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

Slicing dinámico para lenguajes de la familia .NET

Tesis de Licenciatura en Ciencias de la Computación

Federico De Rocco

Director: Alexis Soifer

Buenos Aires, 2020

Slicing dinámico para lenguajes de la familia .NET

Slicing dinámico es una técnica que se utiliza principalmente para facilitar la tarea de debugging[2], reduciendo el esfuerzo del programador encontrando aquellas líneas relevantes para un comportamiento dado[5]. Para construir un slice el enfoque clásico se basa en la búsqueda de dependencias entre las instrucciones del programa. Si bien en sus comienzos la técnica fue prometedora, hoy en día todavía persisten problemas de escalabilidad[30]. El grupo LaFHIS desarrolló en los últimos años **DynAbs**, un slicer de emisión/análisis de traza destinado a analizar código de alto nivel que intenta atacar parcialmente estos problemas de escalabilidad para aplicaciones C#. A pesar de los avances, la herramienta no ataca una de las deficiencias que tienen las herramientas de este tipo, que es la limitación de tratar un único lenguaje.

En este trabajo se presenta una extensión para la herramienta para operar sobre el lenguaje Visual Basic, el cual actualmente no posee ninguna herramienta de slicing compatible exceptuando herramientas multilenguaje como ORBS [27, 28].
Como cualquier herramienta de este tipo, esta consiste en una parte destinada a instrumentar el código cliente y otra parte consumir la traza generada en su ejecución. Por lo tanto este trabajo consistió en implementar por un lado la funcionalidad necesaria para instrumentar código Visual Basic, y por otro, abstraer el módulo de análisis para que trate ambos lenguajes. A lo largo de este trabajo se detallan los conceptos relativos a slicing, los específicos a DynAbs y se explica cómo fue realizada esta extensión. Finalmente, se presentará una evaluación adecuada que determinó que han sido cubiertas suficientes expresiones del lenguaje y que efectivamente ahora se pueden tratar programas que han sido evaluados previamente en C#.

Palabras claves: Program Analysis, Program Slicing, Dynamic Slicing, Tracing

Dynamic Slicing for .NET family languages

Dynamic slicing is a technique that is mainly used by the programmers on debugging[2], reducing their efforts by finding those lines that are relevant to a given behavior[5]. To build a slice, the classic approach is based on finding control and data dependencies between the statements. Although the technique was promising in its beginnings, today persists scalability problems[30]. LaFHIS has developed in recent years **DynAbs**, a classical slicer designed for tracing high-level code that tries to partially solve these scalability problems for C # applications. Despite the advances, the tool does not attack one of the deficiencies that tools of this type have, which is the limitation of dealing with a single language.

This work presents an extension for the tool to operate on the Visual Basic language, which currently does not have any compatible slicing tool except for multilanguage ones such as ORBS [27, 28].
Like any tool of this type, this consists of instrumenting the client source code and consuming the generated trace in its execution. Therefore, this work consists on implementing the tracing module for Visual Basic, and on the other hand, abstracting the analysis module to deal with both languages. In this work are detailed concepts related to slicing, those specific to DynAbs, and how this extension was developed. Finally, an adequate evaluation will be presented. It determined that several expressions of the language have been covered and that programs that have been previously evaluated in C # can be also treated nowadays.

Keywords: Program Analysis, Program Slicing, Dynamic Slicing, Tracing

Índice general

1. INTRODUCCIÓN

Entre las tantas problemáticas presentes en la Ingeniería del Software se encuentran aquellas destinadas simplificar el desarrollo de programas. En este marco podemos destacar metodologías para la detección de errores presentes en el código, lo cual es fundamental para alcanzar la funcionalidad deseada. Si bien hay líneas de investigación destinadas a atacar esta problemática desde el punto de vista *estático*, es decir, sin ejecutar los programas, existe además (por diversas causas) la necesidad de comprender el comportamiento en tiempo de ejecución. Esto lo podemos observar en base a estudios sobre las preguntas que se hacen los desarrolladores [31], además de algunas incipientes técnicas que vienen desarrollandosé los últimos años [32, 33]. Para esta actividad se debe tener en cuenta que una pieza de software posee una complejidad cognitiva, muchas veces independiente del problema en si, que dificulta el estudio de la misma. Simplificar esta actividad es una necesidad para cubrir más eficientemente los casos simples y, a su vez, poder adentrarse en los más complicados con soltura. Entre las soluciones prácticas que han cobrado más fuerza en este último tiempo se encuentra slicing dinámico [5].

Slicing, a modo general, es un método usado por programadores para abstraer el funcionamiento de programas. A partir de un subconjunto del comportamiento de un programa, se construye una versión lo más reducida posible, idealmente mínima con respecto a la cantidad de lineas, tal que el comportamiento original se mantenga. Este programa, llamado *slice*, tiene la cualidad de ser una representación del programa original para ese comportamiento específico. Comúnmente, se denota un conjunto de los valores para un grupo de variables sobre una linea de código como el comportamiento mencionado. Se conoce como slicing *estático* [1] el proceso de producir un slice sin ejecutar el programa, mientras que slicing dinámico es cuando realizamos este proceso recurriendo a la ejecución del programa, fijando los inputs del mismo. Una condición necesaria para resolver adecuadamente los defectos en un programa es comprender su comportamiento, particularmente el que se manifiesta en el error. Este ejercicio; sin embargo, resulta complejo. Es por eso que se entiende menester construir herramientas que favorezcan tanto la comprensión como la búsqueda de defectos. Las aplicaciones prácticas de esta técnica incluyen debugging [2], análisis de impacto [3], testing [4], etc.

Los slicers son las herramientas producen los slices mencionados previamente, analicemos por un momento cual es el estado del arte en cuanto a limitaciones actuales.En el caso de los slicers estáticos si bien existen limitaciones de escala, en la práctica sus limitaciones van por el lado de la precisión. Por otro lado, como toda herramienta de análisis dinámico de código, la escalabilidad de los slicers dinámico es un tema no trivial y esta lejos de resolverse hoy en día [30]. Más allá de las limitaciones técnicas también hay defectos colaterales que hoy en día no son fáciles de resolver, entre ellos, las herramientas suelen estar fuertemente acopladas al lenguaje que analicen. El slicer dinámico del grupo LaFHIS, **DynAbs**, fue desarrollado con la intención de atacar limitaciones de escalabilidad, entraremos en detalle más adelante sobre este punto. Más allá de su propósito original, una observación secundaria fue que era posible sortear otra de las limitaciones comunes para los slicers, el soporte de un único lenguaje. En general estas herramientas suelen im-

plementarse para operar sobre un único lenguaje de programación. Tales son los casos de WET [6, 7, 8] para C o JSlice [9, 10] para Java. Si bien esta limitación se encontraba también en DynAbs, el cual previamente solo brindaba soporte para C#, el diseño del mismo propensó una idea: *abstraer la técnica original utilizando una abstracción del lenguaje que permita que el analizador de código trabaje para más de un lenguaje.*En teoría, agregando un componente puntual que se relaciona directamente con el lenguaje, se podría reutilizar el núcleo de procesamiento para que trate Visual Basic. Esto se debió en gran parte al soporte compartido entre ambos lenguajes, ya que al ser ambos de Microsoft se podían reutilizar las librerías de análisis de código de C# en VB.El aporte de este trabajo fue entonces, el de extender este slicer particular para que soporte Visual Basic. Adicionalmente, se presentaran experimentos para evaluar la efectividad de la extensión realizada.

2. BACKGROUND

2.1. Program Slicing

Inicialmente formulado por Mark Weiser [1] como una técnica de análisis estático. La idea general es reconocer el comportamiento de interés del programa P como un criterio expresado como una tupla (s_i, V), con s_i como el statement i de P, y V un subconjunto de variables de P. A partir de esta definición se calcula el slice S sobre el P, $S \subseteq P$ tal que el valor de V en s_i en S es el mismo que en P para cualquier input.

Una de las técnicas para obtener S se basa en dataflow análisis [12], esta consiste en realizar un corte sobre P incluyendo a los statements de los cuales dependen los cálculos de nuestras variables. Se definen dos tipos de dependencias, la *dependencia de datos* se produce cuando se utilizan variables en un statement s_i cuyo último punto de definición fue en un statement s_j, mientras que la *dependencia de control* se produce hacia aquellos statements de control (if, while, etc.) que determinaron el flujo de la aplicación hacia estos statements.

Incluyendo de forma iterativa a los statements que influyen sobre las variables V en el punto s_i obtenemos un recorte del programa $S \subseteq P$ tal que satisface los requerimientos planteados para un slice. Desde ya, esto no asegura minimalidad, el cual de hecho es un problema considerado *intratable*.

2.2. Program Dependence Graph

El *Program Dependence Graph* (PDG) [13] es la representación de las dependencias de datos y de control, donde cada nodo representa un statement, los ejes representan las dependencias de control y de datos.

Dado un programa P donde G es su PDG, podemos obtener su slice usando el algoritmo BFS para recorrer los nodos alcanzables desde el nodo que representa el statement.

2.3. Dynamic Slicing

Si bien se realizaron muchas investigaciones sobre la temática, las limitaciones del slicing estático planteado por Weiser no tardaron en aparecer. Al exigir que el comportamiento del programa se conserve para todos los posibles inputs muchas veces se termina obteniendo slices que son de un tamaño muy similar al del programa original. Para poder suplir dichas limitaciones Korel y Laski introdujeron el concepto de slicing dinámico [5].

El slicing dinámico consiste en utilizar un input o test puntual como parte de la definición del criterio (X, s_i^k, V), donde X es el conjunto de inputs del programa, s_i es el statement i, k es la k-ésima aparición de la linea y V es el conjunto de variables que se desea analizar.

Esta forma de enfrentar el problema permite obtener slices más precisas a coste de pérdida de generalidad. Esto último se debe a que, evidentemente, se esta limitando más el com-

portamiento reconocido al introducir este nuevo elemento, en comparación con su versión estática que permitía reconocer el comportamiento independientemente del input. Bajo está técnica será necesario poder analizar la ejecución del programa y extraer información sobre este.

2.4. Backward y Forward Slicing

Existen dos formas de recorrer un programa para un criterio dado, para atrás (backward) o hacia adelante (forward). En resumidas cuentas, backward intenta encontrar todos los statements que afectan al criterio, forward por el contrario, busca todos los statements afectados por el criterio.

Originalmente, la definición de Weiser corresponde a backward slicing. Supongamos que tenemos un statement, el cual presenta algún problema en una de sus variables, backward slicing nos permitiría obtener todos los statements que influyeron sobre esa variable. Para el caso de forward, si tenemos un statement que va a ser modificado eventualmente, el conjunto de statements resultado nos muestra donde influirán los cambios de la variable. Los primeros en definir forward slicing fueron Bergeretti y Carre [15].

2.5. Dynamic Dependence Graph

Al igual que en su versión estática, el slicing dinámico puede sacar provecho del PDG para construirse. Agrawal y Horgan [14] propusieron cuatro enfoques para lograrlo (llamados **AH-1**, **AH-2**, **AH-3** y **AH-4**). El PDG construido es llamado Dynamic Dependence Graph (DDG).

2.6. Points-to Graph

Un points-to graph (PTG) es un modelo de la memoria, comúnmente representado como un grafo en el cual cada nodo representa cero, uno o varios nodos en la memoria. Cada objeto alocado en la memoria debería tener su representación en el PTG, mientras que cada relación entre los nodos reales de la memoria tendría que tener su representación en el modelo visualizada a través de sus ejes. Introduciremos como es utilizado este modelo por DynAbs más adelante.

