Optimización de Lenguajes con Comprobación Estática y Dinámica de Tipos[1]

Miguel Garcia, Francisco Ortin

Universidad de Oviedo

Abstract: Los lenguajes con comprobación dinámica de tipos son utilizados comúnmente en diversos escenarios del desarrollo de software, como el desarrollo rápido de prototipos o aplicaciones con requisitos de alta adaptabilidad dinámica. Por otro lado, los lenguajes con comprobación estática de tipos ofrecen innegables ventajas como la detección temprana de errores y un mayor número de optimizaciones por parte del compilador. Dado que ambos enfoques ofrecen diferentes beneficios, en los últimos años han surgido lenguajes de programación con sistemas de tipos híbridos (estáticos y dinámicos). Del mismo modo, algunos lenguajes con comprobación estática de tipos también han incorporado la posibilidad de incluir tipos dinámicos en su sistema de tipos. Sin embargo, estos lenguajes no realizan inferencia de tipos alguna en tiempo de compilación de código con comprobación dinámica de tipos, perdiendo así parte de su robustez y rendimiento. *StaDyn* es una extensión de C# con comprobación de tipos híbrida, que realiza inferencia de tipos tanto en código declarado estático como dinámico. La información inferida por el compilador es usada para optimizar el código generado, obteniendo un mejor rendimiento sin representar ningún coste de memoria en tiempo de ejecución. Este trabajo evalúa la optimización obtenida en el lenguaje *StaDyn*, comparando el rendimiento en tiempo de ejecución y el consumo de memoria con otros lenguajes híbridos existentes sobre el .NET Framework. La utilización de la información de tipos inferida ha implicado una mejora de rendimiento en todos benchmarks utilizados, siendo 1,5 veces más rápido ejecutando código dinámico y 5 veces con código híbrido.

Palabras clave: Sistemas de tipos híbridos, comprobación estática y dinámica de tipos, lenguajes dinámicos, rendimiento, StaDyn, C#, .NET

1 Introducción

Los lenguajes con comprobación dinámica de tipos son ampliamente utilizados en escenarios específicos como la ingeniería Web, el prototipado rápido de aplicaciones, la programación orientada a aspectos dinámica y, en general, cualquier tipo de desarrollo que requiera una alta adaptabilidad en tiempo de ejecución. La principal ventaja de estos lenguajes es la simplicidad que ofrecen para desarrollar y mantener software altamente dependiente del contexto en el que es ejecutado. Las principales características de los lenguajes dinámicos son la metaprogramación, la reflexión, la movilidad y la reconfiguración dinámica. Tomando la ingeniería

[1] Este trabajo ha sido parcialmente financiado por Microsoft Research, dentro del proyecto titulado *Extending dynamic features of the SSCLI*, obtenido en el *Phoenix and SSCLI, Compilation and Management Execution Request for Proposals*. También ha sido financiado por el Ministerio de Ciencia e Innovación mediante el proyecto TIN2011-25978, *Obtención de Software Adaptable, Robusto y Eficiente añadiendo Reflexión Estructural a Lenguajes con Comprobación Estática de Tipos*.

Web como ejemplo, el lenguaje *Ruby* junto con el framework *Ruby On Rails* ha tenido un elevado éxito en el desarrollo de aplicaciones Web, confirmando la simplicidad de estos lenguajes para implementar los principios DRY (*Don't Repeat Yourself*) [HT99] y *Convention over Configuration* [THS+05]. Actualmente, *JavaScript* esta siendo utilizado para el desarrollo de aplicaciones Web interactivas mediante *AJAX* (*Asynchronous JavaScript And XML*). *PHP* (*PHP Hypertext Preprocessor*) es uno de los lenguajes más populares para el desarrollo web de contenido dinámico. Python es utilizado para diferentes propósitos, siendo dos ejemplos conocidos el servidor de aplicaciones *Zope* y el framework de desarrollo de aplicaciones Web *Django*.

Los beneficios ofrecidos por los lenguajes dinámicos han hecho que algunos lenguajes con comprobación estática de tipos hayan añadido comprobación dinámica de tipos. Un claro ejemplo de esta tendencia es el nuevo tipo `dynamic` añadido al lenguaje C# 4.0 [CT]. Con este nuevo tipo, el compilador pospone todas las verificaciones estáticas hasta la ejecución, lo que permite el desarrollo de código más flexible. Mediante el uso del DLR (*Dynamic Language Runtime*), es posible acceder directamente a programas escritos en IronPython, IronRuby y el código JavaScript usado en Silverlight.

Java también ha seguido esta tendencia. Recientemente la JSR 292 (*Java Specification Request*) [Sun] ha sido incorporada a la versión Java 1.7 SE, añadiendo la instrucción `invokedynamic` a la JVM (*Java Virtual Machine*), y el paquete `java.lang.invoke` a la plataforma. De este modo resulta más sencillo implementar lenguajes con comprobación dinámica de tipos para la máquina virtual de Java. La principal ventaja es que ofrece un mecanismo para posponer la resolución de las llamadas a métodos hasta el tiempo de ejecución.

La gran flexibilidad de los lenguajes con comprobación dinámica de tipos contrasta con las limitaciones derivadas de la falta de comprobación estática. La comprobación estática permite la detección temprana de errores, mejor documentación, mayor número de optimizaciones por parte del compilador y mejor rendimiento. Sin embargo, la comprobación dinámica de tipos proporciona una solución a almacenamiento de datos persistentes, comunicación entre procesos, personalización del comportamiento de programas o generación dinámica de programas [ACPP91]. Por ello, existen situaciones en las que se precisa el uso de tipos dinámicos, incluso cuando se utilizan sistemas de tipos avanzados con comprobación estática de tipos [MD04].

