PEF: Python Error Finder

Bordese Andrés M., Hayes Tomás

{amb0109, toh0107}@famaf.unc.edu.ar

Director: **Damián Barsotti**

damian@famaf.unc.edu.ar

Trabajo especial presentado para acceder al título de
Licenciado en Ciencias de la Computación

Grupo de Sistemas Confiables
Facultad de Matemáticas Astronomía y Física
Universidad Nacional de Córdoba
Argentina

17 de diciembre de 2014

PEF: Python Error Finder

Bordese Andrés M., Hayes Tomás
{amb0109, toh0107}@famaf.unc.edu.ar

Trabajo especial presentado para acceder al título de
Licenciado en Ciencias de la Computación
Julio 2014

Palabras clave: Ejecución Simbólica; Python; Proxy; Error; Testing

Clasificación: D.2.5 Testing and Debugging: [Testing tools, Symbolic execution, Debugging aids]

Resumen

La verificación del correcto funcionamiento de los sistemas es uno de los aspectos más importante del desarrollo de software, y a su vez, uno de los más costosos de llevar a cabo. El testing tradicional y la verificación de modelos requiere mucho esfuerzo para lograr una buena cobertura de las posibles ejecuciones. En este trabajo, utilizamos y extendemos una técnica que combina ejecución simbólica, el poder de los razonadores recientes y la versatilidad de lenguajes puramente orientados a objetos, para crear una librería que explora y describe los caminos de un programa, detectando errores de manera automática y facilitando posteriormente, la generación de tests.

Presentaremos a *PEF*, una herramienta novedosa que hace uso de la técnica mencionada para detectar errores en programas escritos en Python 3 con muy poco esfuerzo por parte del usuario. Mostraremos la arquitectura del motor de ejecución simbólica y los aspectos fundamentales del lenguaje necesarios para construir el motor en forma de librería, sin tener que re-implementar el intérprete, como es usual en proyectos similares. También explicaremos el sistema de contratos que incorporamos para indicar pre y post condiciones que se desean aplicar y validar en los programas a ser explorados por PEF.

Declaración

Ninguna parte de esta tesis ha sido presentada en otra institución para conseguir otro grado o calificación. Todo el contenido de la misma fue producido por los autores a menos que lo contrario sea indicado en el texto.

Agradecimientos

Damián Barsotti por guiarnos en el trabajo.

Daniele Venzano por mostrarnos el paper clave.

Familiares y amigos por el apoyo.

Índice general

Capítulo 1

Introducción

Verificar que un sistema funcione de la manera deseada es una de las etapas mas costosas en el desarrollo de software. Se invierten muchas horas detectando errores en los programas que se hacen y modifican cada día. Reducir el tiempo que se invierte en esta etapa de corrección tiene un gran valor en la industria, pues afecta los costos de desarrollo de manera directa.

Tradicionalmente existen dos enfoques para detectar errores, el *Testing* y la *Verificación Formal*. El **Testing** es el método más utilizado en la industria. Generalmente consiste de personas, *Testers*, que generan entradas para el sistema de acuerdo a un criterio de test específico. Luego escriben pequeños programas que ejercitan el sistema bajo análisis con cada una de las entradas a validar, y comparan la salida real del sistema con la que los testers suponen es la correcta. La metodología de trabajo con testing, aunque sencilla, requiere mucho esfuerzo, por lo que se usa para validar una cantidad muy acotada de casos de uso. El análisis en busca de errores a través del testing no es exhaustivo, por lo tanto no se puede garantizar la ausencia de errores.

El otro gran enfoque para detectar errores es la **Verificación Formal**. En esta técnica se representa el sistema como un modelo matemático, generalmente un grafo, donde los nodos representan estados del programa, y las aristas las posibles transiciones entre estados; o como un conjunto de fórmulas que suponiéndolas ciertas, implican la especificación del sistema. Luego, sobre el modelo se realizan las verificaciones deseadas usando herramientas como *SMT-Solvers* o "*razonadores*". Al

trabajar sobre un modelo formal se puede asegurar o refutar el cumplimiento de cualquier propiedad que sea codificable (de manera finita) en el lenguaje del sistema de verificación, exceptuando los casos en que no alcanza el poder de cálculo para una exploración exhaustiva. Tanto el modelo como las propiedades se escriben usualmente en lenguajes diferentes al sistema bajo verificación, lo cual requiere además de personal más especializado. Este enfoque no es muy utilizado en la industria, con la excepción de los sistemas críticos, debido a que o bien la construcción del modelo suele ser una tarea muy compleja, requiriendo que el usuario entienda el sistema con mucho detalle; o el poder de cálculo necesario no escala mas allá de sistemas pequeños. Algunos ejemplos son la Lógica de Hoare [61], Owicki-Gries [63], Rely-Guarantee [62], Model Checking [68], etc.

La *ejecución simbólica* ha surgido como una técnica prometedora para razonar sobre el comportamiento de programas en varias entradas de manera simultánea, y explorar la mayoría de los caminos de ejecución. Recientemente una variedad de herramientas han tenido éxito usando ejecución simbólica para detectar defectos y generar casos de test de manera automática [16] [17]. La ejecución concreta tradicional, lo que típicamente realizan los intérpretes y compiladores, ejecuta el programa objetivo con valores de entrada fijos, como por ejemplo el entero "4". En contraste, la ejecución simbólica razona sobre la ejecución utilizando **conjuntos** de enteros (o conjuntos de otros valores), usualmente representados mediante **restricciones** (ej: los enteros mayores a 2). En consecuencia, utilizar ejecución simbólica suele involucrar la definición de nuevas semánticas simbólicas para el lenguaje objetivo.

Tradicionalmente esta semántica simbólica no estándar es implementada de alguna de las siguientes maneras:

- Escribiendo un nuevo intérprete para el lenguaje, o
- Escribiendo un traductor/compilador que lleva el código a otro lenguaje con soporte simbólico, o
- Escribiendo un instrumentador de código.
  Un programa que inyecta instrucciones en lugares estratégicos del código para lograr soporte simbólico manteniendo el intérprete original.

Por ejemplo, Java PathFinder [16] es un intérprete para bytecode de la JVM y DART [17] es un intérprete para programas binarios x86.

Desafortunadamente, estas formas son de algún modo "pesadas": requieren gran cantidad de código que debe ser desarrollado y mantenido; puede llegar a ser difícil seguir la evolución de un lenguaje y soportar extensiones no estándares; y suele presentar problemas manejando código generado o cargado dinámicamente.

Mostraremos una alternativa novedosa, presentada en 2008 con PeerCheck [2]. En ella se propone una *arquitectura de pares*, donde las semánticas simbólicas se implementan como una librería, en lugar de ser un nueva implementación del compilador/intérprete del lenguaje. La arquitectura de pares explota las capacidades de expansibilidad de los lenguaje puramente orientados a objetos; en particular, la habilidad de despachar dinámicamente operadores primitivos, como los aritméticos (+,-,*,/), acceso a arrays (var[index]), valores de verdad en saltos condicionales, etc.

A pesar de haber sido desacreditada como útil para casos prácticos [5], principalmente por sólo soportar ejecución simbólica para tipos primitivos, en este trabajo mostraremos que es posible extender la técnica propuesta por PeerCheck para cualquier tipo de objetos (primitivos o definidos por el usuario). Esta extensión de la técnica abre un nuevo camino, en el cual pueden construirse sistemas que detecten errores para programas de propósito general con relativamente poco esfuerzo, limitado principalmente por el poder de los razonadores, los cuales se encuentran en constante avance.

En la arquitectura de pares, el **motor de ejecución simbólica**, de ahora en adelante MES, es una librería que se ejecuta en el mismo proceso que el programa objetivo. En vez de correr el programa con parámetros de entrada concretos, el MES envía entradas simbólicas especiales que llamamos *objetos intermediarios*. Estas entradas son un tipo particular de objetos que le permiten al MES conocer la manera en que el programa objetivo manipula internamente sus entradas.

Cuando el programa objetivo realiza una operación sobre algún objeto intermediario, el intérprete despacha de manera dinámica una llamada a una función del MES, permitiéndole al motor llevar un registro de como las entradas y otros valores del programa se relacionan y cambian a medida que el programa es ejecutado.

Cuando el programa objetivo realiza un salto condicional sobre el valor de un objeto intermediario, el MES es informado nuevamente y consulta un *razonador* externo (SMT Solver) para decidir qué camino explorar (siempre que alguno sea alguno). Para obtener un buen cubrimiento de ramas, el MES re-ejecuta el programa objetivo varias veces, explorando cada vez un camino distinto. Esta arquitectura de pares permite una reducción sustancial de la complejidad en la forma de resolver el problema de la semántica simbólica.

Los desarrollos en ejecución simbólica mas importantes como KLEE [47], SAGE [48], Bitblaze [49] y S2E [50] se aplican a lenguajes que se compilan de manera estática, como C o Java. Sin embargo, cada vez son mas populares los lenguajes que se ejecutan directamente por un intérprete, sin ser compilados de manera estática, como Python, Perl o Java Script; que permiten un prototipado ágil y son fáciles de utilizar. Para el caso de Python, la única herramienta madura que los autores conocen es Chef [51], que logra crear un motor de ejecución simbólica utilizando el intérprete como especificación del lenguaje, ejecutando simbólicamente el programa y el intérprete al mismo tiempo. Ésta ultima herramienta tiene la desventaja de necesitar un gran poder de procesamiento, no disponible en computadoras personales. Hasta el día de la redacción de este trabajo, no hemos encontrado otros proyectos que utilizan la técnica de arquitectura de pares además de la propuesta por PeerCheck [2] en el 2011. Lo más cercano (para Python) está en el área de ejecución simbólica dinámica o *concolic testing* [6] por parte de ConPy [52], pero no encontramos referencia a algún software funcional.

Python, y en particular Python 3, fue nuestro lenguaje de elección. Es un lenguaje puramente orientado a objetos, ampliamente usado para implementar funcionalidades de alto nivel en sistemas y servidores web. Python facilita una metodología ágil de prototipado de nuevas funcionalidades. Dado que carece de verificación estática de tipos, tests que generen una buena cobertura de caminos son críticos.

En este trabajo, no sólo presentaremos la extensión desarrollada para la técnica de arquitectura de pares, también mostraremos una novedosa herramienta, creada por los autores, que permite detectar errores de manera automatizada en programas escritos en Python. Esta herramienta, la cual llamamos *PEF (Python Error Finder)*,

utiliza la técnica mejorada de arquitectura de pares antes mencionada y funciona como prueba de concepto de la misma. Además presentamos un sistema de contratos para expresar pre y postcondiciones sobre las funciones a explorar, permitiéndonos así reducir el dominio de entrada y encontrar ejecuciones que contengan errores. Una precondición fallida en la función objetivo indica que los valores de entrada son inapropiados, mientras que una postcondición fallida revela un error, ya sea en el código de la función objetivo o en su especificación.

El trabajo está organizado de la siguiente manera: en el capítulo 2, "Marco Teórico", se introducirán todos los conceptos esenciales para el entendimiento del trabajo. En el capítulo 3, "Trabajo", se explicará de manera detallada los pasos necesarios para la implementación de la antes mencionada arquitectura de pares sobre Python. En el capítulo 4, "Usos de PEF", se mostrarán las formas en que la herramienta puede ser utilizada, según sea el propósito. En el capítulo 5, "Experimentos", se explicarán los resultados obtenidos al utilizar PEF para explorar algunos algoritmos clásicos e incluso, el mismo código fuente de PEF. Finalmente en el capítulo 6, "Trabajos relacionados, aportes y propuestas", se hará un pequeño recorrido por los distintos trabajos que se han realizado en áreas relacionadas. Luego se resaltarán los principales aportes de este trabajo, posibles mejoras a realizar en trabajos posteriores y finalmente, la conclusión.

Capítulo 2

Marco Teórico

En este capítulo explicaremos algunos conceptos esenciales para un buen entendimiento del trabajo. El mismo estará organizado de la siguiente manera: En la sección 2.1, haremos una pequeña introducción sobre los errores en el desarrollo de software y el testing como herramienta para mitigarlos. En la sección 2.2, hablaremos sobre Python, el lenguaje utilizado en este trabajo. En la sección 2.3, mencionaremos de manera sencilla las formas tradicionales de recorrer un árbol. En la sección 2.4, mostraremos los razonadores o SMT-Solvers. Por último, en la sección 2.5, explicamos en detalle la técnica de ejecución simbólica.

2.1. Testing

Un proyecto de desarrollo de software está usualmente dividido en 5 etapas bien definidas:

1. Análisis de Requerimientos
2. Diseño
3. Implementación
4. Validación (Testing)
5. Mantenimiento

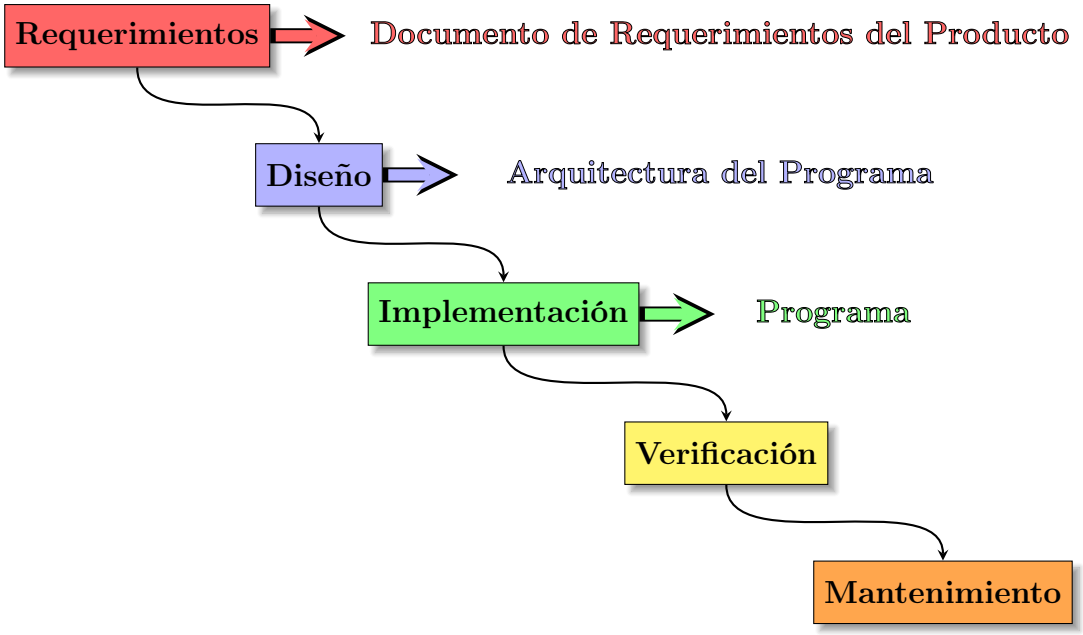


Figura 2.1: Las actividades del desarrollo del software representadas por el modelo de *cascada*, uno de los más utilizados.

El modelo de desarrollo mas famoso es el de *cascada*, que sigue el orden lineal de las 5 etapas recién mencionadas, y se ilustra en la figura 2.1.

En el desarrollo de un programa, los errores pueden ser introducidos en cualquier etapa. Es por eso que los errores se intentan detectar al finalizar cada etapa, con técnicas como la *inspección de código* [33], aunque algunos errores permanecen sin ser descubiertos. Al final, los mismos estarán reflejados en el código. Por ese motivo es altamente probable que el código final contenga algunos errores de diseño y de requerimientos, además de los introducidos durante la codificación per se. **Validación**, o **Testing**, es la actividad donde los errores que quedaron de todas las etapas anteriores deben ser detectados. El rol que cumple esta etapa es sumamente crítico para asegurar calidad en el programa final.

En las siguientes subsecciones explicaremos con mas detalle algunos conceptos complementarios, para luego profundizar sobre el testing.

2.1.1. Especificación de un Programa

Según el contexto, la especificación de un programa puede ser informal o formal. El primer caso surge del desarrollo de software como el acuerdo (o "contrato") que

se realiza entre el cliente y el equipo de desarrollo. Aquí, la especificación de un programa intenta ser una descripción completa del comportamiento del sistema que se va a desarrollar. Como debe ser entendido tanto por el equipo de desarrollo como por el cliente, se emplea un lenguaje informal en su redacción. Incluye además un conjunto de casos de uso que describe todas las interacciones que tendrán los usuarios con el programa.

El estándar IEEE 830-1998 [60], considera que una buena especificación debe ser:

- **Completa**. Todos los requerimientos deben estar reflejados en ella y todas las referencias deben estar definidas.
- **Consistente**. Debe ser coherente con los propios requerimientos y también con otros documentos de especificación.
- **Inequívoca**. La redacción debe ser clara de modo que no se pueda mal interpretar.
- **Correcta**. Todos los requerimientos indicados reflejan las necesidades que deben ser satisfechas por el software.
- **Trazable**. Se refiere a la posibilidad de verificar la historia, ubicación o aplicación de un ítem a través de su identificación almacenada y documentada.
- **Priorizable**. Los requisitos deben poder organizarse jerárquicamente según su relevancia para el negocio y clasificándolos en esenciales, condicionales y opcionales.
- **Modificable**. Aunque todo requerimiento es modificable, se refiere a que debe ser fácilmente modificable.
- **Verificable**. Debe existir un método finito con un costo razonable para que todos los requerimienos de la especificación puedan ser validados por una máquina o una persona.

Si la especificación es formal, la descripción se realiza en un lenguaje matemático, el cual especifica lo que se espera del programa de manera completa. Sirve para

analizar posibles comportamientos en forma exhaustiva, ayudar en el diseño (verificando que se cumplen propiedades deseadas) y construir implementaciones con la ayuda de herramientas. Una descripción formal describe *que* es lo que el sistema debe hacer, pero no *como* lo haría.

Una de las formas usuales de especificar formalmente el comportamiento de un programa es mediante un "**sistema de contratos**". Los sistemas de contratos se utilizan en la etapa de testing para asegurarse que lo diseñado y requerido es consistente con el software desarrollado. Un **contrato** es una expresión, en algún lenguaje formal, que denota qué debe ser cierto en un determinado punto del programa. Los sistemas de contratos, generalmente, dividen sus contratos en dos grandes tipos:

- Contratos de **Precondición**

  Los contratos de precondición se encargan de especificar las propiedades que deben ser ciertas **antes** de ejecutar el programa o función objetivo.

- Contratos de **Postcondición**

  Los contratos de postcondición se encargan de especificar las propiedades que deben ser ciertas **luego** de ejecutar el programa o función objetivo asumiendo la precondición (si es que existe).

La herramienta desarrollada en este trabajo, PEF, implementa un sistema de contratos similar al descripto. Más detalles sobre su funcionamiento e implementación pueden verse en la sección 3.3 página 79.

2.1.2. Error

El término *error* es comúnmente usado con dos acepciones. La primera se refiere a la discrepancia entre un valor calculado, observado o medido y el real, especificado, o valor teóricamente correcto. Es decir que *error* se refiere a la diferencia entre la salida concreta de un programa y la salida esperada o correcta (según la especificación del mismo). En esta interpretación, un *error* es esencialmente una medida de la diferencia entre el sistema real y el ideal. La segunda acepción hace referencia a una acción humana que resulta en un software que contiene fallas o defectos.

2.1.3. Defecto

Un *defecto* es una condición que causa que el sistema falle en el desempeño de una función. También conocido como *bug*, es la razón principal por la cual un programa se comporta de manera incorrecta. El término *error* también suele ser utilizado para referirse a *defectos* (utilizando una pequeña variación de la segunda definición de *error*). En este trabajo vamos a continuar con el uso de los términos de la manera usual y no se va a hacer una diferencia explícita entre *error* y *defecto*.

2.1.4. Falla

Una *falla* es la inhabilidad de un sistema o componente de realizar una tarea requerida de acuerdo a sus especificaciones. Decimos que ocurre una falla si el comportamiento del software es diferente al comportamiento especificado. Las *fallas* pueden presentarse como problemas de funcionalidad o rendimiento. Una *falla* se produce sólo cuando hay un *defecto* presente en el sistema, sin embargo, la presencia de *defectos* no garantiza la manifestación de fallas. En otras palabras, los *defectos* tienen el potencial de causar *fallas* y su presencia es necesaria pero no suficiente para que ocurra una *falla*. Notar que la definición no implica que una *falla* tiene que ser observable. Es posible que una *falla* pueda ocurrir pero que no sea detectada.

2.1.5. Oráculos

Para hacer testing sobre cualquier programa, es necesario tener una descripción de su comportamiento esperado y un método para determinar cuando el comportamiento observado se ajusta al esperado. Para esto necesitamos un *oráculo de test*. Un *oráculo* es un mecanismo, diferente al programa en si mismo, que puede ser utilizado para determinar la corrección de la salida del programa para los *casos de test*. Conceptualmente podemos ver al *testing* como el proceso en que los *casos de test* son "ejecutados" por el *oráculo* y por el programa bajo testing. La salida de los dos es, luego, comparada para determinar si el comportamiento del programa fue correcto para los casos de test, como se muestra en la figura 2.2.

Los *oráculos de test* son necesarios para el *testing*. Idealmente, nos gustaría tener

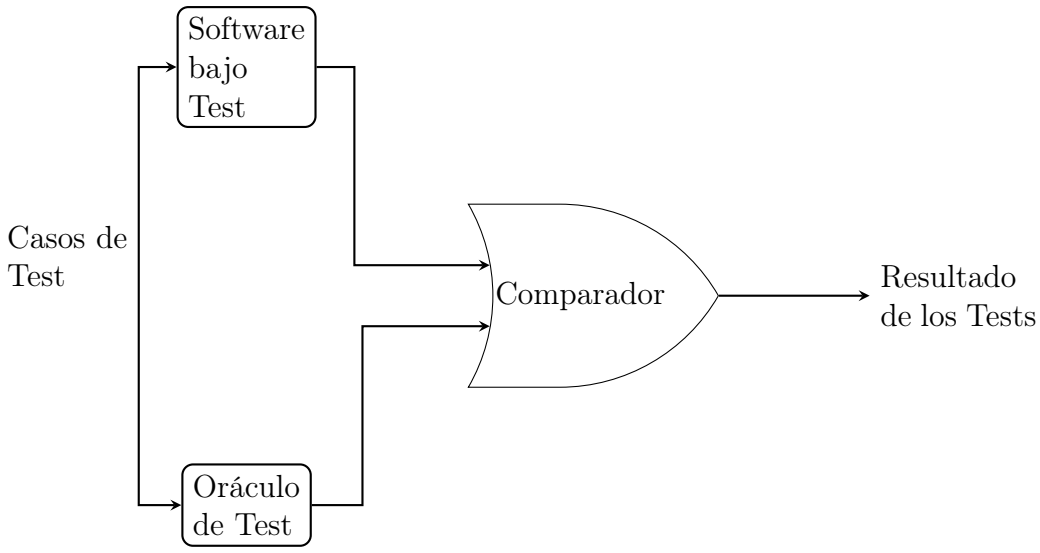


Figura 2.2: Test y Oráculos de Test

un oráculo automatizado que siempre nos diera las respuestas correctas. Sin embargo, los oráculos suelen ser seres humanos y pueden cometer errores. Como resultado, ante discrepancias entre el oráculo y el programa se debe verificar primero que el valor generado por el oráculo es correcto, y sólo luego se puede afirmar que hay un defecto en el programa.

Un oráculo humano generalmente utiliza la especificación del programa para decidir cual debe ser el comportamiento "correcto". Sin embargo, las especificaciones en si mismas pueden contener errores, ser imprecisas, o ser ambiguas.

Existen sistemas en los cuales los oráculos son generados automáticamente a partir de la especificación de los módulos. Estos oráculos eliminan el esfuerzo de determinar el valor esperado para un *caso de test*. Un *oráculo* generado a partir de una especificación sólo va a generar resultados correctos si la especificación es correcta. Usualmente, estos sistemas que generan *oráculos* de manera automática requieren una especificación formal las cuales no suelen ser generadas durante la etapa de diseño.

2.1.6. Casos de Test y Criterio de Selección

Tener casos de test que sean buenos para revelar la presencia de defectos es central para un testing exitoso. La razón de esto es que si hay un defecto en un

programa, éste aún se comporta de manera correcta para muchas otras entradas. Sólo un subconjunto de entradas son las que ejercitan el defecto y causan que el programa no se comporte como es esperado. Se puede afirmar entonces que "el *testing* es tan bueno como lo son sus *casos de test*".

Idealmente nos gustaría determinar un conjunto mínimo de caso de test tales que, la ejecución exitosa de los mismos implique que no hay errores en el programa. Este objetivo ideal usualmente no puede ser cumplido debido a restricciones teóricas y prácticas. Cada caso de test cuesta dinero, ya que se necesita tiempo para generarlo, tiempo de máquina para ejecutarlo, y más esfuerzo es necesario para evaluar los resultados. Por esta razón es que nos gustaría minimizar la cantidad de casos de test. Estos son los dos principales objetivos del testing, **maximizar la cantidad de errores detectados y minimizar el número de casos de test**. Como estos dos objetivos suelen ser contradictorios, el problema de seleccionar el conjunto de casos de test se vuelve más complejo.

Al seleccionar casos de test, el objetivo principal es asegurar que si hay un error o defecto en el programa, el mismo sea ejercitado por "uno" de los casos. Un caso de test ideal es aquel que "pasa" (es decir que su ejecución no revela ningún error) sólo si no hay errores en el programa. Un posible conjunto ideal de casos de test es el que incluye todas las posibles entradas del programa. Esto es llamado **testing exhaustivo**. Sin embargo, el testing exhaustivo no es práctico, ya que incluso para programas pequeños el dominio de entrada puede ser extremadamente grande.

Entonces, ¿Cómo deberían seleccionarse los casos de test? ¿ Con qué principio deberíamos incluir elementos del dominio del programa en el conjunto de casos de test y no incluir otros ? Para esto utilizaremos un *criterio de selección de test*. Dado un programa P y su especificación S, un criterio de selección especifica la condición que debe cumplir un conjunto de casos de test T. Por ejemplo, si el criterio es que todas las sentencias de un programa sean ejecutadas por lo menos una vez durante el testing, un conjunto de casos de test T satisface este criterio para un programa P si la ejecución de P con T asegura que toda sentencia de P es ejecutada por lo menos una vez.

Existen dos propiedades fundamentales que debe cumplir un criterio de test: **con-**

fiabilidad y **validación**. Decimos que un criterio es confiable si todos los posibles conjuntos (de casos de test) que satisfacen el criterio detectan los mismos errores. Decimos que un criterio es válido si para cada error en el programa existe al menos un conjunto que satisface el criterio y es capaz de detectar el error. Un teorema fundamental del testing es que si un criterio de testing es válido y confiable, si un conjunto que satisface ese criterio no revela ningún defecto, entonces no hay errores presentes en el programa. Sin embargo, ha sido mostrado que no existe un algoritmo capaz de generar un criterio válido para cualquier programa arbitrario.
Obtener un criterio que sea confiable, válido y que además pueda ser satisfactible por un número 'manejable' de casos de test suele ser imposible. Es por eso que usualmente los criterios escogidos no son válidos y confiables, como por ejemplo: "El 90 % de las líneas van a ser ejecutadas por lo menos una vez".

2.1.7. Testing de Caja Negra

Hay dos enfoques básicos para hacer testing: de **caja negra** y de **caja blanca**. En el testing de caja negra, también llamado **testing funcional** o de **comportamiento**, la estructura del programa no es tenida en cuenta. El tester (persona encargada de crear los tests) sólo conoce las entradas que pueden ser dadas al sistema y la salida que esperar del mismo. Para crear/agregar casos de test sólo se toma en cuenta la especificación del programa/módulo. El **testing aleatorio** y el **testing exhaustivo** son los ejemplos más simples de testing de caja negra.

2.1.8. Testing de Caja Blanca

Mientras que el testing de caja negra intenta lidiar con la funcionalidad de un programa, al **testing de caja blanca**, o **testing estructural**, le concierne la implementación.
Para hacer testing sobre la estructura o implementación de un programa, el testing de caja blanca apunta a generar casos que logren el *cubrimiento* deseado de diferentes estructuras. Varios son los enfoques existentes para adoptar criterios de test, siendo los más relevantes: **flujo de control** y **flujo de datos**.

Criterio Según Flujo de Control

Aquí se considera el control de flujo del programa para definir un criterio de test. El criterio de test más simple es ejecutar todas sentencias del programa al menos una vez, pero es bastante débil. Otro criterio común pero mas poderoso es visitar cada rama de cada control de flujo, osea, ejercitar cada decisión del programa por la rama verdadera y la falsa. Éste último criterio contiene al anterior, pero hay casos en que no se detectan errores, como cuando las decisiones involucran varias condiciones. Otro criterio aún más general es considerar los caminos de ejecución de todo el programa. El problema con éste último método es que encontraremos una cantidad infinita de caminos en la mayoría de los programas que contengan ciclos, y aún así, hay muchos errores que no pueden ser detectados con este criterio, por ejemplo, cuando se omiten sentencias de validación. Éste es el criterio utilizado por PEF.

Criterio Según Flujo de Datos

Aquí, se toma en cuenta la creación y uso de las variables como criterio de test. Un criterio es el siguiente: Se debe ejercitar la declaración de todas las variables, al menos una vez. Otro puede ser: por cada variable creada, se deben ejercitar todas sus posibles formas de uso.

2.1.9. Generación de Casos de Test

Una vez elegido un criterio de test, se procede por generar un conjunto de entradas para el programa, que idealmente debe ejercitar el criterio de test en su totalidad y ser mínimo en su tamaño. La razón de esto es que la creación y ejecución de los test es una tarea costosa, por lo que se busca abarcar el mayor espacio de búsqueda, usando la menor cantidad de casos de test posible. Encontrar un balance entre estos dos aspectos, generalmente excluyentes, no suele ser una tarea fácil. Luego se ejecuta cada caso de test, y un oráculo intenta detectar fallas, comparando el comportamiento del programa, con la especificación del mismo.

2.2. Python

Python es un lenguaje de programación de alto nivel ampliamente usado; soporta los paradigmas de orientación a objetos, imperativo y funcional. Sus características principales son el tipado dinámico [21] y el manejo de memoria de manera automática. Es un lenguaje interpretado, siendo CPython [22] su implementacón más difundida y la que ha sido utilizada en este trabajo. En esta sección introduciremos algunos detalles de Python3 (la versión de Python usada en este trabajo) y CPython que fueron explotados durante el desarrollo de PEF.

2.2.1. Sintaxis Básica

Variables y Asignación

Como Python es un lenguaje de tipado dinámico, las variables no necesitan ser declaradas antes de ser utilizadas e inclusive puede cambiar el tipo de lo que almacenan durante la ejecución.

```
>>> a = 1    # Le asigno a la variable 'a' el entero 1
>>> a = "aa" # Le asigno a la variable 'a' el string "aa"
```

Python soporta asignaciones múltiples y asignaciones simultaneas.

```
>>> a = b = c = d = 1  # Asignación simultanea
>>> a, b, c = 1 ,2, 3  # Asignación múltiple
```

Bloque de Código

Uno de las características fundamentales de Python es su indentación obligatoria. En vez de utilizar llaves ({}) o palabras claves para delimitar bloques de código, se utiliza la *indentación*. Un incremento en la indentación marca el comienzo de un bloque mientras que un decremento indica el cierre. Los bloques de código no pueden estar vacíos.

