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

Diseño e implementación de un generador de código .NET

Tesis de Licenciatura en Ciencias de la Computación

Francisco Nicolás Curdi

Director: Dr. Edgardo Julio Zoppi

Codirector: Dr. Diego Garbervetsky

Buenos Aires, 2021

Resumen

En esta tesis presentamos el diseño e implementación de un módulo de generación de código para el *framework* de análisis estático de código Analysis.NET. Esta herramienta permite leer ejecutables de .NET y realizar distintos análisis sobre ellos. Si bien la misma existe hace tiempo ya, actualmente tiene la limitante de no poder generar código ejecutable. El módulo agregado busca cubrir esa brecha y lograr una herramienta más completa, como las que podemos encontrar para otras tecnologías similares.

Adicionalmente incluimos otro módulo que permite realizar una conversión entre dos representaciones intermedias que brinda el *framework*. Esto, sumado a la nueva funcionalidad de generación de código, nos permite hacer uso de distintos análisis y transformaciones que provee la herramienta pudiendo impactar cambios en un nuevo ejecutable.

En este trabajo presentamos la plataforma .NET, junto con su arquitectura, y el *framework* Analysis.NET. Detallamos el diseño e implementación de los módulos agregados explicando la problemática que resuelven y las dificultades encontradas durante su implementación.

Nos concentramos en cuatro ejes de estudio que nos ayudarán a entender el valor de la funcionalidad agregada. Estos son Generación, Instrumentación, Optimización y Generación programática, siendo el primero de ellos el foco principal de este trabajo.

Por último realizamos una evaluación empírica, experimentando la herramienta completa con distintos casos de pruebas para los cuales evaluamos los resultados obtenidos para cada eje.

Palabras clave: análisis, programa, estático, .NET, framework, bytecode, generación, metadata, CIL, código, ejecutable.

Índice general

1. INTRODUCCIÓN

El análisis estático de código es el proceso automatizado de analizar características de un programa sin la necesidad de ejecutar el mismo. Este análisis puede realizarse sobre alguna representación del programa como es el código fuente o directamente sobre el binario generado por el compilador. Este tipo de análisis es utilizado en distintas áreas y con diversos objetivos entre los cuales podemos destacar la optimización y correctitud. La primera consiste en cambios o transformaciones que apuntan a mejorar la performance del programa y reducir el uso de recursos. Correctitud en cambio busca asegurar que el programa se comporta como esperamos y que haga lo que tiene que hacer.

En la actualidad muchas de las herramientas utilizadas durante el proceso de desarrollo como pueden ser compiladores, analizadores de código, *integrated development environments* (IDEs) o bien distintas herramientas de *software* utilizan ampliamente este tipo de análisis. Esto permite generar código de mayor calidad y a su vez ayudar al programador facilitando la tarea de desarrollo y previniendo errores. Estas herramientas suelen contar también con otros tipos de análisis, como pueden ser análisis dinámicos de código los cuales evalúan distintas carcterísticas del programa pero en tiempo de ejecución.

En esta tesis trabajamos en la extensión del *framework* de análisis estático de código Analysis.NET. Introducimos primero la tecnología sobre la que opera, .NET, así como también la herramienta en sí misma ya que más adelante en este trabajo hacemos foco en la funcionalidad agregada y en su implementación dentro del *framework* de trabajo.

1.1. .NET

.NET [1] es una plataforma de desarrollo que permite construir distintos tipos de aplicaciones como pueden ser web, mobile, desktop, juegos o sistemas de *internet of things* (IoT). Desarrollado por Microsoft en el año 2002, .NET es altamente utilizado en la industria del *software*.

1.1.1. Lenguajes

Si bien C# [2] es el lenguaje más comunmente asociado con esta plataforma, .NET está diseñado para permitir el uso de distintos lenguajes de programación [3]. De esta forma incorpora lenguajes con distintas características lo cual permite elegir el más adecuado para el problema a resolver. Entre los más populares podemos encontrar C#, F# [4] y Visual Basic [5]:

- **C#:** Se caracteriza por ser fuertemente tipado, orientado a objetos, con aspectos tanto del paradigma imperativo como del funcional y con soporte de polimorfismo

paramétrico[1]. C# fue desarrollado por Microsoft y tiene sus raíces en la familia de lenguajes C por lo que resulta más que amigable para desarrolladores familiarizados con C, C++, Java y similares.

- **F#:** Actúa como contraparte de C# siendo primariamente funcional pero con soporte del paradigma de objetos e imperativo. Esto resulta en un lenguaje de propósito general más versátil que los lenguajes puramente funcionales. Es desarrollado por la F# Software Foundation [6] en conjunto con Microsoft y está influenciado por lenguajes como C#, Python, Haskell, Scala y Erlang.
- **Visual Basic:** Lenguaje de alto nivel orientado a objetos desarrollado por Microsoft. Está diseñado para ser fácil de aprender y cuenta con una sintaxis que busca favorecer la claridad del código. Visual Basic permite ser utilizado tanto de forma fuertemente tipada como de forma dinámica.

1.1.2. Arquitectura

Para poder ser multilenguaje, y multiplataforma, la arquitectura de .NET debe considerar estos aspectos en su diseño. Para ello, se define una infraestructura que establece un terreno común, lo cual permite interoperar entre los distintos lenguajes. La misma es conocida como *Common Language Infrastructure*.

Common Language Infrastructure

Provee una especificación del código ejecutable y entorno de ejecución la cual es independiente de la plataforma subyacente. Esto permite la utilización de distintos lenguajes de alto nivel sin tener que lidiar con características específicas de la arquitectura de cada plataforma particular. *Common Language Infrastructure*, o bien CLI, fue desarrollada por Microsoft y estandarizada por ISO y ECMA [7]. Especifica los siguientes componentes:

- **Common Type System (CTS):** Define un sistema de tipado común para los tipos y operaciones encontradas en los distintos lenguajes soportados por .NET. Más aún, permite la interacción entre los mismos. Un tipo describe un conjunto de valores y operaciones sobre ellos. CTS proveee un modelo orientado a objetos que soporta las características conocidas de este paradigma como son el uso de tipos paramétricos, encapsulamiento y polimorfismo o interfaces. Este sistema clasifica los tipos en *Reference Types* y *Value Types*[2] y provee mecanismos de conversión[3] entre los mismos.

[1] También conocido como *generics*. Se puede encontrar más información en https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/generics.

[2] Dentro de los *Reference Types* podemos encontrar clases e interfaces así como también algunos tipos incluidos en .NET como Object o String. Ejemplos de *Value Types* en cambio pueden ser *enums* o *structs*.

[3] La conversión de *Value Type* a *Reference Type* es conocida como *boxing* mientras que el proceso inverso es denominado *unboxing*.

- **Metadata:** Describe y referencia los tipos y miembros definidos por el CTS junto con la información necesaria que se requiere para localizar y cargar las clases, instanciar los tipos en memoria, resolver invocaciones de métodos y demás cuestiones necesarias durante la ejecución. La metadata se almacena de forma independiente del lenguaje de programación lo cual establece un punto medio para los compiladores, *debuggers* y el entorno de ejecución. La misma se ubica en una porción del ejecutable junto con el código. Durante la ejecución, la metadata se mantiene en memoria lo cual permite referenciar los distintos elementos así como también el uso de *Reflection*[1].

- **Common Language Specification (CLS):** Define el conjunto de reglas y funcionalidades que deben implementar los lenguajes sobre .NET. Es la especificación que debe respetar cualquier lenguaje que quiera ser parte de la plataforma.

- **Virtual Execution System (VES):** Módulo responsable de la carga y ejecución de los programas escritos para el CLI. Provee los servicios necesarios para ejecutar el código usando la metadata para conectar los distintos módulos en tiempo de ejecución.

Common Language Runtime

.NET cuenta con su propio entorno de ejecución el cual consta de una máquina virtual que implementa la especifcacion del VES definido por el CLI. Este módulo se define como *Common Language Runtime*, o bien CLR, y es el encargado de la ejecución de los programas .NET. Provee distintos servicios para el manejo de memoria, chequeo de tipos, manejo de excepciones, *garbage collection*[2] y gestión de concurrencia.

Common Intermediate Language

Lenguaje definido en la especificación del CLI, anteriormente conocido como *Microsoft Intermediate Language* (MSIL). El *Common Intermediate Language*, o CIL, define un set de instrucciones que es independiente de la plataforma y es ejecutado por el CLR. Al igual que otros lenguajes de bajo nivel como Assembler, CIL es basado en *stack*, lo que significa que los parámetros y resultados de cada instrucción se mantienen en una pila en lugar de registros o posiciones de memoria arbitrarias. Sin embargo, es orientado a objetos lo cual lo torna en un lenguaje de más alto nivel que el anteriormente mencionado.

A continuación podemos ver una comparación de una clase en C# con su respectivo código CIL:

[1] Capacidad que tiene un programa para observar y opcionalmente modificar su estructura de alto nivel.
[2] Mecanismo implícito de gestión de memoria que se ocupa de monitorear el ciclo de vida de los objetos con el fin de asignar y liberar memoria de forma eficiente.

```
namespace Sample
{
  public class Person
  {
      public string Dni;

      public Person(string dni)
      {
          Dni = dni;
      }
  }
}
```

Listing 1.1: Clase Person en C#

```
.class public auto ansi beforefieldinit
    Sample.Person
      extends [mscorlib]System.Object
  {
    .field public string Dni

    .method public hidebysig specialname rtspecialname instance void
      .ctor(
        string dni
      ) cil managed
    {
      .maxstack 8

      IL_0000: ldarg.0
      IL_0001: call instance void [mscorlib]System.Object::.ctor()
      IL_0006: nop

      IL_0007: nop

      IL_0008: ldarg.0
      IL_0009: ldarg.1
      IL_000a: stfld string Sample.Person::Dni

      IL_000f: ret
    }
  }
```

Listing 1.2: Clase Person en CIL

En la versión en CIL podemos observar directivas como **.class**, **.method**, **.field** que permiten definir clases, métodos o atributos respectivamente y también algunas que especifican detalles de más bajo nivel como pueden ser **specialname** y **rtspecialname**. Yendo al cuerpo del método encontramos que se define la cantidad máxima de elementos que puede tener la pila mediante la directiva **.maxstack**, se llama al constructor de Object y luego carga el valor del campo dni. Durante este proceso se van leyendo y guardando los parámetros y resultados en la pila.

Cada instrucción se posiciona en un cierto *offset* del ejecutable, el cual es representado por una etiqueta o *label* en el formato de **IL_** seguido de un sufijo hexadecimal. La distancia entre estas etiquetas está asociada al tamaño que ocupa la instrucción etiquetada.

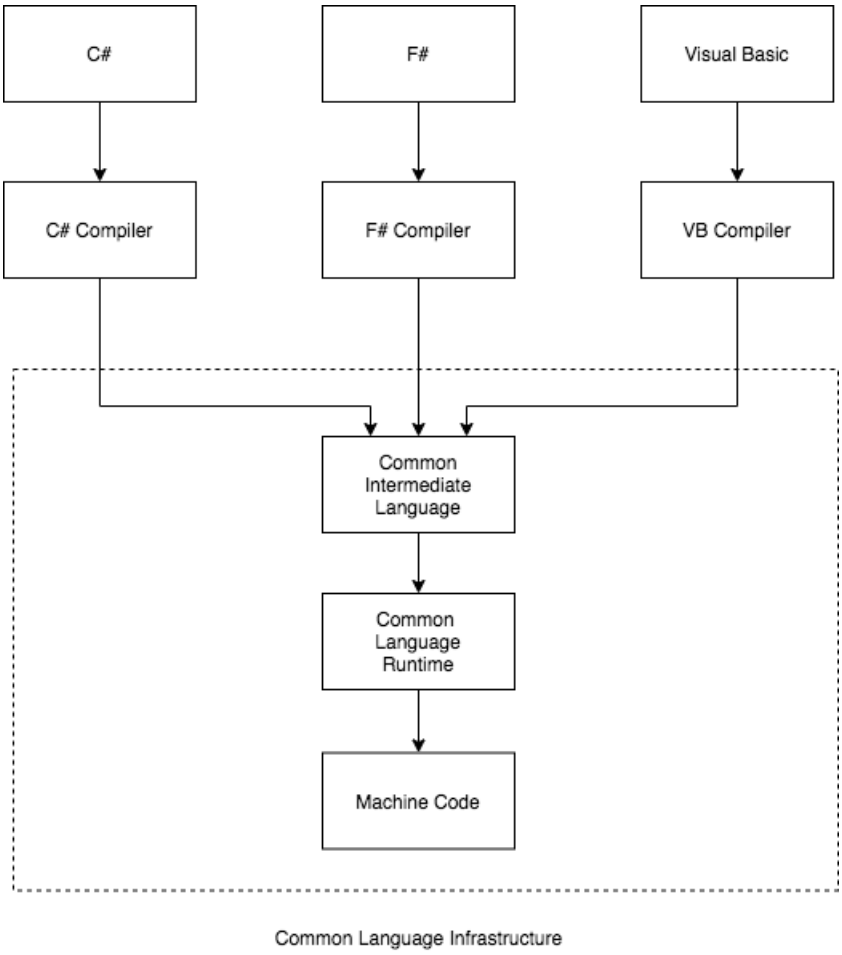

Fig. 1.1: Los compiladores de los distintos lenguajes transforman el código fuente al código CIL. El mismo es luego ejecutado por el CLR, el cual lo traduce finalmente en código que entiende la máquina sobre la que está corriendo el programa.

1.2. Analysis.NET

Analysis.NET[1] es un *framework* de análisis estático de código para programas .NET. Es de código abierto y esta escrito enteramente en C#. Contiene una colección de APIs que permiten ejecutar distintos análisis de código, construir nuevos y además cuenta con algunas herramientas de interfaz gráfica para visualizarlos. Provee distintas representaciones intermedias que permiten implementar estos análisis sobre abstracciones más adecuadas, como *Three Address Code* o *Static Single Assignment* [8], en lugar de trabajar directamente sobre el código fuente o sobre CIL.

Durante este trabajo haremos referencia a la herramienta ya sea por su nombre, Analysis.NET, o bien como *framework*.

1.2.1. Principios de diseño

Podemos destacar tres principios de diseño que caracterizan y definen la esencia del *framework*:

- **Basado en Bytecode:** Analysis.NET no trabaja sobre código fuente sino con *bytecode*. El *framework* parte del CIL y luego opera con distintas representaciones intermedias, lo cual permite trabajar con programas escritos en cualquiera de los lenguajes de .NET. Este enfoque tiene la ventaja de que se soportan automáticamente las distintas funcionalidades de alto nivel de estos lenguajes que se encuentran actualmente en los mismos o bien que se agreguen en un futuro. Permite además trabajar con programas donde no se cuenta con el código fuente original sino únicamente con un ejecutable.

