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

Slicing dinámico para aplicaciones construídas sobre librerías y frameworks

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

Julián Osías Jamardo

Director: Victor Braberman

Buenos Aires, 2017

SLICING DINÁMICO PARA APLICACIONES CONSTRUÍDAS SOBRE LIBRERÍAS Y FRAMEWORKS

En este trabajo se presentarán cuatro herramientas de slicing dinámico, WET para código C, JSlice y JavaSlicer para código Java y NetSlicer para código C#. Se compararán las ventajas y desventajas de las características que componen a los procesos de instrumentación, tracing y slicing para cada una de ellas. Se realizará un análisis comparativo de eficiencia y precisión de cada una. Finalmente se analizará el comportamiento de las herramientas ante programas montados sobre librerías y frameworks.

Palabras claves: Análisis Dinámico de Programas, Slicing, Tracing, Profiling, PDG, DDG, PTG.

DYNAMIC SLICING FOR APPLICATIONS BUILT ON LIBRARIES AND FRAMEWORKS

In this work four tools for dynamic slicing will be presented, WET for C, Jslice and JavaSlicer for Java and NetSlicer for C#. The advantages and disadvantages of the characteristics that make up the instrumentation, tracing and slicing processes for each of them will be compared. A comparative analysis of efficiency and accuracy of each slicer will be carried out. Finally we will analyze the behavior of the tools in programs that use libraries and frameworks.

Palabras claves: Dynamic Program Analysis, Slicing, Tracing, Profiling, PDG, DDG, PTG.

Índice general

1.. Introducción 1

2.. Antecedentes y fundamentos 3
2.1. Program Slicing 3
2.2. Slicing dinámico 6
2.3. Tracing 9
2.4. Correctitud y precisión 10

3.. Herramientas 12
3.1. WET 12
3.2. JSlice 18
3.3. JavaSlicer 21
3.4. NetSlicer 23

4.. Comparativa 37
4.1. Características 37
4.2. Eficiencia 41
4.3. Frameworks y librerías 54
4.4. Precisión 63

5.. Conclusiones 65

1. INTRODUCCIÓN

La comprensión de programas es una actividad vital del mantenimiento y evaluación del software. En particular, estudios recientes sobre las preguntas que se hacen los desarrolladores [23] y algunas incipientes técnicas [21, 38] evidencian que comprender el comportamiento en tiempo de ejecución es particularmente importante. En este contexto se encuadra la problemática del desarrollo de técnicas para responder preguntas relevantes de desarrolladores y arquitectos sobre el comportamiento de una aplicación bajo análisis.

Entre las técnicas "de soporte" más prometedoras para atacar este problema se encuentra el *slicing dinámico* [22] .

El problema de slicing fue inicialmente formulado en los 80' por Weiser [36] como una técnica de análisis estático. Allí se define a un slice de un programa con respecto a un criterio (ej: una locación y una variable en el programa) como el fragmento de un programa cuyo comportamiento debe ser idéntico (o vinculable de alguna forma) al del programa original con respecto al criterio especificado. La formulación de slicing estático indica además que un slice debe ser válido para todas las ejecuciones del programa. La hipótesis principal era que en un programa reducido es más fácil encontrar y resolver errores. La técnica de computo del slicing se basa en el uso de un grafo de dependencias del programa (PDG) que registra las relaciones uso-definición y de control (ej: condicionales, iteraciones) del programa. A partir del PDG se puede calcular un slice, mediante un recorrido desde el criterio hacia el inicio del programa (backward slicing) o hacia el final del programa (forward slicing).

Esta idea, en su momento revolucionaria, generó una intensa actividad de investigación en la temática, así como su aplicación en un gran número de áreas tales como el debugging [16], análisis de impacto [18], testing [7, 17, 7], mantenimiento [4], análisis de cohesión de diseños [11], ingeniería reversa [47, 29, 6], localización de código [28], etc.

Sin embargo, el slicing estático sufre de algunos problemas que dificultan su adopción sobre todo en tareas vinculadas con la locación y comprensión de código. Al generar slices que aplican todos los posibles inputs, tiende a sacrificarse precisión a cambio de corrección, siendo en muchos casos el slice prácticamente del mismo tamaño que el programa original. Con un enfoque más pragmático Korel y Laski definen la idea del *slicing dinámico* [22]. El slicing dinámico apunta a resolver el problema de slicing para un input o test en particular, permitiendo de esta manera obtener slices mucho más precisos (especialmente útiles para localización de fallas), a costa de esta pérdida de generalidad. Estas técnicas se basan en la generación de una traza de ejecución a partir del test, para construir un PDG dinámico a partir de ella y aplicar allí el criterio de slice.

A pesar de lo prometedor del nuevo acercamiento, aparecieron rápidamente limitaciones vinculadas a la aplicabilidad de la técnicas debido a problemas de precisión y de escala [2]. En los siguientes años aparecieron trabajos que apuntan a resolver diferentes problemáticas, por ejemplo, tratar de obtener slices más pequeños [9, 31], combinar análisis estáticos y dinámicos para obtener slices generales pero aún medianamente precisos [33, 20, 26], soportar orientación a objetos [12] o sistemas fuertemente dependientes de base de datos [15], entre otros.

Como vemos el slicing dinámico ha sido un área ampliamente explorada ya que enfrenta desafíos, todavía abiertos, respecto al concepto de relevancia (no saturar al usuario/pro-

gramador con instrucciones irrelevantes), recall (tratar de captar todas las instrucciones relevantes) y de escala [30, 37]. A pesar de avances técnicos en los últimos años en las direcciones mencionadas (ej., [41, 46, 31]) el tamaño de la traza de ejecución y su posterior tratamiento suelen ser impedimentos para que el slicing dinámico sea utilizado como análisis soporte para algunas aplicaciones reales (ej., [38]). De hecho, hay también una cantidad limitada de herramientas disponibles.

Cabe destacarse que estos problemas de escala no solo alcanzan a las técnicas de slicing sino que además aplican a la mayoría de las técnicas de análisis dinámico que requieran información detallada de la ejecución, problemas que no se ha podido resolver aún [5]. Por ejemplo, este problema también es parcialmente atacado en otro tipo de análisis dinámico como es el caso del tainting con trazas de información mínima y una arquitectura desacoplada (ej., [24]) pero los desafíos conceptuales son distintos a los del slicing. Por otro lado, el problema de analizar código no disponible que termina invocando código bajo análisis está siendo recientemente atacado en la comunidad de análisis estático debido a su relevancia en los frameworks modernos de programación (ej., [32]).

En resumen, se trata de un problema que ha sido estudiado en las últimas décadas, pero sobre el que aún existen muchas limitaciones con respecto a la escala, precisión y soporte de paradigmas que se deben investigar y abordar posiblemente con aproximaciones novedosas.

El objetivo principal de este trabajo es el de comparar las herramientas de slicing dinámico más importantes que se encuentran disponibles en la actualidad (la mayoría de naturaleza académica) en aplicaciones modernas, que utilizan lenguajes orientados a objetos o similares, montadas sobre gran cantidad de código de infraestructura, librerías y frameworks (no necesariamente escritos en los mismos lenguajes de programación), o que realizan cómputo intensivo sobre grandes volúmenes de datos en memoria o bases de datos. En general, las herramientas analizadas trabajan traceando el código a nivel de assembler o bytecode y asumen que toda la actividad del código (el propio y el de infraestructura) es traceada a un bajo nivel de abstracción. Esto puede enlentecer las aplicaciones y generar enormes trazas si este mecanismo se aplica en aplicaciones con las características antes mencionadas. Para corroborar esta hipótesis, se utilizarán los slicers sobre programas que simulen aplicaciones modernas, que utilicen herramientas como Hibernate para Java, que contienen la mayoría de las características que se encuentran en frameworks contemporáneos.

El primer slicer a analizar será WET [42, 43, 40] para el lenguaje C. Lo interesante de esta herramienta, es que los involucrados en el proyecto realizaron gran cantidad de aportes a la técnica (ej., [44, 41, 25]). JSlice [34, 35] para Java, será el segundo slicer a analizar debido a que es uno de los trabajos más significativos y sentó las bases para posteriores implementaciones. Luego se analizará JavaSlicer [13], herramienta implementada recientemente, también para Java, que soporta gran cantidad de funcionalidades. Por último se presentará NetSlicer, slicer para C#, desarrollado en el marco de un proyecto de I+D+i llevado a cabo por la Facultad de Ciencias Exactas y Naturales de la UBA, del cual fui partícipe, que se diferencia del resto por no incluir información de tiempo de ejecución, sino que utiliza algoritmos inspirados en el análisis simbólico de programas.

Para cada slicer, serán detalladas las limitaciones encontradas, tanto teóricas como prácticas y se informarán detalles de precisión y performance sobre distintos sujetos.

2. ANTECEDENTES Y FUNDAMENTOS

En este capítulo serán dadas las nociones requeridas para el posterior desarrollo del trabajo. Cada concepto será acompañado con una breve explicación.

2.1. Program Slicing

Program Slicing es una técnica introducida por Mark Weiser [36]. Según la definición original de Weiser, informalmente un *slice* S de un programa P para un criterio $C = (s_i, V)$, donde s_i es el *statement* i de P y V un subconjunto de las variables de P, incluye todos los statements que afectaron a las variables en V para todos los posibles inputs del programa. Weiser define al slice S, como una proyección del comportamiento del programa original, en otras palabras que $P \equiv S$ si para todo input en el dominio de P, P y S computan los mismos valores para las variables de V en el statement s_i.

El algoritmo se basa en el flujo de control y datos, por lo que se establecen dos tipos de dependencias. Se llama *dependencia de datos* del statement s_i al efecto de las variables usadas sobre las definidas en s_i, ya que los valores de las variables definidas se modifican en función de las variables usadas. Dado un statement s_i, obtenemos el conjunto $USE(s_i)$ como las variables usadas y $DEF(s_i)$ como las variables definidas en s_i. Por cada variable v del conjunto USE de un statement, se obtiene el $LD(v)$ (o $LastDef(v)$) que es el conjunto de statments donde esa variable se modificó por última vez, s_i dependerá entonces del conjunto de statements resultado. Por otro lado, existe una *Dependencia de Control* entre un statement y una estructura de control (if, while, do-while, for, etc), si el statement se ve "afectado" por la misma o, en otras palabras, si el statement se encuentra dentro del bloque de la estructura.

Para ejemplificar la idea de slice, consideremos el algoritmo de la Fig. 2.1 y el criterio $C = (s_9, \{z\})$.

```
1 read(x);
2 read(y);
3 t = 0;
4 if x ≤ 1 then
5   t = y;
6 else
7   t = y * y;
8 v = x;
9 z = t;
```

Fig. 2.1: Ejemplo de código simple con dos caminos posibles de ejecución.

La variable z se define en s_9 usando t, que se define en s_5 y s_7 en función de y. Dependiendo del valor de x, el programa puede ejecutar el cuerpo del "then" o el del "else", de manera que es necesario suponer que t pudo haber sido definida en cualquiera

de esas líneas. s_5 y s_7 dependen de la condición del if en s_4, que usa la variable x, por lo que éste statement, debe ligarse por dependencia de datos a s_1. Luego la variable y se define en s_2. Finalmente el conjunto resultado es $\{s_9, s_7, s_5, s_4, s_2, s_1\}$

```
1 read(x);
2 read(y);
3 if x ≤ 1 then
4 |  t = y;
5 else
6 |  t = y * y;
7 z = t;
```

Fig. 2.2: Slice sobre el ejemplo de la Fig. 2.1.

El slice mostrado en la Fig. 2.2 no contiene s_8 porque v no se utiliza en el cómputo de z ni s_3 porque todo camino de ejecución que lleva a s_9 redefine t.

La técnica definida por Weiser computa el slice para todo posible input del programa, y analiza incluso código inalcanzable.

2.1.1. Backward y forward slicing

Un programa puede ser recorrido de dos formas con respecto a un criterio dado, hacia atrás (*backward*) o hacia adelante (*forward*). El cómputo del slice de manera backward, intenta encontrar todos los **statements que afectan al criterio**, mientras que el forward, todos los **statementes afectados por el criterio**. La definición de Weiser corresponde a backward slicing. Bergeretti y Carre fueron los primeros en definir la noción de Forward Slicing [3], sin embargo los primeros que utilizaron el término *forward* fueron Reps y Bricker [27].

Backward slicing es utilizado en varias áreas, como optimización, mantenimiento de software, análisis de software, pero la más común es *debugging*. Dado algún tipo de problema en un statement s_i, por ejemplo un valor erróneo en una variable v, el backward slice con criterio $C = (s_i, \{v\})$ retorna todos los statements que influyeron en el cómputo de esa variable.

Si bien también existen aplicaciones de forward slicing para debugging, en *análisis de impacto* (impacto de un cambio en el código) es donde realmente se destaca. Si existe una modificación prevista para el statement s_i, el conjunto de statements resultado del slice con criterio $C = (s_i, \{v\})$ denota donde influirán los cambios de v.

2.1.2. Program Dependence Graph

El *Program Dependence Graph* (PDG)[8] es el grafo $G = (N, D)$ que resulta de representar las dependencias de control y de datos, donde cada nodo del conjunto N representa un statement, y los ejes en D representan las dependencias tanto de control como de datos.

Esta representación es realmente útil para la técnica de slicing, ya que el slice de un programa P con PDG G, es sencillamente recorrer los nodos alcanzables desde el nodo que

representa el statement contenido en el criterio y esto se puede realizar con un algoritmo simple como BFS.

En la Fig. 2.3 se aprecia el resultado de construir el PDG para el pseudocódigo mostrado en 2.1. Las flechas punteadas representan dependencias de control y las lisas dependencias de datos. Es simple ver que obteniendo los nodos alcanzables el resultado del slice es el esperado.

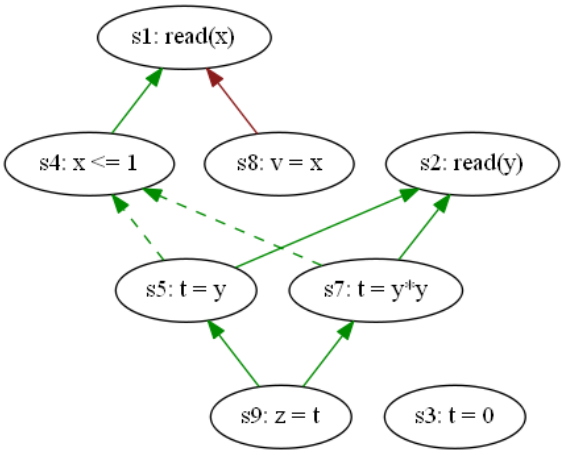

Fig. 2.3: PDG obtenido para el pseudocódigo de la Fig. 2.1.

Otra de las ventajas que tiene esta representación, es que construído el grafo, además del backward es posible realizar también forward slicing. Por ejemplo consideremos el pseudocódigo 2.4 y el PDG obtenido en la Fig. 2.5.

```
1 read(i);
2 x = 1;
3 y = 1;
4 while i ≥ 0 do
5 |   y = y * x;
6 |   x = x + 1;
7 |   i = i − 1;
8 z = y;
```

Fig. 2.4: Ejemplo de código simple con ciclo.

Supongamos que queremos visualizar el impacto de inicializar x en el statement s_2 con otro valor. Antes de computar el slice, se invierten todas las aristas y luego se realiza el recorrido desde el nodo que representa el statement modificado.

En la Fig. 2.6 se muestra el resultado de rotar los ejes. Si obtenemos todos los nodos alcanzables desde el nodo que representa a s_2, obtenemos el forward slice, con criterio $C = (s_2, \{x\})$.

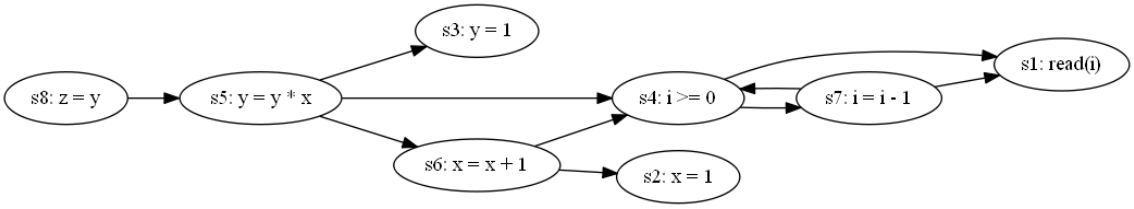

Fig. 2.5: PDG obtenido para el pseudocódigo de la Fig. 2.4.

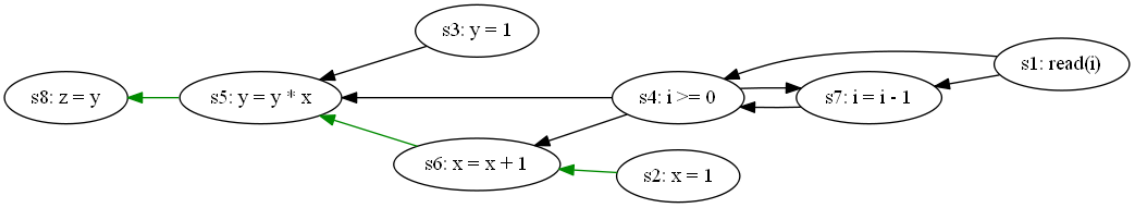

Fig. 2.6: PDG obtenido luego de invertir las aristas, para el pseudocódigo de la Fig. 2.4.

2.2. Slicing dinámico

Dynamic Program Slicing fue definido por Korel y Laski [22], en donde comentan que desde un punto de vista de debugging, donde habitualmente se trata con un input específico, no todos los caminos del código son ejecutados, por lo que el slice puede reducirse. Por ejemplo en la Fig. 2.1, si $x \leq 1$, s_7 nunca se ejercita, entonces el slice no debería contenerlo, quedando como muestra la Fig. 2.7.

```
1 read(x);
2 read(y);
3 if x ≤ 1 then
4     t = y;
5 z = t;
```

Fig. 2.7: Slice dinámico sobre el ejemplo de la Fig. 2.1.

La definición de Korel y Laski contempla un criterio $C = (X, s_i^k, V)$ y una *historia de ejecución*, donde la historia de ejecución es la lista de los statements que se ejercitaron, X es el *input* del programa y el statement s_i ahora se modifica por s_i^k que representa la aparición k en la historia.

2.2.1. Criterio de slicing dinámico

Como en el transcurso del trabajo se analizarán distintos lenguajes, es conveniente definir la notación que se utilizará para describir un criterio de slicing. Dado que se tratará siempre de slicing dinámico, se usará la notación de Korel y Laski por lo que el criterio de slicing se define como $C = (X, s_i^k, V)$, con:

- X el conjunto de inputs del programa. En casos donde el input no es relevante para el criterio de slice, el conjunto se denotará simplemente con X, en caso contrario se expresará explícitamente, por ejemplo $\{x = 1, y = 2\}$ con x e y los argumentos.

- s_i: el statement i, desde el cual se realiza el slice. Un *statement* es la porción más chica de código de un lenguaje de programación que expresa una acción. Un programa se encuentra compuesto de varios statements, que pueden estar compuestos por una o más *expresiones*, que son valores, operadores, funciones, etc, que que se evalúan para retornar otro valor. En un código assembler, una *instrucción* es un statement, mientras que en un lenguaje de alto nivel es una estructura compleja como asignaciones, declaraciones, estructuras de control, returns, etc. Por este motivo, se indicará cuál es la interpretación de statement en cada caso.

- k: la k-ésima aparición de la línea o instrucción. Se escribe $*$ en caso de requerir todas las apariciones, 0 la primera aparición y U la última.

- V: el conjunto de variables de la línea o instrucción que se desea perseguir en el análisis. Generalmente se utilizará el símbolo $*$ para denotar todas las variables en el statement. En caso contrario, el conjunto se expresará explícitamente, por ejemplo $\{a, b\}$ con a y b variables que pertenecen al programa.

Para ejemplificar, si el criterio son todas las variables contenidas en la última aparición de la línea i para un conjunto de inputs X, el criterio sería $C = (X, s_i^U, *)$, o si se desean todas las apariciones $C = (X, s_i^*, *)$.

2.2.2. Dynamic Dependence Graph

El PDG es una buena representación para hallar un slicing estático, por consiguiente es deseable poder utilizarlo en slicing dinámico. Agrawal y Horgan [2] propusieron cuatro enfoques para lograrlo (a los que llamaremos **AH-1**, **AH-2 AH-3** y **AH-4**). El primero consiste en proyectar la *historia de ejecución* del programa sobre el PDG generado previamente. El segundo, también con el PDG computado estáticamente, e imaginando las aristas como líneas punteadas, se van transformando en sólidas a medida que las dependencias aparecen la ejecución. Esto mejora la precisión con respecto al primero. Dado que los primeros dos enfoques se basan en el PDG generado estáticamente, detallarlos escapa al objetivo de este trabajo.

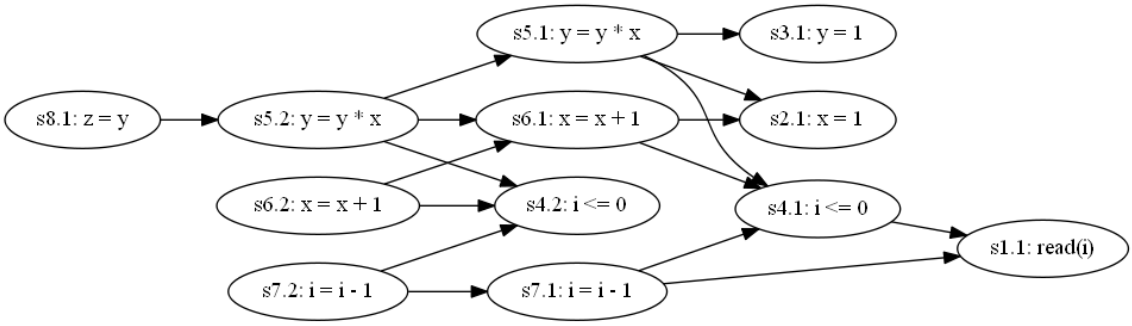

Fig. 2.8: DDG con enfoque AH-3 obtenido para el código de la Fig. 2.4 con dos iteraciones y criterio $C = (\{i = 2\}, s_8^1, \{z\})$

Para mejorar la precisión de los enfoques AH-1 y AH-2, Agrawal y Horgan propusieron el *Dynamic Dependence Graph* (AH-3). Por cada aparición de un statement en la *historia de ejecución* se crea un nodo nuevo, con sus propias dependencias. La Fig. 2.8 muestra como quedaría el DDG utilizando este enfoque para el código de la Fig. 2.4, con criterio $C = (\{i = 2\}, s_8^1, \{z\})$.

Debido a que los ciclos pueden generar grafos demasiado extensos ya que cada nodo de statement se repite por cada aparición, desarrollaron un método para comprimirlo (AH-4).

2.2.3. Slicing interprocedural y aliasing

El análisis interprocedural plantea algunos problemas no contemplados en la definición original de slicing. Uno de ellos está relacionado con los argumentos de las funciones. No necesariamente todos van a influir en valor retornado por la misma, por ejemplo si establecemos el criterio $C = (s_5, \{z\})$ para el slicing del código de la Fig. 2.9, podemos notar que z deberá depender solo de la variable x, pues en $f(a, b)$ (que se asocia al argumento y) b no participa del cómputo de a.

```
1  main:
2  x = 7;
3  y = 42;
4  z = f(x, y);
5  return z;
6
7  f(a,b):
8  b = b + 1;
9  a = a * 5;
10 return a;
```

Fig. 2.9: Ejemplo con llamado al método *f(a,b)*

Otro problema fue el reportado por Horwitz, Reps y Binkley [19], llamado *contexto de llamado* o *calling context*. El slice con criterio $C = (s_{13}, \{i\})$ sobre el código de la Fig. 2.10 debería obviar los statements s_2 y s_4 ya que estos no influyen en el criterio.

```
1  main:
2  x = 1;
3  y = 1;
4  add(x, 5);
5  increment(y);
6
7  add(a,b):
8  c = a + b;
9  return c;
10
11 increment(i):
12 i = add(i, 1);
13 return i;
```

Fig. 2.10: Ejemplo a método desde distintos contextos.