2.7. Tracing

Tracing es una metodología utilizada para recolectar información sobre la ejecución de un programa mediante el uso de logging, es decir, generar logs dentro de la ejecución del programa. Este resulta de gran utilidad para obtener los slices dinámicos, ya que es fundamental para la su cómputo conocer los statements que se ejecutan en el programa, a diferencia de su versión estática que no requiere ejecutar el programa. Con esto en mente, la información generada por el tracing (llamada traza) serán los statements ejecutados durante la corrida.

Para poder asegurarnos de obtener la traza buscada cuando ejecutemos el programa, será necesario someterlo a un proceso de *instrumentación*. Este consiste en insertar el código necesario para producir logs al programa original. En nuestro caso, el objetivo de este proceso es que pueda emitir la traza ya mentada para ser posteriormente analizada.

2.8. Abstract Syntax Tree

Un *Abstract Syntax Tree* (AST), también llamado árbol sintáctico, es una representación como árbol de una estructura sintáctica abstracta del código de un programa. Cada nodo del árbol denota una construcción que ocurre en el código (por ejemplo, la declaración de una variable o un bloque if), normalmente se los llama SyntaxNode.

Durente el proceso de instrumentación, necesitamos recorrer el syntax tree del programa cliente para agregar el código necesario para producir los logs. Es crucial entonces para poder instrumentar el código, poder contar con un compilador y herramientas suficientes de análisis de código para el lenguaje seleccionado.

3. DYNABS

En este capítulo presentaremos una breve introducción a **DynAbs**, el slicer desarrollado por el grupo LaFHIS para el lenguaje C#. Motivaciones teóricas y de diseño adoptadas en el desarrollo de esta herramienta quedan fuera del scope de esta tesis.

3.1. Funcionamiento general

DynAbs es un slicer que implementa una técnica que actualmente está desarrollando el grupo llamada *Abstract Dynamic Slicing*. Esta técnica se basa en el esquema tradicional de producción/análisis de traza pero difiere en un aspecto fundamental: en lugar de analizar direcciones y valores concretos de la memoria, esta se la modela utilizando una representación abstracta de la memoria, que en este caso fue implementada con points-to graph adaptado al contexto del problema.

La motivación principal de este enfoque es reducir varios órdenes de magnitud el tamaño de la traza y acelerar el análisis a costa de pérdida de precisión. Como consecuencia, para reducir la pérdida de precisión, cuando nos encontramos ante la presencia de calls a código no instrumentado actualizamos nuestras estructuras de datos (modelo de memoria, grafo de dependencias) utilizando especificaciones que indican *que sucede* en estos métodos.

Lamentablemente en caso de no contar con una especificación apropiada se procederá a actualizar nuestras estructuras considerando el peor escenario posible que consiste en reproducir todas las lecturas y escrituras que pudieron haber sido realizadas en el callee.

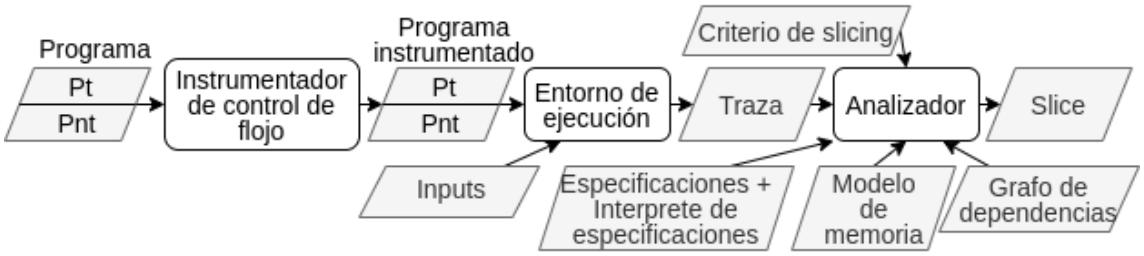

Fig. 3.1: Arquitectura de DynAbs

En la figura 3.1 recibimos el programa cliente el cual podemos diferenciar en 2 partes, *Pt* es el programa a instrumentar mientras que *Pnt* es la parte del programa que no instrumentamos, esta consiste en las librerías externas o puede contener componentes del código cliente que deseamos evitar.

Luego se procede a instrumentar *Pt* de tal forma que al ejecutarlo produzca una traza que solo servirá para seguir el flujo de la ejecución del programa cliente. Notar nuevamente que no hay direcciones de memoria concretas.

Una vez producida la traza el *analizador*, componente que se encargará de procesarla, utiliza principalmente 3 estructuras internas: el modelo de memoria, el grafo de dependencias, y las especificaciones.

El *modelo de memoria* se encargará de mantener una abstracción tal que cada objeto concreto se encuentre representado en un único objeto de la abstracción, y que cada access path (variable + secuencia de fields) en la memoria concreta mapee a los mismos objetos que su contraparte abstracta.

La implementación se realizó con un points-to graph, y esta consiste en mantener un scope con sus respectivas variables para cada método que se está analizando y un grafo asociado donde cada nodo mapea a uno o más nodos concretos y cada eje representa una referencia entre objetos en el heap real. Una formalización de esta abstracción y su implementación es trabajo de investigación del grupo y no será profundizado en este trabajo.

Por otro lado el grafo de dependencias en esta implementación está construido utilizando el approach 4 del trabajo de Agrawal y Horgan [14]. Este enfoque ataca el problema de escalabilidad del approach 3, para que en lugar de agregar un nodo en el grafo por cada nuevo statement analizado, cree un nuevo nodo cuando la ocurrencia de un statement no comparta las mismas dependencias transitivas que la ocurrencia anterior.

Las especificaciones por otro lado tienen el propósito de evitar los peores casos en el momento de replicar en nuestras estructuras que sucede cuando no se genera traza asociada a la ejecución de un método. Estas son constituidas en base a un lenguaje desarrollado especialmente para esta herramienta. La información que provee cada anotación dice que objetos son alocados en memoria, y cada *posible* definición/uso de los fields alcanzables desde el callee. El intérprete por otro lado *entiende* estas especificaciones y actualiza las estructuras acordemente. En el caso de que no exista especificación asociada a un método, entonces se asume el peor caso: este es que todos los objetos accesibles por este código son leídos y escritos, y por lo tanto, todas las conexiones posibles entre estos objetos se puede llegar a efectuar (mediante asignaciones).

Durante el análisis el analizador realiza la misma operatoria que el algoritmo tradicional de slicing manteniendo una tabla de *últimas definiciones* para cada una de las variables o fields de cada objeto. La diferencia en este enfoque radica que en vez de tener una entrada con una dirección concreta de memoria se tiene una entrada *simbólica* compuesta por el scope-variable u objeto-field al que se hace referencia en el modelo de memoria. Una vez completado el análisis, el slice se obtiene a partir del grafo de dependencias construido.

Profundizaremos más en explicar como se instrumenta el programa cliente y las características de la traza producida ya que estos puntos serán claves para entender el trabajo realizado en esta tesis.

3.2. Instrumentación y generación de traza

El proceso de instrumentación consiste en tomar como input al programa original y retornar un programa semánticamente equivalente, que al ejecutarse emita información suficiente que permita seguir el flujo de la ejecución refiriendo a fragmentos de código del programa original.
Para realizar la emisión de traza se asocia un módulo de DynAbs al programa instrumentado que provee una API de emisión de traza que llamamos **Tracer**. Esta API contiene métodos que facilitan el embebido de los calls en el programa instrumentado para la generación de la traza. Diversos detalles de implementación serán presentados en las siguientes secciones.

3.2.1. Claves de la instrumentación

La instrumentación del código se puede realizar tanto a low level como a high level. El grupo decidió enfocar la instrumentación hacia high-level con el fin de reducir el tamaño de la traza emitida. Esto produce que hay que insertar los llamados al tracer de forma muy precisa, ya que no es sencillo seguir con precisión el flujo de la ejecución. Analicemos en que situaciones se agrega traza y porque:

- Propiedades sin código asociado son tratadas como campos.
- Las invocaciones con encapsuladas para poder registrar información sobre el retorno.
- Operaciones lógicas con encapsuladas para poder registrar más información sobre ellas, incluyendo cuando no necesitamos esperar un traza por su parte.
- Los métodos también son encapsulados para poder tratar las excepciones.
- A todas las expresiones que contengan bloques de código (if, while, etc) se les agrega trazas que marcan el comienzo y terminación de los mismos.

3.2.2. Tipos de traza aceptados y traza emitida

Cada traza hace referencia a una porción de código particular, por lo tanto se necesita conocer *el archivo* y la parte del mismo a la que se hace referencia. Esto se logra identificando a cada archivo con un ID y nombrando a cada parte del código mediante un rango desde/hasta de caracteres.

Por otro lado al haber distintos *tipos de traza* (que serán presentados a continuación), cada línea de la traza estará compuesta por una tupla $< FID, SS, SE, TT >$ donde: FID es el identificador del archivo, SS y SE los índices de la posición del primer/último carácter dentro del mismo y TT el cual señala el tipo de la traza.

Presentamos entonces que tipos de traza se emiten en la herramienta y la motivación de cada uno:

- TraceSimpleStatement: es utilizada para marcar un statement particular. Siempre antecede el statement sobre el que opera.

- TraceEnterCondition, TraceEnterMethod, TraceEnterConstructor, TraceEnterStaticMethod, TraceEnterCatch, TraceEnterFinally, TraceEnterFinalCatch, TraceEnterFinalFinally, TraceEnterStaticConstructor y TraceEnterExpression: son utilizados para marcar el ingreso a un determinado tipo de bloque de código.
- TraceExitCondition, TraceExitLoop, TraceExitLoopByException, TraceExitUsing, TraceExitMethod, TraceExitConstructor, TraceExitStaticMethod, TraceExitCatch, TraceExitFinally y TraceExitStaticConstructor: son utilizados para marcar el egreso de un determinado tipo de bloque de código.
- TraceBreak: similar a TraceSimpleStatement, en este caso es usado para marcar exclusivamente los statements break.
- TraceBeforeConstructor: marca el instante previo a al declaración de los constructores de una clase.
- TraceInvocationWrapper: se utiliza para encapsular todo tipo de invocaciones, para ver si se producen callbacks. Esto incluye a expresiones binarias. Ejemplo: "abc" + $variable$.
- TraceInitializationWrapper: se usa para encapsular la asignación de una propiedad en un ObjectCreationExpressionSyntax (es el llamado a un constructor). Un ejemplo de esto es: $newCasa\{propietario = "Christian", direccion = "unaDireccion"\}$. En este caso se encapsularía tanto "$Christian$", como "$unaDireccion$".
- TraceMemberAccessWrapper: encapsula todo tipo de accesos a propiedad. Ejemplo: $base.val$
- TraceInitialMemberAccessWrapper: encapsula la expresión de una asignación sobre una propiedad de un objeto. Ejemplo: $this.b+ = other.b$.
- TraceConditionalAccessExpression: encapsula expresiones del tipo $A?.B$, las son equivalentes a $(A == null?null : A.B)$.
- TraceBaseCall, TraceEndInvocation y TraceEndMemberAccess: sirven para marcar el final de la ejecución de una función creada por el instrumentador previamente para poder analizar la ejecución de un statement particular, el cual requiera esto. Se utiliza TraceEndMemberAccess, en lugar de TraceEndInvocation, si se trata de un acceso a una propiedad. TraceBaseCall es utilizado exclusivamente para el llamado al constructor base en la declaración de un constructor que lo utilice.

Es importante notar que los tipos de traza se pueden separar en dos grupos, los que se dedican a marcar un statement o una posición relativa a otros statements, y los que encapsulan expresiones. De los primeros se entiende su propósito como el de anticipar que traza se espera y brindar la capacidad de extraer toda la información comentada arriba, para cada expresión en el análisis. Los segundos cumplen con el mismo propósito pero requieren otras consideraciones. El problema radica que estas expresión pueden no ejecutarse en el flujo del programa, incluso si se pasa por la linea que los contiene; de hacerlo se debe poder verificar que esto ocurrió.

