UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

DISEÑO, IMPLEMENTACIÓN Y EVALUACIÓN DE UNA COLECCIÓN ADAPTATIVA

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

SERGIO ANDRÉS MAASS OLEA

PROFESOR GUÍA:
ALEXANDRE BERGEL

MIEMBROS DE LA COMISIÓN:
MARÍA CECILIA RIVARA ZÚÑIGA
JAVIER BUSTOS JIMÉNEZ

SANTIAGO DE CHILE
ENERO 2016

RESUMEN

La creación y manipulación de colecciones de valores está ampliamente soportada por los lenguajes de programación modernos. A pesar de que cada lenguaje viene con su propia implementación de colecciones, la mayoría de las implementaciones encontradas son muy similares en estructura y funcionalidad ofrecida. Una excepción notable es el caso de Lua, pues presenta una sola colección: la tabla. Esta estructura es la unificación de una colección expansible secuencial y un arreglo asociativo o diccionario.

La tabla es una estructura híbrida, compuesta por una parte de hash y una parte de arreglo, y hará uso de sus estructuras internas según su uso: si es utilizada como lista hará uso del arreglo y si es usada como diccionario usará el hash. Esto permite flexibilidad y eficiencia pues las estructuras internas sólo se crean cuando requieren ser usadas. Este comportamiento adaptativo es interesante en cuanto sugiere buen rendimiento mientras que se muestra muy simple para el usuario. A pesar de que Lua ha logrado bastante popularidad, las tablas no han sido cuidadosamente estudiadas por la comunidad científica.

La eficiencia no solo depende de la implementación de las colecciones, sino también de si se escoge la colección correcta, con los parámetros adecuados, para cada situación particular. La selección inadecuada de colecciones puede resultar en un sobrecosto severo. Por este motivo, contar con una forma automática de asignar la colección más adecuada para cada caso sería muy valioso en la práctica. El experimento *Chameleon* aborda este problema sobre la JVM, mediante la identificación automática de elecciones inadecuadas de colecciones a través de análisis en tiempo de ejecución. El reemplazo de las colecciones identificadas por otras más adecuadas permitió reducir el consumo de memoria hasta en un 55 % y el tiempo de ejecución en un 60 % en algunas aplicaciones.

Este trabajo enfrenta el problema de determinar la aplicabilidad de la tabla de Lua, o un derivado de esta, en lenguajes con bibliotecas ricas de colecciones. Para esto se desarrollan tres colecciones en el lenguaje Pharo: SLua, SmartCollection y SmartCollection2. Estas luego se evalúan en comparación con las colecciones normales de Pharo y con una versión *lazy* de las mismas mediante distintos tipos de benchmarks que miden su tiempo de ejecución y consumo de memoria, y se realiza análisis dinámico para entender las diferencias en los resultados. Además, se aborda el problema de la selección adaptativa de colecciones y se replica el experimento *Chameleon*. Luego se usa la herramienta de selección y reemplazo automático de colecciones para evaluar las colecciones desarrolladas.

Los resultados de los experimentos realizados mediante instrumentación muestran reducciones de hasta un 30 % en consumo de memoria y hasta un 15 % en tiempo de ejecución en algunos escenarios al reemplazar las colecciones tradicionales por las desarrolladas. Sin embargo, la evaluación posterior mediante la herramienta de reemplazo de colecciones a nivel de código fuente no reveló diferencias estadísticamente significativas entre los tiempos de ejecución y uso de memoria entre las distintas colecciones. Esto indica una amplificación de los resultados introducida por la instrumentación. Las colecciones de las librerías externas utilizadas por las aplicaciones estudiadas no fueron reemplazadas, por lo que se plantea estudiar el impacto del reemplazo de colecciones sobre estas como trabajo futuro.

Tabla de Contenido

Introducción

La creación y manipulación de colecciones de valores está ampliamente soportada por los lenguajes de programación modernos [1]. Un entorno de programación típicamente ofrece una biblioteca de colecciones que soporta un amplio rango de variaciones en la forma en que las colecciones de valores son manejados y manipulados. Las colecciones poseen diversas características [1, 2, 3], incluyendo el ser expansibles o no. Una colección expansible es aquella cuyo tamaño puede variar a medida que los elementos son añadidos o removidos. Las colecciones expansibles son ampliamente usadas en los lenguajes modernos y han sido tema de diversos estudios [4, 5, 6, 7].

A pesar de que cada lenguaje de programación viene con su propia implementación de colecciones, la mayoría de las implementaciones encontradas en los lenguajes modernos son muy similares en estructura y funcionalidad ofrecida. Una excepción notable es el caso de Lua, pues presenta una sola colección: la tabla [8]. En Lua, la tabla es la unificación de una *lista* (ej. *ArrayList* en Java y *OrderedCollection* en Pharo) con lo que comúnmente se conoce como *arreglo asociativo* (ej. *Hashtable* en Java y *Dictionary* en Pharo). Un arreglo asociativo está compuesto por una colección de pares *(llave, valor)*, donde la llave puede ser cualquier objeto.

A partir de Lua 5.0[1], publicado en 2005, la tabla se implementa como una estructura de datos híbrida: compuesta por una parte de hash y una parte de arreglo. La tabla hace uso de sus estructuras internas según el contexto en el que se encuentre: si es utilizada como lista hará uso del arreglo, y en cuanto sea usada como diccionario usará el hash. Estas estructuras son inicializadas de forma perezosa, lo cual permite que la colección sea flexible y a la vez eficiente con respecto al uso de recursos. Su comportamiento adaptativo es interesante en cuanto sugiere un buen rendimiento mientras que expone una API muy simple para el usuario. Además, permite que las optimizaciones realizadas sobre esta colección tengan un alto impacto en las aplicaciones, ya que es la colección fundamental del lenguaje. Lua se ha vuelto un lenguaje altamente popular, encontrándose dentro de los primeros 20 en github[2], y es bastante usado en la industria de los videojuegos. A pesar esto, las tablas no han sido cuidadosamente estudiadas por la comunidad científica.

Que una colección se comporte eficientemente en cuanto a recursos no implica que las aplicaciones que hagan uso de estas funcionarán rápidamente. La eficiencia también depende de la colección que se escoja en cada situación particular, y de los parámetros iniciales que

[1] Lua está ahora en su versión 5.3.1

[2] http://githut.info

se elijan para estas. La selección inadecuada de colecciones, y sus parámetros, puede resultar en un sobrecosto severo. Chis *et al.* [9] estudiaron 34 *snapshots* del *heap* de memoria de 34 aplicaciones reales y en producción. Los autores mostraron que "la mayor parte de las *snapshots* atribuyen 50 % o más de su *heap* a sobrecostos de implementación". Las colecciones inadecuadamente escogidas, o pobremente manejadas, han sido citadas como un problema prominente al identificar las causas del sobrecosto de memoria.

Este problema se acentúa en el caso de lenguajes que cuentan con ricas bibliotecas de colecciones, con diversas implementaciones para los mismos tipos de dato abstractos. Las colecciones son difíciles de usar correctamente por las siguientes razones:

- *Elegir la colección adecuada* – La biblioteca de colecciones de Java consta de 10 clases y la de Pharo cuenta con 77. El elegir la implementación correcta para una situación en particular es una tarea difícil y se ha mostrado que los usuarios se equivocan con frecuencia [10, 7]. Por ejemplo, elegir *LinkedList* en vez de *ArrayList* resultará en un mal rendimiento en una situación que requiera acceso aleatorio a elementos de la colección.
- *Colecciones poco usadas* – Una colección escasamente poblada puede significar un desperdicio de memoria importante [9]. La elección de parámetros iniciales adecuados puede contribuir a reducir significativamente este sobrecosto.

Por estos motivos, contar con una forma automática de asignar la colección más adecuada para cada caso de uso podría ser muy valioso en la práctica, pudiendo lograr beneficios considerables en cuanto a tiempo de ejecución y/o consumo de memoria. El experimento *Chameleon* [7], realizado sobre una máquina virtual de Java (JVM) modificada, propone una técnica de *pattern matching* para enfrentar este problema: se monitorearon los sitios de producción de colecciones de diversas aplicaciones y se analizaron los patrones de uso de cada colección creada, luego en base a un conjunto de reglas predefinidas se identificaron elecciones de colecciones y/o parámetros iniciales potencialmente ineficientes, y finalmente se modificaron por implementaciones más adecuadas. Con esto se logró reducir considerablemente el consumo de memoria —y en algunos casos el tiempo de ejecución— de diversas aplicaciones.

El uso de una plataforma de ejecución modificada no deja claro hasta qué punto los resultados de *Chameleon* son replicables, por lo que la replicación del experimento en una plataforma diferente resultaría valiosa. La obtención de resultados positivos implicaría que una aplicación que se comporta pobremente debido a una mala elección de sus colecciones puede ser automáticamente optimizada, sin necesidad de conocer su implementación. Además, la herramientra planteada permitiría medir fácilmente la magnitud del impacto de reemplazar una colección por otra en diversas situaciones, lo cual contribuye a la evaluación de futuras optimizaciones o nuevas colecciones.

Este trabajo busca principalmente evaluar la aplicabilidad de la tabla de Lua, o un derivado de esta, en lenguajes con bibliotecas ricas de colecciones. La metodología propuesta por *Chameleon* resulta útil para el proceso de evaluación de la colección al permitir reemplazar una colección por otra de forma selectiva (según su procedencia y patrones de uso), con lo que se puede determinar experimentalmente en qué situaciones es más conveniente el uso de la colección desarrollada.

El trabajo se estructura de la siguiente forma. En el capítulo primero se expone un marco

teórico con los conceptos más relevantes sobre los que tratarán los capítulos siguientes. Específicamente se tratará el lenguaje Lua y su estructura tabla; el lenguaje Pharo, en el que desarrolla el trabajo; teoría acerca de los benchmarks, cuál será la metodología estadística para evaluarlos, y el framwork SMark que se usará para parte de estos; finalmente se expone acerca de profiling y se explica el funcionamiento del framework Spy y el profiler MessageTally. En el capítulo segundo se continúa con los objetivos del trabajo y la metodología que se usará para alcanzarlos. En el capítulo tercero se expone el núcleo del trabajo: el desarrollo y evaluación de la colección, que en realidad serán tres desarrolladas incrementalmente; se explicará el funcionamiento de cada colección desarrollada y se harán mediciones de tiempo de ejecución y consumo de memoria para cada una; se hará análisis de uso de colecciones para tratar de entender los resultados, y se compararán. En el capítulo cuarto se abordará el problema de la selección adaptativa de colecciones, y se buscará aplicarlo a la evaluación de las colecciones desarrolladas. En el quinto y último capítulo se describirán brevemente las herramientas más significativas desarrolladas para la evaluación de las colecciones.

Capítulo 1

Marco teórico

En este capítulo se abordan algunos temas fundamentales sobre los que se basa este trabajo. Primero se introduce el lenguaje Lua y se describe su estructura llamada *tabla*, debido a que en base a esta se diseña la colección adaptativa. Luego se describe brevemente al lenguaje Pharo, en cuanto a su semántica y su sintaxis, ya que el trabajo se realiza sobre este. Posteriormente se expone acerca de los *benchmarks*, se detalla la metodología estadística utilizada para analizar sus resultados, y se describe SMark, una herramienta utilizada para realizar parte de los mismos. Finalmente, se expone acerca de *profiling* y se describen dos herramientas útiles para llevarlo a cabo durante este trabajo: Spy y MessageTally.

1.1. Lua

Lua es un lenguaje de programación de propósito general, liviano y multi-paradigma, diseñado originalmente en 1993 para extender aplicaciones. Como lenguaje de extensión, Lua fue pensado para funcionar dentro de un programa huésped, manteniendo una comunicación bidireccional con este. Como tenía que funcionar dentro de otro programa, fue desarrollado para ser muy liviano, por lo que sólo incluía la funcionalidad básica de la mayoría de los lenguajes de programación procedurales. Para permitir casos de usos más complejos, Lua incluyó mecanismos para extender el lenguaje fácilmente. El desarrollo de Lua puso especial énfasis en el rendimiento, portabilidad, extensibilidad, y facilidad de uso [11].

Lua es un lenguaje dinámicamente tipado, y provee los siguientes ocho tipos básicos: *nil*, *boolean*, *number*, *string*, *userdata*, *function*, *thread*, y *table* (tabla). Las funciones son valores de primera clase en Lua, por lo que pueden ser manipulados como cualquier otro valor [12].

Un tipo de especial importancia en Lua es la tabla. La tabla es el único mecanismo de estructuración de datos en Lua, o sea, la única colección. Las tablas son utilizadas para representar arreglos, tablas de símbolos, conjuntos, *structs*, colas, objetos y otras estructuras de forma simple, uniforme, y eficiente. Lua incluso usa las tablas para representar paquetes. Por ejemplo, al escribir io.read nos referimos a "la función *read* del paquete *io*". Pero para Lua, eso significa "el elemento indexado por el string *read* en la tabla *io*" [12].

Lua es multi-paradigma porque provee de mecanismos que permiten extender y modificar su semántica para ajustarse a distintos tipos de problemas, en vez de proveer una especificación más compleja y rígida que cuadre con un paradigma específico. Gran parte de esta flexibilidad se debe a las tablas y su ubicuidad dentro del lenguaje. Por ejemplo, Lua no tiene soporte explícito para la herencia, pero permite su implementación mediante las *metatablas* (tablas que apuntan a tablas). De modo similar, Lua permite la implementación de espacios de nombres, clases, y otras funcionalidades relacionadas mediante el uso de la tabla.

Lua corre mediante la interpretación de bytecode por una máquina virtual basada en registros, la cual cuenta con manejo automático de memoria y recolección de basura incremental [8]. El lenguaje base es muy ligero —el intérprete completo compilado ocupa tan sólo 180kB—y es fácilmente adaptable a un amplio rango de aplicaciones. Estas características lo hacen ideal como lenguaje embebido, para *scripting* y prototipado rápido.

La Tabla

Como fue mencionado anteriormente, Lua ofrece un único tipo de dato abstracto para manipular colecciones: la *tabla*. Esta colección se comporta como un arreglo asociativo que además permite ser utilizado como lista de forma eficiente. Desde Lua 5.0[1], publicada en 2005, la tabla es una estructura de datos híbrida [8]. Esto se refiere a que internamente está compuesta por una parte de hash y una parte de arreglo. La tabla, entonces, hará uso de sus estructuras internas según el contexto en el que se encuentre: si es utilizada como una lista hará uso del arreglo y si es usada como diccionario usará el hash. Esto permite flexibilidad y a la vez eficiencia, pues las estructuras internas son inicializadas de forma perezosa. Para poder enfrentar todos los casos de uso típicos de una colección (ej. filtrar, ordenar, transformar [13]), Lua provee de un conjunto de funciones para manipular las tablas.

Dos usos

El que la tabla sea un arreglo asociativo significa que esta puede estar compuesta por una colección de pares *(llave, valor)*. Por ejemplo, la expresión $\{x = 10, y = 20\}$ crea una tabla con dos llaves, x e y, asociadas a los valores 10 y 20, respectivamente.

Una tabla también es una colección secuencial y expansible (lista), ya que puede crecer dinámicamente cuando se le añaden datos y achicarse cuando los datos son removidos. Por ejemplo, la expresión $\{5, 6, 7\}$ crea una tabla con tres elementos secuencialmente ordenados. Lua ofrece funciones para manipular una tabla e iteradores dedicados a recorrerla. Por ejemplo, *table.insert* y *table.remove* añaden y eliminan elementos.

[1] Lua está ahora en su versión 5.3.1

Estructura interna

La estructura interna de una tabla acomoda estos casos de uso radicalmente distintos a través de su estructura híbrida. Una tabla adapta de forma automática y dinámica sus dos partes internas según su contenido: la parte de arreglo intenta almacenar los valores correspondientes a llaves enteras entre 1 y algún límite n. Los valores correspondientes a llaves no enteras o a llaves enteras fuera del rango son almacenadas en la parte de hash. Considere la tabla $a = \{10, 20, x = 40\}$. Tres llaves son usadas, 1, 2, y x. Las dos primeras son almacenadas en la parte de arreglo y la última en la parte de hash.

Crecimiento

Durante una operación *table.insert()*, la tabla crece si no puede acomodar una llave que está siendo asociada a un valor. Tanto la parte de arreglo como la de hash pueden crecer. Por ejemplo, considere la tabla a descrita anteriormente. La expresión $a[8] = 1$ primero trata de insertar el valor en la parte de arreglo. La inserción en esa parte falla dado que el tamaño del arreglo es $n = 4$. El algoritmo de inserción luego trata de insertar la asociación en la parte de hash. Si la llave no puede ser acomodada en la parte de hash sin que esta requiera crecer, el algoritmo de crecimiento es gatillado. Este algoritmo considera todas las llaves enteras tanto en la parte de arreglo como de hash. Computa el tamaño maximal n, el cual es la mayor potencia de 2 tal que (i) al menos 50 % de los espacios entre 1 y n están en uso y (ii) hay al menos un espacio usado entre $\frac{n}{2} + 1$ y n. El algoritmo asigna un tamaño n a la parte de arreglo e inserta en esta todas las asociaciones con llaves enteras menores o iguales a n que estaban previamente en la parte de hash. Posteriormente, se hace un *rehash* de esta parte.

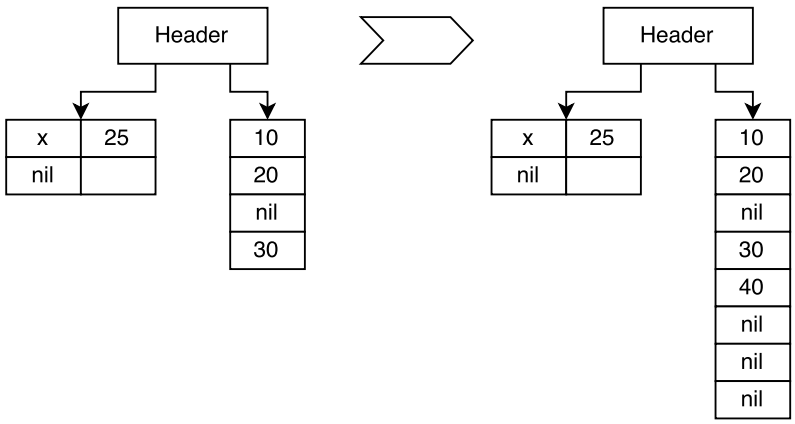

Figura 1.1: Tabla $a = \{10, 20, [4] = 30, x = 25\}$ antes y después de *insert(a, 40)*

Por ejemplo, considere la tabla producida por el código Lua $a = \{10, 20, [4] = 30, x = 25\}$. Esta tabla tiene una parte de hash que almacena la llave x asociada al valor 25 y una parte de arreglo con capacidad 4 pero con un tercer espacio sin uso (ver figura 1.1). Al aplicar la operación $table.insert(a, 40)$ la parte de arreglo crece para acomodar el valor adicional 40, y el nuevo tamaño para este será el doble de la cantidad de espacios usados (n más grande tal

que al menos la mitad de los espacios entre 1 y n están en uso).