- **Extensibilidad:** Al trabajar con representaciones intermedias, Analysis.NET busca desacoplarse de .NET permitiendo así la incorporación de lenguajes fuera del mismo. Por otro lado, los distintos componentes del *framework* estan diseñados de forma extensible por lo que es posible contar con distintas implementaciones para ellos. Si bien se proporciona una serie de análisis estáticos de código incluidos directamente en la herramienta, el *framework* está pensado para que sea sencillo agregar nuevos o bien modificar o extender los actuales.

- **Simplicidad sobre performance:** Analysis.NET busca ser fácil de utilizar y de entender. Evita optimizaciones por performance que comprometan la legibilidad, mantenibilidad y extensibilidad del *framework*.

[1] https://github.com/edgardozoppi/analysis-net.

1.2.2. Representaciones intermedias

Las representaciones intermedias facilitan la tarea de la herramienta ya que simplifican las complejidades de los lenguajes de alto nivel. Trabajar a nivel de código fuente, por ejemplo C#, implica soportar las distintas funcionalidades que provee el lenguaje, cuando éstas en realidad son construcciones sobre operaciones mucho más fundamentales. Ocurre algo similar yendo al otro extremo, con CIL, ya que si bien en este caso el nivel de abstracción es menor, contamos con muchas características de bajo nivel.

Las representaciones intermedias se ubican en el medio de estas dos y permiten abstraer los *features* de tan alto nivel de los lenguajes .NET pero sin tener tampoco que trabajar directamente con el código CIL. Analysis.NET provee distintas representaciones intermedias, cada una adecuada para cumplir cierto rol. En este trabajo haremos énfasis en dos de ellas, pero se puede encontrar el detalle de las demás en el *framework* [9].

1.2.2.1. SIL

Como mencionamos anteriormente, Analysis.NET tiene un enfoque basado en *bytecode* y representaciones intermedias. Como primera de ellas encontramos el *Simplified Bytecode*, o SIL (*Simplified IL*). Esta representación es basada en *stack* y tiene similitudes con CIL, pero busca simplificar algunos detalles de más bajo nivel con los que cuenta el mismo y además abstraerse del entorno, en este caso .NET.

Si bien actualmente Analysis.NET no soporta otros lenguajes que no sean parte de .NET, este enfoque facilita este tipo de extensiblidad ya que permite tener una representación agnóstica del lenguaje y plataforma lo cual constituyen el punto de partida para los distintos procedimientos que se realicen dentro del *framework*. Es decir, podríamos tener distintos lenguajes, dentro y fuera de .NET, para los cuales implementaríamos únicamente el pasaje de su *bytecode* a SIL y viceversa. El resto del *framework* trabaja con SIL o con otras representaciones intermedias para las cuales se necesita una única implementación de conversión entre las mismas. Tomemos como ejemplo Java. Si quisiéramos agregar soporte para este lenguaje, deberíamos únicamente implementar el pasaje del *bytecode* de Java al SIL y su vuelta. Si en cambio trabajaramos directamente con el *bytecode* de Java, además de lo anteriomente mencionado deberíamos tener traducciones, ida y vuelta, del *bytecode* de Java a las distintas representaciones intermedias del *framework*. Esto no solo complejiza el *framework* sino que además no exhibe ninguna ventaja sobre el enfoque actual.

Operaciones unificadas

CIL cuenta con más de 200 instrucciones que, si bien se pueden agrupar en varios segmentos, no deja de ser un número bastante grande. Haciendo un poco de zoom, encontramos que para la mayoría de estos segmentos hay una gran variedad de instrucciones que difieren en los operandos o los tipos que utilizan. Abstrayendo estas diferencias, SIL

representa las mismas operaciones con aproximadamente solo 20 instrucciones. A continuación analizaremos un fragmento de código simple para ver algunas de las diferencias entre CIL y SIL:

```
public float Average(int x, float y) {
    var sum = x + y;
    var result = sum / 2;
    return result;
}
```

Listing 1.3: Método *Average* en C#

```
{
  IL_0000: nop

  IL_0001: ldarg.1
  IL_0002: conv.r4
  IL_0003: ldarg.2
  IL_0004: add
  IL_0005: stloc.0

  IL_0006: ldloc.0
  IL_0007: ldc.r4 2
  IL_000c: div
  IL_000d: stloc.1

  IL_000e: ldloc.1
  IL_000f: stloc.2
  IL_0010: br.s IL_0012

  IL_0012: ldloc.2
  IL_0013: ret
}
```

Listing 1.4: Cuerpo del método *Average* en CIL

```
{
  L_0000: nop

  L_0001: load content of x
  L_0002: conv to Single
  L_0003: load content of y
  L_0004: add
  L_0005: store local_0

  L_0006: load content of local_0
  L_0007: load 2
  L_000c: div
  L_000d: store local_1

  L_000e: load content of local_1
  L_000f: store local_2
  L_0010: goto L_0012

  L_0012: load content of local_2
  L_0013: return
}
```

Listing 1.5: Cuerpo del método *Average* en SIL

Encontramos las siguientes diferencias:

- **ldarg-ldloc:** En el fragmento de CIL encontramos múltiples operaciones de carga de un argumento (**ldarg**) y carga de una variable local (**ldloc**). Todas ellas son instrucciones distintas ya que tienen un operando como parte de la misma que las diferencia: **ldarg.1**, **ldarg.2**, **ldloc.0**, **ldloc1**, **ldloc.2**. En el caso de SIL, tenemos una única instrucción **load** que las unifica. De forma análoga tenemos el caso de las instrucciones de guardado como **stloc**.
- **conv:** En el caso de esta operación de conversión, vemos que aparece como **conv.r4**. La misma está indicando que es una conversión a un tipo de datos *float*. Si bien

en este ejemplo no tenemos más casos de esta operación, podemos ver en el set de instrucciones de CIL que contamos con **conv.i1**, **conv.i2**, **conv.u1**, **conv.u2** y más variantes. En SIL esto se traduce a una única instrucción **conv** la cual cuenta con un parámetro que indica el tipo de dato de la conversión.

- **add-div:** En CIL contamos con una instrucción por cada operación matemática que querramos hacer. En su contraparte en SIL, tenemos una única instrucción que condensa todas estas variantes de operaciones, e incluso otras que no son aritméticas pero se comportan de la misma manera en cuanto a su aridad. En el ejemplo si bien las vemos como **add** y **div** también en SIL, esto es simplemente una ayuda visual al programador ya que a nivel implementación podemos ver que todas ellas se traducen a una única instrucción denominada **Basic Instruction**.

- **br:** En CIL encontramos distintos tipos de saltos, o *branches*, como pueden ser los condicionales e incondicionales. A su vez, CIL provee variantes más eficientes en cuanto espacio para diversas instrucciones en forma de *short forms*, lo cual agrega el sufijo **.s** a la instrucción como podemos ver en el ejemplo en cuestión. Al igual que en los casos anteriores, en SIL contamos con una única instrucción *branch* que se encarga de unificar estas variantes.

1.2.2.2. TAC

Three address code [8], o bien TAC, es un lenguaje intermedio utilizado por los compiladores para lo que respecta transformaciones de código, como pueden ser mejoras y optimizaciones. Cada instrucción puede tener hasta tres operandos, de ahí el nombre, y es generalmente una combinación de asignaciones y operadores de distintos tipos. Las instrucciones respetan el formato $\mathbf{x = y\ op\ z}$, donde **x**, **y**, **z** son nombres, constantes o variables temporales generadas por el compilador mientras que **op** representa un operador. En el caso de *arrays*, se pueden representar las operaciones de lectura y escritura de un elemento como $\mathbf{x = y[z]}$ y $\mathbf{x[y] = z}$ respectivamente. La ejecución del TAC es primariamente secuencial pero también permite instrucciones de control de flujo como saltos condicionales e incondicionales:

- **if x goto L:** Si **x** se cumple continuar en la instrucción con etiqueta **L**. Caso contrario seguir por la instrucción siguiente a la actual.

- **goto L:** Continuar en la instrucción con etiqueta **L**.

Se usan etiquetas, o *labels*, para marcar una instrucción de modo que pueda ser referenciada por una instrucción de control de flujo. Por último, también es posible copiar un valor mediante el uso de la asignación $\mathbf{x = y}$.

A diferencia del CIL y SIL, este lenguaje es basado en registros y no hace uso de una pila. Se utilizan variables simbólicas que luego se traducen a direcciones reales de memoria.

Analysis.NET provee una representación intermedia en TAC. A continuación podemos ver el código del método del ejemplo 1.3 en esta representación intermedia que provee el *framework*:

```
{
    L_0000: nop

    L_0001: $s0 = x
    L_0002: $s0 = $s0 as Single
    L_0003: $s1 = y
    L_0004: $s0 = $s0 + $s1
    L_0005: local_0 = $s0

    L_0006: $s0 = local_0
    L_0007: $s1 = 2
    L_000c: $s0 = $s0 / $s1
    L_000d: local_1 = $s0

    L_000e: $s0 = local_1
    L_000f: local_2 = $s0
    L_0010: goto L_0012

    L_0012: $s0 = local_2
    L_0013: return $s0
}
```

Listing 1.6: Cuerpo del método *Average* en TAC

Podemos observar que las instrucciones respetan el formato descrito anteriormente. Al igual que en CIL y SIL, contamos con etiquetas únicas en las instrucciones. Si bien están presentes en todas ellas, esto es más que nada un tema de prolijidad ya que son necesarias únicamente en aquellas instrucciones que luego son referenciadas por alguna intrucción de control de flujo. Las variables prefijadas con **$** son temporales que se generan para ir guardando los resultados intermedios y para almacenar las partes de las instrucciones de modo que se respete siempre la máxima cantidad de operandos. Estas generalmente se numeran de forma autoincremental. El resto de las variables que vemos son locales al método.

Esta representación es fundamental en el *framework* ya que es la que utilizan los análisis provistos por el mismo. Esto se debe a que es una representación estándar y que está abstraída del lenguaje y plataforma subyacentes por lo que los distintos análisis y algoritmos se suelen pensar sobre este tipo de representaciones.

La versión provista por la herramienta carece inicialmente de información de tipos ya

que las variables temporales representan *slots* en el *stack* y por ende pueden almacenar distintos tipos de valores. Sin embargo, es posible agregarle esta información mediante el uso de los análisis *Type Inference* y *Web Analysis*.

1.2.3. Análisis estático de código

Analysis.NET provee una serie de análisis estáticos de código clásicos ya incluidos dentro del *framework* y además cuenta con una interfaz siemple y extensible que permite el agregado de nuevos análisis. Mencionaremos algunos de ellos que fueron utilizados en este trabajo, ya sea para la implementación de la solución o bien para la generación de los experimentos o casos de prueba. El detalle de estos análisis se puede encontrar en el trabajo original del *framework* [9].

Class Hierarchy Analysis

Este análisis permite entender la jerarquía de clases, donde las nociones de herencia e interfaz conforman el concepto de relación. En .NET, al igual que muchos otros lenguajes orientados a objetos, todas las clases heredan directa o indirectamente de una única clase denominada **Object**. Esta herencia no puede ser múltiple, una clase hereda siempre de alguna otra, pero sí está permitido que una clase implemente más de una interfaz lo cual busca lograr un efecto similar.

Class Hierarchy Analysis construye un árbol donde cada clase tiene como hijos a todas las clases que heredan de ella. Análogamente de cada interfaz descienden todas aquellas clases que la implementan.

Call Graph Analysis

Este análisis permite describir como se relacionan los distintos métodos entre sí, es decir, las invocaciones entre métodos. Para esto se construye un grafo en el cual cada nodo representa un método y donde cada eje denota una invocación entre dos métodos. Un nodo que posee un eje a sí mismo indica un método recursivo.

Control Flow Analysis

Permite construir un grafo, denominado *Control Flow Graph* (CFG), que determina el flujo de ejecución del código mostrando los distintos caminos posibles. El grafo está compuesto por nodos denominados *basic blocks* que contienen la cantidad maximal de instrucciones consecutivas a las que se llega por un único punto de entrada, que cuentan con un único punto de salida y no contienen saltos internos. Los ejes se crean en base a los caminos de los *branches* y permiten conectar los distintos nodos. Los ciclos o iteraciones son representados por ciclos en el grafo. Este tipo de análisis es fundamental para cualquier otro que dependa del flujo de ejecución.

Analysis.NET soporta la creación del CFG para las distintas representaciones intermedias como pueden ser SIL o TAC.

A continuación veremos un ejemplo de como armar el CFG para un programa dado. El mismo, que llamaremos *MaxPlusOne*, calcula el máximo entre dos numeros y luego lo incrementa en uno:

```
var w = 0;
if (x > y)
{
    w = x;
}
else
{
    w = y;
}
w++;
```

Listing 1.7: Cuerpo del método *MaxPlusOne* en C#

```
L1: w = 0
L2: if y ≥ x goto L5
L3: w = x
L4: goto L6
L5: w = y
L6: w = w + 1
```

Listing 1.8: Cuerpo del método *MaxPlusOne* en TAC

En el fragmento de la derecha encontramos una versión simplificada del TAC del método *MaxPlusOne*. Podemos observar que la condición del salto está invertida con respecto a su versión en C#. Esto se debe a que, como detallamos anteriormente al introducir el TAC, el mismo continúa con la instrucción indicada en el *goto* si es verdadera mientras que si es falsa continúa con la instrucción siguiente al salto.