Para explicarlo con más detalle se utilizará el ejemplo de una expresión $A?.B$, esta funciona de la siguiente forma: si A es nulo, entonces se retorna *null*, caso contrario se devuelve $A.B$. Es necesario para poder determinar las relaciones def/use de variables, saber si el valor de A era nulo durante la ejecución. Recordemos que en este enfoque no tenemos acceso a los valores de las variables, como consecuencia la única forma de saber si se va a utilizar solo A (es decir, devolviendo *null*) o $A.B$ es emitiendo una traza entre A y B de tal forma que al analizar el flujo se lea de la siguiente forma: *A no era nulo, por lo tanto se procede a devolver B.*

```
1
2  class EjemploDeEncapsulamiento
3  {
4    static void Main()
5    {
6      Client client1 = null;
7      Client client2 = new Client();
8
9      var a = client1?.getId();
10     var b = client2?.getId();
11     return;
12   }
13
14   class Client
15   {
16     public int getId()
17     {
18       return 1;
19     }
20   }
21 }
```

Tab. 3.1: Ejemplo de encapsulamiendo que ilustra la expresión $A?.B$

Ahora se muestra las lineas $var\ a = client1?.getId();$ y $var\ b = client2?.getId();$ instrumentadas:

```
1 var a = f_1_229_237(Tracer.PipeTraceSender.TraceConditionalAccessExpression
      (client1, 1, 221, 237));
2
3 var b = f_1_268_276(Tracer.PipeTraceSender.TraceConditionalAccessExpression
      (client2, 1, 260, 276));
```

Tab. 3.2: Ejemplo de encapsulamiendo: lineas de las expresiones instrumentadas

Las funciones $f\_1\_229\_237$ y $f\_1\_268\_276$ fueron agregadas en la instrumentación, así es como se ve:

```
1
2 static int? f_1_229_237(Tests.Cases.ConditionalAccessC.Client this_param)
3 {
4   var return_v = this_param?.getId();
5   Tracer.PipeTraceSender.TraceEndInvocation(1, 229, 237);
6   return return_v;
```

```
}

static int? f_1_268_276(Tests.Cases.ConditionalAccessC.Client this_param){
  var return_v = this_param?.getId();
  Tracer.PipeTraceSender.TraceEndInvocation(1, 268, 276);
  return return_v;
}
```

Tab. 3.3: Ejemplo de encapsulamiendo: funciones agregadas en la instrumentacción

Estas funciones se dedican a reproducir la funcionalidad original, con la diferencia que se incluye un llamado a traza dedicado a marcar el final de la invocación. Al tener también la traza TraceConditionalAccessExpression, podemos determinar que se provocó un llamado a un expresión del tipo $A?.B$; lo cual es lo que se busca saber. Para poder entender como funciona, nos remitiremos a código que genera su traza:

```
public static T TraceConditionalAccessExpression<T>(T val, int fileId, int
    spanStart, int spanEnd)
{
  if (val == null)
    TracerClient.Trace(fileId, (int)TraceType.ConditionalAccessIsNull,
    spanStart, spanEnd);
  return val;
}
```

Tab. 3.4: Ejemplo de encapsulamiendo: TraceConditionalAccessExpression

Tenemos dos posibilidades, la primera es que la variable *val*, que representa a $A?.B$ en el ejemplo, sea igual a *null*. En este caso se genera la traza y mediante la constante TraceType.ConditionalAccessIsNull se indica que no es necesario seguir evaluando. En caso contrario se procede a revolver *val* la cual seguirá con la evaluación.

3.2.3. Mecanismo de instrumentación

En primer lugar para analizar el programa cliente se utiliza Roslyn [29], una librería open source desarrollada por Microsoft que permite obtener los syntax tree, la información de tipos asociados a cada elemento del mismo, modificar el código y posteriormente compilarlo.

Una vez *levantado* el programa, se recorren los elementos del syntax tree agregando llamados a la API de emisión de traza según corresponda. Una vez que contamos con los syntax tree instrumentados procedemos a compilar el programa instrumentado (también utilizando el framework Roslyn), para que posteriormente al ejecutarse genere la traza deseada. Analicemos caso por caso como se insertan los llamados al tracer en el código cliente. Tener en cuenta dos cosas, la primera, que la traza agregada debe permitir seguir el flujo de la ejecución, la segunda, es que claramente la traza se agrega de distinta forma en cada lenguaje, pero el concepto de seguir el flujo sería equivalente en cada uno de ellos.

Clases

Se instrumentan todos los métodos contenidos en la clase. Esto puede dar a entender que con este procedimiento se está permitiendo un análisis de toda la funcionalidad que la

clase contiene, pero esto es un error dado que estaríamos olvidando el hecho que existe una parte de esta que no necesariamente se encuentra escrita; estos son los constructores por defecto. Si no se escriben, entonces el compilador garantiza versiones por defecto de estos los cuales no hacen nada, son efectivamente métodos vacíos. Sin embargo, esto no los excluye del análisis; al fin y al cabo pueden ser utilizados al crear el objeto. En definitiva necesitamos instrumentarlos para saber que se ejecutan (aunque adentro no haya nada).

Para ambos tipos de constructores, normales y estáticos; si no se tiene cualquiera de los dos definido, se lo define vacios y se los instrumenta; los detalles de los mismos se comentaran cuando hablemos de instrumentación sobre métodos. Con esto se puede acceder a información de la instanciación de los objetos de la clase si se lo requiere, incluso si el objeto se crea con los constructores por defecto.

En particular en C# puede haber llamados a base, los cuales dificultan seguir el flujo. Para poder seguir con mejor precisión que sucede, se agrega por cada clase un field que emite traza (es una cuestión técnica, pero la idea es que esa traza cae antes que cualquier otra y permite saber a que clase se está llamando con precisión). Este field es el siguiente: $int\ dummy = Tracer.TraceBeforeConstructor(1, 0, 19)$.

Métodos

El contenido del método debe ser instrumentado, lo que queremos saber para cada método es básicamente cuando se entra/sale, y cuándo produce una excepción no controlada. Por lo tanto se encapsula su contenido en un try y dentro del mismo se encapsula su contenido con trazas de entrada y salida, las cuales son de tipo TraceEnterMethod, TraceEnterConstructor o TraceEnterStaticMethod (según corresponda) y TraceExitMethod, TraceExitConstructor o TraceExitStaticMethod (según corresponda también). El catch se construye con un TraceEnterFinalCatch y un statement throw, la idea es entender que el flujo va a cambiar, pero no detener la excepción. El finally incluye simplemente un TraceEnterFinalFinally, este en caso de una excepción termina reemplazando a los exit originales.

Los constructores se tratan de manera particular. Si se los posee, se los instrumenta como cualquier método. Surgen diferencias menores como el uso de TraceEnterConstructor/-TraceExitConstructor, en lugar de los TraceEnter/TraceExit habituales para métodos. Sin embargo si existe una diferencia importante con respecto a cualquier otro método, y esta es la posibilidad de que un constructor invoque explícitamente o no al constructor de clase padre. En el caso de C#, un ejemplo de como esto toma forma sintáctica es la siguiente:

```
1 class ClaseA
2 {
3   class ClaseB
4   {
5     private string texto;
6     public ClaseB(string _texto)
7     {
8       texto = _texto;
9     }
```

```
10    }
11
12    class ClaseC : ClaseB
13    {
14      public ClaseC(string _texto) : base(_texto)
15      {
16      }
17    }
18  }
```

Tab. 3.5: Ejemplo de constructor

Debe notarse que es necesario capturar el comportamiento de este llamado. Más adelante en esta sección se explicará como se instrumentan los llamados a métodos, en los cuales se incluye este escenario.

En el caso de los constructores agregados con la instrumentación de la clase, no incluye el try/catch/finally que encapsula todos los statements anteriores. Esto es porque no hay posibilidad de excepciones dado que el constructor por defecto no posee statements. El procedimiento entonces se resume en agregar tres lineas de traza: los TraceEnter y TraceExit correspondientes, y un TraceEnterFinalFinally al final. Ya sabemos el uso de los dos primeros, el último simplemente sirve para notar el final que todo método instrumentado posee. Cualquier otro de estos tiene que ejecutar su bloque finally, el cual se agregó en la instrumentación, el analizados demanda esta traza como última linea ejecutada. Es por eso que se introduce, aunque no este dentro de un try/catch/finally.

Es importante notar para futuras explicaciones que cada bloque de código poseerá un llamado a la traza de entrada y otro de salida, a no ser que se indique lo contrario.

Statements

A cada statement s_i en un bloque de código se le agrega una nueva sentencia s'_i antes que llama al Tracer con la información de la original de s_i. Esto nuevamente permite entender el flujo. Al ejecutarse la instrucción s'_i se emite una traza indicando que la siguiente línea del código cliente será s_i (desde ya el programa instrumentado refiere al código del programa sin instrumentar).

Condicionales

Las expresiones if poseen una particularidad con respecto a la mayoría de los elementos del lenguaje de programación, la guarda. Esta es sometida al proceso de instrumentación como cualquier otra sentencia y, por lo tanto, las operaciones internas de una guarda requieren poder ser observadas y analizadas, debido a esto es necesario marcarlas para este propósito. La expresión aparecerá como primer parámetro de la operación lógica &&, el segundo será la guarda original instrumentada.

TraceEnterCondition y TraceExitCondition son usadas para la entrada y salida del bloque, respectivamente. Por último aclarar que tanto los else if como los else son instrumentados con los mismos principios que los if.

Los switch funcionan de manera similar a los otros condicionales, dado que se puede entender como un if que puede tener else if's y un else, salvando el detalle de que estos terminan con la operación break. A todos los usos, esta se trata de forma similar a un statement más, la diferencia es que se usa otro tipo de expresión de traza: el TraceBreak. Se opta por utilizar esta, en lugar de TraceSimpleStatement ya que en el caso de ejecutarse, el break sería la última instrucción dentro del bloque; no siendo esta el TraceExitCondition. Por lo tanto requiere un análisis distinto al de una expresión cualquiera. Esto aplica para cualquier uso de break, incluidos los loops.

Loops

Los loops son tratados de forma similar a los condicionales, dado que siempre poseen una condición que requiere mantenerse para continuar la iteración, esta es instrumentada de forma equivalente respecto a las guardas de los if.

Existe más de una forma en la que se puede salir de un ciclo, de la forma tradicional, en la cual la guarda llaga a un valor falso; mediante el uso de statements dedicados a generar un escape de este (break); otra posibilidad es si ocurre una excepción. Para poder capturar este último comportamiento se debe encapsular el ciclo en un bloque try; en los catch y finally se encontrará un llamado a traza del tipo TraceExitLoopByException y TraceExitLoop, respectivamente.

El bloque interno se somete al proceso de instrumentación de todos los bloques vistos previamente, a estos se les agregan al principio un TraceEnterCondition y al final un TraceExitCondition.

El caso de los for's es particular, en estos se tienen dos operaciones además de la guarda; la inicialización de variables internas y la modificación de sus valores al concluir un ciclo. En medio de estas dos está la guarda, que debe ser tratada de la misma forma que en los otros tipos de ciclo. Como cualquiera de estas declaraciones en otros contextos, la inicialización de variables requiere un TraceSimpleStatement para marcarlas, el cual se posicionará antes del for. La modificación con respecto al cambio de las variables en cada ciclo requiere consideraciones especiales, es necesario establecer que ahora el final del ciclo puede estar dado por esta operación, en lugar de la última instrucción al final del bloque. Por lo tanto, el TraceExitCondition debe estar ubicado después de esta operación, y no al final del bloque.

