DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

# Especificación y verificación modular de consumo de memoria

Febrero de 2011 — Tesis de Licenciatura


Jonathan Tapicer

Libreta Universitaria: 371/06

E-mail: jtapicer@dc.uba.ar


## Directores


Diego Garbervetsky
diegog@dc.uba.ar

Martín Rouaux
mrouaux@dc.uba.ar





**Facultad de Ciencias Exactas y Naturales**
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

**Resumen**

Abordamos en este trabajo la problemática de la verificación modular del consumo de memoria en programas orientados a objetos. El enfoque propuesto consiste en permitir al usuario anotar el programa con contratos asociados al consumo de memoria del mismo y luego verificar su correctitud, obteniendo así un *certificado* del consumo máximo de memoria del programa.

Para representar el hecho de que la memoria es recolectada automáticamente por un *garbage collector* en los lenguajes orientados a objetos del estilo de Java y C#, utilizaremos un modelo simplificado de administración de memoria basado en scopes. Definimos la sintaxis y semántica de un lenguaje de anotaciones para escribir contratos modulares de consumo de memoria en el código. Este lenguaje es suficientemente expresivo para brindar información de tiempo de vida de los objetos y contempla mecanismos de ocultamiento de la información para mantener la modularidad.

Se diseñan y detallan un conjunto de técnicas de instrumentación y verificación de los contratos, que apoyadas en un verificador estático permiten desarrollar una solución capaz de verificar la correctitud de los contratos dados en las anotaciones definidas.

La verificación de los contratos requiere muchas veces de una capacidad de análisis aritmético complejo. Dadas las limitaciones en el módulo aritmético de las mayoría de los verificadores estáticos existentes, proponemos un conjunto de técnicas adicionales que permiten integrar herramientas externas al proceso de verificación, haciendo posible la verificación de contratos con requerimientos aritméticos complejos.

Realizamos una implementación de prueba de concepto de las anotaciones y técnicas desarrolladas en el trabajo para la plataforma .NET utilizando las capacidades de verificación de la herramienta Code Contracts.

# Índice

# Ejemplos

# Índice de figuras

# 1. Introducción

## 1.1. Descripción de la problemática

En determinado conjunto de aplicaciones la memoria es un recurso crítico para el correcto funcionamiento del sistema; nos referimos a sistemas donde el uso de memoria debe ser predecible y determinístico tales como:

- Aplicaciones para sistemas embebidos, donde muchas veces por limitantes de hardware la memoria suele ser uno de los recursos más importantes a tener en cuenta.
- Sistemas real-time, donde no son afrontables los recursos consumidos por un garbage collector y su impredecibilidad temporal[1].
- Sistemas que se ejecutan en entornos donde el uso de los recursos, y en particuar el uso de memoria, se cobra; en estos casos no es deseable pagar por el uso una cantidad de memoria que no se necesite.

En estos casos es valioso contar con un *certificado* de que el consumo de memoria del programa nunca excederá un límite dado.

Un ejemplo de un conjunto de herramientas diseñadas para este fin surge a partir de un movimiento llamado Real-time Java que cuenta con una especificación formal definida en [GB00] (Real-time Specification for Java); esta especificación permite implementar Java para sistemas en tiempo real y embebidos con una administración de memoria con consumo predecible (se encuentra disponible una implementación hecha por Sun: [Sun09]).

Es en este marco donde surgen diferentes formas de administración de la memoria dinámica. La administración de memoria por scopes es una de ellas, de la que hablamos en la sección siguiente.

Un enfoque posible para obtener un *certificado* de consumo de memoria es el tomado por trabajos anteriores como [Rou09]. Este consiste en utilizar técnicas de análisis estático para inferir expresiones que determinan el consumo de memoria en función de los parámetros del programa para un esquema de administración de memoria basado en scopes.

Otro enfoque posible consiste en permitir al usuario definir, mediante contratos, su conocimiento sobre el consumo de memoria para luego verificar la correctitud de los mismos. Nuestra propuesta es permitir al programador anotar contratos de consumo de memoria embebidos en el código para luego verificar su correctitud.

[1] Es decir, no se puede determinar el momento en el que el *garbage collector* se ejecutará, y esto podría generar un retraso no afrontable por un sistema en tiempo real.

La inferencia del consumo exacto de memoria de un programa es un problema indecidible [Ghe02], sin embargo en el campo de la verificación estática existen avances significativos que abren muchas posibilidades para su verificación. De esta forma, si el usuario tiene conocimiento acerca de este consumo de memoria, verificar su correctitud es una tarea más fácil de abordar que inferirlo; de la misma manera que es fácil verificar la correctitud de la solución a un problema a pesar de ser muy difícil de obtener dicha solución, la verificación de contratos puede ser posible en muchos casos en que la inferencia no lo es. Por esto es que nuestro trabajo se centra en definir un lenguaje de anotaciones con estas características, creemos que así podremos lograr una herramienta capaz de verificar contratos en programas grandes y complejos donde el patrón de consumo de memoria suele tener una complejidad que a veces se torna inmanejable (o pierde precisión) a la hora de inferir. Creemos que la herramienta ideal debería hacer uso combinado de técnicas de verificación, como las que desarrollamos en este trabajo, con técnicas de inferencia como las mencionadas, por lo que en un futuro pensamos integrar a la herramienta desarrollada técnicas de inferencia.

## 1.2. Objetivos generales

El objetivo general de este trabajo consiste en diseñar e implementar un lenguaje de anotaciones para definir contratos de consumo de memoria para lenguajes orientados a objetos clásicos (del estilo de Java y C#), que permitan desarrollar una herramienta para verificar la correctitud de los mismos.

Definiremos un lenguaje de anotaciones para especificar los contratos tomando como base otros lenguajes para especificar contratos de comportamiento (tales como JML [LRL+00], Spec# [BLS05] y Code Contracts [FBL10]) adaptándolos al caso del consumo de memoria. El lenguaje de anotaciones debe contemplar las siguientes características:

- Deberá ser intuitivo y cómodo para su uso por parte del usuario, esto en parte estará asociado a la implementación hecha como explicaremos más adelante.
- Debe contemplar mecanismos para el ocultamiento de la información entre clases, es decir, cuando un método invoca a otro no debe ser necesario que conozca su implementación para poder razonar acerca de su consumo de memoria.
- Para poder considerar la recolección de memoria es necesario algún mecanismo para especificar el tiempo de vida de los objetos, por lo tanto las anotaciones permitirán al usuario proveer este tipo de información.
- Deberán mantener la modularidad (i.e., las anotaciones son aserciones sólo sobre el consumo del método anotado) y al mismo tiempo las anotaciones que harán de contratos para otro métodos deberán brindar a un método llamador toda la información necesaria para la verificación con sólo conocer el contrato del método llamado.

Aprovecharemos la capacidad de verificadores estáticos existentes, por lo tanto una

vez definida la sintaxis y semántica de las anotaciones a utilizar diseñaremos un conjunto de algoritmos para implementar una verificación de las mismas que describirán cómo transformar las anotaciones de memoria en anotaciones clásicas que entiende un verificador estático (aserciones booleanas que predican sobre expresiones) y cómo insertar el código necesario para mantener la trazabilidad de las alocaciones de memoria a través de la instrumentación de contadores y otros bloques de código.

Para demostrar la factibilidad de la solución propuesta haremos una implementación de la misma como prueba de concepto. La misma se hará sobre la plataforma .NET y será una extensión de la herramienta Code Contracts [FBL10].

Para lograr una solución capaz de verificar contratos con expresiones matemáticas complejas integraremos a la solución herramientas externas, tales como Barvinok [CFGV09], una herramienta implementada con ideas definidas en [CT04] capaz de realizar manipulaciones simbólicas con expresiones paramétricas para resolver operaciones de conteo, maximización y suma bajo un conjunto de restricciones.

## 1.3. Trabajos relacionados

### 1.3.1. Inferencia

Dentro del conjunto de trabajos que abordan la problemática del análisis estático de consumo de memoria dinámica (o recursos en general) algunos de ellos hacen un análisis para inferir resúmenes de consumo de memoria por métodos. Por ejemplo [Gar07] hace un análisis global para inferir el consumo de memoria total, este tipo de análisis tiene problemas de escalabilidad; [Rou09] utiliza un enfoque de especificación modular inferida para resolver estos problemas de escalabilidad. [GK10] propone una extensión de [Rou09] que aumenta las capacidades de inferencia del análisis mejorando la precisión de los resultados. Estos trabajos están limitados por la indecibilidad presente en el análisis de inferencia del consumo de memoria.

Existen otros dos trabajos relacionados ([Gul09] y [GZ10]) que a pesar de no abordar la problemática del consumo de memoria directamente se concentran en el desarrollo de técnicas para determinar automáticamente cotas paramétricas en función de la entrada sobre la cantidad de veces que ciertas líneas de código son ejecutadas. Estas técnicas tienen varios usos tales como el análisis de consumo de recursos (sin contemplar la liberación de los mismos) y la determinación de la complejidad temporal de un algoritmo.

### 1.3.2. Verificación

En la línea de la verificación existen pocos trabajos relacionados; [CNQR05] propone un sistema de tipos con una semántica que permite determinar cuándo se solicita y cuándo se libera la memoria, requiere de muchas anotaciones y soporta sólo expresiones lineales sin productos en las mismas; además se especifica sobre un lenguaje recursivo,

aunque el código Java puede ser traducido al mismo. [BPS05] propone un enfoque similar al abordado en este trabajo, con contratos embebidos en el código, pero las anotaciones no contemplan mecanismos para especificar el tiempo de vida de los objetos instanciados.

El enfoque propuesto en [CEI+07] consiste en combinar un análisis estático con chequeos dinámicos (*run-time*) embebidos en el código, describe un lenguaje para anotar el consumo de recursos y da una semántica operacional similar a la dada por [CNQR05] pero para un lenguaje imperativo minimal. Este enfoque no contempla la liberación de memoria aunque propone una extensión para permitir anotarla.

## 1.4. Aportes

Identificamos en nuestro trabajo los siguientes aportes novedosos:

- La definición de un lenguaje suficientemente expresivo que permite a los usuarios escribir aserciones sobre la cantidad de memoria consumida por un método, además del tipo y la forma del consumo. El lenguaje además permite mantener la modularidad de los contratos y el ocultamiento de la información entre clases.
- Un conjunto de algoritmos que, apoyados en las capacidades de un verificador estático, permiten verificar la correctitud de contratos de consumo de memoria[2].
- Una serie de algoritmos que, apoyados en un análisis estático de aliasing y escape del programa, permiten verificar la correctitud de las anotaciones definidas para especificar el tiempo de vida de los objetos.
- Una implementación basada en Code Contracts [FBL10] de las anotaciones y los algoritmos definidos, la misma está integrada en la IDE utilizada para Code Contracts (Microsoft Visual Studio) y se utiliza de la misma forma que la herramienta original, por lo que un usuario de Code Contracts puede adaptarse fácilmente a la herramienta.

## 1.5. Estructura del trabajo

En la Sección 2 (Conceptos preliminares) resumimos algunos conceptos necesarios para la comprensión global del trabajo.

En la siguiente Sección, 3 (Definiendo un lenguaje de anotaciones), definimos formalmente la sintaxis y semántica de las anotaciones propuestas para describir los contratos de consumo de memoria y damos algunos de ejemplos de su uso.

[2]Notar que estos algoritmos no son definidos para ningún verificador ni lenguaje en particular, por lo que podrían ser aplicados en un futuro para diferentes lenguajes (orientados a objetos) y verificadores.

Luego, en la Sección 4 (Verificación de consumo mediante instrumentación) definimos los algoritmos principales de instrumentación, que junto a un verificador estático permitirán verificar la correctitud de las anotaciones dadas.

En la Sección siguiente, 5 (Verificación de correctitud de las anotaciones de tiempo de vida), damos algoritmos adicionales para implementar la verificación las anotaciones de tiempo de vida, no directamente asociadas a los contratos de consumo de memoria, pero cuya correctitud es necesaria para la correctitud de los contratos.

En la Sección 6 (Incrementando las capacidades aritméticas de la verificación) describimos la forma de implementar un conjunto de técnicas que permiten incrementar la cantidad de casos verificables para verificadores estáticos que no tienen un módulo aritmético capaz de operar con polinomios.

Luego, en la Sección 7 (Overview de la implementación) damos una idea general de la estructura de la solución implementada, primero genéricamente y luego detallando los componentes desarrollados y utilizados en la implementación de prueba de concepto.

En la Sección 8 (Evaluación) evaluamos las capacidades de la herramienta desarrollada mostrando una serie de experimentos y ejemplos que demuestran las capacidades y limitaciones de la misma.

En la Sección 9 (Conclusiones) concluimos el trabajo haciendo un breve resumen de los logros del mismo.

En la Sección 10 (Trabajo futuro) resumimos un conjunto de posibles líneas de trabajo futuro junto a algunas ideas de cómo abordar cada una de ellas.

Finalmente, en la Sección 11 (Anexo) incluimos el código completo de un modelo de clases anotado mencionado durante el trabajo y detallamos algunas cuestiones técnicas necesarias para el correcto uso de la herramienta implementada.

# 2. Conceptos preliminares

## 2.1. Administración de memoria por scopes

Los lenguajes orientados a objetos con manejo de memoria automática utilizan en su mayoría una administración de memoria basada en *garbage collector*, donde la memoria que se dejó de utilizar es periódicamente liberada por un proceso encargado de hacerlo. Este tipo de administración de memoria es generalmente impredecible, dado que no es posible determinar cuándo se liberará cierta memoria alocada.

En este trabajo, para ser capaces de hacer un análisis estático del consumo de memoria, asumimos que los objetos se comportan según un modelo de administración de

memoria basado en scopes, donde la recolección de objetos de memoria ocurre siempre al final de la ejecución de un método. De esta forma, eliminamos la impredictibilidad que hace imposible el análisis.

En la administración de memoria por scopes la alocación y liberación de memoria se debe hacer en momentos específicos predeterminados, los objetos alocados se ubican en espacios de memoria asociados al scope de creación del objeto y a su tiempo de vida.

Los scopes en general se asocian a una invocación a un método, aunque podrían ser más generales o particulares. En este trabajo utilizaremos scopes asociados a la ejecución de un método.

Los objetos asociados a un scope son liberados por un método al final de su ejecución, y un método sólo puede liberar los objetos que él mismo creó o los creados por los métodos que invoca si el tiempo de vida del objeto excede el del método invocado, pero nunca puede liberar objetos pre-existentes, es decir, objetos creados en una invocación anterior en el stack de llamadas.

Para ejemplificar supongamos el siguiente código de dos métodos de una misma clase:

```
1  public void TestCreateNode(int value)
2  {
3    Node n1 = new Node(value);
4    Node n2 = this.CreateNodeLogging(value);
5  }
6
7  public void CreateNodeLogging(int value)
8  {
9    Logger logger = new Logger();
10   logger.log("node created");
11
12   return new Node(value);
13 }
```

Ejemplo 1: Asociación de objetos a scopes

Supongamos una invocación al método `TestCreateNode`, el objeto creado en la línea 3 puede ser asociado al scope del método `TestCreateNode` y liberado al final de la ejecución del mismo, lo mismo sucede con el creado en la línea 9 en el método `CreateNodeLogging`, sin embargo el creado en la línea 12 no puede ser alocado en el espacio de memoria asociado al método `CreateNodeLogging` dado que su tiempo de vida excede el scope de `CreateNodeLogging`, pero sí puede ser asociado al de `TestCreateNode` porque el objeto no es necesario al finalizar su ejecución y éste podrá liberarlo al finalizar.

Un ejemplo particular de la administración de memoria por scopes es la administración de memoria por regiones descripta en [TT97] donde se permite al programador definir regiones de memoria, asociar objetos a las mismas y luego eliminar regiones

liberando todos los objetos alocados en la misma.

En este momento, es lícito aclarar que a pesar de que la herramienta que desarrollaremos estará basada en la plataforma .NET y no existen implementaciones conocidas de dicha plataforma que utilicen un esquema de administración de memoria por scopes, utilizando un garbage collector debidamente configurado[3] el total de memoria consumida por un programa estará siempre acotado superiormente por el total de memoria consumida por el mismo programa utilizando un esquema de administración de memoria por scopes. Así es que la administración de memoria por scopes en este trabajo es utilizada como forma predecible de modelar el uso de memoria, pero aún cuando un programa se ejecutará con una administración de memoria con garbage collector, la verificación de los contratos de consumo de memoria asumiendo un esquema de administración por scopes es útil para conocer el máximo de memoria que un programa puede potencialmente utilizar.

Definiendo un conjunto de anotaciones apropiadamente expresivo seremos de capaces de determinar exactamente el espacio de memoria asociado a un método que contiene a una instancia específica, para esto necesitaremos definir una nomenclatura para el tipo de memoria que cada método tiene de acuerdo al scope al que deberán asociarse los objetos que crea, de esto nos ocupamos en la siguiente sección.

## 2.2. Tipos de memoria

Siguiendo con la nomenclatura definida en [Rou09] vamos a discriminar la memoria alocada por un método en dos tipos: temporal y residual. Primero definiremos informalmente estos conceptos y luego los formalizaremos.

Durante la ejecución de un método $m$ se crean un conjunto de objetos tanto en el método mismo invocado como en todos los métodos que éste invoca. Particionamos a este conjunto de objetos en dos categorías disjuntas: temporales y residuales. Dado que consideramos un esquema de administración de memoria por scopes, los objetos existentes antes de la ejecución de $m$ no son considerados, porque $m$ no puede liberarlos.

Los objetos temporales son los objetos que $m$ utiliza de forma auxiliar, los mismos son utilizados durante la ejecución de $m$ para un cálculo o proceso pero al final de su ejecución ya no son necesarios.

En cambio, los objetos residuales son necesarios luego de la ejecución de $m$, esto puede suceder por diferentes razones, por ejemplo, si el objeto es devuelto. Decimos que estos objetos exceden el tiempo de vida de la ejecución de $m$.

[3] Algunas virtual machines, como la de Java, permiten configurar la máxima memoria utilizable, llegado este límite se forzará una recolección de memoria. Otra alternativa es forzar manualmente la recolección al final de cada método (aunque los garbage collectors suelen ser más agresivos y recolectan los objetos de memoria lo antes posible), para asegurarse que siempre se cumplen las condiciones asumidas por el modelo de memoria que utilizamos.

A continuación definimos formalmente los tipos de memoria descriptos.

**Definición:** Decimos que un objeto ***o*** creado por el método $m$ o por algún método invocado por $m$ **escapa** de $m$ cuando el tiempo de vida de ***o*** podría exceder el de una invocación a $m$.

Identificamos tres formas en que un objeto ***o*** escapa de un método $m$:

1. ***o*** es devuelto por $m$ o puede ser alcanzable desde el objeto devuelto.
2. ***o*** puede ser alcanzable desde algún parámetro de $m$ al finalizar su ejecución.
3. ***o*** puede ser alcanzable desde el scope global al finalizar la ejecución de $m$.

Para determinar si un objeto escapa de un método se puede realizar un análisis estático de Points-To como el descripto en [BFGL07].

**Definición:** Decimos que un objeto ***o*** creado por el método $m$ o por algún método invocado por $m$ es parte de la memoria **temporal** de $m$ si ***o*** **no escapa** de $m$.

**Definición:** Decimos que un objeto ***o*** creado por el método $m$ o por algún método invocado por $m$ es parte de la memoria **residual** de $m$ si ***o*** **escapa** de $m$.

Volviendo al Ejemplo 1 presentado en la sección anterior, el objeto instanciado en la línea 9 del método `CreateNodeLogging` pertenece a la memoria temporal del método y el instanciado en la línea 12 pertenece a la memoria residual y también pertenece a la memoria temporal de `TestCreateNode`, dado que `TestCreateNode` invoca a `CreateNodeLogging`, obtiene dicho objeto y el mismo no escapa de `TestCreateNode`.

Relacionándolo con la administración de memoria por scopes, todo objeto cuyo scope es el método donde se creó y su tiempo de vida no excede dicho método pertenece a la memoria temporal del método, mientras que si es necesario en un scope previo (de un método que invocó al que creó el objeto) pertenecerá a la memoria residual del método que lo creó.

En los contratos discriminaremos la memoria temporal y residual por tipos de objetos (clases), haciendo referencia al tipo de objeto que esa memoria referencia. Por ejemplo, diremos que el método $m$ tiene una memoria temporal de tipo $T$ de cantidad $n$ (donde $n$ puede ser un número o cualquier expresión entera en función de los parámetros de $m$). Además, para el caso de la memoria residual definiremos una notación que nos permitirá comprender a través de qué expresión un objeto (o conjunto de objetos) escapa del método, para luego ser capaces de mantener la trazabilidad del mismo a la hora de verificar los contratos.

Una diferencia importante entre la memoria residual y temporal de un método está relacionada con cómo se trata la memoria de los métodos invocados. Los objetos residuales de un método invocado se acumulan en el método llamador (donde pueden ser

temporales o residuales del mismo), pero lo mismo no sucede con los objetos temporales, dado que éstos no son necesarios en el método llamador, aunque sí deben ser tenidos en cuenta en el momento de calcular la memoria temporal del mismo (recordar que en la definición se tienen en cuenta tanto los objetos creados por el método como los objetos creados por todos los métodos invocados).

Para calcular el requerimiento de memoria temporal de un método tendremos en cuenta la memoria temporal necesaria de acuerdo de los objetos que el método crea directamente y de acuerdo a los objetos que crean los métodos que invoca; sin embargo, en el caso de estos últimos objetos no todos necesitan un espacio de memoria simultáneamente. Veamos el siguiente siguiente ejemplo:

```
1 public void CreateNodeAndAppend(int value, List<Node> list)
2 {
3   Node n = new Node(value);
4   list.Register(n); //Register tiene temporal 2 de tipo Node
5   list.Append(n); //Append tiene temporal 3 de tipo Node
6 }
```

Ejemplo 2: Ejemplo de consumo de memoria temporal bajo invocaciones

El método tiene un consumo de memoria temporal de tipo `Node` por objetos creados directamente de 1 (por el objeto creado en la línea 3). Supongamos además que el método `Register` invocado en la línea 4 crea 2 objetos temporales de tipo `Node` y que el método `Append` invocado en la línea 5 crea 3 objetos temporales de tipo `Node`.