Debido a que ambas aproximaciones poseen diferentes beneficios, ha habido trabajos que permiten ofrecer ambas aproximaciones en el mismo lenguaje. Uno de los primeros trabajos en esta línea es de Meijer y Drayton, donde se proponen sistemas de tipos híbridos, en lugar de obligar a los programadores a elegir exclusivamente entre comprobación estática y dinámica [MD04].

Nuestro trabajo rompe la elección exclusiva entre comprobación dinámica o estática de tipos. Hemos diseñado un lenguaje de programación, denominado *StaDyn* [OZGPSG10], que ofrece tanto comprobación estática como dinámica en el mismo lenguaje de programación – puede consultarse una descripción informal del lenguaje en [Ort]. Este lenguaje de programación combina la robustez y eficiencia de los lenguajes con comprobación estática de tipos con la flexibilidad y adaptabilidad de los lenguajes con comprobación dinámica. El programador indica cuándo requiere alta flexibilidad (tipado dinámico) o la robustez de un sistema de tipos estático. También es posible combinar ambos enfoques, haciendo partes de la aplicación más flexibles, mientras que el resto del programa mantiene su robustez y rendimiento en tiempo de ejecución. La mayor contribución de nuestro lenguaje, comparado con otros existentes que poseen comprobación híbrida, es que el compilador continúa obteniendo información de tipos

del código declarado como dinámico. Esta información es usada para para detectar errores de tipo en tiempo de compilación y optimizar significativamente el código generado, sin implicar por ello un mayor coste de memoria en tiempo de ejecución.

El resto de este artículo se estructura de la siguiente forma. En la siguiente sección, describimos someramente el lenguaje de programación *StaDyn*. En la Sección 3 presentamos una evaluación detallada del rendimiento en tiempo de ejecución, comparando los resultados de diferentes lenguajes. Finalmente, comentamos el trabajo relacionado en la Sección 4 y las conclusiones y trabajo futuro en la Sección 5.

2 El Lenguaje de Programación *StaDyn*

Esta sección presenta una visión general del lenguaje de programación *StaDyn*, las técnicas empleadas para su implementación se detallan en [OZGPSG10] y la descripción formal del sistema de tipos en [Ort11]. El lenguaje de programación *StaDyn* es una extensión de C# 3.0 [Cor], aunque las técnicas utilizadas para su implementación pueden ser aplicadas a cualquier lenguaje de programación orientado a objetos con comprobación estática de tipos. El principal cambio es la extensión del comportamiento del tipado implícito de variables locales de C# 3.0. En *StaDyn*, el tipo de las referencias puede ser declarado de manera explícita, pero también puede ser establecido implícitamente mediante el uso de la palabra reservada `var`. *StaDyn* permite el uso `var` como cualquier otro tipo, mientras que C# 3.0 solamente lo permite en la inicialización de referencias locales. *StaDyn* no ofrece la palabra reservada `dynamic` de C# 4.0. En su lugar permite el uso de referencias `var` estáticas y dinámicas. A continuación se identifican de un modo somero las principales características del lenguaje *StaDyn*.

Los lenguajes con comprobación estática de tipos existentes obligan a una variable declarada con un tipo *T* a tener el mismo tipo *T* dentro de su ámbito. No obstante, los lenguajes dinámicos permiten albergar diferentes tipos en el mismo ámbito. *StaDyn* ofrece esta característica mediante la comprobación estática de tipos, considerando el tipo concreto de cada referencia. En la Figura 1, `figura` tiene más de un tipo en el método `f`. La inferencia de tipo de las variables `var`, se ha implementado mediante una modificación [OZGPSG10] del algoritmo de *Hindley-Milner* [Mil78].

```
 1: using System;
 2: class Circumferencia {
 3:     public var x, y, radio;
 4: }
 5: class Rectangulo {
 6:     public var x, y, ancho, alto;
 7: }
 8: class Triangulo {
 9:     public var x1, y1, x2, y2, x3, y3;
10: }
```

```
11: class Programa {
12:  public static int f() {
13:     var figura;
14:     if (Random.Next() % 2 == 0)
15:         figura = new Circumferencia();
16:     else
17:         figura = new Rectangulo();
18:     // static duck typing
19:     return figura.x; // Se infiere int
20: }
21:}
```

Figura 1: Inferencia de tipos unión.

El *Duck Typing* es una propiedad de los lenguajes dinámicos que significa que un objeto es intercambiable por cualquier otro objeto que implemente la misma interfaz dinámica, independientemente de si tienen una relación de herencia o no. *StaDyn* ofrece *duck typing*, pero con comprobación estática de tipos. Si una referencia `var` apunta a un conjunto de objetos que

implementan un método publico `m`, el mensaje `m` puede ser pasado de forma segura, aunque estos objetos no implementen una interfaz común o una clase (abstracta) con el método `m`. Un ejemplo de esta propiedad es la línea 19 de la Figura 1 (se puede pasar el mensaje `x` a `figura`). Esta propiedad se obtiene a través de la inferencia de tipos unión [Pie92].

StaDyn permite el uso de referencias `var` tanto estáticas como dinámicas. Dependiendo de su dinamismo, la comprobación e inferencia de tipos puede ser pesimista (estática) u optimista (dinámica), sin alterar el significado dinámico del lenguaje de programación. Por ejemplo, si la referencia `figura` fuese dinámica, el mensaje `radio` podría ser pasado. El dinamismo de las referencias `var` es manejado por el IDE, ubicándolo en un fichero separado (un documento XML) [OM11]. Dado que el dinamismo no está declarado explícitamente en el código fuente, *StaDyn* facilita la conversión de referencias dinámicas en estáticas y viceversa. Esta separación permite cambiar de prototipos desarrollados rápidamente a aplicaciones finales robustas y eficientes. También es posible realizar partes de una aplicación más adaptables, manteniendo la robustez del resto del programa.