La sentencia de no operación "pass"

La más simple de todas, la sentencia **pass** literalmente no hace nada. Es el equivalente a un noop (no-operation), o un "skip" en Guarded Command Language

[25]. Generalmente se usa cuando sintácticamente se requiere una sentencia pero el programa no tiene que hacer nada.

```
>>> while 1:
...     pass
```

Ejecución condicional "if"

Quizás una de las sentencias más conocidas, utilizando la palabra clave **if** podemos ejecutar condicionalmente un bloque de código. Se puede utilizar en conjunto de sentencias **elif** y finalmente un **else**, todas opcionales. La palabra clave "elif" es una reducción de "else if", y se utiliza para evitar la indentación excesiva. Una secuencia if ... elif ... elif es una generalización de la sentencia "switch-case" presente en otros lenguajes.

```
>>> x = 2
>>> if x < 0:
...     print("x es negativo")
... elif x == 0:
...     print("x es cero, cero es x")
... else:
...     print("x es positivo")
x es positivo
```

Iterando con un "for"

La sentencia **for** en Python se comporta un poco diferente a otros lenguajes como C [23] o Pascal [24]. En vez de iterar sobre una progresión numérica (como Pascal), o permitirle al usuario realizar cualquier operación sobre la variable iteradora (como C), la sentencia **for** de Python itera sobre elementos de una **secuencia** (ej., listas, strings, tuplas, etc.), respetando el orden en que aparecen en la misma.

```
>>> gadgets = ["laptop", "smartphone", "tablet"]
>>> for i in gadgets:
...     print("Gadget:", i)
...
Gadget: laptop
Gadget: smartphone
Gadget: tablet
```

Naturalmente, no es recomendable modificar la secuencia siendo iterada dentro del cuerpo del for.

Definición de Funciones

La palabra clave **def** introduce la definición de una función. Debe seguir el nombre de la función y entre paréntesis una lista de sus parámetros. El bloque de código que acompaña a la declaración es el cuerpo de la función; la primera línea suele ser un string que se utiliza como documentación de la función y se conoce como **docstring**.

Para que las funciones puedan devolver un valor debemos utilizar la sentencia **return** dentro del cuerpo de la función. Si una función no tiene el comando return la misma devolverá el valor *None*.

```
def mi_funcion(a, b, c):
    """
    Esta es la documentación de la funcion mi_funcion
    """
    d = a + b + c
    return d
```

En 2.2.10 podemos encontrar detalles sobre los diferentes estilos de argumentos de las funciones.

Definición de Clases

Se usa la palabra clave **class** para declarar una nueva clase. Debe seguir el nombre con el que llamaremos a la clase, y entre paréntesis, pueden ir otras clases de las cuales se generará una herencia. Luego en un bloque se definen los atributos y los métodos.

```
class MiClase(Herencia1, Herencia2):
    """
    Esta es la documentación de la clase MiClase
    """
    atributo_1 = "atributo de clase"

    def method_1(self, a, b):
        return "Se ha llamado a un método de la clase MiClase"
```

Cuando se definen métodos de instancia, como '*method_1*' en el ejemplo anterior, siempre se debe debe declarar al menos un parámetro. Por convención llamado '**self**', el primer argumento de todo método de instancia es una referencia a la instancia misma a partir de la cual el método fue llamado. Este parámetro '**self**' es agregado

de manera implícita cada vez que una instancia hace una llamada a un método. Así, por ejemplo:

```
mc = MiClase()     # Creación de una instancia de MiClase
mc.method_1(1, 2) # Se llamada al método ‘method_1‘ de MiClase donde a -> 1
                   # b -> 2. En este caso ‘self‘ sería ‘mc‘. El intérprete
                   # se encarga de agregar este parámetro automáticamente
```

Excepciones

Podemos hacer un try/catch con las palabras clave **try, except, finally**, y generar una excepción con la palabra clave **raise**. A continuación un poco de código que hace uso de esta sintaxis:

```
try:
    r = mi_lista[4]
except IndexError:
    print("Índice fuera de rango.")
finally:
    print("Esto fue todo amigos!")

if r == 42:
    raise AssertionError("No me gusta el 42!")
```

Importando objetos

Se pueden importar objetos de otros módulos (2.2.7) con la palabra clave **import**. Cuando se importa un objeto, se usa el nombre del módulo, quitando el sufijo *.py* y el mismo se agrega al ámbito local (2.2.8). A continuación se muestran los usos mas comunes de la sintaxis, tomando como ejemplo el módulo *math.py*, incluido en las librerías básicas de Python:

```
>>> import math
>>> from math import pi  # Solo se importa el objeto pi, bajo el nombre ‘pi’
>>> import math as math_con_otro_nombre  # útil cuando colisionan los nombres
>>> math.pi  # Nos referimos al objeto particular ‘pi’ dentro del modulo math
3.141592653589793
>>> pi
3.141592653589893
>>> math_con_otro_nombre.pi
3.141592653589893
```

Los módulos son explicados en la sección 2.2.7.

2.2.2. Tipos en Python

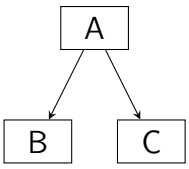
Python es un lenguaje de tipado dinámico [21], es decir que son los valores y no las variables las que tiene asociado un tipo. Se hace uso extenso del Duck Typing (sección 2.2.2, página 20), por lo cual suele tener mas relevancia la manera en que se comportan los objetos, que el tipo al que pertenezcan.

Para obtener el tipo de un objeto se puede utilizar la función predefinida "$type(x)$", la cual tiene como parámetro el objeto a analizar, y devuelve el tipo/clase del mismo.

```
>>> type([1, 2, 3])
list
>>> type({"pef": "Python Error Finder"})
dict
>>> type("PEF")
str
>>> type(0)
int
```

Muchas veces no queremos hacer una verificación tan estricta sobre el tipo. La función predefinida "$isinstance(x, cls)$" nos permite conocer si x es una instancia de la clase cls o de alguna clase que herede de cls (Notar que la herencia es transitiva).

Supongamos la siguiente jerarquía de clases.



Sean a, b y c instancias de A, B y C respectivamente.

```
>>> type(a) == A
True
>>> isinstance(a, A)
True

>>> type(b) == A
False
>>> isinstance(b, A)
True

>>> type(c) == A
False
>>> isinstance(c, A)
True
```

Duck Typing

El Duck Typing es un concepto atribuido a James Whitcomb Riley [20] que puede ser anunciado como:

"Cuando veo un ave que camina como pato, nada como pato y grazna como pato, a esa ave yo la llamo un pato."

En el Duck Typing, el programador sólo se concentra en que los objetos se comporten de la manera esperada, en vez de asegurar que sean de un tipo específico. Son los atributos y métodos de un objeto los que determinan las semánticas válidas. Por ejemplo, supongamos que queremos crear una función que use los métodos *caminar* y *graznar* de un objeto. En un lenguaje que no posee Duck Typing deberíamos requerir que su parámetro sea de tipo Pato. En un lenguaje con Duck Typing, la función tomaría un objeto de cualquier tipo y simplemente llamaría a los métodos caminar y graznar, produciendo errores en tiempo de ejecución si no están definidos.

El Duck Typing puede ser visto en acción en la siguiente figura. En cuanto a la función *en_el_bosque* le concierne, el objeto *persona* es un pato.

```
class Pato:
    def graznar(self):
        print("Cuaaaaaac!")
    def caminar(self):
        print("El pato camina hacia el rio")

class Persona:
    def graznar(self):
        print("La persona imita el graznido del pato")
    def caminar(self):
        print("La persona camina de regreso a su casa")
    def nombre(self):
        print("Tomas Hayes")

def en_el_bosque(pato):
    pato.graznar()
    pato.caminar()

def juego():
    donald = Pato()
    tomas = Persona()
    en_el_bosque(donald)
    en_el_bosque(tomas)

juego()
```

Figura 2.3: Ejemplo de Duck Typing. Patos y Personas.

Mutabilidad de los Objetos

Los objetos en Python pueden ser clasificados en dos grandes grupos **mutables** e **inmutable**. Decimos que una instancia es **mutable** si la misma **puede** ser modificada luego de su creación. Equivalentemente decimos que una instancia es **inmutable** si **no puede** ser modificada después de ser creada.

Ejemplos de tipos mutables son las listas, los diccionarios y los conjuntos. Ejemplos de tipos inmutables son los enteros, las tuplas y los strings.

```
>>> l = [1, 2]  # l es una lista
>>> l[0] = 2    # Modificamos un elemento de la lista
>>> print(l)
[2, 2]   # La lista se modifico!

>>> t = (1, 2)  # t es una tupla
>>> t[0] = 2    # Veamos si se modifica
TypeError: 'tuple' object does not support item assignment
```

Números

En Python disponemos tres tipos numéricos predefinidos:

- **int**: Representan los números enteros. A diferencia de los enteros de C [23], los de Python tienen una precisión arbitraria, pudiendo almacenar números tan grandes como el usuario lo desee.
- **float**: Representan los números reales. Son números de punto flotante de 64 bits al igual que los "doubles" en C.
- **complex**: Representan los números complejos, con una parte real y otra imaginaria.

Secuencias

En Python tenemos los siguientes tipos básicos de secuencias

- **list**: Representan las listas, que pueden contener cualquier objeto, y son mutables.
- **tuple**: Similar a las listas, pero inmutables.

- **str**: Representan cadenas de caracteres (palabras o strings), inmutables.

Todas estos tipos permiten operaciones similares, como acceder a elementos por su posición en la secuencia con la sintaxis [*pos*] (Ejemplo: "hola"[2] devuelve la palabra "l"). A continuación un poco de código que muestra la sintaxis de creación, acceso y modificación de estos objetos:

```
>>> l = [1, 2, 3, "Hola"] # Creación
>>> t = (1, 2, 3, "Hola") # Creación
>>> s = "1, 2, 3, Hola"   # Creación
>>> l[3]  # acceso
"Hola"
>>> t[3]  # acceso
"Hola"
>>> s[3]  # acceso
"2"
>>> l[3] = "Chau" # Modificación
>>> l
[1, 2, 3, "Chau"]
>>> t[3] = "Chau"
TypeError: 'tuple' object does not support item assignment
>>> s[3] = "Chau"
TypeError: 'str' object does not support item assignment
```

Hay algunas operaciones que valen la pena mencionar con un poco mas de detalles:

- La suma ($x + y$) es equivalente a la concatenación (x e y deben ser secuencias del mismo tipo).
- El producto ($x * n$) es equivalente a concatenar x consigo mismo n veces (x es una secuencia y n un entero).
- La comparación ($x \oplus y$, con $\oplus$ algún operador de comparación como $<$, $>$, $<=$, $==$, *etc*) entre secuencias del mismo tipo se llevan a cabo utilizando el orden lexicográfico, con la diferencia que para las listas y las tuplas, la relación punto a punto depende del tipo de los elementos. Por ejemplo, vale que *[1, 2, 3] <[1, 2, 4]*, pero realizar la operación *[1, 2, 3] <['1', 2, 3]* genera la excepción *TypeError* cuando se ejecuta la comparación *1 < '1'*.

Diccionarios

Los diccionarios (palabra clave **dict**) son un conjunto de pares *clave, valor*, que soporta búsqueda eficiente por clave. A continuación un poco de código que muestra

la sintaxis básica de creación, acceso y modificación:

```
>>> d = {"a": "Chau", 3:"Hola"}  # Creación
>>> d[3]  # Acceso
"Hola"
>>> d[3] = "Chau"  # Modificación
>>> d
{"a": "Chau", 3:"Chau"}
```

Conjuntos

Los conjuntos (palabra clave **set**) contienen elementos de forma desordenada, pero soy muy eficientes para decir si un elemento está en el. A continuación un poco de código que muestra algunas operaciones en acción.

```
>>> c = {1, 2, 3, "Hola"}  # Creación
>>> c[1]  # Acceso
TypeError: 'set' object does not support indexing
>>> c.add("Chau")
>>> c
{"Hola", 2, 3, 1, "Chau"}
>>> c.remove("Chau")
>>> c
{"Hola", 2, 3, 1}
>>> "Hola" in c  # Preguntar si contiene un elemento
True
>>> otro = {1}
>>> c.intersection(otro)
{1}
>>> c.union(otro) == c
True
```

Los operadores de comparación (*<*, *<=*, *==*, *>*, *etc*) se interpretan entre conjuntos siguiendo la relación de inclusión. Por ejemplo, vale que *{1} <{1,2}*.

Por último, existe también la versión inmutable de los conjuntos, llamada **frozenset**.

2.2.3. Slices

La manera mas usual de obtener un elemento o una serie de elementos de una secuencia (tupla, string, lista, etc) es a través de los slices. La notación es la siguiente: [*start*:*stop*:*step*], donde *start* indica la posición de inicio, *stop* la posición de parada y *step* la distancia de los saltos al recorrer la secuencia.

Por lo general usamos los slices sin darnos cuenta al obtener un elemento de una lista como *lista[posición]*. En ese caso, *stop* toma el valor de *start + 1* y *step*, *1*.

Se puede leer más acerca de los slices en la documentación oficial de Python: `https://docs.python.org/3/library/functions.html#slice`.

2.2.4. Funciones Útiles

Generando secuencias de enteros

La forma usual de generar una secuencia de enteros es con la función **range**. La función genera una lista con una progresión entera. Podemos usarla con 1, 2 o 3 parámetros para obtener diversos resultados:

- $range(stop)$

  Genera una progresión incremental desde 0 hasta *stop* (sin incluirlo).

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- $range(start, stop)$

  Genera la progresión incremental desde *start* hasta *stop* (sin incluirlo).

```
>>> range(4, 10)
[4, 5, 6, 7, 8, 9]
```

- $range(start, stop, step)$

  Genera la progresión desde *start* hasta *stop* (sin incluirlo) con saltos de *step* unidades.

```
>>> range(0, 10, 2)
[0, 2, 4, 6, 8,]
```

Obteniendo la longitud de un objeto

La función **len** se utiliza para obtener la longitud o tamaño (como cantidad de elementos) de un objeto que contenga otros elementos (listas, palabras, tuplas, diccionarios, conjuntos, etc.)

```
>>> len([1,2,3])
3
>>> len((1,2))
2
```

2.2.5. Despacho dinámico de operadores

Una de las características usualmente encontradas en los lenguajes puramente orientados objetos es el despacho dinámico de operadores. Python es uno de estos lenguajes, y todos los operadores primitivos (ej. "+" o "-") son traducidos por el intérprete como llamadas a métodos sobre los operandos. En la tabla siguiente se muestran todos los operadores de Python (no se pueden agregar nuevos) y sus respectivas traducciones.

Operación	Traducción	Traducción Alternativa
x + y	x.__add__(y)	y.__radd__(x)
x - y	x.__sub__(y)	y.__rsub__(x)
x * y	x.__mul__(y)	y.__rmul__(x)
x / y	x.__truediv__(y)	y.__rtruediv__(x)
x // y	x.__floordiv__(y)	y.__rfloordiv__(x)
x % y	x.__mod__(y)	y.__rmod__(x)
x ** y	x.__pow__(y)	y.__rpow__(x)
- x	x.__neg__()	
~x	x.__invert__()	
x >> y	x.__rshift__(y)	y.__rrshift__(x)
x << y	x.__lshift__(y)	y.__rlshift__(x)
x and y	x.__and__(y)	y.__rand__(x)
x or y	x.__or__(y)	y.__ror__(x)
x >y	x.__gt__(y)	y.__rgt__(x)
x >= y	x.__ge__(y)	y.__rge__(x)
x <y	x.__lt__(y)	y.__rlt__(x)
x <= y	x.__le__(y)	y.__rle__(x)

x == y	x.__gt__(y)	y.__rgt__(x)
x != y	x.__ne__(y)	y.__rne__(x)

Como regla general, cada vez que el intérprete recibe una instrucción con un operador, traduce la misma como una llamada a un método del primer operando, es decir, de la primera a la segunda columna de la tabla superior. En caso de que el operador sea binario, $x \oplus y$, el método traducido puede tener dos comportamientos válidos:

1. Resolver la operación.

   Si el método conoce como tratar con y devuelve el resultado de la operación.

2. Delegar la resolución a la traducción alternativa.

   Suele suceder que los tipos de x e y son diferentes y los métodos de x no saben como tratar con cosas de tipo $type(y)$. En este caso, el método traducido llama a la traducción alternativa, que es un método sobre y. Este nuevo método está obligado a resolver la operación, ya sea computándola o generando un error.

Utilizando el conocimiento que tenemos sobre la traducción de los operadores podemos "sobrecargarlos". Dado que los métodos que definen a los operadores (la traducción y la traducción alternativa) pueden ser implementados o redefinidos en cualquier clase, basta con definirlos para que el operador pueda operar sobre ella.

En la figura 2.4 se muestra una implementación muy simple de números complejos en la cual podemos apreciar como se realiza la sobrecarga del operador "+". Notemos que la clase "Complex" tiene definidos los métodos $\_\_add\_\_$ y $\_\_radd\_\_$. El método $\_\_add\_\_$ verifica que el segundo operando, "other", sea de un tipo válido para sumarlo, sino delega el cálculo a la traducción alternativa. En el caso de $\_\_radd\_\_$, la lógica es similar, pero si el segundo operando no es de un tipo válido se genera un error.

Para la expresión $x + 1$, el método $\_\_add\_\_$ verifica el tipo del argumento *other* (el entero 1, en este caso), y como no es un número complejo, agrega el argumento a la parte real de x. Alternativamente, la evaluación de $1 + x$ resulta en

```
class Complex:
    def __init__(self, real, img):
        self.real = real
        self.img = img

    def __add__(self, other):
        if isinstance(other, Complex):
            return Complex(self.real + other.real, self.img + other.img)
        elif isintance(other, int):
            return Complex(self.real + other.real, self.img)
        else:
            return other.__radd__(self)

    def __radd__(self, other):
        if type(other) not in [Complex, int]:
            raise TypeError("unsupported operand type(s) for +: 'int' and 'str'")
        else:
            return self.__add__(other)
```

Figura 2.4: Implementación de números complejos

la llamada $1.\_\_add\_\_(x)$. Como el entero no sabe como lidiar con un objeto de tipo *Complex*, la suma es delegada a $x.\_\_radd\_\_(1)$. Nuestra implementación de $\_\_radd\_\_$ hace uso de la propiedad conmutativa de la suma para a su vez, delegar el trabajo a $\_\_add\_\_$.

El despacho dinámico de operadores es uno de los pilares fundamentales de la *arquitectura de pares*. Esta capacidad de elegir en tiempo de ejecución la función a ejecutar nos ayudó a crear los objetos intermediarios. Como ya mencionamos, los objetos intermediarios imitan a otros objetos, por lo que sobrecargar los operadores es algo fundamental. Los objetos intermediarios son explicados en la sección 3.2.

2.2.6. Valor de verdad de los objetos

En Python, todos los objetos tienen un valor de verdad asociado. Cada vez que un objeto x necesita ser utilizado como un booleano, se busca por el método $x.\_\_bool\_\_$:

- Si el método no existe: se devuelve por defecto $True$,
- Si el método existe: se devuelve el resultado de $x.\_\_bool\_\_()$, teniendo el mismo que ser de tipo bool.

Esta propiedad es utilizada por PEF para controlar el flujo del control del programa, definiendo dinámicamente y según el contexto el valor de verdad a asignarle a cada objeto intermediario.

2.2.7. Módulos

A medida que los programas se hacen mas grandes, suele ser convenientes dividir el código en varios archivos. Esto tiene varias ventajas, como código mejor organizado y conveniencia para importar partes del mismo desde otros programas. Estos archivos son llamados módulos. Entonces, un módulo de Python es simplemente un archivo que contiene definiciones y sentencias, y su nombre contiene el sufijo *.py*. Cada módulo tiene su propia tabla de símbolos, de manera que no haya conflictos entre los nombres de las variables globales de los módulos cuando éstos se importen (esto se detalla en la sección 2.2.8).

2.2.8. Ámbitos (namespaces) y alcance de las variables

Todos los objetos que se usan en Python deben estar declarados, ya sea en en la raíz del módulo, dentro de una definición de función o predefinidos en el módulo *builtins*. Python cuenta con varios diccionarios encargados de mapear nombres de variables con valores, los cuales se crean durante ejecución, estos diccionarios son los *namespaces*, comúnmente llamados ámbitos o contextos.

Namespace	Alcance	Orden
Local	Declaraciones dentro de la definición de la última función que se llamó y se está ejecutando.	0
No Local	Declaraciones dentro de la definición de las otras funciones, que encierran la declaración de la función que genera el ámbito local.	1
. Global	Declaraciones que están únicamente en la raíz del módulo, y que contiene el código que se está ejecutando.	2
Built-in	Declaraciones predefinidas por Python.	3

Como se muestra en la tabla, se manejan cuatro ámbitos diferentes. Cuando se requiere acceder al valor de una variable se busca su nombre en los diccionarios de ámbitos en el orden indicado. Por ejemplo, cada vez que se ingresa al cuerpo de una función durante la ejecución, el ámbito local es redefinido para esa función. Notar que al importar objetos de otros módulos, éstos se agregan al ámbito local o global según donde haya ocurrido la importación.

2.2.9. Builtins

El intérprete de Python tiene varias funciones, tipos y constantes internamente definidas que se encuentran disponibles para ser utilizadas en cualquier momento de la ejecución. Estos objetos, están en todo momento ya que se encuentran almacenados en el ámbito built-in. En CPython, la implementación del lenguaje utilizada por los autores, cada módulo tiene un ámbito builtin independiente. Este ámbito, a diferencia de los otros, se encuentra accesible a través de un diccionario global con el nombre $\_\_builtin\_\_$. En la tabla siguiente mostramos algunos objetos built-in.

False	Instancia de *bool* con valor de verdad falso.
None	El único valor de tipo *NoneType*. None suele ser utilizado para representar la ausencia de un valor.
len()	Retorna el largo de un objeto. El argumento debe ser un objeto que contenga otros objetos, donde la operación de *contar cuantos hay* tenga sentido.
isinstance()	Indica si un objeto es instancia de una clase dada (aceptando herencia múltiple).

Durante el desarrollo de PEF nos encontramos con funciones internas que hacían que algunas verificaciones de nuestros objetos intermediarios fallen. Para poder resolver este problema, primero creamos funciones internas equivalentes que salteaban esas validaciones; en segundo lugar, reemplazamos en $\_\_builtins\_\_$ las referencias a las funciones originales por sus versiones sin validación. Esto será explicado con mayor detalle en la sección 3.5

2.2.10. Pasaje de parámetros en funciones

Dada una función en Python podemos encontrarnos con dos tipos de argumentos: posicionales y por palabra clave (o simplemente por nombre). Los argumentos por palabra clave necesitan tener un valor por defecto en la definición de la función. En la siguiente función, "a","b" y"c" son argumentos posicionales mientras que "x","y" y "z" son por palabra clave (1, 2 y 3 son sus respectivos valores por defecto).

```
def f(a, b, c, x=2, y=1, z=None):
    pass
```

Figura 2.5: Función que define parámetros posicionales y por palabra clave

Para llamar a una función, los argumentos posicionales son *obligatorios* y se escriben en el orden en que fueron declarados.

```
def f(a, b, c):
    return a * b + c

f(1,2,3)  # a <- 1, b <- 2, c <- 3
```

Figura 2.6: Llamada a función con parámetros posicionales

En el caso de los argumentos por palabra clave, los mismos son *opcionales* (es decir que se pueden no escribir) y pueden ser pasados de dos formas: en el orden en que fueron escritos o utilizando su nombre.

```
def f(x=1, y=2, z=3):
    return a * b + c

# Son optativos
f()
f(z=2)
# En el mismo orden en que fueron definidos
f(11,22,33)  # x <- 11, y <- 22, z <- 33
# Por nombre
f(x=3, y=5, z=3) # Ordenados
f(y=1, z=2, x=2) # Desordenados
```

Figura 2.7: Llamada a función con parámetros por palabra clave

En el caso en que se desee usar parámetros posicionales y por nombre en una función, se deben escribir primero los posicionales y después los por nombre; tal y

como se muestra en la figura 2.5.

*args y **kwargs

Supongamos que queremos programar una función suma, pero queremos que sea capás de sumar una cantidad arbitraria de elementos. Es decir, queremos poder hacer $suma(1, 2)$ y también $suma(1, 2, 3, 4)$. Para esto, Python permite utilizar un nombre de variable con un asterisco adelante, que por convención es ***args**. Cuando una función tiene en su definición ***args**, a la hora de llamarla, todos los parámetros por posición que no estén declarados en la definición de la función serán almacenados en la tupla "args" (y podrán ser accedidos en el cuerpo de la función).

```
def sum(a, b, *args):
    r = a + b
    for i in args:
        r += i
    return r

sum(1,2,3,4,5)  # a <- 1, b <- 2, args <- (3,4,5)
sum(1,2)  # a <- 1, b <- 2, args <- ()
```

Figura 2.8: Llamada a función con *args y parámetros posicionales

Algo similar sucede cuando una función presenta ****kwargs** en su definición. A la hora de llamar a esta función, todos los parámetros pasados por palabra clave (ej x=1, y=2) cuyo nombre no esté en la definición de la función, serán almacenados en el diccionario "kwargs".

```
def mostrar_persona(nombre="yo", **kwargs):
    print("nombre: %s" % nombre)
    for i in kwargs:
        print("%s: %s" %(i, kwargs[i]))

mostrar_persona(nombre="Andrés") # kwargs <- {}
mostrar_persona(nombre="Tomas", altura=1.86) # kwargs <- {altura: 1.86}
```

Figura 2.9: Llamada a función con **kwargs y parámetros por nombre

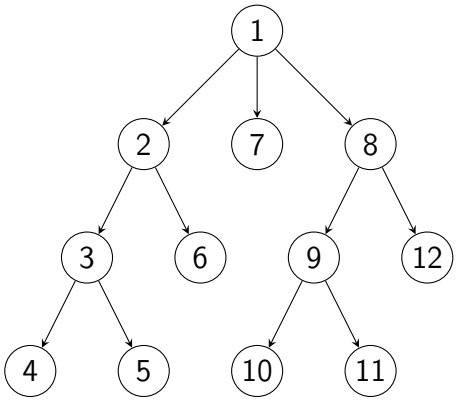


Figura 2.10: Orden en que los nodos del árbol son recorridos usando DFS

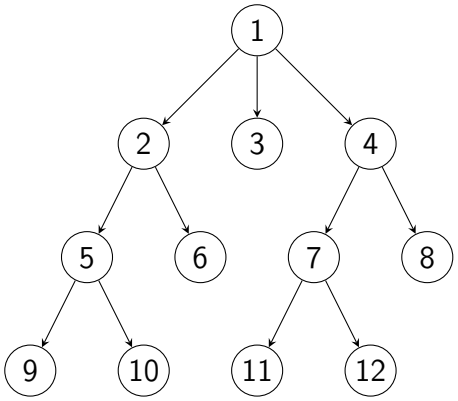


Figura 2.11: Orden en que los nodos del árbol son recorridos usando BFS

2.3. Recorrido de un árbol

La **búsqueda por profundidad** o **DFS** (por sus siglas en ingles depth-first search) es un algoritmo utilizado para recorrer o buscar dentro de una estructura de datos de *árbol* o *grafo*. Se comienza desde el nodo raíz y se explora tan profundo como sea posible por cada rama antes de comenzar el *bactracking*.

Una versión de DFS fue investigada en el siglo 19 por el matemático francés Charles Pierre Trémaux como una estrategia para resolver laberintos.

En la **búsqueda por anchura** o **BFS** (por sus siglas en ingles breadth-first search), se comienza desde el nodo raíz y se explora por nivel, visitando todos los nodos de un nivel, antes de pasar al siguiente.

2.3.1. Complejidad

En general, la complejidad de BFS y DFS es similar. Para el recorrido de un árbol, la complejidad temporal es de $\mathcal{O}(|A|)$, lineal en la cantidad de aristas; y la espacial, en el peor de los casos de $\mathcal{O}(|N|)$, lineal en la cantidad de nodos. Queda claro entonces, que la elección de uno u otro algoritmo no depende de su complejidad sino del orden particular con el que recorren los vértices.

Para aplicaciones de DFS en dominios específicos, cómo la búsqueda de soluciones en inteligencia artificial o los indexadores de la web, el grafo a ser recorrido puede ser muy grande para ser visitado completamente o hasta incluso infinito. En estos casos, la búsqueda se realiza hasta una determinada "*profundidad*"; debido a recursos limitados, como el espacio de memoria o disco. Generalmente no se lleva registro del conjunto de todos los nodos visitados anteriormente. Cuando la búsqueda se limita a una profundidad, el tiempo sigue siendo lineal en término del número de nodos y aristas expandidas (aunque este número no es igual al tamaño total del grafo, ya que algunos nodos pueden ser visitados más de una vez y otros ninguna) pero la complejidad en espacio de esta variante de DFS es sólo proporcional al límite de profundidad, y como resultado, mucho menor al espacio necesario para buscar a la misma profundidad con BFS. Para este tipo de aplicaciones, variaciones de DFS se suelen utilizar como heurísticas para para determinar ramas similares. Cuando el límite de búsqueda no se conoce a priori, una búsqueda por profundidad iterativa [18] va aplicando DFS sistemáticamente, incrementando el límite de de búsqueda en cada iteración.

2.3.2. Pseudocódigo

El algoritmo de DFS toma como **entrada** un grafo *Graph* y un nodo *node*. A la **salida** todos los nodos a los que se puede llegar desde **node** son marcados como *visitados*.

Pseudocódigo recursivo.

```
def DFS (Graph, node):
    node.visited = True
    for m in Graph.adjacentNodes(node):
        if (node, m) in Graph.edges:
            # Se puede ir de node a m
            if not m.visited:
                DFS(Graph, m)
```

Pseudocódigo Iterativo.

```
def DFS_Iterativo (Graph, node):
    stack = [n]
    while stack != []:
        # Quedan nodos por recorrer
        node = stack.pop()
        if not node.visited:
            for m in Graph.adjacentNodes(node):
                if (node, m) in Graph.edges:
                    # Se puede ir de node a m
                    stack.append (m)
```

Estas dos variaciones del pseudocódigo recorren los vecinos de un nodo en sentido opuesto. En el algoritmo recursivo, dado un nodo n el primer vecino de n recorrido es el primer vecino de la lista de nodos adyacentes ($G.adjacentNodes(n)$), mientras que en la versión iterativa el primer vecino visitado es el último de la lista de nodos adyacentes. La implementación no recursiva es similar a la de BFS sólo difiriendo en dos cosas:

1. Usa una pila (stack) en vez de una cola (queue).

2. Retrasa la verificación de si un nodo ha sido visitado o no hasta quitarlo de la pila, en vez de hacerlo antes de meterlo en la pila.

Si bien la implementación recursiva parece más sencilla, en la práctica no suele ser muy utilizada ya que hace uso intensivo de la pila de llamadas recursivas del lenguaje que suele tener un límite (En el intérprete de Python de este autor, la pila de llamadas recursivas de tiene un límite de 989). Esta restricción limita el tamaño de grafos que el algoritmo es capaz de manejar (En el caso de Python a grafos de hasta 989 nodos). El algoritmo iterativo, en cambio, no utiliza el stack del lenguaje sino uno propio que podemos considerar infinito, ya que sólo está limitado por la memoria del sistema.