En presencia de punteros, puede ocurrir que dos variables referencien a la misma

posición de memoria. Este fenómeno es conocido como *Aliasing*. El Aliasing presenta varias dificultades a la hora de realizar slicing, ya que no aplica la definición tradicional de dependencia de datos.

```
1  int a = 0;
2  int *p = &a;
3  *p = *p + 42;
4  int b = a;
```

Fig. 2.11: Aliasing en lenguaje procedural con punteros.

Tomemos el caso de la Fig. 2.11, donde se quiere realizar el slice para el criterio $C = (s_4, \{b\})$, es evidente que existe un problema al definir el $LD(a)$. Si bien la última definición de a es en s_1, si nos atenemos a la definición tradicional de slicing, que indica que un slice se comprende de todos los statements que afectan a los valores de las variables del criterio, parece lógico suponer que el slice debería contener a s_3, ya que en éste es la última vez que se modifica el valor de a pues p es un puntero a su posición de memoria.

El problema no es sólo con punteros, sucede también en lenguajes orientados a objetos como C# o Java, por ejemplo en la Fig. 2.12 se realiza aliasing entre $a1$ y $a2$, por lo tanto $a1.field$ se modificó por última vez en s_3.

```
1  A a1 = new A();
2  A a2 = a1;
3  a2.field = 4;
4  Integer x = a1.field;
```

Fig. 2.12: Aliasing en lenguaje orientado a objetos.

Si bien el aliasing y el análisis interprocedural conllevan problemas complejos para el slicing estático, en el contexto de slicing dinámico pueden resolverse con mayor facilidad. Agrawal, DeMillo y Spafford [1] definieron las dependencias en términos de direcciones de memoria. Esto tiene la ventaja de no ser necesario el análisis de aliasing ni se presentan los problemas interprocedurales. Un enfoque similar es el de Wang et al. [34], donde debido a que en Java las posiciones de memoria para un objeto no son absolutas, utilizan identificadores globales.

Cuando se presenten las herramientas utilizadas, se explicará en detalle cómo resuelven estos problemas cada una.

2.3. Tracing

El *Tracing* es una herramienta utilizada en Ingeniería de Software para recolectar información sobre la ejecución de un programa mediante un tipo especial de logging. Normalmente es utilizada por desarrolladores para debugging, administradores de sistemas y por programas de monitoreo de software para auto diagnosticar problemas. Si bien no existe una distinción clara entre logging y tracing, salvo por el hecho de que no se utiliza tracing como una particularidad funcional de un programa (logging de errores, datos, etc) sino como una herramienta ortogonal.

En Slicing Dinámico es fundamental conocer los statements que se ejecutan del programa tratado, por lo tanto Tracing es una herramienta indispensable para cumplir este objetivo, pues la información generada (que llamaremos *traza*) será los statements ejercitados en tiempo de ejecución. Es importante notar que esta herramienta no es necesaria en Slicing Estático, puesto que se realiza utilizando el programa original sin necesidad de ejecutarse.

Es claro que el software del cual queremos obtener el slice, no genera la traza requerida. Por eso es necesario someterlo a una técnica denominada *instrumentación*, que puede verse como el proceso de insertar código en el programa original modificándolo con un fin específico. Naturalmente, en el caso de Slicing el fin es emitir traza.

El programa se puede instrumentar a alto nivel, como C, C# o Java) o bajo nivel como lenguajes intermedios, Assembler, Bytecode o CIL. Existen ventajas y desventajas con cada uno.

- **Acceso a todo el código**: Con lenguajes de bajo nivel se puede acceder a todo el código que ejecuta el programa, que resulta una ventaja debido a que permite obtener toda la información dinámica necesaria para el cómputo del slice, sin embargo se genera una gran cantidad de traza. Por el contrario, el tracing sobre lenguajes de alto nivel genera menor cantidad de traza, pero la imposibilidad de analizar código de terceros genera la necesidad de deducir el efecto de ese código sobre el slice.

- **Mapeo entre la traza y el fuente**: El tracing de código de bajo nivel genera un problema de trazabilidad entre el código trazado y el fuente original cuando el código que se analiza es de alto nivel. Si por el contrario, se instrumenta y traza el mismo lenguaje que se analiza el problema no sucede.

- **Estructuras sintácticas complejas**: Al instrumentar niveles de abstracción altos surgen problemas en el manejo de sintaxis compleja como expresiones lambdas, iteradores, etc. Por el contrario, los lenguajes de bajo nivel son simples de analizar y contienen una sintaxis mucho más reducida.

A medida que veamos diferentes implementaciones, detallaremos las decisiones tomadas con respecto a la instrumentación y el formato y tratamiento de la traza.

2.4. Correctitud y precisión

En la definición de Slicing Estático propuesta por Weiser y la de Dinámico por Korel y Laski el slice resultante debe ser un programa **ejecutable**, no obstante la mayoría de las técnicas e implementaciones que se presentarán a lo largo de este trabajo, **no** genera un slice compilable. Por ejemplo consideremos la porción de código de la Fig. 2.13 escrito en C++, tomando como criterio la definición de z en la línea 7, x se define dentro del cuerpo del then y el else, por lo que la definición previa (statement s_1) quedaría fuera del slice, pero si la sacamos, está claro que el código no compilaría.

El slice ejecutable tiene la ventaja de que permite verificar la **correctitud** de un slice dinámico bajo la definición original corroborando que tanto el slice como el programa computen los mismos valores del criterio para el input especificado. No obstante, para aquellas técnicas que no retornan una solución ejecutable, es necesario definir otro criterio de correctitud. Una posibilidad es establecerla en función de las dependencias de datos,

```
1  int x = 0;
2  if (cond) {
3      x = y * y;
4  } else {
5      x = sqrt(y);
6  }
7  int z = x;
```

Fig. 2.13: Porción de código compilable escrito en lenguaje C++.

es decir, un slice es correcto si contiene todas las dependencias de datos que afectan las variables del criterio. Aún así esta última definición puede variar, ya que las dependencias de datos pueden definirse de distintas formas, por ejemplo en el caso de la Fig. 2.14 ¿Cuál sería el slice dinámico para el criterio $C = (X, s_4^0, \{d\})$? Algunas implementaciones retornan los statements $\{s_1,s_2,s_3,s_4\}$, sin embargo no es trivial responder si realmente influyen las líneas 1 y 2, pues no queda claro el impacto real de la creación de A y B en el valor final de la variable d, por lo que no sería incorrecto retornar solo $\{s_3, s_4\}$.

```
1  var a = new A();
2  a.b = new B();
3  a.b.c = new C();
4  var d = a.b.c;
```

Fig. 2.14: Ejemplo de múltiples acceso a campos en lenguaje C#.

Dado que Weiser [36] demostró que hallar el slice mínimo de un programa sería equivalente a resolver el *Halting Problem*, es necesario encontrar un slice que sea correcto y que contenga la **mínima cantidad posible de statements irrelevantes** para el criterio dado. A esto lo llamaremos *precisión*.

Es importante notar que el hecho de que sea ejecutable o no, impacta directamente en la precisión, ya que como se aprecia en el ejemplo anterior, la declaración de la variable no aporta información relevante al slice. De igual manera sucede con el ejemplo del algoritmo de la Fig. 2.14, donde incorporar o no ciertas líneas con sus dependencias, puede tener un impacto significativo en el tamaño del resultado.

Los slicers se pueden clasificar como **precisos** o **imprecisos**. Esta distinción no está asociada a las decisiones de precisión en la inclusión explícita o no de líneas debido a los factores anteriormente mencionado, sino al análisis del programa en su totalidad o no, pues si se decide evitar alguna parte del programa (como puede ser código de terceros), es necesario realizar un análisis (que debe ser correcto pero que habitualmente es sobre-estimado) del impacto que puede generar ese código sobre el slice. Esto se verá más claro cuando se analicen las herramientas en el capítulo siguiente.

3. HERRAMIENTAS

En este capítulo se presentarán las cuatro herramientas de slicing dinámico más significativas.

La primera herramienta a analizar será *WET* para código *C*. Lo interesante de esta herramienta, es que los autores realizaron grandes aportes a la técnica y los subsiguientes slicers se basan en sus trabajos. *JSlice* para *Java*, será el segundo slicer analizado debido a que es uno de los trabajos más significativos y al igual que WET, sentó las bases para otras implementaciones. Luego se analizará *JavaSlicer*, herramienta implementada recientemente, basada en JSlice, también para Java, que puede utilizarse en software contemporáneo y que demostró soportar gran cantidad de funcionalidades. Por último se introducirá NetSlicer, herramienta que permite realizar slicing en código *C#*.

Durante este capítulo, cada herramienta contendrá las siguientes secciones:

- **Información general**. Se explicarán brevemente detalles sobre la arquitectura, la infraestructura requerida, el modo de compilación y la forma de uso de la herramienta.
- **Tracing**. Se explicarán los detalles sobre el proceso de tracing, instrumentación y ejecución del programa instrumentado.
- **Slicing**. Se introducirá el algoritmo de slicing, así como las estructuras soporte (DDG, mapas de memoria, etc.) utilizadas.

3.1. WET

El proyecto *WET*[1] es una infraestructura de software de código abierto, que es capaz de rastrear y analizar ejecuciones de programas largos, con personalización, extensibilidad y lo más importante, la capacidad de recopilar distintos tipos de trazas dinámicas para ejecuciones realistas en programas de uno o más threads.

Lo interesante de esta herramienta es que la información y tipo de traza obtenida, le otorga la capacidad de realizar Program Slicing.

La decisión de incorporar WET en este trabajo se debe a que los involucrados en el proyecto (Dennis Jeffrey, Vijay Nagarajan, Rajiv Gupta, Xiangyu Zhang, William Nicholas Sumner, Min Feng, Chen Tian, Yan Wang y Tao Bao) han realizado gran cantidad de aportes en Slicing y Tracing.

3.1.1. Información general

La arquitectura presentada en WET es similar a la utilizada por Zhang et al. [45], y en sucesivos trabajos ([10],[39]). En la Fig. 3.1, se muestran los componentes principales de la misma.

Esta implementación realiza un análisis estático previo para determinar dependencias de control que luego se usarán en el análisis dinámico. Este análisis previo se realiza con

[1] http://wet.cs.ucr.edu/index.html

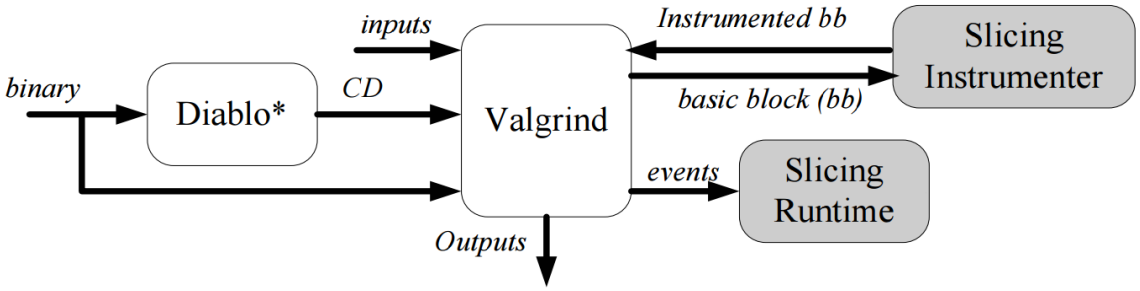

Fig. 3.1: Arquitectura de WET. (Figura extraída de "Pruning Dynamic Slices With Confidence" [39]).

la herramienta Diablo [2]. Luego de calcular la información estática, el programa se ejecuta utilizando Valgrind [3] que instrumenta el binario llamando al Slicing Instrumenter y luego lo ejecuta. En la herramienta WET la instrumentación se realiza gracias a la modificación de la herramienta Lackey [4] incorporada en Valgrind. La ejecución del binario instrumentado lanza distintos eventos, como entradas a funciones, accesos a memoria, operaciones binarias, etc) que serán procesados como callbacks en el motor de Slicing. Además utilizan la técnica de *shadow memory* [45] (que se explicará más adelante), utilizada para manejar las dependencias dinámicas. La información recolectada se guarda como en una estructura similar a un DDG.

La última versión obtenible desde la web proporcionada por los autores requiere el Kernel de Linux 2.6.9-5.0.3.EL. Cualquier versión posterior no permite compilar el Valgrind, porque utiliza algunas particularidades del kernel que desaparecen en nuevas versiones. Además utiliza libc 2.3.4. Nuevamente, cualquier versión posterior genera múltiples errores en la compilación. Para satisfacer los requerimientos se descargó el sistema operativo Centos 4.0, una distribución lanzada el 09/03/2005, lo que demuestra la antigüedad de la herramienta.

El proceso para realizar el slice es el siguiente:

- Compilar utilizando un gcc modificado por los autores, con símbolos de debug (-g), enlace estático (-static) y generación de archivo .map (-Map, genera un mapa de posición de memoria a posición del código).
- Correr Diablo. Utilizando el mapa generado en el primer paso, genera información estática y computa las dependencias de control.
- Correr Valgrind. Existen distintas opciones que se explicarán más adelante.

Como mencionan los autores y se aprecia en la Fig. 3.1, de no ser necesarias las dependencias de control, es posible evitar correr Diablo, por lo que no es necesario utilizar el gcc modificado con la inyección estática de librerías. No obstante evitar estos pasos puede generar un slice incorrecto, como se verá posteriormente.

[2] http://www.elis.ugent.be/diablo/
[3] http://valgrind.org/
[4] http://valgrind.org/docs/manual/lk-manual.html

WET cuenta con dos modos de ejecución y la posibilidad de realizar el slice o no. Los modos de ejecución son los siguiente:

- **Modo exhaustivo**: Realiza una traza "legible" ya que las dependencias se presentan cómo archivos y número de líneas. Esta traza, si bien puede ser muy extensa, es posible seguirla visualmente.
- **Modo limitado**: Contiene sólo dependencias entre direcciones de memoria, pero resulta en una traza mucho más compacta. En este modo además se utiliza un **buffer circular** (que será explicado más adelante) con una cantidad finita de dependencias.

La usabilidad de WET se ve afectada por la complejidad en su ejecución y en la legibilidad de los resultados. El modo exhaustivo retorna una traza que puede ser leída sin inconvenientes. El limitado, sin embargo, tiene el siguiente formato:

```
0x8048242#0 --> 0x8048210#0
0x8048242#0 --> 0x8048225#0
```

Es claro que este formato no es apto para la lectura, lo que resulta en una limitación para WET, el problema es que, como se mostrará más adelante, no es posible utilizarlo en modo exhaustivo debido a graves problemas de performance. Por lo tanto, con el fin de poder utilizar la herramienta, se realizó un proceso de traducción. Utilizando el programa *addr2line*[5], que dado un binario y una dirección de memoria, retorna la línea a la que corresponde, fue posible traducir las dependencias de manera legible.

Si se desea que la herramienta compute el slice, es necesario pasarle un parámetro de la forma `--slice=INSTRUCTION_INSTANCE`. Traduciendo esto a la definición de criterio original propuesta por Korel y Laski, el criterio de slice se define como $C = (X, I^n, *)$
donde I es la **dirección de memoria** correspondiente a la instrucción de la cual se desea partir y n la instancia de esa instrucción (n-ésima aparición). Por ejemplo con `--slice=0x08048248_0` el programa realiza el slice a partir de la dirección `0x08048248` en su primera aparición.

Nuevamente, esto es una limitación en la usabilidad del programa, ya que requiere que el usuario utilice un desensamblador para, dado el binario que quiere analizar, obtener la dirección de memoria correspondiente al archivo y línea de interés.

Incluso el resultado del slice es un problema. El programa escribe el resultado en un archivo que sólo contiene las direcciones de memoria como se ve en el siguiente ejemplo:

```
Computed slice: 4 instruction instances in slice
Instruction instances in slice:
   0x8048213#0
   0x8048236#0
   0x8048239#0
   0x8095B6E#0
```

Para facilitar el uso se realizó un programa que dado el binario, el archivo donde se encuentra el código fuente que se desea analizar, la línea y la aparición (instancia), traduce

[5] https://linux.die.net/man/1/addr2line

la línea a las direcciones hexadecimales correspondientes y utiliza la última de ellas para ejecutar luego el slicer.

Primero se utiliza el GDB[6] (debugger GNU) corriendo el comando *info line*, por ejemplo:

```
(gdb) info line while.c:11
Line 11 of "while.c" starts at pc 0x8048245 and ends at 0x804824b.
```

Este comando retorna el principio y el final en hexadecimal de las direcciones de memoria que componen esa línea, pero es necesario obtener la dirección de las instrucciones. Para ello se utiliza el comando *disas* de la siguiente forma:

```
(gdb) disas 0x8048245 0x804824b
Dump of assembler code from 0x8048245 to 0x804824b:
0x08048245 <main+85>:   mov    0xfffffff8(%ebp),%eax
0x08048248 <main+88>:   mov    %eax,0xfffffff0(%ebp)
End of assembler dump.
```

Por último se extrae la última dirección obtenida (en el ejemplo `0x08048248`) y se corre la herramienta.

Es importante mencionar que este proceso se apoya en la suposición de que la última dirección de memoria extraída del programa *disas*, depende de las anteriores. En la práctica esto es correcto para sentencias del tipo asignación o declaraciones de variables, sin embargo si por ejemplo, el criterio es un statement que sólo contiene un *println*, ninguna de las direcciones obtenidas por *disas* se puede utilizar para realizar el slice, ya que el programa indica que las direcciones no se encuentran en la traza. Por este motivo, todos los criterios se establecieron sobre asignaciones.

Otro problema relacionado a la compilación es la necesidad de utilizar enlace estático, por consiguiente se deben indicar todas las librerías de manera explícita para que éste las introduzca en el binario. Para aplicaciones reales esto sugiere mucho trabajo, ya que una librería que se incluye, podría enlazar dinámicamente muchas otras. Para ejemplificar, con un programa que utiliza la librería de acceso a base de datos PostgreSQL **libpq**[7] con enlace dinámico sólo es necesario agregar `-lpq`, mientras que con compilación estática hay que agregar `-lpq`, `-lssl`, `-lkrb5`, `-lcrypto`, `-lk5crypto`, `-lcom_err`, `-lcrypt` y `-lresolv`. Esto puede no parecer un problema que obstaculice el uso de la herramienta, sin embargo es habitual que los programas utilicen varias librerías dinámicas, y el programador no suele conocer todas las que a su vez cada una de ellas utiliza, por lo que puede ser un proceso bastante complicado. A esto, se suma el hecho de que para compilar se suelen utilizar herramientas como **Autotools**[8], por lo que modificar el proceso de compilación de una herramienta de gran escala puede ser muy complicado.

Sin compilación con enlace estático **Diablo**, herramienta que se debe correr antes de Valgrind para computar las dependencias de control, no funciona correctamente. No obstante, como muestra la Fig 3.1 del capítulo anterior, es posible saltear ese paso, por

[6] https://www.gnu.org/software/gdb/
[7] https://www.postgresql.org/docs/7.4/static/libpq.html
[8] https://www.gnu.org/software/automake/manual/html_node/Autotools-Introduction.html

lo que la compilación estática dejaría de ser una condición necesaria. De hecho, Diablo es la herramienta que limita WET al lenguaje C, ya que Valgrind trabaja sobre assembler, por lo que también es capaz de trazar aplicaciones C++. El problema de saltear el paso de Diablo es que puede traer problemas de correctitud.

```
1  #include "foo.h"
2  int main(int argc, char* argv[]) {
3          struct employee e1;
4          e1.age = atoi(argv[1]);
5          int count = 0;
6          if (e1.age > 18)
7                  count++;
8          int z = count;
9          return 0;
10 }
```

Fig. 3.2: Ejemplo de uso de librería en lenguaje C.

El código de la Fig. 3.2 utiliza una librería que define la estructura *employee*. Utilizando enlace estático el slice resultante sobre el criterio $C = (\{argv[1] = \text{“19”}\}, s_8^0, \{z\})$ es $\{s_2, s_4, s_5, s_6, s_7, s_8\}$, mientras que con enlace dinámico, como imposibilita correr Diablo desaparece la dependencia de control entre la línea 7 y la 6, y por consiguiente la dependencia con la línea 4 (la más relevante pues es la causa de que se incremente la variable *count*) resultando en el conjunto $\{s_2, s_5, s_7, s_8\}$.

3.1.2. Tracing

A continuación se explicarán características relacionadas con el proceso de Tracing de WET.

La instrumentación en WET se realiza a través de la modificación de Valgrind, que acepta binarios compilados con GCC, los instrumenta llamando al componente Slicing Instrumenter, y ejecuta el código instrumentado utilizando el módulo Slicing Runtime. El núcleo de Valgrind es un instrumentador dinámico por sí mismo, que toma el binario y antes de ejecutar cualquier bloque de instrucciones (que no haya sido instrumentado) llama a la función de instrumentación, que es provista por el Slicing Instrumenter, se instrumenta, retorna al núcleo, se guarda en memoria (para poder ser usado nuevamente sin llamar al instrumentador) y se ejecuta.

El trabajo que le da el nombre a la herramienta es el realizado por Xiangyu Zhang y Rajiv Gupta, *Whole Execution Traces* [42] [43]. Existen incontables trabajos realizados en distintos tipos específicos de tracing (Control flow, Value profiles, Address profiles, Dependence profiles).

WET tiene como objetivo unificar estas técnicas y generar una traza que pueda responder a cualquier tipo de análisis, pues ésta es una representación en la cual subyace el historial de ejecución completo, control flow, valores de variables, direcciones de memoria y dependencias tanto de control como de datos. Es esencialmente una representación estática del programa, enriquecida con información obtenida dinámicamente.

WET se encuentra modelado con un grafo de la forma G(N, E(CF, CD, DD)), donde N son los statements ejecutados del programa, E es el conjunto de ejes **bidireccionales**

que pueden ser dependencias de datos (DD), dependencia de control (CD) o dependencias de control de flujo (CF). Es importante notar la similitud de esta representación con un grafo de dependencias dinámico, pues será utilizada directamente en el proceso de slicing.

Nagarajan et al. [25], presentaron **ONTRAC** (ONline TRACing for Debugging), una herramienta que permite computar las dependencias dinámicas de manera "online", es decir a medida que el programa se ejecuta. Al realizar el procedimiento de esta manera, el cómputo de las dependencias y la escritura en un archivo en disco puede aumentar notablemente el tiempo de ejecución con respecto al programa original. Por esta razón, los autores decidieron utilizar un **buffer circular** de dependencias, que limita la cantidad de dependencias que en la historia de ejecución se pueden guardar. El inconveniente es que esta técnica tiene la desventaja de perder dependencias cuando el buffer se llena, sin embargo como ONTRAC se concentra en utilizar slicing como herramienta de debugging, los autores explican que regularmente, dado un problema, los errores en el código se encuentran dentro de una ventana reciente de statements ejecutados.

Es importante aclarar que WET **no** computa las dependencias de manera "online" como lo hace ONTRAC, sin embargo tiene implementado el buffer circular para evitar el crecimiento excesivo de la traza en disco.

El proceso de tracing de WET cuenta con la técnica de *Shadow Memory* para establecer dependencias entre direcciones de memoria, que consiste en el mapeo de bytes pertenecientes a una memoria invisible para el programa original, a bytes de la memoria principal. Por cada byte reservado en el stack o el heap del espacio virtual de la aplicación, se reserva un byte correspondiente en el nuevo espacio, pudiendo accederlo con la misma dirección virtual. Esta técnica fue incorporada en la implementación propuesta por Zhang et al. [45] y en susivos trabajos ([39, 25]).

