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

Una implementación transparente de Memoria Transaccional en un lenguaje dinámico Orientado a Objetos

Tesis presentada para optar al título de
Licenciado en Ciencias de la Computación

Luciano Leveroni

Director: Hernán Wilkinson

Buenos Aires, 2017

UNA IMPLEMENTACIÓN TRANSPARENTE DE MEMORIA TRANSACCIONAL EN UN LENGUAJE DINÁMICO ORIENTADO A OBJETOS

Memoria Transaccional (de ahora en adelante, STM) emerge como una alternativa a técnicas tradicionales de control de concurrencia, basadas principalmente en la idea de *locks* y sus variantes (semáforos, monitores, etc). En los últimos años la investigación sobre este tema se encuentra en auge.

Este trabajo describe una implementación transparente de STM sobre el lenguaje Ruby. Entendemos por transparente a una implementación que, a diferencia de otras implementaciones existentes, no resulta intrusiva para el programador-usuario de esta extensión del lenguaje.

La implementación realizada es compatible con MRI, la implementación oficial de Ruby, así como Rubinius, una implementación alternativa realizada sobre Ruby mismo. La técnica utilizada se basa en la generación dinámica de código, en tiempo de ejecución, fuertemente basada en el trabajo *Transactional Memory for Smalltalk*[7] (2008, Lukas Renggli et. al).

La investigación se centra en exhibir las ventajas y limitaciones del lenguaje Ruby al momento de implementar STM, así como los obstáculos y beneficios particulares de las dos implementaciones de Ruby mencionadas. Se muestra cómo las distintas capacidades reflexivas, características del meta-modelo y de meta-programación del lenguaje inciden en la posibilidad de implementar esta técnica sin modificaciones necesarias a las VMs asociadas (salvo algunas excepciones).

Cómo conclusión, Ruby como lenguaje ofrece en principio las herramientas para lograr una de STM robusta y completa. Sin embargo, la problemática de las primitivas de lenguaje junto con algunas deficiencias de implementación y limitaciones del meta-modelo, derivaron en una implementación más compleja y acoplada a particularidades implementativas. No obstante, se logró una implementación funcional que requiere modificaciones a nivel implementación del lenguaje / VM para solventar las limitaciones expuestas.

Palabras claves: STM, Memoria transaccional, orientado a objetos, dinámicamente tipado, Ruby, MRI, Rubinius, primitivas, meta-modelo Ruby, inmutabilidad, Ruby VM, Smalltalk, generación dinámica de código.

A TRANSPARENT IMPLEMENTATION OF TRANSACTIONAL MEMORY FOR A DYNAMIC OBJECT-ORIENTED LANGUAGE

Transactional Memory (from now on, STM) emerges as an alternative to traditional concurrency control techniques based mainly in the idea of locks and it's variants (semaphores, monitors, etc). Over the last years, research on this topic is booming.

This work describes a transparent implementation of STM for the Ruby language. By transparent, we mean an implementation that, in contrast with existing ones, is non-intrusive for the programmer-user of this language extension.

The implementation presented is compatible with MRI, the official Ruby implementation, as well as Rubinius, an alternative implementation based on Ruby itself. The technique used is strongly based on the work Transactional Memory for Smalltalk (2008, Lukas Renggli et. al.).

Research is focused on showing the advantages and limitations of the Ruby language in relationship with implementing STM, as well as obstacles and benefits particularly linked with the two implementations mentioned. It is shown how reflexive capabilities, meta-programming and meta-model characteristics of the language affect the possibility of implementing this technique without needing to modify the corresponding VMs (with some exceptions).

In conclusion, Ruby as a language offers the tools needed to achieve a robust and complete STM implementation. However, issues with language primitives, together with some implementation deficiencies and limitations of the meta-model, derived in a more complex implementation, somewhat fragile and coupled to implementation details. Nonetheless, a functional implementation was achieved that requires changes at the language implementation level / VM to solve the exposed limitations.

Keywords: STM, Transactional memory, Object oriented, dynamic typing, Ruby, MRI, Rubinius, primitives, Ruby meta-model, inmutability, Ruby VM, Smalltalk, dynamic code generation.

AGRADECIMIENTOS

A mis viejos, por darme todas las herramientas necesarias (y más) desde que nací y el apoyo incondicional, en todo sentido, para poder llegar hasta acá. Hubiese sido imposible sin ustedes.

A mi hermano quien, a su manera, siempre confió y creyó en mí. Que fue partícipe fundamental y cómplice en mi infancia para despertar ese interés por la ciencia y en particular por la computación.

A mis primos Fran y Nahui, que compartieron durante mi infancia y adolescencia ese interés por la computación, por el *hardware*, por Linux y demás ñoñadas.

A los familiares, en especial tías y tíos, que desde siempre creyeron en mí y en mis capacidades. Mención aparte para mi abuela Marta y mi abuelo Luis, por lo que fueron y son.

A Agus, Mati, Fede e Igal por todas las tardes de trabajos prácticos, de juntarse a estudiar para parciales y finales a lo largo de casi toda la carrera, y por su gran amistad en estos 10 años.

A Gonzo, al Negro, Dani y Chulets por las cenas, los partidos de truco y su compañía estos últimos años de carrera.

A mis compañeros de Eryx, que día a día me hacen ir con muchas ganas a trabajar, hace ya más de 6 años.

A Mari, por ese amor que me llena todos los días y por tu hermosa compañía durante este tramo final de mi carrera.

A todos los grandes docentes que tuve en la carrera, que son tantos que no podría enumerarlos a todos. Docentes de teórica, JTPs y ayudantes que hicieron posible que llegue hasta acá.

A quienes luchan y lucharon por una educación pública de calidad, que me permitieron tener una educación de primer nivel, con docentes de una calidad humana y académica que nunca imaginé poder tener.

A los que la vida me cruzó durante estos 10 años, que quizás ahora me esté olvidando porque hoy no están cerca pero en algún momento lo estuvieron.

A mis viejos y a mi hermano.

Índice general

1. INTRODUCCIÓN

En el cómputo concurrente varias operaciones se ejecutan en intervalos de tiempo solapados, en lugar de en forma secuencial donde la siguiente comienza recién cuando termina la última. Visto como paradigma de programación, implica la escritura de programas donde la solución de un problema a resolver se divide en partes de forma tal que las mismas pudiesen ser ejecutadas simultáneamente (concurrentemente). Por si misma, esta técnica resulta interesante a nivel diseño de una aplicación ya que en la práctica distintos problemas del mundo real son naturalmente un conjunto de subproblemas concurrentes.

Como sabemos, el principal desafío de la programación concurrente tiene que ver con el control simultáneo de accesos a recursos compartidos. La ejecución de 2 instancias de un mismo procedimiento en forma concurrente podría derivar en comportamientos inesperados, producto del estado de las variables que se están leyendo y escribiendo en un orden desconocido.

A lo largo de la historia han surgido diversas técnicas para resolver esta problemática, siendo las más populares aquellas basadas principalmente en el concepto de *locking*. Por otra parte, han surgido modelos alternativos como el modelo de Actores, con otro paradigma de computo que evita el uso de *locks*. La desventaja de estos mecanismos es que resultan en tener que re-escribir programas existentes de formas no triviales, incluyendo en muchas ocasiones repensar algoritmos y estructuras de datos fundamentales de otra manera.

En particular, las técnicas mencionadas suelen resultar inconvenientes al momento de aplicarlas en el contexto de un software con un diseño Orientado a Objetos. De esta problemática fundamental surge la motivación de estudiar otras técnicas que interfieran lo menor posible con la forma de razonar al momento de escribir software siguiendo las premisas fundamentales de este paradigma.

El siguiente trabajo se centra un mecanismo de control de concurrencia distinto a las técnicas tradicionales: Memoria Transaccional. El objetivo será realizar una implementación de esta técnica en un lenguaje Orientado a Objetos y analizar las distintas limitaciones y/o ventajas que brinda un lenguaje basado en este paradigma. En particular, se verá cómo las capacidades reflexivas de un lenguaje dinámico inciden en la posibilidad de implementar esta técnica de forma tal que su aplicación resulte transparente para el usuario de la misma. La implementación será realizada en Ruby, un lenguaje dinámico Orientado a Objetos surgido a mediados de la década del '90 que ha ganado popularidad en los últimos años.

1.1. Por qué Memoria Transaccional

Desde el punto de vista de la *performance*, el enfoque optimista de control de concurrencia que brinda Memoria Transaccional (de ahora en adelante, **STM**) implica un incremento en el nivel de concurrencia de un sistema, ya ningún *thread* necesita esperar para poder acceder a un recurso. Este motivo por sí sólo representa una ventaja importante respecto al *locking* tradicional, resultando entonces un motivo suficiente por el cual optar por implementar este mecanismo de control.

Sin embargo, el motivo principal por el cuál esta técnica resulta de interés en el contexto

de la programación Orientada a Objetos radica en cómo la aplicación de la misma simplifica el entendimiento de los programas concurrentes. STM no interfiere con los conceptos de un sistema diseñado Orientado a Objetos sino que funciona en armonía con abstracciones de alto nivel como objetos y clases. Proveer memoria transaccional como facilidad para la ejecución concurrente segura implica solamente el envío de un mensaje a un bloque de código indicando que se desea ejecutar dicho código en forma atómica.

Veamos a continuación un ejemplo de control de concurrencia utilizando un semáforo (izquierda) y STM (derecha):

Listing 1.1: Usando un semáforo

```
an_array = ['a', 'c', 'd']
semaphore = Mutex.new

# writing
semaphore.synchronize do
  an_array.insert(1, 'b')
end

# reading
semaphore.synchronize do
  an_array.at(1)
end
```

Listing 1.2: Usando STM

```
an_array = ['a', 'c', 'd']

# writing
Proc.new do
  an_array.insert(1, 'b')
end.atomic

# reading
an_array.at(1)
```

Como se puede observar, el problema principal de la solución de la izquierda es que se debe utilizar el mismo *lock* para todos los accesos de lectura y escritura del arreglo. Esto implica que el *lock* debe ser preventivamente utilizado previo al envío de cualquier mensaje que forme parte de la interfaz pública del objeto.

En cambio, la solución utilizando STM simplemente necesita indicar que la escritura se realizará en el contexto de una transacción. Para esto, bastará con enviar el mensaje ***atomic*** a un bloque de código que contenga el acceso de escritura al arreglo. De este modo, el protocolo de *commit* optimista garantizará el acceso concurrente al objeto en forma segura. Cabe notar que, por este mismo protocolo, si toda escritura se realiza en forma atómica las lecturas se podrán realizar en forma segura sin necesidad de encontrarse dentro de un bloque atómico (de igual forma en la que se realizan las lecturas en un contexto no concurrente).

Por último, cabe mencionar que este mecanismo de control de concurrencia basado en transacciones es utilizado con éxito en Bases de Datos relacionales desde hace décadas, por lo que resulta natural extenderlo al dominio de los Objetos al momento de persistir los mismos concurrentemente en forma segura.

1.2. Por qué Ruby

La elección de Ruby como lenguaje se basa en distintos aspectos del mismo:

- Es un lenguaje relativamente moderno (su primer versión pública data de Diciembre de 1995), con un mercado interesante y que ha ganado popularidad en los últimos años.

- No existen actualmente implementaciones robustas conocidas de esta técnica en el lenguaje.
- Se trata de uno de los lenguajes más puros Orientados a Objetos y se jacta de ser un lenguaje transparente, por lo que resulta de interés estudiar sus capacidades reflexivas.
- Además de la implementación oficial (Matz's Ruby Interpreter, de ahora en adelante **MRI**) escrita en lenguaje C, existe otra implementación (entre varias otras) llamada Rubinius implementada (en gran parte) en Ruby mismo. Dada esta última particularidad, resulta interesante analizar posibles ventajas y/o limitaciones de implementar STM como extensión a un lenguaje implementado en un lenguaje estructurado y de bajo nivel como C versus un lenguaje Orientado a Objetos con las capacidades ya mencionadas.

1.3. Estado del arte

Si bien se trata de una técnica relativamente reciente, existen implementaciones de Memoria Transaccional para distintos lenguajes de programación utilizados hoy en día, variando entre sí en cuanto a robustez y transparencia.

En primer lugar, existen librerías/*frameworks* que implementan bases de datos de objetos sobre lenguajes dinámicos Orientados a Objetos. Estos utilizan Memoria Transaccional como técnica para manipular los objetos concurrentemente en forma segura. Podemos mencionar:

- **AtomizeJS:** Se trata de una librería externa de Javascript (lenguaje dinámico prototipado) que implementa una arquitectura cliente-servidor para lograr persistir un grafo de objetos distribuidos automáticamente a todos los navegadores cliente conectados. [1]
- **GemStone Smalltalk:** Se trata de una variante propietaria de Smalltalk que implementa una base de datos de objetos distribuida transaccional, bajo una arquitectura cliente-servidor. [2]
- **Ruby MagLev:** Implementación abierta y alternativa del lenguaje Ruby, construida sobre GemStone (y dependiente de este último), que implementa persistencia transparente de objetos distribuida y transaccional. [3]

Respecto a implementaciones de memoria transaccional como extensiones de lenguaje (dinámicos, Orientados a Objetos) podemos mencionar:

- **Python STM:** Prueba de concepto realizada sobre el lenguaje Python. En concreto, se realizaron modificaciones al intérprete de Python oficial escrito en C [4]. Este *fork*, realizado en 2011, no forma parte de la distribución oficial. Sin embargo, una implementación denominada PyPy, alternativa Python a la oficial, incluye soporte en su versión estable, basada en el mencionado *fork* [5].
- **Smalltalk STM:** Se trata de una prueba de concepto implementada en 2008 sobre *Squeak 3.9*. Resulta de interés por su técnica de transformación de código y su implementación escrita en Smalltalk como una propia extensión del lenguaje, sin modificaciones a nivel Virtual Machine. [7]

Para el caso de **Ruby**, encontramos una única implementación conocida distribuida bajo un paquete denominado **Concurrent Ruby**. El mismo se trata de una serie de herramientas para escritura de programas concurrentes. En este caso, la técnica implementada para STM consiste en garantizar atomicidad para aquellas variables de tipo "TVar" (contenedor para cualquier otro tipo de variable) que se encuentren dentro de un bloque "atómico" [6]. Esta característica hace de ésta una implementación muy poco transparente (y poco robusta), ya que requiere modificar el código dentro de cada bloque atómico, reemplazando toda variable por una variable de tipo "TVar". Potencialmente, podría implicar tener que reescribir todo método que pueda a llegar a ser referenciado dentro de un bloque atómico. Se este hecho surge inmediatamente la pregunta: ¿Por qué debe realizar este trabajo el usuario cuando podría hacerlo la computadora?

2. ESQUEMA GENERAL Y ESTRATEGIAS

Podemos definir al esquema general de implementación de STM como una serie de pasos que ocurren desde que el mensaje **atomic** es enviado a un bloque de código. Las etapas en cuestión serán:

1. **Comienzo de la transacción:** Se registra el inicio de una nueva transacción para el *thread* en ejecución.
2. **Registro de cambios / Operando en memoria paralela:** A partir de ese momento, todos los accesos de lectura y/o cambios realizados en los objetos en ese contexto se deberán registrar sobre un espacio de memoria paralelo, ya que los mismos no deben ser visibles hacia otros *threads*.
3. **La fase de *commit*:** Al finalizar, se verifican conflictos entre los cambios realizados en la memoria paralela y el estado actual de la memoria real (relativa a la transacción que está se está *commiteando*) y se procede a aplicar los cambios a esta última (o a desecharlos si hay conflictos).

A continuación abordaremos cada una de ellas, analizando distintas estrategias de implementación para cada una de ellas.

2.1. Comienzo de la transacción

El comienzo de una nueva transacción no presenta mayores dificultades de por sí. La idea sería simplemente registrar en alguna variable que la transacción ha comenzado. Dado que puede haber múltiples ejecuciones concurrentes del mismo código pero tan sólo una transacción por *thread*, resulta natural almacenar la transacción actual relacionándola con el *thread* actualmente en ejecución.

Dado que el lenguaje Ruby está en sí mismo modelado Orientado a Objeto, la solución más razonable consiste en extender la clase que representa un *thread* con un mensaje que permita asociar la nueva transacción (también modelada como un objeto) con el *thread* actual. Se verá más adelante los detalles de implementación en relación a esto.

2.2. Registro de cambios / Operando en memoria paralela

La principal dificultad de implementar STM radica en el hecho de que, implementar el concepto de Transacción en Memoria implica necesariamente realizar las operaciones en un espacio de memoria paralelo a la memoria donde se almacenan los objetos, de modo de poder aislar las transacciones y no afectar el estado de los objetos, antes de finalizar la transacción.

Existen dos caminos posibles para lograr operar en un espacio de memoria paralelo:

1. **Alterando el código original (manualmente).** Por ejemplo, indicando al desarrollador que debe reemplazar todo objeto que participe en la transacción por otro objeto que encapsule al original, encargándose este último de escribir (y leer) las modificaciones en una memoria paralela. Este es el enfoque utilizado por la libería **Concurrent Ruby**, citada previamente.

2. **Sin alterar el código original.** La idea aquí sería que el mismo ambiente intercepte los accesos a memoria de algún modo, de forma tal de realizar las escrituras y lecturas en una memoria paralela.

Dado que se busca realizar una implementación **transparente** de Memoria Transaccional, se optó por investigar soluciones que siguieran el segundo camino.

2.2.1. Primer enfoque: Interceptación directa de asignaciones y lecturas de variables

La forma más directa de lograr trabajar en una memoria paralela sería alterar el algoritmo de asignación (y lectura) de variables del lenguaje. De este modo, se podría marcar el inicio de la ejecución transaccional de algún modo (al comienzo del envío del mensaje **atomic** al bloque de código) y luego desviar todas las asignaciones hacia un espacio de memoria paralelo, hasta el final de la transacción. Se deberían también interceptar las lecturas a variables de modo que se realicen sobre este espacio alternativo.

El problema con este enfoque radica en que tanto el operador de asignación '=' así como la lectura de una variable (invocada implícitamente con el nombre de la misma, en ciertos contextos) se tratan de operaciones muy primitivas del lenguaje. Incluso en un lenguaje Orientado a Objeto con las características reflexivas de *Smalltalk*, dichas operaciones no se encuentran implementadas como mensajes a un objeto en particular.

Para realizar esto en Ruby (tanto en MRI como en Rubinius) se requiere modificar la implementación del lenguaje a muy bajo nivel. En ambos casos, la asignación y lectura de variables ocurre a nivel *Virtual Machine*: El *parser* del lenguaje genera el AST correspondiente y las asignaciones/lecturas se compilan a instrucciones particulares de la VM, que incluso dependen del contexto. La instrucción de asignación de una variable de instancia es distinta a la asignación de una variable local, por citar un ejemplo. Por lo tanto, este camino requiere modificar en forma no trivial la implementación de la *Virtual Machine*.

Un aspecto negativo adicional e importante que conlleva realizar una implementación a nivel *Virtual Machine* tiene que ver con la gran diferencia que existen entre las distintas implementaciones. En caso de que opte por dar soporte para MRI, Rubinius y JRuby (otra implementación muy popular), se deberá utilizar 3 lenguajes distintos: C (MRI), C++ (Rubinius) y Java (JRuby). Notar incluso que, a pesar de la similitud del lenguaje entre la implementación de MRI y Rubinius, las *VMs* de ambas implementaciones difieren significativamente. Rubinius, a diferencia de MRI, está escrito sobre un *framework* de *VMs* llamado LLVM, por lo cual posiblemente habría que diseñar una implementación particular para cada caso.[9]

Aún cuando se implemente para una única *VM*, se debe lidiar con problemas intrínsecos al desarrollo de *VMs*:

- Dificultad en la mantenibilidad de la implementación ante cambios en el lenguaje, producto del desarrollo a tan bajo nivel.
- Dada las optimizaciones implementadas en forma local, transversalmente al código, la dificultad de la implementación probablemente aumentará (aún cuando no se decida optimizar la propia implementación).
- Dado que se trata de una parte delicada del proceso de ejecución de una aplicación, aumentaría la probabilidad de que *bugs* de implementación vuelvan inestable a esta última.

2.2.2. Segundo enfoque: Transformación de código

Una alternativa a la interceptación directa de los accesos a variables es reemplazar dichos accesos por mensajes alternativos, previo a ejecutar el código del bloque que se desea ejecutar atómicamente. Los mensajes alternativos podrían ser mensajes implementados en Ruby, que simplemente trabajen sobre una copia del objeto original. Estos reemplazos deberían aplicarse desde el comienzo de la ejecución del bloque hasta el final, para todo método que se ejecute dentro del contexto del bloque en cuestión.

Transformación en tiempo de *parsing* y compilación

Una forma de implementar esta idea es **alterar el *parser* original del lenguaje**. De este modo, el reemplazo sucedería en tiempo de *parsing* y compilación del programa. Previo a la ejecución del *script* que contiene el código fuente con el bloque, cada vez que se encuentre con un identificador que representa la asignación o la lectura de una variable dentro del contexto de ejecución del bloque se genera código alternativo, enviando un mensaje que se encargará de acceder a una copia de trabajo del objeto. A continuación se presenta un ejemplo de esto:

Listing 2.1: Original

```
Proc.new do
  @a = 1
  puts @a
end.atomic
```

Listing 2.2: Transformado

```
Proc.new do
  instance_variable_set_atomic(:@a, 1)
  puts instance_variable_get_atomic(:@a)
end.atomic
```

Sin embargo, existen dificultades técnicas al alterar el *parser* original. En primer lugar, el proceso de *parsing*, compilación y ejecución de un programa Ruby **no es lineal**: Dado que un *script* Ruby puede incluir otros *scripts* (por ejemplo vía el mensaje **require** implementado en el módulo **Kernel**), el lenguaje parsea y genera el *byte-code* correspondiente al script que se ejecuta. Luego, ejecuta las instrucciones generadas y, al momento de encontrar una instrucción de carga de otro *script* procede a parsear, compilar y ejecutar dicho *script* cargado. Por este motivo, hallar un mecanismo para poder realizar los reemplazos de mensajes solamente en el contexto de ejecución del bloque atómico puede resultar difícil.

Por otra parte, en el caso de MRI, el *parser* forma parte de la implementación (escrito en C), por lo cual habría dificultades similares a las mencionadas respecto de alterar la *VM*. El *parser* de Rubinius, por el contrario, sí está implementado en Ruby e incluso provee una arquitectura para implementar transformaciones AST propias. Igualmente, el problema con alterar el *parser* original es que se necesita hacer esto sobre la base de código original de Rubinius, no pudiendo utilizar la capacidad de apertura de clases provista por Ruby. El motivo de esto es que el compilador que se utiliza es una versión que ya se encuentra pre-compilada a *byte-code* (esto es necesario, ya que para compilar el compilador se necesitaría tener previamente un compilador).

Una alternativa a alterar el parser original provisto por el lenguaje, es realizar la transformación de código **en tiempo de ejecución**. El código transformado luego será parseado, compilado y ejecutado también en tiempo de ejecución.