La reconstrucción de tipos concretos no se limita solamente a las variables locales. *StaDyn* realiza una análisis global sensible al flujo de ejecución de todas las referencias `var` implícitas. Esto implica que es posible declarar tanto atributos (campos) como parámetros `var`. Tal y como se aprecia en la Figura 1, los atributos de `Circunferencia` y `Rectángulo` pueden tener cualquier tipo (son `var`). Para implementar esta característica, se ha ampliado el sistema de tipos, haciendo que sea basado en restricciones [OSW97].

3 Evaluación del rendimiento en tiempo de ejecución

Hemos evaluado el rendimiento y consumo de memoria de *StaDyn*, comparándola con otros lenguajes. El principal objetivo de esta evaluación es conocer la mejora obtenida en el rendimiento gracias a la inferencia estática de tipos y contrastar los resultados con las evaluaciones preliminares llevadas a cabo anteriormente [OZGPSG10].

3.1. Metodología

Para realizar la evaluación del lenguaje de programación *StaDyn* descrito en este trabajo, hemos comparado su rendimiento y consumo de memoria con los lenguajes de programación con comprobación de tipos hibrida existentes para el Framework .NET, todos ellos compilados con sus máximo nivel de optimización. Nos centramos en lenguajes .NET para evitar la introducción de variaciones causadas por el uso de diferentes plataformas (por ejemplo Java o aplicaciones nativas). Los lenguajes que hemos usado para la evaluación son:

- C# 4.0. La última versión de *C#* que combina comprobación de tipos estática y dinámica con el tipo `dynamic`. Hace uso del DLR, la capa sobre el CLR que ofrece un conjunto de servicios que facilita la implementación de lenguajes dinámicos [CT].
- IronPython 2.7.3. Es una implementación de software libre del lenguaje de programación Python integrado con el Framework .NET 4.0, utiliza el DLR. Compila programas Python a código ejecutable .NET [Iro].
- Visual Basic (VB) 10: También ofrece comprobación híbrida. Las referencias dinámicas se declaran con la palabra reservada `Dim`, sin establecer un tipo. Con esta sintaxis,

el compilador no obtiene información de tipos estática y la comprobación de tipos se realiza dinámicamente [Vic07].

- Boo 0.9.4.9: Lenguaje de programación para el CLI con una sintaxis inspirada en Python. Tiene comprobación estática de tipos, pero permite el uso de *duck typing* mediante el tipo especial `duck` [DO].
- Cobra 0.9.1: Se trata de un lenguaje de programación orientado a objetos con comprobación dinámica y estática de tipos, que posee inferencia de tipos en tiempo de compilación. Al igual que C#, utiliza el tipo especial `dynamic` para la comprobación dinámica de tipos [SFM+96].
- Fantom 1.0.63: Es un lenguaje de programación orientado a objetos que genera código para la maquina virtual de *Java (JVM)*, la plataforma .NET y *JavaScript*. Tiene comprobación estática de tipos, pero permite la invocación dinámica de métodos con un operador específico de paso de mensajes [FF12].
- *StaDyn*. Los mismos programas codificados en C# 4.0 se trasforman en *StaDyn* simplemente sustituyendo la palabra reservada `dynamic` por `var` especificando estas variables como dinámicas.

Todos estos lenguajes compilan código para la plataforma .NET, facilitando la comparación de los resultados del rendimiento. Para cada lenguaje hemos evaluado una versión de las siguientes aplicaciones:

- Pybench. Es un benchmark diseñado para medir el rendimiento de implementaciones estándar de Python. Está compuesto por una colección de 52 test que evalúan diferentes aspectos del lenguaje de programación Python.
- Pystone. Se trata de la versión Python del benchmark Dhrystone.
- Un subconjunto de aplicaciones del benchmark *Java Grande* [Kri12]:
  - Section 2 (*Kernels*). FFT, algoritmo de transformación de N números complejos; Heapsort, el algoritmo de ordenación *heap sort* con arrays de números enteros; y SparseMatmult, manejo de matrices dispersas no estructuradas.
  - Section 3 (*Large Scale Applications*). RayTracer, trazador de rayos de escenas 3D que contienen 64 esferas renderizadas a una resolución de 25x25 pixeles.
- Points. Se trata de una aplicación diseñada para evaluar el rendimiento de lenguajes con comprobación estática y dinámica de tipos [Ort11].

Respecto al análisis de los datos, hemos seguido la metodología propuesta en [GBE07] para la evaluación del rendimiento de maquinas virtuales que poseen compilación *JIT*. La metodología define dos pasos para evaluar aplicaciones no servidoras:

1. Medimos el tiempo de ejecución resultante de ejecutar múltiples veces el mismo programa. El resultado son p (tomamos $p = 30$) mediciones x_i con $1 \le i \le p$.
2. Se calcula el intervalo de confianza para un nivel de confianza dado (95%) para eliminar los errores de medición que pueden introducir una variación en la evaluación. El intervalo de confianza se calcula usando la distribución *t de Student* [Lilja2005]. Calculamos el intervalo de confianza $[c1,c2]$ como:

$$c1 = \bar{x} - t_{1-\alpha/2;p-1}\frac{s}{\sqrt{p}} \qquad c2 = \bar{x} + t_{1-\alpha/2;p-1}\frac{s}{\sqrt{p}}$$

Siendo $\bar{x}$ la media aritmética la las mediciones x_i, $\alpha = 0.05$ (95%), s la desviación típica de las mediciones x_i y $t_{1-\alpha/2;p-1}$ se define de tal forma que una variable aleatoria T, que sigue la dis-

tribución *t de Student* con *p -1* grados de libertad, cumple que $Pr[T \leq t_{1-\alpha/2;p-1}] = 1 - \alpha / 2$. El dato obtenido de cada medición es la media del intervalo de confianza y un porcentaje indicando la amplitud del intervalo de confianza relativo a la media.