De esta manera, el método podría ejecutarse de forma segura disponiendo de un espacio para memoria temporal de 6 objetos de tipo `Node`. Sin embargo, los objetos temporales de los dos métodos invocados nunca existen simultáneamente en memoria, porque los mismos son eliminados de memoria al finalizar la ejecución de cada método (esto sucede siempre con los objetos temporales de un método). De esta forma, con disponer de un espacio de memoria de 4 objetos de tipo `Node` también podremos ejecutar de forma segura el método; el espacio utilizado por el método `Register` es liberado al finalizar su ejecución y luego reutilizado por el método `Append`, los 3 objetos temporales que crea sumado al creado por el método que estamos analizando son la máxima cantidad de memoria temporal necesaria en cualquier momento durante la ejecución del mismo.

Generalizando este razonamiento, decimos que la memoria necesaria por un método bajo la presencia de invocaciones a otros métodos es la memoria que utiliza el método directamente sumado a la máxima memoria temporal utilizada por los métodos que invoca. No sucede lo mismo con la memoria residual, dado que la memoria residual de un método invocado no es liberada al finalizar la ejecución del método porque sus objetos pueden ser utilizados en el método llamador.

Si en un método llamamos $Tmp_T$ a la cantidad de memoria temporal de tipo $T$ que el mismo requiere y a la residual del mismo tipo $Rsd_T$, y todos los tipos de objetos instanciados por el método y por los métodos que invoca son $T_1$, $T_2$, ..., $T_n$ entonces

definimos a $Req$, la cantidad de memoria necesaria para la ejecución del método, como:

$$Req = \sum_{i=1}^{n} size(T_i) * Tmp_{T_i} + size(T_i) * Rsd_{T_i}$$

Donde $size(T)$ es el tamaño máximo que un objeto de tipo $T$ puede ocupar en memoria.

Es decir, la memoria requerida por un método es el espacio de memoria que ocupan todos los objetos temporales y residuales que crea él mismo y los métodos que invoca.

Notar que en la especificación permitiremos especificar a $Rsd_T$ particionada de acuerdo a la forma en que escapa del método, por lo que tendremos un conjunto de valores $Rsd_{T_1}, \ldots, Rsd_{T_m}$ para cada tipo, en ese caso consideraremos $Rsd_T = \sum_{i=1}^{m} Rsd_{T_i}$.

El cálculo de $Req$ introduce una sobre-aproximación del consumo de memoria del programa, la misma surge en la suma de la memoria temporal, la cual no tiene en cuenta que el mismo espacio de memoria podría ser utilizado por diferentes objetos temporales que no son necesarios en memoria en el mismo momento durante la ejecución de un método. Esta sobre-aproximación aparece a raíz del modelo de memoria utilizado, donde asumimos que los objetos temporales son eliminados de memoria recién al final de la ejecución del método.

En la sección siguiente definimos un lenguaje de anotaciones para especificar estos requerimientos de memoria junto a la información necesaria para verificar su correctitud.

# 3. Definiendo un lenguaje de anotaciones

En esta sección introducimos todos los tipos de anotaciones que soportaremos, tanto para definir contratos de consumo de memoria como para brindar información sobre el tiempo de vida de los objetos necesaria para la verificación.

Daremos la sintaxis de las anotaciones y su semántica informal, mediante una definición intuitiva.

La sintaxis de las anotaciones está inspirada en la de Code Contracts [FBL10], donde los contratos son parte del código, a diferencia de otros lenguajes como JML donde las anotaciones son comentarios. En todas las anotaciones definidas es posible utilizar cualquier expresión válida en el cuerpo del método anotado, siempre y cuando los tipos sean correctos.

Al final damos algunos ejemplos de métodos anotados para mostrar su modo de uso.

## 3.1. Introducción

Los contratos que definiremos en las subsiguientes secciones se basan en los contratos clásicos utilizados en diferentes lenguajes y verificadores estáticos para especificar un método. Estos contratos son una formalización de los requisitos de un método (precondiciones) y de las garantías luego de su ejecución (postcondiciones). Veamos el siguiente ejemplo que utiliza la sintaxis de contratos de Code Contracts:

```
1 public double RaizCuadrada(double n)
2 {
3   Contract.Requires(n >= 0);
4   Contract.Ensures(Contract.Result() * Contract.Result() == n);
5   Contract.Ensures(Contract.Result() >= 0);
6
7   return Math.Sqrt(n);
8 }
```

Ejemplo 3: Ejemplo de método con contratos de Code Contracts

El contrato del método especifica que siempre y cuando se cumpla que $n \geq 0$ el resultado devuelto multiplicado por sí mismo será igual a $n$ y será positivo, es decir, especifica formalmente que devuelve la raíz cuadrada positiva del parámetro $n$ siempre que éste sea positivo.

La postcondición del método `RaizCuadrada` puede ser vista como un *certificado* de que el método cumple con la condición especificada. De la misma forma, los contratos de memoria que describimos en la siguiente sección funcionarán como certificados del consumo de memoria del método y los mismos pueden ser vistos de la misma manera

que las postcondiciones en los contratos clásicos como el del ejemplo.

Sin aún definir la semántica de las anotaciones veamos un ejemplo de uso de la anotación utilizada para especificar el consumo de memoria temporal de un método[4]:

```
1  public void ProcessOrders(int n)
2  {
3    Contract.Requires(n >= 0);
4    Contract.Memory.Tmp<OrderProcessor>(n);
5
6    for (int i = 1; i <= n; i++)
7    {
8      OrderProcessor op = new OrderProcessor(i);
9      op.Process();
10   }
11 }
```

Ejemplo 4: Ejemplo de contrato de memoria temporal

El contrato de la línea 4 puede ser interpretado como un `Ensures` que especifica que la memoria temporal de tipo `OrderProcessor` del método `ProcessOrders` será menor o igual a `n`. Notar que especificamos un `Requires` en la línea 3, sin el mismo el contrato no sería correcto, dado que un número negativo invalidaría el contrato de consumo de memoria. En este caso el contrato es correcto, dado que el ciclo crea `n` objetos de tipo `OrderProcessor` y los mismos pertenecen a la memoria temporal del método porque son auxiliares.

Supongamos este otro método:

```
1  public void ProcessTwoOrders(int n1, int n2)
2  {
3    Contract.Requires(n >= 0);
4    Contract.Memory.Tmp<OrderProcessor>(1);
5
6    OrderProcessor op1 = new OrderProcessor(n1);
7    op1.Process();
8
9    OrderProcessor op2 = new OrderProcessor(n2);
10   op2.Process();
11 }
```

Ejemplo 5: Ejemplo de contrato de memoria temporal

El contrato de este método es incorrecto, dado que el mismo especifica que se crea a lo sumo un objeto temporal de tipo `OrderProcessor`, sin embargo se crean dos. La

[4]En estos ejemplos omitimos otras anotaciones necesarias para la verificación para mantener la simplicidad.

herramienta que desarrollaremos describirá la forma de verificar la correctitud de estos nuevos contratos.

## 3.2. Anotaciones para contratos de memoria

Las anotaciones que describimos a continuación son las que permiten describir los contratos de consumo de memoria de un método, y deben escribirse al principio de cada método anotado.

- `Contract.Memory.Tmp<T>(int n [, bool cond]);`

  Esta anotación define que el método requiere de una memoria temporal de tipo `T` menor o igual a `n`, como se explico anteriormente esta cantidad debe incluir la de los método invocados. `cond` es una condición booleana opcional que se tiene cumplir para que aplique el contrato.

  Se puede dar varios contratos para el mismo método siempre y cuando las condiciones sean disjuntas.

  **Ejemplo:**

```
public void ProcesarLista(List<Nodo> lista, int tipo)
{
  Contract.Memory.Tmp<Nodo>(lista.Count, tipo == 1);
  Contract.Memory.Tmp<Nodo>(lista.Count + 1, tipo == 2);
  ...
}
```

Ejemplo 6: Ejemplo de `Contract.Memory.Tmp`

- `Contract.Memory.Rsd<T>(Contract.Memory.RsdType rsd_name, int n [, bool cond]);`

  Define que el método requiere una memoria residual de tipo `T`, que escapa del método por la expresión asociada al residual de nombre `rsd_name`, menor o igual a `n`. `cond` es una condición booleana opcional que se tiene cumplir para que aplique el contrato.

  Se puede dar varios contratos para el mismo método siempre y cuando las condiciones sean disjuntas.

  El conjunto de objetos asociados a un mismo nombre de residual deben compartir un mismo tiempo de vida. El que se indique el nombre del residual por el que la memoria escapa permite al llamador referenciar a este residual por el mismo nombre y además permite asociarle una expresión que utilice atributos privados de la clase del método anotado, ocultando así las estructuras internas de la clase hacia el llamador. En la Sección 3.4 (Tipos de residuales) describimos en detalle la forma de declarar nombres de residuales y asociarle una expresión.

  **Ejemplo:**

```
public void ProcesarLista(List<Nodo> lista, int tipo)
{
  Contract.Memory.Rsd<Nodo>(rsdLista, lista.Count, tipo == 1);
  Contract.Memory.Rsd<Nodo>(rsdLista, lista.Count + 1, tipo == 2);
  ...
}
```

Ejemplo 7: Ejemplo de `Contract.Memory.Rsd`

## 3.3. Anotaciones para destino de un *new*

Las siguientes anotaciones son utilizadas para determinar si un objeto forma parte de la memoria temporal o residual del método. Es necesario que el usuario las provea si escribió un contrato para el tipo del objeto, y se verificará su correctitud.

- `Contract.Memory.DestTmp([bool trust]);`

  Esta anotación va antes de un *new*[5], e indica que el destino del objeto es el temporal del método del tipo del siguiente objeto creado.

  El parámetro `trust` es opcional, si se pasa en `true` se asumirá que la anotación es correcta y no se intentará verificarla.

  **Ejemplo:**

```
public void ProcesarLista(List<Nodo> lista, int tipo)
{
  ...
  Contract.Memory.DestTmp();
  Nodo n = new Nodo();
  ...
}
```

Ejemplo 8: Ejemplo de `Contract.Memory.DestTmp`

- `Contract.Memory.DestRsd(Contract.Memory.RsdType rsd_name [, bool trust]);`

  Esta anotación va antes de un *new*[6], e indica que el destino del objeto es el residual de nombre `rsd_name` del tipo del siguiente objeto creado.

  El parámetro `trust` es opcional, si se pasa en `true` se asumirá que la anotación es correcta y no se intentará verificarla.

  **Ejemplo:**

[5] Notar que esta anotación es ambigua cuando ocurre más de un *new* a continuación de la anotación, por ejemplo como parámetros de un método. En ese caso la semántica no está definida, sin embargo, siempre es posible reescribir el código de tal forma que esta ambigüedad no exista.

[6] Notar que esta anotación es ambigua cuando ocurre más de un *new* a continuación de la anotación, por ejemplo como parámetros de un método. En ese caso la semántica no está definida, sin embargo, siempre es posible reescribir el código de tal forma que esta ambigüedad no exista.

```
public void ProcesarLista(List<Nodo> lista, int tipo)
{
  ...
  Contract.Memory.DestRsd(rsdLista);
  Nodo n = new Nodo();
  ...
}
```

Ejemplo 9: Ejemplo de `Contract.Memory.DestRsd`

## 3.4. Anotaciones para definir tipos de residuales

Como ya mencionamos anteriormente, cada vez que se define un contrato de memoria residual, se debe asociar el mismo a un tipo de residual; las siguientes anotaciones permiten definir tipos de residuales y asociarlos a expresiones.

- `public static Contract.Memory.RsdType rsd_name;`

  El tipo `Contract.Memory.RsdType` es el utilizado para definir nombres de residuales, para definir un nuevo nombre de residual simplemente se declara un atributo estático en la clase y el nombre del atributo será el nombre del residual (`rsd_name` es este caso).

  Un nombre de residual se debe utilizar para definir la cantidad de objetos de un tipo dado que escapan del método a través de una misma expresión y comparten un tiempo de vida similar.

  El atributo debe ser estático y público para que otros métodos externos puedan referenciar al residual por su nombre escribiendo `Clase.rsd_name`.

  **Ejemplo:**

```
public static Contract.Memory.RsdType rsdLista;
public void ProcesarLista(List<Nodo> lista, int tipo)
{
  ...
}
```

Ejemplo 10: Ejemplo de `Contract.Memory.RsdType`

- `Contract.Memory.BindRsd(Contract.Memory.RsdType rsd_name, object`[7] `local_expr);`

  Esta anotación debe ir al comienzo del método, junto a los contratos de consumo de memoria y define que el residual de nombre `rsd_name` referencia a la expresión `local_expr` (en función de los parámetros del método, `this` o atributos estáticos).

[7] En C# `object` es una superclase de todas las clases, por lo que este tipo permite escribir cualquier expresión válida en el cuerpo del método.

Esta anotación es la que permite ocultar las estructuras internas de la clase, dado que la expresión `local_expr` puede hacer referencia a atributos privados de la clase, pero un método externo referenciará al residual por su nombre.

**Ejemplo:**

```
public static Contract.Memory.RsdType rsdLista;
public void ProcesarLista(List<Nodo> lista, int tipo)
{
  Contract.Memory.BindRsd(rsdLista, lista);
  ...
}
```

Ejemplo 11: Ejemplo de `Contract.Memory.BindRsd`

- `Contract.Memory.Return`

  Este es un nombre de residual (atributo de tipo `Contract.Memory.RsdType`) predefinido usado para referenciar a la memoria residual que escapa por el objeto devuelto por el método (no hace falta asociarlos con `Contract.Memory.BindRsd`).

  **Ejemplo:**

```
public List<Nodo> ProcesarLista(List<Nodo> lista, int tipo)
{
  Contract.Memory.Rsd<Nodo>(Contract.Memory.Return, lista.Count);
  ...
}
```

Ejemplo 12: Ejemplo de `Contract.Memory.Return`

- `Contract.Memory.This`

  Este es un nombre de residual (atributo de tipo `Contract.Memory.RsdType`) predefinido usado para referenciar a la memoria residual que escapa por la expresión `this` (no hace falta asociarlos con `Contract.Memory.BindRsd`).

  **Ejemplo:**

```
public void ProcesarLista(List<Nodo> lista, int tipo)
{
  Contract.Memory.Rsd<Nodo>(Contract.Memory.This, lista.Count);
  ...
}
```

Ejemplo 13: Ejemplo de `Contract.Memory.This`

## 3.5. Anotaciones para transferencia de residuales en invocaciones a métodos

Cuando un método $m_1$ invoca a un método $m_2$ que tiene un contrato de memoria residual, es necesario que el método $m_1$ especifique para cada tipo de residual cuál es

el destino de la memoria (si los objetos residuales de $m_2$ van a la memoria temporal o residual de $m_1$, y a cuál memoria residual se transfiere en caso de ir a ésta). Las siguientes anotaciones permiten anotar estas transferencias. La anotación debe estar presente en caso de que el método invocado tenga un contrato de memoria residual y el método que se está verificando tenga algún contrato para el mismo tipo de memoria.

- `Contract.Memory.AddRsd(Contract.Memory.RsdType rsd_name_local, Contract.Memory.RsdType rsd_name_call [, bool trust]);`

  Esta anotación va antes de una invocación[8], e indica que el residual de nombre `rsd_name_call` del método llamado se va a transferir al residual de nombre `rsd_name_local` del llamador.

  El parámetro `trust` es opcional, si se pasa en `true` se asumirá que la anotación es correcta y no se intentará verificarla.

  **Ejemplo:**

```
public void ProcesarLista(List<Nodo> lista, int tipo)
{
  Contract.Memory.Rsd<Nodo>(Contract.Memory.This, 1);
  ...
  Contract.Memory.AddRsd(Contract.Memory.This, Contract.Memory.
     Return);
  this._head = this.ProcesarNodo(nodo);
  ...
}
```

Ejemplo 14: Ejemplo de `Contract.Memory.AddRsd`

- `Contract.Memory.AddTmp(Contract.Memory.RsdType rsd_name_call [, bool trust]);`

  Esta anotación va antes de una invocación[9], e indica que el residual de nombre `rsd_name_call` del método llamado se va a transferir al temporal del llamador.

  El parámetro `trust` es opcional, si se pasa en `true` se asumirá que la anotación es correcta y no se intentará verificarla.

  **Ejemplo:**

```
public void ProcesarLista(List<Nodo> lista, int tipo)
{
  Contract.Memory.Tmp<Nodo>(1);
  ...
  Contract.Memory.AddTmp(Contract.Memory.Return);
  Nodo nodo = this.ProcesarNodo(nodo);
  ...
```

[8]Notar que esta anotación es ambigua cuando ocurre más de una invocación a continuación de la anotación, por ejemplo por un anidamiento de invocaciones. En ese caso la semántica no está definida, sin embargo, siempre es posible reescribir el código de tal forma que esta ambigüedad no exista.

[9]Notar que esta anotación es ambigua cuando ocurre más de una invocación a continuación de la anotación, por ejemplo por un anidamiento de invocaciones. En ese caso la semántica no está definida, sin embargo, siempre es posible reescribir el código de tal forma que esta ambigüedad no exista.

```
}
```

Ejemplo 15: Ejemplo de `Contract.Memory.AddTmp`

## 3.6. Anotaciones para definir espacios de iteración

La siguiente anotación permite definir el espacio de iteración de un ciclo.

Para la implementación particular que hacemos en este trabajo, esta anotación no es necesaria para la verificación utilizando los algoritmos de la Sección 4 (Verificación de consumo mediante instrumentación) dado que para los mismos el verificador utilizado es capaz de inferir los espacios de iteración, pero sí es necesaria para los algoritmos de la Sección 6 (Incrementando las capacidades aritméticas de la verificación) donde utilizamos una herramienta externa al verificador que necesita de esta información.

- `Contract.Memory.IterationSpace(bool cond);`

  Se sitúa adentro de un ciclo (`for` o `while`) inmediatamente después del comienzo del ciclo, `cond` es una expresión booleana que indica el espacio de iteración del ciclo. La condición puede ser cualquier expresión válida en el cuerpo del método que debe estar en función de los parámetros del mismo y de variables locales que se modifican dentro del ciclo.

  **Ejemplo:**

```
public void ProcesarLista(List<Nodo> lista, int tipo)
{
  ...
  for (int i = 1; i <= lista.Count; i++)
  {
    Contract.Memory.IterationSpace(1 <= i && i <= lista.Count);
    ...
  }
  ...
}
```

Ejemplo 16: Ejemplo de `Contract.Memory.IterationSpace`

El espacio de iteración puede ser fácilmente calculado conociendo el invariante de ciclo. La conjunción lógica del invariante de ciclo y la guarda resulta en el espacio de iteración. En el ejemplo, el invariante de ciclo es

$$1 \leq i \leq lista.Count + 1$$

y la guarda es

$$i \leq lista.Count$$

La conjunción de ambos es

$$1 \leq i \leq lista.Count + 1 \wedge i \leq lista.Count = 1 \leq i \leq lista.Count$$

que es el espacio de iteración.

En un futuro pensamos extender esta anotación para permitir una mayor granularidad, dando la posibilidad de anotar un espacio de iteración específico para cada punto dentro del ciclo. Esto puede ser necesario en casos donde dentro del ciclo existe un control de flujo y algunos puntos del mismo tienen un espacio de iteración más restringido.

## 3.7. Ejemplos

Para ejemplificar el uso de las anotaciones damos a continuación algunos ejemplos de uso que combinan diferentes tipos de anotaciones.

El código de los ejemplos está tomado de un programa anotado con contratos disponible en la Sección 11.4 (Ejemplo completo anotado).

En el siguiente ejemplo mostramos la especificación de un contrato de memoria temporal:

```
1  public Address(string street, string city, string state)
2  {
3    Contract.Memory.Tmp<AddressValidator>(1);
4
5    Contract.Memory.DestTmp();
6    AddressValidator validator = new AddressValidator();
7
8    if (!validator.IsAddressValid(street, city, state))
9    {
10     throw new Exception("Address is invalid.");
11   }
12
13   this.Street = street;
14   this.City = city;
15   this.State = state;
16 }
```

Ejemplo 17: Especificación de temporal

El contrato del método en la línea 3 especifica que tiene un temporal de objetos de tipo `AddressValidator` menor o igual a 1, y en la línea 5 se utiliza `DestTmp` para indicar que el objeto instanciado en la siguiente línea tiene como destino la memoria temporal.

En el siguiente ejemplo veremos el uso del `BindRsd` y la transferencia de residuales:

```
1  public static Contract.Memory.RsdType rsd_MovePersonTo_person;
2
3  public static void MovePersonTo(Person person, string street, string
       city, string state)
```

```
4  {
5    Contract.Memory.BindRsd(rsd_MovePersonTo_person, person);
6    Contract.Memory.Rsd<Address>(rsd_MovePersonTo_person, 1);
7    Contract.Memory.Tmp<AddressValidator>(1);
8
9    Contract.Memory.AddRsd(rsd_MovePersonTo_person, Contract.Memory.This)
       ;
10   person.MoveTo(street, city, state);
11 }
```

Ejemplo 18: Bind y transferencia de residuales

En la primera línea se declara el tipo de residual y en la línea 5 se lo asocia con la expresión `person` (un parámetro). Luego, en la línea 6 se especifica que por el residual recién mencionado escapa a lo sumo un objeto de tipo `Address`. En la línea 9 vemos que el residual del método `MoveTo` (invocado en la siguiente línea) por `this` se transfiere al residual local `rsd_MovePersonTo_person`. Si miramos el contrato del método `MoveTo` de la clase `Person` (disponible en el anexo), veremos que tiene un residual por `this` de a lo sumo 1 objeto de tipo `Address`, lo que hace válido el contrato del residual `rsd_MovePersonTo_person`.

El contrato de memoria temporal de tipo `AddressValidator` que tiene el método en la línea 7 es producto de la invocación de la línea 10, el método `MoveTo` tiene una memoria temporal de tipo `AddressValidator`, por lo tanto el método que la invoca necesita reservar memoria de ese tipo, a pesar de no usarla directamente.