Transformación de código en tiempo de ejecución

A diferencia de la transformación realizada en tiempos de *parsing* y compilación, la transformación en tiempo de ejecución implica generar en *runtime* un nuevo ***proc*** (objeto de la clase **Proc** que reifica el concepto de *closure* en Ruby) que se pueda ejecutar en forma atómica. El código del nuevo *proc* será equivalente al anterior, pero con dos cambios:

1. Al igual que antes, se reemplazan los mensajes de asignación y lectura de variables de instancia contenidos dentro del bloque original.

2. Se reemplazarán todos los envíos de mensajes por mensajes alternativos 'atómicos'.

El segundo punto es necesario para poder luego transformar, también en tiempo de ejecución, todo método que se llame en el contexto del *proc*. Dichos métodos serán transformados con los mismos dos cambios mencionados. De este modo, toda variable de instancia involucrada en los métodos relacionados también será accedida a través de un mensaje alternativo, logrando efectivamente trabajar sobre un espacio de memoria paralelo. Adicionalmente, todo mensaje enviado en el cuerpo de los métodos transformados también será reemplazado por mensajes alternativos atómicos, a los que luego se le aplicarán las mismas transformaciones a los métodos relacionados. El proceso se repetirá recursivamente por todos los caminos de ejecución que surjan a partir del *proc* original.

Para lograr realizar las transformaciones a través de todo el árbol de ejecución, una vez generado el nuevo *proc* con código transformado, se procederá a ejecutar el mismo. Al intentar ejecutarse, Ruby no encontrará ninguno de los mensajes alternativos 'atómicos' ya que los mismos no se encuentran definidos. Al ejecutar el algoritmo de *method lookup* y no encontrar el método definido en ninguna clase, el lenguaje enviará el mensaje **method_missing** implementado en la clase **Object**. La técnica consiste entonces en alterar la implementación de **method_missing** de modo de generar dinámicamente y en tiempo de ejecución, las versiones transformadas de los métodos 'atómicos'. Para esto, se obtendrá el código fuente del método, se realizarán las transformaciones mencionadas y luego se insertará el nuevo método en la clase del método original y en todas sus subclases (veremos más adelante el porqué de esto último). Todo esto se realizará utilizando las capacidades de metaprogramación provistas por el lenguaje.

Veamos un ejemplo. Sea la siguiente definición de la clase Messenger:

```
class Messenger
  def initilize
    @message = 'hi'
  end
  def say
    puts @message
  end
  def change_message(new_message)
    @message = new_message
  end
end
```

Supongamos que definimos la variable **a_messager** como una instancia de la clase **Messenger** y ejecutamos atómicamente el siguiente bloque:

```
Proc.new {
  a_messager.change_message('bye')
}
```

Al enviarse el mensaje **atomic** al bloque, se realizará la transformación de código del *proc*, obteniendo el siguiente *proc* equivalente:

```
Proc.new {
  a_messager.__atomic__change_message('bye')
}
```

Luego, se procederá a compilar y ejecutar el nuevo *proc*.

Como se ha mencionado, dado que el método **__atomic__change_message** no se encuentra definido en la clase **Messenger**, el intérprete de Ruby enviará el mensaje **method_missing**. El mismo se encargará de obtener el código fuente del método **change_message** y generar la siguiente definición atómica del mismo:

```
def __atomic__change_message(new_message)
    __atomic__instance_variable_set(:@message, new_message)
end
```

Dicho método será primero compilado e insertado en la clase **Messenger** (y sus subclases si las hubiese) y luego ejecutado. Cómo se observa en el código, la ejecución enviará el mensaje **__atomic__instance_variable_set**. El mismo se trata de un mensaje (predefinido en **Object**) que almacenará en el valor de **new_message** en la variable de instancia **@message** de una copia del objeto **a_messager** instanciado al comienzo. De este modo, el cambio al objeto se realiza sobre una memoria paralela, tal como se deseaba.

Análogamente, en caso de realizarse una lectura a una variable de instancia, se procederá del mismo modo. Por ejemplo, en caso de ejecutar atómicamente el bloque

```
Proc.new {
  puts a_messager.say
}
```

el mismo será transformado al siguiente bloque equivalente[1]:

```
Proc.new {
  puts a_messager.__atomic__say
}
```

Una vez que dicho bloque se compile y ejecute, Ruby no encontrará definición para **__atomic__say** y adjuntará a la clase **Messenger** una versión atómica del método original **say** como la siguiente[1]:

```
def __atomic__say
    puts __atomic__instance_variable_get(:@message)
end
```

Al igual que **__atomic__instance_variable_set**, el método **__atomic__instance_variable_get** también se encontrará predefinido en **Object**, realizando una lectura de la variable de instancia **@message** sobre la copia actual del objeto sobre el cual se está trabajando (en el contexto de la transacción actual).

2.3. La fase de *commit*

Cómo se mencionó al comienzo, la fase de *commit* en STM consiste en 2 etapas: Verificación de conflictos y aplicación de cambios a la memoria real (en caso de que no haya conflictos).

La verificación de conflictos consiste en observar los objetos que fueron accedidos durante la transacción (tanto por lectura como por escritura) y verificar que los mismos no hayan sido alterados desde que se hizo el primer acceso hasta el momento del *commit*. En caso de que se haya modificado externamente algún objeto, la transacción hará *rollback* (simplemente NO aplicando los cambios efectuados en la memoria paralela).

La estrategia utilizada para poder implementar este control es sencilla y directa. Al momento de acceder a un objeto, se realizarán 2 copias del objeto original: la copia de trabajo sobre el cual se efectuarán los cambios y una copia adicional local a la transacción que no se modificará. Adicionalmente, se mantendrá una referencia al objeto original. De este modo, la verificación consiste en comparar el estado interno del objeto original con el estado de la copia adicional mencionada, logrando así saber si el objeto en cuestión fue modificado por otro *thread* durante la transacción actual. Si existiese conflicto, basta con lanzar una excepción que interrumpa la operatoria. Otra opción es llamar a un bloque alternativo para continuar la ejecución por otro camino o reintentar la ejecución. Como veremos luego al analizar la implementación, ninguna de estas variantes presenta desafíos particulares.

En caso de que ninguno de los cambios conflictúe, se procederá a aplicar los cambios a los objetos originales. Nuevamente, aquí la operatoria no presenta demasiados inconvenientes en la teoría. La idea será ir copiando los valores de las variables de instancia de la copia de trabajo al objeto original. Para ahorrar tiempo, antes de realizar la copia, podemos verificar fácilmente si el objeto cambio, simplemente comparando la copia de trabajo con la copia original. Veremos más adelante los obstáculos de implementación respecto a la comparación y copia de estado de un objeto hacia otro.

El algoritmo recién descrito en principio no presenta mayores dificultades. Sin embargo, hay un tema importante a considerar: Si se realizaran cambios a los objetos involucrados en la transacción durante la fase de *commit* se podrían generar problemas. Por ejemplo, que un objeto se modifique luego de que se efectúe la verificación de conflictos pero antes de aplicar los cambios. La manera canónica de evitar esto es aplicando un *lock* global antes de comenzar con la operatoria y liberándolo una vez que se apliquen los cambios a la memoria real (o al hacer el *rollback*, en caso de que haya conflictos).

[1] A modo de simplificar el ejemplo, se omitió la transformación del mensaje **puts**, que debiera ser también definida su versión atómica.

Por último, cabe notar que hasta el momento nos hemos referido exclusivamente a cambios de estado de *objetos* pero no a cambios de estado en general. Si bien en teoría en un lenguaje puro de objetos no intervienen otros elementos, la realidad es que todo lenguaje suele proveer otras formas de estado bajo mecanismos de Entrada/Salida. Así entonces, podría ocurrir que se realice una escritura a disco durante una transacción, que debiera ser potencialmente revertida. Este tipo de situaciones está por fuera del objeto de estudio de este trabajo, aunque se hará una mención al tema en la sección Trabajo Futuro.

3. IMPLEMENTACIÓN: DESCRIPCIÓN Y ANÁLISIS

En el presente capítulo ahondaremos en detalle en la implementación realizada, analizando al mismo tiempo las dificultades enfrentadas y las ventajas y/o limitaciones del lenguaje al momento de implementar STM en un lenguaje dinámico orientado a objetos como Ruby.

La implementación realizada requirió alterar el comportamiento de entidades fundamentales del lenguaje, en buena parte propias del meta-modelo de este último. Ruby posee la característica de permitir añadir nuevos métodos a clases ya definidas previamente. Esto nos permitió añadir comportamiento fácilmente sin tener que editar directamente el código fuente de la distribución.

Durante el proceso de investigación y desarrollo de la solución, se fueron implementando casos de *tests* utilizando la librería **rspec** de Ruby. Los mismos no serán explicitados aquí pero se anexan junto con el código fuente que acompaña este documento.

Por otra parte, también durante la investigación, se detectaron algunos errores o inconsistencias de la implementación **Rubinius**. Los mismos serán mencionados y tratados en un capítulo aparte para no quitar el foco en los aspectos interesantes de las soluciones implementadas.

Por último, la implementación a presentar se centrará en las variables de instancia como noción central de estado de los objetos. Sin embargo, es común que existan otras formas de almacenar estado. En Ruby por ejemplo, existen las variables de clase, variables globales y constantes (que a pesar de su nombre, pueden mutar), entre otros. El trabajo no abarca el estudio de estos cambios, aunque se analizará brevemente en el capítulo de Trabajo Futuro acerca de estas variantes.

Abordaremos a continuación todas las extensiones realizadas al lenguaje en el contexto del trabajo, incluyendo las extensiones al *core* así como toda nueva clase definida.

3.1. Extendiendo *Proc*: El punto de entrada

Desde el punto de vista del desarrollador, el proceso se inicia mediante el envío del mensaje **atomic** a un bloque de código, que se desea ejecutar en forma atómica. Por este motivo, la primer clase extendida fue la clase **Proc**. En este contexto, se implementaron los siguientes métodos:

```
class Proc
  def atomic
    MemoryTransaction.do(to_atomic)
  end

  def atomic_if_conflict(a_block)
    MemoryTransaction.do_if_conflict(to_atomic, a_block)
  end

  def atomic_retry
    MemoryTransaction.do_and_retry(to_atomic)
```

```
  end

  def to_atomic
    has_arguments = self.arity != 0
    if has_arguments
      raise 'Cannot atomize a proc with arguments!'
    end
    AtomicProc.from(self)
  end

  def source_code
    ProcSourceCode.new(self)
  end
end
```

Los primeros 3 métodos se tratan de las distintas versiones del mensaje **atomic**, que serán analizadas luego en el contexto del análisis de **MemoryTransaction**.

El método **to_atomic** será el encargado de generar la versión atómica transformada del *proc*. Dado que convertir un bloque con parámetros carece de sentido, validaremos esto primero utilizando el mensaje **arity** implementado en **Proc**, que retorna la aridad del bloque, como su nombre lo indica. Analizaremos **AtomicProc** en una futura sección.

Por último, el mensaje **source_code** será utilizado para obtener el código fuente del mismo. Veremos más adelante cuando se hará uso de este mensaje y cómo se logra obtener el código fuente, cerca del final del trabajo.

3.2. Extendiendo *UnboundMethod*: Código fuente y versión atómica

Además de los bloques de código, los métodos en Ruby deberán ser transformados también atómicamente. Extenderemos entonces la clase **UnboundMethod** con los mismos mensajes recién definidos en **Proc**, cómo se observa a continuación.

```
class UnboundMethod
  def source_code
    UnboundMethodSourceCode.new(self)
  end

  def to_atomic
    AtomicUnboundMethod.from(self)
  end
end
```

Las nuevas responsabilidades añadidas serán proveer el código fuente del método y generar su versión atómica. Como se observa en los métodos **source_code** y **to_atomic**, la representación de estos dos objetos se encuentra implementadas en las clases **UnboundMethodSourceCode** y **AtomicUnboundMethod** que serán analizadas posteriormente.

3.3. El núcleo: Memoria transaccional y registro de cambios

A continuación describiremos los nuevos conceptos centrales a Memoria Transaccional, introducidos a través de nuevas clases.

3.3.1. *MemoryTransaction*

La clase **MemoryTransaction** representa una transacción en memoria, idea fundamental de la implementación. Su responsabilidad principal será la de generar una transacción en memoria a partir de un *proc* que se desea ejecutar encapsulado en una transacción (es decir un bloque que el desarrollador desea ejecutar atómicamente). Esta responsabilidad está implementada en el mensaje **do_if_conflict(atomic_proc, on_conflict_proc)**:

```
def do_if_conflict(atomic_proc, on_conflict_proc)
  Thread.register_current_transaction(self)
  begin
    result = atomic_proc.call
  ensure
    Thread.unregister_current_transaction
  end
  commit_if_conflict(on_conflict_proc)
  result
end
```

Tal como se observa, en primer lugar se procede a registrar la transacción actual para el *thread* que se encuentra ejecutando este método. A continuación, se procede a obtener el resultado de la ejecución del *proc* que se desea ejecutar en el contexto de una transacción, pasado como argumento (*atomic_proc*).

Como se verá luego, *atomic_proc* no es el *proc* original sino que se trata de un *proc* equivalente generado dinámicamente, que se comporta exactamente igual al original, sólo que todos los accesos y escrituras que se hagan a variables de instancia de los objetos involucrados no se harán sobre los objetos reales sino sobre copias que son locales a la transacción que se está ejecutando. Veremos más adelante en detalle cómo se logra esto vía transformación de código.

Todos los cambios realizados se registran en la variable de instancia **@object_changes** de la transacción, un diccionario que contiene una instancia de la clase **ObjectChange** por cada objeto que participa de la misma.

Una vez ejecutado, haya sido o no satisfactoria la ejecución, se quita el registro de la transacción actual. La motivación detrás de esto es evitar que se realicen cambios a la transacción, posteriores a la ejecución del *proc* y durante la fase de *commit*. Eso podría suceder mediante el acceso en forma asincrónica (vía otro *thread* inicializado dentro del *proc* por ejemplo) a los objetos que forman parte de la transacción. En caso de que se lance una excepción durante la ejecución del bloque, también es deseable quitar el registro de la transacción para evitar que esta sea accedida luego globalmente (vía **MemoryTransaction.current**).

Una vez ejecutado el bloque, se procede a intentar realizar el commit de la transacción mediante el mensaje **commit_if_conflict(on_conflict_proc)**:

```
def commit_if_conflict(on_conflict_block)
  MemoryTransaction.commit_lock
  @object_changes.each_value do |change|
    if change.has_conflict?
      MemoryTransaction.commit_unlock
      return on_conflict_block.call(change.original)
    end
  end
  @object_changes.each_value do |change|
    if change.has_changed?
      change.apply
    end
  end
  MemoryTransaction.commit_unlock
  nil
end
```

En primer lugar, se procede a obtener un *lock* global. Este *lock*, al ser global (se trata de una instancia de **Mutex** y se encuentra almacenado como una variable de instancia de clase en **MemoryTransaction**), evitará que otros *threads* alteren la variable de instancia **@object_changes** durante la duración de la fase de commit. Esto es importante ya que, un cambio de contexto durante la ejecución concurrente de varios *threads* podría originar que un *thread* A intente cambiar un objeto durante la transacción mientras que un *thread* B se encuentra verificando posibles conflictos en fase de *commit*.

Una vez adquirido el *lock*, se procede a verificar la existencia de conflictos. En caso de que se encuentre un conflicto (veremos más adelante como se realiza esta verificación mediante el método **has_conflict?** definido en la clase **ObjectChange**), el método retornará, llamando al bloque de código pasado como argumento (previo *unlock*). Al retornar el método, se estará abortando implícitamente la transacción ya que el *commit* no continuará y los cambios no se aplicarán.

En caso de que no existan conflictos, se procederá a aplicar los cambios realizados en las copias de cada objeto hacia los objetos originales. La aplicación se hará vía el método **apply** sólo en caso de que se hayan registrado cambios al objeto. Veremos más detalles a continuación al estudiar la clase **ObjectChange**.

Finalmente, se libera el *lock*.

Una vez realizado el *commit*, el método **do_if_conflict** retorna el resultado del bloque obtenido previamente.

Además de **do_if_conflict**, se implementaron otras 2 variantes: **do**, que simplemente lanzará una excepción al haber conflicto, indicando el objeto original involucrado, y **do_and_retry** que reintentará la ejecución hasta que esta sea satisfactoria:

```
def self.do(atomic_proc)
  self.do_if_conflict(atomic_proc,
    MemoryTransaction.signal_conflict_block)
end
```

```
def self.do_and_retry(atomic_proc)
  self.do_if_conflict(atomic_proc,
    Proc.new { self.do_and_retry(atomic_proc) })
end

def self.signal_conflict_block
  Proc.new do |conflicting_obj|
    raise "CommitConflict: #{conflicting_obj} was changed (...)"
  end
end
```

Transacción actual y cambios del objeto

Adicionalmente a los métodos principales mencionados, se agregaron los siguientes mensajes:

```
def self.current
  unless is_there_a_transaction_running?
    raise 'No current transaction!'
  end
  Thread.current_transaction
end

def self.is_there_a_transaction_running?
  Thread.is_there_a_current_transaction_registered?
end

def change_for(an_object)
  @object_changes[an_object] ||= ObjectChange.new(an_object)
end
```

Se verá en próximas secciones sus contextos de uso, la extensión realizada a la clase **Thread** y la definición de **ObjectChange**.

3.3.2. *ObjectChange*: Registrando cambios

La clase **ObjectChange** representa todo cambio realizado a un objeto durante una transacción. Se presenta su definición a continuación:

```
class ObjectChange
  def initialize(object)
    @original = object
    @working = object.shallow_copy
    @previous = object.shallow_copy
  end
```

```
  def working
    @working
  end

  def original
    @original
  end

  def apply
    @original.copy_object_from(@working)
  end

  def has_conflict?
    not @original.has_same_object_state?(@previous)
  end

  def has_changed?
    not @working.has_same_object_state?(@previous)
  end
end
```

Las instancias de **ObjectChange** se inicializan recibiendo un objeto como parámetro, para el cual se registrarán los cambios. Se guardan 2 copias del objeto y 1 referencia al objeto original:

- **@original:** Se tratará de una referencia al objeto original. Se utilizará para luego aplicar los cambios en la fase de *commit*, así como verificar posibles conflictos.
- **@working:** Se trata de la copia sobre la cual se realizará en cuestión al objeto. Esta copia también se utiliza para verificar si el objeto fue alterado por otra transacción.
- **@previous:** Se trata de una copia que no se modificará durante la transacción, reflejando el estado original del objecto al comienzo. Se utiliza para verificar conflictos y verificar también si el objecto cambió.

Notar que las copias realizadas son de tipo *shallow* ya que se quiere copiar el estado del objeto con sus referencias originales. La copia se realiza mediante un método **shallow_copy** implementado en **Object**, que será presentado más adelante.

La verificación de cambios, conflicto y aplicación de los cambios se efectúan mediante los métodos:

- **has_conflict?** Para saber si existe un conflicto, basta con comparar la copia original del objecto (**@original**), con la copia previa realizada al comienzo de la transacción (**@previous**). Si el original cambió respecto a la copia previa, quiere decir que otro *thread* alteró el objeto original durante la transacción, generando un conflicto.
- **has_changed?** Para saber si el objeto fue modificado, bastante con comparar la copia actual (**@working**) sobre la que se realizan los cambios durante la transacción con la copia previa (**@previous**). Si la copia actual tiene el mismo estado, significa que el objeto no fue alterado por la transacción.

- **apply** Para aplicar los cambios, basta con copiar el estado interno de la copia de trabajo (**@working**) al objeto original (**@original**).

Los mensajes **has_same_object_state?** y **copy_object_from** fueron implementados en **Object** como parte de este trabajo y serán descriptos más adelante.

3.4. Extendiendo *Thread*: Registro de la transacción actual

Se extendió la clase **Thread** para registrar la transacción actual relativa al *thread* que se está ejecutando, como se muestra a continuación:

```
class Thread
  def self.register_current_transaction(a_transaction)
    raise 'A transaction has already been registered (...) !' if
        is_there_a_current_transaction_registered?
    Thread.current[:current_transaction] = a_transaction
  end

  def self.current_transaction
    raise 'No transaction registered!' unless
        is_there_a_current_transaction_registered?
    return Thread.current[:current_transaction]
  end

  def self.is_there_a_current_transaction_registered?
    return !Thread.current[:current_transaction].nil?
  end

  def self.unregister_current_transaction
    raise 'No transaction registered!' unless
        is_there_a_current_transaction_registered?
    Thread.current[:current_transaction] = nil
  end
end
```

Como se puede observar en el cuerpo del método **register_current_transaction**, la transacción actual se almacena mediante el mensaje []= provisto por el lenguaje (análogamente, se recupera mediante el mensaje []). Como indica la documentación oficial, dichos mensajes almacenan variables locales al *thread* (el *thread* actual se obtiene mediante **Thread.current**), de modo que habrá una transacción actual registrada por *thread*. En rigor, las variables en realidad son *fiber-local*[1] en lugar de *thread-local*. Si bien la clase **Thread** provee otro mensaje para almacenamiento *thread-local*, se optó por esta decisión para soportar el uso de transacciones en el contexto de concurrencia vía *fibers*. Notar que

[1] Los *fibers* son un concepto que presenta Ruby (como parte del *core*) para implementar concurrencia cooperativa, similar a los *threads*, difiriendo de este último en que el *scheduling* no es controlado por la VM sino que debe ser realizado por el programador. Más información en https://ruby-doc.org/core-2.3.3/Fiber.html

por cuestiones implementativas, *fiber-local* **implica** *thread-local*, por lo cual las variables también serán locales a los *threads*.

3.5. Extendiendo *Symbol*: Mensajes auxiliares para manejar nombres de métodos

En el lenguaje Ruby los nombres de los métodos son simplemente símbolos, instancias de la clase **Symbol**. Dado que desearemos realizar conversiones entre nombres de métodos para la transformación de código, se extenderá la clase con mensajes auxiliares relativos a esto. Veamos a continuación los más relevantes:

```
def is_an_atomic_method_name?
  self.to_s.start_with? atomic_method_prefix
end

def to_atomic_method_name
  if is_an_atomic_method_name?
    self
  else
    if is_operator?
      non_atomic_name = operator_to_name(self)
    else
      non_atomic_name = self
    end
    (atomic_method_prefix + non_atomic_name.to_s).to_sym
  end
end

def to_nonatomic_method_name
  if is_an_atomic_method_name?
    non_atomic_name =
      self.to_s.sub(atomic_method_prefix, '').to_sym
    if is_an_operator_name?(non_atomic_name)
      name_to_operator(non_atomic_name)
    else
      non_atomic_name
    end
  else
    self
  end
end
```

En primer lugar, **is_an_atomic_method_name?** simplemente verificará que se trata de un nombre de método atómico, verificando que el mismo comience con un prefijo especial.

El método **to_atomic_method_name** será el responsable de convertir un nombre de método a un nombre atómico. Si bien en primer instancia bastaría con prefijar el método

con el prefijo mencionado, como se observa en la definición, existe una diferenciación para el caso de que estemos tratando con un **operador** de Ruby. El motivo detrás de esto es que el lenguaje no permite utilizar caracteres 'operadores' como parte del nombre de un método. Por ejemplo, '__**atomic**__+' no es un nombre de método sintácticamente válido para el *parser* del lenguaje, a pesar de que '+' sí lo es. Esta limitación sintáctica arbitraria se trata de una de las primeras limitaciones de Ruby encontradas. Nótese que se trata de un accidente sintáctico y no de una limitación propia de los lenguajes dinámicos Orientados a Objetos.