Para cada benchmark hemos medido la diferencia entre el comienzo y el final de cada invocación, para obtener el tiempo de ejecución de cada benchmark. El consumo de memoria sigue la misma metodología, midiendo la memoria usada por todo el proceso (propiedad `PeakWorkingSet`). Todos los test han sido ejecutados en un sistema Intel Core 2 Duo P7450 a 2,13Ghz con 4GB de memoria RAM ejecutando una versión actualizada de *Windows 7 Home Premium Sp1*.

3.2. Comprobación dinámica de tipos

Hemos evaluado diferentes benchmarks con tipado dinámico, haciendo que todas las referencias del código fuente sean declaradas como dinámicas, sin usar ninguna declaración explícita de tipo. En las siguientes secciones utilizaremos código fuente con comprobación de tipos hibrida y declaración explícita de tipos, con el fin de medir y comparar distintos escenarios.

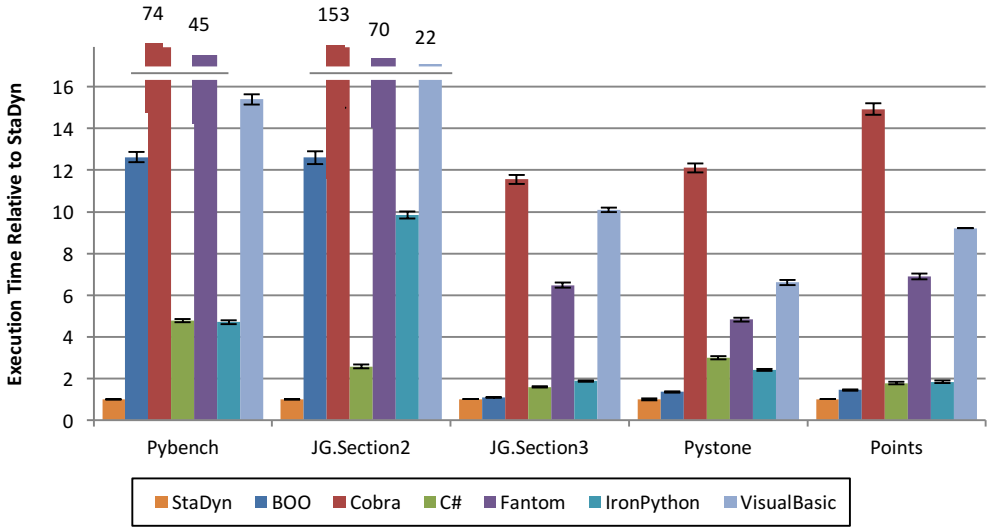

Figura 2: Tiempo de ejecución de los benchmarks con comprobación dinámica de tipos.

La Figura 2 muestra el tiempo medio de ejecución relativo a *StaDyn* de cada uno de los benchmarks evaluados. *StaDyn* presenta el mejor rendimiento en todos los test. Ejecutando el Pybench *StaDyn* es 4, 4, 12, 14, 45 y 74 veces más rápido que IronPython, C#, Boo, VB, Fantom y Cobra, respectivamente. La diferencia entre *StaDyn* y los demás lenguajes es debida a que en este benchmark se prueban características específicas de los lenguajes (como las operaciones aritméticas, invocación a métodos, manejo de listas, acceso a atributos, etc.) utilizando en todos los casos variables dinámicas. Mientras que *StaDyn* es capaz de inferir los tipos de la mayoría de las variables dinámicas en tiempo de compilación, los demás lenguajes utilizan reflexión para ejecutar las distintas operaciones. La diferencia entre C# y IronPython y el resto de lenguajes (Boo, VB, Fantom y Cobra) se debe a las optimizaciones implementadas por el

DLR (empleado por estos dos lenguajes), que utiliza una caché para reducir la penalización del uso de reflexión.

La ejecución de los test de JG.Section2 muestra importantes diferencias de rendimiento. *StaDyn* es 1,57 veces más rápido que el segundo que mejor rendimiento tiene (C#) y llega a ser 2,18 ordenes de magnitud más rápido que el peor de todos (Cobra). Todos los algoritmos de esta sección, a pesar de ser diferentes, utilizan las mismas características del lenguaje: gran cantidad de operaciones aritméticas y múltiples accesos a arrays usando variables declaradas como dinámicas. *StaDyn* infiere el tipo de casi todas las variables dinámicas y puede utilizar instrucciones específicas, por ejemplo, para acceder a los elementos de los arrays. No obstante, *StaDyn* no infiere información de tipos de los parámetros declarados como dinámicos [OZGPSG10]. En esos casos, al igual que los demás, usa reflexión en tiempo de ejecución. El resto de lenguajes usan reflexión en todos los casos, conllevando una importante penalización en el rendimiento.

En el caso de la aplicación JG.RayTracer (JG.Section3), las diferencias en el rendimiento son significativamente menores que en las otras aplicaciones. *StaDyn* es un 8%, 58%, 87%, 547%, 909% y 1.055% más rápido que Boo, C#, IronPython, Fantom, VB y Cobra, respectivamente. A lo largo de todo el código, hay múltiples invocaciones a métodos con parámetros declarados como dinámicos. Dado que en estos casos *StaDyn* no posee información de tipos para optimizar el código, el uso de reflexión causa un menor rendimiento. Sin embargo, *StaDyn* continúa siendo la implementación más rápida, debido al beneficio obtenido de la recopilación de información de tipos de otras expresiones en el código. Los resultados obtenidos en la evaluación de esta aplicación real confirman los obtenidos en las evaluaciones preliminares [OZGPSG10], donde *StaDyn* presentaba el mejor rendimiento cuando no se infería ninguna información de tipos. El resultado del benchmark Pystone es similar al del JG.RayTracer, puesto que también utiliza gran cantidad de parámetros declarados como dinámicos. También se ha incluido en la Figura 2 la evaluación de una versión modificada de la aplicación Points [Ort11]. Este código, originalmente hibrido, ha sido modificado en esta sección para que sea completamente dinámico. El resultado obtenido es similar al de los benchmarks JG.RayTracer y Pystone.

