Especificación y verificación modular de consumo de memoria

Jonathan Tapicer

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

Febrero de 2011

Directores: Diego Garbervetsky, Martín Rouaux

Especificación y verificación modular
de consumo de memoria

Jonathan Tapicer

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

Febrero de 2011

Directores: Diego Garbervetsky, Martín Rouaux

1. Decir el título
2. Contar que vamos a empezar describiendo a grandes rasgos el objetivo del trabajo

Objetivo

- Dados:
  - Un programa escrito en un lenguaje orientado a objetos, y
  - Un conjunto de contratos que especifican su consumo de memoria
- Queremos:
  - Verificar la correctitud de los contratos
- Para esto:
  - Diseñamos un lenguaje de especificación y un conjunto de técnicas que realizan un análisis estático del programa, y
  - Construimos una herramienta que utiliza estas técnicas

Objetivo
- Dados:
  - Un programa escrito en un lenguaje orientado a objetos, y
  - Un conjunto de contratos que especifican su consumo de memoria
- Queremos:
  - Verificar la correctitud de los contratos
- Para esto:
  - Diseñamos un lenguaje de especificación y un conjunto de técnicas que realizan un análisis estático del programa, y
  - Construimos una herramienta que utiliza estas técnicas

1. Tenemos un programa en un lenguaje orientado a objetos
2. El usuario nos dice de alguna forma su consumo de memoria
3. Lo verificamos, hacemos algoritmos y una implementación

Índice

1. Cómo se va a estructurar la presentación
2. Introducción: discutimos alternativas, damos un pantallazo de la solución
3. Anotaciones: algunos conceptos preliminares para introducir las anotaciones, cómo permitimos al usuario darnos los contratos de consumo
4. Verificación: cómo verificamos que los contratos son correctos
5. Limitaciones y trabajo futuro: contamos limitaciones que encontramos, decimos cómo se puede seguir trabajando
6. Conclusiones: resumimos todo lo presentado

Índice

Verificación

- ¿Por qué analizar el consumo de memoria?
  - Sistemas con requerimientos estrictos de memoria: tiempo real, embebidos, misión crítica
  - Entornos donde se cobra el uso de recursos (cloud)
  - Es valioso contar con un *certificado* de la memoria requerida para una ejecución segura
- Inferir vs. Verificar
  - Análisis de consumo de memoria en general: es indecidible
  - Verificar es "más fácil" que inferir, analogía del laberinto
  - El usuario *razona*, tiene un conocimiento profundo del programa
  - La combinación usuario + verificación puede ser mejor que la inferencia
- Verificación de consumo de memoria: área poco explorada

Verificación
- ¿Por qué analizar el consumo de memoria?
  - Sistemas con requerimientos estrictos de memoria: tiempo real, embebidos, misión crítica
  - Entornos donde se cobra el uso de recursos (cloud)
  - Es valioso contar con un *certificado* de la memoria requerida para una ejecución segura
- Inferir vs. Verificar
  - Análisis de consumo de memoria en general: es indecidible
  - Verificar es "más fácil" que inferir, analogía del laberinto
  - El usuario razona, tiene un conocimiento profundo del programa
  - La combinación usuario + verificación puede ser mejor que la inferencia
- Verificación de consumo de memoria: área poco explorada

1. Sistemas donde se cobra: no queremos pagar de más, pero hay que saber cuánto consume
2. Contar diferencias entre inferir y verificar
3. Queremos conocer la memoria necesaria para que el programa se ejecute de forma segura
4. Laberinto: difícil de resolver, fácil de verificar solución
5. Podemos pensar que tenemos un problema (saber el consumo, como el laberinto) y usuario nos da la solución, tenemos que verificarla
6. Existen para inferir: Diego, Martín, extensión de Gastón y Matías
7. Existen para verificar, implementaciones: JML, Spec#, Code Contracts

Verificación

- ¿Por qué analizar el consumo de memoria?
  - Sistemas con requerimientos estrictos de memoria: tiempo real, embebidos, misión crítica
  - Entornos donde se cobra el uso de recursos (cloud)
  - Es valioso contar con un *certificado* de la memoria requerida para una ejecución segura
- Inferir vs. Verificar
  - Análisis de consumo de memoria en general: es indecidible
  - Verificar es "más fácil" que inferir, analogía del laberinto
  - El usuario *razona*, tiene un conocimiento profundo del programa
  - La combinación usuario + verificación puede ser mejor que la inferencia
- Verificación de consumo de memoria: área poco explorada

Verificación
- ¿Por qué analizar el consumo de memoria?
  - Sistemas con requerimientos estrictos de memoria: tiempo real, embebidos, misión crítica
  - Entornos donde se cobra el uso de recursos (cloud)
  - Es valioso contar con un *certificado* de la memoria requerida para una ejecución segura
- Inferir vs. Verificar
  - Análisis de consumo de memoria en general: es indecidible
  - Verificar es "más fácil" que inferir, analogía del laberinto
  - El usuario *razona*, tiene un conocimiento profundo del programa
  - La combinación usuario + verificación puede ser mejor que la inferencia
- Verificación de consumo de memoria: área poco explorada

1. Sistemas donde se cobra: no queremos pagar de más, pero hay que saber cuánto consume
2. Contar diferencias entre inferir y verificar
3. Queremos conocer la memoria necesaria para que el programa se ejecute de forma segura
4. Laberinto: difícil de resolver, fácil de verificar solución
5. Podemos pensar que tenemos un problema (saber el consumo, como el laberinto) y usuario nos da la solución, tenemos que verificarla
6. Existen para inferir: Diego, Martín, extensión de Gastón y Matías
7. Existen para verificar, implementaciones: JML, Spec#, Code Contracts

Verificación

- ¿Por qué analizar el consumo de memoria?
  - Sistemas con requerimientos estrictos de memoria: tiempo real, embebidos, misión crítica
  - Entornos donde se cobra el uso de recursos (cloud)
  - Es valioso contar con un *certificado* de la memoria requerida para una ejecución segura
- Inferir vs. Verificar
  - Análisis de consumo de memoria en general: es indecidible
  - Verificar es "más fácil" que inferir, analogía del laberinto
  - El usuario *razona*, tiene un conocimiento profundo del programa
  - La combinación usuario + verificación puede ser mejor que la inferencia
- Verificación de consumo de memoria: área poco explorada

Verificación
- ¿Por qué analizar el consumo de memoria?
  - Sistemas con requerimientos estrictos de memoria: tiempo real, embebidos, misión crítica
  - Entornos donde se cobra el uso de recursos (cloud)
  - Es valioso contar con un *certificado* de la memoria requerida para una ejecución segura
- Inferir vs. Verificar
  - Análisis de consumo de memoria en general: es indecidible
  - Verificar es "más fácil" que inferir, analogía del laberinto
  - El usuario *razona*, tiene un conocimiento profundo del programa
  - La combinación usuario + verificación puede ser mejor que la inferencia
- Verificación de consumo de memoria: área poco explorada

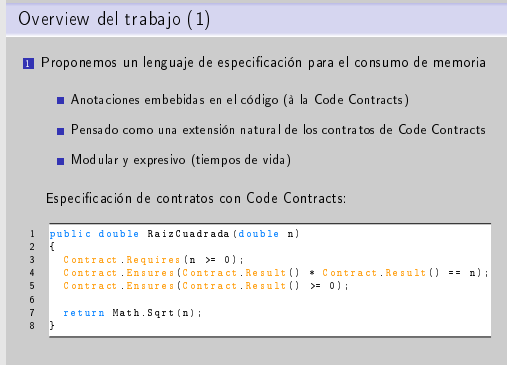
1. Sistemas donde se cobra: no queremos pagar de más, pero hay que saber cuánto consume
2. Contar diferencias entre inferir y verificar
3. Queremos conocer la memoria necesaria para que el programa se ejecute de forma segura
4. Laberinto: difícil de resolver, fácil de verificar solución
5. Podemos pensar que tenemos un problema (saber el consumo, como el laberinto) y usuario nos da la solución, tenemos que verificarla
6. Existen para inferir: Diego, Martín, extensión de Gastón y Matías
7. Existen para verificar, implementaciones: JML, Spec#, Code Contracts

Overview del trabajo (1)

1. Proponemos un lenguaje de especificación para el consumo de memoria
   - Anotaciones embebidas en el código (à la Code Contracts)
   - Pensado como una extensión natural de los contratos de Code Contracts
   - Modular y expresivo (tiempos de vida)

Especificación de contratos con 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  }
```



1. Contamos el desarrollo del trabajo, detallamos cada parte más adelante
2. Usar el estilo de Code Contracts es bueno porque: fácil de escribir (no hace falta aprender nueva sintaxis), integrado a la IDE
3. Lenguaje: modular, a nivel método, de un método sólo se ven sus contratos de consumo, no es necesario conocer su implementación
4. Lenguaje: expresivo, tiene en cuenta que los objetos creados por un método pueden tener diferentes tiempos de vida

Overview del trabajo (2)

2. Proponemos un conjunto de técnicas para verificar la correctitud de las anotaciones
   - Apoyándose en un verificador estático
   - Usando técnicas de análisis de tiempo de vida
   - Teniendo en cuenta la necesidad de verificar aritmética no lineal
3. Construimos un prototipo de la solución propuesta para .NET extendiendo Code Contracts
   - Buena integración con la IDE (Visual Studio)
   - Disponibilidad de herramientas para análisis estático e instrumentación
   - Verificador estático que requiere poca asistencia (uso de Interpretación Abstracta)

Overview del trabajo (2)
2. Proponemos un conjunto de técnicas para verificar la correctitud de las anotaciones
   - Apoyándose en un verificador estático
   - Usando técnicas de análisis de tiempo de vida
   - Teniendo en cuenta la necesidad de verificar aritmética no lineal
3. Construimos un prototipo de la solución propuesta para .NET extendiendo Code Contracts
   - Buena integración con la IDE (Visual Studio)
   - Disponibilidad de herramientas para análisis estático e instrumentación
   - Verificador estático que requiere poca asistencia (uso de Interpretación Abstracta)

1. Herramientas para análisis estático: CCI
2. A nivel CIL (∼bytecode): permite usar la herramienta en cualquier lenguaje de .NET: C#, VB, C++, IronPython, IronRuby
3. Nosotros probamos (y mostramos ejemplos) sólo con C#

Overview del trabajo (2)

2. Proponemos un conjunto de técnicas para verificar la correctitud de las anotaciones
   - Apoyándose en un verificador estático
   - Usando técnicas de análisis de tiempo de vida
   - Teniendo en cuenta la necesidad de verificar aritmética no lineal
3. Construimos un prototipo de la solución propuesta para .NET extendiendo Code Contracts
   - Buena integración con la IDE (Visual Studio)
   - Disponibilidad de herramientas para análisis estático e instrumentación
   - Verificador estático que requiere poca asistencia (uso de Interpretación Abstracta)

Overview del trabajo (2)

- Proponemos un conjunto de técnicas para verificar la correctitud de las anotaciones
  - Apoyándose en un verificador estático
  - Usando técnicas de análisis de tiempo de vida
  - Teniendo en cuenta la necesidad de verificar aritmética no lineal
- Construimos un prototipo de la solución propuesta para .NET extendiendo Code Contracts
  - Buena integración con la IDE (Visual Studio)
  - Disponibilidad de herramientas para análisis estático e instrumentación
  - Verificador estático que requiere poca asistencia (uso de Interpretación Abstracta)

1. Herramientas para análisis estático: CCI
2. A nivel CIL (∼bytecode): permite usar la herramienta en cualquier lenguaje de .NET: C#, VB, C++, IronPython, IronRuby
3. Nosotros probamos (y mostramos ejemplos) sólo con C#

Índice

Modelando el uso de memoria

- Los lenguajes orientados a objetos modernos usan un GC para administrar la memoria, impredecible
- Vamos a usar un modelo del consumo de memoria predecible
  - Sobreaproxima los requerimientos reales, permitiendo obtener una cota del consumo
  - Asumimos que los objetos son recolectados al final del método
  - Sólo tiene en cuenta objetos **creados** por el método analizado y los métodos que este invoca
    - Objetos temporales: no necesarios después de la ejecución del método, se pueden eliminar de memoria
    - Objetos residuales: exceden el tiempo de vida del método, deben seguir vivos luego de que finalice la ejecución del método

1. Necesitamos modelar el uso de memoria de una forma donde la alocación y liberación sea predecible
2. Obtenemos una cota superior: los GC son mejores (más agresivos) que al final del método, por eso sobreaproximamos
3. Partimos al conjunto de objetos creados por un método en dos: tmp y rsd
4. EXPLICAR RÁPIDO Y DETALLAR EN EL EJEMPLO
5. Temporales: objetos usados para un cálculo o proceso auxiliar durante la ejecución
6. Residuales: de diferentes formas (devuelto, parámetros, globales)

Modelando el uso de memoria

- Los lenguajes orientados a objetos modernos usan un GC para administrar la memoria, impredecible
- Vamos a usar un modelo del consumo de memoria predecible
  - Sobreaproxima los requerimientos reales, permitiendo obtener una cota del consumo
  - Asumimos que los objetos son recolectados al final del método
  - Sólo tiene en cuenta objetos **creados** por el método analizado y los métodos que este invoca
    - Objetos temporales: no necesarios después de la ejecución del método, se pueden eliminar de memoria
    - Objetos residuales: exceden el tiempo de vida del método, deben seguir vivos luego de que finalice la ejecución del método

1. Necesitamos modelar el uso de memoria de una forma donde la alocación y liberación sea predecible
2. Obtenemos una cota superior: los GC son mejores (más agresivos) que al final del método, por eso sobreaproximamos
3. Partimos al conjunto de objetos creados por un método en dos: tmp y rsd
4. EXPLICAR RÁPIDO Y DETALLAR EN EL EJEMPLO
5. Temporales: objetos usados para un cálculo o proceso auxiliar durante la ejecución
6. Residuales: de diferentes formas (devuelto, parámetros, globales)

Modelando el uso de memoria

- Los lenguajes orientados a objetos modernos usan un GC para administrar la memoria, impredecible
- Vamos a usar un modelo del consumo de memoria predecible
  - Sobreaproxima los requerimientos reales, permitiendo obtener una cota del consumo
  - Asumimos que los objetos son recolectados al final del método
  - Sólo tiene en cuenta objetos **creados** por el método analizado y los métodos que este invoca
    - Objetos temporales: no necesarios después de la ejecución del método, se pueden eliminar de memoria
    - Objetos residuales: exceden el tiempo de vida del método, deben seguir vivos luego de que finalice la ejecución del método

1. Necesitamos modelar el uso de memoria de una forma donde la alocación y liberación sea predecible
2. Obtenemos una cota superior: los GC son mejores (más agresivos) que al final del método, por eso sobreaproximamos
3. Partimos al conjunto de objetos creados por un método en dos: tmp y rsd
4. EXPLICAR RÁPIDO Y DETALLAR EN EL EJEMPLO
5. Temporales: objetos usados para un cálculo o proceso auxiliar durante la ejecución
6. Residuales: de diferentes formas (devuelto, parámetros, globales)

Modelando el uso de memoria

- Los lenguajes orientados a objetos modernos usan un GC para administrar la memoria, impredecible
- Vamos a usar un modelo del consumo de memoria predecible
  - Sobreaproxima los requerimientos reales, permitiendo obtener una cota del consumo
  - Asumimos que los objetos son recolectados al final del método
  - Sólo tiene en cuenta objetos **creados** por el método analizado y los métodos que este invoca
    - Objetos temporales: no necesarios después de la ejecución del método, se pueden eliminar de memoria
    - Objetos residuales: exceden el tiempo de vida del método, deben seguir vivos luego de que finalice la ejecución del método

Modelando el uso de memoria

- Los lenguajes orientados a objetos modernos usan un GC para administrar la memoria, impredecible
- Vamos a usar un modelo del consumo de memoria predecible
  - Sobreaproxima los requerimientos reales, permitiendo obtener una cota del consumo
  - Asumimos que los objetos son recolectados al final del método
  - Sólo tiene en cuenta objetos **creados** por el método analizado y los métodos que este invoca
    - Objetos temporales: no necesarios después de la ejecución del método, se pueden eliminar de memoria
    - Objetos residuales: exceden el tiempo de vida del método, deben seguir vivos luego de que finalice la ejecución del método

1. Necesitamos modelar el uso de memoria de una forma donde la alocación y liberación sea predecible
2. Obtenemos una cota superior: los GC son mejores (más agresivos) que al final del método, por eso sobreaproximamos
3. Partimos al conjunto de objetos creados por un método en dos: tmp y rsd
4. EXPLICAR RÁPIDO Y DETALLAR EN EL EJEMPLO
5. Temporales: objetos usados para un cálculo o proceso auxiliar durante la ejecución
6. Residuales: de diferentes formas (devuelto, parámetros, globales)

Modelando el uso de memoria

- Los lenguajes orientados a objetos modernos usan un GC para administrar la memoria, impredecible
- Vamos a usar un modelo del consumo de memoria predecible
  - Sobreaproxima los requerimientos reales, permitiendo obtener una cota del consumo
  - Asumimos que los objetos son recolectados al final del método
  - Sólo tiene en cuenta objetos **creados** por el método analizado y los métodos que este invoca
    - Objetos temporales: no necesarios después de la ejecución del método, se pueden eliminar de memoria
    - Objetos residuales: exceden el tiempo de vida del método, deben seguir vivos luego de que finalice la ejecución del método

Modelando el uso de memoria

- Los lenguajes orientados a objetos modernos usan un GC para administrar la memoria, impredecible
- Vamos a usar un modelo del consumo de memoria predecible
  - Sobreaproxima los requerimientos reales, permitiendo obtener una cota del consumo
  - Asumimos que los objetos son recolectados al final del método
  - Sólo tiene en cuenta objetos **creados** por el método analizado y los métodos que este invoca
    - Objetos temporales: no necesarios después de la ejecución del método, se pueden eliminar de memoria
    - Objetos residuales: exceden el tiempo de vida del método, deben seguir vivos luego de que finalice la ejecución del método

1. Necesitamos modelar el uso de memoria de una forma donde la alocación y liberación sea predecible
2. Obtenemos una cota superior: los GC son mejores (más agresivos) que al final del método, por eso sobreaproximamos
3. Partimos al conjunto de objetos creados por un método en dos: tmp y rsd
4. EXPLICAR RÁPIDO Y DETALLAR EN EL EJEMPLO
5. Temporales: objetos usados para un cálculo o proceso auxiliar durante la ejecución
6. Residuales: de diferentes formas (devuelto, parámetros, globales)

Modelando el uso de memoria

- Los lenguajes orientados a objetos modernos usan un GC para administrar la memoria, impredecible
- Vamos a usar un modelo del consumo de memoria predecible
  - Sobreaproxima los requerimientos reales, permitiendo obtener una cota del consumo
  - Asumimos que los objetos son recolectados al final del método
  - Sólo tiene en cuenta objetos **creados** por el método analizado y los métodos que este invoca
    - Objetos temporales: no necesarios después de la ejecución del método, se pueden eliminar de memoria
    - Objetos residuales: exceden el tiempo de vida del método, deben seguir vivos luego de que finalice la ejecución del método

Modelando el uso de memoria

- Los lenguajes orientados a objetos modernos usan un GC para administrar la memoria, impredecible
- Vamos a usar un modelo del consumo de memoria predecible
  - Sobreaproxima los requerimientos reales, permitiendo obtener una cota del consumo
  - Asumimos que los objetos son recolectados al final del método
  - Sólo tiene en cuenta objetos **creados** por el método analizado y los métodos que este invoca
    - Objetos temporales: no necesarios después de la ejecución del método, se pueden eliminar de memoria
    - Objetos residuales: exceden el tiempo de vida del método, deben seguir vivos luego de que finalice la ejecución del método

1. Necesitamos modelar el uso de memoria de una forma donde la alocación y liberación sea predecible
2. Obtenemos una cota superior: los GC son mejores (más agresivos) que al final del método, por eso sobreaproximamos
3. Partimos al conjunto de objetos creados por un método en dos: tmp y rsd
4. EXPLICAR RÁPIDO Y DETALLAR EN EL EJEMPLO
5. Temporales: objetos usados para un cálculo o proceso auxiliar durante la ejecución
6. Residuales: de diferentes formas (devuelto, parámetros, globales)

Modelando el uso de memoria, ejemplo