La manera de brindar soporte también a los operadores para poder generar la versión atómica de los mismos consistió en realizar un mapeo entre operadores y nombres sintácticamente válidos, mediante el siguiente método:

```
def operators_renaming_map
  {
      :+ => :add,
      :- => :substract,
      :* => :multiply,
      :** => :power,
      :/ => :divide,
      # ...
  }
end
```

La principal limitación de realizar el mapeo resulta evidente: En caso de que una futura versión de Ruby agregue un nuevo operador, se deberá alterar este diccionario. Una manera de solucionar esto (aunque poco elegante) consistiría en realizar un mapeo *dinámico*, donde cada método sintácticamente inválido sea mapeado a una cadena de caracteres aleatoria y única. Mediante metaprogramación se podría realizar la evaluación sintáctica en tiempo de ejecución y verificar esto.

La conversión de nombre de método no atómico a atómico realizada por **to_nonatomic_method_name** se trata simplemente de la operación inversa de **to_atomic_method_name**.

3.6. Extendiendo *Object* (1ra parte): *Method missing* y la copia de trabajo actual

Se añadirán 3 responsabilidades nuevas para **Object**:

1. La definición dinámica de métodos atómicos a través de **method_missing**.
2. El acceso y el reenvío de mensajes hacia la copia de trabajo actual, en conjunto con la definición de la versión atómica de los mensajes **instance_variable_get** e **instance_variable_set**, que permitirán leer y escribir las variables de instancias sobre la copia.
3. La verificación y aplicación de cambios de estado entre objetos, necesario para evaluar conflictos transaccionales y aplicar cambios en los objetos reales. Este último ítem será analizado en una próxima sección, como una segunda parte de la extensión.

3.6.1. Definición dinámica de métodos vía *method missing*

A continuación se presenta la extensión del algoritmo:

```
def method_missing(method_name, *args, &block)
  if method_name.is_an_atomic_method_name?
    define_and_dispatch_atomic_variant(method_name, args, block)
  else
    super
  end
end
```

Como se puede observar, en caso de que el método no definido se trate de un método atómico, se procederá a definir la variante atómica y reenviar el mensaje a este último, mediante **define_and_dispatch_atomic_variant** definido a continuación.

```
def define_and_dispatch_atomic_variant(method_name, args, block)
  original_method_name = method_name.to_nonatomic_method_name
  assert_original_method_is_defined(original_method_name)
  original_unbound_method = method(original_method_name).unbind
  atomic_unbound_method = original_unbound_method.to_atomic
  atomic_method = atomic_unbound_method.bind(self)
  atomic_method.call(*args, &block)
end
```

Los puntos importantes a remarcar son:

- La obtención del objeto que representa el método "suelto" (**UnboundMethod**), mediante los mensajes **method** y **unbind**.
- La generación en sí del método atómico mediante el mensaje **to_atomic** definido en **UnboundMethod** (analizado más adelante al estudiar los cambios implementados en esta última clase).
- El *binding* del método atómico al objeto mediante **bind**.
- La llamada al nuevo método atómico envíando el mensaje **call**.

Antes de analizar estos puntos, notar que la existencia del mensaje **method_missing** es condición **fundamental** para poder realizar esta técnica. El envío de un mensaje particular en caso de que el algoritmo de *method lookup* no halle el método en la jerarquía es una decisión de diseño del lenguaje, implementada a nivel *VM* y por ende no revertible facilmente por un usuario del lenguaje. En este caso, la posibilidad de redefinir el flujo de ejecución mediante la imlementación de un mensaje en tal caso no es sólo una ventaja propia de un lenguaje dinámico robusto sino más bien una condición para lograr la implementación (sin tener recurrir a cambios a nivel VM). El poder contar con este mensaje abre las puertas a la metaprogramación, permitiendo realizar técnicas como las aquí presentadas.

Respecto al primer punto, cabe notar la existencia del mensaje **method** y el mensaje **unbind**, ambos provistos por el lenguaje. **Object#method**, permite obtener el método (**Method**) definido para un objeto a partir de su nombre, mientras que **unbind** retorna la versión "suelta" del método, es decir, aquella no ligada a un objeto en particular. A partir de este último objeto se generará la versión atómica del mismo.

Una vez generado el método atómico (instancia de **AtomicUnboundMethod**), se procederá a volver a unir el método obtenido con el objeto actual que se encuentra enviando el mensaje. Este paso es necesario para poder llamar al nuevo método. Haremos uso de **bind** implementado en **UnboundMethod** para lograr esto. Cabe notar que el método atómico responde **bind** porque la nueva clase **AtomicUnboundMethod** es completamente polifórmica con **UnboundMethod**, como veremos luego.

Por último, se llamará al nuevo método atómico mediante el mensaje **call** (también implementado en **UnboundMethod**), utilizando los parámetros originales.

3.6.2. Accesos a través de la copia de trabajo actual

El siguiente punto a definir serán los métodos alternativos de acceso y escritura de las variables de instancia en la copia de trabajo del objeto en cuestión. Para esto, simplemente basta con definir los métodos **__atomic__instance_variable_get** y **__atomic__instance_variable_set** en **Object**, del siguiente modo:

```
define_method(:instance_variable_set.to_atomic_method_name) do
    |var_name, value|
  working_copy.instance_variable_set(var_name, value)
end

define_method(:instance_variable_get.to_atomic_method_name) do
    |var_name|
  working_copy.instance_variable_get(var_name)
end
```

Notar como nuevamente se hace uso de las capacidades de meta-programación del lenguaje, utilizando el mensaje de clase **define_method** para agregar el método a la clase. Usaremos el mismo en lugar del keyword *def* del lenguaje para poder determinar dinámicamente el nombre del mismo (ya que el prefijo __atomic__ es simplemente una convención definida en **Symbol#atomic_method_prefix**).

Como se observa, los métodos simplemente llamarán a **instance_variable_get** e **instance_variable_set** respectivamente sobre la copia de trabajo actual. Veamos la implementación del mensaje **working_copy**, también definido en **Object**, a continuación:

```
def working_copy
  MemoryTransaction.current.change_for(self).working
end
```

El método buscará la copia de trabajo (mediante **working**) de la instancia de **ObjectChange** retornada por **change_for** en el contexto de la transacción actual (obtenida por el método de clase **current** de **MemoryTransaction**).

Adicionalmente, se definirá el siguiente método para enviar mensajes a través de la copia de trabajo. El mismo será utilizado posteriormente en el contexto de la definición de métodos atómicos.

```
def send_through_working_copy(message, *args, &block)
  working_copy.send(message, *args, &block)
end
```

3.7. Extendiendo *Object* (2da parte): Copia y comparación de objetos

3.7.1. *Shallow copy*

Como se mencionó previamente, para registrar los cambios y poder trabajar una memoria paralela, en primer instancia se necesita poder realizar copias de los objetos.

En principio, Ruby posee los mensajes **clone** y **dup** para efectuar copias de tipo *shallow*. Sin embargo, según la documentación oficial ambos mensajes, luego de realizar la copia, enviarán el mensaje **initialize_copy** que podrá ser implementado por cada clase para realizar acciones posteriores. A diferencia de *Smalltalk*, que posee 2 mensajes separados (*copy* que enviará *postCopy* para acciones adicionales y *shallowCopy* que simplemente realiza la copia)[10, p. 97-99], lamentablemente en Ruby no se incluyó un mensaje que simplemente haga la copia *shallow* sin ejecutar código adicional.

Para obtener el algoritmo de copia, se investigó la implementación en C de **clone** presentada junto a la documentación oficial de MRI. Asimismo, se consultó con la implementación realizada en Ruby que forma parte del *core* de Rubinius.

A continuación se presenta la implementación propuesta:

```
def shallow_copy
  copy = start_new_copy
  copy.copy_object_from(self)

  copy.freeze if frozen?
  copy
end

def start_new_copy
  self.class.allocate
end
```

En primer lugar, a través de la llamada a **start_new_copy**, haremos uso del mensaje **Class#allocate** que permite crear una nueva instancia de una clase sin inicializarla (es decir, sin enviar el mensaje **initialize**). Luego, copiaremos el objeto a partir del objeto original mediante **copy_object_from**, que será presentado a posteriormente.

Por último, por consistencia copiaremos el estado *freeze* del objeto original. Se trata de un *flag* especial que forma parte la estructura de todo objeto en Ruby, propiamente documentado.

Para el caso de **Rubinius**, dado que está implementado sobre Ruby mismo, la implementación del lenguaje está compuesta por otras clases de objetos más primitivos que no

forman parte de la documentación de MRI (implementación oficial). Haciendo un recorrido por las implementaciones de **clone** específicas para otras clases, podemos notar que existen algunas clases que utilizan métodos de alocación de memoria alternativos. Este es el caso por ejemplo de **Tuple** y **ByteArray**. Se trata en ambos casos de clases de objetos de tamaño variable, necesarios para implementar otros objetos cuyo estado sea de un tamaño arbitrario. Para ambos, el mensaje **allocate** lanzará una excepción, ya que se necesita especificar el tamaño del objeto a inicializar. **ByteArray#allocate_sized** y **Tuple#new** son los 2 mensajes alternativos que a hacen las veces de **allocate** para estas clases, recibiendo un valor entero con el tamaño a inicializar.

Por este motivo, se implementó una extensión para estas clases, redefiniendo el mensaje **start_new_copy** como se vé a continuación:

```
module Rubinius
  class ByteArray
    def start_new_copy
      self.class.allocate_sized(self.size)
    end
  end

  class Tuple
    def start_new_copy
      self.class.new(self.size)
    end
  end

  class Proc
    def start_new_copy
      self.class.__allocate__
    end
  end
end
```

3.7.2. Copia de la representación interna de un objeto

En Ruby (y en general en todo lenguaje orientados a objetos con clases), todo objeto a nivel implementación se puede definir como una combinación de un puntero a una clase y a un arreglo de variables de instancia [9, p. 107]. Estos dos elementos conforman la representación interna de todo objeto en Ruby. Considerando esta definición, para copiar un objeto a partir de otro basta con copiar estas dos partes.

Respecto al puntero a la clase, el envío del mensaje **allocate** para instanciar una nueva copia a partir de la clase del objeto original ya garantizará que el nuevo objeto tenga el mismo puntero que el objeto original, ya que es una nueva instancia de la misma clase.

Por consecuencia de lo anterior, una implementación completa para **copy_object_from** sería:

```
def copy_object_from(an_obj)
```

```
  an_obj.instance_variables.each do |ivar_name|
    self.instance_variable_set(ivar_name,
           an_obj.instance_variable_get(ivar_name))
  end
end
```

Como se observa, simplemente se recorrerá la colección de variables de instancia (accesibles vía **instance_variables**) y se copiaran los valores vía **instancia_variable_set** e **instancia_variable_get**. Esta implementación es muy similar a la utilizada en la implementación sobre Smalltalk.

En primera instancia, la implementación presentada bastaría. Sin embargo, ocurre que **no todos los objetos almacenan su estado en variables de instancia**. Los objetos más primitivos del lenguaje, que forman parte del *core* del lenguaje como **String** y **Array**, se encuentran implementados a bajo nivel, almacenando sus datos a nivel de *VM*. Al enviar el mensaje **instance_variables** a estos objetos, se puede corroborar que se retorna un arreglo vacío en ambas implementaciones estudiadas.

Notar que sólamente interesa analizar aquellos objetos primitivos que sean **mutables**. Sin embargo, aún con esta restricción, no existe una manera confiable de saber si se trata realmente de objetos primitivos mutables. Si bien las capacidades reflexivas nos permiten saber si hay variables de instancia definidas, no podremos saber si se trata de un objeto sin variables de instancia definido por el usuario o de uno primitivo mutable que almacena su estado a nivel *VM*.

El lenguaje cuenta también con un método denominado **frozen?** que informa si un objeto fue marcado como inmutable. Si bien podría ser útil potencialmente, se trata de una condición suficiente de inmutabilidad pero no necesaria, por lo que no servirá para poder separar a todo objeto entre mutable e inmutable.

Otra opción analizada consistía en hacer uso de la convención de métodos de Ruby, que impone al desarrollador nombrar los métodos que modifican el estado de un objeto con el sufijo **!**. Sin embargo, no se trata más que de una convención, por lo cuál no habrá garantía de que todo objeto mutable vaya a contener un método terminado en **!** y viceversa.

Dado este escenario, no queda otra opción que analizar manualmente el código fuente de la implementación para identificar que clases de objetos son mutables y definir el mensaje **copy_object_from** para cada caso. Aquí es donde nos encontramos con la primera gran limitación de la implementación oficial MRI: Todas las clases que forman parte del *core* de la implementación (es decir todo menos las pertenecientes a la librería *stdlib*) se encuentran implementadas en lenguaje C, por lo que no será posible copiar o modificar el estado de estos objetos en lenguaje Ruby, sino que deberá modificarse directamente la implementación en C.

Como alternativa a esta problemática, en algunos casos es posible definir el mensaje **copy_object_from** utilizando otros mensajes públicos específicos de cada clase, que compartan esta semántica. Por ejemplo, las clases **Array**, **String** y **Hash** comparten el protocolo **replace**, que se encargará de reemplazar el "contenido"de instancias de estos objetos por otra, por lo que resulta un buen candidato para esta tarea.

Si bien las mencionadas son heurísticas funcionales, la única manera de lograr copiar de forma totalmente segura fiable para todos los objetos sería implementar una versión en C de este mensaje para la clase en **Object**. Quedará como trabajo futuro investigar cómo implementarlo para MRI.

Para el caso de Rubinius, prácticamente toda la librería *core* se encuentra implementada en Ruby por lo que los objetos variables más primitivos sí tendrán variables de instancia. Sin embargo, los objetos más primitivos de todos, como los mencionados **ByteArray** y **Tuple** que se usan como estructuras fundamentales, son mutables pero NO contienen variables de instancia. Si bien en principio los objetos fundacionales podrían ser objetos inmutables (como los números enteros), en el caso que querer contar con estructuras de tamaño variable será necesario alguna estrategia especial. Explicitar variables de instancia no es una solución, dado que el número no se conoce de antemano. Si bien implementar estas clases con mensajes primitivos es una solución, lamentablemente no existe forma de identificar cuales son estas clases. Por el contrario, en Smalltalk existe el mensaje **isVariable** que permite distinguir si una clase es de tamaño variable (es decir, con una cantidad variable de colaboradores). Para estos casos, cuenta además con los mensajes **at:** y **at: put:** que permiten acceder al estado de estas clases de tamaño variable. En Rubinius, al no existir un modo uniforme de acceso para este tipo de clases (como sí ocurre para las de tamaño fijo vía **instance_variables**, **instance_variable_get** e **instance_variable_set**) deberíamos realizar una implementación particular para cada clase. Afortunadamente, dado que la implementación se encuentra en Ruby, podremos inspeccionar como **clone** resuelve esta problemática. En el código fuente, se puede observar que para realizar la copia se invoca a una primitiva llamada *:object_copy_object*. La misma se encuentra implementada en C++ y permitirá copiar todos los objetos, sin importar la representación de su estado. Por lo tanto, redefiniremos **copy_object_from** del siguiente modo para Rubinius:

```
def copy_object_from(an_obj)
  Rubinius.invoke_primitive :object_copy_object, self, an_obj
end
```

De este modo, vemos que en Rubinius fue posible realizar esta implementación sin necesidad de modificar la *VM*.

El problema de las clases *singleton*

Aún resuelta la copia del estado de los objetos a nivel VM, existe otra problemática. El metamodelo de Ruby tiene una característica particular que añade una complejidad adicional. El lenguaje introduce la idea de clases ***singleton***. Se trata de clases especiales que permiten agregar comportamiento a un objeto en particular, que lo distinguirá del resto de las instancias de la misma clase. Este mismo concepto es utilizado para implementar mensajes de clase. Estos últimos de implementan como métodos dentro de la clase *singleton* de la clase en cuestión. Esta clase funciona como metaclase de la primera.

Al igual que ocurre con las metaclases en Smalltalk, las clases *singleton* son creadas por Ruby automáticamente al momento de crear una nueva clase. Para el resto de los objetos, las clases *singleton* se crean dinámicamente al momento de añadir un método a un objeto. A nivel implementativo, se introduce dinámicamente dentro de la jerarquía de clases. El puntero de clase de un objeto pasa a apuntar a la nueva clase *singleton*, y esta última pasa a tener a la clase original como superclase. De este modo, el algoritmo de *method lookup* no sufrirá modificaciones. Sin embargo, para el desarrollador, al enviar el mensaje **class**, la clase de un objeto seguirá siendo la misma, disponga o no de una clase *singleton* asociada. Esta última será visible mediante el envío del mensaje **singleton_class**.

La implicancia que tendrá este modelo en nuestra implementación se puede observar con claridad al analizar qué ocurre al copiar una clase mediante **shallow_copy**. Al enviar el mensaje **start_new_copy**, se creará una nueva instancia de la clase original. En el caso de una clase, se tratará de una instancia de **Class**. Una vez copiada las variables de instancia mediante **copy_object_from** la clase copiada tendrá el mismo estado interno que la original. Sin embargo, no podrá responder a los mensajes de clase que tenía la clase original. El motivo de esto es que los mismos se encuentran definidos en la *singleton class* y no en la clase **Class**, de donde provino la copia. Análogamente, lo mismo ocurriría con un objeto que no sea una clase, y tenga una clase *singleton* asociada con un método particular.

La primera solución inmediata que surge a este problema es enviar el mensaje **allocate** (dentro de **start_new_copy**) siempre a la clase *singleton* en lugar de a la clase original. Sin embargo, esto lanzará una excepción: No es posible crear instancias de una clase *singleton*. Conceptualmente, esto tiene sentido ya que una clase *singleton* es simplemente una clase asociada a un único objeto específico por lo que crear múltiples instancias carece de sentido.

Entonces, ¿cómo es que el mensaje **clone** provisto por el lenguaje no presenta este problema? Vemos la implementación de este mensaje en Rubinius:

```
def clone
  copy = Rubinius::Type.object_class(self).allocate

  Rubinius.invoke_primitive :object_copy_object, copy, self
  Rubinius.invoke_primitive :object_copy_singleton_class, copy, self

  Rubinius::Type.object_initialize_clone self, copy

  copy.freeze if frozen?
  copy
end
```

Notar que luego de invocar a la primitiva *:object_copy_object*, se invocará a una primitiva llamada *:object_copy_singleton_class*. Esta misma, como su nombre sugiere, será la encargada de copiar también la clase *singleton* original a la copia.

Renombraremos entonces el mensaje **copy_object_from** como **copy_object_state** y definiremos **copy_object_from** como:

```
def copy_object_from(an_obj)
  copy_object_state(an_obj)
  copy_singleton_class(an_obj)
end
```

Para implementar **copy_singleton_class** en Rubinius, simplemente podemos invocar a la primitiva utilizada en **clone** para realizar la copia. En el caso de MRI, un primer enfoque sería clonar la clase *singleton*. Sin embargo, el intérprete no permitirá clonar la clase ya que las clases *singleton* deberán ser únicas como se mencionó (son, en efecto, *singletons*).

Una alternativa consiste en adjuntar las partes de la clase *singleton* original a la copia. Una clase *singleton*, como toda clase en Ruby, está compuesta por un diccionario de métodos, nombres de las variables de instancia y un diccionario de constantes [9, p. 124].

El problema que surge al intentar adjuntar métodos de la clase *singleton* original es que Ruby no permite anexar métodos de una clase en otra clase que no pertenece a la misma jerarquía. Un posible *workaround* consiste en convertir el método a un *proc* y luego adjuntarlo. Sin embargo, se puede mostrar con un simple caso de *test* que aquellos métodos que modifiquen el estado del objeto van a modificar el estado del objeto original, en lugar de la copia.

Aquí nos encontramos con una limitación de expresividad del lenguaje que no permite realizar esta operación sin tener que recurrir a alterar la VM original en C. Realizar esta copia no parece presentar dificultades mayores y quedará como trabajo futuro resolver esta problemática en MRI.

3.7.3. Verificación de cambios de estado de las copias

Para la fase de *commit* se necesita también poder comparar el estado de 2 objetos (un objeto y una copia de este). En este caso (al igual que Smalltalk), el lenguaje Ruby tampoco posee una implementación para esta tarea por lo cual se realizó la implementación presentada a continuación.

```
def has_same_object_state?(an_obj)
  instance_variables.all? do |ivar_name|
    ivar_self = self.instance_variable_get(ivar_name)
    ivar_an_obj = an_obj.instance_variable_get(ivar_name)
    ivar_self.equal?(ivar_an_obj)
  end
end
```

La forma estándar de comparar el estado de 2 objetos consiste en ver si los objetos a los que referencian ambos son iguales (mediante la comparación de sus variables de instancia). Se utilizará el mensaje **equal?** provisto por el lenguaje, que se trata de igualdad a nivel de posición de memoria, es decir, indicará si 2 referencias son efectivamente el mismo objeto. El mismo se encuentra implementado en **BasicObject** (raíz) y su documentación exige que no sea sobrecargado en las subclases (al contrario de lo que ocurre con el mensaje ==).

Al igual que ocurrió con la copia, aquellos objetos primitivos mutables que no cuenten con variables de instancia retornaran siempre *true*, a pesar de que quizás si hayan mutado. Nuevamente, para el caso de MRI, no será posible hacer esto en Ruby para estos casos, aunque sí podremos utilizar también mensajes publicos de cada clase, que comparan la semántica de comparación. Por ejemplo, para la clase **Array**, la implementación de == está documentada como una comparación de cantidad de elementos e igualdad (==) de cada uno de ellos. Algo similar ocurre con **Hash#==** y **String#eql?**, donde semánticamente dichos mensajes implican la comparación de estado entre 2 instancias del tipo. Para estas clases, dichos mensajes podrían servir como implementación para el mensaje en cuestión.

Para el caso de Rubinius, lamentablemente no existe una primitiva para realizar tal tarea como sí ocurre con la copia. Sin embargo, al realizar una investigación sobre el código fuente, se puede concluir que **ByteArray** y **Tuple** son las únicas primitivas mutables cuyo estado no se encuentra almacenado en variables de instancia. Inspeccionando la implementación de ambas clases, se optó por realizar las siguientes extensiones:

```
module Rubinius
  class ByteArray
    def has_same_object_state?(other_bytearray)
      other_bytearray.size.times.all? do |i|
        get_byte(i).equal?(other_bytearray.get_byte(i))
      end
    end
  end
end
```

```
module Rubinius
  class Tuple
    def has_same_object_state?(other_tuple)
      other_tuple.length.times.all? do |i|
        self[i].equal?(other_tuple[i])
      end
    end
  end
end
```

En el caso de **ByteArray**, haremos uso de los mensajes **get_byte** y **size** (estos, a su vez, invocarán a una primitiva de igual nombre) para obtener el estado interno del objeto. Análogamente, recurriremos a los mensajes **length** y [] para el caso de **Tuple** (nuevamente, estos invocarán a una primitiva).