En cuanto al consumo de memoria, *StaDyn* presenta el menor consumo de memoria en todas las pruebas. VB, Cobra, Boo, Fantom, C# y IronPython requieren 16%, 32%, 34%, 64%, 82% y 303% más memoria que *StaDyn*, respectivamente. Los lenguajes que mayor consumo de memoria presentan son los que hacen uso del DLR (C# y IronPython). Vemos pues cómo la mejora en el rendimiento ofrecida por el DLR también implica un mayor consumo de memoria. *StaDyn*, no obstante, ofrece un beneficio de rendimiento sin penalizar el consumo de memoria.

3.3. Comprobación estática y dinámica de tipos

Hemos modificado el código fuente de los benchmarks anteriores, declarado explícitamente el tipo de los atributos y los parámetros de los métodos. En el resto de los casos hemos usado tipado dinámico. Dado que el benchmark Pybench no tiene ni parámetros ni atributos de clase, no se ha incluido en esta evaluación. Tampoco hemos incluido IronPython, puesto que carece de comprobación estática de tipos.

La Figura 3 muestra los resultados de la evaluación estos benchmarks, donde *StaDyn* presenta el mejor rendimiento en todos los programas. En la ejecución de los programas de

JG.Section2, *StaDyn* es 3,3 veces más rápido que el segundo lenguaje con mejor rendimiento (C#) y 2,35 órdenes de magnitud que el que peor rendimiento presenta (Cobra). Esto es debido a que *StaDyn* infiere el tipo de las variables declaradas como dinámicas, y por lo tanto, genera código optimizado con la información estática de tipos inferida. En el caso de los arrays declarados como dinámicos, Boo y VB poseen operaciones especiales para su acceso (`SetSlice` y `LateIndexSet` respectivamente) incurriendo en una penalización menor que C#, Fantom y Cobra, que usan reflexión.

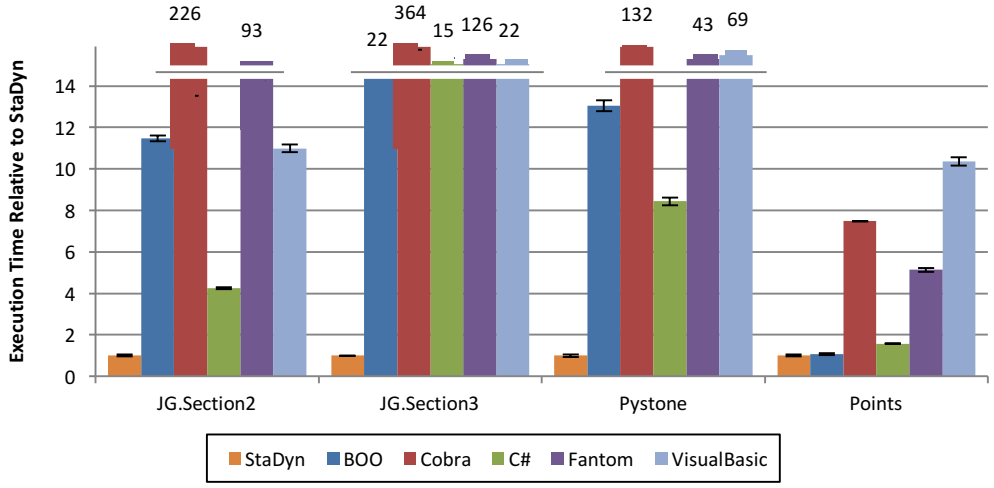

Figura 3: Tiempo de ejecución de los benchmarks con comprobación estática y dinámica de tipos.

Comparando estos resultados con los obtenidos utilizando solamente comprobación dinámica de tipos (Figura 2), tanto Boo como VB reducen el tiempo de ejecución relativo a *StaDyn* (Boo pasa de 1.200% a 1.050% y VB de 2.200% a 1.000%), al contrario que C#, Fantom y Cobra cuya diferencia respecto a *StaDyn* es mayor con comprobación estática y dinámica de tipos (C# pasa de 150% a 325%, Fantom de 700% a 930% y Cobra de 1.530% a 2.260%). Estas diferencias se deben a que, usando comprobación híbrida, por un lado, *StaDyn* mejora su rendimiento dado que los tipos de los parámetros de los métodos son explícitamente declarados; pero, por otro lado, la declaración explícita de los tipos de los atributos no implica un beneficio para *StaDyn*, ya que usando comprobación dinámica de tipos ya infería estos tipos. Para el resto de lenguajes, la declaración explícita de los tipos de los parámetros y atributos de clase implica una mejora en el rendimiento, que es más significativa para Boo y VB que para C#, Fantom y Cobra, debido a la forma que utilizan para acceder a los arrays declarados dinámicamente.

En la aplicación JG.RayTracer *StaDyn* presenta la mayor diferencia de rendimiento comparado con el resto de lenguajes, siendo 15, 21, 21, 126 y 364 veces más rápido que C#, Boo, VB, Fantom y Cobra, respectivamente. Esto se debe a que, al contrario que los demás lenguajes, en *StaDyn* infiere el tipo de todas las variables `var`. Comparando estos datos con su versión exclusivamente dinámica, el tiempo de ejecución de *StaDyn* es 128 veces más rápido, mientras que los demás mejoran entre 4 (Cobra) y 50 (VB) veces. Esto es debido a que la versión dinámica presenta el peor escenario para *StaDyn* (gran cantidad de parámetros declarados como dinámicos), mientras que para la versión híbrida infiere el tipo de todas las variables `var`.

El resultado del benchmark Pystone es similar al de JG.Section2. *StaDyn* es aproximadamente 7 veces más rápido que C# (el segundo mejor) y 2,1 órdenes de magnitud que Fantom (el de peor rendimiento). La comparación con la versión 100% dinámica de Pystone es similar al caso de JG.RayTracer: *StaDyn* infiere el tipo de todas las variables declaradas como dinámicas y su rendimiento mejora 20 veces, mientras que los demás lenguajes mejoran aproximadamente 2.