Llamadas a métodos/properties

El análisis para estos casos presenta un nuevo desafió a la hora de poder seguir el flujo, teniendo en cuenta que cada método o property a una librería no instrumentada puede enviar *callbacks* el código cliente, instrumentado. Para poder determinar, para un callback, *donde* se produjo el último call desde el cliente, debemos saber para cada método, si este es el que produjo o no un callback.
Dado que en una misma linea podrían existir varios llamados a métodos y propiedades distintas, queremos saber cuando cada uno termina de ejecutarse para saber si el callback provino de este. Esta necesidad cambia la forma en la cual se debe instrumentar, ya no se

puede llevar acabo simplemente asignando una instrucción de traza específica para marcar la linea. Debe poder aislarse cada uno de los llamados, y entonces se podrá analizarlos de forma independiente.

El procedimiento de instrumentación tiene como finalidad enviar una traza indicando que un call *ha terminado*, para lograrlo, reemplazamos la llamada al método original por un método propio el cual además de llamar al original envía una traza de finalización. Este nuevo método se crea con un valor de retorno igual que el de la expresión original y, en el caso de tenerlos, parámetros del mismo tipo. Por lo tanto, el llamado original es reemplazado por el llamado a este nuevo método. Al final de esta sección veremos un ejemplo completo.

Control de excepciones

Para el caso del try se limita a instrumentar los statements dentro del bloque. El catch añade al principio y al final un TraceEnterCatch y TraceExitCatch, respectivamente. Lo mismo ocurre con finally TraceEnterFinally y TraceExitFinally. No se agregan TraceEnter ni TraceExit dentro del bloque try debido a que siempre podemos tener certeza que va a ingresar a este; a diferencia de un if o while que requieren que se cumpla una condición.
El objetivo primario de este encapsulamiento es trackear las excepciones no capturadas por el cliente para poder relizar los ajustes necesarios en nuestra estructura. Es decir, si por ejemplo se produce una división por cero en un método m_k cuyo pila de llamados es $m_1, ..., m_{k-1}$, y recién esta excepción se capturara en m_1, tenemos que actualizar el estado de la memoria realizando el cambio de contexto (recordemos que nuestra memoria es simbólica y tenemos que hacer un tracking de lo que sucede en la ejecución del programa cliente).

EJEMPLO COMPLETO

Para mostrar como resulta un código cliente instrumentado luego de todo lo expuesto previamente, analicemos el siguiente ejemplo:

```
1  internal partial class ProgramaEjemplo
2  {
3    public static int Main()
4    {
5      int entero1 = 0;
6      int entero2 = 1;
7
8      while (entero1 < entero2)
9      {
10       var objetoDeclarado = new object();
11       entero2 -= 1;
12     }
13
14     return entero2;
15   }
16 }
```

Tab. 3.6: Ejemplo para slicing de DynAbs

El código instrumentado se puede obsertar en la imagen. Como podemos ver el ejemplo consiste en una clase con un único método. En este empezamos con la declaración y asignación de dos variables. Continuamos con un ciclo while, el cual podemos ver que se ejecuta exactamente una vez. Su contenido se limita a la declaración de una variable tipo object, con un llamado al constructor; el ciclo termina modificando una de las variables para asegurar que este no se ejecute una segunda vez. Finalmente se retorna el valor de una de las variables.

La instrumentación comienza por la clase, en esta se realizan dos operaciones. La primera, agregar dos constructores vacíos con los tipos public y static, para poder obtener información de ellos se les agregan los llamados a traza para marcar su principio y final, TraceEnterStaticConstructor o TraceEnterConstructor y TraceExitStaticConstructor o TraceExitConstructor, según el tipo. Para ambos casos, TraceEnterFinalFinally es el último llamado que se agrega. Todo esto permite obtener información de la creación del objeto, la cual solamente se podrá llevar acabo usando los constructores por defecto, dato que el ejemplo no provee otros. Los constructores agregados representan a estos últimos.

En segundo lugar esta la instrumentación del método. Para empezar, todo el contenido del mismo queda encapsulado en un bloque try/catch/finally, el cual posee las instrucciones de traza necesarias para que se puedan controlar las excepciones y obtener información de ellas si fuera necesario. Dentro del try, se encuentra la instrumentación del método, este es precedido por un llamado a traza de tipo TraceEnterMethod que se encarga de marcar el comienzo de la ejecución del mismo. A su vez, el método instrumentado precede el llamado a un statement de tipo TraceExitMethod, que indica que final de su ejecución.

De un total de tres declaraciones de variables, con sus respectivas asignaciones, podemos ver que existe una diferencia entre las dos primeras; que poseen una asignación "simple.[a] diferencia de la última que llama a un constructor. Las dos primeras requieren solamente una expresión TraceSimpleStatement. La última necesita más que esto para poder realizar el análisis. Si bien se incluye el statement simple, también se modifica la linea original cambiando la llamada al constructor por una llamada a una función introducida en la instrumentación (f_1_177_189), con el propósito de reproducir la ejecución del constructor y marcar el final de dicha ejecución usando la expresión TraceEndInvocation, esto se realiza antes de devolver el valor. Recordemos que el llamado al constructor entra entre los llamados a métodos, por lo tanto no se puede asumir que el mismo sea el único de estos en la linea, aunque en este caso lo es; el análisis por lo tanto, demanda el uso de esta nueva función para poder garantizar el análisis.

La instrumentación del while consiste en tres partes: se encapsula el while dentro de un bloque try/catch/finally para poder capturar información de las excepciones si estas ocurren; se instrumenta el contenido de la guarda agregando un llamado de traza simple como primer parámetro en una operación &&, esto permite asegurar que se ejecutara antes del llamado a la guarda original y nos permite marcarla como si fuera un statement mas; por último se debe instrumentar el contenido del bloque.

Para este último ya hemos explicado como se procede con la declaración de la variable de tipo objeto. Queda por notar, la inclusión de statements de traza para marcar el comienzo y final de la ejecución del bloque, estas están dadas por TraceEnterCondition y TraceExitCondition. La última linea en el ciclo es una operación simple que se encarga de disminuir el valor de la variable *entero2*. Esta requiere no más que un TraceSimpleStatement para poder analizarse, dado que es una operación *simple*, a la par de una asignación.

Finalmente, tenemos la operación de retorno del método. Esta no requiere más que un TraceSimpleStatement para poder garantizar el análisis.

```
internal partial class ProgramaEjemplo
{
  public static int Main()
  {
    try
    {
      Tracer.TraceEnterStaticMethod(1,45,295);

      Tracer.TraceSimpleStatement(1,86,102);
      int entero1 = 0;

      Tracer.TraceSimpleStatement(1,112,128);
      int entero2 = 1;
      try {
        while (Tracer.TraceSimpleStatement(1,138,261) && (entero1<entero2))
        {
          Tracer.TraceEnterCondition(1,138,261);

          Tracer.TraceSimpleStatement(1,188,223);
          var objetoDeclarado = f_1_210_222();

          Tracer.TraceSimpleStatement(1,237,250);
          entero2 -= 1;

          Tracer.PipeTraceSender.TraceExitCondition(1,138,261);
        }
      } catch(System.Exception) {
        Tracer.TraceExitLoopByException(1,138,261);
        throw;
      } finally { Tracer.TraceExitLoop(1,138,261); }

      Tracer.TraceSimpleStatement(1,273,288);
      return entero2;

      Tracer.TraceExitStaticMethod(1,45,295);
    }
    catch
    {
      Tracer.TraceEnterFinalCatch(1,45,295);
      throw;
    }
    finally { Tracer.TraceEnterFinalFinally(1,45,295); }
  }

  int dummy=Tracer.TraceBeforeConstructor(1,0,298);
  public ProgramaEjemplo()
  {
    Tracer.TraceEnterConstructor(1,0,298);
    Tracer.TraceExitConstructor(1,0,298);
    Tracer.TraceEnterFinalFinally(1,0,298);
  }
  static ProgramaEjemplo()
  {
    Tracer.TraceEnterStaticConstructor(1,0,298);
    Tracer.TraceExitStaticConstructor(1,0,298);
    Tracer.TraceEnterFinalFinally(1,0,298);
  }

  static object f_1_210_222()
  {
    var return_v = new object();
    Tracer.TraceEndInvocation(1, 210, 222);
    return return_v;
  }
}
```

Tab. 3.7: Ejemplo para slicing de DynAbs después de la instrumentación

4. EXTENSIÓN MULTILENGUAJE

4.1. Limitaciones actuales

Los slicers enfrentan diversas dificultades en su diseño y construcción. El lograr superar varias de ellos, tales como la escalabilidad y la eficiencia; es fundamental para lograr introducirlas como herramientas útiles para desarrolladores.

El desarrollo de slicers tienen un objetivo claro, el cubrimiento sobre todas las expresiones del lenguaje. Con esto en mente se debe considerar todos los problemas a la hora de implementar y mantener uno de estos. Particularmente no se posee una concepción de como se debería realizar el testing sobre estas y esto provoca que sea difícil tener claro en que momento pueden surgir errores, si bien se han realizado avances[26]. Esto hace que, como en la mayoría de los casos, no se pueda, o sea muy difícil, tener una certeza total sobre la correctivo de el slicer dado lo costoso que resultan las revisiones manuales.

El crear un slicer para un lenguaje específico desde cero es una tarea ardua, la cual implica un conocimiento completo de las expresiones de ese lenguaje. Existen notables variaciones entre slicers de distintos lenguajes, incluso a veces del mismo lenguaje, respecto a los procesos; las herramientas que utiliza y de las que depende; incluso el desarrollo de pruebas y la documentación.

Además de los problemas de típicos en la mayoría de los desarrollos de software, los lenguajes de programación no se caracterizan por ser invariables en el tiempo, constantes actualizaciones sobre estos agregan nuevas expresiones, sobre las cuales el slicer debe estas preparado para actuar. Esto introduce una presión significativa en el mantenimiento de estas herramientas a largo plazo. Un ejemplo de esto lo tenemos con Java (cuya primera versión data de 1996) que, hasta la versión SE 8, no poseía expresiones lambda; esta fue introducida el 18 de marzo de 2014. Debe notarse que, en este caso particular, se está hablando de un nuevo tipo de expresión que puede requerir un tiempo sustancial para alcanzar una versión operativa del slicer, y que requiere un conocimiento mínimo sobre la herramienta para llevar a cabo esta actualización con éxito. Se puede concluir que el mantenimiento de uno de estos programas requiere, potencialmente, realizarse a través de años, incluso décadas; sin mucha previsibilidad. Esto es especialmente problemático para los slicers, ya que el desarrollo de la mayoría de ellos fue iniciado por laboratorios como LaFHIS, esto puede provocar que en el futuro se dependa de estos mismos o de contribuciones independientes; suponiendo que el código este liberado. Con el desinterés y casi total ausencia de otros agentes para trabajar en el mantenimiento.

La limitación estudiada y enfrentada en este trabajo, el mono-soporte de lenguaje, es uno de los problemas más comunes en las herramientas de slicing. Dadas todas las complicaciones que se tienen que pasar para poder tener un slicer funcional, y todo lo que este demanda en términos de mantenimiento en el tiempo, puede ser verdaderamente problemático que se termine teniendo una herramienta que funciona sobre un solo lenguaje; de tantos que existen y se utilizan. Hay que tener en cuenta que los incentivos para continuar el desarrollo

de un slicer, se pueden ver gravemente mermados cuando este solamente será útil para un único lenguaje. Existe una herramienta que permite realizar el slicing independientemente del lenguaje, ORBS [27, 28]. Lamentablemente esta posee una complejidad computacional tan grande que la pone fuera de cualquier posible implementación al día de hoy que pueda considerarse para el uso industrial; dado que presenta largos tiempos de ejecución hasta para experimentos simples.