Variables de instancias en Rubinius

Respecto a **Rubinius**, al recorrer el código fuente de la implementación, podemos observar que las clases primitivas como **String** o **Array** cuya estructura de datos se encuentra almacenada en C++, sí poseen variables de instancia. Sin embargo, al enviar el mensaje **instance_variables** veremos que se retorna un arreglo vacío, al igual que en MRI. Por una decisión técnica del equipo de desarrollo, este mensaje no retornará aquellas variables de instancia de objetos *core* del lenguaje que se correspondan con campos implementados en C++ en la capa bajo nivel de la VM. Curiosamente, el mensaje **instance_variable_get** sí retorna su valor al enviar el mensaje con cada una de las variables de instancia como parámetro, que se observan en el código fuente de cada clase implementada.

Por suerte, la implementación provee una clase **Mirror** que permite obtener estas variables mediante el mensaje **instance_fields**. Utilizando esta facilidad, se implementó el mensaje **all_instance_variables** que retornará estas variables de instancia "ocultas", además de las variables de instancia convencionales:

```
def all_instance_variables
  reflection_api = Rubinius::Mirror.reflect(self)
  instance_variables = reflection_api.instance_variables
  all_instance_variables = Set.new(instance_variables)
  all_instance_variables.merge(reflection_api.instance_fields)
end
```

Este método simplemente calculará la unión entre el conjunto de variables de instancias normalmente visible y el conjunto de variables ocultas (*instance fields*). Utilizando este nuevo mensaje, simplemente bastaría con reescribir el mensaje **has_same_object_state?** utilizando **all_instance_variables** en lugar de **instance_variables**.

A modo de conclusión, se puede decir que la implementación basada mayormente en Ruby mismo presenta notables ventajas al momento de realizar este tipo de operaciones primitivas. En MRI, por contraposición, resulta necesario realizar una implementación directamente en lenguaje C. Sin embargo, la implementación de objetos de tamaño variable en Rubinius no resulta tan transparente como podría, recurriendo al uso de primitivas para acceder al estado de dichos objetos. Por el contrario, la implementación de STM en *Smalltalk*, al contar con el concepto mencionado de clases de tamaño variable, logra implementar fácilmente los mensajes de verificación y copia (**isIdenticalToSnapshot** y **restoreSnapshot**, respectivamente), sin necesidad de recurrir a primitivas[3].

3.8. Extendiendo *Module / Class* (1ra parte)

En Ruby, la clase **Module** es la encargada de aquellas responsabilidades compartidas por los módulos y clases del lenguaje. Analizaremos a continuación algunas nuevas responsabilidades implementadas, que servirán para complementar la implementación de la generación de métodos "atómicos" (aquellos que se ejecutan correctamente dentro de una transacción), analizada posteriormente.

3.8.1. Definición de métodos mediante código fuente

Para lograr insertar métodos atómicos a una clase, se necesitará en general realizar esta operación a partir del código fuente del método. El motivo detrás de esto es que se realizarán transformaciones a partir del código fuente del método original, tal como se mencionó al comienzo del trabajo.

Ruby no permite crear un objeto de tipo **UnboundMethod** a partir de código fuente (ni tampoco a partir de *bytecode*), aunque cuanta al menos con facilidades para evaluar código en el contexto de una clase/módulo. Se optó entonces por añadir un mensaje llamado **define_method_using_source_code** a la clase **Module** haciendo uso de esta facilidad:

[3] **restoreSnapshot** se encuentra implementado mediante una primitiva pero por temas de eficiencia, ya que se trata de la misma primitiva utilizada en **Object#copyFrom**, mensaje implementado también vía código Smalltalk.

```
def define_method_using_source_code(name, meth_source_code)
  if name != meth_source_code.name_in_definition
    meth_source_code.change_name_in_definition!(name)
  end
  if RUBY_ENGINE == 'rbx'
    require_relative 'rbx/module'
    class_eval_with_kernel_code_support(meth_source_code.to_s)
  else
    class_eval(meth_source_code.to_s)
  end
end
```

Por consistencia con **define_method** (ya presente en la implementación original del lenguaje) el primer parámetro será el nombre del método a insertar. Dado que la definición de un método ya incluye su nombre, en caso de ser distinto al pasado como parámetros, se alterará la definición mediante **change_name_in_definition!** (implementado en **UnboundMethodSourceCode**) de modo de insertar el método con el nombre correcto.

Para definir el método en cuestión, se utilizará el método **class_eval** provisto por el lenguaje. Este se encargará de evaluar el código fuente del método en el contexto de la clase actual, logrando de este modo insertar dinámicamente el método en la clase, del mismo modo que si fuese definido dentro de la clase manualmente por el desarrollador.

En el caso de la implementación **Rubinius**, surge un inconveniente al momento de utilizar el método **class_eval** para esta tarea. Los métodos más primitivos del lenguaje que pertenecen al *kernel* de Rubinius, también implementados en Ruby, sufren en algunos casos transformaciones de código realizadas por el compilador, donde el código es compilado utilizando un *bytecode* especial en caso de encontrar ciertos "métodos" en el código fuente. Estos métodos, que no están definidos como parte del lenguaje sino que se tratan de un metalenguaje utilizado como señuelo para el compilador, terminarán en *method missing* en caso de que sean compilados y ejecutados normalmente. El compilador utilizado en la implementación de **class_eval** no realiza por defecto estas transformaciones, ya que las mismas estan pensadas para ser aplicadas sólo al *kernel* del lenguaje y no a código escrito por el usuario. Sin embargo, dado que el compilador también se encuentra escrito en Ruby, se realizó una implementación alternativa de **class_eval** llamada **class_eval_with_kernel_code_support** (tal como se observa en el código mostrado anteriormente) que se encargará de usar un compilador alternativo que habilitará el soporte para las transformaciones de código del *kernel*. De este modo, el código de los métodos atómicos transformados será insertado en la clase en cuestión correctamente, con las transformaciones del compilador aplicadas para aquellos métodos cuya versión original forma parte del *kernel* de Rubinius.

3.8.2. Registro y remoción de métodos atómicos

Se extenderá **Module** con un mensaje de clase llamado **register_module_with_an_atomic_method** que permitirá registrar aquellos módulos que contengan métodos atómicos. Veamoe el código fuente vemos a continuación:

```
def self.register_module_with_an_atomic_method(a_module)
  @@modules_with_atomic_methods.add(a_module)
end
```

Tal como ser observa, el registro consiste añadir al módulo a una conjunto de módulos almacenados en la variable de clase[2] **@@modules_with_atomic_methods**. El objetivo de registrar aquellos módulos que tienen métodos atómicos consiste en minimizar drásticamente la búsqueda de métodos atómicos al momento de querer hallar todos los métodos insertados en el sistema hasta el mismo. Esto servirá, por ejemplo, para poder remover fácilmente todos los métodos atómicos insertados. Para esta última tarea, se implementaron los siguientes métodos:

```
def self.remove_all_atomic_methods
  @@modules_with_atomic_methods.each do |a_module|
    a_module.remove_atomic_methods
    @@modules_with_atomic_methods.delete(a_module)
  end
end

def remove_atomic_methods
  instance_methods(include_super=false)
      .find_all { |method| method.is_an_atomic_method_name? }
      .each { |method| remove_method(method) }
end
```

El método principal **remove_all_atomic_methods** se trata de un mensaje de clase que simplemente recorrerá la variable de clase mencionada y enviará el mensaje **remove_atomic_methods**, que eliminará todos los métodos atómicos definidos en el módulo (instancias de **Module**). Es interesante observar como las capacidades introspectivas del lenguaje, manifestada en la presencia de los métodos **instance_methods** y **remove_method** hacen posible esta tarea de manera sencilla.

3.8.3. Obtención de métodos alias

En Ruby, un mismo método puede tener distintos nombres conocidos como **alias**. La tercera extensión realizada a **Module** consiste en un mensaje que permitirá obtener todos los **aliases** definidos en una clase o módulo para un método en particular. Este mensaje es utilizado al definir los métodos atómicos, ya que deseamos crear un método atómico alias del método original por cada alias que tenga este último.

Se presenta el código fuente a continuación:

```
def aliased_methods(method)
  instance_methods(false).collect do |other_meth_name|
```

[2] Se utilizó una variable de clase en lugar de una variable de instancia de clase, ya que es necesario compartir dichos valores con la clase hija **Class**, dado que se registrarán tanto módulos como clases.

```
    instance_method(other_meth_name) == method &&
        other_meth_name != method.name
  end
end
```

Para obtener los métodos alias, basta con recorrer todos los métodos de instancia y buscar aquellos que sean iguales al método pasado como parámetro. Según la documentación oficial, el mensaje == responde verdadero para aquellos métodos que tengan la misma definición. De este modo, los alias serán aquellos con misma definición pero distinto nombre.

Si bien esta característica particular del lenguaje aumentó la complejidad de la implementación, la presencia de los mensajes **instance_methods** e **instance_method** en **Module** así como **name** de **UnboundMethod** fueron fundamentales para resolver este problema facilmente.

3.8.4. *Class*: Subclases que implementan un mensaje en particular

Por último, la última extensión auxiliar consiste en implementar un mensaje llamado **subclasses_implementing_message** en **Class**, la clase de las clases en Ruby. El mismo retornará todos los métodos de un mensaje pasado como parámetro para todas las subclases de la clase en cuestión. La necesidad de contar con este mensaje radica en un problema accidental de la inserción de método atómicos, que será más adelante explicado como parte de la implementación de **AtomicUnboundMethod**.

Dado que el concepto de subclase sólo está definido para clases (y no módulos), la extensión se realizó sobre la clase **Class**.

Antes de implementar el mensaje, notemos primero que Ruby **no** posee una implementación para obtener todas las subclases de una clase y menos aún aquellas que implementan un mensaje en particular, aunque si provee facilidades para poder realizar un algoritmo simple al momento de implementar la funcionalidad.

En primer lugar, se implementó el mensaje **all_subclasses** como se vé a continuación:

```
def all_subclasses
  metaclass = self.singleton_class
  subclasses_including_self = ObjectSpace.each_object(metaclass)
  subclasses_including_self.reject { |klass| klass == self }
end
```

Para poder obtener todas las subclases de la clase recorreremos primero el espacio de todos los objetos vivos en el ambiente, concepto reificado en Ruby como **ObjectSpace**, mediante el mensaje **each_object**. Este mensaje permite opcionalmente filtrar aquellos objetos que sean instancias de una jerarquía de clases pasando el nodo raíz del árbol de clases como parámetro. Dado que nos interesan aquellos objetos que sean subclases de la clase actual, pasaremos como parámetro de **each_object** a la metaclase de la clase actual. De este modo, las instancias obtenidas serán todas las instancias del árbol de metaclases, es decir, todas las subclases de la clase, incluyendo a ella misma. Por último, filtraremos la clase en cuestión, ya que nos interesan las subclases únicamente.

Mediante la ayuda de este método, podremos obtener fácilmente aquellas subclases que implementan el mensaje en cuestión, tal como se observa a continuación:

```
def subclasses_implementing_message(message)
  self.all_subclasses.select do |klass|
    klass.instance_methods(false).include?(message)
  end
end
```

Una vez obtenidas las subclases, se aplicará un filtro para seleccionar sólo las subclases que tengan el método de instancia en cuestión (nuevamente, mediante una facilidad reflexiva del lenguaje a través del método **instance_methods**).

3.9. Extendiendo *Module / Class* (2da parte): Clasificando primitivas

En toda implementación de un lenguaje de alto nivel existen partes de la misma que necesitan implementarse en un lenguaje de más bajo nivel. Comúnmente, se suele llamar primitivas del lenguaje a estas piezas. En otros casos, el uso de una primitiva no es estrictamente necesario pero igualmente se realiza por temas de *performance*.

En el caso de MRI, los componentes *core* del lenguaje (es decir las clases más fundacionales, que no incluyen a la librería estándar *stdlib*) se encuentran implementados por completo en lenguaje C. Para el caso de Rubinius, la situación es bastante distinta. El lenguaje se encuentra "íntegramente" implementado sobre Ruby mismo, y para evaluar las primitivas utiliza 2 macros embebidas en el código fuente: ***Rubinius.primitive*** y ***Rubinius.invoke_primitive***. El intérprete luego se encargará de convertir dichas macros en un *byte-code* alternativo al momento parsear y compilar el código fuente. Es decir, en lugar de interpretar dicha cadena de caracteres como un envío de mensaje hacia el objeto Rubinius, generará un *byte-code* especial que ejecutará código implementado en C++ como parte de la VM.

El problema central que presentan las primitivas es que su código no es transformable. Esto resulta una barrera importante para la técnica aquí presentada.

Un búsqueda rápida por el código fuente de la versión 3.69 de Rubinius, muestra alrededor de 550 mensajes implementados utilizando primitivas. Por el lado de MRI, la situación es aún más crítica: Son 123 las clases de objetos que forman parte de la documentación oficial implementadas en lenguaje C. Lamentablemente, en este aspecto el lenguaje no presenta capacidades reflexivas que permitan obtener características sobre estas primitivas. Las existencia de las mismas es transversal al meta-modelo, para ambas implementaciones.

Dada la magnitud de código que posiblemente deba ser re-implementado, se decidió atacar este problema analizando las características de estos métodos para ambas implementaciones, de modo de intentar hallar grupos para los cuales se pueda dar un tratamiento especial.

En primer lugar, se realizó una recorrida superficial (aunque exhaustiva) por todas las clases que componen la implementación Rubinius, de modo intentar encontrar patrones y entender en que contextos se utilizan las primitivas. Una primer conclusión obtenida es que el uso de ***Rubinius.invoke_primitive*** se diferencia de ***Rubinius.primitive***. En el caso del primero, se utiliza para invocar primitivas que retornen valores pero que no detengan

el flujo de ejecución; se trata como si fuese un mensaje más, aunque implementado en C++. Su uso está bastante menos extendido que el de ***Rubinius.primitive*** y en general está limitado a ciertos usos particulares como las primitivas de copia de objetos. Sin embargo, lamentablemente se utiliza también en algunos contextos donde se podría utilizar ***Rubinius.primitive***, lo que dificulta contemplarlo como un caso aparte.

La conclusión más importante obtenida es que no se halló un patrón bien definido que permita clasificar en forma clara a las primitivas, para ninguna de las dos implementaciones. Lamentablemente tampoco existe documentación asociada a ninguna de las implementaciones acerca de cuales son los objetos primitivos ni características que permitan encontrar una solución automatizada para todos los casos.

No obstante, se propone a continuación una clasificación que permitirá tratar algunos grupos de primitivas de forma particular, que evitarán tener que re-implementar nuevamente estos mensajes en Ruby. Asimismo, resulta una base de conocimiento para intentar entender problemas y soluciones asociadas a las primitivas en el contexto de cualquier lenguaje orientado a objetos.

El objeto de estudio serán las clases del lenguaje. Se propone particionar el conjunto de clases en los siguientes subconjuntos (disjuntos):

- **Clases inmutables:** Denominaremos como clase inmutable a aquellas clases cuyas instancias serán siempre inmutables. A pesar de ser un lenguaje altamente dinámico, Ruby posee una forma de garantizar inmutabilidad para ciertas clases. Identificar cuales son evitará la necesidad de transformar los métodos primitivos (y no primitivos) asociados a estas instancias, ya que los dichos objetos no serán nunca modificados durante la ejecución.

- **Clases mutables:** Las clases para las cuales no podremos garantizar la inmutabilidad entran en este conjunto. A su vez, podemos identificar 2 grupos característicos:

  - **Con colaboradores internos:** Se trata del caso general al implementar un objeto mutable. En general, los mensajes de estas clases deberán ser transformados normalmente. Sin embargo, en el caso de Rubinius existen algunas clases de objetos que invocan a primitivas pero al mismo tiempo definen variables de instancia y tienen otros mensajes que se ejecutan normalmente en código Ruby. Veremos luego en detalle qué mensajes entran en esta categoría y que hacer con ellos.
  - **Sin colaboradores internos:** En Ruby, los objetos mutables más fundamentales no almacenan su estado a través de variables de instancia, sino que se utilizan macros primitivas para leer y escribirlos, ya que los mismos se encuentran implementados a nivel VM. Las mensajes de las instancias de estas clases no serán necesario transformalos sino que bastará con reenviar el mensaje hacia una copia del objeto, como veremos luego.

Una problemática que tiene este clasificación es que está centrada en el concepto de clases. Ruby, por otra parte, define además el concepto de módulo. Este último no se puede instanciar[9, p. 135] y su uso principal es ser utilizado como un conjunto de mensajes anexados a otra clase (idea conocida como *mixin*). En este sentido, se puede pensar a las clases del lenguaje como la suma de sus mensajes para los mensajes de los módulos incluidos por esta última, por lo que no sería relevante analizar los módulos en forma aislada. Sin embargo, los módulos también pueden ser utilizados en forma autónoma, a través

de mensajes de clase e incluso definir colaboradores internos. Si bien estos serán objetos *singletons* (ya que los módulos no son instanciables), pueden interactuar potencialmente con colaboradores internos y por lo tanto alterar su estado en un contexto transaccional. Quedará como trabajo futuro analizar cómo abordar aquellos módulos que se encuentran implementados en forma primitiva y que potencialmente deberían ser transformados. En la práctica, la investigación arrojó que son pocos módulos que no se implementan como *mixins* y forman parte de la distribución del lenguaje. En MRI estos son: GC, GC::Profiler, Marshal, Math, Process, Process::GID, Process::Sys, Process::UID, Signal, ObjectSpace, FileTest, Random::Formatter, Errno, File::Constants.

Por último, además de esta clasificación, es útil analizar los siguientes grupos de primitivas (ortogonales a la clasificación mencionada):

- **Primitivas de infraestructura:** Se trata de métodos primitivos que forman parte de piezas estructurales del lenguaje. En muchos casos se trata de mensajes fundacionales imposibles de implementar fuera de la VM. En general estos métodos no deseamos transformarlos por lo que identificarlos ayudará a acortar la cantidad de primitivas a reescribir.
- **Primitivas opcionales (Rubinius):** Al igual que ocurren en Smalltalk, varias primitivas en Rubinius disponen de su código fuente alternativo justo debajo de la llamada primitiva. Identificar estos mensajes permitirá poder realizar su transformación sin necesidad de reescribirlos.

Analizaremos a continuación los grupos mencionados y su impacto en la implementación.

3.9.1. Clases inmutables

Debido a la naturaleza de los lenguajes dinámicos, en principio resulta imposible clasificar a una clase de objetos como inmutables. Dado que las clases son alterables para el usuario del lenguaje, se podrían implementar métodos que alteren el estado de objetos que originalmente eran inmutables. Sin embargo, Ruby posee un mensaje denominado **freeze** que permite "congelar" un objeto, haciéndolo efectivamente inmutable. Una particularidad de esto es que la acción no puede ser revertida. Más aún, la documentación de este mensaje afirma que todas las instancias de las siguientes clases están siempre "congeladas": **Fixnum**, **Bignum**, **Float**, **Symbol**.

Por otra parte, las clases **TrueClass**, **FalseClass** y **NilClass** son clases *singleton* que no permiten generar nuevas instancias. Las mismas poseen una sola instancia global y se puede verificar que la misma está congelada, mediante el envío del mensaje **frozen?** a los objetos *true*, *false* y *nil*.

Cómo conclusión, podemos afirmar que todas las instancias de **Fixnum**, **Bignum**, **Float**, **Symbol**, **TrueClass**, **FalseClass** y **NilClass** siempre serán inmutables.

Este conocimiento permitirá luego evitar realizar transformación de código para mensajes que respondan instancias de las clases mencionadas. Incluso resulta necesario evitar estas transformaciones ya que tanto la implementación oficial como Rubinius impiden la alocación de recursos para realizar copias de estos objetos. Por lo tanto, la implementación de **shallow_copy** presentada anteriormente no funcionará.

Para lograr identificar estas clases, se implementó el siguiente mensaje en la clase **Class**:

```
def immutable_instances?
  false
end
```

Dado que por defecto las instancias de una clase no son inmutables, el mensaje retornará *false*. El mismo mensaje fue sobre escrito para las clases mencionadas. Por ejemplo:

```
class Bignum
  def self.immutable_instances?
    true
  end
end
```

Veremos luego cómo haremos uso de este mensaje para reconocer las clases cuyos métodos no deben ser transformados, al analizar la clase **AtomicUnboundMethod** en una próxima sección.

3.9.2. Clases mutables

En general, toda clase mutable tendrá colaboradores internos, ya que para que tenga sentido una mutación de estado, debe haber un colaborador que mute. El enfoque para estas clases, como se mencionó al comienzo del trabajo, consistirá en transformar los métodos de modo que los accesos a memoria se hagan en memoria paralela. Además, los mensajes a colaboradores internos deberán a su vez ser transformados.

En el caso de MRI, las clases o bien se encuentran implementadas en Ruby o íntegramente como una estructura de C, por lo cual no existen objetos que invoquen a primitivas y a la vez tengan colaboradores internos. Por el contrario, en Rubinius sí nos encontramos con objetos primitivos implementados en Ruby con colaboradores internos, pero que a la vez invocan a primitivas. Estos objetos también se encuentran implementados como una clase en C++. La VM se encargará de mapear los campos de C++ con las variables de instancia correspondientes.

Lamentablemente no existe una manera sencilla de identificar estas clases de objetos mutables con variables de instancia y primitivas. En un análisis preliminar realizado sobre el código fuente de Rubinius se identificó a **String**, **Time** y **Regexp** como clases documentadas, con colaboradores y a su vez con llamadas a primitivas. La principal complejidad no radica en hallar estas clases sino en saber qué hacen las primitivas. Para ello, habrá que estudiar el código fuente C++ de ellas y determinar si envían mensajes a los colaboradores. En este último caso, no quedará alternativa que re-implementarlos en Ruby. Se podría realizar un análisis automatizado mediante heurísticas basadas en sintaxis (buscando por ejemplo, entre definiciones de clases con *Rubinius.primitive* y al mismo tiempo referencias a *@variable.mensaje*) para identificar estos métodos y sus clases relacionadas, para luego definir qué acción tomar en cada caso. Quedará como trabajo futuro esta tarea.

Para aquellos mensajes que invoquen una primitiva en C++ y que esta última no colabore con otro objeto, se implementará un mensaje en **Class** para identificarlos:

```
class Class
  def mutable_primitives(*messages)
    (@mutable_primitives ||= Set.new).merge(messages)
  end

  def is_a_mutable_primitive?(message)
    !@mutable_primitives.nil? &&
        @mutable_primitives.include?(message)
  end
end
```

De este modo, para aquellas clases en Rubinius que sean mutables y tengan primitivas que NO envíen mensajes a colaboradores internos, estos mensajes serán luego ejecutados a través de la copia de trabajo actual de modo de trabajar en memoria paralela. Se verá luego cómo se utiliza este mensaje en el contexto de **AtomicUnboundMethod** para este propósito.

Por otra parte, las clases mutables sin colaboradores internos existen tanto en Ruby como en Rubinius. En el primero, se trata de las clases mutables *core* del lenguaje, implementadas en C. Este es el caso, por ejemplo, de **Array**, **String** y **Hash**. En el segundo, se trata de clases implementadas en Ruby que no poseen colaboradores internos pero llaman a primitivas que acceden a estado. Las dos clases ejemplares en el caso de Rubinius son **ByteArray** y **Tuple**, utilizadas por otras clases para implementar estructuras de tamaño variable.