```
1  class IntLinkedList
2  {
3      private Node Head;
4
5      public void PushFront(Node node)
6      {
7          Logger logger = new Logger();
8          node.Next = this.Head;
9          this.Head = node;
10         logger.Log("PushFront done");
11     }
12
13     public void Fill(int n)
14     {
15         for (int i = 1; i <= n; i++)
16         {
17             Node node = new Node(i);
18             this.PushFront(node);
19         }
20     }
21 }
```

Modelando el uso de memoria, ejemplo

```
1  class IntLinkedList
2  {
3     private Node Head;
4
5     public void PushFront(Node node)
6     {
7        Logger logger = new Logger();
8        node.Next = this.Head;
9        this.Head = node;
10       logger.Log("PushFront done");
11    }
12
13    public void Fill(int n)
14    {
15       for (int i = 1; i <= n; i++)
16       {
17          Node node = new Node(i);
18          this.PushFront(node);
19       }
20    }
21 }
```

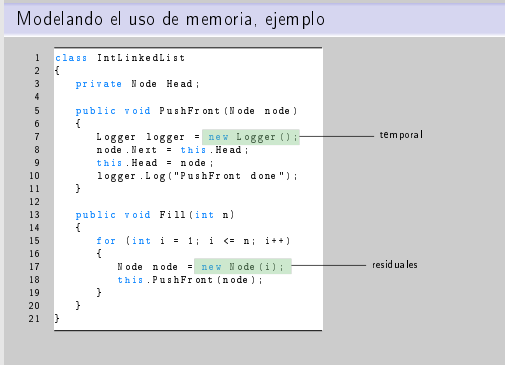
1. Explicar código
2. Decir que lo vamos a usar de ejemplo en el resto de la presentación
3. Explicar por qué son tmp y rsd cada uno

Modelando el uso de memoria, ejemplo

```
1  class IntLinkedList
2  {
3      private Node Head;
4
5      public void PushFront(Node node)
6      {
7          Logger logger = new Logger();
8          node.Next = this.Head;
9          this.Head = node;
10         logger.Log("PushFront done");
11     }
12
13     public void Fill(int n)
14     {
15         for (int i = 1; i <= n; i++)
16         {
17             Node node = new Node(i);
18             this.PushFront(node);
19         }
20     }
21 }
```

temporal

residuales



1. Explicar código
2. Decir que lo vamos a usar de ejemplo en el resto de la presentación
3. Explicar por qué son tmp y rsd cada uno

Anotaciones, ejemplo

```
public void PushFront(Node node)
{


   Logger logger = new Logger();
   node.Next = this.Head;
   this.Head = node;
   logger.Log("PushFront done");
}

public void Fill(int n)
{



   for (int i = 1; i <= n; i++)
   {

      Node node = new Node(i);
      this.PushFront(node);
   }
}
```

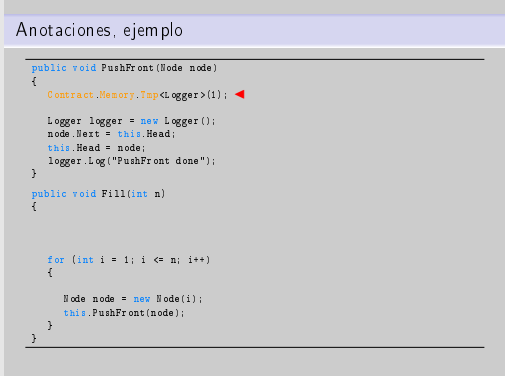
Anotaciones, ejemplo

```
public void PushFront(Node node)
{


   Logger logger = new Logger();
   node.Next = this.Head;
   this.Head = node;
   logger.Log("PushFront done");
}

public void Fill(int n)
{



   for (int i = 1; i <= n; i++)
   {

      Node node = new Node(i);
      this.PushFront(node);
   }
}
```

1. Mostrar de a poco y explicar cada anotación
2. Vean cómo se parecen a las de Code Contracts
3. Explicar que Fill tiene tmp porque para la ejecución del método es necesario el espacio de memoria de un Logger
4. Para que el contrato sea correcto n tiene que ser positivo

Anotaciones, ejemplo

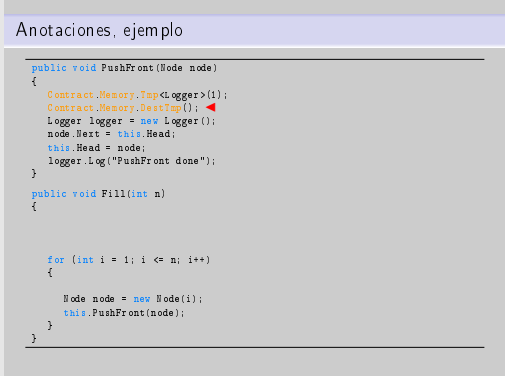
```
public void PushFront(Node node)
{
   Contract.Memory.Tmp<Logger>(1);

   Logger logger = new Logger();
   node.Next = this.Head;
   this.Head = node;
   logger.Log("PushFront done");
}

public void Fill(int n)
{



   for (int i = 1; i <= n; i++)
   {

      Node node = new Node(i);
      this.PushFront(node);
   }
}
```



1. Mostrar de a poco y explicar cada anotación
2. Vean cómo se parecen a las de Code Contracts
3. Explicar que Fill tiene tmp porque para la ejecución del método es necesario el espacio de memoria de un Logger
4. Para que el contrato sea correcto n tiene que ser positivo

Anotaciones, ejemplo

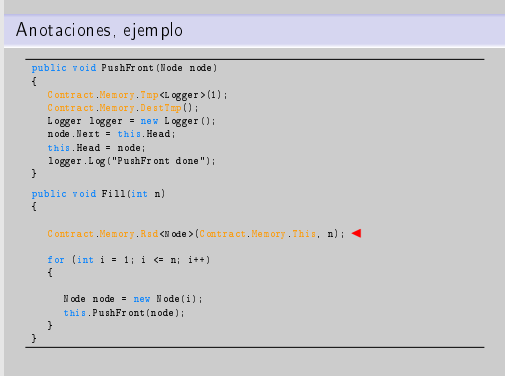
```
public void PushFront(Node node)
{
   Contract.Memory.Tmp<Logger>(1);
   Contract.Memory.DestTmp(); ◀
   Logger logger = new Logger();
   node.Next = this.Head;
   this.Head = node;
   logger.Log("PushFront done");
}

public void Fill(int n)
{



   for (int i = 1; i <= n; i++)
   {

      Node node = new Node(i);
      this.PushFront(node);
   }
}
```



1. Mostrar de a poco y explicar cada anotación
2. Vean cómo se parecen a las de Code Contracts
3. Explicar que Fill tiene tmp porque para la ejecución del método es necesario el espacio de memoria de un Logger
4. Para que el contrato sea correcto n tiene que ser positivo

Anotaciones, ejemplo

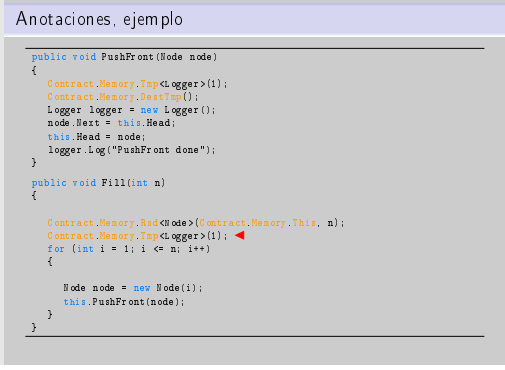
```
public void PushFront(Node node)
{
   Contract.Memory.Tmp<Logger>(1);
   Contract.Memory.DestTmp();
   Logger logger = new Logger();
   node.Next = this.Head;
   this.Head = node;
   logger.Log("PushFront done");
}

public void Fill(int n)
{

   Contract.Memory.Rsd<Node>(Contract.Memory.This, n);

   for (int i = 1; i <= n; i++)
   {

      Node node = new Node(i);
      this.PushFront(node);
   }
}
```



1. Mostrar de a poco y explicar cada anotación
2. Vean cómo se parecen a las de Code Contracts
3. Explicar que Fill tiene tmp porque para la ejecución del método es necesario el espacio de memoria de un Logger
4. Para que el contrato sea correcto n tiene que ser positivo

Anotaciones, ejemplo

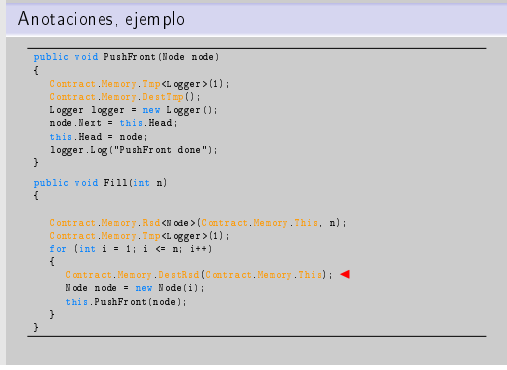
```
public void PushFront(Node node)
{
   Contract.Memory.Tmp<Logger>(1);
   Contract.Memory.DestTmp();
   Logger logger = new Logger();
   node.Next = this.Head;
   this.Head = node;
   logger.Log("PushFront done");
}

public void Fill(int n)
{

   Contract.Memory.Rsd<Node>(Contract.Memory.This, n);
   Contract.Memory.Tmp<Logger>(1);
   for (int i = 1; i <= n; i++)
   {

      Node node = new Node(i);
      this.PushFront(node);
   }
}
```



1. Mostrar de a poco y explicar cada anotación
2. Vean cómo se parecen a las de Code Contracts
3. Explicar que Fill tiene tmp porque para la ejecución del método es necesario el espacio de memoria de un Logger
4. Para que el contrato sea correcto n tiene que ser positivo

Anotaciones, ejemplo

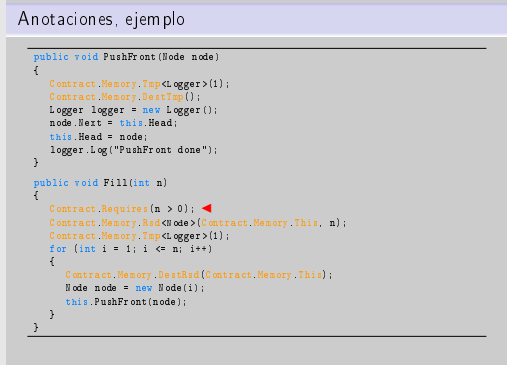
```
public void PushFront(Node node)
{
   Contract.Memory.Tmp<Logger>(1);
   Contract.Memory.DestTmp();
   Logger logger = new Logger();
   node.Next = this.Head;
   this.Head = node;
   logger.Log("PushFront done");
}

public void Fill(int n)
{

   Contract.Memory.Rsd<Node>(Contract.Memory.This, n);
   Contract.Memory.Tmp<Logger>(1);
   for (int i = 1; i <= n; i++)
   {
      Contract.Memory.DestRsd(Contract.Memory.This);
      Node node = new Node(i);
      this.PushFront(node);
   }
}
```



1. Mostrar de a poco y explicar cada anotación
2. Vean cómo se parecen a las de Code Contracts
3. Explicar que Fill tiene tmp porque para la ejecución del método es necesario el espacio de memoria de un Logger
4. Para que el contrato sea correcto n tiene que ser positivo

Anotaciones, ejemplo

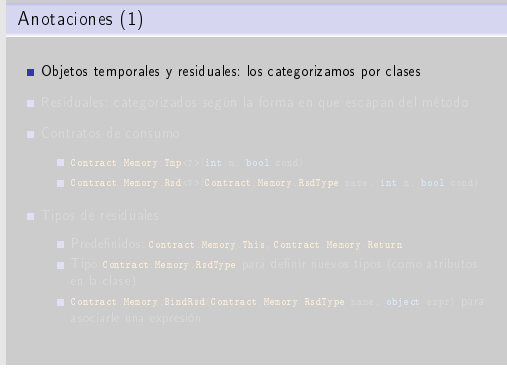
```
public void PushFront(Node node)
{
   Contract.Memory.Tmp<Logger>(1);
   Contract.Memory.DestTmp();
   Logger logger = new Logger();
   node.Next = this.Head;
   this.Head = node;
   logger.Log("PushFront done");
}