Es importante ver que debido al uso de la shadow memory, las dependencias no se establecen en función de los nombres de las variables sino con las posiciones de memoria, o sea que utiliza el mismo enfoque de Agrawal et al.[1]. Por este motivo la traza de WET no contiene el flujo de control del programa, sino que retorna directamente las dependencias entre direcciones. Una dependencia de datos existe entre dos instrucciones si una de ellas usa un valor definida por la otra. Esto se representa por un par de tuplas de la forma:

$$(instr_id_{use}, instance_{use}) \rightarrow (instr_id_{def}, instance_{def})$$

Existe una dependencia de control entre dos instrucciones cuando una de ellas (el predicado de control), controla la ejecución de la otra. Esto se representa como:

$$(instr_id_{use}, instance_{use}) \rightarrow (instr_id_{predicate}, instance_{predicate})$$

Notar que esta representación es exactamente el output del programa, ya que se compone de la dirección de la instrucción y la instancia en que fue ejecutada.

3.1.3. Slicing

La representación WET explicada anteriormente, contiene toda la información dinámica necesaria para computar slices dinámicos. Según indican los autores, la versión compacta (limitada) puede quedar guardada en memoria y usarse para computar slices dinámicos

en tiempos razonables en programas grandes. Dado un valor calculado mediante la ejecución de una instrucción de código, el *WET Slice*[42, 43] es un backward slice sobre la representación WET a partir del valor de interés. Este slice captura el flujo completo de control, el flujo de valores a través de las dependencias y las referencias de direcciones que directa o indirectamente afectaron el cálculo del valor de interés. Por lo tanto, un WET Slice proporciona un superset de la información proporcionada el enfoque tradicional de Korel y Laski.

3.2. JSlice

El slicer dinámico JSlice, es la materialización del trabajo de Tao Wang y Abhik Roychoudhury [34], en donde hacen hincapié en el formato de representación de la traza. Es el primer trabajo en utilizar un método de compresión sobre la traza y en generar el slice leyendo esta información, sin necesidad de descomprimirla.

3.2.1. Información general

La herramienta JSlice[9] es una modificación de una máquina virtual de Java. Esta arquitectura le proporciona dos ventajas:

- No instrumenta ni realiza tracing. En lugar de esto, monitorea los bytecodes ejecutados.
- Cómputo de dependencias "online". A medida que ejecuta los bytecodes, realiza el cómputo de dependencias.

La VM seleccionada fue Kaffe[10] (proyecto inactivo desde el año 2011), debido a que la máquina virtual más usada, de la empresa SUN (hoy Oracle) no se encontraba publicada con licencia de código abierto. Esto presenta dos problemas de infraestructura. El primero es que Kaffe se encuentra escrito en C y la herramienta proporcionada por Wang et al. solo compila con GCC 2.95 y ejecuta sobre una distribución específica de Linux (Fedora 4) que se encuentra ya en desuso. El segundo problema es la antigüedad de la versión de Kaffe utilizada, que es compatible con la JDK de Oracle hasta la versión 1.4.

El uso de la herramienta es realmente simple. Primero se debe crear un archivo que puede contener más de un criterio de slice, con el número de línea, el nombre del archivo y si es la última aparición o todas, es decir que permite dos tipos de criterio, $C = (X, s_i^U, *)$ para computar el slice a partir de la última aparición o $C = (X, s_i^*, *)$ que incluye en el resultados todos los slices de todas las apariciones. Luego se debe compilar el programa con el compilador modificado y por último se corre con el runtime modificado agregando el parámetro `-slicing` y el archivo de criterios.

Además JSlice cuenta con un plugin para Eclipse que facilita la realización y visualización del slice. Lamentablemente no funciona correctamente cuando se incluyen librerías externas.

[9] http://jslice.sourceforge.net/
[10] http://www.kaffe.org/

3.2.2. Tracing

El proceso de tracing no es como el del resto de las herramientas ya que en lugar de realizar una traza con información dinámica y flujo de control para su posterior análisis, al ser una modificación de la VM es capaz de monitorear directamente los bytecodes que se interpretan.

La información generada durante la ejecución se va guardando en memoria. El problema es que no escala, debido a que las trazas sobre bytecodes tienden a ser muy grandes, por este motivo Wang et al.[34] propusieron comprimirla. Además, para no tener que escribirla a disco y poder seguir trabajando en la memoria, realizaron una extensión al algoritmo SEQUITUR[11] al que llamaron Run-Length Encoding SEQUITUR (RLESe), que permite comprimir la traza on-the-fly, es decir a medida que se va generando. Más aún, lograron que el proceso posterior de Slicing, realice un recorrido backward sin necesidad de descomprimirla.

La traza de bajo nivel le otorga a JSlice la posibilidad de evitar ciertos bytecodes que no influyen en el slice, por lo tanto necesitan sólo los siguientes tres tipos de bytecodes para reconstruir el flujo de control:

- Bytecodes de invocaciones a métodos. Los bytecodes invokevirtual e invokeinterface pueden ejecutar diferentes métodos en diferentes instancias de ejecución. Estos métodos tienen exactamente el mismo nombre y firma, pero pertenecen a diferentes clases, debido al polimorfismo. Por eso, cada vez que se ejecuta alguno de estos métodos se traza la **clase efectiva del método que se ejecuta**.
- Bytecodes de acceso a memoria. A diferencia de WET, en Java no es posible trazar direcciones de memoria pues no siempre el objeto al que apunta una variable se encuentra en la misma dirección, por esto, para los bytecodes que acceden a campos de los objetos o arrays, **se traza un identificador del objeto**, lo que resuelve el problema de aliasing. Para las variables locales, como son referenciadas utilizando el índice dentro del stack frame del llamado al método que las contine, se genera una simulación del stack agregando un nuevo marco a la pila cuando se entra a un método y se sacándolo cuando se sale. Vale aclarar que JSlice no soporta excepciones, donde este método falla (Al hablar de JavaSlicer se volverá sobre este asunto).
- Bytecodes de flujo. Para cada salto condicional, se traza el bytecode que se ejecuta inmediatamente después.

3.2.3. Slicing

La solución de JSlice es similar al algoritmo preciso **NP** (No Preprocessing) introducido por Zhang et al. [44] autores de la herramienta WET (que no forma parte de WET por ser desestimado posteriormente por los mismos autores, ej., [41]). El algoritmo NP no computa el grafo dinámico de dependencias en su totalidad para disminuir el consumo de memoria. En lugar de eso las dependencias de datos dinámicas se extraen bajo demanda de la traza mientras se realiza el proceso de slicing. Por lo tanto, cada vez que se examina la traza, sólo se extraen las dependencias relevantes al slice que se calcula. El procedimiento consiste en recorrer la traza de atrás hacia adelante (backward) e ir capturando las dependencias que

[11] http://www.sequitur.info/

influyen en el criterio de slice dado, si v es la variable del criterio se recorre la traza hasta encontrar su última definición y se incluye en el conjunto resultado, luego si está definido en función del valor de otra variable w, ésta también se incluye y se recorre el slice hasta encontrar su última definición y así sucesivamente.

Dado que la traza es una secuencia de bytecodes ejecutados, el criterio establecido en el fuente Java, se traduce como $C = (X, \alpha, V)$, con X el conjunto de inputs, α conjunto de bytecodes y V conjunto de variables incluidas en esos bytecodes.

En primer lugar se computa el grafo de dependencias de control (CDG) estáticamente. A medida que se realiza el slicing se mantiene una lista δ de variables cuyos valores necesitan ser "explicados" (es necesario obtener las definiciones de las variables usadas para su última definición), un conjunto γ de bytecodes en los que es necesario buscar si pertenecen a alguna estructura de control y el conjunto resultado del slice ϕ. Se recorre la traza y por cada bytecode se realizan las siguientes operaciones:

- Si es la última ocurrencia del bytecode β y éste pertenece a α, entonces se insertan en δ las variables utilizadas en β (ya que hay que buscar sus definiciones) y el bytecode en γ y ϕ.
- Usando el CDG, se verifica si algún bytecode que se encuentra en γ depende por control de β y en caso afirmativo, se incorpora en la solución ϕ.
- Si la variable definida en β aparece en δ se agrega en γ y ϕ y se actualiza δ removiéndola e insertando las usadas.

Además del aporte de la compresión de traza, Wang et al. [34], incorporaron un análisis interesante sobre la técnica de slicing en el contexto de debugging. El slice computado por JSlice contiene líneas no contempladas en la definición original que pueden ser fundamentales para hallar un error, por ejemplo en la Fig. 3.3, el slice con criterio $C = (X, s_7, \{x\})$ retorna $\{s_7, s_3\}$, no obstante si el valor de x esperado es 2, el error se podría encontrar en el valor de a o b cuyas definiciones no pertenecen al resultado. Para incluir estas lineas los autores agregaron una característica que permite computar "dependencias potenciales" lo que permite agregar el statement s_2 en el slice.

```
1 a = 0;
2 b = 1;
3 x = 1;
4 if a < 1 then
5 |   if b < 1 then
6 |   |   x = 2;
7 ... = x;
```

Fig. 3.3: Fragmento de programa con error. (Figura extraída de "Using Compressed Bytecode Traces for Slicing Java Programs" [34]).

Este es un caso claro de decisiones de precisión, en el que se agregan líneas no por ser un algoritmo impreciso en sí, sino con otros fines.

3.3. JavaSlicer

JavaSlicer [14], realizado por Clemens Hammacher en su tesis de licenciatura en el año 2008 es una herramienta para Java basada en JSlice.

3.3.1. Información general

El principal aporte de JavaSlicer[12] yace en la usabilidad de la herramienta. Clemens Hammacher[13], identificó las limitaciones de JSlice con respecto a la arquitectura (modificación de una VM de Java) y presentó una implementación alternativa que permite que su slicer pueda ejecutarse con software contemporáneo, utilizando la VM de Oracle y en cualquier sistema operativo.

Para lograr la independencia de la máquina virtual, JavaSlicer utiliza *Java Agents*[13] que permiten instrumentar el bytecode de la clase antes de ser ejecutada. Las clases se cargan bajo demanda utilizando *Class Loaders*[14]. Una vez que el class loader obtiene el bytecode, se llama al framework de instrumentación que llama a "transformadores" registrados previamente. Los Agents, que se agregan como argumentos de la VM, tienen la posibilidad de agregar transformadores. Lo interesante de los transformadores es que pueden agregar, modificar y quitar métodos, campos, interfaces implementadas, etc.

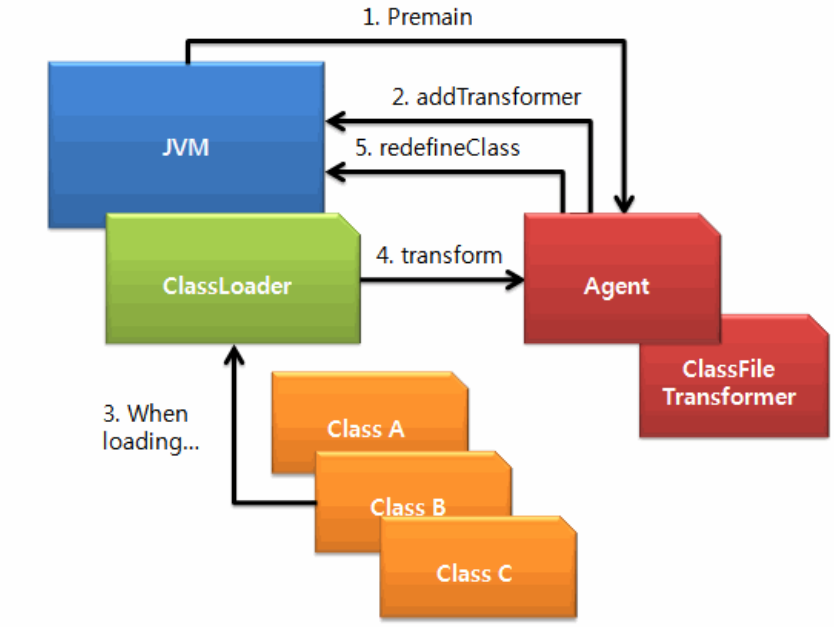

Fig. 3.4: Procedimiento de aplicación de Agents en máquina virtual Java.

El procedimiento completo se aprecia en la Fig. 3.4, se carga el Agent por argumento, éste se ejecuta al comienzo y agrega los transformadores necesarios, luego los class loaders al cargar las clases antes de transformar el bytecode en código máquina llaman a los transformadores para finalmente terminar cargados en la JVM.

[12] https://github.com/hammacher/javaslicer
[13] https://docs.oracle.com/javase/6/docs/api/java/lang/instrument/package-summary.html
[14] https://docs.oracle.com/javase/tutorial/ext/basics/load.html

El uso de la herramienta es aún más sencillo que el de su predecesor. Se debe realizar un paquete jar con el programa a analizar. Luego se ejecuta el runtime que se tenga instalado incorporando el agent que realiza la traza con el parámetro `-javaagent` y el nombre del archivo de traza. Por último se ejecuta el slicer con la traza como argumento.

3.3.2. Tracing

JavaSlicer utiliza Java Agents para instrumentar el bytecode de todas las clases que se cargan a medida que se ejecuta el programa. Esto incluye a las clases contenidas en la JRE, pues con instrumentar las clases incorporadas por el usuario, no es suficiente para poder trazar toda la ejecución del programa.

El proceso de instrumentación inserta nuevos bytecodes antes o después de ciertas instrucciones, que tienen como objetivo escribir un archivo con la traza, que contiene toda la información necesaria para computar el slice dinámico.

Es importante considerar cualquier salto en la ejecución, es decir, la ejecución de dos instrucciones que no son vecinas directas en el bytecode. Los diferentes tipos de saltos son: instrucciones de saltos incondicionales (goto, jsr, excepciones, invocaciones, etc), instrucciones de saltos condicionales (ifnull, ifle, etc). El enfoque tomado por JavaSlicer está basado en el hecho de que cada vez que sucede un salto en en flujo del bytecode, siempre continúa a partir de una etiqueta. A cada instrucción se le asigna un identificador y cada vez que se atraviesa una etiqueta, se guarda el identificador de la instrucción que se ejecutó justo antes. De esta forma, es posible reconstruir la secuencia de bytecodes que se ejecutaron de forma **backward**. El problema con este enfoque es que si bien resuelve los saltos incondicionales, condicionales, el salto hacia la instrucción que corresponde a un método no se realiza a través de una etiqueta, no obstante se resuelve agregando etiquetas al principio del cuerpo de los métodos y luego de cada instrucción de invocación.

JavaSlicer comparte las mismas características que su predecesor JSlice, por consiguiente en el proceso de Tracing la información extraída por JavaSlicer es similar a la extraída por el segundo, se traza la clase efectiva de las invocaciones para resolver el problema del dispatch dinámico y se trazan identificadores de los objetos de manera de seguir los usos y definiciones de los mismos en ejecución. Para las variables locales utiliza una simulación del stack, sin embargo a diferencia de JSlice, soporta excepciones. Como cuando se lanza una excepción, el stack puede saltar varios frames y saltear la instrumentación que envía información de flujo de control JavaSlicer agrega en todos los métodos, un **try** que engloba todo el cuerpo del método y un **catch** que permite trazar la excepción, restablecer el stack, para luego volver a lanzar la excepción.

JavaSlicer implementa varios tipos de compresión, entre ellos una modificación de SEQUITUR para luego recorrerla de manera backward sin descomprimir.

3.3.3. Slicing

El procedimiento de slicing es esencialmente el mismo que JSlice, primero se instrumenta y ejecuta el programa extrayendo la traza comprimida, luego el slice se construye recorriendola de atrás hacia adelante. Así como su antecesor computa un CDG estático previamente que será usado para corroborar dependencias de control, en JavaSlicer se computa un *Control Flow Graph* por cada método y se guardan las dependencias para cada instrucción.

3.4. NetSlicer

NetSlicer es una herramienta de slicing para código *C#* desarrollado en la Facultad de Ciencias Exactas de la Universidad de Buenos Aires, cuyo objetivo es el de poder manejar aplicaciones reales, es decir, aplicaciones desarrolladas normalmente en la industria, las cuales contienen uso extensivo de frameworks, librerías externas, bases de datos, etc.

Dado que en el contexto de debugging los desarrolladores no examinan los fuentes de terceros (por no estar disponible o simplemente por ser confiable), el proceso de slicing puede acotarse al código conocido. Es en este punto donde yace la principal diferencia entre NetSlicer y las herramientas antes mencionadas.

3.4.1. Información general

La herramienta consta de dos procesos: el programa original modificado, y otro que llamamos "backend". El *Backend* se compone de dos etapas. En la primera etapa **instrumenta** el código fuente del programa que quiere analizar y lo **compila**. En la segunda **ejecuta** el programa modificado, obtiene la traza que genera, la procesa y realiza el slicing. Esta etapa puede ser **diferida** o **a demanda**. En la modalidad diferida el programa original se ejecuta completamente, **escribe la traza a disco** y luego el backend la procesa. En la modalidad a demanda, el programa instrumentado escribe la traza en un stream que el backend consume y procesa en simultáneo (similar al enfoque de Nagarajan et al. [25]). Esta última modalidad admite PIPE o socket UDP.

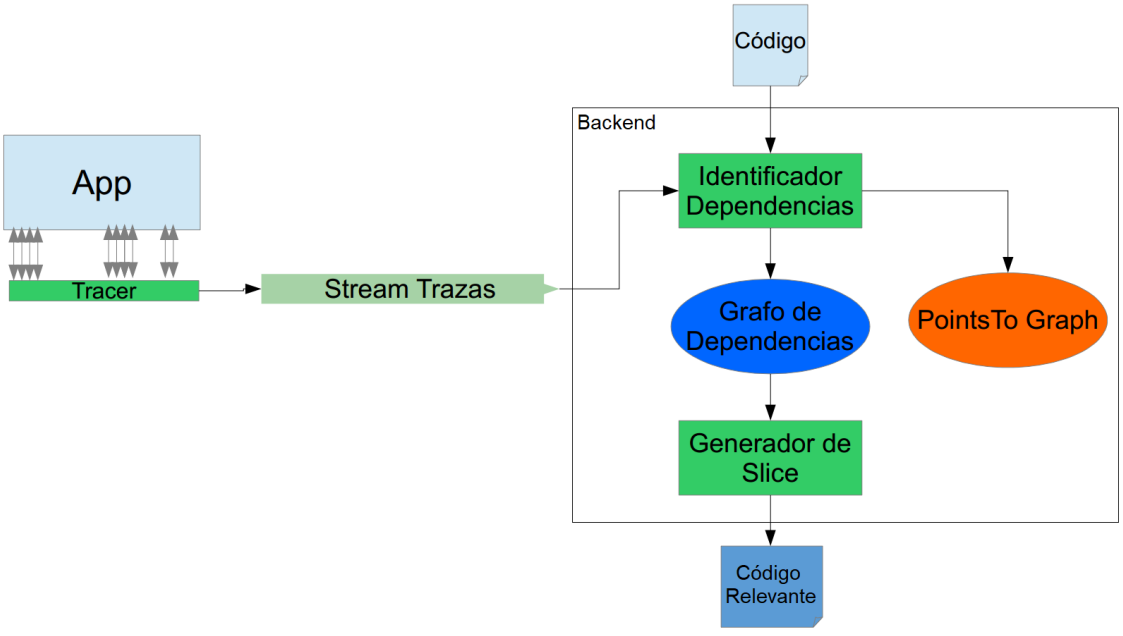

Fig. 3.5: Arquitectura de NetSlicer.

En el lado izquierdo de la Fig. 3.5 se identifica el programa modificado (App) que llama al *Tracer*. Éste escribe la traza en el Stream. Sobre el lado derecho se encuentra el backend y los elementos que lo componen. El *Identificador de Dependencias* es el núcleo de NetSlicer, utilizando el código original del programa, la traza y una estructura soporte

PointsTo Graph (que se explicará en el transcurso de esta sección), se construye el *Grafo de Dependencias* (PDG), que será utilizado para el cómputo del slice.

Tanto la instrumentación como el análisis del procesamiento de la traza se realiza con el framework *Roslyn* [15]. Roslyn provee compiladores .Net con APIs de análisis de código, las mismas que son utilizadas por versiones recientes de la IDE oficial de Microsoft Visual Studio. Utilizando estas herramientas, al instrumentar se recorre el AST agregando en el árbol sintáctico los llamados al Tracer y se compila la sintaxis modificada. En la segunda etapa, el programa modificado envía la información necesaria para que backend, también usando Roslyn y el código fuente original, identifique la porción de código que se va a ejecutar para poder analizarla.

3.4.2. Tracing

Roslyn provee herramientas para visitar y modificar el código. El procedimiento de instrumentación consiste en recorrer el AST y por cada statement s_i, de cada bloque de código (estructura sintáctica entre llaves) se agrega otro s_i' antes de s_i, que llama al Tracer con información de s_i.

En la traza se escriben tuplas de la forma $< FID, SS, SE, TT >$. FID es el el identificador del archivo (File ID) que se utiliza para poder obtener el AST del código que se ejecuta. SS (Span Start) es el índice de la posición del primer caracter del comienzo de la porción de código que se va a ejecutar (siendo la posición cero el comienzo del archivo) y SE (Span End) corresponde al índice final de la porción de código, con estos dos valores, se obtiene la porción de sintaxis que se va a analizar. El último entero en la tupla TT (Trace Type) es el tipo de traza que se envía, que lo determina el nombre del método llamado en el tracer (ej. si el método es TraceSimpleStatement, la información en TT será "SimpleStatement"). El tipo de traza es de suma importancia ya que determina el flujo del programa, ésta puede ser un statement normal de un bloque de código (SimpleStatement), la entrada a un método de instancia (EnterMethod), la entrada a un bloque de una estructura de control (EnterCondition), las entradas a bloques **catch** en el caso de excepciones (EnterCatch), entre otros, por ejemplo en la Fig. 3.7 se observa el resultado de instrumentar el método "Method" de la Fig. 3.6.

```
1  static void Method()
2  {
3    int x = 7;
4    int i = 0;
5    while (i < 3)
6    {
7      x = x * x;
8      i++;
9    }
10   int y = x;
11 }
```

Fig. 3.6: Ejemplo de método completo en lenguaje *C#*.

[15] https://github.com/dotnet/roslyn

```
1  static void Method()
2  {
3    try
4    {
5      Tracer.TraceEnterStaticMethod(1,176,408);
6      Tracer.TraceSimpleStatement(1,232,242);
7      int x = 7;
8      Tracer.TraceSimpleStatement(1,256,266);
9      int i = 0;
10     while (Tracer.TraceSimpleStatement(1,280,373) && (i < 3))
11     {
12       Tracer.TraceEnterCondition(1,280,373);
13       Tracer.TraceSimpleStatement(1,326,336);
14       x = x * x;
15       Tracer.TraceSimpleStatement(1,354,358);
16       i++;
17       Tracer.TraceExitCondition(1,280,373);
18     }
19     Tracer.TraceSimpleStatement(1,387,397);
20     int y = x;
21     Tracer.TraceExitStaticMethod(1,176,408);
22   }
23   catch
24   {
25     Tracer.TraceEnterFinalCatch(1,176,408);
26     throw;
27   }
28 }
```

Fig. 3.7: Resultado de la instrumentación del ejemplo de la Fig. 3.6.

Como se comentó previamente, cuando se computan las dependencias de un statement se obtienen los *LastDef* con los nombres de las variables que se usan. El problema que conlleva esto, es que en diferentes métodos, los identificadores se pueden repetir, más aún los nombres de las variables que se pasan como argumentos, habitualmente difieren de los nombres de los parámetros declarados. Es por esto que se mantiene una pila de llamados a métodos paralelo al *stack* de ejecución. Ante un *EnterMethod*, los nombres de los argumentos se deben ligar a los nombres de los parámetros y se apilan las estructuras necesarias. Al recibir *ExitMethod* o procesar un *return*, se realiza el proceso inverso. Por otro parte, esta traza es utilizada para determinar si el método al que se va ingresar es o no es instrumentado. Si se está analizando una porción de código que contiene un llamado a un método, se puede observar si el próximo elemento en la traza es un EnterMethod, en caso de serlo, se realiza el procedimiento antes explicado, en caso contrario, se asume que no pertenece al código fuente.