El siguiente ejemplo es un método más complejo que utiliza varias anotaciones que luego detallaremos:

```
1  /// <summary>
2  /// Creates a family with the combination of all first and last names
3  /// </summary>
4  public static Person[] CreateCombinedFamily(List<string> firstNames,
     List<string> lastNames, string street, string city, string state)
5  {
6      System.Diagnostics.Contracts.Contract.Requires(firstNames.Count >
         0);
7      System.Diagnostics.Contracts.Contract.Requires(firstNames.Count ==
         lastNames.Count);
8
9      Contract.Memory.Rsd<Person>(Contract.Memory.Return, firstNames.
         Count * lastNames.Count);
10     Contract.Memory.Rsd<Person[]>(Contract.Memory.Return, 1);
11     Contract.Memory.Rsd<Address>(Contract.Memory.Return, firstNames.
         Count * lastNames.Count);
12     Contract.Memory.Tmp<AddressValidator>(1);
13
14     Contract.Memory.DestRsd(Contract.Memory.Return);
15     Person[] family = new Person[firstNames.Count * lastNames.Count];
16
17     for (int i = 0; i < firstNames.Count; i++)
18     {
19       Contract.Memory.IterationSpace(0 <= i && i < firstNames.Count);
```

```
20          for (int j = 0; j < lastNames.Count; j++)
21          {
22            Contract.Memory.IterationSpace(0 <= j && j < lastNames.Count);
23            Contract.Memory.AddRsd(Contract.Memory.Return, Contract.Memory.
                  This);
24            Contract.Memory.DestRsd(Contract.Memory.Return);
25            Person p = new Person(firstNames[i], lastNames[j], street, city
                  , state);
26            family[i * firstNames.Count + j] = p;
27          }
28        }
29
30        return family;
31    }
```

Ejemplo 19: Especificación de residuales con transferencias y loops

En este ejemplo observamos varias cosas: por un lado vemos que se usa en las líneas 6 y 7 la anotación `Requires` para especificar precondiciones del método, es lícito (y a veces necesario) utilizarlas para la correcta verificación de los contratos de memoria. El verificador de contratos de memoria que implementaremos se apoya en un verificador estático, por lo que los contratos de ambos pueden cooperar en la verificación final.

El método tiene tres contratos residuales (líneas 9, 10 y 11) donde las expresiones del contrato ya no son sólo números enteros, si no expresiones dependientes de los parámetros. Luego, vemos el uso de anotaciones para especificar los espacios de iteración de ciclos en las líneas 19 y 22. Finalmente vemos un uso de la anotación `DestRsd` en la línea 24 que especifica que el objeto instanciado en la línea siguiente tiene como destino la memoria residual que escapa por `return`, lo mismo sucede para el objeto `family` en la línea 14.

Con la presentación de estos ejemplos, vimos casos de uso de casi todas las anotaciones disponibles, ejemplificando su modo de uso y el aspecto de las mismas en el código. En las siguientes secciones se describe la solución implementada y los algoritmos que la hacen posible.

# 4. Verificación de consumo mediante instrumentación

En esta sección describimos los algoritmos de instrumentación del código que insertan contadores para registrar el consumo de memoria y la transformación de los contratos de memoria en contratos que entenderá el verificador estático, ambas tareas juntas permitirán verificar la correctitud de los contratos de memoria provistos por el usuario.

## 4.1. Ejemplo motivacional

Antes de comenzar a describir los algoritmos veamos un ejemplo motivacional para ilustrar el objetivo de la instrumentación.

Supongamos el siguiente método anotado:

```
1  public void ProcessOrders(int n)
2  {
3    Contract.Requires(n > 0);
4    Contract.Memory.Tmp<OrderProcessor>(n + 1);
5    Contract.Memory.Tmp<OrderProcessor[]>(1);
6
7    Contract.Memory.DestTmp();
8    OrderProcessor processor = new OrderProcessor(0);
9    processor.Process();
10
11   Contract.Memory.DestTmp();
12   OrderProcessor[] processors = new OrderProcessor[n];
13
14   for (int i = 1; i <= n; i++)
15   {
16     Contract.Memory.DestTmp();
17     processors[i - 1] = new OrderProcessor(i);
18     processors[i - 1].Process();
19   }
20 }
```

Ejemplo 20: Ejemplo motivacional para instrumentación

Asumiendo que el método `Process` de la clase `OrderProcessor` no tiene consumo de memoria temporal de tipo `OrderProcessor`, es fácil concluir que el contrato de memoria provisto es correcto, dado que el ciclo de la línea 14 consume `n` objetos de tipo `OrderProcessor` y en la línea 8 se consume uno más, dando un total de `n + 1`.

El objetivo de los algoritmos que describiremos a continuación es transformar este método en uno equivalente funcionalmente pero con contadores y contratos que permitan

a un verificador estático verificar la correctitud del contrato dado por el usuario. Para este ejemplo en particular buscamos obtener el siguiente código:

```
1  public static int Order_ProcessOrders_tmp_OrderProcessor;
2  public static int Order_ProcessOrders_tmp_OrderProcessor_arr
       ;
3
4  public void ProcessOrders(int n)
5  {
6    Contract.Requires(n > 0);
7    Contract.Memory.Tmp<OrderProcessor>(n + 1);
8    Contract.Memory.Tmp<OrderProcessor[]>(1);
9    Contract.Ensures(Order_ProcessOrders_tmp_OrderProcessor <=
         n + 1);
10   Contract.Ensures(
       Order_ProcessOrders_tmp_OrderProcessor_arr <= 1);
11
12   Order_ProcessOrders_tmp_OrderProcessor = 0;
13
14   Contract.Memory.DestTmp();
15   Order_ProcessOrders_tmp_OrderProcessor++;
16   OrderProcessor processor = new OrderProcessor(0);
17   processor.Process();
18
19   Contract.Memory.DestTmp();
20   Order_ProcessOrders_tmp_OrderProcessor_arr++;
21   OrderProcessor[] processors = new OrderProcessor[n];
22
23   for (int i = 1; i <= n; i++)
24   {
25     Contract.Memory.DestTmp();
26     Order_ProcessOrders_tmp_OrderProcessor++;
27     processors[i - 1] = new OrderProcessor(i);
28     processors[i - 1].Process();
29   }
30 }
```

Ejemplo 21: Ejemplo motivacional para instrumentación (instrumentado)

Notar los siguientes cambios:

- Se insertaron dos campos públicos y estáticos en la clase contenedora del método, `Order_ProcessOrders_tmp_OrderProcessor` y `Order_ProcessOrders_tmp_OrderProcessor_arr`, estos campos serán contadores utilizados para registrar la cantidad de objetos creados de cada tipo en el método `ProcessOrders`. Además son públicos y estáticos porque deben ser visibles por

otros métodos de otras clases para utilizar su contrato (más adelante detallamos la forma en que se utilizan).

- Se insertó un incremento de los contadores en cada *new* del tipo correspondiente dentro del método.
- Se insertaron contratos que especifican que los contadores alcanzan el valor máximo indicado en los contratos dados por el usuario.

Con estas adiciones, un verificador debería ser capaz de verificar la correctitud de los nuevos contratos insertados, probando así la correctitud de los contratos provistos por el usuario.

Vale aclarar que a pesar de existir casos donde la memoria utilizada por un objeto puede ser reutilizada por otro (por ejemplo: en el último ejemplo si cada instancia de `OrderProcessor` creada dentro del loop se asignase a una variable temporal del loop en lugar de almacenarse en un array, se podría utilizar el mismo espacio de memoria para todas las instancias, porque dicho objeto es temporal dentro del loop), nosotros no tendremos en cuenta esos casos particulares, y consideraremos siempre el peor caso, donde cada instancia tiene su espacio de memoria propio. Esto se debe a que todas las anotaciones tienen un alcance limitado al scope del método y no se permite una granularidad limitada al scope de un ciclo; por esta razón una posible forma de lograr el efecto de la reutilización del espacio de memoria es abstraer el cuerpo del ciclo en otro método. En un futuro pensamos automatizar esta abstracción, pero por el momento el usuario debe realizarla manualmente en caso de requerirla para la definición de contratos.

Otra aclaración pertinente es que el usuario es responsable de anotar un método con todos los contratos que se deben verificar para que la verificación sea correcta, la no existencia de un contrato de cierto tipo implica que el consumo de memoria de ese tipo no se verificará (incluso si hubiesen otra anotaciones adicionales para ese tipo de memoria); de la misma forma es responsable de definir todas las otras anotaciones necesarias para la verificación, tales como `DestTmp` o `DestRsd` para cada objeto creado, la ausencia de esta anotación hará que el objeto correspondiente se ignore en la verificación[10]. En un futuro pensamos agregar a la herramienta la capacidad de alertar sobre anotaciones faltantes para solventar este problema. Una forma de hacer esto es crear un contador que colecte la cantidad de memoria no anotada y luego agregar una postcondición al método que defina que este contador debe ser igual a cero, así los contratos del método no se verificarán cuando no se anote el destino de cierta memoria; esto también obligará al usuario a escribir un contrato de consumo de memoria para todos los tipos de objetos creados en el método, validación que no se hace actualmente en el prototipo implementado.

El objetivo de los siguientes algoritmos es detallar la forma en que se transforman todas las anotaciones dadas por el usuario en una instrumentación de contadores e inserción de contratos en forma similar a la hecha en el ejemplo dado.

[10] Esta es una limitación en la implementación hecha en la prueba de concepto.

## 4.2. Memoria temporal

Supongamos que estamos analizando el método `M` de la clase `C`.

### 4.2.1. Contratos

Para cada aparición de `Contract.Memory.Tmp<T>(int n [, bool cond]);`:

- Agregar en la clase `C`:

  `public static int C_M_tmp_T;`

  Este es el campo que se utilizará en el método `M` para contabilizar las instancias de tipo `T` que vayan a la memoria temporal.

- Al comienzo del método `M` insertar la siguiente asignación:

  `C_M_tmp_T = 0;`

  Incializando de esta forma el contador en 0.

- Si no se proveyó el parámetro `cond` insertar en el método el contrato:

  `Contract.Ensures(C_M_tmp_T <= n);`

- Si se proveyó el parámetro `cond` (en ese caso puede haber varios contratos, el usuario es responsable de que las condiciones sean disjuntas[11] y la instrumentación descripta se debe hacer para cada contrato) insertar el contrato:

  `Contract.Ensures(!cond || C_M_tmp_T <= n);`

  Esta condición tiene la misma semántica lógica que: $cond \rightarrow C\_M\_tmp\_T \leq n$, y determina que la condición de la derecha debe ser verdadera siempre que *cond* lo sea.

### 4.2.2. Destino de un *new*

Para cada aparición de `Contract.Memory.DestTmp();`:

Obtener el tipo `T` del *new* del siguiente statement (si no hay un *new* o no hay contrato para ese tipo de temporal se ignora la anotación), e introducir luego de la anotación el statement:

`C_M_tmp_T++;`

[11] Una forma fácil de verificar esto es insertando una anotación `Assert` que asevere que nunca occurren simultáneamente todas las condiciones.

### 4.2.3. Invocaciones fuera de loops

Recordemos de la Sección 2.2 (Tipos de memoria) que la memoria necesaria por un método bajo la presencia de invocaciones a otros métodos es la memoria que utiliza el método directamente sumado a la máxima memoria temporal utilizada por los métodos invocados. A continuación describimos la forma de realizar la instrumentación para las invocaciones, de forma de calcular el máximo de las memoria temporal requerida por los métodos invocados.

Para todas las invocaciones fuera de un loop, `call_1`, ..., `call_n`, para cada temporal en el método de tipo `T`, introducir el statement para cada `i` entre `1` y `n`:

`int call_i_tmp_T = limit_call_i_T;` después de cada invocación.

Donde `limit_call_i_T` es:

Si `call_i` es sobre la clase `C'` y método `M'`, y el método `M'` tiene un contrato para el temporal de tipo `T` entonces:

`limit_call_i_T` = `C'.C'_M'_tmp_T`

Si no tiene contrato para el temporal `T` entonces:

`limit_call_i_T` = `0`

De esta forma tenemos declarada en el método una variable entera para cada invocación con la cantidad de memoria temporal consumida por el mismo según el campo generado para verificar el consumo (el verificador luego será capaz de utilizar el contrato del método para determinar el valor de la variable local).

Luego, introducir al final del método el statement (sólo si el método tiene un contrato para el temporal de tipo `T`):

```
C_M_tmp_T += Math.Max(call_1_tmp_T, ..., call_n_tmp_T);        (1)
```

Así estaremos sumando al contador de la memoria temporal de tipo `T` el máximo de memoria temporal requerida por las invocaciones fuera de loops.

### 4.2.4. Invocaciones dentro de loops

Tratamos a las invocaciones dentro de loops aparte porque al momento de calcular el máximo de la memoria requerida por el método analizado se debe tener en cuenta que las invocaciones ocurren múltiples veces con parámetros posiblemente variantes. Para mostrar la forma en que los implementamos, podemos pensar que los loops pueden ser *desenrrollados* convirtiéndose en varios statements, por ejemplo, el siguiente loop:

```
for (int i = 0; i < n; i++)
{
  M(i);
}
```

Ejemplo 22: Ejemplo de loop

Puede ser visto como el siguiente código equivalente:

```
M(0);
M(1);
.
.
.
M(n - 2);
M(n - 1);
```

Ejemplo 23: Ejemplo de loop *desenrrollado*

De esta manera podemos aplicar la misma lógica explicada anteriormente sobre cada invocación dentro del loop, sólo debemos obtener el máximo de todas las invocaciones considerando el loop *desenrrollado*. A continuación explicamos cómo realizar la instrumentación para lograr esto.

Consideremos la siguiente invocación dentro de un loop:

```
for/while (...)
{
  .
  .
  .
  C'.call_loop(...);
  .
  .
  .
}
```

Ejemplo 24: Loop no instrumentado

Para cada temporal de tipo `T` que tenga un contrato en `C.M` y en `C'.call_loop`[12]:

- Agregar afuera del loop más externo (la invocación puede estar en un loop anidado, pero nos interesa cubrir todo el espacio de iteración) una variable local inicializada en 0:

[12] En caso de que el contrato esté presente sólo en `C'.call_loop` se debe alertar al usuario que falta un contrato para memoria de tipo `T` en `C.M`.

```
int max_call_loop_T = 0;
```

- Luego de la invocación introducir el statement:

```
max_call_loop_T = Math.Max(max_call_loop_T, C'.C'_call_loop_tmp_T);
```

Obteniendo así un loop instrumentado de la siguiente manera:

```
1  int max_call_loop_T = 0;
2  for/while (...)
3  {
4     .
5     .
6     .
7     C'.call_loop(...);
8     max_call_loop_T = Math.Max(max_call_loop_T, C'.C'
          _call_loop_tmp_T);
9     .
10    .
11    .
12 }
```

Ejemplo 25: Loop instrumentado

Finalmente, se debe agregar `max_call_loop_T` al máximo definido en (1), incluyendo así el máximo de las invocaciones dentro de loops en el máximo de las invocaciones fuera de loops, sumando al contador de la memoria temporal la cantidad de memoria efectivamente necesaria para la ejecución del método.

Nota: la posibilidad de determinar este máximo para diferentes tipos de loops dependerá de las capacidades del verificador estático utilizado. En la Sección 6 (Incrementando las capacidades aritméticas de la verificación) aplicamos ciertas técnicas que ayudan al verificador a verificar la correctitud de contratos más complejos bajo la presencia diferentes tipos de loops.

## 4.3. Memoria residual

Supongamos que estamos analizando el método `M` de la clase `C`.

### 4.3.1. Contratos

Para cada aparición de `Contract.Memory.Rsd<T>(Contract.Memory.RsdType rsd_name, int n [, bool cond]);`

- Agregar en la clase `C`:

  `public static int C_M_rsd_name_T;`

  Este es el campo que se utilizará en el método `M` para contabilizar las instancias de tipo `T` que vayan a la memoria residual.

- Al comienzo del método `M` insertar la siguiente asignación:

  `C_M_rsd_name_T = 0;`

  Incializando de esta forma el contador en 0.

- Si no se proveyó el parámetro `cond` insertar en el método el contrato:

  `Contract.Ensures(C_M_rsd_name_T <= n);`

- Si se proveyó el parámetro `cond` (en ese caso puede haber varios contratos, el usuario es responsable de que las condiciones sean disjuntas[13] y la instrumentación descripta se debe hacer para cada contrato) insertar el contrato:

  `Contract.Ensures(!cond || C_M_rsd_name_T <= n);`

  Esta condición tiene la misma semántica lógica que: $cond \rightarrow C\_M\_rsd\_name\_T \leq n$, y determina que la condición de la derecha debe ser verdadera siempre que *cond* lo sea.

### 4.3.2. Destino de un *new*

Para cada aparición de `Contract.Memory.DestRsd(rsd_name);`:

Obtener el tipo `T` del *new* del siguiente statement (si no hay un *new* o no hay contrato para ese tipo de residual se ignora la anotación), e introducir luego de la anotación el statement:

```
C_M_rsd_name_T++;
```

### 4.3.3. Transferencia a memoria temporal

Para cada anotación

```
Contract.Memory.AddTmp(Contract.Memory.RsdType rsd_name_call
[, bool trust]);
```

y para cada residual con nombre `rsd_name_call` y tipo `T` del método invocado (notar que `T` no es único porque el mismo nombre podría reutilizarse para diferentes tipos) realizar la siguiente instrumentación (si no hay una invocación, el residual no pertenece al método llamado o no hay un temporal local de ese tipo se ignora la anotación):

[13] Una forma fácil de verificar esto es insertando una anotación `Assert` que asevere que nunca occurren simultáneamente todas las condiciones.

Si la invocación es `C'.M'(...)` entonces introducir luego de la anotación el statement:

```
C_M_tmp_T += C'.C'_M'_rsd_name_call_T;
```

Así, la cantidad de memoria residual del método invocado es sumada a la memoria temporal del método local.

No hacemos distinción entre la localización de esta anotación dentro de un loop o no, dado que, al igual que en el caso de la suma de temporales, asumimos el peor caso, es decir, que la memoria temporal destino de la transferencia no se reutiliza.

### 4.3.4. Transferencia a memoria residual

Para cada anotación

```
Contract.Memory.AddRsd(Contract.Memory.RsdType rsd_name_local,
Contract.Memory.RsdType rsd_name_call [, bool trust]);
```

y para cada residual con nombre `rsd_name_call` y tipo `T` del método invocado realizar la siguiente instrumentación (si no hay una invocación, el residual no pertenece al método llamado o no hay un contrato local para el residual de nombre `rsd_name_local` y tipo `T` se ignora la anotación):

Si la invocación es `C'.M'(...)` entonces introducir luego de la anotación el statement:

```
C_M_rsd_name_local_T += C'.C'_M'_rsd_name_call_T;
```

De esta forma, la cantidad de memoria residual del método invocado es sumada a la memoria residual del método local.

La misma aclaración hecha anteriormente sobre loops es válida para este caso, no consideramos el reuso de espacio de memoria residual.

# 5. Verificación de correctitud de las anotaciones de tiempo de vida

En esta sección describimos la forma en que se verifica la correctitud de las anotaciones asociadas al tiempo de vida de los objetos, que a pesar de no formar parte de los contratos[14], su correctitud es necesaria para la correctitud de los contratos provistos por el usuario.

Para ejemplificar la necesidad de verificar estas anotaciones, supongamos que un *new* es anotado con `DestTmp` y que el contrato de memoria temporal es escrito y verificado acorde a esta anotación, pero en realidad el objeto no va a la memoria temporal, si no a la memoria residual. En ese caso se está asumiendo la correctitud de algo que no es cierto, lo que podría generar inconsistencias con otros contratos y se podría terminar aceptando la correctitud de aserciones incorrectas.

Por esta razón es que en esta sección explicamos cómo se verifica la correctitud de las anotaciones `DestTmp`, `DestRsd`, `AddTmp` y `AddRsd`.

## 5.1. Análisis de points-to y escape

Para determinar la correctitud de las anotaciones que mencionamos es necesario tener un conocimiento sobre el aliasing que ocurre durante la ejecución de un método, esta información la puede brindar un Points-to Graph. Con el mismo podremos determinar si ciertos objetos escapan o no de un método y a través de qué expresiones lo hacen.

Un Points-to Graph, en rasgos generales, es un grafo dirigido que representa la información de aliasing del método analizado, donde los nodos representan objetos y los arcos representan referencias entre estos objetos. Existen dos tipos de nodos: inside, que representan conjuntos de objetos creados por el método analizado en un mismo punto, y nodos outside, que representan conjuntos de objetos creados por otros métodos en un mismo punto. Para el análisis que haremos sólo nos importarán los nodos inside, porque deberemos verificar aserciones sobre el tiempo de vida de los objetos creados por el método que estamos analizando.

Para estas verificaciones utilizaremos una implementación del análisis de points-to y escape desarrollado en [BFGL07]. A continuación definiremos un conjunto de funciones, las cuales pueden ser resueltas analizando la información devuelta por el análisis. En la Sección 11.5 (Información obtenida del análisis de points-to y escape) del anexo describimos la información que obtenemos del análisis, haciendo consultas sobre esta información podemos resolver las siguientes funciones:

Un nodo está asociado a un *new* específico dentro del cuerpo del método (en la

[14] Más de allá de que todas las anotaciones se ubiquen bajo la clase `Contract.Memory`, las únicas que son estrictamente contratos son `Contract.Memory.Tmp` y `Contract.Memory.Rsd`.

implementación particular que utilizamos se identifica con el número de línea y columna donde se hace el *new*).

- $escapes(m,\ node)$: determina si el nodo *node* escapa del método $m$.
- $escapes\_through(m,\ node,\ expr)$: determina si el nodo *node* escapa del método $m$ a través de la expresión *expr*. Esto sucede si *node* es alcanzable por una expresión tal que *expr* es una subexpresión a izquierda[15] de la misma.

  Por ejemplo: si el nodo $n$ en el método $m$ escapa por la expresión $this.f.g$ entonces $escapes\_through(m,\ n,\ this.f)$ es verdadero.
- $escapes\_expr(m,\ expr)$: determina si algún nodo alcanzable por la expresión *expr* escapa del método $m$.
- $escapes\_expr\_through\_expr(m,\ expr1,\ expr2)$: determina si algún nodo alcanzable por la expresión $expr1$ escapa del método $m$ a través de $expr2$ (es decir, si alguno de esos nodos es alcanzable por una expresión tal que $expr2$ es subexpresión a izquierda de la misma).

Para ejemplificar el uso de las funciones definidas supongamos el siguiente Points-to Graph simplificado, el mismo contiene nodos que representan objetos y punteros entre los mismos que representan un vínculo a través del campo que indica la flecha.