public void Fill(int n)
{
   Contract.Requires(n > 0); ◀
   Contract.Memory.Rsd<Node>(Contract.Memory.This, n);
   Contract.Memory.Tmp<Logger>(1);
   for (int i = 1; i <= n; i++)
   {
      Contract.Memory.DestRsd(Contract.Memory.This);
      Node node = new Node(i);
      this.PushFront(node);
   }
}
```



1. Mostrar de a poco y explicar cada anotación
2. Vean cómo se parecen a las de Code Contracts
3. Explicar que Fill tiene tmp porque para la ejecución del método es necesario el espacio de memoria de un Logger
4. Para que el contrato sea correcto n tiene que ser positivo

Anotaciones (1)

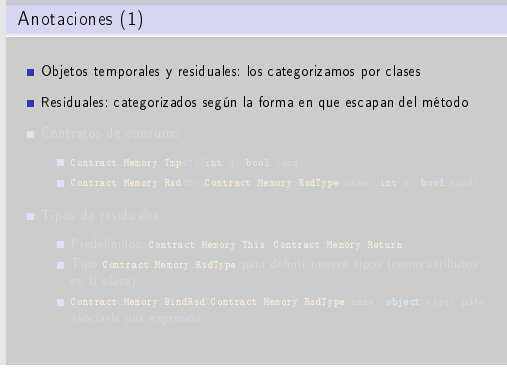
- Objetos temporales y residuales: los categorizamos por clases
- Residuales: categorizados según la forma en que escapan del método
- Contratos de consumo
  - `Contract.Memory.Tmp<T>(int n, bool cond)`
  - `Contract.Memory.Rsd<T>(Contract.Memory.RsdType name, int n, bool cond)`
- Tipos de residuales
  - Predefinidos: `Contract.Memory.This`, `Contract.Memory.Return`
  - Tipo `Contract.Memory.RsdType` para definir nuevos tipos (como atributos en la clase)
  - `Contract.Memory.BindRsd(Contract.Memory.RsdType name, object expr)` para asociarle una expresión



1. Vimos ejemplos, ahora definimos formalmente las anotaciones
2. Rsd: aclarar escapa por, porque los agrupamos por tiempo de vida similar, lo necesitamos para identificar la forma en que se usan los objetos residuales en el método llamador
3. Contratos: para definir el consumo de memoria de un método, discriminado por tipos
4. Tipos de residuales: los que hay, y para definir nuevos tipos

Anotaciones (1)

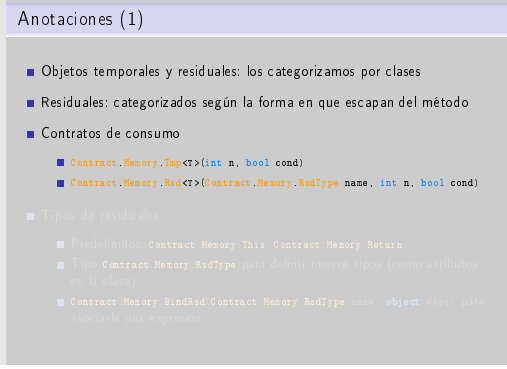
- Objetos temporales y residuales: los categorizamos por clases
- Residuales: categorizados según la forma en que escapan del método
- Contratos de consumo
  - `Contract.Memory.Tmp<T>(int n, bool cond)`
  - `Contract.Memory.Rsd<T>(Contract.Memory.RsdType name, int n, bool cond)`
- Tipos de residuales
  - Predefinidos: `Contract.Memory.This`, `Contract.Memory.Return`
  - Tipo `Contract.Memory.RsdType` para definir nuevos tipos (como atributos en la clase)
  - `Contract.Memory.BindRsd(Contract.Memory.RsdType name, object expr)` para asociarle una expresión



1. Vimos ejemplos, ahora definimos formalmente las anotaciones
2. Rsd: aclarar escapa por, porque los agrupamos por tiempo de vida similar, lo necesitamos para identificar la forma en que se usan los objetos residuales en el método llamador
3. Contratos: para definir el consumo de memoria de un método, discriminado por tipos
4. Tipos de residuales: los que hay, y para definir nuevos tipos

Anotaciones (1)

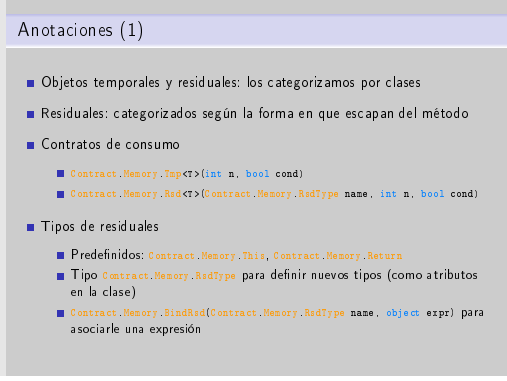
- Objetos temporales y residuales: los categorizamos por clases
- Residuales: categorizados según la forma en que escapan del método
- Contratos de consumo
  - `Contract.Memory.Tmp<T>(int n, bool cond)`
  - `Contract.Memory.Rsd<T>(Contract.Memory.RsdType name, int n, bool cond)`
- Tipos de residuales
  - Predefinidos: `Contract.Memory.This`, `Contract.Memory.Return`
  - Tipo `Contract.Memory.RsdType` para definir nuevos tipos (como atributos en la clase)
  - `Contract.Memory.BindRsd(Contract.Memory.RsdType name, object expr)` para asociarle una expresión



1. Vimos ejemplos, ahora definimos formalmente las anotaciones
2. Rsd: aclarar escapa por, porque los agrupamos por tiempo de vida similar, lo necesitamos para identificar la forma en que se usan los objetos residuales en el método llamador
3. Contratos: para definir el consumo de memoria de un método, discriminado por tipos
4. Tipos de residuales: los que hay, y para definir nuevos tipos

Anotaciones (1)

- Objetos temporales y residuales: los categorizamos por clases
- Residuales: categorizados según la forma en que escapan del método
- Contratos de consumo
  - `Contract.Memory.Tmp<T>(int n, bool cond)`
  - `Contract.Memory.Rsd<T>(Contract.Memory.RsdType name, int n, bool cond)`
- Tipos de residuales
  - Predefinidos: `Contract.Memory.This`, `Contract.Memory.Return`
  - Tipo `Contract.Memory.RsdType` para definir nuevos tipos (como atributos en la clase)
  - `Contract.Memory.BindRsd(Contract.Memory.RsdType name, object expr)` para asociarle una expresión



1. Vimos ejemplos, ahora definimos formalmente las anotaciones
2. Rsd: aclarar escapa por, porque los agrupamos por tiempo de vida similar, lo necesitamos para identificar la forma en que se usan los objetos residuales en el método llamador
3. Contratos: para definir el consumo de memoria de un método, discriminado por tipos
4. Tipos de residuales: los que hay, y para definir nuevos tipos

Anotaciones (2)

- Tiempo de vida
  - `Contract.Memory.DestTmp()` para definir que el objeto pertenece a la memoria temporal del método
  - `Contract.Memory.DestRsd(Contract.Memory.RsdType name)` para definir que el objeto pertenece a la memoria residual de nombre `name` del método
  - `Contract.Memory.AddTmp(Contract.Memory.RsdType name_call)` y `Contract.Memory.AddRsd(Contract.Memory.RsdType name_local, Contract.Memory.RsdType name_call)` para transferencia de residuales en calls
- Espacios de iteración
  - `Contract.Memory.IterationSpace(bool cond)` para definir espacios de iteración en loops

Anotaciones (2)

- Tiempo de vida
  - Contract.Memory.DestTmp() para definir que el objeto pertenece a la memoria temporal del método
  - Contract.Memory.DestRsd(Contract.Memory.RsdType name) para definir que el objeto pertenece a la memoria residual de nombre name del método
  - Contract.Memory.AddTmp(Contract.Memory.RsdType name_call) y Contract.Memory.AddRsd(Contract.Memory.RsdType name_local, Contract.Memory.RsdType name_call) para transferencia de residuales en calls
- Espacios de iteración
  - Contract.Memory.IterationSpace(bool cond) para definir espacios de iteración en loops

1. Tiempo de vida: para anotar cuáles objetos son tmp y cuáles rsd, y qué pasa con los objetos rsd de un método invocado
2. Espacios de iteración: para entender la cantidad de veces que ocurren los consumos de memoria adentro de loops

Anotaciones (2)

- Tiempo de vida
  - `Contract.Memory.DestTmp()` para definir que el objeto pertenece a la memoria temporal del método
  - `Contract.Memory.DestRsd(Contract.Memory.RsdType name)` para definir que el objeto pertenece a la memoria residual de nombre `name` del método
  - `Contract.Memory.AddTmp(Contract.Memory.RsdType name_call)` y `Contract.Memory.AddRsd(Contract.Memory.RsdType name_local, Contract.Memory.RsdType name_call)` para transferencia de residuales en calls
- Espacios de iteración
  - `Contract.Memory.IterationSpace(bool cond)` para definir espacios de iteración en loops

Anotaciones (2)

- Tiempo de vida
  - Contract.Memory.DestTmp() para definir que el objeto pertenece a la memoria temporal del método
  - Contract.Memory.DestRsd(Contract.Memory.RsdType name) para definir que el objeto pertenece a la memoria residual de nombre name del método
  - Contract.Memory.AddTmp(Contract.Memory.RsdType name_call) y Contract.Memory.AddRsd(Contract.Memory.RsdType name_local, Contract.Memory.RsdType name_call) para transferencia de residuales en calls
- Espacios de iteración
  - Contract.Memory.IterationSpace(bool cond) para definir espacios de iteración en loops

1. Tiempo de vida: para anotar cuáles objetos son tmp y cuáles rsd, y qué pasa con los objetos rsd de un método invocado
2. Espacios de iteración: para entender la cantidad de veces que ocurren los consumos de memoria adentro de loops

Índice

Verificación

- ¿Cómo verificamos que las anotaciones son correctas?
- Contratos
  - Correctitud de las anotaciones `Contract.Memory.Tmp` y `Contract.Memory.Rsd`
  - Instrumentación y uso de verificador estático
  - Soporte de herramienta aritmética
- Tiempo de vida
  - Correctitud de las anotaciones `Contract.Memory.DestTmp`, `Contract.Memory.DestRsd`, `Contract.Memory.AddTmp` y `Contract.Memory.AddRsd`
  - Análisis de points-to y escape

Verificación

1. Esta parte es el desarrollo central del trabajo
2. Verificar que el consumo que el usuario dice que tiene su programa sea correcto
3. Hay diferentes cosas a verificar, cada una la resolvemos de diferente forma
4. Instrumentación: generar un programa equivalente pero con código insertado que nos va a permitir que Code Contracts verifique los contratos de memoria
5. Empezamos viendo un ejemplo de la instrumentación básica

Verificación

- ¿Cómo verificamos que las anotaciones son correctas?
- Contratos
  - Correctitud de las anotaciones `Contract.Memory.Tmp` y `Contract.Memory.Rsd`
  - Instrumentación y uso de verificador estático
  - Soporte de herramienta aritmética
- Tiempo de vida
  - Correctitud de las anotaciones `Contract.Memory.DestTmp`, `Contract.Memory.DestRsd`, `Contract.Memory.AddTmp` y `Contract.Memory.AddRsd`
  - Análisis de points-to y escape

Verificación

1. Esta parte es el desarrollo central del trabajo
2. Verificar que el consumo que el usuario dice que tiene su programa sea correcto
3. Hay diferentes cosas a verificar, cada una la resolvemos de diferente forma
4. Instrumentación: generar un programa equivalente pero con código insertado que nos va a permitir que Code Contracts verifique los contratos de memoria
5. Empezamos viendo un ejemplo de la instrumentación básica

Verificación

- ¿Cómo verificamos que las anotaciones son correctas?
- Contratos
  - Correctitud de las anotaciones `Contract.Memory.Tmp` y `Contract.Memory.Rsd`
  - Instrumentación y uso de verificador estático
  - Soporte de herramienta aritmética
- Tiempo de vida
  - Correctitud de las anotaciones `Contract.Memory.DestTmp`, `Contract.Memory.DestRsd`, `Contract.Memory.AddTmp` y `Contract.Memory.AddRsd`
  - Análisis de points-to y escape

Verificación

1. Esta parte es el desarrollo central del trabajo
2. Verificar que el consumo que el usuario dice que tiene su programa sea correcto
3. Hay diferentes cosas a verificar, cada una la resolvemos de diferente forma
4. Instrumentación: generar un programa equivalente pero con código insertado que nos va a permitir que Code Contracts verifique los contratos de memoria
5. Empezamos viendo un ejemplo de la instrumentación básica

Verificación mediante instrumentación, ejemplo 1

```
public void PushFront(Node node)
{
   Contract.Memory.Tmp<Logger>(1);


   Contract.Memory.DestTmp();

   Logger logger = new Logger();
   node.Next = this.Head;
   this.Head = node;
   logger.Log("PushFront done");
}
```

Verificación mediante instrumentación, ejemplo 1

```
public void PushFront(Node node)
{
    Contract.Memory.Tmp<Logger>(1);


    Contract.Memory.DestTmp();

    Logger logger = new Logger();
    node.Next = this.Head;
    this.Head = node;
    logger.Log("PushFront done");
}
```

1. Mostrar de a poco y explicar la instrumentación

Verificación mediante instrumentación, ejemplo 1

```
public static int IntLinkedList_PushFront_Tmp_Logger;

public void PushFront(Node node)
{
   Contract.Memory.Tmp<Logger>(1);


   Contract.Memory.DestTmp();

   Logger logger = new Logger();
   node.Next = this.Head;
   this.Head = node;
   logger.Log("PushFront done");
}
```

Verificación mediante instrumentación, ejemplo 1

```
public static int IntLinkedList_PushFront_Tmp_Logger;

public void PushFront(Node node)
{
    Contract.Memory.Tmp<Logger>(1);


    Contract.Memory.DestTmp();

    Logger logger = new Logger();
    node.Next = this.Head;
    this.Head = node;
    logger.Log("PushFront done");
}
```

1. Mostrar de a poco y explicar la instrumentación

Verificación mediante instrumentación, ejemplo 1

```
public static int IntLinkedList_PushFront_Tmp_Logger;

public void PushFront(Node node)
{
   Contract.Memory.Tmp<Logger>(1);

   IntLinkedList_PushFront_Tmp_Logger = 0;
   Contract.Memory.DestTmp();

   Logger logger = new Logger();
   node.Next = this.Head;
   this.Head = node;
   logger.Log("PushFront done");
}
```

```
public static int IntLinkedList_PushFront_Tmp_Logger;

public void PushFront(Node node)
{
    Contract.Memory.Tmp<Logger>(1);

    IntLinkedList_PushFront_Tmp_Logger = 0;
    Contract.Memory.DestTmp();

    Logger logger = new Logger();
    node.Next = this.Head;
    this.Head = node;
    logger.Log("PushFront done");
}
```

1. Mostrar de a poco y explicar la instrumentación

Verificación mediante instrumentación, ejemplo 1

```
public static int IntLinkedList_PushFront_Tmp_Logger;

public void PushFront(Node node)
{
   Contract.Memory.Tmp<Logger>(1);

   IntLinkedList_PushFront_Tmp_Logger = 0;
   Contract.Memory.DestTmp();
   IntLinkedList_PushFront_Tmp_Logger++;
   Logger logger = new Logger();
   node.Next = this.Head;
   this.Head = node;
   logger.Log("PushFront done");
}
```

```
public static int IntLinkedList_PushFront_Tmp_Logger;

public void PushFront(Node node)
{
    Contract.Memory.Tmp<Logger>(1);

    IntLinkedList_PushFront_Tmp_Logger = 0;
    Contract.Memory.DestTmp();
    IntLinkedList_PushFront_Tmp_Logger++;
    Logger logger = new Logger();
    node.Next = this.Head;
    this.Head = node;
    logger.Log("PushFront done");
}
```

1. Mostrar de a poco y explicar la instrumentación

Verificación mediante instrumentación, ejemplo 1

```
public static int IntLinkedList_PushFront_Tmp_Logger;

public void PushFront(Node node)
{
   Contract.Memory.Tmp<Logger>(1);
   Contract.Ensures(IntLinkedList_PushFront_Tmp_Logger <= 1);
   IntLinkedList_PushFront_Tmp_Logger = 0;
   Contract.Memory.DestTmp();
   IntLinkedList_PushFront_Tmp_Logger++;
   Logger logger = new Logger();
   node.Next = this.Head;
   this.Head = node;
   logger.Log("PushFront done");
}
```

```
public static int IntLinkedList_PushFront_Tmp_Logger;

public void PushFront(Node node)
{
    Contract.Memory.Tmp<Logger>(1);
    Contract.Ensures(IntLinkedList_PushFront_Tmp_Logger <= 1);
    IntLinkedList_PushFront_Tmp_Logger = 0;
    Contract.Memory.DestTmp();
    IntLinkedList_PushFront_Tmp_Logger++;
    Logger logger = new Logger();
    node.Next = this.Head;
    this.Head = node;
    logger.Log("PushFront done");
}
```

1. Mostrar de a poco y explicar la instrumentación

Verificación mediante instrumentación, ejemplo 2

```
public void Fill(int n)
{
   Contract.Requires(n > 0);
   Contract.Memory.Rsd<Node>(Contract.Memory.This, n);
   Contract.Memory.Tmp<Logger>(1);




   for (int i = 1; i <= n; i++)
   {
      Contract.Memory.DestRsd(Contract.Memory.This);

      Node node = new Node(i);
      this.PushFront(node);