Para identificar a estas clases, se implementará otro nuevo mensaje en **Class**:

```
class Class
  def mutable_primitive_instances?
    false
  end
end
```

Aquellas clases cuyas instancias sean primitivas mutables redefinirán este mensaje retornando *true*. Quedará como trabajo futuro hacer un listado exhaustivo de estas clases, tanto para MRI como para Rubinius.

Como se verá luego, este mensaje se utilizará para identificar estas clases en **AtomicUnboundMethod**. Al igual que antes, se enviarán los mensajes a través de la copia actual. En este caso, evitaremos tener que re-implementar todas estas clases primitivas, ya que no será necesario: Dado que estas estructuras no poseen colaboradores, no hace falta transformar mensajes de otros objetos recursivamente, sino que se logrará trabajar en memoria paralela, simplemente reenviando el mensaje original a la copia.

Como conclusión, vale la pena observar que en el caso de MRI, a pesar contar con un mayor número de código implementado en un lenguaje de bajo nivel, resulta menos

problemático en este aspecto que Rubinius, ya que todas sus clases primitivas no cuentan con colaboradores, por lo que no es necesario transformar dicho código ni analizar mensajes individualmente. Sin embargo, en ambos casos, el abuso de primitivas hace que sea necesario identificar manualmente muchísimas clases de objetos. Esto implica que futuras versiones de las implementaciones quizás tengan que redefinir estos casos. En el caso de Rubinius, esto sería potencialmente evitable ya que se podría disminuir considerablemente el uso de primitivas obligatorias en muchos casos, sin tener que reescribir la implementación prácticamente por completo.

3.9.3. Primitivas de infraestructura

Podemos clasificar las primitivas de infraestructura en 2 grandes grupos: De meta-modelo y de interfaz con el Sistema Operativo.

Gran parte de las clases / módulos incluidos en la distribución del lenguaje conforman el meta-modelo de este último. Clases como **Class**, **Module**, **Object**, **Method**, **UnboundMethod**, **Binding** y **Proc** modelan conceptos del propio lenguaje. La transformación de los mensajes de estos módulos traerá consecuencias indeseadas, difíciles de lidiar. Por ejemplo, un caso patológico claro podría ser la propia redefinición del método **atomic** definido en **Proc**, dentro de un contexto transaccional. En principio este tipo de casos no resulta de interés por varios motivos. En primer lugar, no es un uso habitual querer realizar este tipo de operaciones de meta-modelo en un contexto transaccional. En segundo lugar, los lenguajes que exhiben el meta-modelo al programador ya permiten a este alterar el lenguaje de forma de dejarlo totalmente inutilizable fácilmente (basta con intercambiar *true* por *false*, por citar un ejemplo obvio).

Sin embargo, existe casos habituales donde sí se intentarán transformar mensajes del meta-modelo. Un ejemplo es la utilización de un *proc* dentro de un bloque atómico, una práctica habitual en Ruby. Al querer evaluar un *proc* se enviará el mensaje **call**. Dado el contexto atómico, el mismo será renombrado como **__atomic__call**. Al enviarse este último, se intentará definir su variantes atómica. Tanto en MRI como en Rubinius, este mensaje es primitivo. En caso de querer reenviarse a través de la copia actual, se intentará hacer una copia *shallow* del bloque. En general esto fallará ya que no es posible crear un *proc* alocando memoria vía **allocate**. En este caso, el camino sencillo será omitir la transformación de estos mensaje, evitando las copias de los mismos.

El otro grupo que resulta potencialmente problemático está compuesto por aquellos mensajes de más bajo nivel, que interactuarán con el SO. Aquí se pueden identificar una variedad de mensajes. Uno de ellos serán los mensajes de alocación de memoria, como el mensaje de clase **allocate** definido en **Class**. Ejecutar este mensaje en memoria paralela carece de sentido (¿Qué es "pedir memoria" en forma atómica?) por lo que tiene sentido omitir su ejecución. Otro grupo de mensajes serán aquellos relacionados con el manejo de la concurrencia, como los que pertenecen a las clases **Thread** y **Fiber**. Detener un hilo de ejecución en forma atómica tampoco tiene una semántica clara. En la práctica, enviar el mensaje **Thread#stop** en memoria paralela a través de la copia de trabajo actual hará que la aplicación nunca termine de ejecutarse (en los experimentos realizados, esto ocurre en MRI pero no en Rubinius, por causas que se desconocen).

Como conclusión, una heurística razonable consiste en no transformar estos mensajes. Sin embargo, para obtener una solución robusta, se debería estudiar con mayor profundidad cada caso para determinar la mejor acción posible. Por los experimentos y el análisis

realizado al código fuente de las implementaciones (principalmente en Rubinius), el primero de los grupos (meta-modelo) parece ser sumamente más complejo de abordar.

Para marcar estos mensajes y evitar su transformación, se propone la siguiente extensión a **Module**:

```
class Module
  def infrastructure_primitives(*method_names)
    (@infrastructure_primitives ||= Set.new).merge(method_names)
  end

  def is_infraestructure_primitive?(method_name)
    !@infrastructure_primitives.nil? &&
        @infrastructure_primitives.include?(method_name)
  end
end
```

Cabe notar que en este caso la implementación se realiza en **Module** y no en **Class**, ya que las primitivas de infraestructura no son un concepto único de las clases sino de métodos en general, que pueden estar presente en módulos.

3.9.4. Primitivas opcionales (Rubinius)

Otro grupo de primitivas de interés son aquellas primitivas que se encuentran implementadas también en Ruby pero que existen en versión C++ por temas de *performance*. La estrategia utilizada aquí consistió en remover los nodos de llamadas a primitivas en Rubinius. De este modo, aquellos mensajes implementados en Ruby, al no existir la macro de invocación de la primitiva, el resto del método se ejecutará normalmente y podrá ser transformado sin problemas. Veremos más detalles sobre la implementación al analizar **AtomicUnboundMethod** y en particular el *rewriter* AST **RemoveRbxPrimitivesRewriter**.

3.10. La jerarquía *AtomicExecutable*: Representando bloques y métodos 'atómicos'

Definimos como *ejecutable* a todo objeto que represente código que puede ser ejecutado. Los métodos y los *procs* entra en esta categoría. Debido a que deseamos construir instancias de métodos y de bloques que se ejecuten en un contexto atómico y sus transformaciones serán similares, se procedió a definir una nueva jerarquía de clase donde **AtomicExecutable** será una clase padre abstracta con comportamiento común para ambos tipos de objetos, con los hijos **AtomicProc** y **AtomicUnboundMethod** para la implementación específica de cada caso.

Veamos a continuación la definición de **AtomicExecutable**. Comenzaremos describiendo su interfaz pública:

```
class AtomicExecutable
  def self.from(executable)
```

```
    new(executable)
  end

  def initialize(executable)
    @original_exec = executable
    @atomic_exec = generate_atomic
  end

  def source_code
    source_code = original.source_code
    transform_to_atomic(source_code)
  end

  def to_atomic
    self
  end

  # ...
end
```

La inicialización consiste en guardar una referencia al ejecutable original y enviar el mensaje **generate_atomic** para generar un nuevo objeto ejecutable atómico.

El mensaje **source_code** servirá para retornar el código fuente del objeto atómico. El algoritmo consiste en obtener el código mediante el mensaje **source_code** al ejecutable y aplicar las transformaciones correspondientes para retornar la versión atómica del código.

Respecto a la parte privada, se definieron los siguientes mensajes:

```
class AtomicExecutable
  # ...

  def generate_atomic
    raise NotImplementedError
  end

  def original
    @original_exec
  end

  def transform_to_atomic(source_code)
    atomic_send_transformation!(source_code)
    atomic_state_access_transformation!(source_code)
    source_code
  end

  def atomic_state_access_transformation!(source_code)
    state_access_rewriter = AtomicStateAccessRewriter.new
    source_code.apply_ast_transformation!(state_access_rewriter)
```

```
  end

  def atomic_send_transformation!(source_code)
    send_rewriter = atomic_send_rewriter
    source_code.apply_ast_transformation!(send_rewriter)
  end

  def atomic_send_rewriter
    AtomicSendRewriter.new
  end

  def method_missing(symbol, *args)
    @atomic_exec.send(symbol, *args)
  end
end
```

En primer lugar, definimos **generate_atomic** como un mensaje abstracto que será redefinido para **AtomicProc** y **AtomicUnboundMethod**, ya que la generación del objeto atómico depende fuertemente del tipo de objeto.

El mensaje **transform_to_atomic** aplicará las 2 transformaciones que fueron enunciadas al comienzo del trabajo:

- **atomic_send_transformation!**: Se inicializa un nuevo *rewriter* de AST (**AtomicSendRewriter**, que será analizado posteriormente) y se aplica la transformación sobre el código fuente enviando el mensaje **apply_ast_transformation!** (parte de **ExecutableSourceCode**, a ser analizado posteriormente). Esta transformación se encargará de reemplazar todos los envíos de mensaje por su variante atómica (prefijada con *__atomic__*).
- **atomic_state_access_transformation!**: Similar a la anterior, se inicializará el *rewriter* **AtomicStateAccessRewriter** que realizará las transformaciones AST relacionadas a la lectura y escritura del estado de los objetos en memoria paralela. Se analizará en detalle posteriormente.

Por último, se sobrescribió **method_missing** de modo tal que se reenvíe todo mensaje no encontrado al nuevo ejecutable atómico generado. Notar que la interfaz pública aquí presentada también fue implementada en la clases de los objetos originales **Proc** y **UnboundMethod**. Por lo tanto, al reenviar todo el resto de los mensajes no definidos, logramos que las instancias de **AtomicExecutable** sean completamente polimórficas con los ejecutables en todo su conjunto de mensajes.

3.10.1. *AtomicProc*: Generando *procs* atómicos

Un *proc* atómico se construye a partir de un *proc* no-atómico. Como mencionamos, se trata de una subclase de **AtomicExecutable**. Veamos a continuación el código fuente:

```
class AtomicProc < AtomicExecutable
  private
```

```
  def generate_atomic
    Proc.new do
      original_.binding.eval(source_code.to_s,
                             *original.source_location)
    end
  end

  def atomic_send_rewriter
    AtomicSendRewriter.new(original.binding)
  end
end
```

La generación del bloque atómico (vía **generate_atomic**) consistirá en crear un nuevo *proc* (usando el mecanismo normal de creación de un bloque) a partir del código fuente retornado por el mensaje **source_code**, ya transformado atómicamente. Para lograr crear un bloque a partir de código fuente, haremos uso del método **eval** (definido en la clase **Binding** provista por el lenguaje) que permite evaluar código dinámicamente (realizando el parsing y compilación correspondiente en tiempo de ejecución) utilizando el contexto (*binding* en terminología de Ruby) de un objeto en particular. Dado que las variables referenciadas en el código fuente del *proc* original serán las mismas que las referencias en el código transformado, evaluaremos el código fuente utilizando el mismo contexto que el *proc* original. Para acceder al objeto que representa su contexto basta con enviar el mensaje **binding** al mismo.

Por último, redefiniremos el mensaje **atomic_send_rewriter**. En este caso, a diferencia de la implementación original, se pasará el *binding* del bloque como parámetro al constructor de **AtomicSendRewriter**. Veremos por qué esto es necesario al ver la implementación de este último más adelante.

3.10.2. *AtomicUnboundMethod*: Generación de métodos atómicos

Respecto a **AtomicUnboundMethod**, veamos en primer lugar su interfaz pública:

```
class AtomicUnboundMethod < AtomicExecutable
  def name
    original.name.to_atomic_method_name
  end

  def owner
    original.owner
  end

  def source_code
    source_code = original.source_code
    source_code.change_name_in_definition!(name)
    transform_to_atomic(source_code)
  end
```

```
  # ...
end
```

En primer lugar, el mensaje **name** deberá retornar la versión atómica del nombre del método original. Por otra parte, el mensaje **owner**, también implementado originalmente en **UnboundMethod**, reenviará el mensaje al método original. El *owner* representa a la clase donde se encuentra originalmente definido el método. El nuevo método atómico será definido en la misma clase que el método original.

El mensaje **source_code** será alterado respecto de la definición canónica presentada en **AtomicExecutable**. La diferencia es que en este caso no basta sólo con realizar la transformación del código fuente del cuerpo del método sino que será necesario también alterar su encabezado, mediante el método **change_name_in_definition!(name)**. Se usará como nombre del método su versión atómica.

Respecto a la generación del método atómico, se propuso en primera instancia la siguiente implementación de **generate_atomic**:

```
def generate_atomic
  define_in_owner
  define_aliases
  Module.register_module_with_an_atomic_method(owner)
  owner.instance_method(name)
end
```

Como se puede observar, el algoritmo cuanta con una serie de pasos, concretados mediante el envío de diferentes mensajes.

En primer lugar, se procede a concretar la definición del método en sí, dentro de la clase dueña (**owner**) del método. Dado que esto presenta una serie de problemáticas, será analizado posteriormente (en una sección aparte).

Luego, se enviará el mensaje **define_aliases**. En Ruby, un mismo método puede tener distintos nombres conocidos como *alias*. Dado que el método original podría tener distintos alias, desearemos que la versión atómica del método también sea definida para dichos alias. Para lograr esto, se implementó el siguiente mensaje (en **AtomicUnboundMethod**):

```
def define_aliases
  owner.aliased_methods(original) do |meth_name|
    alias_method(meth_name.to_atomic_method_name,
                 name)
  end
end
```

El mensaje buscará todos los metodos alias del original mediante el mensaje **aliased_methods** (definido como parte del trabajo, como extensión de la clase **Module**, presentada previamente) enviado a la clase dueña del método. Luego se utilizará el mensaje **alias_method** provisto por el lenguaje para crear el alias, por cada uno de los alias que haya.

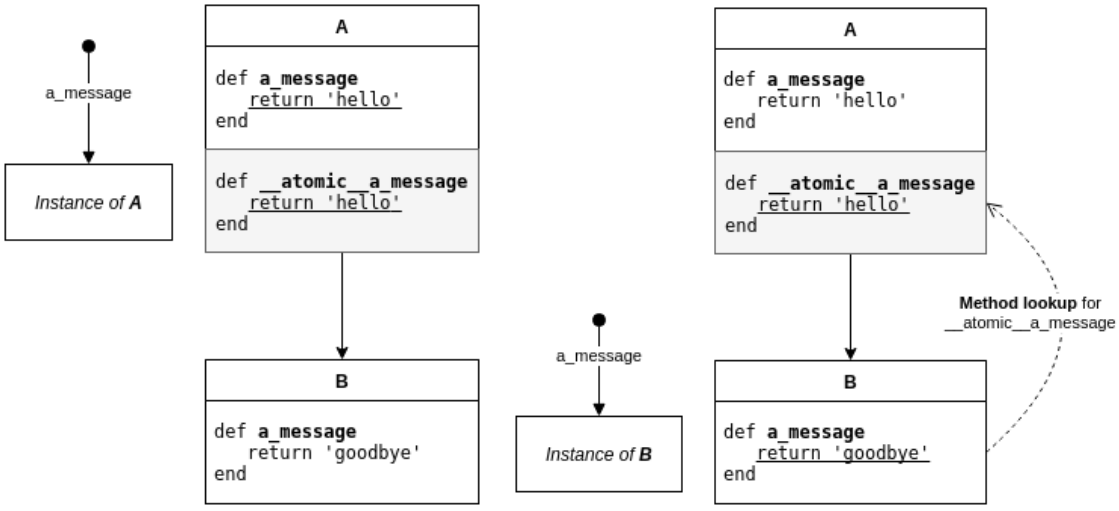
Volviendo a **generate_atomic**, en tercer lugar se procederá a registrar a la clase (módulo) dueña del método como una clase con métodos atómicos, mediante el envío del mensaje de clase **register_module_with_an_atomic_method** a **Module**. El objetivo de registrar aquellos módulos que tienen métodos atómicos consiste en minimizar drásticamente la búsqueda de métodos atómicos al momento de querer hallar todos los métodos insertados en el sistema hasta el mismo. Como se observó anteriormente, esto servirá, por ejemplo, para poder remover fácilmente todos los métodos atómicos insertados.

Por último, el mensaje retornará una instancia de **UnboundMethod** con el método recién definido. Se utiliza el mensaje **Module#instance_method** provisto por el lenguaje para lograr esto.

3.10.3. *AtomicUnboundMethod*: Corrigiendo comportamientos inesperados del algoritmo de *method lookup*

La implementación recién presentada se encarga de definir el método atómico para la clase dueña del método original. Sin embargo, es interesante notar que esto no es suficiente para que la técnica presentada sea satisfactoria. El problema, que radica en el algoritmo de *method lookup*, se puede observar en el siguiente escenario:

Sea **A** una clase padre de **B** que define un mensaje llamado **a_message** y la clase B define el mismo mensaje con un comportamiento alternativo. Supongamos luego que un objeto le envía el mensaje **a_message** a una instancia de **A**. Cómo vimos hasta el momento, al intentar buscar la versión atómica de **a_message**, el mismo no será hallado, por lo que se enviará el mensaje **method_missing** que definirá un método nuevo atómico en la clase **A**, tal como se muestra en la figura 3.1a a continuación.



(a) Se envía **a_message** a una instancia de **A** y se define **__atomic__a_message** en la clase **A** correctamente.

(b) Se envía **a_message** a una instancia de **B** y el algoritmo de *method lookup* encuentra **__atomic__a_message** definido en **A**, ejecutando una implementación distinta a la esperada.

Supongamos que luego un objeto le envía **a_message** a una instancia de **B**. Esta vez, al no hallarse definido el método en B, en lugar de enviarse **method_missing** inmediatamente, el algoritmo de *method_lookup* irá a buscar el método en la clase padre **A**. Dado que dicho método fue generado previamente en el paso anterior, esta vez sí será hallado.

Sin embargo, el código atómico a ejecutar corresponderá a una implementación relativa al mensaje definido en **A**, en lugar de comportarse como se encuentra definido en **B**, que era la intención original, tal como se observa en la figura 3.1b.

Existen 2 maneras naturales de lidear con este problema. La primera consistiría en alterar el algoritmo de *method lookup* para comportarse de forma distinta ante métodos atómicos. Sin embargo, esta solución requeriría realizar modificaciones a nivel VM, donde se encuentra implementado el algoritmo.

El segundo camino para evitar el problema consiste en definir los métodos atómicos no sólo en la clase dueña del método original, sino de toda subclase de esta última al momento de evaluar el primer método atómico. De este modo, se definirán las versiones atómicas de los métodos de mismo nombre de toda la jerarquía, de modo de que siempre el algoritmo encuentre el método correcto (siempre hallará la versión atómica en la clase sobre la cual se está enviando el mensaje).

Para lograr esto, se modificó ligeramente el algoritmo de **generate_atomic** del siguiente modo:

```
def generate_atomic
  define_in_owner
  define_aliases
  Module.register_module_with_an_atomic_method(owner)
  if owner.is_a?(Class)
    generate_atomic_methods_for_subclasses
  end
  owner.instance_method(name)
end
```

Antes de retonarar el nuevo método, se procederá a definir los métodos atómicos mediante **generate_atomic_methods_for_subclasses** (siempre y cuando el dueño se trate de una clase y no de un módulo, ya que estos últimos no tienen subclases).

```
def generate_atomic_methods_for_subclasses
  subclasses = owner.subclasses_implementing_message(original.name)
  subclasses.each do |subclass|
    subclass_methods = subclass.instance_methods(false)
    unless subclass_methods.include?(name)
      original_method = subclass.instance_method(original.name)
      AtomicUnboundMethod.from(original_method)
    end
  end
end
```

En primer lugar, se obtendrán todas las subclases que implementan el mensaje relacionado con el método original mediante el mensaje **subclasses_implementing_message** (ya presentado anteriormente). Luego, se recorrerán las subclases y se generarán nuevos métodos atómicos mediante el constructor **AtomicUnboundMethod::from**. El mismo enviará luego le mensaje **generate_atomic** que realizará la operación sobre la subclase.

De este modo, se irán definiendo recursivamente todos los métodos de la jerarquía, recorriendo el árbol de clases en profundidad (*DFS*). Notar que previo a insertar el método, se verificará que el mismo no haya sido previamente definido para dicha subclase, ya que podría suceder que se haya enviado previamente el mensaje a una de las subclases antes que a la clase padre.

Si bien se trata de una operación muy costosa en terminos computacionales (el árbol de clases podría ser potencialmente muy grande), notar que sólo se efecturá esta operación la primera vez que el mensaje original sea envíado, durante todo el ciclo de vida de la aplicación.

Por último, merece ser destacado que este último mensaje resulto natural y sencillo de implementar gracias a las notables capacidades reflexivas del lenguaje.

3.10.4. *AtomicUnboundMethod*: Definiendo el método atómico en la clase

Retomando el análisis de **generate_atomic**, queda pendiente la implementación del mensaje **define_in_owner**, encargado de definir eefectivamente el método en la clase *owner*.

A primera vista, la definición de un nuevo método atómico no presenta desafíos. La implementación natural que surge es:

```
def define_in_owner
  owner.send(:define_method_using_source_code, name, source_code)
end
```

Sin embargo, cómo se anticipó al analizar las extensiones a **Module**, aquí es donde aparece la primera gran dificultad de la implementación: No todos los métodos tienen su código fuente disponible.

Para el caso de aquellos objetos que se encuentren implementados en C/C++ y sean mutables, una opción es reenviar sus mensajes originales a través de la copia de trabajo actual del objeto (*working copy*). Dado que todo su estado se encuentra contenido en el mismo objeto (a bajo nivel, una clase de C++ o *struct* de C), estos objetos no cuentan con colaboradores internos. Por lo tanto, no hay envíos de mensajes a colaboradores por lo que no hay otros métodos que deban ser transformados. De esto modo, reenviar el mensaje vía la copia actual hará que se trabaje en memorial paralela correctamente.

Por otro lado, como se analizó previamente, en el caso de Rubinius existen objetos mutables que sí cuentan con colaboradores internos pero tienen algunos mensajes implementados como primitivas. En estos casos, reenviaremos también estos mensajes a través de la copia de trabajo actual.

En base a estos dos casos, procedemos a definir el siguiente mensaje:

```
def should_send_through_working_copy?
  owner.is_a?(Class) && (owner.mutable_primitive_instances? ||
      owner.is_a_mutable_primitive?(original.name))
end
```

Notar que en estos dos casos aplican únicamente a clases y no a módulos, por lo que se verifica previamente que el dueño del métodos sea efectivamente una clase.

Teniendo esta consideración, se sugiere la siguiente implementación para **define_in_owner**:

```
def define_in_owner
  if should_send_through_working_copy?
    owner.send(:define_method, name, original_through_working_copy)
  else
    owner.send(:define_method_using_source_code, name, source_code)
  end
end
```

Para realizar la definición vía *working copy* utilizaremos el mensaje **define_method** definido en **Module** (nuevamente, privado). Según la documentación oficial, el mismo puede tomar un bloque de código (*proc*) que servirá como cuerpo del método. En este caso, construiremos el bloque vía el mensaje **original_through_working_copy** implementado del siguiente modo:

```
def original_through_working_copy
  ->(*args, &block) do
    send_through_working_copy(__method__.to_nonatomic_method_name,
                              *args, &block)
  end
end
```

Notar que utilizamos una sintaxis alternativa para definir el bloque, el lugar de *Proc.new*. Esto se debe a que esta última forma no permite pasar otros bloques como parámetro. Dado que deseamos pasar los argumentos original del método, necesitamos contemplar la posibilidad de que uno de ellos sea un bloque, vía el parámetro *&block*.