Para armar el CFG entonces procedemos a descomponer el programa en *basic blocks*. Estos bloques se arman en base a la sentencia que tiene el salto, la cual nos separa en un bloque que agrupa todas las instrucciones hasta el mismo, dos bloques que son los caminos a tomar según el resultado del salto y por último un bloque con las instrucciones restantes. Se agregan además dos nodos, *Entry* y *Exit*, para representar el único punto de entrada y de salida del programa. A continuación podemos ver como queda armado el grafo:

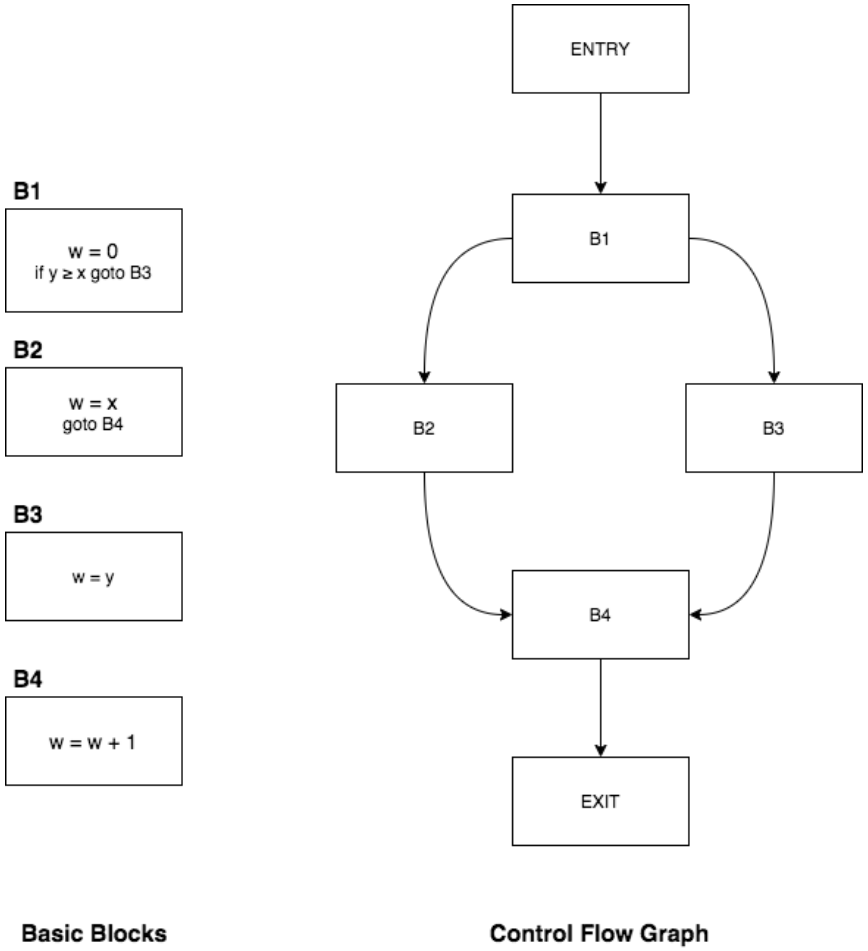

Fig. 1.2: CFG del método *MaxPlusOne*

Web Analysis

Este análisis permite junto con el *Type Inference*, tipar el cuerpo de un método en formato TAC, lo cual es necesario ya que la versión provista por el *framework* es por defecto no tipada. Como vimos anteriormente, el TAC es una representación basada en registros en vez de pila y utiliza variables temporales para ir almacenando los resultados intermedios. Estas variables al no ser tipadas, son naturalmente reutilizadas para almacenar distintos valores de distintos tipos de datos. *Web Analysis* nos permite separar estas variables en variables frescas, las cuales pueden ser tipadas sin ningún conflicto de tipos. Una variable puede convertirse entonces en una o más variables según la cantidad de reusos que la misma tenga. Estos reusos de cada variable forman el concepto de *webs*. Cada *web* se compone del conjunto de referencias (definiciones y usos) de variables minimal que contiene por cada definición todos sus usos y por cada uso sus definiciones. El análisis consiste en asignar una variable fresca por cada *web* y renombrar todas sus referencias.

El resultado de este análisis nos otorga un TAC con esta información de las *webs* lo cual funciona como entrada del *Type Inference*.

Type Inference Analysis

Tomando como entrada el resultado del análisis anterior, el objetivo es obtener un TAC tipado. Para esto se propagan los tipos del lado derecho de cada asignación a su variable destino. En caso de tener operaciones algebraicas o conversiones de tipos, las mismas deben considerarse al momento de inferir el tipo del resultado. No encontraremos conflictos de tipos ya que esto es garantizado por el *Web Analysis*.

Si bien en cada paso el análisis busca resolver el tipo de cada variable, existen algunos casos especiales para los cuales es necesario más información para determinar el tipo de la que tenemos disponible. El ejemplo más claro de esto es el caso de valores *nulls* ya que estos son válidos para distintos tipos de datos (valor polimórfico). En estos casos lo que se hace es retrasar la inferencia de tipo hasta algún momento posterior en el cual contemos con la información necesaria para determinarlo de forma correcta.

2. MOTIVACIÓN

Analysis.NET surge de la falta de herramientas y la dificultad de analizar programas .NET. Si bien existen herramientas como Roslyn [10], CCI [11], Cecil [12] e IlSpy [13], su foco no es el análisis estático de código y no ofrecen la posiblidad al usuario de implementar sus propios análisis. Algunas de ellas están deprecadas o cumplen primariamente otra función como es el caso de un compilador del código fuente (Roslyn) o bien un decompilador de *bytecode* (IlSpy). Estas herramientas además trabajan sobre el código fuente en lugar de ofrecer representaciones intermedias.

En otros lenguajes existen herramientas similares que ofrecen la funcionalidad provista por Analysis.NET. Si tomamos como ejemplo Java, tenemos Soot [14], que ofrece una herramienta para leer el *bytecode* de Java, múltiples representaciones intermedias, análisis estático de código y finalmente la posiblidad de impactar estas transformaciones generando un nuevo ejecutable. Si comparamos Analysis.NET con Soot, una de las limitaciones que encontrarmos es justamente este último punto. Es decir, los análisis y transformaciones que se hagan del código no pueden impactarse en un nuevo ejecutable por lo que el *framework* se podría decir que funciona en un modo solo lectura.

En el presente trabajo buscamos cubrir esa brecha agregando toda la parte de generación de código, lo cual potencia el poder de la herramienta haciendo posible el camino completo de lectura-analsis/transformación-generación de código.

2.1. Ejes de estudio

La habilidad de generar código ejecutable nos abre un abanico de posiblidades que antes no teníamos. Hablaremos de cuatro ejes que nos permiten evaluar el valor agregado de esta funcionalidad. Estos son Generación, Optimización, Instrumentación y Generación programática.

Generación

Trataremos la generación de código en sí. Partimos de un ejecutable y generamos uno nuevo semánticamente equivalente que no agrega ningún valor sobre el original. Este es el eje principal del trabajo y es el cimiento que hace posibles a todos los demás.

Instrumentación

La instrumentación de código es un agregado de funcionalidad al mismo que no altera la semántica del programa pero que lo enriquece de alguna manera. Tener el camino completo de lectura y generación nos permite, dado un programa, posicionarnos en alguna parte del

mismo instrumentando alguna porción de código y finalmente generar un nuevo ejecutable que tiene estas características.

Entre las instrumentaciones más conocidas podemos encontrar el *logging* y los *benchmarks*[1].

Optimización

Una optimización consiste en una transformación de código que aporta cierto valor y se considera una mejora, según algún aspecto, sobre el programa original. Analysis.NET ofrece una batería de análisis estáticos de código, además de la posiblidad de construir nuevos, los cuales observan distintas características del mismo. Sobre ellos podemos construir optimizaciones e impactarlas generando un nuevo archivo ejecutable que cuenta con esas mejoras.

Generación programática

Con generación programática nos referimos al proceso de generar un ejecutable desde cero utilizando Analysis.NET y sin necesidad de partir de un programa ya existente. Directamente utilizando el modelo de objetos provisto, podemos generar programas de forma totalmente programática.

En la sección de evaluación, veremos ejemplos de estos ejes para comprender el valor que agrega a la herramienta y se detallará cómo validamos los mismos. Si bien trataremos los cuatro ejes presentados, es importante destacar que el foco principal de este trabajo es la generación de código. La instrumentación y optimización son problemas interesantes y complejos por sí solos. Por este motivo, para los demás ejes que no son generación, lo que veremos serán ejemplos más simples para demostrar el potencial de la herramienta pero no profundizaremos en estas problemáticas.

[1] El *logging* consiste en al agregado de reportes que pueden ser para auditoría de alguna funcionalidad o bien como ayuda al programador para hacer *debugging*. De manera similar, *benchmarks* también consiste en el agregado de reportes pero que en este caso se enfocan en mediciones relacionadas a performance o algún aspecto del programa que luego es útil para evaluar el funcionamiento del mismo o bien para realizar comparaciones.

3. IMPLEMENTACIÓN

Habiendo presentado la plataforma sobre la que trabajaremos, el *framework* de análisis y los ejes de estudio en los que nos enfocaremos durante este trabajo, en esta sección entraremos en detalle en la extensión realizada, es decir, los módulos incorporados. Si bien el *framework* cuenta con muchos componentes y herramientas, si nos enfocamos en los aspectos de lectura, transformación y generación de código podemos simplificarlo en la interacción de cuatro componentes principales que son *Loader*, *Generator*, *Disassembler* y *Assembler*.

Loader

Encargado de la lectura del programa de entrada. Recibe un archivo ejecutable y construye un modelo de objetos que representa ese programa. Además es el responsable de transformar el cuerpo de los métodos de CIL a SIL.

Generator

Genera un nuevo archivo ejecutable a partir del modelo de objetos. Análogamente al *Loader*, es el encargado de traducir el código SIL a su contraparte en CIL.

Disassembler

Transforma el código SIL a su representación equivalente en TAC.

Assembler

Análogo al *Disassembler*, se encarga de transformar el código TAC de vuelta a SIL.

Los componentes *Generator* y *Assembler* son los que se agregan en este trabajo. Al igual que el *Loader*, el *Generator* se implementa de forma extensible, definiendo una interfaz *IGenerator* lo cual habilita el uso de distintas implementaciones para proveer dicho comportamiento. Actualmente el *Loader* cuenta con dos implementaciones, la original que utiliza CCI [11] y la última desarrollada que usa *System Reflection Metadata* (SRM) [15]. En el caso del *Generator* por el momento se incluye una única implementación utilizando SRM también.

Para el *Disassembler* y *Assembler*, actualmente hay una única implementación de los mismos y no están pensados para soportar más de una. Esto se debe a que, a diferencia del caso del *Loader* y *Generator*, en estos componentes ya no nos encontramos a bajo nivel y acoplados a .NET, sino que tratamos con representaciones intermedias provistas por el *framework*. Es por eso que a priori no parece necesario poder proveer más de una forma

de transformar SIL a TAC y viceversa. Sin embargo, llegado el caso de que surgiera esa necesidad, no sería complicado hacer un *refactor* que lo permita.

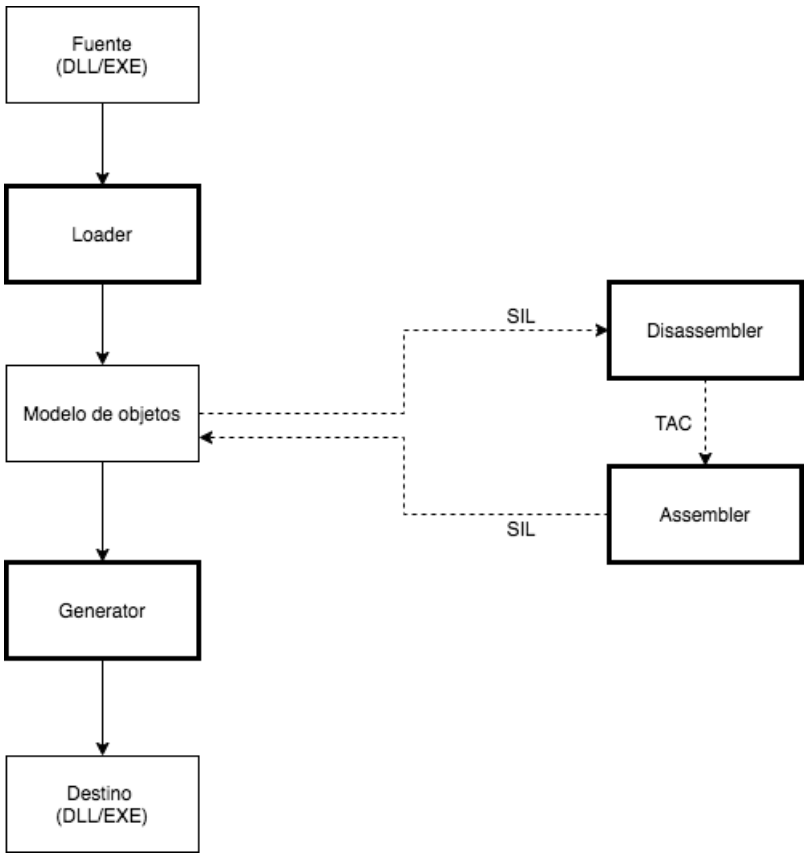

Fig. 3.1: Diagrama de componentes de Analysis.NET

3.1. Generación de código: Generator

El objetivo de la generación de código es pasar del modelo de objetos que provee el *framework* a un ejecutable de .NET. *Portable Executable* [16], o PE, es el formato binario de archivos ejecutables utilizado por este tipo de programas. Entre sus extensiones más utilizadas encontramos **.exe** para ejecutables y **.dll** para librerías. Como su nombre lo indica, es portable, lo que significa que esta diseñado para ser independiente de la arquitectura en la cual se ejecuta. A lo largo del presente trabajo hablaremos de ejecutable para referirnos a los archivos en formato PE, indistintamente si son librerías o programas ejecutables.

Los archivos PE son complejos ya que deben incluir no solo el código CIL del programa en cuestión, sino que además se requiere una gran cantidad de datos adicionales, llamados metadata, que relacionan las distintas partes del CIL y son utilizados por el CLR durante la ejecución. Por este motivo la generación del archivo ejecutable es una labor compleja ya que

es necesario respetar todas las reglas y lineamientos que define el formato para obtener un ejecutable correcto. Optamos entonces por utilizar la librería *System Reflection Metadata* (SRM) que provee una interfaz de un poco más alto nivel en comparación con tener que generar directamente el binario y se encarga de manejar los detalles más específicos del formato del ejecutable. La misma ya estaba siendo utilizada en Analysis.NET para lo que respecta a la lectura de estos archivos. SRM es desarrollada y mantenida por Microsoft, forma parte del *Core* de .NET y es utilizada actualmente en Roslyn, el compilador de C#. El mismo debe ser eficiente por lo que esta librería parte con este requerimiento desde el diseño y es por eso que notamos muchos aspectos de muy bajo nivel en la misma[1]. Esto la convierte en una librería compleja de utilizar que, sumado a la prácticamente nula documentación, establece una pendiente bastante grande en la curva de aprendizaje. No obstante, consideramos que vale la pena el esfuerzo por los beneficios que provee.

Metadata

Si bien el archivo ejecutable contiene mucha información, en este trabajo nos enfocaremos en dos de los aspectos más importantes que son el CIL y la Metadata [17]. La misma es toda la información necesaria para definir y referenciar clases, instanciarlas, resolver invocaciones de métodos y ayudar en la traducción del código CIL al código máquina. En un modo simplificado podemos decir que CIL es el conjunto de instrucciones que se usa en los cuerpos de los métodos, mientras que la metadata define toda la estructura de clases, métodos y demás aspectos del programa en cojunto con los mecanismos necesarios para referenciarlos.