   }

}
```

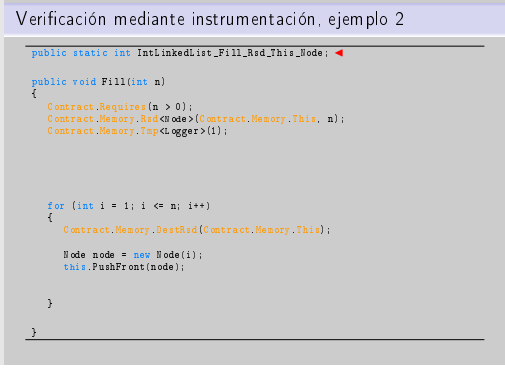
Verificación mediante instrumentación, ejemplo 2

```
public void Fill(int n)
{
   Contract.Requires(n > 0);
   Contract.Memory.Rsd<Node>(Contract.Memory.This, n);
   Contract.Memory.Tmp<Logger>(1);



   for (int i = 1; i <= n; i++)
   {
      Contract.Memory.DestRsd(Contract.Memory.This);

      Node node = new Node(i);
      this.PushFront(node);

   }

}
```

1. Este método es más complejo porque hay un call y un loop
2. Hay que tener en cuenta el tmp del método llamado
3. Si tuviese rsd también hay que tenerlo en cuenta
4. Maximizamos el tmp del callee porque necesitamos conocer el máximo de memoria temporal necesario en cualquier invocación, no la suma porque se libera cuando termina el método
5. Aclarar: anotaciones AddTmp y AddRsd no las mostramos porque no son necesarias, pero la instrumentación es muy similar

Verificación mediante instrumentación, ejemplo 2

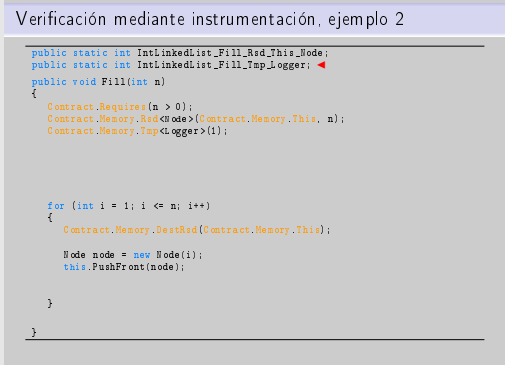
```
public static int IntLinkedList_Fill_Rsd_This_Node; ◀

public void Fill(int n)
{
   Contract.Requires(n > 0);
   Contract.Memory.Rsd<Node>(Contract.Memory.This, n);
   Contract.Memory.Tmp<Logger>(1);




   for (int i = 1; i <= n; i++)
   {
      Contract.Memory.DestRsd(Contract.Memory.This);

      Node node = new Node(i);
      this.PushFront(node);


   }

}
```



1. Este método es más complejo porque hay un call y un loop
2. Hay que tener en cuenta el tmp del método llamado
3. Si tuviese rsd también hay que tenerlo en cuenta
4. Maximizamos el tmp del callee porque necesitamos conocer el máximo de memoria temporal necesario en cualquier invocación, no la suma porque se libera cuando termina el método
5. Aclarar: anotaciones AddTmp y AddRsd no las mostramos porque no son necesarias, pero la instrumentación es muy similar

Verificación mediante instrumentación, ejemplo 2

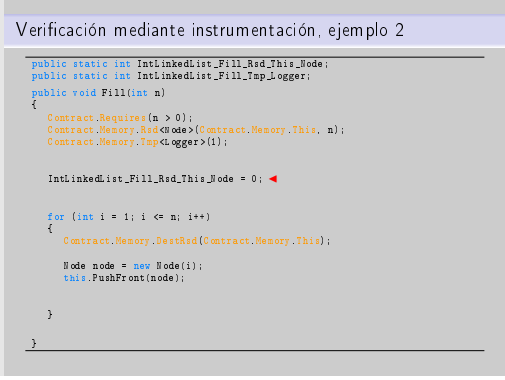
```
public static int IntLinkedList_Fill_Rsd_This_Node;
public static int IntLinkedList_Fill_Tmp_Logger; ◀

public void Fill(int n)
{
   Contract.Requires(n > 0);
   Contract.Memory.Rsd<Node>(Contract.Memory.This, n);
   Contract.Memory.Tmp<Logger>(1);









   for (int i = 1; i <= n; i++)
   {
      Contract.Memory.DestRsd(Contract.Memory.This);

      Node node = new Node(i);
      this.PushFront(node);


   }

}
```



1. Este método es más complejo porque hay un call y un loop
2. Hay que tener en cuenta el tmp del método llamado
3. Si tuviese rsd también hay que tenerlo en cuenta
4. Maximizamos el tmp del callee porque necesitamos conocer el máximo de memoria temporal necesario en cualquier invocación, no la suma porque se libera cuando termina el método
5. Aclarar: anotaciones AddTmp y AddRsd no las mostramos porque no son necesarias, pero la instrumentación es muy similar

Verificación mediante instrumentación, ejemplo 2

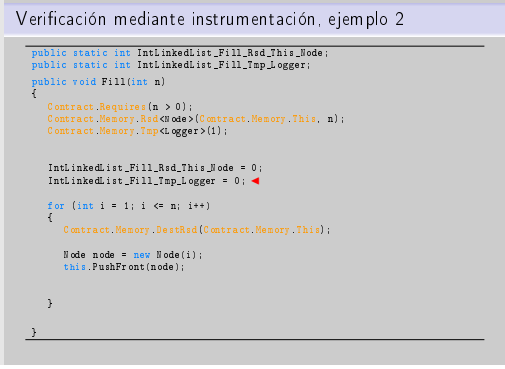
```
public static int IntLinkedList_Fill_Rsd_This_Node;
public static int IntLinkedList_Fill_Tmp_Logger;

public void Fill(int n)
{
   Contract.Requires(n > 0);
   Contract.Memory.Rsd<Node>(Contract.Memory.This, n);
   Contract.Memory.Tmp<Logger>(1);


   IntLinkedList_Fill_Rsd_This_Node = 0; ◀


   for (int i = 1; i <= n; i++)
   {
      Contract.Memory.DestRsd(Contract.Memory.This);

      Node node = new Node(i);
      this.PushFront(node);


   }

}
```



1. Este método es más complejo porque hay un call y un loop
2. Hay que tener en cuenta el tmp del método llamado
3. Si tuviese rsd también hay que tenerlo en cuenta
4. Maximizamos el tmp del callee porque necesitamos conocer el máximo de memoria temporal necesario en cualquier invocación, no la suma porque se libera cuando termina el método
5. Aclarar: anotaciones AddTmp y AddRsd no las mostramos porque no son necesarias, pero la instrumentación es muy similar

Verificación mediante instrumentación, ejemplo 2

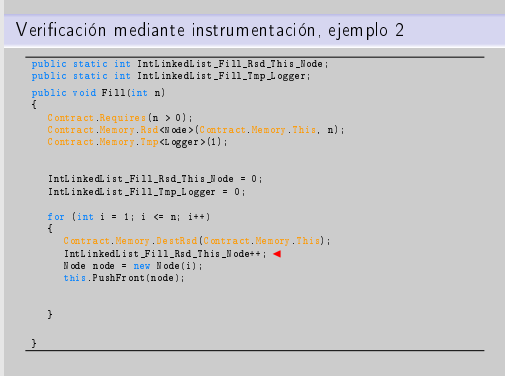
```
public static int IntLinkedList_Fill_Rsd_This_Node;
public static int IntLinkedList_Fill_Tmp_Logger;

public void Fill(int n)
{
   Contract.Requires(n > 0);
   Contract.Memory.Rsd<Node>(Contract.Memory.This, n);
   Contract.Memory.Tmp<Logger>(1);


   IntLinkedList_Fill_Rsd_This_Node = 0;
   IntLinkedList_Fill_Tmp_Logger = 0;

   for (int i = 1; i <= n; i++)
   {
      Contract.Memory.DestRsd(Contract.Memory.This);

      Node node = new Node(i);
      this.PushFront(node);


   }

}
```



1. Este método es más complejo porque hay un call y un loop
2. Hay que tener en cuenta el tmp del método llamado
3. Si tuviese rsd también hay que tenerlo en cuenta
4. Maximizamos el tmp del callee porque necesitamos conocer el máximo de memoria temporal necesario en cualquier invocación, no la suma porque se libera cuando termina el método
5. Aclarar: anotaciones AddTmp y AddRsd no las mostramos porque no son necesarias, pero la instrumentación es muy similar

Verificación mediante instrumentación, ejemplo 2

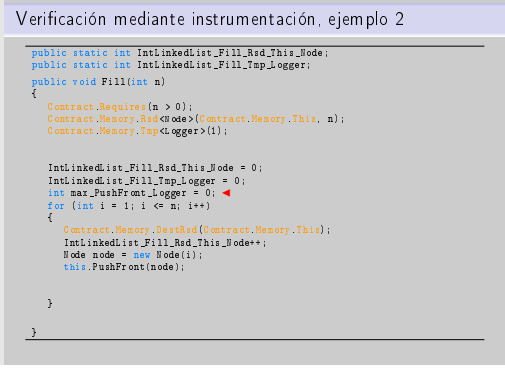
```
public static int IntLinkedList_Fill_Rsd_This_Node;
public static int IntLinkedList_Fill_Tmp_Logger;

public void Fill(int n)
{
   Contract.Requires(n > 0);
   Contract.Memory.Rsd<Node>(Contract.Memory.This, n);
   Contract.Memory.Tmp<Logger>(1);


   IntLinkedList_Fill_Rsd_This_Node = 0;
   IntLinkedList_Fill_Tmp_Logger = 0;

   for (int i = 1; i <= n; i++)
   {
      Contract.Memory.DestRsd(Contract.Memory.This);
      IntLinkedList_Fill_Rsd_This_Node++;
      Node node = new Node(i);
      this.PushFront(node);


   }

}
```



1. Este método es más complejo porque hay un call y un loop
2. Hay que tener en cuenta el tmp del método llamado
3. Si tuviese rsd también hay que tenerlo en cuenta
4. Maximizamos el tmp del callee porque necesitamos conocer el máximo de memoria temporal necesario en cualquier invocación, no la suma porque se libera cuando termina el método
5. Aclarar: anotaciones AddTmp y AddRsd no las mostramos porque no son necesarias, pero la instrumentación es muy similar

Verificación mediante instrumentación, ejemplo 2

```
public static int IntLinkedList_Fill_Rsd_This_Node;
public static int IntLinkedList_Fill_Tmp_Logger;

public void Fill(int n)
{
   Contract.Requires(n > 0);
   Contract.Memory.Rsd<Node>(Contract.Memory.This, n);
   Contract.Memory.Tmp<Logger>(1);


   IntLinkedList_Fill_Rsd_This_Node = 0;
   IntLinkedList_Fill_Tmp_Logger = 0;
   int max_PushFront_Logger = 0;
   for (int i = 1; i <= n; i++)
   {
      Contract.Memory.DestRsd(Contract.Memory.This);
      IntLinkedList_Fill_Rsd_This_Node++;
      Node node = new Node(i);
      this.PushFront(node);


   }

}
```



1. Este método es más complejo porque hay un call y un loop
2. Hay que tener en cuenta el tmp del método llamado
3. Si tuviese rsd también hay que tenerlo en cuenta
4. Maximizamos el tmp del callee porque necesitamos conocer el máximo de memoria temporal necesario en cualquier invocación, no la suma porque se libera cuando termina el método
5. Aclarar: anotaciones AddTmp y AddRsd no las mostramos porque no son necesarias, pero la instrumentación es muy similar

Verificación mediante instrumentación, ejemplo 2

```
public static int IntLinkedList_Fill_Rsd_This_Node;
public static int IntLinkedList_Fill_Tmp_Logger;

public void Fill(int n)
{
   Contract.Requires(n > 0);
   Contract.Memory.Rsd<Node>(Contract.Memory.This, n);
   Contract.Memory.Tmp<Logger>(1);


   IntLinkedList_Fill_Rsd_This_Node = 0;
   IntLinkedList_Fill_Tmp_Logger = 0;
   int max_PushFront_Logger = 0;
   for (int i = 1; i <= n; i++)
   {
      Contract.Memory.DestRsd(Contract.Memory.This);
      IntLinkedList_Fill_Rsd_This_Node++;
      Node node = new Node(i);
      this.PushFront(node);
      max_PushFront_Logger = Math.Max(IntLinkedList_PushFront_Tmp_Logger,
                                      max_PushFront_Logger);
   }

}
```

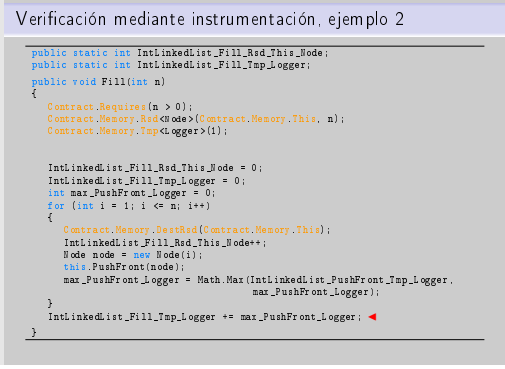
Verificación mediante instrumentación, ejemplo 2

```
public static int IntLinkedList_Fill_Rsd_This_Node;
public static int IntLinkedList_Fill_Tmp_Logger;

public void Fill(int n)
{
   Contract.Requires(n > 0);
   Contract.Memory.Rsd<Node>(Contract.Memory.This, n);
   Contract.Memory.Tmp<Logger>(1);


   IntLinkedList_Fill_Rsd_This_Node = 0;
   IntLinkedList_Fill_Tmp_Logger = 0;
   int max_PushFront_Logger = 0;
   for (int i = 1; i <= n; i++)
   {
      Contract.Memory.DestRsd(Contract.Memory.This);
      IntLinkedList_Fill_Rsd_This_Node++;
      Node node = new Node(i);
      this.PushFront(node);
      max_PushFront_Logger = Math.Max(IntLinkedList_PushFront_Tmp_Logger,
                                      max_PushFront_Logger);
   }

}
```

1. Este método es más complejo porque hay un call y un loop
2. Hay que tener en cuenta el tmp del método llamado
3. Si tuviese rsd también hay que tenerlo en cuenta
4. Maximizamos el tmp del callee porque necesitamos conocer el máximo de memoria temporal necesario en cualquier invocación, no la suma porque se libera cuando termina el método
5. Aclarar: anotaciones AddTmp y AddRsd no las mostramos porque no son necesarias, pero la instrumentación es muy similar

Verificación mediante instrumentación, ejemplo 2

```
public static int IntLinkedList_Fill_Rsd_This_Node;
public static int IntLinkedList_Fill_Tmp_Logger;

public void Fill(int n)
{
   Contract.Requires(n > 0);
   Contract.Memory.Rsd<Node>(Contract.Memory.This, n);
   Contract.Memory.Tmp<Logger>(1);


   IntLinkedList_Fill_Rsd_This_Node = 0;
   IntLinkedList_Fill_Tmp_Logger = 0;
   int max_PushFront_Logger = 0;
   for (int i = 1; i <= n; i++)
   {
      Contract.Memory.DestRsd(Contract.Memory.This);
      IntLinkedList_Fill_Rsd_This_Node++;
      Node node = new Node(i);
      this.PushFront(node);
      max_PushFront_Logger = Math.Max(IntLinkedList_PushFront_Tmp_Logger,
                                      max_PushFront_Logger);
   }
   IntLinkedList_Fill_Tmp_Logger += max_PushFront_Logger; ◀
}
```



1. Este método es más complejo porque hay un call y un loop
2. Hay que tener en cuenta el tmp del método llamado
3. Si tuviese rsd también hay que tenerlo en cuenta
4. Maximizamos el tmp del callee porque necesitamos conocer el máximo de memoria temporal necesario en cualquier invocación, no la suma porque se libera cuando termina el método
5. Aclarar: anotaciones AddTmp y AddRsd no las mostramos porque no son necesarias, pero la instrumentación es muy similar

Verificación mediante instrumentación, ejemplo 2

```
public static int IntLinkedList_Fill_Rsd_This_Node;
public static int IntLinkedList_Fill_Tmp_Logger;

public void Fill(int n)
{
   Contract.Requires(n > 0);
   Contract.Memory.Rsd<Node>(Contract.Memory.This, n);
   Contract.Memory.Tmp<Logger>(1);
   Contract.Ensures(IntLinkedList_Fill_Rsd_This_Node <= n); ◀

   IntLinkedList_Fill_Rsd_This_Node = 0;
   IntLinkedList_Fill_Tmp_Logger = 0;
   int max_PushFront_Logger = 0;
   for (int i = 1; i <= n; i++)
   {
      Contract.Memory.DestRsd(Contract.Memory.This);
      IntLinkedList_Fill_Rsd_This_Node++;
      Node node = new Node(i);
      this.PushFront(node);
      max_PushFront_Logger = Math.Max(IntLinkedList_PushFront_Tmp_Logger,
                                      max_PushFront_Logger);
   }
   IntLinkedList_Fill_Tmp_Logger += max_PushFront_Logger;
}
```

Verificación mediante instrumentación, ejemplo 2