Para realizar el reenvío del mensaje, simplemente enviaremos el mismo en forma dinámica utilizando **send_through_working_copy**, definido previamente en **Object**. Notar el uso del mensaje **__method__** provisto por el lenguaje, que retorna el nombre del método que se está ejecutando actualmente. Al momento de evaluarse el código, este se corresponderá con el nombre del método atómico ejecutado. Dado que se quiere enviar el mensaje original, se convertirá el nombre vía el mensaje **to_nonatomic_method_name**.

Además del caso del envío a través de la copia actual, tal como se mencionó también durante la extensión presentada para **Module** / **Class**, existen otros mensajes primitivos que podremos evitar transformar.

En primer lugar, otro grupo de mensajes que deseamos evitar transformar serán los mensajes dirigidos hacia objetos inmutables. Dado que su estado permanece constante, no será necesario que los métodos se ejecuten en memoria paralela. En segundo lugar, se encuentran las primitivas que denominamos “de infraestructura”.

Nuevamente, estas afirmaciones dan lugar a la implementación del siguiente mensaje:

```
def should_not_transform?
  owner.is_a?(Class) && owner.immutable_instances? ||
      owner.is_infraestructure_primitive?(original.name)
end
```

Utilizando este mensaje, definimos finalmente la siguiente implementación para **define_in_owner**:

```
def define_in_owner
  if should_not_transform?
    owner.send(:define_method, name, original)
  elsif should_send_through_working_copy?
    owner.send(:define_method, name,
               original_through_working_copy)
  else
    owner.send(:define_method_using_source_code, name,
               source_code)
  end
end
```

Notemos entonces que en el caso de que los mensajes primitivos no se encuentren en ninguno de los grupos distinguidos, se realizará la definición del métodos mediante su código fuente. En este caso, fallará si el mismo no está disponible. En estos casos, dichos mensajes deberán ser rescritos en Ruby para poder ser transformados, o distinguidos especialmente para ser tratados con algunas de las otras dos estrategias (reenvío del mensaje original, ya sea a través de la copia de trabajo, o directamente sobre el objeto original).

3.10.5. *AtomicUnboundMethod*: Particularidades de Rubinius

Durante la investigación se detectaron otras problemáticas relacionadas a la transformación de código, vinculadas específicamente a Rubinius.

Como vimos anteriormente, en primer lugar, en Rubinius existen primitivas opcionales. Se tratan de mensajes que invocan a una primitiva (vía la macro ya mencionada) pero a continuación tienen un código fuente alternativo escrito en forma nativa. Al igual que ocurren en Smalltalk, este es el caso de primitivas que se implementan por temas de *performance* puntualmente.

Por ejemplo, este es el caso del mensaje **at** definido en **Array**.

```
def at(idx)
  Rubinius.primitive :array_aref
  idx = Rubinius::Type.coerce_to_collection_index idx

  total = @start + @total

  if idx < 0
```

```
    idx += total
  else
    idx += @start
  end

  if idx >= @start and idx < total
    return @tuple.at idx
  end
end
```

Dado que no hay una forma determinista de distinguir primitivas opcionales de requeridas, en principio habría que redefinir aparte la versión atómica de estos métodos. Para evitar esto, se optó por implementar una transformación AST específica para Rubinius que elimine el nodo relacionado con la macro *Rubinius.primitive*. En el caso de los mensajes donde no exista código Ruby alternativo, se lanzará una excepción tipo **PrimitiveFailed**. Si bien en principio esto es problemático, estos mensajes no podrían ser transformados inicialmente, con lo cual que fallen explícitamente es más bien una mejor política para estos casos (considerando el principio de *fail-fast*).

Para aplicar esta transformación, definiremos un nuevo mensaje en **AtomicUnboundMethod**:

```
def remove_rbx_primitives!(source_code)
  src_rewriter = source_code.new_source_rewriter
  remove_primitives = RemoveRbxPrimitivesRewriter.new(src_rewriter)
  src_rewriter = remove_primitives.do(source_code.to_ast)
  source_code.apply_source_rewrite!(src_rewriter)
end
```

El mensaje tomará como entrada el código fuente a transformar, creará un nuevo **SourceRewriter** (facilidad provista por la librería ***parser***, que se mencionará luego) enviando el mensaje **new_source_rewriter** (que será presentado luego como parte de **ExecutableSourceCode**). Luego instanciaría **RemoveRbxPrimitivesRewriter** que será la encargada de realizar el trabajo, mediante el mensaje **do**. Luego se hará efectiva esta reescritura al código fuente vía **apply_source_rewrite!**, también a ser presentado posteriormente.

Finalmente, para poder aplicar esta transformación, sobrecargaremos el mensaje **transform_to_atomic** definido en **AtomicExecutable**, de modo que se envíe primero el mensaje recién presentado:

```
if RUBY_ENGINE == 'rbx'
  def transform_to_atomic(source_code)
    remove_rbx_primitives!(source_code)
    super
  end
end
```

La segunda problemática encontrada radica en ciertas consideraciones especiales a tener en cuenta al reescribir los envíos de mensajes de forma atómica en Rubinius. Por este motivo, se implementó una clase alternativa denominada **AtomicSendOnRbxRewriter**. Se verá en detalle el trabajo de esta clase en las próximas secciones.

Para poder realizar la transformación con la clase alternativa, en lugar la original, se redefinió **atomic_send_rewriter**:

```
if RUBY_ENGINE == 'rbx'
  def atomic_send_rewriter
    AtomicSendOnRbxRewriter.new
  end
end
```

3.11. AST *Rewriters*: Reescribiendo el árbol AST

En esta sección analizaremos un conjunto de clases nuevas que se encargarán de realizar todas las transformaciones de código ya mencionadas. Llamaremos *rewriters* a estos objetos.

Para realizar estas operaciones, se recurrió al uso de una librería de terceros denominada **parser** que provee mecanismos para convertir código fuente Ruby en un árbol AST y luego procesarlo.

La forma de realizar las escrituras consiste en implementar un procesador AST heredando de la clase **Parser::AST::Processor** e implementando distintos mensajes que se envíen durante el recorrido del árbol.

Existen otro tipo de reescrituras que permiten realizar otras operaciones como renombrar fragmentos de nodos de texto (como el nombre de un método) o incluso eliminarlos. Para estas tareas, también utilizaremos un procesador AST pero se hará en conjunto con un **SourceRewriter** (provisto también por la librería), que podrá realizar estas operaciones específicas.

Las re-escrituras implementadas son:

- **AtomicSendRewriter**: Transformación del envío de mensajes.
- **AtomicSendOnRbxRewriter**: Variante de la primera, específica para Rubinius.
- **AtomicStateAccessRewriter**: Transformación de acceso a estado (misma implementación para ambas plataformas).
- **RemoveRbxPrimitivesRewriter**: Remoción de nodos primitivos para Rubinius.

Analizaremos a continuación cada una de estas en detalle.

3.11.1. *AtomicSendRewriter*: Transformación del envío de mensajes

Como se comentó al comienzo, el objetivo de la transformación será reemplazar todos los envios de mensaje por mensajes atómicos (aquellos prefijados con '__atomic__'). El procesador original **Parser::AST::Processor** recorre el nodo de envío enviando el mensaje **on_send**. Veamos su definición a modo de referencia:

```ruby
def on_send(node)
  receiver_node, method_name, *arg_nodes = *node

  receiver_node = process(receiver_node) if receiver_node
  node.updated(nil, [
    receiver_node, method_name, *process_all(arg_nodes)
  ])
end
```

El procesamiento del nodo consiste en procesar el nodo receptor del mensaje (si existe) y los argumentos del mensaje recursivamente, y luego actualizar el nodo con estos valores. Dado que el interés es simplemente renombrar el nombre del método, una implementación de **on_send** que cumpla este objetivo sería:

```ruby
def on_send(node)
  receiver_node, method_name, *arg_nodes = *node

  receiver_node = process(receiver_node) if receiver_node

  method_name = method_name.to_atomic_method_name  # nueva linea

  node.updated(nil, [
      receiver_node, method_name, *process_all(arg_nodes)
  ])
end
```

Sin embargo, esta implementación tiene un **problema** particular producto de un **accidente sintáctico** del lenguaje al momento de querer transformar un nodo de envío de mensaje en un contexto particular. En Ruby, sin el contexto apropiado, **el envío de un mensaje sin argumentos es indistinguible sintácticamente del acceso a una variable local**. Por lo tanto, al realizar el parseo del código fuente de un *proc* por ejemplo, una variable local definida por fuera del bloque (que es accesible en el *scope* del *proc*) podría ser confundida con el envío de un mensaje del mismo nombre. Notar que esto no es una deficiencia del parser sino del uso particular que se le está dando: Realizar una transformación de un fragmento de código sin contexto. En caso de que el código parseado incluya la definición de toda variable local que se menciona dentro del contexto de un *proc* el parser puede distinguir entre ambos y determinar que se trata de una acceso a una variable y no la llamada a un método del mismo nombre. Sin embargo, por el enfoque presentado de este trabajo, no será posible contar con todo el código fuente que rodea al bloque a ser transformado, ya que potencialmente las variables locales podrían estar definidas incluso en otros archivos (cargados mediante **requiere**).

Si bien no es posible parsear todo el código necesario para determinar el contexto, sí es posible contar con el contexto de ejecución del **proc** original mediante el mensaje **binding** como vimos anteriormente. De este modo, podemos facilitar a las instancias de **AtomicSendRewriter** con el *binding* del *proc* al momento de su creación. La nueva inicialización de la instancia será:

```
def initialize(a_binding=nil)
  if !a_binding.nil?
    @local_vars_in_scope = LocalVarsInScope.new(a_binding)
  else
    @local_vars_in_scope = Set.new
  end
end
```

La instancia de **LocalVarsInScope** representará al conjunto de variables locales en el contexto. Veamos la definición de la clase auxiliar:

```
class LocalVarsInScope
  def initialize(binding)
    @source_binding = binding
  end

  def include?(var_name)
    begin
      return @source_binding.local_variable_defined?(var_name)
    rescue NameError
      false
    end
  end
end
```

El mensaje **include?** simplemente verificará que el *binding* tenga definida el nombre en cuestión como una variable local, mediante **local_variable_defined?** (provisto por el lenguaje). Otra vez más, se puede ver como las capacidades reflexivas del lenguaje permiten realizar análisis del código imposibles de realizar de otro modo.

Finalmente, la nueva variante del mensaje **on_send** será:

```
def on_send(node)
  receiver_node, method_name, *arg_nodes = *node

  receiver_node = process(receiver_node) if receiver_node

  if should_transform_to_atomic?(node)
    method_name = method_name.to_atomic_method_name
  end

  node.updated(nil, [
      receiver_node, method_name, *process_all(arg_nodes)
  ])
end
```

...donde **should_transform_to_atomic?** se define como:

```
def should_transform_to_atomic?(send_node)
  _, method_name, *_ = *send_node

  !@local_vars_in_scope.include?(method_name)
end
```

Cómo se verá a continuación, este último mensaje será reimplementado para Rubinius, ya que la implementación hasta aquí presentada funcionará correctamente en la distribución oficial de Ruby, pero sin embargo seguirá trayendo inconvenientes en la distribución alternativa.

3.11.2. *AtomicSendOnRbxRewriter*: Manejo especial para macros de compilador presentes en Rubinius

Debido a diferencias de implementación, se definió la clase **AtomicSendOnRbxRewriter**, que hereda de **AtomicSendRewriter**, con comportamiento específico necesario para Rubinius. Principalmente, se definirán nuevos criterios de cuando no transformar un nodo de "envío de mensaje", agregando nuevos casos a **should_transform_to_atomic?**.

Como se comentó en secciones anteriores, no todos los mensajes presentes en el código del *kernel* de Rubinius son realmente mensajes. En algunos casos se trata de primitivas del lenguaje que no deben ser transformadas en versiones atómicas. Si bien este escenario parece similar a los métodos nativos de *MRI*, en caso de realizar la transformación, el algoritmo de *method missing* no podrá hallar ni siquiera la versión nativa de estos métodos ya que los métodos no están directamente definidos como tales, sino que son simplemente palabras claves para que el compilador los transforme en un *bytecode* especial, como se mencionó también previamente.

Para resolver esta problemática, se realizó una investigación acerca de la transformaciones de tipo *kernel* aplicadas en Rubinius para tratar de manera especial explícitamente a estos nodos, recorriendo el código fuente de la implementación (no se halló documentación al respecto).

Observando las fuentes definidas en lib/rubinius/code/ast/transforms.rb, se listan las siguientes clases de transformaciones: **BlockGiven**, **AccessUndefined**, **SendPrimitive**, **CheckFrozen**, **InvokePrimitive**, **CallCustom**, **SingleBlockArg**, **InlineAssembly**, **SendPrivately**, **SendFastNew**, **SendKernelMethod**, **SendFastCoerceTo**.

En primera instancia, la primer medida consistió en omitir la transformación del envío de mensajes para todas las macros encontradas. Para esto, se implementó el siguiente mensaje:

```
def is_a_rbx_undefined_method_node?(node)
  receiver_node, method_name, *_ = *node

  if not receiver_node.nil?
    primitives = [:primitive, :invoke_primitive, :check_frozen,
                  :call_custom, :single_block_arg, :asm,
                  :privately]
```

```
    is_rbx_primitive = receiver_node.children[1] == :Rubinius &&
        primitives.include?(method_name)

    undef_equal_send = receiver_node.children[1] == :undefined &&
        method_name == :equal?

    is_rbx_primitive || undef_equal_send
  else
    [:undefined, :block_given?, :iterator?].include?(method_name)
  end
end
```

La lista de la variable local *primitives* muestra los mensajes que serán omitidos, que comienzan con **Rubinius.**. Esto hará que se puedan aplicar originalmente las transformaciones **SendPrimitive**, **CheckFrozen**, **InvokePrimitive**, **CallCustom**, **SingleBlockArg**, **InlineAssembly**, **SendPrivately**.

Respecto a **BlockGiven**, **SendFastNew**, **SendKernelMethod**, **SendFastCoerceTo** se trata simplemente de optimizaciones a mensajes que sí se encuentran definidos en Ruby por lo cual no es necesario considerarlos.

La tranformación restante **AccessUndefined** es tratada aparte ya que el keyword *undefined* se utiliza en como "receptor" especial del mensaje *equal?* (y no es un "mensaje" enviado a *Rubinius*).

De este modo, la implementación alternativa para **should_transform_to_atomic?** sería:

```
def should_transform_to_atomic?(send_node)
  _, method_name, *_ = *send_node

  unless super
    return false
  end

  if is_a_rbx_undefined_method_node?(send_node)
    return false
  end

  true
end
```

Sin embargo, las transformaciones **InlineAssembly** y **SendPrivately** presentan complicaciones adicionales analizadas a continuación.

La transformación InlineAssembly

InlineAssembly, cómo su nombre lo indica, sirve para embeber *byte-code* de forma *inline* en el contexto de un método Ruby. Por ejemplo, se utiliza en la implementación de **ByteArrray::new**:

```
def self.new(cnt)
  obj = allocate_sized cnt
  Rubinius.asm(obj) do |obj|
    run obj
    send :initialize, 0, true
  end

  obj
end
```

La macro toma un bloque como parámetro que contiene las instrucciones de la VM. Si no hacemos un tratamiento especial para este caso, las instrucciones serán interpretadas también como mensajes y la ejecución del código fallará.

La solución en este caso consiste en detectar cuando se está dentro de un bloque invocado por *Rubinius.asm* y omitir la transformación en tal caso. Por suerte, el procesador AST nos permite detectar esto a través del mensaje *on_block*, que será invocado al procesar los nodos relativos a los bloques.

Se implementa entonces el siguiente mensaje en **AtomicSendOnRbxRewriter**:

```
def on_block(node)
  send_node, _, body_node = *node
  receiver_node, method_name, *_ = *send_node

  if is_rbx_asm_node?(receiver_node, method_name)
    @currently_inside_rbx_asm_block = true
    processed_node = super
    @currently_inside_rbx_asm_block = false
    return processed_node
  end

  super
end
```

Básicamente, en caso de que el bloque en cuestión pertenezca a un nodo de tipo Rubinius.asm, marcaremos ese estado mediante la variable de instancia *booleana @currently_inside_rbx_asm_block*. A continuación procesaremos el nodo normalmente (invocando a *super*) y re-estableceremos el valor de la variable en cuestión (la misma será inicializada en *false* al construir el *rewriter*).

La definición de **is_rbx_asm_node?** resulta relativamente simple gracias a la librería utilizada:

```
def is_rbx_privately_node?(receiver_node, method_name)
  !receiver_node.nil? && receiver_node.children[1] == :Rubinius &&
      method_name == :privately
end
```

Una vez implementado esto, basta con alterar nuevamente **should_transform_to_atomic?** agregando el nuevo caso:

```
def should_transform_to_atomic?(send_node)
  _, method_name, *_ = *send_node

  unless super
    return false
  end

  if is_a_rbx_undefined_method_node?(send_node)
    return false
  end

  if currently_inside_rbx_asm_block?
    return false
  end

  true
end
```

La transformación SendPrivately

Para el caso de la transformación **SendPrivately** ocurre algo similar a **InlineAssembly**: La macro toma un bloque como parámetro que debe ser interpretado distinto.

SendPrivately es una macro que permite englobar un bloque de código de modo de forzar que los envíos de mensajes que hagan en forma privada, sin tener que recurrir a enviar estos mensajes vía **send**. Por ejemplo, permitiendo escribir *obj.private_message* en lugar de *obj.send(:private_message)*. Su existencia es más bien un *hack* (descripto así por Brian Shirai, uno de los desarrolladores principales de Rubinius) y tiene que ver con que los mensajes envíados vía *send* no se almacenarían en la *caché* de métodos de la VM por motivos técnicos.

La consecuencia de esta macro es que el bloque encapsulado por *Rubinius.privately* no es realmente un bloque y el contenido tiene que ser interpretado como que si se ejecutara fuera del bloque de la macro. Esto es particularmente problemático ya que las variables locales definidas dentro del "bloque" no serán consideradas variables locales al *scope* externo. Veamos por ejemplo el caso de **Hash#[]**:

```
def [](key)
  if @trie
    Rubinius.privately { key_hash = key.hash }
    if item = @trie.lookup(@state, key, key_hash)
      return item.value
    end
  end
```

```
  default key
end
```

La variable *key_hash* en principio es local al bloque *Rubinius.privately*. Sin embargo, es utilizada luego como parámetro en *@trie.lookup*, escapando al *scope*. Si el parser no tiene una consideración especial, interpretará el segundo *key_hash* como un envío de mensaje, ya que la variable local no existe en el *scope* del método.

Para solucionar este problema, la idea será incorporar todas las variables locales dentro de los bloques *Rubinius.privately* al contexto externo. Luego, se optará por omitir la transformación a dichos "mensajes" al igual que en los casos anteriores.

Para acumular las variables, podemos alterar la definición de **on_block**, de modo de acumular las variables que se encuentran dentro de ese bloque.

```
def on_block(node)
  send_node, _, body_node = *node
  receiver_node, method_name, *_ = *send_node

  # -- nuevo fragmento --
  if is_rbx_privately_node?(receiver_node, method_name)
    @local_vars_in_privately_node.add_from_tree(body_node)
  end
  # -- fin nuevo fragmento --

  if is_rbx_asm_node?(receiver_node, method_name)
    @currently_inside_rbx_asm_block = true
    processed_node = super
    @currently_inside_rbx_asm_block = false
    return processed_node
  end

  super
end
```

La definición de **is_rbx_privately_node?** será análoga a **is_rbx_asm_node?**, vista anteriomente.

La variable de instancia *@local_vars_in_privately_node* será inicializada al construir el *rewriter*:

```
def initialize(a_binding=nil)
  super

  @local_vars_in_privately_node = LocalVarsInAST.new
  @currently_inside_rbx_asm_block = false
end
```

La clase **LocalVarsInAST**, representará el conjunto de variables locales, similar a **LocalVarsInScope**. Sin embargo, a diferencia de la solución exhibida para la diferenciación entre variables locales y mensajes sin argumentos, en este caso las variables locales deberán ser obtenidas vía *parsing* y no por el contexto, ya que el bloque analizado no es el mismo que el *proc* originalmente transformado. Veamos a continuación la definición de **LocalVarsInAST**, basada nuevamente en un *AST Processor*.

```
class LocalVarsInAST < Parser::AST::Processor
  def initialize
    @local_vars_in_ast = Set.new
  end

  def add_from_tree(root_node)
    process(root_node)
  end

  def on_lvasgn(node) # e.g.: local_var = "some_value"
    var_name, _ = *node
    @local_vars_in_ast.add(var_name)

    on_vasgn(node)
  end

  def include?(var_name)
    @local_vars_in_ast.include?(var_name)
  end
end
```

Como se puede observar, la implementación consiste en acumular las variables locales dentro de un conjunto, a través de implementar el mensaje **on_lvasgn**, específico del recorrido del subárbol relacionado a la definición de una variable local.

El mensaje **add_from_tree** se trata de un simple renombre de **process**, mensaje implementado en **Parser::AST::Processor**, que se encargará de procesar el árbol en cuestión.

Una vez agregadas las variables locales, bastará con contemplar este nuevo caso en **should_transform_to_atomic?**. La versión definitiva de este último entonces será:

```
def should_transform_to_atomic?(send_node)
  _, method_name, *_ = *send_node

  unless super
    return false
  end

  if @local_vars_in_privately_node.include?(method_name)
    return false
```

```
  end

  if is_a_rbx_undefined_method_node?(send_node)
    return false
  end

  if currently_inside_rbx_asm_block?
    return false
  end

  true
end
```

De este modo, se dará soporte para que todas las transformaciones AST realizadas por Rubinius que se aplican sobre el código fuente no interfieran con la transformación de envío de mensajes realizada por **AtomicSendOnRbxRewriter**.

Como conclusión, se puede observar como aquí nos enfrentamos a limitaciones de la implementación, producto del uso de palabras reservadas adicionales del lenguaje. Peor aún, en este caso están enmascaradas como mensajes dentro del código fuente del mismo, lo que conflictúa con el modelo de envío de mensajes e incluso rompe con la semántica de bloques para algunos casos. Otras implementaciones de lenguajes dinámicos como Pharo Smalltalk hacen uso también de sintáxis especial para anotaciones como las *pragmas*, pero en este caso son totalmente distinguibles del envío de mensajes y hasta forman parte del metamodelo del lenguaje.

3.11.3. *AtomicStateAccessRewriter*: Transformación de acceso a estado

Para el caso de la asignación y escritura de variables, el algoritmo resulta bastante más sencillo y menos problemático. Veamos primero como referencia la implementación del procesador que simplemente recorre el árbol de un nodo de asignación sin hacer ninguna transformación:

```
def on_ivasgn(node)
  name, value_node = *node

  node.updated(nil, [
    name, process(value_node)
  ])
end
```

El algoritmo simplemente procesa recursivamente el nodo que contiene el valor a asignar a la variable (*value_node*) y actualiza al nodo.