4.2. Solución propuesta para el mono-lenguaje de DynAbs

Bajo la estructura planteada en la sección anterior, existen dos grandes procesos que componen el DynAbs, instrumentador y backend. Para poder integrar un nuevo lenguaje al Slicer, es necesario modificar ambas partes. Por un lado, el Tracer debe incorporar la funcionalidad para instrumentar código en Visual Basic. Por el otro, tenemos el backend, que requiere modificaciones para soportar el procesamiento de la traza; sin embargo, se encontró una forma de crear una abstracción que permite generalizar este proceso de forma independiente al lenguaje. Otra dificultad que se presentó fue el fuerte acoplamiento del backend al lenguaje C#.

Al igual que en C#, la instrumentación y el análisis de la traza para Visual Basic se puede llevar a cabo utilizando el framework Roslyn[29]. Esta cualidad permite reutilizar gran parte del código existente. El desarrollo se organizó en varios pasos que se explican a lo largo de esta sección.

4.2.1. Parametrización del lenguaje

Previamente la herramienta solamente operaba con C#, para esta extensión fue necesario establecer una forma de reconocer sobre cuál de los dos lenguajes se va a trabajar. En base a cual de estos se trabaja, el instrumentador actúa de forma diferente; para el caso de Visual Basic se llama a la funcionalidad del instrumentador para este lenguaje. Esta modificación es meramente un valor que distingue entre estas dos posibilidades; es entonces el usuario quien puede asigna dicho valor, como un parámetro. Para conservar experimentos y trabajos previos, el valor por defecto es C#, con lo que no es necesario cambiarlos para tomar esto en cuenta. Se debe añadir que este parámetro podría ser deducido a partir de las extensiones de los proyectos (*.vbproj* o *.chproj*) o la de los archivos (*vb* o *csharp*).

4.2.2. Tracer para Visual Basic

C# y Visual Basic son dos lenguajes de programación que poseen muchas características comunes, tanto sintácticas como semánticas. Esto se debe a la coincidencia de algunos paradigmas que representan, y el soporte compartido de muchas herramientas y frameworks, principalmente aportados por Microsoft Corporation. Sin embargo, también poseen multitud de diferencias en estos aspectos. A continuación se listaran algunas de ellas.

- El bloque for es diferente en ambos lenguajes, concretamente en Visual Basic no posee un condicional sino que se limita a iterar secuencias de objetos (comúnmente números).
- Invocar al constructor base de la clase padre se hace de manera diferente. En el caso de C# se especifica usando el formato: “public Class() : base()”, en cambio

para Visual Basic, el constructor se invoca de la misma manera que cualquier otro mensaje usando la keyword de la clase base. Además el constructor base exige que se lo llame como la primera instrucción. Esto también es obligatorio para la invocación de otro constructor de la misma clase usando Me.New.

- ReDim es una instrucción para cambiar el tamaño de arrays. No existe en C#.
- With permite ejecutar una serie de instrucciones que refieren a un objeto puntual. No existe en C#.
- En Visual Basic no se admite un Return de ningún tipo en una función que devuelve un nuevo valor para un foreach. Solo se puede usar Yield.
- Erase se usa para borrar el contenido de arrays y liberar la memoria asociada a los mismos. No existe en C#.
- Resume se usa para reanuda la ejecución una vez que finaliza una rutina de manejo de errores. No existe Resume en C#. No se le puede aplicar un equivalente en C# para probar debido a que Resume no soporta estar dentro de un TryCatch.
- No existe Module en C#. Su equivalente es static class.
- En Visual Basic los métodos del Module no necesitan ser declarados como shared. Se los considera por defecto métodos estáticos.
- En Visual Basic los Module sólo se pueden declarar en un archivo o un namespace, nunca dentro de un Class o Structure.
- En Visual Basic no es posible usar el operador = para realizar una asignación doble, y además no existe operador que reemplace esta funcionalidad.

Todas estas diferencia, y muchas más, se ven reflejadas en el árbol sintáctico de cada programa, en cualquiera de los dos lenguajes. Por lo tanto, para poder realizar el proceso de instrumentación no es suficiente con modificar el ya existente; sino que fue necesario realizar una implementación de este exclusivamente para Visual Basic. Este toma la misma forma que el original y utiliza componentes equivalentes a los implementados para C#.

Sin embargo, se presentaron algunas dificultades que valen la pena discutir, una de ellas es el caso de los Else If. El problema se origina debido a que C# funcionaba de tal manera que el Visitor visita tanto el If, como a el Else If de la misma forma y por eso, los Else se instrumentaban igual que los Else If. En C# existe una modificación adicional que siempre ocurre en la instrumentación, mientras que en Visual Basic fue necesario replicar la estructura de forma artificial. En otras palabras, en Visual Basic para cada ocurrencia de un Else If se la modifica, como parte del proceso de instrumentación, de tal manera que, primero; pase a su equivalente usando un bloque de código Else, el cual, tendrá dentro un bloque If con la misma guarda y expresiones, y segundo; se debe realizar la instrumentación sobre su nueva estructura, como si fuera la original.

Otro problema, que resulto muy complejo de solucionar, fue el del constructor base. El problema no fue tanto que el llamado al constructor base sea una linea más en el constructor, sino el echo de que esta linea tiene que ser la primera. De lo contrario, existe un error

sintáctico que no permite ni siquiera la compilación del programa. Esto significa que no existe una forma de colocar el llamado a traza sobre la llamada al constructor padre. En C# esto nunca fue un problema, porque el elemento sintáctico BeforeConstructor, se encargaba del llamado. En Visual Basic, este se ejecuta después. Para resolver este problema se implemento una funcionalidad que se encarga de realizar un *"lookup"*, con el objetivo de observar si el constructor al que se esta llamando está instrumentado. A partir de esto, se puede colocar la traza del base delante de las otras.
Sin duda, el mayor problema en esta instancia fue dado por los bloques de código.

4.2.3. Instrumentación de bloques de código en Visual Basic

La instrumentación del código se realiza agregando instrucciones de tracing, entre las propias del programa. Con el instrumentador se ataca este problema trabajando sobre bloques de código, en lugar de instrucciones aisladas. En C# cada expresión que puede encapsular a otras es un SyntaxNode particular llamado BlockSyntax, que representa un bloque de código. Sin embargo, Visual Basic no posee un BlockSyntax genérico para cada SyntaxNode en el árbol que pueda contener otras expresiones (tales como los bloques For, While, If, etc). Cada bloque representa un tipo particular y, además, no poseen relación con los otros.

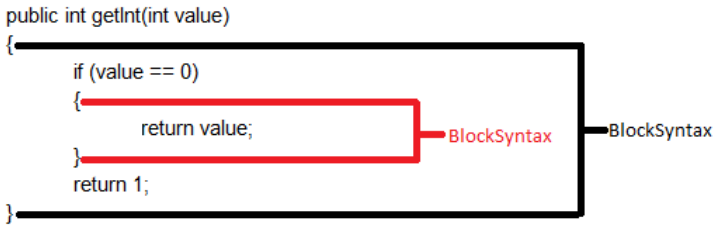

Fig. 4.1: Ejemplo de Block Syntax

Fue necesario generar una abstracción que represente un equivalente al BlockSyntax para Visual Basic. En esta tarea se tuvo en cuenta la existencia de algunos mensajes que comparten todos estos nodos, entre ellos Statements y WithStatements. Los cuales, respectivamente, devuelven y asignan las expresiones que son englobadas dentro del bloque. Puede notarse que, al poseer esta funcionalidad, se puede instrumentar el código de cualquiera de estos nodos. La necesidad de poder representar estos bloques con estas funcionalidades llevó a crear la siguiente interfaz.

```
1 public interface BlockSyntax
2 {
3   SyntaxList<StatementSyntax> Statements { get; }
4   BlockSyntax WithStatements(SyntaxList<StatementSyntax> statements);
5 }
```

Tab. 4.1: Interface general de BlockSyntax

Usando esta como base se pudieron crear BlockSyntaxWrappers; los cuales son objetos dedicados a encapsular la funcionalidad de cada tipo de bloque que represente un BlockSyntax. Al implementar uno de estos para un tipo de bloque particular, se puede garantizar

que al momento de instrumentarlos solo será necesario actuar a través de su Wrapper. Gracias a esto, fue posible facilitar la instrumentación sin generar excesivo código repetido. En forma de ejemplo, se muestra el WhileBlockSyntaxWrappers que se usa para encapsular como bloque al WhileSyntaxBlock de Visual Basic.

```
1  public class WhileBlockSyntaxWrapper : BlockSyntax
2  {
3    private WhileBlockSyntax _someClass;
4
5    public WhileBlockSyntaxWrapper(WhileBlockSyntax someClass)
6    {
7      _someClass = someClass;
8    }
9
10   public SyntaxList<StatementSyntax> Statements
11   {
12     get
13     {
14       return _someClass.Statements;
15     }
16   }
17   public BlockSyntax WithStatements(SyntaxList<StatementSyntax> statements)
18   {
19     _someClass = _someClass.WithStatements(statements);
20     return this;
21   }
22
23   public static implicit operator WhileBlockSyntax(WhileBlockSyntaxWrapper
       blockSyntaxWrapper)
24   {
25     return blockSyntaxWrapper._someClass;
26   }
27 }
```

Tab. 4.2: Ejemplo de Wrappers para WhileBlockSyntax

4.3. Backend para Visual Basic: Procesamiento de la traza

Una vez modificado el instrumentador, si se lograba emitir traza en el mismo orden que en C#, también lograría funcionar el backend para Visual Basic. Sin embargo, esto requerío cambios en el procesamiento para poder funcionar, un ejemplo de estas modificaciones es el tratamiento de la invocación al constructor base como un statement más, uno que además debe ser el primero del constructor. Si bien las diferencias sintácticas y semánticas entre ambos lenguajes impiden reutilizar el procesamiento de la traza, fue posible adaptar la pasada versión para que pueda soportar Visual Basic, aprovechando ahora las similitudes que ambos lenguajes poseen en sus SyntaxTree's.

Antes de comenzar a hablar de los cambios realizados, es importante notar que el diseño del procesamiento de la traza posee, de por si, caracteristicas que facilitan enormemente extenderlo para soportar otro lenguaje. Esto hace referencia principalmente al IOperation, el cual es la clase raíz para representar la semántica abstracta de los statements de C# y Visual Basic. Gracias a esto, cada una de las expresiones que se debe analizar son distinguidas por el IOperation que representan, y en consecuencia tratadas de forma distinta.

Sin embargo, esto no es suficiente para que el procesamiento de la traza sea multilenguaje; consideremos que las IOperation no son suficientes para obtener toda la información que se puede necesitar sobre una expresión, esta solo se puede obtener mirando directamente al árbol sintáctico, nodo a nodo.

Abstracción para los SyntaxNode

Los dos lenguajes poseen elementos sintácticos comunes en la programación imperativa, tales como el If, While, Switch, etc, los cuales no difieren demasiado si el código está en C# o Visual Basic. Es por esto que una forma de sacar provecho a la funcionalidad existente fue crear una nueva capa de abstracción, la cual permite acceder a las funcionalidades comunes de forma independiente del lenguaje. Esta capa, llamada abstracciones del lenguaje, básicamente encapsulan la funcionalidad de un SyntaxNode que puede pertenecer a un lenguaje u otro, y proveen una serie de mensajes que representen funcionalidades necesarias para su procesamiento, que ambos nodos potenciales comparten. De esta forma se puede recurrir, según sea necesario, a estos objetos encapsulados en vez de a sus originales. Por ejemplo, supongamos que se tiene el nodo que representa un If y que es necesario acceder a la guarda del mismo, utilizando esta abstracción se podría simplemente crear la funcionalidad que permite acceder a dicha información y utilizarla sin tomar en cuenta sobre que lenguaje se está trabajando. La misma adoptará la responsabilidad de preguntar sobre que lenguaje se actúa y, en consecuencia, realizar la operación apropiada sobre el mismo nodo que pertenece a uno de los lenguajes.