En lenguajes con excepciones como *C#*, una excepción lanzada en un método, puede saltar varios *frames* del *stack* hasta llegar a un bloque **catch** que la maneje. La excepción evita que se envíe el ExitMethod instrumentado, generando una inconsistencia en la pila de métodos simulada. Para evitar este problema, se implementó la misma idea realizada en JavaSlicer, a todos los métodos se les agrega el try-catch, de forma que si se lanza una excepción, se puede enviar la información necesaria para que el backend modifique la pila

adecuadamente.

3.4.3. Slicing

El enfoque que adopta NetSlicer puede verse como **híbrido**, ya que dinámicamente se obtiene la historia de ejecución, pero no se envía información dinámica como identificadores de objetos, direcciones de memoria o valores de las variables, sino que se envía sólo información de control de flujo, para luego realizarse un análisis estático sobre el código fuente. A medida que se realiza el análisis se va construyendo un grafo de dependencias.

El PDG construido por el backend es el Dynamic Dependence Graph (DDG) de Agrawal y Horgan, en la versión AH-3 o AH-4. Para programas reales, el uso de AH-3 es inviable por el alto consumo de memoria, por lo que habitualmente se utiliza AH-4, pues la compresión de éste es prácticamente imperceptible en los tiempos totales de cómputo del slice. El grafo es global para toda la ejecución del programa y cada nodo representa una línea del código fuente. El uso del DDG le permite a NetSlicer realizar tanto **backward** como **forward** slicing.

Dado que no utiliza información dinámica como identificadores de objetos o direcciones de memoria, para solucionar los problemas de aliasing, NetSlicer implementa una estructura de soporte llamada points-to graph (PTG). A partir del PTG se construye el DDG, ya que cada vértice del grafo de points-to, contiene una lista de *LastDefs* (nodos del DDG) de cada campo o propiedad que se define, de manera que para obtener la última vez que se definió un término del tipo acceso a miembro (ej. $a.b.c$), se debe recorrer el grafo de points-to.

```
1  A a = new A();
2  B b = new B();
3  A c = a;
4  c.f1 = b;
5  b.f2 = c;
6  d = a.f1;
```

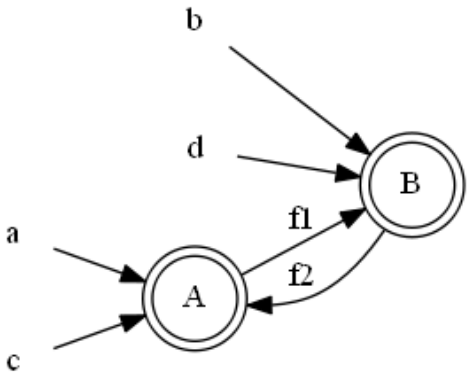

Fig. 3.8: Ejemplo de PTG (derecha) sobre el código *C#* (izquierda).

El lado derecho de la Fig 3.8 muestra el PTG del código de la izquierda. Los nodos del grafo representan los objetos en memoria. Las variables a, b, c y d son puntos de entrada al grafo y los campos o propiedades de los objetos que apunten a otros objetos son ejes. En el ejemplo, en la primera línea se crea el nodo que representa el objeto del tipo A y su punto de entrada es la variable a. De la misma manera sucede con la línea 2. En la 3, se genera aliasing entre a y c, por lo que c pasa a apuntar al objeto que apuntaba a. En la línea 4, se genera aliasing del objeto apuntado por b, al campo $f1$ del objeto que apunta c, por lo que se agrega un eje con etiqueta $f1$ entre el objeto del tipo A y el del tipo B. Ídem en la línea 5. En la línea 6 se requiere obtener el objeto al que que apunta $a.f1$, usando a

como punto de entrada al grafo y navegando por la etiqueta $f1$ se obtiene el objeto B.

Al igual que el DDG, el PTG es global a toda la ejecución del programa. Por esto, al entrar a un método nuevo, los puntos de entradas al PTG locales al método llamador se eliminan y se **simula aliasing** entre los argumentos y los parámetros de entrada y entre el objeto retornado (en el caso que exista) y la variable asignada en el método llamador.

En la Fig. 3.9 en el llamado a M se genera aliasing entre a y el parámetro p de manera que en la ejecución del método apunte al objeto creado en la línea 3 de $Main$. Al salir de M, a la variable $b1$ se le asigna el resultado del método, por lo que debe apuntar al objeto del tipo B creado en la línea 10. En la figura se aprecia el efecto del método M sobre el PTG. El objeto $B1$, creado en el statement 4, deja de ser alcanzable por lo que debe ser recolectado por el garbage collector. Si se establece como criterio de slice $C = (X, s_6^U, \{b2\})$, se debe obtener la última vez que se definió $a.f$. Recorriendo el PTG se obtiene que $a.f$ no apunta a $B1$ sino a $B2$, creado en el statement 10 y asignado al field f en 11, por lo que el slice debe retornar las líneas 10 y 11 que causan este efecto y no la línea 4.

```
1  static void Main()
2  {
3      A a = new A();
4      a.f = new B();
5      B b1 = M(a);
6      B b2 = a.f;
7  }
8  static B M(A p)
9  {
10     B b = new B();
11     p.f = b;
12     return b;
13 }
```

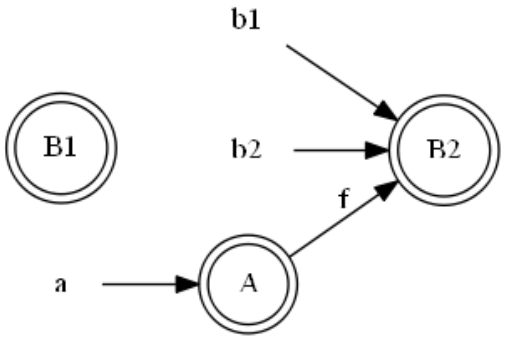

Fig. 3.9: Ejemplo de PTG (derecha) sobre el código C# (izquierda) con invocación a método.

La Fig. 3.10 contiene el pseudocódigo para el algoritmo para obtener los nodos del PTG a partir de una variable o acceso a miembro. Dada una expresión de acceso a un campo o propiedad (*memberAccessExp*), se obtiene el nombre de la variable del objeto con la función *Var*, que funciona de punto de entrada al PTG y los campos (con la función *Fields*) que pueden ser uno o varios (ej. $\text{Var}(a.f.g.h) = a$, $\text{Fields}(a.f.g.h) = \{f, g, h\}$). Usando el punto de entrada i obtenido con *Var* se obtiene el conjunto I de nodos a los que apunta, que se asigna a la solución final S. Luego se recorren los campos obtenidos con *Fields* en orden. Por cada campo f se construye un conjunto C con todos los nodos que serán alcanzables por el campo f desde los nodos en S. Por lo tanto, por cada nodo n en la solución, se obtiene el conjunto de todos los nodos adyacentes a n por el campo f y se une a C. Luego la solución pasa a ser esos nodos alcanzados C. Al terminar de iterar sobre los campos, el conjunto S contendrá todos los nodos que representan los objetos a los que puede apuntar la expresión analizada.

La Fig. 3.11 refleja cómo va quedando el conjunto S en cada paso en un grafo aleatorio para $a.b.c$. Como se puede ver, inicialmente se obtienen los nodos utilizando el punto de

```
1  i = Var(memberAccessExp);
2  F = Fields(memberAccessExpr);
3  I = InitNodes(PTG, i);
4  S = I;
5  for f in F do
6      C = ∅;
7      for n in S do
8          A = Adjacencies(PTG, n, f);
9          C = C ∪ A;
10     S = C;
11 return S;
```

Fig. 3.10: Algoritmo de obtención de nodos a partir de un acceso a miembro.

entrada y cada vez que se mueve a través de un campo, el conjunto solución se modifica, quedando finalmente los objetos a los que apunta esa expresión.

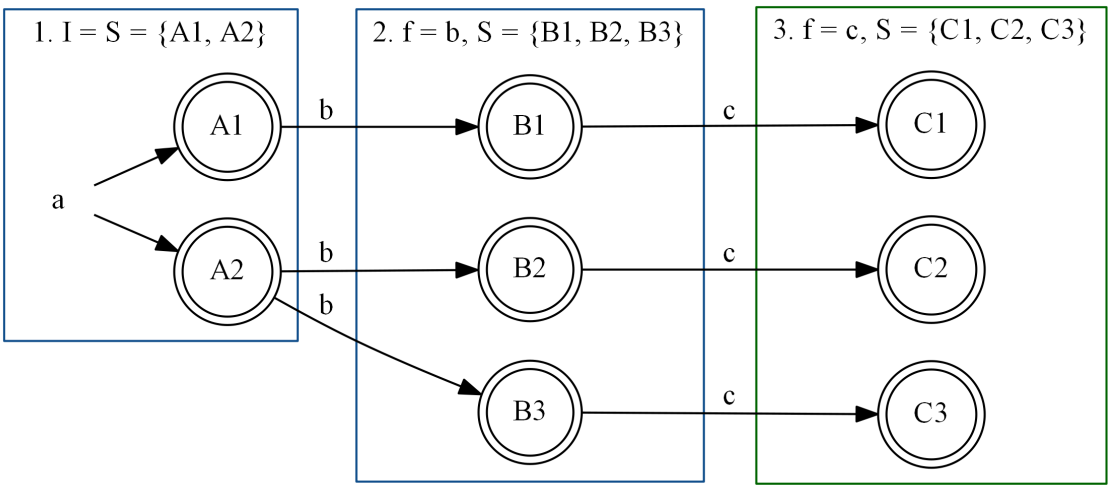

Fig. 3.11: Representación gráfica del algoritmo de obtención de nodos para un grafo aleatorio y acceso a campo del tipo *a.b.c*.

El PTG se construye fácilmente y otorga una forma práctica de representar la memoria, sin necesidad de obtener información dinámica. No obstante la existencia de código no instrumentado genera un problema, ya que no se conoce el efecto del método sobre el PTG. Por ejemplo siendo el método M de la Fig. 3.9 **no instrumentado** (no hay traza alguna sobre ese método), si no se toma consideración alguna sobre el efecto del método sobre el PTG, $a.f$ continuaría apuntando a $B1$, conduciendo a problemas de correctitud. Un inconveniente que tiene la idea tradicional de PTG es que no contempla el efecto de métodos cuyo código se desconoce. En estos casos se agrega un nuevo tipo de nodo que llamaremos *Hub*. Los "hubs" representan regiones de la memoria que son desconocidas por el PTG y se asocian al llamado al método no instrumentado. En estas regiones podrían

existir otros objetos que se pueden ligar mediante campos y propiedades con los objetos pasados por argumento, lo que obliga a considerar todo posible aliasing para mantener correctitud. La Fig. 3.12 es el resultado del PTG para el código de la Fig. 3.9 asumiendo *M* no instrumentada.

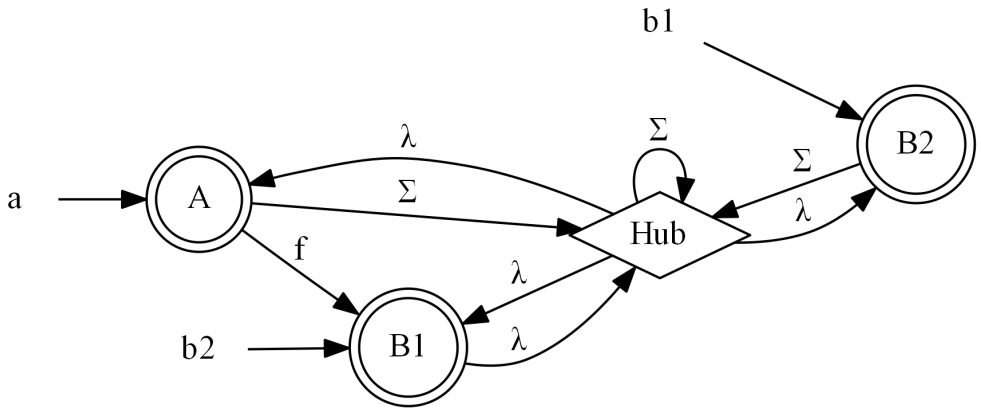

Fig. 3.12: PTG para 3.9 con *M* no instrumentada.

El tratamiento de no instrumentado agrega dos tipos de ejes nuevos en el PTG, λ y Σ. Las aristas de tipo Σ representan todos los campos o propiedades del objeto que pueden haber sido modificados, mientras que las aristas de tipo λ representan un posible aliasing entre dos objetos, es decir que si dos nodos están ligados por un eje λ, toda navegación que dé como resultado uno, debe retornar también el otro. Es importante notar que un método no instrumentado, no solo puede modificar los campos de los objetos pasados como argumentos sino también los campos de esos campos, por tal motivo, existe un bucle Σ en el hub.

```
 1 i = Var(memberAccessExp);
 2 F = Fields(memberAccessExpr);
 3 I = InitNodes(PTG, i);
 4 S = LambdaClosure(I);
 5 for f in F do
 6 |   C = ∅;
 7 |   for n in S do
 8 |   |   A = Adjacencies(PTG, n, f);
 9 |   |   A_Σ = Adjacencies(PTG, n, Σ);
10 |   |   A' = A ∪ A_Σ;
11 |   |   C = C ∪ A';
12 |   S = LambdaClosure(C);
13 return S;
```

Fig. 3.13: Algoritmo de obtención de nodos del PTG agregando ejes λ y Σ.

En la Fig. 3.13 se encuentra el pseudocódigo del algoritmo para la obteción de nodos

del PTG extendido a partir de una variable o acceso a miembro. La modificación más importante sobre el algoritmo original es la clausura λ y las búsquedas de nodos adyacentes a través de las aristas Σ. El cuerpo del algoritmo en esencia es similar al original. Antes de ingresar al ciclo se realiza la función *LambdaClosure*, que realiza la clausura transitiva por los ejes λ, sobre los nodos iniciales, debido a que en situaciones como las de la Fig. 3.12, si se estuviera realizando el recorrido a partir de *b2*, como podría ser tanto el objeto *B1* como algun otro creado perteneciente a la región *Hub*, hay que obtener ambos y sin la clausura inicial, el hub no se incluiría. El resto del procedimiento es similar, por cada campo f de la expresión se obtienen los nodos adyacentes por *f*, pero además se incorporan todos los adyacentes por Σ.

Asumiendo M no instrumentada para el código de la Fig. 3.9, $a.f$ puede seguir siendo el objeto instanciado en la línea 4, pero puede haberse modificado en M por lo que el slice deberá mostrar ambas líneas, por lo tanto al momento de obtener $a.f$, primero se utiliza a como punto de entrada y luego se navega por Σ y por el campo f, obteniendo el nodo $B1$, creado en la línea 4 y el hub, que representa la invocación de la línea 5.

El eje Σ posee la ventaja de representar aquellos campos que no necesariamente fueron asignados en el código instrumentado. Para ejemplificar, asumiendo M no instrumentada y **eliminando la línea 4**, en la línea 6, se accede a $a.f$ que no se asigno en el código instrumentado, pero puede haber sido inicializada en M, por lo que el slice debe retornar la línea 5 (el llamado a M) que se obtiene retornando el nodo hub al recorrer el PTG. Por lo tanto al momento de obtener $a.f$, primero se utiliza a como punto de entrada y luego se navega por Σ.

La creación de objetos (método *new*) posee un efecto ligeramente diferente en el PTG. Ante la creación de un objeto cuyo tipo es no instrumentado se crea el objeto conocido por el código analizado, y se agrega una región. Como cualquier campo o propiedad del objeto se pudo haber modificado en la región se agrega un eje Σ desde el nuevo objeto hacia la región y dado que pudo haberse generado aliasing con un objeto de la región se agrega λ en sentido inverso. La Fig. 3.14 ejemplifica este efecto al ejecutar "`var a = new A()`", con la clase A no instrumentada.

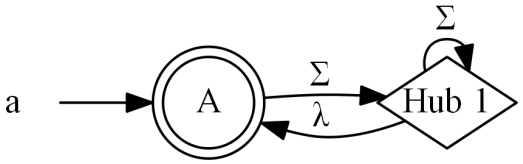

Fig. 3.14: PTG resultante de la creación de un objeto cuya clase es no instrumentada.

En ciertos casos, la creación de un Hub por método no instrumentado puede resultar en un PTG muy extenso, por ejemplo, en la Fig. 3.16 se aprecia el PTG que resulta de la ejecución del código de la Fig. 3.15 que consta de un simple ciclo (en este caso de dos iteraciones) que agrega elementos a una lista. Dado que List es un objeto que pertenece al framework .NET, se genera en el grafo el efecto de una construcción de objeto no instrumentada. Luego en la primera llamada al método no instrumentado Add se crea el *Hub 2* usando *list* y *tmp* como los parámetros, y ligando todos sus alcanzables (en este caso el *Hub 1*) con λ ida y vuelta contra *Hub 2*. El procedimiento en el segundo Add es

análogo, se crea el *Hub 3* y se liga con el conjunto alcanzable (que contiene al *Hub 1* y al *Hub 2*) por λ en ambas direcciones.

```
1  public static void Main()
2  {
3      var list = new List<A>();
4      A tmp = null;
5      for(var i = 0; i < 2; i++)
6      {
7          tmp = new A();
8          list.Add(tmp);
9      }
10 }
```

Fig. 3.15: Ejemplo de reiterados llamados a métodos no instrumentados en *C#*.

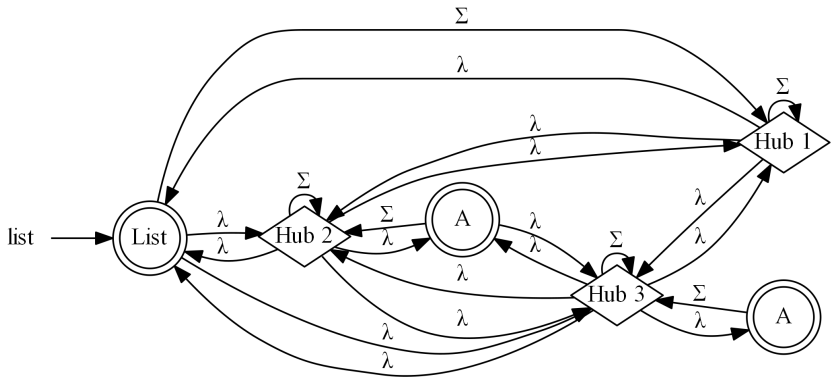

Fig. 3.16: PTG resultante del análisis del código de la Fig. 3.15.

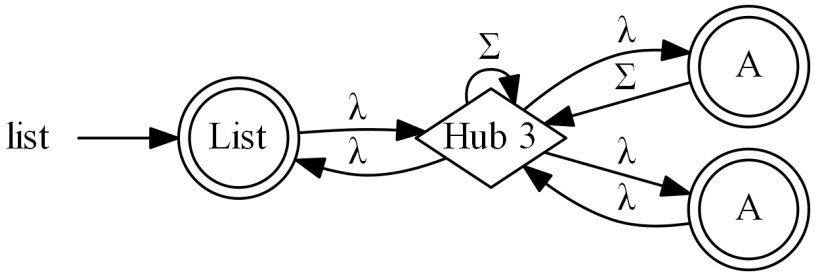

Fig. 3.17: PTG resultante del análisis del código de la Fig. 3.15 luego de la compresión.

.

Dado que el crecimiento del grafo degrada el algoritmo de recorrido, el PTG cuenta con un algoritmo de compresión basado en la observación de que si dos hubs se conectan por λ tanto de ida como de vuelta, ambos se pueden comprimir en uno, debido a que al realizar la clausura siempre se van a obtener ambas regiones. El procedimiento es simple, cada vez que se crea un nuevo hub y este se va a asociar a otro, se elimina el previo incorporando sus ejes al nuevo. La Fig. 3.17 muestra el grafo con la compresión.

La utilización de hubs conlleva a un problema de tipos que no existe en el PTG original. Dado el código de la Fig. 3.18 con la clase Binary perteneciente a una librería no instrumentada, el efecto del llamado al constructor de Binary con *a* es que podría asignarse *a.f* a *a*, a algún objeto creado en el llamado, o al mismo objeto *bin* retornado (Notar que no se tiene en cuenta tipos de datos). Por lo tanto al navegar con *a.f* sobre el PTG (Fig 3.19) se obtiene la región y los objetos *A*, *B1* y *Binary*. Sin embargo, es claro que *a.f* que es de tipo *B*, no puede ser ni *A* ni *Binary*.

```
1  public void Method()
2  {
3      A a = new A();
4      a.f = new B();
5      Binary bin = new Binary(a);
6      object o = a.f;
7  }
```

Fig. 3.18: Ejemplo de llamado a método no instrumentado en *C#*.

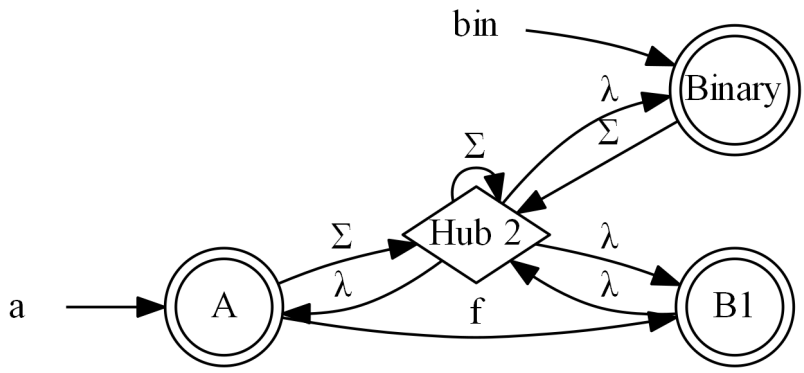

Fig. 3.19: Estado del PTG luego de la ejecución del statement 5, para el código de la Fig. 3.18.

.

Apoyados en el compilador .Net de Roslyn que infiere los tipos de los símbolos, se modificó tanto la función *Adjacencies* como *LambdaClosure* del algoritmo de recorrido del PTG de la Fig. 3.13, para que filtre nodos utilizando la jerarquía del tipo que tiene el término por el cual se recorre el grafo. Como los hubs representan muchos objetos se asume que es de tipo **object**. De esta forma, los objetos que se asignan a *o* luego de la línea 4 del ejemplo de la Fig. 3.18, son sólo el *Hub 2* y *B1*, como muestra la Fig. 3.20.

Es normal que un método no instrumentado perteneciente a un framework o librería usada, genere callbacks sobre el código analizado, por ejemplo la ejecución de un método que inicie un formulario sobre un framework GUI, genera tantos callbacks como acciones del usuario sobre la interfaz.

Dado que NetSlicer no analiza código multithread, todo método no instrumentado se asume sincrónico, por lo que es posible identificar el método que generó el callback. De esta forma, cuando se ejecuta un método correspondiente a un callback, los parámetros, el objeto receptor y el objeto que retorna el callback (en caso de que exista) pueden participar

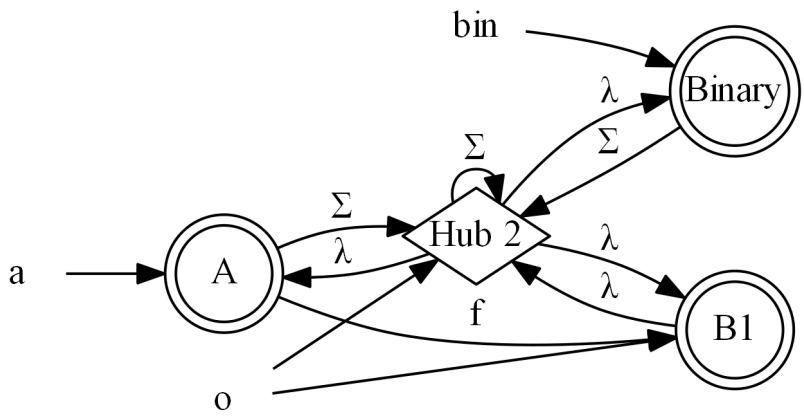

Fig. 3.20: Estado del PTG luego de la ejecución del statement 6, para el código de la Fig. 3.18.