La transformación a realizar para este caso consitirá en convertir el código de asignación en un nodo de envío de mensaje que envíe el mensaje **__atomic__instance_variable_set** usando como parámetro el nombre de la variable y el valor a asignar (*name* y *value_node* respectivamente). La nueva implementación definida en **ASTAtomicRewriter** será:

```
def on_ivasgn(node) # e.g.: @instance_var = "some_value"
  var_name, value_node = *node

  node.updated(:send, [
      Parser::AST::Node.new(:self),
      :instance_variable_set.to_atomic_method_name,
      Parser::AST::Node.new(:sym, [var_name.to_sym]),
      process(value_node)
  ])
end
```

El caso de lectura de una variable de instancia es prácticamente equivalente, con la diferencia que se trata de un nodo hoja ya que es simplemente un palabra que referencia a la variable de instancia:

```
def on_ivar(node) # e.g.: @instance_var
  var_name = node.children.first

  node.updated(:send, [
      Parser::AST::Node.new(:self),
      :instance_variable_get.to_atomic_method_name,
      Parser::AST::Node.new(:sym, [var_name.to_sym])
  ])
end
```

En este caso, para estas transformaciones no es necesario definir ningún comportamiento particular para Rubinius, respecto a implementación canónica aquí presentada.

3.11.4. *RemoveRbxPrimitivesRewriter*: Remoción de nodos primitivos (Rubinius)

Como se presentó anteriormente, otro de las transformaciones que deseamos realizar consiste en eliminar aquellos nodos AST que invocan a primitivas en Rubinius, de modo de evitar su ejecución (vía la transformación AST que forma parte del *kernel*).

Nuevamente, haremos uso de un procesador AST para recorrer el árbol y encontrar dichos nodos. Sin embargo, para poder borrar los nodos, el procesador no lo permite. Para esto, usaremos la clase **SourceRewriter** provista por la librería que permite reescribir el código fuente, eliminando nodos, entre otras operaciones. Veamos la definición del nuevo procesador, que utiliza una instancia de **SourceRewriter**.

```
class RemoveRbxPrimitivesRewriter < Parser::AST::Processor
  def initialize(source_rewriter)
    @source_rewriter = source_rewriter
  end
```

```
  def do(ast)
    process(ast)
    @source_rewriter
  end

  def on_send(node) # e.g.: obj.a_message
    if is_a_rbx_primitive_call_node?(node)
      @source_rewriter.remove(node.location.expression)
    end

    super
  end

  def is_a_rbx_primitive_call_node?(node)
    receiver_node, method_name, *_ = *node

    !receiver_node.nil? && receiver_node.children[1] == :Rubinius
      && method_name == :primitive
  end
end
```

La implementación consiste en identificar los nodos en cuestión sobreescribiendo **on_send**, utilizando la misma técnica que para las soluciones anteriores. Para la remoción del nodo en cuestión, utilizaremos el mensaje **remove** sobre el *source rewriter*.

Notar que sólo se removerán primitivas invocadas con *Rubinius.primitive* y no con *Rubinius.invoke_primitive*. Al realizar una investigación sobre su uso, se detecto que esta última se utiliza para invocar primitivas cuyo resultado luego se asigna a una variable. A diferencia de *Rubinius.primitive*, estas primitivas no frenan la ejecución del código Ruby sino que se comportan como si fuesen un envío de mensaje más (a pesar de que igualmente no se trata de un mensaje) en cuanto al flujo de ejecución. Por lo tanto, estas primitivas no deben ser removidas.

3.12. Código fuente de *procs* y métodos: Representación y transformaciones

Restan presentar 2 entidades fundamentales: Los objetos que representarán el código fuente de *procs* y métodos, y **SourceCodeReader** que se encargará de realizar la extracción del código de un bloque o un método a partir del archivo de código fuente donde se encuentra definido. Se abordarán dichas implementaciones, junto con un análisis de las limitaciones encontradas a continuación.

3.12.1. La jerarquía *ExecutableSourceCode*: Representando y transformando código fuente de *procs* y métodos

Si bien a simple vista el código fuente no es más que una cadena de caracteres (es decir, un objeto *string*), se optó por modelar dicho concepto ya que el código fuente se puede también representar como un árbol AST y puede ser transformado.

Para la conversión de *string* a árbol AST se hará uso de la librería **parser** mencionada previamente (utilizada para la manipulación del árbol AST en los *rewriters* implementados). Dado que se necesitará también convertir el árbol AST nuevamente a *string*, se hará uso de otra librería llamada **unparser** para este trabajo.

Existe comportamiento compartido entre las instancias de código fuente de un *proc* y un método. Más aún, comparte la parte central de su interfaz que se trata de permitir aplicar una transformación AST, y representarse como árbol AST y como *string* simultáneamente. La diferencia central radicará en cómo se obtiene el nodo AST del código fuente del *proc* o método a partir de la lectura de la expresión completa de su declaración. Veamos a continuación la definición completa de **ExecutableSourceCode**:

```
class ExecutableSourceCode
  def initialize(obj)
    @obj = obj
    @ast = Parser::CurrentRuby.parse(parse_source_code)
  end

  def to_s
    Unparser.unparse(@ast)
  end

  def to_ast
    @ast
  end

  def apply_ast_transformation!(ast_processor)
    @ast = ast_processor.process(to_ast)
  end

  def apply_source_rewrite!(source_rewriter)
    new_source_code = source_rewriter.process
    @ast = Parser::CurrentRuby.parse(new_source_code)
  end

  def new_source_rewriter
    buffer = Parser::Source::Buffer.new('(method_buffer)')
    buffer.source = to_s
    Parser::Source::Rewriter.new(buffer)
  end

  private

  def parse_source_code
    exp_src = get_source_code_expression
    parsed_node = Parser::CurrentRuby.parse(exp_src)
    source_code_node = find_source_code_node(parsed_node)
    Unparser.unparse(source_code_node)
```

```
  end

  def get_source_code_expression
    if !@obj.source_location.nil?
      file, line = @obj.source_location
      src_reader = SourceCodeReader.new
      src_reader.get_src_of_first_expression_in(file, line)
    else
      raise "Source code is not available for #{@obj}"
    end
  end

  def find_source_code_node(parsed_node)
    raise NotImplementedError
  end
end
```

El mensaje central de la implementación es **parse_source_code**. Como se observa, en primer lugar obtiene la expresión del código fuente a parsear enviando **get_source_code_expression**. Este mensaje a su vez obtendrá la ubicación del código fuente del ejecutable (mediante el mensaje **source_location**, polimórfico en **Proc** y en **UnboundMethod**, definido como parte del lenguaje), para luego obtener la primer expresión hallada allí presente utilizando una instancia de **SourceCodeReader** (analizado posteriormente). A continuación se convierte la expresión a AST y luego se invoca al método **find_source_code_node** que se encargará de encontrar el nodo relevante. Finalmente se vuelve a convertir el código a *string* antes de ser retornado, mediante el mensaje **unparse**.

Tal como se observa, el método **find_source_code_node** es abstracto y deberá ser definido en cada subclase.

Veamos a continuación la definición de **ProcSourceCode**, que simplemente implementa este último método:

```
class ProcSourceCode < ExecutableSourceCode
  private

  def find_source_code_node(parsed_node)
    if parsed_node.type == :block
      block_node = parsed_node
    else
      block_node = parsed_node.children.find do |child|
        child.is_a?(Parser::AST::Node) && child.type == :block
      end
      unless block_node
        raise "Could not find definition for #{@obj}"
      end
    end
    block_node.children[2]
  end
```

```
end
```

En el caso de un *proc*, dependiendo de como se haya creado al mismo, el nodo hallado será directamente el bloque en cuestión, o sino será uno de sus nodos hijo. Una vez ubicado el nodo que representa al bloque, se retornará el nodo de su cuerpo (siempre ubicado en la tercera posición) que será el código fuente buscado.

Para el caso del código fuente de un método, el algoritmo sería en principio bastante similar. Sin embargo, ocurre que Ruby posee un mecanismo de generación dinámica de métodos mediante los mensajes de clase **:attr_accessor**, **:attr_reader** y **:attr_writer**, que genera automáticamente métodos de acceso y escritura de variables de instancia (análogo a los *getters* y *setters*). Dado que en este caso el código fuente de los métodos asociados no se encuentra explícitamente presente, habrá que generar el mismo también dinámicamente. Afortunadamente, los mismos tiene siempre una misma estructura sencilla, por lo que no resulta complicado generar su árbol AST manualmente utilizando las herramientas de la librería **parser**. Dado que el código fuente de esta generación automática es relativamente extenso y no forma parte del objetivo central del trabajo, se omitirá su análisis.

La implementación final de **find_source_code_node** en **UnboundMethodSourceCode** será:

```
def find_source_code_node(parsed_node)
  if is_an_attr_def_node?(parsed_node)
    generate_attr_method_node
  else
    method_def_node = search_for_method_def_node(parsed_node)
    unless method_def_node
      raise "Could not find definition for #{@obj}"
    end
    method_def_node
  end
end
```

Cómo acabamos de mencionar, en primer lugar se verá si se trata de un nodo de tipo '**:attr_**' y en tal caso se generará el árbol manualmente mediante **generate_attr_method_node**. En caso de que no sea así, se enviará el mensaje **search_for_method_def_node**, que tendrá un comportamiento similar al comportamiento de **ProcSourceCode#find_source_code_node** presentado previamente.

Por otra parte, durante la inicialización de **UnboundMethodSourceCode** se realizará una transformación adicional al código fuente, enviando el mensaje **remove_method_receiver_if_prese** Este último método será el encargado de remover del código fuente todo receptor explícito que forme parte del encabezado del método: En Ruby, la sintáxis del *keyword* '*def*' permite explicitar un objeto donde será implementado el método, que por defecto se inserta como un método de instancia de la clase que engloba. Por ejemplo, si se define un método mediante 'def self.new_method', el nuevo método será anexado a la clase de **self** (donde el valor de **self** normalmente corresponde a la clase sobre la que se está definiendo el método, por lo que el método será anexado a la clase de la clase en cuestión, lo que lo convertirá en un método de instancia de clase). Dado que el código fuente del método es en principio

independiente de donde se realice la inserción del método, removeremos este receptor de la definición.

Para realizar esta transformación, haremos uso de una facilidad de la librería **parser** llamada **SourceRewriter**, que permite editar fácilmente el código en forma de *string* en lugar de como árbol AST. Dado que deseamos eliminar un fragmento de código, esta opción resultará práctica. Para generar un nuevo *source rewriter* acudiremos al mensaje **new_source_rewriter** definido en ObjectSourceCode, exhibido anteriomente. Por su parte, la aplicación de la reescritura se hará mediante **apply_source_rewrite!**, también presentado.

Por último, las instancias de **UnboundMethodSourceCode** también deberán contar con el método **change_name_in_definition!** que permite alterar el nombre del método embebido en el encabezado de la definición. Como se vió anteriormente, el mismo será usado al momento de definir un método atómico, alterando el nombre del mismo por su versión atómica. Para lograr el renombre, se hará también uso de un objeto de tipo *source rewriter*.

3.12.2. *SourceCodeReader*: Obtención del código fuente de una expresión

Tanto los *procs* como los métodos de Ruby cuentan con el método **source_location** para obtener el nombre del archivo y la línea donde se encuentra definido un método. Lamentablemente, este es el único mecanismo que poseen para poder conocer el código fuente del mismo.

Debido a la sintáxis del lenguaje, este mecanismo no será suficiente para hallar en todos los casos el código deseado. Veamos el siguiente código de ejemplo:

```
inc1 = Proc.new { |a| a + 1 }; dec1 = Proc.new { |a| a - 1 }
```

Al enviar el mensaje **source_location**, tanto *inc1* como *dec1* darán el mismo archivo y número de línea. Al realizar luego el *parsing* de esa línea, no será posible saber cuál de los dos cuerpos de bloque corresponde a cada uno. Si bien el uso de ";"para incluir 2 sentencias en una misma línea es desaconsejado y poco habitual en la práctica, es parte de la sintáxis válida y aquellos *procs* que estén definidos de este modo no podrán ser transformados atómicamente, por no poder contar con su código fuente.

Un problema similar se presenta para aquellos métodos o *procs* que están definidos mediante mecanismos de metaprogramación como **class_eval**. En este caso, **source_location** retornará la línea donde se invocó a **class_eval**. Sin embargo, dado que el código es un *string* que puede estar definido dinámicamente por variables *inline* será imposible (o al menos órdenes de magnitud más difícil) encontrar un algoritmo para *parsear* el caso general.

Exceptuando estos casos, podremos extraer el código fuente de cualquier *proc* o método con un algoritmo goloso relativamente sencillo, compuesto por los siguientes pasos:

1. Obtener todas las líneas del archivo fuente a partir del número de línea indicado por **source_location**.

2. Ir intentando construir una expresión concatenando línea por línea las líneas obtenidas. Se comienza por la primer línea y se consulta si la misma es una expresión

completa. Si no lo es, se continúa concatenando la próxima línea hasta encontrar la primera expresión completa. Si lo es, se retorna la expresión hallada.

Una expresión es considerada completa cuando no finaliza con un caractér que indique una línea incompleta (caracteres ',' y '/' que permiten definir expresiones de múltiples líneas) y además es una expresión válida del lenguaje.

Esta idea fue originalmente extraída desde el código fuente de una librería de terceros llamada **method_source** (citar fuente?). Dada la sencillez del algoritmo y la falta de mantenimiento de la librería, se optó por implementar una versión propia siguiendo la misma idea. A continuación se presenta el código fuente de la clase **SourceCodeReader**:

```
class SourceCodeReader
  def get_src_of_first_expression_in(file, linenum)
    all_lines = IO.readlines(file)
    lines_from_linenum_till_end = all_lines[linenum-1..-1]
    extract_first_expression(lines_from_linenum_till_end)
  end

  private

  def extract_first_expression(src_code_lines)
    expression = ''
    src_code_lines.each do |v|
      expression << v
      return expression if complete_expression?(expression)
    end
    raise SyntaxError, 'unexpected $end'
  end

  def complete_expression?(src_code)
    !ends_with_incomplete_line_marker?(src_code) &&
        valid_expression?(src_code)
  end

  def valid_expression?(src_code)
    begin
      catch(:valid) do
        eval("BEGIN { throw :valid }\n#{src_code}")
      end
      true
    rescue IncompleteExpressionError
      false
    end
  end

  def ends_with_incomplete_line_marker?(src_code)
    src_code =~ /[,\\]\s*\z/
```

```
  end
end
```

El aspecto más interesante es probablemente la implementación de **valid_expression?**: Para poder validar la expresión sin ejecutar su código, se evaluará su código fuente utilizando **eval** pero anteponiendo el lanzamiento de una excepción, de modo tal prevenir la ejecución. Mediante el método **catch** capturamos la excepción y damos por válida la expresión. En caso de que no sea una expresión válida, la evaluación de código lanzará una excepción de tipo **SyntaxError**. En principio, en tal caso, se cortaría la ejecución. Dado que el comportamiento esperado es continuar leyendo las siguientes líneas, se capturará esta excepción y se retornará *false*.

Notar que en lugar de atrapar excepciones de tipo **SyntaxError**, se atrapará solamente las **IncompleteExpressionError**. Dicha clase de excepción será una subclase de **SyntaxError** (definida como parte del trabajo) que sólamente contemplará aquellos errores sintácticos que son producto de haber leído una expresión incompleta que luego podrá ser potencialmente válida si se continúa concatenando las siguientes líneas de código.

Si bien esta estrategia resulta efectiva en la práctica, posee varias **limitaciones**:

- Como se mencionó en el ejemplo más arriba, retornará todas las expresiones completas que se hallen en una misma línea sin distinción entre ellas.
- La implementación (sobre todo **IncompleteExpressionError**) depende del parser del lenguaje. Existen diferencias entre los errores sintácticos entre las distintas implementaciones de Ruby y no hay documentación oficial al respecto. Cualquier cambio en el parser respecto a esto podría implicar tener que actualizar esta implementación.
- Por el punto anterior, se desprende que el concepto de expresión completa no está bien definido por lo que podrían existir casos donde el resultado no sea el esperado.

Aún bajo estas consideraciones, los casos de prueba implementados como parte del trabajo muestran un comportamiento correcto para ejemplos reales (aunque sencillos).

4. PROBLEMÁTICAS DE IMPLEMENTACIÓN ESPECÍFICAS DE RUBINIUS

A lo largo de la investigación y el desarrollo de la implementación fueron apareciendo algunas problemáticas específicas en relación a la compatibilidad con Rubinius. Las mismas no hacen a lo central del trabajo, por lo que se optó por comentarlas aparte. En algunos casos se trata de falencias de la implementación, mientras que en otro de particularidades no necesariamente erróneas.

Antes de comenzar, es interesante notar que, con excepción al primer punto (*bugs* de la implementación), se trata de hallazgos directamente relacionados con el hecho de que Rubinius se encuentra implementado sobre Ruby mismo. Este hecho tuvo como consecuencia que las pruebas (casos de test) implementadas impliquen la transformación de gran cantidad de código de las clases *core* del lenguaje y por ende un árbol de ejecución teórico mucho más profundo que en la ejecución bajo MRI. Esto ayudó a encontrar distintas problemáticas de implementación que posiblemente hubiesen pasado desapercibidas.

Bugs en source_location y *workarounds*

En primer lugar, se encontró un problema con la implementación de **source_location** al momento de retornar la ubicación del código *core* de Rubinius. Se utilizan rutas relativas en lugar de absolutas como ocurre en MRI. Se implementó un *workaround* simple hasta que se implemente una solución en la distribución oficial (el problema fue reportado).

Un segundo error encontrado, también relacionado a **source_location**, es que este último no reporta correctamente la ubicación de métodos que se encuentren definidos implícitamente vía *accessors* (mensajes de clase **attr_***), como sí ocurren en MRI. En este caso, el *workaround* fue algo más complejo. En primer lugar, se alteró el mensaje **UnboundMethodSourceCode#get_source_code_expression** para que, en caso de que se trate de un método de tipo **accessor**, se retorne directamente el *string* correspondiente. Por ejemplo, si el método buscado se llama *method* y es de tipo *attr_reader*, se retornará "*attr_reader :method*". La dificultad radicó en poder distinguir qué métodos eran de este tipo y cuales no. Para ello, se implementó los siguientes mensajes en **UnboundMethod** (sólo para Rubinius):

```
def is_attr_method?
  self.executable.kind_of? Rubinius::AccessVariable
end

def is_attr_writer?
  is_attr_method? && self.executable.arity == 1
end

def is_attr_reader?
  is_attr_method? && self.executable.arity == 0
end
```

Afortunadamente, dado que la implementación se encuentra sobre Ruby, el objeto **UnboundMethod** cuenta con capacidades reflexivas adicionales a las documentadas. Cuenta en particular con un colaborador *executable* que permite distinguir estas variantes.

Implementación de Module#class_eval_with_kernel_support

Cómo se comentó al momento de ver la extensión realizada a **Module** para definir métodos vía **class_eval**, para Rubinius se implementó la variante **class_eval_with_kernel_support**, cuyo código vemos a continuación:

```
def module_eval_with_kernel_code_support(string,
    filename="(eval)", line=1)
  string = StringValue(string)
  filename = StringValue(filename)

  cs = Rubinius::LexicalScope.new self,
      Rubinius::LexicalScope.of_sender

  binding = Binding.setup(Rubinius::VariableScope.of_sender,
                          Rubinius::CompiledCode.of_sender,
                          cs)

  c = KernelEvalCompiler
  be = c.construct_block string, binding, filename, line

  be.call_under self, cs, true, self
end

alias_method :class_eval_with_kernel_code_support,
    :module_eval_with_kernel_code_support
```

Nota al margen: La utilización *alias method* es para mantener consistencia con la implementación original, dado que **class_eval** es un alias de **module_eval**.

El código fuente es prácticamente una réplica (simplificada) del método **module_eval** original, sustituyendo el compilador original por **KernelEvalCompiler**, cuya definición vemos a continuación:

```
class KernelEvalCompiler < Rubinius::ToolSets::Runtime::Compiler
  def self.compile_eval(string, variable_scope, file="(eval)",
      line=1)
    if ec = @eval_cache
      layout = variable_scope.local_layout
      if code = ec.retrieve([string, layout, file, line])
        return code
      end
    end
```

```
    compiler = new :eval, :compiled_code

    parser = compiler.parser
    parser.root CodeTools::AST::EvalExpression
    parser.default_transforms
    parser.enable_category :kernel # UNICA LINEA AGREGADA
    parser.input string, file, line

    compiler.generator.variable_scope = variable_scope

    code = compiler.run

    code.add_metadata :for_eval, true

    if ec and parser.should_cache?
      ec.set([string.dup, layout, file, line], code)
    end

    return code
  end
end
```

Esta clase simplemente sobrecarga el mensaje de clase **compile_eval** con una implementación idéntica a la original pero que habilita las transformaciones AST a nivel de *kernel* vía el envío de mensaje **enable_category** al parser.

Notemos que nos encontramos tratando con piezas fundamentales del lenguaje como su parser y compilador, y aún así no hubo que recurrir en modificaciones a nivel VM. Sin embargo, como crítica se puede sostener que la habilitación de las transformaciones AST (o al menos el uso de un compilador alternativo) debería resultar más sencillo y sin incurrir a tanta repetición de comportamiento.

Uso de meta-programación en la librería *thread_safe*

La librería *unparser* utilizada depende indirectamente de otra librería llamada *thread_safe*. Esta última, hace uso de metaprogramación (vía **class_eval**, durante la carga de la librería) para definir métodos en **Hash** y **Array** en forma dinámica, particularmente en un caso aplicado específicamente a Rubinius. Dado que la implementación de STM presentada no soporta la transformación de métodos generados dinámicamente, esto hace que las pruebas realizadas sobre **Array** y **Hash** no se ejecuten en forma satisfactoria. En consecuencia, se implementó la siguiente extensión a **ThreadSafe**.

```
module ThreadSafe
  class Hash < ::Hash;
  end
  class Array < ::Array;
  end
```

```
[Hash, Array].each do |klass|
  klass.superclass.instance_methods(false).each do |method|
    klass.class_eval <<-RUBY_EVAL, __FILE__, __LINE__ + 1
        def #{method.to_atomic_method_name}(*args)
          @_monitor.synchronize { super }
        end
      RUBY_EVAL
    end
  end
end
```

La idea consistió en generar las versiones atómicas de los métodos también mediante meta-programación (notar el envío de mensaje **to_atomic_method_name**). Dado que el código fuente original consiste en simplemente encapsular las llamadas bajo un monitor, se mantuvo el mismo código para las versiones atómicas.

Conflicto con la librería *ice_nine*

Otras de las librerías que entraron en conflicto fue *ice_nine*, también una dependencia indirecta de *unparser*. Al cargar la librería en Rubinius (versión 3.69), se lanza una excepción **NoMethodError**, cuestión que no ocurre en MRI. Al reportar este error al equipo de Rubinius, uno de los desarrolladores principales indicó que es más bien un *bug* propio de la librería y que esta última esta rota a nivel diseño y probablemente surjan incompatibilidades a futuro, aunque esta se solucione. La librería se entromete con el estado interno de estructuras fundamentales como **Hash**, lo que genera incompatibilidades con Rubinius ya que este último, a diferencia de MRI, expone su estado y su implementación hacia el desarrollador.