En cuanto a la aplicación Points, a pesar de que *StaDyn* es el lenguaje con mejor rendimiento, la diferencia con los demás lenguajes es mucho menor que en las otras pruebas presentadas en esta sección. El tiempo de ejecución de *StaDyn* es un 7% y 58% mejor que el de Boo y C# (segundos lenguajes con mejor rendimiento), respectivamente. Esto es debido a que en este programa, la mayoría de las variables declaradas como dinámicas son parámetros de métodos para los que *StaDyn* utiliza reflexión. Comparado con la versión dinámica, esta versión es un 15% más rápida.

En términos de consumo de memoria, *StaDyn* vuelve a ser el lenguaje que menos recursos consume, siendo un 21%, 43%, 48%, 93% y 93% menor que VB, Boo, Cobra, Fantom y C#, respectivamente.

3.4. Declaración explícita de tipos

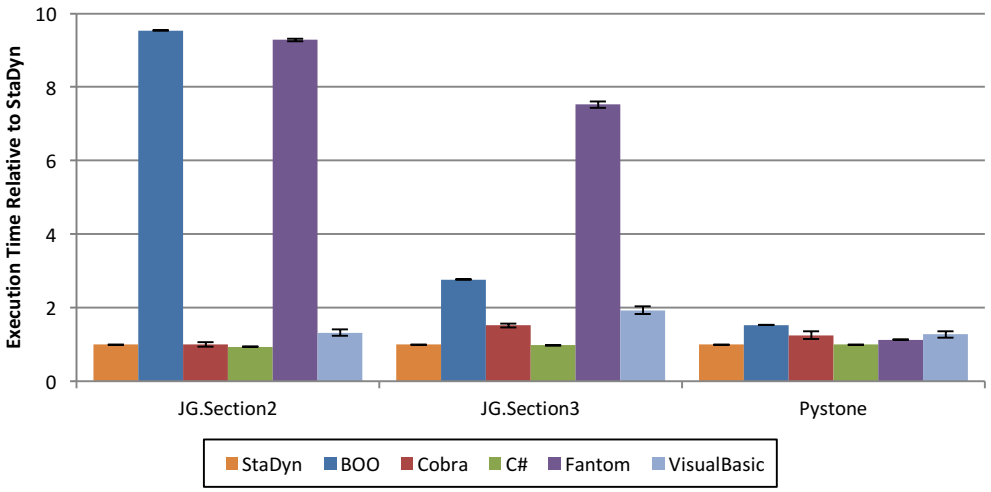

Figura 4: Tiempo de ejecución de los benchmarks con declaración explícita de tipos.

En esta sección hemos evaluado código que únicamente posee declaración explícita, y por tanto estática, de tipos. El principal objetivo de esta evaluación es comparar las optimizaciones que realiza cada compilador cuando no se usa ninguna declaración dinámica de tipos. Hemos modificado los benchmarks de tal forma que el tipo de todas las variables sea declara explícitamente. La aplicación Points no ha sido incluida en esta sección, dado que se diseño para que no fuese posible implementarla sin usar variables dinámicas [Ort11].

La Figura 3 muestra el tiempo medio de ejecución relativo a *StaDyn* de los benchmarks Java Grande y Pystone. C# es el lenguaje más rápido en todas las pruebas, siendo un 2,5% más rápido que *StaDyn*, que es el segundo con mejor rendimiento. Este contraste muestra las dife-

rencias entre las optimizaciones hechas por el compilador comercial de C# y *StaDyn*. Comparándolo con los otros lenguajes, *StaDyn* es, de media, un 24%, 48%, 243% y 335% más rápido que Cobra, VB, Boo y Fantom, respectivamente. Estos resultados son similares a los obtenidos en las evaluaciones preliminares previas [OZGPSG10].

En las aplicaciones de Java Grande, el rendimiento de Boo y Fantom es significativamente peor que el resto de lenguajes; de media requieren un 414% y 735% más de tiempo de ejecución que *StaDyn*. Esta diferencia se debe a que tanto Boo como Fantom utilizan operaciones con tipos dinámicos, incluso cuando se utiliza declaración explícita de tipos.

El benchmark Pystone presenta una situación diferente, siendo el rendimiento de todos los lenguajes muy similar. Fantom, Cobra, VB, y Boo requieren un 13%, 25%, 27% y 53% más de tiempo de ejecución que *StaDyn* para ejecutar el Pystone. En esta aplicación, apenas se realizan accesos a arrays u otras operaciones que hagan que alguno de los lenguajes utilice sus propias operaciones para ejecutarlas. De esta forma, todos utilizan las instrucciones IL estáticamente tipadas, y las diferencias de rendimiento se reducen considerablemente.

StaDyn y C# son los lenguajes con menor consumo de memoria en este tipo de programas, siendo prácticamente iguales (la diferencia es menor que el intervalo de error). VB, Boo, Cobra y Fantom requieren un 17%, 43%, 59% y 95% más memoria que *StaDyn*.

3.5. Influencia del tipado en el rendimiento

La Figura 4 muestra la influencia de la comprobación dinámica, híbrida y estática en el tiempo de ejecución de cada uno de los lenguajes. Los valores en la Figura 4 son relativos al tiempo de ejecución de cada lenguaje para la declaración explícita (estática) de tipos. Por tanto, la figura representa el coste de utilizar comprobación dinámica e híbrida de tipos, para cada lenguaje.