La metadata se organiza en dos estructuras: *heaps* y tablas. Los *heaps* son utilizados para almecanar los *strings*, identificadores únicos y algunos datos con el fin de optimizar el espacio de almacenamiento. Las tablas en cambio son las encargadas de almacenar la información que define y relaciona las distintas entidades del ejecutable. Cada entrada en una tabla puede ser una constante o bien un índice que referencie una entrada en la misma, otra tabla o bien en un *heap*. Estos índices pueden ser simples, referencian entradas en una única tabla, o bien codificados lo que implica que pueden referenciar entradas de múltiples tablas. Las tablas cuentan con un nombre y un código hexadecimal que las identifica, por ejemplo **Assembly 0x20**. Estas tablas poseen una clave primaria, algunas de ellas requieren estar ordenadas e incluso no aceptar entradas duplicadas.

Si bien cada tabla guarda información particular al dominio de la misma, en esencia podemos clasificar sus entradas en tres categorías:

- **Definiciones:** Definen un ítem de metadata. Incluyen toda la información que caracteristiza al ítem en cuestión.

[1] Por ejemplo la librería está diseñada para evitar la alocación de objetos en el *heap* en favor del *stack* para mayor performance.

- **Referencias:** Permiten referenciar un ítem de metadata con la información mínima indispensable. Las referencias son necesarias para poder hablar de ítems que se encuentran fuera del programa que se esta generando como pueden ser librerías externas que se utilizan. Sin embargo también son útiles para referenciar ítems definidos dentro del mismo programa, por ejemplo por no tener disponible la definición acorde.

- **Firmas:** Representan características de un ítem de metadata. Sirven para especificar firmas de métodos, tipos de *fields*, firmas de variables locales de un método, entre otros. Además son utilizadas en lo que es *generics* para referirse a instanciaciones de un tipo o método genérico.

Hablaremos de ítems para hacer referencia a tipos, métodos, *fields* y demás componentes del código. Algunos ejemplos de estas tablas son:

- **TypeDef (0x02):** Almacena una definición de tipo. Esto incluye datos como su nombre, *namespace* que lo contiene y aquellos métodos y *fields* que define.

- **TypeRef (0x01):** Guarda las referencias a tipos. A diferencia de las definiciones, almacenan solo la información necesaria para poder determinar el tipo referido. Esencialmente nombre, *namespace* que lo contiene y *assembly* o tipo que lo define.

- **TypeSpec (0x1B):** Cada entrada es una firma que representa una instanciación de un tipo genérico.

- **MethodDef (0x06):** Definiciones de métodos. Almacena, entre otros datos, el nombre del método y firma del mismo, la cual incluye principalmente los tipos de los parámetros y el tipo de retorno. El cuerpo del método no esta contenido en la definición, sino que se almacena de forma separada, pero es referenciado por la misma.

- **MethodSpec (0x2B):** Análogo a TypeSpec, almacena las instanciaciones de métodos genéricos.

- **Field (0x04):** Definiciones de *fields*. Incluye, entre otros, el nombre y una referencia al tipo que lo contiene.

- **MemberRef (0x0A):** Representa referencias tanto para *fields* como para métodos.

Estos son algunos ejemplos nomás pero cabe destacar que la metadata se compone de más de 30 tablas. Las mismas almacenan cierta información pero es luego la referencia entre ellas que forma el modelo completo del programa.

3.1.1. Estrategia

Se evaluaron dos enfoques para la generación de código que consisten en hacer la generación en una única pasada o llevarlo a cabo en múltiples pasadas.

En el caso de múltiples pasadas, se opera principalmente con definiciones y se usan referencias solamente cuando es necesario, es decir, cuando las definiciones no estan incluidas en el mismo programa. El orden en el que se encuentran los tipos, métodos y demás en el modelo de objetos, o incluso en el CIL, nos obliga a recorrer varias veces con el objetivo de ir consiguiendo las definiciones de todo lo que precisamos. A modo de ejemplo, ocurre que al procesar un tipo que tiene un método, el mismo puede llamar a otro método o referenciar un tipo que aún no fue procesado y por ende no tenemos aún su definición. Sin ir más lejos, esto se puede dar con los mismos métodos de un tipo ya que puede haber un método que es llamado por otro que aún no se llegó a procesar.

El enfoque de única pasada si bien a priori puede sonar más complejo, en la práctica resulta más sencillo e intuitivo. Esto se debe a que a pesar de que las definiciones y referencias se almacenan en tablas distintas (y son elementos distintos de metadata), éstas son por la mayor parte intercambiables en su uso. Es decir, las entradas de las tablas que tienen alguna columna que es un índice a otra tabla, en su mayoría son índices a definiciones o referencias, es decir, índices codificados. Un ejemplo de esto podría ser una instrucción que invoca un método, la instrucción acepta tanto una definición como una referencia del mismo. O bien en la definición de un tipo, para especificar su tipo base se puede usar tanto una definición como una referencia. Como mencionabamos antes, esto es cierto para la mayoría de las tablas, siendo muy pocos los casos donde se necesita una definición o bien una referencia de forma excluyente. La estrategia de única pasada aprovecha esta característica y consiste entonces en ir recorriendo los distintos ítems del modelo de objetos, ir generando las definiciones e ir creando referencias para todo aquello que aún no se haya procesado.

Almacenamiento de referencias y firmas

En la forma más simple de la estrategia de única pasada, agregaríamos potencialmente muchas referencias y firmas de un mismo ítem de metadata además de su definición. Esto se debe a que un ítem puede ser referenciado en distintos contextos, por ejemplo un método que se usa en múltiples instrucciones de tipo *call*. Esto se traduce en un aumento del tamaño del archivo generado, ya que se incrementa el tamaño de las tablas de la sección de metadata. Sumado a que algunas tablas tienen la restricción de no permitir duplicados, resulta más que lógico ir almacenando estas referencias y firmas de modo de poder reutilizarlas. Cabe aclarar que en la estrategia de múltiples pasadas también necesitamos almacenar estos ítems para aquellas tablas que no admiten duplicados. No obstante, el tamaño del archivo generado por la estrategia de única pasada siempre es

mayor ya que potencialmente tenemos una definición, una referencia y una firma para cada ítem.

Tablas ordenadas

Del mismo modo que algunas tablas no admiten duplicados, algunas de ellas imponen un orden en sus elementos. El mismo se da en base a su clave primaria. Esto presenta un problema, en ambas estrategias, ya que no tenemos la posiblidad de ordenar esas tablas sino que tenemos que asegurarnos que la inserción respete el orden impuesto. Esto condicionaría el orden en el que se deben procesar los distintos ítems para ir agregando sus elementos correspondientes de metadata. Como solución a este problema se optó por un agregado lógico de los mismos. La implementación entonces registra las entradas que debería agregar en estas tablas, que necesitan estar ordenadas, al momento de procesar ese ítem pero recién se agregan a la tabla al final de la generación lo cual permite ordenar las entradas previamente.

Podemos resumir ambas estrategias en la siguiente tabla comparativa:

Aspecto	Única pasada	Múltiples pasadas
Algoritmo	más simple	más complejo
Tamaño del ejecutable	mayor	menor
Almacenamiento de referencias y firmas	necesario	necesario
Lógica de tablas ordenadas	necesario	necesario

Tab. 3.1: Comparativa de estrategias de generación

Ambos enfoques son válidos y tienen sus pros y contras. Para la implemetación se optó por el enfoque de única pasada.

3.1.2. Componentes

Podemos identificar 3 componentes principales en el *Generator*:

- **Generators:** Asociados a la generación de algún aspecto en particular, son los encargados de procesar y agregar las definiciones. Podemos encontrar un *generator* para los tipos, métodos, *fields*, *properties*, interfaces y demás ítems.
- **Encoders:** Encargados del armado de las firmas que se precisan para definiciones y referencias. Al igual que los *generators*, encontraremos uno para los distintos ítems.
- **Resolver:** Módulo encargado de proveer firmas y referencias a demanda. Los distintos componentes interactúan con él para obtener las firmas o referencias que precisan. Es el responsable de administrar la base de referencias y firmas mencionada anteriormente.

3.1.3. SIL a CIL

Esta traducción consiste en llevar los cuerpos de los métodos que se encuentran en una representación intermedia del *framework*, SIL, a su equivalente en CIL. Adicionalmente debemos tener en cuenta el flujo de ejecución del código, generar la información de manejo de excepciones y calcular el *max stack* (profundidad máxima del *stack*).

Traducción de instrucciones

Como se mencionó al introducir SIL, el mismo cuenta con un conjunto de instrucciones muy reducido si lo comparamos con su contraparte en CIL. Esta etapa consiste en recorrer cada instrucción del SIL y generar una instrucción equivalente en CIL. Es el proceso inverso al que realiza el *Loader* al leer el ejecutable y armar el modelo de objetos. Para esta etapa nos basamos en el set de instrucciones del ECMA [18], el cual detalla todas las instrucciones del CIL y sus particularidades.

Por la mayor parte es una tarea principalmente tediosa y en la cual hay que ser riguroso ya que una instrucción mal generada puede resultar en un CIL que no se comporte como debería. Al haber tantas instrucciones, algunas que son variantes similares o incluso intercambiables, la tarea se torna un poco propensa a errores.

Otra complejidad que encontramos es el hecho de que en este punto se efectúa el pasaje de un modelo a otro por lo que entra en juego el uso de definiciones, referencias y firmas que las instrucciones puedan necesitar. Una instrucción en SIL puede estar relacionada con algún otro ítem del modelo, pero esto se maneja todo a nivel objetos. Por ejemplo en el caso de una instrucción que llama a algún método, tenemos que la misma contiene un atributo que es el método al que llama, un objeto. Mientras que en CIL, necesitamos una definición, referencia o firma, es decir un puntero a una entrada en las tablas de metadata, lo cual significa que comenzamos a tratar con entidades de otro modelo.

Control de flujo

Cuando hablamos de control de flujo, hacemos referencia a aquellas construcciones o lógicas que pueden alterar el orden de ejecución de las instrucciones. En particular hablaremos del manejo de excepciones y de los *branches* o saltos.

En el primer caso, CIL cuenta con cláusulas e instrucciones para el manejo de excepciones que son similares a las que encontramos en los lenguajes de alto nivel. Se definen regiones que se componen de:

- **Bloques protegidos:** El fragmento de código en el cual se espera que pueda ocurrir alguna excepción.
- **Handlers:** Bloque de código que contiene la lógica de manejo de la excepción.
- **Filtros:** Deciden si un *handler* maneja o no una excepción.

Además, CIL cuenta con algunas instrucciones específicas para lanzar excepciones.

Los saltos en cambio son instrucciones que permiten, de forma condicional o incondicional, continuar la ejecución desde una instrucción determinada. En los lenguajes de alto nivel los encontramos generalmente en forma de *if-else* o *switch* y ciclos como *while*, *for* y *foreach*.

Ambos casos tienen una similitud y es que consisten en poder referenciar una instrucción, donde continuar en el caso de los saltos y donde arrancar o terminar una región para las excepciones. Esto nos introduce una problemática ya que no podemos conocer la ubicación de las instrucciones que aún no procesamos. Cuando introdujimos CIL, mencionamos el uso de *offsets*. Estos permiten marcar, de forma relativa, la ubicación de cada instrucción. La librería de SRM utilizada se encarga de generar y gestionar estos *offsets* ya que dependen del tamaño que ocupa cada instrucción en el ejecutable. El problema es que partimos del SIL, el cual esta desacoplado del CIL y por ende no conoce el tamaño que ocupará cada instrucción al ser traducida. Estas referencias de instrucciones que mencionábamos para los saltos y manejo de excepciones son a nivel de *offsets*. Lo que significa que al momento de traducir una de ellas, no sabemos aún en que *offset* se va a posicionar la instrucción que queremos referenciar. SRM esta preparada para esta problemática y lo que provee es un sistema de etiquetas virtuales. Básicamente permite generar instrucciones de CIL para los saltos y manejo de excepciones que referencian una etiqueta aún no creada. El proceso entonces consiste en ir haciendo este procedimiento a medida que se van procesando estas intrucciones y a su vez ir creando esas etiquetas cuando hacemos la traducción de la instrucción a la cual le corresponde. Esto permite obtener la ubicación real de la instrucción referenciada.

Max Stack

Hasta el momento la traducción de las instrucciones la realizamos de forma secuencial recorriendo las mismas en el orden en el que se encuentran en el método en SIL. Esto tiene que ser así ya que su contraparte en CIL debe respetar el mismo orden, sino estaríamos alterando la semántica del método original. El primer enfoque entonces fue recorrer las instrucciones una por una y traducir. Si bien esto nos genera un CIL correcto en cuanto a las instrucciones, encontramos un problema a la hora de calcular el *max stack*.

El *max stack* es la máxima cantidad de elementos que puede haber en la pila en todo momento en el cuerpo de un método (recordemos que CIL es basado en *stack*). Si bien es información que se puede obtener al leer el ejecutable, es un detalle muy específico del mismo que no es posible ir manteniendo a lo largo de las transformaciones de código ya que está relacionado directamente con las operaciones del CIL y recordemos que una vez convertido en SIL, este puede luego pasarse a TAC (basado en registros) o sufrir distintas modificaciones. Por este motivo no se espera un *max stack* calculado de antemano, sino que lo calculamos durante la traducción de las instrucciones.

El ECMA especifica las transformaciones que ocurren en el *stack* para cada instrucción de CIL. Por lo que el procedimiento consiste en ir aumentando o disminuyendo la cantidad de elementos que puede haber en la pila en base a las instrucciones que se van procesando, es decir si éstas operan o no con el *stack*, ya sea para dejar un resultado o leer una entrada por ejemplo.

La primera estrategia fue simplemente ir sumando y restando cuantos elementos había en el *stack* a medida que recorríamos y traducíamos las instrucciones. Si bien esto nos generaba por la mayor parte resultados correctos (nunca menor a lo necesario), en muchos casos calculaba un *max stack* mayor al que realmente se necesitaba. Esto se debe a que este cálculo no es tan lineal sino que tiene que tener en cuenta el control de flujo del código. El caso más evidente de esto son los saltos. En la ejecución se va a tomar un camino u otro pero nunca ambos. O mismo si tenemos saltos incondicionales, podríamos encontrarnos con partes que incluso nunca se ejecutan. Por lo que el enfoque original tiene el problema de que recorre todos los caminos y uno detrás del otro.

La solución para este problema es hacer un análisis más inteligente de cómo se compone el código. Analysis.NET cuenta ya con un análisis de control de flujo el cual nos permite armar el *Control Flow Graph* (CFG). El mismo agrupa las instrucciones en bloques y los conecta según el flujo de ejecución.