La AbstractionLanguage, así llamada, permite diferenciar entre la funcionalidad de un nodo, para un determinado tipo, independientemente del lenguaje en el que estén, sin la necesidad de modificar el código del procesamiento de la traza de forma que se incluya la funcionalidad requerida por duplicación. La única modificación que se requería en estos casos sería reemplazar el CSharpSyntaxNode por una de estas abstracciones dependiendo de cual sea. Adicionalmente, este objeto requiere poseer la capacidad de reconocer su propio tipo, por ejemplo que un AbstractionLanguage el cual representa un While pueda decir si un nodo, tanto si es de C# o Visual Basic, representa un While.

La implementación de esta funcionalidad consistió en asociar un tipo determinado de nodo, con el conjunto de mensajes que el nodo posee. Poder entender el tipo concreto es simple; basta con preguntar si es de C# o Visual Basic y el tipo de nodo en cada caso.

La funcionalidad por otro lado se debe implementar y completar con los mensajes que sean necesarios, en ningún momento se planteo hacerla total, sino que se la completó de acuerdo con la necesidad. Esta fue nombrada como LanguageAbstractionWrapper, y se creó a partir de los nodos originales. Cada mensajes de este objeto debe ser equivalente a enviar el mismo mensaje al nodo original. Como se explicó anteriormente, esta debe incluir una distinción de tipo y, posteriormente, ejecutar la funcionalidad equivalente al mismo. Otros LanguageAbstractionWrapper fueron creados para los resultados de estas operaciones. Esto se debió a que estos eran SyntaxNodes, los cuales también dependían del lenguaje.

Con estos cambios, y con la ayuda de las características del diseño original, fue posible realizar el procesamiento de la traza para Visual Basic usando la funcionalidad existente para C#. A parte de las modificaciones mencionadas, solamente fueron requeridos cambios puntuales en la funcionalidad para atender problemas incompatibles con esta abstracción, como por ejemplo el caso de los constructores base, mencionado anteriormente.

5. EVALUACIÓN

Para evaluar está nueva extensión de DynAbs se utilizaron casos unitarios de test para abordar las distintas expresiones sintácticas del lenguaje y se aprovechó un benchmark con el cual el grupo vino trabajando en C#, usándolo para pruebas generales, todo esto con el fin de evaluar precisión y performance.

5.1. Cubrimiento del lenguaje y expresiones sintácticas básicas

5.1.1. Objetivos

Los objetivos primarios del testing son verificar que el slicer funcione para pequeñas estructuras y, que el mismo desarrollo no este introduciendo fallas para C#.

A la hora de realizar el desarrollo y verificación del programa, fue esencial mantener una colección de test suites adecuados. Entre los mismos, se pueden distinguir dos grupos: los diseñados para aproximar una instrumentación correcta y otro para verificar el proceso de slicing completo. La creación y mantenimiento de estos casos de test fue necesario durante la totalidad del desarrollo.

5.1.2. Metodologías

Previamente a poder obtener el slice de un programa, es necesario realizar el proceso de instrumentación, dicha funcionalidad debió ser implementada (al menos parcialmente) como el primer paso en el desarrollo. No está de más notar que el testing en esta instancia presenta algunas dificultades. Para empezar, se requiere poseer fragmentos de código, y las versiones de los mismos después de la instrumentación. Esto es un proceso difícil de llevar a cabo y, además, debe tenerse en cuenta que la variedad del testing debe ser considerable, y suficiente para cubrir casos esenciales.

Para resolver el problema de alcanzar una instrumentación correcta para Visual Basic se construyó un oráculo parcial, esto fue posible asumiendo la correctitud de la funcionalidad previa dedicada a C#. En la implementación del mismo fue importante tener en cuenta las similitudes semánticas que comparten el código instrumentado en los dos lenguajes, esto se deriva de las similitudes propias de estos. Un ejemplo simple puede ser la declaración de una variable, por más diferencias que existan en la sintaxis, semánticamente para la mayoría de los casos es indistinta.

La instrumentación también comparte similitudes, las mismas se pueden resumir con una nueva línea, que indica el tipo de la traza y el cambio de la funcionalidad referente al valor que se le asigna, si es que se lo hace, según corresponda.

En cuanto al oráculo en sí mismo, cada caso de prueba se construye en base a dos piezas de código, en las mismas se pretende observar semánticas equivalentes para los dos lenguajes. El oráculo está compuesto por los siguientes pasos:

1. Corroborar que el código original en los dos lenguajes compile sin errores.
2. Realizar la instrumentación para cada uno.
3. Corroborar que el código instrumentado en los dos lenguajes compile sin errores.
4. En ambos casos la cantidad de instrucciones referentes a la traza generada deben ser equivalentes (misma cantidad, tipo y posición relativa).

Debido que la cantidad de líneas, su largo y posición; varían debido a la sintaxis. No se utilizó mucha más información para nutrir al oráculo de precisión. Sin embargo, este resultó en una herramienta muy útil a la hora de desarrollar este primer paso para el slicing de Visual Basic.

Todo el testing presentado hasta este punto se centró en el primer proceso, la instrumentación. Este además, cuenta con la característica de aprovechar la funcionalidad previa de C# para potenciar el desarrollo de la extensión. Aparte de esto, aun sigue siendo necesario evaluar la funcionalidad en su totalidad, de la misma forma en la que se hizo, y continua haciéndose, para C#.

Equivalente al testing pre-existente en C#, se realizó el propio para Visual Basic. Por cada pieza de código en la que se pretende realizar la prueba, se ejecuta el slicing y se corrobora que las líneas resultantes coinciden con las del resultado esperado. En un principio se combinaron casos propios, los cuales permiten verificar la correctitud para situaciones puntuales o casos exclusivos de Visual Basic, con versiones semánticamente iguales a las de pruebas realizadas previamente en C#.

Un punto importante a tener en cuenta fue la constante necesidad de realizar pruebas de regresión usando el testing para C#. Esto se debió a la necesidad de modificar el proceso de slicing, usando las ya explicadas abstracciones del lenguaje, adicional a modificaciones pensadas para atender situaciones propias de Visual Basic.

Limitaciones de traducción

Como se mencionó anteriormente las características comunes entre los lenguajes son muy útiles para lograr un testing adecuado sobre la instrumentación, mediante la comparación de instrucciones de traza. Si bien esta aproximación resultó efectiva para un gran número de expresiones, también existen casos donde no se puede depender enteramente de estas similitudes. En ocasiones no existe una traducción entre la pieza de código escrito en un lenguaje, que permita garantizar esta propiedad para su posterior análisis. Un caso de esto es la expresión For. La forma en que esta funciona en C# varía de acuerdo al tipo, se tiene entonces el clásico ciclo sobre variables declaradas, una condición y el cambio de dichas sobre estas variables al final. Esta forma es la más conocida y puede llevarse a cabo de forma equivalente usando un While sin demasiados cambios. Por otro lado tenemos el ForEach, que itera directamente sobre un conjunto de valores u objetos, esta expresión existe también en Visual Basic. Otra forma en la que se manifiesta la podemos encontrar en Visual Basic, esta se nombra como ForTo. La misma consiste en modificar de forma periódica, por cada ciclo, una variable puntual entre un valor inicial y uno final, estable-

ciendo la modificación paso a paso de la misma.

Esta distinción genera problemas a la hora de instrumentar debido a que el ForTo no posee un condicional explícito a cumplir y, dado que este detalle afecta el resultado, no puede establecer una relación directa. No resultó trivial sortear estas incompatibilidades a la hora del testing y, en última instancia, se decidió no enfrentarlas. En caso de que se quisiera hacer esto en futuros trabajos se deberían establecer un conjunto de reglas para cada uno de los casos donde estas diferencias aparecen y cómo sortearlas; para tratar de aproximar el oráculo a uno más competente.

A pesar de sus limitaciones, fue posible aprovechar este oráculo para llevar a cabo la implementación del proceso de instrumentación.

5.1.3. Resultados

Si bien no se pudo alcanzar el cubrimiento de todos los posibles casos del lenguaje, se pudo construir un test suite adecuado para la mayoría de los casos equivalentes que ya existían para C#. Por otro lado, se completaron algunas pruebas especializadas en circunstancias puntuales de Visual Basic.

El desarrollo se centró en satisfacer casos conformados por las expresiones "más comunes", tales como If y Switch; los distintos tipos de ciclos como While, For y ForTo; pasando por declaración y uso de variables, e incluyendo distintas formas en las cuales una clase o módulo pueden constituirse, tanto por atributos como operaciones.

Para la instrumentación se alcanzaron un total de 269 test cases, de los cuales 228 corrieron correctamente, lo cual es aproximadamente 84,7 % . Se debe añadir que este último número está excluyendo casos donde el oráculo lanzara un falso negativo debido a diferencias irreconciliables entre los lenguajes, como se detalló anteriormente. Los casos que no se pudieron cubrir contienen expresiones que no consideraron necesarias para completar este trabajo, no son evaluadas en sucesivos experimentos y se las consideró de menor prioridad comparadas a las que efectivamente se cubrieron.

Para el caso de las pruebas sobre slicing completo, se desarrolló un total de 179 test cases, de los cuales se obtuvieron resultados exitosos para 155, lo cual es aproximadamente 86,6 %. Similar al caso de la instrumentación, los casos que no se cubrieron contienen expresiones que se valoraron como no prioritarias. Un ejemplo de test case para slicing completo, en el cual se marcan las lineas coincidentes, es el siguiente:

```
1 internal partial class Ejemplo_While{
2   public static int Main(){
3       int entero1 = 0;
4       int entero2 = 1;
5
6       while (entero1 <entero2) {
7         var objetoDeclarado = new object();
8         entero2 -= 1;
```

```
9       }
10
11      return entero2;
12    }
13  }
```

Tab. 5.1: Ejemplo para slicing completo C#

```
1  Class Ejemplo_While
2    Public Shared Function Main() As Integer
3      Dim entero1 As Integer = 0
4      Dim entero2 As Integer = 1
5
6      While entero1 <entero2
7        Dim objetoDeclarado = New Object
8        entero2 -= 1
9      End While
10
11     Return entero2
12   End Function
13 End Class
```

Tab. 5.2: Ejemplo para slicing completo Visual Basic

A pesar de todo el trabajo realizado, todavía existen casos en los cuales el slicer no ha podido demostrar plena operatividad. En la sección de futuros trabajos se detallaran algunos de estos.

5.2. Evaluación general

5.2.1. Objetivo

Si bien en la sección anterior se habló de el uso de casos unitarios de testing, estos están destinados a evaluar el correcto funcionamiento. Se necesita, además, plantear una evaluación de la precisión y performance.

5.2.2. Metodología

Para poder establecer pruebas sobre ejemplos reales se utilizó el benchmark Olden[17], que anteriormente ya había sido de utilidad para probar el Slicer en C#. Los experimentos se realizaron sobre los programas: BH, BiSort, Em3d, Health, MST, Power, TreeAdd, TSP y Voronoi. Además, no se utilizó la modalidad de optimización de ciclos.