2.4. Razonadores

Llamamos razonadores a herramientas como los SMT-Solvers [31] o CSP-Solvers [32], que para un conjunto de objetos dado, buscan satisfacer una serie de restricciones sobre el estado de los mismos. Los razonadores se utilizan para resolver problemas de satisfactibilidad en expresiones escritas en alguna teoría soportada. Se pueden pensar como una generalización del conocido problema de satisfactibilidad de expresiones booleanas (SAT), donde se busca una asignación de valores para las variables involucradas en la expresión, de forma que la misma adquiera el valor de verdad deseado; donde ahora además se permite que las variables y las operaciones sean parte de otras teorías, como por ejemplo, la de los números reales, o conjuntos.

Los razonadores son utilizados para verificar corrección de programas, crear fragmentos de programas que satisfagan alguna postcondición, resolver problemas de decisión (Muy usado en IA), encontrar soluciones a ecuaciones y probar teoremas.

En este trabajo utilizamos el SMT Solver Z3 [58] de Microsoft, que es bastante poderoso para resolver ecuaciones que involucran teorías de enteros, booleanos, arreglos de enteros, ordenes parciales, cuantificadores y lógica temporal.

2.4.1. ¿Por qué necesitamos razonadores?

Uno de los problemas fundamentales de la Ciencias de la Computación teórica es la **satisfactibilidad**, es decir, el problema de determinar cuando una expresión o una restricción tiene solución. Los problemas de satisfacción, también llamados como *Problemas de Satisfacción de Restricciones* (CSP por sus siglas en inglés), se presentan en áreas muy diversas, desde hardware y verificación de software hasta planificación.

La *Satisfactibilidad Proposicional*, **SAT** [38], es uno de los problemas de resolución de restricciones más investigados. Se busca decidir si una fórmula proposicional es satisfactible, es decir, si existe una asignación de valores a las variables que hace que la fórmula sea verdadera. Las herramientas que implementan los algoritmos para decidir problemas SAT son llamados **SAT Solvers**.

Durante la última década, SAT ha recibido mucha atención por parte de la

comunidad científica. Se produjeron grandes mejoras en eficiencia para los SAT Solvers, con las cuales las aplicaciones que dependen de ellos (siendo la IA una de la áreas fundamentales) también se han beneficiado. Aunque SAT es un problema NP-Completo, en la práctica muchas instancias del problema son "tratables".

Sin embargo, existen problemas que pueden describirse de manera más natural en lógicas más expresivas, como por ejemplo, la lógica de primer orden. Intuitivamente, se puede pensar que estos problemas pueden resolverse con un probador de teoremas [39] de propósitos generales, pero usualmente este no es el caso. La razón principal es que la mayoría de las aplicaciones no requiere satisfactibilidad general de primer orden, sino satisfactibilidad con respecto a un **teoría** [40] que fije la interpretación de algunos predicados y funciones.

Llamamos **Satisfactibilidad Modulo Teorías** (SMT) al problema de decidir la satisfactibilidad de fórmulas con respecto a una teoría τ.

Los métodos de razonamiento de propósito general pueden ser forzados a considerar sólo interpretaciones consistentes con una teoría τ, pero sólo explícitamente incorporando los axiomas de la teoría a la fórmula de entrada. Incluso cuando esto es posible, es decir, cuando la teoría es finitamente axiomatizable, la performance de los probadores es inaceptable. Una alternativa más viable es utilizar métodos de razonamiento optimizados para la teoría en cuestión.

Con los trabajos pioneros de Nelson y Oppen [34] [35] [36] [37], se desarrollaron e implementaron algoritmos eficientes para decidir consistencia sobre conjunción de expresiones atómicas en teorías de primer orden decidibles. Estos algoritmos han sido implementados para un gran número de teorías: función sin interpretación, aritmética lineal, teorías de vectores de bits, de arrays, de listas, entre otras. Sin embargo, el dominio de trabajo de muchas aplicaciones requiere verificar la satisfactibilidad de fórmulas que son combinaciones booleanas de expresiones atómicas y proposiciones atómicas en alguna teoría, por lo que el razonamiento proposicional debe combinarse de manera eficiente con el razonamiento específico de la teoría. Los algoritmos de decisión específicos para una teoría sólo pueden trabajar con conjunciones de expresiones atómicas, y no son capaces de razonar con Booleanos.

Actualmente, el trabajo en SMT se concentra en construir algoritmos de decisión

(SMT Solvers) que combinen de manera eficiente el razonamiento proposicional y el específico de una teoría. Existen dos grandes enfoques: *eager* y *lazy*. El enfoque *eager* [41] [42] [43] [44] [46] codifica el problema (una fórmula SMT) como una fórmula proposicional equisatisfactible [45] utilizando consecuencias (propiedades) de la teoría de interés, y luego ejecutar un SAT Solver con esa fórmula como entrada. Este enfoque se aplica, en principio, a cualquier teoría con una base decidible para el problema de satisfactibilidad. El costo de este enfoque es el gran tamaño que suele adquirir la traducción. La traducción obtenida puede ser resuelta por un SAT Solver de propósitos generales. Su viabilidad depende de la habilidad del SAT Solver utilizado de procesar rápidamente la información codificada (en una gran fórmula SAT).

El enfoque *lazy* combina algoritmos de decisión específicos de la teoría (conocidos como $\tau - solvers$) con un SAT Solver eficiente. De este modo, el sistema conjunto ($\tau - solver + SAT - Solver$) es capaz de resolver fórmulas en lógica de primer orden (libres de cuantificadores) con una estructura booleana arbitraria. Su principal ventaja es que, al utilizar razonadores específicos para una teoría se pueden utilizar las estructuras de datos y algoritmos de decisión que mejor se adapten a la teoría en cuestión, generando una mejor performance del sistema. Actualmente, el enfoque *lazy* es el utilizado por los SMT Solvers modernos.

En los últimos años se ha producido un gran interés en la comunidad científica sobre los aspectos teóricos y prácticos que envuelven a SMT debido a todos los campos a los que puede aplicarse (ej. Planificación [53], Razonamiento Temporal [54], Verificación de pipelines en circuitos [55], Verificación de proof-obligations en Sistemas [56], Verificación de sistemas en tiempo real [57], etc.). Muchos SMT Solver han sido creados en ámbitos académicos como z3 [58] o CVC4 [59].

2.4.2. Z3

Z3 es un SMT Solver de alto rendimiento desarrollado en Microsoft Research por Leonardo de Moura y Nikolaj Bjørner [58]. Este razonador es activamente utilizado en un amplio abanico de tareas: verificación y testing de software/hardware [15] [76], resolución de restricciones [77], análisis de sistemas híbridos [78], seguridad, biología

(análisis in silico) [79] y problemas geométricos [80]. Sus principales características son:

- Soporte builtin para varias teorías:
  - Vectores de Bits de tamaño fijo.
  - Aritmética Lineal entera y real.
  - Arrays.
  - Funciones sin interpretación.
  - Cuantificadores.
- Permite la generación de modelos.
- Es un proyecto de código abierto. Su código fuente puede ser obtenido desde `http://z3.codeplex.com`
- Puede manejar varios formatos de entradas (Simplify, SMT-LIB, Z3, Dimacs)
- Tiene una API extensiva (C/C++, .Net, OCaml, Python)

Z3 cuenta con una versión exclusiva encargada de mejorar la interfaz del razonador con python: *z3Py* [81]. Esta es la versión de z3 que hemos utilizado en este trabajo y sobre la cual nos centraremos.

Sorts

En z3, los tipos de datos que puede manejar (y por ende las teorías que puede resolver) son denominados **Sorts**. En z3Py cada sort esta implementado como una clase de *Python*. A continuación, podemos apreciar algunos de los sorts más conocidos que z3 brinda al usuario:

Tipo	Sort
Enteros	*IntSort*
Reales	*RealSort*
Booleanos	*BoolSort*
Arreglos	*ArraySort*
...	

Los *sorts* pueden ser utilizados para crear, a través de la función **Const**, constantes del tipo requerido. Cada constante tiene asociada un nombre único interno a z3.

```
>>> from z3 import *
>>> x = Const('x', IntSort())  # x es una constante entera con valor no definido
                               # y nombre 'x'
>>> b = Const('b', BoolSort()) # b es una constante booleana con valor no definido
                               # y nombre 'b'
```

El uso de constantes en z3 es el siguiente: se declaran de algún sort en particular (su valor en un principio se desconoce), se van acumulando restricciones sobre su valor y finalmente se le consulta al razonador si es posible que exista una constante con esas propiedades. De ser necesario, el razonador puede incluso dar un valor para la constante en cuestión.

Enteros en z3

Como ya mencionamos, en z3 los enteros se declaran como *IntSort*. z3 brinda dos funciones que simplifican la creación de constantes de tipo IntSort: *Int* e *Ints*. Estas funciones son equivalentes a declarar *Const(nombre, IntSort())* una o más veces.

```
>>> from z3 import *
# Creamos una constante 'x' de tipo IntSort
>>> x = Int('x') # Equivalente a Const('x', IntSort())
# Creamos 3 constantes, a, b y c, de tipo IntSort
>>> a, b, c = Ints("a b c") # Equivalente a: a = Const('a', IntSort())
                            # b = Const('b', IntSort())
                            # c = Const('c', IntSort())
```

El razonador es capaz de manejar los operadores usuales sobre enteros: $+, -, *, \backslash, \%$, etc. Las constantes enteras generadas por *Cons*, *Int* o *Ints* pueden ser utilizadas junto a los operadores para generar **expresiones** de z3. Crear expresiones nuevas de z3 es muy simple, se escriben iguales a cualquier expresión sobre enteros. El despacho dinámico de operadores de Python se encarga de todo el trabajo.

```
>>> from z3 import *
>>> x, y = Ints('x y')
>>> a = x + y # a contiene la expresión del razonador x + y
>>> b = x * y # b contiene la expresión del razonador x * y
>>> c = x / y # c contiene la expresión del razonador x / y
>>> d = x % y # d contiene la expresión del razonador x % y
```

Pueden aplicarse relaciones sobre las constantes y expresiones generando **fórmulas** en z3. Las relaciones soportadas sobre los enteros son las usuales: $<$, $>$, $\geq$, $\leq$, $=$, etc. Al igual que con las expresiones, para crear fórmulas se trabaja como si fueran valores concretos de Python; encargándose el despacho dinámico de operadores de llamar a la función interna correcta.

```
>>> from z3 import *
>>> x, y = Ints('x y')
>>> a = x < y  # a contiene la expresión del razonador x < y
>>> b = x >= y # b contiene la expresión del razonador x >= y
>>> c = x == y # c contiene la expresión del razonador x == y
```

Arreglos en z3

Como mencionamos anteriormente, el razonador es capaz de resolver fórmulas que involucren arreglos. Para z3, un arreglo es un mapeo de un *Sort A* a un *Sort B*, donde los elementos de sort *A* se usarán para indexar elementos de *sort B*. Son similares a los diccionarios que provee *Python*, aunque en estos, los índices y valores almacenados pueden ser de distintos tipos.

A la hora de crear una constante de tipo arreglo, necesitamos indicar el *sort* por el cual indexaremos y el sort de datos a almacenar. Existen dos formas de crear constantes de tipo arreglo: con la función *Const* y con la función *Array*.

```
>>> from z3 import *
# a1 será un mapeo int -> int
>>> a1 = Const('a1', ArraySort(IntSort(), IntSort()))
# a2 será un mapeo int -> int
>>> a2 = Array('a2', IntSort(), IntSort())
```

Existen dos operaciones fundamentales que podemos aplicar sobre arreglos para generar expresiones y fórmulas:

- *Select*.

  Esta función, toma como parámetro un arreglo y un índice. *Select* permite crear expresiones o fórmulas que involucren elementos almacenados en posiciones específicas de un arreglo. Por ejemplo, *Select(a, 2)* permite referenciar al elemento del arreglo "a" ubicado en la posición 2.

  ```
  >>> from z3 import *
  >>> a = Array('a', IntSort(), IntSort())
  >>> e = Select(a, 2) + 2 # e es la expresión a[2] + 2
  ```

- *Store*.

  Esta función, toma como parámetro un arreglo, un índice y un valor a almacenar. *Store* permite realizar afirmaciones sobre los valores almacenados en un arreglo. Por ejemplo, la expresión *Store(a, 1, 100)* afirma que el arreglo "a" contiene el valor 100 en la posición 1.

```
>>> from z3 import *
>>> a = Array('a', IntSort(), IntSort())
>>> e = Store(a, 0, 2)
```

Resolución y modelado

Todo el poder de los razonadores reside en la capacidad de resolver formulas de una teoría. z3 tiene el poder no sólo de decidir **satisfactibilidad** de fórmulas sino también de encontrar un **modelo** (es decir una asignación de valores a las variables) que hace verdadera la fórmula.

Para resolver una fórmula necesitamos crear una instancia de la clase *Solver*, agregar la formula al solver y posteriormente pedirle que verifique si es satisfactible mediante el método *check*

```
>>> from z3 import *
>>> x, y = Ints('x y')
>>> f = x < y  # a contiene la expresión del razonador x < y

>>> s = Solver() # Creamos una instancia de Solver
>>> s.add(f)     # Agregamos la fórmula al solver
>>> s.check()    # Verificamos si es sat o unsat
sat
```

Para generar un modelo que haga verdadera una fórmula basta con crear un *Solver*, verificar si la fórmula es satisfactible, y si lo es, posteriormente, llamar el método *model* de *Solver* para que z3 nos genere algún modelo.

```
>>> from z3 import *
>>> x = Int('x')
# Busquemos una raíz de x^2 - 1
>>> f = (x * x - 1 == 0)
>>> s = Solver()
>>> s.add(f)
>>> s.check() # sat
sat
>>> s.model() # Obtengamos el modelo
[x = 1] # z3 genera el modelo x <- 1
```

2.5. Ejecución Simbólica

Como vimos en la sección 2.1, uno de los problemas fundamentales del testing es la selección de los casos de test. Esta tarea generalmente requiere la elección de un **criterio de selección** y la posterior adopción de un conjunto (mínimo) de entradas que lo satisfagan.

En PEF, el criterio de selección utilizado para la generación de test es **según el flujo de control** (2.1.8). Este criterio busca recorrer todos [1] los caminos de ejecución de un programa. Siguiendo la naturaleza "automática" de la herramienta, la tarea de escoger los casos de test también debería serlo. PEF resuelve este problema con la ayuda de la **ejecución simbólica**. Esta técnica permite analizar un programa y determinar qué entradas causan la ejecución de cada camino del programa.

La ejecución simbólica [1], es una técnica intermedia entre el testing y la verificación formal de programas. Su autor, James C. King, la define como un testing "mejorado", donde en lugar de ejecutar un programa con valores de entrada concretos (casos de test), se lo ejecuta con **clases de entradas**. Una clase de entrada es un conjunto de valores. Las clases de entradas caracterizan los valores que puede tomar la entrada de un programa para seguir un camino específico. Esto permite que cada ejecución simbólica sea equivalente a un gran número de ejecuciones concretas. Como ejemplo, en el programa de la figura 2.13, las clases de entrada son los conjuntos $\{(a, b) \mid a < b\}$ y $\{(a, b) \mid a >= b\}$.

El conjunto de valores que representan las clases de entrada es determinado por la dependencia del flujo de control del programa sobre las entradas. Si el flujo de control del programa es totalmente independiente de las entradas (ej. figura 2.12), una sola clase de entrada bastará para analizar todos los posibles caminos del programa. Si por el contrario, el flujo de control depende de alguna de sus entrada (ej. figura 2.13), se generarán tantas clases de entrada como sea necesario para lograr que la ejecución simbólica ejercite los diferentes caminos. A menudo, la cantidad de clases de entrada necesarias para recorrer de manera exhaustiva todos los caminos suele

[1] No siempre es posible, como se explica en 2.1.8, la cantidad de caminos de ejecución suelen ser infinitos

ser infinito, sobre todo cuando hay bucles involucrados.

```
def f(a, b):
    c = a + b
    return c
```

Figura 2.12: 'f' suma dos naturales. El flujo del programa no depende de la entrada.

```
def min(a, b):
    if (a < b):
        return a
    else:
        return b
```

Figura 2.13: 'min' calcula el mínimo entre dos números naturales. El flujo del programa depende de las entradas.

Veamos la técnica en acción; al iniciar la ejecución simbólica de un programa, hay una única clase entrada y no contiene restricciones ($\{(...) \mid true\}$). A medida que avanza la ejecución, aparecen sentencias de control de flujo; en esos momentos la ejecución se divide en dos, una por cada camino del control, donde además cada una modificará su propia clase de entrada con las restricciones que el camino elegido impuso. Al finalizar la ejecución simbólica de una rama del programa, la clase de entrada contendrá el conjunto de los posibles valores para los cuales la función recorrerá el mismo camino si se la ejecuta concretamente, y cualquier asignación de valores que estén fuera del conjunto, recorrerán un camino diferente.

Luego de ejecutar simbólicamente un programa, obtenemos un conjunto de pares *(clase en entrada, resultado)* donde *resultado* es una fórmula sobre las variables de entrada, que describe el valor de retorno de la función.

2.5.1. Ejecución Simbólica en un Ambiente Controlado

Dado que la ejecución simbólica es una técnica aplicada a un lenguaje de programación, introduciremos un lenguaje simple sobre el cual haremos los cambios necesarios para ejecutar programas de manera simbólica. Utilizaremos una versión reducida de *Python* con las siguientes restricciones:

- El único **tipo de datos** que dispone son los **enteros**.
  - Las únicas **operaciones** disponibles son $+$, $-$ y $*$.
  - La única **relación** disponible es $\geq$.

- Las únicas **sentencias** disponibles son:
  - **if - : - else**. Dentro de la condición booleana sólo se puede verificar que $valor1 \geq valor2$.
  - **Asignación** de variables.
  - Definición y llamada a funciones.

Todos los lenguajes tienen su propia semántica de ejecución, en ella se describe cómo los objetos son representados por variables, cómo las sentencias manipulan objetos y cómo se controla el flujo de un programa. Podemos definir una nueva semántica, una "semántica simbólica", donde disponemos de **símbolos arbitrarios** que representan conjuntos de valores y pueden ser asignados a variables o utilizados para realizar operaciones. Estos símbolos serán llamados **entradas simbólicas** o **variables simbólicas**. Al adoptar una nueva semántica es necesario extender las definiciones de los operadores del lenguaje para aceptar y producir formulas sobre estos símbolos (fórmulas simbólicas).

Para ejecutar una función de manera simbólica no es necesario modificar ni la sintaxis ni su contenido, sólo basta con utilizar entradas simbólicas (símbolos) como parámetros. Por simplicidad, asumamos que las entradas simbólicas pertenecen al siguiente alfabeto $\{s_0, s_1, s_2, ...\}$. La ejecución simbólica, entonces, puede verse como una extensión natural de la ejecución concreta, computando la ejecución concreta como un caso especial (cuando los símbolos representan un singletón).

Las **clases de entrada** que genera como resultado la ejecución simbólica se irán construyendo a medida que el programa se ejecuta. Se agregará al concepto usual de **estado de un programa** una lista de expresiones, que llamaremos ***cc*** "condición de camino". *cc* es una lista de expresiones booleanas (restricciones) sobre los símbolos s_i. Estas restricciones son las que nos permiten ir refinando los conjuntos de valores que representan los símbolos a medida que se ejecuta simbólicamente el programa. Al finalizar la ejecución simbólica de un camino, la lista ***cc*** son las restricciones que definen una **clase de entrada**. A continuación mostramos una condición de camino de ejemplo. La misma expresa las restricciones adquiridas después de ejecutar las

sentencias *if* de la figura 2.14, línea 5:

$$[s_0 \geq 0, s_0 + 2 * s_1 \geq 0, \neg(s_1 \geq 10)]$$

```
1 def fun(a, b):
2     if a >= 0:
3         if a + 2 * b >= 0:
4             if not (b >= 10):
5                 pass
```

Figura 2.14: Ejemplo para ver la lista de condición de camino que se genera en la línea 5

Comencemos a analizar en más detalle los cambios que necesitamos introducir en nuestra semántica simbólica. Las reglas de evaluación para las expresiones aritméticas, usadas en una *asignación* o dentro de un *if*, deben extenderse para soportar valores simbólicos. En la ejecución normal las expresiones aritméticas siempre evalúan a enteros, ahora serán expresiones que involucran símbolos (s_i) y enteros, llamadas **fórmulas simbólicas**. Para este caso particular, las expresiones pueden verse como **polinomios** con coeficientes enteros, donde las variables polinomiales serán los símbolos s_i, y los operadores los provistos por el lenguaje $(+, -, *)$. En la figura 2.15 se muestra un ejemplo donde se generan fórmulas simbólicas. La función es ejecutada con la asignación simbólica $y \leftarrow s_0$ sobre el parámetro de entrada.

```
def f(y):
    # Supongamos y ← s0
    x = 1          # Asignación simple; polinomio de grado 0.
    return y * y + x  # Se retorna el polinomio s0^2 + 1.
```

Figura 2.15: Ejemplo de formula simbólica. La función computa el polinomio $y^2 + 1$

La ejecución simbólica de las **asignaciones** consiste en evaluar la parte derecha de la misma, reemplazando variables de lenguaje (si es que las hay) por sus respectivas fórmulas simbólicas. El resultado es otra fórmula (o un entero en el caso trivial) que es almacenado en la variable de la parte izquierda de la asignación. Ya que eventualmente las entradas del programa son asignadas a variables (y estas son entradas simbólicas), permitiremos que las variables puedan almacenar, a demás de enteros, fórmulas simbólicas.

La **conjunción** de los elemento de ***cc*** es la **propiedad** que las entradas deben satisfacer para seguir el **camino** asociado a la ejecución. A medida que el programa se ejecuta se presentan expresiones condicionales (**if**) que involucran símbolos. Es necesario, entonces, realizar suposiciones sobre estos símbolos para poder continuar la ejecución, ya sea por la rama verdadera o la falsa. Cada vez que se hace una de estas suposiciones, la misma es insertada al final de la cola ***cc***.

La ejecución simbólica de un **if** comienza igual que su contraparte concreta, se evalúa la expresión booleana asociada reemplazando las variables por sus valores. La condición, que llamaremos E e involucra enteros y variables simbólicas, puede reducirse en nuestro caso a una expresión de la forma $\widetilde{E} \geq 0$ (puesto que la única relación de la que disponemos es $\geq$). Entonces una y sólo una de las siguientes tiene que ser verdadera:

1. $\boldsymbol{cc \models E}$

   Si E se deduce de ***cc*** bajo las suposiciones actuales sobre los símbolos (i.e. la condición de camino), la condición del **if** es verdadera. En este caso la ejecución continua de manera normal por la rama del **then**. Decimos que en este caso que el flujo de control tomó una *decisión forzada*.

2. $\boldsymbol{cc \models \neg E}$

   Si $\neg E$ se deduce de ***cc*** bajo las suposiciones actuales sobre los símbolos (i.e. la condición de camino), la condición del **if** es falsa. La ejecución continua de manera normal por la rama del **else**. En este caso también decimos que el control del flujo del programa fue decidido de manera *forzada*.

3. $\neg(\ 1) \wedge \neg(2)$

   Este es el caso más interesante. Con las restricciones actuales sobre los símbolos en ***cc***, la condición del **if** puede ser tanto verdadera como falsa, y el control del flujo del programa *no está forzado* por el estado. La ejecución simbólica procede, en este caso, dividiendo la ejecución actual en dos. Estas nuevas ejecuciones mantienen el estado anterior e intentan continuar por los caminos del **then** y del **else**. Si elegimos el camino del **then** se asume que E es satisfactible, y se inserta al final de *cc* la fórmula E. De manera similar si elegimos el

camino del **else**, se inserta al final de *cc* la fórmula $\neg E$.

A esta altura debería ser claro por que denominamos a *cc* como *condición de camino*. La conjunción de sus elementos determina un único flujo de control a través del programa. Cada división de la ejecución en un **if**, es decir por una decisión no forzada, agrega una nueva restricción sobre los símbolos. Las decisiones forzadas no agregan nada a *cc* ya que no se requiere hacer ninguna suposición. Inicialmente *cc* es la lista vacía.

Una vez que se termina la ejecución simbólica por uno de los caminos, utilizando la información almacenada en *cc* podemos caracterizar la clase de entrada que nos lleva por el camino e inclusive elegir un representante concreto para cada variable simbólica (ej. de la condición de camino $[s_0 \geq 0]$ podemos elegir como representante de s_0 al 0). Notar que nunca sucede que $cc \models False$ (La condición de camino no es satisfactible) por la manera en que *cc* es construida (sólo se agregan elementos a *cc* cuando ambos caminos de un *if* son posibles).

Las llamadas a funciones que suceden durante una ejecución simbólica son llevadas a cabo de manera idéntica a una ejecución concreta: se copian los valores en las variables de la definición de la función a ser llamada, y se ejecuta el código dentro de un nuevo ambiente. Al finalizar la función, el valor de retorno es devuelto al cuerpo de la función que hizo el primer llamado, y la ejecución continúan con la próxima instrucción. Por lo tanto, la ejecución simbólica sigue los caminos por las llamadas a otras funciones. Puede pensarse, simplificadamente, que el código de las funciones se "pegan" en una gran función, la cual luego se ejecuta simbólicamente.

A continuación mostraremos un ejemplo ilustrador para un caso sencillo:

```
1 def duplicar_valor (a):
2     result = 2 * a
3     if result == 12:
4         fail()
5     return result
```

Figura 2.16: La función duplica_valor genera un error si se la llama con 6; de lo contrario retorna el doble del valor ingresado

Si ejecutamos simbólicamente la función *duplicar_ valor* con una variable simbólica s_0 de tipo numérica, al llegar al *if* de la línea 3, tendremos la expresión $2 * a = 12$,

la cual no se puede deducir verdadera o falsa ya que ***cc*** está vacío. Estamos en el caso interesante donde ambos caminos son posibles. A partir de aquí la ejecución se divide en dos, una continúa por la rama del **then** con ***cc*** como $[\boldsymbol{s_0 * 2 = 12}]$, y se ejecuta la línea 4, donde aparece una falla al ejecutar $fail()$; mientras que la otra continúa por la rama del **else** donde ***cc*** es $[s_0 * 2 \neq 12]$ y se ejecuta la línea 5, retornando la función el valor de la variable *result*. Si luego deseamos encontrar un valor concreto para realizar tests que ejerciten ambos caminos de la función, basta con encontrar un modelo de las restricciones de ***cc*** para cada camino; tendríamos así que para ejecutar la rama del **then** necesitamos que $s_0 = 6$, mientras que para la otra, $s_0 \neq 6$, cualquier valor de s_0 distinto de 6 es válido.

2.5.2. Implementación de una Ejecución Simbólica

Tal y como mencionamos anteriormente, para implementar ejecución simbólica en un lenguaje de programación particular, sólo es necesario modificar su *semántica de ejecución*. Existen varios enfoques para lograr este objetivo:

- Reimplementar el intérprete/compilador del lenguaje.

  La opción más natural para implementar una nueva semántica, por lo menos para los autores, es reimplementar (o tan sólo modificar) el intérprete/compilador del lenguaje. Se vuelve trivial dónde se tiene que introducir la nueva semántica, aunque generalmente implica muchas horas/hombre tanto para su desarrollo como para su mantención. Es el método más lento de todos. Éste enfoque ha sido utilizado con éxito en herramientas como Java PathFinder [16] y DART [17].

- Traducir el código.

  Se traduce el programas del lenguaje original a un nuevo lenguaje, el cual ya posee la capacidad de ejecutarse simbólicamente. Este enfoque evita la implementación de nuevas semánticas de ejecución para el lenguaje objetivo. Todo el trabajo recae en la programación del traductor. Cabe notar que la mayoría de las veces una traducción automática es imposible, dada las diferencias en la arquitectura y paradigmas de los lenguajes en cuestión.

- Instrumentar el código.

  Se ejecuta una copia del código, donde se insertan sentencias en lugares estratégicos, generalmente antes de entrar en un control de flujo y en el comienzo de cada rama del mismo, con el objetivo de realizar las deducciones y cambios necesarios sobre las variables durante la ejecución. El problema de este enfoque es que es necesario tener completo acceso al código fuente, al ejecutar un código distinto al original, puede ser mas difícil encontrar y corregir los errores, y puede ser complicado implementarlo para algunos lenguajes poco flexibles.

- Utilizar una arquitectura de pares.

  La arquitectura de pares explota las características de los lenguajes puramente orientados a objetos con despacho dinámico de operadores primitivos. Se modifica explícitamente la semántica de ejecución creando objetos simbólicos como clases en el mismo lenguaje. Este enfoque requiere menos esfuerzo que los otros métodos y brinda un mayor control sobre la ejecución simbólica que la traducción o la instrumentación de código (cuando son posibles de implementar).

El enfoque más apropiado a implementar dependerá de las propiedades del lenguaje objetivo, del tiempo que pretendamos invertir y del grado de complejidad con el que estemos dispuestos a lidiar.

En este trabajo utilizamos la arquitectura de pares para realizar ejecución simbólica, la cual extendimos para dar soporte a objetos de cualquier clase, incluyendo a las creados por el usuario, superando así la mayor limitación por la cual esta técnica era considerada inútil para usos prácticos [5].

2.5.3. Los Razonadores en la Ejecución Simbólica

Hasta ahora no explicamos de qué manera se llevan a cabo las deducciones sobre las expresiones y las condiciones de camino. Las mismas son realizadas por un razonador o SMT-Solver, que dado un conjunto de fórmulas, tienen la capacidad de indicar si la conjunción es satisfactible, y en caso de serlo, pueden dar una asig-

nación concreta de las variables. Entonces, la versatilidad de la ejecución simbólica esta fuertemente relacionada con el razonador que se utilice.

2.5.4. Árbol de ejecución simbólica

Durante la ejecución simbólica de un programa es posible ir creando un **árbol de ejecución** que caracteriza cada uno de los caminos que la técnica recorrió en el programa ejecutado. Se asocia un nodo a cada sentencia ejecutada (en este caso usamos el número de línea como nombre) y una arista dirigida por cada transición entre sentencias. Cuando se produce una división por un **if**, del nodo asociado salen dos aristas:

- Una con nombre "T", representando la elección del camino **then**.
- Otra con nombre "F", representando la elección del camino **else**.

A demás, cada nodo tiene asociado el estado del programa en ese punto (*cc*, linea a ejecutar, mapeo de variables). El árbol de ejecución para la función *duplicar_valor* de la figura 2.16 se muestra en la figura 2.17.

Los árboles generados por la ejecución simbólica tiene propiedades interesantes:

1. Para cada hoja del árbol (correspondiente a un camino completo de ejecución) existe alguna entrada concreta, que al ejecutar el programa de manera concreta con esa entrada se sigue el mismo camino. Esto es equivalente a decir que la conjunción de la *cc* nunca se hace $False$.

2. Dos hojas diferentes tienen diferentes condiciones de camino y más aún, vale que $\neg(cc_1 \wedge cc_2)$. Los dos caminos parten de una misma raíz (el comienzo de la ejecución simbólica), por lo que hay un único nodo a partir del cual los dos caminos divergen. En ese nodo se agrega la condición q a cc_1, mientras que $\neg q$ se agrega a cc_2.