Cambiamos entonces el primer enfoque a uno que hiciera uso del CFG para recorrer las instrucciones. Sin embargo más allá de construirlo, debemos ordenar los nodos tal que respeten el orden en el cual se encuentran las instrucciones en el código SIL de modo de recorrerlas y traducirlas en el orden original. Para lograr esto, tomamos la primera instrucción de cada nodo (bloque de n instrucciones) como representante de ese nodo y ordenamos los mismos en base a la ubicación, el índice, de esa instrucción en el método original. Esto nos garantiza el orden que precisamos.

Se recorren entonces las instrucciones, en base a los nodos, y para cada una además de traducirla a su contraparte en CIL calculamos cómo modifica el *stack*. Adicionalmente tenemos que ir manteniendo el estado del *stack* en cada uno de los nodos. Esto nos permite restaurarlo ante cada salto que implique más de un posible camino. De esta forma recorremos todos los caminos posibles pero con el *stack* como debería estar antes de entrar a cada uno. Esto es lo que no ocurría en la primera solución, ya que ante una bifurcación pasabamos por ambos caminos, uno tras otro, llegando al segundo con el estado del *stack* como resultado de haber pasado por el primero.

3.1.4. Precondiciones y Postcondiciones

Precondiciones

Como entrada del *Generator* precisamos un modelo de objetos del *framework* donde todos los métodos tienen sus instrucciones en SIL.

Postcondiciones

Como salida obtenemos un archivo ejecutable donde el código CIL es semánticamente equivalente al SIL de entrada. Sin embargo este CIL no es necesariamente óptimo en terminos de tamaño del mismo o instrucciones utilizadas.

3.2. Traducción de TAC a SIL: Assembler

En este caso encontramos un problema de un poco menor escala que el de generación ya que este módulo se encarga únicamente de la traducción de los cuerpos de los métodos, a diferencia del *generator* que se ocupa además de toda la generación de la metadata y requisitos necesarios del ejecutable. Más aún, en este segmento todo ocurre dentro del *framework* y con representaciones intermedias del mismo por lo que no es necesario el uso de ninguna librería ni de lidiar con detalles específicos de .NET. Este módulo sería la contraparte del *Disassembler*.

3.2.1. TAC a SIL

Al igual que en la generación de código (SIL a CIL), la traducción consiste en recorrer las distintas instrucciones del TAC y llevarlas a su equivalente en SIL.

Traducción de instrucciones

En el caso del TAC nos encontramos con una representación basada en registros en lugar de *stack*. Esto facilita un poco el proceso ya que en la traducción inversa, en el *Disassembler*, el pasaje de *stack* a registros es más complejo debido a que hay que ir simulando la pila de forma de ir generando las distintas variables a utilizar por el TAC. En este caso en cambio la traducción es más directa y en la mayoría de los casos consiste en traducir una a una las instrucciones. La razón de esto es que la implementación del TAC del *framework* hace uso de variables temporales entre cada operación, lo cual produce un código si se quiere más similar al SIL que queremos llegar. A continuación mostramos un fragmento de código muy sencillo en C# y como queda el TAC generado por el *framework* y luego el SIL generado por el módulo que describimos en esta sección:

```
var x = 1;
var y = 2;
var z = x + y;
```

Listing 3.1: Fragmento en C#

```
L_0001:  $s0 = 1
L_0002:  local_0 = $s0
L_0003:  $s0 = 2
L_0004:  local_1 = $s0
L_0005:  $s0 = local_0
L_0006:  $s1 = local_1
L_0007:  $s0 = $s0 + $s1
L_0008:  local_2 = $s0
```

Listing 3.2: Fragmento en TAC

```
L_0001:  load Value of 1
L_0002:  store local_0
L_0003:  load Value of 2
L_0004:  store local_1
L_0005:  load Content of local_0
L_0006:  load Content of local_1
L_0007:  Add
L_0008:  store local_2
```

Listing 3.3: Fragmento en SIL

Como podemos observar, esta implementación del TAC nos simplifica un poco la traducción ya que tenemos una correspondencia uno a uno de las instrucciones. Si en cambio no se utilizaran variables temporales, cada sentencia de asignación del código C# sería una única sentencia en TAC que luego se desdoblaría en dos instrucciones en SIL, un *load* y un *store*. No obstante, al igual que de SIL a TAC, el mapeo de instrucciones no es uno a uno en todos los casos por lo que hay que lidiar con algunas instrucciones que se transforman en más de una o bien en varias que se condensan en una sola. Además, hay que tratar cada caso de variable en particular, sea una local o temporal generada para respetar el formato de 3 operandos.

Manejo de excepciones

En el pasaje de CIL a SIL, el manejo de excepciones es más directo ya que en ambos es información adicional a las instrucciones. Es decir, no contamos con instrucciones que indican un *Try* o un *Catch*, sino que tenemos un apartado que indica donde comienzan y el alcance de estas regiones de excepciones. En el TAC pasa lo contrario ya que todo está plasmado como instrucciones. Es por eso que durante la traducción es necesario ir recopilando toda la información relacionada al manejo de excepciones e ir consolidándola en el formato que SIL necesita.

El apartado de manejo de excepciones en SIL consiste de una lista de bloques donde cada uno esta formado por un *try* y uno o más *handlers*. Ambos cuentan con *labels* que

indican el inicio y fin de esa cláusula. Un *try* puede tener múltiples *handlers* y esto se modela como múltiples bloques donde el *try* es el mismo (tiene los mismos *labels*) pero lo que difiere son los *handlers*. Además estos bloques pueden estar anidados, es decir, que podemos tener un *try* dentro de otro.

El proceso entonces consiste en, durante la traducción de instrucciones de TAC a SIL, ir armando estos bloques a medida que nos topamos con instrucciones de manejo de excepciones. Utilizamos una pila para ir guardando los bloques que estan en proceso y los sacamos cuando estan completos. Definimos que un bloque está completo cuando tiene su *try* y se agregaron sus *handlers*, todos con sus respectivos *labels* de inicio y fin.

Algorithm 1 Traducción de información de manejo de excepciones

Entrada instruccionesTac: instrucciones del método en formato TAC.
Salida resultado: regiones de manejo de excepciones en formato SIL.

```
1:  resultado ← Lista()
2:  bloques ← Pila()                                          ▷ bloques en proceso
3:  for instrucción en instruccionesTac do
4:      if instrucción es try then
5:          if instrucción anterior es try then               ▷ mismo try, distinto handler
6:              incrementar la cantidad esperada de handlers de bloques.tope()
7:          else
8:              bloques.apilar(Bloque())                      ▷ inicializar nuevo bloque
9:      else if instrucción es handler then
10:         agregar handler a bloques.tope()
11:     else if instrucción puede salir de un bloque then
12:         if bloques.tope() está completa then
13:             resultado ← bloques.desapilar()
    return resultado
```

En el pseudocódigo anterior podemos ver una versión simplificada del proceso de armado de la información de manejo de excepciones. Los *handlers* son las instrucciones de tipo **catch**, **finally**, **filter** y **fault**. Las instrucciones que pueden salir de un bloque son por ejemplo **throw**, **leave** y **endfinally**.

3.2.2. Precondiciones y Postcondiciones

Precondiciones

El *Assembler* toma como entrada un TAC que cumpla con tener etiquetas únicas. Esto se refiere a que no haya dos instrucciones con un mismo *label*, ya que recordemos que esto se utiliza para todo lo que es control de flujo. El TAC que recibe el *Assembler* no es necesariamente aquel que solo fue traducido del SIL original, sino que puede haber sido sometido a distintos análisis y pasado por diversas transformaciones. En estos casos es necesario garantizar la unicidad de los *labels* y más aún, la consistencia. Por ejemplo si se eliminan instrucciones, es necesario cambiar aquellas que referenciaban esos *labels*.

Por otro lado, el TAC debe ser tipado. Esto es necesario ya que el SIL es tipado por lo que necesitamos esa información para poder hacer la traducción. Analysis.NET provee una versión de TAC no tipado, pero como vimos anteriormente es posible adicionarlo utilizando los análisis *Type Inference* y *Web Analysis*.

Por último, el TAC debe tener el mismo formato que el generado por el *Disassembler*. Es decir, debe hacer uso de variables temporales entre las operaciones tal como vimos en secciones anteriores.

Postcondiciones

El *framework* maneja dos identificadores de instrucciones que son *labels* y *offsets*. El primero, como hemos mencionado, consiste de etiquetas que identifican la instrucción y permiten referenciarla desde otra. Estos *labels* si bien suelen ser autoincrementales, no tienen por qué serlo. Lo único que necesitamos es que sean únicos, pero tranquilamente podrían ser aleatorios. Los *offsets* en cambio son las posiciones de las instrucciones en CIL. Estos sí son autoincrementales y la distancia entre dos representa el tamaño que ocupa la instrucción. En particular en CIL, los *labels* y los *offsets* son lo mismo. Estos *offsets* los levanta el *framework* al leer el CIL y pueden ser útiles para algunos análisis a nivel SIL. Pero una vez que se traduce de SIL a TAC, ya no hay garantías sobre esos *offsets*. Esto se debe a que, a este nivel, no es posible respetar los *offsets* ya que justamente estamos en una capa que ya es independiente del CIL, por lo que no podríamos saber los tamaños de las instrucciones. Además, sobre ese TAC se pueden ir agregando, modificando o eliminando instrucciones e incluso en la misma traducción podría haber algún cambio.

La salida del *Assembler* entonces es un SIL semánticamente equivalente al TAC de entrada y garantiza *labels* únicos y *offsets* virtuales. Decimos virtuales en lugar de nuevos ya que en sí reutiliza los que vienen como información del TAC que a su vez vinieron de la traducción desde el SIL. Esto significa que si solo se hizo la traducción de SIL a TAC y ninguna modificación, podría ser que estos offsets todavía tengan sentido. No obstante, el *Assembler* no da garantías sobre esto.

4. EVALUACIÓN

En esta sección pondremos a prueba los módulos incorporados con el objetivo de constatar su valor agregado y observar las características del resultado obtenido. Para esto divideremos la verificación de los ejecutables generados en tres etapas que son validación del formato del ejecutable, validación de la preservación de semántica y por último inspección manual.

Formato

Como mencionamos anteriormente, el archivo ejecutable es bastante complejo ya que contiene mucha información y muchas reglas de formato y estructura que se deben cumplir. Por este motivo, validar el ejecutable generado no es una tarea para nada sencilla. Microsoft provee una herramienta que se encarga de esto mismo llamada PEVerify [19]. Su función es validar si el ejecutable cumple con todas las reglas especificadas en el ECMA, lo cual resulta una herramienta muy útil para todo desarrollo que implique generación de código CIL.

Utilizaremos entonces esta herramienta para validar el formato del ejecutable generado.

Semántica

El ejecutable generado debe preservar la semántica del programa original. Esto es intuitivo en los casos de lectura y generación o bien en los de lectura, optimización y generación. En el caso de instrumentación también se debe respetar esto ya que esta acción agrega algún valor pero no debe alterar la semántica original. Dejaremos de lado este punto para lo que es generación programática de código ya que en ese caso no se lee un ejecutable, sino que se genera todo desde cero por lo que no hay semántica que preservar.

A diferencia de la validación de formato, no existe una herramienta que pueda verificar la preservación de la semántica o bien la correctiud del programa generado. Para este caso nos conformaremos con el uso de *testing* automatizado como herramienta que, teniendo una buena cobertura de tests, nos dará una aproximación de la preservación de la semántica.

En .NET, al igual que en otros lenguajes y plataformas, existen diversas herramientas para soportar *testing* automatizado. Muchas de ellas estan integradas dentro de los entornos de desarrollo lo cual permite ejecutar los tests de forma cómoda y conveniente. Sin embargo en nuestro caso esto no nos sirve ya que lo que precisamos es poder correr los tests de un ejecutable que generamos y del cual no poseemos el código fuente. Por este motivo decidimos utilizar Nunit [20] que cuenta con una herramienta que nos permite hacer lo anteriomente mencionado directamente desde una línea de comandos.

Inspección manual

Esta estapa consiste en observar características del resultado obtenido como pueden ser tamaño o calidad del ejecutable. Adicionalmente, para los ejes de instrumentación y optimización, debemos verificar la presencia de la mejora o valor agregado.

Es importante destacar que los dos primeros aspectos son complementarios. Si bien el formato nos asegura un programa válido, nada garantiza que el mismo preserve la semántica. Un ejemplo de esto sería una instrucción mal generada como podría ser el hecho de intercambiar sumas por restas. De la misma manera no es suficiente contar solo con el chequeo semántico. Si bien esto nos garantiza un ejecutable válido en el sentido de que se ejecuta correctamente y pasa los tests, podría haber algún aspecto interno que difiera del original o algo que no surja en los tests pero que podría ser un problema a futuro. En ambos casos estaríamos perdiendo calidad si comparamos la generación contra el original.

4.1. Casos de prueba

Habiendo definido las formas en las que vamos a evaluar la generación de código, precisamos de una serie de casos de estudio que nos permitan poner a prueba los ejes anteriormente descritos. Para esto vamos a experimentar tanto con ejemplos construidos para ejercitar específicamente ciertos aspectos de la generación como con distintas librerías de terceros ya existentes para .NET.

Si bien la cantidad de librerías que podemos encontrar es muy grande, se tuvieron en cuenta los siguientes aspectos para elegirlas:

- **Complejidad:** Buscamos librerías que no sean casos muy simples y que presenten cierta complejidad para considerarlas representativas. Consideramos que la librería es compleja sí
  - tiene muchas clases, métodos, *fields* y demás estructuras del lenguaje.
  - utiliza funcionalidades no triviales como puede ser el uso de *generics* y diversas construcciones que provee el lenguaje.
  - resuelve un problema interesante y no trivial.
- **Testing:** Cuenta con una batería de tests que garantiza una buena cobertura del código.
- **Popularidad:** Es utilizada en la comunidad ya sea industria o academia por diversos desarrolladores. En este punto nos guiamos por algunos indicadores que nos ofrece GitHub [21] como son:

- Issues: Errores, *bugs* o mejoras reportadas.
- Pull Requests: Pedidos de incorporación de nuevas funcionalidades, *refactors* o solución de *issues*.
- Watches: Usuarios que siguen atentamente los cambios del repositorio. Los mismos son notificados cuando hay alguna interacción como pueden ser *commits*, incorporación de *pull requests*, solución de *issues*, entre otros.
- Forks: Cantidad de usuarios que crean una copia del repositorio. Esto permite hacer *pull requests* al original o bien experimentar con cambios propios.
- Stars: Usuarios que tienen destacado este repositorio.