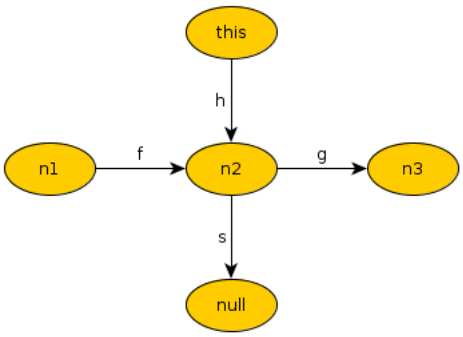


Figura 1: Points-to Graph de ejemplo

Si el Points-to Graph pertenece al método $m$, el único parámetro del mismo es *this* y no devuelve nada, entonces:

- $escapes(m,\ n1)$ es falso porque no es alcanzable por *this* ni por el scope global y no hay otros parámetros ni se devuelve nada.
- $escapes(m,\ n3)$ es verdadero porque $n3$ es alcanzable por *this*, que es un parámetro (implícito).

[15] Decimos que una expresión $x$ es subexpresión a izquierda de $e_1.e_2 \dots e_{n-1}.e_n$ si $x = e_1$ o es de la forma $e_1 \dots e_j$ con $2 \leq j \leq n$.

- *escapes_through*($m$, $n1$, *this.h*) es falso.
- *escapes_through*($m$, $n3$, *this.h*) es verdadero, porque $n3$ es alcanzable por *this.h.g* y *this.h* es subexpresión a izquierda de la misma.
- *escapes_expr*($m$, *this.h.s*) es falso porque el nodo alcanzado por dicha expresión es nulo.
- *escapes_expr*($m$, *this.h.g*) es verdadero.
- *escapes_expr_through_expr*($m$, *this.h*, *this.h.s*) es falso porque ningún objeto escapa a través de *this.h.s*.
- *escapes_expr_through_expr*($m$, *this.h*, *this.h.g*) es verdadero porque *this.h* alcanza al nodo $n3$ que escapa por la expresión *this.h.g*.

En algunos casos el análisis de escape realiza sobre-aproximaciones y determina que ciertos nodos escapan del método cuando no lo hacen; para subsanar este incoveniente en la verificación, en todas las anotaciones de tiempo de vida, damos la posibilidad de omitir la verificación utilizando el parámetro `trust`.

## 5.2. Verificación de `DestTmp`

Para cada anotación `Contract.Memory.DestTmp([bool trust])` en el método $m$ si `trust` no es `true`:

Buscar el nodo *node* correspondiente al próximo *new* (es decir, el primer *new* que ocurra después de la anotación) y verificar que *escapes*($m$, *node*) sea falso, es decir, que *node* no escape de $m$. En caso de que escape generar una alerta que reporte que la anotación podría ser incorrecta.

## 5.3. Verificación de `DestRsd`

Para cada anotación `Contract.Memory.DestRsd(Contract.Memory.RsdType rsd_name [, bool trust])` en el método $m$ si `trust` no es `true`:

Buscar el nodo *node* correspondiente al próximo *new* y verificar que *escapes*($m$, *node*) sea verdadero, es decir, que *node* escape de $m$. En caso de que no escape generar una alerta que reporte que la anotación es incorrecta.

Además en caso de que sí escape el nodo, tomar la expresión *expr* asociada según el `BindRsd` dado para el residual `rsd_name` y verificar que *escapes_through*($m$, *node*, *expr*) sea verdadero, en caso de que no lo sea generar una alerta que reporte que la anotación es incorrecta.

## 5.4. Verificación de `AddTmp`

Para cada anotación `Contract.Memory.AddTmp(Contract.Memory.RsdType rsd_name_call [, bool trust])`; en el método $m$ si `trust` no es `true`:

Buscar según la expresión dada en `BindRsd` del método llamado para el residual de nombre `rsd_name_call` el objeto local al que escapa el residual del método llamado.

Si la expresión es `this` y la llamada es `o.M()` entonces la expresión a evaluar es `o`.

Si la expresión es `return` y la llamada es `r = o.M()` entonces la expresión a evaluar es `r`.

Para el caso de los parámetros es necesario hacer un reemplazo del nombre utilizado en la expresión dada en `BindRsd` por el nombre del objeto local pasado de parámetro. Por ejemplo, si la expresión es `p.f` donde `p` es el primer parámetro del método y la llamada es `o.M(s)` entonces la expresión a evaluar es `s.f`, porque el nombre de `p` en el método llamador es `s`. Esta misma lógica aplica si la expresión contiene a `this`, pero el reemplazo a hacer en ese caso será de `this` por el objeto sobre el que se invoca el método.

Una vez obtenida esta expresión (llamémosla *expr*) verificar que $escapes\_expr(m,\ expr)$ sea falso, es decir, que el objeto correspondiente a *expr* no escape del método. En caso de que no lo sea generar una alerta que reporte que la anotación podría ser incorrecta.

Veamos un ejemplo:

```
1 public void ProcesarLista(List<Nodo> lista, int tipo)
2 {
3   Contract.Memory.Tmp<Nodo>(1);
4   ...
5   Contract.Memory.AddTmp(Contract.Memory.Return);
6   Nodo nodo = this.ProcesarNodo(nodo);
7   ...
8 }
```

Ejemplo 26: Ejemplo de verificación de `AddTmp`

Para verificar la correctitud de la anotación de la línea 5 debemos buscar la expresión local asociada a la expresión `return` del método `ProcesarNodo` (el nombre de residual `Contract.Memory.Return` se asocia siempre a la expresión `return` que representa el objeto devuelto por el método). En este caso la expresión local es `nodo`, entonces debemos verificar que $escapes\_expr(ProcesarLista,\ nodo)$ sea falso, es decir, que el objeto al que apunta `nodo` no escape del método.

Veamos otro ejemplo donde sea necesario hacer un reemplazo de nombres de acuerdo a la expresión dada en una anotación `BindRsd`:

```
1 Contract.Memory.RsdType rsd_ApilarElemento_pila;
2
```

```
3  public void ApilarElemento(Pila pila, int valor)
4  {
5    Contract.Memory.BindRsd(rsd_ApilarElemento_pila, pila.Tope);
6    Contract.Memory.Rsd<NodoPila>(rsd_ApilarElemento_pila, 1);
7    ...
8    pila.Tope = new ElementoPila(valor);
9    ...
10 }
11
12 public void ProbarPila()
13 {
14   Contract.Memory.Tmp<NodoPila>(1);
15   ...
16   Pila pilaPrueba = new Pila();
17   Contract.Memory.AddTmp(rsd_ApilarElemento_pila);
18   this.ApilarElemento(pilaPrueba);
19   ...
20 }
```

Ejemplo 27: Ejemplo de verificación de `AddTmp`

En el método `ProbarPila` se indica que la memoria residual del método `ApilarElemento` es transferida a la memoria temporal local, dicha memoria residual escapa del método `ApilarElemento` a través de `pila.Tope`, debemos reemplazar en esta expresión `pila` por `pilaPrueba` dado que ese es el parámetro pasado al método, entonces, para verificar la anotación `AddTmp` del método `ProbarPila` debemos verificar que $escapes\_expr(ProbarPila, pilaPrueba.Tope)$ sea falso, es decir, que el objeto al que apunta `pilaPrueba.Tope` no escape del método.

## 5.5. Verificación de `AddRsd`

Para cada anotación `Contract.Memory.AddRsd(Contract.Memory.RsdType rsd_name_local, Contract.Memory.RsdType rsd_name_call [, bool trust]);` en el método $m$ si `trust` no es `true`:

Buscar según la expresión dada en `BindRsd` del método llamado para el residual de nombre `rsd_name_call` el objeto local al que escapa el residual del método llamado usando la misma lógica descripta en la sección anterior.

Una vez obtenida esta expresión (llamémosla $expr$) verificar que $escapes\_expr(m, expr)$ sea verdadero, es decir, que el objeto correspondiente a $expr$ escape del método. En caso de que no lo sea generar una alerta que reporte que la anotación es incorrecta.

Además, en caso de que sí escape, tomar la expresión asociada según el `BindRsd` dado para el residual `rsd_name_local` en el método $m$ (llamémosla $expr\_local$) y verificar que $escapes\_expr\_through\_expr(m, expr, expr\_local)$ sea verdadero, en caso de que no lo sea generar una alerta que reporte que la anotación es incorrecta.

Veamos un ejemplo:

```
1 public void ProcesarLista(List<Nodo> lista, int tipo)
2 {
3   Contract.Memory.Rsd<Nodo>(Contract.Memory.This, 1);
4   ...
5   Contract.Memory.AddRsd(Contract.Memory.This, Contract.Memory.Return);
6   this._head = this.ProcesarNodo(nodo);
7   ...
8 }
```

Ejemplo 28: Ejemplo de verificación de `AddRsd`

En este caso, para verificar la correctitud de la anotación de la línea 5 debemos verificar dos cosas: primero, que el objeto transferido del método invocado (`ProcesarNodo`) escape del método, y segundo, que si escapa, lo haga a través de la expresión asociada al residual local.

Debemos buscar la expresión asociada al residual del método invocado, al igual que en el ejemplo anterior, el objeto escapa por `return`, entonces la expresión asociada es `this._head`.

Primero verificamos que $escapes\_expr(ProcesarLista,\ this.\_head)$ sea verdadero, es decir, que el objeto al que apunta `this._head` escape del método, en este caso lo será porque el nodo `this` escapa por ser un parámetro. Luego debemos verificar que escape a través de la expresión asociada al residual `Contract.Memory.This` que es `this`, entonces verificamos que $escapes\_expr\_through\_expr(ProcesarLista,\ this.\_head,\ this)$ sea verdadero, que también lo será porque `this._head` escapa a través de `this`.

# 6. Incrementando las capacidades aritméticas de la verificación

Muchos verificadores estáticos (el de Code Contracts incluido) poseen un módulo aritmético con capacidades muy limitadas; la mayoría no son capaces de verificar la correctitud de relaciones entre expresiones donde aparecen multiplicaciones (por ejemplo, no son capaces de determinar la veracidad de $n^2 \geq n+2 \ \ \forall\, n \geq 2$). En general, sólo son capaces de verificar contratos con expresiones que tienen relaciones lineales.

Como ya vimos en algunos ejemplos, es muy común que aparezcan multiplicaciones en los contratos de consumo de memoria, en el Ejemplo 19 la multiplicación surge de dos loops con el segundo dentro del primero, por lo tanto, para implementar una herramienta capaz de verificar todo tipo de contratos es necesario abordar esta problemática.

En esta sección describimos un conjunto de instrumentaciones adicionales que apoyadas en el uso de alguna herramienta capaz de resolver operaciones aritméticas complejas permitirán incrementar las capacidades de verificación de la solución descripta hasta el momento para soportar contratos con expresiones no lineales.

## 6.1. Ejemplo motivacional

Comenzaremos mostrando un ejemplo para motivar la idea detrás de los algoritmos que describimos en las siguientes secciones.

Dado el siguiente método:

```
1  public void ConsumoCubicoCombinado(int n)
2  {
3    Contract.Requires(n > 0);
4    Contract.Memory.Tmp<C>(n * n * n);
5
6    for (int i = 1; i <= n; i++)
7    {
8      for (int j = 1; j <= n; j++)
9      {
10       ConsumoCuadratico(i, j); // tiene un temporal de tipo C de i * j
11     }
12   }
13
14   ConsumoCubico(n); // tiene un temporal de tipo C de n * n * n
15 }
```

Ejemplo 29: Ejemplo motivacional para verificación de contratos con aritmética no lineal

El contrato dado en el ejemplo es correcto, de la invocación de la línea 10 se debe obtener el máximo temporal en alguna de las invocaciones dentro de los dos loops, este máximo se da cuando `i = n` y `j = n`, dando así un temporal necesario de $n^2$, luego se

debe calcular el máximo temporal entre $n^2$ y $n^3$ (producto de la invocación de la línea 14). Dicho máximo es $n^3$, por lo tanto el contrato es correcto. Sin embargo, el verificador de Code Contracts no es capaz de verificar la correctitud de este contrato[16].

Supongamos que disponemos de un oráculo capaz de aseverar que ciertas relaciones son verdaderas (por ejemplo, que $max(n^2,\ n^3) = n^3$ es cierto $\forall\ n > 0$), y que esta información puede ser provista al verificador. La herramienta que utilizaremos para resolver las operaciones aritméticas complejas debe realizar el trabajo de este oráculo y la información que nos brinda la podemos inyectar en el código reemplazando algunas de las instrumentaciones descriptas en la Sección 4 (Verificación de consumo mediante instrumentación), sumado a algunas asunciones que insertaremos, logrando así que el verificador utilizado verifique el contrato dado por el usuario.

Las asunciones las insertaremos utilizando la anotación `Contract.Assume` de Code Contracts (también disponible en otros lenguajes como JML y utilizada por otros verificadores), que recibe una expresión booleana que el verificador incorporará en su base de conocimiento durante la verificación asumiendo su correctitud, nos aseguraremos de proveer siempre expresiones correctas, en caso de no hacerlo podríamos generar una inconsistencia lógica.

Notar que, como se desprenderá de los algoritmos presentados a continuación, es necesario que todos los loops sean anotados con el espacio de iteración utilizando la anotación `Contract.Memory.IterationSpace`. En caso de no estar presente ésta en un loop, no se podrá hacer la verificación utilizando las técnicas que presentamos a continuación y se hará la instrumentación regular, descripta en secciones anteriores. En un futuro pensamos extraer el espacio de iteración calculado por el verificador (si es que lo calcula) para utilizarlo en estas técnicas de verificación.

## 6.2. Requisitos de la herramienta aritmética

La herramienta aritmética que utilicemos debe ser capaz de resolver las operaciones para polinomios con varias variables que definimos a continuación.

$\mathbb{E}[\overline{P}]$ denota una expresión paramétrica entera en función de las variables del conjunto $\overline{P}$, por ejemplo:

- $\mathbb{E}[\overline{P}] = n^2 + m^2$ con $\overline{P} = \{n,\ m\}$.
- $\mathbb{E}[\overline{P}] = x + y$ con $\overline{P} = \{x\}$.

$\mathbb{E}_b[\overline{P}]$ denota una expresión paramétrica booleana en función de las variables del conjunto $\overline{P}$, por ejemplo:

[16] Esto sucede con la mayoría de los verificadores estáticos, dado que los mismos no están preparados para resolver aritmética si no para manejar aserciones lógicas.

- $\mathbb{E}_b[\overline{P}] = n^2 \geq 0$ con $\overline{P} = \{n\}$.
- $\mathbb{E}_b[\overline{P}] = 1 \leq x \leq y$ con $\overline{P} = \{x\}$.

Las operaciones que requerimos son:

- $count(inv,\ freeVars) : \mathbb{E}_b[\overline{P}] \times \overline{P'} \mapsto \mathbb{E}[\overline{P'}]$ con $\overline{P'} \subseteq \overline{P}$

  Cuenta la cantidad de puntos enteros restringidos a la expresión paramétrica $inv$ considerando a $\overline{P}$ como el conjunto variables libres en $inv$ (el resto de las variables son ligadas, para el uso que le daremos, por un ciclo), devolviendo una expresión paramétrica en función las variables libres.

  **Ejemplo:** $count(1 \leq i \leq n\ \wedge\ 1 \leq j \leq m,\ \{n,\ m\}) = n * m$

- $maximize(e,\ inv,\ freeVars) : \mathbb{E}[\overline{P}] \times \mathbb{E}_b[\overline{P}] \times \overline{P'} \mapsto \mathbb{E}[\overline{P'}]$ con $\overline{P'} \subseteq \overline{P}$

  Determina el máximo valor entero que alcanza la expresión $e$ bajo las condiciones $inv$, considerando a $\overline{P}$ como el conjunto variables libres; devolviendo una expresión paramétrica que representa este valor máximo en función de las variables libres.

  **Ejemplo:** $maximize(i * j,\ 1 \leq i \leq n\ \wedge\ 1 \leq j \leq m,\ \{n,\ m\}) = n * m$

- $summate(e,\ inv,\ freeVars) : \mathbb{E}[\overline{P}] \times \mathbb{E}_b[\overline{P}] \times \overline{P'} \mapsto \mathbb{E}[\overline{P'}]$ con $\overline{P'} \subseteq \overline{P}$

  Determina la suma de todos los puntos enteros de la expresión $e$ que cumplen las condiciones dadas por $inv$, considerando a $\overline{P}$ como el conjunto variables libres; devolviendo una expresión paramétrica que representa esta suma en función de las variables libres.

  **Ejemplo:** $summate(i,\ 1 \leq i \leq n,\ \{n\}) = \frac{n*(n+1)}{2}$

- $greater(e_1,\ \ldots,\ e_n,\ inv) : \mathbb{E}[\overline{P}] \times \ldots \times \mathbb{E}[\overline{P}] \times \mathbb{E}_b[\overline{P}] \mapsto \mathbb{E}[\overline{P}] \times \mathbb{E}_b[\overline{P}]$

  Determina cuál es la mayor expresión dentro de $e_1,\ \ldots,\ e_n$ bajo las condiciones $inv$. Dado que esta expresión puede no ser única, el resultado podrá estar restringido a una condición que también es devuelta.

  **Ejemplo 1:** $greater(n^2,\ n^3,\ n \geq 0) = (n^3,\ n \geq 0)$

  **Ejemplo 2:** $greater(n^2,\ n + 2,\ n \geq 0) = (n^2,\ n \geq 2)$[17]

En nuestra implementación utilizaremos, para resolver las operaciones descriptas, la herramienta Barvinok [CFGV09]. Esta herramienta en particular soporta para las expresiones enteras sólo expresiones polinomiales (con varias variables) y para las expresiones booleanas sólo expresiones lineales (también con varias variables); sin embargo nada impide reemplazar esta herramienta en un futuro por otra con mayores capacidades.

[17] Notar que en este caso existe más de un posible resultado, sin embargo, por el momento el prototipo desarrollado sólo es capaz de operar con uno solo.

Definiremos una operación auxiliar para determinar las variables libres de un método. La operación $FreeVars(m)$ devuelve todas las subexpresiones a izquierda del cuerpo del método $m$ que comienzan con *this* o con alguno de los parámetros del método y que no se modifican durante la ejecución del mismo. Por ejemplo, para el siguiente método:

```
1  public void BuscarElemento(List<Nodo> lista, Logger logger)
2  {
3    logger.File.Open();
4    logger.Log("Inicio");
5    for (int i = 0; i < lista.Count; i++)
6    {
7      if (lista[i] == this.ElementoBuscado)
8      {
9        this.ElementoEncontrado = lista[i];
10       logger.Log("Elemento encontrado");
11       break;
12     }
13   }
14   logger.File.Close();
15 }
```

Ejemplo 30: Ejemplo de *FreeVars*

En este caso $FreeVars(BuscarElemento) = \{this,\ lista,\ logger,\ logger.File,\ lista.Count,\ this.ElementoBuscado\}$

En las siguientes secciones describimos la forma de utilizar las operaciones definidas para incrementar las capacidades de verificación de contratos con aritmética compleja para verificadores que no tienen estas capacidades.

## 6.3. Memoria temporal

Para un temporal de tipo `T` en el método `M` de la clase `C` con los siguientes contratos:

```
Contract.Memory.Tmp<T>(lim_1, cond_1);

  ⋮

Contract.Memory.Tmp<T>(lim_k, cond_k);

  ⋮

Contract.Memory.Tmp<T>(lim_n, cond_n);
```

Para este temporal anotado, en las siguientes subsecciones describimos cómo reemplazar, en caso de ser posible, la instrumentación descripta en la Sección 4 (Verificación

de consumo mediante instrumentación) por una nueva instrumentación para tres casos: anotaciones adentro del cuerpo de loops, cálculo del máximo de memoria temporal de las invocaciones hechas y, finalmente, la verificación global (a nivel método) del contrato mismo.

### 6.3.1. Loops

Para todos los loops anotados con `IterationSpace`, consideremos a `inv` como el espacio de iteración anotado en el loop; en caso de subloops, todos los subloops deben tener un espacio de iteración anotado y se debe considerar a `inv` como todos ellos unidos por la operación lógica $y$:

- Para cada `DestTmp` donde el objeto creado en la línea siguiente es de tipo `T`, en lugar de incrementar el contador correspondiente como se describió en los algoritmos de instrumentación, calcular:

  $$cant\_directa\_loop\_i = count(inv,\ FreeVars(M))$$

  Es decir, la suma de la cantidad de veces que se hace un *new* restringido al invariante. $i$ es el número de ocurrencia de este `DestTmp` entre todos los loops.

  Luego, insertar después del loop más externo el statement:

  ```
  C_M_tmp_T += cant_directa_loop_i;
  ```

- Para cada invocación donde el método invocado tiene un contrato de memoria temporal de tipo `T`, en lugar de hacer la instrumentación calcular:

  $$max\_call\_loop\_i = maximize(tmp\_call,\ inv,\ FreeVars(M))$$

  Donde $i$ es el número de invocación entre todos los loops y $tmp\_call$ es la expresión del temporal de tipo `T` del método llamado (asumimos que es único, no se soportan en este caso contratos de memoria temporal con condiciones dado que implica una explosión exponencial en la instrumentación[18]).

  Esta expresión calcula el máximo valor que puede alcanzar la memoria temporal dentro del loop, reemplazando el máximo calculado en la instrumentación regular con `Math.Max`.

  Luego, insertar afuera del loop más externo:

  ```
  max_i_T = max_call_loop_i;
  ```

[18] Una alternativa es tomar el máximo entre todos los contratos, sin embargo esto no siempre es posible y puede generar una pérdida de precisión importante; por esta razón en estos casos se realiza la instrumentación regular que en algunos casos es verificable.

Donde `max_i_T` es la variable insertada antes del loop por la instrumentación explicada anteriormente en la Sección 4.2.4 (Invocaciones dentro de loops) utilizada para calcular el máximo de todas las invocaciones dentro de loops.

- Para cada `AddTmp(rsd)` donde existe un residual de nombre `rsd` y tipo `T` en el siguiente método invocado, en lugar de insertar la suma correspondiente calcular:

$$cant\_transf\_loop\_i = summate(rsd\_call,\ inv,\ FreeVars(M))$$

Donde $i$ es el número de `AddTmp` entre todos los loops y $rsd\_call$ es la expresión del residual con nombre `rsd` y tipo `T` del método llamado (al igual que con los temporales, no se soportan residuales con condiciones).