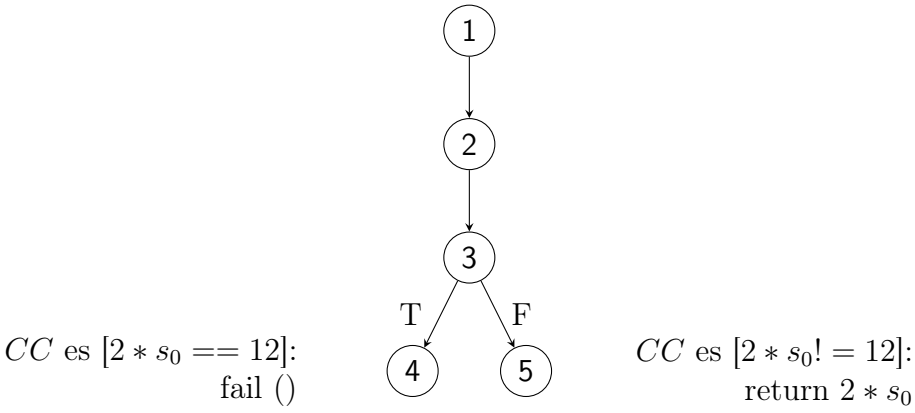


Figura 2.17: Árbol de ejecución para el programa duplicar_valor(s_0) de la figura 2.16

2.5.5. Conmutatividad entre semánticas de ejecución

Definamos primero dos conceptos, "instanciación de una variable simbólica" e "instanciación de un resultado de la ejecución simbólica". Entendemos por **instanciación de una variable simbólica** al hecho de reemplazar la variable por un valor concreto. En el caso de la **instanciación del resultado simbólico** hay dos formas equivalentes de definirla:

- Sustituir en las variables del programa y *cc* de cada hoja del árbol de ejecución simbólica los símbolos s_i por sus valores de instanciación j_i; y finalmente utilizar como valor de resultado la hoja cuya condición de camino haya quedado verdadera, que será a lo sumo una.

- Buscar la clase de entrada que contenga la entrada concreta formada por los j_i y luego instanciar s_i por j_i sobre el resultado asociado a la clase.

La semántica de ejecución simbólica que definimos para nuestro lenguaje reducido satisface una interesante **propiedad de conmutatividad**. Si instanciamos entradas simbólicas s_i por enteros j_i y ejecutamos concretamente un programa, el resultado será el mismo que al de ejecutar simbólicamente el programa y luego "instanciar el resultado simbólico". Esta propiedad conmutativa puede apreciarse en la figura 2.18, donde:

- P es un programa,

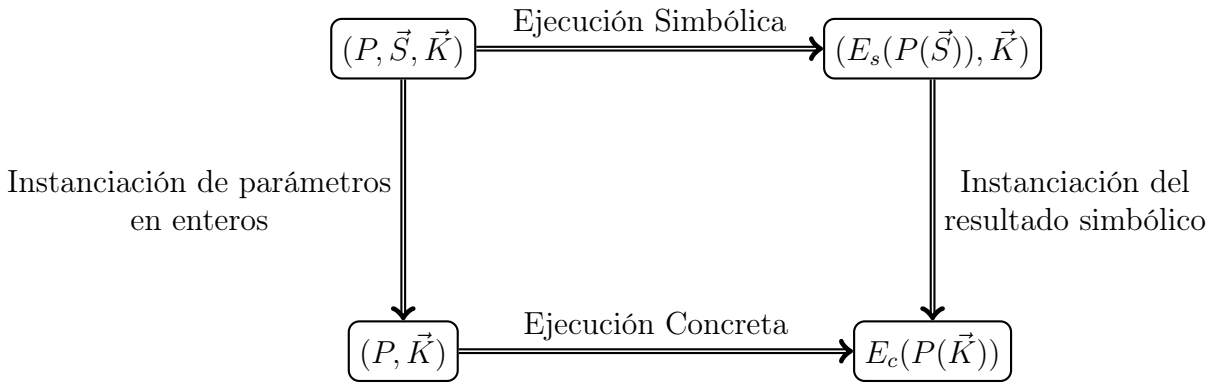


Figura 2.18: Diagrama de conmutatividad de semánticas

- $\vec{S}$ es un vector de entradas simbólica,
- $\vec{K}$ es un vector de entradas enteras,
- $E_s(P(\vec{S}))$ es el resultado de ejecutar simbólicamente el programa P
- $E_c(P(\vec{K}))$ es el resultado de ejecutar concretamente P con los parámetros $\vec{K}$.

Notemos que esta relación de conmutatividad es por lo que la ejecución simbólica es interesante. La ejecución simbólica captura exactamente los mismos efectos que la ejecución concreta. No es una ejecución alternativa cualquiera sino una extensión natural de su definición concreta.

2.5.6. El problema de los bucles

La mayor limitación de la ejecución simbólica reside en los bucles del programa, ya que un solo bucle puede contener infinitos caminos de ejecución simbólica. Se puede entender esto fácilmente pensando a los bucles como una serie (usualmente infinita) de if's anidados (son equivalentes).

Más formalmente, la ejecución simbólica de un **while** es muy similar a la de un **if**. Se evalúa la expresión booleana asociada, que llamaremos E, reemplazando las variables por sus valores. Luego, al igual que en un **if**, una y sólo una de las siguientes afirmaciones se hace verdadera:

1. $\boldsymbol{cc} \models \boldsymbol{E}$

   Si E se deduce de $\boldsymbol{cc}$, la condición del **while** es verdadera. Al igual que en la

ejecución concreta, se continua con el cuerpo del **while** y al finalizarlo se re-ejecuta simbólicamente el **while**. Esta se considera como una *decisión forzada*.

2. $\boldsymbol{cc} \models \neg \boldsymbol{E}$

   Si $\neg E$ se deduce de $\boldsymbol{cc}$, la condición del **while** es falsa. La ejecución continua de manera normal, sin ejecutar el cuerpo del **while**. En este caso también decimos que el control del flujo del programa fue decidido de manera *forzada*.

3. $\neg(\ 1) \wedge \neg(2)$

   Al no poder deducirse ni E ni $\neg E$ de $\boldsymbol{cc}$, los dos caminos del **while** (ejecutar o no el cuerpo) son posibles y consideramos a esta una decisión *no forzada*. La ejecución simbólica, al igual que en el **if**, se divide en dos. La primera agrega a $\boldsymbol{cc}$ la restricción E y ejecuta el cuerpo del **while** para luego volver a ejecutar la sentencia del **while**. La segunda división de la ejecución simbólica agrega a $\boldsymbol{cc}$ la restricción $\neg E$ y continúa la ejecución salteando el cuerpo del **while**.

```
1  def f(x):
2      i = 0
3      while i < x:
4          i += 1
5      return i
```

Figura 2.19: Un *while* dependiendo de una entrada es un gran problema para la ejecución simbólica

.

Supongamos que queremos ejecutar de manera simbólica la función $\boldsymbol{f(s_0)}$ de la figura 2.19. La ejecución simbólica comienza sin dividirse hasta la línea 3. En esa línea, el programa se pregunta si $0 < s_0$, pero como $\boldsymbol{cc}$ está vacío la guarda del **while** puede ser tanto verdadera como falsa y la ejecución se divide en dos partes. Una parte continua con $\boldsymbol{cc} = [\boldsymbol{\neg(0 < s_0)}]$ e inmediatamente retorna con valor 0; la otra, con $\boldsymbol{cc} = [\boldsymbol{0 < s_0}]$, ejecuta el incremento de la variable i (quedando $i = 1$) y vuelve a ejecutar el **while**. Se procede, entonces, a evaluar de nuevo la guarda, $1 < s_0$, y nuevamente no se puede deducir si es verdadera o falsa dividiendo la ejecución en dos. Una, continuando con $\boldsymbol{cc} = [\boldsymbol{0 < s_0, \neg(1 < s_0)}]$ e inmediatamente retornando el valor 1; y otra, con $\boldsymbol{cc} = [\boldsymbol{0 < s_0, 1 < s_0}]$, que ejecuta el incremento de la variable

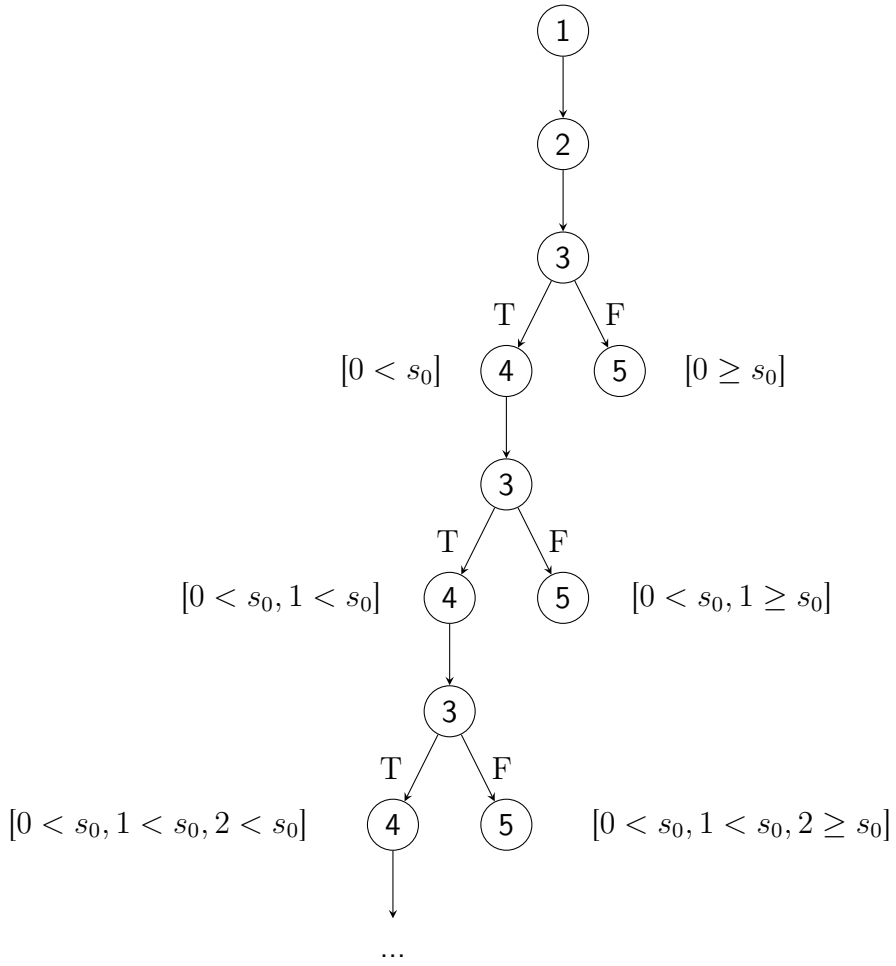


Figura 2.20: Al ejecutar con s_0 la función f de la figura 2.19 el árbol de ejecución obtenido es infinito.

i (ahora $i = 2$) y vuelve a ejecutar el **while**. Como el lector puede haber notado, esta ejecución simbólica es infinita. Cada vez que llegamos al **while** la condición de camino es de la forma $[0 < s_0, 1 < s_0, ..., j < s_0]$ mientras que la guarda evalúa a $j + 1 < s_0$, por lo que siempre puede ser tanto verdadera como falsa. El árbol de ejecución simbólica de este ejemplo puede apreciarse en la figura 2.20.

Se han desarrollado varias heurísticas [64] [65] [66] [67] para tratar con este problema, como por ejemplo deducir el invariante del ciclo para obtener expresiones que relacionen los valores de las variables con la cantidad de veces que se recorre el bucle, pero el problema de la explosión de caminos siempre está presente. Por lo tanto, es necesario limitar la profundidad de la exploración de los caminos para asegurar terminación en la ejecución simbólica. Limitar la profundidad de exploración es equivalente a limitar la cantidad de elementos que puede contener cc, la cantidad

de niveles alcanzados en el árbol de ejecución o, lo que es equivalente, la cantidad máxima de decisiones no forzadas que el programa puede tomar. Cuando el límite de profundidad es alcanzado durante una ejecución, la misma se interrumpe, y el camino se descarta.

2.5.7. Sistema de Contratos y Ejecución Simbólica

Como ya fue mencionado en la sección 2.1.1, una de las formas usuales de especificar formalmente el comportamiento de un programa es mediante un **sistema de contratos**. La ejecución simbólica en PEF puede verificar la validez de los contratos, o ser asistida por estos, obteniendo una detección automática de caminos erróneos o mejoras de performance, al reducir el espacio de búsqueda.

Los contratos de **precondición** se encargan de expresar las propiedades que deben ser ciertas **antes** de ejecutar el programa o función objetivo. Estos contratos cumplen dos funciones:

1. Al comienzo de la ejecución simbólica, la precondición de la función objetivo se **asume verdadera**. De esta manera las entradas que no la satisfacen son descartadas, logrando podar de manera anticipada el árbol de ejecución. En otras palabras, son usados para restringir las ramas a explorar.

2. Cada vez que se ejecuta una función, los contratos de precondiciones son utilizados para determinar si los parámetros con los que fue llamada son válidos. Si la entrada de la función no satisface el contrato decimos que se produjo una **falla de contrato**.

Cuando la aplicación de una precondición causa que algunas instrucciones nunca sean ejecutadas, éstas pueden considerarse código muerto respecto a la especificación que la tiene en cuenta como verdadera.

Por otra parte, los contratos de **postcondición** expresan las propiedades que deben ser ciertas luego de ejecutar una función/programa. Cada vez que una función retorna un valor, se verifica que la postcondición se satisfaga en ese estado. Al igual que en los contratos de precondición, si el estado no logra satisfacer la postcondición se genera una falla de contrato.

La ejecución simbólica junto a los contratos permiten la automatización de los oráculos, diferenciando sin intervención del usuario resultados correctos de erróneos.

Veamos ahora el ejemplo de un pequeño programa con contratos (figura 2.21), y observemos los diferentes árboles de ejecución que se producen cuando son utilizados. Asumiremos que los contratos se insertan dentro de la documentación de la función y que las postcondiciones son capaces de hacer referencia al valor de retorno de la función con el nombre *returnv*. Un contrato debe ser una fórmula válida del lenguaje.

```
1  def f(x):
2      """
3      precondition: x >= 0
4      postcondition: returnv > x
5      """
6      if x < 0:
7          return x * 2
8      else:
9          if x == 0:
10             return 0
11         else:
12             return x * x
```

Figura 2.21: Una función con contratos de precondición y postcondición en su docstring.

Si no utilizamos contratos, el árbol de ejecución obtenido es el usual, y se muestra en la figura 2.22. Si por el contrario utilizamos los contratos, se asumirá como verdadero $x \geq 0$; al ejecutarse simbólicamente la línea 6, debido a lo que acabamos de asumir, no se entrará a la rama del *then* y nos moveremos directamente al *else*. Al ejecutar el return de la línea 10, automáticamente es evaluada la postcondición $returnv > x$. Reemplazando las variables por sus valores obtenemos la expresión falsa $0 > 0$; la ejecución simbólica se encarga en este momento de "marcar" esta hoja del árbol como una **falla de contrato**. El árbol de ejecución para este caso puede observarse en la figura 2.23.

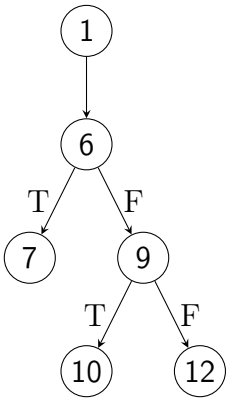


Figura 2.22: Árbol de ejecución simbólica generado al ejecutar la función f de la figura 2.21 **sin utilizar el sistema de contratos**.

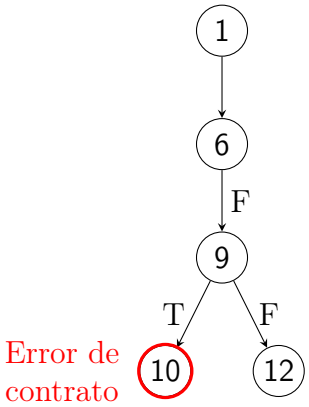


Figura 2.23: Árbol de ejecución simbólica generado al ejecutar la función f de la figura 2.21 **utilizando el sistema de contratos**.

Teniendo una clara idea de las ventajas de utilizar contratos, es hora de analizar cómo incluirlos a la ejecución simbólica. Haremos un análisis intuitivo y no muy profundo de los cambios que se deben introducir. La implementación de los contratos en PEF se puede encontrar en la sección 3.3, página 79.

Para poder manejar contratos, deben introducirse tres modificaciones esenciales al algoritmo genérico de ejecución simbólica:

1. Al comenzar la ejecución simbólica de la función/programa objetivo, en vez de hacerlo con una lista de *condición de camino* vacía, se lo hace con una lista con la precondición como único elemento, es decir $\boldsymbol{cc} = [\boldsymbol{prec}]$ (donde *prec* es la precondición con las variables de entrada reemplazadas por sus respectivos valores).

2. Antes de explorar simbólicamente cualquier función, si la misma tiene contratos de precondición declarados, se debe corroborar si el estado actual satisface tales condiciones. Sea *prec* la precondición con las variables de entrada reemplazadas por sus respectivos valores:

   - Si $\boldsymbol{cc} \models \boldsymbol{prec}$, la precondición se deduce de la condición de camino. La ejecución simbólica continúa habiendo asegurado que la función fue llamada con parámetros válidos.

- Si $\boldsymbol{cc \models \neg prec}$, ninguna asignación posible a las variables simbólicas satisfará la precondición. Se genera un error de contrato y se interrumpe la ejecución simbólica.
- Finalmente, si no ocurre ninguno de los anteriores, las variables simbólicas pueden tanto satisfacer como no la precondición. La ejecución simbólica se divide en dos, por un lado agregando la precondición a *cc* y continuando normalmente; por el otro, agregando $\neg prec$ a *cc*, generando un error de contrato e interrumpiendo la ejecución.

3. Se debe modificar la semántica de ejecución de funciones para evaluar la postcondición. Cuando la función finaliza, se verifica que la expresión de la postcondición se deduzca de la condición de camino, reemplazando *returnv* por el valor de retorno de la función.
   - Si $\boldsymbol{cc \models post}$, la postcondición se deduce de la condición de camino. La ejecución simbólica continúa habiendo asegurado que la función retornó un valor adecuado.
   - Si $\boldsymbol{cc \models \neg post}$, ninguna asignación posible de las variables simbólicas satisfará la postcondición. Se genera un error de contrato y se interrumpe la ejecución simbólica.
   - Finalmente, si ninguno de los anteriores sucede, las variables simbólicas pueden tanto satisfacer como no la postcondición. La ejecución simbólica se divide en dos, por un lado agregando *post* a *cc* y continuando normalmente; por el otro, agregando $\neg post$ a *cc* y generando un error de contrato.

Volvamos a ejecutar y analizar el programa de la figura 2.21 para observar los nuevos cambios que introdujimos. A causa del punto 1, comenzamos a ejecutar simbólicamente f con s_0 como parámetro y $cc = [s_0 \geq 0]$. Inmediatamente, por el punto 2, se verifica que se cumple la precondición; para la primer ejecución, al haber sido agregada la precondición a *cc*, $cc \models precondition$, por lo que la ejecución continúa de manera usual. Al ejecutar simbólicamente el *if* de la la línea 6, debido a que

$[s_0 \geq 0] \models \neg(s_0 < 0)$, el flujo se dirige directamente al *else*. En la próxima instrucción (línea 9), la ejecución se divide en 2. La primera división asume la condición cierta, agregando a *cc* $s_0 = 0$ y ejecutando el *return*. A la hora de retornar el valor, por el punto 3, comprobamos la postcondición; pero $[s_0 \geq 0, s_0 == 0] \models \neg(0 > s_0)$, por lo que esta rama genera un **error de contrato**. En la otra división, *cc* adopta el valor $[s_0 \geq 0, s_0 \neq 0]$ y ejecutamos el return de la línea 12. Nuevamente, volvemos a intentar validar la postcondición, donde para este caso que tenemos $[s_0 \geq 0, s_0 \neq 0] \models s_0 * 2 > s_0$ la postcondición es válida. El árbol de ejecución obtenido por esta ejecución es el mostrado en la figura 2.23.

Capítulo 3

Trabajo

La herramienta desarrollada en este trabajo, **Python Error Finder** o simplemente **PEF**, es un poderoso detector automático de errores para programas escritos en Python 3. Su componente principal, el motor de ejecución simbólica, se encarga de ejecutar en forma reiterada el programa objetivo con objetos intermediarios como argumentos. Los objetos intermediarios se encargan de interceptar, registrar e informar al razonador de todas las operaciones en las que intervienen. La información recogida por los objetos intermediarios durante la ejecución es utilizada para construir nuevos objetos que explorarán las sucesivas ejecuciones y caracterizarán los caminos recorridos.

Nuestra herramienta utiliza un sistema de contratos para verificar que cada uno de los pares *(entrada, salida)*, generados por el motor de ejecución simbólica, cumpla con las restricciones impuestas por el usuario. De esta manera se logra automatizar no sólo la exploración de caminos sino también los oráculos que validan los mismos.

Para el desarrollo de PEF se utilizó el módulo para Python de Z3 [81], un razonador muy poderoso para enteros y arreglos de enteros. Otros tipos de datos no están implementados de forma nativa en Z3, y aunque es posible extenderlo, el esfuerzo que eso implica es demasiado y fuera del alcance de este trabajo. Por esa razón, el soporte de tipos en PEF al momento de escribir este trabajo es limitado: enteros, booleanos, listas de enteros, strings y clases definidas por el usuario[1].

[1]Para que una clase definidas por el usuario sea soportada de manera completa, ésta debe estar compuesta por objetos de tipo soportado.

Este capítulo se organiza en 4 secciones. En la sección 3.1 mostramos la forma en que realizamos la ejecución simbólica en PEF. En la sección 3.2 introducimos los objetos intermediarios junto con detalles de implementación y algunas propiedades de posible interés para el lector. Posteriormente, en 3.3, introduciremos el sistema de contratos que ha sido desarrollado para la herramienta. En la sección 3.4 contamos cómo logramos extender la técnica de arquitectura de pares, donde agregamos la habilidad de crear objetos intermediarios sobre clases definidas por el usuario de forma recursiva. Luego, en la sección 3.5 mencionaremos algunos retoques que tuvimos que realizar para lidiar con algunos aspectos internos de Python que generaban problemas con PEF. Finalmente, en la sección 3.6 mostraremos un ejemplo integrador utilizando todos los conceptos desarrollados a lo largo de este capítulo.

3.1. Ejecución simbólica en PEF

Si bien es posible implementar la técnica de ejecución simbólica de muchas formas, PEF utiliza el enfoque de *arquitectura de pares* [2]. En este enfoque, la ejecución simbólica es realizada por un **Motor de Ejecución Simbólica** (MES) en forma de librería. El MES coordina muchos componentes de PEF para poder llevar a cabo la ejecución simbólica de una función o método objetivo. En la figura 3.1 podemos apreciar la arquitectura interna de PEF y el trabajo central que realiza el MES.

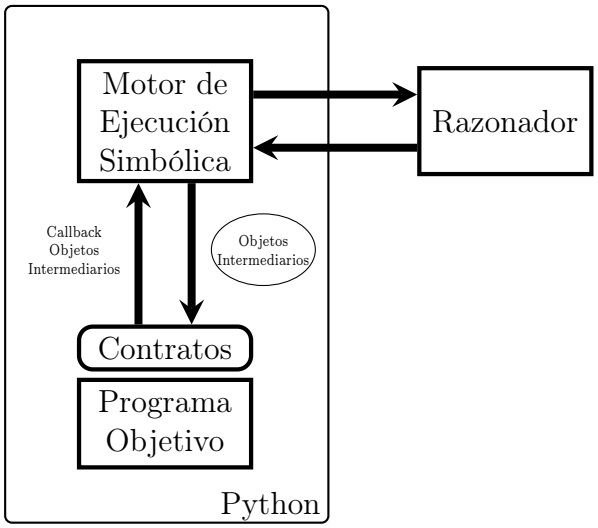


Figura 3.1: Arquitectura de PEF.

Los valores simbólicos que PEF utiliza son llamados *objetos intermediarios* y se explicarán en detalle en la sección 3.2. El MES recibe notificaciones de estos *objetos intermediarios* y luego traduce e informa las restricciones al razonador. Cuando una sentencia de control de flujo necesita el valor de verdad de un objeto intermediario (llamada a __bool__), el motor consulta al razonador para resolver el valor de verdad y devolverlo al control de flujo que lo requería. Cuando el objeto intermediario pueda tomar los dos valores de verdad, el motor se encarga de escoger uno para continuar la ejecución, dejando al otro para una ejecución futura. Cada vez que la ejecución simbólica se encuentra con llamadas a funciones o métodos, los mismos son ejecutados simbólicamente y su valor de retorno (generalmente simbólico) es utilizado para continuar la ejecución original.

Existen algunas diferencias relevantes entre la ejecución simbólica introducida en 2.5.1 y la implementada en PEF. Al momento de escribir este documento, PEF es puramente secuencial y por lo tanto la ejecución nunca se divide en threads; en su lugar, el programa se ejecuta en reiteradas oportunidades recorriendo siempre caminos diferentes. Cada vez que hay que tomar una decisión no forzada, PEF escoge uno de los posibles caminos para continuar el recorrido y programa una futura ejecución para recorrer el otro camino. Esta diferencia hace que a la herramienta deba usar dos listas (pilas), en vez de una, para realizar la exploración del programa objetivo:

- **_pathcondition**

  Es la **condición de camino**. La i-ésima posición indica las restricciones que surgieron al elegir una rama en la i-ésima bifurcación no forzada. La rama elegida está indicada por _path[i]. Contiene las restricciones a ser usadas por el razonador.

- **_path**

  Contiene los booleanos que indican las decisiones que fueron tomadas durante la ejecución. _path[i] indica el valor de verdad asignado a la i-ésima decisión no forzada. Cada ejecución del programa objetivo genera un _path único. PEF utiliza la lista _path para programar los recorridos del programa.

Las variables *_path* y *_path_condition* son **globales**.

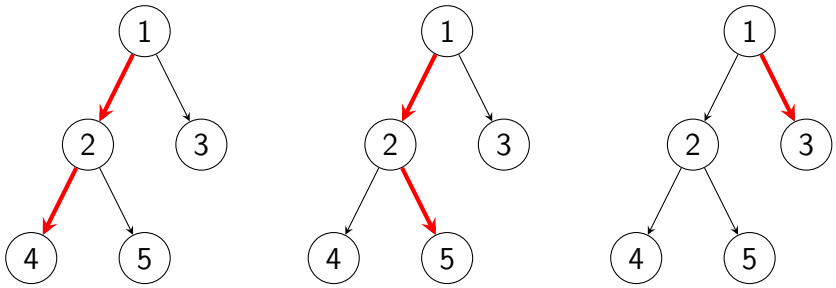


Figura 3.2: De izquierda a derecha, las diferentes ejecuciones de PEF que van explorando el árbol de ejecución.

La ejecución simbólica de un programa objetivo se realiza llamando al método **explore** (figura 3.3), pasando como argumentos el programa y las entradas simbólicas. Cada iteración dentro del bucle principal recorre un nuevo camino del programa objetivo y almacena el valor de retorno o excepción generada. Se obtiene un recorrido DFS (2.3) de las ramas del programa objetivo, empezando siempre por la rama verdadera. El recorrido DFS obtenido puede apreciarse en la figura 3.2. PEF utiliza la información almacenada en _path, generada durante la ejecución simbólica, para recrear el tramo inicial del nuevo camino a recorrer. Al comienzo de cada iteración la cola *_pathcondition* es vaciada; al final, el posfijo de booleanos falsos en *_path* es removido, y el último booleano verdadero es invertido a falso. Esto es lo que permite recorrer también las ramas falsas de los flujos de control. Cuando *_path* queda vacío, todas las ramas explorables fueron visitadas y la exploración termina. Las ramas explorables son todas las ramas módulo la profundidad de búsqueda, donde además el razonador pueda decidir la satisfactilidad de las expresiones involucradas. PEF informa al usuario sobre las ramas que son ignoradas durante la exploración.

Como mencionamos en 2.5.6, la ejecución simbólica necesita ser modificada para poder asegurar terminación. En PEF este problema se solucionó limitando la profundidad de exploración durante la ejecución simbólica. La cota por defecto fue tomado de “A Peer Architecture for Lighweight Symbolic Execution” [2], pero puede cambiarse fácilmente si es requerido. El valor por defecto resultó en la práctica ser suficiente para los programas de ejemplo que han verificado los autores. Más información sobre estas mediciones será presentado en 5.4.

La función *explore* de la figura 3.3 está implementada como un generador, donde por cada rama recorrida se retorna una tupla con:

- Argumentos posicionales con los que se llamaron al programa objetivo (args)
- Argumentos no posicionales con los que se llamaron al programa objetivo (kwargs)
- Resultado
- Excepción generada
- Escrituras al *stdout* (llamadas a *print*())
- El *_path* que caracteriza el camino recorrido
- *_pathcondition* con las restricciones producidas a lo largo del recorrido.

La información en *_path* y *_pathcondition* puede usarse para generar informes y/o gráficos para la visualización de los caminos recorridos.

```
def explore(function, args, kwargs):
    """
    Explora todos los caminos alcanzables de la funcion 'function' con los
    argumentos 'args' y 'kwargs'.
    """
    _path = []
    have_paths_to_explore = True
    while have_paths_to_explore:
        _pathcondition = []
        result, error = None, None
        try:
            result = function(*args, **kwargs)
        except Exception as e:
            error = e
        finally:
            # Concretizamos objetos intermediarios
            cargs, ckwargs = concretize_args(pargs, pkwargs)
            result = concretize_result(result)
            yield (cargs, pkargs, result, error, _path, _pathcondition)
        while len(_path) > 0 and not _path[-1]:
            path.pop()
        if not _path:
            have_paths_to_explore = False
        else:
            _path[-1] = False
```

Figura 3.3: Pseudocódigo de la función de exploración de caminos *explore*.

3.2. Objetos Intermediarios

Como se explicó en 2.2,en *Python* todos los operadores (+, >, <, and, etc) se traducen a llamadas a métodos sobre los objetos que intervienen en la operación. Por ejemplo, '==' se traduce a una llamada al método __eq__ (o su alternativo __req__) con los objetos que se comparan como parámetros. Podemos suponer entonces, que toda la funcionalidad de una clase, sin importar si es builtin o definida por el usuario, está dada sólo por sus atributos y métodos. Esta propiedad del lenguaje, conocida como Duck Typing, es explicada en la sección 2.2.2.

Los objetos intermediarios son la base de la ejecución simbólica en PEF, de hecho, son los valores simbólicos. Se pueden pensar como "espías", que se comportan como una instancia de otra clase, emulando su comportamiento y observando todas las operaciones que lo acceden o lo modifican. El razonador usa la información recogida por nuestros objetos intermediarios para decidir el flujo de control a tomar en el programa objetivo.

