Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Implementación de un Entorno de Ejecución Reflexivo mediante compiladores de Meta-Trazas

Tesis de Licenciatura en Ciencias de la Computación

Dardo Gustavo Marasca

Director: Guido Chari

Codirector: Diego Garbervetsky

Buenos Aires, 2020

RESUMEN

Los lenguajes dinámicos permiten al programador construir sistemas de software de forma ágil debido a que eliminan la necesidad de indicar tipos estrictos a todas las variables o entidades que forman parte del programa. Usualmente estos lenguajes son ejecutados sobre una máquina virtual que abstrae el hardware o sistema subyacente. Gracias a esto, un mismo producto de software puede ser ejecutado en diferentes entornos sin grandes modificaciones. Si bien esto último permite gran flexibilidad, la indirección adicional provoca que en términos generales los lenguajes de tipado dinámico sean significativamente más lentos que aquellos lenguajes que tienen la capacidad de generar ejecutables en código nativo.

Para mejorar las capacidades evolutivas de las aplicaciones en tiempo de ejecución, recientemente se introdujo a los entornos de ejecución reflexivos, máquinas virtuales que permiten, a los programas que ejecutan soportados por ella, inspeccionarla y modificarla según sean sus necesidades. Debido a la flexibilidad que poseen los lenguajes dinámicos y la gran cantidad de estudios relacionados a la construcción de máquinas virtuales para ellos, las primeras implementaciones de entornos de ejecución reflexivos están siendo construidos para esta familia de lenguajes. En recientes estudios fue desarrollado *MATE*, un modelo para la construcción de máquinas virtuales reflexivas que, mediante un protocolo basado en meta-objetos, permite al usuario modificar el comportamiento de diferentes componentes de la máquina virtual subyacente, como por ejemplo, la organización de la memoria o el proceso ejecución de métodos. Gracias a una implementación basada en meta-compiladores con 'evaluación parcial', se demostró que los costos adicionales en términos de rendimiento introducidos por las funcionalidades reflexivas, pueden ser mitigados considerablemente.

En este trabajo se explora una implementación alternativa de *MATE*, basada en compiladores por *metra-trazas*, con el fin de verificar si los resultados previos son generalizables a esta familia de compiladores. Para ello, implementamos una máquina virtual, mediante el framework *Pypy*, comparable en funcionalidades a las implementaciones previas de *MATE*. Esta tecnología, diseñada para la construcción de máquinas virtuales, nos permitió adicionalmente, y de forma muy sencilla, dotar a nuestra máquina virtual de un compilador *JIT* para mejorar su rendimiento.

Para comprobar nuestra hipótesis, comparamos el rendimiento de ambas soluciones ante un conjunto de pruebas diseñadas para evaluar el impacto inherente del soporte a *MATE* y su desempeño en casos de uso posibles de las capacidades reflexivas. En nuestro análisis logramos comprobar que ambas soluciones presentan un rendimiento asintóticamente similar, siguiendo la misma linea que los resultados de estudios previos que comparan compiladores *JIT* basados en *evaluación-parcial* y *meta-trazas*.

Palabras claves: máquinas virtuales, reflexión, compiladores dinámicos, compiladores de trazas, evaluación parcial, especulación, adaptación de software, evolución de software.

AGRADECIMIENTOS

A mi amada esposa que me ayudo durante el desarrollo de mi tesis y nunca dejó que bajara los brazos. A mis padres que me dieron el empujón a estudiar Ciencias de la Computación en la Universidad de Buenos Aires. Finalmente quiero darle gracias a mi abuela que me compró mi primer computadora a los 6 años (una 286 con escasos megas de memoria RAM y disco rígido) y me introdujo a este fascinante mundo de la computación.

Índice general

1. INTRODUCCIÓN

Según su estrategia de tipado, los lenguajes de programación pueden ser clasificados en dos grandes familias: lenguajes de tipado estático y lenguajes de tipado dinámico. La principal diferencia entre ambas familias radica en que los primeros requieren que el programador declare explícitamente el tipo de las variables y objetos que forman parte del programa, mientras que en el segundo grupo esto no es necesario. La información de tipos en el código fuente del programa permite, entre otras cosas, que este sea optimizado para aumentar su rendimiento. El esquema más tradicional de ejecución en los lenguajes de tipado estático consiste en compilar el código original en un ejecutable nativo para que pueda ser interpretado directamente por el hardware subyacente. A diferencias de estos últimos, la mayoría de los lenguajes de tipado dinámico requiere de un interprete adicional encargado de ejecutar el código fuente, agregando una indirección adicional durante la ejecución del programa.

Los lenguajes de tipado dinámico corren generalmente sobre máquinas virtuales que proporcionan una amplia gama de funcionalidades heterogéneas que incluyen, entre otras, la implementación de la semántica de algún lenguaje, la gestión automática de la memoria, la compilación eficiente de los programas (a código nativo), controles de seguridad y la interacción con el sistema operativo.

La naturaleza de bajo nivel y la interdependencia de estas funcionalidades provocan que los entornos de ejecución sean, en general, artefactos complejos [Hau+09]. El desempeño del lenguaje depende de la implementación y eficiencia de su máquina virtual, esto conlleva a que la mayoría de las soluciones de uso comercial suelen ser construcciones estáticas que priorizan el desempeño sobre la flexibilidad. Por lo tanto, agregar o modificar características en tiempo de ejecución, como por ejemplo, modificar el gestor de memoria, suele ser una tarea muy compleja, dependiente de la máquina virtual que se esta utilizando y muchas veces con protocolos no accesibles desde el lenguaje en sí.

Para que los lenguajes de tipado dinámico [Tra09] puedan lograr alcanzar un desempeño similar a los sistemas estáticamente compilados, las implementaciones de uso general utilizan compiladores dinámicos de código en tiempo de ejecución (Just-In-Time Compilers, JIT desde ahora). Este tipo de compiladores analizan, con diferentes estrategias, el código mientras el programa esta siendo ejecutado, detectando posibles piezas o fragmentos del mismo que puedan ser compilados a código nativo con el fin de incrementar su desempeño. Podemos encontrar los orígenes de esta tecnología en las primeras implementaciones de LISP y Smalltalk-80 [Ayc03]. En la actualidad existen dos familias predominantes de compiladores JIT, aquellas que se basan en especialización de métodos y otras basadas en detección y optimización de trazas [MD15].

Recientemente se introdujeron las máquinas virtuales reflexivas [CGM16], entornos de ejecución con la capacidad de ser modificados en tiempo de ejecución mediante interfaces que exponen la estructura y el comportamiento interno a las aplicaciones que ejecutan. Estas implementaciones, soportan escenarios adaptativos en tiempo de ejecución que incluyen desde simples cambios en la semántica del lenguaje a re-implementaciones completas de

alguno sus componentes. En consecuencia, estos entornos pueden favorecer a soluciones de software que puedan explotar esta flexibilidad de forma inteligente logrando que el entorno de ejecución se adapte al problema que se esta tratando de modelar.

MATE [Cha+18b] es una arquitectura particular de máquina virtual reflexiva basada en un protocolo de meta-objetos que permite realizar adaptaciones a ciertos aspectos la misma en **tiempo de ejecución**. TruffleMATE [Cha+18a] es una implementación de MATE que permite realizar operaciones sobre un subconjunto de componentes internos de la máquina virtual, suficientes para mejorar el estado del arte respecto a escenarios de adaptación dinámica. Esta solución utiliza Graal [Gral] para mejorar su rendimiento mediante la utilización de técnicas de meta-compilación por evaluación parcial [Fut99]. TruffleMATE demostró que es posible construir una máquina virtual reflexiva sin una perdida considerable de rendimiento [Cha+18a].

Si bien existen resultados que demuestran que es posible construir máquinas virtuales clásicas usando compiladores *JIT* de diferentes familias con desempeños asintóticamente similares [MD15], no existen estudios para el caso particular de las máquinas virtuales reflexivas.

En este trabajo exploramos la factibilidad de implementar una máquina virtual reflexiva con soporte para MATE utilizando compiladores de *meta-trazas*, con la hipótesis de obtener resultados similares, en términos de desempeño, a los presentados con TruffleMATE. Para el desarrollo de esta nueva máquina virtual extendimos una máquina virtual clásica de Smalltalk construida utilizando el framework *Pypy*. Esta herramienta no solo simplifica la construcción una máquina virtual, sino que también facilita la inclusión de un compilador *JIT* basado en *meta-trazas* para incrementar su rendimiento. Esta máquina virtual fue modificada para ser equiparable en términos de funcionalidades a TruffleMATE, para lo cual fue necesario realizar varias modificaciones a la implementación base para soportar el protocolo de *meta-objetos* propuesto por MATE, así como realizar diversas optimizaciones en la forma en que el código es interpretado. Entre las optimizaciones más importantes destacamos la utilización de *Dispatch-chains* [Woe+14] (una generalización de *Polymorphic inline caches*[HCU91]) para evitar repetir operaciones recurrentes dentro del marco de la implementación de *MATE*.

Luego evaluamos ambas soluciones mediante una serie de pruebas que buscan medir el rendimiento asintótico en diferentes contextos. Por un lado, realizamos pruebas para medir el impacto inherente de disponibilizar el soporte para *MATE* sin que el mismo sea utilizado. Por otro, implementamos dos posibles casos de uso de las capacidades reflexivas de MATE: una solución para asegurar inmutabilidad de objetos y otra para rastrear la ejecución del código en un contexto determinado.

Como resultado de nuestro trabajo, encontramos que nuestra implementación de MATE mediante compiladores de *meta-trazas*, tiene un rendimiento asintóticamente similar a la implementación TruffleMATE, siguiendo la misma línea de los estudios previos comparando ambas familias de compiladores *JIT* [MD15].

Nuestra tesis esta organizada de la siguiente forma: Primero presentaremos los antecedentes relacionados a nuestro estudio, destacando aquellos que fueron la base fundamental en el desarrollo de nuestra solución. A continuación explicaremos la implementación haciendo énfasis en los detalles particulares de nuestra versión en relación a TruffleMATE, así

como las dificultades que encontramos durante el proceso de desarrollo. Luego pasaremos a explicar el proceso de experimentación utilizado así como una presentación de los resultados y conclusiones sobre los mismos. Finalmente cerraremos este trabajo con algunas conclusiones generales que nos dejó el estudio de este tema y la construcción de nuestra solución. Adicionalmente en el anexo se puede encontrar el detalle de todas las ejecuciones de los experimentos realizados.

2. ANTECEDENTES

2.1. Contexto

La eficiencia en términos temporales con la que corren los programas es un aspecto relevante de estudio en el dominio de las máquinas virtuales. Debido a la flexibilidad que presentan la familia de lenguajes dinámicos, sus entornos de ejecución son generalmente menos eficientes que las soluciones construidas para lenguajes compilados. Esta falencia de los lenguajes dinámicos se puede atribuir en parte a la imposibilidad de la máquina virtual de optimizar de forma eficiente la ejecución del programa debido a la falta de información de tipos y a la indirección adicional generada por si misma [Tra09].

Con el fin de lograr desempeños similares a los sistemas construidos con lenguajes compilados, las máquinas virtuales de lenguajes dinámicos buscan compilar código durante la ejecución del programa (Just-in-time). Estas tecnologías se integran a la máquina virtual y analizan el código mientras se esta ejecutando, con el fin almacenar información y detectar patrones en el código que puedan ser compilados posteriormente a código de máquina [Ayc03]. Existen diferentes enfoques para decidir cuándo y qué sección del código se deberá compilar, entre los más utilizados podemos distinguir:

- Compilación por método.
- Compilación por trazas.

Los compiladores que encontramos en la primer familia optimizan la ejecución mediante la compilación de métodos o sub-árboles del AST (árbol abstracto de sintaxis), se basan en el análisis de tipos utilizados con frecuencia y en la generación de diferentes versiones del código generado según estos. Como en general las llamadas a dichos métodos suelen con tipos muy similares, esta estrategia es muy efectiva mejorando el rendimiento en ordenes de magnitud. Los primeros compiladores *JIT* implementados formaban parte de esta familia, destacando las versiones implementadas para *LISP* y *Smalltalk-80* por sus ideas precursoras en el tema [Ayc03]. En la actualidad la mayoría de las máquinas virtuales comerciales que implementan compiladores *JIT* basan su diseño en la solución propuesta por *Self* [CU89] durante inicios de la década de los 90. Entre las más utilizadas en la industria podemos distinguir la máquina virtual *HotSpot* para Java [Kot+08] y *V8* [GoV8] para Javascript.

Por otro lado, la segunda familia de compiladores basan su estrategia en analizar las trazas de ejecución del programa huésped con el fin de detectar segmentos de código que se ejecutan "frecuentemente", en general asociadas a ciclos [Van+15]. El compilador JIT trabaja en conjunto al interprete de la máquina virtual analizando trazas de código, buscando aquellas que se ejecutan reiteradas veces de forma iterativa. Luego de detectar dichas trazas, el compilador *JIT* genera una versión compilada de la misma y esta reemplaza al código interpretado original en la ejecución del programa. Adicionalmente, al bloque de código compilado se lo protege con diferentes guardas, en general de tipos, para asegurar la correctitud del mismo. En posteriores capítulos, y debido a la relevancia para nuestro

trabajo, se explicará con detalle el funcionamiento de este tipo de compiladores y en especial *Pypy* [Pypy], el framework utilizado para la construcción de nuestra máquina virtual reflexiva. Si bien podemos encontrar algunos estudios que abarcaron ideas similares a esta familia de compiladores *JIT* durante la década del 70' [Mit70], no fue hasta el siglo XXI donde esta tecnología se comenzó a utilizar de forma masiva, por ejemplo en máquinas virtuales comerciales como TraceMonkey [TrcM] o *Pypy* [Pypy].

Realizar estudios empíricos sobre máquinas virtuales tiene un costo excesivo en términos de desarrollo de infraestructura debido a la naturaleza compleja y demasiado abarcativa de estos sistemas. La construcción de una máquina virtual desde cero que se optimice utilizando cualquiera de los dos enfoques mencionados anteriormente es una tarea compleja que suele requerir una gran cantidad de recursos y tiempo de desarrolladores expertos. Las tecnologías de meta interpretación ayudan en la implementación de lenguajes eficientes con mucho menor esfuerzo dado que facilitan la inclusión de funcionalidades relativamente complejas en la máquina virtual. En estos casos, el desarrollador generalmente solo implementa el intérprete con la semántica del lenguaje y el framework produce una máquina virtual con un *Garbage Collector* y un compilador JIT automáticamente. Podemos destacar dos frameworks de meta intérpretes: para la primer familia de compiladores a Truffle/Graal [Truf], mientras que para la segunda a *Pypy* [Pypy].

Truffle aplica la técnica de evaluación parcial [Fut99] en la versión especializada del intérprete para determinar la unidad de compilación que se corresponde con un fragmento relevante del programa, la cual luego es optimizada y compilada a código nativo usando optimizaciones clásicas de los compiladores, tales como *in-lining* y *escape-analysis*. El usuario de este framework debe guiar, mediante un *DSL* específico definido por *Truffle*, cuando y de qué forma se deben especializar las distintas unidades de código según la información de tipos recopilada durante la ejecución del programa [WW12].

Pypy recurre al análisis de trazas ejecutadas por el intérprete para detectar código a ser potencialmente compilado. Las trazas corresponden a unidades de código que se ejecutan frecuentemente (generalmente ciclos) en el programa huésped. Las máquinas virtuales construidas con *Pypy* realizan la detección de estos patrones de código mediante simples ayudas adicionadas al intérprete del lenguaje. De esta forma con muy poco esfuerzo y modificaciones se puede dotar al intérprete de un compilador JIT [Bol+09].

Un experimento previo sobre un subconjunto de instrucciones del lenguaje Smalltalk [MD15] demostró que, para cierta batería de pruebas, las características de eficiencia de máquinas virtuales implementadas mediante los dos enfoques mencionados presentan un rendimiento similar y comparable a otras máquinas virtuales muy utilizadas en la industria.