Como consecuencia, si la tabla está siendo usada como arreglo, se comporta como arreglo siempre y cuando este sea denso (al menos 50 % de los espacios están en uso). Si las llaves están ordenadas consecutivamente, esto se cumple. En caso que sólo se use la parte de arreglo, la parte de hash no incurre en gasto de memoria ya que ni siquiera se ha creado. Del mismo modo, si la tabla está siendo usada como arreglo asociativo lo más probable es que la parte de arreglo no se haya creado por lo que no incurrirá en ninguna penalización.

1.2. Pharo

Pharo es un lenguaje y entorno de programación, puramente orientado a objetos, dinámicamente tipado, de código abierto, basado en Smalltalk. Pharo deriva de Squeak, una reimplementación del clásico sistema Smalltalk-80 [14]. Mientras que Squeak fue desarrollado principalmente como plataforma de desarrollo de software educacional experimental, Pharo intenta ofrecer una plataforma productiva y de código abierto para desarrollar software profesional, y también ofrecer una plataforma robusta y estable para la investigación y desarrollo de los lenguajes dinámicos y los entornos de programación [15].

Pharo es altamente portable: incluso su máquina virtual está escrita en sí mismo [16], haciéndola fácil de depurar, analizar, y cambiar. El proyecto fue iniciado en marzo del 2008 como un *fork* de Squeak, y mediante desarrollo incremental e iterativo ha ido evolucionando, alejándose de a poco del Smalltalk original mediante la inclusión de nuevas características. Sin embargo, su sintaxis y la esencia de su semántica siguen siendo las mismas. A continuación estas se describen.

Smalltalk

Smalltalk es un lenguaje orientado a objetos, dinámicamente tipado y reflexivo. Smalltalk fue creado como el lenguaje que daría sustento al "nuevo mundo" de la computación, ejemplificado por la "simbiosis humano-computadora" [17]. Fue diseñado y creado, en parte para uso educativo, en el *Learning Research Group* (LRG) de Xerox PARC por Alan Kay, Dan Ingalls, Adele Goldberg, Ted Kaehler, Scott Wallace y otros durante la década de 1970.

Orientación a objetos

Smalltalk fue uno de los lenguajes que surgieron inspirados en Simula, el primer lenguaje orientado a objetos, y ha influenciado a prácticamente todos los lenguajes OOP que aparecieron después. En Smalltalk, el concepto central es el *objeto*. Un objeto siempre es una instancia de una clase, la cual describe las propiedades y comportamiento de sus instancias. Un objeto de Smalltalk puede hacer exáctamente tres cosas:

1. Mantener *estado* (referencias a otros objetos).

2. Recibir mensajes desde sí mismo u otros objetos.

3. Al recibir un mensaje puede enviar mensajes a sí mismo u otros objetos, o modificar su estado.

El estado que un objeto mantiene es siempre privado para ese objeto. Otros objetos pueden consultar o cambiar ese estado sólo mediante el envío de mensajes al objeto para que él mismo lo haga. Cualquier mensaje puede ser enviado a cualquier objeto: cuando un mensaje es recibido, el receptor es responsable de determinar si el mensaje es apropiado, y reaccionar de acuerdo.

Smalltalk es *puramente* orientado a objetos en el sentido de que no existe distinción entre valores que son objetos y valores que son tipos primitivos. En Smalltalk, valores primitivos como los enteros y los caracteres también son objetos, en el sentido de que son instancias de sus clases correspondientes y las operaciones sobre ellos son invocadas mediante el envío de mensajes. Un programador puede cambiar o extender (mediante subclases) las clases que implementan valores primitivos, de modo que nuevo comportamiento puede ser definido para sus instancias — por ejemplo, para implementar nuevas estructuras de control — o incluso se puede cambiar su comportamiento existente. Este hecho es resumido en la frase "en Smalltalk todo es un objeto", lo cual puede ser expresado más correctamente como "todos los valores son objetos", ya que las variables no lo son.

Como todos los valores son objetos, las clases también lo son. Cada clase es una instancia de la *metaclase* de esa clase. Las metaclases, a su vez, también son objetos, y todas son instancias de una clase llamada *Metaclass*. Los *bloques* — funciones anónimas en Smalltalk — también son objetos [18].

Reflexión

Reflexión es la capacidad que poseen algunos programas de inspeccionar su propia estructura. En un principio, la reflexión fue una cualidad eminentemente de lenguajes interpretados como Smalltalk [2] y Lisp. El hecho de que las expresiones sean interpretadas significa que los programas tienen acceso a información creada a medida que fue parseada y pueden, a menudo, modificar su propia estructura.

La reflexión también es una característica que surge a partir de la existencia de un meta-modelo como el que tiene Smalltalk. El meta-modelo es el modelo que describe el lenguaje en sí, y los desarrolladores pueden usar el meta-modelo para hacer cosas como examinar y modificar el *parse tree* de un objeto, o encontrar todas las instancias de una determinada clase.

Smalltalk-80 es un sistema totalmente reflexivo, implementado en sí mismo. La estructura

[2]Hoy en día los programas Smalltalk suelen ser compilados a bytecode, el cual es luego interpretado por una máquina virtual o dinámicamente traducido a código máquina nativo por un comilador JIT.

de todo el sistema está definida por objetos del lenguaje, incluyendo las clases, métodos, bloques, tipos primitivos e incluso los métodos compilados. Estos últimos son representados por instancias de *CompiledMethod*, los cuales son asociados a su clase respectiva mediante su adición al diccionario de métodos de la misma, por lo que es posible añadir y modificar métodos en tiempo de ejecución, del mismo modo que es posible la definición, creación y modificación de clases dinámicamente. Estas cualidades permiten modificaciones muy profundas en el sistema en tiempo de ejecución.

Como las clases son objetos, estas pueden ser consultadas —mediante el envío de mensajes— para responder preguntas tales como "¿qué métodos implementas?" o "¿qué variables de instancia defines?". Entonces los objetos pueden ser fácilmente inspeccionados, copiados, serializados, deserializados, etc, usando código genérico válido para cualquier objeto del sistema. [19]

Smalltalk-80 también provee reflexión computacional, la habilidad de observar el estado computacional del sistema. En lenguajes derivados del Smalltalk-80 original —como Pharo— se puede acceder al *contexto*[3] de activación del método actual como un objeto nombrado mediante la pseudo-variable (una de seis palabras reservadas) *thisContext*. Mediante el envío de mensajes a *thisContext* el método activo actualmente puede hacer preguntas como "¿quién me envió este mensaje?". El sistema de excepciones es implementado usando esta característica.

Un ejemplo de aplicación de la reflexión en Smalltalk es el mecanismo para manejar errores. Cuando un objeto recibe un mensaje que no implementa, la máquina virtual envía el mensaje *#doesNotUnderstand:* con una reificación del mensaje como argumento. El mensaje (otro objeto, instancia de *Message*) contiene el selector del mensaje y un arreglo (instancia de *Array*) con sus argumentos. En un sistema interactivo de Smalltalk la implementación por defecto de *#doesNotUnderstand:* es una que abre una ventana reportando el error al usuario. A través de esta el usuario puede examinar el contexto en el cual el error ocurrió, redefinir el código defectuoso, y continuar, todo dentro del sistema, usando las capacidades reflexivas de Smalltalk-80. [20].

La imagen

Los sistemas de programación más populares separan el código estático del programa (definiciones de clases, funciones o procedimientos) del estado dinámico (como objetos u otras formas de datos). Cargan el código del programa cuando este comienza, y cualquier estado previo del programa debe ser recreado explícitamente a partir de archivos de configuración u otras fuentes de datos.

Sin embargo, los sistemas Smalltalk no hacen diferencia entre los datos del programa (objetos) y el código (clases). De hecho, como ya se mencionó, las clases son objetos. Por lo tanto, la mayoría de los sistemas Smalltalk almacenan el estado completo del programa (incluyendo los objetos que son clases y los que no lo son) en un archivo llamado *imagen*. La

[3]Un contexto representa el estado de una activación de método. El contexto es creado cuando un método se activa, y es terminado cuando el método retorna. En Smalltalk, un contexto es distinto de un *stack frame* porque este es manipulado como cualquier objeto y además puede editar su *sender* (el contexto que lo activó).

imagen luego puede ser cargada por la máquina virtual para restaurar el sistema a un estado previo. Esto fue inspirado por *FLEX*, un lenguaje creado por Alan Key y descrito en su tesis de magíster. [21]

Sintaxis

La sintaxis de Smalltalk es particularmente concisa. A continuación se detalla informalmente:

- **Comentarios**
  "Los comentarios se escriben entre comillas"
- **Variables temporales**
  |var|
  |var1 var2|
- **Asignación**
  var := statement
- **Statements**
  aStatement1. aStatement2
  aStatement1. aStatement2. aStatement3
- **Mensajes**
  receptor mensaje *(mensaje unario)*
  receptor + argumento *(mensaje binario)*
  receptor mensaje: argumento *(mensaje keyword)*
  receptor mensaje: argumento1 with: argumento2 *(mensaje keyword)*
- **Cascada de mensajes**
  receptor mensaje1; mensaje2
  receptor mensaje1; mensaje2: arg2; mensaje3: arg3
- **Bloques**
  [aStatement1. aStatement2]
  [:argument1 | aStatement1. aStatement2]
  [:argument1 :argument2 | | temp1 temp2 | aStatement1]
- **Retorno**
  ^ aStatement

Con respecto a los mensajes, los unarios no reciben argumentos, los binarios reciben exáctamente un argumento, y los mensjaes *keyword* reciben uno o más argumentos. Los mensajes unarios tienen la mayor precedencia, luego vienen los binarios, y por último los *keyword*. Los mensajes se envían de izquierda a derecha, pero pueden usarse paréntesis para alterar el orden. Por ejemplo, al evaluar la expresión $1 + 2 * 3$ obtenemos 9 como resultado, mientras que si evaluamos $1 + (2 * 3)$ obtenemos 7.

Las palabras reservadas del lenguaje son las siguientes:

- **self**, el receptor del mensaje.
- **super**, el receptor del mensaje, pero el *method lookup* comienza en la superclase.

- **nil**, la instancia única de la clase *UndefinedObject*.
- **true**, la instancia única de la clase *True*.
- **false**, la instancia única de la clase *False*.
- **thisContext**, el contexto de ejecución actual.

Los siguientes ejemplos ilustran los objetos más comunes que pueden ser escritos como valores literales en Smalltalk-80.

- **Integer**
  123
  2r1111011 *(123 en binario)*
  16r7B *(123 en hexadecimal)*
- **Float**
  123.4
  1.23e-4
- **Character**
  $a
- **String**
  'abc'
- **Symbol**
  #abc
- **Array**
  #(123 123.4 $a 'abc' #abc)

En Smalltalk es común referirse a los métodos según su identificador en forma de símbolo, también llamado *selector*. Por ejemplo, si se tiene una clase *A* con un método ***foo:*** *arg1* ***bar:*** *arg2* es común referirse a este como *#foo:bar:* sin anotar los argumentos. Si se quiere hacer mención al método con su clase respectiva la notación utilizada es *A»#foo:bar:*. Además, en Pharo el mensaje *» aSymbol* es implementado por la clase *Behavior* (superclase de *Class* y *Metaclass*) y permite obtener el *CompiledMethod* asociado al identificador *aSymbol*.

1.3. Benchmarks

En computación, un benchmark es la acción de ejecutar un programa computacional, un conjunto de programas, u otras operaciones, con el objetivo de medir el rendimiento relativo de un objeto, normalmente mediante la ejecución de varias pruebas. El término "benchmark" es también utilizado en referencia a elaborados programas diseñados para realizar benchmarks.

La realización de benchmarks suele estar asociada a la medición del rendimiento de ciertas características del hardware. Por ejemplo, del rendimiento de la operación de punto flotante de una CPU. Sin embargo, los benchmarks también son aplicables a software. Existen diversos tipos de benchmark: de componentes (microbenchmarks), de aplicaciones reales, sintéticos, de

entrada/salida, de bases de datos, paralelos, etc. Para este trabajo interesan los dos primeros, a los cuales nos referiremos respectivamente como *micro-benchmarks* y *macro-benchmarks*.

Micro-benchmarks: Miden el rendimiento de componentes aisladas del software. En el contexto de nuestro trabajo serán usados para medir el rendimiento de algunas operaciones relevantes de las colecciones de forma aislada.

Macro-benchmarks: Se refieren a la ejecución de software real. En este trabajo se utilizarán para medir el rendimiento de las colecciones en escenarios reales de uso.

Los benchmarks son de importancia central para este trabajo, ya que constituyen la forma de evaluar el rendimiento de la colección adaptativa diseñada en relación con el resto de las colecciones. Con respecto a la evaluación, interesa destacar los siguientes aspectos fundamentales:

Reproducibilidad: Los benchmarks deben ser reproducibles con mínimo esfuerzo.

Evaluación estadística rigurosa: Los resultados de las ejecuciones de los benchmarks deben ser evaluados estadísticamente y presentados de acuerdo a esta evaluación.

El principio de reproducibilidad es fundamental para el método científico: los resultados deben ser reproducibles para que tengan valor. Esto implica que debemos contar con una infraestructura que permita escribir los benchmarks una vez y luego ejecutarlos cuando se necesite.

La evaluación estadística rigurosa es también muy importante, ya que los benchmarks son afectados por una gran cantidad de factores que hacen que los resultados varíen entre una ejecución y otra. Por esto, los resultados deben presentarse como un promedio de varias ejecuciones, y debe realizarse un análisis estadístico para determinar la existencia de diferencias estadísticamente significativas entre las alternativas en comparación, y la magnitud de estas diferencias.

En la sección siguiente se describirá la metodología de evaluación estadística que se utilizará para los benchmarks a lo largo de este trabajo. Luego, se describirá el framework de benchmarks *SMark*, el cual se utilizará para la realización de los micro-benchmarks. Los macro-benchmarks harán uso de una infraestructura propia.

1.3.1. Evaluación estadística

El rendimiento de las aplicaciones es difícil de medir debido a que es afectado por diversos factores como su *input*, la máquina virtual, el recolector de basura, el tamaño del *heap*, etc. Además, el no-determinismo puede causar que el tiempo medido varíe entre una ejecución y otra. Hay diversas fuentes de no-determinismo, como la compilación *Just-In-Time* (JIT), la planificación de procesos, la recolección de basura, y otros efectos del sistema [22, 23].

Por esto, para obtener buenas conclusiones es imprescindible que el análisis de los datos sea estadísticamente riguroso. A continuación se expone algo de teoría estadística fundamental,

y se discute cómo esta puede aplicarse en el contexto del análisis de rendimiento de software.

Errores

Los errores experimentales se clasifican en dos grupos: errores sistemáticos y errores aleatorios. Los errores sistemáticos son típicamente debido a problemas experimentales o procedimientos incorrectos que introducen un sesgo en las mediciones. Estos errores obviamente afectan la precisión de los resultados. Es responsabilidad del experimentador controlar y eliminar los errores sistemáticos. Si no las conclusiones, incluso con un análisis estadísticamente riguroso de los datos, pueden ser engañosas.

Los errores aleatorios, en cambio, son impredecibles y no-determinísticos. No están sesgados, ya que pueden disminuir o incrementar el valor de la medición. Pueden haber muchas fuentes de errores aleatorios en el sistema. En la práctica, una precoupación importante es la presencia de eventos perturbantes que no están relacionados a lo que el experimentador intenta medir, tales como los eventos externos del sistema, que producen la aparición de *outliers*. Los *outliers* deben ser examinados con cuidado, y si son resultado de perturbaciones externas deben ser descartados.

Si bien es imposible predecir los errores aleatorios, sí es posible desarrollar un modelo estadístico que describa el efecto de los errores aleatorios en los resultados experimentales, lo cual haremos a continuación.

Intervalo de confianza de la media

En cada experimento se toma una serie de muestras de una población subyacente. Un intervalo de confianza para la media derivado a partir de estas muestras cuantifica el rango de valores que tiene una cierta probabilidad de incluir la media de la población. A pesar que la forma de computar el intervalo de confianza de la media es similar para todos los experimentos, debe hacerse una distinción dependiendo del número de muestras obtenidas [24]: (i) el número n de muestras es grande (típicamente $n \geq 30$), y (ii) el número n de muestras es pequeño (típicamente $n < 30$). Debido a que en este trabajo para cada *benchmark* se tomaron al menos 30 mediciones, sólo se discutirá el primer caso.

La construcción de un intervalo de confianza requiere de diversas mediciones $x_\mathrm{i}, 1 \leq \mathrm{i} \leq n$, de una población con media μ y varianza σ^2. La media $\bar{x}$ de estas mediciones viene dada por

$$\bar{x} = \frac{\sum_{\mathrm{i}=0}^{n} x_\mathrm{i}}{n}. \tag{1.1}$$

Aproximaremos el valor de μ por la media $\bar{x}$ de nuestras mediciones y computaremos un rango de valores $[c_1, c_2]$ en torno a $\bar{x}$ que define el intervalo de confianza para una probabilidad dada (llamada el nivel de confianza). El *intervalo de confianza* $[c_1, c_2]$ es tal que la probabilidad de que μ se encuentre entre c_1 y c_2 es igual a $1 - \alpha$; α es denominado *nivel de*

significación y $1-\alpha$ es llamado *nivel de confianza.*

El cómputo del intervalo de confianza está basado en el teorema central del límite. Este dice que, para valores grandes de n (típicamente $n \geq 30$), la distribución de $\bar{x}$ se aproxima a una distribución normal con media μ y desviación estandar $\sigma/\sqrt{n}$ [25]. A esta desviación estandar (de la distribución de $\bar{x}$) se le llama *error estandar*, y se suele denotar por $SE_{\bar{x}}$.

Debido a que el nivel de significación α es escogido a priori, debemos encontrar c_1 y c_2 tales que $\Pr[c_1 \leq \mu \leq c_2] = 1-\alpha$. Típicamente, c_1 y c_2 son escogidos para formar un intervalo simétrico en torno a $\bar{x}$, o sea, $\Pr[\mu < c_1] = \Pr[\mu > c_2] = \alpha/2$. Aplicando el teorema central del límite encontramos que

$$c_{1,2} = \bar{x} \ \pm \ z_{1-\alpha/2}\frac{s}{\sqrt{n}}, \tag{1.2}$$

donde $\bar{x}$ es la media de la muestra, n es el número de mediciones y s es la desviación estandar de la muestra, computada como sigue:

$$s = \sqrt{\frac{\sum_{\mathrm{i}=1}^{n}(x_{\mathrm{i}} - \bar{x})^2}{n-1}}. \tag{1.3}$$

El valor $z_{1-\alpha/2}$ es tal que, si Z es una variable aleatoria distribuida normalmente con media $\mu = 0$ y varianza $\sigma^2 = 1$, se cumple que:

$$\Pr[Z \leq z_{1-\alpha/2}] = 1 - \alpha/2. \tag{1.4}$$

La tabla 1.1 muestra algunos valores comunes de $z_{1-\alpha/2}$ en función del nivel de confianza $1-\alpha$.