Supongamos que definimos una clase **MiString** que tenga los mismos atributos y los mismos métodos que el builtin **str**, con igual semántica pero una implementación posiblemente diferente. Dado que MiString y str presentan la misma interfase al exterior, los objetos de las clases serían prácticamente[2] indistinguibles para *Python* e inclusive hasta intercambiables. Esta es la idea que esta detrás del concepto de los *objetos intermediarios*.

Sea p una instancia de la clase P y q una instancia de una clase Q, decimos que q es un objeto intermediario de P si:

1. P y Q presentan la misma interfase externa.

2. Exceptuando la validación de tipos e ids, a la hora de ejecutar un programa, p y q son **indistinguibles**.

3. q puede representar y emular tanto instancias concretas de P como también un **conjunto de instancias**.

[2]Con excepción de los operadores builtins *type* e *isinstance*.

4. q se comunica con un **razonador** y éste registra todas las operaciones en las cuales está involucrado q y le son relevantes.

Es importante remarcar el punto 3. Los objetos intermediarios pueden representar conjuntos de valores por compresión, estos conjuntos son determinados por las fórmulas presentes en el razonador. Por ejemplo, sea i un objeto intermediario de enteros, si en el razonador existe la restricción $i > 0$, entonces i representa a todos los enteros mayores a cero. Las restricciones sobre los valores se van adquiriendo y refinando a medida que se ejecuta el programa (punto 4).

Debido a que **los objetos intermediarios son nuestros valores simbólicos**, necesitamos tenerlos para todos los tipos de dato de Python. Hemos implementado clases intermediarias para algunos tipos básicos, y diseñamos un algoritmo (3.4) que construye clases intermediarias a partir de clases definidas por el usuario.

Clase	Clase Intermediaras
int	IntProxy (3.2.3)
bool	BoolProxy (3.2.4)
list	ListProxy (3.2.5)
slice	SliceProxy (3.2.6)
str	StringProxy (3.2.7)

Lo que resta de esta sección está dividida de la siguiente manera: en primer lugar, en 3.2.1, explicaremos una interesante propiedad de los objetos intermediarios, la *indistinguibilidad*. Luego, en 3.2.2, mostraremos la interfaz de los objetos intermediarios con el razonador. Más adelante, en 3.2.3, mencionaremos detalles sobre los valores simbólicos para los enteros, los *IntProxy*. Siguiendo con los objetos intermediarios, en 3.2.4, haremos hincapié en los valores simbólicos más importantes, los *BoolProxy*; posteriormente, mostraremos uno de los tipos más complejos, las listas simbólicas o *ListProxy* (3.2.5). Finalmente, en la última parte de esta sección hablaremos sobre los *SliceProxy* (3.2.6) y los *StringProxy* (3.2.7).

3.2.1. Indistinguibilidad de los objetos intermediarios

Es necesario definir mejor la noción intuitiva de objetos "indistinguibles" que utilizamos en el ítem 2 de la definición de objetos intermediarios.

Sea Φ el conjunto de instancias de clases intermediarias y Σ el conjunto de todas las instancias:

- $Proxify :: \Sigma \to \Phi$, crear un objeto intermediario que representa una instancia concreta. Para instancias intermediarias, se comporta como la función identidad.
- $Concretize :: \Sigma \to \Sigma$, retorna uno de los objeto concretos representados por un objeto intermediario. Para instancias concretas, se comporta como la función identidad.
- $\forall i \in \Sigma, \quad Concretize(Proxify(i)) == i)$

Decimos que las instancias concretas y las intermediarias son indistinguibles cuando $\forall f \in FuncionesDePython$, $\forall i_1, .., i_n \in \Sigma - \Phi$, $v_1, .., v_n \in \Sigma \cup \Phi$ donde además, para cada v_i vale alguna de las siguientes: $v_i = i_i$, o bien $v_i = Proxify(i_i)$, tenemos que:

1. $f(i_i, ..., i_n)$ genera una excepción e si y sólo si $f(v_i, ..., v_n)$ genera una excepción e'. Además $type(e) = type(e')$.
2. Si no se genera una excepción, entonces
   $f(i_i, ..., i_n) == Concretize(f(v_i, ..., v_n))$

Notar que v_i puede ser tanto una variable simbólica como concreta.

Por razones de simplicidad en la notación, obviamos involucrar en la definición los argumentos en formato *clave=valor* (kwargs) presentes en Python. Sin embargo, la noción de indistinguibilidad sigue valiendo.

Los fundamentos de esta noción de indistinguibilidad están fuertemente relacionados con los de la conmutatividad de la semántica de ejecución, explicada en la sección 2.5.5, página 51.

3.2.2. Interfaz con el Razonador

Los valores simbólicos, u *objetos intermediarios*, son los pilares esenciales de la ejecución simbólica; ellos son los encargados de comunicar el estado del programa al razonador, el cual para nuestro caso se trata de Z3 [58], el razonador creado por Microsoft Research. Para trabajar con el razonador, generalmente es necesario realizar varias operaciones internas que pueden distraer la atención del lector; es por eso que introduciremos una interfaz simplificada para utilizarlo. Notemos que esta interfaz es sólo para fines didácticos y no se encuentra presente en el código de PEF.

Operación	Función	Ejemplo
smt.fresh_var(tipo)	Genera en el razonador una nueva variable del tipo especificado.	*smt.fresh_var(int)*
smt.make_symbolic(valor)	Convierte un valor concreto en simbólico para que pueda ser utilizado en operaciones del razonador. Se comporta de manera idempotente si el valor ya es simbólico.	smt.make_symbolic(valor)
smt.op(op, valor1, valor2)	Crea un término con el resultado de aplica cualquier operación aritmética (+, -, *, /) a dos términos.	*smt.op("+", x, y)*
smt.pred(rel, valor1, valor2)	Crea una nueva fórmula aplicando una relación aritmética ($=, <, \leq, >, \geq$) sobre dos términos.	*smt.pred("=", x, x)*
smt.formula(op, valor1, valor2)	Crea un nueva fórmula aplicando una operación boolena ($\wedge, \vee, \neg$) sobre dos fórmulas, o una.	*smt.formula("|", True, False)*

smt.solve(formula)	Permite resolver una fórmula creada por *smt.pred* o *smt.formula*.	*smt.solve(smt.pred("≥", smt.make_symbolic(2), smt.make_symbolic(2)))*

3.2.3. IntProxy

Como su nombre lo indica, la clase IntProxy es la clase intermediaria para los enteros en PEF. El objetivo de esta clase es traducir las operaciones aritméticas al razonador, manteniendo la semántica de los enteros nativos de Python.

Mostramos en la figura 3.4 una versión simplificada de la implementación de IntProxy, dando soporte a las operaciones de suma e igualdad a través de los métodos __add__, __radd__, __eq__ y __req__.

Considere la expresión $x + 1 == y$, donde x es un IntProxy representando la variable del razonador s_0, e y una variable entera builtin que contiene el valor 42. La evaluación de esta expresión comienza con el llamado a $x.\_\_add\_\_(1)$, la cual crea la expresión $s_0 + 1$ en el razonador llamando al método apropiado, y devuelve un nuevo IntProxy (supongamos, z) representando esa expresión. Luego $z.\_\_eq\_\_(y)$ es llamado. Este método crea un fórmula en el razonador equivalente a $s_0 + 1 = 42$ y luego retorna a nuevo BoolProxy representando esta fórmula.

Entonces, los IntProxy simplemente construyen expresiones en el razonador que representan la forma en que son usadas en el programa.

Un entero intermediario i debería soportar las siguientes operaciones básicas, donde j puede ser un objeto concreto o simbólico:

Operación	Ejemplo
Suma	$i + j$
Resta	$i - j$
Multiplicación	$i * j$
División	i/j
División Entera	$i//j$
Módulo	$i \% j$

Igualdad	$i == j$
Menor	$i < j$
Menor o igual	$i <= j$
Mayor	$i > j$
Mayor o igual	$i >= j$
...	

```
1  class IntProxy(ProxyObject):
2      """
3      Clase para los enteros simbolicos
4      """
5
6      def __init__(self, value=None):
7          if value is None:
8              self.real = smt.fresh_var(int)
9          else:
10             self.real = smt.make_symbolic(value)
11
12     def __add__(self, other):
13         """
14         x.__add__(y) <==> x+y
15         """
16         return IntProxy(smt.op('+', self.real, other.real))
17
18     def __radd__(self, other):
19         """
20         x.__radd__(y) <==> y+x
21         """
22         return self.__add__(other)
23
24     def __eq__(self, other):
25         """
26         x.__eq__(y) <==> x==y
27         """
28         return BoolProxy(smt.pred('=', self.real, other.real))
29
30     def __req__(self, other):
31         """
32         x.__req__(y) <==> y==x
33         """
34         return self.__eq__(other)
```

Figura 3.4: Código reducido de 'IntProxy'

Notar que para el caso de *IntProxy*, la implementación es trivial, ya que casi todas las operaciones se reducen de manera inmediata a expresiones del razonador.

La división, la única excepción, pregunta si el divisor es cero (y en caso de serlo, se genera una excepción). A demás de ser la operación más compleja, la división es la única operación que puede generar **ramificación en la ejecución simbólica** (pues es la única con sentencias de control de flujo). En el apéndice A.1, página 144, se muestra la implementación completa de *IntProxy*.

3.2.4. BoolProxy

Cada vez que un predicado condicional involucra objetos intermediarios, el método *__bool__* de la clase BoolProxy es llamado por el intérprete de Python para decidir que camino tomar. Se consulta al razonador por la factibilidad de los caminos, esto es, si la fórmula que contiene la instancia de BoolProxy o su negación son satisfactibles junto al resto de la fórmulas en *_pathcondition*. Luego se continúa de la siguiente manera (Código reducido de BoolProxy mostrado en la figura 3.5):

1. Si sólo uno de los caminos es posible, la ejecución sigue por ese camino, devolviendo el método el valor correspondiente, sin alterar las colas *_path* y *_pathcondition* (líneas 16-23).

2. Sino (ambos caminos son posibles).

   a) Si *_path* contiene valores adicionales sobre el camino a tomar, se toma ese camino, actualizando *_pathcondition* según corresponda (líneas 25-32).

   b) Si el límite de profundidad de exploración fue alcanzado, la ejecución del camino es terminada (lineas 35-37).

   c) Sino, se explora el camino correspondiente al valor verdadero de la condición, y se agregan a *_path* y *_pathcondition* el valor True y la fórmula correspondiente (líneas 39-41). El camino por la otra rama (condición falsa) será ejecutado en alguna ejecución futura.

Notar que *_path* y *_pathcondition* sólo se modifican con decisiones no forzadas.

```
class BoolProxy(ProxyObject):
    """
    Clase para los booleanos simbolicos
    """
    def __init__(self, formula=None):
        if formula is None:
            self.formula = smt.fresh_var(bool)
        else:
            self.formula = smt.make_symbolic(formula)

    def __not__(self):
        return BoolProxy(smt.formula('!', self.formula))

    def __bool__(self):
        # Vemos las ramas que pueden ser recorridas
        true_cond = smt.solve(smt.formula('&', _pathcondition, self.formula))
        false_cond = smt.solve(smt.formula('&', _pathcondition,
                     smt.formula('!', self.formula)))
        # Si sólo un camino puede ser elegido, _path y _pathcondition quedan igual
        if true_cond and !false_cond:
            return True
        if !true_cond and false_cond:
            return False

        if len(_path) > len(_pathcondition):
            # Tenenmos que recuperar un estado anterior
            branch = _path[len(self._pathcondition)]
            if branch:
                _pathcondition.append(self.formula)
            else:
                _pathcondition.append.(smt.formula('!', self.formula)
            return branch

        # Si len(_path) >= MAX_DEPTH, se alcanzó el limite máximo de exploracion.
        if len(_path) >= MAX_DEPTH:
            # Paramos la exploración de esta rama.
            raise MaxDepthError()
        # Seguimos el camino que asigna True a la guarda.
        _path.append(True)
        _pathcondition.append(self.formula)
        return True
```

Figura 3.5: Código reducido de 'BoolProxy'

3.2.5. ListProxy

Si bien las implementaciones de **IntProxy** y **BoolProxy** son bastante directas, a la hora de trabajar con listas no sucede lo mismo. Sea l una *lista intermediaria*, la listas simbólica tiene que ser capaz de soportar las siguientes operaciones básicas.

Operación	Ejemplo	Nota
Acceso a posición	$l[i]$	i puede ser un objeto concreto o simbólico
Asignar a posición	$l[i] = j$	i, j pueden ser objetos concretos o simbólicos
Obtener el largo	$len(l)$	El largo de una lista simbólica es un entero simbólico
Sumar otra lista	$l + m$	m puede ser un iterable concreto o simbólico
Replicar la lista	$l * i$	i puede ser un entero concreto o simbólico
Insertar un objeto	$l.insert(i, j)$	i, j pueden ser objetos concretos o simbólicos
Agregar al final	$l.append(i)$	i puede ser un objeto concreto o simbólico
Comparación	$l < m$	m puede ser un iterable concreto o simbólico
...		Hay muchas más operaciones

Las listas de Python, a diferencia de las de C++ [26] o Haskell [30], son contenedores genéricos, es decir que pueden almacenar elementos de diferentes tipos. Por ejemplo, $l = [1, \text{“}hola\text{”}, \{1 : 2\}]$ es una lista válida en Python.

Ya que conocemos las operaciones y el comportamiento que deben tener las listas, la creación de una lista simbólica o ProxyList se puede reducir a resolver los siguientes problemas:

1. Representar secuencias mutables en el razonador.

2. Hacer que los elementos de la secuencia puedan ser de diferentes tipos.

Para representar secuencias mutables en Z3 existen dos enfoques: utilizar arreglos simbólicos o crear un tipo de datos propio en z3. Decidimos utilizar los arreglos por su sencillo uso y la posibilidad de obtener, de manera integrada, las operaciones de acceso y asignación a posiciones ($array[i]$, $array[i] = j$). Los arreglos de Z3 son mapeos de un tipo I a otro T, donde I es el tipo de datos por el cual se indexan

los elementos y T el tipo de datos a almacenar (I y T son tipos de Z3). Como nos interesa acceder a posiciones mediante números, haremos que I sean los enteros, mientras que T será del tipo necesario para contener los elementos de nuestra lista.

El punto 2 fue descartado y agregado como posible trabajo futuro a causa del tiempo que requería su implementación al momento de escribir este trabajo. Sin embargo, es necesario recalcar que si bien no podemos crear listas simbólicas con elementos de diferentes tipos, es posible generar listas concretas con elementos simbólicos de diferentes tipos, por ejemplo $[IntProxy(), BoolProxy()]$. Nos centraremos entonces en las **listas de enteros**.

Si bien utilizamos arreglos simbólicos para representar listas intermediarias, estos arreglos son mucho más expresivos que las listas que queremos representar. El problema es que son infinitos, no tienen la noción de largo (se pueden indexar elementos desde $-\infty$ hasta ∞). Para solucionarlo, decidimos agregar a la representación de listas un entero simbólico que indique el largo. Ya que los largos son siempre positivos, necesitamos comunicarle al razonador esta restricción. El algoritmo de ejecución simbólica fue aumentado teniendo, a demás de las listas *_path* y *_pathcondition*, una lista de *hechos* llamada *_facts*. Los **hechos** son condiciones que deben ser asumidas como verdaderas a lo largo de toda la ejecución. Al no formar parte de *_pathcondition* no afectan el límite de profundidad. En la figura 3.6 podemos apreciar como se utiliza _facts en la creación de listas simbólicas.

```
class ListProxy(ProxyObject):
    """
    Esta clase implementa listas simbólicas
    """
    def __init__(self, initial=None):
        if not initial:
            self._array = smt.fresh_var(array_int)
            self._len = smt.fresh_var(int)
            # Le exigimos al razonador que el largo sea positivo
            _facts.append(self._len >= 0)
        else:
            self._array = smt.make_symbolic(initial)
            self._len = smt.make_symbolic(len(intitial))
```

Figura 3.6: PEF representa las listas simbólicas como un arreglo de z3 (razonador) y un entero que representa el largo

A diferencia de los enteros intermediarios, en donde casi la totalidad de sus

operaciones no genera la "ramificación" del árbol de ejecución, y en el peor de los casos se divide sólo en 2; las operaciones sobre listas simbólicas generan, en su mayoría, una gran cantidad de caminos a explorar. Esta diferencia puede entenderse por el hecho de que los los enteros son soportados de manera nativa por el razonador, mientras las listas no, sumado además a la naturaleza intrínseca de las mismas: son secuencias de datos. Se pueden generan muchos errores al manipular listas, que se traducen en caminos, así también como generar mucha ramificación cuando el largo simbólico de la misma no se encuentra debidamente restringido.

A continuación, mostraremos un programa de ejemplo, el cual ejecutaremos simbólicamente con una lista intermediaria para apreciar algunas operaciones que generan grandes ramificaciones. Consideremos el programa de la figura 3.7. Al ejecutar la función f con una lista simbólica (l_0), en la línea 2 se debe debe ejecutar la sentencia *if*. La ejecución simbólica puede continuar por tres caminos diferentes:

- Asumir $len(l_0) == 0$

  Generar un *IndexError*, pues el largo de la lista simbólica es 0 y por lo tanto no se puede evaluar $l[0]$.

- Asumir $l_0[0] < 0$.

  No solo se asume como verdadera la condición del *if*, sino que también se supone que el largo de la lista es mayor a cero ($cc = [l_0[0] < 0, len(l_0) > 0]$). Se procede ejecutando el bloque del código del *then*. Posteriormente, al ejecutar la línea 3 ya no se puede producir un error de índice, pues la condición de camino dice que esto no puede suceder, y finalmente se ejecuta el *return*.

- Asumir $\neg(l_0[0] < 0)$.

  Nuevamente, no solo se asume como falsa la condición del *if*, sino que también se supone que el largo de la lista es mayor a cero ($cc = [\neg(l_0[0] < 0), len(l_0) > 0]$). Se continúa con el bloque del *else* y se sigue en la línea 5, con una iteración sobre la lista simbólica. Debido a que la condición de camino nos dice que $len(l_0) > 0$, la primera iteración se realiza sin ningún problema. En cualquier iteración posterior a la primera, la condición de camino es igual a $[\neg(l_0[0] < 0), len(l_0) > 0, ..., len(l_0) > i]$ (con i el número de iteración), y la ejecución

```
1 def f(l):
2     if l[0] < 0:
3         l[0] = 0
4     else:
5         for j in l:
6             print(j)
7     return None
```

Figura 3.7: Una función muy sencilla puede producir infinitos caminos a recorrer. En este caso, la ejecución es podada por el límite de exploración.

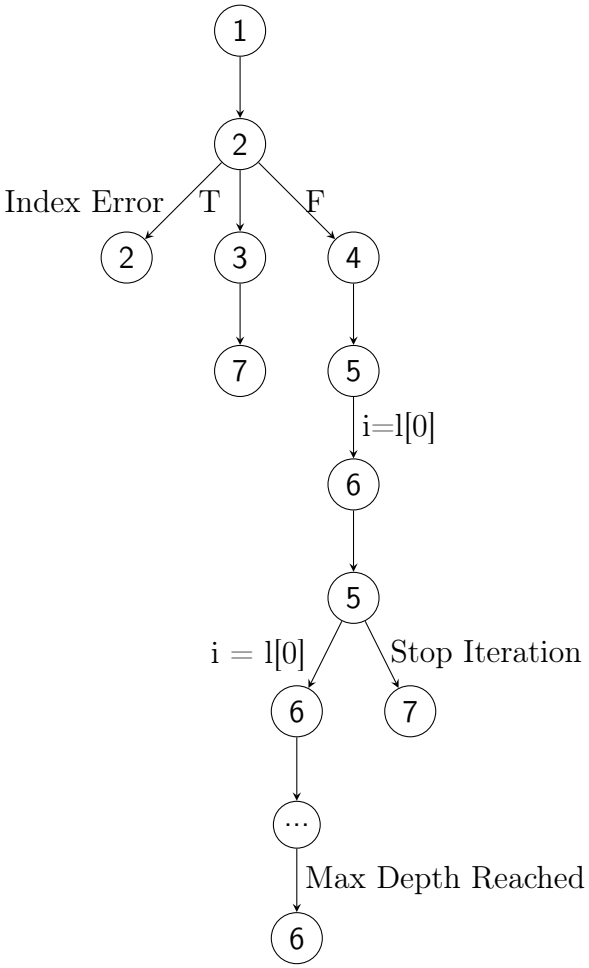


Figura 3.8: Árbol de ejecución generado por la función f de la figura 3.7.

simbólica dispone de dos caminos para recorrer: que se termine de iterar e inmediatamente se retorne *none*; o que se continúe iterando. Cada vez que se decide continuar iterando, se agrega a cc la expresión $len(l_0) > i$, con i el número de iteración. Como el largo de la lista puede crecer tanto como sea necesario, las iteraciones continuarán hasta que se llegue a la **máxima profundidad de exploración**. Si la máxima profundidad es fijada en 3, la 3^{ra} iteración violará el largo máximo ($cc = [\ \neg(l_0[0] < 0), len(l_0) > 0, len(l_0) > 1, len(l_0) > 1\ ]$) y la ejecución simbólica será interrumpida.

El árbol de ejecución simbólica del ejemplo anterior se encuentra disponible para el lector en la figura 3.8.

Varias optimizaciones han sido realizadas a las listas simbólicas para reducir la

explotación de estados; aún así hay operaciones como la iteración sobre las mismas que, como vimos en el ejemplo anterior, obligan a recorrerlas completamente y generalmente provocan la finalización de la ejecución simbólica por alcance del límite de exploración.

Resumiendo, *ListProxy* es la clase intermediara usadas para las listas. Está implementado como un arreglo de Z3 y una variable entera intermediaria que indica el largo del mismo. Las listas intermediarias solo soportan el almacenamiento de enteros; sin embargo, en la mayoría de los casos es suficiente con usar listas concretas (nativas) con objetos intermediarios, lo que mitiga esa restricción. La implementación completa de este tipo puede leerse en el apéndice A.3 (página 150).

3.2.6. SliceProxy

Para brindar soporte a todas las operaciones sobre listas, fue necesario implementar slices intermediarios que permitan operaciones como ListProxy[IntProxy(): IntProxy()]. Las operaciones sobre slices generan ramificación en la ejecución simbólica, aunque mucho menos que las listas simbólicas. La implementación completa puede leerse en el apéndice A.5 (página 175).

Como caso anecdótico, tuvimos que leer la implementación de CPython de la función *indices* para poder emular el comportamiento, el cual no resultaba trivial. Gracias a que utilizamos PEF para testear nuestra propia implementación (más acerca de esto en la sección 4.2), descubrimos muchos errores causados por una incorrecta interpretación de la manera en que debía comportarse esta función.

3.2.7. StringProxy

Al igual que en *ListProxy*, el razonador utilizado no brinda un soporte nativo para *strings*. Si bien han sido desarrollados complementos para trabajar con *strings* en *Z3* [73], estas extensiones sólo soportan versiones antiguas del razonador y no pudieron ser aplicadas a PEF.

En *Python 3* los *strings* son secuencias de caracteres con una codificación *unicode* [74]. Son similares a las listas en funcionalidad, aunque **inmutables**. Python no

ofrece soporte para el tipo caracter, acceder a una posición de un *string* retorna otro *string* de largo uno. Sus operaciones más utilizadas son las siguientes:

Operación	Ejemplo	Nota
Acceso a posición	$s[i]$	i puede ser un objeto concreto o simbólico. Retorna un string de largo 1.
Obtener el largo	$len(s)$	El largo de un string simbólico es un entero simbólico
Concatenar strings	$s + m$	Se obtiene un string nuevo
Replicar strings	$s * i$	i puede ser un entero concreto o simbólico
Comparar string	$s < m$	m tiene que ser un string
...		Hay muchas más operaciones

Como podemos apreciar, las operaciones de *strings* son muy similares a las de listas simbólicas, podemos considerar a los *strings* como un tipo especial de listas con algunas restricciones. Como ya mencionamos, no disponemos de soporte nativo de *strings* en el razonador. PEF se encarga de implementar los *strings simbólicos* utilizando la misma receta de los *ListProxy*: un arreglo y un entero simbólicos, aunque en este caso los arreglos son de enteros más pequeños. En la figura 3.9 se puede apreciar el pseudocódigo para la creación de un *StringProxy*.

```
class StringProxy(ProxyObject):
    """
    Esta clase implementa strings simbólicos
    """
    def __init__(self, initial=None):
        if initial is none:
            self._array = smt.fresh_var(smt.array_int_16^3)
            self._len = smt.fresh_var(int)
            # Le exigimos al razonador que el largo del string
            # sea positivo
            _facts.append(self._len >= 0)
        else:
            self._array = smt.make_symbolic(initial)
            self._len = smt.make_symbolic(len(intitial))
```

Figura 3.9: PEF representa los strings simbólicos como un arreglo de Z3 (razonador) y un entero que representa el largo

PEF representa los caracteres como enteros simbólicos de 16 bits. Se utilizan 16 bits ya que este es el tamaño (en bits) de caracteres que utiliza la codificación *unicode*. Al igual que en *ListProxy*, se utiliza un entero simbólico para representar el largo de la palabra.

Unicode es un gran diccionario que mapea enteros de 16 bits a caracteres y propiedades. En *ASCII* [75], otra codificación para caracteres, el bit 5 (100000b) es utilizado para diferenciar mayúsculas y minúsculas. En *unicode* ese tipo de "trucos" para razonar no se encuentran disponibles. Verificar cualquier propiedad sobre un caracter implica buscarlo en el diccionario. Debido a la falta de "trucos" y a la falta de soporte para diccionarios de nuestro motor de ejecución simbólica, **razonar sobre caracteres codificados en *unicode* es difícil**. Las versiones preliminares de PEF codificaban los caracteres en *ASCII*, aunque esto simplifica muchas operaciones sobre strings, se pierde soporte para muchos caracteres esenciales y la implementación se aleja de los *strings* reales.

La implementación completa de los *strings intermediarios* puede leerse en el apéndice A.4 (página 158).

3.3. Contratos

Sin ayuda del usuario, *PEF* no puede hacer mucho mas que mostrar las excepciones que se generan durante la exploración, como candidatos a fallas en el código (no siempre lo son). Para mejorar la detección de errores, incorporamos un sistema de contratos, donde se pueden describir **tipos y propiedades en general** a ser validadas o impuestas en cada ejecución. Los sistemas de contratos, explicados en 2.1.1 y 2.5.7, clasifican las propiedades en dos grupos: **precondiciones** y **postcondiciones**. Estas propiedades permiten indicar qué restricciones deben valer justo antes y después de la ejecución de la función. Es necesario destacar que PEF valida las cláusulas de los contratos sin terminación, es decir que si la función objetivo no termina la postcondición no será validada.

[3] smt.array_int_16 es un tipo de datos presente en el razonador permite representar arreglos que contienen enteros de 16 bits.

Implementamos el sistema de contratos usando el *docstring* de las funciones a explorar, esto es, definimos una sintaxis para describir las propiedades en la documentación del código, sin tener que recurrir a otros métodos más invasivos, como la introducción de *decoradores* o sentencias dentro del código ejecutable (*Instrumentación de código*). El BNF de la sintaxis antes mencionada será mostrado en 3.3.1, 3.3.2, 3.3.3 y 3.3.4. La sintaxis creada se ajusta a las prácticas recomendadas [69] [70] [71] [72] para documentar código Python, y creemos además es de utilidad para el programador que lee el código. Por comodidad declaramos dos atajos sintácticos para describir precondiciones sobre los tipos de los argumentos, los cuales llamamos *Contratos de Tipo*, y para describir las aserciones que deben valer al generarse una excepción, que llamamos *Contratos de Excepciones*. Los *Contratos de Tipo* son esenciales para PEF, son los encargados de guiar la ejecución simbólica disparando la creación de objetos intermediarios.

A diferencia de la validación de contratos explicado en 2.5.7, PEF no verifica contratos en llamadas a funciones *durante* la ejecución simbólica. Dada una función objetivo, PEF utiliza los contratos únicamente en dos momentos: antes de comenzar con la ejecución simbólica, para reducir el espacio de búsqueda; y al finalizar la ejecución simbólica para validar la postcondición. Es decir, si una función objetivo contiene llamadas a otras funciones con contratos en su cuerpo, estos no son validados. En 6.3.2 se propone como mejora implementar la validación de contratos durante la ejecución.

Las expresiones de los contratos (llamadas *<python_ exprs>* en las BNF) pueden referenciar cualquier objeto dentro del ámbito que contiene a la función donde son declarados.

A continuación detallaremos diferentes tipos de contratos.

3.3.1. Contratos de Precondiciones

Como su nombre lo sugiere, los *contratos de precondiciones* permiten hablar sobre las propiedades que tienen que ser válidas antes de ejecutar la función objetivo. Estos contratos son utilizados por PEF para restringir el conjunto inicial de valores con lo

```
def division_de_positivos(i, j):
    """
    :assume: i > 0, j > 0
    """
    i // j
```

Figura 3.10: Ejemplo pequeño de una función que define un contrato de precondición

cuales explorar la función (sólo se consideran los que hacen válida la precondición).

$$< \textbf{\textit{con\_pre}} > \quad ::= \text{“ : } \textbf{\textit{assume}} \text{ : ”} \quad < \textbf{\textit{python\_exprs}} >$$

$$< \textbf{\textit{python\_exprs}} > ::= \quad < \textbf{\textit{python\_expr}} >$$

$$\mid \quad < \textbf{\textit{python\_expr}} > \text{ “,” } < \textbf{\textit{python\_exprs}} >$$

Donde <*python_ expr*> es cualquier expresión válida de Python.

Dentro de un contrato de precondiciones, el usuario puede referenciar cualquiera de los parámetros de entrada y utilizar todas las funciones *builtin* o definidas en el mismo contexto de la función objetivo.

Usualmente, un contrato de precondición consiste en una o varias expresiones que relacionan los parámetros de entrada. Si se quiere escribir mas de una propiedad, basta con separar las expresiones por comas (‘,’). PEF interpretará las expresiones separadas por comas como la conjunción de las mismas.

En la figura 3.10 mostramos un pequeño ejemplo donde se utiliza un contrato de precondición para indicar que ambos argumentos deben ser mayores a 0.

3.3.2. Contratos de Postcondiciones

Análogo a los contratos de precondiciones, los *contratos de postcondiciones* permiten hablar sobre las propiedades que deben mantenerse luego de ejecutar la función objetivo. Estos contratos son utilizados por PEF como un *oráculo de test* (2.1.5)

para detectar **errores semánticos** en la función.

$$< con\_post > \quad ::= \text{“} : ensure : \text{”} \ < python\_exprs >$$

$$< python\_exprs > ::= < python\_expr >$$

$$| < python\_expr > \ \text{“,”} \ < python\_exprs >$$

Donde <*python_ expr*> es cualquier expresión válida de Python.

Dentro de un contrato de postcondiciones, el usuario puede referenciar el valor de retorno de la función con la palabra reservada **returnv** (Si un argumento es nombrado returnv, se generará un error durante el parseo del contrato). Cuando se hace referencia una variable declarada en la definición de la función, la validación del contrato utilizará los valores que las mismas contengan al comenzar la ejecución de la función (Poder hacer referencia a los valores en sus estados finales es propuesto como mejora en 6.3.2)

Usualmente, un contrato de postcondición consiste en una o varias expresiones que relacionan los parámetros de entrada con la salida. Si se quiere escribir mas de una propiedad, basta con separar las expresiones con comas (‘,’) .PEF interpretará las expresiones separadas por comas como la conjunción de las mismas.