.

posteriormente de operaciones de aliasing con los argumentos y el objeto receptor del método no instrumentado llamado desde el código analizado que genera el callback.

El código de la Fig. 3.21, es una extensión del ejemplo 3.18, con una llamado a un método no instrumentado que genera un callback, que contempla varios elementos que se pueden dar en un escenario de callback. Binary es una clase definida en una librería que no se instrumenta, y la clase A implementa el método Callback de una interfaz definida también en la librería. Cuando se envía el mensaje NonInstrumentedMethod a *bin* la librería ejecuta Callback.

```
1  public void Method()
2  {
3      A a = new A();
4      a.f = new B();
5      Binary bin = new Binary(a);
6      bin.NonInstrumentedMethod(new B());
7      object o = a.f;
8  }
9
10 /*** In class A ***/
11 public object Callback(object arg)
12 {
13     this.f = (B)arg;
14     return new B();
15 }
```

Fig. 3.21: Ejemplo de llamados a método no instrumentado con callback en *C#*.

Para el criterio de slicing $C = (X, s_7^U, *)$, se debe obtener la última vez que se definió *a.f*, por lo que los objetos a los que podría apuntar son:

1. En primer objeto B asignado a *a.f*. Si bien en el cuerpo del callback, el campo *f* se sobrescribe, no necesariamente el objeto receptor del mensaje Callback es *a*, la librería

podría haber creado otro objeto A, por lo tanto este objeto es un posible candidato.

2. Algún objeto creado en el constructor de Binary. La librería podría crear uno o más objetos y asignarlos al campo, ya que posee una referencia de *a*. De igual forma sucede con el llamado al método no instrumentado.

3. El segundo objeto B, pasado por parámetro al método no instrumentado.

4. Los objetos que representan el parámetro *arg* del método Callback asignado a *this.f*.

5. El último objeto B retornado por el método Callback. La librería podría asignar a *a.f* al objeto retornado.

Hasta la ejecución del statement 5, el grafo es el mostrado anteriormente en la Fig. 3.19. Se crea la región representada por el *Hub 2*, en la invocación del *new* que ejecuta el constructor no instrumentado de Binary.

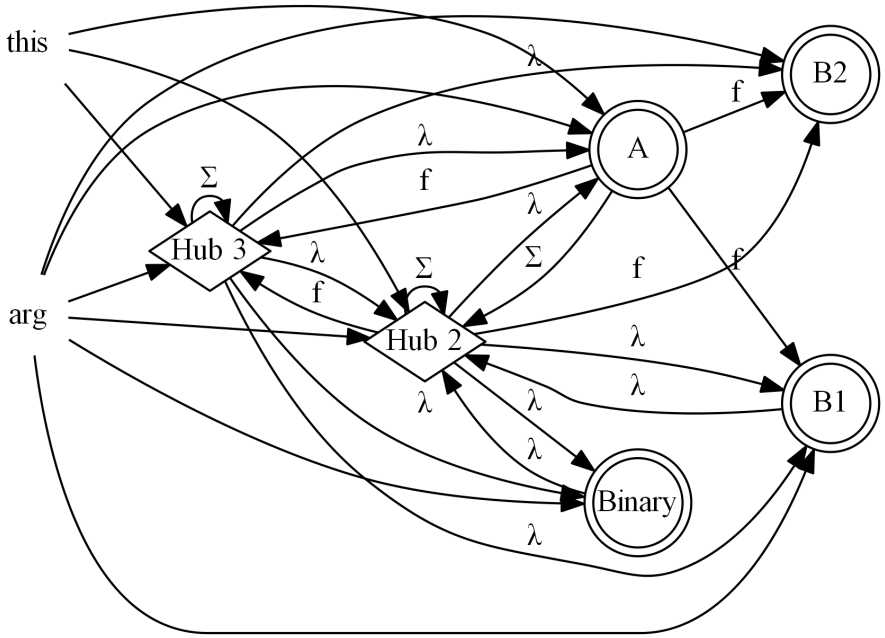

Fig. 3.22: Estado del PTG luego de la ejecución del statement 13, para el código de la Fig. 3.21.

Dado que un método o constructor no instrumentado puede ejecutar callbacks, se utiliza el flujo de control para saber si el próximo statement a ejecutar es el inmediato siguiente dentro del mismo bloque de código, o corresponde a un método nuevo.

Por ejemplo, en la ejecución del statement 6 del código de la Fig. 3.21, el siguiente elemento de la traza es el *EnterMethod* del método Callback, por lo tanto inicia el manejo del callback sobre el PTG. En este punto se crea una nueva región (*Hub 3*), que representa las modificaciones sobre el estado de la memoria realizadas por la librería entre el llamado a *NonInstrumentedMethod* y el callback. Este hub, debe apuntar por lambda a todo lo alcanzable por el this (*bin*) y los argumentos (el nuevo objeto B creado antes de llamar al método), pues podría existir aliasing con objetos creados en ese intervalo. Para crear las nuevas entradas *this* y *arg*, se realiza la clausura lambda partiendo desde el nuevo hub, filtrando por tipos. En el ejemplo, *this* queda apuntando a la región nueva, a la que

representa el llamado al constructor de Binary (el objeto podría haber sido creado en el new de Binary o antes de entrar a Callback), y al objeto A. Por otro lado, *arg* apunta a todos los objetos por ser del tipo *object*.

Luego, al ejecutar el statement 13, se asigna al campo f de todos los posibles objetos *this*, los objetos a los que apunta *arg* cuyo tipo sea compatible con f y no salga previamente por λ o σ (dado que la navegación está asegurada). La Fig. 3.22 muestra el estado del PTG luego de la asignación.

Al terminar el callback, es necesario crear otra región que representa los objetos creados en el intervalo entre el fin y el retorno del método no instrumentado (*Hub 4*). Este procedimiento es idéntico al realizado antes sin la presencia de callbacks, pero tratando los objetos retornados en cada callback, de la misma forma que con el receptor y los argumentos; se agrega un eje Σ desde cada uno hacia el hub (si no existe eje λ previo) y un eje λ en dirección inversa. Luego a todo lo alcanzable desde estos nodos, se agregan ejes λ de ida y vuelta. El resultado de este procedimiento en el ejemplo se aprecia en la Fig. 3.23.

Dado que los hubs *Hub 2* y *Hub 3* quedan con λ de ida y vuelta hacia el *Hub 4* desaparecen debido al **proceso de compresión** antes mencionado.

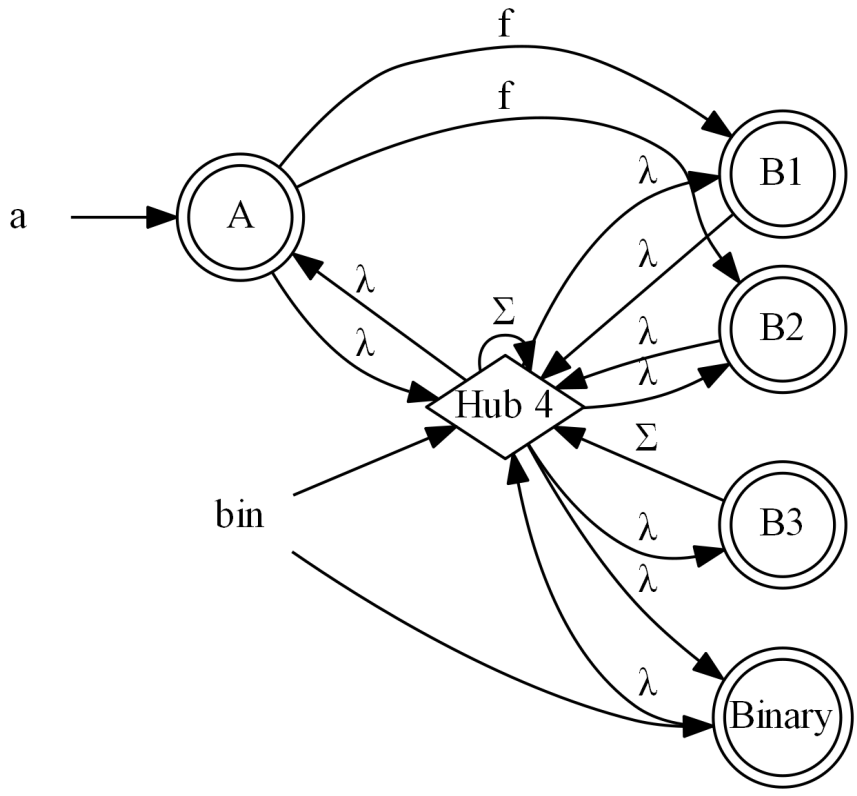

Fig. 3.23: Estado del PTG luego de la ejecución del statement 6 de la Fig. 3.21.

Finalmente, al recorrer el PTG para el término *a.f*, se obtienen los nodos que representan todos los objetos listados anteriormente, aquellos creados por la librería (representados por el *Hub 4*) y los nodos del tipo B creados en el código analizado. La Fig. 3.24 muestra el resultado de la asignación de *o* a *a.f*.

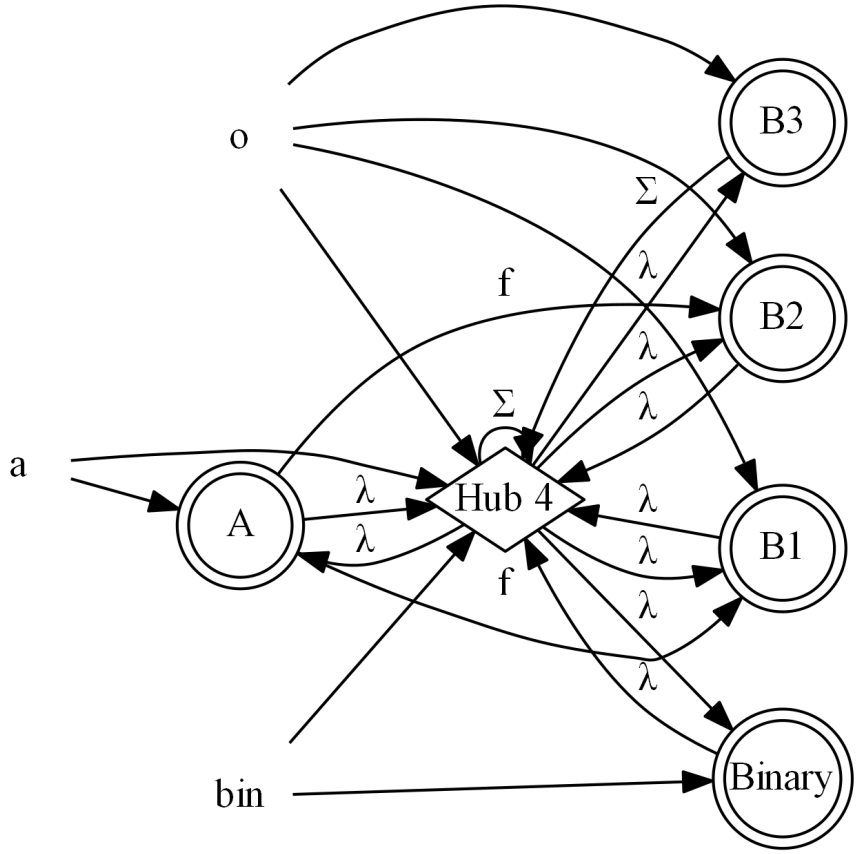

Fig. 3.24: Estado del PTG luego de la ejecución del statement 7 de la Fig. 3.21.

4. COMPARATIVA

En esta sección se compararán las cuatro herramientas presentadas en el capítulo anterior, tanto en las ventajas y desventajas de sus características, como de su eficiencia y su comportamiento ante aplicaciones reales.

4.1. Características

A continuación se realizará una comparativa sobre las características principales en el proceso de instrumentación, traza y slice de cada herramienta, analizando brevemente las ventajas y desventajas de cada una.

4.1.1. Instrumentación

La tabla 4.1 presenta una comparativa entre las principales características de las herramientas en el proceso de instrumentación.

Característica	WET	JSlice	JavaSlicer	NetSlicer
Tipo	Dinámica	No	Dinámica	Estática
Lenguaje	Assembler	-	Bytecode	C#
Compresión	No	Sí	Sí	No

Tab. 4.1: Comparativa entre las principales características del proceso de instrumentación para cada herramienta.

Tipo

La solución de JSlice es posiblemente la más interesante pues evita el tiempo de instrumentación y traza sólo los bytecodes de interés al momento de interpretarlos. Sin embargo para lograrlo es necesario modificar una máquina virtual, que conlleva a los problemas vistos anteriormente.

Las soluciones de WET y JavaSlicer tienen la ventaja de instrumentar sólo lo que se ejecuta. Este proceso disminuye los tiempos del programa original, pero ambas herramientas lo realizan sólo una vez por bloque de código ejecutado. La principal desventaja de realizar la instrumentación de forma dinámica, es que cada vez que se ejecuta la herramienta se debe volver a realizar, problema que no se encuentra presente en NetSlicer, pues lo evita realizando la instrumentación estática. De esta forma es posible ejecutar el programa con distintos criterios y distintos caminos de ejecución. Instrumentar todo el código toma más tiempo que sólo la porción necesaria, sin embargo debido a que NetSlicer no instrumenta código de terceros se realiza rápidamente.

Lenguaje

Esta característica refiere al lenguaje sobre el que se instrumenta. WET instrumenta sobre el Assembler y JavaSlicer lo hace sobre el lenguaje intermedio Bytecode. Una ventaja de utilizar un lenguaje de bajo nivel, es que permite eliminar ciertas instrucciones no deseadas. Por ejemplo, la mayoría de las técnicas de análisis basadas en traza, se concentran en el control flow o el acceso a memoria como es el caso de WET que traza las dependencias entre direcciones de memoria, abstrayendo los problemas de aliasing entre objetos. Otra ventaja es que la complejidad de instrumentar un conjunto limitado de instrucciones simples, es mucho menor que la de hacerlo sobre un lenguaje de alto nivel con gran cantidad de expresiones complejas, como lambdas, delegates y "syntax sugars". No obstante, la decisión en NetSlicer de instrumentar código C# radica en el hecho de que la cantidad instrucciones ejecutadas en un lenguaje de bajo nivel, es muy superior a la cantidad de statements ejecutados en un lenguaje de alto nivel, lo que genera archivos de trazas muy grandes y tiempos de cómputo muy elevados.

Compresión

Dado que la representación de WET es compacta y escribe la traza a disco, no necesita realizar compresión. NetSlicer tampoco necesita comprimir debido a que el tamaño de traza que genera es considerablemente menor que la generada por las otras herramientas. Si bien JavaSlicer comprime, no es realmente necesario, ya que escribe la traza a disco. JSlice es el único que **debe** comprimir, debido a que toda la traza la maneja en memoria.

4.1.2. Traza

La tabla 4.1.2 presenta una comparativa entre las principales características de las herramientas en el proceso de tracing.

Característica	WET	JSlice	JavaSlicer	NetSlicer
Información	DD DC	FC	FC	FC
Inf. Dinámica	Dir. Memoria	Id. Objeto	Id. Objeto	Ninguna
Tipo de Cómputo	Backward	Backward	Backward	Forward

Tab. 4.2: Comparativa entre las principales características del proceso de tracing para cada herramienta.

Información

La información trazada, depende del tipo de análisis que se realice sobre la misma. En las herramientas de slicing utilizadas se pueden observar dos formas. WET traza las dependencias de datos y control (DD y DC respectivamente) entre las direcciones de memoria utilizadas por las instrucciones en el programa, por lo tanto, cuando se realiza el proceso de lectura de la traza y slice, las dependencias ya se encuentran computadas. Por otro lado JSlice, JavaSlicer y NetSlicer trazan el flujo de control (FC) del programa, es decir el flujo de instrucciones o statements que se ejercitan en tiempo de ejecución.

Con esta información, las herramientas son capaces de reconstruir el flujo de ejecución y establecer las dependencias correspondientes.

Información dinámica

Si bien la definición de Korel y Laski de Slicing Dinámico se limita a utilizar un historial de ejecución o control flow dinámico para los inputs del programa determinados en el criterio de slice, en la mayoría de las herramientas se agregó en la traza información dinámica (como por ejemplo las direcciones de memoria para resolver aliasing). Como ya se mencionó, WET traza las direcciones de memoria lo que permite saber exactamente las dependencias entre las instrucciones del programa. De manera similar, tanto JSlice como JavaSlicer trazan identificadores únicos de los objetos implicados en las instrucciones que se trazan, de forma de determinar cuáles objetos se leen y cuáles se escriben. Por otro lado, NetSlicer es el único que no envía ningún tipo de información dinámica, ya que con el historial de statements ejecutados determina la dependencia entre los símbolos que contiene cada uno.

Tipo de cómputo

El cómputo de la traza backward, tiene la ventaja de computar el *backward slice* de manera simultánea, ya que ante la aparición del criterio, se empiezan a procesar las dependencias que son estrictamente necesarias. En contraposición, el cómputo forward de NetSlicer facilita la construcción del grafo de points-to.

4.1.3. Slice

La tabla 4.1.3 presenta una comparativa entre las principales características de las herramientas en el proceso de slicing.

Caracteristica	WET	JSlice	JavaSlicer	NetSlicer
C. Tercero	Sí	Sí	Sí	No
Cómputo DC	Estático (opcional)	Estático	Estático	Dinámico (opcional)
Tipo de Slice	Backward	Backward	Backward	Backward, Forward
Criterio	$(X, s_i^k, *)$	$(X, s_i^U, *)$, $(X, s_i^*, *)$	(X, s_i^U, V)	$(X, s_i^0, *)$, $(X, s_i^U, *)$
Ejecutable	No	No	No	No
PDG	Si	No	No	Si
Multithreading	Si	Si	Parcial	No
Algoritmo	Preciso	Preciso	Preciso	Impreciso

Tab. 4.3: Comparativa entre las principales características del proceso de slicing para cada herramienta.

Código de terceros

Este punto fue analizado en más de una oportunidad. Dado que WET, JSlice y JavaSlicer instrumentan y trazan lenguaje de bajo nivel, son capaces de analizar código de terceros, mientras que NetSlicer no. Sin embargo es importante comprender que desde un punto de vista de debugging, usualmente se encuentra bajo análisis el código propio del programa y no de los frameworks o librerías que utiliza. Por tal motivo, es un punto de tradeoff para NetSlicer, por un lado evitar el análisis de código de terceros elimina tiempos de cómputo innecesarios, pero a la vez pierde precisión.

Cómputo de dependencias de control

Como se explicó en el capítulo 2, en WET es opcional computar las dependencias de control ejecutando o no la herramienta Diablo. Por otro lado, si bien no se explicó previamente, NetSlicer también puede no computarlas.

El pre cómputo estático de dependencias de control facilita el proceso de slicing y el seguimiento del flujo de control. Cuando se analiza una instrucción o statement, se consulta al CDG y si éste tiene una dependencia contra una estructura de control se incorpora. De esta forma no es necesario tener un control de los scopes de los que se entra y se sale como hace NetSlicer. No existe una limitación por la que NetSlicer no pueda realizar este procedimiento previamente, no obstante el tiempo de cómputo destinado al mantenimiento de las estructuras de datos que controlan las dependencias de control es despreciable con respecto al cómputo general del slicer.

Tipo de slice

Dado que JSlice y JavaSlicer realizan un recorrido backward que computa las dependencias a demanda, no están habilitados a realizar el recorrido forward. En el caso de WET, si bien construye un grafo similar a un DDG, la definición de WET Slice se limita al recorrido backward, por lo que es el único modo disponible. En lo que respecta a NetSlicer, una vez realizado el cómputo del DDG, permite realizar el slice tanto backward como forward, invirtiendo los ejes.

Criterio

El statement s_i representa la línea i del código fuente de alto nivel, exceptuando por WET, donde representa una instrucción de assembler. En WET, JSlice y NetSlicer, el conjunto de variables seguidas no se especifica, mientras que en JavaSlicer sí (dando la opción de ser todas $*$). Todas las soluciones computan el slice a partir de la última aparición, pero JSlice agrega la posibilidad de generar un slice con la unión de todas las apariciones del statement, NetSlicer, permite realizar el slice a partir de la primera aparición y WET es el único que toma cualquier aparición.

Slice ejecutable

Como se explicó en el primer capítulo, ninguna de las herramientas utilizadas genera un slice ejecutable como indica la definición original de Korel y Laski. No obstante, como el objetivo de las mismas es la aplicación de slicing en debugging, no es un requerimiento necesario.

Generación de PDG

En la sección 3.2.3 del capítulo anterior, se explicó que el algoritmo utilizado por JSlice y JavaSlicer no requieren construir un PDG para luego recorrerlo. WET, por otro lado, realiza una representación similar a un dependence graph, y define el backward slice como el recorrido sobre esa representación. NetSlicer realiza el PDG con enfoque **AH-4** de Agrawal Y Horgan.

Soporte multi threading

Dado al avance sobre los procesadores actuales para ejecutar tareas en paralelo, la mayoría de las aplicaciones reales que se pueden encontrar en la industria son multi thread. Por este motivo, es deseable que la herramienta admita analizar programas con esta característica. El principal problema al analizar una aplicación multi thread, es el de establecer correctamente las dependencias entre los threads.

El soporte multithread de WET se apoya en la capacidad de ejecución de aplicaciones con esta característica de Valgrind. Éste ejecuta todos los threads linealmente tomando ciertas decisiones de scheduling, que pueden ser guardadas en un archivo. Ese archivo puede utilizarse para volver a correr Valgrind, de manera que tome las mismas decisiones de scheduling. Este determinismo le permite a WET establecer las dependencias correspondientes entre threads.

JSlice también soporta múltiples threads en la versión 2.0, sin embargo los autores no especifican cómo lo realizan.

Con respecto a JavaSlicer, el soporte multi thread es "parcial". Es capaz de analizar una aplicación multi thread, pero estableciendo las dependencias por cada thread separado.

NetSlicer no posee soporte alguno multi thread, pero se incorporará en versiones futuras.

Algoritmo

WET, JSlice y JavaSlicer utilizan información dinámica y analizan todo el código que ejecuta el programa. Esto los ubica dentro de la categoría de precisos, ya que con la información dinámica pueden distinguir por ejemplo en un array cuál es la posición a la cual se está accediendo, o en llamados a métodos de código de terceros conocer el efecto exacto. El análisis híbrido de NetSlicer lo convierte en impreciso, dado que el análisis de points-to otorga una sobreestimación de los efectos del aliasing o los métodos no instrumentados.

4.2. Eficiencia

En esta sección se realizarán slices sobre programas simples con las diferentes herramientas, de manera de analizar la escalabilidad de cada una y compararlas entre sí.

Prueba.1: Correr cada una de las herramientas en un ciclo simple (Fig. 4.1), observando los tiempos de cómputo, el slice resultante y el tamaño de la traza a medida que aumenta la cantidad de iteraciones.

- *Criterio*: $C = (I, s_{11}^{U}, *)$, statement 11, última aparición y cantidad de iteraciones I como input.

- *Slice esperado*: {11, 5}. La variable z, en línea 11, depende de y, que se define en la línea 5.

```
1  #include <stdio.h>
2  int main(int argc, char* argv[])
3  {
4          int x = 2;
5          int y = 3;
6          int i = 0;
7          while (i < argv[1]) {
8                  x = x + 1;
9                  i++;
10         }
11         int z = y;
12         return 0;
13 }
```

Fig. 4.1: Ciclo simple en lenguaje C++, con cantidad de iteraciones definidas por argumento.

- **WET**

  Como se comentó en el capítulo anterior, WET posee dos modos de ejecución, el *exhaustivo* y el *limitado*. Visto que el programa admite la posibilidad de realizar la traza sin ejecutar el slice, se pueden distinguir tres formas de ejecutar la herramienta.

  - **LS**: Modo limitado y cómputo de slice. Se utiliza el modo limitado (con buffer circular) y se realiza el cómputo del slice.
  - **ES**: Modo exhaustivo y cómputo de slice. Se utiliza el modo exhaustivo y se realiza el cómputo del slice.
  - **E**: Modo exhaustivo. Se utiliza sólo el formato extendido sin cómputo del slice. Gracias a la legibilidad de la traza, se corroboran visualmente las dependencias directamente sobre la traza.