```
public static int IntLinkedList_Fill_Rsd_This_Node;
public static int IntLinkedList_Fill_Tmp_Logger;

public void Fill(int n)
{
   Contract.Requires(n > 0);
   Contract.Memory.Rsd<Node>(Contract.Memory.This, n);
   Contract.Memory.Tmp<Logger>(1);
   Contract.Ensures(IntLinkedList_Fill_Rsd_This_Node <= n);

   IntLinkedList_Fill_Rsd_This_Node = 0;
   IntLinkedList_Fill_Tmp_Logger = 0;
   int max_PushFront_Logger = 0;
   for (int i = 1; i <= n; i++)
   {
      Contract.Memory.DestRsd(Contract.Memory.This);
      IntLinkedList_Fill_Rsd_This_Node++;
      Node node = new Node(i);
      this.PushFront(node);
      max_PushFront_Logger = Math.Max(IntLinkedList_PushFront_Tmp_Logger,
                                      max_PushFront_Logger);
   }
   IntLinkedList_Fill_Tmp_Logger += max_PushFront_Logger;
}
```

1. Este método es más complejo porque hay un call y un loop
2. Hay que tener en cuenta el tmp del método llamado
3. Si tuviese rsd también hay que tenerlo en cuenta
4. Maximizamos el tmp del callee porque necesitamos conocer el máximo de memoria temporal necesario en cualquier invocación, no la suma porque se libera cuando termina el método
5. Aclarar: anotaciones AddTmp y AddRsd no las mostramos porque no son necesarias, pero la instrumentación es muy similar

Verificación mediante instrumentación, ejemplo 2

```
public static int IntLinkedList_Fill_Rsd_This_Node;
public static int IntLinkedList_Fill_Tmp_Logger;

public void Fill(int n)
{
   Contract.Requires(n > 0);
   Contract.Memory.Rsd<Node>(Contract.Memory.This, n);
   Contract.Memory.Tmp<Logger>(1);
   Contract.Ensures(IntLinkedList_Fill_Rsd_This_Node <= n);
   Contract.Ensures(IntLinkedList_Fill_Tmp_Logger <= 1);
   IntLinkedList_Fill_Rsd_This_Node = 0;
   IntLinkedList_Fill_Tmp_Logger = 0;
   int max_PushFront_Logger = 0;
   for (int i = 1; i <= n; i++)
   {
      Contract.Memory.DestRsd(Contract.Memory.This);
      IntLinkedList_Fill_Rsd_This_Node++;
      Node node = new Node(i);
      this.PushFront(node);
      max_PushFront_Logger = Math.Max(IntLinkedList_PushFront_Tmp_Logger,
                                      max_PushFront_Logger);
   }
   IntLinkedList_Fill_Tmp_Logger += max_PushFront_Logger;
}
```

Verificación mediante instrumentación, ejemplo 2

```
public static int IntLinkedList_Fill_Rsd_This_Node;
public static int IntLinkedList_Fill_Tmp_Logger;

public void Fill(int n)
{
   Contract.Requires(n > 0);
   Contract.Memory.Rsd<Node>(Contract.Memory.This, n);
   Contract.Memory.Tmp<Logger>(1);
   Contract.Ensures(IntLinkedList_Fill_Rsd_This_Node <= n);
   Contract.Ensures(IntLinkedList_Fill_Tmp_Logger <= 1);
   IntLinkedList_Fill_Rsd_This_Node = 0;
   IntLinkedList_Fill_Tmp_Logger = 0;
   int max_PushFront_Logger = 0;
   for (int i = 1; i <= n; i++)
   {
      Contract.Memory.DestRsd(Contract.Memory.This);
      IntLinkedList_Fill_Rsd_This_Node++;
      Node node = new Node(i);
      this.PushFront(node);
      max_PushFront_Logger = Math.Max(IntLinkedList_PushFront_Tmp_Logger,
                                      max_PushFront_Logger);
   }
   IntLinkedList_Fill_Tmp_Logger += max_PushFront_Logger;
}
```

1. Este método es más complejo porque hay un call y un loop
2. Hay que tener en cuenta el tmp del método llamado
3. Si tuviese rsd también hay que tenerlo en cuenta
4. Maximizamos el tmp del callee porque necesitamos conocer el máximo de memoria temporal necesario en cualquier invocación, no la suma porque se libera cuando termina el método
5. Aclarar: anotaciones AddTmp y AddRsd no las mostramos porque no son necesarias, pero la instrumentación es muy similar

Verificación de anotaciones de tiempo de vida

- Necesitamos verificar que las anotaciones `Contract.Memory.DestTmp`, `Contract.Memory.DestRsd`, `Contract.Memory.AddTmp` y `Contract.Memory.AddRsd` son correctas
- Construimos un Points-to Graph (PTG) del método haciendo un análisis de points-to, grafo dirigido donde los nodos son objetos (o conjuntos de) y los arcos referencias entre ellos
- A partir del PTG hacemos un análisis de escape de objetos para ver qué objetos exceden el tiempo de vida del método
- Verificamos
  - Que los objetos escapen del método si y sólo si están anotados como residuales
  - Que los objetos residuales escapen del método a través de la expresión indicada

Verificación de anotaciones de tiempo de vida

- Necesitamos verificar que las anotaciones `Contract.Memory.DestTmp`, `Contract.Memory.DestRsd`, `Contract.Memory.AddTmp` y `Contract.Memory.AddRsd` son correctas
- Construimos un Points-to Graph (PTG) del método haciendo un análisis de points-to, grafo dirigido donde los nodos son objetos (o conjuntos de) y los arcos referencias entre ellos
- A partir del PTG hacemos un análisis de escape de objetos para ver qué objetos exceden el tiempo de vida del método
- Verificamos
  - Que los objetos escapen del método si y sólo si están anotados como residuales
  - Que los objetos residuales escapen del método a través de la expresión indicada

1. Para que los contratos sean correctos, las anotaciones de tiempo de vida tienen que serlo

Verificación de anotaciones de tiempo de vida

- Necesitamos verificar que las anotaciones `Contract.Memory.DestTmp`, `Contract.Memory.DestRsd`, `Contract.Memory.AddTmp` y `Contract.Memory.AddRsd` son correctas
- Construimos un Points-to Graph (PTG) del método haciendo un análisis de points-to, grafo dirigido donde los nodos son objetos (o conjuntos de) y los arcos referencias entre ellos
- A partir del PTG hacemos un análisis de escape de objetos para ver qué objetos exceden el tiempo de vida del método
- Verificamos
  - Que los objetos escapen del método si y sólo si están anotados como residuales
  - Que los objetos residuales escapen del método a través de la expresión indicada

Verificación de anotaciones de tiempo de vida

- Necesitamos verificar que las anotaciones `Contract.Memory.DestTmp`, `Contract.Memory.DestRsd`, `Contract.Memory.AddTmp` y `Contract.Memory.AddRsd` son correctas
- Construimos un Points-to Graph (PTG) del método haciendo un análisis de points-to, grafo dirigido donde los nodos son objetos (o conjuntos de) y los arcos referencias entre ellos
- A partir del PTG hacemos un análisis de escape de objetos para ver qué objetos exceden el tiempo de vida del método
- Verificamos
  - Que los objetos escapen del método si y sólo si están anotados como residuales
  - Que los objetos residuales escapen del método a través de la expresión indicada

1. Para que los contratos sean correctos, las anotaciones de tiempo de vida tienen que serlo

Verificación de anotaciones de tiempo de vida

- Necesitamos verificar que las anotaciones `Contract.Memory.DestTmp`, `Contract.Memory.DestRsd`, `Contract.Memory.AddTmp` y `Contract.Memory.AddRsd` son correctas
- Construimos un Points-to Graph (PTG) del método haciendo un análisis de points-to, grafo dirigido donde los nodos son objetos (o conjuntos de) y los arcos referencias entre ellos
- A partir del PTG hacemos un análisis de escape de objetos para ver qué objetos exceden el tiempo de vida del método
- Verificamos
  - Que los objetos escapen del método si y sólo si están anotados como residuales
  - Que los objetos residuales escapen del método a través de la expresión indicada

Verificación de anotaciones de tiempo de vida

- Necesitamos verificar que las anotaciones `Contract.Memory.DestTmp`, `Contract.Memory.DestRsd`, `Contract.Memory.AddTmp` y `Contract.Memory.AddRsd` son correctas
- Construimos un Points-to Graph (PTG) del método haciendo un análisis de points-to, grafo dirigido donde los nodos son objetos (o conjuntos de) y los arcos referencias entre ellos
- A partir del PTG hacemos un análisis de escape de objetos para ver qué objetos exceden el tiempo de vida del método
- Verificamos
  - Que los objetos escapen del método si y sólo si están anotados como residuales
  - Que los objetos residuales escapen del método a través de la expresión indicada

1. Para que los contratos sean correctos, las anotaciones de tiempo de vida tienen que serlo

Verificación de anotaciones de tiempo de vida

- Necesitamos verificar que las anotaciones `Contract.Memory.DestTmp`, `Contract.Memory.DestRsd`, `Contract.Memory.AddTmp` y `Contract.Memory.AddRsd` son correctas
- Construimos un Points-to Graph (PTG) del método haciendo un análisis de points-to, grafo dirigido donde los nodos son objetos (o conjuntos de) y los arcos referencias entre ellos
- A partir del PTG hacemos un análisis de escape de objetos para ver qué objetos exceden el tiempo de vida del método
- Verificamos
  - Que los objetos escapen del método si y sólo si están anotados como residuales
  - Que los objetos residuales escapen del método a través de la expresión indicada

Verificación de anotaciones de tiempo de vida

- Necesitamos verificar que las anotaciones `Contract.Memory.DestTmp`, `Contract.Memory.DestRsd`, `Contract.Memory.AddTmp` y `Contract.Memory.AddRsd` son correctas
- Construimos un Points-to Graph (PTG) del método haciendo un análisis de points-to, grafo dirigido donde los nodos son objetos (o conjuntos de) y los arcos referencias entre ellos
- A partir del PTG hacemos un análisis de escape de objetos para ver qué objetos exceden el tiempo de vida del método
- Verificamos
  - Que los objetos escapen del método si y sólo si están anotados como residuales
  - Que los objetos residuales escapen del método a través de la expresión indicada

1. Para que los contratos sean correctos, las anotaciones de tiempo de vida tienen que serlo

Verificación de anotaciones de tiempo de vida, ejemplo

```
1  public void Fill(int n)
2  {
3    Contract.Requires(n > 0);
4    Contract.Memory.Rsd<Node>(Contract.Memory.This, n);
5    Contract.Memory.Tmp<Logger>(1);
6
7    for (int i = 1; i <= n; i++)
8    {
9      Contract.Memory.DestRsd(Contract.Memory.This);
10     Node node = new Node(i);
11     this.PushFront(node);
12   }
13 }
```

Verificación de anotaciones de tiempo de vida, ejemplo

```
1  public void Fill(int n)
2  {
3    Contract.Requires(n > 0);
4    Contract.Memory.Rsd<Node>(Contract.Memory.This, n);
5    Contract.Memory.Tmp<Logger>(1);
6
7    for (int i = 1; i <= n; i++)
8    {
9      Contract.Memory.DestRsd(Contract.Memory.This);
10     Node node = new Node(i);
11     this.PushFront(node);
12   }
13 }
```

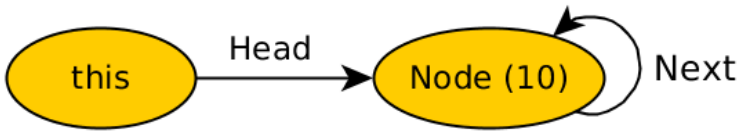
1. Método Fill anotado
2. Vamos a ver la info que obtener del análisis de points-to y escape
3. Y cómo usamos esa info para verificar
4. Con la misma info podemos verificar que AddTmp y AddRsd sean correctos, es decir que los objetos residuales de un método invocado se conviertan en lo tmp o rsd locales según la anotación dada diga

Verificación de anotaciones de tiempo de vida, ejemplo

```
1  public void Fill(int n)
2  {
3    Contract.Requires(n > 0);
4    Contract.Memory.Rsd<Node>(Contract.Memory.This, n);
5    Contract.Memory.Tmp<Logger>(1);
6
7    for (int i = 1; i <= n; i++)
8    {
9      Contract.Memory.DestRsd(Contract.Memory.This);
10     Node node = new Node(i);
11     this.PushFront(node);
12   }
13 }
```

Del análisis de points-to y escape obtenemos los siguientes datos:

- Points-to Graph:



- El objeto de la línea 10 escapa

Verificación de anotaciones de tiempo de vida, ejemplo

```
1  public void Fill(int n)
2  {
3    Contract.Requires(n > 0);
4    Contract.Memory.Rsd<Node>(Contract.Memory.This, n);
5    Contract.Memory.Tmp<Logger>(1);
6
7    for (int i = 1; i <= n; i++)
8    {
9      Contract.Memory.DestRsd(Contract.Memory.This);
10     Node node = new Node(i);
11     this.PushFront(node);
12   }
13 }
```

Del análisis de points-to y escape obtenemos los siguientes datos:

- Points-to Graph: 

- El objeto de la línea 10 escapa

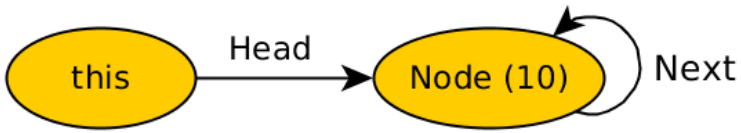
1. Método Fill anotado
2. Vamos a ver la info que obtener del análisis de points-to y escape
3. Y cómo usamos esa info para verificar
4. Con la misma info podemos verificar que AddTmp y AddRsd sean correctos, es decir que los objetos residuales de un método invocado se conviertan en lo tmp o rsd locales según la anotación dada diga

Verificación de anotaciones de tiempo de vida, ejemplo

```
1  public void Fill(int n)
2  {
3    Contract.Requires(n > 0);
4    Contract.Memory.Rsd<Node>(Contract.Memory.This, n);
5    Contract.Memory.Tmp<Logger>(1);
6
7    for (int i = 1; i <= n; i++)
8    {
9      Contract.Memory.DestRsd(Contract.Memory.This);
10     Node node = new Node(i);
11     this.PushFront(node);
12   }
13 }
```

Del análisis de points-to y escape obtenemos los siguientes datos:

- Points-to Graph:



- El objeto de la línea 10 escapa $\Longrightarrow$ es residual, `DestRsd` es correcta

Verificación de anotaciones de tiempo de vida, ejemplo

```
1  public void Fill(int n)
2  {
3    Contract.Requires(n > 0);
4    Contract.Memory.Rsd<Node>(Contract.Memory.This, n);
5    Contract.Memory.Tmp<Logger>(1);
6
7    for (int i = 1; i <= n; i++)
8    {
9      Contract.Memory.DestRsd(Contract.Memory.This);
10     Node node = new Node(i);
11     this.PushFront(node);
12   }
13 }
```

Del análisis de points-to y escape obtenemos los siguientes datos:

- Points-to Graph:

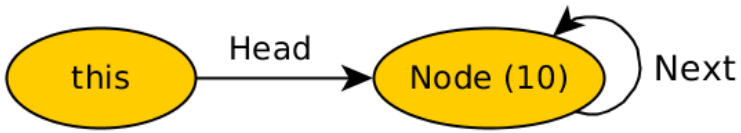
- El objeto de la línea 10 escapa $\Longrightarrow$ es residual, `DestRsd` es correcta

1. Método Fill anotado
2. Vamos a ver la info que obtener del análisis de points-to y escape
3. Y cómo usamos esa info para verificar
4. Con la misma info podemos verificar que AddTmp y AddRsd sean correctos, es decir que los objetos residuales de un método invocado se conviertan en lo tmp o rsd locales según la anotación dada diga

Verificación de anotaciones de tiempo de vida, ejemplo

```
1  public void Fill(int n)
2  {
3    Contract.Requires(n > 0);
4    Contract.Memory.Rsd<Node>(Contract.Memory.This, n);
5    Contract.Memory.Tmp<Logger>(1);
6
7    for (int i = 1; i <= n; i++)
8    {
9      Contract.Memory.DestRsd(Contract.Memory.This);
10     Node node = new Node(i);
11     this.PushFront(node);
12   }
13 }
```

Del análisis de points-to y escape obtenemos los siguientes datos:

- Points-to Graph: 

$\Longrightarrow$ escapa por `this`
- El objeto de la línea 10 escapa $\Longrightarrow$ es residual, `DestRsd` es correcta

Verificación de anotaciones de tiempo de vida, ejemplo