Durante el desarrollo y experimentación se trabajó con las siguientes librerías disponibles públicamente en GitHub:

- **Examples:** DLL con ejemplos chicos que muestran distintas funcionalidades del lenguaje. Se generó para los inicios de la implementación del módulo de generación ya que provee casos más acotados que los de una librería y por ende un ambiente más controlado para ir haciendo pruebas. Actualmente su valor ya no es significativo en comparación a las demás librerías utilizadas. Se incluye como parte de la solución[1].
- **Optional:** Provee una implementación robusta de una abstracción del paradigma funcional que es *Option/Maybe*[2]. La misma permite tratar con valores que pueden ser *null*, de una forma más polimórfica lo cual permite construir soluciones más robustas y elegantes y sobretodo menos propensas a errores como el famoso *Null Pointer Exception*.
- **TinyCsvParser:** Permite leer y parsear archivos en formato CSV a un modelo fuertemente tipado de objetos[3]. Es altamente configurable de modo de proveer alta flexibilidad pero a su vez ofrece una interfaz limpia y alta performance en su ejecución.
- **Fleck:** Implementación de un servidor de *Web Sockets*[4].
- **DSA:** Provee implementaciones de una gran cantidad de estructuras de datos como arboles, colas, grafos y una serie de algoritmos sobre ellas. De muy simple a más complejo, podemos encontrar las estructuras y algoritmos más conocidos y utlizados[5].

[1] https://github.com/fcurdi/analysis-net/blob/code-generator/Examples/Examples.cs.
[2] https://github.com/nlkl/Optional.
[3] https://github.com/bytefish/TinyCsvParser.
[4] https://github.com/statianzo/Fleck.
[5] https://github.com/abdonkov/DSA.

Como mencionamos anteriormente, utilizaremos Nunit para correr los tests de los ejecutables generados. Como no todas las librerías elegidas utilizaban este *framework* de *testing*, se realizaron *forks*[1] de las mismas y se adaptaron para utilizar esta herramienta. El cambio es muy sencillo y consiste en cambiar anotaciones de los tests a su equivalente en Nunit y lo mismo para algunas aserciones.

4.2. Resultados

En esta sección evaluaremos los resultados obtenidos para cada eje de estudio. Podemos describir el *pipeline* general de experimentación con el siguiente gráfico:

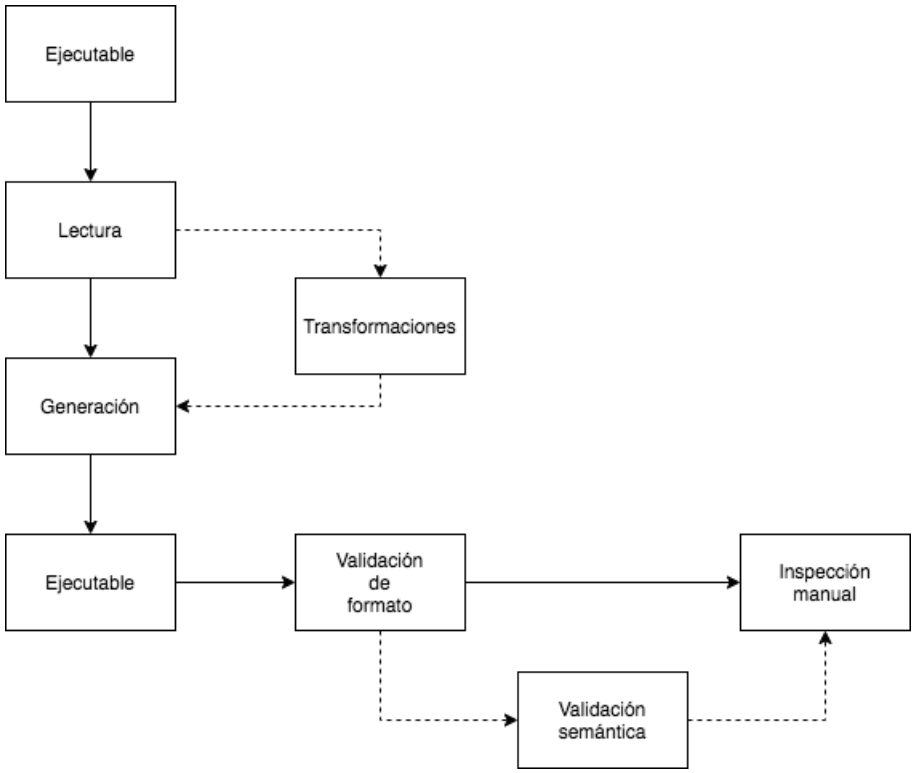

Fig. 4.1: Pipeline general del proceso de experimentación

Cada eje sin embargo tiene sus particularidades tanto en el proceso de armado del experimento como en la etapa de validación. Por ejemplo las transformaciones no ocurren en todos los ejes y algunos de ellos no cuentan con validación semántica, lo cual denotamos con lineas punteadas.

Presentaremos la experimentación para cada eje bajo el siguiente formato:

1. **Armado del experimento:** Detallaremos en qué consiste el mismo y qué casos de prueba se utilizaron.

[1] *Forks* disponibles en https://github.com/fcurdi.

2. **Validación semántica y de formato:** Explicaremos qué validaciones se hicieron.
3. **Inspección manual:** Expondremos las observaciones realizadas sobre los resultados.

4.2.1. Generación

Como ya hemos mencionado anteriormente, el eje de generación es el foco principal del trabajo. La validación del mismo es crucial ya que los demás ejes construyen sobre él. Pondremos a prueba los dos módulos principales construidos: *Generator* y *Assembler*. Esto implica que además de probar la generación en sí, también usaremos este eje para validar las transformaciones de TAC a SIL. Esto se debe a que no tenemos herramientas para probar la equivalencia semántica entre un modelo en SIL y un modelo en TAC. Por este motivo usaremos la misma generación, con sus validaciones, para confirmar que esta traducción también es correcta.

4.2.1.1. Armado del experimento

Los pasos a seguir por el expermiento son:

1. Se lee el ejecutable de entrada y se arma el modelo de objetos.
    - Opcionalmente se buscan todos aquellos métodos que tengan cuerpo (al menos una instrucción) y se los transforma de SIL a TAC y luego de TAC a SIL. Esta última transformación luego se tipa usando *Type Inference* y *Web Analysis*.
2. Se genera el nuevo ejecutable a partir del modelo de objetos.

Como casos de prueba se utilizan todas las librerías mencionadas. Cada una de ellas cuenta con una (o más) **dll** principal y luego una para sus tests. Con el objetivo de ser lo más riguroso posible, se probaron las siguientes combinaciones:

- Generación de la **dll** principal únicamente. El resultado será constatado luego contra los tests originales.
- Generación de la **dll** de tests únicamente. El código original será puesto a prueba contra la *suite* de tests generada.
- Generación de ambas **dll**. En este caso se pone a prueba todo generado.

Los primeros dos puntos son fundamentales ya que si hiciéramos solo el último, podría ser que estuviéramos generando código que no respete la semántica del original y luego tests que prueben eso o bien menos de lo que deberían.

Para cada una de estas combinaciones se realizaron dos casos, uno con el paso opcional de transformación de código mencionado anteriormente y otro sin.

4.2.1.2. Validación semántica y de formato

Para este eje contamos con ambas validaciones por lo cual se corrió PEVerify para validar el formato y luego los tests provistos para verificar que se preservó la semántica.

4.2.1.3. Inspeción manual

En una inspección rápida y de alto nivel del resultado no se ven casi diferencias entre la versión original y la generada. Sin embargo si entramos más en detalle podemos identificar algunas. Detallaremos las mismas diferenciando dos aspectos que son la comparación de CIL y de las tablas de metadata.

Comaparación de CIL

La diferencia más notable cuando empezamos a comparar el código CIL generado contra el original es el uso de referencias. Recordemos que la solución hace uso constante de las mismas para realizar el proceso de generación en una sola pasada. Lo que notaremos en el CIL es que vemos referencias incluso a tipos que estan declarados dentro del mismo ejecutable. A continuación mostramos un método de la librería TinyCsvParser, tanto en su versión original como generada.

```
.method public hidebysig
    instance class [mscorlib]System.Tuple`2<int32, int32> GetOffsetAndLength
    ( int32 length ) cil managed
{
    .maxstack 3
    .locals init (
        [0] class [mscorlib]System.Tuple`2<int32, int32>
    )

    IL_0000: nop
    IL_0001: ldarg.0
    IL_0002: call instance int32 TinyCsvParser.Ranges.RangeDefinition::
        get_Start()
    IL_0007: ldarg.0
    IL_0008: call instance int32 TinyCsvParser.Ranges.RangeDefinition::
        get_End()
    IL_000d: ldarg.0
    IL_000e: call instance int32 TinyCsvParser.Ranges.RangeDefinition::
        get_Start()
    IL_0013: sub
    IL_0014: call class [mscorlib]System.Tuple`2<!!0, !!1> [mscorlib]System.
        Tuple::Create<int32, int32>(!!0, !!1)
    IL_0019: stloc.0
    IL_001a: br.s IL_001c

    IL_001c: ldloc.0
    IL_001d: ret
}
```

Listing 4.1: Método GetOffsetAndLength original

```
.method public hidebysig
    instance class [mscorlib]System.Tuple`2<int32, int32> GetOffsetAndLength
    ( int32 length ) cil managed
{
    .maxstack 3
    .locals init (
        [0] class [mscorlib]System.Tuple`2<int32, int32>
    )

    IL_0000: nop
    IL_0001: ldarg.0
    IL_0002: call instance int32 [TinyCsvParser.dll]TinyCsvParser.Ranges.
        RangeDefinition::get_Start()
    IL_0007: ldarg.0
    IL_0008: call instance int32 [TinyCsvParser.dll]TinyCsvParser.Ranges.
        RangeDefinition::get_End()
    IL_000d: ldarg.0
    IL_000e: call instance int32 [TinyCsvParser.dll]TinyCsvParser.Ranges.
        RangeDefinition::get_Start()
    IL_0013: sub
    IL_0014: call class [mscorlib]System.Tuple`2<!!0, !!1> [mscorlib]System.
        Tuple::Create<int32, int32>(!!0, !!1)
    IL_0019: stloc.0
    IL_001a: br IL_001f

    IL_001f: ldloc.0
    IL_0020: ret
}
```

Listing 4.2: Método GetOffsetAndLength generado

Podemos observar que en el generado todo lo que referencia a tipos de la librería está prefijado por **[TinyCsvParser.dll]** mientras que en el original sólo vemos referencias para lo que son librerías externas como lo es **[mscorlib]**, una de las librerías principales de .NET.

El uso de referencias sin embargo no es exclusivo a métodos sino que se puede ver en todo lo que involucre tipos, por ejemplo en *fields* o interfaces. En el siguiente fragmento de la misma librería podemos ver un ejemplo de estos casos mencionados:

```
.namespace TinyCsvParser.Mapping
{
 .class public auto ansi beforefieldinit TinyCsvParser.Mapping.
     CsvCollectionPropertyMapping`2<class .ctor TEntity, TProperty>
   extends [mscorlib]System.Object
   implements class TinyCsvParser.Mapping.ICsvPropertyMapping`2<!TEntity,
       string[]>
 {
  .field private initonly string propertyName
  .field private initonly class
   TinyCsvParser.TypeConverter.IArrayTypeConverter`1<!TProperty>
       propertyConverter
 }
}
```

Listing 4.3: Fragmento original de la librería TinyCsvParser

```
.namespace TinyCsvParser.Mapping
{
 .class public auto ansi beforefieldinit TinyCsvParser.Mapping.
     CsvCollectionPropertyMapping`2<class .ctor TEntity, TProperty>
   extends [mscorlib]System.Object
   implements class [TinyCsvParser.dll]TinyCsvParser.Mapping.
       ICsvPropertyMapping`2<!TEntity, string[]>
 {
  .field private initonly string propertyName
  .field private initonly class
   [TinyCsvParser.dll]TinyCsvParser.TypeConverter.IArrayTypeConverter`1<!
       TProperty> propertyConverter
 }
}
```

Listing 4.4: Fragmento generado de la librería TinyCsvParser

Si volvemos al ejemplo 4.2 podemos observar una diferencia más, que también se puede ver en los distintos métodos que generemos. Esto es la generación de las instrucciones *branch*. Si comparamos el método original con el generado, se observa que en el primero vemos una instrucción **br.s** mientras que en el segundo utilizamos simplemente **br**. El set de instrucciones del CIL provee, para muchas de ellas, una versión normal y una más eficiente en tamaño denominada *short form*. Esta última no aplica para todos los casos, pero cuando lo hace es conveniente usarla ya que el tamaño que ocupa la misma es menor. El problema en este caso es que la posiblidad de usar esta versión más eficiente depende de la distancia del salto que se haga. A su vez, esto depende de la posición en la que se encuentre este salto en el cuerpo del método, su *offset*. Como la traducción a CIL la

hacemos a medida que vamos recorriendo el método en SIL, a priori no podemos saber cuanto van a ocupar las instrucciones ni donde van a posicionarse por lo que no es posible determinar si el caso aplica o no para utilizar una *short form*. La problemática es similar a lo que ocurría con los *labels* de los saltos pero con la diferencia de que en este caso la herramienta de SRM no nos provee una forma de solucionarlo. Como consecuencia de esto también se puede ver una diferencia en las etiquetas generadas, lo cual es coherente ya que la instrucción que no es *short form* ocupa más espacio.

El caso de los saltos es el único donde tenemos este problema, ya que en otras instrucciones lo que determina si podemos utilizar o no una *short form* está dado por el tamaño o tipo del operando.

La última diferencia que notamos es la falta de cierta información de *debugging* como pueden ser los nombres de las variables locales en los métodos. Pero esto no es por el funcionamiento de la solución propuesta sino simplemente porque, como mencionaremos más adelante, no estamos generando el archivo que contiene la información de *debug* asociada al programa.

Si bien estos son los puntos que notamos, no quiere decir que no haya más diferencias sino que no las percibimos al inspeccionar el CIL. Puede que existan algunos atributos o *flags* que no estemos generando o que sean algo distintos. Recordemos que después de todo, la herramienta implementada no es el compilador de .NET.

Comparación de tablas de metadata

Comparar el tamaño del ejecutable no resulta una métrica ideal ya que la versión generada puede carecer de algunas cosas como mencionabamos anteriormente. Por lo que no es necesariamente representativo comparar los tamaños del archivo obtenido con el original. Podría pasar que los tamaños sean similares y en realidad pasara que hay que cosas que no se generan en uno y si en el otro que ocupan más lugar (por ejemplo las instrucciones *branch* siempre en su version regular).

Optaremos entonces por comparar las tablas de metadata. Para hacer esto utilizamos la herramienta PEVerify que permite ver distintas informaciones sobre el ejecutable entre las cuales se encuentran las distintas tablas populadas con el detalle de sus filas.

En lo siguiente veremos comparativas entre las tablas para las distintas librerías utilizadas. Esto lo expondremos para las **dlls** principales y no para sus ejecutables que contienen los tests con el objetivo de que no se vuelva repetitivo.