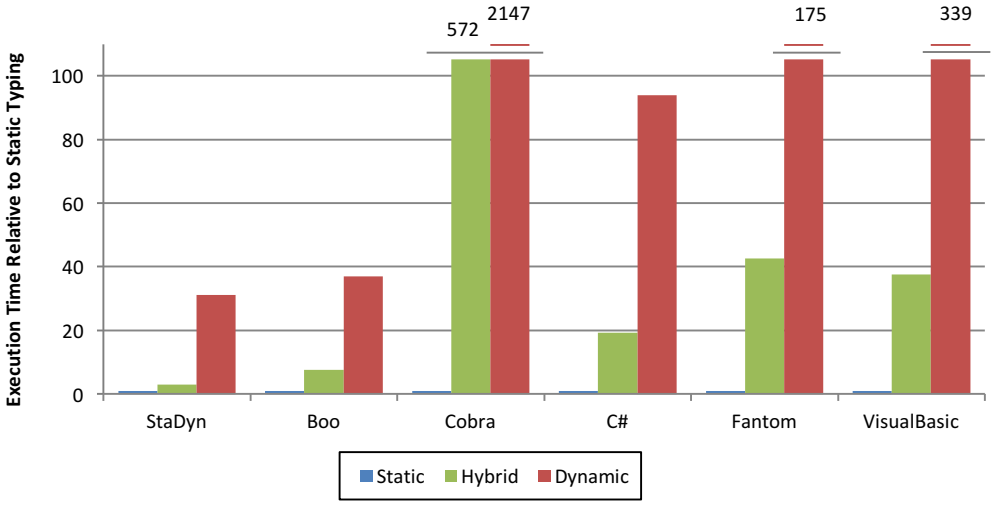

Figura 5: Influencia en el rendimiento de la comprobación dinámica de tipos.

StaDyn es el lenguaje que introduce la menor penalización en el rendimiento, al usar tanto comprobación dinámica como híbrida frente a la estática. Comparando la declaración explícita con la comprobación híbrida, el coste en StaDyn supone dos factores. En Boo, C#, VB, Fantom y Cobra este coste es de 7, 18, 36, 41 y 571 veces. La diferencia con StaDyn es que estos

lenguajes no infieren información de tipos de las variables declaradas como dinámicas durante la compilación.

Por último, con *StaDyn* los programas con declaración explícita de tipos son 1.4 órdenes de magnitud más rápidos que con comprobación dinámica. Este valor es 1,5; 1,9; 2,2; 2,5 y 3,3 órdenes de magnitud para Boo, C#, Fantom, VB y Cobra, respectivamente.

4 Trabajo Relacionado

Dado que tanto la comprobación dinámica como estática de tipos ofrecen distintos beneficios, existen diversos trabajos enfocados a obtener las ventajas de ambos. Uno de los primeros trabajos en este campo fue *Soft Typing* [CF91], que propone añadir comprobación estática de tipos a un lenguaje dinámico como *Scheme*. La aproximación propuesta en [ACPP91] añade un tipo *Dynamic* al cálculo lambda, incluyendo dos operaciones de conversión (`dynamic` and `typecase`), generando un código prolijo y altamente dependiente de su dinamismo.

Los trabajos presentados en *Quasi-Static Typing* [Tha90], *Hybrid Typing* [FFT06] and *Gradual Typing* [ST07] realizan la conversión implícita entre código dinámico y estático, empleando relaciones de subtipado en los dos primeros, y una relación de *consistencia* en el último. El trabajo *Gradual Typing* propone, al igual que *StaDyn*, la resolución de restricciones mediante unificación para integrar la comprobación estática y dinámica de tipos [SV08].

Las implementaciones de lenguajes de programación como Boo, Visual Basic (VB) .NET, Cobra, Dylan, Strongtalk, y *C#* 4.0 han incluido parte de trabajos teóricos que combinan comprobación de tipos estática y dinámica [BMT10]. Algunos lenguajes de programación han seguido el enfoque de añadir un nuevo tipo dinámico [ACPP91] (`dynamic` en C# y Cobra, y `duck` en Boo), mientras que otros representan los tipos dinámicos eliminando los tipos en la declaración de las variables (*VB* y *Dylan*) [Vic07]. Strongtalk sigue un enfoque completamente diferente basado en el concepto de sistema de tipos intercambiable (*pluggable*) [Bra04]. En este lenguaje, los tipos dinámicos son implícitamente forzados a tipos estáticos siguiendo el enfoque descrito en [Tha90] y [ST07], contrariamente al uso de una instrucción explícita para la conversión como se propone en [ACPP91]. Dado que estas conversiones implícitas pueden fallar en tiempo de ejecución, se introduce una verificación dinámica de tipo en el código generado [FFT06].

También existen trabajos enfocados a realizar inferencia de tipos estática en lenguajes con comprobación dinámica, descubriendo errores de tipo antes de la ejecución de un programa. *Diamondback Ruby (DRuby)* es una herramienta que combina el sistema de tipos dinámico de Ruby con un método de comprobación estática de tipos [FAFH09]. Anderson, Giannini y Drossopoulou formalizaron un subconjunto de JavaScript *(JS0)*, definiendo un algoritmo de inferencia de tipos que es parecido a un sistema de tipos [AGD05]. Ninguno de estos trabajos (DRuby and JS0) usa la información de tipos estática inferida para generar código optimizado.

5 Conclusiones

El lenguaje de programación *StaDyn* muestra cómo la obtención de información de tipos en lenguajes con comprobación híbrida puede ser utilizada para realizar importantes optimizaciones de rendimiento sin incurrir en un consumo adicional de memoria. *StaDyn* realiza la inferencia y comprobación de tipos incluso sobre referencias declaradas como dinámicas, ofre-

ciendo el nivel de flexibilidad de los lenguajes dinámicos, y manteniendo la eficiencia y robustez de los lenguajes con comprobación estática de tipos.

Hemos comparado el rendimiento y consumo de memoria de *StaDyn* con los lenguajes híbridos existentes sobre el .NET Framework, utilizando un amplio conjunto de benchmarks. *StaDyn* ha mostrado el mejor rendimiento en todos los programas que usan alguna referencia dinámica, siendo al menos 1,2 veces más rápido ejecutando código dinámico y 5 veces con código hibrido.