$\mathbf{1-\alpha}$	$\mathbf{z_{1-\alpha/2}}$
68.27 %	1.000
80.00 %	1.282
90.00 %	1.645
95.00 %	1.960
95.45 %	2.000
99.00 %	2.576
99.73 %	3.000

Tabla 1.1: Valores comunes de $z_{1-\alpha/2}$ en función de $1-\alpha$

Es importante enfatizar que el cálculo de intervalos de confianza no requiere que los datos subyacentes se distribuyan normalmente. El teorema central del límite indica que $\bar{x}$ sigue una distribución normal independientemente de la distribución de la población de la que se tomaron las mediciones. En otras palabras, aunque la población no distribuya normalmente la media de sus mediciones sí lo hará si las mediciones son independientes entre sí.

Comparación de alternativas

Hasta ahora hemos calculado el intervalo de confianza de la media para un solo sistema. En relación a los benchmarks, esto permite obtener un intervalo de confianza para el tiempo de ejecución de cada uno de ellos. Pero, ¿cómo comparar los resultados de un mismo benchmark al cual se le han variado algunos parámetros (por ejemplo, la implementación de colecciones usada)? A continuación abordaremos este problema.

El enfoque más simple para comparar dos alternativas consiste en determinar si los intervalos de confianza para los dos conjuntos de mediciones se sobreponen. Si se sobreponen, no podemos concluir que las diferencias observadas en los valores de la media no sean debido a fluctuaciones aleatorias en las mediciones. O sea, la diferencia observada es posiblemente debida a efectos aleatorios. Si los intervalos de confianza no se sobreponen, concluimos que no hay evidencia que sugiera que no hay una diferencia estadísticamente significativa. Nótese el uso cuidadoso de las palabras. Sigue existiendo una probabilidad α de que las diferencias observadas en nuestras mediciones sean simplemente debido a efectos aleatorios de las mediciones. En otras palabras, no podemos asegurar con 100 % de certeza que hay una verdadera diferencia entre las alternativas comparadas. Sin embargo, esto es lo mejor que se puede hacer dada la naturaleza estadística de las mediciones [22].

Considérense ahora dos alternativas con n_1 mediciones y n_2 mediciones, respectivamente. Primero determinamos las medias muestrales $\bar{x_1}$ y $\bar{x_2}$, y las desviaciones estandar de las muestras s_1 y s_2. Luego calculamos la diferencia de las medias como $\bar{x} = \bar{x_1} - \bar{x_2}$. La desviación estandar s_x de esta diferencia de valores de la media viene dada por:

$$\bar{s_x} = \sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}. \tag{1.5}$$

Entonces, el intervalo de confianza $[c_1, c_2]$ de la media de las diferencias queda como:

$$c_{1,2} = \bar{x} \ \pm \ z_{1-\alpha/2} s_x. \tag{1.6}$$

Si este intervalo incluye cero podemos concluir que, al nivel de confianza escogido, no hay diferencia estadísticamente significativa entre las alternativas.

Si bien la diferencia absoluta entre las medias es útil, la variación porcentual entre las mismas permite observar de forma más clara la magnitud del cambio en rendimiento de un benchmark entre una configuración y otra. La variación porcentual $f(x, y)$ de un valor inicial x a un valor final y viene dada por:

$$f(x, y) = 100 * \frac{y - x}{x}. \tag{1.7}$$

Para obtener un intervalo de confianza para la variación porcentual entre las medias mues-

trales $\bar{x_1}$ y $\bar{x_2}$, tenemos que considerar la propagación de los errores. En este caso, buscamos obtener la varianza de $f(x, y)$. Tenemos que

$$\begin{aligned} \mathrm{Var}\left(f(x,y)\right) &= \mathrm{Var}\left(100 * \frac{y-x}{x}\right) \\ &= 100^2 * \mathrm{Var}\left(\frac{y}{x} - 1\right) \\ &= 100^2 * \mathrm{Var}\left(\frac{y}{x}\right) \end{aligned} \tag{1.8}$$

Suponiendo que x e y son variables aleatorias independientes que distribuyen normalmente, tenemos que [26]:

$$\mathrm{Var}\left(\frac{y}{x}\right) \approx \left(\frac{y}{x}\right)^2 \left(\frac{s_x^2}{x^2} + \frac{s_y^2}{y^2}\right), \tag{1.9}$$

donde s_x representa la desviación estandar de x y s_y representa la desviación estandar de y. Sea s_f la desviación estandar de f, a partir de 1.8 y 1.9 tenemos que:

$$s_f(\bar{x_1}, \bar{x_2}) = 100 * \left|\frac{\bar{x_2}}{\bar{x_1}}\right| \sqrt{\frac{s_1^2}{\bar{x_1}^2 n_1} + \frac{s_2^2}{\bar{x_2}^2 n_2}}, \tag{1.10}$$

donde s_x y s_y han sido reemplazados por el error estandar de las medias $\bar{x_1}$ y $\bar{x_2}$ respectivamente. Finalmente, el intervalo de confianza $[c_1, c_2]$ para la variación porcentual de las medias $\Delta\bar{x}$ viene dado por:

$$c_{1,2} = 100 * \left(\frac{\bar{x_2} - \bar{x_1}}{\bar{x_1}} \quad \pm \quad z_{1-\alpha/2} \left|\frac{\bar{x_2}}{\bar{x_1}}\right| \sqrt{\frac{s_1^2}{\bar{x_1}^2 n_1} + \frac{s_2^2}{\bar{x_2}^2 n_2}}\right) \tag{1.11}$$

Nuevamente, si este intervalo incluye cero podemos concluir que, al nivel de confianza escogido, no hay diferencia estadísticamente significativa entre las alternativas.

Outliers

El no-determinismo en el tiempo de ejecución de una aplicación debido a los múltiples factores externos que pueden perturbarla —como son el recolector de basura, la planificación de procesos, el compilador JIT, las llamadas a sistema, etc— es una fuente de *outliers*. Los *outliers* son valores que se encuentran muy lejos del resto de los valores de la medición, y suelen indicar eventos perturbadores. Por ejemplo, si se gatilla una recolección de basura completa durante una de las iteraciones del benchmark pero no en las demás, esa perturbación incrementará notablemente el tiempo de ejecución de esa iteración, distorsionando la

estimación de la media. Por eso, es importante en primer lugar tratar de evitar la aparición de *outliers* minimizando las fuentes de perturbaciones externas. En el caso de la recolección de basura, forzando una recolección completa antes de cada iteración del benchmark. Sin embargo, las fuentes de perturbaciones son diversas y complejas, por lo que el experimentador nunca podrá evitar con certeza la aparición de *outliers*.

Es entonces relevante contar con una forma de identificar *outliers* para luego eliminarlos de los resultados, y así obtener estimaciones más precisas de las variables que se quieren medir. En este trabajo se hará uso del criterio de Peirce cuando se sospeche la existencia de *outliers* en las mediciones. Este criterio es un método riguroso que permite la eliminación de los *outliers* basado en fundamentos sólidos de teoría de probabilidades [27].

El criterio de Peirce se basa en siguiente principio:

> *"... las observaciones propuestas deben ser rechazadas cuando la probabilidad del sistema de errores obtenido al retenerlas es menor que la del sistema de errores obtenido al rechazarlas multiplicado por la probabilidad de hacer tal cantidad, y no más, de observaciones anormales" [28]*

El método original propuesto por Peirce es complicado de usar, por lo que este fue simplificado mediante el uso de tablas pre-calculadas, derivadas del trabajo de Peirce. Estas tablas contienen el valor para un parámetro R en función del número de observaciones y el número de estas que son dudosas. La tabla 1.2 muestra los valores de R para un número de hasta 12 observaciones. El parámetro R satisface la siguiente ecuación:

$$R = \frac{|x_\mathrm{i} - x_m|_{max}}{\sigma}, \tag{1.12}$$

donde σ es la desviación estandar de la muestra, x_i es el i-ésimo valor de la muestra y x_m es la media de la misma. $|x_\mathrm{i} - x_m|_{max}$ corresponde a la máxima desviación de la media permitida para un dato.

El método para determinar qué valores deben ser rechazados se describe a continuación:

1. Se calcula la media y desviación estandar del conjunto de datos.

2. Se obtiene el R correspondiente al número de mediciones, a partir de la tabla. Se supone el caso de *una* sola observación dudosa, incluso aunque parezca haber más de una.

3. Se calcula la máxima desviación permitida según la ecuación 1.12.

4. Para cada elemento x_i de la muestra:

   - Se calcula $|x_\mathrm{i} - x_m|$
   - Si $|x_\mathrm{i} - x_m| > |x_\mathrm{i} - x_m|_{max}$, se elimina x_i del conjunto de datos a considerar.

5. Si el paso anterior resulta en la eliminación de uno o más elementos, se actualiza el valor de R usando como número de elementos "dudosos" la cantidad de elementos eliminados

hasta ahora más uno, y se vuelve al paso 3. El promedio y desviación estandar obtenidos en el paso 1 se mantienen constantes.

6. Se calcula el nuevo valor para la media y desviación estandar de la muestra sin los *outliers* eliminados. Termina el proceso.

	Observaciones dudosas						
Total observaciones	1	2	3	4	5	6	7
3	1.196						
4	1.383	1.078					
5	1.509	1.200					
6	1.610	1.299	1.099				
7	1.693	1.382	1.187	1.022			
8	1.763	1.453	1.261	1.109			
9	1.824	1.515	1.324	1.178	1.045		
10	1.878	1.570	1.380	1.237	1.114		
11	1.925	1.619	1.430	1.289	1.172	1.059	
12	1.969	1.663	1.475	1.336	1.221	1.118	1.009

Tabla 1.2: Valores de R

1.3.2. SMark

SMark, antes llamado *PBenchmark*, es un framework para Pharo que permite escribir benchmarks como si fueran tests unitarios [29]. El framework ejecuta cada benchmark n veces, recopila cada resultado $x_\mathrm{i}, 1 \leq \mathrm{i} \leq n$, y luego presenta el intervalo de confianza $c_{1,2}$, dado por la expresión 1.2, para un nivel de confianza $1 - \alpha = 90\,\%$.

Uso del framework

En SMark los benchmarks son agrupados en *suites*. Cada *suite* es una subclase de *SMarkSuite*, y puede contener múltiples benchmarks que son definidos como métodos de la *suite*. Para que un método sea considerado benchmark por el framework, el identificador del mismo debe contener el prefijo *bench*. Al llamar al método de clase *#run:* de la *suite* con un parámetro entero n, el framework recopilará todos los métodos que comiencen con el prefijo *bench* y ejecutará cada uno de ellos n veces, midiendo el tiempo de cada ejecución. Luego se calcularán, para cada benchmark, los intervalos de confianza de la media de los tiempos según fue mencionado anteriormente, y se presentarán los resultados al usuario.

Cada *suite* puede además proveer los métodos *#setUp* y *#tearDown*, comunes en los frameworks de *unit testing*, que sirven para realizar tareas de inicialización y limpieza respectivamente. El método *#setUp* se ejecutará antes de la ejecución de cada iteración de cada benchmark, y el método *#tearDown* se ejecutará después de la ejecución de cada iteración de cada benchmark. Adicionalmente, es posible usar *setUp* y *tearDown* como prefijos

de los nombres de método de los benchmarks, en cuyo caso se ejecutarán antes/después del benchmark en cuestión.

Por ejemplo, si una subclase *S* de *SMarkSuite* define los métodos *#benchFoo*, *#setUp*, *#setUpBenchFoo* y *#tearDown*, al llamar *S run: 100* se creará una instancia *s* de *S* y se ejecutará 100 veces la siguiente secuencia: *s setUp*, *s setUpBenchFoo*, *s benchFoo*, *s tearDown*. Se tomarán los tiempos de cada ejecución de *#benchFoo* y al final se entregará el intervalo de confianza del tiempo promedio para el benchmark *Foo*.

1.4. Profiling

En ingeniería de software, *profiling* se refiere a la obtención de información dinámica a partir de una ejecución controlada de un programa [30]. Es una técnica usada para una amplia variedad de tareas. Entre sus usos más comunes encontramos: el monitoreo del tiempo de ejecución, el monitoreo del consumo de memoria, la evaluación de la cobertura del código por parte de los tests unitarios, la *retroalimentación de tipos*[4], y la comprensión del software.

El análisis de la información recopilada en tiempo de ejecución provee pistas importantes acerca de cómo mejorar la ejecución de un programa. Esta información suele ser presentada en forma de mediciones numéricas, tales como el número de invocaciones de métodos o el número de objetos creados en un método, lo cual la hace fácil de comparar entre una ejecución y otra.

El *profiling* se realiza a través de una herramienta llamada *profiler*. Los *profilers* usan una amplia variedad de técnicas para recopilar información, incluyendo las interrupciones de hardware, interrupciones de software, instrumentación del código, simulación del conjunto de instrucciones de la CPU, entre otras. Según cómo recopilen la información, los *profilers* se dividen en varios tipos. En el caso de este trabajo interesan los dos siguientes:

Instrumentación Esta técnica consiste en añadir instrucciones al programa objetivo para recolectar la información requerida. La instrumentación de un programa puede causar cambios en el rendimiento, por lo que puede llevar a resultados imprecisos en algunos casos. El efecto dependerá de qué información se está recopilando, y del nivel de detalle requerido. La instrumentación puede realizarse en diferentes niveles, como el código fuente, el binario compilado, o en tiempo de ejecución.

Muestreo Algunos profilers operan mediante muestreo. Un profiler de este tipo obtiene muestras del *puntero de instrucciones* del programa objetivo a intervalos regulares de tiempo mediante interrupciones. Los *profiles* obtenidos mediante muestreo suelen ser menos precisos numéricamente, pero permiten que el programa objetivo funcione casi inalterado. Los datos resultantes son una aproximación estadística. En la práctica, muchas veces los profilers de

[4]La retroalimentación de tipos (*type feedback*) se refiere a la extracción de información de tipos a partir de una ejecución para luego entregarle esta información al compilador [31].

muestro proveen una imagen más precisa de la ejecución del programa objetivo que otros enfoques, debido a que producen menos efectos colaterales.

A continuación se expone acerca de dos herramientas de *profiling* usadas para el desarrollo de este trabajo: el framework *Spy*, y el profiler *MessageTally*.

1.4.1. Spy

Spy es un framework para el desarrollo de profilers, basado en la instrumentación en tiempo de ejecución. La información dinámica retornada por el profiler está estructurada según la estructura estática del programa, expresada en términos de paquetes, clases y métodos [30].

Spy ha sido usado para implementar diversos tipos de profilers, incluyendo profilers de ejecución, de consumo de memoria, de cobertura de tests, un mecanismo de retroalimentación de tipos, analizadores de dependencias entre métodos y clases, entre otros. La creación de un nuevo profiler es de bajo costo ya que Spy libera al programador de la necesidad de realizar monitoreo de bajo nivel.

Funcionamiento del framework

Las clases principales de Spy son *Profiler*, *PackageSpy*, *ClassSpy* y *MessageSpy*, las cuales se explican más adelante. Para crear un profiler se debe extender de la clase *Profiler*. Esta clase permite la obtención de información en tiempo de ejecución mediante el *profiling* de la ejecución de un bloque de código Smalltalk. La clase *Profiler* ofrece diversos métodos de clase públicos para controlar el profiling. El método *#profile:inPackagesNamed:* acepta como primer parámetro un bloque y como segundo parámetro una colección de nombres de paquetes. El efecto de llamar a este método es (i) instrumentar los paquetes especificados; (ii) ejecutar el bloque provisto; (iii) desinstrumentar los paquetes instrumentados; y (iv) retornar los datos obtenidos en la forma de una instancia de la clase que contiene instancias de las clases descritas a continuación, esencialmente imitando la estructura del programa.

PackageSpy contiene los datos de profiling para un paquete. Cada instancia tiene un nombre y contiene un conjunto de *class spies*. Para cada clase en el paquete correspondiente se crea un *class spy*.

ClassSpy describe una clase de Pharo. Sus atributos son: nombre, *superclass spy*, *metaclass spy*, y un conjunto de *method spies*. Para cada método en la clase correspondiente, la instancia *class spy* crea un *method spy*.

MethodSpy envuelve un método de Pharo y acumula información durante la ejecución del programa[5]. Posee un nombre de selector y pertenece a un *class spy*. MethodSpy es central para Spy ya que posee los *hooks* para realizar la recopilación de la información en tiempo de ejecución. Se proveen dos métodos para este propósito: *#before-*

[5]MethodSpy es implementado como un *wrapper* del método.

Run:with:in: y *#afterRun:with:in:*, los cuales son ejecutados antes y después del método Pharo correspondiente. Estos métodos están vacíos por defecto; deben ser sobreescritos en las subclases de *MethodSpy* para recolectar la información dinámica relevante. Los parámetros pasados a estos métodos son: el nombre del método (como símbolo), la lista de argumentos, y el objeto que recibe el mensaje interceptado.

El framework Spy se usa creando subclases de PackageSpy, ClassSpy, MethodSpy y Profiler, las cuales deben especializarse para obtener la información en tiempo de ejecución precisa que se requiere para una tarea y sistema en particular.

Actualmente, el framework se encuentra en su segunda versión (Spy2), y sus servicios disponibles han aumentado. Dentro de las nuevas prestaciones destaca el sistema de *plugins* que permite modificar el funcionamiento del profiler mediante la inclusión de módulos reutilizables. Estos *plugins* permiten personalizar la instrumentación para, por ejemplo, capturar algún método en particular de una clase que no es parte de los paquetes a instrumentar, o para reemplazar un determinado *literal*[6] por otro en tiempo de instrumentación de cada método. Estas capacidades son de gran utilidad para desarrollar la evaluación de la colección adaptativa en este trabajo.

1.4.2. MessageTally

MessageTally es un profiler que opera mediante muestreo de la ejecución, y permite obtener la distribución de tiempo de ejecución de una computación. Está implementado como una clase única del mismo nombre, y funciona mediante la recepción del mensaje *#spyOn:* junto a un bloque que se pasa como argumento. El bloque es evaluado y al finalizar la ejecución se despliega una ventana con la siguiente información [32]:

1. un árbol jerárquico mostrando los métodos ejecutados con su tiempo de ejecución asociado durante la evaluación de la expresión contenida en el bloque.

2. los métodos hoja de la ejecución. Un método hoja es aquel que no invoca a otros métodos.

3. estadísticas acerca del consumo de memoria y el funcionamiento del recolector de basura.