Esta expresión calcula la suma de memoria residual transferida del método llamado al temporal local.

Luego, insertar afuera del loop más externo:

```
C_M_tmp_T += cant_transf_loop_i;
```

### 6.3.2. Máximo de memoria temporal de invocaciones

En la Sección 4.2.3 (Invocaciones fuera de loops) describimos en la ecuación (1) el incremento del contador correspondiente al temporal de tipo `T` con el máximo de los temporales de todas las invocaciones hechas (recordar que a este máximo luego se incorporan los máximos calculados dentro de loops).

El verificador de Code Contracts no es capaz de resolver esta maximización para casos no lineales o con más de una variable involucrada, pero utilizando las operaciones definidas podremos resolverla correctamente.

Si para todas las invocaciones hechas dentro de loops se pudo calcular un polinomio que represente el consumo de memoria temporal dentro de ese loop, entonces se tendrán una serie de expresiones $tmp_1,\ \ldots\ ,\ tmp_n$ que representan el requerimiento de memoria temporal de todos los métodos invocados; en ese caso, podemos calcular:

$$max\_calls = greater(tmp_1,\ \ldots\ ,\ tmp_n,\ inv)$$

Donde $inv$ son todas las precondiciones en los contratos del método (condiciones dadas con `Contract.Requires`).

Y luego reemplazar la instrumentación hecha en (1) por:

```
C_M_tmp_T += max_calls;
```

Este incremento puede estar restringido a una condición, en cuyo caso lo hacemos bajo un `if` y si no se cumple la condición hacemos que falle la verificación insertando una anotación `Contract.Assert(false);`.

En el Ejemplo 29 presentado al comienzo de la sección se debería calcular el máximo de $n^2$, calculado usando el segundo algoritmo de la Sección 6.3.1 (Loops) y $n^3$, resultado de la llamada a `ConsumoCubico`. Usando *greater* obtenemos que este máximo es $n^3$ siempre que `n >= 0` por lo que insertamos el siguiente código en lugar de la instrumentación:

```
1 if (n >= 0)
2 {
3   C_M_tmp_T += n * n * n;
4 }
5 else
6 {
7   Contract.Assert(false);
8 }
```

Ejemplo 31: Ejemplo de código insertado de máximo calculado con *greater*

En caso de que no se cumpla la condición hacemos que falle la verificación usando `Contract.Assert(false)` porque no sabemos qué sucede con el contador dado que la herramienta aritmética no fue capaz de calcularlo.

Notar que esta instrumentación pierda la sensibilidad a flujo para el análisis y genera una sobre-aproximación, en futuras versiones pensamos solucionar este problema.

### 6.3.3. Verificación global

Con las instrumentaciones descriptas hasta el momento, se insertan en el código varios resultados pre-calculados que muchos verificadores no son capaces de calcular; sin embargo, aún existirán contratos que no serán capaces de ser verificados. Supongamos el siguiente método:

```
1  public void ConsumoCubico(int n)
2  {
3    Contract.Requires(n >= 2);
4    Contract.Memory.Tmp<C>(n * n * n);
5
6    for (int i = 1; i <= n; i++)
7    {
8      Contract.Memory.DestTmp();
9      C c = new C();
10   }
11
```

```
12    ConsumoCuadratico(n); // tiene un temporal de tipo C de n * n
13  }
```

Ejemplo 32: Ejemplo de contrato no verificable por Code Contracts

El consumo exacto de este método es $n + n^2$ ($n$ en el loop y $n^2$ en la invocación de la línea 12), dado que $n^3 \geq n^2 + n \ \ \forall\ n \geq 2$ el contrato dado en el método es correcto; sin embargo contiene una relación no lineal.

Utilizaremos las funciones definidas para intentar verificar la correctitud del contrato con el oráculo, y en caso de ser cierto insertaremos un `Contract.Assume` que perimitará al verificador verificar el contrato. Lo que hacemos en este caso es adelantar la verificación, y en caso de poder hacerla le informamos al verificador mediante el `Contract.Assume` que el contrato es correcto.

En rasgos generales, generaremos una expresión resultado de la suma de todas las expresiones en que se incrementa el contador `C_M_tmp_T` para luego verificar con la herramienta aritmética si dicha expresión es menor o igual que la expresión dada en el contrato, teniendo en cuenta las posibles condiciones dadas para cada contrato. A continuación describimos en detalle la forma en que esto se debe implementar.

Sean las siguientes expresiones:

- $cant\_directa\_loop\_i$: definido anteriormente, es la cantidad de `DestTmp` hechas adentro de loops.

  $i$ va desde 1 hasta $num\_DestTmp$, el número total de `DestTmp` anotados en el método.

- $max\_call\_loop\_i$: definido anteriormente, es el máximo de cada invocación adentro de loops.

  $i$ va desde 1 hasta $num\_CallsLoops$, el número total de invocación hechas dentro de loops.

- $cant\_transf\_loop\_i$: definido anteriormente, es la cantidad de temporal transferida desde residuales al temporal local adentro de loops.

  $i$ va desde 1 hasta $num\_AddTmpLoops$, el número total de `AddTmp` anotados en el método dentro de loops.

- $cant\_directa$: cantidad de news de tipo `T` hechos fuera de loops anotados con `DestTmp`.

- $tmp\_call\_i$: es la cantidad de memoria temporal de tipo `T` de las invocaciones fuera de loops (`i` es el número de invocación fuera de loops).

  $i$ va desde 1 hasta $num\_Calls$, el número total de invocaciones hechas fuera de loops.

- $cant\_transf\_i$: es la cantidad de temporal transferida desde residuales al temporal local fuera de loops (`i` es el número de invocación fuera de loops).

$i$ va desde 1 hasta $num\_AddTmp$, el número total de `AddTmp` anotados en el método fuera de loops.

Sea la expresión:

$$cand\_tmp = \sum_{i=1}^{num\_DestTmp} cant\_directa\_loop\_i \; + \; cant\_directa \; +$$

$$\sum_{i=1}^{num\_AddTmpLoops} cant\_transf\_loop\_i \; + \sum_{i=1}^{num\_AddTmp} cant\_transf\_i \; +$$

$$greater(max\_call\_loop\_1, \; \ldots, \; max\_call\_loop\_num\_CallsLoops,$$

$$tmp\_call\_1, \; \ldots, \; tmp\_call\_num\_Calls, \; preconditions)$$

Donde $preconditions$ son las precondiciones del método `M`.

$cand\_tmp$ es la suma de la memoria consumida directamente por $news$ (dentro y fuera de loops), más la memoria transferida de residuales de invocaciones al temporal local (dentro y fuera de loops) más el máximo de memoria temporal de todas las invocaciones hechas (dentro y fuera de loops).

Calcular para cada $1 \leq k \leq n$:

$$max = greater(cand\_tmp, \; lim\_k, \; cond\_k)$$

En caso de no haber una condición, utilizar $true$ como invariante.

Si se determina que $max = lim\_k$, es decir, se cumple que $cand\_tmp \leq lim\_k$ bajo $cond\_k$, insertar el statement:

```
Contract.Assume(C_M_tmp_T <= lim_k);
```

Si se determina que $max = lim\_k$ bajo ciertas condiciones `conds`, insertar el statement:

```
if (conds)
{
  Contract.Assume(C_M_tmp_T <= lim_k);
```

```
}
else
{
  Contract.Assert(false);
}
```

De esta forma, el verificador podrá utilizar estos assumes para determinar la correctitud de los contratos dados por el usuario, siempre y cuando se cumplan las condiciones, caso contrario fallará la verificación.

## 6.4. Memoria residual

Las instrumentaciones y verificaciones hechas para la memoria residual son muy similares a las presentadas para la memoria temporal, pero sin tener el cuenta el cálculo del máximo de la memoria requerida por los métodos invocados.

Para un residual de tipo `T` y nombre `rsd_local` en el método `M` de la clase `C` con los siguientes contratos:

```
Contract.Memory.Rsd<T>(rsd_local, lim_1, cond_1);

  ⋮

Contract.Memory.Rsd<T>(rsd_local, lim_k, cond_k);

  ⋮

Contract.Memory.Rsd<T>(rsd_local, lim_n, cond_n);
```

Para este residual anotado, en las siguientes subsecciones describimos cómo reemplazar, en caso de ser posible, la instrumentación descripta en la Sección 4 (Verificación de consumo mediante instrumentación) por una nueva instrumentación para dos casos: anotaciones adentro del cuerpo de loops y la verificación global (a nivel método) del contrato mismo.

### 6.4.1. Loops

Para todos los loops anotados con `IterationSpace`, consideremos a `inv` como el espacio de iteración anotado en el loop; en caso de subloops, todos los subloops deben tener un espacio de iteración anotado y se debe considerar a `inv` como todos ellos unidos por la operación lógica $y$:

- Para cada `DestRsd(rsd_local)` donde el objeto creado en la línea siguiente es de

tipo `T`, en lugar de incrementar el contador correspondiente como se describió en los algoritmos de instrumentación, calcular:

$$cant\_directa\_loop\_i = count(inv,\ FreeVars(M))$$

Es decir, la suma de la cantidad de veces que se hace un *new* restringido al invariante. $i$ es el número de ocurrencia de este `DestRsd(rsd_local)` entre todos los loops.

Luego, insertar después del loop más externo el statement:

```
C_M_rsd_local_T += cant_directa_loop_i;
```

- Para cada `AddRsd(rsd_local, rsd_method)` donde existe un residual de nombre `rsd_method` y tipo `T` en el siguiente método invocado, en lugar de insertar la suma correspondiente calcular:

$$cant\_transf\_loop\_i = summate(rsd\_call,\ inv,\ FreeVars(M))$$

Donde $i$ es el número de `AddRsd(rsd_local)` entre todos los loops y *rsd_call* es la expresión del residual con nombre `rsd` y tipo `T` del método llamado (al igual que con los temporales, no se soportan residuales con condiciones).

Esta expresión calcula la suma de memoria residual transferida del método llamado al residual local.

Luego, insertar afuera del loop más externo:

```
C_M_rsd_local_T += cant_transf_loop_i;
```

### 6.4.2. Verificación global

De la misma forma que con la memoria temporal, hacemos una verificación de los contratos e insertamos los `Contract.Assume` correspondientes en caso de poder hacerlo.

En rasgos generales, generaremos una expresión resultado de la suma de todas las expresiones en que se incrementa el contador `C_M_rsd_local_T` para luego verificar con la herramienta aritmética si dicha expresión es menor o igual que la expresión dada en el contrato, teniendo en cuenta las posibles condiciones dadas para cada contrato. A continuación describimos en detalle la forma en que esto se debe implementar.

Sean las siguientes expresiones:

- *cant_directa_loop_i*: definido anteriormente, es la cantidad de `DestRsd(rsd_local)` hechas adentro de loops.

  $i$ va desde 1 hasta *num_DestRsd*, el número total de `DestRsd` anotados en el método.

- $cant\_transf\_loop\_i$: definido anteriormente, es la cantidad de residual transferida desde residuales al residual local adentro de loops.

  $i$ va desde 1 hasta $num\_AddRsdLoops$, el número total de `AddRsd` anotados en el método dentro de loops.

- $cant\_directa$: cantidad de news de tipo `T` hechos fuera de loops anotados con `DestRsd(rsd_local)`.

- $cant\_transf\_i$: es la cantidad de residual transferida desde residuales al residual local fuera de loops (`i` es el número de invocación fuera de loops).

  $i$ va desde 1 hasta $num\_AddRsd$, el número total de `AddRsd` anotados en el método fuera de loops.

Sea la expresión:

$$cand\_rsd = \sum_{i=1}^{num\_DestRsd} cant\_directa\_loop\_i \; + \; cant\_directa \; +$$

$$\sum_{i=1}^{num\_AddRsdLoops} cant\_transf\_loop\_i \; + \sum_{i=1}^{num\_AddRsd} cant\_transf\_i$$

$cand\_rsd$ es la suma de la memoria consumida directamente por *news* (dentro y fuera de loops) más la memoria transferida de residuales de invocaciones al residual local (dentro y fuera de loops).

Calcular para cada $1 \leq k \leq n$:

$$max = greater(cand\_rsd, \; lim\_k, \; cond\_k)$$

En caso de no haber una condición, utilizar *true* como invariante.

Si se determina que $max = lim\_k$, es decir, se cumple que $cand\_rsd \leq lim\_k$ bajo $cond\_k$, insertar el statement:

```
Contract.Assume(C_M_rsd_local_T <= lim_k);
```

Si se determina que $max = lim\_k$ bajo ciertas condiciones `conds`, insertar el statement:

```
if (conds)
{
  Contract.Assume(C_M_rsd_local_T <= lim_k);
```

```
}
else
{
  Contract.Assert(false);
}
```

De esta forma, el verificador podrá utilizar estos assumes para determinar la correctitud de los contratos dados por el usuario, siempre y cuando se cumplan las condiciones, caso contrario fallará la verificación.

# 7. Overview de la implementación

## 7.1. Arquitectura genérica

Comenzaremos primero describiendo la arquitectura de la solución genéricamente sin tener en cuenta la implementación particular hecha, describiendo los requerimientos de cada componente. Luego, en las siguientes secciones detallamos la arquitectura de la herramienta desarrollada.

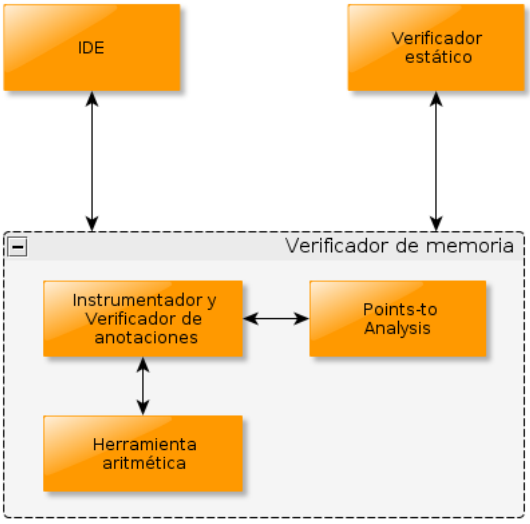


Figura 2: Arquitectura genérica

En la Figura 2 vemos la arquitectura genérica de la solución propuesta a lo largo del trabajo.

En la Figura 3 vemos un diagrama de secuencia que describe el flujo del control durante la verificación. La IDE es la encargada de iniciar la verificación luego de la compilación del código. Según la plataforma, la IDE enviará al verificador de memoria el código compilado o el código fuente, luego el verificador de memoria hará la verificación y devolverá a la IDE el resultado de la verificación para que sea mostrada al usuario.

El verificador de memoria realiza la instrumentación del código y la verificación de anotaciones como se describió en las secciones anteriores, utilizando el análisis de points-to y escape para obtener información del tiempo de vida de los objetos y la herramienta aritmética para enriquecer el código instrumentado con información que ayude a la verificación.

El código instrumentado es enviado al verificador estático, el cual verifica la correctitud de los nuevos contratos insertados y devuelve los resultados al verificador de memoria, éste deberá agregar a estos resultados los de la verificación de anotaciones hecha para devolverlos a la IDE.

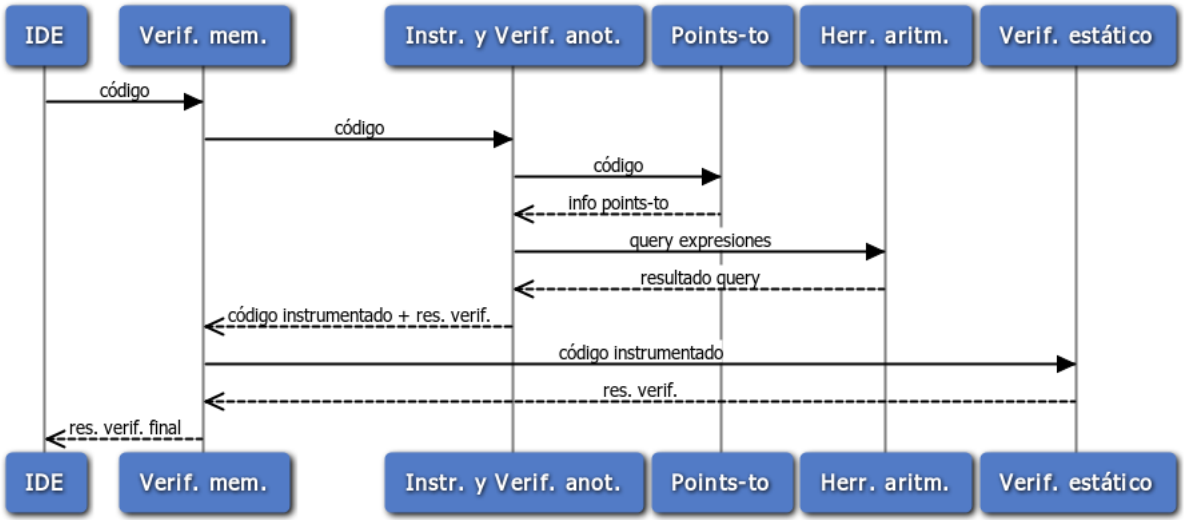


Figura 3: Diagrama de secuencia mostrando el flujo del control durante la verificación

A partir de ahora nos concentraremos en explicar la forma en que esta arquitectura se utilizó en la implementación de prueba de concepto hecha para este trabajo, utilizando Microsoft Visual Studio como IDE, el verificador de Code Contracts como verificador estático y como verificador de memoria una implementación de los algoritmos descriptos junto a la herramienta aritmética Barvinok y el análisis de points-to y escape de [BFGL07].

## 7.2. Plataforma y herramientas utilizadas

Como ya mencionamos, hicimos una implementación de la solución propuesta para la plataforma .NET usando el analizador estático de la herramienta Code Contracts (Clousot, [FL09]). Elegimos esta plataforma y herramienta dado que son activamente desarrolladas y mantenidas en la actualidad, además tienen una sintaxis para escribir anotaciones muy intuitiva e integrada con la IDE (Microsoft Visual Studio) que fácilmente podemos extender para soportar nuestras nuevas anotaciones.

El verificador estático de Code Contracts (Clousot) utiliza técnicas de interpretación abstracta para intentar demostrar la correctitud de los contratos, a diferencia de los verificadores estáticos más comunes que se basan en calcular y demostrar la precondición más débil. El uso de interpretación abstracta permite al verificador inferir automáticamente los invariantes de ciclo e integrar en la verificación diferentes heurísticas para ajustar el tradeoff entre el costo de la verificación y su precisión.

Dado que en Code Contracts los contratos son código embebido, pudimos extender estos contratos permitiendo al usuario escribir los nuevos contratos de consumo de memoria de la misma forma que lo harían con contratos de Code Contracts, aprovechando las ventajas que la IDE brinda, tales como documentación inline mientras se escriben las anotaciones y chequeo de tipos y sintaxis de las mismas.

Una ventaja importante es que la plataforma .NET cuenta con un framework para análisis estático e instrumentación de código llamada `CCI` (Common Compiler Infraestructure: [Res10b], [Res10a]) que es de código abierto, tiene buena documentación de uso y soporte nativo para inyectar contratos de Code Contracts.

Otra característica interesante de Code Contracts es que todos los contratos escritos por el usuario quedarán embebidos en el código compilado, por lo que al distribuirlo se estarán proveyendo contratos de consumo de memoria en caso de que el código estuviese anotado; también es posible distribuir los contratos y el código en archivos separados.

## 7.3. Arquitectura de la solución implementada

Para comprender la arquitectura de la herramienta desarrollada y el flujo de los elementos entre sus diferentes partes, es primero conveniente conocer la forma en que funciona la verificación estática hecha por Code Contracts, dado que nuestra solución es una extensión de la misma.

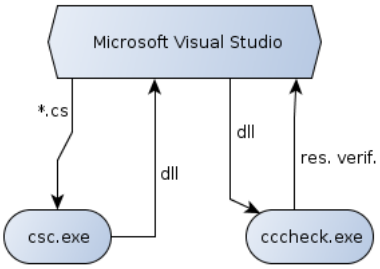


Figura 4: Arquitectura de la verificación estática de Code Contracts

La verificación comienza cuando el usuario compila el código en la IDE (Ver Figura 4). Todos los archivos con extensión `cs` (asumiendo código C#) de un mismo proyecto son enviados al compilador `csc.exe` que devuelve una `dll` con el código compilado, luego, en caso de tener activada la verificación estática, esta `dll` es enviada al verificador estático de Code Contracts llamado `cccheck.exe` el cual analiza los contratos, intenta verificarlos y devuelve el resultado a la IDE (este resultado es un texto con un formato acordado) la cual muestra los resultados al usuario en una forma amigable. Vale aclarar que este proceso se realiza por separado para cada proyecto compilado (un proyecto equivale a una `dll` o assembly en .NET).

La extensión que hacemos del verificador nativo, en rasgos generales, intercepta la `dll` antes de que sea enviada a `cccheck.exe`, la analiza y modifica y luego la envía a `cccheck.exe`, tomando su salida para devolvérsela a la IDE agregándole resultados de ciertas verificaciones hechas. En la Figura 5 se puede ver la arquitectura, ahora explicaremos en detalle el flujo de la información en esta arquitectura y la responsabilidad de cada componente. Los números entre paréntesis indican el orden dentro del flujo del control durante la ejecución.