```
1  public void Fill(int n)
2  {
3    Contract.Requires(n > 0);
4    Contract.Memory.Rsd<Node>(Contract.Memory.This, n);
5    Contract.Memory.Tmp<Logger>(1);
6
7    for (int i = 1; i <= n; i++)
8    {
9      Contract.Memory.DestRsd(Contract.Memory.This);
10     Node node = new Node(i);
11     this.PushFront(node);
12   }
13 }
```

Del análisis de points-to y escape obtenemos los siguientes datos:

- Points-to Graph: 

$\Longrightarrow$ escapa por `this`
- El objeto de la línea 10 escapa $\Longrightarrow$ es residual, `DestRsd` es correcta

1. Método Fill anotado
2. Vamos a ver la info que obtener del análisis de points-to y escape
3. Y cómo usamos esa info para verificar
4. Con la misma info podemos verificar que AddTmp y AddRsd sean correctos, es decir que los objetos residuales de un método invocado se conviertan en lo tmp o rsd locales según la anotación dada diga

Verificación con aritmética no lineal

- El verificador de Code Contracts sólo soporta aritmética lineal
- Integramos una herramienta externa, Barvinok, capaz de resolver operaciones con polinomios
- Durante la verificación
  - Usamos la herramienta para calcular expresiones polinomiales e incrementamos los contadores con valores pre-calculados
  - Obtenemos aserciones acerca de la validez de los contratos y las brindamos al verificador mediante la anotación `Assume`
- Requerimos para estos casos que se anote un `IterationSpace` para los ciclos

Verificación con aritmética no lineal
- El verificador de Code Contracts sólo soporta aritmética lineal
- Integramos una herramienta externa, Barvinok, capaz de resolver operaciones con polinomios
- Durante la verificación
  - Usamos la herramienta para calcular expresiones polinomiales e incrementamos los contadores con valores pre-calculados
  - Obtenemos aserciones acerca de la validez de los contratos y las brindamos al verificador mediante la anotación `Assume`
- Requerimos para estos casos que se anote un `IterationSpace` para los ciclos

1. Aclarar que otros verificadores estáticos también manejan sólo aritmética lineal
2. En gral, aparece una multiplicación e ignoran, no pueden probar
3. En consumo de memoria aparecen fácil las multiplicaciones, pensar en loop anidado, parecido a análisis de complejidad temporal

Verificación con aritmética no lineal

- El verificador de Code Contracts sólo soporta aritmética lineal
- Integramos una herramienta externa, Barvinok, capaz de resolver operaciones con polinomios
- Durante la verificación
  - Usamos la herramienta para calcular expresiones polinomiales e incrementamos los contadores con valores pre-calculados
  - Obtenemos aserciones acerca de la validez de los contratos y las brindamos al verificador mediante la anotación `Assume`
- Requerimos para estos casos que se anote un `IterationSpace` para los ciclos

Verificación con aritmética no lineal
- El verificador de Code Contracts sólo soporta aritmética lineal
- Integramos una herramienta externa, Barvinok, capaz de resolver operaciones con polinomios
- Durante la verificación
  - Usamos la herramienta para calcular expresiones polinomiales e incrementamos los contadores con valores pre-calculados
  - Obtenemos aserciones acerca de la validez de los contratos y las brindamos al verificador mediante la anotación `Assume`
- Requerimos para estos casos que se anote un `IterationSpace` para los ciclos

1. Aclarar que otros verificadores estáticos también manejan sólo aritmética lineal
2. En gral, aparece una multiplicación e ignoran, no pueden probar
3. En consumo de memoria aparecen fácil las multiplicaciones, pensar en loop anidado, parecido a análisis de complejidad temporal

Verificación con aritmética no lineal

- El verificador de Code Contracts sólo soporta aritmética lineal
- Integramos una herramienta externa, Barvinok, capaz de resolver operaciones con polinomios
- Durante la verificación
  - Usamos la herramienta para calcular expresiones polinomiales e incrementamos los contadores con valores pre-calculados
  - Obtenemos aserciones acerca de la validez de los contratos y las brindamos al verificador mediante la anotación `Assume`
- Requerimos para estos casos que se anote un `IterationSpace` para los ciclos

Verificación con aritmética no lineal
- El verificador de Code Contracts sólo soporta aritmética lineal
- Integramos una herramienta externa, Barvinok, capaz de resolver operaciones con polinomios
- Durante la verificación
  - Usamos la herramienta para calcular expresiones polinomiales e incrementamos los contadores con valores pre-calculados
  - Obtenemos aserciones acerca de la validez de los contratos y las brindamos al verificador mediante la anotación `Assume`
- Requerimos para estos casos que se anote un `IterationSpace` para los ciclos

1. Aclarar que otros verificadores estáticos también manejan sólo aritmética lineal
2. En gral, aparece una multiplicación e ignoran, no pueden probar
3. En consumo de memoria aparecen fácil las multiplicaciones, pensar en loop anidado, parecido a análisis de complejidad temporal

Verificación con aritmética no lineal

- El verificador de Code Contracts sólo soporta aritmética lineal
- Integramos una herramienta externa, Barvinok, capaz de resolver operaciones con polinomios
- Durante la verificación
  - Usamos la herramienta para calcular expresiones polinomiales e incrementamos los contadores con valores pre-calculados
  - Obtenemos aserciones acerca de la validez de los contratos y las brindamos al verificador mediante la anotación `Assume`
- Requerimos para estos casos que se anote un `IterationSpace` para los ciclos

Verificación con aritmética no lineal

- El verificador de Code Contracts sólo soporta aritmética lineal
- Integramos una herramienta externa, Barvinok, capaz de resolver operaciones con polinomios
- Durante la verificación
  - Usamos la herramienta para calcular expresiones polinomiales e incrementamos los contadores con valores pre-calculados
  - Obtenemos aserciones acerca de la validez de los contratos y las brindamos al verificador mediante la anotación `Assume`
- Requerimos para estos casos que se anote un `IterationSpace` para los ciclos

1. Aclarar que otros verificadores estáticos también manejan sólo aritmética lineal
2. En gral, aparece una multiplicación e ignoran, no pueden probar
3. En consumo de memoria aparecen fácil las multiplicaciones, pensar en loop anidado, parecido a análisis de complejidad temporal

Verificación con aritmética no lineal, ejemplo

```
1  public void ConsumoCuadratico(int n)
2  {
3      Contract.Requires(n > 0);
4      Contract.Memory.Tmp<Logger>(n * n);
5
6      for (int i = 1; i <= n; i++)
7      {
8          for (int j = 1; j <= n; j++)
9          {
10             Contract.Memory.DestTmp();
11             Logger logger = new Logger();
12             logger.Log("Log " + (i * j).ToString());
13         }
14     }
15 }
```

- El contrato dado es correcto
- El verificador de Code Contracts no es capaz de verificarlo con la instrumentación descripta hasta el momento
- Con un poco de ayuda del usuario y de una herramienta externa podemos verificarlo

Verificación con aritmética no lineal, ejemplo

```
1  public void ConsumoCuadratico(int n)
2  {
3      Contract.Requires(n > 0);
4      Contract.Memory.Tmp<Logger>(n * n);
5
6      for (int i = 1; i <= n; i++)
7      {
8          for (int j = 1; j <= n; j++)
9          {
10             Contract.Memory.DestTmp();
11             Logger logger = new Logger();
12             logger.Log("Log " + (i * j).ToString());
13         }
14     }
15 }
```

- El contrato dado es correcto
- El verificador de Code Contracts no es capaz de verificarlo con la instrumentación descripta hasta el momento
- Con un poco de ayuda del usuario y de una herramienta externa podemos verificarlo

1. Ejemplo que no puede verificar Code Contracts

Verificación con aritmética no lineal, ejemplo

```
1  public void ConsumoCuadratico(int n)
2  {
3      Contract.Requires(n > 0);
4      Contract.Memory.Tmp<Logger>(n * n);
5
6      for (int i = 1; i <= n; i++)
7      {
8          for (int j = 1; j <= n; j++)
9          {
10             Contract.Memory.DestTmp();
11             Logger logger = new Logger();
12             logger.Log("Log " + (i * j).ToString());
13         }
14     }
15 }
```

- El contrato dado es correcto
- El verificador de Code Contracts no es capaz de verificarlo con la instrumentación descripta hasta el momento
- Con un poco de ayuda del usuario y de una herramienta externa podemos verificarlo

Verificación con aritmética no lineal, ejemplo

```
1  public void ConsumoCuadratico(int n)
2  {
3      Contract.Requires(n > 0);
4      Contract.Memory.Tmp<Logger>(n * n);
5
6      for (int i = 1; i <= n; i++)
7      {
8          for (int j = 1; j <= n; j++)
9          {
10             Contract.Memory.DestTmp();
11             Logger logger = new Logger();
12             logger.Log("Log " + (i * j).ToString());
13         }
14     }
15 }
```

- El contrato dado es correcto
- El verificador de Code Contracts no es capaz de verificarlo con la instrumentación descripta hasta el momento
- Con un poco de ayuda del usuario y de una herramienta externa podemos verificarlo

1. Ejemplo que no puede verificar Code Contracts

Verificación con aritmética no lineal, ejemplo

```
1  public void ConsumoCuadratico(int n)
2  {
3      Contract.Requires(n > 0);
4      Contract.Memory.Tmp<Logger>(n * n);
5
6      for (int i = 1; i <= n; i++)
7      {
8          for (int j = 1; j <= n; j++)
9          {
10             Contract.Memory.DestTmp();
11             Logger logger = new Logger();
12             logger.Log("Log " + (i * j).ToString());
13         }
14     }
15 }
```

- El contrato dado es correcto
- El verificador de Code Contracts no es capaz de verificarlo con la instrumentación descripta hasta el momento
- Con un poco de ayuda del usuario y de una herramienta externa podemos verificarlo

Verificación con aritmética no lineal, ejemplo

```
1  public void ConsumoCuadratico(int n)
2  {
3      Contract.Requires(n > 0);
4      Contract.Memory.Tmp<Logger>(n * n);
5
6      for (int i = 1; i <= n; i++)
7      {
8          for (int j = 1; j <= n; j++)
9          {
10             Contract.Memory.DestTmp();
11             Logger logger = new Logger();
12             logger.Log("Log " + (i * j).ToString());
13         }
14     }
15 }
```

- El contrato dado es correcto
- El verificador de Code Contracts no es capaz de verificarlo con la instrumentación descripta hasta el momento
- Con un poco de ayuda del usuario y de una herramienta externa podemos verificarlo

1. Ejemplo que no puede verificar Code Contracts

Verificación con aritmética no lineal, ejemplo

```
1  public void ConsumoCuadratico(int n)
2  {
3      Contract.Requires(n > 0);
4      Contract.Memory.Tmp<Logger>(n * n);
5
6      for (int i = 1; i <= n; i++)
7      {
8          for (int j = 1; j <= n; j++)
9          {
10             Contract.Memory.DestTmp();
11             Logger logger = new Logger();
12             logger.Log("Log " + (i * j).ToString());
13         }
14     }
15 }
```

- El contrato dado es correcto
- El verificador de Code Contracts no es capaz de verificarlo con la instrumentación descripta hasta el momento
- Con un poco de ayuda del usuario y de una herramienta externa podemos verificarlo

Verificación con aritmética no lineal, ejemplo

```
1  public void ConsumoCuadratico(int n)
2  {
3      Contract.Requires(n > 0);
4      Contract.Memory.Tmp<Logger>(n * n);
5
6      for (int i = 1; i <= n; i++)
7      {
8          for (int j = 1; j <= n; j++)
9          {
10             Contract.Memory.DestTmp();
11             Logger logger = new Logger();
12             logger.Log("Log " + (i * j).ToString());
13         }
14     }
15 }
```

- El contrato dado es correcto
- El verificador de Code Contracts no es capaz de verificarlo con la instrumentación descripta hasta el momento
- Con un poco de ayuda del usuario y de una herramienta externa podemos verificarlo

1. Ejemplo que no puede verificar Code Contracts

Verificación con aritmética no lineal, ejemplo

Para poder calcular la cantidad de objetos temporales necesarios necesitamos ayuda del usuario del para entender los espacios de iteración:

```
public void ConsumoCuadratico(int n)
{
   Contract.Requires(n > 0);
   Contract.Memory.Tmp<Logger>(n * n);

   for (int i = 1; i <= n; i++)
   {

      for (int j = 1; j <= n; j++)
      {

         Contract.Memory.DestTmp();
         Logger logger = new Logger();
         logger.Log("Log "+ (i * j).ToString());
      }
   }
}
```

Verificación con aritmética no lineal, ejemplo

Para poder calcular la cantidad de objetos temporales necesarios necesitamos ayuda del usuario del para entender los espacios de iteración:

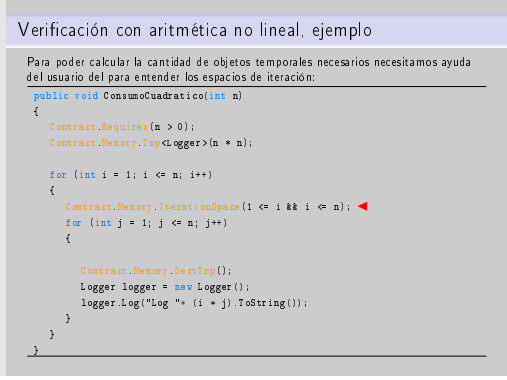
```
public void ConsumoCuadratico(int n)
{
   Contract.Requires(n > 0);
   Contract.Memory.Tmp<Logger>(n * n);

   for (int i = 1; i <= n; i++)
   {

      for (int j = 1; j <= n; j++)
      {

         Contract.Memory.DestTmp();
         Logger logger = new Logger();
         logger.Log("Log "+ (i * j).ToString());
      }
   }
}
```

1. Si el usuario anota los espacios de iteración (fácil), podemos hacer una instrumentación diferente
2. Ahora vemos la instrumentación nueva

Verificación con aritmética no lineal, ejemplo

Para poder calcular la cantidad de objetos temporales necesarios necesitamos ayuda del usuario del para entender los espacios de iteración:

```
public void ConsumoCuadratico(int n)
{
   Contract.Requires(n > 0);
   Contract.Memory.Tmp<Logger>(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.DestTmp();
         Logger logger = new Logger();
         logger.Log("Log "+ (i * j).ToString());
      }
   }
}
```

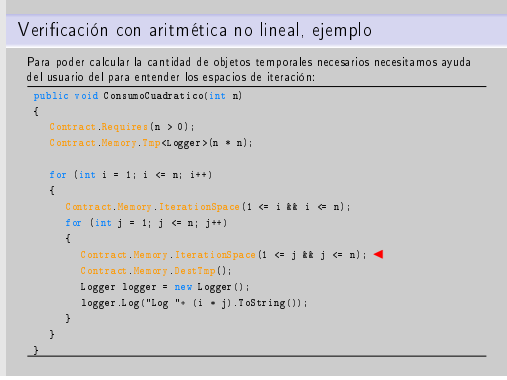


1. Si el usuario anota los espacios de iteración (fácil), podemos hacer una instrumentación diferente
2. Ahora vemos la instrumentación nueva

Verificación con aritmética no lineal, ejemplo

Para poder calcular la cantidad de objetos temporales necesarios necesitamos ayuda del usuario del para entender los espacios de iteración:

```
public void ConsumoCuadratico(int n)
{
   Contract.Requires(n > 0);
   Contract.Memory.Tmp<Logger>(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();
         Logger logger = new Logger();
         logger.Log("Log "+ (i * j).ToString());
      }
   }
}
```



1. Si el usuario anota los espacios de iteración (fácil), podemos hacer una instrumentación diferente
2. Ahora vemos la instrumentación nueva

Verificación con aritmética no lineal, ejemplo

```
public void ConsumoCuadratico(int n)
{
   Contract.Requires(n > 0);
   Contract.Memory.Tmp<Logger>(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();
         Logger logger = new Logger();
         logger.Log("Log "+ (i * j).ToString());
      }
   }


}
```

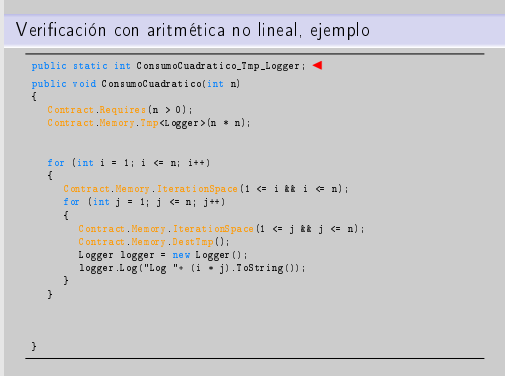
Verificación con aritmética no lineal, ejemplo