El árbol muestra el tiempo que tardó la ejecución de cada método, y cuánto tardaron sus nodos hijos (métodos a los cuales llama). Los métodos se ordenan según tiempo de ejecución, de mayor a menor, y junto al tiempo se provee un porcentaje representando la proporción del tiempo total de la ejecución del bloque.

La parte de consumo de memoria muestra los cambios observados en la cantidad de memoria asignada y el uso del recolector de basura. En Pharo, el recolector de basura funciona bajo el principio de que un objeto *viejo* tiene una mayor probabilidad de seguir referenciado

[6]En un método compilado de Pharo (instancia de *CompiledMethod*), se denomina literal a cada identificador de clase o método presente en el código del método.

en el futuro que uno *joven*. Los objetos jóvenes pueden ser promovidos a viejos. MessageTally describe el uso de memoria usando cuatro valores:

1. el valor *old* cuantifica la variación del espacio de memoria dedicado a los objetos viejos. Un objeto califica como "viejo" cuando su localización en la memoria física está en el "espacio viejo". Esto ocurre cuando se gatilla una recolección de basura completa, o cuando hay muchos objetos sobrevivientes (de acuerdo a un umbral especificado en la máquina virtual). Este espacio de memoria sólo se limpia durante una recolección de basura completa, por lo que un recolector de basura incremental no reduce su tamaño.

2. el valor *young* cuantifica la variación del espacio de memoria dedicado a los objetos jóvenes. Cuando un objeto es creado, este se ubica físicamente en el "espacio joven" de la memoria. El tamaño de este varía frecuentemente.

3. el valor *used* corresponde al total de memoria usada y se calcula como *old* + *young*.

4. el valor *free* cuantifica la variación en la memoria disponible, y corresponde a $-$*used*.

El profiler también provee estadísticas acerca del recolector de basura, representadas por los siguientes valores:

1. el valor *full* indica la cantidad de recolecciones de basura completas que se llevaron a cabo y el tiempo que tardó. La recolección de basura completa no es muy frecuente, y suele ser resultado de la asignación de grandes porciones de memoria.

2. el valor *incr* indica las recolecciones de basura incrementales. La recolección incremental es usada frecuentemente (varias veces por segundo) y toma poco tiempo (alrededor de 1 ms). La cantidad de tiempo invertida en recolecciones de basura incrementales durante la ejecución no debiera superar el 10 % del total.

3. el valor *tenures* señala la cantidad de objetos que migraron al "espacio viejo" de memoria. Esta migración ocurre cuando el tamaño del "espacio joven" está sobre cierto umbral. Esto suele suceder al lanzar una aplicación, cuando todos los objetos necesarios aún no han sido creados y referenciados.

Para conseguir estas mediciones, MessageTally hace uso de las capacidades reflexivas de Pharo. El método *spyEvery: millisecs on: aBlock* contiene toda la lógica del *profiling*. Este método es llamado indirectamente por *#spyOn:*. El valor *millisecs* es la cantidad de milisegundos entre cada muestra, y por defecto es 1. El bloque sobre el que se realiza el profiling es *aBlock*.

El muestreo está controlado por un objeto *Timer*. Timer es un nuevo proceso al cual se le asigna una alta prioridad, y está a cargo de monitorear *aBlock*. Para esto, simplemente crea un ciclo infinito que espera cierta cantidad de milisegundos entre cada ciclo y obtiene una *snapshot* del *call stack* de métodos.

La idea es asociar a cada contexto de método un contador. Esta asociación se realiza mediante instancias de la clase *MessageTally* (la cual define las variables *class*, *method* y *process*). En cada ciclo de muestreo, el contador de cada *frame* del *stack* es incrementado

con la cantidad de milisegundos transcurridos. El *stack frame* se obtiene enviando el mensaje *#suspendedContext* al proceso que ha sido interrumpido.

Las estadísticas de memoria se obtienen mediante la diferencia entre la cantidad de memoria consumida antes y después del *profiling*. *Smalltalk*, una instancia de la clase *SmalltalkImage*, contiene diversos métodos que permiten consultar la cantidad de memoria disponible.

Capítulo 2

Objetivos y Metodología

A continuación se describen las hipótesis que se busca validar o rechazar, el objetivo general y los objetivos específicos de este trabajo, y la metodología escogida para llevarlo a cabo.

2.1. Hipótesis

En este trabajo se desarrollará una colección inspirada en las tablas de Lua, y se estudiarán los beneficios de su utilización con respecto a las colecciones tradicionales. Específicamente, se busca confirmar o refutar las siguientes hipótesis:

H1 La colección adaptativa logra reducir, en promedio, el tiempo de ejecución de las aplicaciones al reemplazar colecciones tradicionales.

H2 La colección adaptativa logra reducir, en promedio, el consumo de memoria de las aplicaciones al reemplazar colecciones tradicionales.

2.2. Objetivo General

Diseñar e implementar una colección adaptativa que, tras una evaluación rigurosa, demuestre mejor rendimiento con respecto al de las colecciones expansibles tradicionales más utilizadas en lenguajes que poseen bibliotecas de colecciones sofisticadas.

2.2.1. Objetivos Específicos

1. Diseñar e implementar una colección adaptativa inspirada en las tablas de Lua.

2. Caracterizar el rendimiento en cuanto a uso de memoria y tiempo de ejecución de la colección adaptativa.

3. Comparar rigurosamente el rendimiento de la colección adaptativa con el de las colecciones tradicionales más comunes.

4. Identificar si el uso de una colección adaptativa de esta naturaleza es benéfico en el contexto de un lenguaje con una biblioteca de colecciones sofisticada.

2.3. Metodología

Para estudiar la colección adaptativa se hará uso del lenguaje/entorno de programación Pharo, ya que el lenguaje ofrece una rica biblioteca de colecciones y también posee capacidades reflexivas de alta flexibilidad y expresividad, las cuales son clave para llevar a cabo las mediciones detalladas a las que apuntamos. Además, provee herramientas de mucha utilidad para este propósito, como SMark[1], Spy[2] y MessageTally[3].

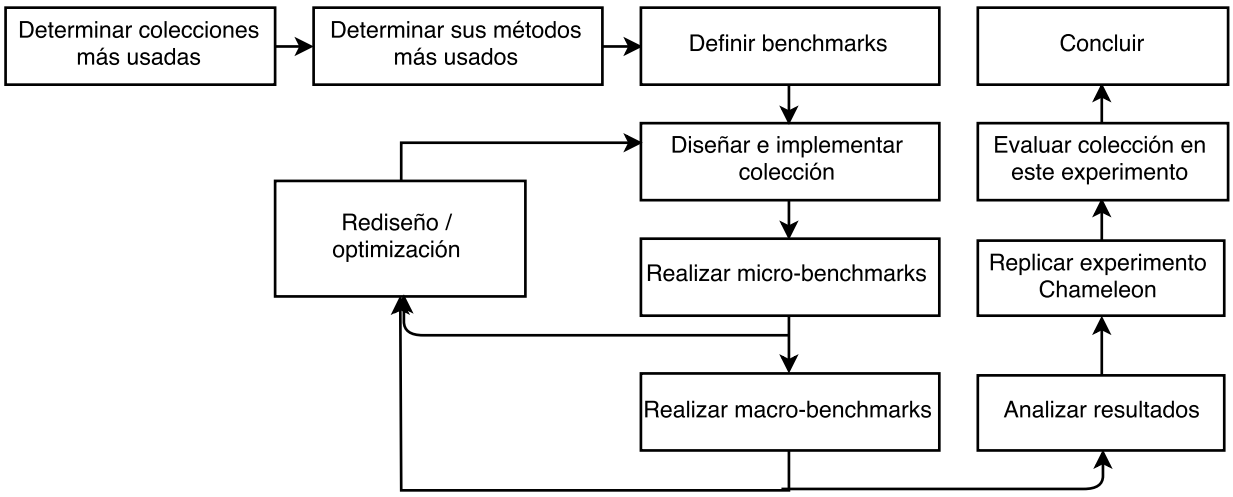

Figura 2.1: Metodología

La metodología escogida se ilustra en la figura 2.1 y se explica en los siguientes pasos:

1. Determinar las colecciones expansibles frecuentemente utilizadas en Pharo. Estas colecciones luego serán evaluadas en relación a la colección adaptativa.

2. Determinar los métodos comúnmente utilizados por las colecciones identificadas en el paso 1.

3. Definir un *benchmark*. Se determinará un conjunto de métricas a estudiar y se diseñará

[1] Ver 1.3.2
[2] Ver 1.4.1
[3] Ver 1.4.2

un *benchmark* [33, 23] para medirlas en un conjunto representativo de escenarios de uso real de colecciones.

4. Diseñar e implementar en Pharo la colección adaptativa. La colección debe ser polimórfica estructuralmente con respecto a las clases identificadas en el paso 1 en relación a sus métodos identificados en el paso 2.

5. Realizar micro-benchmarks para medir el tiempo de ejecución de los métodos identificados en el paso 2, y evaluar su costo individual.

6. Realizar macro-benchmarks para medir tiempo de ejecución y consumo de memoria usando aplicaciones de Pharo representativas. Estas aplicaciones serán adaptadas para reemplazar las colecciones expansibles populares identificadas en el paso 1 por la colección adaptativa.

7. Análisis de los resultados de los experimentos. En particular se comparará el rendimiento en cuanto a uso de memoria y tiempo de ejecución, entre la colección desarrollada y las colecciones normales, en los distintos escenarios ejercitados.

8. Rediseño u optimización de la colección basado en los resultados de los benchmarks.

9. Replicar experimento *Chameleon* en Pharo, y luego evaluar la colección adaptativa —en cuanto a tiempo de ejecución y consumo de memoria— como parte de este experimento.

10. Discutir resultados en relación a las hipótesis planteadas y concluir.

Para llevar a cabo esta metodología se requiere desarrollar algunas herramientas de análisis dinámico, en particular:

- Una infraestructura de benchmarks que permita reemplazar todas las referencias a una determinada clase por referencias a otra clase antes de ejecutar los experimentos. Debe medir tiempo de ejecución y consumo de memoria.
- Una profiler que permita obtener estadísticas de uso de las colecciones agrupadas según el sitio de creación de cada colección.
- Una herramienta que reciba como entrada estadísticas de uso de colecciones según sitio de creación y entregue recomendaciones de reemplazo de colecciones según un conjunto de reglas de reemplazo predefinidas. La herramienta además debe poder aplicar los cambios de forma automática.

El detalle acerca del diseño e implementación de estas herramientas se expone en el capítulo 4.3.

Capítulo 3

Desarrollo y Evaluación de la Colección

En esta sección se detalla el desarrollo de la colección adaptativa basada en las tablas de Lua, la cual en realidad se subdivide en tres versiones que fueron desarrolladas incrementalmente: SLua, SmartCollection y SmartCollection2. SLua es la más cercana a la tabla de Lua original, y fue desarrollada para ser lo más parecida posible a esta. SmartCollection surge como una mejora frente a algunas limitaciones que se identificaron en SLua, debido a diferencias en los ecosistemas de Lua y Pharo, las cuales se detallan más adelante. SmartCollection2 es un intento por mejorar aún más SmartCollection mediante la modificación de su estructura interna. Las tres colecciones son detalladas en cuanto a su estructura y funcionamiento. Adicionalmente, se presentan micro-benchmarks y macro-benchmarks para comparar las colecciones desarrolladas con las colecciones expansibles más populares de Pharo.

El capítulo comienza con el análisis de colecciones más comunes de Pharo, y los métodos más usados de estas, y continúa con la descripción de cómo se llevaron a cabo los micro-benchmarks expuestos más adelante. Luego se expone acerca de cada colección desarrollada y los resultados obtenidos para los micro-benchmarks. Finalmente se comparan las colecciones desarrolladas mediante macro-benchmarks.

3.1. Análisis de colecciones en Pharo

Antes de comenzar a describir las colecciones desarrolladas, se identificarán las colecciones expansibles más populares en Pharo y sus métodos más usados. De este modo se contará con una referencia frente a la cual comparar las colecciones desarrolladas mediante los benchmarks.

3.1.1. Colecciones expansibles populares en Pharo

Tomamos 124.292 métodos a partir de una gran muestra de código fuente de Pharo. De esos métodos, 11.995 (9.64 %) hacen uso de una colección. A partir de los métodos identificados,

encontramos que:

- 2.281 métodos (19 %) referencian a la clase *OrderedCollection*
- 2.067 métodos (17 %) referencian a la clase *String*
- 1.967 métodos (16 %) referencian a la clase *Array*
- 774 métodos (6 %) referencian a la clase *Dictionary*

Los restantes 4.906 métodos referencian al menos a una de las otras 150 clases de colecciones. Debido a que las clases *Array* y *String* corresponden a colecciones no expansibles, no pueden ser reemplazadas por nuestra implementación de la tabla de Lua. Como *OrderedCollection* y *Dictionary* son las dos colecciones expansibles más populares en Pharo, nuestras mediciones compararán la tabla de Lua con esas dos colecciones.

3.1.2. Métodos comúnmente usados

Las clases *OrderedCollection* y *Dictionary* definen 73 métodos y 82 métodos, respectivamente. Algunos de esos métodos son privados o raramente usados. Hemos analizado el uso de estas dos clases durante la ejecución de 5 aplicaciones grandes de Pharo. Identificamos 7 métodos no privados que son los más frecuentemente usados. Excluimos los métodos privados ya que no están hechos para ser invocados por usuarios finales de aplicaciones.

En la clase *OrderedCollection*, los métodos más frecuentemente usados son *addFirst:* (añade un elemento al principio de la colección), *addLast:* (añade un elemento al final), *do:* (itera sobre la colección), *reduce:* (operación *fold* de la programación funcional), y *remove:* (remueve un elemento en particular). En la clase *Dictionary*, el método *at:ifAbsent:* (obtiene un valor de una llave o retorna un valor por defecto en caso que la llave no esté presente) y *at:put:* (inserta un valor para una llave dada).

Debido a que estos métodos son frecuentemente usados, nuestros micro-benchmarks miden el costo de cada uno de ellos.

3.2. Micro-benchmarks

Para los micro-benchmarks se usó el framework *SMark* (originalmente *PBenchmark*[29]) de Pharo, el cual permite escribir benchmarks de modo similar a los test unitarios. Estos micro-benchmarks son útiles para verificar el correcto funcionamiento en cuanto a rendimiento de las colecciones desarrolladas, identificar métodos que requieren optimización, y para descubrir de forma rápida posibles regresiones en el rendimiento producidos por cambios en la colección.

Los micro-benchmarks utilizados hacen uso de colecciones de gran tamaño para ejercitar intensamente los métodos. Esto implica que representan escenarios de uso arbitrarios, por lo que la información que proporcionan no es extrapolable a todo caso de uso. Las colecciones pueden tener variaciones importantes de rendimiento dependiendo de la cantidad de

elementos que estén manejando y de la distribución de los mismos (en casos como #remove: y #at:put:, por ejemplo), por lo que para un análisis más detallado se requieren resultados en función de estos parámetros. En vez de eso, para la evaluación rigurosa se opta por macro-benchmarks que ejercitan escenarios de ejecución realistas.

3.3. SLua

Se desarrolló una colección en Pharo inspirada en las tablas de Lua. Nuestra implementación, a la cual nos referiremos como *SLua*, tomó en consideración la documentación de Lua, el código fuente de la tabla, y el código fuente de la máquina virtual de Lua.

Del mismo modo que las tablas de Lua, SLua es una combinación eficiente de un *arreglo asociativo* y una *colección expansible secuencial*, por lo que puede ser utilizada tanto como diccionario como lista. En cuanto a su implementación, esta también es una estructura híbrida que consta de una parte ordenada representada mediante un arreglo, y una parte de tabla de hash. Ambas estructuras internas son inicializadas de forma perezosa, por lo que sólo consumirán memoria si están siendo utilizadas.

SLua mantiene el invariante de la tabla de Lua: la parte de arreglo almacena los valores correspondientes a llaves enteras entre 1 y n, donde n es la mayor potencia de 2 tal que (i) al menos 50 % de los espacios entre 1 y n están en uso y (ii) hay al menos un espacio usado entre $\frac{n}{2}+1$ y n. Para esto, hace uso del mismo algoritmo de crecimiento descrito en la sección 1.1.

Sin embargo, debido a que la naturaleza de Pharo difiere de la de Lua, la tabla no fue portada tal cual, sino que fue necesario ajustarla. SLua difiere de las tablas de Lua en los siguientes aspectos:

- Lua es un lenguaje procedural mientras que Pharo es un lenguaje orientado a objetos basado en clases. Lua ofrece numerosas funciones para manipular las tablas. Estas funciones fueron implementadas como métodos en SLua.
- SLua extiende de *Collection*, superclase de todas las colecciones de Pharo, y posee una API con los mismos identificadores de los métodos de *OrderedCollection* y *Dictionary*, lo cual la hace polimórfica a estas clases. Esto implica que instancias de estas clases pueden reemplazarse directamente por instancias de SLua, lo que permite simplificar la implementación de los macro-benchmarks.

A pesar de estas diferencias, nuestra implementación es bastante cercana a la implementación original de las tablas. Mientras la tabla de Lua es una estructura que se manipula mediante las funciones incluidas en Lua, SLua es una clase cuyas instancias son objetos que mutan mediante la recepción de mensajes.

Los métodos definidos en SLua que corresponden a los métodos de *OrderedCollection* afectarán principalmente a la parte de arreglo mientras que los métodos que corresponden a *Dictionary* afectarán la parte de hash.

3.3.1. Limitaciones

Sea c una instancia de SLua, c_{i} un elemento de c indexado por i, y sea $c_{a,\mathrm{i}}$ el i-ésimo espacio del arreglo interno de c. Si i es un entero tal que $\mathrm{i} \leq n$, donde n es la capacidad computada para el arreglo interno de c, entonces tenemos que $c_{\mathrm{i}} = c(a, \mathrm{i})$. Esto viene dado por la estructura de la colección y es igual en la tabla de Lua.

Lo anterior implica que si una instancia de SLua está siendo utilizada como lista (*ie.* los elementos son añadidos de forma contígua, comenzando del índice 1) y luego se requiere agregar un elemento al principio de esta, necesariamente hay que copiar todos los elementos un espacio hacia adelante para liberar el primer espacio. Esto significa que la operación que añade un elemento al inicio de la colección (*#addFirst:* en el caso de la biblioteca de colecciones de Pharo) tomará un tiempo $O(s)$, donde s es la cantidad elementos de la lista.

En contraste, la implementación de *OrderedCollection*[1] garantiza que las adiciones tanto al inicio como al final de la colección sean en tiempo $O(1)$, lo cual haría a SLua muy inferior en cuanto a rendimiento en escenarios que hagan uso intensivo de *#addFirst:*. Debido a que este método está dentro de los más utilizados en las aplicaciones inspeccionadas, es probable que esta limitación tenga un impacto negativo bastante significativo en la práctica.