Fleck

Tabla	Filas (Original)	Filas (Generado)
Module	1	1
TypeRef	**108**	**144**
TypeDef	50	50
Field	183	183
Method	335	335
Param	293	293
InterfaceImpl	8	8
MemberRef	**231**	**583**
Constant	42	42
CustomAttribute	188	188
ClassLayout	1	1
StandaloneSig	64	64
PropertyMap	12	12
Property	83	83
MethodSemantics	130	130
TypeSpec	34	34
FieldRVA	1	1
Assembly	1	1
AssemblyRef	3	3
NestedClass	20	20
GenericParam	2	2
MethodSpec	17	17

Tab. 4.1: Comparativa de tablas para la librería Fleck

Podemos observar un incremento de alrededor de un 33 % en la tabla de referencias de tipos mientras que vemos un poco más del doble en el valor para la tabla de referencias a métodos y *fields*.

TinyCsvParser

Tabla	Filas (Original)	Filas (Generado)
Module	1	1
TypeRef	**95**	**173**
TypeDef	93	93
Field	131	131
Method	313	313
Param	318	318
InterfaceImpl	23	23
MemberRef	**303**	**514**
Constant	10	10
CustomAttribute	93	93
StandaloneSig	40	40
PropertyMap	11	11
Property	21	21
MethodSemantics	31	31
MethodImpl	7	7
TypeSpec	129	129
Assembly	1	1
AssemblyRef	3	3
NestedClass	20	20
GenericParam	52	52
MethodSpec	95	95
GenericParamConstraint	2	2

Tab. 4.2: Comparativa de tablas para la librería TinyCsvParser

En este caso vemos un aumento mayor en la tabla de referencias a tipos pero más parejo entre las tablas de referencia de miembros. Tenemos un 82 % y un 69 % en lo que es TypeRef y MemberRef respectivamente.

DSA

Tabla	Filas (Original)	Filas (Generado)
Module	1	1
TypeRef	**81**	**333**
TypeDef	265	265
Field	1462	1462
Method	2538	2538
Param	2319	2319
InterfaceImpl	488	488
MemberRef	**3140**	**3458**
Constant	22	22
CustomAttribute	1759	1759
ClassLayout	1	1
StandaloneSig	496	496
PropertyMap	160	160
Property	437	437
MethodSemantics	605	605
MethodImpl	728	728
TypeSpec	517	517
FieldRVA	1	1
Assembly	1	1
AssemblyRef	3	3
NestedClass	167	167
GenericParam	685	685
MethodSpec	268	268
GenericParamConstraint	261	261

Tab. 4.3: Comparativa de tablas para la librería DSA

DSA es la librería más grande de las cuatro, esto se puede ver reflejado en las cantidad de entradas de sus tablas de metadata. Podemos observar un valor muy grande en la tabla de referencias de tipos, alrededor del cuadruple del original, pero sin embargo vemos solo un 10 % de aumento en la tabla MemberRef.

Por último resta la librería Optional. La misma en sí se compone de varias **dlls**, sin embargo mostraremos solo la principal para no extender tanto la comparación.

Optional

Tabla	Filas (Original)	Filas (Generado)
Module	1	1
TypeRef	**49**	**78**
TypeDef	46	46
Field	93	93
Method	263	263
Param	208	208
InterfaceImpl	35	35
MemberRef	**350**	**348**
CustomAttribute	155	155
StandaloneSig	58	58
PropertyMap	9	9
Property	19	19
MethodSemantics	19	19
MethodImpl	45	45
TypeSpec	150	150
Assembly	1	1
AssemblyRef	1	1
NestedClass	37	37
GenericParam	181	181
MethodSpec	41	41
GenericParamConstraint	5	5

Tab. 4.4: Comparativa de tablas para la librería Optional

En este caso encontramos casi un 60 % de aumento para la tabla TypeRef respecto de la versión original, mientras que para MemberRef observamos un valor prácticamente idéntico pero favoreciendo la versión generada.

Como podemos observar en las comparativas anteriores, la cantidad de elementos de las distintas tablas son idénticos salvo para dos tablas: TypeRef y MemberRef. En ambas vemos un aumento como esperábamos ya que la solución hace uso de referencias constantemente como vimos en secciones anteriores. Sin embargo no existe un patrón claro ya que vemos desde incrementos muy chicos como un 10 % hasta muy grandes como 300 %. Y también tenemos el caso de Optional, donde se obtuvo incluso un valor mejor para una de las tablas. Sin duda depende mucho de la librería y de cómo se relacionan y referencian los distintos ítems de la misma. Pero es claro que en la mayoría de los casos el ejecutable generado va a tener mayor tamaño de tablas de metadata lo que probablemente termine resultando en un mayor tamaño final del ejecutable.

4.2.2. Instrumentación

Sobre este eje hay diversos tipos de intrumentaciones que podríamos realizar. En esta ocasión optamos por una de las más conocidas: *logging*. La misma consiste en el agregado de reportes o bien impresiones por consola ya sea para auditoría o *debugging*.

4.2.2.1. Armado del experimento

Agregaremos entonces un *log*, es decir, una impresión por consola, de un mensaje con el nombre del método a cada uno de ellos. El experimento consiste de los siguientes pasos:

1. Se lee el ejecutable de entrada y se arma el modelo de objetos.
2. Se buscan todos aquellos métodos que tengan cuerpo (al menos una instrucción) y se transforma cada cuerpo de cada método de su representación en SIL a su equivalente en TAC. Se adiciona la información de tipos utilizando *Type Inference* y *Web Analysis*.
3. Se agrega un log con el nombre del método como primera instrucción en el cuerpo. Para ello es necesario agregar dos instrucciones, una para hacer la carga del mensaje y otra para imprimirlo.
4. Se convierte de TAC a SIL.
5. Se genera el nuevo ejecutable a partir del modelo de objetos.

Este proceso puede realizarse también sin convertir los cuerpos de SIL a TAC y viceversa, y hacer la instrumentación directamente sobre el SIL. Sin embargo, en la práctica los análisis y transformaciones seguramente se hagan sobre el TAC por lo que esta prueba parece ser más representativa.

El experimento se realizó con todas las librerías de prueba mencionadas anteriormente.

4.2.2.2. Validación semántica y de formato

Para validar el resultado, al igual que en el eje de generación, se corren los chequeos de semántica y formato utilizando la batería de tests y la herramienta de PEVerify respectivamente.

4.2.2.3. Inspeción manual

En esta instancia debemos detectar el valor agregado que en este caso es una impresión en pantalla. Esto es algo que podemos comprobar inspeccionando el ejecutable generando utilizando la herramienta ILSpy, para un enfoque más gráfico, o bien desde una terminal utilizando la herramienta *monodis* [22]. En el ejecutable debemos encontrar dos nuevas

instrucciones que se agregan al comienzo de cada método, que cumplen la función de imprimir el mensaje por pantalla. Adicionalmente, también se puede observar la salida del proceso de correr los tests, ya que ahí podemos ver como se van imprimiendo los mensajes a medida que los distintos métodos se van llamando.

A continuación mostramos un ejemplo de instrumentación de un método de la librería de algoritmos y estructuras de datos, DSA:

```
.method public hidebysig virtual
        instance void Clear () cil managed
{
    .maxstack 8

    IL_0000: nop
    IL_0001: ldarg.0
    IL_0002: call instance void class DSA.DataStructures.Trees.
        BinarySearchTree`1<!T>::Clear()
    IL_0007: nop
    IL_0008: ret
}
```

Listing 4.5: Método original

```
.method public hidebysig virtual
        instance void Clear () cil managed
{
    .maxstack 8

    IL_0000: ldstr "Entering method: Clear of class: SplayTree"
    IL_0005: call void [mscorlib]System.Console::WriteLine(string)
    IL_000a: nop
    IL_000b: ldarg.0
    IL_000c: call instance void class [DSA.dll]DSA.DataStructures.Trees.
        BinarySearchTree`1<!T>::Clear()
    IL_0011: nop
    IL_0012: ret
}
```

Listing 4.6: Método instrumentado

En el método instrumentado podemos ver que se agregan dos instrucciones, una para la carga del mensaje a imprimir y luego otra que realiza la llamada al método **System.Console.WriteLine**, encargado de mostrar el mensaje por pantalla. El resto del método se mantiene intacto a como estaba originalmente.

4.2.3. Optimización

Al igual que la instrumentación, el eje de optimización es un problema interesante y complejo por sí solo. Por este motivo, armaremos un ejemplo simple pero sobre el cual haremos una optimización interesante.

4.2.3.1. Armado del experimento

Definiremos un ejecutable que cuenta con una clase con su método *Main* y dos métodos más de los cuales uno es llamado por *Main* mientras que el otro no es utilizado. El objetivo es eliminar este último método de forma automática. Este tipo de optimización se denomina *dead code elimination* y sirve para reducir el tamaño de los ejecutables. Para esto utilizaremos el análisis *Call Graph* provisto por el *framework*. El mismo nos permite entender que método llama a cuál y de ahí podemos obtener el método que no es llamado por nadie y eliminarlo.

Los pasos del experimento son:

1. Se lee el ejecutable de entrada y se arma el modelo de objetos.
2. Se buscan todos aquellos métodos que tengan cuerpo (al menos una instrucción) y se transforma cada cuerpo de cada método de su representación en SIL a su equivalente en TAC. Se adiciona la información de tipos utilizando *Type Inference* y *Web Analysis*.
3. Se corre el análisis *Call Graph*.
4. Se eliminan aquellos métodos que no son parte del grafo, es decir, aquellos que no son invocados por ningún otro método.
5. Se convierte de TAC a SIL.
6. Se genera el nuevo ejecutable a partir del modelo de objetos.

4.2.3.2. Validación semántica y de formato

Podremos hacer una validación de formato usando PEVerify y tambien ejecutar el programa para ver que se comporta como el original. Pero al ser un ejemplo que no tiene tests automatizados, no tenemos un chequeo de preservación de semántica. Tampoco haría mucha diferencia sin embargo tener una *suite* de tests ya que el ejemplo es relativamente sencillo por lo que puede comprobarse ejecutándolo manualmente e inspeccionando el código generado.

4.2.3.3. Inspeción manual

A continuación veremos el código CIL del ejecutable original y el generado. Podremos ver de forma clara la optimización ya que la diferencia va a estar en que hay un método que no se encuentra presente en el ejecutable generado.

```
.class public auto ansi abstract sealed beforefieldinit ExamplesEXE.
    MainClass
        extends [mscorlib]System.Object
{
    .method public hidebysig static
        void Main (
            string[] args
        ) cil managed
    {
        .maxstack 8
        .entrypoint

        IL_0000: nop
        IL_0001: call void ExamplesEXE.MainClass::UsedMethod()
        IL_0006: nop
        IL_0007: ret
    }

    .method private hidebysig static
        void UsedMethod () cil managed
    {
        .maxstack 8

        IL_0000: nop
        IL_0001: ret
    }

    .method private hidebysig static
        void UnusedMethod () cil managed
    {
        .maxstack 8

        IL_0000: nop
        IL_0001: ret
    }
}
```

Listing 4.7: Ejecutable original

```
.class public auto ansi abstract sealed beforefieldinit ExamplesEXE.
    MainClass
    extends [mscorlib]System.Object
{
    .method public hidebysig static
        void Main (
            string[] args
        ) cil managed
    {
        .maxstack 8
        .entrypoint

        IL_0000: nop
        IL_0001: call void [ExamplesEXE.exe]ExamplesEXE.MainClass::
            UsedMethod()
        IL_0006: nop
        IL_0007: ret
    }

    .method private hidebysig static
        void UsedMethod () cil managed
    {
        .maxstack 8

        IL_0000: nop
        IL_0001: ret
    }
}
```

Listing 4.8: Ejecutable generado con la optimización

Si comparamos los ejemplos anteriores, podemos observar que efectivamente los métodos *Main* y *UsedMethod* se mantienen intactos mientras que desaparece el método *UnusedMethod* que es el que no estaba siendo utilizado. No se encuentran otras diferencias, más que las ya anteriormente mencionadas como el uso de referencias.

4.2.4. Generación programática

En este último eje buscamos probar la habilidad de generar código de forma dinámica a partir del modelo de objetos. Como veremos, el *pipeline* en este caso será mucho más reducido ya que no hay casos de prueba involucrados.

4.2.4.1. Armado del experimento

Este experimento consiste en armar un programa desde cero por medio de código utilizando directamente el modelo de objetos y luego generar un ejecutable para el mismo. Al igual que el eje de optimización, optamos por armar un caso simple pero que tuviera más de un componente como para que no fuera trivial. El ejemplo armado consiste de dos *namespaces*, Console y Languages. El segundo cuenta con dos clases que representan el idioma español e inglés. Cada una de ellas cuenta con un método *greet* que devuelve un saludo en ese idioma. El *namespace* Console cuenta con una clase Program, que tiene el método **Main** el cual instancia ambas clases, llama a los métodos mencionados anteriormente y luego imprime esos mensajes por pantalla.

El experimento cuenta con un único paso que es la generación del ejecutable a partir del modelo de objetos que representa la jerarquía de clases anteriormente descrita.

4.2.4.2. Validación semántica y de formato

Para este eje el código es generado directamente desde el programa, por lo que no contamos con una lectura de un ejecutable ni con transformaciones de código. Por este motivo no tendremos una validación semántica corriendo tests como en casos anteriores. Sin embargo se generó un **.exe**, en lugar de una librería, para este caso y se ejecutó corroborando que el resultado era el esperado. Adicionalmente realizamos la validación de formato utilizando PEVerify como en los demás ejes.