El trabajo futuro más inmediato es realizar especialización de funciones con la información de tipos de los parámetros. Los resultados obtenidos de esta evaluación indican que esta mejora podría significar mayores optimizaciones en el rendimiento del lenguaje. La versión actual del lenguaje de programación *StaDyn* y su código fuente está disponible en:

http://www.reflection.uniovi.es/stadyn

References

[ACPP91] Martín Abadi, Luca Cardelli, Benjamin Crawford Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, 1991.

[AGD05] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards type inference for JavaScript. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 428–452, Glasgow, UK, 9-11 June 2005. Springer.

[BMT10] Gavin Bierman, Erik Meijer, and Mads Torgersen. Adding dynamic types to C#. In *Proceedings of the 24th European Conference on Object-Oriented Programming*, ECOOP'10, pages 76–100, Maribor, Slovenia, 21-25 June 2010. Springer-Verlag.

[Bra04] Gilad Bracha. Pluggable Type Systems. In *Proceedings of the OOPSLA 2004 Workshop on Revival of Dynamic Languages*, Vancouver, Canada, October 2004. ACM.

[CF91] Robert Cartwright and Miken Fagan. Soft Typing. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 278–292, Toronto, Canada, 26-28 June 1991. ACM.

[Cor] Microsoft Corporation. The C# Programming Language.

[CT] Bill Chiles and Alex Turner. Dynamic Language Runtime. http://www.codeplex.com/-Download?ProjectName=dlr&DownloadId=127512.

[DO] Rodrigo B. De Oliveira. The Boo programming language. http://boo.codehaus.org.

[FAFH09] Michael Furr, Jong-hoon David An, Jeffrey S. Foster, and Michael Hicks. Static type inference for Ruby. In *Proceedings of the ACM symposium on Applied Computing (SAC)*, pages 1859–1866, Honolulu, Hawaii, 9-12 March 2009. ACM.

[FF12] Brian Frank and Andy Frank. Fantom, the language formerly known as Fan. http://-fantom.org, 2012.

[FFT06] Cormac Flanagan, Stephen N. Freund, and Aaron Tomb. Hybrid types, invariants, and refinements for imperative objects. In *Proceedings of the International Workshop on Foundations and Developments of Object-Oriented Languages (FOOL)*, San Antonio, Texas, 23 January 2006. ACM.

[GBE07] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. *ACM SIGPLAN Notices*, 42(10):57–76, 2007.

[HT99] Andrew Hunt and David Thomas. *The pragmatic programmer: from journeyman to master*. Addison-Wesley Longman Publishing Co., Inc., Boston, Massachusetts, 1999.

[Iro] IronPython. The IronPython programming language. http://ironpython.net/.

[Kri12] Krintz, Chandra. A collection of phoenix-compatible C# benchmarks. http://-www.cs.ucsb.edu/~ckrintz/racelab/PhxCSBenchmarks, 2012.

[MD04] Erik Meijer and Peter Drayton. Static typing where possible dynamic typing when needed: The end of the cold war between programming languages. In *Proceedings of the OOPSLA Workshop on Revival of Dynamic Languages*, Vancouver, Canada, 24-28 October 2004. ACM.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[OM11] Francisco Ortin and Anton Morant. Ide support to facilitate the transition from rapid prototyping to robust software production. In *Proceedings of the 1st Workshop on Developing Tools as Plug-ins*, TOPI '11, pages 40–43, New York, NY, USA, 2011. ACM.

[Ort] Francisco Ortin. The StaDyn programming language. http://www.reflection.uniovi.es/-stadyn.

[Ort11] Francisco Ortin. Type inference to optimize a hybrid statically and dynamically typed language. *Computer Journal*, 54(11):1901–1924, November 2011.

[OSW97] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. In *Fourth International Workshop on Foundations of Object-Oriented Programming (FOOL)*, 1997.

[OZGPSG10] Francisco Ortin, Daniel Zapico, Jose Baltasar Garcia Perez-Schofield, and Miguel Garcia. Including both static and dynamic typing in the same programming language. *IET Software*, 4(4):268–282, 2010.

[Pie92] Benjamin Crawford Pierce. Programming with intersection types and bounded polymorphism. Technical Report CMU-CS-91-106, School of Computer Science, Pittsburgh, PA, USA, 1992.

[SFM+96] Jon Siegel, Dan Frantz, Hal Mirsky, Raghu Hudli, Peter de Jong, Alan Klein, Brent Wilkins, Alex Thomas, Wilf Coles, Sean Baker, and Maurice Balick. *COBRA fundamentals and programming*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[ST07] Jeremy G. Siek and Walid Taha. Gradual typing for objects. In *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP)*, pages 2–27, Berlin, Germany, 30 July - 3 August 2007. Springer-Verlag.

[Sun] Sun Microsystems. JSR 292, supporting dynamically typed languages on the java platform. http://www.jcp.org/en/jsr/detail?id=292.

[SV08] Jeremy G. Siek and Manish Vachharajani. Gradual typing with unification-based inference. In *Proceedings of the Dynamic Languages Symposium*, pages 7:1–7:12, Paphos, Cyprus, 25 July 2008. ACM.

[Tha90] Satish Thatte. Quasi-static typing. In *Proceedings of the 17th symposium on Principles of programming languages (POPL)*, pages 367–381, San Francisco, California, United States, January 1990. ACM.

[THS+05] Dave Thomas, David Heinemeier Hansson, Andrea Schwarz, Thomas Fuchs, Leon Breedt, and Mike Clark. *Agile Web Development with Rails. A Pragmatic Guide*. Pragmatic Bookshelf, Raleigh, North Carolina, 2005.

[Vic07] Paul Vick. *The Microsoft Visual Basic Language Specification*. Microsoft Corporation, Redmond, Washington, 2007.