2.2. Entornos de ejecución para lenguajes dinámicos

La flexibilidad que presentan los lenguajes dinámicos en contraposición de los lenguajes compilados no son pocas, y en los últimos años la industria a adoptado estas tecnologías para el desarrollo de gran número de soluciones. La velocidad de desarrollo sumado a su flexibilidad, ecosistema y la proliferación de tecnologías web, entre otras, provocaron que los lenguajes dinámicos se conviertan en opciones muy populares entre ingenieros de

software.

En general, los programas escritos en lenguajes dinámicos no son ejecutados directamente por el hardware subyacente, sino que son interpretados por una máquina virtual. Si bien, hay excepciones a esta regla [Clasp][Fact][Cyth], la mayoría de los lenguajes dinámicos populares, siguen este diseño. Esta "doble" interpretación conlleva a una pérdida de desempeño significativo que limita el alcance de las soluciones con esta familia de lenguajes.

Durante años se idearon diferentes estrategias con el fin de incrementar la velocidad estos lenguajes, entre las cuales podemos destacar:

- Optimizaciones de intérpretes.
- Inclusión de compiladores en tiempo de ejecución, *Just-In-Time*.

La primer estrategia consiste en realizar diferentes optimizaciones con el fin de reducir al mínimo el costo de interpretación de instrucciones del programa. Aunque existen diferentes técnicas para optimizar operaciones como *method-lookup* o *unboxing/boxing* de datos primitivos, para la mayoría de los lenguajes dinámicos, el costo de interpretación y la indirección adicional sigue siendo el factor decisivo en términos de velocidad.

Por otro lado, los compiladores JIT [Ayc03] demostraron ser una forma muy efectiva para atacar problemas de desempeño introducidos por esta interpretación adicional. La característica principal de los compiladores JIT es la generación, cuando es necesario, de código de máquina durante el tiempo de ejecución del programa, generalmente detectando fragmentos de código que se ejecutan frecuentemente.

Si bien el desarrollo de una máquina virtual con compiladores JIT parece ser una solución efectiva para la implementación de lenguajes dinámicos, durante mucho tiempo su construcción fue muy compleja, requiriendo mucho esfuerzo, tiempo y recursos. Por suerte durante los últimos años surgieron diferentes herramientas que nos permiten implementar con menor esfuerzo máquinas virtuales que incluyan compiladores *JIT*. Con estas nuevas herramientas el desarrollador de la máquina virtual puede enfocarse en la implementación del interprete del lenguaje, y luego mediante diferentes anotaciones provee de información al framework para que incluya automáticamente un compilador JIT.

Entre este tipo de herramientas podemos destacar a *Truffle* [Truf] y *Pypy* [Pypy], la primera utiliza técnicas basadas en evaluación parcial y especialización; y la segunda en detección y compilación de trazas de ejecución. En este trabajo nos enfocaremos en explicar esta segunda familia, dado que fue la utilizada para la construcción de nuestra máquina virtual. Primero explicaremos los conceptos en los que se basan los compiladores de trazas, para luego detallar el framework Pypy y la implementación de nuestra máquina virtual reflexiva.

2.2.1. Compiladores de trazas

Los compiladores de trazar son construidos en base a los siguientes supuestos:

- Los programas pasan la mayor parte del tiempo ejecutando ciclos.
- Iteraciones secuenciales del mismo ciclo tomarán caminos de código similares.

La idea general de los compiladores de trazas es generar código compilado para los ciclos frecuentemente ejecutados e interpretar el resto del programa. A su vez, el código compilado generado para estos ciclos es fuertemente optimizado utilizando diferentes técnicas, tales como *Allocation-Removal* [Bol+11] o *Redundant-Guard-Removal* [ABF12].

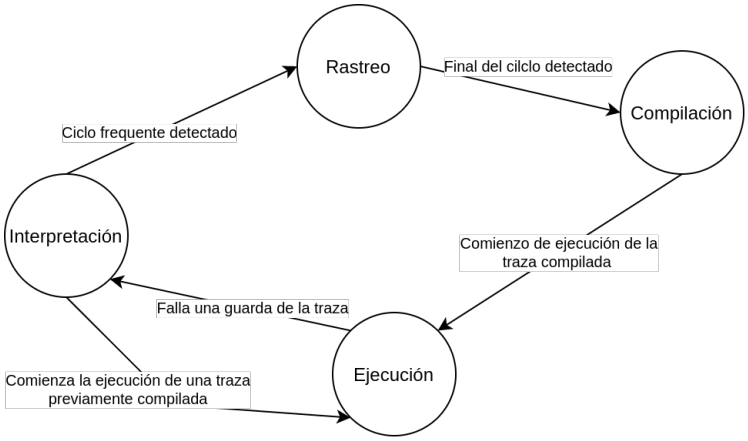
Una máquina virtual con un compilador de trazas cambia entre diferentes estados cuando se encuentra ejecutando un programa:

- **Interpretación**: Cuando el programa comienza a ejecutar, todo el código es interpretado. El intérprete se comporta como cualquier otro, con código adicional para analizar y detectar ciclos frecuentes. Para la detección de los mismos, asocia un contador a cada uno, el cual es incrementado cada vez que se ejecuta un salto al comienzo del mismo. Cuando el contador alcanza un límite pre-establecido, la máquina virtual pasa al estado de **Rastreo**.
- **Rastreo**: Durante esta fase, el intérprete comienza a grabar un listado de todas las operaciones que va ejecutando hasta completar una iteración del ciclo marcado para ser analizado. Para detectar la finalización del ciclo, el intérprete busca una operación que provoque que la ejecución vuelva a una posición previamente alcanzada dentro del historial de operaciones. Este listado de operaciones se denomina traza y, luego de ser generadora, es transferida al compilador *JIT*.
- **Compilación**: El objetivo del compilador es convertir la traza previamente detectada en código nativo fuertemente optimizado. Antes de compilar la traza, la misma es optimizada con el fin de minimizar la cantidad de operaciones y disminuir las indirecciones. Una vez generado el código correspondiente a la traza, este puede ser utilizado inmediatamente en la próxima iteración del ciclo asociado. Para asegurarse que la traza sea válida en todo momento, esta es protegida con diversas guardas, generalemente en puntos donde la ejecución interna del ciclo pueda bifurcarse.
- **Ejecución**: Aquí, el código del compilador *JIT* es ejecutado. Dado que representa un ciclo, el mismo será utilizado repetidas veces hasta que alguna condición de salida sea ejecutada, ya sea por haber finalizado el ciclo o porque algunas de las guardas fue violada. Luego la máquina virtual vuelve a estar en la fase de interpretación comenzando nuevamente con todo el proceso.

Al ser secuencial, una traza sólo representa uno de los muchos posibles caminos que puede tomar la ejecución del ciclo. Como mencionamos anteriormente, para asegurar la correctitud de la ejecución, la traza es enriquecida con guardas en cada punto donde el código pudiera haber tomado otro camino. Cuando se genera el código de máquina, cada guarda es transformada en un chequeo rápido que asegura que el camino que se está ejecutando aún es válido. Si una de estas guardas falla, la máquina virtual inmediatamente aborta la ejecución del código de máquina y vuelve a la fase de interpretación.

Dependiendo de la técnica de rastreo, cada traza puede tener asociado un contador de fallas, similar al contador de ciclos mencionado previamente. Este contador es incrementado cada vez que la guarda falla hasta alcanzar un límite, en este caso se considera frecuente el código ejecutado al fallar la traza y con lo cual el mismo se compila como un camino alternativo en la ejecución del ciclo. Debido a este fenómeno la traza pierde su condición de linealidad y migra a un comportamiento más arbóreo.

A continuación se presenta la máquina de estados de un caso simple (sin contador de guardas fallidas):



Como se mencionó anteriormente, una vez que la traza es completada, la máquina virtual pasa a la fase de compilación. En esta fase se intenta generar código de máquina muy eficiente para que pueda ser ejecutado en los ciclos de la iteración que se acaba de rastrear. Como la traza representa un único y concreto ciclo de iteración es necesario generalizarla para que la misma pueda ser reutilizada en subsiguientes iteraciones. Debido a esto, siempre hay tensión entre tratar de generalizar lo más posible y generar código eficiente:

- Si generalizamos demasiado podemos llegar a terminar con código de máquina nativo poco eficiente.
- Si no generalizamos lo suficiente, el código compilado no se podrá reutilizar para las siguientes iteraciones.

Encontrar el punto de generalización justo es un trabajo crucial en la implementación de compiladores *JIT* para obtener código eficiente y al mismo tiempo maximizar el tiempo que se invierte en la ejecución del mismo.

2.3. Entornos de ejecución reflexivos

2.3.1. Definición

Como mencionamos anteriormente, generalmente las máquinas virtuales de uso masivo se presentan al programador como una caja negra que no puede ser inspeccionada y mucho menos modificarse. Estas implementaciones están optimizadas para la ejecución generalizada de diferentes sistemas de software, con resultados exitosos en la mayoría de los casos. El software cambia y se complejiza a medida que pasa el tiempo, y estas soluciones 'generales' pueden llegar a limitar la capacidad expresiva del sistema modelado y dificultar su evolución. Cada vez es más frecuente el requisito de poder romper la barrera que separa el sistema de su entorno de ejecución, para que el mismo se adapte al software que ejecuta y sirva a sus necesidades [Ghe11].

A diferencia de los entornos de ejecución mencionados, los entornos de ejecución reflexivos (desde ahora *EER*) se abren a la posibilidad de ser modificados en **tiempo de ejecución** por el software que ejecutan según este lo requiera. Debido a esto, los *EER* no son una caja negra, presentan una interfaz mediante la cual pueden ser inspeccionados y modificados desde el sistema huésped [CGM17].

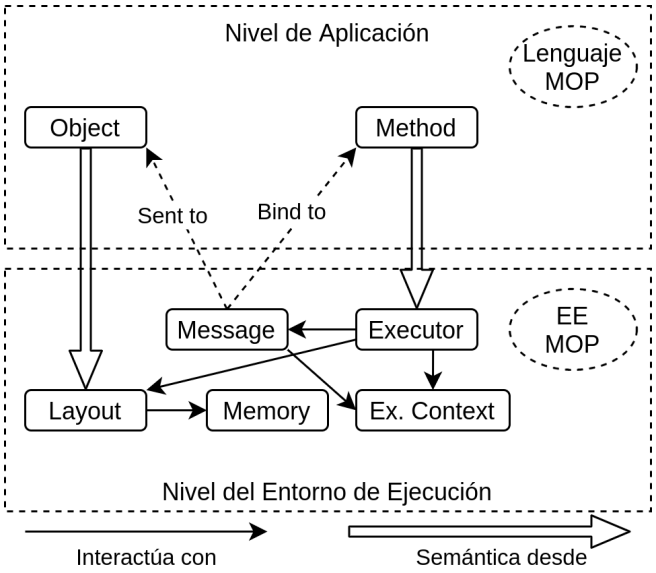
Con el fin de separar responsabilidades, la aplicación deberá enfocarse en resolver los problemas de dominio, mientras que los temas ortogonales relacionados a capacidades reflexivas deberán ser provistos de forma separada por el entorno. Es importante que exista una clara separación entre las abstracciones necesarias para la implementación de las capacidades reflexivas y la aplicación huésped.

En los capítulos siguientes se explicará brevemente la arquitectura de *MATE*, diseñada en trabajos de investigación previos [CGM17], que utilizaremos como base para la implementación de nuestro entorno de ejecución reflexivo.

2.3.2. MATE

Como el objetivo de nuestro estudio es evaluar la posibilidad de construcción de un entorno de ejecución reflexivo mediante el uso de compiladores de *metra-trazas* y luego compararlo con una solución previa basada evaluación parcial, es necesario que utilicemos el mismo modelo reflexivo utilizado en esta.

La arquitectura que utilizamos como base para el desarrollo nuestra máquina virtual fue *MATE*, presentada originalmente en [Cha+18b] y utilizada en la implementación de *TruffleMate*. *MATE* se basa en la exposición de diferentes entidades subyacentes en la máquina virtual mediante un protocolo de *meta-objetos* (desde ahora MOP) como podemos observar en la siguiente figura:

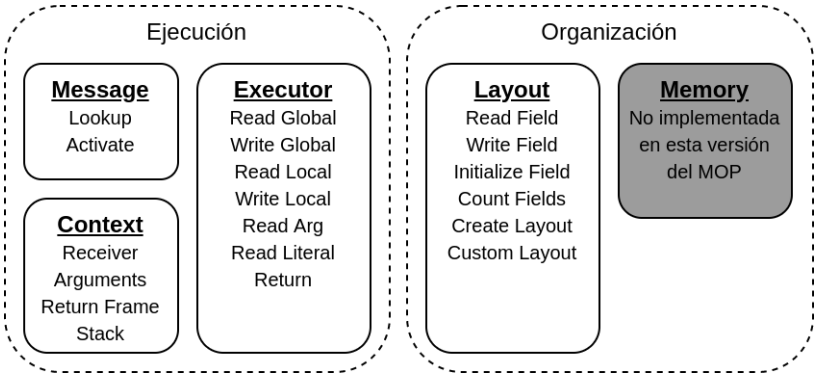


Como se observa en la figura, la arquitectura esta dividida en dos capas, una representando el nivel de aplicación y la otra la máquina virtual, con flechas representando diferentes interacciones entre los componentes. Para representar la mayor cantidad de lenguajes

orientados a objetos, *MATE* solo utiliza objetos y métodos como elementos principales, y propone un protocolo de meta-objetos como interfaz para utilizar las capacidades reflexivas de la máquina virtual. Este protocolo expone a cada una de la entidades de la máquina virtual (representadas en la capa inferior de la figura) a través de una (o varias) *meta-clases* en el lenguaje huésped. A su vez, estas clases presentan una interfaz sencilla que permite modificar el comportamiento del entorno en tiempo de ejecución. En el siguiente capítulo explicaremos en detalle el protocolo de *meta-objetos*.

2.3.3. Protocolo de meta-objetos

Las *meta-clases* responsables de la implementación del *MOP* pueden ser categorizadas en dos grupos: uno relacionado al contexto de ejecución y el otro a la organización de los datos, como podemos observar en la siguiente imagen:



Las meta-clases dividen las responsabilidades de la siguiente forma:

- **Message:** Permite a los desarrolladores modificar el algoritmo de *method-lookup* y el mecanismo de activación de los métodos.
- **Executor:** Permite redefinir el comportamiento y semántica de cada operación definida por el lenguaje. Por ejemplo, el acceso a variables, lectura de argumentos o retorno de valores por los métodos.
- **Execution context:** Permite observar el contexto de ejecución durante la activación de un método, por ejemplo: el receptor del mensaje, las variables locales, etc.
- **Layout:** Provee un mecanismo para modificar el comportamiento de las operaciones que realizan interacción con la estructura interna de los objetos. Permitiría, por ejemplo, observar o modificar como sus atributos internos son almacenados en memoria.

Cada *meta-clase* provee un conjunto de métodos que el programador podrá re-implementar para acceder a las capacidades reflexivas de la máquina virtual. Para la utilización de este *MOP*, el programador deberá crear una una subclase de alguna de las *meta-clases* mencionadas anteriormente redefiniendo el método específico al comportamiento que quiere adaptar.

Para utilizar los *meta-objetos* que redefinen ciertas capacidades reflexivas de la máquina virtual, el programador deberá agrupar estos objetos dentro de un *meta-entorno* e insta-

larlo en el contexto correspondiente. El *MOP* permite la instalación de este *meta-entorno* en dos niveles diferentes: en un objeto en particular o bien dentro del contexto de ejecución de un método.

Por ejemplo, supongamos que el usuario de *MATE* desea modificar el comportamiento del *method-lookup* para un objeto en particular. Para ello debería realizar los siguientes pasos:

- Definir un *meta-objeto* que implementente el protocolo definido por *MATE* para sobrescribir la operación *method-lookup*.
- Crear un nuevo *meta-entorno* conteniendo al *meta-objeto*.
- Instalar el *meta-entorno* en el objeto deseado.

A continuación se ejemplifica con detalle este procedimiento.