En la figura 3.11 mostramos un ejemplo de contrato de postcondición, donde se indica que el resultado debe ser mayor o igual a la suma de ambos argumentos.

```
def f(a, b):
    """
    :ensure: returnv >= a + b
    """
    return abs(a) + abs(b)
```

Figura 3.11: Ejemplo pequeño de una función que define un contrato de postcondiciones

3.3.3. Contratos de Tipo

Los contratos de tipo son atajos sintácticos para definir precondiciones sobre el tipo de los argumentos.

Recordemos que Python es un lenguaje dinámicamente tipado y carece de de-

finición explícita de tipos. Los contratos de tipos son los encargados de especificar el tipo (o tipos) de las variables de entrada de un método o función. Esta clase de contratos es esencial para nuestra herramienta, sin ellos no se podrían crear las entradas simbólicas correspondientes.

$$
\begin{aligned}
&< con\_type > && ::= \text{“ : } type \text{ : ”} \ < args\_types > \\
&< args\_types > && ::= \ < arg > \ \text{“ : ”} \ < type\_def > \\
&&& \ | \ \ < args\_types > \ \text{“ , ”} \ < args\_types > \\
&< type\_def > && ::= \ < type > \ | \ \text{“ [”} \ < types > \ \text{“] ”} \\
&< types > && ::= \ < type > \ | \ < type >, < types >
\end{aligned}
$$

Donde <*type*> es cualquier tipo builtin o definido por el usuario.

La sintaxis permite indicar que los argumentos pueden ser de mas de un tipo. En éste caso, *PEF* realiza un ejecución simbólica con cada uno. Cuando son varios argumentos los que definen mas de un tipo, se producen tantas ejecuciones simbólicas como combinaciones de tipos haya en los argumentos. En la figura 3.12 encontraremos un ejemplo simple que contiene declaraciones de tipo de los argumentos. Notar que para la variable *b* se indican dos posibles tipos, resultando en 2 ejecuciones simbólicas de la función cuando sea explorado por *PEF*, una en donde tendremos a ambas variables de tipo *IntProxy*, y otra donde *a* será de tipo *IntProxy*, mientras *b* será de tipo *StringProxy*.

```
def multiplicacion_polimorfica(a, b):
    """
    :types: a: int, b: [int, str]
    """
    return a * b
```

Figura 3.12: Ejemplo pequeño de una función que define un contrato de tipos

Deducción automática de tipos

Si bien los contratos de tipos son **necesarios** para que la herramienta ejecute simbólicamente un método/función, existen situaciones donde *PEF* es capaz de deducir correctamente el tipo de algunos de sus argumentos:

- En el caso de los métodos, siempre se cumple que el primer argumento, por convención llamado *self*, es del mismo tipo que la clase que lo contiene. Si PEF no encuentra contrato de tipos para este argumento, hará uso del razonamiento anterior; de modo que el tipo para este argumento puede ser obviado.

- Cuando un método/función declara en su prototipo un *vararg* (por convención llamados **args*): si no hay un contrato de tipo que caracterice el *vararg*, PEF asumirá que es de tipo *lista*. Al igual que con *self*, el tipo del *vararg* puede ser obviado.

- Cuando un contrato de tipos define un parámetro de tipo *lista*, debido a que PEF sólo es capaz de manejar listas de enteros **asumirá que la lista es de enteros**.

Cada vez que un argumento no esté contemplado dentro del contrato de tipo, el mismo intentará ser deducido usando los puntos detallados anteriormente. En caso de no ser posible su deducción, se generará una excepción de contrato de tipo interrumpiendo la ejecución simbólica.

3.3.4. Contratos de excepciones

Los contratos de *Tipos*, *Precondición* y *Postcondición* nos permiten hablar sobre el comportamiento del programa. Sin embargo, estos contratos no nos permiten describir que sucede cuando una excepción es generada. No toda excepción es un error. En muchas oportunidades, se desea que una función genere excepciones para comunicar su estado interno. Es por esto que es menester para nuestro sistema de contratos poder diferenciar excepciones esperadas de excepciones producidas por errores. Incorporamos a nuestro sistema de contratos un nuevo tipo, **contrato de excepciones**, capaz que realizar este trabajo. A continuación presentamos el *BNF*

de estos contratos.

$$< con\_exc > \ ::= \text{“}:raises:\text{”} \ < exc\_expr >$$

$$< exc\_expr > ::= < exc\_type > \ \text{“}:\text{”} \ < python\_expr >$$

$$\mid < exc\_expr > \ \text{“},\text{”} \ < exc\_expr >$$

$$< exc\_type > ::= \text{“}any\text{”} \mid < exc\_type\_in\_python\_scope >$$

Donde *<exc_type_in_python_scope>* es cualquier excepción builtin o definida por el usuario que se encuentre en el scope de la función.

```
def division_entera(i, j):
    """
    :raises: ZeroDivisionError: j == 0
    """
    return i // j
```

Figura 3.13: Ejemplo pequeño de una función que define un contrato errores

En caso de generarse una excepción mientras PEF recorre un camino del programa objetivo, si la misma está contemplada en un contrato de excepciones y la expresión relacionada se evalúa a True, tal ejecución se considera válida. En caso contrario, se reporta como un error.

Al ocurrir una excepción, PEF busca el primer contrato que coincide con el tipo de la excepción, recorriendo la lista de arriba hacia abajo en las declaraciones, análogo al pattern matching de Haskell [30].

En el ejemplo de la figura 3.13 podemos ver un contrato de excepción en acción, el mismo indica que en caso de llamar la función con $j == 0$, la excepción de tipo *ZeroDivisionError* no es un error, sino un comportamiento deseado.

3.3.5. Contratos en PEF

A continuación mostraremos la manera en que PEF integra los contratos con la ejecución simbólica.

1. Antes de comenzar la ejecución simbólica, PEF busca contratos de tipo en la función a explorar.

- Si no encuentra contratos de tipos, intenta deducirlos según los explicado en 3.3.3.
- Si no puede deducirlos, se genera un error de contrato.

2. PEF planifica ejecuciones simbólicas para cada posible combinación de tipos de los argumentos.

3. Antes de comenzar la primer ejecución simbólica, si existe un contrato de precondición en la función objetivo, la expresión de la misma se agrega como un hecho, en la lista *_facts*. Esto reduce el espacio de búsqueda.

4. Si se genera una excepción durante la ejecución, la cual no es manejada por ningún *try*, se buscan contratos de excepción. Si ninguno de los contratos de excepción coincide con el tipo de la excepción generada, la ejecución correspondiente se considera con presencia de errores. Si por el contrario alguno coincide, se evalúa la expresión asociada; si el razonador valida la expresión como cierta, entonces la ejecución no se considera un error.

5. Al finalizar cada ejecución (explorar una rama del programa), si existe un contrato de postcondición en la función objetivo, se valida el cumplimiento de la expresión que contenga.

   - Si el razonador puede hacerla falsa bajo las restricciones encontradas al finalizar la ejecución, entonces se genera un error de contrato.
   - Si no se puede invalidar, la ejecución se interpreta como válida.
   - Si el razonador no puede deducir si la expresión asociada es satisfactible o no, PEF genera una excepción interna y la misma es notificada al usuario.

Notar que si la ejecución termina por una excepción, únicamente se evalúan los contratos de excepciones. Notar además, que si el programa no tiene bucles, no se alcanza el límite de exploración y el razonador puede deducir todas las ejecuciones como válidas, entonces el programa es correcto con respecto a su pre y postcondición.

3.4. Clases de usuario intermediarias

La *arquitectura de pares* ha sido desacreditada [5] por funcionar únicamente para clases builtin. Sin embargo, en este trabajo mostraremos que es posible extender la técnica de arquitectura de pares para generar, de manera automática, valores simbólicos para clases definidas por el usuario.

3.4.1. Generación de objetos simbólicos de cualquier tipo

Hemos desarrollado la función *proxify*, que dado un objeto concreto, es capaz de generar objetos simbólicos equivalentes, cualquiera sea su tipo. A continuación, mostraremos la idea general para la generación valores simbólicos "frescos" (creados a partir de una clase, en vez de una instancia), ya que generar valores simbólicos a partir de una instancia de clase es un caso particular. Lo haremos sobre una versión reducida de *proxify* que llamaremos *proxify_simple*, para lograr abstraernos de algunos detalles de implementación de PEF, los cuales serán tratados en detalle más adelante en esta sección.

La función *proxify_simple(t)*, toma como parámetro un tipo de dato t y retorna un valor simbólico nuevo capaz de representar instancias de tipo t. La misma se encuentra definida de manera recursiva de la siguiente forma:

1. Si t es un tipo builtin (una clase nativa de Python):
   Se crea un nuevo objeto simbólico del tipo correspondiente para el tipo t (*IntProxy* para *int*, *ListProxy* para *list*, *BoolProxy* para *bool*, ...).

2. Si por el contrario, t es una clase definida por el usuario:
   Se explora el método *__init__* de la clase en busca de un contrato de tipo (3.3.3) para sus parámetros. Se ejecuta recursivamente *proxify_simple* con los tipos indicados por el contrato y se retorna una nueva instancia de la clase, donde *__init__* es ejecutado con los parámetros simbólicos recién generados. Esta instancia es un objeto intermediario para la clase t o instancia simbólica de t.

   Si el tipo de alguno de los parámetros es de una clase definida por el usuario,

por hipótesis inductiva, *proxify_simple* habrá devuelto una instancia intermediaria que representa objetos de esa clase.

```
1  class Stack(object):
2      def __init__(self, initial=None):
3          """
4          :types: initial: list
5          """
6          if initial is None:
7              self.stack = []
8          else:
9              self.stack = initial
10
11     def push(self, item):
12         self.stack.append(item)
13
14     def pop(self):
15         try:
16             return self.stack.pop(0)  # Sacamos el primer elemento
17         except IndexError:
18             raise IndexError("Pop from empty stack")
19
20     def __len__(self):
21         return len(self.stack)
22
23
24 def pop_n_times(stack, times):
25     """
26     :types: stack: Stack, times: int
27     """
28
29     for i in range(times):
30         stack.pop()
31
32     if len(stack) > 0:
33         print("El stack aun contiene elementos")
```

Figura 3.14: Ejemplo de simbolización de una clase definida por el usuario

Mostraremos el algoritmo en acción con el ejemplo de la figura 3.14, donde se define la clase *Stack* y la función *pop_n_times*; la misma, se encarga de ejecutar el método *pop()* de la clase *times* veces y luego imprimir el mensaje "El stack aun contiene elementos", en caso de que eso suceda. Para ejecutar simbólicamente la función *pop_n_times* necesitaremos crear objetos intermediarios para los argumentos *stack* y *times*. Sigamos el algoritmo de cerca:

- Se llama a *proxify_simple* para los argumentos *stack* y *times*.

- Para el caso de *times*, como el tipo indicado es *int*, se crea un *IntProxy* de manera usual.
- En el caso del argumento *stack*, como el nombre del tipo declarado en el contrato no coincide con el de ninguna clase interna de Python, se exploran los ámbitos en busca de una definición para la clase *Stack*. Una vez localizada, se procede a crear un objeto intermediario de ese tipo.
  - Se explora el docstring del método $\_\_init\_\_$ de la clase *Stack* en busca de contratos de tipo.
  - Se procede de manera recursiva a generar objetos intermediarios para cada argumento del método (en este caso, el argumento *initial*, de tipo lista).
  - Se genera una instancia de *Stack*, pasando una lista intermediaria como argumento al método $\_\_init\_\_$ de la misma. Esta instancia es un objeto intermediario de la clase *Stack*.
- Se procede con la ejecución simbólica normalmente.

Notemos que, una vez creadas las instancias intermediarias de clases de usuario, el algoritmo de ejecución simbólica se mantiene sin modificaciones; esto se debe a que eventualmente todas las operaciones se traducen a operaciones entre objetos de tipos nativos de Python. Para una instancia de *Stack*, este comportamiento es observable a través del atributo *stack*, de tipo *list*; eventualmente todas las operaciones que alteran el estado de la instancia modifican este atributo para reflejar el cambio, por lo tanto, "proxificar" esta variable es suficiente.

3.4.2. Detalles de implementación de *proxify*

Como dijimos en 3.4.1, si bien *proxify_simple* mantiene la idea general de la función $proxify$, abstrae algunos detalles internos de PEF. Es hora que nos centremos en ellos.

En primer lugar, supongamos que queremos generar una instancia intermediaria de la clase *NumeroOString* definida en la figura 3.15. Cuando $proxify$ explora en

busca de contratos de tipos, encuentra que el parámetro *nos* puede tomar dos tipos diferentes, *int* y *str*. Debido a que PEF no soporta valores simbólicos "multi-tipo" (es decir, que puedan representar valores de varios tipos), $proxify$ en realidad genera un **conjunto de instancias simbólicas**, donde cada elemento del conjunto es una combinación diferente de tipos de entradas. En el caso de la clase *NumeroOString*, $proxify(NumeroOString)$ generará el conjunto $\{NumeroOString(IntProxy()), NumeroOString(StringProxy())\}$.

```
1 class NumeroOString(object):
2     def __init__(self, nos):
3         """
4         :type: nos: [int, string]
5         """
6         self.nos = nos
```

Figura 3.15: La clase *NumeroOString* tiene un atributo que puede ser una cadena de texto o un entero. El tipo del atributo depende del parámetro con que $\_\_init\_\_$ es llamada.

El segundo gran problema al que se enfrenta *proxify_simple* es cuando el $\_\_init\_\_$ presenta varios caminos de ejecución. Consideremos la clase *MayorQueCinco* de la figura 3.16. Cuando *proxify_simple* intente generar una instancia simbólica de *MayorQueCinco* ejecutará el $\_\_init\_\_$ de la clase con un *IntProxy* como parámetro. Aquí llega la parte interesante, al ejecutar la línea 6 se está comparando una variable simbólica con una concreta. Para que esta comparación tenga sentido, es necesario ejecutar el $\_\_init\_\_$ simbólicamente. Es por eso que para este caso nuestra implementación de $proxify$, al igual que el caso anterior, genera un conjunto de instancias simbólicas de la clase, donde cada elemento es un resultado de la ejecución simbólica. En el caso de nuestra clase *MayorQueCinco*, $proxify(MayorQueCinco)$ retornará un conjunto de dos elementos $\{s_0, s_1\}$, donde $s_0.value = i_0$, $s_1.value = i_0 + 5$ e i_0 es un entero simbólico, con las restricciones impuestas por el control de flujo de la línea 6 correspondiente para cada caso.

```
1 class MayorQueCinco(object):
2     def __init__(self, n):
3         """
4         :type: n: int
5         """
6         if n > 5:
7             self.valor = n
8         else:
9             self.valor = n + 5
```

Figura 3.16: La clase *MayorQueCinco* necesita ser ejecutada simbólicamente para generar dos tipos de instancias diferentes

Resumiendo, si bien *proxify* y *proxify_simple* siguen la misma idea básica para proxificar clases, hay dos grandes diferencias:

- *proxify* genera conjuntos de instancias, cada una distinta a todas las demás del conjunto.
- *proxify* ejecuta simbólicamente el *__init__* de la clase a proxificar con valores simbólicos.

Supongamos ahora que queremos ejecutar simbólicamente la función de la figura 3.17. Comenzamos por crear, utilizando *proxify*, instancias simbólicas de tipo A, B y C. Esta operación nos generará tres conjuntos S_A, S_B y S_C. Luego, para ejecutar simbólicamente la función f se hacen $|S_A|*|S_B|*|S_C|$ ejecuciones simbólicas, correspondientes a cada elemento del producto cartesiano $S_A \times S_B \times S_C$. Supongamos que $S_A = \{a_0, a_1\}$, $S_B = \{b_0\}$ y $S_C = \{c_0, c_1\}$, para ejecutar simbólicamente f tenemos que realizar las siguientes ejecuciones simbólicas y luego combinar sus resultados:

- $f(a_0, b_0, c_0)$
- $f(a_0, b_0, c_1)$
- $f(a_1, b_0, c_0)$
- $f(a_1, b_0, c_1)$

```
def f(a, b, c):
    """
    :type: a: A, b:B, c:C
    """
    ...
```

Figura 3.17: Función genérica con varios parámetros

La implementación completa de la función *proxify* puede leerse en el apéndice A.6 (página 176).

3.4.3. Limitaciones en la *proxificación* de clases

Si bien la técnica de generación de instancias simbólicas que introducimos en esta sección permite superar los problemas sugeridos en [5], nuestra solución también presenta limitaciones. Estas limitaciones están relacionadas a la forma en que *proxify* se encuentra implementada en PEF al momento de escribir este trabajo. Únicamente las variables declaradas en el método $\_\_init\_\_()$ de la clase son proxificadas, por lo tanto, instancias que modifiquen variables globales o de clase, puede que no queden convertidas a intermediarias de forma completa, salteando posiblemente algunas ramas de ejecución para el programa bajo exploración.

A continuación, mostraremos un caso de uso particular en el que PEF no es capaz de simbolizar correctamente una clase definida por el usuario. La clase "*ClaseUnica*" de la figura 3.18 almacena en un atributo x un entero, y a demás tiene una variable de clase, *contador*, que registra la cantidad de instancias creadas de la clase. Consideremos ahora la función, *test_clase_unica* de la figura 3.18 que genera una excepción si se ha creado más de una instancia de *ClaseUnica*.

Para este caso de estudio, PEF no es capaz de generar instancias de ClaseUnica que generen y no generen la excepción, analicemos porque. Para ejecutarla simbólicamente *test_clase_unica*, primero debemos generar una instancia simbólica de *ClaseUnica*. Al llamar a *proxify(ClaseUnica)* se explora el $\_\_init\_\_$ en busca de contratos, se crea una instancia de enteros simbólico y se llama al $\_\_init\_\_$ con ese entero como parámetro. Se obtiene como resultado un conjunto con una única instancia simbólica de *ClaseUnica*, llamemos al elemento *inst*. Ahora podemos proceder a ejecutar simbólicamente *test_clase_unica*. Al momento de ejecutar la línea

14, como *obj.contador* no es un valor simbólico, no se producirá la ramificación de la ejecución simbólica y sólo se obtendrá un único resultado (ya sea generando o no una excepción).

```
1  class ClaseUnica(object):
2      contador = 0
3      def __init__(self, x):
4          """
5          :type: x: int
6          """
7          self.x = x
8          ClaseUnica.contador = ClaseUnica.contador + 1
9
10 def test_clase_unica(obj):
11     """
12     :type: obj: ClaseUnica
13     """
14     if obj.contador >= 2:
15         raise Exception("Ding ding ding!")
```

Figura 3.18: PEF no es capaz de *proxificar* de manera correcta a *ClaseUnica*.

Las limitaciones de esta implementación suceden con el uso de variables de clases y variables globales.

3.5. Builtins

Para que PEF pueda explorar los caminos del programa objetivo, es necesario que no sea posible para el mismo diferenciar entre un objeto y su equivalente intermediario.
Tuvimos entonces que modificar algunas funciones internas de Python que no permitían el comportamiento deseado.
El funcionamiento de los módulos y sus referencias a otros objetos en Python puede leerse en la sección 2.2.8, página 28.

- **len**: La implementación nativa llama al método __len__ del objeto pasado como argumento y verifica que el valor de retorno sea una instancia de *int*. La nuestra, permite que se devuelva además algo de tipo *IntProxy* cuando el objeto es de tipo intermediario.

- **type**: Necesitamos "mentirle" al programa objetivo cuando pregunte por el tipo de un objeto intermediario. Cuando se llama con un objeto intermediario, nuestra implementación devuelve la clase que el objeto intermediario emula. Como ejemplo, en la ejecución del programa objetivo valdría que **type**(ListProxy()) == **type**([1,2,3]) == **list**

- **isinstance**: Análogo a la modificación realizada para *type()*, devolvemos True cuando se pregunta si un objeto intermediario es una instancia de la clase que emula o viceversa. Como ejemplo, valdría que **isinstance**(IntProxy(3), **int**) en la ejecución del programa objetivo.

- **range**: La implementación nativa verifica que cada valor entregado por el generador que retorna, sea de tipo *int*. Nuestra implementación quita esa restricción para las clases intermediarias, permitiendo entregar instancias de *IntProxy*.

- **enumerate**: Esta función retorna un iterador, que devuelve 2-uplas con los elementos del iterador (argumento) y su posición u orden. La implementación nativa verifica que la posición sea algo de tipo *int*, nuestra versión permite que sean además objetos de tipo *IntProxy*.

3.6. PEF en acción

A continuación mostraremos en un breve ejemplo la manera en que PEF explora una función. En la figura 3.19 se declara la función *divide_elementos* que, dada una lista l y un entero x, retorna una nueva lista donde cada elemento es el resultado de la división de los elementos de l por x.

Se declararon varios contratos; contrato de tipo, donde se especifica que la variable l es una lista (de enteros) y x un entero; contrato de precondición (*assume*), donde se indica que la lista debe contener sólo 2 elementos; contrato de excepción (*raises*), donde se indica que para el caso en que x sea 0, la excepción *ZeroDivisionError* es el comportamiento esperado; contrato de postcondición (*ensure*), donde se indica que la lista que devuelve la función debe contener la misma cantidad de elementos que la lista con la que fue llamada.

Veamos qué sucede en PEF si se ejecuta *explore(divide_ elementos)* (recordemos que 'explore' fue introducida en 3.1). Primero se explora el docstring de la función en busca de contratos, los mismo son los descriptos en el párrafo anterior. Los contratos de tipo devuelven un diccionario de la forma {*nombre_ variable*: *tipos*}, esa información es utilizada para crear las variables intermediarias que correspondan en cada ejecución simbólica (ver 3.3.5 para mas detalle sobre este paso). Luego se ingresa en el bucle principal de la ejecución simbólica, el cual continúa mientras queden ramas por explorar dentro de la función. En cada iteración se crean las variables intermediaras correspondientes para ejecutar la función objetivo. La exploración de las ramas se realiza en orden DFS. Al finalizar una ejecución o recorrido de una rama, se guardan el resultado, el estado de las variables intermediarias al finalizar, las escrituras en la salida estándar (llamadas a print) y las excepciones generadas en caso de haber ocurrido. Además, antes de comenzar con cada ejecución se realizan copias de las variables intermediarias de entrada para poder recuperar sus valores al finalizar, en caso que hayan sido modificadas, y lograr así la correcta caracterización de la ejecución.

Entremos dentro del bucle de la ejecución simbólica con este ejemplo: Al comenzar la primer ejecución, se aplica la precondición de la función como un hecho sobre el estado de las variables, de esta manera, la variable l comienza siendo una lista simbólica que contiene 2 enteros intermediarios, mientas x es un entero intermediario, ambos sin otras restricciones (por ahora).

La ejecución continúa normalmente hasta llegar a la sentencia *for* de la línea 9, allí, el intérprete de Python le consulta a l si tiene al menos un elemento, l responde afirmativamente, luego el intérprete solicita el primer elemento a l y se lo asigna a la variable e. La ejecución continúa hasta la línea 10, donde se agrega el primer elemento a la lista r (creada en la línea 8), el cual consiste en el resultado de computar la división entera entre los enteros simbólicos e (primer elemento de la lista l) y x. El código de la división entera para los enteros analiza si el divisor es distinto de 0, generando la excepción *ZeroDivisionError* en caso negativo. Luego de consultar el valor de x se abren dos ramas de ejecución, una donde el valor de x es distinto de 0, y otra donde es igual a 0.

- Rama *x != 0*:

  Se comienza explorando la rama afirmativa, agregando al comienzo de las listas *_path* y *_pathcondition* los valores *True* y *smt.pred('!=', x, 0)* respectivamente. Seguimos por la rama afirmativa, donde el intérprete vuelve a ejecutar la línea 9. Nuevamente le consulta a l si tiene 2 o más elementos, l responde afirmativamente (recuerden que tenemos el hecho de que *len(l) == 2*) y se asigna a e la variable entera simbólica que corresponde a la segunda posición de la lista l. Al ejecutar la línea 10, se vuelve a computar la división entera, y con ella, se pregunta sobre la presencia de un valor distinto de 0 en x. En este momento, como el razonador ya contiene la expresión *x != 0*, se continúa la ejecución por ese caso, sin producir ramificaciones en la ejecución simbólica. En la próxima ejecución de la línea 9, el intérprete saltea la ejecución del bucle, ya que l indica que ya no posee elementos para iterar. Se ejecuta la línea 11, devolviendo la función una lista concreta que contiene dos enteros simbólicos, que son el resultado de dividir otros enteros simbólicos sin restricciones por un valor no nulo. PEF recibe el resultado de la función, los parámetros de entrada y salida, y verifica que la ejecución sea válida, evaluando los contratos de postcondición. En este caso tenemos la expresión *len(returnv) == len(l)*, PEF reemplaza la variable *returnv* por el valor de retorno de la función, es decir, la lista almacenada en la variable r. Luego verifica el valor de verdad de la expresión *len(r) == len(l)*, la cual al ser verdadera indica que nos encontramos en una ejecución aparentemente sin errores.

- Rama *x == 0*:

  En esta rama de ejecución se genera la excepción *ZeroDivisionError* en la división, finalizando la ejecución, pues la misma no es atrapada dentro de la función que estamos explorando. PEF recibe la excepción y verifica si la misma es contemplada como una ejecución válida en algún contrato de excepción. El tipo de la excepción coincide con la del contrato, por lo que se evalúa la expresión *x == 0*. En este caso, al ser la expresión asociada verdadera, se considerando que la ejecución no posee errores aparentes.

```
1  def divide_elementos(l, x):
2      """
3      :type: l: list, x: int
4      :assume: len(l) == 2
5      :raises: ZeroDivisionError: x == 0
6      :ensure: len(returnv) == len(l)
7      """
8      r = []
9      for e in l:
10         r.append(e // x)
11     return r
```

Figura 3.19: Pequeño ejemplo integrador del funcionamiento de PEF

Capítulo 4

Usos de PEF

En los capítulos anteriores, nos hemos centrado en los conceptos y fundamentos que hacen posible la construcción y ejecución de la herramienta PEF. Sin embargo, todavía no hemos mencionado las bondades y posibilidades de uso real de la misma. A lo largo de este capítulo explicaremos y ejemplificaremos los diferentes modos de uso que brinda la herramienta. El capítulo se encuentra dividido en dos secciones: En la sección 4.1, se analizan los dos grandes modos de uso de la herramienta; En la sección 4.2, se utiliza uno de esos modos de uso para verificar la misma herramienta.

4.1. Modos de uso de PEF

PEF está diseñado para ser utilizado de diferentes maneras, pudiendo agruparse en dos grandes categorías: como **ejecutable independiente** o como **librería**. En esta sección explicaremos las diferencias entre estos modos de uso, como así también detalles sobre la manera de utilizar PEF con cada uno.

4.1.1. Ejecutable Independiente

PEF puede funcionar como un ejecutable independiente, esto es, ser llamado desde la línea de comando. Los parámetros de ejecución junto al programa a explorar son proporcionados como argumentos del programa. La sintaxis para utilizar la herramienta en la versión que se entrega junto con este trabajo, es la siguiente:

```
#!/bin/bash
./pef [-d] <program_file>:<function_name>
```

Donde:

- *<program_file>* es la ruta del archivo donde se encuentra la función que queremos explorar.
- *<function_name>* es el nombre de la función a explorar dentro del archivo *<program_file>*.
- *-d* especifica un valor opcional para el límite de profundidad a usar en la exploración (por defecto es 10).

En este modo, PEF busca contratos y recorre los caminos según el límite de exploración correspondiente; luego imprime en pantalla representantes de los parámetros de entrada y salida que caracterizan a cada camino. Recordemos que al finalizar una ejecución simbólica, PEF obtiene restricciones que caracterizan la entrada y la salida como fórmulas sobre las variables simbólicas. La herramienta se encarga de seleccionar un representante concreto para cada variable simbólica. Estos representantes permiten mostrar de manera mas simple e intuitiva el camino ejecutado, e incluso pueden ser utilizado posteriormente para crear tests que recorran el camino representado (útil para tests de regresión).

En el caso de la salida, PEF indica si hubo un valor de retorno o una excepción y las escrituras a la salida estándar. Además, en caso de que se haya encontrado un contrato de postcondición, se indican las salidas que no validen el mismo. La sintaxis que utiliza PEF para mostrar la información generada es la siguiente:

```
<function>(<parameters>) [-> <return_value>]
 [<program_output>]
     [<exception_generated>]
```

Los corchetes ([,]) representan salidas opcionales, es decir que depende del resultado de la ejecución si son mostrados. <function> es la función a ejecutar; <parameters> son los parámetros con los que la función fue ejecutada; <return_value>

será el valor obtenido al ejecutar <function>(<parameters>), y sólo se muestra si no se generaron excepciones; <program_output> será una lista con todas las escrituras a la salida estándar que ocurrieron dentro de la función, y solo será mostrada si hubo alguna; <exception_generated> mostrará, en caso de que se produzca, la excepción generada al ejecutar <function>(<parameters>).

A continuación mostraremos la salida de PEF para un ejemplo reducido, el código del ejemplo se muestra en la figura 4.1 mientras que la salida puede apreciarse en la figura 4.2. Se puede notar como PEF realiza el recorrido de los caminos en orden DFS, comenzando por la rama verdadera.

```
def suma_con_bug(a, b):
    """
    :types: a: int, b: int
    """
    result = a + b
    if result == 42:
        print("La gran respuesta.")
    if a - b == 42:
        raise Exception("Bug")
    return result
```

Figura 4.1: Pequeño ejemplo del uso de PEF como ejecutable independiente

```
./pef.py examples.py:suma_con_bug -d 10

suma_con_bug(42, 0)
['La gran respuesta.']
    Exception('Bug',)

suma_con_bug(41, 1) -> 42
['La gran respuesta.']

suma_con_bug(43, 1)
    Exception('Bug',)

suma_con_bug(42, -1) -> 41

4 paths explored.
```

Figura 4.2: Salida por pantalla del ejemplo de uso de PEF como ejecutable independiente

4.1.2. Librería

PEF puede además ser utilizado como librería. Este modo de uso es muy útil cuando se desea personalizar el comportamiento de PEF, o bien realizar tareas adicionales con datos internos, como la pila de restricciones de camino. Un ejemplo puede ser que el usuario desee utilizar sus propios oráculos en lugar de los proporcionados por PEF a través de los contratos. La manera de utilizar la herramienta como librería es importando los módulos que resulten de utilidad.

PEF se encuentra modularizado de manera que cada componente funcional pueda ser fácilmente incluido en otros módulos:

- *proxy.py* contiene:
  - Los objetos intermediarios.
  - La función para proxificar objetos *proxify*, explicado en la sección 3.4.
  - La rutina básica de exploración *explore*, explicada en la sección 3.1).
- *contracts.py* contiene:
  - Los contratos, explicado en la sección 3.3.
  - La función *find_ all_ contracts*, que parsea y genera los contratos a partir de un docstring.
- *smt.py* contiene:
  - Interfaz para cambiar el razonador y la manera en que se resuelven las operaciones que despachan los objetos intermediarios.
- *pef.py*

  Este módulo es el ejecutable de PEF. Puede servir como referencia de uso de los módulos mencionados en los puntos anteriores.
- *implementation_ test.py*

  En este módulo se implementan tests automáticos para validar la implementación de los objetos intermediarios. Los mismo son explicados en la sección 4.2. Este es un buen ejemplo de uso de PEF como librería, y muy útil como referencia.