Al inspeccionar la librería *unparser* y notar que el uso de esta dependencia no es crítico, se optó por realizar un *fork* de la librería, removiendo esta dependencia. La implementación de STM utilizará este *fork* en lugar de la librería original, logrando que esta se cargue ahora correctamente.

5. CONCLUSIONES

La capacidad de alterar el comportamiento de las clases provistas como parte de la distribución del lenguaje fue el primer indicio de que estar frente a un lenguaje pensado para ser extendido. La re-apertura de definición de una clase, permitiendo agregar o sobrescribir sus mensajes, es un distintivo del lenguaje que permite posible este tipo de extensiones, sin tener que realizar un *fork* del proyecto original.

Como vimos durante el análisis, las capacidades reflexivas del lenguaje en conjunto con su meta-modelo expuesto en términos de objetos fueron fundamentales para poder realizar la implementación. Listamos a continuación algunas de las características más sobresalientes explotadas durante la implementación.

- Bloques y métodos como objetos (**Proc** y **UnboundMethod**).
- El mensaje **method_missing** invocado automáticamente al no hallarse la definición de un método fue clave para la generación de métodos en tiempo de ejecución.
- Un modelo uniforme de variables de instancia (en Python el concepto de variable de instancia no está claramente definido), complementado con mensajes para leerlas, escribirlas y recorrerlas (**instance_variable_set**, **instance_variables_get** e **instance_variables**).
- **define_method** en **Module** como facilidad de metaprogramación para la definición dinámica de métodos, en conjunto con **instance_methods** (e **instance_method**) para la obtención de los métodos de una clase.
- El envío dinámico de mensajes vía **send** en **Object**.
- La posibilidad de evaluar código fuente dinámicamente vía **class_eval**.
- El *singleton* **ObjectSpace** permitió poder encontrar todas las subclases de una clase.
- La posibilidad de obtener el contexto de ejecución de un bloque o método a través del objeto **Binding** fue fundamental para la generación del bloque transformado, así como para resolver el problema de *parsing* de código sin contexto.

Sin embargo, por momentos se notaron falencias de expresividad del meta-modelo. Por ejemplo, al momento de generar un nuevo método, el lenguaje no cuenta con la posibilidad de crear nuevas instancias de **UnboundMethod** directamente. Para lograr esto, hubo que utilizar la evaluación directa del código fuente (un *string*) dentro del contexto de una clase (**class_eval**). A su vez, el código fuente no forma parte del meta-modelo y, si bien pudo ser agregado, no existen medios que provean una forma confiable de obtenerlo. La única facilidad presente fue el mensaje **source_location** que, como se observó, tiene importantes limitaciones. Otro falencia radicó en no contar con un mensaje de copia tipo *shallow copy*, cuya implementación resultó problemática.

Respecto al meta-modelo del lenguaje en sí mismo, algunas particularidades del mismo dificultaron la implementación. En primera instancia, el concepto de clase *singleton* agregó

una complejidad adicional inesperada al comienzo. Si bien en principio no interceden en el algoritmo de *method lookup* (ya que se las trata como una clase más), estas clases son únicas por cada objeto, por lo que la copia no resulta posible sin tener que alterar el lenguaje a nivel VM (o invocando primitivas existente, en el caso de Rubinius). La existencia de estas entidades implica que el concepto de estado por objeto no está limitado a sus variables de instancia únicamente sino también a estas clases anexas. Además, el hecho de que se puedan generan dinámicamente (por ejemplo, al agregar un método a un objeto) le agrega un nivel más de complejidad en caso de querer soportar este tipo de cambios en el contexto de una transacción.

Otro aspecto que agrega complejidad al lenguaje es el concepto de **módulo** y sus usos asociados. Los módulos pueden hacer las veces de *mixins* en Ruby. A diferencia de otros modelos como los *traits*, los *mixins* pueden agregar estado a objetos que previamente no lo tuvieran[13]. En principio, esto no sería problema ya que las variable de instancia se consultan por objeto, independientemente de donde fueron definidas. Sin embargo, al momento de intentar resolver la problemática de objetos primitivos, la presencia de módulos dificultó el tratamiento de estos últimos, como se observó previamente.

En relación al lenguaje en sí mismo y su sintaxis, se puede concluir que la presencia de gran cantidad de *keywords* aumenta la complejidad de implementaciones que utilizan la característica de transformación de código. Dado que todo código deberá ser potencialmente transformado, aquellas palabras clave del lenguaje que accedan o modifiquen estado deberán tener un tratamiento especial. Por ejemplo, la definición de un método mediante la palabra clave *def* implica el cambio de estado de una clase. Si bien en este trabajo no se puso foco en soportar cambios transaccionales a nivel modelo, en caso de querer contar con una implementación completa, se deberá contemplar aparte este caso. Un recorrido por la documentación permite detectar varios otros *keywords* que deberían tratarse aparte potencialmente: **alias**, **class**, **defined?**, **module** y **undef**. Por otra parte, al querer realizar una *suite* de casos de prueba completa para probar la correcta transformación de código, la cantidad de casos tendrá una correlación lineal con la cantidad de *keywords* del lenguaje (en el mejor de los casos).

Otra problemática sintáctica fue la distinción especial que Ruby hace con los operadores. Los mismos son nombres de métodos válidos pero sin embargo no pueden ser prefijados (ni sufijados) por otra cadena de caracteres. Hubo que recurrir a una solución *ad-hoc* para estos casos, teniendo que listarlos explícitamente.

En lo que respecta a la implementación del lenguaje, en ambas implementaciones analizadas, la principal dificultad surgió a raíz de la existencia de mensajes y objetos primitivos. En el caso de MRI, esto abarca absolutamente toda la librería *core* del lenguaje mientras que en Rubinius, abarca a un subconjunto minoritario de objetos y mensajes.

La primera consecuencia negativa de que MRI se encuentre implementado en C se observó al momento de intentar implementar la copia de estado de los objetos. Los objetos primitivos no cuentan con variables de instancia, por lo que no resulta posible en principio copiar su estado ni accederlo para compararlo directamente. Se propuso ciertas heurísticas para algunos tipos básicos como **Array**, **Hash** y **String** que logran copiar y comparar utilizando otros mensajes públicos. Sin embargo, aún queda por resolver la copia de la clase *singleton*, que en principio no parece posible sin tener que recurrir a modificar la implementación en C, ya que se necesita modificar el comportamiento de las clases *singleton* a un nivel fundamental.

La siguiente barrera ocurrió al momento de realizar la transformación de los métodos de objetos primitivos. Dado que los métodos se encuentran implementados en C, no fue posible obtener su código fuente y transformarlos. Sin embargo, el hecho de que los objetos fueran íntegramente primitivos (implementados como estructuras de C), paradójicamente simplificó la problemática. Dado que el objeto no cuenta con colaboradores internos, no hay mensajes adicionales para transformar. Por lo tanto, alcanza con reenviar el mensaje original a una copia del objeto, para que la mutación ocurra sobre la copia. De todos modos, deberá explicitarse la lista de clases primitivas, que deberá ser mantenida en las sucesivas versiones del lenguaje. Por otra parte, se observó que ciertos objetos potencialmente mutables como los *procs* o instancias de **Thread** no pueden ser copiados o su copia tendrá efectos colaterales no deseados. En consecuencia, la técnica de reenvío sobre copia no servirá en todos los casos, por lo que esta problemática no se encuentra totalmente resuelta.

Con respecto a Rubinius, dado que se encuentra implementado sobre Ruby, en principio se conjeturó que debería ser más sencilla la implementación de STM. En lo que respecta a la copia y comparación de objetos, también existieron inconvenientes. Si bien todos los objetos que forman parte de la distribución oficial cuentan con variables de instancia, estos colaboran con objetos más primitivos que sientan las bases de la implementación. Además de los objetos inmutables, existen clases como **ByteArray** y **Tuple**, utilizados para modelar objetos de tamaño variable, que son mutables y no cuentan con variables de instancia. Como vimos en su momento, esto implica una falta de uniformidad en cuanto al acceso a estado para los objetos. Por suerte, el lenguaje cuenta con la posibilidad de invocar a primitivas de copia ya implementadas que permitirán copiar el estado de cualquier objeto e incluso su clase *singleton*. Sin embargo, no hay primitivas para poder acceder al estado de estos objetos, independientemente de su clase. Como conclusión, podemos decir que si bien se pudo llegar a una solución más completa que en el caso de MRI, deficiencias en el diseño de estas clases primitivas impidieron lograr una implementación robusta y clara de copia y comparación de estado, como la lograda sobre Smalltalk.

En relación a la transformación de métodos, en principio Rubinius presenta una clara ventaja ya que gran parte de los métodos de la librería *core* se encuentran implementados sobre Ruby. Al igual que ocurre con MRI, existen clases de objetos mutables donde todos sus mensajes son primitivos (como **ByteArray** y **Tuple** recién mencionados). Para estos casos, también será posible reenviar todos sus mensajes en su forma original hacia una copia de estos objetos. Y también existen primitivas de infraestructura que no deberían ser transformadas. Sin embargo, a diferencia de MRI, existen clases mutables que cuentan con mensajes primitivos y a su vez colaboradores internos como el caso de **String**. Esto hace que estos mensajes deban ser reescritos y/o analizados individualmente (quizás no necesiten ser ejecutados en el contexto transaccional), lo que implica un grado de complejidad adicional respecto a la implementación MRI. Sin embargo, dado que todos los objetos se corresponden con una clase definida en Ruby, todos estos problemas resultan al menos abordables sin tener que pensar de inmediato en cambios a nivel VM, como ocurriría con MRI. Bastará con, en primer lugar, reimplementar una versión en Ruby para aquellas primitivas para las que cuales esto sea posible. Luego, las primitivas esenciales deberán ser clasificadas entre aquellas que directamente no se deban (o necesiten) transformar, y aquellas que pueden ser re-enviadas a través de la copia de trabajo.

En relación a las primitivas y Rubinius, también viene al caso mencionar la forma en que se implementó la invocación a las mismas. Como vimos, en lugar de utilizar una sintaxis

adicional por fuera del lenguaje Ruby (como los *pragmas* de Smalltalk), utiliza igual sintaxis que un envío de mensaje (cuando en realidad no lo es tal) para realizar operaciones primitivas. Al momento de implementar la reescritura de los métodos, está decisión implicó tener que contemplar estas macros de compilador en forma especial durante la etapa de *parsing*, lo que hizo significativamente más compleja la implementación. Por otro lado, la solución resulta fuertemente acoplada a las particularidades de estas macros, que podrían cambiar.

Como conclusión general, tal como se podía esperar, un lenguaje más complejo desde el punto de vista sintáctico (mayor cantidad de palabras clave, operadores distinguidos) con un meta-modelo también más sofisticado (módulos, clases *singleton*, métodos alias, *accessors*) en comparación con Smalltalk, resultaron en una implementación más compleja y con más limitaciones.

En cuanto a una comparación entre ambas implementaciones, podemos afirmar que Rubinius posee mayor potencial respecto a la posibilidad de poder contar en un futuro cercano con una implementación totalmente robusta y completa para el lenguaje, que haga uso de esta técnica. Sin embargo, decisiones de diseño de la implementación como la falta de un protocolo uniforme de acceso al estado de un objeto o la sintaxis utilizada para invocar a primitivas, dificultaron la implementación notablemente.

Respecto a la problemática de las primitivas, dejando de lado el tema sintáctico recién comentado, Rubinius cuenta con el mismo problema que Smalltalk, donde cada primitiva debe ser analizada y abordada manualmente. Tanto Smalltalk como Rubinius no proveen reflexividad a nivel de primitivas. Si bien Smalltalk modela los *pragmas*, esto se trata de una sintaxis especial para agregar meta-datos a un método, algo más general que el concepto de primitiva. Incorporar las primitivas al meta-modelo permitiría por ejemplo distinguir entre aquellas primitivas que trabajan con la memoria del objeto, de aquellas que interactúan con un dispositivo de E/S. Esta capacidad permitiría tratar distintas clases de primitivas de distinta forma, de modo de re-enviar el mensaje original a la copia o simplemente continuar la ejecución sin aplicar ninguna transformación según cada caso. Esto haría que la implementación sea independiente de las primitivas que utilice la VM.

6. TRABAJO FUTURO

Existen múltiples áreas para expandir el trabajo aquí presentado. Quedaron algunas problemáticas abiertas por resolver, centrales al objetivo del trabajo. Además, existen otros puntos donde se podría mejorar, profundizar y extender las capacidades de la solución propuesta.

Copia y comparación de estado robusta para MRI y Rubinius (sólo comparación)

La primera problemática importante abierta consiste en implementar una mecanismo de copia y comparación de objetos en C para la implementación MRI, que cubra los objetos más primitivos y copie también la clase *singleton* de los objetos. Para el caso de Rubinius, habría que implementar una nueva primitiva en C++ que permita comparar el estado de dos objetos, de modo de evitar tener que tratar cada objeto de tamaño variable en forma particular. A primera vista no parecen extensiones muy complejas ni grandes de realizar dada la existencia de métodos de copia primitivos como referencia, pero sin duda involucrarán la necesidad de un entendimiento bastante más profundo acerca de ambas implementación, a más bajo nivel.

Soluciones al problema de las primitivas

La segunda problemática importante radica en la existencia de los mensajes y objetos primitivos y la dificultad de aplicar la técnica de transformación de código.

Para el caso general, un enfoque posible consistiría en lograr que todos los objetos primitivos (o una mayor cantidad) sean inmutables. Se anunció que la próxima versión mayor oficial de Ruby (3.0) tendrá como novedad que las instancias de **String** serán inmutables[14], por lo que sería interesante un trabajo en esta dirección. Existen también librerías código abierto que proveen alternativas inmutables para estructuras básicas para Ruby[15]. Los objetos inmutables, por su naturaleza, evitan directamente el problema de la concurrencia, por lo que una solución por este camino seguramente simplifique mucho la implementación.

Otro camino menos radical sería mantener estructuras fundamentales mutables como los arreglos pero agregar capacidades reflexivas al lenguaje que permitan diferenciar los inmutables de los mutables, tal como se propuso al abordar esta temática (ver sección 3.9). Un enfoque más artesanal y simple sería profundizar la idea propuesta de clasificar manualmente todas las clases *core* del lenguaje como clases inmutables, mutables con y sin colaboradores internos.

Otro trabajo a realizar sobre las primitivas consiste en distinguir las primitivas de interfaz con el Sistema Operativo o dispositivos de Entrada/Salida, de las primitivas que lean o escriban el estado de un objeto. Nuevamente, se puede optar por un análisis manual de las implementaciones ya existentes o proponer cambios profundos que permitan identificar este tipo de primitivas automáticamente.

En el caso de Rubinius, tanto se trata de un análisis manual como de efectuar cambios que impliquen la distinción automática de los distintos tipos de objetos y mensajes

primitivos agregando capacidades reflexivas al lenguaje, posiblemente resulten más fáciles de abordar que en el caso de MRI. En Rubinius, si bien existen muchas invocaciones primitivas, se trata de puntos de la implementación acotados y fácilmente identificables, mientras que en MRI se trata de rever la implementación prácticamente en su totalidad.

Por último, otro trabajo interesante relacionado sería convertir las invocaciones a primitivas como una extensión del lenguaje Ruby, en lugar de utilizar la misma sintaxis existente para un envío de mensaje común.

Soporte para *keywords* adicionales de cambio de estado

Como vimos, existen *keywords* del lenguaje que implican potencialmente la consulta o modificación del estado de un objeto. Por ejemplo, el uso del *keyword* ***def*** implica la definición de un nuevo método, lo que altera el estado de una clase. Al no tratar de un mensaje, el mismo no será enviado en forma atómica, por lo que una definición de un método en un contexto transaccional será registrado en la clase original, en lugar de una copia de esta.

Una forma de atacar este problema consiste en implementar un *rewriter* AST adicional que emita los mensajes equivalentes para este tipo de palabras clave, cuando se esté en un contexto de transformación de código. Su implementación no debería demandar mayores dificultades.

Un recorrido por la documentación permite detectar varios otros *keywords* que deberían tratarse aparte potencialmente: **alias**, **class**, **defined?**, **module** y **undef**.

Soporte para operadores compuestos

Debido a que los operadores en Ruby debieron ser tratados en forma particular (ver sección 3.5), se detectaron otros operadores que no fueron contemplados. Se trata de operadores compuestos como "+=". Dado que se trata de una operación combinada que implica también la asignación de una variable, posiblemente deba contemplarse al momento de la transformación, ya que podría tratarse de la asignación de una variable de instancia. Si bien no parece un tema particularmente complejo, no se trata de simplemente extender la lista de operadores con mapeo a nombres.

Soporte para transacciones anidadas

Una posible extensión de la implementación consistiría en soportar bloques atómicos anidados. La única dificultad o consideración a tener en cuenta es que se debería evitar transformar el envío **atomic** en sí mismo, que ocurrirá dentro del contexto transaccional. Más allá de esto, al observar la implementación realizada en Smalltalk, se puede concluir que no habrá mayores dificultades.

Soporte a otras formas de cambios de estado de objetos

Aparte del estado en forma de variables de instancia, Ruby cuenta con otras formas de estado en el contexto de objetos:

- **Variables de clase.** Prefijadas con @@. Uso poco frecuente.

- **Variables globales.** Prefijadas con $. También poco frecuentes y uso no recomendado.
- **Constantes.** Comienzan con mayúscula. En Ruby las constantes, a pesar de su nombre, sí pueden ser modificadas (simplemente se lanzará una advertencia en caso de hacerlo), por lo que debería contemplarse como un posible cambio de estado. Su uso es habitual, particularmente en el *framework* Ruby on Rails.

Agregar soporte para estos tipos de cambios no debería presentar grandes desafíos, ya que la librería *parser* del lenguaje distingue las asignaciones y lecturas de cada tipo de variables en forma individual.

Soporte para otros tipos de cambios

Además de los cambios a nivel objetos, pueden ocurrir otros tipos de cambios de estado del programa, como la escritura a un archivo por ejemplo. Para soportar este tipo de cambios, bastaría con crear una clase nueva de cambios que cuya aplicación consista en la evaluación de un bloque que aplique el cambio deseado. Luego, se deberían implementar manualmente versiones atómicas de los métodos de E/S que agreguen este cambio a la transacción, para que luego sean aplicados en la fase de *commit*.

Generación de métodos atómicos a partir de su *bytecode* original

Una alternativa a explorar consiste en generar las versiones atómicas de cada método a partir de su *byte-code*, en lugar de obtener el código fuente vía **source_location**. En MRI existe el mensaje **InstructionSequence::disasm** que permite obtener el *byte-code* a partir de un bloque de código o método. Una opción es luego construir el código fuente a partir del *byte-code*. Sin embargo, habría que implementar un traductor para tal fin. Otra alternativa consiste en realizar la transformación a nivel byte-code, es decir, cambiar los accesos a estado y los envíos de mensajes con prefijo atómico a nivel de byte-code en lugar de alterar el código fuente.

Lo interesante de este enfoque es que permitiría convertir métodos que se hayan definido por mecanismos de metaprogramación o cuyo código fuente no esté disponible (por ejemplo, por estar implementado originalmente en C).

Análisis de impacto de *performance* y optimizaciones

Otro trabajo interesante consiste en analizar el impacto de *performance* que tendrá la aplicación de la técnica presentada en un contexto real. Se espera que la generación dinámica de código tenga un impacto significativo en el proceso de ejecución, cada vez que un nuevo método deba ser transformado.

Realizar optimizaciones a nivel VM sin duda será necesario para lograr un nivel de *performance* que permita ser usado en escenarios reales.

7. APÉNDICE: IMPACTO EN LA COMUNIDAD RUBINIUS

A lo largo de la investigación realizada, nos hemos puesto en contacto con el equipo de desarrollo de Rubinius, a través de su canal de Gitter[1]. Brian Shirai, uno de los desarrolladores principales del proyecto, ha manifestado interés en el trabajo prácticamente desde el comienzo. Su colaboración resultó de gran ayuda en múltiples oportunidades al momento de entender particularidades de la implementación, suplantando por momentos la falta de documentación.

Durante el desarrollo, se encontraron dos *bugs* relacionados con la implementación de **source_location** que fueron debidamente reportados. Estos son "*UnboundMethod#source_location returns nil for attr_* methods*"[2] y "*UnboundMethod#source_location uses relative paths instead of absolute paths for core code*"[3].

Sobre a la problemática mencionada en la sección 3.7.3 respecto a las variables de instancia de Rubinius (no expuestas por defecto para las clases *core*), se consultó con el equipo de Rubinius por este asunto. En ese momento Rubinius no poseía la capacidad de acceder a dichas variables. Dado nuestro requerimiento, Brian accedió a extender su facilidad de **Mirror**, permitiendo el acceso a estas variables vía **Mirror#instance_fields**. Este cambio fue incluído como parte de la versión 3.52 [4].

Por su parte, Brian colaboró también directamente a este proyecto mediante un *pull request*[5] sugiriendo un cambio menor (en concreto, mover el ejemplo hallado en el *README* a un *script* aparte).

Por último, cabe destacar que el equipo de Rubinius ha mostrado interés en conocer la versión final de este trabajo, deseando poder incluir STM en futuras versiones de la implementación.[6]

Referencias

1. `http://gitter.im/rubinius/rubinius`
2. `https://github.com/rubinius/rubinius/issues/3727`
3. `https://github.com/rubinius/rubinius/issues/3729`
4. `https://github.com/rubinius/rubinius/releases/tag/v3.52`
5. `https://github.com/lucianolev/ruby-stm/pull/1`
6. `https://gitter.im/rubinius/rubinius?at=57d88213b37816b14fb531d1`

Bibliografía

[1] AtomizeJS: Introduction

`http://atomizejs.github.io/`

[2] GemStone FAQ

`http://www.ipass.net/vmalik/gemstone.html`

[3] MagLev

`http://maglev.github.io/`

[4] STM on CPython. August 2011. PyPy mailing list discussion.

`https://mail.python.org/pipermail/pypy-dev/2011-August/008153.html`

[5] PyPy Status Blog: STM with threads

`http://morepypy.blogspot.com.ar/2012/06/stm-with-threads.html`

[6] Concurrent Ruby: TVar

`http://ruby-concurrency.github.io/concurrent-ruby/Concurrent/TVar.html`

[7] Lukas Renggli and Oscar Nierstrasz. 2007. *Transactional Memory for Smalltalk.* Software Composition Group, University of Bern.

[8] Lukas Renggli and Oscar Nierstrasz. 2008. *Transactional memory in a dynamic language.* Software Composition Group, University of Berne, 3012 Bern, Switzerland.

[9] Pat Shaughnessy. 2014. *Ruby Under a Microscope.*

[10] Adele Goldberg and David Robson. 1983. *Smalltalk-80: The language and its implementation.* Xerox Palo Alto Research Center.

[11] Paolo Perrotta. 2010. *Metaprogramming Ruby.* The Pragmatic Programmers.

[12] Official documentation for the Ruby programming language.

`http://ruby-doc.org/`

[13] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. 2003. **Traits: Composable Units of Behaviour**. Software Composition Group, University of Bern, Switzerland.

[14] Feature #11473: Immutable String literal in Ruby 3

`https://bugs.ruby-lang.org/issues/11473`

[15] Hamster: Efficient, Immutable, Thread-Safe Collection classes for Ruby

`https://github.com/hamstergem/hamster`