Primero definimos el nuevo *meta-objeto* que cambia la semántica del *method-lookup*:

```
1 MessageSendSemanticsMO = MessageMO (
2     find: aSymbol since: aClass = (
3     ...
4     ...
5     ...
6     ↑method.
7 )
```

Paso a paso:

- *Línea 1:* Heredamos nuestro *meta-objeto* de la *meta-clase* definida en el framework asociada a la entidad responsable de resolver la búsqueda de los métodos.
- *Línea 2:* Sobrescribimos el método asociado al *method-lookup* con la interfaz definida por *MATE*. La llamada recibe el nombre del método a buscar y la clase donde se inicia la búsqueda.
- *Lineas 3, 4 y 5:* Aquí debería estar nuestra implementación especial del *method-lookup*.
- *Línea 6:* Finalmente y respetando la interfaz de *MATE*, devolvemos el método a utilizar.

A continuación es necesario definir un *meta-entorno* conteniendo al nuevo *meta-objeto*:

```
1 environment := EnvironmentMO
2     operationalSemantics: nil
3     message: MessageSendSemanticsMO new
4     layout: nil.
```

Finalmente instalamos el *meta-entorno* dentro de objeto a modificar:

```
1 object installEnvironment: environment.
```

Tanto *TruffleMate* como *RTruffleMate* implementan exactamente la misma interfaz, permitiendo que todo el código diseñado para evaluar correctitud y rendimiento entre ambas soluciones pueda ser re-utilizado. Por otro lado, debido a la complejidad de su implementación, las versiones actuales de ambas máquinas virtuales no permiten alterar el comportamiento de las entidades relacionadas a la administración de la memoria de la máquina virtual.

La implementación del protocolo de *meta-objetos* fue realizada mediante la utilización de *intercession-handlers* (IHs). En las siguientes secciones relacionadas a nuestra implementación se detalla este mecanismo muy relevante en ambas soluciones de *MATE*.

3. IMPLEMENTACIÓN DEL ENTORNO DE EJECUCIÓN REFLEXIVO

Siguiendo las ideas presentadas en antecedentes, en esta sección vamos a explicar el desarrollo de nuestra máquina virtual reflexiva. En las primeras secciones vamos a presentar una alternativa a la familia de compiladores de trazas explicados previamente, detallando en particular el framework *Pypy*, el cual utilizamos para nuestra implementación. Luego vamos a exponer los detalles acerca de la construcción de *RTruffleMATE*, destacando aquellas técnicas que utilizamos para optimizar su desempeño, así como las singularidades de nuestra solución en relación a su homónima en *Truffle*. Al final de esta sección se podrá encontrar enlaces a todos los artefactos construidos durante el desarrollo de este trabajo.

3.1. Compiladores de meta-trazas: Pypy

Como se mencionó previamente, la construcción de una máquina virtual eficiente que contenga un compilador *JIT* es un trabajo complejo y requiere un equipo especializado. Por fortuna, en los últimos años aparecieron frameworks que facilitan este proceso, brindándonos un conjunto de herramientas que nos permiten construir máquinas virtuales con compiladores *JIT* integrados con mucho menos esfuerzo.

Para nuestra implementación decidimos utilizar *Pypy* [Pypy], una herramienta con varios años de maduración que nos permitió desarrollar una máquina virtual eficiente con un compilador *JIT* integrado. La construcción de una máquina virtual mediante *Pypy* se realiza implementando el intérprete en *RPython* (un lenguaje muy similar a *Python* pero con ciertas limitaciones) y enriqueciendo el código fuente con anotaciones que le permiten a esta tecnología, entre otras cosas, optimizar el intérprete e incluir el compilador *JIT* en la máquina virtual final.

Hay una diferencia sustancial entre el enfoque de *Pypy* y los compiladores por trazas convencionales. Cuando en los compiladores convencionales las trazas son generadas almacenando un listado de operaciones correspondientes al programa del usuario, en el caso de *Pypy*, estas se generan a partir de las instrucciones ejecutadas por el intérprete. En otras palabras, el compilador *JIT* genera trazas con operaciones del interprete de la máquina virtual (a nivel "meta", de aquí el nombre de *metra-trazas*). Esta estrategia permite a la herramienta generar compiladores *JIT* para todos los intérpretes que puedan ser implementados mediante este framework.

Desafortunadamente, no es posible aplicar a ciegas las técnicas de los compiladores de trazas convencionales, cada iteración del ciclo de interpretación corresponde a la ejecución de solo una instrucción del programa del usuario, y es muy poco probable que el intérprete ejecute exactamente la misma instrucción muchas veces seguidas. Como consecuencia, nunca se va a generar una traza lineal "estable" que se repita durante varias iteraciones. Para confrontar este problema, los compiladores de *meta-trazas* intentan rastrear los ciclos

del programa usuario mediante trazas dentro del código del interprete, las cuales en la mayoría de los casos corresponden a varias iteraciones del ciclo de interpretación.

Como mencionamos, un ciclo del programa del usuario incluye muchas iteraciones del ciclo principal del intérprete. Entonces, para poder generar una traza de una iteración de un ciclo del usuario, bastaría con "desenrollar" los ciclos del interprete correspondientes a las operaciones que forman parte del ciclo del programa. Esta situación es fácilmente detectable por el intérprete simplemente controlando aquellas variables que representan el *program-counter*(PC) dado que esta almacena la dirección o posición de la instrucción que se esta ejecutando y sería fácil detectar cuando se realiza un salto hacia "atrás" en la ejecución del programa huésped. Durante el rastreo de la traza, el intérprete almacena los valores del *PC*, y cuando este toma un valor previamente visto, puede considerar que un ciclo del usuario fue cerrado.

Como el compilador de trazas no tiene ningún tipo de conocimiento acerca de cómo fue implementado el intérprete y, por lo tanto, cómo es construido el *PC*, es necesario indicarle mediante anotaciones cuales de las variables en la implementación del intérprete corresponden al *PC*.

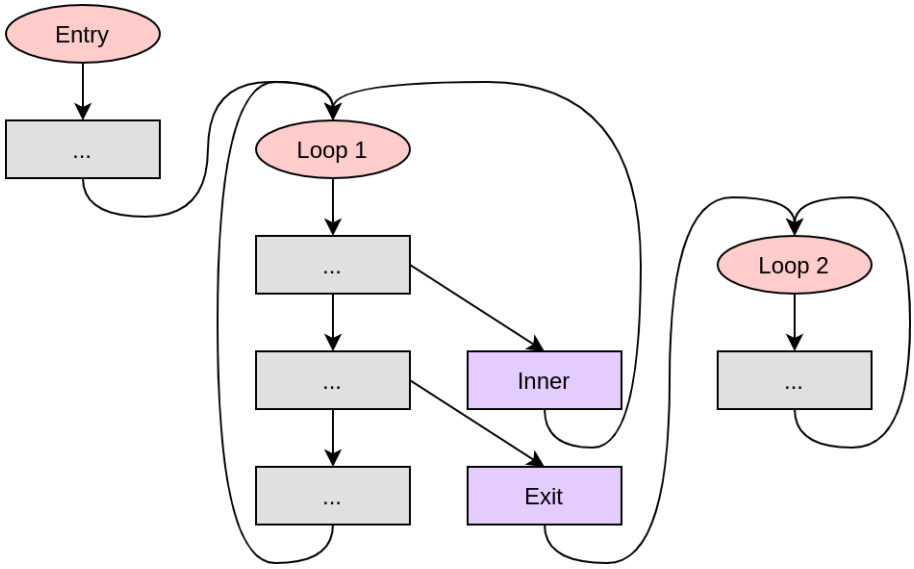
Adicionalmente, y por cuestiones de eficiencia, el intérprete de trazas no puede chequear la posición del *PC* en cada instrucción ejecutada, por lo tanto es necesario indicarle mediante anotaciones aquellas instrucciones que potencialmente pueden provocar saltos hacia "atrás" en la ejecución.

Una vez que un ciclo del usuario es detectado, la estrategia es similar a los compiladores de trazas tradicionales, cuando el contador de "ejecuciones" del ciclo alcanza un limite pre-establecido, la traza es optimizada y posteriormente compilada a código nativo.

Como habíamos mencionado anteriormente, es importante encontrar un equilibrio entre la generación de trazas muy especializadas y trazas muy generales, así como también, considerar la ejecución de caminos alternativos dentro de la ejecución del un mismo ciclo. Para abordar este problema, *Pypy* define cuatro tipos de trazas diferentes:

- **Loop**, son las trazas que representan los ciclos de usuarios en si mismos. La optimización del código compilado para este tipo de trazas es fundamental para obtener buen desempeño en la ejecución del programa.
- **Entry-Bridge**, son trazas que se ejecutan solamente una vez antes de ingresar a una traza *Loop*. Como las trazas de ciclos, tienen cierto grado de generalización. Los *Entry-Bridges* se encargan de la inicialización de las variables y parámetros necesarios para las trazas de ciclos.
- **Inner-Bridge**, son trazas que se generan cuando una guarda dentro del ciclo comienza a fallar de forma reiterada. Son muy útiles cuando un ciclo contiene más de un camino rápido. Luego de la ejecución del mismo, se continúa con la traza original del ciclo desde el inicio.
- **Exit-Bridge**, funcionan de forma similar al *Inner-Bridge*, pero en lugar de volver al ciclo original, saltan a otra traza de ciclo.

En el siguiente esquema podemos observar la relación entre los diferentes tipos de trazas, y la forma que el flujo de ejecución es transferido entre ellos:



Un ciclo de usuario genera una traza lineal mientras que todas sus guardas internas sean válidas. Cuando algunas de ellas comienza a fallar de forma frecuente, se generan trazas "alternativas" conectadas mediante *Inner-Bridges* o *Exit-Bridges* para dotar de cierta "generalidad" a la traza compilada y así lograr mantener un buen desempeño.

El resto del funcionamiento del compilador *JIT* generado por *Pypy* se comporta de forma muy similar a los compiladores de trazas convencionales, intercalando ejecuciones de código interpretado con trazas compiladas como fue detallado en la sección anterior.

Adicionalmente es necesario que el intérprete de la máquina virtual tenga buen desempeño dado que el programa del usuario puede pasar bastante tiempo siendo interpretado hasta que las trazas son detectadas y compiladas. Para lograr esto *Pypy* implementa un sistema de inferencia de tipos que, combinado con un traductor y compilador de C, termina generando un interprete eficiente compilado a código nativo. Este último punto es el principal responsable de las limitaciones que tiene *RPython* en relación a *Python*, dado que permitir demasiado "dinamismo.[en] el código del interprete haría imposible esta tarea.

3.2. Implementación

Para la implementación de nuestra máquina virtual reflexiva usamos como base *RTruffleSOM* [RTSom], una máquina virtual homónima a *TruffleSOM* (utilizada como base en *TruffleMATE*), implementada completamente utilizando *Pypy*. Esta máquina virtual implementa *SOM* (*SimpleObjectMachine*), una variante reducida de Smalltalk muy utilizada para enseñanza e investigación en temas relacionados a la construcción de máquinas virtuales [SOM]. *RTruffleSOM* además del soporte básico para interpretar el lenguaje *SOM*, contiene un compilador *JIT* implementado mediante las facilidades que provee *Pypy*.

Algunas de las características de esta máquina virtual son:

- Clases básicas de *SOM*.

- Entidades de objetos construidas con las estructuras y facilidades disponibles en *Pypy*.
- Utilización del sistema de gestión de memoria provisto por *Pypy*.
- Generación de un *AST* mediante un parser `top-down` implementado completamente en *RPython*.
- Especialización del *AST* en tiempo de ejecución para el acceso a los campos atributos de los objetos.
- Mejoras de desempeño mediante la utilización de *Polymorphic-Inline-caching* [HCU91] implementadas mediante *Dispatch-chains*.

El desarrollo de la nueva máquina virtual reflexiva consistió en los siguientes pasos:

- Implementación de funcionalidades faltantes de *RTruffleSOM* para poder ejecutar las pruebas compartidas con *TruffleMATE*.
- Implementación del soporte para *MATE*.
- Optimizaciones propias de *RTruffleMATE* para mejorar el rendimiento.

En las siguientes secciones vamos a detallar cada uno de los pasos en la implementación de la máquina virtual.

3.2.1. Funcionalidades Faltantes

Si bien el soporte a *SOM* de la máquina virtual base era bastante completo, antes de comenzar a desarrollar su versión reflexiva fue necesario ampliar y modificar la misma para que sea comparable a *TruffleSOM*. Esto fue necesario dado que varias de las pruebas utilizadas en *TruffleMATE* requerían características faltantes en *RTruffleSOM*.

Entre los cambios realizados podemos destacar:

- Se agregó la clase *Char*.
- Se agregó la clase *String*.
- Se agregaron varios métodos faltantes en clases bases.
- Se implementaron diversas clases para soportar la lectura de archivos mediante la jerarquía de clase Stream.

Adicionalmente realizamos cambios menores y fuimos arreglando errores hasta lograr que la máquina virtual pueda correr correctamente en toda la suite de pruebas implementada en *TruffleMATE*.

3.2.2. Intercession Handlers

Los *intercession-handlers* (IH desde ahora) son fragmentos de código ubicados estratégicamente en los puntos de entrada de la máquina virtual donde la semántica de cada operación es implementada. El objetivo de estos es detectar si el comportamiento fue modificado, y

si es necesario, ejecutar la nueva semántica. En cada operación soportada por *MATE*, los IHs buscan la existencia de un *meta-objeto* ya sea en el objeto o en el contexto de ejecución del método. En caso de encontrar un *meta-objeto* definido, en lugar de utilizar la semántica predeterminada por la máquina virtual, delega la responsabilidad de la operación al *meta-objeto*.

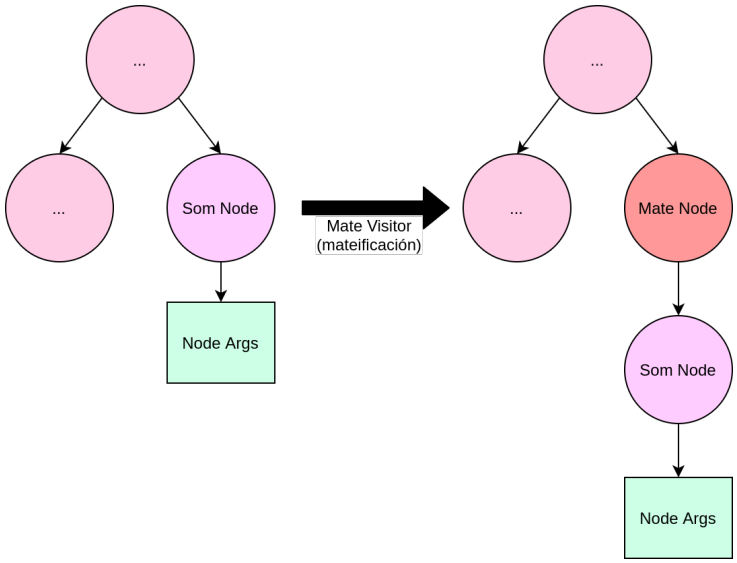
En ambas máquinas virtuales, los IHs fueron implementados como nodos del AST, encapsulando a los originales para controlar la ejecución de las operaciones. Este paso adicional es necesario realizarlo en casi todas las operaciones base de la máquina virtual, por lo tanto el rendimiento general de la máquina virtual depende en gran medida de implementación de los mismos.

3.2.3. Implementación de RTruffleMATE

Como mencionamos anteriormente y siguiendo la línea de desarrollo utilizada por TruffleMATE, para realizar la implementación de los *intercession-handlers*, se encapsularon los nodos originales por nodos especiales de *MATE*. Para este proceso de "Mateificación" nuestra estrategia fue introducir cambios poco intrusivos a la solución actual, y por ello decidimos recurrir al patrón Visitor [Gam+95] para recorrer y modificar el AST agregando los nuevos nodos de *MATE*. Fue necesario implementar en las clases del AST el protocolo necesario para soportar el patrón mencionado dado que el código original de *RTruffleSOM* no poseía soporte para este.

Implementamos nuestro *visitor* para que cuando detecte un nodo 'interesante', es decir, algún nodo que implementa alguna de las operaciones que queremos dotar de reflexión, crea el nodo de *MATE* correspondiente según la operación y reemplaza el nodo original por este, manteniendo el anterior como sucesor del nuevo nodo.