4.2. Auto-verificación de PEF

Por el momento, el único caso real de uso de PEF es un programa creado por los autores llamado *implementation_ test.py*. El programa se encarga de **comparar la implementación interna de los objetos intermediarios de PEF contra los objetos nativos equivalentes**. A lo largo del desarrollo de PEF, nos encontramos con muchas operaciones de los objetos intermediarios que eran, en cierto modo, difíciles de implementar (ya sea por documentación deficiente, o por la complejidad misma de la operación). A causa de esas dificultades fue que nació *implementation_ test.py*, encargándose de comparar las implementaciones de los métodos de los objetos intermediarios con los métodos "reales" de las clases *builtins*.

Por cada clase intermediaria construida, *implementation_ test.py* se encarga de ejecutar PEF en cada uno de sus métodos, comparando la entrada y la salida (concretizada) contra el método correspondiente de la clase *builtin* real emulada (IntProxy emula enteros, StringProxy emula strings y así sucesivamente). De esta manera, utilizamos a PEF como una **librería** y a las clases reales emuladas como **oráculos** para comparar los resultados obtenidos en los caminos de los métodos de las clases intermediaras. De manera más detallada, "Implementation Test" funciona de la siguiente manera:

1. Busca de manera automática todos los ProxyObjects (objetos intermediarios).

2. Identifica la clase "real" que representa cada ProxyObject. Para una clase proxy c, llamemos $real(c)$ a la clase real emulada.

3. Por cada método m de cada clase ProxyObject c, se utiliza PEF para explorar los caminos del método. Recordemos que estos caminos son en realidad n-uplas ($entrada$, $salida$, $excepcion$, $output$, $condicion\_de\_camino$).

4. Por cada camino explorado, se *concretiza entrada* y *salida*. Para *concretizar* estos valores, construimos un modelo que satisfaga la $condicion\_de\_camino$ y luego instanciamos en *entrada* y *salida* los valores simbólicos pos los asignados por el modelo. Notar que una vez concretizada *entrada*, si se reemplazan las variables de la expresión en *salida* por los valores obtenidos, se obtiene el valor

que corresponde a la salida que el programa hubiera devuelto al ser ejecutado con dicha entrada. (Explicado en 2.5.5).

5. Se ejecuta el método m de la clase $real(c)$ con *entrada* (concretizada) como parámetro y se analiza lo siguiente:
   - Si el método real genera una excepción y el camino explorado generó una excepción del mismo tipo: el camino se considera **correcto**.
   - Si el método real y el camino explorado (*salida* concretizada) retornan el mismo valor: el camino se considera **correcto**.
   - Si no se produce ninguna de las anteriores, el camino es considerado **incorrecto**.
6. Por cada método, si todos los caminos son considerados correctos, también será considerada correcta la implementación del método; de lo contrario la implementación es incorrecta y se marca como un error.

El código fuente de *implementation_test.py* se encuentra a disposición del lector en el apéndice B.1, página 178.

Como el lector puede haber notado, *implementation_test.py* utiliza a PEF para validar las implementaciones internas de PEF. Aunque esto puede sonar extraño y anti-intuitivo, alcanza con tener una mínima porción funcional de BoolProxy (el método *__bool__*, encargado de ramifica la exploración simbólica. Ver 3.2.4) y el algoritmo general de exploración (función *explore*, ver 3.1) para que esta técnica funcione. Creemos interesante notar que además proporcionamos contratos de tipo a los métodos de las clases intermediarias, logrando así, que los test se ejecuten de forma completamente automática.

La salida de *Implementation Test* es bastante intuitiva, por cada clase intermediaria se muestra el resultado de comparar sus métodos con los de las clases reales emuladas. Por cada comparación se detalla el resultado de la misma (✓|✗), el nombre del método ejecutado y la cantidad de caminos explorados. En caso de que la comparación halla sido fallida, se indica cual fue la falla encontrada y con que parámetros es necesario ejecutar la función para replicar el error.

A continuación mostramos la salida que genera Implementation Test al ejecutarlo para la clase StringProxy:

```
> Running automated tests cases for class ListProxy
    ✓  __add__  [paths: 56]
    ✓  __contains__ [paths: 11]
    ✓  __delitem__  [paths: 20]
    ✓  __eq__   [paths: 11]
    ✓  __ge__   [paths: 20]
    ✓  __getitem__  [paths: 121]
    ✓  __gt__   [paths: 19]
    ✓  __iadd__ [paths: 11]
    ✓  __imul__ [paths: 39]
    ✗  __iter__ [paths: 1]
    ERROR FOUND: running with params []
        Expected result: <list_iterator object at 0x111582750>, but
        ProxyObject returned: <generator object __iter__ at 0x1115710f0>
    ✓  __le__   [paths: 19]
    ✓  __len__  [paths: 1]
    ✓  __lt__   [paths: 22]
    ✓  __mul__  [paths: 39]
    ✓  __rmul__ [paths: 39]
    ✓  __setitem__  [paths: 116]
    ✓  append   [paths: 1]
    ✓  clear    [paths: 1]
    ✓  copy [paths: 1]
    ✓  count    [paths: 63]
    ✓  extend   [paths: 11]
    ✓  index    [paths: 62]
    ✓  insert   [paths: 76]
    ✓  pop  [paths: 11]
    ✓  remove   [paths: 31]
    ✓  reverse  [paths: 2]
    ✗  sort [paths: 0]
    ContractError: sort has not type contract and types cannot be infered

 < Runned 27 tests: 25/2 [passed/failed]
```

Capítulo 5

Experimentos

En capítulos anteriores hemos introducido los conceptos teóricos que sustentan a PEF, así también como detallado su interfaz de uso. Si bien hemos mostrado a la herramienta en acción, no nos hemos detenido a medir que tan "bueno" es PEF realizando su trabajo. En este capítulo introduciremos métricas y registraremos el desempeño de **PEF** al ejecutar algunos algoritmos clásicos. También analizaremos los diferentes tipos de programas que hacen uso en mayor o en menor medida las capacidades simbólicas de la herramienta.

Considerando que PEF hace uso de ejecución simbólica y por ende tiene un SMT Solver por detrás, es esperable que la eficiencia de la herramienta sea varios órdenes de magnitud más lenta que la ejecución concreta. También, es necesario recalcar que esta primera iteración de PEF no está centrada en la eficiencia, buscando ser meramente una prueba de concepto. Futuras iteraciones de la herramienta podrán encargarse de estos aspectos.

El capítulo se encuentra dividido de la siguiente manera: en primer lugar, en la sección 5.1, introduciremos las métricas a ser utilizadas; en la sección 5.2, analizaremos en detalle un algoritmo clásico que hace un uso intensivo de la herramienta. Mas adelante, en la sección 5.3, estudiaremos un programa que produce una pequeña sobrecarga en el tiempo en que PEF realiza la exploración. Finalmente, en la sección 5.4, mostraremos el resultado de ejecutar PEF para un gran conjunto de funciones de ejemplo.

5.1. Métricas

Para realizar las mediciones de performance, primero necesitamos fijar las métricas con las cuales juzgaremos el desempeño de nuestra aplicación. Decidimos basar las mediciones en tres puntos claves:

- **Porcentaje código cubierto**

  El objetivo principal de PEF es recorrer un programa e intentar ejecutar la mayor cantidad posible de caminos sobre el mismo. Debido a es esto, la cobertura de código es una de las propiedades fundamentales y buscaremos obtener una cobertura tan alta como sea posible. Existen varios criterios de medir cobertura de código, nosotros utilizaremos 2:

  - *Cobertura de caminos*

    Es la manera ideal de medir cobertura de código. Se registra la cantidad de caminos ejecutados en el programa objetivo. Conseguir que PEF logre un 100 % de cobertura de caminos (es decir que se hayan recorrido todos los caminos del programa), es equivalente a lograr corrección total. Lograr un 100 % de cobertura asegura que todas las postcondiciones serán **probadas o refutadas**.

    La cantidad de caminos explorados por PEF se puede medir de manera directa. Hay tantos caminos explorados (no cortados por máxima profundidad de exploración) como salidas generadas por la función *explore* (3.1).

    Si bien utilizar cobertura de caminos es la mejor forma de medir cobertura de código, la cantidad de caminos de un programa suele ser infinita a causa de los bucles. Si la cantidad de caminos es infinita, medir el porcentaje de caminos recorridos no aporta ningún dato relevante.

  - *Cobertura de ramas*

    Otra forma de medir cobertura de código es mediante la *cobertura de ramas*. En este criterio, buscamos que PEF recorra todos los controles de flujo (*if*, *for*, *while*, ...) al menos una vez por la rama verdadera y por

la falsa. Para el siguiente ejemplo, bastara con ejecutar *f(0)* y *f(2)* para obtener un 100 % de cobertura de ramas (mientras que sólo obtendríamos un 66.66 % de cobertura de caminos)

```
def f(x):
    if x == 0:
        x += 1
    if x == 1:
        x -= 1
    return x
```

- **Recursos utilizados**

  Registraremos el uso de recursos de la herramienta midiendo:

  - Memoria Ram: máxima memoria ram consumida por la herramienta. Realizaremos un promedio de n ejecuciones para evitar datos incorrectos.
  - Tiempo de ejecución: tiempo que demora la herramienta en explorar completamente una función objetivo. Realizaremos un promedio de n ejecuciones para evitar datos incorrectos.

- **Overhead introducido por la herramienta**

  Mediremos la sobrecarga que introduce ejecutar simbólicamente un programa en PEF con respecto a la ejecución concreta. Obtendremos porcentaje de carga extra de CPU y porcentaje extra de memoria ram utilizada.

  Los datos se obtendrán de la siguiente manera:

  1. Se hará un ejecución de PEF para obtener los valores de entrada con los que la herramienta explora la función.
  2. Se concretizarán las entradas generadas en el punto anterior.
  3. Se promediarán los resultados de realizar n veces el siguiente experimento:
     - Se ejecutará PEF y registrará el tiempo y la memoria máxima utilizada.
     - Se ejecutará concretamente la función objetivo con todos los valores de entradas generados por PEF en 2 y se registrará el tiempo total y la memoria máxima utilizada.

4. Finalmente, se cocientará:
   - $\frac{memoria\_ram\_pef}{memoria\_ram\_concreta}$ para obtener el overhead de memoria.
   - $\frac{tiempo\_pef}{tiempo\_concreto}$ para obtener el overhead de tiempo de ejecución.

Todos los experimentos que se mostrarán a continuación fueron realizados en una computadora portátil con procesador i5-4258U @ 2.40GHz, 8GB de ram 1600Hz y sistema operativo OSX 10.10.

5.2. Quicksort

Comenzaremos las mediciones de desempeño con una implementación recursiva de la clásica función de ordenamiento de listas/arreglos *quicksort*. Hemos agregado, a modo de ejemplo, un contrato de postcondición que verifica que el resultado obtenido sea el correcto. El código fuente de la función puede ser encontrado en la figura 5.1. Recordamos al lector para los contratos de PEF, las listas únicamente contienen enteros.

```
1  def quicksort(l):
2      """
3      Quicksort using list comprehensions
4
5      :types: l: list
6      :ensures: returnv == sorted(returnv)
7      """
8      if not l:
9          return []
10     else:
11         pivot = l[0]
12         lesser = [x for x in l[1:] if x < pivot]
13         greater = [x for x in l[1:] if x >= pivot]
14         lesser_sorted = quicksort(lesser)
15         greater_sorted = quicksort(greater)
16         return lesser_sorted + [pivot] + greater_sorted
```

Figura 5.1: Implementación recursiva de QuickSort

Recordemos en que consiste el algoritmo clásico de *quicksort*:

1. Se escoge un elemento de la lista a ordenar como *pivote*. En nuestra implementación este será el primer elemento de la lista (línea 11).

2. Se crean dos listas auxiliares separando los elementos menores y mayores que el pivota (líneas 12 y 13)

3. Se llama de manera recursiva a *quicksort* con la lista de elementos menores y mayores (líneas 14 y 15).

4. Finalmente, se combinan las listas de menores y mayores ordenados con el *pívot* obteniendo una versión ordenada de la lista original de entrada.

Para la implementación de *quicksort* que analizaremos, dada una lista de n elementos a ordenar, en el mejor caso la complejidad [83] es de $\mathcal{O}(n * log(n))$, en el peor de los casos $\mathcal{O}(n^2)$ y en el caso promedio $\mathcal{O}(n * log(n))$.

Ya que *quicksort* es una función de ordenamiento, debe comparar una gran cantidad de elementos (de hecho estas son casi todas sus operaciones). Estas acciones, al ejecutar simbólicamente la función se convierten en comparaciones simbólicas y por ende, en trabajo para el SMT Solver. Debido a esto, analizar *quicksort* resulta extremadamente intensivo para PEF.

5.2.1. Análisis de exploración

Utilizando las métricas que introducimos en la sección 5.1, comenzaremos a realizar pruebas sobre la función *quicksort* de la figura 5.1, pero antes de comenzar, es necesario aclarar algunas decisiones que hemos tomada para realizar las pruebas. Debido a que *quicksort* es una función que itera sobre un arreglo/lista, presenta una cantidad **infinita** de caminos. Hemos decidido *"podar"* estos caminos agregando un contrato de precondición a la función, encargándose el mismo de limitar el tamaño de las listas. El contrato asociado puede apreciarse a continuación:

```
def quicksort(l):
    """
    :assume: len(l) == n
    """
```

De manera que ahora, nuestras pruebas consiste en ejecutar con PEF la función *quicksort* variando el largo de la lista. Ésto no sólo hace que la cantidad de caminos se haga finita (siendo igual a $n!$), sino que también nos permite hacer un análisis

más detallado de como responde la herramienta frente al incremento del tamaño de entradas.

Por cada valor de largo de listas n entre 0 y 6, ejecutamos PEF 5 veces, midiendo promedios en *tiempo de ejecución*, *memoria ram*, *caminos explorados*, *ramas exploradas* y *profundidad de exploración* necesaria para obtener un 100 % de cobertura de caminos. Así mismo, también ejecutamos concretamente la función *quicksort* 5 veces por cada caminos explorado por PEF y compararemos la diferencia de performance entre la ejecución simbólica y concreta (más detalles sobre este procedimiento en 5.1). Al igual que lo realizado con PEF, las ejecuciones concretas se repitieron 5 veces y luego se promediaron los resultados. A continuación, en cada una de las siguientes subsecciones, analizaremos en detalle los aspectos más relevantes de los resultados obtenidos.

Detalle de Caminos y Cobertura

Comenzaremos analizando las métricas de cobertura: *caminos explorados*, *profundidad de exploración* y *porcentaje de ramas exploradas*. A continuación mostraremos una tabla con los datos obtenidos:

Tamaño entrada	Caminos Explorados	Profundidad Exploración	Porcentaje de Ramas Exploradas
0	1	0	16
1	1	0	33
2	2	1	66
3	6	3	100
4	24	6	100
5	120	10	100
6	720	15	100

- **Caminos Explorados**

  Recordemos que para estos casos siempre logramos un 100 % de cobertura de caminos, por lo que este valor representa la cantidad de caminos presentes en *quicksort* para un tamaño de entrada n. Las mediciones obtenidas no presentan ningún patrón extraño, es fácil de calcular que la cantidad de caminos en

quicksort es igual al factorial del largo de la entrada. Esto se debe a que al ordenar hay un camino por cada orden diferente del arreglo/lista (si la misma tiene n elementos hay $n!$ permutaciones).

- **Profundidad de Exploración**

  La profundidad de exploración se encarga de medir la cantidad de decisiones no forzadas que fueron tomadas durante la ejecución, en este caso para alcanzar 100 % de cobertura de caminos. Notar lo pequeño que es este valor en comparación a la cantidad de caminos explorados (es de alrededor al 2 % de la cantidad de caminos cuando la lista tiene 6 elementos). Específicamente para *quicksort* la profundidad de exploración se encuentra descripta por $\frac{n^2-n}{2}$.

- **Porcentaje de Ramas Exploradas**

  La tabla nos hace notar que basta ejecutar *quicksort* con 3 elementos para alcanzar un 100 % de cobertura de ramas. Esto es relevante, pues nos indica que no es necesario seguir ejecutando simbólicamente la función con listas más largas para obtener este porcentaje. Notemos que para alcanzar el 100 % de cobertura de ramas sólo hace falta buscar con una profundidad de exploración de también 3. La relación entre profundidad de exploración y cobertura de código seguirá siendo analizada para otros ejemplos en la sección 5.4.

Detalles de Tiempos de Ejecución

Comenzaremos presentando en una tabla los resultados obtenidos en las mediciones de tiempos de ejecución. La primer columna muestra el largo de la lista que recibe como argumento la función; la segunda, muestra el tiempo total (en milisegundos) para realizar la ejecución simbólica de todos los caminos; la tercera columna muestra el tiempo (en microsegundos) total que le llevó al sistema ejecutar la función de manera concreta para cada camino explorado en la etapa simbólica; la cuarta y última columna muestra el porcentaje de overhead del tiempo de ejecución simbólico respecto al concreto. Es importante notar que **la segunda y la tercera columna se presentan en unidades de tiempo diferentes**.

Tamaño entrada	Tiempo PEF total [ms]	Tiempo Concreto total [μs]	Porcentaje overhead de tiempo
0	165,29	195,59	84408,41
1	309,68	185,1	$1{,}67 \cdot 10^5$
2	641,28	196,24	$3{,}27 \cdot 10^5$
3	2263,67	221,02	$1{,}02 \cdot 10^6$
4	10610,27	398,12	$2{,}66 \cdot 10^6$
5	62431,88	1522,04	$4{,}1 \cdot 10^6$
6	$4{,}7 \cdot 10^5$	9430,11	$4{,}98 \cdot 10^6$

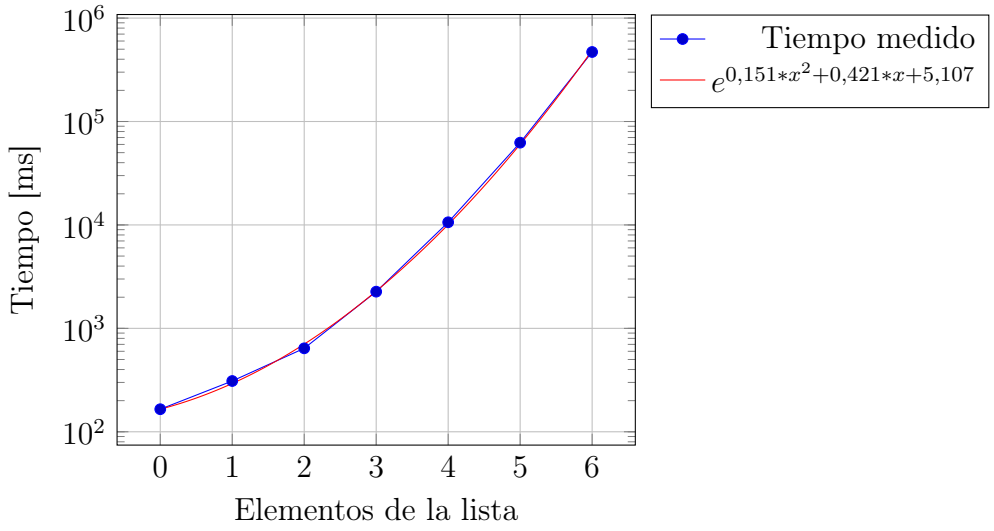
Dividiremos el análisis sobre el tiempo de ejecución en dos: **tiempo de ejecución en función del tamaño de la entrada** y observaciones sobre la **sobrecarga que introduce PEF** frente a la ejecución concreta. Analicemos cuidadosamente los tiempos de ejecución de PEF en la tabla y en la figura 5.2. Claramente se observa que a medida que más grande es la entrada más tiempo es necesario para explorar la función. De hecho, los incrementos en los tiempos parecen ser mayores a los de una exponencial usual (e^x). Utilizando la técnica de regresión exponencial y mínimos cuadrados, hemos encontramos que la función exponencial que mejor se ajusta a la relación entre tamaño de entrada y tiempo necesario para explorar esta descripta por la siguiente función $f(x) = e^{0{,}151*x^2+0{,}421*x+5{,}107}$. Si bien los tiempos de ejecución de PEF que hemos mesurado son extremadamente altos, esto es esperable pues *quicksort* hace un uso intensivo de decisiones que involucran variables simbólicas.

Ahora analizaremos la tercera y más interesante columna de la tabla de tiempos. Si bien el tiempo que demora PEF en explorar una función depende de la misma, si analizamos el overhead que PEF introduce con respecto a la ejecución concreta podemos independizarnos un poco (no totalmente) de la de la función en cuestión.

Los datos obtenidos fueron, en principio, contra-intuitivos. Los autores esperaban que al igual que con los tiempos de ejecución se obtuviera un crecimiento exponencial del overhead. Sin embargo, observando la figura 5.3, podemos notar que existen tres zonas bien diferenciadas:

1. De 0 a 2: Se observa un crecimiento pequeño del overhead. El pendiente del

Figura 5.2: Tiempo consumido en función del largo de entrada



crecimiento es positivo. Esto es razonable ya que para esta cantidad reducida de elementos hay cantidades similares de caminos.

2. De 2 a 4: Se observa un gran incremento del overhead. La pendiente de crecimiento es positivo y mayor que en la zona anterior. La diferencia entre cantidad de caminos entre valor a valor es cada vez mayor.

3. De 4 a 6: Se continúa observando crecimiento, aunque esta vez la pendiente de crecimiento es negativa.

No somos capaces de explicar por qué se da una disminución del crecimiento del overhead en la tercer zona de la función. De igual manera, la sobrecarga introducida por PEF es sumamente alta, variando desde el 84400 % hasta 4900000 %. Como ya hemos mencionado, estos overheads tan significativos están ligados a la gran cantidad de comparaciones simbólicas que ocurren dentro de la función analizada y resultan en un mayor trabajo de z3. En la sección 5.3 mostraremos una función que, a diferencia de *quicksort*, introduce un overhead mínimo al ser ejecutada simbólicamente por PEF.

Detalle de Memoria Ram Consumida

Con respecto a la memoria consumida, mostraremos en la siguiente tabla la cantidad de memoria ram utilizada por PEF y por las ejecuciones concretas al variar

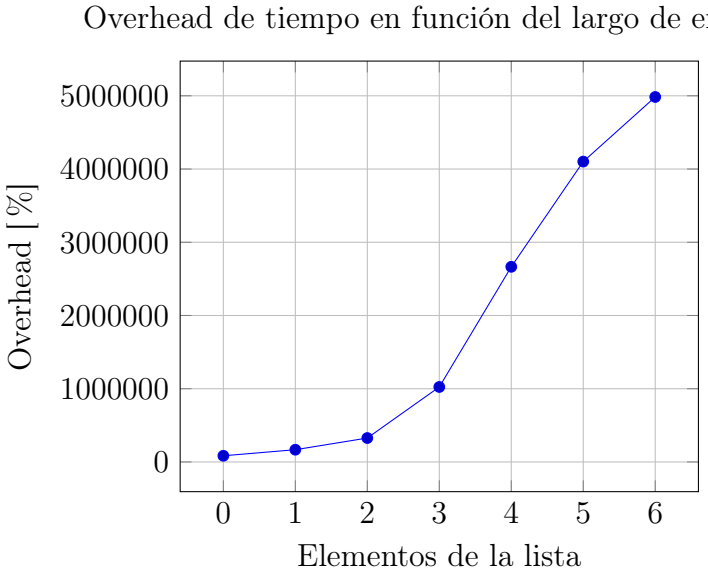


Figura 5.3: Overhead de tiempo en función del largo de entrada. Los datos, en principio contraintuitivos nos hacen notar que el overhead parece estabilizarse para tamaños mayores de listas.

el largo de las listas entre 0 y 6.

Tamaño entrada	Memoria utilizada por PEF [MB]	Memoria ejecución Concreta [KB]	Overhead Memoria [%]
0	19,56	33,35	59958
1	24,26	33,26	74591
2	26,94	33,17	83067
3	27,31	33,81	82613
4	27,44	33,09	84815
5	27,93	32,36	88281
6	28,82	34,4	85679

Al igual que con el tiempo de ejecución, dividiremos el análisis en dos partes: variación de la memoria con respecto al tamaño de entrada, y variación de la sobrecarga en función al tamaño de la entrada.

Si analizamos los datos de la tabla u observamos la figura 5.4 podemos hacer varias observaciones:

- La carga inicial de PEF, principalmente contribuida por la carga de z3, es de alrededor de 20 MB.

- Aumentar el tamaño de las entradas provoca el incremento de la memoria RAM consumida.
- El consumo de RAM de la herramienta se encuentra entre 20MB y 30MB.
- Los datos obtenidos sugieren que el consumo de memoria de la herramienta se presenta en forma de exponencial negativa.

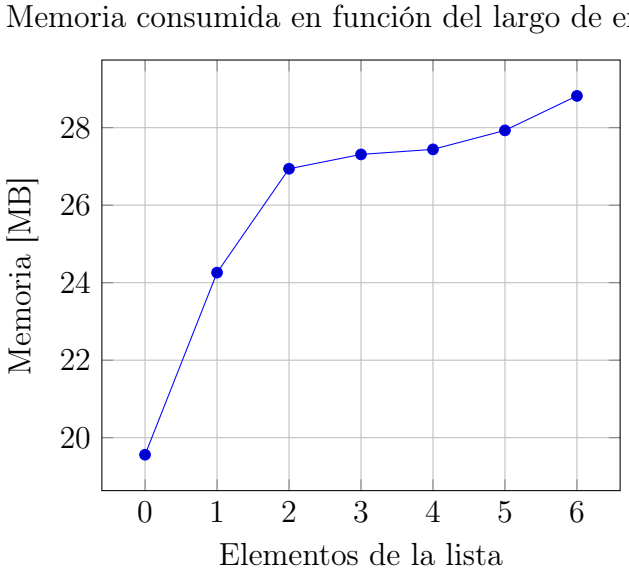


Figura 5.4: Memoria consumida en función del largo de entrada

Si ahora intentamos comparar la memoria consumida por PEF contra la memoria consumida por ejecuciones concretas, nuevamente nos encontraremos con una diferencia gigantesca. Notemos que la memoria de PEF está expresada en MB y la concreta en KB. Al igual que en el caso del consumo de RAM y como puede observarse en la figura 5.5, el overhead se presenta en forma aparente de exponencial negativa. Sin embargo y a diferencia del caso anterior, **el overhead oscila**; de largo 2 a 3 y de largo 5 a 6 el mismo disminuye. Analizando los datos obtenidos, podemos considerar que el overhead de memoria se encuentra alrededor del 80000 %, una cifra despreciable si tenemos en cuenta que el overhead de tiempo rondaba el 4900000 %. Notemos que por los gráficos de sobrecarga obtenidos, los overheads de tiempo y memoria no se comportan de igual manera al aumentar el tamaño de la lista de entrada.

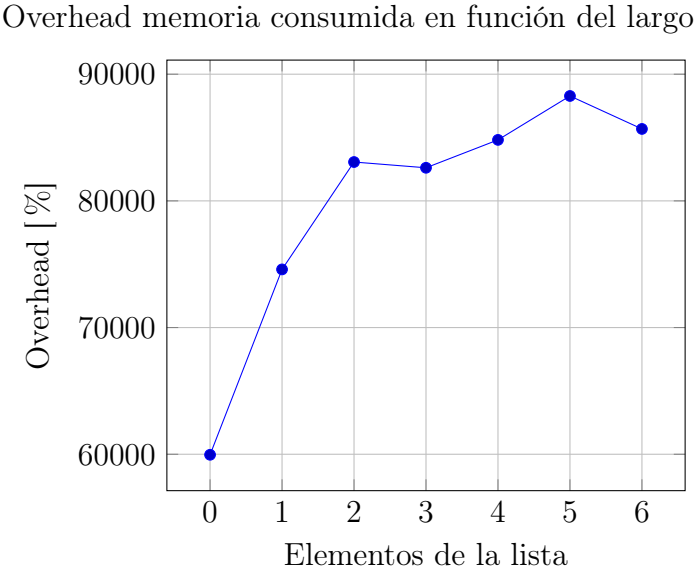


Figura 5.5: Overhead memoria consumida en función del largo de entrada

5.2.2. Impacto de Optimizaciones

Hemos mostrado en 5.2.1 que el overhead introducido por PEF al ejecutar quicksort es realmente considerable. Si bien debemos recordar que esta primera iteración no esta pensada para ser eficiente y que la ejecución simbólica es inherentemente lenta, es posible analizar el impacto de algunas optimizaciones sobre el desempeño de la herramienta. Analizaremos como es afectada la performance de PEF con la introducción conjunta (teórica) de dos optimizaciones propuestas en 6.3: salvar el estado de ejecución y paralelizar la ejecución de PEF.

Al momento de presentar este documento, PEF posee una implementación puramente secuencial. Cada vez que se presenta una posible bifurcación en la ejecución simbólica, se continúa por uno de los caminos, programando la otra posibilidad para una futura ejecución. Considerando lo anterior, el código de PEF es altamente paralelizable. La idea, esencialmente, es cada vez que se llega a una bifurcación donde ambos caminos son posibles, continuar con el hilo actual por uno de los caminos e iniciar un nuevo hilo de ejecución para ejercitar el otro. Suponiendo que tenemos acceso a una cantidad infinita de threads, no sólo la implementación se parecería más a la ejecución simbólica presentada en 2.5, sino que los tiempos de ejecución de la herramienta bajarían considerablemente (a costa de más carga en CPU). Notar

que para hacer más fáciles los cálculos supondremos que podemos almacenar el estado del programa, y por lo tanto, cuando iniciamos un nuevo thread este comienza desde el momento de la bifurcación.

Para comenzar a tener una idea intuitiva de la ganancia que podría obtenerse paralelizando el código, hemos realizado mediciones registrando el tiempo promedio (5 ejecuciones) que demora cada camino simbólico en ser ejecutado (figura 5.6). De esta manera si bien explorar simbólicamente *quicksort* con una lista de 5 elementos demora 62.43s, recorrer cada camino de la misma demora sólo 601.39ms en promedio.