3.3.2. Micro-benchmarks

Se realizaron micro-benchmarks para comparar SLua con OrderedCollection en los métodos *#addLast:*, *#do:*, *#reduce:* y *#remove:*. El método *#addFirst:* fue omitido debido a la diferencia en complejidad de tiempo entre la implementación de OrderedCollection y la de SLua, la cual hace innecesaria una comparación pues sabemos que la de OrderedCollection será muchísimo más rápida. Los resultados de los micro-benchmarks se ven en la tabla 3.1.

Benchmark	**OrderedCollection**	**SLua**	$\boldsymbol{\Delta\bar{\mathbf{x}}}$ **(%)**
addLast:	38.72 ± 0.19	68.46 ± 0.21	76.81 ± 0.38
do:	9.23 ± 0.07	9.12 ± 0.04	-1.19 ± 0.65
reduce:	204.9 ± 3.8	242.27 ± 0.67	18.24 ± 0.53
remove:	107.42 ± 0.63	107.18 ± 0.62	-0.22 ± 3.03

Tabla 3.1: Micro-benchmarks SLua vs OrderedCollection, i = 1000

Las columnas segunda y tercera muestran la media de los i resultados de cada benchmark para OrderedCollection y SLua, respectivamente. La cuarta columna de la tabla, etiquetada por $\boldsymbol{\Delta\bar{\mathbf{x}}}$, expone la variación porcentual del tiempo de ejecución entre los resultados de OrderedCollection y SLua para ese benchmark. Todos los resultados son presentados en forma de intervalos de confianza al 90 %, calculados según 1.2 y 1.11.

En este caso se observa un menor rendimiento de parte de SLua en comparación con OrderedCollection en dos de los benchmarks, y un rendimiento prácticamente idéntico en los casos de *#do:* y *#remove:*. El rendimiento significativamente menor ($\approx$ 76 % de *overhead*)

[1] OrderedCollection es el equivalente a *ArrayList* en Pharo.

en *#addLast:* se explica mediante dos razones: (1) la lógica involucrada en cada inserción de elemento en SLua es más compleja que la de OrderedCollection, ya que *#addLast:* opera llamando a *#at:put:*, que se ocupa para la adición de elementos en cualquier índice de la colección, por lo que se hacen diversos chequeos, y (2) el algoritmo de crecimiento de la colección (ver 1.1) es más complejo y por ende más costoso en cuanto a tiempo. El rendimiento de *#reduce:* es menor en SLua ($\approx 18\,\%$ de *overhead*) debido a que como esta puede contener elementos en la parte de arreglo (contíguos o no, por lo que puede haber espacios con *nil* entre un elemento del arreglo y otro) y también en la parte de hash, se debe recorrer primero toda la colección para tomar todos sus elementos y luego recién aplicar la operación *#reduce:*. En contraste, OrderedCollection contiene todos sus elementos en un mismo arreglo, y todos estos se encuentran contíguos, por lo que *#reduce:* se aplica directamente sobre ellos. Finalmente, en los casos de *#do:* y *#remove:* los tiempos son muy similares debido a que no hay diferencias importantes con respecto a cómo operan.

Si bien los resultados de los micro-benchmarks no son necesariamente un reflejo de lo que ocurrirá al usar la colección en escenarios realistas —ya que por ejemplo puede haber diferencias en el consumo de memoria que impacten también en el tiempo de ejecución, y puede que la operación *#addFirst:* no sea invocada muchas veces en la práctica (el análisis métodos más usados que se describe en es estático, no dinámico)— el hecho de que *#addFirst:* tenga una complejidad de tiempo $O(n)$ en SLua versus el tiempo amortizado $O(1)$ que esta misma operación tiene en la implementación tradicional de una colección expansible secuencial en Pharo (OrderedCollection) plantea la necesidad de hacer ajustes para que esta colección funcione más rápidamente en presencia de esta operación. El impacto en la práctica de la mayor complejidad de *#addLast:* se medirá con los macro-benchmarks, al final de este capítulo.

3.4. SmartCollection

Con la finalidad de reducir las limitaciones de SLua expuestas anteriormente, se desarrolló otra versión de la colección a la cual llamaremos SmartCollection. SmartCollection mantiene tal cual la funcionalidad de SLua pero cambia la forma como almacena sus elementos. Ahora la parte de arreglo contiene todos los elementos cuyo índice es un entero menor o igual a s, donde s es el número de espacios ocupados (y no la capacidad) del arreglo interno. El funcionamiento del arreglo interno se explica a continuación.

3.4.1. Uso híbrido

Antes de pasar a los detalles del funcionamiento de la parte ordenada de la colección, mencionaremos algunos aspectos relevantes de su uso híbrido que deben ser considerados para el desarrollo de la parte secuencial.

Si una instancia c de SmartCollection es usada como lista y tabla de hash durante una misma ejecución, puede darse el caso que un elemento fue insertado en un índice entero positivo n fuera del rango $[1, s]$ y por ende quedó en la parte de hash, y luego la parte de

arreglo crece mediante inserciones hasta alcanzar ese índice. En ese caso, es necesario contar con un mecanismo para que cuando $s = n - 1$, el elemento indexado por n pase de la parte de hash a la de arreglo. Si es necesario transferir al elemento indexado por n a la parte de arreglo, también habrá que transferir a los elementos cuyo índice sea contíguo a n.

Definición 3.1 *Sea c una instancia de SmartCollection, sea s la cantidad de elementos en su parte secuencial, y sea I_c el conjunto de los índices enteros que tienen elementos asociados en c. El conjunto I' de índices de elementos en la parte de hash de c que son contíguos a la parte de arreglo de c se define por:*

1. $n_0 = s + 1 \quad (n_0 \in I)$

2. $n_i = n_{i-1} + 1 \quad (\{n_{i-1}, n_i\} \subset I)$

Para transferir los elementos de la parte de hash que son contíguos a la parte de arreglo según la definición anterior se desarrolló el método *#transferNextValuesFromHash*, el cual transfiere los elementos de forma recursiva a partir del índice $s + 1$.

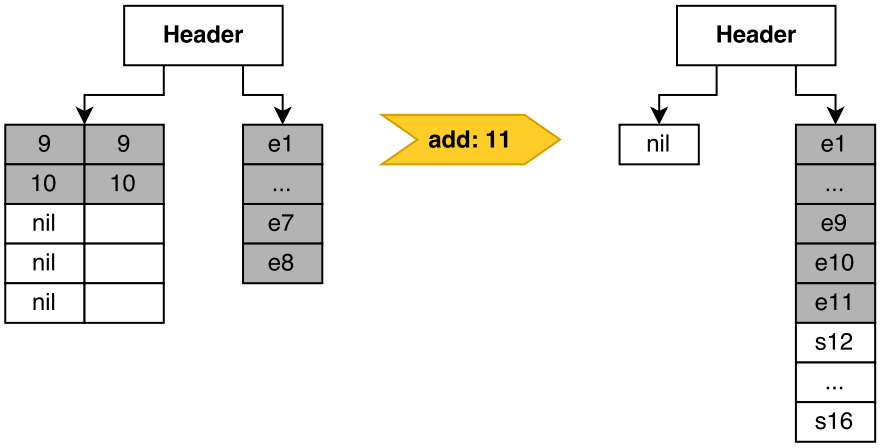

Figura 3.1: Transferencia de valores asociados a índices enteros desde la parte de hash a la de arreglo

Esta situación se ilustra en el siguiente ejemplo, que está representado gráficamente en la figura3.1. Supongamos que a es una instancia de *SmartCollection* que contiene 7 elementos en su parte ordenada, la cual es de capacidad 8, y estos elementos ocupan los 7 primeros espacios. Si luego agregamos dos elementos mediante a at: 9 put: 9; a at: 10 put: 10, se inicializará la parte de hash y se agregarán el 9 en el índice 9, y 10 en el índice 10 de esta, pues ambos índices están fuera del rango $[1, s]$ de a. Si añadimos otro elemento mediante a add: 8, este quedará en la parte ordenada, la cual ahora estará completamente ocupada. Al añadir un décimo elemento mediante a add: 11, este debería quedar en la posición 11 porque los índices 9 y 10 ya están ocupados, lo que significa que esos elementos en los índices 9 y 10, que originalmente se insertaron en la parte de hash, ahora deben reubicarse a la parte de arreglo. El método *#transferNextValuesFromHash* se encargará de la reubicación, el arreglo interno tendrá que crecer para reacomodar los elementos, y finalmente obtendremos una parte ordenada con los espacios del 1 al 11 ocupados, y la parte de hash vacía dejará de ser referenciada para liberar ese espacio.

3.4.2. Parte secuencial

Sea c una instancia de SmartCollection con s elementos en su parte secuencial, con un arreglo interno a de capacidad n cuyo i-ésimo espacio se denota por a_{i}. La instancia tiene asociados un par de punteros p_{ini} y p_{fin}. Al momento de creación, c es inicializada con los valores $p_{ini} := 1$, $p_{fin} := 0$ y $a := nil$. La cantidad s de elementos en la parte secuencial viene determinada por $p_{fin} - p_{ini} + 1$. Los métodos de adición de un elemento al final de la colección, adición de un elemento al principio de la colección, y remoción de un elemento (*#add:*, *#addFirst:* y *#remove:*, respectivamente), son explicados a continuación.

#add: e

1. Si la parte de hash ha sido creada y contiene un índice entero h_{i} tal que $s + 1 = h_{\text{i}}$:
   - Se llama a *#transferNextValuesFromHash*
2. Si $a \neq nil$ y $p_{fin} + 1 \leq n$:
   - $p_{fin} := p_{fin} + 1$
   - $a_{p_{fin}} :=$ e
3. Si no:
   - Se llama a *#makeRoomAtLast*
   - Se vuelve a 1

El algoritmo *makeRoomAtLast* funciona del siguiente modo:

- Si $a = nil$ (no hay arreglo interno aún):
  - Se inicializa a con un arreglo de capacidad n
  - Retorna
- Si $s * 2 > p_{fin}$:
  - Se crea un nuevo arreglo a' de capacidad $2s$
  - Se copian todos los espacios a_{i} en a'_{i}
  - $a := a'$
- Si no:
  - Se crea $p'_{fin} := \lfloor \frac{p_{fin}}{2} \rfloor$
  - Se crea $p'_{ini} := p'_{fin} - p_{fin} + p_{ini}$
  - Se copian en a todos los elementos de los espacios en el intervalo $[p_{ini}, p_{fin}]$ al intervalo $[p'_{ini}, p'_{fin}]$
  - Se asignan $p_{fin} := p'_{fin}$ y $p_{ini} := p'_{ini}$

#addFirst: e

1. Si la parte de hash ha sido creada y contiene un índice entero h_i tal que $s + 1 = h_i$:
   - Se llama a *#transferNextValuesFromHash*
2. Si $a \neq nil$ y $p_{ini} > 1$:
   - Se asigna $p_{ini} := p_{ini} - 1$
   - Se asigna $a_{p_{ini}} :=$ e
3. Si no:
   - Se llama a *#makeRoomAtFirst*
   - Se vuelve a 1

El algoritmo *makeRoomAtFirst* funciona del siguiente modo:

- Si $a = nil$ (no hay arreglo interno aún):
  - Se inicializa a con un arreglo de capacidad n
  - Retorna
- Si s * 2 > *capacidad(a)*:
  - Se crea un nuevo arreglo a' de capacidad $2s$
  - Se crea $p'_{ini} := 2s - capacidad(a) + p_{ini}$
  - Se crea $p'_{fin} := p'_{ini} + p_{fin} - p_{ini}$
  - Se copian todos los espacios de a en el intervalo $[p_{ini}, p_{fin}]$ a a' en el intervalo $[p'_{ini}, p'_{fin}]$
  - Se asignan $a := a'$, $p_{ini} := p'_{ini}$ y $p_{fin} := p'_{fin}$
- Si no:
  - Se crea $p'_{ini} := \lfloor \frac{capacidad(a)}{2} \rfloor + 1$
  - Se crea $p'_{fin} := p'_{ini} - p_{ini} + p_{fin}$
  - Se copian en a todos los elementos de los espacios en el intervalo $[p_{ini}, p_{fin}]$ al intervalo $[p'_{ini}, p'_{fin}]$
  - Se asignan $p_{fin} := p'_{fin}$ y $p_{ini} := p'_{ini}$

#remove: e

El método *#remove:* llama al método *#remove:ifAbsent:* que permite proporcionar un bloque[2] que no recibe argumentos para ejecutarse en caso de que el elemento e a eliminar no se encuentre en la colección. El bloque que *#remove:* le pasa a *#remove:ifAbsent:* al ejecutarse lanza un error indicando que el elemento no fue encontrado. *#remove:ifAbsent:* sólo opera sobre la parte de arreglo (ie. no busca elementos en la parte de hash), ya que en

[2]Función anónima

la API de colecciones de Pharo este método no existe para los diccionarios pues para estos debe usarse *#removeKey:*. Este último también permite eliminar un elemento asociado a una llave de la parte de arreglo, o sea, cuyo índice es un entero entre 1 y s.

A continuación se detalla el funcionamiento de *#remove:ifAbsent:*.

#remove: e ifAbsent: absentBlock

- Para cada i $\in [p_{ini}, p_{fin}]$:
  - Si $a_i = e$:
    - Para todo $j \in [i, p_{fin} - 1]$:
      - $a_j := a_{j+1}$
    - $a_{p_{fin}} := nil$
    - $p_{fin} := p_{fin} - 1$
    - Retorna e
- Se ejecuta *absentBlock*

Consecuencias

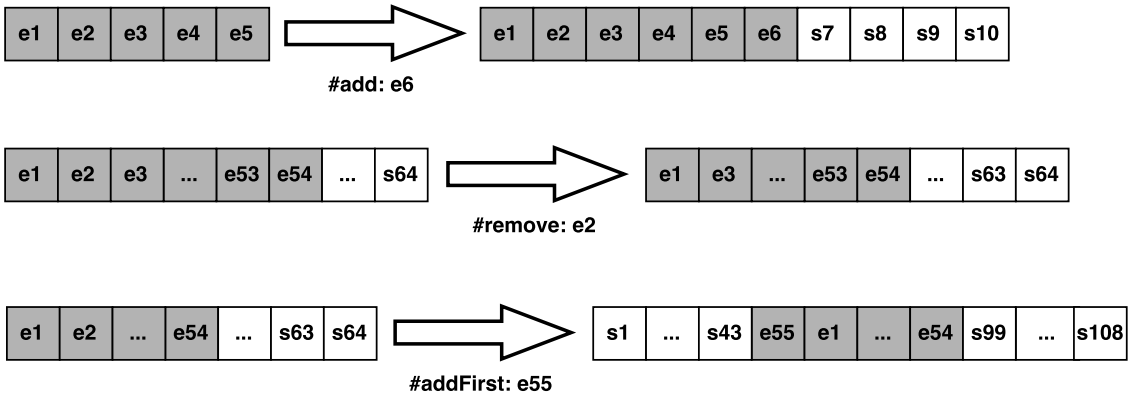

Figura 3.2: Operaciones #add:, #remove: y #addFirst: sobre el arreglo interno de *SmartCollection*.

El funcionamiento de *#add:*, *#addFirst:* y *#remove:* puede observarse en la figura 3.2. Este comportamiento para la parte de arreglo de *SmartCollection*, que es análogo al de *OrderedCollection*, implica que las operaciones de adición al principio y al final de la colección serán en tiempo amortizado $O(1)$, mientras que la remoción de un elemento arbitrario será en tiempo $O(s)$. Esto supone una ventaja con respecto a SLua, la cual se medirá experimentalmente más adelante.

3.4.3. Micro-benchmarks

Se realizaron micro-benchmarks para comparar SmartCollection con OrderedCollection en los métodos *#addFirst:*, *#addLast:*, *#do:*, *#reduce:* y *#remove:*. Los resultados de los micro-benchmarks se ven en la tabla 3.2. Las columnas segunda y tercera indican la media de los tiempos (en milisegundos) obtenidos en las i iteraciones de cada benchmark, para OrderedCollection y SmartCollection respectivamente. La cuarta columna muestra la variación porcentual del tiempo de ejecución entre los resultados de OrderedCollection y SmartCollection para cada benchmark. Todos los resultados son presentados en forma de intervalos de confianza al 90 %, calculados según 1.2 y 1.11.

Benchmark	**OrderedCollection**	**SmartCollection**	$\Delta\bar{x}$
addFirst:	30.56 ± 0.11	32.38 ± 0.17	5.9 ± 0.7
addLast:	34.5 ± 0.13	41.42 ± 0.14	20 ± 0.6
do:	9.2 ± 0.084	9.34 ± 0.088	1.5 ± 1.3
reduce:	199.35 ± 0.11	237.76 ± 0.1	19.3 ± 0.1
remove:	130.9 ± 0.26	105.86 ± 0.25	-19.2 ± 0.2

Tabla 3.2: Micro-benchmarks OrderedCollection vs SmartCollection, i = 1000

El rendimiento de los métodos de SmartCollection sigue siendo menor que el de los mismos en OrderedCollection, pero hay mejoras con respecto a SLua. Fundamentalmente, SmartCollection posee un *#addFirst:* cuyo tiempo amortizado es de complejidad $O(1)$ al igual que la implementación de OrderedCollection. La implementación de SmartCollection es levemente más lenta ($\approx 6\,\%$) debido a que hay algunas comparaciones adicionales que deben hacerse, con respecto a la transferencia de valores asociados a índices enteros desde la parte de hash mencionada en la sección 3.4.1. *#addLast:* mejor bastante con respecto a SLua (de $\approx 76\,\%$ a $\approx 20\,\%$ de overhead) pero aún es bastante más lento que el caso de OrderedCollection. No tenemos claro por qué *#addFirst:* posee un rendimiento más cercano al de OrderedCollection que *#addLast:*, siendo que sus implementaciones son muy similares. Los casos de *#do:* y *#reduce:* se mantienen prácticamente iguales con respecto a SLua, por los mismo motivos mencionados para esta. *#remove:* resulta ser bastante más rápido en SmartCollection que en OrderedCollection ($\approx 20\,\%$), sin embargo aún no estamos seguros de las razones de esta diferencia.

También se realizaron micro-benchmarks para comparar SmartCollection con Dictionary en los métodos *#at:ifAbsent:* y *#at:put:*. Los resultados de los benchmarks se ven en la tabla 3.3.

Benchmark	**Dictionary**	**SmartCollection**	$\Delta\bar{x}$
at:ifAbsent:	21.83 ± 0.27	22.47 ± 0.24	2.9 ± 1.7
at:put:	30.92 ± 0.14	32.63 ± 0.28	5.5 ± 1

Tabla 3.3: Micro-benchmarks Dictionary vs SmartCollection, i = 1000

Los benchmarks para *#at:ifAbsent:* y *#at:put:* muestran un overhead por parte de SmartCollection, de aproximadamente 3 % y 5 %, respectivamente. Esto muy probablemente se

debe a la mayor indirección de estos métodos en SmartCollection, ya que el índice al que se quiere acceder puede estar en la parte de arreglo o de hash.