La figura a continuación muestra la transformación que recibe el AST luego del proceso de "Mateificación":



Los nodos de *MATE* fueron creados bajo la misma jerarquía que los nodos normales del AST, implementando la interfaz mínima que era necesaria para formar parte del árbol. El mayoría de los nodos del AST, el método más importante es *execute*, el cual se encarga de ejecutar la operación correspondiente durante la ejecución del programa usuario. Aquí es donde los nodos de *MATE* realizan su función de *intercession-handling* cambiando la semántica de la operación si el usuario así lo configuró. Para cada operación posible de redefinir en *MATE* creamos clases de nodos especificas dado que las interfaces en el *MOP* para cada caso son ligeramente diferentes. Todos estos nodos son subclase de el nodo principal *MateNode* que implementa la lógica base de *intercession-handling*:

```
1  class MateNode(ExpressionNode):
2
3      def execute(self, frame):
4
5          value = None
6          if not frame.meta_level():
7              receiver = frame.get_self()
8              value = self.do_mate_semantics(frame, receiver)
9
10         if value is None:
11             return self._som_node.execute(frame)
12         else:
13             return value
```

Detalle:

- *Línea 1:* Definimos a nuestros Nodos de *MATE* como subclase de *ExpressionNode* para poder formar parte del AST.
- *Línea 3:* Re-implementamos el método *execute* para interceptar la ejecución de la semántica del nodo.
- *Línea 5:* Utilizamos la variable local *value* para almacenar temporalmente el resultado de aplicar una semántica redefinida por *MATE* si es el caso.
- *Línea 6:* Tenemos que asegurarnos de no aplicar *MATE* si ya estamos ejecutando código dentro del *meta-level* de *MATE*. Sino hacemos esto entramos en un bucle infinito.
- *Línea 7:* Buscamos el receptor original del mensaje u operación.
- *Línea 8:* Intento aplicar la semántica de *MATE*, en caso de no estar redefinida, *do_mate_semantics* retorna *None*.
- *Línea 10:* Chequeo si la semántica del nodo fue redefinida por *MATE*.
- *Línea 11:* Sino, delego la operación al nodo original almacenado en el atributo *_som_node*.
- *Línea 13:* Si fue redefinida, retorno el valor retornado por la nueva semántica.

La implementación del método *_do_mate_semantics* es la siguiente:

```
1  class MateNode(ExpressionNode):
2
3      def do_mate_semantics(self, frame, receiver):
4
5          environment = frame.get_meta_object_environment() or receiver.
               get_meta_object_environment()
6          if environment is None:
7              return None
8
9          method = self._lookup_meta_invokable(environment)
10         if method is None:
11             return None
12
13         args = self.get_meta_args(frame)
14         return method.invoke_to_mate(receiver, args,  frame)
```

Detalle:

- *Línea 1:* Definimos a nuestros Nodos de MATE como subclase de *ExpressionNode* para poder formar parte del AST.
- *Línea 3:* La función que ejecuta la semántica de MATE recibe el entorno de ejecución y el objeto receptor del mensaje/operación como parámetros.
- *Línea 5:* Buscamos el *meta-entorno* de MATE primero en el entorno, y si no esta definido, en el objeto.
- *Línea 6:* Controlamos la existencia de un *meta-entorno* de MATE definido.
- *Línea 7:* Sino hay ningún *meta-entorno* definido abortamos la ejecución del método.
- *Línea 9:* Buscamos si existe algún método que redefina la semántica para operación que se esta ejecutando.
- *Línea 10 y 11:* Sino esta redefinida, abortamos el método retornando *None*.
- *Línea 13:* Delegamos al tipo de nodo MATE especializado la generación de los parámetros para el *meta-método*.
- *Línea 14:* Invocamos al *meta-método* con los parámetros calculados y retornamos su resultado.

Para almacenar el *meta-entorno*, en ambos casos, decidimos simplemente agregar un campo adicional en las entidades *Object* y *Frame* previamente existentes en la versión original de *RTruffleSOM*. Luego fue necesario exponer nuevas primitivas para que sea posible acceder y configurar este objeto desde el código huésped según fue definido por el *MOP*.

Como se mencionó anteriormente, debido a las diferentes interfaces que son necesarias según el *MOP*, fue necesario crear nodos de *MATE* específicos cada una de ellas. Los nodos reemplazados según cada operación subyacente de la máquina virtual fueron los siguientes:

- **Message send**: Nodo `UninitializedMessageNode`. Encargado de ejecutar el *method-lookup* para encontrar el método a ejecutar según el objeto receptor.

- **Message activation**: Nodo `Invokable`. Responsable de crear el contexto y ejecutar los métodos.
- **Argument read**: Nodo `LocalArgumentReadNode`. Nodo responsable de la lectura de los parámetros utilizados en el envío de los mensajes.
- **Variable read**: Nodo `UninitializedReadNode`. Este nodo es la versión sin especializar del nodo encargado de la lectura de las variables locales y globales. Luego de una serie de iteraciones, la máquina virtual especializa el nodo para acelerar el acceso a tipos primitivos.
- **Variable write**: Nodo `UninitializedWriteNode`. Idem el nodo anterior pero para el caso de escritura.
- **Field write**: Nodo `FieldWriteNode`. Responsable de la escritura de los colaboradores internos del objeto.
- **Field read**: Nodo `FieldReadNode`. Idem nodo anterior, pero para el caso de lectura.
- **Layout field read**: Nodo `UninitializedReadFieldNode`. Encargado de realizar la lectura del esquema de almacenamiento de los colaboradores internos de un objeto.
- **Layout field write**: Nodo `UninitializedWriteFieldNode`. Idem el nodo anterior pero referente a operaciones de escritura.

Si bien la jerarquía de nodos que utiliza la versión original de *RTruffleSOM* es mucho mayor, con solo implementar los nodos mencionados fue suficiente para proveer a la máquina virtual de las características reflexivas.

3.2.4. Optimizaciones

Uno de los puntos claves en nuestra máquina virtual es identificar rápidamente la existencia de un método de usuario que redefina el comportamiento de alguna entidad subyacente mediante el *MOP*. Para ello recurrimos a la utilización de *Dispatch-chains* para evitar búsquedas consecutivas en un nodo particular para el mismo *meta-entorno*. Algo que consideramos que debería suceder generalmente dado que no esperamos que los *meta-entornos* cambien demasiado.

El nodo inicial de la *meta-Dispatch-chain* es hijo del nodo de *MATE* en el AST, el cual delega en este la búsqueda del *meta-método* mediante una llamada a *_lookup_meta_invokable*:

```
1 class MateNode(ExpressionNode):
2
3     def _lookup_meta_invokable(self, environment):
4         return self._lookup_node.lookup_meta_invokable(environment)
```

La *Dispatch-chain* fue implementada como una lista enlazada de nodos con un limite pre-establecido. Cada nodo contiene una referencia al *meta-entorno*, al *meta-método* y al siguiente nodo de la cadena. La búsqueda del método dentro de la lista simplemente recorre de forma secuencial los nodos hasta llegar el final de la misma, y si no se alcanzó el limite pre-establecido, se crea un nuevo nodo especifico para este caso.

La *Dispatch-chain* fue implementada por la siguiente jerarquía de nodos:

- **UninitializedMateLookUpNode:** Esta clase representa un nodo sin inicializar, ubicándose siempre al final de la lista. Cuando este nodo es alcanzado durante la búsqueda del *meta-método*, se especializa en *_CachedMateLookUpObjectCheckNode* si aún no se alcanzó el limite del tamaño de la lista o en *_GenericMateLookUpNode* de lo contrario.
- **_CachedMateLookUpObjectCheckNode:** Esta clase representa un nodo especializado de la cache y contiene una referencia al *meta-entorno* y al *meta-método* asociado.
- **_GenericMateLookUpNode:** Esta clase representa el nodo final de la lista cuando la misma alcanza el limite pre-configurado, implementa la búsqueda completa del *meta-método*.

Con esta jerarquía, la búsqueda dentro de la cache se realiza de la siguiente forma en cada tipo de nodo:

```
1  class UninitializedMateLookUpNode(_AbstractMateLookUpWithReflectiveOp):
2
3      def lookup_meta_invokable(self, environment):
4          return self._specialize(environment).lookup_meta_invokable(
               environment)
5
6
7  class _GenericMateLookUpNode(_AbstractMateLookUpWithReflectiveOp):
8
9      def lookup_meta_invokable(self, environment):
10         return mate.interpreter.mop.lookup_invokable(self._universe, self.
               _reflective_op, environment)
11
12
13 class _CachedMateLookUpObjectCheckNode(_AbstractMateLookUpNode):
14
15     def lookup_meta_invokable(self, environment):
16         if environment == self._expected_environment:
17             return self._cached_method
18         else:
19             return self._next.lookup_meta_invokable(environment)
```

Detalle:

- *Línea 4:* Este nodo no ejecuta ninguna lógica para la búsqueda del *meta-entorno*, simplemente se especializa en alguno de los otro dos nodos y delega la búsqueda.
- *Línea 10:* Este nodo implementa una búsqueda genérica, buscando dentro del *meta-entorno* si existe el método definido por el MOP asociado a la operación del nodo.
- *Línea 16-19:* Este nodo chequea si el *meta-entorno* es el asociado a este, y en caso positivo, retorna la referencia al *meta-método*. En caso contrario, delega la búsqueda al siguiente nodo en la cadena.

Adicionalmente se realizaron diferentes optimizaciones menores propias de *Pypy* para mejorar el desempeño de la máquina virtual, como por ejemplo, indicar con anotaciones

inmutabilidad de objetos y funciones o modificar los ajustes internos por defecto del compilador *JIT*.

3.2.5. Artefactos implementados

Como resultado de esta tesis implementamos una serie de artefactos y soluciones que están disponibles a toda la comunidad:

- **RTruffleMate**: La máquina virtual reflexiva implementada para esta tesis. La misma fue implementada usando el framework Pypy y su código fuente fue desarrollado en RPython. Pueden encontrar su implementación en `https://github.com/gefarion/RTruffleMate/tree/metaobjectInObject`.
- **Kit de reproducibilidad**: Adicionalmente proveemos un repositorio con instrucciones y facilidades para reproducir todos los experimentos presentados en este trabajo. Este repositorio puede encontrarse en `https://github.com/gefarion/MatePerformance`.

4. EXPERIMENTACIÓN Y EVALUACIÓN

Como mencionamos en anteriores secciones, nuestro objetivo es comprobar si el rendimiento de nuestra solución es comparable con *TruffleMate*. Para ello ejecutamos los mismos experimentos que fueron utilizados en anteriores estudios de esta implementación [Cha+18a] y comparamos sus resultados.

4.1. Metodología

Cada prueba fue ejecutada 200 veces en sucesivas iteraciones, tomando en cada una los tiempos de ejecución tanto de la máquina virtual evaluada como su control (al cual nos referiremos como *baseline*). Para comparar las soluciones definimos la métrica *Factor de Sobrecarga* (desde ahora *FS*) como el promedio de la relación de rendimiento entre la máquina virtual experimento y su control para todas las iteraciones. Vamos a utilizar esta métrica en todos los experimentos para evaluar el rendimiento de nuestra solución salvo que se indique lo contrario.

Como muchas de pruebas que realizamos eran demasiado rápidas, fue necesario en cada iteración de las mismas ejecutar un serie de "iteraciones internas", cuya cantidad depende de cada caso. El tiempo final de la iteración se calculó como la suma de los tiempos de todas estas "iteraciones internas".

Debido al funcionamiento de los compiladores de *meta-trazas*, explicados en anteriores secciones, para obtener resultados limpios del rendimiento de cada prueba, antes de comenzar a tomar las muestras fue necesario dejar que las soluciones "calienten", dándole tiempo al compilador JIT para que pueda realizar las optimizaciones necesarias. Nos referiremos a este periodo inicial como *Warmup* del experimento. Con nuestro análisis, explicado en una sección posterior, encontramos que el compilador JIT provisto por *Pypy* no necesita una gran cantidad de iteraciones de *Warmup*, por lo tanto mediante experimentación consideramos que 50 son suficientes para que el rendimiento se estabilice.

Para facilitar la ejecución de los los experimentos y la toma de las muestras, utilizamos el framework *ReBench*[Rbch]. Por otro lado, construimos un entorno fácilmente replicable mediante la utilización de contenedores *Docker* [Dokr].

Todos los experimentos fueron ejecutados en la siguiente máquina:

- Procesador: Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz.
- Memoria RAM: 20 GB DDR4 a 2133 MHz (4 + 16).
- Arquitectura: Amd64.
- Sistema Operativo: Linux Mint 18.3 Sylvia
- Versión de Docker: 18.09.7, build 2d0083d

En el apéndice de este trabajo se podrá encontrar el detalle de los tiempos de ejecución de todos los experimentos en cada de una de las máquina virtuales utilizadas.

En la mayoría de los experimentos utilizamos un conjunto de pruebas estándar para evaluar rendimiento de máquinas virtuales. A continuación se lista cada uno con una breve descripción:

- **Bounce:** Simulación de rebotes de pelotas dentro de una caja.
- **CD:** Simulación de detección de colisiones de un avión.
- **DeltaBlue:** Implementación orientada a objetos de un *constraint solver* clásico.
- **Havlak:** Algoritmo de reconocimiento de ciclos en grafos.
- **Json:** Un *parser* de cadenas JSON.
- **List:** Creación y recorrido de listas enlazadas.
- **Mandelbrot:** Cómputo de un Conjunto de Mandelbrot.
- **NBody:** Simulación de órbitas de planetas Jovianos.
- **Permute:** Generación de todas las permutaciones posibles de elementos de un array.
- **Queens:** Algoritmo para resolver el problema clásico de las ocho reinas.
- **Sieve:** Simulación de la Criba de Eratóstenes.
- **Storage:** Generación arboles de arrays con el fin de sobrecargar al *Garbage-Collector*.
- **Towers:** Algoritmo para resolver el problema de las Torres de Hanoi.
- **BubbleSort:** Algoritmo clásico de ordenamiento por burbujeo.
- **Dispatch:** Este prueba simplemente envía el mismo método 20000 a un mismo objeto.
- **Fannkuch:** Esta prueba involucra una gran cantidad de operaciones sobre enteros y arrays de enteros.
- **Fibonacci:** Calculo de la serie de Fibonacci utilizando su versión recursiva.
- **FieldLoop:** Acceso reiterado dentro un ciclo del mismo campo de un objeto.
- **Loop:** Computo de la sumatoria de todos los números menores de 100 mediante un ciclo *for*.
- **QuickSort:** Implementación *in-place* de algoritmo de ordenamiento.
- **Recurse:** Esta prueba implemente una simple recursión lineal.
- **WhileLoop:** Suma iterativa de un conjunto lineal de números mediante un ciclo *while*.

Las razones para utilizar este set de pruebas son:

- No requieren ningún tipo de soporte reflexivo de la máquina virtual.
- Al ser muy utilizados, ya existen bastantes implementaciones en otros lenguajes.

- Fueron diseñados para evaluar las partes fundamentales de los lenguajes de tipado dinámico.

4.2. Experimentos

4.2.1. Rendimiento Base

Con este experimento buscamos evaluar el rendimiento de *RTruffleSOM* , el cual utilizamos para nuestra implementación de *MATE*, en relación a otras máquinas virtuales de uso comercial. Para ello utilizamos sub-conjunto de las pruebas mencionadas anteriormente. En este experimento comparamos el rendimiento relativo de *RTruffleSOM* contra *OpenJDK* y *NodeJS*, la primera para el lenguaje Java y la segunda para Javascript. En cada caso, cada una de las pruebas fue implementada en el lenguaje correspondiente.

4.2.2. Sobrecarga Inherente

Con este conjunto de pruebas buscamos medir el impacto que tiene la implementación del soporte de *MATE* sin que este sea utilizado. Gracias a este experimento vamos a poder evaluar el costo computacional de la evaluación de los *intesession-handlers* en la ejecución del programa.