  Además se agregó una cuarta forma (**L+PS**) que realiza la traza en modo limitado y realiza un pos procesamiento de la misma que será explicado más adelante.

  Notar que el modo limitado sin cómputo de slice (al que se podría nombrar **L**) no se considera debido a que las dependencias en la traza no son legibles como fue mencionado en el capítulo anterior.

  La Tab. 4.4 contiene los resultados obtenidos para los cuatros modos de ejecución mencionados aumentando la cantidad de iteraciones del ciclo de 10^2 a 10^8. Por cada modo existen dos tres filas, la primera el tiempo de cómputo del slice, la segunda el tamaño de traza y la tercera el resultado obtenido. En el caso de L+PS el tiempo se encuentra dividido entre el tiempo de traza y el tiempo del pos procesamiento. Además se incluyó el tiempo del programa original (T.Ori). La notación E.M indica "error de memoria", es decir el slicer terminó por falta de memoria y T.O indica que el programa se paró luego del tiempo mostrado.

Primero se realizó la prueba con buffer circular y cómputo de slicing **LS**. Para 10^2 iteraciones, el resultado del slice es correcto, pero algo impreciso. Las líneas 11 y 5 son las esperadas pero se incluye la línea 3 correspondiente a la llave de apertura del cuerpo del método, que puede representar la declaración de la función y por lo tanto, que 5 y 11 se encuentren ligadas por control. Es importante ver que al aumentar la cantidad de iteraciones a 10^3, la línea 3 desaparece, que podría indicar overflow del buffer circular. No obstante, si bien esta línea desaparece, el slice sigue siendo correcto. Como se aprecia en la Tab. 4.4 a partir de 10^5 iteraciones no fue posible obtener resultados, ya que al momento del cómputo del slice, en unos pocos segundos WET consume 1GB de memoria (en la tabla se aprecia como *E.M*), lo que conlleva a que termine abruptamente, pues aparenta tener una estructura limitada en ese tamaño que se intentó, sin éxito, hallar para incrementar su capacidad.

Conf.	10^2	10^3	10^4	10^5	10^6	10^7	10^8
T.Ori	1.94x10^{-6}s	3.48x10^{-6}s	2.27x10^{-5}s	0.000246s	0.00236s	0.0233s	0.244s
LS	4.307s	1.921s	1.476s	7.155s	21.755s	195.455s	2055.34s
	1.3MB	2.8MB	3.0MB	3.2MB	3.4MB	3.6MB	3.8MB
	{11,3,5}	{11,5}	{11,5}	E.M	E.M	E.M	E.M
ES	5.386s	972.779s	T.O (10hs)	-	-	-	-
	0.9MB	7.5MB	91MB	-	-	-	-
	{11,3,5}	{11,3,5}	-	-	-	-	-
E	0.649s	3.476s	37.960s	87.521s	84.964s	82.931s	81.584s
	0.9MB	7.5MB	91MB	-	-	-	-
	ok	ok	ok	E.M	E.M	E.M	E.M
L+PS	0.497s + 3.28s	0.682s + 3.72s	0.823s + 3.78s	2.320s + 3.91s	18.526s + 3.99s	194.335s + 3.89s	2038.08s + 3.81s
	1.3MB	2.8MB	3.0MB	3.2MB	3.4MB	3.6MB	3.8MB
	{3,5,6,7,9, 11,12,13}	{11,5,7,9}	{11,5,7,9}	{11,5,7,9}	{11,5,7,9}	{11,5,7,9}	{11,5,7,9}

Tab. 4.4: Comparación entre los cuatro modos de ejecución de WET sobre el código de la Fig. 4.1.

Utilizando el modo exhaustivo **ES**, el slice obtenido para 10^3 difiere del obtenido con LS, ya que en éste aparece nuevamente la línea 3. Esto parece confirmar la sospecha de que en **LS** hubo pérdidas de dependencias. El problema con este modo es que los tiempos de cómputo son extremadamente altos. Con 10^3 iteraciones, tardó más de 16 minutos en realizar el slice, contra los casi 2 segundos que tomó LS. Esta gran diferencia en tiempo de ejecución evidencia la necesidad de utilizar el buffer circular. Con 10^4 el programa fue detenido luego de ejecutar durante 10 horas sin retornar resultados.

Las pruebas con WET en modo exhaustivo sin computar slice (**E**), parecen demostrar que los tiempos de cómputo elevados se encuentran en el cómputo del slice. La ejecución para 10^3 iteraciones tarda aproximadamente 4 segundos y con 10^4 solo 40 segundos, contra los 16 minutos y timeout respectivamente de ES.

Dado que la traza del modo exhaustivo (E) es extensa y dificulta su posterior análisis, que luego de 10^5 arroja problemas de memoria similares a los de LS pero en tiempo de traza y que el cómputo del slice sobre este modo (ES) arroja tiempos tan elevados,

se descartaron estos modos de ejecución definitivamente, concentrando los esfuerzos en el modo limitado.

Realizando las pruebas se notó que en modo limitado, el proceso de generación de traza se realiza rápida y exitosamente, mientras que el tiempo y memoria consumidos en la etapa de slice son considerablemente altos. Se decidió implementar un **pos procesamiento de la traza** que realiza el slice, basado en el hecho de que según la definición de *WET Slice*, es simplemente el recorrido backward sobre la representación WET, pues ésta contiene un superconjunto de la información necesaria para realizarlo. El pos procesamiento consiste entonces, en generar el DDG sobre las dependencias y luego realizar un BFS desde la dirección de memoria de interés. En la Tab. 4.4 se encuentra la fila **L+PS** (modo limitado con pos procesamiento) y los resultados obtenidos.

En primer lugar es importante destacar que debido a que la traza no supera los 3.8MB, el DDG en memoria no ocupa mucho más que eso, por lo que con L+PS el consumo de memoria no es un problema. Gracias a esto, se obtuvieron resultados con tiempos considerablemente buenos hasta con 10^8 iteraciones. Dado que el tamaño de la traza no crece gracias al buffer circular, es coherente que el tiempo consumido por el pos procesamiento sea constante, sin embargo el tiempo consumido al generar la traza en todos los casos se mantiene algo inferior a 10.000 veces más lento que el tiempo del programa original.

Con respecto a los resultados obtenidos, para 10^2 se observa que sólo quedan afuera las relacionadas con la variable x, que es correcto ya que no afecta para nada la variable y, sin embargo se obtuvieron resultados correctos pero muy imprecisos ya que contiene líneas cómo la 7 y 9 que no deberían aparecer en el slice. Esto se puede deber a que la representación WET contiene más información que la requerida por el método de slicing tradicional.

Por otro lado, al igual que con LS, en 10^3, la cantidad de líneas disminuye, pues la traza es la misma que la obtenida anteriormente.

Es evidente la diferencia de precisión entre el cómputo del slice realizado por WET y el realizado por el pos procesamiento. Es claro que el proceso de slice de WET no sólo recorre la traza de forma backward sobre las dependencias como indican sus autores sino que realiza algún otro tipo de cómputo, lo que explicaría también los tiempos elevados de ejecución.

Si bien esta prueba parece indicar overflow del buffer circular, no es posible concluir que WET tenga problemas de correctitud. Sin embargo, existe la posibilidad que WET realice cómputo de dependencias de forma estática, ya que en uno de los trabajos relacionados al proyecto, Xiangyu Zhang y Rajiv Gupta [41] presentaron una optimización que consiste en que, dada una variable v con uso de otra w cuya última definición se encuentra en el mismo scope, se establece una dependencia de datos estática de v a w. No obstante, más adelante se mostrará un caso en que se evidencia la pérdida de dependencias.

- **JSlice**

Con JSlice solo es posible mostrar los tiempos de ejecución del proceso de slicing ya que éste se genera "on-the-fly", no instrumenta ni genera archivo con la traza. La Tab. 4.5, muestra los resultados obtenidos para un código análogo al de la Fig. 4.1. Es notable la diferencia de performance con respecto a WET. La memoria consumida es despreciable y la cantidad de iteraciones alcanza las 10^8 en tiempo aceptable (al rededor de **110** veces el del programa original) procesando todos los bytecodes ejecutados.

El conjunto de statements obtenidos fue exactamente el esperado {11,5} en todos los casos pues no existen condiciones que modifiquen el resultado (como el buffer circular de WET). En este caso no se agregaron líneas por debugging.

Iteraciones	10^2	10^3	10^4	10^5	10^6	10^7	10^8
Tiempo	0.31s	0.33s	0.34s	0.47s	1.92s	22.75s	190.56s
T.Original	5.0x10^{-6}s	2.6x10^{-5}s	0.000185s	0.00156s	0.0169s	0.175s	1.742s

Tab. 4.5: Resultados de ejecución de JSlice para un código Java análogo al de la Fig. 4.1.

- **JavaSlicer**

  La ejecución de JavaSlicer permite discriminar entre el tiempo de cómputo consumido al realizar la traza y el del proceso de slicing, así como también el tamaño de la traza obtenida.

  Observando la Tab. 4.6 donde se encuentran los resultados obtenidos para el código Java análogo al de la Fig. 4.1 con el que se ejecutó JSlice, se aprecia que entre 10^2 y 10^4 iteraciones, el tamaño de la traza prácticamente no crece, así como tampoco los tiempos de generación de traza, sin embargo son mucho más elevados que los de su predecesor. Esto puede deberse a la instrumentación de las clases de la JRE. A partir de 10^5 se empieza a apreciar un incremento en el tamaño de la traza, lo que conlleva al aumento en el tiempo de slicing, pero que no lleva una relación directa entre el tamaño y el tiempo.

  Para 10^8 con una traza de solo 772MB el proceso termina con una excepción por falta de memoria (en la tabla como *E.M*) "*GC overhead limit exceeded*", que es lanzada por la JVM si gasta más del 98 % del tiempo total recolectando la memoria y cuando después de recolectar sólo se recupera menos del 2 % del heap. Esto significa que la pequeña cantidad de memoria del heap que el garbage collector puede limpiar probablemente se llena de nuevo rápidamente, forzando al GC a reiniciar el proceso de limpieza nuevamente. Esto forma un círculo vicioso donde la CPU está al 100 % ocupada con el proceso de garbage collection y deja de computar el proceso real.

  Se aumentó la cantidad de memoria utilizada por la VM con el parámetro **-Xmx** a 5GB pero aún así, el proceso termina con el mismo error luego de 2200 segundos aproximadamente.

  Al igual que con JSlice, el conjunto obtenido en todos los casos fue el correcto, {11,5}.

- **NetSlicer**

Iteraciones	10^2	10^3	10^4	10^5	10^6	10^7	10^8
Traza	2.4MB	2.5MB	2.5MB	3.2MB	11MB	80MB	772MB
Tiempo	22.28s + 2.86s	22.19s + 2.30s	23.27s + 3.67s	23.07s + 6.70s	25.77s + 18.93s	27.11s + 43.01s	29.92s + 2229.16s (E.M)
T.Original	1.0x10^{-6}s	2.0x10^{-5}s	9.9x10^{-5}s	0.000715s	0.00721s	0.0763s	0.771s

Tab. 4.6: Resultados de ejecución de JavaSlicer para un código Java análogo al de la Fig. 4.1.

Como fue mencionado en el capítulo anterior, NetSlicer tiene dos posibles modos de ejecución, puede escribir la traza en un archivo y luego procesarlo, o puede utilizar comunicación entre procesos (IPC) mediante Pipe para evitar el retraso generado por la escritura a disco, no obstante las pruebas sólo fueron realizadas escribiendo a disco con el objetivo de obtener los tiempos del proceso de trazado y slicing por separado. En la Tab. 4.7 se pueden ver los resultados de la ejecución de un código escrito en C# análogo al de la Fig. 4.1.

Es llamativo el excesivo crecimiento de la traza donde se aprecia un evidente crecimiento lineal con respecto a las iteraciones. Para 10^7 resulta mucho mayor que el resto de las herramientas, cuando la hipótesis que sostiene es precisamente que instrumentar código de alto nivel genera trazas de menor tamaño que hacerlo sobre bajo nivel. Si bien posteriormente se verá que esto es cierto en la mayoría de los casos, este es un ejemplo particular donde la cantidad de bytecodes del ciclo, no es muy superior al código original (por ser statements muy simples), y el tamaño en bytes de los datos trazados por NetSlicer es superior al de JSlice y JavaSlicer. Para 10^8 iteraciones el tamaño de la traza no cabe en memoria donde se realizaron las pruebas (intel core i5 con 8gb de memoria ram), pero se puede estimar el tiempo de slicing en 9000 segundos, dado que al igual que el tamaño de traza, el tiempo va incrementando proporcionalmente.

Iteraciones	10^2	10^3	10^4	10^5	10^6	10^7	10^8
Traza	7KB	64KB	635KB	6.19MB	61.9MB	619MB	6.19GB
Tiempo	0.009s + 0.219s	0.037s + 0.109s	0.223s + 0.937s	1.997s + 9.094s	20.952s + 91.311s	207.014s + 925.918s	2174.732s + 9000s (E)
T.Original	4.89x10^{-7}s	3.25x19^{-6}s	0.0000371s	0.000269s	0.00249s	0.0252s	0.273s

Tab. 4.7: Resultados de ejecución de NetSlicer para un código C# análogo al de la Fig. 4.1.

Prueba.2: Correr cada una de las herramientas sobre el código de la Fig. 4.2, que consiste en la multiplicación de dos matrices cuadradas con valores aleatorios, observando los tiempos de cómputo, el slice resultante y el tamaño de la traza a medida que aumenta el tamaño de las matrices.

- *Criterio*: $C = (X, s^U_{63}, *)$, última aparición de la línea 63. El input es el tamaño de la matriz cuadrada, mayor o igual que 110.

- *Slice esperado*: $\{9, 21, 45, 46, 47, 48, 59, 63\}$. La variable x, en línea 63, depende del valor de la matriz *ans* en la posición 50x50. La posición pedida se calcula sumando los índices provistos a los punteros donde comienza la matriz en memoria, por lo tanto la declaración de la matriz inicial *ans* (línea 59) debe aparecer, así como los arrays que contiene (líneas 9 y 21 que no se encuentran en la porción de código mostrada). El contenido en la posición 50x50 se modifica en la línea 48, que depende por control de las líneas 47, 46 y 45.

```
42 void multiply_matrices(int n, int** m1, int** m2, int** ans)
43 {
44     int i, j, k;
45     for(i = 0; i < n; i++){
46         for(j = 0; j < n; j++){
47             for(k = 0; k < n; k++){
48                 ans[i][j] += m1[i][k] * m2[k][j];
49             }
50         }
51     }
52 }
53
54 int main(int argc, char* argv[])
55 {
56     int n = atoi(argv[1]);
57     int** m1 = rand_matrix(n);
58     int** m2 = rand_matrix(n);
59     int** ans = new_matrix(n);
60
61     multiply_matrices(n, m1, m2, ans);
62
63     int x = ans[1][1];
64
65     free_matrix(n, ans);
66     free_matrix(n, m1);
67     free_matrix(n, m2);
68     return 0;
69 }
```

Fig. 4.2: Porción de código de algoritmo cúbico de multiplicación de matrices en C, con tamaño de matrices definidas por argumento.

- **WET**

  Las pruebas presentadas corresponden a la ejecución de WET en el modo L+PS. Esto se debe a que como se mostró anteriormente realizando el cómputo del slice (LS), WET termina por falta de memoria con 10^5 iteraciones, considerablemente menos que las realizadas por el algoritmo $O(n^3)$ utilizado con matrices de 110x110 a 200x200.

  En la Tab. 4.8 se encuentran los resultados obtenidos. Para matrices entre 110x110 y 150x150, los resultados obtenidos son los esperados, con algunas líneas de más. Estas diferencias probablemente se deban a la incorporación de dependencias de

110	**120**	**130**	**140**	**150**
5.385s + 3.694s	6.618s + 3.894s	8.625s + 3.890s	10.107s + 4.134s	12.384s + 4.543s
2.9MB	2.8MB	2.8MB	2.7MB	2.7MB
{21.43,45, 46,47,48,52, 59,61,63,9}	{21.43,45, 46,47,48,52, 59,61,63,9}	{21.43,45, 46,47,48,52, 59,61,63,9}	{21.43,45, 46,47,48,52, 59,61,63,9}	{21.43,45, 46,47,48,52, 59,61,63,9}
160	**170**	**180**	**190**	**200**
14.989s + 4.317s	18.331s + 3.975s	20.878s + 3.809s	24.626s + 4.103s	29.422s + 4.202s
2.7M	2.7M	2.7M	2.7M	2.7M
{63}	{63}	{63}	{63}	{63}

Tab. 4.8: Resultados de ejecución de WET para el código de la Fig. 4.2, con L+PS con buffer en 100.000.

control con respecto a los llamados a función, pues se agregan líneas 43 y 52 que corresponden al inicio y fin de la función multiply_matrices así como también el llamado a la misma (línea 59).

110	**120**	**130**	**140**	**150**
5.867s + 6.599s	7.375s + 7.691s	9.388s + 6.761s	10.905s + 7.008s	13.3396s + 6.965s
6.2MB	6.1MB	6.1MB	6.0MB	6.0MB
{21.43,45, 46,47,48,52, 59,61,63,9}	{21.43,45, 46,47,48,52, 59,61,63,9}	{21.43,45, 46,47,48,52, 59,61,63,9}	{21.43,45, 46,47,48,52, 59,61,63,9}	{21.43,45, 46,47,48,52, 59,61,63,9}
160	**170**	**180**	**190**	**200**
15.438s + 7.201s	18.193s + 6.972s	21.688s + 7.684s	25.293s + 7.523s	29.178s + 7.326s
5.9MB	5.9MB	5.8MB	5.8MB	5.7MB
{21.43,45, 46,47,48,52, 59,61,63,9}	{21.43,45, 46,47,48,52, 59,61,63,9}	{21.43,45, 46,47,48,52, 59,61,63,9}	{21.43,45, 46,47,48,52, 59,61,63,9}	{21.43,45, 46,47,48,52, 59,61,63,9}

Tab. 4.9: Resultados de ejecución de WET para el código de la Fig. 4.2, con L+PS con buffer en 200.000.

Lo interesante de esta prueba es que finalmente se obtuvieron problemas de correctitud debido a la pérdida de dependencias, a partir de la multiplicación de matrices de 160x160, donde retorna sólo la línea 63 correspondiente al criterio. Se decidió aumentar el tamaño con la intensión de obtener las dependencias perdidas. Dado que no existe argumento de la aplicación que permita modificar el tamaño del buffer, se buscó en el código de la aplicación el momento en que define el número máximo de dependencias y se modificó de 100.000 (valor default como indica la documentación [1]) a 200.000. La Tab. 4.9 muestra los resultados de la ejecución de WET con esta modificación. Como se puede apreciar, los resultados a partir de 160x160 son los

[1] http://wet.cs.ucr.edu/trace_format.html

esperados, lo que indica que no se perdieron dependencias. Es importante notar que como era de esperarse, duplicar el buffer generó que el tamaño de traza se duplicara así como también se aprecia aumento en los tiempos de cómputo del proceso de slicing.

La Tab. 4.10, compara los tiempos del proceso de tracing para el código de la Fig. 4.2, contra los tiempos de la ejecución del programa original. Como muestra la tabla, existe una relación lineal de 530 veces entre ambos con respecto al tamaño de la entrada, lo que parece indicar que WET computa el slice linealmente en función de la cantidad de instrucciones que ejecuta. Esto se puede apreciar gráficamente en la Fig. 4.3, donde se compara el tiempo de WET, el comportamiento cúbico del algoritmo y los los tiempos del programa original.

Tamaño	**T. WET L+PS (s)**	**T. Original (s)**
110	5.867	0.0109235
120	7.375	0.013535
130	9.388	0.0167456
140	10.905	0.0203703
150	13.339	0.0248197
160	15.438	0.0301067
170	18.193	0.035513
180	21.688	0.042513
190	25.293	0.0495438
200	29.178	0.0560234

Tab. 4.10: Tiempos de ejecución de WET para el código de la Fig. 4.2, con L+PS con buffer en 200.000.

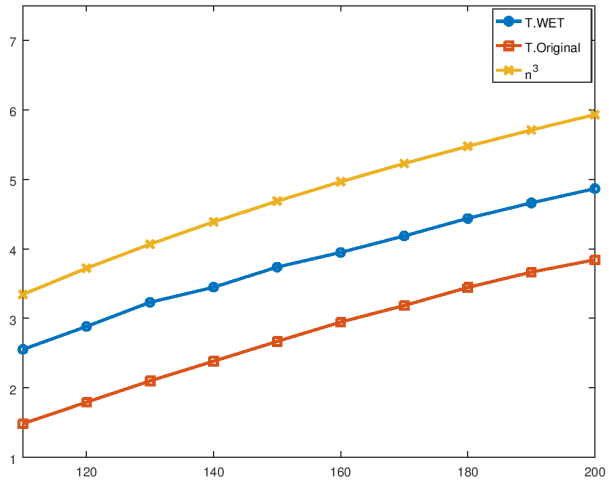

Fig. 4.3: Tiempos de ejecución de WET para el código de la Fig. 4.2, con L+PS con buffer en 200.000. Tiempo en función de la entrada en escala logarítmica.

- **JSlice**

En la Tab. 4.11 se encuentran los resultados obtenidos del análisis utilizando JSlice de un programa de multiplicación de matrices cuadradas similar al de la Fig. 4.2. Nuevamente JSlice muestra tiempos de ejecución realmente buenos. En este caso, obtenemos una diferencia de tiempos con respecto al código original del orden de 21 veces más lento, lo que muestra al igual que WET una relación lineal con la cantidad de sentencias que se ejecutan, lo que se ve con claridad en la Fig. 4.4.

Tamaño	T.JSlice (s)	T.Original (s)
100	7.38	0.363
110	10.45	0.498
120	12.25	0.615
130	16.01	0.786
140	18.64	0.9
150	21.9	1.109
160	26.68	1.299
170	31.54	1.546
180	37.98	1.817
190	44.34	2.102
200	52.32	2.478

Tab. 4.11: Tiempos de ejecución del algoritmo de multiplicación de matrices para JSlice.

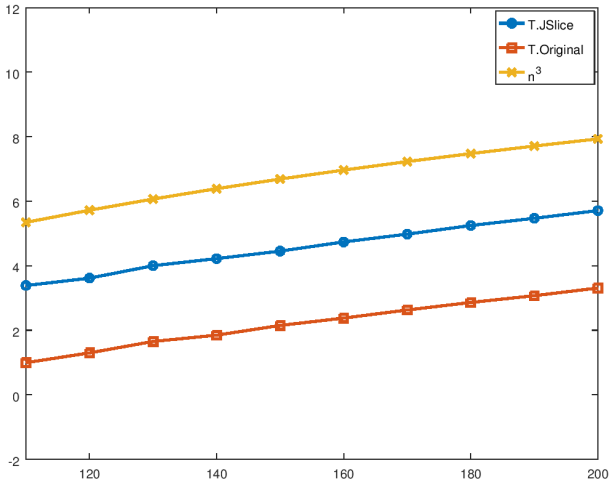

Fig. 4.4: Tiempos de ejecución de JSlice para el algoritmo de multiplicación de matrices.

- **JavaSlicer**

  La Tab. 4.12 contiene los resultados para el algoritmo de multiplicación de matrices analizado con JavaSlicer. Se logró correr satisfactoriamente con matrices de hasta 170x170, para matrices de 180x180 se obtuvo la excepción de memoria del garbage collector que se comentó en la Prueba.1.

Tamaño	T.JavaSlicer (s)	T.Original (s)
110	33.912 + 93.180	0.0665
120	33.024 + 118.312	0.0813
130	34.624 + 151.942	0.102
140	35.856 + 190.528	0.125
150	37.584 + 239.956	0.155
160	41.180 + 289.516	0.182
170	44.376 + 950.836	0.217

Tab. 4.12: Tiempos de ejecución del algoritmo de multiplicación de matrices para JavaSlicer.