4.2.4.3. Inspeción manual

En este ambito inspeccionaremos con ILSpy el ejecutable generado para ver que respete la jerarquía de clases y componentes propuesta. A continuación mostramos el ejemplo generado:

```
.namespace Console
{
  .class public auto ansi abstract sealed Console.Program
      extends [mscorlib]System.Object
  {
    .method public hidebysig static
        void Main () cil managed
    {
        .maxstack 8
        .entrypoint

        IL_0000: nop
        IL_0001: newobj instance void [Gen.exe]Languages.English::.ctor()
        IL_0006: call instance string [Gen.exe]Languages.English::Greet()
        IL_000b: call void [mscorlib]System.Console::WriteLine(string)
        IL_0010: nop
        IL_0011: newobj instance void [Gen.exe]Languages.Spanish::.ctor()
        IL_0016: call instance string [Gen.exe]Languages.Spanish::Greet()
        IL_001b: call void [mscorlib]System.Console::WriteLine(string)
        IL_0020: nop
        IL_0021: ret
    }
  }
}
```

Listing 4.9: Namespace Console

Gen.exe sería el ejecutable en cuestión. Podemos observar el namespace *Console* tal cual fue descrito con su clase *Program* y su método Main el cual se encarga de instanciar las clases que veremos en el siguiente *namespace* e invocar a los métodos de las mismas imprimiendo su resultado por consola.

```
.namespace Languages
{
  .class public auto ansi Languages.English
      extends [mscorlib]System.Object
  {
      .method public hidebysig specialname rtspecialname
          instance void .ctor () cil managed
      {
          .maxstack 8
          IL_0000: ldarg.0
          IL_0001: call instance void [mscorlib]System.Object::.ctor()
          IL_0006: nop
          IL_0007: ret
      }

      .method public hidebysig
          instance string Greet () cil managed
      {
          .maxstack 8
          IL_0000: ldstr "Hello"
          IL_0005: ret
      }
  }

  .class public auto ansi Languages.Spanish
      extends [mscorlib]System.Object
  {
      .method public hidebysig
          instance string Greet () cil managed
      {
          .maxstack 8
          IL_0000: ldstr "Hola"
          IL_0005: ret
      }

      .method public hidebysig specialname rtspecialname
          instance void .ctor () cil managed
      {
          .maxstack 8
          IL_0000: ldarg.0
          IL_0001: call instance void [mscorlib]System.Object::.ctor()
          IL_0006: nop
          IL_0007: ret
      }
  }
}
```

Listing 4.10: Namespace Languages

En esta segunda parte del ejecutable generado podemos encontrar el *namespace* de los idiomas el cual contiene una clase para cada uno de ellos con sus respectivos métodos de saludo (además de sus constructores).

5. TRABAJO RELACIONADO

En esta sección mencionaremos brevemente algunas librerías y herramientas que cumplen una función similar respecto de Analysis.NET. Desde *frameworks* similares para otros lenguajes (Soot), compiladores (Roslyn) y herramientas para la inspección, lectura y generación de código .NET como lo son Cecil, CCI o ILSpy. En particular las últimas dos fueron utilizadas por el *framework* ya sea en el proceso de extensión del mismo o bien en su implementación.

Soot

Soot [14] es un *framework* de optimización para código Java desarrollado en el año 2000. Desde entonces fue ganando madurez y actualmente es utilizado en distintos ambitos desde lo educacional hasta la investigación y desarrollo. Cuenta con múltiples representaciones intermedias entre las cuales encontramos un *Three Address Code* (TAC) y un *Static Single Assignment* (SSA). Provee distintos análisis, como por ejemplo *pointer analysis*, cuenta con la construcción de un *call graph* y permite también generar código ejecutable.

Analysis.NET está principalmente inspirado en este *framework* pero focalizado en los programas .NET. Sin embargo, como hemos mencionado anteriormente, el mismo trabaja con representaciones intermedias y busca estar desacoplado de .NET. Por lo que incorporando módulos especiales para lo que es lectura y escritura del ejecutable, sería posible soportar otros lenguajes fuera de la familia de .NET, incluso Java.

Roslyn

Roslyn [10] es el compilador oficial de .NET para C# y Visual Basic. Desarrollado por Microsoft como un proyecto de código abierto, Roslyn cuenta con una numerosa comunidad que lo utiliza y brinda soporte.

Los compiladores suelen ser programas de caja negra, que toman como entrada el código fuente y generan código objeto. Roslyn se aleja de este enfoque tradicional y toma una estrategia distinta ya que está estructurado como una plataforma, un conjunto de APIs que permiten a los desarrolladores ser partícipes de la basta información que los compiladores manejan durante su proceso. Esto habilita a los desarrolladores a involucrarse en las distintas etapas del mismo como análisis sintáctico, chequeos semánticos y la generación de código. Lo cual abre puertas a la innovación en áreas como meta programación o generación de código dinámico en tiempo de ejecución.

Roslyn permite a los desarrolladores la construcción de analizadores de código, que entienden la sintaxis y estructura del código, y *code fixes* que permiten actuar sobre los

resultados obtenidos por estos analizadores. Actualmente los entornos de desarrollo más usados, como Visual Studio [23] o JetBrains Rider [24], hacen uso de las APIs expuestas por este compilador.

Roslyn provee estructuras de arboles, como *syntax-tree* asi también como *abstract syntax stree*, para los lenguajes que soporta. Pero lo que no ofrece son representaciones intermedias, como lo es TAC, las cuales son útiles para construir análisis estáticos de código. Además, al ser un compilador, trabaja con código fuente únicamente por lo que no es posible por ejemplo el análisis de librerías que son referenciadas por el código pero para las cuales no se cuenta con su código fuente. De la misma forma, tampoco es posible analizar programas que esten escritos en otros lenguajes que no sean C# o Visual Basic, lo que deja afuera diversos lenguajes incluso otros de .NET como F#. Tampoco provee análisis clasicos de *control-flow* y *data-flow*.

CCI

Common Compiler Infrastructure [11], o bien CCI, es un conjunto de librerías de código abierto desarrolladas por Microsoft Research [25] para la creación, lectura y manipulación de metadata y código CIL. CCI permite leer y modificar *assemblies*, módulos y archivos de *debugging* (PDBs). Sin embargo, al igual que Roslyn, no provee representaciones intermedias que facilitan la labor de construir y trabajar con análisis estáticos de código.

CCI fue originalmente usada para la lectura de ejecutables en Analysis.NET pero fue posteriomente remplazada ya que la misma se encuentra actualmente deprecada por lo que no cuenta con soporte como antes. Su remplazo fue utilizar la librería de *System Reflection Metadata* que también es desarrollada por Microsoft, forma parte del *Core* de .NET y además es multiplataforma. No obstante, sigue disponible en el *framework* por lo que es posible utilizar cualquiera de las dos para la etapa de lectura y transformación al SIL.

Cecil

Cecil [12] es una librería de código abierto desarrollada por el proyecto Mono [26], que permite inspeccionar y generar programas y librerías escritas en CIL. Cecil permite leer binarios de .NET mediante el uso de un modelo de objetos, sin la necesidad de cargar *assemblies* y usar *reflection*. También permite la modificación de binarios de .NET agregando nueva metadata o bien modificando el código CIL. Actualmente es ampliamente utilizada por la comunidad ya que soporta las funcionalidades más nuevas del lenguaje y cuenta con soporte activo.

Al igual que CCI y Roslyn, no provee representaciones intermedias ni herramientas para construir análisis estáticos de código. Podría ser utilizada por Analysis.NET para manejar la parte de lectura de los ejecutables, como se usó CCI o SRM, o bien para la

etapa de generación de código que se incluye en este trabajo utilizando SRM.

ILSpy

ILSpy [13] es una herramienta de código abierto que permite inspeccionar metadata y decompilar ejecutables de .NET. Internamente utiliza Cecil para cargar la metadata y el código CIL. El proceso de decompilado consiste de varias etapas de transformación de código que convierten el CIL en una representacion basada en arbol similar a un *Abstract Syntax Tree*.

Si bien ILSpy incluye algunos análisis estáticos que son requeridos para el proceso de decompilado, su objetivo no es dar una herramienta de análisis de código sino una forma amigable de inspeccionar binarios de .NET en su formato CIL o bien en su representación en C#. Existen herramientas gráficas[1] que hacen uso de ILSpy asi también como extensiones para Visual Studio y otros entornos de desarrollo.

[1] https://github.com/icsharpcode/AvaloniaILSpy.

6. CONCLUSIONES

Durante esta tesis extendimos la herramienta Analysis.NET agregando la funcionalidad de generar código ejecutable. La misma era una de las limitantes más fuertes con las que contaba el *framework* si la comparabamos con las herramientas similares que tenemos para otros lenguajes, como puede ser Soot en el caso de Java.

Adicionalmente agregamos la traducción de la representación intermedia TAC a SIL, lo cual nos habilita el uso de todos los análisis con los que cuenta el *framework* que están construidos sobre esta representación. Con estos análisis podemos hacer instrumentaciones, optimizaciones o distintas transformaciones de código que antes no eran posibles.

Combinando ambas funcionalidades, permitimos leer archivos ejecutables, realizar análisis y transformaciones de código y luego generar nuevamente un archivo ejecutable que impacta estos cambios. El resultado es una herramienta completa y potente que aporta mucha funcionalidad y potencial en el área de análisis estático de código para programas .NET.

6.1. Trabajo futuro

En esta sección cubriremos algunos aspectos que se pueden mejorar de la solución actual o bien que la misma no contempla y aportaría valor sumarlos. Si bien los módulos agregados brindan la funcionalidad esperada, siempre hay espacio para mejora.

Optimización de metadata e instrucciones

Como vimos en la sección de evaluación, la implementación propuesta no genera tablas óptimas de metadata sino que maneja más entradas en algunas de ellas de las que realmente necesita. Esto se debe principalmente al uso de referencias en lugar de definiciones. Como mejora sobre este tema se podría implementar una forma de ir reutilizando aquellas definiciones que se van agregando. Esto consiste en usar referencias mientras no se tenga la definición y al momento de tenerla desde ese punto en adelante usar la definición. También tenemos la posibilidad de cambiar la estrategia utilizada y optar por la de múltiples pasadas.

Otro punto posible a optimizar es la generación de instrucciones de tipo *branch*. Recordemos que la solución propuesta no es capaz de usar la versión eficiente de estas instrucciones (*short form*) por como va traduciendo y generando el CIL. El problema con esto es que la librería utilizada no permite editar las instrucciones una vez agregadas. Quizás se pueda idear una solución, también en múltiples pasadas, para esta etapa de modo de ir agregando lógicamente las instrucciones y calculando el tamaño que van a ocupar una

vez que se generen. De esta forma podemos luego hacer la generación ya sabiendo si estas instrucciones pueden o no usar su *short form*.

Soporte de distintas implementaciones de TAC

Como explicamos en la sección del *Assembler*, el mismo actualmente no acepta cualquier tipo de TAC sino solo el que es generado por el *Disassembler*. Esto a priori no es un inconveniente ya que todo el *framework* opera únicamente con esta implementación. Sin embargo, si se quisiera hacer algún trabajo enfocado en el eje de generación programática, podría ser más cómodo poder proveer cualquier tipo de TAC y que el *Assembler* genere el SIL correspondiente. Esto en sí también podría ser una mejora sobre el *Disassembler*, logrando así que la herramienta como un todo soporte más de una implementación de TAC.

Generación del archivo PDB

Durante este trabajo nos enfocamos en la generación del ejecutable ya sea una **dll** o bien un **.exe**. Sin embargo adicionalmente a estos, existen los archivos *Portable Data Base* o PDB. Los mismos son auxiliares generados por el compilaldor que proveen información sobre el ejecutable y sobre cómo fue producido lo cual que es útil para algunas herramientas, principalmente para los *debuggers*. Un ejemplo de esto es a la hora de configurar *breakpoints*, donde el *debugger* lee el archivo PDB asociado para relacionar la línea de código de algún programa con su ubicación en el ejecutable.

La librería de *System Reflection Metadata* utilizada en el *framework* provee un apartado para lo que respecta la lectura y generación de este tipo de archivos. De este modo sería posible agregar esta funcionalidad con las mismas herramientas que utiliza el *framework* actualmente.

Testing

Analysis.NET actualmente no cuenta con una batería de tests que permita la prueba automatizada de la herramienta. El *framework* se forma de muchos componentes y muchos de ellos son complejos por lo que este agregado sería ideal. Por el momento la forma con la que contamos para validar las etapas de lectura, transformaciones y generación son pruebas manuales con ejecutables, similar a lo que detallamos en la sección de evaluación.

La tarea de agregar esto sin embargo no debe ser subestimada ya que como decíamos, la herramienta es compleja. Esto no se debe solo a la basta funcionalidad que tiene sino tambien a que lee y genera ejecutables. Por este mótivo habría que analizar cual es la mejor forma de hacer estos tests. Probablemente lo conveniente sea probar unitariamente los distintos módulos y sus interacciones con las librerías utilizadas pero a su vez también incorporar PEVerify o alguna otra herramienta para validar los ejecutables.

Migración a NETCore

Analysis.NET actualmente está construido sobre .NET Framework en lugar de su versión de código abierto y multiplataforma NETCore por lo que algunas de sus funcionalidades, como las de interfaz gráfica, estan limitadas al sistema operativo Windows. Sin embargo la mayor parte de la herramienta puede ser utilizada también en otras plataformas utilizando Mono. De hecho este trabajo fue desarrollado en macOS.

La migración de Analysis.NET a NETCore podría ser una buena adición a la herramienta, facilitando aún más su utilización en las distintas plataformas.

REFERENCIAS

[1] .NET. https://dotnet.microsoft.com/learn/dotnet/what-is-dotnet.

[2] C#. https://docs.microsoft.com/en-us/dotnet/csharp/.

[3] .NET programming languages. https://dotnet.microsoft.com/languages.

[4] F#. https://fsharp.org/.

[5] Visual Basic. https://docs.microsoft.com/en-us/dotnet/visual-basic/.

[6] F# Software Foundation. https://foundation.fsharp.org/.

[7] Standard ECMA-335. http://www.ecma-international.org/publications/standards/Ecma-335.htm.

[8] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*, chapter Intermediate Code Generation. Addison-Wesley, 1986.

[9] Edgardo Julio Zoppi. *Análisis estático de programas .NET*. Tesis Doctoral. Universidad de Buenos Aires. Facultad de Ciencias Exactas y Naturales, 2019.

[10] Roslyn. https://github.com/dotnet/roslyn.

[11] Common Compiler Infrastructure. https://github.com/Microsoft/cci.

[12] Mono.Cecil. http://cecil.pe.

[13] ILSpy. http://www.ilspy.net/.

[14] Soot. https://github.com/soot-oss/soot.

[15] System Reflection Metadata. https://github.com/dotnet/runtime/tree/master/src/libraries/System.Reflection.Metadata.

[16] Portable Executable Format. https://docs.microsoft.com/en-us/windows/win32/debug/pe-format.

[17] Metadata. https://docs.microsoft.com/en-us/dotnet/standard/metadata-and-self-describing-components.

[18] Standard ECMA-335. Partition III: CIL Instruction Set. http://www.ecma-international.org/publications/standards/Ecma-335.htm.

[19] PEVerify. https://docs.microsoft.com/en-us/dotnet/framework/tools/peverify-exe-peverify-tool.

[20] Nunit. https://nunit.org/.

[21] GitHub. https://github.com.

[22] Dis/Assembling CIL Code. https://www.mono-project.com/docs/tools+libraries/tools/monodis/.

[23] Visual Studio. https://visualstudio.microsoft.com/.

[24] JetBrains Rider. https://www.jetbrains.com/rider/.

[25] Microsoft Research. https://www.microsoft.com/en-us/research/.

[26] Mono. https://www.mono-project.com/.