Para estas pruebas comparamos el rendimiento relativo de *RTruffleMATE* en comparación a *RTruffleSOM*, la cual utilizamos como máquina virtual de control al no poseer implementado ninguno de los mecanismos necesarios para *MATE*.

4.2.3. Operaciones Individuales

En estas pruebas buscamos evaluar el rendimiento de *MATE* al reemplazar cada operación relevante de la máquina virtual base con una versión implementada mediante el framework, replicando exactamente el comportamiento original. De esta forma es posible evaluar la pérdida de rendimiento que conlleva el uso de *MATE* sin modificar el comportamiento original de cada operación. Con el fin de realizar una evaluación mas precisa y acotada, implementamos pruebas diferentes para cada tipo de operación.

Para ilustrar un ejemplo de estas pruebas, este es el código que utilizamos para evaluar la operación de lectura de un atributo de objeto:

```
1  FieldRead = Benchmark (
2       | object |
3
4       oneTimeSetup = (
5            Random initialize.
6            object := Pair new.
7            object key: 1.
8       )
9
10      benchmark = (
11           object key: (Random next rem: 100) + 1.
```

```
12          ↑object key.
13      )
14
15      verifyResult: result = (
16        ↑result <= 100
17      )
18  )
19
20
21  FieldReadSemanticsMO = OperationalSemanticsMO (
22      read: anIndex = (↑self instVarAt: anIndex)
23  )
24
25
26  VMReflectiveFieldRead = FieldRead (
27      oneTimeSetup = (
28          | environment |
29          super oneTimeSetup.
30          environment := EnvironmentMO
31                                  operationalSemantics: FieldReadSemanticsMO new
32                                  message: nil
33                                  layout: nil.
34          object installEnvironment: environment.
35      )
36  )
```

Detalle:

- *Línea 1-18:* Esta es la definición del *benchmark*, en cada iteración se genera un numero aleatorio y se asigna a un atributo del objeto. Es necesario aleatorizar el valor asignado para evitar que el compilador optimice demasiado el código llegando posiblemente al punto donde se elimine completamente la operación.
- *Línea 21-23:* Estas lineas crean una subclase de *MATE* para redefinir la lectura de los atributos del objeto manteniendo el mismo comportamiento.
- *Línea 27:* Este método se ejecuta una sola vez al inicio del *benchmark* y lo utilizamos para configurar el *meta-entorno* que vamos a utilizar.
- *Línea 30:* Creamos un nuevo *meta-entorno* y asociamos la nueva semántica de lectura.
- *Línea 34:* Instalamos el *meta-entorno* en el objeto principal del *benchmark*.

Nombramos cada prueba según la operación especifica que se esté re-implementando.

4.2.4. Inmutabilidad

En esta prueba tratamos de simular un posible caso de uso de *MATE*. Implementamos una solución de *Inmutabilidad* a nivel de objetos y sus referencias utilizando las funcionalidades reflexivas que el framework provee.

Por un lado, para asegurar la inmutabilidad de los objetos fue necesario crear un *meta-objeto* que redefina el comportamiento de la operación de escritura para asegurarnos que

esta no cambie el estado del mismo. El código encargado de realizar esto es el siguiente:

```
1 FieldRead = Benchmark (
2 ImmutableSemanticsForHandlesMO = OperationalSemanticsMO (
3     write: anIndex value: aValue = (
4         ↑aValue
5     )
6 )
```

- *Línea 1* Declaramos un nuevo *handler* para redefinir la semántica de escritura de los atributos del objeto.
- *Línea 2-4:* En la nueva semántica de escritura no modificamos el estado del objeto.

A si mismo, para asegurarnos que también sus referencias sean inmutables, fue necesario modificar el acceso a las mismas. Para ello, nuevamente mediante *MATE*, interceptamos la lectura de los atributos y retornamos una versión inmutable de los mismos. El siguiente código ejemplifica esta operación:

```
1 ImmutableSemanticsForHandlesMO = OperationalSemanticsMO (
2     read: anIndex = (
3         ↑(self instVarAt: anIndex) readOnly
4     )
5 )
```

- *Línea 1* Definimos un nuevo *intersession-handler* para modificar la semántica de lectura de los atributos del objeto.
- *Línea 2-4:* Invocamos el método *readOnly* sobre el objeto que se esta intentando acceder. Este mensaje retorna una versión *read-only* del receptor.

Para evaluar nuestra solución, realizamos dos experimentos:

- Ejecutamos un sub-conjunto de las pruebas que utilizamos en los experimentos anteriores pero incluyendo la condición de inmutabilidad en los objetos mediante *MATE*. En este caso, comparamos el rendimiento de la ejecución esta variante con su versión base.
- Comparamos nuestra solución de inmutabilidad contra otra solución equivalente implementada sin hacer uso de las capacidades reflexivas. Esta variante fue construida mediante el uso de *DelegationProxies* [Wer+14]. Para este caso, la prueba consiste en la generación de listas enlazadas de tuplas las cuales luego son iteradas, modificadas y evaluadas para comprobar que se mantenga la condición de inmutabilidad de las mismas.

4.2.5. Trazado de código

En este experimento buscamos evaluar la sobrecarga de *MATE* cuando una gran cantidad de *intersession-handlers* activan la ejecución contante de los *meta-objetos* del framework.

Para medir este comportamiento agregamos un *meta-objeto* al contexto de ejecución. Este *meta-objeto* tiene la particularidad de propagarse así mismo en los nuevos contextos generados tras las activaciones de métodos subsecuentes. Para asegurarnos la correctitud de la prueba incluimos un contador que se irá incrementando a medida que sucesivas llamadas de métodos van ejecutando, realizando una suerte de "Trazado".

Nuestra solución consiste en la creación de un nuevo *meta-objeto* encargado de interceptar las llamadas a los métodos. Este es incluido en un *meta-entorno* para luego instalarlo en el contexto de ejecución que nos interesa evaluar. Adicionalmente, fue necesario propagar este *meta-entorno* hacia todos los nuevos contexto creados en las sucesivas ejecuciones de métodos.

La implementación del *meta-objeto* es la siguiente:

```
1  TracingMessageMO = MessageMO (
2      activate: aMethod withArguments: arguments = (
3          | realArguments |
4          realArguments := arguments copy.
5          realArguments at: 1 put: MessageMO instrumentationMO.
6          Counter addMethodCall.
7          ↑realArguments.
8      )
9      ----------------------------
10     | instrumentationMO |
11     instrumentationMO = (↑instrumentationMO)
12     instrumentationMO: aMetaobject = (
13         instrumentationMO := aMetaobject
14     )
15 )
```

- *Línea 1* Definimos un nuevo *meta-objeto* para reemplazar la semántica de ejecución de métodos.
- *Línea 2:* Implementamos el método *activate* que se activa previamente a la ejecución del método y nos permite modificar algunas características del contexto del mismo. Entre ellas, en el primer argumento podemos definir un *meta-entorno* que se propagara al nuevo contexto de ejecución.
- *Línea 5:* Para propagar el *meta-objeto* simplemente insertamos *instrumentationMO* en la primer posición del array. Este objeto simplemente es un *meta-entorno* que contiene a *MessageMO*.
- *Línea 6:* Incrementamos el contador de llamadas enviando el mensaje *addMethodCall* a la clase global *Counter*.
- *Línea 10-14:* Definimos el atributo de instancia *instrumentationMO* que referencia a un *meta-entorno* que contiene a *self*.

Para este primer experimento ejecutamos un sub-conjunto de las pruebas que se utilizaron en los anteriores: *CD*, *DeltaBlue*, *Json*, *Mandelbrot*, *NBody* y *Quicksort*.

4.3. Resultados

Esta esta sección vamos a presentar los resultados de cada experimento y vamos a compararlo, si es posible, contra resultados de estudios previos relacionados a *TruffleMate*.

4.3.1. Rendimiento Base

Para este experimento ejecutamos la batería de pruebas mencionadas en la sección anterior en tres máquinas virtuales diferentes: *RTruffleSOM*, *OpenJDK*[Ojdk] y *NodeJS*[NdJS]. En cada prueba calculamos el *FS* de *RTruffleSOM* utilizando como *baseline* las dos máquinas virtuales mencionadas anteriormente. Obtuvimos los siguientes resultados:

Tab. 4.1: Rendimiento de RTruffleSOM en comparación a OpenJDK y NodeJS

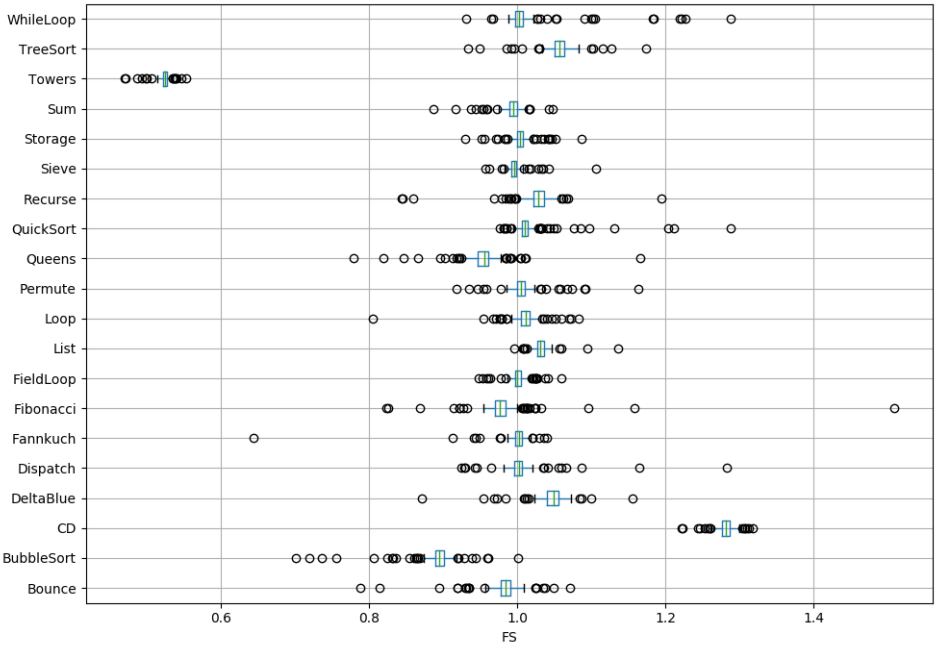
Benchmark	Baseline	FS	Sd.	Min	Max	Median	p95
DeltaBlue	OpenJDK	4.69	0.56	1.71	5.29	4.88	5.22
	NodeJS	1.96	0.27	1.29	2.38	2.06	2.3
Towers	OpenJDK	2.67	0.12	2.3	3.74	2.68	2.76
	NodeJS	1.56	0.08	1.28	2.19	1.57	1.61
Mandelbrot	OpenJDK	2.21	0.03	2.08	2.35	2.21	2.25
	NodeJS	2.46	0.03	2.41	2.68	2.46	2.5
List	OpenJDK	11.71	0.24	10.34	12.67	11.72	11.96
	NodeJS	6.07	0.13	5.13	6.46	6.09	6.19
Bounce	OpenJDK	2.19	0.1	1.82	2.49	2.22	2.3
	NodeJS	0.32	0.01	0.31	0.43	0.32	0.33
CD	OpenJDK	8.93	1.14	2.92	9.55	9.32	9.48
	NodeJS	4.81	0.42	3.63	9.36	4.82	4.95
Json	OpenJDK	2.22	0.23	1.17	3.66	2.24	2.44
	NodeJS	1.71	0.27	1.56	4.8	1.68	1.87
Permute	OpenJDK	6.96	0.33	5.87	9.33	6.93	7.44
	NodeJS	3.6	0.15	3.34	4.64	3.57	3.86
Sieve	OpenJDK	4.52	0.16	3.59	4.92	4.54	4.82
	NodeJS	2.14	0.04	1.94	2.23	2.14	2.17
Richards	OpenJDK	2.29	0.11	2.07	2.83	2.26	2.46
	NodeJS	1.32	0.07	1.19	1.92	1.31	1.4
Storage	OpenJDK	4.3	0.5	2.29	4.76	4.46	4.61
	NodeJS	2.78	0.09	2.51	3.31	2.78	2.89
Queens	OpenJDK	5.93	0.13	5.31	6.4	5.94	6.1
	NodeJS	3.06	0.07	2.73	3.3	3.06	3.17
Havlak	OpenJDK	11.04	1.36	3.9	12.22	11.39	11.95
	NodeJS	1.89	0.06	1.64	2.29	1.88	1.95
NBody	OpenJDK	3.33	0.14	2.38	3.68	3.33	3.5
	NodeJS	1.66	0.05	1.46	1.85	1.66	1.73

Como se puede apreciar en la tabla 4.1, el *FS* promedio de *RTruffleSOM* en relación a *OpenJDK* es de 5.21x y con respecto a *NodeJS* es de 2.52x. Si bien nuestra máquina virtual es más lenta en comparación a las soluciones comerciales, consideramos que tiene un buen desempeño para ser un prototipo.

4.3.2. Sobrecarga Inherente

Para este experimento evaluamos el *FS* de *RTruffleMATE* contra *RTruffleSOM*, la cual utilizamos como *baseline*. Obtuvimos los siguientes resultados:

Fig. 4.1: Sobrecarga Inherente - Factor de sobrecarga



Como se puede observar en la figura 4.1, la versión de *RTruffleMate* muestra *FS* promedio de 0.99x con respecto a la máquina virtual base, siendo *CD* la prueba que registró una mayor perdida de desempeño con 1.28x. TruffleMate en la ejecución de estas mismas pruebas presentó un rendimiento de 0.81x [Cha17] bajo las mismas condiciones de evaluación.

Nuestros resultados indican que al menos en este conjunto de pruebas utilizadas, la presencia de los cambios en nuestra máquina virtual para soportar el protocolo de *MATE* no tiene ningún tipo de impacto significativo en el rendimiento de la máquina virtual.

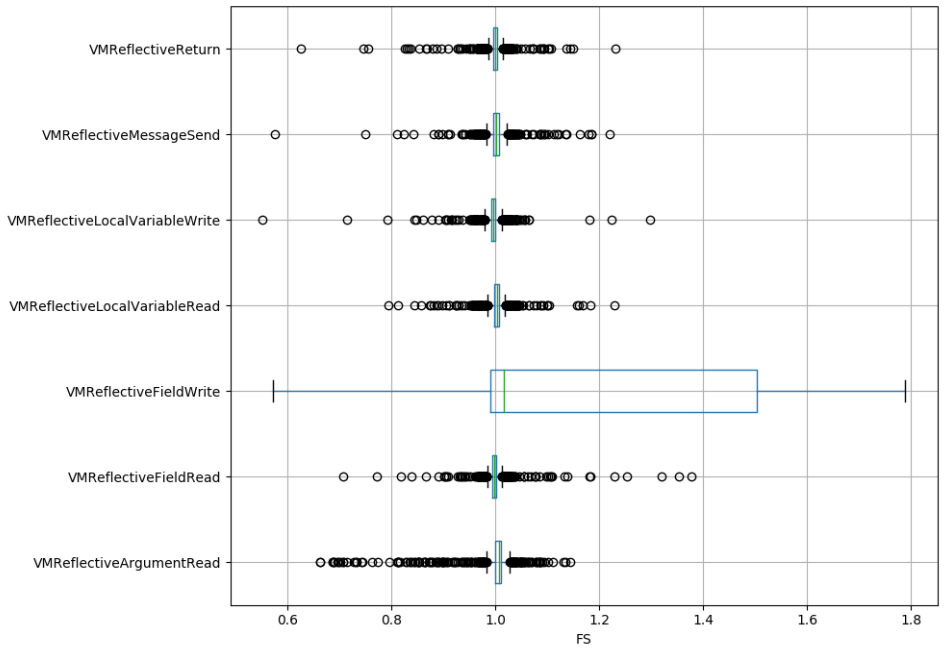
Tab. 4.2: Sobrecarga Inherente - Detalle

Benchmark	**FS**	**Sd.**	**Min**	**Max**	**Median**	**p95**
Bounce	0.98	0.03	0.79	1.07	0.98	1.01
BubbleSort	0.89	0.03	0.7	1.0	0.9	0.92
CD	1.28	0.01	1.22	1.32	1.28	1.3
DeltaBlue	1.05	0.02	0.87	1.16	1.05	1.07
Dispatch	1.0	0.03	0.93	1.28	1.0	1.04
Fannkuch	1.0	0.03	0.64	1.04	1.0	1.02
Fibonacci	0.98	0.05	0.82	1.51	0.98	1.02
FieldLoop	1.0	0.01	0.95	1.06	1.0	1.02
List	1.03	0.01	1.0	1.14	1.03	1.04
Loop	1.01	0.02	0.81	1.08	1.01	1.03
Permute	1.01	0.02	0.92	1.16	1.01	1.04
Queens	0.95	0.03	0.78	1.17	0.96	0.99
QuickSort	1.02	0.04	0.98	1.29	1.01	1.05
Recurse	1.03	0.03	0.84	1.19	1.03	1.05
Sieve	1.0	0.01	0.96	1.11	1.0	1.01
Storage	1.0	0.02	0.93	1.09	1.0	1.03
Sum	0.99	0.02	0.89	1.05	0.99	1.01
Towers	0.52	0.01	0.47	0.55	0.52	0.54
TreeSort	1.06	0.02	0.93	1.17	1.06	1.08
WhileLoop	1.01	0.05	0.93	1.29	1.0	1.1

4.3.3. Operaciones Individuales

Para esta prueba medimos el *FS* de las operaciones re-implementadas mediante *MATE* en relación a las operaciones originales de la máquina virtual. Obtuvimos los siguientes resultados:

Fig. 4.2: Operaciones Individuales - Factor de sobrecarga



Como se puede observar en la figura 4.2, las re-implementaciones de las funcionalidades mediante el framework *MATE* presentan un *FS* promedio de 1.01x en relación a la máquina virtual base. Es un resultado muy interesante dado que estas operaciones son fundamentales en la ejecución de la máquina virtual y las mismas no sufrieron un deterioro de rendimiento mayor. En relación a *TruffleMate*, para este conjunto de pruebas se registró un rendimiento de 0.99x [Cha17], un resultado muy similar al que obtuvimos para el mismo conjunto de pruebas.