En la Fig. 4.5 se compara el comportamiento de la herramienta en función del tamaño de la matriz, con el tiempo del algoritmo original y n^3, con los valores extraídos de la Tab. 4.12. Si bien con matrices de 170x170 JavaSlicer logra finalizar satisfactoriamente y entregar el resultado esperado, se aprecia una clara degradación de la performance.

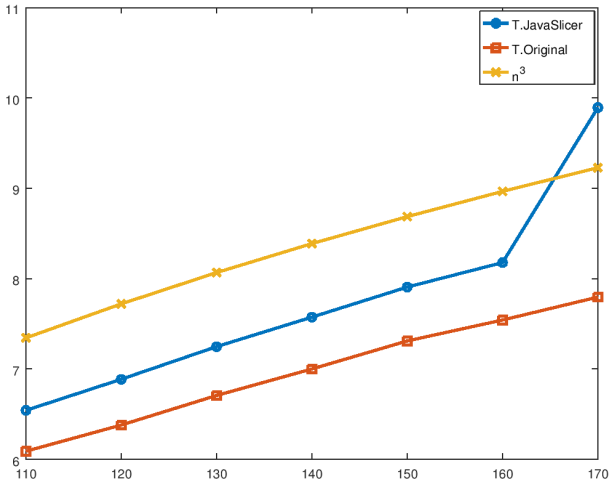

Fig. 4.5: Tiempos de ejecución de JavaSlicer para el algoritmo de multiplicación de matrices.

Comparado con su antecesor, los tiempos de ejecución son más elevados y si bien parece comportarse linealmente con respecto a la cantidad de bytecodes ejecutados, el tiempo total empleado es 1900 veces más lento que el tiempo consumido por el algoritmo original, siendo el proceso de tracing aproximadamente 500 veces el original, sin embargo si bien comienza siendo un 30 % de la ejecución con matrices de 110x110, no aumenta proporcionalmente a medida que se incrementa la entrada, lo que parece indicar que la mayor parte del tiempo se lo lleva la instrumentación de la JRE y clases necesarias, y no la escritura de la traza en disco.

- **NetSlicer**

  En la Tab. 4.13 se muestran los resultados obtenidos para NetSlicer. Los tiempos obtenidos y la proporción con respecto a al código original son considerablemente más elevados que los del resto de las herramientas.

Tamaño	T.NetSlicer(s)	T.Original(s)
100	24.36 + 667.92	0.00712
110	26.88 + 985.54	0.0123
120	36.09 + 1383.23	0.0148
130	43.79 + 1926.49	0.0175
140	58.34 + 2589.31	0.0229
150	72.91 + 4871.32	0.0269

Tab. 4.13: Tiempos de ejecución del algoritmo de multiplicación de matrices para NetSlicer.

En las pruebas con matrices de 160x160 el programa ejecutó durante 9 horas en situación de *thrashing* luego de haber consumido 7GB de memoria ram, sin retornar resultado alguno. Esto sucedió en menor medida con 150x150, donde el slicer no llegó a consumir la totalidad de la memoria, pero se produjo un deterioro en el comportamiento que se visualiza en el incremento de tiempos en la Fig. 4.6.

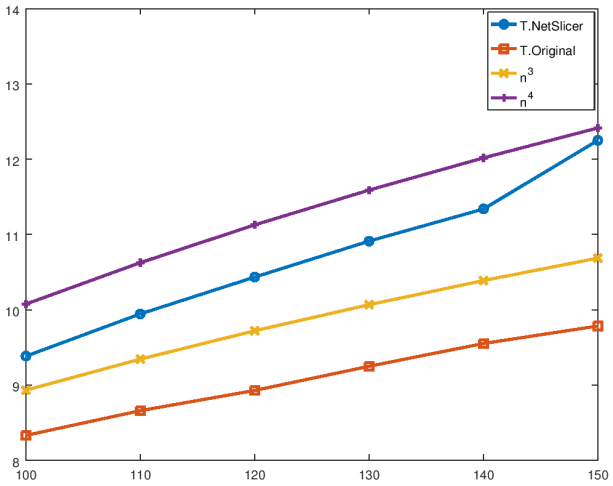

Fig. 4.6: Tiempos de ejecución de NetSlicer para el algoritmo de multiplicación de matrices, en escala logarítimica

El problema de performance de NetSlicer se encuentra en el ciclo con creación de objetos, del algoritmo de multiplicación de matrices probado, que constituye uno de los peores casos de análisis.

```
37  int[][] m1 = new int[n][];
38  for (i = 0; i < n; i++)
39      m1[i] = new int[n];
40
41  for (o = 0; o < n; o++)
42      for (i = 0; i < n; i++)
43          m1[o][i] = value;
```

Fig. 4.7: Porción de código del algoritmo cúbico de multiplicación de matrices en lenguaje C#, análogo al mostrado en la Fig. 4.2.

El algoritmo de multiplicación de matrices cuadradas presentado en el ejemplo tiene complejidad $O(n^3)$ ya que ejecuta un statement que ejecuta n^2 veces un ciclo de n instrucciones. Por lo tanto, NetSlicer debe analizar este statement n^3 veces también, pero no lo hace en $O(1)$.

Para entender el comportamiento de NetSlicer, se muestra el código adaptado en C# en la Fig. 4.7 del código de la Fig. 4.2 donde se crea y carga una de las matrices (método *rand_matrix*) y la Fig. 4.8 muestra el PTG resultante para la ejecución de ese código con $n = 2$.

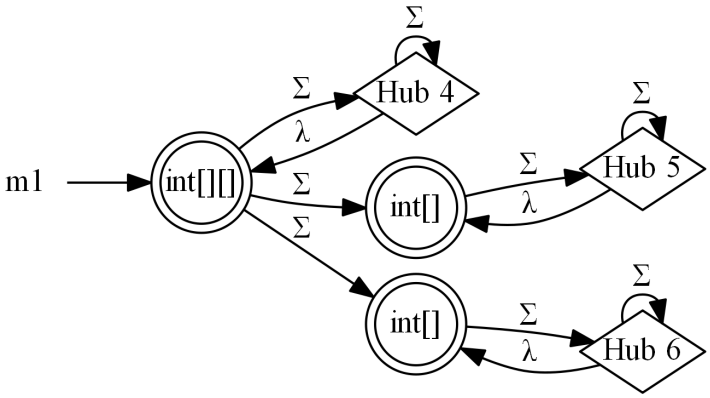

Fig. 4.8: Subgrafo del PTG generado para el algoritmo de multiplicación de matrices.

En el PTG se observa el punto de entrada *m1* que apunta al objeto que representa la matriz (o arreglo de arreglos), éste por Σ llega a cada uno de los nodos que representan los arreglos de enteros que se crean en el primer ciclo, es decir que si las matrices son de n cuadradas, tendrá n cantidad de vértices alcanzables.

A causa del análisis simbólico que realiza NetSlicer, no es posible discriminar el índice sobre el PTG pues se desconoce el valor de i al momento de ser asignado, por lo que un acceso indexado ($[i]$) debe retornar todos los objetos del arreglo. Por este motivo, cada cada vez que se accede a una posición (i, j) de la matriz es necesario recorrer los n nodos alcanzables por Σ. La complejidad del análisis resultante entonces, es $O(n^4)$, pues el algoritmo original es n^3 y debe recorrer n nodos en el PTG.

En la prueba **P1**, el código contenido en el cuerpo del ciclo, se traduce en menos cantidad de statements de bajo nivel. Por tal motivo la cantidad de traza emitida por las herramientas que instrumentan bajo nivel, es menor que la escrita por NetSlicer, por tal motivo NetSlicer produce gran cantidad de traza y arroja tiempos de cómputo muy elevados. Sin embargo, este caso podría considerarse "extremo", ya que como se verá posteriormente, la traza generada por NetSlicer es siempre menor a la del resto de las herramientas.

Los resultados de WET fueron desalentadores, los tiempos de cómputo de los modos de ejecución que computan el slice son muy elevados y prácticamente no se pueden utilizar. JSlice es el más eficiente, posiblemente porque sólo monitorea los bytecodes de asignación y no los de cómputo, por lo que la traza es compacta y la puede procesar velozmente.

JavaSlicer, por otro lado si bien arroja resultados muy buenos, el consumo excesivo de memoria no le permite escalar como su predecesor.

Con **P2** se pudo observar la pérdida de dependencias de WET y la necesidad de aumentar el tamaño del buffer. Nuevamente JSlice es el que mejor se comporta, seguido por JavaSlicer, que al igual que en P1, el consumo de memoria limita su escalabilidad. Por otro lado, la naturaleza imprecisa de NetSlicer lo obliga a tomar decisiones conservativas ante cada llamado a método no instrumentado que resultan en un PTG cuya morfología degrada su eficiencia. No obstante, es importante notar que NetSlicer tiene como objetivo otro tipo de aplicaciones, con uso intensivo de código de terceros. En P2, las creaciones de los arrays (`new int[n]`), implican entre tres o cuatro statements de bajo nivel, por lo que no produce gran cantidad de traza en NetSlicer, JSlice y JavaSlicer, sin embargo para conservar la correctitud, NetSlicer debe tratarlo como cualquier código de terceros.

4.3. Frameworks y librerías

La mayoría de las aplicaciones modernas se montan sobre librerías y frameworks para facilitar y agilizar el desarrollo, proveyendo funcionalidades que son de uso común. En esta sección se mostrarán limitaciones encontradas, los tiempos de cómputo y memoria consumida de cada una de las herramienta de slicing, con este tipo de aplicaciones.

4.3.1. WET

En primer lugar se probó la librería de acceso a base de datos PostgreSQL *libpq*. Pese a los problemas con la compilación con lo enlace estático que se comentó en el capítulo anterior, se logró compilar el programa, sin embargo Diablo retornó error en el parsing del mapa generado en la compilación, con un error relacionado con la librería k5crypto. Se intentó entender y resolver el problema pero sin resultado. A causa de esto, se decidió intentar realizar el slice obviando correr Diablo, pero aún evitándolo, Valgrind arrojó problemas aparentemente por un error de "sanidad" en las instrucciones del binario. Nuevamente, se intentó arreglar el error sin éxito.

Pese a los intentos fallidos con linpq, se intentó utilizar otra librería, en este caso una aplicación con entorno gráfico utilizando **gtk-2**[2]. Lamentablemente se encontraron problemas similares a los antes descritos.

Por otro lado, los requerimientos de software mencionados en el capítulo anterior, complican la tarea de utilizar librerías actuales, utilizar frameworks o instalar software, debido a la ausencia de repositorios activos.

Dados los problemas obtenidos, **se abandonó la idea de ejecutar WET con aplicaciones reales**.

4.3.2. JSlice

Como se explicó previamente, JSlice es una modificación de la máquina virtual de licencia GNU Kaffee. Esta decisión de permite evitar la instrumentación, el proceso de tracing y la realización del cómputo del slice sobre la marcha, lo que otorga (como se pudo ver en la sección anterior) resultados de eficiencia realmente buenos. No obstante, el uso de la VM Kaffee lo transformó en una herramienta obsoleta a la hora de probar frameworks o librerías contemporáneas.

[2] https://www.gtk.org/

Al igual que sucede con WET, los requerimientos de S.O. y software, limita cualquier intento de instalar o descargar librerías o frameworks, por lo que las pruebas deben limitarse a lo que se puede realizar dentro de este entorno.

Pese a las limitaciones mencionadas, se intentó probar una aplicación con la librería gráfica **Swing** incluída en Fedora, realizando una aplicación sencilla. Lamentablemente no fue posible correr la aplicación con la VM modificada por un problema de compatibilidad entre Swing y la librería AWT (interfaz que utiliza código nativo para dibujar las ventanas) de JSlice. Cuando se ejecutaba el programa con la VM Kaffee, la **libawt** que utilizaba no contenía el método *setDropTarget* que requería Swing. Este tipo de errores se dieron en pruebas con otras librerías, intentando resolver los problemas encontrados en cada caso, pero lamentablemente sin éxito.

Dadas las limitaciones técnicas encontradas, no fue posible probar el comportamiento de JSlice con el uso de frameworks y librerías, por lo que al igual que con WET, **se abandonó la idea de ejecutarlo con aplicaciones reales**.

4.3.3. JavaSlicer

Dado que JavaSlicer fue mantenido hasta mediados del año 2016 y que a diferencia de JSlice, puede ejecutar en cualquier implementación de JVM mientras que implemente Agents, no se obtuvieron grandes dificultades para ejecutarlo sobre aplicaciones reales. Sólo se tuvieron algunos inconvenientes que fueron solucionados, estos fueron:

- Corrupción en la traza por excepción por variable en null, en la ejecución del método *Transform* del Agent. Se solucionó agregando una comprobación y retorno en caso de ser null. Esto evita el error en el archivo y permite realizar el proceso de slicing con éxito.
- Corrupción en la traza por excepción por método muy grande. La máxima longitud en bytecodes que puede tener un método en java es 64KB. Debido a la instrumentación, ciertos métodos superaban ese tamaño. El fuente de JavaSlicer contempla esto y revierte el método en caso de ser demasiado extenso, sin embargo tenía un error. Se corrigió aplicando el procedimiento existente. Es importante notar que esto evidencia un posible **problema de correctitud**, ya que se desconoce el efecto sobre los objetos de los métodos que no se instrumentan.
- Error en proceso de tracing por error de verificación (*VerifyError*). Esto puede ser causado cuando se compila contra una librería que es diferente a la que se utiliza en tiempo de ejecución hallada en el classpath. No fue posible encontrar la fuente del problema, pero se saltea utilizando el parámetro de ejecución *-noverify*. Esta solución no mostró tener efectos negativos sobre el programa.

Además, existen problemas no resueltos, detallados por el autor, que limitan el uso de la herramienta en aplicaciones complejas y pueden generar problemas de correctitud:

- **Código Nativo**. Ciertos métodos de Java se implementan en C/C++ y son llamados por la JNI (Java Native Interface). Lógicamente estos métodos no pueden ser accedidos por los transformadores por lo que no pueden ser instrumentados. Si bien no es común encontrar proyectos que contengan código nativo, muchos métodos de la JRE utilizan código nativo.

- **Reflection**. El framework de reflection de Java, se encuentra programado con código nativo, por motivo del punto anterior, no es posible analizarlo. Existen varios Frameworks frecuentemente utilizados en la industria que utilizan esta característica, por lo que se convierte en una limitación crítica.

4.3.4. NetSlicer

Tanto NetSlicer como Roslyn, compilador .NET en el cual se apoya la herramienta, se encuentran en actual desarrollo, por lo que existen algunos bugs y problemas todavía no resueltos. Por ejemplo NetSlicer no es capaz de analizar código que contenga el iterador "*yield*", o la abstracción del AST que provee Roslyn falla al utilizar referencias dinámicas ("*dynamic*"). No obstante, dado que el slicer se concibió para ser utilizado en aplicaciones que hacen uso de frameworks y librerías, corre satisfactoriamente sobre cualquier tipo de programa.

4.3.5. JavaSlicer vs NetSlicer

Dada la imposibilidad de ejecutar WET y JSlice en frameworks o librerías, las pruebas se redujeron a comparar JavaSlicer y NetSlicer que permiten utilizarlos en aplicaciones contemporáneas. Estas dos herramientas conforman dos implementaciones de la técnica completamente diferentes. JavaSlicer es el representante del algoritmo utilzado en JSlice (basado en el algoritmo preciso NP de Zhang et. al), que recorre **todos** los bytecodes (incluso los de códigos de terceros) ejecutados de forma backward, reconstruyendo las dependencias sin utilizar grafo de dependencias y siguiendo las variables a partir de la información dinámica obtenida (identificadores de los objetos que representan), por otro lado, NetSlicer realiza un análisis simbólico forward sólo con información de flujo de control, modelando la memoria con un PTG, construyendo el enfoque AH-4 de Agrawal et. al. y evitando el análisis de código de terceros.

En esta sección se analizará el comportamiento de los dos slicers ante el uso intensivo del framework ORM Hibernate para java y su equivalente en .Net NHibernate. Para esto se construyó una aplicación que realiza consultas sobre la base de datos de ejemplo que se encuentra en la documentación MySql, *Employees*[3].

Q.1: Realizar consulta sobre los empleados de la base de datos (Fig. 4.9) utilizando *Criteria Queries*, observando los tiempos de cómputo y tamaño de la traza a medida que aumenta el límite de la consulta.

- *Criterio*: $C = (L, s^U_{24}, \{s\})$, variable s del statement 24, con L límite de la consulta.
- *Slice esperado*: Todos. ($\{15, 16, 17, 18, 19, 22, 24\}$). El statement s_{21} puede o no ser incluído en el slice (ver sección de correctitud y precisión del capítulo uno).

Esta consulta tiene dos características:

- Uso limitado del framework. Se establece el límite y se pide la lista de empleados.
- Genera callbacks. La consulta desata la creación de objetos del tipo *Employee*, cuyos atributos son cargados por Hibernate utilizando los setters publicos. Por tal motivo, cada setter ejecutado es un callback de *c.List()*.

[3] https://dev.mysql.com/doc/employee/en/

```
15  var c = session.CreateCriteria(typeof(Employee));
16  c.SetMaxResults(limit);
17  var results = c.List();
18  string s = "||";
19  for (int i = 0; i < results.Count; i++)
20  {
21      Employee employee = (Employee)results[i];
22      s = s + employee.FirstName + "|" + employee.LastName;
23  }
24  s = s + "||";
```

Fig. 4.9: Consulta NHibernate Q1 sobre base de datos de Empleados en lenguaje C#.

La Tab. 4.15 contiene los resultados de la ejecución de la consulta Q1, con distintos límites. Los tiempos de ejecución iniciales de JavaSlicer son algo más elevados que los de NetSlicer, pero a medida que aumenta el límite de la consulta, el segundo aumenta de manera notable sobrepasando al primero, lo que se aprecia de manera gráfica en la Fig. 4.10.

Límite	**T.NetSlicer (s)**	**T.JavaSlicer (s)**	**Traza NetSlicer**	**Traza JavaSlicer**
1000	18.14	274.17	301KB	168MB
1500	48.87	307.21	451KB	211MB
2000	103.74	333.67	602KB	257MB
2500	189.26	400.05	752KB	301MB
3000	309.58	417.12	903KB	345MB
3500	467.26	438.82	1.02MB	393MB
4000	667.93	460.39	1.17MB	437MB
4500	1,191.28	494.48	1.32MB	480MB

Tab. 4.14: Comparación de tiempos y tamaño de traza para consulta Hibernate con callbacks (Consulta Q1).

El comportamiento de la función de los tiempos de NetSlicer se puede acotar entre n^2 y n^3 con n el límite de empleados en la query (Fig. 4.11), exceptuando el valor para 4500, donde se observa deterioro en la performance debido a falta de memoria que conduce al efecto de thrashing mencionado en la sección anterior. Al igual que los resultados obtenidos con el algoritmo de multiplicación de matrices, la explicación de los tiempos elevados de NetSlicer se halla en la forma que adquiere el PTG.

La Fig. 4.12 muestra un subgrafo obtenido luego de la ejecución del statement s_{17} (método *c.List()*) con límite en tres. En él, se ven cada uno de los nodos que representan los objetos Employee creados, el objeto que representa la lista que los contiene, la región (Hub 12) resultado de la compresión de aquellos hubs creados en los callbacks y los puntos de entrada *employee* y *results*. El recorrido de una propiedad del objeto employee (por ejemplo FirstName) sobre este PTG se acota por n^2. Primero por *employee* se accede al grafo, lo que retorna la región, se clausura por lambda como indica el algoritmo, que resulta en todos los nodos de los empleados (n), luego se navega por la propiedad y por Σ, lo que retorna nuevamente hub y por último se vuelve a clausurar por lambda que termina dando **todos** los nodos del grafo ($\geq n$). Este procedimiento lo hace n veces (por cada empleado resultado), recorriendo n nodos y luego por cada uno, otra vez n nodos, es

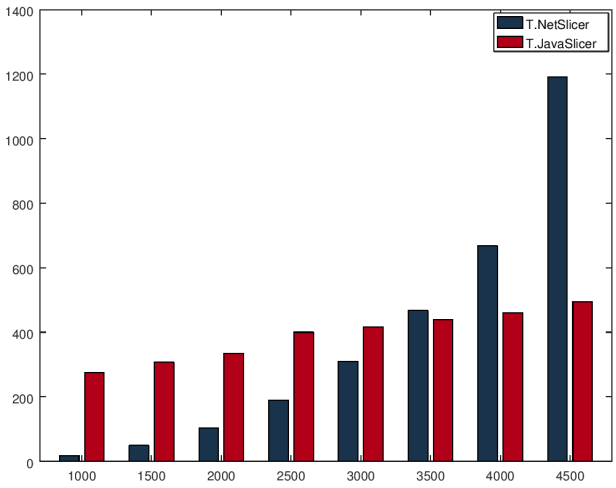

Fig. 4.10: Comparativa entre NetSlicer y JavaSlicer para consulta Hibernate con callbacks (Consulta Q1).

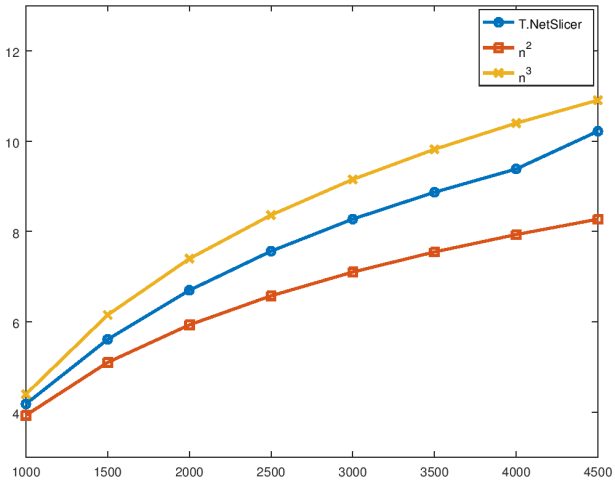

Fig. 4.11: Comportamiento de NetSlicer para consulta Hibernate con callbacks (Consulta Q1).

decir toma n^3 operaciones. Como se dijo anteriormente, no llega a ser n^3 debido a que el algoritmo de recorrido no es exactamente el del pseudocódigo de la Fig. 3.13, sino que se optimiza evitando recorrer nuevamente nodos que ya se encuentran en el resultado, por lo que queda acotado entre n^3 y n^2.

Con respecto a los tiempos de cómputo obtenidos con JavaSlicer, no parecen estar relacionados directamente al tamaño de la entrada (Fig. 4.13), pero si al tamaño de la traza (Fig. 4.14) ya que el tiempo de cómputo está asociado la cantidad de bytecodes ejecutados y estos crecen a medida que se incrementa el límite de empleados en la consulta y esta crece linealmente a razón de 40MB en cada aumento de la entrada. Este incremento paulatino, le permite a la herramienta analizar el programa con límites muy elevados, se

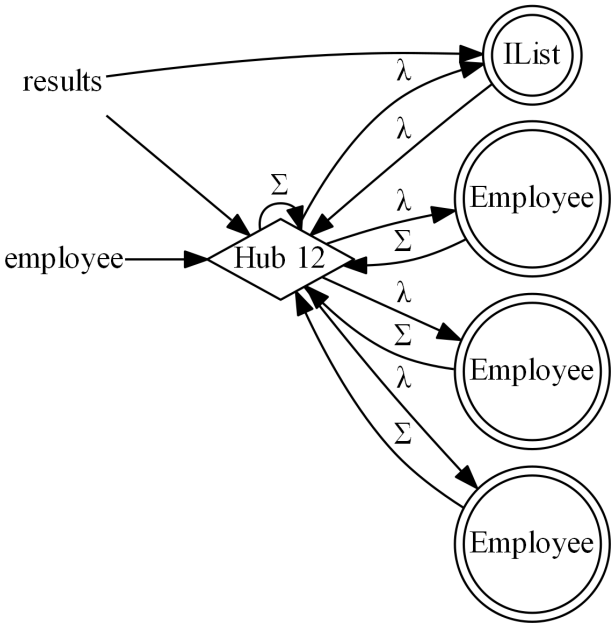

Fig. 4.12: Subgrafo del PTG generado para la consulta de empleados en NHibernate.

lograron resultados con éxito para 10000 empleados 943.25 segundos.