Dado que el benchmark está en C#, fue necesario convertirlo a Visual Basic, esto se debe hacer manteniendo el comportamiento, ya que sera necesario para comparar los resultados. Para esto se sacó provecho de un plugin para Visual Studio que transforma de manera automática código entre los dos lenguajes[16]. Sin embargo, este presentó inconvenientes; a continuación se listan los mismos:

- La formas en las que representa ciclos *For* que consistan en el decremento de un valor. Para estos la traducción correcta siempre debería incluir *Step -1*; sin embargo, el plugin nunca tomaba en cuenta esto. Por lo cual, nunca se entraba a estos ciclos.
- La división entera se representa de la misma forma que la flotante en C#. Visual Basic no comparte esta cualidad, para este caso se requiere el operador \ en lugar de /.
- La asignación múltiple ($i = j = k$) no existe en Visual Basic. Para resolver este problema el plugin crea una función nueva que se dedica a replicar esta funcionalidad y la utiliza en los casos donde debería aparecer. Sin embargo, esto falla si el segundo elemento es un valor dentro de un array, evitando que la asignación tome lugar para este.
- El operador & tiene su uso aplicado entre elementos de tipo bool o integer. En Visual Basic se utiliza en cambio AND. El plugin convierte correctamente a & para el caso de los bool; pero lo conserva para integer. Dado que estos operadores no son equivalentes en Visual Basic, este cambio altera el comportamiento del programa.
- Por defecto, la herramienta traduce la conversión de un float o double, a entero, utilizando CInt. Si bien esa función cubre este propósito, existen ocasiones en las cuales el resultado puede variar. Por ejemplo si se tiene el valor 4.999999999; en C# el resultado sería 4, en Visual Basic 5. Para corregir este problema se puede recurrir a la función Microsoft.VisualBasic.Conversion.Int.
- La inferencia del slicer sobre el tipo de una variable declarada, puede dar lugar a problemas en muchas ocaciones. Este se presentó especialmente sobre listas. Para solucionar este error se debe especificar siempre el tipo de la variable.

Todos estos alteran el comportamiento de la versión Visual Basic, provocando que difiera de la original. Para sobrepasar este problema, se optó por corregir estos defectos a mano.

Con este proceso completado, se tiene el mismo benchmark que antes, y su versión en Visual Basic, equivalentes en comportamiento. Sin embargo, esto no significa que exista una coincidencia total en las lineas de cada caso.

La evaluación para esta extensión, según lo que se creyo necesario, requiere someterla a pruebas semejantes a las de la versión original. Para C# particularmente, en trabajos previos, se realizaron experimentos que consistieron en ejecutar el benchmark citado, capturando datos; tanto los que son relativos al slicing, como de la ejecución propia del programa. Esto último hace referencia a tiempos de ejecución, tanto para el proceso completo como para sus dos principales partes; la instrumentación y la ejecución de la traza y, además, se incluye el input en cada caso, junto al criterio. Los datos de interés particulares del slicing son la cantidad de lineas ejecutadas, el tiempo de ejecución y la cantidad de lineas resultantes del slicing, expresadas como cantidad y porcentaje sobre el total.

Este experimento tomó forma a partir de los resultados del Slicer sobre cada uno de los componentes del benchmark, y la comparación de los mismos. Se puede afirmar que, dado que el comportamiento de cada programa y su versión para el otro lenguaje, comparten

el comportamiento, si para cada componente del benchmark se conservan el mismo comportamiento después del slicing, tenemos que se cubre tan exitosamente el benchmark en C#, como en Visual Basic.

La forma más adecuada para comparar el comportamiento de los dos programas, en este particular escenario, es realizando una comparación de las lineas resultantes del proceso de slicing. La tarea de verificar la coincidencia de lineas puede llegar a ser larga y tediosa, para simplificarla fue necesario modificar el código de los experimentos. Por cada uno de estos, se buscó hacerlo lo más similar posible, entre lenguajes, en cuanto a la posición de las lineas, dejando espacios en blanco si era necesario. De esta forma, a la hora de verificar la equivalencia se puede observar, al menos en parte, que las lineas que se obtiene como resultado del slicing coincidan.

Para poder automatizar este proceso se modificó la herramienta SliceResultsComparison, que ya formaba parte de DynAbs para que pudiera soportar Visual Basic. Básicamente, esta tiene el proposito de distinguir las diferencias y coincidencias, utilizando una interfaz gráfica para comparar las lineas, entre dos archivos de lineas resultantes y las piezas de código a los que estos refieren. Además, se realizaron cambios para que la misma pueda cargar resultados del slicing, sin forzar uno de los dos a ser exclusivamente de Java, como funcionaba anteriormente. Con todo esto, basta con brindarle la ubicación de los resultados del slicing, y esta mostrará los mismos marcando tanto coincidencias como diferencias.

5.2.3. Resultados

			DynAbsVisualBasic				DynAbs			
Programa	Input	Criterio	#Lineas ejecutadas	Tiempo (Segundos)	#Lineas sliceadas	Tamaño de la Traza (KB)	#Lineas ejecutadas	Tiempo (Segundos)	#Lineas sliceadas	Tamaño de la Traza (KB)
BH	-b 26 -s 2	1	383	18.3	53	2925	383	17.22	53	3490
		2	384	18.32	311	2925	384	17.23	311	3490
		3	385	18.33	53	2925	385	17.23	53	3490
		4	386	18.33	311	2925	386	17.23	311	3490
		5	387	18.33	53	2925	387	17.23	53	3490
		6	388	18.33	303	2925	388	17.25	303	3490
BiSort	-s 12	1	127	0.59	87	29	127	0.47	87	30
		2	128	0.61	93	29	128	0.49	93	30
		3	129	0.61	92	29	129	0.49	92	30
Em3d	-n 2 -d 2 -i 13 -s 12	1	144	0.52	88	12	144	0.45	88	13
		2	145	0.54	58	12	145	0.46	58	13
		3	146	0.54	72	12	146	0.46	72	13
		4	147	0.54	51	12	147	0.46	51	13
		5	148	0.54	58	12	148	0.46	58	13
Health	-l 4 -t 14 -s 12	1	236	20.25	167	2761	236	20.43	167	2944
		2	237	20.27	164	2761	237	20.45	164	2944
		3	238	20.27	171	2761	238	20.45	171	2944
MST	-v 4	1	195	0.57	148	24	187	0.52	140	25
Power		1	294	21.96	251	4721	294	23.08	251	5660
		2	295	21.98	251	4721	295	23.1	251	5660
		3	296	21.98	251	4721	296	23.1	251	5660
		4	297	21.98	251	4721	297	23.1	251	5660
TreeAdd	-l 2	1	44	0.4	26	2	44	0.38	26	2
TSP	-c 20	1	148	1.05	25	143	145	0.99	25	147
		2	149	1.06	26	143	146	1.01	26	147
		3	150	1.07	102	143	147	1.01	99	147
		4	151	1.07	102	143	148	1.01	99	147
		5	152	1.07	102	143	149	1.01	99	147
Voronoi	-n 9	1	353	5.61	249	521	349	5.4	247	519
		2	354	5.64	249	521	350	5.42	247	519
		3	355	5.64	249	521	351	5.43	247	519
		4	356	5.66	249	521	352	5.43	247	519
		5	357	5.66	249	521	353	5.44	247	519
		6	358	5.67	249	521	354	5.45	247	519

Tab. 5.3: Resultados de Olden

Dado que la coincidencia de lineas no es total, se puede esperar diferencias en la cantidad de lineas ejecutadas. Por esto se altera también los porcentajes de lineas resultantes del slicing y, en muchos casos, también el total de lineas filtradas se ve alterado. Por lo explicado anteriormente, es evidente que la diferencia entre el total de lineas ejecutadas no es un problema, siempre y cuando, las lineas de más no alteren el comportamiento esperado y su coincidencia para los dos casos. Algo similar ocurre con las lineas resultantes del slicing. Más adelante, se justificará porque esto no es un problema en este caso y, además, la diferencia de lineas está justificada.

Como podemos observar, los tiempos de ejecución para ambos casos se mantienen en rangos similares. Podemos concluir que no se presentan cambios en ese aspecto. Al igual que el tamaño de las trazas.

La diferencias entre lineas resultantes del slicing para los dos lenguajes

Los experimentos mostraron diferencias respecto a las lineas resultantes del slicing entre los dos lenguajes. Esto podría indicar que los experimentos no obtuvieron los resultados esperados, ya que esta diferencia implicaría que el comportamiento de los programas resultantes del slicing difieren. Sin embargo, en todos los casos encontrados, las causas fueron por motivos sintácticos; imposibles de arreglar para estos experimentos y, por lo tanto, no afectan la correctitud; dado que los programas después del slicer mantiene comportamientos equivalentes. A continuación se detallaran las diferencias encontradas. A la hora de mostrarlas, se utilizará ■ para las coincidentes y ■ para las diferentes.

La diferencia de For y ForTo

Uno de los motivos que provocaron estas diferencias fue que el For, de C#, y el ForTo, de Visual Basic, no proveen conversiones que se puedan representar con una cantidad igual de lineas para todos los casos. Por lo tanto, la conversión muchas veces depende de reemplazar el For por un While. A continuación se muestra uno de las ocurrencias de este error; se la extrajo del archivo Graph de el experimento MST.

```
1  private void addEdges(int numvert)
2  {
3     int count1 = 0;
4     for (Vertex tmp = nodes[0]; tmp != null; tmp = tmp.Next())
5     {
6        Hashtable hash = tmp.Neighbors();
7        for (int i = 0; i <numvert; i++)
8        {
9           if (i != count1)
10          {
11             int dist = computeDist(i, count1, numvert);
12             hash.put(nodes[i], dist);
13          }
```

```
14      }
15      count1++;
16    }
17  }
```

Tab. 5.4: Resultados de Olden: diferencia de For y ForTo C#

```
1  Private Sub addEdges(ByVal numvert As Integer)
2
3    Dim count1 As Integer = 0
4    Dim tmp As Vertex = nodes(0)
5    While tmp IsNot Nothing
6      Dim hash As Hashtable = tmp.Neighbors
7      For i As Integer = 0 To numvert - 1
8
9        If i <>count1 Then
10
11         Dim dist As Integer = computeDist(i, count1, numvert)
12         hash.put(nodes(i), dist)
13       End If
14     Next
15     count1 += 1
16     tmp = tmp.[next]
17   End While
18 End Sub
```

Tab. 5.5: Resultados de Olden: diferencia de For y ForTo Visual Basic

Para este componente existen tres lineas de más en el código Visual Basic. Para empezar tenemos a la función `addEdges` que es la responsable de dos lineas adicionales. Se puede observar que el traslado de un lenguaje a otro dejó esta función con una pequeña diferencia. Al no poder traducirse sin cambiar lineas, aparecen dos que deben agregarse al slice, la linea 5 y 14. Se puede deducir que esto no afecta el resultado de DynAbs para este caso, ya que estas lineas simplemente son resultado de la ausencia de una traducción adecuada, y su inclusión es necesaria para mantener el comportamiento.

La diferencia provocada por una asignación múltiple

Anteriormente, se mencionó a la doble asignación como un caso en el cual la traducción no puede darse de forma completa, dado que Visual Basic no la soporta. Con lo cual, esta linea de más es necesaria en tanto la linea correspondiente a la doble asignación en C# lo sea.

```
1  public Graph(int numvert)
2  {
3    nodes = new Vertex[numvert];
4    Vertex v = null;
```

```
5    // the original C code creates them in reverse order
6    for (int i = numvert - 1; i >= 0; i--)
7    {
8        Vertex tmp = nodes[i] = new Vertex(v, numvert);
9        v = tmp;
10   }
11
12   addEdges(numvert);
13 }
```

Tab. 5.6: Resultados de Olden: diferencia provocada por una asignación múltiple C#

```
1  Public Sub New(ByVal numvert As Integer)
2
3    nodes = New Vertex(numvert - 1)
4    Dim v As Vertex = Nothing
5
6    For i As Integer = numvert - 1 To 0 Step -1
7
8        Dim tmp As Vertex = New Vertex(v, numvert)
9        nodes(i) = tmp
10       v = tmp
11   Next
12
13
14   addEdges(numvert)
15 End Sub
```

Tab. 5.7: Resultados de Olden: diferencia provocada por una asignación múltiple Visual Basic

En este ejemplo podemos observar una diferencia entre las lineas de ambos constructores, concretamente dada por la linea 10. Nuevamente tenemos un problema dado por la traducción, esto se puede explica observando el caso de C#. Podemos ver que en la linea 8 se realiza una doble asignación, en la cual uno de los elementos es un acceso al array. Como no se puede presentar el mismo comportamiento en Visual Basic usando una sola linea, es innevitable que el slicing de como resultado esa diferencia.