Tab. 4.3: Operaciones Individuales - Detalle

Benchmark	**FS**	**Sd.**	**Min**	**Max**	**Median**	**p95**
ArgumentRead	0.99	0.06	0.66	1.14	1.01	1.05
FieldRead	1.0	0.04	0.71	1.38	1.0	1.03
FieldWrite	1.14	0.29	0.57	1.79	1.02	1.63
LocalVariableRead	1.0	0.03	0.79	1.23	1.0	1.04
LocalVariableWrite	1.0	0.03	0.55	1.3	1.0	1.03
MessageSend	1.0	0.03	0.58	1.22	1.0	1.04
Return	1.0	0.03	0.63	1.23	1.0	1.03

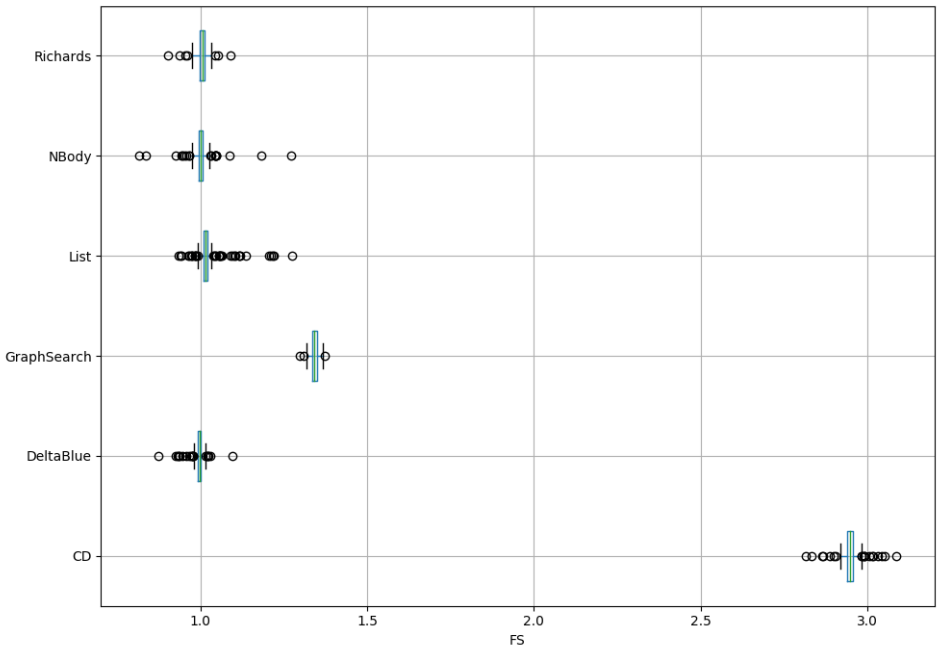
4.3.4. Inmutabilidad

Como mencionamos anteriormente, en este experimento hicimos dos pruebas diferentes: una comparando versiones inmutables de algunas de las pruebas utilizadas en los anteriores experimento; y otra evaluando la implementación de inmutabilidad de *MATE* contra otra versión implementada sin utilizar las capacidades reflexivas.

4.3.5. Rendimiento base

En estas pruebas comparamos el rendimiento de nuestra solución contra una versión que no implementa inmutabilidad. Como casos de prueba utilizamos versiones modificadas de las pruebas para incluir inmutabilidad en los objetos principales. Obtuvimos lo siguientes resultados:

Fig. 4.3: Inmutabilidad, Rendimiento Base - Factor de sobrecarga



Tab. 4.4: Inmutabilidad, Rendimiento Base - Detalle

Benchmark	FS	Sd.	Min	Max	Median	p95
CD	2.95	0.03	2.82	3.09	2.95	2.99
DeltaBlue	1.0	0.02	0.87	1.1	1.0	1.01
GraphSearch	1.34	0.01	1.3	1.37	1.34	1.36
List	1.02	0.05	0.94	1.27	1.01	1.12
NBody	1.0	0.04	0.82	1.27	1.0	1.04
Richards	1.01	0.02	0.9	1.09	1.01	1.02

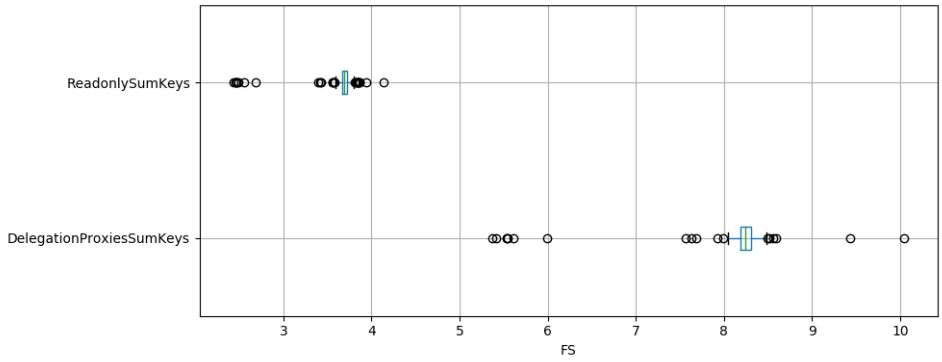
Como podemos observar en los resultados presentados en la figura 4.3, el *FS* promedio fue de 1.38x, siendo la prueba *CD* la que sufrió una mayor perdida en rendimiento con un factor de sobrecarga de 2.95x en relación a su versión sin inmutabilidad. Creemos que gran parte de la diferencia de desempeño de *CD* con el resto de las pruebas, se encuentra que *CD* opera de forma dinámica con una gran cantidad de objetos no primitivos, por lo tanto la cantidad de objetos inmutables "creados" es mayor que el resto.

Para este conjunto de experimentos aún no tenemos ningún estudio previo sobre el rendimiento de *TruffleMate*, con lo cual no podemos hacer una comparación de ambas soluciones.

4.3.6. Rendimiento contra una solución nativa

En este experimento comparamos ambas soluciones mediante una prueba que crea una referencia de solo lectura a una lista enlazada la cual luego es recorrida y modificada. Es importante destacar que durante el recorrido de la misma es necesario ir generando nodos de solo lectura para asegurar inmutabilidad completa de toda la lista. Como consecuencia estamos ante un escenario donde se crean de forma dinámica una gran cantidad objetos. Como base de comparación, en ambos casos, vamos a utilizar una implementación que realiza exactamente lo mismo pero sin asegurar inmutabilidad de la lista.

Fig. 4.4: Inmutabilidad, Solución Nativa - Factor de sobrecarga



Tab. 4.5: Inmutabilidad, Solución Nativa - Detalle

Benchmark	**FS**	**Sd.**	**Min**	**Max**	**Median**	**p95**
DelegationProxiesSumKeys	8.13	0.61	5.38	10.04	8.25	8.46
ReadonlySumKeys	3.64	0.28	2.43	4.13	3.69	3.84

Como se puede observar en la figura 4.4, nuestra solución basada en *MATE* presenta un *FS* promedio aproximadamente dos veces mejor que la solución basada en *DelagationProxies*. En *TruffleMate*, la misma solución con *MATE* registró un *FS* de 1.91x y su variante con *DelagationProxies* uno de 4.17x [Cha17].

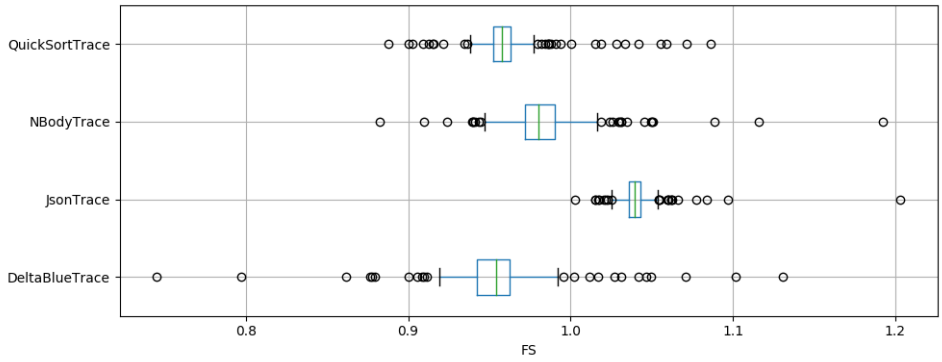
Si bien la solución con *TruffleMate* tiene mejor rendimiento que la nuestra, en ambos casos la solución mediante *MATE* presenta un rendimiento superir a la versión implementada mediante *DelagationProxies*. Creemos que al implementar la solución mediante las características de reflexivas de la máquina virtual, ambos compiladores JIT pueden optimizar de forma más eficiente la ejecución.

Adicionalmente, es interesante destacar que la solución mediante *MATE* fue más sencilla y requirió significativamente menos líneas de código para su implementación en relación a las necesarias para la solución basada en *DelegationProxies*.

4.3.7. Trazado de código

Para este experimento ejecutamos un sub-conjunto de las pruebas utilizadas en el experimento *Rendimiento Base*, incluyendo durante la ejecución de la misma el *meta-entorno* propiamente definido para este experimento. Obtuvimos los siguientes resultados:

Fig. 4.5: Trazado de código - Factor de sobrecarga



Tab. 4.6: Trazado de código - Detalle

Benchmark	FS	Sd.	Min	Max	Median	p95
DeltaBlue	0.95	0.04	0.75	1.13	0.95	1.0
Json	1.04	0.02	1.0	1.2	1.04	1.06
NBody	0.98	0.03	0.88	1.19	0.98	1.03
QuickSort	0.96	0.02	0.89	1.09	0.96	0.99

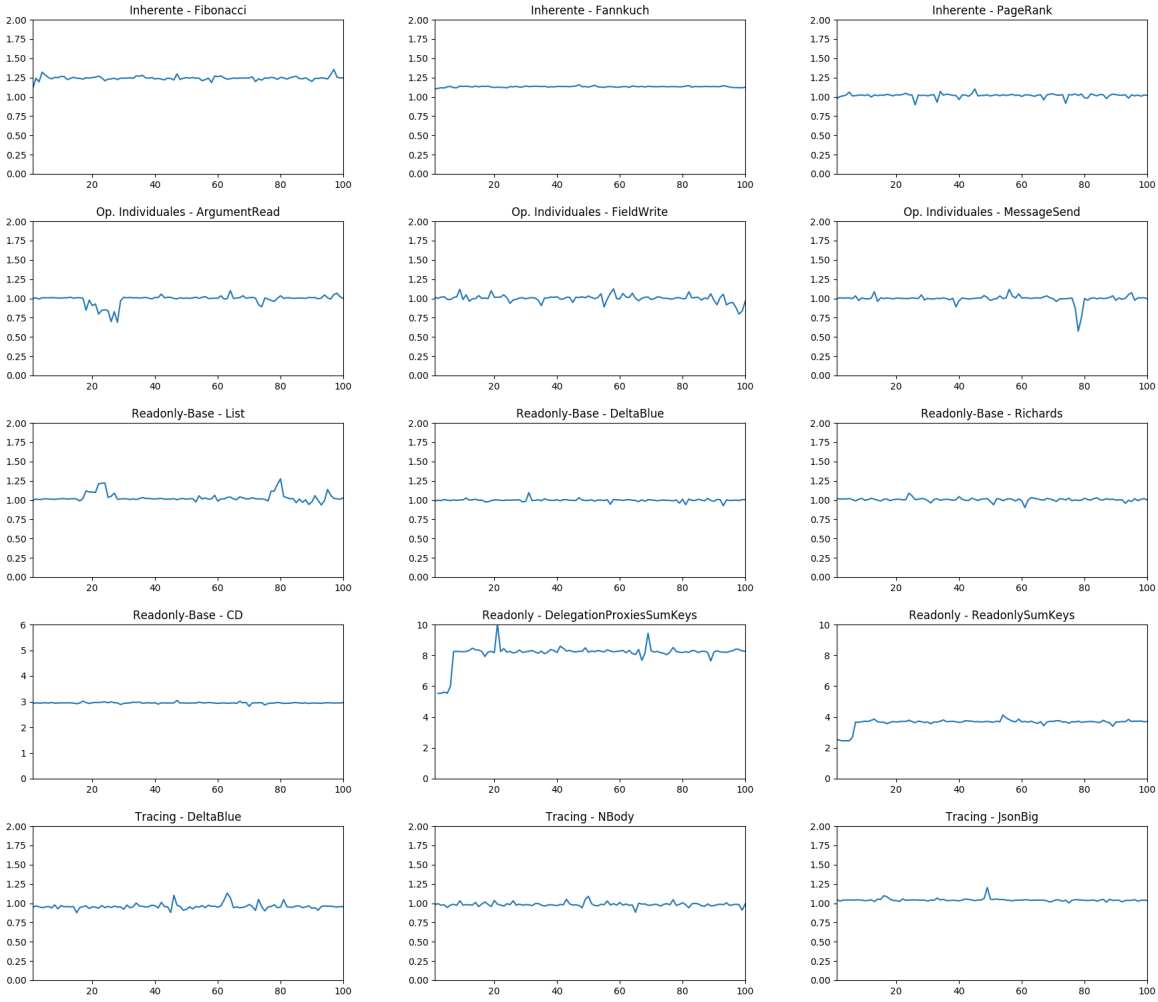
Como podemos observar en la figura 4.5, la inclusión del *meta-objeto* no presenta una reducción de rendimiento significativo en las pruebas. Estos resultados son mejores a los 3.6x obtenidos durante la evaluación de *TruffleMate* en estas mismas condiciones. Creemos que parte de esta diferencia de rendimiento puede ser explicada por las diferentes políticas de *inlining* que existen en las implementaciones, aunque ajustando ciertas configuraciones relacionadas a esto en *TruffleMate* debería ser posible mejorar su rendimiento [Cha17].

4.4. Warmup

Para analizar el tiempo de *Warmup* de nuestra solución realizamos un muestreo de las primeras 100 iteraciones de las pruebas que utilizamos en nuestro trabajo. En nuestras mediciones pudimos observar que no son necesarias muchas iteraciones para que la máquina virtual alcance su pico de rendimiento y se mantenga estable. Este resultado sigue la linea de otros estudios [MD15] donde ya se comprobó que el tiempo necesario por compiladores JIT de trazas para optimizar el rendimiento suele ser poco.