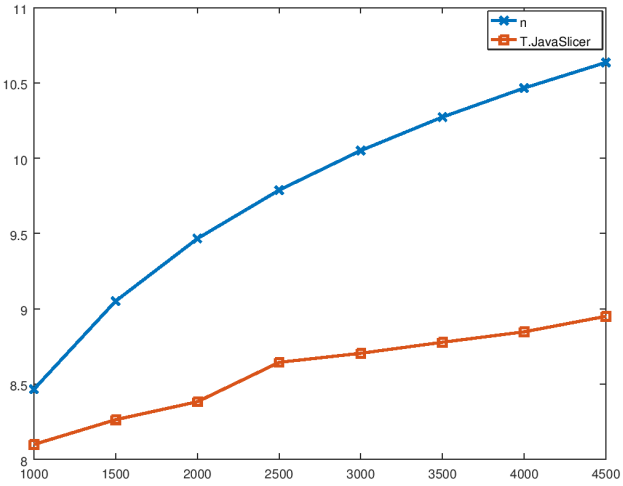

Fig. 4.13: Comportamiento de JavaSlicer para consulta Hibernate con callbacks (Consulta Q1).

El resultado del slice para NetSlicer fue el esperado, pero JavaSlicer incluyó el statement s_{21} y eliminó el s_{18} lo que es un error de correctitud, ya que la variable s se utiliza para redefinirse en la línea 22, pues el valor del string contendrá el valor de la línea 18 también. Esto parece ser un error asociado a strings, pues con cualquier otro tipo la dependencia se incluye.

En los programas con entorno gráfico que requieren participación del usuario, es ha-

Límite	E.Original C# (s)	E.NetSlicer (s)	E.Original Java (s)	E.JavaSlicer (s)
1000	1.41	1.56	3.13	89.17
1500	1.42	1.62	3.18	91.21
2000	1.45	1.72	3.23	93.67
2500	1.47	1.78	3.26	95.05
3000	1.49	1.82	3.29	97.12
3500	1.51	1.91	3.34	99.82
4000	1.52	1.95	3.41	101.39
4500	1.55	1.99	3.44	102.48

Tab. 4.15: Comparación de tiempos entre el programa original y el instrumentado para consulta Hibernate con callbacks (Consulta Q1).

bitual reproducir un error, interactuando con el programa repitiendo una serie de pasos. Estos pasos pueden ser el "input" para un criterio de slice determinado. Entonces la experiencia del usuario con el programa instrumentado generando traza debe ser eficiente. Este es un punto que se debe analizar.

La Fig. 4.15, contiene los tiempos de la consulta Q1 original y la instrumentada para ambos slicers. Mientras que el programa instrumentado por JavaSlicer aumenta su tiempo de cómputo aproximadamente 30 veces más, con NetSlicer sólo aumenta en 1.2. Es razonable pensar que cuanto más uso intensivo de framework utilice la aplicación, más es el tiempo en que va a tardar un programa instrumentado por JavaSlicer debido a que instrumenta absolutamente todo el código, mientras que con NetSlicer la diferencia sería prácticamente imperceptible. Es claro que el slicer de .NET, logra el objetivo que se propuso en la decisión de instrumentar fuente y no CIL (o lenguaje de bajo nivel), como el resto de las herramientas.

Q.2: Proyectar ciertos atributos de los empleados de la base de datos que son ingenieros y se contrataron antes del año 1990 (Fig. 4.14) utilizando *Criteria Queries*, observando los tiempos de cómputo y tamaño de la traza a medida que aumenta el límite de la consulta.

- *Criterio:* $C = (L, s^U_{43}, \{s\})$, variable s del statement 43, con L límite de la consulta.
- *Slice esperado*: Todos. ({18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 37, 38, 39, 40, 41, 43}).

Esta consulta tiene dos características:

- Mayor uso del framework respecto a Q1. Se utilizan más métodos para armar la consulta, alias, restricciones y proyecciones.
- No genera callbacks. Utilizando proyecciones el framework genera una lista de lista de objetos, cada lista en el resultado contiene los valores proyectados. De esta forma se evitan callbacks.

La Tab. 4.16, muestra los tiempos obtenidos. Es notable la diferencia obtenida en tiempos para los slicers con respecto a Q1. En este caso NetSlicer puede escalar considerablemente mejor que JavaSlicer. Ambos incrementan los tiempos de forma lineal con respecto al tamaño de la entrada, el primero a razón de 25 segundos mientras que el segundo 150 segundos. A partir de 30000, JavaSlicer empieza a tener los problemas de memoria

```
18  var c = session.CreateCriteria(typeof(Employee), "employee");
19  c.CreateAlias("employee.DeptEmps", "deptEmps");
20  c.CreateAlias("deptEmps.Department", "department");
21  var co = Restrictions.Conjunction()
22                .Add(Restrictions.Eq("TitleId.Title", "Engineer"));
23  c.CreateCriteria("Titles").Add(co);
24  ProjectionList proList = Projections.ProjectionList();
25  proList.Add(Projections.Property("employee.FirstName"));
26  proList.Add(Projections.Property("employee.LastName"));
27  proList.Add(Projections.Property("employee.HireDate"));
28  proList.Add(Projections.Property("department.DeptName"));
29  var d1990 = QueryUtils.DateParse("1990-01-01");
30  c.Add(Restrictions.Le("employee.HireDate", d1990));
31  c.SetProjection(proList);
32  c.SetMaxResults(limit);
33  var results = c.List();
34  string s = "||";
35  foreach (object[] result in results)
36  {
37    string firstName = (string)result[0];
38    string lastName = (string)result[1];
39    DateTime hireDate = (DateTime)result[2];
40    string deptName = (string)result[3];
41    s += firstName+"|"+lastName+"|"+deptName+"|"+hireDate;
42  }
43  s = s + "||";
```

Fig. 4.14: Consulta NHibernate Q2 sobre base de datos de Empleados en lenguaje C#.

antes mencionados, lo que resulta en un salto en tiempos que no condice con el incremento linea y para 35000 el programa termina por límite de memoria excedido por el GC. El gráfico de barras de la Fig. 4.15 compara los tiempos entre NetSlicer y JavaSlicer evitando los valores para 30000 y 35000.

Limite	**T.NetSlicer (s)**	**T.JavaSlicer (s)**	**Traza NetSlicer**	**Traza JavaSlicer**
5000	23.88	342.07	632KB	248MB
10000	49.49	464.39	1.23MB	420MB
15000	73.51	609.39	1.84MB	591MB
20000	100.17	728.81	2.46MB	763MB
25000	126.82	899.33	3.07MB	935MB
30000	157.11	3665.29	3.69MB	1.1GB
35000	180.08	7718.02 (E.M)	4.21MB	1.3GB

Tab. 4.16: Comparación de tiempos y tamaño de traza para consulta Hibernate sin callbacks (Consulta Q2).

La mejora en performance de NetSlicer con respecto a la consulta Q1 está en la ausencia de callback. Como se comentó en el capítulo anterior, el tratamiento de callbacks es complejo, toma tiempo de cómputo y desata también la compresión de hubs, lo que le da una ventaja en este caso. No obstante, el verdadero problema en Q1 era el hub que conectaba todos los nodos del grafo que se generaba por a los callbacks. Si bien en este

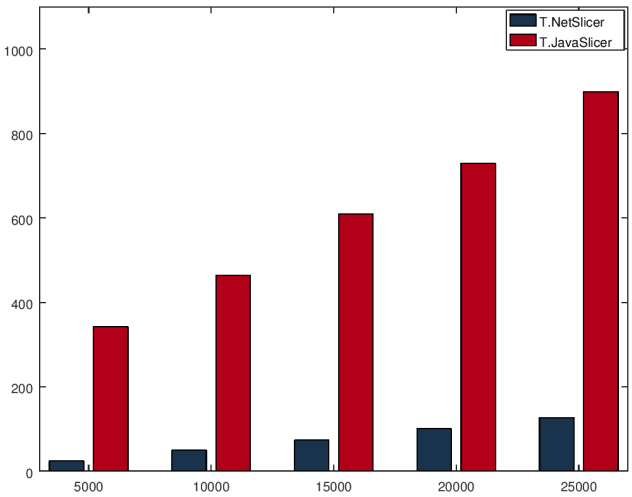

Fig. 4.15: Comparativa entre NetSlicer y JavaSlicer para consulta Hibernate sin callbacks (Consulta Q2).

caso el PTG es más grande que en el anterior (no se muestra por su extensión incluso para límite en 1), la obtención de los nodos que representan el acceso indexado a *result* (*result[i]*), sólo recorre un subgrafo de unos pocos nodos.

El resultado obtenido por NetSlicer fue el esperado, el de JavaSlicer por otro lado no contiene varios statements.

- No contiene la declaración del string *s* como en Q1
- No aparecen todos los statements relacionados a la lista de proyecciones (s_{24}, s_{25}, s_{26}, s_{27}, s_{28} y s_{30}). Puede ser anti intuitivo que no aparezcan estas líneas pues la lista de proyecciones finalmente va a definir que objectos son retornados en la lista, por lo que no parece del todo correcto que se eviten. Para el análisis de NetSlicer la dependencia de *s* con *proList* es clara, *results* es el resultado de *c.List()*, que puede haber usado cualquier objeto al que pueda referenciar, y proList es uno de ellos por la ejecución del método *SetProyection*, no obstante puede que JavaSlicer los evite porque los valores obtenidos (*firstName* por ejemplo) se extraen de base de datos y no tiene una dependencia de datos real con la proyección. Este es un caso claro de diferencia en el criterio de "correctitud" presentado la sección en el capítulo uno.
- No se encuentran las líneas que corresponden a las componentes del string, *firstName*, *lastName*, etc. (s_{36}, s_{37}, s_{38} y s_{39}). Esto parece un error de correctitud, pues no existe una explicación clara para que desaparezcan. Puede que sea por algún tipo de optimización realizada en el bytecode, en cuyo caso podría estar evidenciando problemas de trazabilidad entre el bytecode y el código que devienen en errores de correctitud.

En la Tab. 4.17 se encuentran los resultados obtenidos para la ejecución del programa con y sin instrumentar tanto para con ambos slicers. De la misma forma que con Q1, la

Limite	E.Original C# (s)	E.NetSlicer (s)	E.Original Java (s)	E.JavaSlicer (s)
5000	1.57	1.92	3.32	85.65
10000	1.67	2.03	3.34	105.48
15000	1.83	2.47	3.39	116.33
20000	2.07	2.78	3.44	127.25
25000	2.12	3.15	3.63	136.51
30000	2.23	3.41	3.82	157.93
35000	2.43	3.79	3.95	162.66

Tab. 4.17: Comparación de tiempos entre el programa original y el instrumentado para consulta Hibernate sin callbacks (Consulta Q2).

diferencia entre el código C# y el instrumentado por NetSlicer es mínima no así entre el original Java y el instrumentado por JavaSlicer.

Este tipo de resultados se repiten sea cual sea el programa analizado, pues siempre la traza de código de bajo nivel es mucho más voluminosa que la de alto nivel, como se aprecia en los tamaños de archivos de traza obtenidos para Q1 y Q2 (Fig. 4.14 y Fig. 4.16 respectivamente).

En la consulta **Q1** JavaSlicer supera a NetSlicer debido a dos factores; por un lado la cantidad de bytecodes ejecutados por Hibernate, no es suficiente como para resultar un problema en JavaSlicer, por otro, el programa genera un efecto similar al de la multiplicación de matrices en el PTG de NetSlicer. El algoritmo conservativo crea una región que debe englobar a todos los empleados (pues el framework puede realizar aliasing entre todos los objetos), lo que produce que al obtener un campo de *Employee*, se obtengan todos los objetos implicados.

En contraposición, NetSlicer es superior en la prueba **Q2**. En este caso, si bien el grafo es considerablemente más grande que en Q1, el método *List()* sólo retorna un objeto del tipo array (no genera callbacks) que se liga a la región que representa le llamado al método y en ésta se encuentran representados todos los objetos creados. De este modo, cuando se accede a un objeto de la región, el recorrido retorna una cantidad limitada de nodos.

4.4. Precisión

En el marco del desarrollo de NetSlicer, se realizó un análisis de comparativo de precisión contra JavaSlicer. En esta sección se mostrarán brevemente esos resultados.

Las pruebas se realizaron sobre una adaptación de Java y de C# del benchmark Olden[4], desarrollado originalmente en C, para pruebas de paralelismo en supercomputadoras *Thinking Machines CM-5*. Las adaptaciones no conservan las características paralelas sino que son útiles en la medición de performance. En estas pruebas sólo se utilizaron para tener dos implementaciones prácticamente iguales de los mismos programas y visualizar cuán precisos son las herramientas para cada uno.

La Tab. 4.18 muestra los resultados obtenidos para algunos de los programas incluidos en la suite.

En la mayoría de los casos, se ve que el tiempo de ejecución de NetSlicer es menor con inputs pequeños pero mayor a medida que se aumenta, hecho que se vio en pruebas

[4] http://www.martincarlisle.com/olden.html

		NetSlicer			JavaSlicer		
Programa	**Inputs**	**Tiempo (s)**	**# Líneas**	**% S/T**	**Tiempo (s)**	**# Líneas**	**% S/T**
BH	2;10	2	233	57	16	214	52
BH	40;10	126	264	64	30	264	64
BiSort	1000	68	95	81	22	78	67
BiSort	2000	218	95	81	27	75	64
Em3d	50;10;3	6	82	61	16	42	31
Health	4;2;8	5	30	16	13	30	16
Health	4;5;20	224	152	79	23	119	62
TSP	20	1	160	65	13	146	59
TSP	100	6	170	69	14	156	63

Tab. 4.18: Comparación de precisión entre NetSlicer y JavaSlicer para el benchmark Olden.

realizadas anteriormente. En cuanto a la precisión, el porcentaje de líneas obtenidas en el slice sobre el total de líneas (%S/T) es mayor en NetSlicer que en JavaSlicer, que puede deberse no sólo a casos similares a los presentados en la sección anterior, sino también al hecho de que el algoritmo de JavaSlicer es considerado preciso (por estar basado en el algoritmo NP de Zhang et. al) mientras que el de NetSlicer, es impreciso por no analizar los llamados a métodos no instrumentados.

Como es esperado, los resultados favorecen a JavaSlicer por ser un slicer preciso, por lo que logra incluir en sus resultados menos líneas que NetSlicer. No obstance, en la mayoría de los resultados no se aprecian diferencias realmente significativas, lo que justifica la decisión de NetSlicer de sacrificar precisión en pos de evitar el análisis de código externo.

5. CONCLUSIONES

En este trabajo se compararon cuatro herramientas de la técnica de Slicing Dinámico, WET para código C, JSlice y JavaSlicer para código Java y por último NetSlicer para código C#. Para cada uno de ellos se presentaron fundamentos teóricos, los algoritmos utilizados y las decisiones de diseño de arquitectura tomadas y se presentaron ventajas y desventajas de cada uno de estos aspectos. Además, de cada uno de las herramientas se detallo el uso de estructuras de soporte auxiliares como el el Grafo Dinámico de Dependencias (DDG), o el grafo de Points-To (PTG) utilizado por NetSlicer.

Se introdujeron dos tipos de algoritmos. Los algoritmos llamados *precisos*, utilizan información dinámica, como identificadores de objetos o direcciones de memoria que tienen como objetivo seguir las modificaciones de las variables y solucionar los problemas de aliasing, por lo que están obligados a trazar todo el programa incluyendo el código de terceros. Este tipo de algoritmos es utilizado en WET, JSlice y JavaSlicer. NetSlicer por otro lado, realiza un procedimiento basado en análisis simbólico sobre el control flow del programa y no analiza código de terceros, por lo que debe predecir el efecto que tiene un método cuyo código es desconocido sobre las dependencias de forma conservativa para mantener correctitud, lo que lo convierte en un algoritmo *impreciso*.

Se realizaron pruebas comparativas de eficiencia y precisión, así como también de la capacidad de slicing de programas reales, más complejos que los casos de estudios académicos, construidos sobre frameworks y librerías, con el objetivo de analizar la escalabilidad de la técnica en programas reales.

En las pruebas de eficiencia la solución propuesta por JSlice, que consta en la modificación de una VM para analizar cada bytecode ejecutado sin la necesidad de instrumentar y trazar, fue claramente superior al del resto de las herramientas. El slicer es considerablemente rápido y logra escalar mucho más que el resto. Pese a los problemas encontrados en WET, el modo de ejecución limitado más el pos procesamiento de la traza (L+PS) arrojó buenos resultados y logró escalar sin inconvenientes. Con JavaSlicer se obtuvieron tiempos de ejecución algo más elevados y con problemas de consumo de memoria en casos donde los anteriores no arrojaron problemas. Por último el carácter impreciso de NetSlicer resulta una desventaja en casos donde se analizan funciones no instrumentadas de forma conservativa, pero que no implican gran cantidad de cómputo en las herramientas precisas pues no se traducen en una cantidad considerable de statements de bajo nivel.

Si bien JSlice y WET parecen ser las mejores soluciones en performance, las decisiones en la arquitectura los convierten en herramientas obsoletas, no sólo por ser antiguas (ambas realizadas en el 2005), sino por estar montadas sobre infraestructura que aún re implementándolos nuevamente, no podrían ser mantenidas con facilidad. WET requiere de la herramienta Diablo para las dependencias de control y que todo el código se compile de forma estática, además al montarse en Valgrind, requiere condiciones específicas para ser compilado y ejecutado. JSlice por otro lado, es una modificación de una máquina virtual java específica de código abierto, sin embargo el usuario general de Java, utiliza la provista por Oracle pero esta es privada y no se puede modificar. Además al ser una VM modificada arroja problemas al utilizar librerías que requieren de elementos que no implementan.

Por los motivos antes presentados, las pruebas de aplicaciones modernas se realizaron sólo con JavaSlicer y NetSlicer. El uso de librerías y frameworks no modifican el comportamiento de JavaSlicer, pues éste está ligado a la cantidad de bytecodes que se ejecutan, sean o no del programa analizado. En contraposición, NetSlicer realiza un análisis impreciso que modifica su comportamiento dependiendo el efecto de las llamadas no instrumentadas sobre los objetos alcanzables por las variables del programa. En una prueba, el análisis conservativo de NetSlicer obliga a operar sobre prácticamente la totalidad de los objetos creados, lo que resulta en tiempos de cómputo considerablemente mayores que los de JavaSlicer, sin embargo, en otra prueba se corroboró que la decisión de no analizar código de terceros puede otorgarle a NetSlicer gran capacidad de escalabilidad.

En conclusión, el slicing dinámico no es utilizado en la actualidad debido al hecho de que las aplicaciones se apoyan cada vez más en librerías y frameworks, lo que es un problema para la escalabilidad de los slicers precisos como WET, JSlice y JavaSlicer. Sin embargo NetSlicer presenta una alternativa, que de resolverse los problemas presentados en el trabajo, puede llegar a convertir a la técnica, por primera vez, en una herramienta capaz de analizar aplicaciones reales.

Bibliografía

[1] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Dynamic slicing in the presence of unconstrained pointers. In *Proceedings of the symposium on Testing, analysis, and verification*, pages 60–73. ACM, 1991.

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

[3] 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.

[4] Á. Beszedes, T. Gergely, Z. M. Szabo, J. Csirik, and T. Gyimothy. Dynamic slicing method for maintenance of large c programs. In *Software Maintenance and Reengineering, 2001. Fifth European Conference on*, pages 105–113. IEEE, 2001.

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

[6] G. Canfora, A. Cimitile, and M. Munro. Re2: Reverse-engineering and reuse re-engineering. *Journal of Software: Evolution and Process*, 6(2):53–72, 1994.

[7] 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.

[8] 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.

[9] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE transactions on software engineering*, 17(8):751–761, 1991.

[10] N. Gupta, H. He, X. Zhang, and R. Gupta. Locating faulty code using failure-inducing chops. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 263–272. ACM, 2005.

[11] N. Gupta and P. Rao. Program execution based module cohesion measurement. In *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*, pages 144–153. IEEE, 2001.

[12] R. Gupta and M. L. Soffa. Hybrid slicing: An approach for refining static slices using dynamic information. *ACM SIGSOFT Software Engineering Notes*, 20(4):29–40, 1995.

[13] C. Hammacher. Design and implementation of an efficient dynamic slicer for java. *Bachelor's Thesis, November*, 2008.

[14] 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.

[15] M. Harman, S. Danicic, Y. Sivagurunathan, and D. Simpson. The next 700 slicing criteria. In *Proceedings of the 2nd UK Workshop on Program Comprehension*, pages 1–16, 1996.

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

[17] R. Hierons, M. Harman, and S. Danicic. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability*, 9(4):233–262, 1999.

[18] 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.

[19] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, 1990.

[20] D. Jackson and E. J. Rollins. Chopping: A generalization of slicing. Technical report, PA, USA, 1994.

[21] A. J. Ko and B. A. Myers. Extracting and answering why and why not questions about java program output. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(2):4, 2010.

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

[23] 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.

[24] J. Ming, D. Wu, G. Xiao, J. Wang, and P. Liu. Taintpipe: Pipelined symbolic taint analysis. In *USENIX Security Symposium*, pages 65–80, 2015.

[25] V. Nagarajan, D. Jeffrey, R. Gupta, and N. Gupta. Ontrac: A system for efficient online tracing for debugging. In *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, pages 445–454. IEEE, 2007.

[26] J. Q. Ning, A. Engberts, and W. V. Kozaczynski. Automated support for legacy code understanding. *Communications of the ACM*, 37(5):50–58, 1994.

[27] T. Reps and T. Bricker. *Illustrating interference in interfering versions of programs*, volume 14. ACM, 1989.

[28] N. F. Rodrigues and L. S. Barbosa. Slicing for architectural analysis. *Science of Computer Programming*, 75(10):828–847, 2010.

[29] Y. Sazeides. Instruction–isomorphism in program execution. *Journal of Instruction-Level Parallelism*, 5:1–22, 2003.

[30] J. Silva. A vocabulary of program slicing-based techniques. *ACM computing surveys (CSUR)*, 44(3):12, 2012.

[31] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. *ACM SIGPLAN Notices*, 42(6):112–122, 2007.

[32] H. Tang, X. Wang, L. Zhang, B. Xie, L. Zhang, and H. Mei. Summary-based context-sensitive data-dependence analysis in presence of callbacks. In *ACM SIGPLAN Notices*, volume 50, pages 83–95. ACM, 2015.

[33] G. A. Venkatesh. The semantic approach to program slicing. In *ACM SIGPLAN Notices*, volume 26, pages 107–119. ACM, 1991.

[34] 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.

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

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

[37] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes*, 30(2):1–36, 2005.

[38] G. Xu, N. Mitchell, M. Arnold, A. Rountev, E. Schonberg, and G. Sevitsky. Scalable runtime bloat detection using abstract dynamic slicing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(3):23, 2014.

[39] X. Zhang, N. Gupta, and R. Gupta. Pruning dynamic slices with confidence. In *ACM SIGPLAN Notices*, volume 41, pages 169–180. ACM, 2006.

[40] 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.

[41] X. Zhang and R. Gupta. Cost effective dynamic program slicing. In *ACM SIGPLAN Notices*, volume 39, pages 94–106. ACM, 2004.

[42] 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.

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

[44] 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.

[45] 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.

[46] X. Zhang, S. Tallam, and R. Gupta. Dynamic slicing long running programs through execution fast forwarding. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 81–91. ACM, 2006.

[47] C. B. Zilles and G. S. Sohi. *Understanding the backward slices of performance degrading instructions*, volume 28. ACM, 2000.