Al igual que en el caso anterior, la verificación comienza cuando el usuario compila

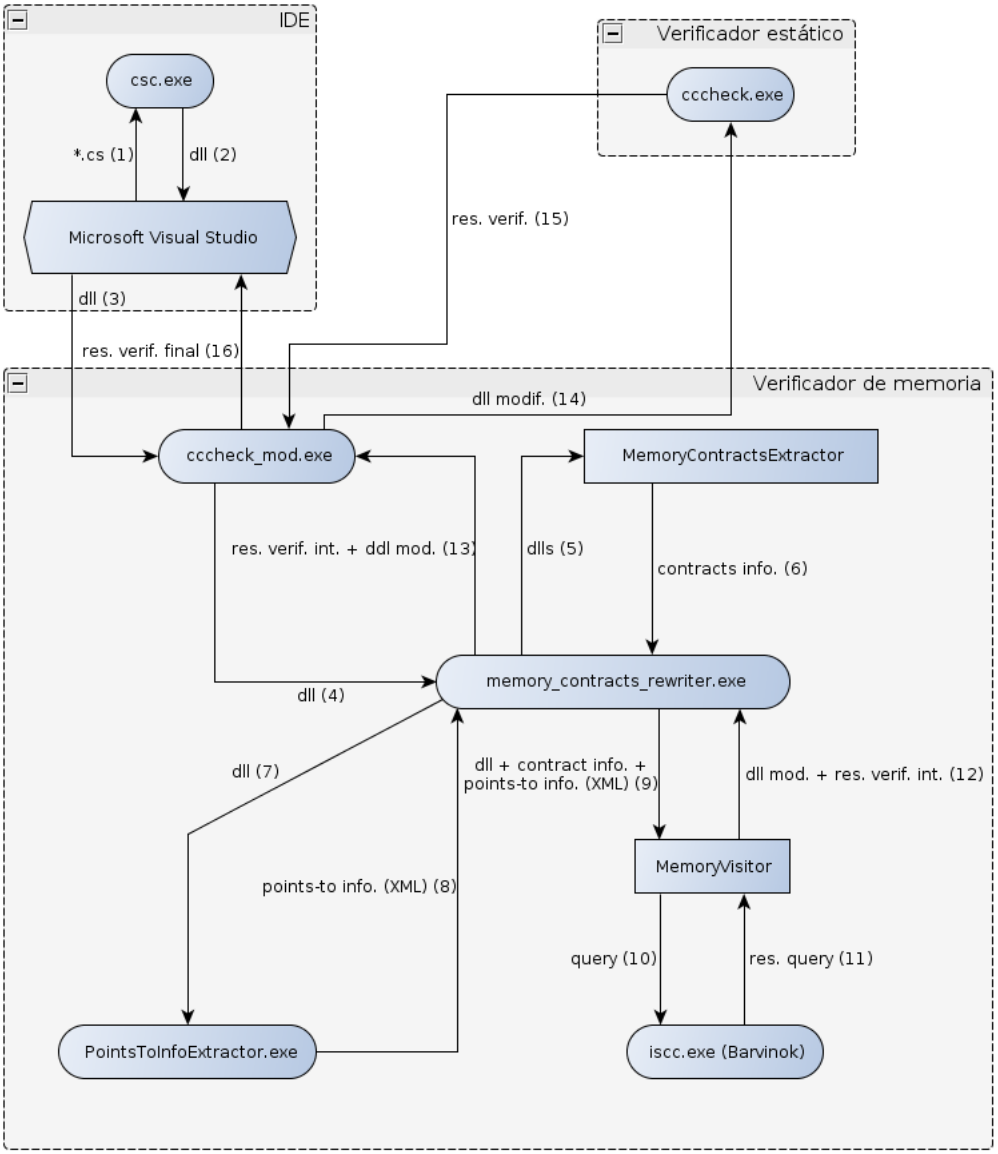


Figura 5: Arquitectura de la verificación estática de Code Contracts con memoria

el código en la IDE, luego de enviarse al compilador (1) y obtener la `dll` (2) ésta es enviada a verificar. Modificando la configuración de Code Contracts es posible hacer que la IDE envíe la `dll` a verificar a nuestro componente, llamado `cccheck_mod.exe` en lugar de al componente de Code Contracts (3). Dado que este componente respeta la misma interfaz que `cccheck.exe` (recibe una `dll` y devuelve en la salida estándar un texto con el resultado de la verificación), el cambio es transparente para la IDE.

Lo siguiente que hace `cccheck_mod.exe` es enviar la `dll` al componente llamado `memory_contracts_rewriter.exe` (4), éste es el componente central de nuestra herramienta, encargado de coordinar la instrumentación del código y verificación de las anotaciones. El mismo se encarga de parte de la verificación pero no toda, dado que la correctitud de ciertas anotaciones es verificada por este componente (por ejemplo de `DestTmp` y `DestRsd`) mientras que otras (como los contratos) son transformadas en

contratos nativos de Code Contracts que luego `cccheck.exe` verificará gracias a la instrumentación de código hecha.

`memory_contracts_rewriter.exe` primero extrae de la `dll`, utilizando otras componentes, un conjunto de información necesaria para la siguiente tarea. Utilizando la clase `MemoryContractsExtractor` (5 y 6), implementada con `CCI`, extrae los contratos de memoria de la `dll` y de todas las `dlls` a las que ésta referencia (notar que si en un futuro los contratos se distribuyen separados de las `dlls` compiladas, se debería modificar sólo esta clase). Además, utilizando el componente `PointsToInfoExtractor.exe` obtiene un Points-to Graph junto con información de escape (en formato XML) de todos los métodos de la `dll` (7 y 8). Este componente está basado en un análisis disponible en `CCI` hecho por Diego Garbervetsky para Spec#, implementado según [BFGL07].

Luego, una vez recolectada esta información, la misma junto a la `dll` original se envía a la clase `MemoryVisitor` (9), esta clase está implementada con `CCI` y se encarga de analizar cada método de cada clase contenida en la `dll`, este análisis realiza tres tareas principales:

- Hace la instrumentación de código necesaria e inserción de contratos de Code Contracts (generados a partir de contratos de memoria), para la posterior verificación. Para esto se utilizan los algoritmos descriptos en la Sección 4 (Verificación de consumo mediante instrumentación).
- Verifica la correctitud de otras anotaciones que no pueden ser verificadas por `cccheck.exe`. Estas verificaciones se realizan utilizando las técnicas descriptas en la Sección 5 (Verificación de correctitud de las anotaciones de tiempo de vida).
- Haciendo uso de `iscc.exe` (la calculadora de la herramienta `Barvinok`) se intenta obtener aserciones sobre el consumo de memoria que son inyectadas en el código junto a los contratos (10 y 11), ayudando así a la posterior verificación; se utilizan para estos los algoritmos descriptos la Sección 6 (Incrementando las capacidades aritméticas de la verificación).

Para todos los ítems se utiliza la información de contratos extraida anteriormente y para el segundo se utiliza la información de points-to y escape. Como resultado de todo el proceso se obtiene una nueva `dll` con código instrumentado y nuevos contratos insertados, y un conjunto de aserciones resultado de las verificaciones hechas en el segundo punto.

Esta `dll` modificada y las aserciones son devueltas a `memory_contracts_rewriter.exe` (12) el cual a su vez las devuelve a `cccheck_mod.exe` (13), éste envía la `dll` modificada a `cccheck.exe` (14) el cual hace la verificación de los nuevos contratos generados y devuelve el resultado (15), este resultado es unido al resultado de la verificación hecha en `MemoryVisitor` (llamado `res. verif. int.` en el diagrama) y finalmente devuelto a la IDE (16) que muestra los resultados de la verificación hecha por `cccheck.exe` del código instrumentado junto a las verificaciones hechas por `MemoryVisitor` directamente.

# 8. Evaluación

En esta sección mostramos una serie de experimentos hechos utilizando la herramienta desarrollada para evaluar sus capacidades. A lo largo de los mismos veremos la forma en que se realiza la instrumentación y verificación para cada uno de ellos y observaremos algunas limitaciones de la herramienta.

Todos los experimentos consisten en una o más clases con sus métodos anotados con contratos de consumo de memoria y otras anotaciones necesarias para la verificación. Veremos en algunos casos el código instrumentado y en otros analizaremos la forma en la que la misma es resuelta por el verificador.

## 8.1. Verificación en el ejemplo dado en el Anexo

Veremos la forma en que se realiza la instrumentación para algunos métodos de las clases presentadas en la Sección 11.4 (Ejemplo completo anotado) del Anexo.

El ejemplo dado en el Anexo consiste de cuatro clases (`Address`, `AddressValidator`, `Person` y `PeopleManager`). Todos los contratos del mismo son verificables por la herramienta desarrollada y cubre todos los tipos de anotaciones disponibles. Veremos la instrumentación de un método utilizando las técnicas de la Sección 4 (Verificación de consumo mediante instrumentación) y la de otro usando las técnicas de la Sección 6 (Incrementando las capacidades aritméticas de la verificación).

El método `CreateFamily` de la clase `PeopleManager` no puede ser instrumentado utilizando las técnicas de la Sección 6 (Incrementando las capacidades aritméticas de la verificación) dado que tiene un ciclo que no está anotado con el espacio de iteración, sin embargo puede ser verificado con la instrumentación regular porque sus contratos sólo tienen expresiones lineales. El siguiente es el código de dicho método instrumentado (los nombres de los campos para contadores se renombraron por cuestiones de espacio):

```
1  public static Person[] CreateFamily(List<string> firstNames, string lastName, string
       street, string city, string state)
2  {
3      Contract.Requires(firstNames.Count > 0);
4      Contract.Ensures(tmp_AddressValidator <= 1);
5      Contract.Ensures(rsd_Return_Address <= firstNames.Count);
6      Contract.Ensures(rsd_Return_Person <= firstNames.Count);
7      Contract.Ensures(rsd_Return_Person_arr <= 1);
8      rsd_Return_Address = 0;
9      rsd_Return_Person = 0;
10     rsd_Return_Person_arr = 0;
11     int tmp_call_1 = 0;
12     tmp_AddressValidator = 0;
13     Contract.Memory.Rsd<Person>(Contract.Memory.Return, firstNames.Count);
14     Contract.Memory.Rsd<Person[]>(Contract.Memory.Return, 1);
15     Contract.Memory.Rsd<Address>(Contract.Memory.Return, firstNames.Count);
16     Contract.Memory.Tmp<AddressValidator>(1);
17
18     Contract.Memory.DestRsd(Contract.Memory.Return);
19     rsd_Return_Person_arr++;
20     Person[] family = new Person[firstNames.Count];
```

```
21
22      for (int i = 0; i < firstNames.Count; i++)
23      {
24          Contract.Memory.AddRsd(Contract.Memory.Return, Contract.Memory.This);
25          Contract.Memory.DestRsd(Contract.Memory.Return);
26          rsd_Return_Person++;
27          Person person = new Person(firstNames[i], lastName, street, city, state);
28          rsd_Return_Address += Person.rsd_This_Address;
29          tmp_call_1 = Math.Max(tmp_call_1, Person.tmp_AddressValidator);
30          family[i] = person;
31      }
32      tmp_AddressValidator += tmp_call_1;
33      return family;
34  }
```

Ejemplo 33: Método `CreateFamily` instrumentado

En las líneas 4 a 7 se ven los contratos insertados por la herramienta, luego en las líneas 8 a 12 se ve la inicialización de los contadores en 0 y la declaración de la variable local `tmp_call_1` que será utilizada para calcular el máximo del temporal de la invocación de la línea 27 (el *new*).

Adentro del loop, se incrementan los contadores correspondientes a la creación del objeto `Person` en la línea 24 y de la transferencia del residual del método llamado al residual local. En la línea 29 se encuentra la instrumentación hecha con `Math.Max` para calcular el máximo temporal consumido por la invocación dentro del loop. Este máximo queda al final del loop en la variable `tmp_call_1` que es sumada el contador correspondiente luego del loop.

Hemos comprobado que este tipo de instrumentación utilizando `Math.Max` no funciona para loops anidados o con un espacio de iteración complejo cuando la verificación se hace con el verificador de Code Contracts, para poder verificar contratos bajo esas condiciones es necesario anotar los espacios de iteración, como veremos en el siguiente ejemplo.

Uno de los métodos más complejos en este conjunto de clases es el método `CreateCombinedFamily` de la clase `PeopleManager`. El resultado de la instrumentación de este método es el siguiente (los nombres de los campos para contadores se renombraron por cuestiones de espacio):

```
1   public static Person[] CreateCombinedFamily(List<string> firstNames, List<string>
        lastNames, string street, string city, string state)
2   {
3       Contract.Requires(firstNames.Count > 0);
4       Contract.Requires(firstNames.Count == lastNames.Count);
5       Contract.Ensures(tmp_AddressValidator <= 1);
6       Contract.Ensures(rsd_Return_Address <= (firstNames.Count * lastNames.Count));
7       Contract.Ensures(rsd_Return_Person <= (firstNames.Count * lastNames.Count));
8       Contract.Ensures(rsd_Return_Person_arr <= 1);
9       rsd_Return_Address = 0;
10      rsd_Return_Person = 0;
11      tmp_AddressValidator = 0;
12      Contract.Memory.Rsd<Person>(Contract.Memory.Return, firstNames.Count * lastNames.
            Count);
13      Contract.Memory.Rsd<Person[]>(Contract.Memory.Return, 1);
14      Contract.Memory.Rsd<Address>(Contract.Memory.Return, firstNames.Count * lastNames.
            Count);
15      Contract.Memory.Tmp<AddressValidator>(1);
```

```
16      Contract.Memory.DestRsd(Contract.Memory.Return);
17      Person[] family = new Person[firstNames.Count * lastNames.Count];
18
19      for (int i = 0; i < firstNames.Count; i++)
20      {
21          Contract.Memory.IterationSpace((1 <= i) && (i <= firstNames.Count));
22          for (int j = 0; j < lastNames.Count; j++)
23          {
24              Contract.Memory.IterationSpace((1 <= j) && (j <= lastNames.Count));
25              Contract.Memory.AddRsd(Contract.Memory.Return, Contract.Memory.This);
26              Contract.Memory.DestRsd(Contract.Memory.Return);
27              Person person = new Person(firstNames[i], lastNames[j], street, city, state
                    );
28              family[(i * firstNames.Count) + j] = person;
29          }
30      }
31
32      if ((lastNames.Count >= 1) && (firstNames.Count >= 1))
33      {
34          rsd_Return_Address += firstNames.Count * lastNames.Count;
35      }
36      else
37      {
38          Contract.Assert(false);
39      }
40      if ((lastNames.Count >= 1) && (firstNames.Count >= 1))
41      {
42          rsd_Return_Person += firstNames.Count * lastNames.Count;
43      }
44      else
45      {
46          Contract.Assert(false);
47      }
48      if ((lastNames.Count >= 1) && (firstNames.Count >= 1))
49      {
50          rsd_Return_Person_arr++;
51      }
52      else
53      {
54          Contract.Assert(false);
55      }
56      if ((lastNames.Count == firstNames.Count) && (firstNames.Count >= 1))
57      {
58          tmp_AddressValidator++;
59      }
60      else
61      {
62          Contract.Assert(false);
63      }
64      if ((lastNames.Count == firstNames.Count) && (firstNames.Count >= 1))
65      {
66          Contract.Assume(rsd_Return_Address <= (firstNames.Count * lastNames.Count));
67      }
68      else
69      {
70          Contract.Assert(false);
71      }
72      if ((lastNames.Count == firstNames.Count) && (firstNames.Count >= 1))
73      {
74          Contract.Assume(rsd_Return_Person <= (firstNames.Count * lastNames.Count));
75      }
76      else
77      {
78          Contract.Assert(false);
79      }
80      if ((lastNames.Count == firstNames.Count) && (firstNames.Count >= 1))
81      {
82          Contract.Assume(rsd_Return_Person_arr <= 1);
83      }
84      else
85      {
86          Contract.Assert(false);
```

```
87      }
88      if ((lastNames.Count == firstNames.Count) && (firstNames.Count >= 1))
89      {
90          Contract.Assume(tmp_AddressValidator <= 1);
91      }
92      else
93      {
94          Contract.Assert(false);
95      }
96      return family;
97  }
```

Ejemplo 34: Método `CreateCombinedFamily` instrumentado

Los contratos de las línea 5 a 8 son los insertados por la instrumentación, y son las aserciones que el verificador de Code Contracts intentará verificar. Vemos que para todos los *news* e invocaciones hechas no se incrementan los contadores como se definió en la Sección 4 (Verificación de consumo mediante instrumentación), dado que en este caso se pudo aplicar para todos los casos las técnicas descriptas en la Sección 6 (Incrementando las capacidades aritméticas de la verificación), haciendo así posible realizar una instrumentación más precisa y una verificación adelantada de los contratos.

En las líneas 40 a 47 se incrementa el contador para el residual por `return` de tipo `Person` correspondiente al *new* hecho en la línea 27, esta expresión pudo ser calculada gracias a los espacios de iteración anotados. El incremento se hace bajo las condiciones dadas por Barvinok y se fuerza que falle la verificación en caso de que las mismas no se cumplan.

En las líneas 32 a 39 se hace un incremento similar al anterior pero para el residual de tipo `Address`, producto del `AddRsd` de la línea 25.

En las líneas 56 a 63 se hace el incremento del contador para el temporal de tipo `AddressValidator`. Este consumo de temporal viene del máximo de las invocaciones al constructor de la clase `Person` en la línea 27, que es siempre 1, por lo tanto el máximo bajo las restricciones dadas por los espacios de iteración es 1.

En las líneas 64 a 95 vemos los `Assume` insertados por las verificaciones hechas con Barvinok, los tres asumen la misma expresión dada en los contratos, dado que fue posible verificar con Barvinok que los mismos son correctos.

## 8.2. Contratos con polinomios y uso de varias anotaciones

Daremos en esta sección un conjunto de ejemplos donde los contratos tienen un consumo dependiente de una expresión polinomial. Primero definiremos algunos métodos simples con diferentes tipos de consumo que luego invocaremos en los métodos con contratos complejos, todos los métodos se encuentran dentro de la clase `CompleteTest`:

**Nota:** en estos ejemplos omitiremos los contratos del tipo `CompleteTest[]` dado que sólo nos interesa observar la forma en que se instrumentan los contratos con consumos

no lineales.

```
public void CuadraticTmp(int n)
{
  System.Diagnostics.Contracts.Contract.Requires(n >= 0);

  Contract.Memory.Tmp<CompleteTest>(n * n);

  for (int i = 1; i <= n; i++)
  {
    Contract.Memory.IterationSpace(1 <= i && i <= n);
    for (int j = 1; j <= n; j++)
    {
      Contract.Memory.IterationSpace(1 <= j && j <= n);
      Contract.Memory.DestTmp();
      new CompleteTest();
    }
  }
}

public CompleteTest[] CuadraticRsd(int n, int m)
{
  System.Diagnostics.Contracts.Contract.Requires(n >= 0);
  System.Diagnostics.Contracts.Contract.Requires(m >= 0);

  Contract.Memory.Rsd<CompleteTest>(Contract.Memory.Return, n * m);

  CompleteTest[] elems = new CompleteTest[n * m];

  for (int i = 0; i < n; i++)
  {
    Contract.Memory.IterationSpace(0 <= i && i < n);

    for (int j = 0; j < m; j++)
    {
      Contract.Memory.IterationSpace(0 <= j && j < m);

      Contract.Memory.DestRsd(Contract.Memory.Return);
      elems[i * n + j] = new CompleteTest();
    }
  }

  return elems;
}

public void CuadraticTmpMaxLoop(int n)
{
  System.Diagnostics.Contracts.Contract.Requires(n >= 0);

  Contract.Memory.Tmp<CompleteTest>(n * n);

  for (int i = 1; i <= n; i++)
  {
    Contract.Memory.IterationSpace(1 <= i && i <= n);

    this.CuadraticTmp(i);
  }
```

```
56  }
57
58  public void CuadraticTmpAddLoop(int n)
59  {
60    System.Diagnostics.Contracts.Contract.Requires(n >= 0);
61
62    Contract.Memory.Tmp<CompleteTest>(n / 6 + (n * n) / 2 + (n * n * n) /
          3);
63
64    for (int i = 1; i <= n; i++)
65    {
66      Contract.Memory.IterationSpace(1 <= i && i <= n);
67
68      Contract.Memory.AddTmp(Contract.Memory.Return);
69      CuadraticRsd(i, i);
70    }
71  }
72
73  public CompleteTest[] CuadraticRsdAddLoop(int n)
74  {
75    System.Diagnostics.Contracts.Contract.Requires(n >= 0);
76
77    Contract.Memory.Rsd<CompleteTest>(Contract.Memory.Return, n / 6 + (n
         * n) / 2 + (n * n * n) / 3);
78
79    CompleteTest[] r = null;
80
81    for (int i = 1; i <= n; i++)
82    {
83      Contract.Memory.IterationSpace(1 <= i && i <= n);
84
85      Contract.Memory.AddRsd(Contract.Memory.Return, Contract.Memory.
            Return);
86      r = CuadraticRsd(i, i);
87    }
88
89    return r;
90  }
```

Ejemplo 35: Métodos auxiliares para ejemplo con contratos con polinomios

Resumimos los tipos de contratos de cada método auxiliar:

- `void CuadraticTmp(int n)`: tiene un temporal de tipo `CompleteTest` de `n * n`.
- `CompleteTest[] CuadraticRsd(int n, int m)`: tiene un residual de tipo `CompleteTest` por `return` de `n * m`.
- `void CuadraticTmpMaxLoop(int n)`: tiene un temporal de tipo `CompleteTest` de `n * n` producto de una invocación a `CuadraticTmp` adentro de un loop.
- `void CuadraticTmpAddLoop(int n)`: tiene un temporal de tipo `CompleteTest` de `n / 6 + ( n * n ) / 2 + ( n * n * n ) / 3` producto de una invocación a `CuadraticRsd` adentro de un loop, la expresión sale de la suma $\sum_{i=1}^{n} i^2$.

- `CompleteTest[] CuadraticRsdAddLoop(int n)`: tiene un residual de tipo `CompleteTest` por `return` de `n / 6 + ( n * n ) / 2 + ( n * n * n ) / 3` producto de la invocación adentro de un loop, sale de la misma suma anterior.