Justificaciones a las diferencias encontradas

Para los casos de `BH`, `BiSort`, `Power`, `TreeAdd`, `Em3d` y `Health`; se observo una coincidencia total. Los otros experimentos poseen diferencias entre lineas.

	Clases			
Lineas	MST	Hashtable	Graph	Vertex
1	+3/-0	+2/-0	+3/-0	+0/-0

Tab. 5.8: Resultados del experimento sobre MST

Es provocada por la diferencia entre For y ForTo en los archivos `Graph`, `MST` y `Hashtable`. Además, `Graph` posee una asignación doble.

	Clases		
Lineas	TSP	Tree	RandomGenerator
1	+0/-0	+0/-0	+0/-0
2	+0/-0	+0/-0	+0/-0
3	+0/-0	+3/-0	+0/-0
4	+0/-0	+3/-0	+0/-0
5	+0/-0	+3/-0	+0/-0

Tab. 5.9: Resultados del experimento sobre TSP

Se observó que en todos los casos las lineas de más están dadas por la diferencia entre For y ForTo.

En este caso tenemos un componente llamado RandomGenerator, su proposito es el de eliminar la aleatorierdad del experimento, dado que este podría hacer variar los resultados. Su evaluación de lineas no es importante, aunque la coincidencia sea total.

	Clases					
Lineas	Vertex	Edge	EdgePair	Vec2	MyDouble	Voronoi
1	+1/-0	+0/-0	+0/-0	+0/-0	+0/-0	+0/-0
2	+1/-0	+0/-0	+0/-0	+0/-0	+0/-0	+0/-0
3	+1/-0	+0/-0	+0/-0	+0/-0	+0/-0	+0/-0
4	+1/-0	+0/-0	+0/-0	+0/-0	+0/-0	+0/-0
5	+1/-0	+0/-0	+0/-0	+0/-0	+0/-0	+0/-0
6	+1/-0	+0/-0	+0/-0	+0/-0	+0/-0	+0/-0

Tab. 5.10: Resultados del experimento sobre Voronoi

En todos los casos la diferencia es provocada por asignaciones múltiples.

Con todo esto, podemos concluir que las diferencias de lineas estan justificadas en cada uno de los casos, ya que estos mantienen el comportamiento antes y después del slicing. Con lo cual, todos los experimentos sobre el benchmark Olden indican que el slicer funciona apropiadamente comparandolo con su versión en C#.

6. CONCLUSIONES

Implementar un slicer multilenguaje para .Net resultó no ser trivial. Se han detallado en secciones anteriores, la variedad de problemas que se detectaron al introducir nuevas expresiones.

A lo largo de este trabajo, se demostró que era posible mantener un mismo analizador semántico para distintas estructuras sintácticas. Y por lo tanto, que este no se viera con la necesidad de modificaciones, al introducirse un nuevo lenguaje.

Es necesario aclarar que, técnicamente, existe un slicer multilenguaje para .Net [18], el cual es previo a esta extensión de DynAbs. Mediante el código disponible y una conversación con uno de los principales responsable del proyecto, se concluyó que este slicer no está disponible actualmente, y no debería tomarse en cuenta. Las principales razones son que no tiene mantenimiento reciente; no es una herramienta productiva; está deprecado y, si bien se posee acceso al código del Tracer, no es el caso para el slicer, muy probablemente por las razones anteriores. Por lo tanto, no podemos someter a DynAbs a pruebas similares, ni experimentar para compararlo con este proyecto.

Se concluye, por todo lo anterior, que esta extensión convierte a DynAbs en el único slicer dinámico de emisión y tratamiento de traza actualmente funcional para Visual Basic.

7. FUTUROS TRABAJOS

Si bien el trabajo presentado, hasta el momento de publicación del informe, abarca el slicing sobre muchas de las expresiones del lenguaje para Visual Basic; todavía existen algunas que no se han implementado en DynAbs. Se debe tener en cuenta que, para garantizar una funcionalidad optima y completa, es necesario abarcar todas las posibles expresiones del lenguaje. A su vez, existen algunas de estas que, si bien están implementadas, no mantienen un comportamiento correcto para todos los casos.

Además de las expresiones no soportadas, también es necesario introducir testing para dichas expresiones.

A continuación se listaran algunos de los cambios pendientes.

- Hace falta implementar el constructor con with.
- Hace falta implementar Resume.
- Hace falta implementar Query.
- Hace falta implementar todo lo relativo a XML.
- Hace falta implementar Lambda functions.
- Existen problemas cuando se realiza el slicing en un ForEach cuya variable esté declarara fuera del statement.
- Existen problemas para realizar el slicing con un ForEach que usa la expresión Yiel.
- En el caso de los constructores base, es necesario hacerlo funcionar para los casos en que llame a una función y tomé el resultado como parámetro.
- Abreviaciones de librerías presentan problemas. Un ejemplo de esto es la de Microsoft.VisualBasic.Conversion.Int, esta función debería poder llamarse usando la abreviación Microsoft.VisualBasic.Int; sin embargo, esto provoca errores en la instrumentación.
- Si no se especifica el tipo de ciertas estructuras en su creación, ocurren problemas en tiempo de ejecución, esto afecta el resultado de los slice en diversas formas. Esto ocurre especialmente en la declaración de variables.
- La solución para el problema de los constructores base requiere generalizarse, se realizó pensando en el benchmark, y esto provoca que en niveles mayores de profundidad en el llamado de constructores base se generen problemas.

Existen otras tareas pendientes que no están recogidas en esta lista. Muchas de ellas podrán ser analizadas en mayor detalle simplemente adaptando la totalidad del test suite de C#, a Visual Basic. Actualmente, solo una parte del mismo, la cual se consideró prioritaria, fue integrada en la extensión.

Bibliografía

[1] M. Weiser. *Program slicing. In Proceedings of the 5th international conference on Software engineering.* , pages 439–449. IEEE Press, 1981.

[2] M. Harman and R. Hierons. *An overview of program slicing. Software Focus.* , 2(3):85–92, 2001.

[3] Y. Hong and X. Bao-wen. *Design and implementation of a pss/ada program slicing system.* Journal of Computer Research and Development, 34(3):217–222, 1997.

[4] E. Duesterwald, R. Gupta, and M. L. Soffa. *Rigorous data flow testing through output influences.* In Proceedings of the 2nd Irvine Software Symposium, pages 131–145, 1992.

[5] B. Korel and J. Laski. *Dynamic program slicing.* Information processing letters, 29(3):155–163, 1988.

[6] X. Zhang, N. Gupta, and R. Gupta. *Whole execution traces and their use in debugging.* The Compiler Design Handbook: Optimizations and Machine Code Generation, 2007.

[7] X. Zhang and R. Gupta. *Whole execution traces.* In Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture, pages 105–116. IEEE Computer Society, 2004.

[8] X. Zhang and R. Gupta. *Whole execution traces and their applications.* ACM Transactions on Architecture and Code Optimization (TACO), 2(3):301–334, 2005.

[9] T. Wang and A. Roychoudhury. *Using compressed bytecode traces for slicing java programs.* In Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on, pages 512–521. IEEE, 2004.

[10] T. Wang and A. Roychoudhury. *Dynamic slicing on java bytecode traces.* ACM Transactions on Programming Languages and Systems (TOPLAS), 30(2):10, 2008.

[11] J. Jamardo and V. Braberman. *Slicing dinámico para aplicaciones construidas sobre librerías y frameworks*, tesis de licenciatura, 2017

[12] Hecht, Matthew S. (1977-05-03). *Flow Analysis of Computer Programs. Programming Languages Series*, 5. Elsevier North-Holland Inc. ISBN 978-0-44400210-5.

[13] *J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization*, ACM Transactions on Programming Languages and Systems (TOPLAS), 9(3):319–349, 1987.

[14] *H. Agrawal and J. R. Horgan. Dynamic program slicing.* In Acm Sigplan Notices, volume 25, pages 246–256. ACM, 1990.

[15] *J.-F. Bergeretti and B. A. Carré. Information-flow and data-flow analysis of while-programs.* ACM Transactions on Programming Languages and Systems (TOPLAS), 7(1):37–61, 1985.

[16] *https://marketplace.visualstudio.com/items?itemName=SharpDevelopTeam.CodeConverter*

[17] *https://github.com/kirshanthans/ECE565_benchmark*

[18] *http://oot.zcu.cz/NET_2005/Papers/Full/B41-full.pdf*

[19] *C. Hammacher, K. Streit, S. Hack, and A. Zeller. Profiling java programs for parallelism.* In Multicore Software Engineering, 2009. IWMSE'09. ICSE Workshop on, pages 49–55. IEEE, 2009.

[20] *X. Zhang, H. He, N. Gupta, and R. Gupta. Experimental evaluation of using dynamic slices for fault location.* In Proceedings of the sixth international symposium on Automated analysis-driven debugging, pages 33–42. ACM, 2005.

[21] *http://www.elis.ugent.be/diablo/*

[22] *http://valgrind.org/*

[23] *X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms.* In Software Engineering, 2003. Proceedings. 25th International Conference on, pages 319–329. IEEE, 2003.

[24] *https://docs.oracle.com/javase/6/docs/api/java/lang/instrument/package-summary.html*

[25] *https://docs.oracle.com/javase/tutorial/ext/basics/load.html*

[26] *David Insa, Sergio Pérez and Josep Silva. How to construct a suite of program slices* Departamento de Sistemas Informáticos y Computación Universitat Politècnica de València E-46022 Valencia, Spain.

[27] *David Binkley, Nicolas Gold, Mark Harman, Jens Krinke, and Shin Yoo. Observation-BasedSlicing.* Loyola University and University College London. June 20, 2013

[28] *David Binkley, Nicolas Gold, Mark Harman, Syed Islam, Jens Krinke, and Shin Yoo Loyola. ORBS and the Limits of Static Slicing.* University Maryland, 4501 N. Charles St., Baltimore, MD 21210-2699, USA College London, Gower Street, London, WC1E 6BT, UK University of East London, University Way, London E16 2RD KAIST, 291 Daehak-ro, Yuseong-gu, Daejeon, 305-701, Republic of Korea.

[29] *K. Ng, M. Warren, P. Golde, and A. Hejlsberg. The roslyn project, exposing the c# and vb compiler's code analysis.* White paper, Microsoft, 2011.

[30] *B. Burg. Understanding Dynamic Software Behavior with Tools for Retroactive Investigation.* PhD thesis, University of Washington, 2015.

[31] *T. D. LaToza and B. A. Myers. Developers ask reachability questions.* In Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering-Volume 1, pages 185–194. ACM, 2010.

[32] *A. J. Ko and B. A. Myers. Extracting and answering why and why not questions about java program output.* ACM Trans. Softw. Eng. Methodol., 20(2):4:1–4:36, Sept. 2010.

[33] *G. Xu, N. Mitchell, M. Arnold, A. Rountev, E. Schonberg, and G. Sevitsky. Scalable runtime bloat detection using abstract dynamic slicing.* ACM Trans. Softw. Eng. Methodol., 23(3):23:1–23:50, June 2014.