3.5. SmartCollection2

SmartCollection logró mejorar el tiempo de adición de elementos al inicio de la colección con respecto a *SLua*, pasando de un tiempo $O(s)$ para cada adición, a tiempo amortizado $O(1)$. Sin embargo, la operación de remoción de un elemento, *#remove:*, toma tiempo promedio $O(s)$, de igual modo que las operaciones *#add:before:* y *#add:after:* que añaden un elemento antes y después, respectivamente, de un elemento dado. Esto ocurre debido a que deben moverse todos los elementos de índice mayor al que se va a quitar o agregar. *SmartCollection2* surge como un intento por mejorar los tiempos de estas operaciones, cuyo tiempo depende linealmente de la cantidad s de espacios utilizados en el arreglo de la parte ordenada de *SmartCollection*.

3.5.1. Estructura interna

Buscando mejorar la complejidad de tiempo de las operaciones que en *SmartCollection* dependen linealmente de s, se optó por usar un conjunto de arreglos para la parte secuencial. Con esto, se reduce la cantidad de elementos que es necesario copiar tras cada expansión o contracción de la colección. Además, el asociar una gran porción continua de memoria a una colección puede llevar a un rendimiento impredecible debido a pausas del recolector de basura [34], por lo que la división del arreglo interno en varios *arreglos discontinuos* permite darle un mayor determinismo al rendimiento, lo cual es particularmente adecuado para sistemas en tiempo real y embebidos [6, 35, 36, 37].

SmartCollection2 en un principio comienza con un solo arreglo y si se agota la capacidad de este para añadir elementos en alguno de los dos sentidos (hacia el inicio o hacia el final), se agrega un nuevo arreglo hacia ese lado y en este se añade el nuevo elemento. Luego, si se siguen añadiendo elementos y se agota la capacidad de este nuevo arreglo, se agrega otro (ver figura 3.3, operaciones #add: y #addFirst:). El tamaño del arreglo que se agregue dependerá de cuantas veces ha sido necesario hacer crecer la colección en esa dirección. Con esto ahora sólo es necesario mover elementos dentro de un sub-arreglo al realizar una operación de adición/remoción de elementos, en contraste con lo que ocurría en el caso de *SmartCollection* (ver figura 3.2, operación #remove:). Sin embargo, este enfoque también aumenta el tiempo de búsqueda de un elemento dentro de la colección al tener que pasar de un arreglo a otro para buscar, y además impacta a las operaciones #at: y #at:put: cuando trabajan sobre la parte ordenada pues dejan de ejecutarse en tiempo constante y pasan a depender linealmente de la cantidad de sub-arreglos anteriores al índice que se busca. Por lo tanto, es necesario evaluar cuidadosa y cuantitativamente los beneficios y perjuicios de esta diferencia en la implementación, lo cual se detallará más adelante.

Desde el punto de vista del usuario, *SmartCollection2* se comporta de forma idéntica a

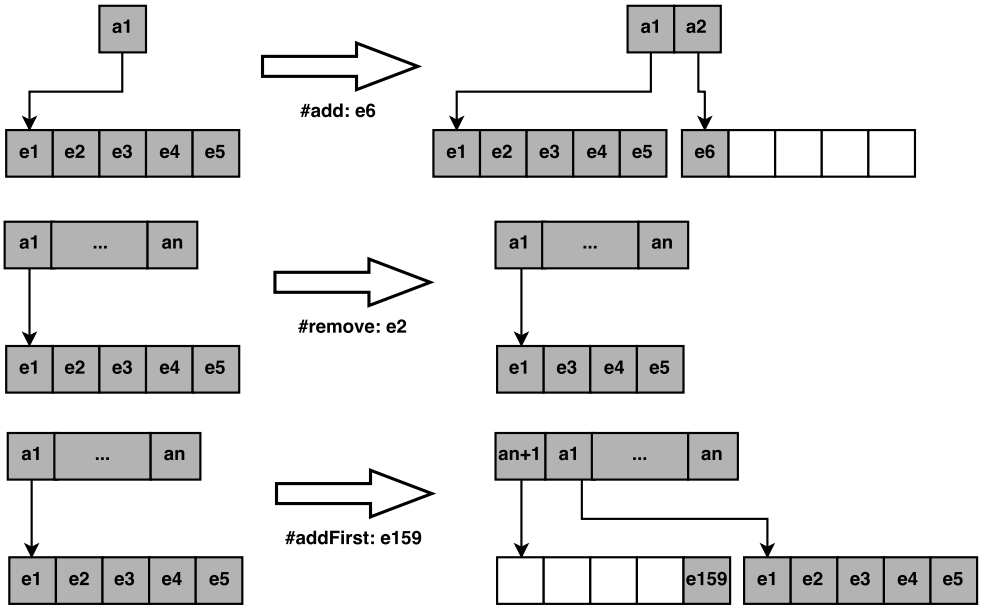

Figura 3.3: Operaciones #add:, #remove: y #addFirst: sobre la parte ordenada de *SmartCollection2*.

SmartCollection y *SLua*, es polimórfica a estas, y exhibe el mismo comportamiento híbrido.

3.5.2. Micro-benchmarks

Se realizaron micro-benchmarks para comparar SmartCollection2 con OrderedCollection en los métodos *#addFirst:*, *#addLast:*, *#do:*, *#reduce:* y *#remove:*. Los resultados de los benchmarks se ven en la tabla 3.4. Las columnas segunda y tercera indican la media de los tiempos (en milisegundos) obtenidos en las i iteraciones de cada benchmark, para OrderedCollection y SmartCollection2 respectivamente. La cuarta columna muestra la variación porcentual entre los resultados de OrderedCollection y SmartCollection2 para cada benchmark. Todos los resultados son presentados en forma de intervalos de confianza al 90 %, calculados según 1.2 y 1.11.

Benchmark	**OrderedCollection**	**SmartCollection2**	$\Delta\bar{x}$
addFirst:	30.51 ± 0.12 ms	26.93 ± 0.14 ms	-11.8 ± 0.6
addLast:	36.35 ± 0.15 ms	30.72 ± 0.13 ms	-15.5 ± 0.5
do:	9.19 ± 0.05 ms	9.16 ± 0.05 ms	-0.33 ± 1.82
reduce:	198.15 ± 0.45 ms	237.7 ± 3.5 ms	20 ± 2
remove:	124.68 ± 0.78 ms	101.04 ± 0.33 ms	-19 ± 0.6

Tabla 3.4: Micro-benchmarks SmartCollection2 vs OrderedCollection, i = 1000

El rendimiento de los métodos de SmartCollection2 mejora notablemente con respecto a SmartCollection, e incluso supera en varios casos al de los de OrderedCollection. *#addFirst:* y *#addLast:* logran superar en aproximadamente un 11 %*yun*15 %, respectivamente, a sus

contrapartes de OrderedCollection, lo cual indica que el evitar la copia de la totalidad del contenido del arreglo interno al momento de crecer tiene un impacto significativo al menos en el caso de colecciones con muchos elementos como las que prueban los micro-benchmarks. El método *#remove:* también mejora el rendimiento con respecto al de OrderedCollection, en aproximadamente un 19 %, debido a que la operación de remoción sólo requiere trabajar con un subarreglo y no con toda la parte ordenada, lo cual se muestra efectivo en el caso de arreglos grandes. Las operaciones *#do:* y *#reduce:* mantienen un rendimiento muy parecido al de SmartCollection. Esto sorprende ya que se esperaría cierta degradación de rendimiento debido a que ahora para recorrer la parte secuencial de la colección se debe "saltar" de arreglo en arreglo, lo cual reduce la localidad de las referencias.

También se realizaron micro-benchmarks para comparar SmartCollection2 con Dictionary en los métodos *#at:ifAbsent:* y *#at:put:*. Los resultados de los benchmarks se ven en la tabla 3.5.

Benchmark	**Dictionary**	**SmartCollection2**	$\mathbf{\Delta\bar{x}}$
at:ifAbsent:	20.89 ± 0.22 ms	21.71 ± 0.24 ms	4 ± 1.6
at:put:	35.7 ± 3.8 ms	37.7 ± 4 ms	6 ± 1.6

Tabla 3.5: Micro-benchmarks SmartCollection2 vs Dictionary, i = 1000

Los benchmarks para *#at:ifAbsent:* y *#at:put:* muestran un overhead por parte de SmartCollection2 bastante similar al que presentaba SmartCollection. Esto se debe a que la implementación con respecto a estos métodos no cambió, y por ende el argumento de indirección enunciado para SmartCollection sigue siendo válido para SmartCollection2.

3.6. Tests

Las tres colecciones anteriormente explicadas, SLua, SmartCollection, y SmartCollection2, aprovechan los *traits* de testing que provee la biblioteca de colecciones de Pharo. Estos *traits* permiten reutilizar tests que son comunes a distintas colecciones [3]. En particular, se reutilizaron los *traits*: *TIndexAccess*, *TStructuralEqualityTest*, *TSequencedStructuralEqualityTest*, *TAddTest*, y *TCopySequenceableSameContents*. Además se añadieron otros tests necesarios que no estaban cubiertos por los traits antes mencionados. Con esto se logró una cobertura de tests del 91.67 % para SLua, de 94.23 % para SmartCollection, y de 96.15 % para SmartCollection2, según indica la herramienta *Hapao* [38].

3.7. Macro-benchmarks

Se realizaron una serie de macro-benchmarks para medir el rendimiento de las colecciones desarrolladas en relación a las colecciones expansibles más comunes de Pharo (*OrderedCollection* y *Dictionary*) en ejecuciones de aplicaciones representativas. Se comparó el rendimiento en dos aspectos complementarios: tiempo de ejecución y consumo de memoria.

Para llevar a cabo las mediciones se definió un conjunto de benchmarks que hacen uso intensivo de colecciones. El conjunto consiste en cinco benchmarks, cada uno de los cuales produce y manipula varios miles de colecciones de datos. Llevar a cabo benchmarks significativos es conocido por ser difícil [39]. Por este motivo, se han tomado las siguientes medidas: (i) limpiar la memoria gatillando el recolector de basura previo a cada ejecución, (ii) *calentar* el compilador *just-in-time* (JIT) y los cachés con algunas ejecuciones iniciales[3], (iii) ejecutar múltiples veces cada benchmark.

Las cinco aplicaciones seleccionadas han sido diseñadas usando las colecciones estandar de Pharo, y no las colecciones desarrolladas en este trabajo. Cada benchmark — con sus n iteraciones — es realizado varias veces, una primera vez usando las colecciones estandar de Pharo y otra por cada colección desarrollada a comparar con estas. Debido a que las colecciones desarrolladas en este trabajo fueron diseñadas para ser polimórficas a *OrderedCollection* y *Dictionary*, es posible reemplazar todas las referencias a esas dos clases por cualquiera de las clases desarrolladas. La consecuencia es la producción de instancias de las colecciones desarrolladas en vez de las colecciones normales de Pharo por parte de la aplicación cuando el benchmark es ejecutado.

El entorno de ejecución de Pharo no ha sido modificado: esto significa que si una aplicación invoca una API ofrecida por la biblioteca estandar de Pharo (ej. socket), entonces la API producirá una colección estandar de Pharo. Este comportamiento es crítico para los experimentos ya que no se quiere medir el efecto de las colecciones desarrolladas en las bibliotecas estandar de Pharo. El reemplazo de clases ha sido realizado mediante un *profiler* creado con el framework Spy[4].

Las colecciones desarrolladas utilizan inicialización perezosa para sus colecciones internas, y es conocido que este mecanismo reduce el sobrecosto al instanciar *OrderedCollection* y *Dictionary* [40]. Por esto, se opta por añadir a la comparación versiones *lazy* de las colecciones tradicionales, con la finalidad de medir la contribución de esta optimización en particular, y de poder descontar su efecto al analizar el rendimiento de las colecciones desarrolladas.

3.7.1. Consumo de memoria

Para medir el consumo de memoria de los macro-benchmarks se desarrolló un *profiler* dedicado[5] que monitorea las instancias de colecciones producidas en los paquetes de los benchmarks, y además realiza reemplazo de colecciones antes de ejecutar cada uno para cambiar *OrderedCollection* y *Dictionary* por la colección a estudiar. Luego de la ejecución, el *profiler* entrega información con respecto a la memoria consumida por estas colecciones.

Se midió el consumo de memoria para cuatro de los cinco benchmarks definidos, ya que el caso de *Spectrograph* falló por falta de memoria de la máquina virtual de Pharo debido a que el

[3]Esto se debe a que la máquina virtual Cog, que usa Pharo, ingresa al método al *inline cache* la primera vez que es ejecutado; la segunda vez se encuentra en el *inline cache*, lo cual gatilla que el compilador JIT produzca código; la tercera ejecución y las que siguen ya se consideran en *steady state* [16].

[4]Ver 1.4.1

[5]Ver 4.6.

monitoreo es costoso en cuanto a este recurso. En cada caso, se computó la totalidad de bytes asignados a las colecciones a estudiar durante la ejecución, en los escenarios con colecciones normales, colecciones perezosas y las colecciones desarrolladas. Los resultados se muestran en la tabla 3.6. La primera columna lista los benchmarks. La segunda columna muestra la memoria total (en KB) asignada a las colecciones, al usar la versión normal de las mismas (NC). La tercera columna muestra la variación porcentual en el consumo de memoria, con respecto a la segunda columna, al usar las versiones perezosas (LC) de OrderedCollection y Dictionary. La cuarta, quinta y sexta columna muestran la variación porcentual en el consumo de memoria al usar SLua, SmartCollection (SC) y SmartCollection2 (SC2), respectivamente, en vez de OrderedCollection y Dictionary.

Benchmark	**NC (KB)**	**LC (%)**	**SLua (%)**	**SC (%)**	**SC2 (%)**
CircularTreeMap	1994.5	-0.2	-1.1	-2.0	6.3
ForceBasedLayout	92.7	-0.1	-32.8	-20.2	39.5
Mondrian	5876.48	-3.0	-26.2	-16.8	26.7
NameCloud	3218.4	-0.01	-31.2	-18.8	43.6

Tabla 3.6: Memoria total asignada a colecciones al usar las normales (NC), las perezosas (LC), SLua, SmartCollection (SC) y SmartCollection2 (SC2)

La tabla 3.6 muestra que mientras las colecciones perezosas de Pharo (LC) consumen marginalmente menos memoria que NC, SLua consume significativamente menos en tres de los benchmarks. SC también consume bastante menos memoria de NC, aunque en menor magnitud que SLua. SC2 consume significativamente más memoria que el resto de las colecciones.

3.7.2. Tiempo de ejecución

Para realizar los benchmarks es importante contar con una infraestructura adecuada que los soporte. Inicialmente se construyó una solución en torno a *SMark* que consistía en realizar una instrumentación con un profiler liviano, que sólo hace el reemplazo de clases, antes de correr cada benchmark (usando los métodos *#setUp* y *#tearDown* descritos en 1.3.2). Sin embargo, esta solución resultó ser insuficiente debido a que era necesario instrumentar antes de cada ejecución del benchmark y desinstrumentar después de las mismas, por lo cual el proceso se hacía muy lento.

Como alternativa, se desarrolló una infraestructura[6] que permite instrumentar una sola vez por cada benchmark, luego correr todas las ejecuciones necesarias del mismo y tomar sus tiempos, y luego desinstrumentar. Esto hace que el proceso sea más rápido y los resultados sean más precisos.

[6] Ver 4.4

Resultados

A continuación se presenta una tabla comparativa de los tiempos de ejecución obtenidos para los cinco benchmarks definidos anteriormente.

Benchmark	**NC (s)**	**LC (%)**	**SLua (%)**	**SC (%)**	**SC2 (%)**
CircularTreeMap	58.26 ± 0.92	-8.3 ± 2.7	-8.7 ± 0.6	-15.7 ± 2.2	-15.1 ± 2.2
ForceBasedLayout	73.02 ± 0.37	0.9 ± 1.1	2.1 ± 0.5	0.9 ± 1.6	-2.2 ± 0.9
Mondrian	29.2 ± 0.03	-7.1 ± 0.3	-5.6 ± 0.2	-11.8 ± 0.2	-1.5 ± 0.3
NameCloud	2.31 ± 0.02	-0.2 ± 0.7	1.1 ± 0.9	-1.5 ± 0.5	-1.3 ± 0.6
Spectrograph	15.81 ± 0.02	2.2 ± 0.3	1.3 ± 0.2	2.5 ± 0.4	0.5 ± 0.4

Tabla 3.7: Comparación de tiempo de ejecución entre colecciones normales (NC), colecciones perezosas (LC), SLua, SmartCollection (SC) y SmartCollection2 (SC2)

La tabla 3.7 muestra los resultados de las ejecuciones de los benchmarks con respecto a tiempo de ejecución. Cada benchmark es ejecutado 35 veces, y luego los *outliers* son filtrados mediante el criterio de Peirce[7]. La primera columna lista los macro-benchmarks. La segunda columna entrega una estimación del tiempo de ejecución, en segundos, al usar las colecciones estandar (NC). Las columnas tercera, cuarta, quinta y sexta muestran estimaciones de la variación porcentual en el tiempo de ejecución obtenida al optar por las colecciones perezosas (LC), SLua, SmartCollection (SC), y SmartCollection (SC2) en vez de las colecciones normales. Una variación negativa indica una reducción en el tiempo de ejecución del benchmark mientras que una variación positiva indica mayor tiempo de ejecución. Todos los resultados expuestos en la tabla corresponden a intervalos de confianza al 90 %, calculados según la expresión 1.2 en el caso de los tiempos (segunda columna) y 1.11 para las variaciones porcentuales.

De la tabla 3.7 se observa lo siguiente:

1. En el benchmark *CircularTreeMap* se observa que LC, SLua, SC y SC2 funcionan más rápido que NC. En el caso de LC y SLua la reducción en tiempo de ejecución es de aproximadamente un 8 %, mientras que SC y SC2 reducen el tiempo de ejecución en aproximadamente un 15 %. El *speedup* de 8 % en el caso de LC indica que la inicialización perezosa de los arreglos internos en las colecciones expansibles puede contribuir a reducir el tiempo de ejecución por sí sola. El que SLua, SC y SC2 cuenten con inicialización perezosa sugiere que una buena parte del *speedup* registrado puede deberse a eso. Sin embargo, al menos en los casos de SC y SC2 esto no sería el único factor de importancia.

2. En el caso de *ForceBasedLayout*, LC y SC no muestran diferencias estadísticamente significativas con respecto a NC. SLua presenta cierto *overhead* con respecto a NC y SC2 muestra un *speedup* con respecto a este. En ambos casos esta diferencia parece ser cercana al 2 %, pero los márgenes de error relativamente altos no dejan muy clara la magnitud de esta diferencia.