```
public void ConsumoCuadratico(int n)
{
   Contract.Requires(n > 0);
   Contract.Memory.Tmp<Logger>(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();
         Logger logger = new Logger();
         logger.Log("Log "+ (i * j).ToString());
      }
   }


}
```

1. Mostrar de a poco y explicar
2. En lugar de incrementar el contador después del new, lo incrementamos afuera del loop con la cantidad de veces que se debería incrementar
3. El `n * n` sale de la herramienta aritmética usando la info de los espacios de iteración
4. El if assume también, pero de verificar que el contrato es correcto, en casos más complejos es necesario, pensar en sumas de varias cosas

Verificación con aritmética no lineal, ejemplo

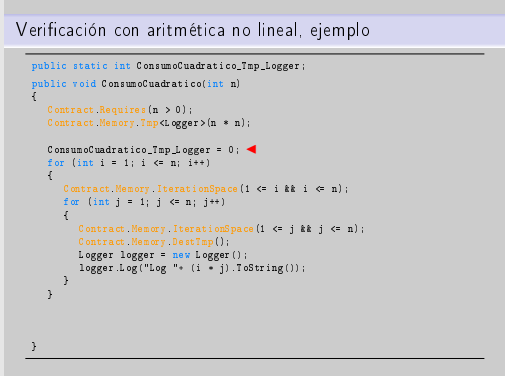
```
public static int ConsumoCuadratico_Tmp_Logger;

public void ConsumoCuadratico(int n)
{
   Contract.Requires(n > 0);
   Contract.Memory.Tmp<Logger>(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();
         Logger logger = new Logger();
         logger.Log("Log "+ (i * j).ToString());
      }
   }


}
```



1. Mostrar de a poco y explicar
2. En lugar de incrementar el contador después del new, lo incrementamos afuera del loop con la cantidad de veces que se debería incrementar
3. El `n * n` sale de la herramienta aritmética usando la info de los espacios de iteración
4. El if assume también, pero de verificar que el contrato es correcto, en casos más complejos es necesario, pensar en sumas de varias cosas

Verificación con aritmética no lineal, ejemplo

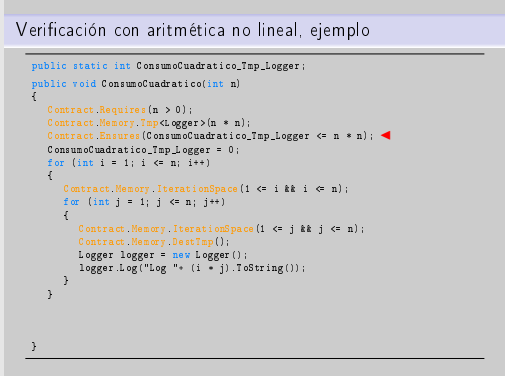
```
public static int ConsumoCuadratico_Tmp_Logger;

public void ConsumoCuadratico(int n)
{
   Contract.Requires(n > 0);
   Contract.Memory.Tmp<Logger>(n * n);

   ConsumoCuadratico_Tmp_Logger = 0;
   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();
         Logger logger = new Logger();
         logger.Log("Log "+ (i * j).ToString());
      }
   }


}
```



1. Mostrar de a poco y explicar
2. En lugar de incrementar el contador después del new, lo incrementamos afuera del loop con la cantidad de veces que se debería incrementar
3. El `n * n` sale de la herramienta aritmética usando la info de los espacios de iteración
4. El if assume también, pero de verificar que el contrato es correcto, en casos más complejos es necesario, pensar en sumas de varias cosas

Verificación con aritmética no lineal, ejemplo

```
public static int ConsumoCuadratico_Tmp_Logger;

public void ConsumoCuadratico(int n)
{
   Contract.Requires(n > 0);
   Contract.Memory.Tmp<Logger>(n * n);
   Contract.Ensures(ConsumoCuadratico_Tmp_Logger <= n * n);
   ConsumoCuadratico_Tmp_Logger = 0;
   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();
         Logger logger = new Logger();
         logger.Log("Log "+ (i * j).ToString());
      }
   }



}
```



1. Mostrar de a poco y explicar
2. En lugar de incrementar el contador después del new, lo incrementamos afuera del loop con la cantidad de veces que se debería incrementar
3. El `n * n` sale de la herramienta aritmética usando la info de los espacios de iteración
4. El if assume también, pero de verificar que el contrato es correcto, en casos más complejos es necesario, pensar en sumas de varias cosas

Verificación con aritmética no lineal, ejemplo

```
public static int ConsumoCuadratico_Tmp_Logger;

public void ConsumoCuadratico(int n)
{
   Contract.Requires(n > 0);
   Contract.Memory.Tmp<Logger>(n * n);
   Contract.Ensures(ConsumoCuadratico_Tmp_Logger <= n * n);
   ConsumoCuadratico_Tmp_Logger = 0;
   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();
         Logger logger = new Logger();
         logger.Log("Log "+ (i * j).ToString());
      }
   }
   ConsumoCuadratico_Tmp_Logger += n * n;


}
```

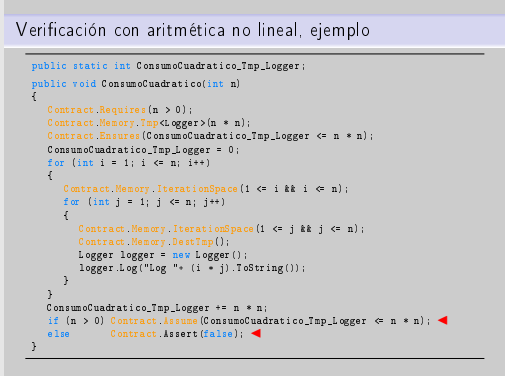
Verificación con aritmética no lineal, ejemplo

```
public static int ConsumoCuadratico_Tmp_Logger;
public void ConsumoCuadratico(int n)
{
    Contract.Requires(n > 0);
    Contract.Memory.Tmp<Logger>(n * n);
    Contract.Ensures(ConsumoCuadratico_Tmp_Logger <= n * n);
    ConsumoCuadratico_Tmp_Logger = 0;
    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();
            Logger logger = new Logger();
            logger.Log("Log "+ (i * j).ToString());
        }
    }
    ConsumoCuadratico_Tmp_Logger += n * n;


}
```

1. Mostrar de a poco y explicar
2. En lugar de incrementar el contador después del new, lo incrementamos afuera del loop con la cantidad de veces que se debería incrementar
3. El `n * n` sale de la herramienta aritmética usando la info de los espacios de iteración
4. El if assume también, pero de verificar que el contrato es correcto, en casos más complejos es necesario, pensar en sumas de varias cosas

Verificación con aritmética no lineal, ejemplo

```
public static int ConsumoCuadratico_Tmp_Logger;