Definidos estos métodos, mostraremos dos métodos que los utilizan generando un consumo dependiente de una expresión compleja:

```
1  public void CompleteTmp(int n)
2  {
3    ////cota exacta
4    //System.Diagnostics.Contracts.Contract.Requires(n >= 1);
5    //Contract.Memory.Tmp<CompleteTest>(1 + n * n + n * n + n + n * n + n
         / 6 + (n * n) / 2 + (n * n * n) / 3);
6
7    ////cota superior
8    //System.Diagnostics.Contracts.Contract.Requires(n >= 1);
9    //Contract.Memory.Tmp<CompleteTest>(n * n * n + 4 * n * n + 2 * n);
10
11   //cota superior considerando n == 0, con condiciones
12   System.Diagnostics.Contracts.Contract.Requires(n >= 0);
13   Contract.Memory.Tmp<CompleteTest>(1, n == 0);
14   Contract.Memory.Tmp<CompleteTest>(n * n * n + 4 * n * n + 2 * n, n >=
         1);
15
16   Contract.Memory.DestTmp();
17   new CompleteTest(); //cant_directa --> 1
18
19   CuadraticTmpMaxLoop(n); //tmp_call_1 --> n * n
20
21   Contract.Memory.AddTmp(Contract.Memory.Return);
22   CuadraticRsd(n, n); //cant_transf_1 --> n * n
23
24   for (int i = 1; i <= n; i++)
25   {
26     Contract.Memory.IterationSpace(1 <= i && i <= n);
27
28     Contract.Memory.DestTmp();
29     new CompleteTest(); //cant_directa_loop_1 --> n
30
31     CuadraticTmpMaxLoop(i); //max_call_loop_1 --> n * n
32
33     Contract.Memory.AddTmp(Contract.Memory.Return);
34     CuadraticRsd(i, i); //cant_transf_loop_1 --> n / 6 + (n * n) / 2 +
           (n * n * n) / 3
35   }
36 }
37
38 public CompleteTest[] CompleteRsd(int n)
39 {
40   ////cota exacta
41   //System.Diagnostics.Contracts.Contract.Requires(n >= 1);
42   //Contract.Memory.Rsd<CompleteTest>(Contract.Memory.Return, n * n + 1
         + n / 6 + (n * n) / 2 + (n * n * n) / 3 + n);
43
44   ////cota superior
```

```
45    //System.Diagnostics.Contracts.Contract.Requires(n >= 1);
46    //Contract.Memory.Rsd<CompleteTest>(Contract.Memory.Return, n * n * n
          + 2 * n * n + 2 * n + 1);
47
48    //cota superior considerando n == 0, con condiciones
49    System.Diagnostics.Contracts.Contract.Requires(n >= 0);
50    Contract.Memory.Rsd<CompleteTest>(Contract.Memory.Return, 1, n == 0);
51    Contract.Memory.Rsd<CompleteTest>(Contract.Memory.Return, n * n * n +
          2 * n * n + 2 * n, n >= 1);
52
53    CompleteTest[] r = new CompleteTest[1];
54
55    Contract.Memory.AddRsd(Contract.Memory.Return, Contract.Memory.Return
          );
56    r = CuadraticRsd(n, n); //cant_transf_1 --> n * n
57
58    Contract.Memory.DestRsd(Contract.Memory.Return);
59    r[0] = new CompleteTest(); //cant_directa --> 1
60
61    for (int i = 1; i <= n; i++)
62    {
63      Contract.Memory.IterationSpace(1 <= i && i <= n);
64
65      Contract.Memory.AddRsd(Contract.Memory.Return, Contract.Memory.
            Return);
66      r = CuadraticRsd(i, i); //cant_transf_loop_1 --> n / 6 + (n * n) /
            2 + (n * n * n) / 3
67
68      Contract.Memory.DestRsd(Contract.Memory.Return);
69      r[0] = new CompleteTest(); //cant_directa_loop_1 --> n
70    }
71
72    return r;
73  }
```

Ejemplo 36: Métodos con contratos con polinomios

Ambos métodos son similares, el primero tiene contratos de memoria temporal, mientras que el segundo de memoria residual; por lo tanto detallaremos sólo el primero (`CompleteTmp`). Dado que ambos tienen sus loops con un espacio de iteración anotado, se utilizan en ambos las técnicas de la Sección 6 (Incrementando las capacidades aritméticas de la verificación) para hacer la verificación. En los comentarios se pueden ver los nombres de las expresiones y las cantidades recolectadas en cada caso para hacer la verificación global descripta en la Sección 6 (Incrementando las capacidades aritméticas de la verificación).

Detallaremos ahora los tres conjuntos de contratos que tiene el método `CompleteTmp`, los primeros dos se encuentran comentados. El primer contrato es una cota exacta del consumo, es decir, si se suman los polinomios consumidos en cada instancia donde se consume memoria temporal se obtiene exactamente el polinomio dado. El segundo es una cota superior y es más parecido a lo que un usuario daría como contrato, dado que en general se tiene una idea del orden del consumo, pero no de la cantidad exacta. Ambos contratos son verificables por Barvinok sólo para $n \geq 1$ (a pesar de que también son

correctos para $n = 0$ pero las heurísticas que utiliza no incluyen al caso de $n = 0$), para verificarlo con $n = 0$ podemos utilizar el parámetro de condición en la especificación del contrato como en el tercer contrato. Allí se utiliza la misma cota que en el segundo caso para $n \geq 1$ y el valor 1 para $n = 0$ (si en el polinomio del primer contrato reemplazamos $n$ por 0 obtenemos 1), de esta forma se verifica el consumo de memoria para todo $n \geq 0$.

A continuación mostramos el código del método `CompleteTmp` luego de la instrumentación:

```
1  public void CompleteTmp(int n)
2  {
3    Contract.Requires(n >= 0);
4    Contract.Ensures((n >= 1) ? (CompleteTmp_tmp_CompleteTest <= ((((n *
       n) * n) + ((4 * n) * n)) + (2 * n))) : true);
5    Contract.Ensures((n == 0) ? (CompleteTmp_tmp_CompleteTest <= 1) :
       true);
6    int tmp_call_2 = 0;
7    CompleteTmp_tmp_CompleteTest = 0;
8    Contract.Memory.Tmp<CompleteTest>(1, n == 0);
9    Contract.Memory.Tmp<CompleteTest>((((n * n) * n) + ((4 * n) * n)) +
       (2 * n), n >= 1);
10   Contract.Memory.DestTmp();
11   CompleteTmp_tmp_CompleteTest++;
12   new CompleteTest();
13   this.CuadraticTmpMaxLoop(n);
14   int num = CompleteTest.CuadraticTmpMaxLoop_tmp_CompleteTest;
15   Contract.Memory.AddTmp(Contract.Memory.Return);
16   this.CuadraticRsd(n, n);
17   CompleteTmp_tmp_CompleteTest += CompleteTest.rsd_return_CompleteTest;
18   for (int i = 1; i <= n; i++)
19   {
20     Contract.Memory.IterationSpace((1 <= i) && (i <= n));
21     Contract.Memory.DestTmp();
22     new CompleteTest();
23     this.CuadraticTmpMaxLoop(i);
24     Contract.Memory.AddTmp(Contract.Memory.Return);
25     this.CuadraticRsd(i, i);
26   }
27   if (n >= 1)
28   {
29     CompleteTmp_tmp_CompleteTest += (n / 6) + (((n * n) / 2) + (((n * n
         ) * n) / 3));
30   }
31   else
32   {
33     Contract.Assert(false);
34   }
35   if (n >= 1)
36   {
37     CompleteTmp_tmp_CompleteTest += n;
38   }
39   else
40   {
41     Contract.Assert(false);
42   }
43   if (n >= 1)
```

```
44    {
45      tmp_call_2 = n * n;
46    }
47    else
48    {
49      Contract.Assert(false);
50    }
51    if (n >= 1)
52    {
53      CompleteTmp_tmp_CompleteTest += n * n;
54    }
55    else
56    {
57      Contract.Assert(false);
58    }
59    if (n >= 1)
60    {
61      Contract.Assume(CompleteTmp_tmp_CompleteTest <= ((((n * n) * n) +
          ((4 * n) * n)) + (2 * n)));
62    }
63    else
64    {
65      Contract.Assert(false);
66    }
67  }
```

Ejemplo 37: Código del método `CompleteTmp` instrumentado

Como vemos, para todos los casos se pudo calcular un polinomio utilizando la herramienta aritmética, haciendo posible la inserción de los `Assume` necesarios para la verificación de los contratos.

## 8.3. Uso de condiciones en los contratos

En el siguiente ejemplo mostramos el uso de condiciones en los contratos y algunas limitaciones que el mismo tiene:

**Nota:** en estos ejemplos omitiremos los contratos del tipo `ConditionsTests[]` dado que sólo nos interesa observar el uso de condiciones para un contrato.

```
1   class ConditionsTests
2   {
3     public ConditionsTests[] TestCondSimple(bool b)
4     {
5       Contract.Memory.Rsd<ConditionsTests>(Contract.Memory.Return, 1, b);
6       Contract.Memory.Rsd<ConditionsTests>(Contract.Memory.Return, 2, !b)
            ;
7
8       if (b)
9       {
10        Contract.Memory.DestRsd(Contract.Memory.Return);
11        ConditionsTests c = new ConditionsTests();
```

```
12
13        return new ConditionsTests[] { c };
14      }
15      else
16      {
17        Contract.Memory.DestRsd(Contract.Memory.Return);
18        ConditionsTests c1 = new ConditionsTests();
19
20        Contract.Memory.DestRsd(Contract.Memory.Return);
21        ConditionsTests c2 = new ConditionsTests();
22
23        return new ConditionsTests[] { c1, c2 };
24      }
25    }
26
27    public void TestCondLoop(int n, bool b)
28    {
29      System.Diagnostics.Contracts.Contract.Requires(n >= 1);
30
31      //no verificables
32      Contract.Memory.Tmp<ConditionsTests>(n, b);
33      Contract.Memory.Tmp<ConditionsTests>(2 * n, !b);
34
35      for (int i = 1; i <= n; i++)
36      {
37        Contract.Memory.AddTmp(Contract.Memory.Return);
38        TestCondSimple(b);
39      }
40    }
41  }
```

Ejemplo 38: Ejemplo con condiciones en los contratos

El método `TestCondSimple` tiene un consumo dependiente de un parámetro booleano, sus contratos se verifican sin problemas porque el verificador de Code Contracts es capaz de determinar el flujo del control según el valor del parámetro y asociarlo a los contratos. Sin embargo, los contratos del método `TestCondLoop` no pueden ser verificados a pesar de ser correctos. Veamos la instrumentación hecha para comprender la razón:

```
1   public void TestCondLoop(int n, bool b)
2   {
3     Contract.Requires(n >= 1);
4     Contract.Ensures(!b ? (TestCondLoop_tmp_ConditionsTests <= (2 * n)) :
          true);
5     Contract.Ensures(b ? (TestCondLoop_tmp_ConditionsTests <= n) : true);
6     TestCondLoop_tmp_ConditionsTests = 0;
7     Contract.Memory.Tmp<ConditionsTests>(n, b);
8     Contract.Memory.Tmp<ConditionsTests>(2 * n, !b);
9     for (int i = 1; i <= n; i++)
10    {
11      Contract.Memory.AddTmp(Contract.Memory.Return);
12      this.TestCondSimple(b);
13      TestCondLoop_tmp_ConditionsTests +=
            ConditionsTests_TestCondSimple_rsd_return_ConditionsTests;
14    }
15  }
```

Ejemplo 39: Instrumentación del método `TestCondSimple`

Code Contracts no es capaz de verificar la correctitud de los contratos utilizando la suma de la línea 13, en combinación con el loop y el contrato con condiciones en el método invocado. Eliminado el loop y cambiando el contrato de forma acorde, la verificación se resuelve correctamente, y lo mismo sucede si el contrato del método invocado no tiene condiciones (y por ende tampoco las tendrá el método que estamos analizando). Creemos que el problema radica en las heurísticas que utiliza el verificador para analizar ciclos, que combinadas con el análisis aritmético hecho por el verificador (bastante limitado), no es capaz de determinar el valor total acumulado en la variable `TestCondLoop_tmp_ConditionsTests`.

En este caso, la limitación radica en el verificador, el mismo podría ser mejorado en una versión futura y podría ser capaz de verificar este contrato.

Anotar un espacio de iteración en el ciclo tampoco ayudaría, ya que como mencionamos en la sección correspondiente, no utilizamos Barvinok cuando el contrato del método invocado tiene condiciones, por lo que la instrumentación hecha sería la misma que sin un espacio de iteración anotado.

Una posible solución para este problema es usar siempre una cota superior de ambos contratos en lugar de usar una condición, en este caso se podría usar `2 * n` para ambos casos (dando un único contrato); muchas veces esto es conveniente dado que generalmente al momento de invocar un método será muy difícil para el verificador determinar la condición que se cumple de entre las dadas para los contratos, por lo que tampoco se podría determinar cuál utilizar.

## 8.4. Uso de *trust* en anotaciones

Aquí veremos un ejemplo donde la verificación de correctitud de algunas anotaciones falla y es necesario el uso del parámetro *trust*.

```
1  class PointsFail
2  {
3      public PointsFail attr;
4
5      public PointsFail RsdPorRet()
6      {
7          Contract.Memory.Rsd<PointsFail>(Contract.Memory.Return, 1);
8
9          Contract.Memory.DestRsd(Contract.Memory.Return);
10         return new PointsFail();
11     }
12
13     public PointsFail NecesitaTrust()
14     {
```

```
15          Contract.Memory.Tmp<PointsFail>(3);
16          Contract.Memory.Rsd<PointsFail>(Contract.Memory.Return, 1);
17
18          Contract.Memory.DestTmp();
19          PointsFail p1 = new PointsFail();
20
21          Contract.Memory.AddRsd(Contract.Memory.Return, Contract.Memory.
                Return);
22          p1.attr = RsdPorRet();
23
24          Contract.Memory.DestTmp();
25          PointsFail p2 = new PointsFail();
26
27          Contract.Memory.AddTmp(Contract.Memory.Return, true); //se
                necesita trust
28          p2.attr = RsdPorRet();
29
30          return p1.attr;
31      }
32  }
```

Ejemplo 40: Ejemplo donde *trust* es necesario

El método `RsdPorRet` tiene un residual de 1 de tipo `PointsFail` por `return`, luego el método `NecesitaTrust` lo invoca dos veces (líneas 22 y 28).

El método `NecesitaTrust` crea un objeto `p1` y luego al atributo `attr` de dicho objeto le asigna el objeto creado por `RsdPorRet`, luego realiza lo mismo con `p2`. Dado que devuelve `p1.attr` y este objeto no referencia a ningún otro, ése es el único objeto que escapa del método y va al residual, el resto (`p1`, `p2` y `p2.attr`) van al temporal. Los contratos indican esto y son correctos.

Sin embargo, al verificar esta clase, el verificador alertará que la anotación de la línea 27 es incorrecta (si no está el *trust*), diciendo que el objeto asociado (`p2.attr`) escapa del método y debería ir al residual en lugar de al temporal. Esto es incorrecto.

El problema se origina en el análisis de points-to y escape utilizado; dicho análisis no es sensitivo a contexto y por lo tanto crea un único nodo en el Points-to Graph para el objeto creado en el método `RsdPorRet` sin tener en cuenta las diferentes invocaciones, por lo tanto `p1.attr` y `p2.attr` quedan asociados a un mismo objeto (a pesar de no estarlo), lo que implica que ambos escapan del método porque el primero es devuelto.

Para estos casos introducimos el parámetro *trust*, que permite saltear la verificación, indicando al verificador que confíe en que la anotación es correcta. La anotación también es útil para casos similares en los que el análisis de points-to y escape hace aproximaciones o asunciones que generan que pierda precisión. En un futuro pensamos mejorar este análisis o reemplazarlo por otro más preciso, sin embargo, el utilizado actualmente arroja resultados correctos para la mayoría de los casos más comunes de uso, y falla en casos particulares como el explicado.

Dado que el análisis de points-to y escape no es un punto indispensable para la verificación de los contratos (sólo se utiliza para la verificación de anotaciones auxiliares y no contratos), la imprecisión del mismo no afecta directamente a las capacidades de verificación de los contratos y si el usuario que especifica los contratos es realmente consciente del consumo de memoria del método que está anotando podrá utilizar el parámetro *trust* para los casos en que falle la verificación de las anotaciones adicionales.

## 8.5. Recursión

Una de las limitaciones de algunas técnicas de inferencia de consumo de memoria es el análisis de métodos recursivos. [Gar07] no es capaz de analizar métodos recursivos, mientras que [Rou09] sólo permite analizarlos si el programador especifica un resumen del método.

Nuestra herramienta permite especificar contratos para métodos recursivos sin ningún problema dado que los contratos son modulares y los contratos de los métodos invocados son utilizados durante la verificación sin importar si se invoca el mismo método que se está definiendo.

Las mismas limitaciones que existen con métodos comunes aplican a métodos recursivos, dado que para los mismos se utiliza el mismo método de instrumentación y verificación.

A continuación mostramos un ejemplo de un método recursivo con un contrato:

```
1  class Recursive
2  {
3      public void Rec(int n)
4      {
5          System.Diagnostics.Contracts.Contract.Requires(n >= 1);
6          Contract.Memory.Tmp<Recursive>(n);
7
8          Contract.Memory.DestTmp();
9          new Recursive();
10
11         if (n > 1)
12         {
13             Rec(n - 1);
14         }
15     }
16 }
```

Ejemplo 41: Clase con método recursivo y contrato

Para una invocación al método `Rec(m)` la llamada recursiva en la línea 13 requiere una memoria temporal de tipo `Recursive` de `m - 1`, esto sumado al temporal del *new*

de la línea 9 da un total de `m`. Para el caso base donde `m = 1`, no se entra al if de la línea 11 y el temporal es exactamente 1 aportado por el *new* de la línea 9 dando un total de 1, que también hace válido el contrato.

Para el análisis de métodos recursivos no podríamos utilizar las técnicas de la Sección 6 (Incrementando las capacidades aritméticas de la verificación) dado que no permitimos especificar espacios de iteración recursivos, en un futuro podríamos incorporarlos.

Notar que al utilizar el mismo contador en la instrumentación para todas las llamadas recursivas, podría surgir un problema en que el valor del contador se pise en otra invocación. Esto se puede solucionar fácilmente guardando el valor del contador en una variable local antes de cada llamada recursiva y asignando ese valor nuevamente al contador luego de la misma.

## 8.6. Limitaciones

A continuación resumimos un conjunto de limitaciones asociadas a la implementación del prototipo desarrollado y las herramientas y análisis externos utilizados.

- La implementación del análisis de points-to y escape utilizada para verificar las anotaciones de tiempo de vida no tiene sensibilidad a contexto durante el análisis generando que ciertas anotaciones sean indicadas por el verificador como incorrectas a pesar de no serlo. En estos casos se puede utilizar el parámetro *trust* para evitar la verificación.
- La información de los condicionales de los contratos no es transferida entre métodos en algunos casos por el verificador, imposibilitando la verificación (como en el ejemplo de la Sección 8.3 (Uso de condiciones en los contratos)).
- No se soportan métodos sobrecargados (métodos en una misma clase con igual nombre pero diferentes tipos y/o cantidades de parámetros). En caso de existir dos métodos con igual nombre en la misma clase, se considerarán como el mismo método.
- No se verifica la correctitud de los espacios de iteración dados por el usuario; en un futuro pensamos inferirlos automáticamente por lo esta verificación ya no será necesaria.
- Para la correcta verificación se necesita de un conjunto archivos generados por el compilador que sólo están disponibles cuando el código se compila en modo *Debug*, por lo que la verificación debe ser hecha compilando en dicho modo. En caso de agregar referencias a `dlls` externas con contratos de memoria, se debe disponer además de la `dll` del archivo `pdb` generado cuando el mismo fue compilado en modo *Debug* para poder leer los contratos de memoria del mismo.
- Por el momento no se dispone de una forma de anotar contratos en bibliotecas de código cerrado y distribuirlas por separado, aunque utilizando una funcionalidad

de Code Contracts destinada para tal fin podríamos adaptar la herramienta para que lo soporte. Por el momento se puede lograr el mismo efecto escribiendo clases *wrapper* de las bibliotecas cerradas con los contratos de memoria anotados en las mismas.

Por otro lado, existen las siguientes limitaciones que surgen en las técnicas de instrumentación y verificación propuestas:

- La instrumentación y verificación utilizando los algoritmos de la Sección 6 (Incrementando las capacidades aritméticas de la verificación) no se hace en caso de que los contratos de los métodos invocados estén particionados con condiciones, creando una imposibilidad de verificar algunos contratos del método llamador.
- El análisis de points-to y escape es impreciso debido a sobre-aproximaciones y generalizaciones hechas en el análisis. No es posible utilizar un análisis de points-to correcto para todos los casos dado que todo análisis de points-to es indecidible por naturaleza. Además, todo análisis de points-to correcto es aproximado. Por lo tanto, siempre existirá la posibilidad de que anotaciones correctas no sean probadas. El parámetro *trust* en las anotaciones de tiempo de vida permite evitar la verificación para estos casos.

# 9. Conclusiones

Presentamos en este trabajo un conjunto de algoritmos y técnicas que permiten implementar un verificador de contratos de consumo de memoria apoyándose en un verificador estático de contratos. Además presentamos una implementación de prueba de concepto de las técnicas especificadas para la plataforma .NET.

A pesar de que aún existen ciertas limitaciones en el prototipo implementado, creemos que el mismo puede ser utilizado en un entorno real para obtener un certificado que verifique el consumo de memoria de un programa completo.

Una desventaja de la solución presentada es que se necesita de una comprensión de la forma en que se deben utilizar las anotaciones y de su semántica para ser capaz de escribir contratos correctos, sin embargo esta barrera también existe al anotar contratos de comportamiento, y conocer estos tipos de contratos ayudará a entender la forma de utilizar estos nuevos tipos de contratos.

Podríamos considerar a la necesidad de la asistencia completa del programador para lograr una verificación correcta una desventaja, sin embargo creemos que esta característica le da a la herramienta una ventaja interesante con respecto a otras técnicas con el mismo objetivo que abordan la problemática haciendo inferencia, la misma consiste en la capacidad de lograr una mejor precisión y permitir al programador anotar contratos posiblemente no inferibles pero quizá sí verificables. En un futuro pensamos integrar en la herramienta algún plug-in para la IDE que asista al programador durante la escritura de contratos de consumo de memoria, haciendo más fácil la escritura de los mismos.

Una ventaja importante que encontramos en la herramienta es la completa integración con la IDE de desarrollo estándar para la plataforma .NET (Microsoft Visual Studio). Esta integración se da en dos aspectos: en la capacidad de escribir contratos contando con asistencia directa de la IDE (*IntelliSense* y verificación de sintaxis y tipos) y en la posibilidad de ver los resultados de la verificación en la IDE misma luego de compilar, junto a los resultados de la verificación de los contratos de Code Contracts.

Otra característica favorable de la solución propuesta es la modularidad de los contratos anotados. La especificación modular permite realizar una verificación manejable a grandes escalas y al mismo tiempo precisa, en nuestro caso, gracias a la información de tiempo de vida de los objetos que el usuario provee. Esta verificación provee una garantía sobre el consumo de memoria del programa o biblioteca de código, brindando así un contrato modular del mismo para posibles usuarios.