3. En el benchmark *Mondrian*, LC, SLua, SC y SC2 funcionan más rápido que NC. SC

[7] Ver 1.3.1.

obtiene el mejor rendimiento, con un *speedup* de más de un 11 %, mientras que SC2 obtiene el *speedup* más pequeño, cercano al 1 %. Las colecciones perezosas logran una mejora de 7 %. Este caso es interesante pues contrasta con *CircularTreeMap*. La inicialización perezosa parece contribuir a reducir el tiempo de ejecución, pero este efecto se ve reducido en SLua y sobre todo en SC2, mientras que en SC se ve potenciado.

4. En el caso de *NameCloud*, LC y SLua no presentan diferencias estadísticamente significativas. SC y SC2 muestran cierto *speedup* de poca magnitud.

5. En *Spectrograph* se observa una pequeña reducción en el tiempo de ejecución para LC, SLua y SC con respecto a NC. SC2 no muestra una diferencia estadísticamente significativa con respecto a NC.

Análisis de uso de colecciones en los benchmarks

Para estudiar la causa de estas diferencias se usó *profiling*, en particular el profiler de uso de colecciones descrito en la sección 4.5 y MessageTally, para analizar el uso de colecciones por parte de los benchmarks. A continuación se exponen resultados de este análisis:

1. Con respecto a *CircularTreeMap* se identificó que se crean 7160 instancias de Dictionary y 5300 instancias de OrderedCollection. Los métodos más llamados son de adición (*#addLast:*) en el caso de OrderedCollection, y de acceso (*#at:*) en el caso de Dictionary. También se realizan algunas operaciones de remoción de elementos, pero en menor medida. 120 instancias de OrderedCollection permanecen vacías, 80 instancias alcanzan un tamaño de exactamente 357 elementos, y el resto de las instancias (5010) alcanzan, en promedio, un tamaño de 6 elementos cada una. Según análisis mediante *MessageTally*[8], el benchmark invierte la mayor parte de su tiempo ($\approx$ 61,5 %) en cálculos para la disposición de los elementos (*layout*), los cuales modifican propiedades de los elementos referenciados por las colecciones, pero no a las colecciones en sí.

2. *ForceBasedLayout* produce 6 instancias de Dictionary y 2642 de OrderedCollection. En cuanto a las operaciones, gran parte de estas ocurren sobre dos instancias de Dictionary (más de 1 millón de *#at:* y más de 700000 *#at:put:*, entre otras). Las instancias de OrderedCollection en general no superan los 200 elementos y su llenado viene dado por la operación *#addLast:*, algunas de las operaciones más comunes son de iteración (*#do:*), y de acceso (en una instancia en particular se realizan 12015 operaciones *#at:*). El análisis con MessageTally muestra que alrededor del 98 % del tiempo invertido tiene que ver con operaciones relacionadas al *layout* de los elementos almacenados en las colecciones.

3. En *Mondrian*, la cantidad de colecciones producidas es bastante mayor: 147 IdentityDictionary, 30 SortedCollection, 4409 Dictionary y 71171 OrderedCollection. También se realizan operaciones más diversas, incluyendo un gran número de adiciones —la mayor parte de estas al final de la colección, pero también una pequeña proporción al medio (en 20 instancias que alcanzan un promedio de 5380 elementos se realizan, en

[8] Ver 1.4.2

promedio, 5380 operaciones *#insert:before:*) y al inicio (30 instancias de OrderedCollection que realizan, en promedio, 120 operaciones *#addFirst:* cada una)— una gran cantidad de búsquedas dentro de la colección —8851 instancias de OrderedCollection de tamaño pequeño (menos de 200 elementos) que realizan en promedio 4.5 operaciones *#indexOf:* cada una, y 30 instancias de mayor tamaño (1936 elementos en promedio) que realizan alrededor de 1498 operaciones *#indexOf:* cada una— una cantidad considerable de chequeos de inclusión —8789 instancias de OrderedCollection que realizan la operación *#inclues:* un promedio de 4 veces cada una, y 30 instancias de tamaño mayor a 1500 elementos que realizan la operación un promedio de 1498 veces cada una— y un número no despreciable de remociones —destacan 10 instancias de OrderedCollection de tamaño superior a 5000 que realizan en promedio 4365 operaciones *#remove:* cada una—. Con respecto a los tamaños, hay una gran cantidad de colecciones pequeñas: 4409 instancias de Dictionary (o sea, la totalidad de estas) cuyo tamaño es menor o igual a 5, y 70620 instancias cuyo tamaño no sobrepasa los 10 elementos.

4. En *NameCloud* se crean 451 instancias de la clase Set, 6 de SortedCollection, 2 de Dictionary y 52094 de OrderedCollection. De las instancias de OrderedCollection, 51941 poseen 4 o menos elementos (5 se mantienen vacías) y el resto tiene en promedio 100 elementos. Las instancias se llenan mediante *#addLast:* y sobre ellas se realiza la operacion *#do:* en promedio 1 vez por cada una. De las dos instancias de Dictionary, una se mantiene vacía y la otra acomoda 316 elementos. Sobre esa instancia se realizan operaciones de acceso y reemplazo de elementos (64080 *#at:ifAbsent:* y 32040 *#at:put:*) y también operaciones de enumeración (450 *#valuesDo:*, método que genera una instancia de OrderedCollection a la cual le añaden todos los valores del diccionario, y luego se realiza *#do:* sobre esa instancia. Las instancias generadas dentro de *#valuesDo:* también son capturadas por el profiler, por lo que se consideran dentro de las mencionadas anteriormente). El análisis de MessageTally indica que cerca del 28 % del tiempo es invertido en cálculos para el *layout*. Además aproximadamente un 20 % del tiempo se ocupa en el método *CompiledMethod»#sourceCode* —ya que la *nube de palabras* que se construye en ese benchmark toma texto a partir del código fuente de miles de métodos. Este método puede ser una fuente importante de no-determinismo ya que si el código fuente de un método no está en memoria deberá ser leído del disco.

5. No se pudo realizar el análisis detallado de colecciones con *Spectrograph* debido a que la máquina virtual de Pharo llegaba a su límite de memoria, por lo que se sospecha que se genera un número bastante grande de colecciones (el profiler almacena información para cada instancia generada, y luego las agrupa y genera estadísticas según sitio de producción). Sin embargo, el análisis con MessageTally indica que cerca de un 50 % de la ejecución transcurre sobre objetos que no fueron instrumentados, y cerca de un 40 % corresponde al *layout*. Un 4 % del tiempo es invertido en la operación *#do:*, y un 3.6 % en la operación *#reduce:*, ambas de OrderedCollection.

Discusión

Los resultados obtenidos sugieren que:

CircularTreeMap

- El *speedup* obtenido por LC con respecto a NC se debe puramente a la inicialización perezosa de las estructuras internas, pues esta es la única diferencia en implementación entre ambas colecciones. Sin embargo, el número de instancias que permanecen vacías es pequeño (lo cual se refleja en un ahorro de memoria de sólo un 0.2 %). Esto sugiere que el hecho de dilatar la instanciación puede producir una mejora en el rendimiento sin que esto se correlacione con un ahorro de memoria, lo cual tiene que deberse a algún aspecto del funcionamiento de la máquina virtual que hasta ahora se desconoce.
- El *speedup* de SLua con respecto a NC se debe en parte a la inicialización perezosa y al pequeño ahorro de memoria, al igual que SC. El caso de SC2 es diferente porque esta consume más memoria que NC. En su caso, la reducción en tiempo de ejecución probablemente se debe a su estructura interna, que le ahorra operaciones de crecimiento de arreglos internos y copias de elementos dentro de estos.

ForceBasedLayout

- El hecho de que los cálculos asociados al *layout* de los elementos ocupen el 98 % del tiempo explica que no hayan diferencias estadísticamente significativas entre las alternativas.

Mondrian

- SLua tiene menor rendimiento con respecto a LC posiblemente debido a las operaciones *#addFirst:*.
- El buen rendimiento de SC con respecto a NC se explica posiblemente a través de la inicialización perezosa y su mayor ahorro de memoria con respecto a LC debido a la menor cantidad de variables de instancia que poseen sus instancias.
- SC2 presenta un rendimiento bastante reducido en relación a SC posiblemente debido a las operaciones *#insert:before:* y *#indexOf:*, ya que al poseer múltiples arreglos internos se rompe la localidad de las referencias, lo cual puede ocasionar más fallos de caché. Además, SC2 presenta el mayor consumo de memoria.

Capítulo 4

Selección adaptativa de colecciones

En esta parte se aborda el problema de la selección correcta de colecciones y sus parámetros iniciales dentro aplicaciones. La forma de enfrentar el problema en este trabajo consiste en monitorear los patrones de uso de colecciones durante la ejecución de la aplicación para encontrar situaciones en que la elección de una colección o sus parámetros podría ser mejorada mediante una modificación sencilla (o bien un reemplazo de la colección en sí, o una modificación de sus parámetros). Luego de identificar estas situaciones es posible realizar una optimización automática del programa analizado.

La forma de enfrentar este problema en el presente trabajo sigue la línea del experimento *Chameleon* [7], que propone una técnica de *pattern matching*. Chameleon hace uso de una máquina virtual de Java modificada, por lo que no queda claro hasta qué punto sus resultados son replicables. Dadas las cualidades reflexivas de Pharo, el esfuerzo que se requiere para llevar a cabo el experimento en este lenguaje es mucho menor al requerido para aplicarlo en Java. Por eso, en primer lugar se busca replicar los resultados, pero con la diferencia de que la herramienta creada para este propósito no sólo entregará recomendaciones para los reemplazos de colecciones, sino que aplicará estas sugerencias de forma automática. Además, la modificación será a nivel del código fuente —el cual será recompilado— por lo que no se necesita dejar instrumentado el código, con lo que el sesgo de medición se reduce, junto al tiempo requerido para ejecutar los benchmarks.

Finalmente, se hará uso de esta herramienta para evaluar las colecciones desarrolladas en el capítulo 3.

4.1. *Pattern matching* sobre el uso de colecciones

El proceso de selección adaptativa de colecciones se divide en tres etapas: monitoreo de los sitios de producción de colecciones, aplicación de reglas de reemplazo (pattern matching), y reemplazo automático de colecciones. A continuación se detalla cada una de estas etapas.

4.1.1. Monitoreo de los sitios de producción de colecciones

Esta etapa consiste en asociar meta-información a cada colección para monitorear su procedencia, su uso de memoria, y las operaciones invocadas. En particular, para cada colección c — instancia de una clase C que pertenece a la biblioteca de colecciones — se asocia una tupla $(p, ops, icap, maxsize)$, donde p es un sitio de producción definido por un identificador de método y un intervalo de caracteres en el código fuente; ops es una lista de pares (m, f), donde m es un mensaje y f es un entero que representa la cantidad de veces que m fue recibido por c; $icap$ es la capacidad inicial de c, y $maxsize$ es el tamaño máximo que alcanza c.

Luego de la ejecución de la aplicación, se examinan todas las colecciones c_{i} producidas en el sitio p y se computa un $(p, OPS, ICAP, MAXSIZE)$ donde OPS, $ICAP$, $MAXSIZE$ son el promedio y desviación estándar de cada métrica individual.

4.1.2. Pattern matching

Las estadísticas obtenidas para cada sitio de producción de colecciones en el paso anterior son utilizadas para identificar sitios en los que la elección de colección o sus parámetros pueden ser mejorados. Para esto se consideran diversas reglas, definidas arbitrariamente, que describen patrones de uso deficiente de colecciones. Cada regla se implementa como una función que recibe una tupla $(p, OPS, ICAP, MAXSIZE)$ y entrega un puntaje entre 0 y 1. Un puntaje mayor a cierto umbral indica que la tupla cumple con el patrón definido. Si una misma tupla cumple con varios patrones, se escoge el con mayor puntaje. Las reglas consideradas son las siguientes: s

- *Colección vacía* — Las colecciones pueden mantenerse vacías luego de su creación: no se les ha añadido ningún valor. Pero se efectúan operaciones de todos modos, tales como enumeración y chequeo de inclusión. Conviene usar colecciones con inicialización perezosa de su arreglo interno.
- *Colecciones no usadas* — Algunas colecciones puede que nunca se usen. En ese caso, una colección no usada puede ser asignada de forma perezosa para evitar su creación innecesaria.
- *Tabla de hash pequeña* — Una tabla de hash presenta cierto consumo mínimo de memoria debido a la propiedad de *hashing* con la que tiene que cumplir. Una tabla de hash muy pequeña (5 o menos elementos) consume memoria de forma innecesaria. El no cumplir con la propiedad de distribución de *hash* en el caso de colecciones muy pequeñas reduce el consumo de memoria y tiempo de ejecución (en Pharo, la clase SmallDictionary implementa un diccionario de este tipo).
- *Capacidad inadecuada* — Las colecciones expandibles, tales como *ArrayList* y *HashMap*, tienen una capacidad correspondiente al tamaño de su arreglo interno. Una colección puede almacenar valores hasta el punto en que su tamaño (elementos almacenados) iguala a su capacidad. Cuando la capacidad es alcanzada, la colección debe crecer para acomodar más elementos. Para esto debe crear un nuevo arreglo interno y copiar en este los valores del anterior. El tener una capacidad inadecuada puede resultar en un

consumo de memoria excesivo [40]. Crear una colección con capacidad adecuada evita operaciones de crecimiento innecesarias.

- *Ordenar la colección* — Las colecciones pueden ser ordenadas de acuerdo a criterios de ordenamiento. Pero puede suceder que una colección sea ordenada múltiples veces, sin añadir o remover elementos. El usar una *SortedCollection* evita las operaciones de ordenamiento innecesarias.
- *Remoción de duplicados* — Si se realizan operaciones de remoción de duplicados repetidamente sobre un *ArrayList*, esta costosa operación podría ser evitada al usar un *Set*.

4.1.3. Reemplazo automático

Cada una de las reglas enunciadas anteriormente está asociada a una transformación de código fuente. Cuando una tupla $(p, OPS, ICAP, MAXSIZE)$ cumple con una regla, la transformación asociada es realizada de forma automática sobre el sitio del código fuente indicado en p por la herramienta de análisis dinámico que se diseñó e implementó, y el método o bloque es recompilado. Se mantiene un historial de cada reemplazo, el cual permite revertir los cambios.

4.2. Experimentos

4.2.1. Entorno experimental

Se implementó una herramienta de análisis dinámico inspirada en *Chameleon*, en el lenguaje de programación Pharo. La metodología empleada por la herramienta se detalla en los siguientes pasos:

1. *Profiling* — Los paquetes que contienen la aplicación a ser monitoreada son instrumentados antes de la ejecución del benchmark. Todos los sitios de producción de colecciones son identificados y el bytecode es instrumentado para generar los valores métricos mencionados en la sección 4.1.2.

2. *Candidatos a optimización* — Se obtienen estadísticas para cada sitio de producción de colecciones que luego son entregadas a un motor de reglas, el cual entrega una lista de modificaciones de código fuente sugeridas.

3. *Ejecución del benchmark* — Los benchmarks son ejecutados varias veces. Se usó un número arbitrario de 30 ejecuciones en los que se monitoreó el tiempo de ejecución y consumo de memoria.

4. *Aplicación de optimizaciones* — Las optimizaciones sugeridas por el paso 2 se aplican de forma automática sobre el código fuente original, el cual es recompilado.

5. *Ejecución del benchmark* — Los benchmarks son ejecutados nuevamente para medir la

variación de rendimiento.

4.2.2. Evitar el sesgo de medición

Es importante considerar el sesgo de medición cuidadosamente, ya que una configuración incorrecta puede llevar a resultados erróneos que llevan a conclusiones incorrectas. A pesar de las múltiples metodologías disponibles, es conocido que el evitar el sesgo de medición es difícil [22, 39, 23].

La recolección de basura copia y une diversas porciones de memoria para reducir la fragmentación [6]. Copiar y escanear porciones grandes de memoria, tal como las colecciones, puede causar pausas de recolección de memoria largas e impredecibles. El recolector de basura fue activado varias veces antes de correr cada benchmark para reducir este no-determinismo.

También se pasa por una etapa de *calentamiento* que consiste en ejecutar el benchmark antes de hacer cualquier medición. Esto tiene el efecto de activar el compilador JIT antes de realizar las mediciones.

4.2.3. Resultados

La herramienta de análisis se ejecutó sobre los mismos ejemplos de la sección 3.7. Para medir la memoria se usó MessageTally, en particular el valor *used*[1]. Cada benchmark se ejecutó 100 veces, y un resumen de los resultados se muestra en la tabla 4.1. Los resultados se presentan como intervalos de confianza al 90 % para la variación porcentual de la media de los resultados de tiempo y memoria, respectivamente, al pasar de la versión no optimizada a la versión optimizada. El intervalo de confianza para la variación porcentual se calcula según la expresión 1.11. La variación en tiempo de ejecución está indicada por $\mathbf{\Delta t}$ % y la variación en el consumo de memoria por $\mathbf{\Delta m}$ %.

Benchmark	$\mathbf{\Delta t}$ %	$\mathbf{\Delta m}$ %
CircularTreeMap	-8.35 ± 0.11	-3.57 ± 0.01
Mondrian	-2.51 ± 0.36	-1.0 ± 0.13
NameCloud	-2.64 ± 0.48	-3.7 ± 0.43

Tabla 4.1: Reemplazo selectivo de colecciones

La tabla 4.1 muestra que la aplicación de las sugerencias de optimización redujo tanto el tiempo de ejecución como el consumo de memoria en los tres ejemplos mostrados, principalmente en el caso de CircularTreeMap, que vio su tiempo de ejecución reducido en un 8 % y su consumo de memoria en 3.5 %. Las reglas que más se aplicaron fueron de capacidad inadecuada y de tabla de hash pequeña.

Los resultados obtenidos indican que efectivamente el reemplazo selectivo de colecciones en base a sus patrones de uso en tiempo de ejecución puede beneficiar tanto el tiempo

[1] Ver 1.4.2

de ejecución como el consumo de memoria de una aplicación. El desafío restante está en la definición de buenas reglas sobre las cuales hacer *pattern matching* y buenas transformaciones de código fuente asociadas a estas.

4.3. Experimento con las colecciones desarrolladas

Se crearon 3 reglas simples que retornan 1 siempre que se encuentran en presencia de sitios de produccion de instancias de OrderedCollection o Dictionary, y cuya transformación de código fuente consiste en reemplazar la clase por SLua, SmartCollection o SmartCollection2, respectivamente. Luego se usó la herramienta para selección adaptativa de colecciones mencionada anteriormente, y se ejecutaron los benchmarks 3 veces —30 iteraciones cada vez— uno por cada regla creada. De esta forma, se reemplazaron todas las instancias de OrderedCollection y Dictionary por las colecciones a comparar en cada benchmark.

Los resultados no entregaron diferencias estadísticamente significativas según los criterios especificados en la sección 1.3.1. Esto contrasta con los resultados expuestos en la tabla 3.7, e indica que la instrumentación amplifica considerablemente el impacto en la ejecución.