public void ConsumoCuadratico(int n)
{
   Contract.Requires(n > 0);
   Contract.Memory.Tmp<Logger>(n * n);
   Contract.Ensures(ConsumoCuadratico_Tmp_Logger <= n * n);
   ConsumoCuadratico_Tmp_Logger = 0;
   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();
         Logger logger = new Logger();
         logger.Log("Log "+ (i * j).ToString());
      }
   }
   ConsumoCuadratico_Tmp_Logger += n * n;
   if (n > 0) Contract.Assume(ConsumoCuadratico_Tmp_Logger <= n * n); ◀
   else       Contract.Assert(false); ◀
}
```



1. Mostrar de a poco y explicar
2. En lugar de incrementar el contador después del new, lo incrementamos afuera del loop con la cantidad de veces que se debería incrementar
3. El `n * n` sale de la herramienta aritmética usando la info de los espacios de iteración
4. El if assume también, pero de verificar que el contrato es correcto, en casos más complejos es necesario, pensar en sumas de varias cosas

Verificación, demo

- Cuando compilamos en Visual Studio se dispara la verificación
- Los resultados se ven en el área de mensajes del compilador
- Vimos algunos ejemplos en funcionamiento
  - Verificación correcta
  - Verificación con contrato incorrecto
  - Verificación con una anotación de tiempo de vida incorrecta

1. Vamos a ver un poco el prototipo funcionando
2. Para mostrar cómo se integra en la IDE

Verificación, demo

- Cuando compilamos en Visual Studio se dispara la verificación
- Los resultados se ven en el área de mensajes del compilador
- Vimos algunos ejemplos en funcionamiento
  - Verificación correcta
  - Verificación con contrato incorrecto
  - Verificación con una anotación de tiempo de vida incorrecta

1. Vamos a ver un poco el prototipo funcionando
2. Para mostrar cómo se integra en la IDE

Verificación, demo

- Cuando compilamos en Visual Studio se dispara la verificación
- Los resultados se ven en el área de mensajes del compilador
- Vimos algunos ejemplos en funcionamiento
  - Verificación correcta
  - Verificación con contrato incorrecto
  - Verificación con una anotación de tiempo de vida incorrecta

- Cuando compilamos en Visual Studio se dispara la verificación
- Los resultados se ven en el área de mensajes del compilador
- Vimos algunos ejemplos en funcionamiento
  - Verificación correcta
  - Verificación con contrato incorrecto
  - Verificación con una anotación de tiempo de vida incorrecta

1. Vamos a ver un poco el prototipo funcionando
2. Para mostrar cómo se integra en la IDE

Índice

Limitaciones

- Estamos limitados por las capacidades de:
  - El verificador estático
  - La herramienta para resolver operaciones con aritmética no lineal
  - $\rightsquigarrow$ El diseño de las técnicas permite reemplazarlas por herramientas con mayores o mejores capacidades cuando existan
- El análisis de tiempo de vida es sobre-aproximado
  - $\rightsquigarrow$ El análisis de points-to es un problema indecidible
  - $\rightsquigarrow$ Permitimos obviar la verificación para casos en que falla
- Requerimos que el usuario provea todas las anotaciones de tiempo de vida para la verificación, y los espacios de iteración para utilizar la herramienta para aritmética no lineal
  - $\rightsquigarrow$ En un futuro pensamos inferir estas anotaciones

Limitaciones
- Estamos limitados por las capacidades de:
  - El verificador estático
  - La herramienta para resolver operaciones con aritmética no lineal
  - ⇝ El diseño de las técnicas permite reemplazarlas por herramientas con mayores o mejores capacidades cuando existan
- El análisis de tiempo de vida es sobre-aproximado
  - ⇝ El análisis de points-to es un problema indecidible
  - ⇝ Permitimos obviar la verificación para casos en que falla
- Requerimos que el usuario provea todas las anotaciones de tiempo de vida para la verificación, y los espacios de iteración para utilizar la herramienta para aritmética no lineal
  - ⇝ En un futuro pensamos inferir estas anotaciones

1. Limitaciones, tanto por las herramientas usadas como inherentes a las técnicas propuestas
2. Como dependemos de otras herramientas (verificador, aritmética), siempre vamos a estar limitados
3. Análisis de tiempo de vida, siempre va a ser un problema
4. Muchas anotaciones requeridas, inferencia ayudaría

Limitaciones

- Estamos limitados por las capacidades de:
  - El verificador estático
  - La herramienta para resolver operaciones con aritmética no lineal
  - $\rightsquigarrow$ El diseño de las técnicas permite reemplazarlas por herramientas con mayores o mejores capacidades cuando existan
- El análisis de tiempo de vida es sobre-aproximado
  - $\rightsquigarrow$ El análisis de points-to es un problema indecidible
  - $\rightsquigarrow$ Permitimos obviar la verificación para casos en que falla
- Requerimos que el usuario provea todas las anotaciones de tiempo de vida para la verificación, y los espacios de iteración para utilizar la herramienta para aritmética no lineal
  - $\rightsquigarrow$ En un futuro pensamos inferir estas anotaciones

Limitaciones
- Estamos limitados por las capacidades de:
  - El verificador estático
  - La herramienta para resolver operaciones con aritmética no lineal
  - ⇝ El diseño de las técnicas permite reemplazarlas por herramientas con mayores o mejores capacidades cuando existan
- El análisis de tiempo de vida es sobre-aproximado
  - ⇝ El análisis de points-to es un problema indecidible
  - ⇝ Permitimos obviar la verificación para casos en que falla
- Requerimos que el usuario provea todas las anotaciones de tiempo de vida para la verificación, y los espacios de iteración para utilizar la herramienta para aritmética no lineal
  - ⇝ En un futuro pensamos inferir estas anotaciones

1. Limitaciones, tanto por las herramientas usadas como inherentes a las técnicas propuestas
2. Como dependemos de otras herramientas (verificador, aritmética), siempre vamos a estar limitados
3. Análisis de tiempo de vida, siempre va a ser un problema
4. Muchas anotaciones requeridas, inferencia ayudaría

Limitaciones

- Estamos limitados por las capacidades de:
  - El verificador estático
  - La herramienta para resolver operaciones con aritmética no lineal
  - $\rightsquigarrow$ El diseño de las técnicas permite reemplazarlas por herramientas con mayores o mejores capacidades cuando existan
- El análisis de tiempo de vida es sobre-aproximado
  - $\rightsquigarrow$ El análisis de points-to es un problema indecidible
  - $\rightsquigarrow$ Permitimos obviar la verificación para casos en que falla
- Requerimos que el usuario provea todas las anotaciones de tiempo de vida para la verificación, y los espacios de iteración para utilizar la herramienta para aritmética no lineal
  - $\rightsquigarrow$ En un futuro pensamos inferir estas anotaciones

Limitaciones
- Estamos limitados por las capacidades de:
  - El verificador estático
  - La herramienta para resolver operaciones con aritmética no lineal
  - ⇝ El diseño de las técnicas permite reemplazarlas por herramientas con mayores o mejores capacidades cuando existan
- El análisis de tiempo de vida es sobre-aproximado
  - ⇝ El análisis de points-to es un problema indecidible
  - ⇝ Permitimos obviar la verificación para casos en que falla
- Requerimos que el usuario provea todas las anotaciones de tiempo de vida para la verificación, y los espacios de iteración para utilizar la herramienta para aritmética no lineal
  - ⇝ En un futuro pensamos inferir estas anotaciones

1. Limitaciones, tanto por las herramientas usadas como inherentes a las técnicas propuestas
2. Como dependemos de otras herramientas (verificador, aritmética), siempre vamos a estar limitados
3. Análisis de tiempo de vida, siempre va a ser un problema
4. Muchas anotaciones requeridas, inferencia ayudaría

Limitaciones

- Estamos limitados por las capacidades de:
  - El verificador estático
  - La herramienta para resolver operaciones con aritmética no lineal
  - $\rightsquigarrow$ El diseño de las técnicas permite reemplazarlas por herramientas con mayores o mejores capacidades cuando existan
- El análisis de tiempo de vida es sobre-aproximado
  - $\rightsquigarrow$ El análisis de points-to es un problema indecidible
  - $\rightsquigarrow$ Permitimos obviar la verificación para casos en que falla
- Requerimos que el usuario provea todas las anotaciones de tiempo de vida para la verificación, y los espacios de iteración para utilizar la herramienta para aritmética no lineal
  - $\rightsquigarrow$ En un futuro pensamos inferir estas anotaciones

Limitaciones

- Estamos limitados por las capacidades de:
  - El verificador estático
  - La herramienta para resolver operaciones con aritmética no lineal
  - ⇝ El diseño de las técnicas permite reemplazarlas por herramientas con mayores o mejores capacidades cuando existan
- El análisis de tiempo de vida es sobre-aproximado
  - ⇝ El análisis de points-to es un problema indecidible
  - ⇝ Permitimos obviar la verificación para casos en que falla
- Requerimos que el usuario provea todas las anotaciones de tiempo de vida para la verificación, y los espacios de iteración para utilizar la herramienta para aritmética no lineal
  - ⇝ En un futuro pensamos inferir estas anotaciones

1. Limitaciones, tanto por las herramientas usadas como inherentes a las técnicas propuestas
2. Como dependemos de otras herramientas (verificador, aritmética), siempre vamos a estar limitados
3. Análisis de tiempo de vida, siempre va a ser un problema
4. Muchas anotaciones requeridas, inferencia ayudaría

Limitaciones

- Estamos limitados por las capacidades de:
  - El verificador estático
  - La herramienta para resolver operaciones con aritmética no lineal
  - $\rightsquigarrow$ El diseño de las técnicas permite reemplazarlas por herramientas con mayores o mejores capacidades cuando existan
- El análisis de tiempo de vida es sobre-aproximado
  - $\rightsquigarrow$ El análisis de points-to es un problema indecidible
  - $\rightsquigarrow$ Permitimos obviar la verificación para casos en que falla
- Requerimos que el usuario provea todas las anotaciones de tiempo de vida para la verificación, y los espacios de iteración para utilizar la herramienta para aritmética no lineal
  - $\rightsquigarrow$ En un futuro pensamos inferir estas anotaciones

Limitaciones

- Estamos limitados por las capacidades de:
  - El verificador estático
  - La herramienta para resolver operaciones con aritmética no lineal
  - ⇝ El diseño de las técnicas permite reemplazarlas por herramientas con mayores o mejores capacidades cuando existan
- El análisis de tiempo de vida es sobre-aproximado
  - ⇝ El análisis de points-to es un problema indecidible
  - ⇝ Permitimos obviar la verificación para casos en que falla
- Requerimos que el usuario provea todas las anotaciones de tiempo de vida para la verificación, y los espacios de iteración para utilizar la herramienta para aritmética no lineal
  - ⇝ En un futuro pensamos inferir estas anotaciones

1. Limitaciones, tanto por las herramientas usadas como inherentes a las técnicas propuestas
2. Como dependemos de otras herramientas (verificador, aritmética), siempre vamos a estar limitados
3. Análisis de tiempo de vida, siempre va a ser un problema
4. Muchas anotaciones requeridas, inferencia ayudaría

Limitaciones

- Estamos limitados por las capacidades de:
  - El verificador estático
  - La herramienta para resolver operaciones con aritmética no lineal
  - $\rightsquigarrow$ El diseño de las técnicas permite reemplazarlas por herramientas con mayores o mejores capacidades cuando existan
- El análisis de tiempo de vida es sobre-aproximado
  - $\rightsquigarrow$ El análisis de points-to es un problema indecidible
  - $\rightsquigarrow$ Permitimos obviar la verificación para casos en que falla
- Requerimos que el usuario provea todas las anotaciones de tiempo de vida para la verificación, y los espacios de iteración para utilizar la herramienta para aritmética no lineal
  - $\rightsquigarrow$ En un futuro pensamos inferir estas anotaciones

Limitaciones

- Estamos limitados por las capacidades de:
  - El verificador estático
  - La herramienta para resolver operaciones con aritmética no lineal
  - ⇝ El diseño de las técnicas permite reemplazarlas por herramientas con mayores o mejores capacidades cuando existan
- El análisis de tiempo de vida es sobre-aproximado
  - ⇝ El análisis de points-to es un problema indecidible
  - ⇝ Permitimos obviar la verificación para casos en que falla
- Requerimos que el usuario provea todas las anotaciones de tiempo de vida para la verificación, y los espacios de iteración para utilizar la herramienta para aritmética no lineal
  - ⇝ En un futuro pensamos inferir estas anotaciones

1. Limitaciones, tanto por las herramientas usadas como inherentes a las técnicas propuestas
2. Como dependemos de otras herramientas (verificador, aritmética), siempre vamos a estar limitados
3. Análisis de tiempo de vida, siempre va a ser un problema
4. Muchas anotaciones requeridas, inferencia ayudaría

Trabajo futuro

- Incorporar capacidades de inferencia permitiendo al usuario proveer sólo los contratos de consumo y no las anotaciones adicionales actualmente requeridas
  - De las anotaciones de tiempo de vida
  - De los espacios de iteración de los ciclos
- Experimentar con otros verificadores estáticos
  - ESC/Java2 para Java: requeriría implementar la instrumentación para Java
  - Z3 para Java o .NET: requeriría traducir el código a un lenguaje intermedio que utiliza (Boogie)
- Extender las capacidades del análisis con aritmética no lineal
  - Mejorar la capacidad de cálculo de máximos entre polinomios
  - Evaluar/modificar la herramienta utilizada o reemplazarla por otra
- Mejorar la usabilidad e integración con la IDE
  - Desarrollando un plug-in que autocomplete anotaciones de acuerdo a los contratos existentes

Trabajo futuro
- Incorporar capacidades de inferencia permitiendo al usuario proveer sólo los contratos de consumo y no las anotaciones adicionales actualmente requeridas
  - De las anotaciones de tiempo de vida
  - De los espacios de iteración de los ciclos
- Experimentar con otros verificadores estáticos
  - ESC/Java2 para Java: requeriría implementar la instrumentación para Java
  - Z3 para Java o .NET: requeriría traducir el código a un lenguaje intermedio que utiliza (Boogie)
- Extender las capacidades del análisis con aritmética no lineal
  - Mejorar la capacidad de cálculo de máximos entre polinomios
  - Evaluar/modificar la herramienta utilizada o reemplazarla por otra
- Mejorar la usabilidad e integración con la IDE
  - Desarrollando un plug-in que autocomplete anotaciones de acuerdo a los contratos existentes

1. Podemos integrar otros trabajos sobre inferencia
2. Hay muchos verificadores, Z3 podría integrarse sin ningún trabajo cuando Code Contracts lo soporte
3. Ya desarrollamos un plugin para instalar el tool como ext de VS y habilitar/deshabilitar contratos de mem, la idea es extenderlo para asistencia de anotaciones

Trabajo futuro

- Incorporar capacidades de inferencia permitiendo al usuario proveer sólo los contratos de consumo y no las anotaciones adicionales actualmente requeridas
  - De las anotaciones de tiempo de vida
  - De los espacios de iteración de los ciclos
- Experimentar con otros verificadores estáticos
  - `ESC/Java2` para Java: requeriría implementar la instrumentación para Java
  - Z3 para Java o .NET: requeriría traducir el código a un lenguaje intermedio que utiliza (`Boogie`)
- Extender las capacidades del análisis con aritmética no lineal
  - Mejorar la capacidad de cálculo de máximos entre polinomios
  - Evaluar/modificar la herramienta utilizada o reemplazarla por otra
- Mejorar la usabilidad e integración con la IDE
  - Desarrollando un plug-in que autocomplete anotaciones de acuerdo a los contratos existentes

Trabajo futuro
- Incorporar capacidades de inferencia permitiendo al usuario proveer sólo los contratos de consumo y no las anotaciones adicionales actualmente requeridas
  - De las anotaciones de tiempo de vida
  - De los espacios de iteración de los ciclos
- Experimentar con otros verificadores estáticos
  - ESC/Java2 para Java: requeriría implementar la instrumentación para Java
  - Z3 para Java o .NET: requeriría traducir el código a un lenguaje intermedio que utiliza (Boogie)
- Extender las capacidades del análisis con aritmética no lineal
  - Mejorar la capacidad de cálculo de máximos entre polinomios
  - Evaluar/modificar la herramienta utilizada o reemplazarla por otra
- Mejorar la usabilidad e integración con la IDE
  - Desarrollando un plug-in que autocomplete anotaciones de acuerdo a los contratos existentes

1. Podemos integrar otros trabajos sobre inferencia
2. Hay muchos verificadores, Z3 podría integrarse sin ningún trabajo cuando Code Contracts lo soporte
3. Ya desarrollamos un plugin para instalar el tool como ext de VS y habilitar/deshabilitar contratos de mem, la idea es extenderlo para asistencia de anotaciones

Trabajo futuro

- Incorporar capacidades de inferencia permitiendo al usuario proveer sólo los contratos de consumo y no las anotaciones adicionales actualmente requeridas
  - De las anotaciones de tiempo de vida
  - De los espacios de iteración de los ciclos
- Experimentar con otros verificadores estáticos
  - `ESC/Java2` para Java: requeriría implementar la instrumentación para Java
  - Z3 para Java o .NET: requeriría traducir el código a un lenguaje intermedio que utiliza (`Boogie`)
- Extender las capacidades del análisis con aritmética no lineal
  - Mejorar la capacidad de cálculo de máximos entre polinomios
  - Evaluar/modificar la herramienta utilizada o reemplazarla por otra
- Mejorar la usabilidad e integración con la IDE
  - Desarrollando un plug-in que autocomplete anotaciones de acuerdo a los contratos existentes

Trabajo futuro
- Incorporar capacidades de inferencia permitiendo al usuario proveer sólo los contratos de consumo y no las anotaciones adicionales actualmente requeridas
  - De las anotaciones de tiempo de vida
  - De los espacios de iteración de los ciclos
- Experimentar con otros verificadores estáticos
  - ESC/Java2 para Java: requeriría implementar la instrumentación para Java
  - Z3 para Java o .NET: requeriría traducir el código a un lenguaje intermedio que utiliza (Boogie)
- Extender las capacidades del análisis con aritmética no lineal
  - Mejorar la capacidad de cálculo de máximos entre polinomios
  - Evaluar/modificar la herramienta utilizada o reemplazarla por otra
- Mejorar la usabilidad e integración con la IDE
  - Desarrollando un plug-in que autocomplete anotaciones de acuerdo a los contratos existentes

1. Podemos integrar otros trabajos sobre inferencia
2. Hay muchos verificadores, Z3 podría integrarse sin ningún trabajo cuando Code Contracts lo soporte
3. Ya desarrollamos un plugin para instalar el tool como ext de VS y habilitar/deshabilitar contratos de mem, la idea es extenderlo para asistencia de anotaciones

Trabajo futuro

- Incorporar capacidades de inferencia permitiendo al usuario proveer sólo los contratos de consumo y no las anotaciones adicionales actualmente requeridas
  - De las anotaciones de tiempo de vida
  - De los espacios de iteración de los ciclos
- Experimentar con otros verificadores estáticos
  - `ESC/Java2` para Java: requeriría implementar la instrumentación para Java
  - Z3 para Java o .NET: requeriría traducir el código a un lenguaje intermedio que utiliza (`Boogie`)
- Extender las capacidades del análisis con aritmética no lineal
  - Mejorar la capacidad de cálculo de máximos entre polinomios
  - Evaluar/modificar la herramienta utilizada o reemplazarla por otra
- Mejorar la usabilidad e integración con la IDE
  - Desarrollando un plug-in que autocomplete anotaciones de acuerdo a los contratos existentes

Trabajo futuro
- Incorporar capacidades de inferencia permitiendo al usuario proveer sólo los contratos de consumo y no las anotaciones adicionales actualmente requeridas
  - De las anotaciones de tiempo de vida
  - De los espacios de iteración de los ciclos
- Experimentar con otros verificadores estáticos
  - ESC/Java2 para Java: requeriría implementar la instrumentación para Java
  - Z3 para Java o .NET: requeriría traducir el código a un lenguaje intermedio que utiliza (Boogie)
- Extender las capacidades del análisis con aritmética no lineal
  - Mejorar la capacidad de cálculo de máximos entre polinomios
  - Evaluar/modificar la herramienta utilizada o reemplazarla por otra
- Mejorar la usabilidad e integración con la IDE
  - Desarrollando un plug-in que autocomplete anotaciones de acuerdo a los contratos existentes

1. Podemos integrar otros trabajos sobre inferencia
2. Hay muchos verificadores, Z3 podría integrarse sin ningún trabajo cuando Code Contracts lo soporte
3. Ya desarrollamos un plugin para instalar el tool como ext de VS y habilitar/deshabilitar contratos de mem, la idea es extenderlo para asistencia de anotaciones

Índice

Conclusiones

- Presentamos un conjunto de algoritmos y técnicas para verificar el consumo de memoria de un programa
  - Haciendo uso de las capacidades de análisis de un verificador estático
  - Integrando herramientas externas para incrementar las capacidades de análisis
- Implementamos un prototipo
  - Para .NET, usando el verificador estático de Code Contracts
  - Con integración en la IDE
- Evaluamos su uso bajo diferentes condiciones
  - Identificando las limitaciones
- Creemos que la solución ideal debe hacer un uso combinado de inferencia y verificación
  - Inferir anotaciones y contratos fácilmente deducibles
  - Verificar contratos complejos anotados por el usuario
- El trabajo presentado es un buen punto de partida para una solución utilizable en un entorno real para obtener un certificado del consumo de memoria

Conclusiones

1. Ideal: inferir lo fácil, verificar lo dificíl que anota el usuario
2. Bueno punto de partida para seguir trabajando

Conclusiones

- Presentamos un conjunto de algoritmos y técnicas para verificar el consumo de memoria de un programa
  - Haciendo uso de las capacidades de análisis de un verificador estático
  - Integrando herramientas externas para incrementar las capacidades de análisis
- Implementamos un prototipo
  - Para .NET, usando el verificador estático de Code Contracts
  - Con integración en la IDE
- Evaluamos su uso bajo diferentes condiciones
  - Identificando las limitaciones
- Creemos que la solución ideal debe hacer un uso combinado de inferencia y verificación
  - Inferir anotaciones y contratos fácilmente deducibles
  - Verificar contratos complejos anotados por el usuario
- El trabajo presentado es un buen punto de partida para una solución utilizable en un entorno real para obtener un certificado del consumo de memoria

Conclusiones

1. Ideal: inferir lo fácil, verificar lo dificíl que anota el usuario
2. Bueno punto de partida para seguir trabajando

Conclusiones

- Presentamos un conjunto de algoritmos y técnicas para verificar el consumo de memoria de un programa
  - Haciendo uso de las capacidades de análisis de un verificador estático
  - Integrando herramientas externas para incrementar las capacidades de análisis
- Implementamos un prototipo
  - Para .NET, usando el verificador estático de Code Contracts
  - Con integración en la IDE
- Evaluamos su uso bajo diferentes condiciones
  - Identificando las limitaciones
- Creemos que la solución ideal debe hacer un uso combinado de inferencia y verificación
  - Inferir anotaciones y contratos fácilmente deducibles
  - Verificar contratos complejos anotados por el usuario
- El trabajo presentado es un buen punto de partida para una solución utilizable en un entorno real para obtener un certificado del consumo de memoria

Conclusiones

- Presentamos un conjunto de algoritmos y técnicas para verificar el consumo de memoria de un programa
  - Haciendo uso de las capacidades de análisis de un verificador estático
  - Integrando herramientas externas para incrementar las capacidades de análisis
- Implementamos un prototipo
  - Para .NET, usando el verificador estático de Code Contracts
  - Con integración en la IDE
- Evaluamos su uso bajo diferentes condiciones
  - Identificando las limitaciones
- Creemos que la solución ideal debe hacer un uso combinado de inferencia y verificación
  - Inferir anotaciones y contratos fácilmente deducibles
  - Verificar contratos complejos anotados por el usuario
- El trabajo presentado es un buen punto de partida para una solución utilizable en un entorno real para obtener un certificado del consumo de memoria

1. Ideal: inferir lo fácil, verificar lo dificíl que anota el usuario
2. Bueno punto de partida para seguir trabajando

Conclusiones

- Presentamos un conjunto de algoritmos y técnicas para verificar el consumo de memoria de un programa
  - Haciendo uso de las capacidades de análisis de un verificador estático
  - Integrando herramientas externas para incrementar las capacidades de análisis
- Implementamos un prototipo
  - Para .NET, usando el verificador estático de Code Contracts
  - Con integración en la IDE
- Evaluamos su uso bajo diferentes condiciones
  - Identificando las limitaciones
- Creemos que la solución ideal debe hacer un uso combinado de inferencia y verificación
  - Inferir anotaciones y contratos fácilmente deducibles
  - Verificar contratos complejos anotados por el usuario
- El trabajo presentado es un buen punto de partida para una solución utilizable en un entorno real para obtener un certificado del consumo de memoria

Conclusiones

1. Ideal: inferir lo fácil, verificar lo dificíl que anota el usuario
2. Bueno punto de partida para seguir trabajando

Conclusiones

- Presentamos un conjunto de algoritmos y técnicas para verificar el consumo de memoria de un programa
  - Haciendo uso de las capacidades de análisis de un verificador estático
  - Integrando herramientas externas para incrementar las capacidades de análisis
- Implementamos un prototipo
  - Para .NET, usando el verificador estático de Code Contracts
  - Con integración en la IDE
- Evaluamos su uso bajo diferentes condiciones
  - Identificando las limitaciones
- Creemos que la solución ideal debe hacer un uso combinado de inferencia y verificación
  - Inferir anotaciones y contratos fácilmente deducibles
  - Verificar contratos complejos anotados por el usuario
- El trabajo presentado es un buen punto de partida para una solución utilizable en un entorno real para obtener un certificado del consumo de memoria

Conclusiones

- Presentamos un conjunto de algoritmos y técnicas para verificar el consumo de memoria de un programa
  - Haciendo uso de las capacidades de análisis de un verificador estático
  - Integrando herramientas externas para incrementar las capacidades de análisis
- Implementamos un prototipo
  - Para .NET, usando el verificador estático de Code Contracts
  - Con integración en la IDE
- Evaluamos su uso bajo diferentes condiciones
  - Identificando las limitaciones
- Creemos que la solución ideal debe hacer un uso combinado de inferencia y verificación
  - Inferir anotaciones y contratos fácilmente deducibles
  - Verificar contratos complejos anotados por el usuario
- El trabajo presentado es un buen punto de partida para una solución utilizable en un entorno real para obtener un certificado del consumo de memoria

1. Ideal: inferir lo fácil, verificar lo dificíl que anota el usuario
2. Bueno punto de partida para seguir trabajando

Fin ■

¿Preguntas?