En conclusión, diseñamos e implementamos una herramienta capaz de ser utilizada en un contexto real certificando la correctitud de contratos de consumo de memoria, herramienta que a pesar de necesitar aún de ciertas mejoras y extensiones para lograr una mejor precisión, puede ser utilizada para obtener resultados aproximados aún útiles bajo ciertas circunstancias. Creemos que la que herramienta construida puede utilizarse como una base sólida para el desarrollo de una futura herramienta con mayores capacidades de verificación.

# 10. Trabajo futuro

Existe un conjunto de características que por cuestiones de complejidad o limitaciones técnicas exceden el alcance del trabajo presentado, sin embargo pueden ser abordadas en futuros trabajos extendiendo las capacidades y precisión de la herramienta desarrollada.

Una de las líneas de trabajo a abordar está relacionada con el análisis de points-to y escape utilizado. El mismo es impreciso debido a sobre-aproximaciones hechas y además no tiene sensibilidad a contexto durante el análisis interprocedural, estas deficiencias no permiten a la herramienta probar la correctitud de algunas anotaciones de tiempo de vida. Como trabajo futuro, este problema puede ser solucionado mejorando el análisis utilizado para solventar los inconvenientes descriptos.

El verificador estático utilizado por Spec#, Z3 [dMB09], tiene capacidades de verificación en muchos casos superior a Clousot. Creemos que la herramienta desarrollada contaría con capacidades de verificación superiores en caso de utilizar este verificador. El mismo puede ser utilizado para código de la plataforma .NET, sin embargo requiere la traducción del código a un lenguaje intermedio de verificación utilizado por Z3 llamado Boogie [BCD+06]. Creemos que esta es un área de trabajo futuro interesante para mejorar los resultados de verificación de la herramienta.

Otra de las principales áreas para continuar trabajando está relacionada a la utilización de la herramienta Barvinok. La instrumentación y verificación utilizando la misma descripta en la Sección 6 (Incrementando las capacidades aritméticas de la verificación) tiene ciertos casos donde la instrumentación o verificación no es realizada debido a limitaciones de la herramienta o por cuestiones algorítmicas, generando así la imposibilidad de verificar la correctitud de ciertos contratos. Existen dos puntos en que vemos posibles mejoras futuras en este aspecto: el primero está relacionado al cálculo del máximo entre polinomios, en algunos casos Barvinok no es capaz de determinar el máximo entre dos ó más polinomios (por ejemplo cuando los mismos se intersecan en algún punto del dominio) dado que las heurísticas que utiliza no se lo permiten detectar, para corregir esto podríamos apoyarnos en otra herramienta externa para partir el dominio y hacer consultas a Barvinok por partes, o modificar Barvinok para que soporte este tipo de consultas; el segundo punto está relacionado a los contratos de consumo bajo condiciones, que directamente son ignorados por la herramienta para la verificación con Barvinok debido a que generaría una explosión combinatoria de subcasos a verificar, para este punto creemos que se necesita abordar estos casos con un enfoque diferente que permita asociar las expresiones calculadas a las diferentes condiciones en los contratos de los métodos invocados para poder verificar los contratos del método bajo análisis sin perder precisión.

Otra línea de trabajo futuro consiste en integrar a la herramienta una capacidad de inferencia de anotaciones y contratos para asistir y completar los contratos dados por el usuario. De esta forma lograríamos una herramienta capaz de verificar la correctitud de contratos complejos que sólo el usuario es capaz de determinar (no inferibles) y al mismo tiempo evitar al usuario escribir contratos simples fácilmente inferibles. Las anotaciones de soporte `DestTmp`, `DestRsd`, `AddTmp` y `AddRsd` pueden ser inferidas utili-

zando un análisis de points-to y escape similar al utilizado actualmente y la anotación `IterationSpace` puede ser inferida utilizando herramientas de análisis estático como la descripta en [EPG+07]. También es posible extraer los invariantes de ciclo calculados por el verificador Clousot para incorporarlos en la verificación realizada utilizando la herramienta aritmética.

Con respecto a la integración de capacidades de inferencia en la herramienta, una posible forma de abordarlo consiste en basarse en el trabajo [GK10], que propone un algoritmo de inferencia de resúmenes de consumo de memoria que logra mayor precisión que el de [Rou09]. Utilizando este algoritmo podríamos generar un conjunto de contratos iniciales que luego el usuario podría enriquecer antes de la verificación.

Una variable en los lenguajes orientados a objetos no tenida en cuenta en el trabajo presentado es el polimorfismo. Es posible integrar a la herramienta análisis heurísticos que determinen el tipo *run-time* de cada objeto creado para poder realizar una verificación más precisa; y además sería posible modificar los algoritmos presentados para tener en cuenta los contratos de las superclases de una clase que no tiene un contrato definido.

El último punto que consideraremos para trabajo futuro está relacionado a la usabilidad de la herramienta. Para permitir a los usuarios anotar contratos de una forma más simple y cómoda se podría desarrollar un plug-in para la IDE que genere automáticamente cierto código necesario para la anotación de los contratos, como ser los campos de tipo `RsdType` utilizados para declarar tipos de residuales y que también asista en la escritura de otras anotaciones, por ejemplo, agregando automáticamente las anotaciones `AddTmp` y `AddRsd` luego de cada invocación evitando así al programador recordar los tipos de contratos en cada método. El plug-in también podría mostrar los contratos de consumo de memoria en la documentación mostrada por el *IntelliSense* de la IDE, como hace lo hace un plug-in oficial para los contratos de Code Contracts [Mic10].

# 11. Anexo

## 11.1. Archivos adjuntos

Adjunto al trabajo entregamos un conjunto de archivos necesarios para su instalación, uso y comprensión. La estructura de los directorios y su contenido es el siguiente:

- `bin`: contiene los archivos binarios necesarios para la instalación y el uso de la herramienta.
- `src`: contiene el código de fuente de la herramienta desarrollada.
- `doc`: contiene la documentación necesaria para la comprensión y el uso de la herramienta y de otras herramientas asociadas.
- `examples`: contiene todos los ejemplos anotados utilizados en este trabajo.

## 11.2. Instalación

### 11.2.1. Prerequisitos

La herramienta desarrollada funciona sólo en sistemas operativos Microsoft Windows debido a su integración con la plataforma .NET y Microsoft Visual Studio. Los prerequisitos de software para la instalación y uso de la misma son:

- Microsoft Visual Studio versión 2008 o 2010 edición Professional o superior (Code Contracts no se puede utilizar en la versión Express).

  Hemos probado la herramienta con ambas versiones, sin embargo recomendamos la versión 2010 por su mejor integración con Code Contracts y para poder probar todos los ejemplos dados (para probarlos en 2008 se necesitará hacer algunos cambios debido a una relocalización de las clases asociadas a contratos de Code Contracts).

- Code Contracts, recomendamos la versión académica, que se puede obtener gratuitamente para uso no comercial en `http://research.microsoft.com/en-us/downloads/4ed7dd5f-490b-489e-8ca8-109324279968/default.aspx`.

### 11.2.2. Instalación del verificador

Una vez instalados los prerequisitos se debe localizar el directorio de instalación de Code Contracts, el mismo normalmente es `C:\Program Files\Microsoft\Contracts`

en sistemas operativos en inglés y `C:\Program Files (x86)\Microsoft\Contracts` si el sistema operativo es de 64 bits.

Localizado el directorio la instalación simplemente consiste en copiar todo el contenido del directorio `bin` en los archivos adjuntos en el directorio de instalación de Code Contracts, pisando los archivos existentes. Es posible que sean necesarios permisos de administrador para realizar esta tarea.

## 11.3. Uso integrado en Microsoft Visual Studio

Para utilizar la herramienta en Microsoft Visual Studio sólo hace falta dos configuraciones:

1. Se debe agregar en el proyecto una referencia a la `dll` que tiene las clases que permite escribir contratos, la misma se llama `Contracts.dll` y se encuentra en el directorio `bin` dentro del directorio de instalación de Code Contracts.

2. Se debe activar la verificación estática de contratos en el proyecto. Esto se realiza yendo a las propiedades del proyecto, luego a la solapa *Code Contracts* y luego marcando la opción *Perform Static Contract Checking*.

Una vez hecho esto, al compilar el proyecto se invocará automáticamente al verificador de contratos de memoria y en la salida de errores se verá el resultado de la verificación. En caso de una verificación correcta se verá una salida similar a la siguiente:

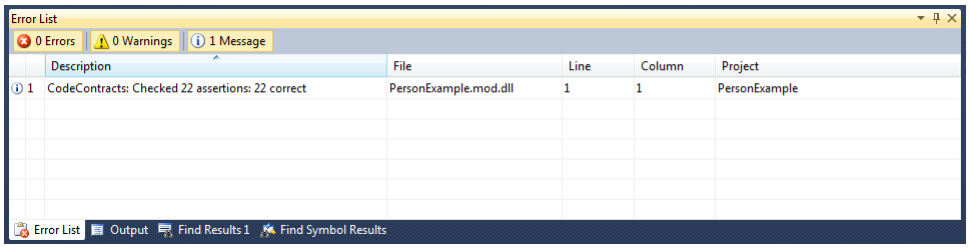


Figura 6: Resultado de verificación correcta

Mientras que si falló la verificación de algún contrato se verá una salida como la siguiente:

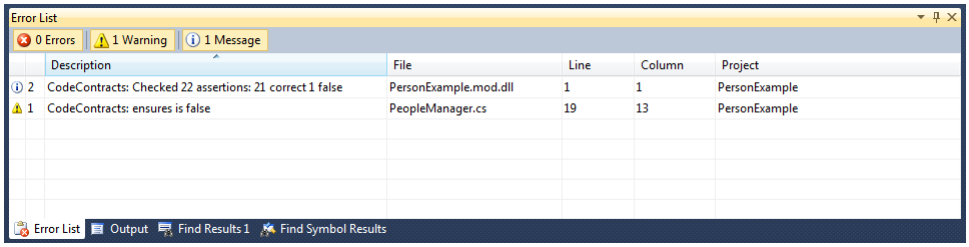


Figura 7: Resultado de verificación con contratos no verificados

## 11.4. Ejemplo completo anotado

En esta sección presentamos un conjunto de clases que están completamente anotadas con contratos de memoria que se pueden verificar correctamente con la herramienta desarrollada. Las cuatro clases forman parte de una misma biblioteca de código cuyo objetivo es modelar personas con sus direcciones físicas.

```
1  public class Address
2  {
3    public string Street { get; private set; }
4    public string City { get; private set; }
5    public string State { get; private set; }
6
7    public Address(string street, string city, string state)
8    {
9      Contract.Memory.Tmp<AddressValidator>(1);
10
11     Contract.Memory.DestTmp();
12     AddressValidator validator = new AddressValidator();
13
14     if (!validator.IsAddressValid(street, city, state))
15     {
16       throw new Exception("Address is invalid.");
17     }
18
19     this.Street = street;
20     this.City = city;
21     this.State = state;
22   }
23 }
```

Ejemplo 42: Clase Address

```
1  public class AddressValidator
2  {
3    public bool IsAddressValid(string street, string city, string state)
4    {
5      return !(String.IsNullOrEmpty(street) ||
6             String.IsNullOrEmpty(city) ||
7             String.IsNullOrEmpty(state));
8    }
9  }
```

Ejemplo 43: Clase AddressValidator

```
public class Person
{
  public string FirtName { get; private set; }
  public string LastName { get; private set; }
  public Address Address { get; private set; }

  public void MoveTo(string street, string city, string state)
  {
    Contract.Memory.Rsd<Address>(Contract.Memory.This, 1);
    Contract.Memory.Tmp<AddressValidator>(1);

    Contract.Memory.DestRsd(Contract.Memory.This);
    this.Address = new Address(street, city, state);
  }

  public Person(string firstName, string lastName, string street,
      string city, string state)
  {
    Contract.Memory.Rsd<Address>(Contract.Memory.This, 1);
    Contract.Memory.Tmp<AddressValidator>(1);

    this.FirtName = firstName;
    this.LastName = lastName;

    Contract.Memory.AddRsd(Contract.Memory.This, Contract.Memory.This);
    this.MoveTo(street, city, state);
  }
}
```

Ejemplo 44: Clase Person

```
public class PeopleManager
{
  public static Person CreatePerson(string firstName, string lastName,
      string street, string city, string state)
  {
    Contract.Memory.Rsd<Person>(Contract.Memory.Return, 1);
    Contract.Memory.Rsd<Address>(Contract.Memory.Return, 1);
    Contract.Memory.Tmp<AddressValidator>(1);

    Contract.Memory.AddRsd(Contract.Memory.Return, Contract.Memory.This
        );
    Contract.Memory.DestRsd(Contract.Memory.Return);
    Person person = new Person(firstName, lastName, street, city, state
        );

    return person;
  }

  public static Contract.Memory.RsdType rsd_MovePersonTo_person;

  public static void MovePersonTo(Person person, string street, string
      city, string state)
  {
```

```
20      Contract.Memory.BindRsd(rsd_MovePersonTo_person, person);
21      Contract.Memory.Rsd<Address>(rsd_MovePersonTo_person, 1);
22      Contract.Memory.Tmp<AddressValidator>(1);
23
24      Contract.Memory.AddRsd(rsd_MovePersonTo_person, Contract.Memory.
            This);
25      person.MoveTo(street, city, state);
26    }
27
28    public static Person[] CreateFamily(List<string> firstNames, string
          lastName, string street, string city, string state)
29    {
30      System.Diagnostics.Contracts.Contract.Requires(firstNames.Count >
            0);
31
32      Contract.Memory.Rsd<Person>(Contract.Memory.Return, firstNames.
            Count);
33      Contract.Memory.Rsd<Person[]>(Contract.Memory.Return, 1);
34      Contract.Memory.Rsd<Address>(Contract.Memory.Return, firstNames.
            Count);
35      Contract.Memory.Tmp<AddressValidator>(1);
36
37      Contract.Memory.DestRsd(Contract.Memory.Return);
38      Person[] family = new Person[firstNames.Count];
39
40      for (int i = 0; i < firstNames.Count; i++)
41      {
42        Contract.Memory.AddRsd(Contract.Memory.Return, Contract.Memory.
              This);
43        Contract.Memory.DestRsd(Contract.Memory.Return);
44        Person p = new Person(firstNames[i], lastName, street, city,
              state);
45        family[i] = p;
46      }
47
48      return family;
49    }
50
51    /// <summary>
52    /// Creates a family with the combination of all first and last names
53    /// </summary>
54    public static Person[] CreateCombinedFamily(List<string> firstNames,
          List<string> lastNames, string street, string city, string state)
55    {
56        System.Diagnostics.Contracts.Contract.Requires(firstNames.Count >
              0);
57        System.Diagnostics.Contracts.Contract.Requires(firstNames.Count
              == lastNames.Count);
58
59        Contract.Memory.Rsd<Person>(Contract.Memory.Return, firstNames.
              Count * lastNames.Count);
60        Contract.Memory.Rsd<Person[]>(Contract.Memory.Return, 1);
61        Contract.Memory.Rsd<Address>(Contract.Memory.Return, firstNames.
              Count * lastNames.Count);
62        Contract.Memory.Tmp<AddressValidator>(1);
63
64        Contract.Memory.DestRsd(Contract.Memory.Return);
```

```
65        Person[] family = new Person[firstNames.Count * lastNames.Count];
66
67        for (int i = 0; i < firstNames.Count; i++)
68        {
69          Contract.Memory.IterationSpace(0 <= i && i < firstNames.Count);
70          for (int j = 0; j < lastNames.Count; j++)
71          {
72            Contract.Memory.IterationSpace(0 <= j && j < lastNames.Count)
                ;
73            Contract.Memory.AddRsd(Contract.Memory.Return, Contract.
                Memory.This);
74            Contract.Memory.DestRsd(Contract.Memory.Return);
75            Person p = new Person(firstNames[i], lastNames[j], street,
                city, state);
76            family[i * firstNames.Count + j] = p;
77          }
78        }
79
80        return family;
81    }
82  }
```

Ejemplo 45: Clase PeopleManager

## 11.5. Información obtenida del análisis de points-to y escape

Del análisis de points-to y escape obtenemos un XML que resume la información de todos los métodos de todas las clases del assembly analizado. El XML devuelto tiene la siguiente estructura:

```
1  <ptg>
2    <method type="class" name="method1">
3      <escape> <!-- nodos que escapan -->
4        <node name="node1" />
5        <node name="node2" />
6      </escape>
7      <expressions>
8        <expr val="expr1">
9          <reaches> <!-- nodos alcanzables por expr1 -->
10           <node name="node1">
11             <expr val="." /> <!-- node1 es alcanzado expr1 misma -->
12           </node>
13         </reaches>
14       </expr>
15       <expr val="expr2">
16         <reaches>
17           <node name="node1">
18             <expr val="f" /> <!-- node1 es alcanzado por expr2.f -->
19           </node>
20           <node name="node2">
21             <expr val="." />
22             <expr val="f" /> <!-- node2 es alcanzado por expr y por
                 expr2.f -->
```

```
23            </node>
24          </reaches>
25        </expr>
26      </expressions>
27    </method>
28    <method type="class" name="method2">
29      ...
30    </method>
31  </ptg>
```

Ejemplo 46: Ejemplo de XML del análisis de points-to y escape

El XML especifica para cada método cuáles son los nodos que escapan del mismo (un nodo representa una instancia en memoria o conjunto de instancias en algunos casos) y además especifica las expresiones que alcanzan a cada nodo. En los comentarios se pueden observar ejemplos del tipo de información que nos brinda.

# 12. Bibliografía

## Referencias


[BCD+06] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *FMCO 2005*, volume 4111, pages 364–387. Springer, September 2006.

[BFGL07] Mike Barnett, Manuel Fähndrich, Diego Garbervetsky, and Francesco Logozzo. Annotations for (more) Precise Points-to Analysis. In *IWACO 2007: ECOOP International Workshop on Aliasing, Confinement and Ownership in object-oriented programming*, jul 2007.

[BLS05] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# Programming System: An Overview (http://research.microsoft.com/en-us/projects/specsharp/). In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *CASSIS 2004*, volume 3362, pages 49–69. Springer, 2005.

[BPS05] Gilles Barthe, Mariela Pavlova, and Gerardo Schneider. Precise analysis of memory consumption using program logics. In Bernhard K. Aichernig and Bernhard Beckert, editors, *SEFM*, pages 86–95. IEEE Computer Society, 2005.

[CEI+07] Ajay Chander, David Espinosa, Nayeem Islam, Peter Lee, and George Necula. Enforcing resource bounds via static verification of dynamic checks. *ACM Trans. Program. Lang. Syst.*, 29(5):28, 2007.

[CFGV09] P. Clauss, F.J. Fernandez, D. Garbervetsky, and S. Verdoolaege. Symbolic polynomial maximization over convex sets and its application to memory requirement estimation. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 17(8):983–996, 2009.

[CNQR05] Wei-Ngan Chin, Huu Hai Nguyen, Shengchao Qin, and Martin Rinard. Memory Usage Verification for OO Programs. In *International Simposium of Static Analysis (SAS'05)*, LNCS, pages 70–86, 2005.

[CT04] Philippe Clauss and Irina Tchoupaeva. A symbolic approach to bernstein expansion for program analysis and optimization. In Evelyn Duesterwald, editor, *13th International Conference on Compiler Construction, CC 2004*, volume 2985 of *LNCS*, pages 120–133. Springer, April 2004.

[dMB09] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver (http://research.microsoft.com/en-us/um/redmond/projects/z3/). 2009.

[EPG+07] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for

dynamic detection of likely invariant. *Science of Computer Programming*, 69(1–3):35–45, December 2007.

[FBL10] Manuel Fähndrich, Michael Barnett, and Francesco Logozzo. Embedded Contract Languages (http://research.microsoft.com/en-us/projects/contracts/). In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 2103–2110, New York, NY, USA, 2010. ACM.

[FL09] Manuel Fähndrich and Francesco Logozzo. Clousot: a language agnostic abstract interpretation-based static analyzer for .NET. http://research.microsoft.com/en-us/people/logozzo/, 2009.

[Gar07] Diego Garbervetsky. *Parametric specification of dymamic memory utilization*. PhD thesis, Departamento de Computación, FCEyN, UBA, November 2007.

[GB00] James Gosling and Greg Bollella. *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[Ghe02] Ovidiu Gheorghioiu. Statically determining memory consumption of real-time java threads. MEng thesis, Massachusetts Institute of Technology, June 2002.

[GK10] Matías Grunberg and Gastón Krasny. Un análisis composicional para la inferencia de resúmenes de consumo de memoria. Tesis de Licenciatura, Departamento de Computación, FCEyN, UBA, 2010.

[Gul09] Sumit Gulwani. Speed: Symbolic complexity bound analysis. In Ahmed Bouajjani and Oded Maler, editors, *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 51–62. Springer, 2009.

[GZ10] Sumit Gulwani and Florian Zuleger. The reachability-bound problem. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 292–304, New York, NY, USA, 2010. ACM.

[LRL+00] Gary T. Leavens, K. Rustan, M. Leino, Erik Poll, Clyde Ruby, Bart Jacobs, Gary T. Leavens, and Clyde Ruby. Jml: notations and tools supporting detailed design in java. In *In OOPSLA 2000 Companion*, pages 105–106. ACM, 2000.

[Mic10] Microsoft. Code Contracts Editor Extensions (http://visualstudiogallery.msdn.microsoft.com/en-us/85f0aa38-a8a8-4811-8b86-e7f0b8d8c71b). 2010.

[Res10a] Microsoft Research. Common Compiler Infrastructure: Code Model and AST API (http://cciast.codeplex.com/). 2010.

[Res10b] Microsoft Research. Common Compiler Infrastructure: Metadata API (http://ccimetadata.codeplex.com/). 2010.

[Rou09] Martín Rouaux. Predicción paramétrica de requerimientos de memoria. Especificación modular. Tesis de Licenciatura, Departamento de Computación, FCEyN, UBA, 2009.

[Sun09] Sun. Sun Java Real-Time System (http://java.sun.com/javase/technologies/realtime/). 2009.

[TT97] Mads Tofte and Jean-Pierre Talpin. Region-Based Memory Management. *Inf. Comput.*, 132:109–176, February 1997.