A continuación ilustramos mediante experimentación como varia el *FS* a los largo de las primeras 100 iteraciones para algunos de las pruebas (el resto pueden ser encontrado en el apéndice de este trabajo):

Fig. 4.6: Factor de Sobrecarga en las primeras 100 iteraciones



4.5. Discusión

Como pudimos ver en las pruebas de *Rendimiento Base*, el desempeño de *RTruffleSOM* a pesar de ser un prototipo es bastante bueno. Recordemos que en general las máquinas virtuales que se utilizan en la industria suelen tener equipos expertos que trabajan durante varios años para lograr que sean muy eficientes. Más aún, si tomamos como referencia la máquina virtual *OpenJDK*, donde adicionalmente a lo mencionado, el lenguaje Java al proveer información de tipos, facilita el proceso de optimización, caso contrario como sucede en lenguajes completamente dinámicos como *Smalltalk* o *Javascript*. Por ejemplo, en anteriores estudios [Cha17] el rendimiento de *Pharo*[Pharo], una implementación de *Smalltalk* muy utilizada, en comparación a *OpenJDK* era de casi 8x más lenta para el mismo conjunto de pruebas. Por lo tanto creemos que es correcto decir que para el conjunto de experimentos que realizamos, el rendimiento de *RTruffleSOM* es competente.

Por otro lado, en nuestras pruebas de *Sobrecarga Inherente* logramos comprobar que si no utilizamos las capacidades reflexivas, nuestra máquina virtual no pierde rendimiento en relación a su implementación original, resultados similares a los ya encontrados en *TruffleMate* [Cha17].

En los experimentos como *Operaciones Individuales* y *Trazado* pudimos verificar que si la semántica implementada con *MATE* es lo suficientemente simple, el compilador *JIT* provisto por *Pypy* puede optimizar lo suficiente el código generado hasta alcanzar una pérdida de rendimiento casi imperceptible.

Con los experimentos de *Inmutabilidad*, pudimos comprobar que en ciertos casos la utilización de *MATE* para realizar algunos cambios en el comportamiento de la máquina virtual presenta un mejor rendimiento en comparación soluciones que buscan lo mismo pero implementadas enteramente en el lenguaje huésped. Adicionalmente en nuestro caso, la solución con *MATE* fue significativamente más simple de implementar que su homónima con *DelegationProxies*.

Finalmente podemos concluir que el rendimiento general de *RTruffleMate* se encuentra en el mismo orden de magnitud a *TruffleMate* para los experimentos que realizamos, ratificando a su vez los resultados de otros estudios anteriores que compararon ambas familias de compiladores [MD15]. Si bien en algunos casos una solución presentó mejor rendimiento que la otra, creemos que al ser prototipos aún hay mucho margen de mejora y podríamos lograr que el desempeño de ambas se asemeje aún más.

5. CONCLUSIONES

En esta tesis exploramos la construcción de una máquina virtual reflexiva siguiendo la arquitectura propuesta por *MATE*. Durante la implementación de *RTruffleMATE* una vez que logramos adaptarnos a la utilización del framework *Pypy*, el desarrollo de nuestra solución fue relativamente ágil. Si bien *TruffleMATE* también implementa la arquitectura de *MATE*, debido a las diferencias en las tecnologías utilizadas, ambas soluciones fueron drásticamente diferentes y, en linea a otros estudios previos[MD15], para realizar la implementación de *RTruffleMATE* fue necesario un menor esfuerzo.

Este trabajo presenta pruebas de que es posible implementar una máquina virtual reflexiva mediante la utilización de un compilador *JIT* de trazas que tenga un desempeño aceptable en comparación a soluciones comerciales mucho más avanzadas. Para ello estudiamos el comportamiento de la máquina virtual en varios escenarios y con diferentes conjuntos de experimentos, que fueron desde pruebas para estresar diferentes aspectos de la máquina virtual y el lenguaje, hasta usos posibles de las capacidades reflexivas. Además, constatamos mediante nuestro experimento de inmutabilidad que la utilización de *MATE* para implementar ciertas semánticas "meta" puede no solo simplificar la codificación, sino también mejorar el rendimiento de la misma. En linea a esto, a su vez probamos que el impacto de la utilización de las capacidades reflexivas de la máquina virtual se va a encontrar directamente asociado a la funcionalidad que se quiera implementar, y no al soporte del framework en sí mismo.

Finalmente, validando nuestra hipótesis, pudimos comprobar mediante experimentación que el rendimiento de *RTruffleMATE* y *TruffleMATE* son similares en términos de magnitud, confirmando nuevamente resultados anteriores sobre el desempeño de las dos familias de compiladores *JIT* [MD15]. Si bien en algunas pruebas encontramos ciertas diferencias de rendimiento, las mismas pueden ser explicadas por las diferentes estrategias de optimización que usa cada una de las tecnologías. Pero, dado que ambas máquinas virtual son prototipos, confiamos que en sucesivas iteraciones es posible seguir mejorando las implementaciones y alcanzar un rendimiento asintótico similar.

5.1. Trabajo Futuro

En nuestro trabajo utilizamos como punto de evaluación el rendimiento en termino de tiempo computacional, si bien este acercamiento inicial es aceptable existen otros factores que son importantes al momento evaluar una máquina virtual. Por ejemplo, en ciertos contextos la memoria puede ser un recurso escaso y por lo tanto es necesario medir la utilización la misma. Consideramos que puede ser interesante evaluar el impacto de nuestra solución en términos de este recurso e implementar ciertas optimizaciones si fuera necesario.

Por otro lado, nos gustaría continuar implementando nuevos casos de uso de las capacidades reflexivas de *MATE* con el fin de explorar nuevos limites y posibilidades del framework.

En nuestros experimentos comprobamos que dependiendo las características de la solución desarrollada mediante *MATE*, el rendimiento puede variar significativamente. Por lo tanto, seguir explorando las posibilidades de *RTruffleMATE* es indispensable para continuar mejorando su rendimiento.

Finalmente, ninguna de implementaciones actuales de *MATE* permiten interactuar con todos los componentes internos de la máquina virtual. Por ejemplo, el *parser* o el módulo de administración de memoria aún no están expuestos por el protocolo de *meta-objetos*. Para completar las capacidades reflexivas de la máquina virtual, es necesario dar soporte a todos los componentes faltantes.

6. APÉNDICE

6.1. Detalle de ejecución

En esta sección se listan los tiempos de ejecución de cada uno de los experimentos utilizados en nuestra investigación.

Tab. 6.1: Rendimiento Base - Ejecución en ms

Benchmark	Baseline	Mean	Sd.	Min	Max	Median	p95
DeltaBlue	RTruffleSOM	150.35	38.99	141.69	620.93	144.91	154.25
	OpenJDK	33.31	17.99	28.66	234.01	29.69	40.24
	NodeJS	78.08	22.04	63.92	302.1	70.66	95.92
Towers	RTruffleSOM	239.81	10.79	235.73	331.37	238.29	244.41
	OpenJDK	89.79	2.94	87.85	106.95	88.87	96.0
	NodeJS	153.4	5.26	150.87	184.77	151.67	161.83
Mandelbrot	RTruffleSOM	339.15	4.92	334.82	383.43	338.56	344.46
	OpenJDK	153.28	1.63	152.96	168.53	153.06	153.39
	NodeJS	137.87	0.61	137.58	142.83	137.71	138.53
List	RTruffleSOM	687.24	8.88	681.32	760.13	685.06	697.85
	OpenJDK	58.71	1.48	57.46	69.04	58.5	60.03
	NodeJS	113.2	2.43	111.8	133.72	112.58	117.4
Bounce	RTruffleSOM	110.71	3.79	108.4	151.06	109.9	113.09
	OpenJDK	50.61	3.53	48.23	82.98	49.44	55.93
	NodeJS	345.22	3.07	342.29	358.6	344.12	352.13
CD	RTruffleSOM	521.15	108.44	503.73	1835.95	510.87	519.35
	OpenJDK	61.1	29.44	54.02	357.2	54.79	88.85
	NodeJS	107.93	8.69	103.79	196.2	105.98	119.62
Json	RTruffleSOM	243.1	52.64	228.92	861.56	235.5	266.69
	OpenJDK	109.77	16.6	103.03	235.24	104.27	134.29
	NodeJS	141.31	3.7	138.95	179.64	140.51	145.39
Sieve	RTruffleSOM	370.46	2.68	367.11	386.07	369.9	374.4
	OpenJDK	82.06	3.21	75.39	107.49	81.42	86.36
	NodeJS	173.47	3.03	171.24	192.14	172.69	177.93
Permute	RTruffleSOM	385.67	16.29	377.98	516.86	381.15	414.45
	OpenJDK	55.45	1.95	54.59	73.1	54.92	58.17
	NodeJS	107.23	1.19	106.23	114.08	106.82	109.37

Tab. 6.2: Rendimiento Base - Ejecución en ms (continuación)

Benchmark	**Baseline**	**Mean**	**Sd.**	**Min**	**Max**	**Median**	**p95**
Richards	RTruffleSOM	241.59	16.0	234.87	403.69	237.7	255.69
	OpenJDK	105.39	4.85	102.19	159.77	105.39	107.12
	NodeJS	182.43	3.4	180.26	210.55	181.52	185.78
NBody	RTruffleSOM	90.04	2.35	88.14	107.17	89.33	94.4
	OpenJDK	27.14	1.54	26.03	38.41	26.8	28.04
	NodeJS	54.17	1.45	52.9	62.83	53.76	56.25
Storage	RTruffleSOM	465.16	13.94	457.47	626.46	463.31	471.22
	OpenJDK	110.49	19.95	100.98	202.58	103.97	163.31
	NodeJS	167.17	4.72	160.33	189.39	167.65	174.78
Havlak	RTruffleSOM	1203.86	46.53	1176.02	1732.09	1199.27	1233.38
	OpenJDK	113.02	36.48	100.92	444.53	105.06	141.66
	NodeJS	638.73	19.16	608.41	757.57	635.48	666.68
Queens	RTruffleSOM	288.9	3.62	285.16	309.97	287.89	295.92
	OpenJDK	48.8	1.23	47.84	58.99	48.67	49.98
	NodeJS	94.7	2.5	93.19	114.96	94.06	97.29

Tab. 6.3: Sobrecarga Inherente - Ejecución en ms

Benchmark	Ejecutor	Sd.	Min	Max	Median	p95
WhileLoop	RTruffleSOM	3.08	348.92	376.82	350.94	355.43
	RTruffleMate	16.37	349.17	451.35	351.51	385.79
DeltaBlue	RTruffleSOM	3.79	166.57	201.06	167.96	174.39
	RTruffleMate	2.04	174.7	194.19	176.22	179.61
Fannkuch	RTruffleSOM	97.18	2021.43	3188.76	2038.66	2075.55
	RTruffleMate	13.5	2032.11	2142.35	2044.53	2065.25
Towers	RTruffleSOM	6.6	381.25	429.71	384.72	393.33
	RTruffleMate	1.82	200.33	213.01	201.59	205.57
QuickSort	RTruffleSOM	4.61	691.03	717.89	694.62	706.76
	RTruffleMate	24.87	696.31	896.11	701.51	731.42
Sum	RTruffleSOM	5.5	341.04	386.24	343.49	355.13
	RTruffleMate	2.47	340.13	358.26	341.59	345.59
List	RTruffleSOM	4.59	673.41	701.67	679.4	690.7
	RTruffleMate	7.48	695.65	771.06	700.67	709.51
Dispatch	RTruffleSOM	3.33	224.09	243.5	225.62	230.37
	RTruffleMate	6.45	224.58	288.65	226.16	233.41
CD	RTruffleSOM	4.21	502.21	530.4	505.87	513.64
	RTruffleMate	3.96	644.24	670.39	648.32	656.09
BubbleSort	RTruffleSOM	6.82	164.26	212.67	166.22	178.18
	RTruffleMate	2.41	147.06	165.25	148.87	152.42
Storage	RTruffleSOM	4.89	445.15	486.69	451.0	456.73
	RTruffleMate	5.1	448.76	489.22	452.65	465.09
TreeSort	RTruffleSOM	5.03	254.67	294.57	257.16	264.86
	RTruffleMate	5.24	269.08	323.45	271.83	278.18
Bounce	RTruffleSOM	3.32	109.21	137.07	110.18	115.89
	RTruffleMate	1.45	107.38	118.26	108.5	110.75
Queens	RTruffleSOM	8.29	284.75	351.01	287.0	298.72
	RTruffleMate	5.82	272.23	334.46	274.25	282.77
Sieve	RTruffleSOM	2.1	369.44	386.69	372.11	375.47
	RTruffleMate	4.37	367.79	410.65	370.24	375.13
FieldLoop	RTruffleSOM	0.9	93.37	98.86	93.74	95.32
	RTruffleMate	0.96	93.39	100.53	93.85	95.68
Recurse	RTruffleSOM	2.76	88.0	108.69	88.67	92.36
	RTruffleMate	1.46	90.38	105.76	91.32	93.56
Permute	RTruffleSOM	5.21	371.46	410.11	374.55	382.54
	RTruffleMate	7.24	373.06	435.89	376.56	387.75
Loop	RTruffleSOM	6.05	261.62	330.48	263.13	269.7
	RTruffleMate	3.28	264.23	283.59	265.77	271.34
Fibonacci	RTruffleSOM	0.79	31.12	37.02	31.33	32.54
	RTruffleMate	1.49	30.39	47.11	30.59	31.77

Tab. 6.4: Operaciones Individuales - Ejecución en ms

Benchmark	Ejecutor	Sd.	Min	Max	Median	p95
FieldWrite	Baseline	0.05	0.34	0.6	0.35	0.49
	Reflective	0.09	0.34	0.61	0.35	0.57
Return	Baseline	0.59	16.3	26.42	16.5	17.05
	Reflective	0.31	16.38	20.27	16.51	16.97
FieldRead	Baseline	0.42	16.27	23.36	16.54	17.0
	Reflective	0.54	16.38	22.91	16.51	17.0
ArgumentRead	Baseline	0.96	12.1	18.66	12.23	14.46
	Reflective	0.27	12.21	15.53	12.31	12.82
MessageSend	Baseline	0.41	11.26	19.72	11.36	11.74
	Reflective	0.26	11.26	13.85	11.37	11.77
LocalVariableRead	Baseline	0.3	12.18	15.42	12.27	12.75
	Reflective	0.24	12.2	15.12	12.3	12.72
LocalVariableWrite	Baseline	0.47	12.14	22.09	12.25	12.67
	Reflective	0.24	12.11	16.01	12.2	12.62

Tab. 6.5: Inmutabilidad, Rendimiento Base - Ejecución en ms

Benchmark	Ejecutor	Sd.	Min	Max	Median	p95
DeltaBlue	Baseline	3.33	176.75	204.03	178.24	183.35
	Readonly	1.82	176.54	194.82	177.87	181.05
List	Baseline	14.27	680.64	794.76	685.07	714.14
	Readonly	32.26	689.71	871.32	694.45	768.11
CD	Baseline	4.9	649.4	688.16	653.11	659.67
	Readonly	14.34	1914.72	2011.51	1925.34	1950.36
GraphSearch	Baseline	4.63	695.78	725.38	705.49	711.53
	Readonly	5.73	935.3	969.11	945.24	956.52
Richards	Baseline	3.73	246.75	278.69	249.36	256.29
	Readonly	2.65	247.65	270.74	251.03	254.66
NBody	Baseline	2.41	89.51	110.34	90.24	92.91
	Readonly	2.63	89.36	114.41	90.23	93.9

Tab. 6.6: Inmutabilidad, Solución Nativa - Ejecución en ms

Benchmark	Ejecutor	Sd.	Min	Max	Median	p95
ReadonlySumKeys	Baseline	0.45	4.09	6.32	4.13	5.13
	Readonly	0.25	15.07	17.05	15.25	15.81
DelegationProxiesSumKeys	Baseline	0.42	4.09	6.32	4.13	4.47
	Readonly	0.77	33.67	41.33	34.02	34.85