Sin embargo, cabe la posibilidad de que mediante reglas más específicas, que hagan uso de las fortalezas de cada colección y seleccionen de acuerdo a eso para cada contexto en particular, se pueda impactar positivamente en el consumo de memoria y/o tiempo de ejecución de aplicaciones mediante la utilización de las colecciones desarrolladas en este trabajo.

Conclusión

Se desarrollaron tres colecciones basadas en las tablas de Lua: SLua, SmartCollection y SmartCollection2. Las tres fueron evaluadas en base a micro y macro benchmarks, comparándolas con las colecciones tradicionales de Pharo y con una versión *lazy* de las mismas. Los macro-benchmarks midieron tiempo de ejecución y consumo de memoria mediante profiling con instrumentación en tiempo de ejecución. Adicionalmente se desarrolló una herramienta para hacer reemplazo selectivo de colecciones a nivel del código fuente en base a *pattern matching* entre un conjunto de reglas y el conjunto de patrones de uso de las colecciones según su sitio de producción. Luego se usó esta herramienta para evaluar nuevamente las colecciones desarrolladas, pero en un ambiente sin instrumentación de su código fuente.

Se plantearon las siguientes hipótesis:

H1 La colección adaptativa logra reducir, en promedio, el tiempo de ejecución de las aplicaciones al reemplazar colecciones tradicionales.

H2 La colección adaptativa logra reducir, en promedio, el consumo de memoria de las aplicaciones al reemplazar colecciones tradicionales.

Estas no pudieron confirmarse con los experimentos realizados. Al observar los macro-benchmarks de la sección 3.7 pareciera que sí se confirman, sin embargo estos resultados no tienen el mismo impacto cuando el reemplazo de clases se realiza a nivel del código fuente y no a través de instrumentación, según sugieren los resultados del experimento de la sección 4.3.

Si bien el objetivo general no se cumplió satisfactoriamente debido a la no confirmación de las hipótesis planteadas, sí se cumplieron objetivos específicos. Con respecto a estos:

1. Se diseñó no sólo una colección inspirada en las tablas de Lua, sino tres.

2. Se caracterizó el rendimiento en cuanto a uso de memoria y tiempo de ejecución mediante benchmarks y análisis del uso de colecciones durante los mismos.

3. Se comparó de forma rigurosa el rendimiento de las colecciones desarrolladas con el de las colecciones tradicionales. Esto incluyó diversas formas de medir el tiempo de ejecución y consumo de memoria, y un análisis estadístico riguroso de los resultados obtenidos.

4. Se identificó que el uso de las colecciones desarrolladas efectivamente puede ser bené-

fico, ya que al menos en algunos escenarios se comportan más rápido que las colecciones tradicionales, y en general consumen menos memoria. Si se definen las reglas adecuadas se podría usar la herramienta de reemplazo selectivo de colecciones para mejorar el rendimiento de una aplicación, tal vez con una mezcla de las tres colecciones desarrolladas.

Con respecto al impacto del trabajo, se logró poner a prueba a la tabla de Lua en un lenguaje con biblioteca de colecciones compleja, y se identificaron sus fortalezas y debilidades. En base a esta colección se desarrollaron dos más que sugieren ventajas de rendimiento. Además, se logró replicar el experimento Chameleon en Pharo, y se confirmó que se puede mejorar automáticamente el rendimiento de las aplicaciones mediante un análisis del uso de colecciones según su sitio de producción.

En el proceso de desarrollo y evaluación de las colecciones se aprendió acerca de análisis dinámico y de los muchos factores que pueden afectar los resultados. Se aprendió también que el comportamiento de las colecciones en la práctica no necesariamente se condice con el de los experimentos artificiales (micro-benchmarks en este caso), pues depende del uso que se les de a las mismas, del entorno de ejecución, la máquina virtual, del resto de los procesos que se estén ejecutando, y otros factores. Se aprendió también acerca de reflexión, y se puso en práctica mediante el desarrollo de herramientas como el profiler de uso de colecciones descrito en la sección 4.5.

A futuro, para mejorar el trabajo se podría incrementar la cantidad de benchmarks con escenarios más diversos que no fueron explorados, y hacer análisis del uso de las colecciones de los mismos para identificar patrones que mejoran o degradan el rendimiento. Luego, a partir del conocimiento obtenido se podrían desarrollar reglas de reemplazo de colecciones que ataquen patrones de uso más específicos, con lo que se podría mejorar el rendimiento de las aplicaciones.

Bibliografía

[1] William R. Cook. On understanding data abstraction, revisited. *SIGPLAN Not.*, 44(10):557–572, 2009.

[2] Klaus Wolfmaier, Rudolf Ramler, and Heinz Dobler. Issues in testing collection class libraries. In *Proceedings of the 1st Workshop on Testing Object-Oriented Systems*, ETOOS '10, pages 4:1–4:8, New York, NY, USA, 2010. ACM.

[3] Stéphane Ducasse, Damien Pollet, Alexandre Bergel, and Damien Cassou. Reusing and composing tests with traits. In *Proceedings of the 47th International Conference on Objects, Models, Components, Patterns (TOOLS'09)*, pages 252–271, Zurich, Switzerland, June 2009.

[4] Joseph (Yossi) Gil and Yuval Shimron. Smaller footprint for java collections. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, SPLASH '11, pages 191–192, New York, NY, USA, 2011. ACM.

[5] Carl Friedrich Bolz, Lukas Diekmann, and Laurence Tratt. Storage strategies for collections in dynamically typed languages. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 167–182, New York, NY, USA, 2013. ACM.

[6] Stelios Joannou and Rajeev Raman. An empirical evaluation of extendible arrays. In *Proceedings of the 10th International Conference on Experimental Algorithms*, SEA'11, pages 447–458, Berlin, Heidelberg, 2011. Springer-Verlag.

[7] Ohad Shacham, Martin Vechev, and Eran Yahav. Chameleon: Adaptive selection of collections. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 408–418, New York, NY, USA, 2009. ACM.

[8] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The implementation of lua 5.0. *Journal of Universal Computer Science*, 11(7):1159–1176, 2005.

[9] Adriana E. Chis, Nick Mitchell, Edith Schonberg, Gary Sevitsky, Patrick O'Sullivan, Trevor Parsons, and John Murphy. Patterns of memory inefficiency. In *Proceedings of the 25th European Conference on Object-oriented Programming*, ECOOP'11, pages

383–407, Berlin, Heidelberg, 2011. Springer-Verlag.

[10] Guoqing Xu. Coco: Sound and adaptive replacement of java collections. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 1–26, Berlin, Heidelberg, 2013. Springer-Verlag.

[11] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. Lua — an extensible extension language. *Softw. Pract. Exper.*, 26(6):635–652, June 1996.

[12] Roberto Ierusalimschy. *Programming in Lua, Second Edition*. Lua.Org, 2006.

[13] Damien Cassou, Stéphane Ducasse, and Roel Wuyts. Traits at work: the design of a new trait-based stream library. *Journal of Computer Languages, Systems and Structures*, 35(1):2–20, 2009.

[14] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, Alan Kay, and Walt Disney Imagineering. Back to the future: The story of squeak, a practical smalltalk written in itself. In *In Proceedings OOPSLA '97, ACM SIGPLAN Notices*, pages 318–326. ACM Press, 1997.

[15] Andrew Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Square Bracket Associates, 2009.

[16] Eliot Miranda. The cog smalltalk virtual machine — writing a jit in a high-level dynamic language. In *5th workshop on Virtual Machines and Intermediate Languages, VMIL at SPLASH '11*, 2011.

[17] Alan C. Kay. The early history of smalltalk. In Thomas J. Bergin, Jr. and Richard G. Gibson, Jr., editors, *History of Programming languages—II*, pages 511–598. ACM, New York, NY, USA, 1996.

[18] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[19] A.N. Clark. Metaclasses and reflection in smalltalk, 1997.

[20] B. Foote and R. E. Johnson. Reflective facilities in smalltalk-80. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '89, pages 327–335, New York, NY, USA, 1989. ACM.

[21] Alan Kay. Flex - a flexible extendable language. Master's thesis, University of Utah, 1968.

[22] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 57–76, New York, NY, USA, 2007. ACM.

[23] Tomas Kalibera and Richard Jones. Rigorous benchmarking in reasonable time. In

Proceedings of the 2013 International Symposium on Memory Management, ISMM '13, pages 63–74, New York, NY, USA, 2013. ACM.

[24] David J. Lilja. *Measuring computer performance : a practitioner's guide*. Cambridge University Press, Cambridge, New York, Merlbourne, 2000. Appendices. Index.

[25] Larry. Wasserman. *All of Statistics*. Springer, 2004.

[26] H. H. Ku. Notes on the use of propagation of error formulas. *Journal of Research of the National Bureau of Standards. C, Engineering and instrumentation*, 70C(4):262, October 1966.

[27] Stephen M. Ross. Peirce's criterion for the elimination of suspect experimental data. *Journal of Engineering Technology*, 2003.

[28] Benjamin Peirce. Criterion for the rejection of doubtful observations. *Astronomical Journal*, 2(45):161–163, July 1852.

[29] Camillo Bruni. Optimizing Pinocchio. Master's thesis, University of Bern, January 2011.

[30] Alexandre Bergel, Felipe Bañados, Romain Robbes, and David Röthlisberger. Spy: A flexible code profiling framework. *Journal of Computer Languages, Systems and Structures*, 38(1):16–28, December 2011.

[31] Ole Agesen. Type feedback vs. concrete type inference: A comparison of optimization techniques for object-oriented languages. In *In Proceedings of the 1995 ACM Conference on Object Oriented Programming Systems, Languages, and Applications*, pages 91–107. ACM, 1995.

[32] Alexandre Bergel, Stephane Ducasse, Damien Cassou, and Jannik Laval. *Deep into Pharo*. Square Bracket LLC, Sep 2014.

[33] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM.

[34] PaulR. Wilson, MarkS. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In HenryG. Baler, editor, *Memory Management*, volume 986 of *Lecture Notes in Computer Science*, pages 1–116. Springer Berlin Heidelberg, 1995.

[35] Jennifer B. Sartor, Stephen M. Blackburn, Daniel Frampton, Martin Hirzel, and Kathryn S. McKinley. Z-rays: Divide arrays and conquer speed and flexibility. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and*

Implementation, PLDI '10, pages 471–482, New York, NY, USA, 2010. ACM.

[36] David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 285–298, New York, NY, USA, 2003. ACM.

[37] G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, B. Mathiske, and M. Wolczko. Heap compression for memory-constrained java environments. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, OOPSLA '03, pages 282–301, New York, NY, USA, 2003. ACM.

[38] Alexandre Bergel and Vanessa Peña. Increasing test coverage with hapao. *Science of Computer Programming*, 79(1):86–100, 2012.

[39] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 265–276, New York, NY, USA, 2009. ACM.

[40] Alexandre Bergel, Alejandro Infante, and Juan Pablo Sandoval Alcocer. Reducing waste in expandable collections: The pharo case. Technical Report TR/DCC-2015-5, University of Chile, November 2015.

Anexo

A continuación se expone brevemente acerca del diseño e implementación de las herramientas que fueron desarrolladas para llevar a cabo la evaluación de las colecciones.

4.4. Infraestructura para Macro-benchmarks

Inicialmente, para realizar los benchmarks se construyó una solución en torno a *SMark*. Esta consistía en realizar una instrumentación con un profiler liviano, que sólo hace el reemplazo de clases, antes de correr cada benchmark (usando los métodos *#setUp* y *#tearDown* descritos en 1.3.2). Sin embargo, la solución resultó ser muy lenta ya que era necesario instrumentar antes de cada ejecución del benchmark y desinstrumentar después de las mismas.

Como solución a ese problema, se desarrolló una infraestructura sencilla que permite instrumentar una sola vez por cada benchmark, luego correr todas las ejecuciones necesarias del mismo y tomar sus tiempos, y luego desinstrumentar. El diseño de esta infraestructura se basó en SMark en cuanto posee una clase principal *ClassReplacementBenchmark* que encapsula la funcionalidad de realizar los benchmarks, y de la cual se debe extender para formar *suites* de benchmarks[2]. Cada *suite* define sus benchmarks como métodos, y estos se ejecutan llamando al método *#run:* que recibe como parámetro el número de iteraciones que se quiere ejecutar. Cuando una instancia a de alguna subclase A de *ClassReplacementBenchmark* recibe el mensaje *#run:*, este recopila todos los métodos de a que comienzan con el prefijo "bench", luego instrumenta los paquetes definidos en el benchmark, realiza el reemplazo de clases correspondiente, ejecuta los benchmarks, desinstrumenta y luego entrega los resultados.

Cada método que define un benchmark debe retornar una asociación[3] donde la llave es un conjunto de paquetes y el valor un bloque con el benchmark en sí. De este modo, cuando se llame a *#run:* el framework sabrá qué paquetes instrumentar antes de ejecutar el benchmark. Para definir qué clases reemplazar se utiliza el método *#for:insteadOf:* el cual recibe como primer argumento una lista de clases a reemplazar y como segundo argumento una lista de clases por las cuales reemplazarlas.

La instrumentación se lleva a cabo con un *profiler* creado en Spy que sólo hace el reemplazo

[2] Ver 1.3.2

[3] En Pharo, una *asociación* es un par (llave, valor).

de clases. Antes de medir el tiempo se realiza una recolección de basura completa y se ejecuta el bloque del benchmark un par de veces para que el compilador JIT produzca el código necesario. El tiempo se mide enviando el mensaje timeToRun al bloque que contiene el benchmark por cada iteración que se quiera medir. Los resultados se entregan como una lista de tiempos medidos.

Para realizar la evaluación estadística de los resultados descrita en 1.3.1 se creó la clase *StatisticsHelper*, que implementa funciones para calcular intervalos de confianza de la media y para eliminar outliers.

4.5. Profiler de Uso de Colecciones

Para obtener estadísticas de uso de las colecciones según su sitio de creación se desarrolló, mediante Spy, el profiler *CollUsage*. El profiler instrumenta los paquetes a analizar, realiza reemplazo de clases, y también instrumenta otros objetos para garantizar la captura de todas las instancias de colecciones que se generen durante la ejecución de código en los paquetes instrumentados. Por ejemplo, para copiar una colección *a* podemos enviarle a esta el mensaje *#copy*, sin embargo ese método está implementado en la clase *Object* por lo que no estará instrumentado si solamente se instrumentan las clases sobre las cuales se ejecutará el benchmark. Es necesario entonces modificar el método *#shallowCopy* (que es llamado por *#copy*) de la clase *Object* para que esa instancia pueda ser capturada. Por el mismo motivo se instrumentan también *#basicNew* y *#as:*.

Luego de capturar una colección recientemente creada, es necesario obtener el sitio de donde provino. Para esto se hace uso de las capacidades reflexivas de Pharo: mediante una llamada a *thisContext* se obtiene el contexto de ejecución actual, a través del cual se retrocede contexto en contexto hasta encontrar el sitio de donde provino la colección. Este sitio corresponde al primer contexto cuyo receptor no es una colección, ni una clase de colección, ni una clase de Spy. Luego, a partir del contexto encontrado se crea una instancia de *ContextInfo*, que contendrá una referencia al método y bloque (si existe) que lo originan y el valor del *instruction pointer* (IP) en ese momento (este valor sirve para identificar el contexto, pues el mero método no basta). A través del IP también se obtiene la porción de código fuente que origina a la colección, gracias al *debugger* de Pharo.

Cada colección que se cree estará asociada a un objeto *UsageData* que almacenará todos los mensajes que la colección reciba, su capacidad inicial, su tamaño máximo alcanzado, y el contexto que la originó. Al terminar la ejecución, las instancias de *UsageData* se agruparán según contexto para formar instancias de *UsageStatistics*, las cuales resumen la información recopilada para todas las instancias de colecciones generadas en ese contexto[4]. El resultado del profiler es un conjunto de *UsageStatistics*.

[4] Ver 4.1.1

4.6. Profiler de Memoria para Colecciones

Se desarrolló un profiler muy sencillo para medir el consumo de memoria de las colecciones. El profiler captura las colecciones generadas a través de llamadas a *#basicNew* y a *#shallowCopy*, y las almacena agrupadas según clase. Cuando la ejecución concluye, el profiler calcula el consumo de memoria de cada colección mediante el mensaje *#sizeInMemory* que entrega el tamaño en bytes del objeto, y a eso se le suma la capacidad de la colección multiplicada por 4 (cada puntero a elemento ocupa 4 bytes).

4.7. Selección Adaptativa de Colecciones

Esta sección describe la herramienta de selección adaptativa de colecciones, cuyo propósito es mejorar el tiempo de ejecución y/o consumo de memoria de una aplicación mediante el reemplazo automático y posterior recompilación de porciones del código fuente. Para lograrlo, la herramienta sigue la metodología planteada en 4.1.2. Esto es:

(i) obtener estadísticas de uso de las colecciones producidas en cada sitio de producción de colecciones de la aplicación,

(ii) pasar las estadísticas por un proceso de *pattern matching* en el cual algunas serán seleccionadas en base a un conjunto de reglas predefinido,

(iii) realizar la modificación del código fuente a para los sitios de producción que hayan sido seleccionados en el paso anterior

Para el paso (i) la herramienta hace uso del profiler descrito en 4.5, la cual entrega un conjunto de *UsageStatistics*. Para el paso (ii), las *UsageStatistics* se pasan como entrada a una instancia de la clase *RulesEngine*, la cual tiene un conjunto de reglas sobre las cuales se prueba cada instancia de *UsageStatistics*. Las instancias que son seleccionadas por alguna regla son candidatas a modificación de código fuente (del método al cual referencian)[5]. Las reglas se definen como subclases de la clase *CollRule*, y deben implementar los métodos *#computeScore* y *#replacementSuggestionFor:*. El primero entrega el puntaje entre 0 y 1 asignado al *UsageStatistics* actual y el segundo entrega una sugerencia de modificación de código fuente para un *ContextInfo* dado.

El paso (iii) también lo lleva a cabo una instancia de *RulesEngine*, la cual modifica el código y lo recompila según las recomendaciones entregadas por las reglas, y mantiene un historial con estas modificaciones lo cual le permite deshacerlas posteriormente.

Para realizar los benchmarks se hace uso de la clase *CollOptimizer*, cuyas instancias reciben un bloque a optimizar, los paquetes sobre los que cuales el análisis, un conjunto de reglas, y un número i de iteraciones. Luego, la instancia de *CollOptimizer* realiza el proceso indicado en los pasos (i), (ii) y (iii) usando las reglas provistas, y posteriormente ejecuta i mediciones para el tiempo e i mediciones para la memoria. El consumo de memoria se mide usando MessageTally.

[5] Ver 4.1.2

Posteriormente, se deshacen los cambios y se vuelven a hacer las i mediciones. Finalmente se comparan los resultados con y sin optimización, siguiendo la evaluación estadística indicada en la sección 1.3.1.