Tab. 6.7: Trazado de código - Detalle

Benchmark	Ejecutor	Sd.	Min	Max	Median	p95
Json	Baseline	18.39	3201.95	3330.88	3219.67	3254.26
	Tracing	45.39	3324.89	3882.07	3346.63	3406.66
DeltaBlue	Baseline	3.78	125.95	163.09	127.37	133.37
	Tracing	3.23	120.41	144.91	121.6	128.93
QuickSort	Baseline	9.6	789.79	855.32	796.54	813.78
	Tracing	17.48	756.24	883.28	762.5	795.49
NBody	Baseline	0.85	57.59	64.26	57.92	59.86
	Tracing	1.4	56.37	68.95	56.85	59.88

6.2. Warmup

Fig. 6.1: Warmup en las primeras 100 iteraciones

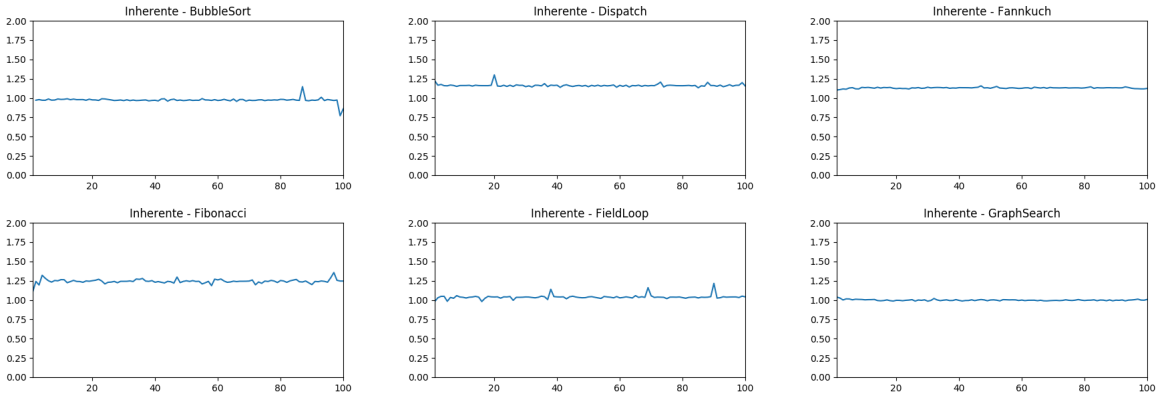


Fig. 6.2: Warmup en las primeras 100 iteraciones (continuación)

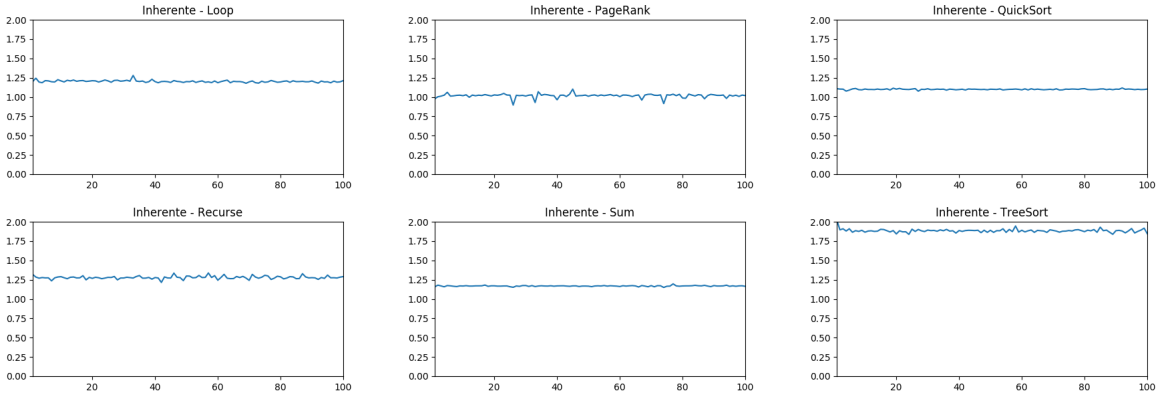


Fig. 6.3: Warmup en las primeras 100 iteraciones (continuación)

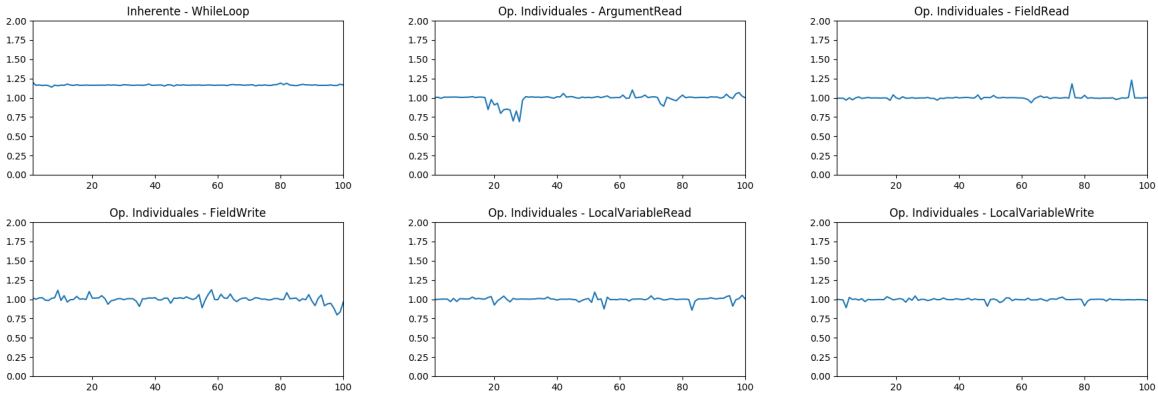


Fig. 6.4: Warmup en las primeras 100 iteraciones (continuación)

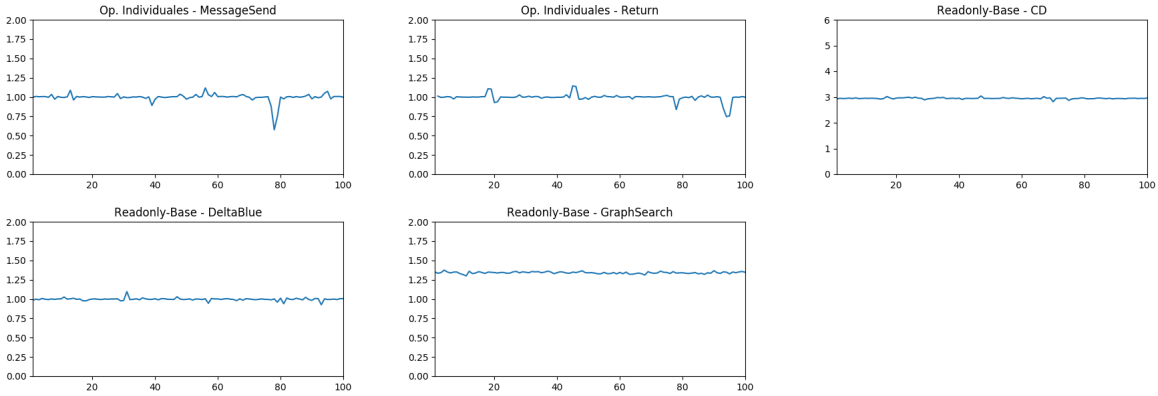


Fig. 6.5: Warmup en las primeras 100 iteraciones (continuación)

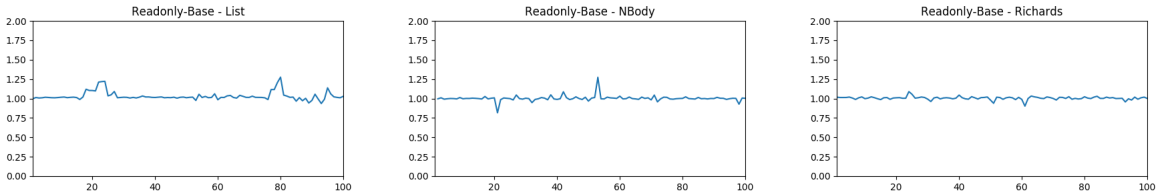


Fig. 6.6: Warmup en las primeras 100 iteraciones (continuación)

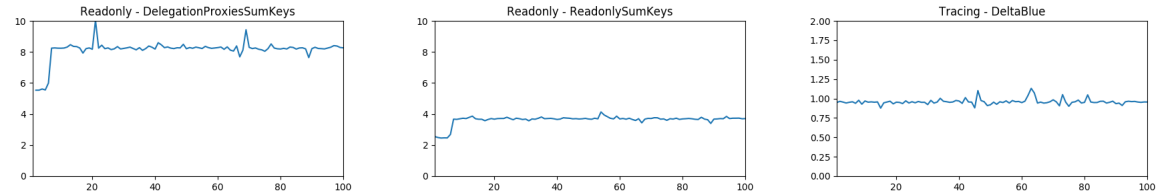
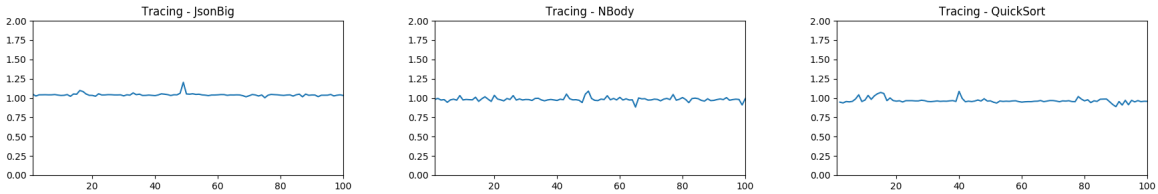


Fig. 6.7: Warmup en las primeras 100 iteraciones (continuación)



BIBLIOGRAFíA

[ABF12] Håkan Ardö, Carl Friedrich Bolz y Maciej FijaBkowski. «Loop-aware optimizations in PyPy's tracing JIT». En: vol. 48. Oct. de 2012, págs. 63-72. DOI: 10.1145/2384577.2384586.

[Ayc03] John Aycock. «A Brief History of Just-In-Time». En: *ACM Comput. Surv.* 35 (jun. de 2003), págs. 97-113. DOI: 10.1145/857076.857077.

[Bol+09] Carl Friedrich Bolz y col. «Tracing the meta-level: PyPy's tracing JIT compiler». En: ene. de 2009, págs. 18-25. ISBN: 9781605585413. DOI: 10.1145/1565824.1565827.

[Bol+11] Carl Friedrich Bolz y col. «Allocation removal by partial evaluation in a tracing JIT». En: ene. de 2011, págs. 43-52. DOI: 10.1145/1929501.1929508.

[CGM16] Guido Chari, Diego Garbervetsky y Stefan Marr. «Building Efficient and Highly Run-time Adaptable Virtual Machines». En: nov. de 2016, págs. 60-71. DOI: 10.1145/2989225.2989234.

[CGM17] Guido Chari, Diego Garbervetsky y Stefan Marr. «Fully-Reflective VMs for Ruling Software Adaptation». En: mayo de 2017, págs. 229-231. ISBN: 9781538615898. DOI: 10.1109/ICSE-C.2017.144.

[Cha+18a] Guido Chari y col. «Fully Reflective Execution Environments». En: *IEEE Transactions on Software Engineering* PP (mar. de 2018), págs. 1-1. DOI: 10.1109/TSE.2018.2812715.

[Cha+18b] Guido Chari y col. «Fully Reflective Execution Environments : Virtual Machines for More Flexible Software». En: (mayo de 2018).

[Cha17] Guido Chari. «Plataformas de ejecución de software reflexivas». En: (dic. de 2017).

[Clasp] *Clasp — Bringing Common Lisp and C++ Together.* URL: https://github.com/clasp-developers/clasp.

[CU89] Craig Chambers y David Ungar. «Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language». En: vol. 24. Jul. de 1989, págs. 295-312. DOI: 10.1145/74818.74831.

[Cyth] *Cython: an optimising static compiler for Python and Cython.* URL: https://cython.org/.

[Dokr] *Docker: Empowering App Development for Developers.* URL: https://www.docker.com/.

[Fact] *Factor programming language.* URL: https://factorcode.org/.

[Fut99] Yoshihiko Futamura. «Partial Evaluation of Computation Process–An Approach to a Compiler-Compiler». En: *Higher-Order and Symbolic Computation* 12 (dic. de 1999), págs. 381-391. DOI: 10.1023/A:1010095604496.

[Gam+95] Gamma y col. «Design Patterns: Elements of Reusable Object-Oriented Software». En: *Reading, MA: Addison-Wesley* 49 (ene. de 1995).

[Ghe11] Carlo Ghezzi. «The Fading Boundary between Development Time and Run Time». En: oct. de 2011, págs. 11-11. DOI: 10.1109/ECOWS.2011.33.

[GoV8] *V8: Google's open source high-performance JavaScript and WebAssembly engine.* URL: https://v8.dev/.

[Gral] *GraalVM: Run Programs Faster Anywhere.* URL: https://github.com/oracle/graal/.

[Hau+09] Michael Haupt y col. «Disentangling virtual machine architecture». En: *Software, IET* 3 (jul. de 2009), págs. 201-218. DOI: 10.1049/iet-sen.2007.0121.

[HCU91] Urs Hölzle, Craig Chambers y David Ungar. «Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches.» En: jul. de 1991, págs. 21-38. DOI: 10.1007/BFb0057013.

[Kot+08] Thomas Kotzmann y col. «Design of the Java HotSpot™ client compiler for Java 6». En: *TACO* 5 (mayo de 2008). DOI: 10.1145/1369396.1370017.

[MD15] Stefan Marr y Stéphane Ducasse. «Tracing vs. Partial Evaluation: Comparing Meta-Compilation Approaches for Self-Optimizing Interpreters». En: oct. de 2015. DOI: 10.1145/2814270.2814275.

[Mit70] James Mitchell. *The Design and Construction of Flexible and Efficient Interactive Programming Systems.* Ene. de 1970. ISBN: 0-8240-4414-2.

[NdJS] *NodeJS: entorno de ejecución para JavaScript construido con el motor de JavaScript V8.* URL: https://nodejs.org/es/.

[Ojdk] *OpenJDK: versión libre de la plataforma de desarrollo Java.* URL: https://openjdk.java.net/.

[Pharo] *Pharo: The immersive programming experience.* URL: https://pharo.org/web.

[Pypy] *PyPy: Intérprete y compilador JIT para el lenguaje Python.* URL: https://www.pypy.org/.

[Rbch] *ReBench: Execute and Document Benchmarks Reproducibly.* URL: https://github.com/smarr/ReBench.

[RTSom] *RTruffleSOM - The Simple Object Machine Smalltalk combining Self-Optimizing Interpreters with Meta-Tracing.* URL: https://github.com/SOM-st/RTruffleSOM.

[SOM] *SOM (SimpleObjectMachine): A minimal Smalltalk for teaching of and research on Virtual Machines.* URL: http://som-st.github.io/.

[Tra09] Laurence Tratt. «Dynamically Typed Languages». En: *Advances in Computers* 77 (jul. de 2009). Ed. por Marvin V. Zelkowitz, págs. 149-184.

[TrcM] *TraceMonkey: JavaScript Lightspeed.* URL: https://brendaneich.com/2008/08/tracemonkey-javascript-lightspeed/.

[Truf] *The Truffle Language Implementation Framework.* URL: https://github.com/oracle/graal/tree/master/truffle.

[Van+15] Maarten Vandercammen y col. «A formal foundation for trace based jit compilers». En: oct. de 2015. DOI: 10.1145/2823363.2823369.

[Wer+14] Erwann Wernli y col. «Delegation proxies». En: abr. de 2014, págs. 1-12. ISBN: 9781450327725. DOI: 10.1145/2584469.2577081.

[Woe+14] Andreas Woess y col. «An object storage model for the truffle language implementation framework». En: sep. de 2014. DOI: 10.1145/2647508.2647517.

[WW12] Christian Wimmer y Thomas Wuerthinger. «Truffle: A self-optimizing runtime system». En: oct. de 2012, págs. 13-14. DOI: 10.1145/2384716.2384723.