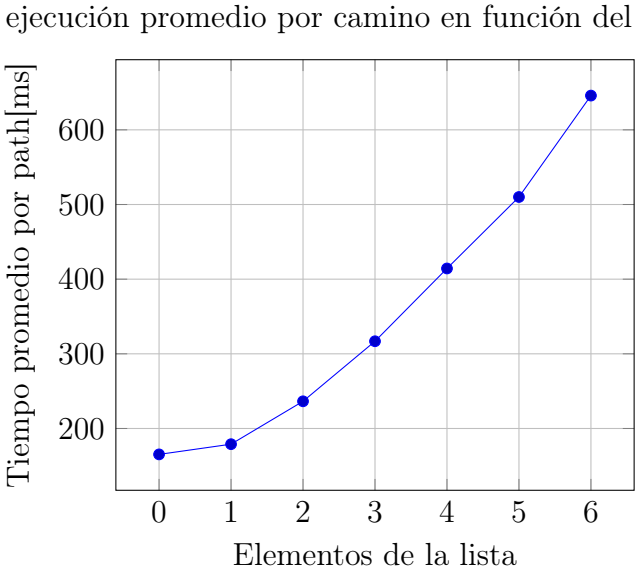


Figura 5.6: Tiempo promedio de ejecución de cada camino en función del tamaño de entrada

En realidad, y para ser más exactos, como todos los caminos se ejecutan en paralelo el cuello de botella paralelizando PEF esta dado por el thread (camino) más lento. Por esto y para conseguir una cota más exacta de la posible ganancia en performance que se puede lograr, medimos el tiempo que toma recorrer el camino más largo de la función a explorar. Nuevamente, realizamos las mediciones de este valor promediando 5 ejecuciones de PEF obteniendo los resultados presentados a continuación y observables en la figura 5.7.

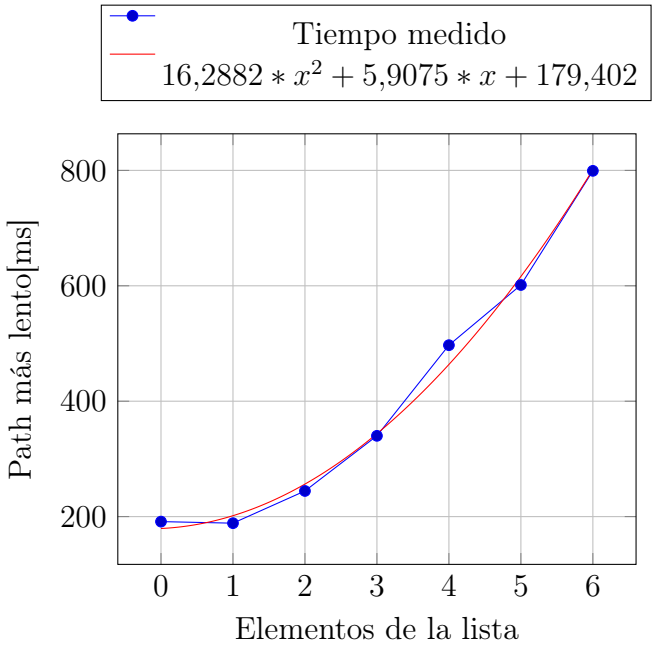


Figura 5.7: Tiempo de ejecución del camino más lento en función del tamaño de entrada

Tamaño lista	Tiempo PEF [ms]	Tiempo camino más largo [ms]
0	165,29	191,34
1	309,68	188,62
2	641,28	244,44
3	2263,67	340
4	10610,27	497,09
5	62431,88	601,39
6	$4{,}7 \cdot 10^5$	799,22

Es fácil de notar que a medida que crece la cantidad de elementos de la lista, mayor es la diferencia porcentual entre el tiempo de PEF y el de PEF paralelizado teórico. Si consideramos un tamaño de lista de 6 elementos se disminuye el tiempo de ejecución de 7.834 minutos a 0.799 segundos, una mejora del 58700 %.

Más aún, si analizamos los gráficos de tiempo de PEF usual vs PEF paralelizado (teórico) podemos notar, que el tiempo pasa de incrementarse exponencialmente a hacerlo cuadráticamente de acuerdo a la siguiente función $16{,}2882x^2 + 5{,}9075x + 179{,}402$. La aproximación fue conseguida por mínimos cuadrados.

5.3. Sobrecarga y Performance

Al analizar los resultados de PEF ejecutando la función *quicksort* se obtienen resultados sumamente altos en términos de overhead de tiempo y memoria. En esta sección mostraremos que ésto está fuertemente relacionado al tipo de programa que *quicksort* es. Mostraremos un programa que hace uso en menor medida de las operaciones simbólicas y mediremos cual es el overhead introducido en este caso por la herramienta.

La función *adivina* de la figura 5.8 toma como parámetro un entero, genera un millón de números aleatorios (función *random.randint*) y luego verifica si la entrada ingresada es igual a un nuevo número aleatorio entre 0 y 42, imprimiendo el mensaje "Adivinaste!" en caso afirmativo. Esta función que en principio es difícil verificar con tests de unidad, es muy fácil de analizar y verificar con PEF.

```
1 def adivina(x) :
2     """
3     :type: x: int
4     """
5     import random
6     for i in range(0, 1000000):
7         random.randint(0, 42)
8     if x == random.randint(0, 42):
9         print("Adivinaste!")
```

Figura 5.8: Ejemplo donde la mayoría de las operaciones no dependen de la entrada

Ahora nos encargaremos de ejecutar la función *adivina* con PEF y de manera concreta; mediremos *tiempo de ejecución* y *memoria RAM consumida*, así también como sus respectivas sobrecargas. A continuación presentamos los resultados obtenidos al promediar 5 ejecuciones simbólicas (hechas por PEF) y 10 ejecuciones concretas de la función *adivina*. Notar que cuando ejecutamos concretamente nos hacen falta dos ejecuciones secuenciales de la función, ya que cada ejecución simbólica ejecuta la función 2 veces, una por cada rama del *if* de la línea 8.

Recurso	PEF	Concreto	Overhead [%]
Tiempo	5.057363 s	5.087728 s	-0.5968 %
Memoria	20.40 MB	6.89 MB	196 %

Los datos obtenidos claramente muestran que **la sobrecarga de tiempo introducido por PEF para este ejemplo es despreciable**. De hecho, en las mediciones se obtiene un overhead negativo, el cual no tiene sentido y probablemente sea producto de la administración interna de procesos y CPU del sistema operativo. La razón de la poca diferencia en el tiempo de ejecución es que la ejecución simbólica de esta función requiere únicamente una comparación simbólica en línea 8, el resto del programa se ejecuta de manera idéntica para ambos casos.

Con respecto a la memoria RAM, el overhead introducido medido es relevante, de alrededor del 200 %. Sin embargo, es necesario mencionar que si lo comparamos con la sobrecarga mínima medida en *quicksort*, de alrededor de 59000 %, este número se vuelve despreciable.

Como conclusión, podemos afirmar que los overheads introducidos, y por ende la performance de PEF, son sumamente dependientes de la función a analizar. Se obtienen overheads mínimos cuando la función hace un uso mínimo de las capacidades simbólicas; y se obtienen overheads extremadamente grandes cuando la función hace uso intensivo de las capacidades simbólicas.

5.4. Otros Resultados

En este capítulo hemos analizado con profundidad el rendimiento de PEF en dos casos que podríamos llamar extremos: *quicksort* y *adivina*. En unos de los casos la performance es la peor que podamos medir y en otro la mejor. En esta sección, mostraremos los resultados de performance de PEF explorando otras funciones.

A lo largo del desarrollo de la herramienta hemos creado un archivo con 30 ejemplos que se encargaban de probar diferentes capacidades de la misma. Ninguna de estas funciones posee "podadas" sus entradas, como quicksort, y en muchas de ellas la cantidad de caminos a explorar es infinita. Mostraremos ahora los resultados de ejecutar PEF sobre estas funciones. A diferencia de las secciones anteriores, aquí sólo mostraremos los datos y haremos un pequeño análisis superficial. El archivo de ejemplos, *examples.py*, se encuentra disponible con el código fuente de PEF o en el apéndice C.1. Es importante remarcar que dentro de las funciones de ejemplo se

encuentran las de bibliografía *sum1* y *power* tomadas del paper de James King [1], y la función *abs1* del paper de Bruni [2].

Al igual que en con *quicksort*, dividiremos los resultados en tres partes: mediciones de tiempo, mediciones de memoria y mediciones de exploración.

Mediciones de Exploración

Para realizar todos los experimentos que veremos a continuación se limitó la **profundidad máxima de exploración** de PEF a 10. Luego de ejecutar 5 veces las funciones tanto simbólica como concretamente [1], se obtuvieron los resultados presentados en la tabla de la figura 5.9. En la misma se presenta la función a medir, la cantidad de caminos explorados, la profundidad máxima con que PEF exploró la función y la cantidad de caminos podados.

La cantidad de caminos nos habla mucho sobre el tipo de función que estamos midiendo, ya que la profundidad de exploración esta limitada, una función con una gran cantidad de caminos es una función con una cantidad mayor de operaciones simbólicas y un árbol de ejecución más ancho.

Las columnas *profundidad de exploración* y *caminos podados* deberían leerse juntas. Notemos que en las funciones de prueba, las únicas que necesitaron podar caminos, pues llegaron a la profundidad máxima de exploración, fueron aquellas funciones que presentaban una cantidad de caminos infinitas, y su poda fue acertada. Podemos considerar que la profundidad de exploración de 10 fue suficiente para estas funciones de ejemplo, sin embargo este tamaño puede no ser suficiente para otras funciones. PEF permite que los usuarios puedan modificar el valor de este tamaño máximo de exploración.

5.4.1. Mediciones de Tiempo

Luego de ejecutar 5 veces las funciones tanto simbólica como concretamente, se obtuvieron los resultados presentados en la tabla de la figura 5.10. En la misma se

[1] Cuando decimos ejecutar 5 veces la función concretamente nos referimos a ejecutar 5 veces la función de manera concreta por cada camino explorado por PEF. Más información sobre este procedimiento puede ser encontrado en 5.1

Función	Caminos Explorados	Profundidad de Exploración	Caminos Podados
simple	2	1	0
simple_bool	1	0	0
fun	2	1	0
funab	2	1	0
funfor	11	10	1
funlist_in	7	4	0
funlist_iter	11	10	1
funlist_slice	103	9	0
assert_not_divisible	19	10	2
divisible	21	10	2
test_Test	4	2	0
test_mock_Test	4	2	0
test_gt	2	1	0
test_kw_args	2	1	0
simple_test	2	1	0
example00	36	10	6
example01	118	10	38
example02	3	2	0
multitypes	4	0	0
test_strings1	20	10	2
print_species	11	10	1
sum_lista	20	10	0
insertLeft	64	10	0
sum_fac	11	10	1
sum_fac2	11	10	1
test_sum_fac	11	10	1
fact_spec	11	10	1
faulty_fact	11	10	1
suma_con_bug	4	2	0
f_l_j	4	2	0
sum1	1	0	0
power	11	10	1
abs1	2	1	0

Figura 5.9: Información sobre los caminos explorados para las funciones de ejemplo. Podemos apreciar cantidad de caminos explorados, profundidad de exploración alcanzada y cantidad de caminos podados.

presenta la función a medir, el tiempo que demoró PEF en explorarla, el tiempo concreto que se necesita para ejecutar todos los caminos recorridos por PEF y el overhead de tiempo.

El overhead promedio medido fue de 3595674 %, mientras que la mediana (quizás más descriptiva) fue de 1434171 %. Si analizamos detalladamente las tres funciones con menor overhead: *multitypes*, *simple* y *fun*, podemos observar que las tres funciones son sumamente sencillas, presentando todas una cantidad finita de caminos.

Si ahora buscamos las tres funciones con mayor overhead: *sum_fact2*, *funlist_slice* e *insertLeft*, es un poco más difícil encontrar algún patrón entre ellas. En el caso de *insertLeft*, es una función muy sencilla pero necesita construir instancias simbólicas de una clase recursiva creada por el usuario, por lo que presenta una cantidad infinita de caminos. La función *sum_fact2* es una función recursiva que calcula el factorial por lo que también posee una cantidad infinita de caminos. El caso de *funlist_slice* es un poco especial, esta función que toma un slice simbólico sobre una lista simbólica, su overhead tan grande se debe al arduo razonamiento sobre listas que debe hacer. De hecho si miramos el overhead de otras funciones que usen listas, notaremos que todas tiene un gran overhead. Concluyendo, PEF no se lleva bien con las funciones recursivas y las listas ya que generalmente involucran una cantidad infinito de caminos y necesitan que la herramienta pode la ejecución desperdiciando mucho razonamiento.

Mediciones de Memoria

Luego de ejecutar 5 veces las funciones tanto simbólica como concretamente, se obtuvieron los resultados presentados en la tabla de la figura 5.11. En la misma se presenta la función a medir, la memoria consumida por PEF en explorarla, la memoria necesaria para ejecutar concretamente todos los caminos recorridos por PEF y el overhead de memoria. Las entradas de la tabla que poseen un "-" no pudieron ser medidas debido a una limitación con la librería *pickle* [2] de Python.

El overhead promedio medido fue de 75382 %, mientras que la mediana fue de

[2]El módulo *pickle* de Python implementa un algoritmo poderoso para serializar y deserializar objetos. Documentación del módulo: `https://docs.python.org/2/library/pickle.html`

Función	Tiempo PEF [s]	Tiempo Concreto [s]	Overhead [%]
simple	$3{,}5 \cdot 10^{-2}$	$4{,}2 \cdot 10^{-5}$	83250
simple_bool	$6{,}68 \cdot 10^{-3}$	$2 \cdot 10^{-6}$	$3{,}34 \cdot 10^{5}$
fun	$3{,}13 \cdot 10^{-2}$	$3{,}7 \cdot 10^{-5}$	84556,76
funab	$4{,}06 \cdot 10^{-2}$	$6 \cdot 10^{-6}$	$6{,}76 \cdot 10^{5}$
funfor	1,31	$4{,}5 \cdot 10^{-5}$	$2{,}92 \cdot 10^{6}$
funlist_in	0,37	$7{,}2 \cdot 10^{-5}$	$5{,}17 \cdot 10^{5}$
funlist_iter	1,45	$3{,}08 \cdot 10^{-4}$	$4{,}72 \cdot 10^{5}$
funlist_slice	31,71	$2{,}52 \cdot 10^{-4}$	$1{,}26 \cdot 10^{7}$
assert_not_divisible	2,46	$1{,}57 \cdot 10^{-3}$	$1{,}56 \cdot 10^{5}$
divisible	2,26	$1{,}33 \cdot 10^{-3}$	$1{,}7 \cdot 10^{5}$
test_Test	0,21	$2{,}2 \cdot 10^{-5}$	$9{,}68 \cdot 10^{5}$
test_mock_Test	0,2	$1{,}4 \cdot 10^{-5}$	$1{,}43 \cdot 10^{6}$
test_gt	$4{,}69 \cdot 10^{-2}$	$4 \cdot 10^{-6}$	$1{,}17 \cdot 10^{6}$
test_kw_args	$4{,}93 \cdot 10^{-2}$	$3 \cdot 10^{-6}$	$1{,}64 \cdot 10^{6}$
simple_test	$4{,}79 \cdot 10^{-2}$	$4 \cdot 10^{-6}$	$1{,}2 \cdot 10^{6}$
example00	4,34	$2{,}27 \cdot 10^{-4}$	$1{,}91 \cdot 10^{6}$
example01	43,73	$1{,}06 \cdot 10^{-3}$	$4{,}11 \cdot 10^{6}$
example02	$9{,}06 \cdot 10^{-2}$	$5 \cdot 10^{-6}$	$1{,}81 \cdot 10^{6}$
multitypes	$2{,}48 \cdot 10^{-2}$	$7 \cdot 10^{-5}$	35392,86
test_strings1	4,52	$6{,}8 \cdot 10^{-5}$	$6{,}65 \cdot 10^{6}$
print_species	1,54	$5{,}82 \cdot 10^{-4}$	$2{,}64 \cdot 10^{5}$
sum_lista	1,84	$1 \cdot 10^{-4}$	$1{,}84 \cdot 10^{6}$
insertLeft	76,3	$1{,}65 \cdot 10^{-4}$	$4{,}62 \cdot 10^{7}$
sum_fac	0,95	$6{,}3 \cdot 10^{-5}$	$1{,}5 \cdot 10^{6}$
sum_fac2	3,25	$4{,}6 \cdot 10^{-5}$	$7{,}06 \cdot 10^{6}$
test_sum_fac	3,98	$1{,}33 \cdot 10^{-4}$	$2{,}99 \cdot 10^{6}$
fact_spec	0,81	$1{,}4 \cdot 10^{-5}$	$5{,}75 \cdot 10^{6}$
faulty_fact	0,82	$2{,}5 \cdot 10^{-5}$	$3{,}28 \cdot 10^{6}$
suma_con_bug	0,11	$4{,}2 \cdot 10^{-5}$	$2{,}54 \cdot 10^{5}$
f_l_j	0,42	$9 \cdot 10^{-6}$	$4{,}67 \cdot 10^{6}$
sum1	$6{,}26 \cdot 10^{-3}$	$2 \cdot 10^{-6}$	$3{,}13 \cdot 10^{5}$
power	0,79	$1{,}8 \cdot 10^{-5}$	$4{,}38 \cdot 10^{6}$
abs1	$3{,}55 \cdot 10^{-2}$	$3 \cdot 10^{-6}$	$1{,}18 \cdot 10^{6}$

Figura 5.10: Resultados de tiempo para las funciones de ejemplo. Podemos apreciar tiempo simbólico, tiempo concreto y overhead de tiempo.

80321.17 %. Al igual que con el tiempo presentaremos las funciones que mayor y menor overhead presentan. Las tres funciones que menor overhead de tiempo presentaron fueron: *simple_bool*, *sum1* y *multitypes*. Las tres funciones que mayor overhead obtuvieron fueron *test_strings1*, *funfor* y *power*. En este caso el análisis de los datos es muy sencillo, las funciones que menor cantidad de caminos exploran, y por ende resuelven menos *constraints* a lo largo de la ejecución, son la que menor overhead presentan; mientras que las funciones que hacen lo inverso, presentan los overheads más altos.

Función	Memoria PEF [MB]	Memoria Concreta [KB]	Overhead [%]
simple	21.33	32.78	66531.84
simple_bool	16.97	33.42	51896.64
fun	22.11	33.58	67322.98
funab	22.12	32.68	69211.13
funfor	27.56	33.20	84904.33
funlist_in	27.12	33.99	81603.08
funlist_iter	27.48	34.37	81772.33
funlist_slice	29.35	36.82	81525.20
assert_not_divisible	27.78	38.96	72915.19
divisible	27.67	38.68	73152.53
test_Test	-	-	-
test_mock_Test	26.14	33.69	79351.94
test_gt	22.87	33.06	70737.50
test_kw_args	21.14	33.32	64868.06
simple_test	-	-	-
example00	27.46	33.85	82969.54
example01	-	-	-
example02	24.56	33.10	75880.18
multitypes	20.83	33.57	63438.63
test_strings1	29.22	33.38	89538.34
print_species	27.15	34.57	80321.17
sum_lista	-	-	-
insertLeft	-	-	-
sum_fac	27.25	33.55	83071.38
sum_fac2	27.43	33.58	83545.97
test_sum_fac	27.42	33.48	83765.23
fact_spec	27.14	33.26	83457.90
faulty_fact	27.13	33.26	83427.11
suma_con_bug	22.95	33.39	70282.74
f_l_j	26.73	33.18	82394.03
sum1	17.36	33.252	53368.06
power	27.22	32.94	84520.61
abs1	20.91	32.92	64949.20

Figura 5.11: Resultados de memoria para las funciones de ejemplo. Podemos apreciar memoria simbólica, memoria concreta y overhead de memoria.

Capítulo 6

Trabajos relacionados, aportes y propuestas

En este último capítulo, y como su título lo sugiere, analizaremos trabajos relacionados, los aportes que este trabajo brinda y propuestas de trabajos futuros. Este capítulo 6 se encuentra dividido de la siguiente manera: en la sección 6.1 de comenzaremos revisando algunos trabajos relacionados, desde el primer estudio sobre ejecución simbólica, hasta la actualidad, donde se usan técnicas que combinan ejecución simbólica con valores concretos; en la sección 6.2, revisaremos los aportes realizados en este trabajo; luego, en 6.3, mostraremos propuestas para mejorar diferentes aspectos de la herramienta. Finalmente, en sección 6.4 se encuentra una pequeña conclusión del trabajo.

6.1. Trabajos Relacionados

La ejecución simbólica fue estudiada originalmente por King [1] en el año 1976, quién construyó un ambiente de programación llamado *EFFIGY* que permitía interpretaciones simbólicas para los programas escritos en su lenguaje.

A partir del año 2000 ha habido un creciente interés por la ejecución simbólica. Meudec [8] usa programación lógica con restricciones y ejecución simbólica para generar datos para la verificación de forma automática. Balser [9] prueba propiedades en sistemas concurrentes de manera interactiva. Khurshid [10] instrumenta código

java en Java Pathfinder para combinar ejecución simbólica con verificación formal y combatir la explosión de estados. Pasareanu y Visser [11] desarrollaron un método para encontrar y probar invariantes de ciclos en código Java. Berdine [12] usa ejecución simbólica junto a lógica de separación para probar ternas de Hoare de forma automática.

También ha crecido el interés en utilizar la ejecución simbólica para generar tests de forma automática. Xie [3] presenta un entorno de trabajo llamado Symstra capaz de generar tests de unidad orientados a objetos usando ejecución simbólica. Tillmann y Schulte [4] describen tests de unidad parametrizados combinados con ejecución simbólica como forma de guiar la generación de tests de forma automática. Godefroid [7] ha utilizado la ejecución simbólica para generar suite de tests para programas escritos en C logrando gran cobertura.

Nuevamente mejorando a JavaPahFinder, Anand [13] incorporó aspectos de ejecución simbólica a la herramienta, estudiando y desarrollaron una manera de interpretar estructuras de datos, como las clases de Java, a través de instanciación *lazy* de referencias. Pasareanu [14], extendió el trabajo hecho por Anand, pero en vez de instrumentar código, su sistema es un intérprete “no-estándar"del *bytecode* [28], usando las ejecuciones concretas a nivel del sistema para mejorar la generación de tests de unidad.

Similar al trabajo hecho sobre Java Pathfinder, Tillmann y Halleux [15] implementaron un motor de ejecución simbólica para la plataforma .NET [27] llamado Pex, usando el razonador Z3 [58] para resolver las restricciones.

La técnica de ejecución simbólica utilizada en PEF fue introducida por Bruni, Disney y Flanagan [2] en el 2011. Nuestro trabajo está inspirado por este paper, el cual propone la ejecución simbólica con objetos intermediarios para algunas clases nativas de Python.

Otra técnica relacionada con la ejecución simbólica es la denominada ejecución *concólica* (en inglés, *concolic execution*), que combina la ejecución simbólica con la concreta para complementar las áreas donde la ejecución puramente simbólica se ve limitada. Sen [6] estudió y desarrolló el motor para tests de unidad Concólico para programas escritos en C, CUTE, que agrega un paso de ejecución concreta

con valores obtenidos luego del análisis simbólico; además, CUTE, permite hacer ejecuciones simbólicas sobre programas con entradas de datos dinámicas (punteros). SAGE [82] es una propuesta de Microsoft que ejecuta de manera concólica binarios de Windows. KLEE [47] actúa como un intérprete para ejecutar de manera concólica el bytecode producido por LLVM [29]. Pex [15] adopta una metodología de instrumentación dinámica para ejecutar de manera concólica código .NET. Conpy [52] se basa, al igual que PEF, en el principio de Python de que "todo es un objeto" para crear clases que extienden los objetos nativos de Python con atributos adicionales para indicar los valores simbólicos relacionados al objeto. El paper no detalla mucho la implementación del motor de ejecución, pero por lo que entendemos, es similar a una ejecución de PEF, donde las ejecuciones son con valores concretos, durante la cual se registran las expresiones de los controles de flujo. Al final de una ejecución, se deducen las condiciones necesarias para pasar por un camino nuevo, y se pide un modelo para empezar de nuevo (por otra rama). Se puede entender como una forma inversa a la que realiza PEF, pero curiosamente, el hecho de que en Python todo sea un objeto implica que para ambas técnicas, no haya gran diferencia entre valores simbólicos o "reales". Por lo que la metodología concólica no aporta beneficios reales para la técnica de arquitectura de pares.

6.2. Aportes

Este trabajo presenta dos grandes contribuciones: la extensión de la técnica de arquitectura de pares para clases definidas por el usuario; y la creación de PEF, una herramienta que utiliza la arquitectura de pares mejorada para detectar errores en *Python* de manera automática.

La arquitectura de pares es una técnica de ejecución simbólica para lenguajes interpretados desarrollada por Bruni, Disney y Flanagan [2] en el 2011. Las clases definidas por el usuario eran consideradas uno de los mayores impedimentos [5] para considerarla una técnica realmente útil. En este trabajo, presentamos en 3.4 un algoritmo recursivo que permite de manera sencilla mitigar las limitaciones antes mencionadas.

El segundo aporte mencionado, PEF, es una herramienta para detección automática de errores, utilizando contratos y ejecución simbólica, en programas escritos en Python. La misma funciona como una prueba de concepto de la extensión de la técnica de arquitectura de pares. El desarrollo de la herramienta se muestra a lo largo del capítulo 3. El código fuente de la herramienta es provisto junto con este trabajo.

6.3. Propuestas

Varias mejoras pueden ser aplicadas a PEF para lograr mayor eficiencia, cobertura y soporte para distintos tipos de programas. A continuación mencionaremos algunas de las más importantes.

6.3.1. Mejoras de rendimiento

- Guardar estados de ejecución: Si se guardan las colas de condición de camino (*path_ condition*) se pueden iniciar las ejecuciones desde alguna sentencia deseada sin necesidad de tener que volver a ejecutar todo el camino hasta ese punto (siempre que se haya ejecutado antes). Esto brindaría una gran mejora de performance.

- Multi-threads: Es posible utilizar threads para paralelizar la exploración de cada rama del programa. Esta mejora aportaría una importantísima reducción del tiempo de exploración.

El impacto de estas dos mejoras combinadas ha sido analizado en detalle para la función *quicksort* en la sección 5.2.2.

6.3.2. Mejoras de cobertura

- Post-condición: Referencia a estados de inicio y fin de la ejecución en las variables de las expresiones: Actualmente las post-condiciones están muy limitadas, sólo pueden referirse al valor que tienen las variables de entrada al comenzar la ejecución de la función.

- Aliasing: PEF no razona sobre aliasing. Pero es posible implementarlo y aumentar la cobertura de caminos.

- Validar contratos en llamadas a funciones durante la ejecución: Verificar los contratos de las funciones que son llamadas durante una exploración brindaría una mayor cobertura de exploración y robustez en la herramienta.

6.3.3. Otras Mejoras

- Construir el árbol de ejecución: Las salidas de PEF contienen casi toda la información necesaria para recrear los árboles de ejecución de manera ilustrativa, incluso en tiempo real. Esta mejora puede ser de gran utilidad para fines educativos.

- Detección de código muerto: A través de las precondiciones es posible implementar la detección de código muerto dentro de las funciones.

6.3.4. ProxyObjects Genéricos

Creemos que es posible partir desde un lado mucho mas interesante, donde tenemos un objeto intermediario que es capaz de elegir la manera de comportarse a medida que es ejecutado el programa. La idea es que este objeto intermediario lleve un registro de posibles objetos en los cuales puede convertirse según las operaciones en las que ha sido involucrado hasta un punto dado de la ejecución. Lograr algo así requiere de una lógica mucho más compleja, sobretodo si queremos tomar en cuenta clases definidas por el usuario.

6.4. Conclusión

La técnica de arquitectura de pares presentada en Peercheck [2] es una técnica prometedora para analizar programas a través de la ejecución simbólica. Descansa en el hecho de que lenguajes puramente orientados a objetos y con despacho dinámico de operaciones pueden realizar ejecución simbólica sin necesidad de modificar el intérprete, usando los mismos objetos del lenguaje como valores simbólicos.

En este trabajo expandimos esa técnica para soportar la generación de valores simbólicos para objetos que pertenecen a clases creadas por el usuario, lo cual era considerado una imposibilidad de la técnica [5] y por lo tanto, razón suficiente para desacreditarse como útil para casos de uso real.

Además, creamos y proporcionamos con este trabajo la herramienta PEF (Python Error Finder), que aplica los conceptos y técnicas mencionadas para realizar exploración de caminos y detección de errores en programas escritos en Python de manera automatizada, gracias a un sistema de contratos que implementamos, que permite especificar validaciones y restricciones sobre la función a explorar, las cuales utilizan Python 3 como lenguajes de especificación. PEF sirve como prototipo y prueba de concepto de la técnica de arquitectura de pares.

Al momento de escribir este trabajo, PEF presenta varias limitaciones: no contempla aliasing, el uso de librerías externas es limitado y no se simbolizan las variables globales y de clases; otras están directamente relacionadas a las limitaciones del razonador, como el soporte de tipos, o la incapacidad de resolver expresiones muy complejas. Sin embargo, Creemos que a medida que los razonadores sean más poderosos, técnicas como la arquitectura de pares permitirá realizar verificaciones y análisis de programas de manera mucho más sencilla, completa y rápida.

Bibliografía

[1] James C. King.
"Symbolic execution and program testing".
Communications of the ACM, volumen 19, número 7, páginas 385–394, 1976.

[2] Alessandro Bruni, Tim Disney, Cormac Flanagan,
"A Peer Architecture for Lightweight Symbolic Execution".
Universidad de California, Santa Cruz, 2011.

[3] Tao Xie, Darko Marinov, Wolfram Schulte, y David Notkin.
"Symstra: A framework for generating object-oriented unit tests using symbolic execution".
Tools and Algorithms for the Construction and Analysis of Systems,
Lecture Notes in Computer Science volumen 3440, páginas 365–381. Springer Berlin / Heidelberg, 2005.

[4] N. Tillmann y W. Schulte.
"Unit tests reloaded: Parameterized unit testing with symbolic execution.
IEEE software, 23(4), páginas 38–47, 2006.

[5] Ruowen Wang, Peng Ning, Tao Xie, Quan Chen.
"MetaSymploit: Day-One Defense Against Script-based Attacks with Security-Enhanced Symbolic Analysis".
Departamento de Ciencias de la Computación, Universidad estatal de Carolina del Norte, Raleigh, NC, USA, página 14.

[6] K. Sen, D. Marinov, y G. Agha.
"CUTE: A concolic unit testing engine for C."

Proceedings of the 10th European software engineering conference, páginas 263–272. ACM, 2005.

[7] P. Godefroid, N. Klarlund, y K. Sen.
"DART: Directed automated random testing".
Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, páginas 213–223. ACM, 2005.

[8] C. Meudec.
"ATGen: automatic test data generation using constraint logic programming and symbolic execution."
Software Testing, Verification and Reliability, 11(2), páginas 81–96, 2001.

[9] Michael Balser, Christoph Duelli, Wolfgang Reif, y Gerhard Schellhorn.
"Verifying Concurrent Systems with Symbolic Execution".
Journal of Logic and Computation, 12(4),
páginas 549–560, 2002.

[10] Sarfraz Khurshid, Corina Psreanu, y Willem Visser.
"Generalized symbolic execution for model checking and testing".
Volumen 2619 de Lecture Notes in Computer Science, páginas 553–568, editorial Springer Berlin / Heidelberg, 2003.

[11] C.S. Pasareanu y W. Visser.
"Verification of Java programs using symbolic execution and invariant generation".
Model Checking Software, páginas 164–181, Editorial Springer Berlin / Heidelberg, 2004.

[12] Josh Berdine, Cristiano Calcagno, and Peter OHearn.
"Symbolic execution with separation logic".
Volumen 3780 de Lecture Notes in Computer Science, páginas 52–68, Editorial Springer Berlin / Heidelberg, 2005.

[13] S. Anand, C. Pasareanu, y W. Visser.
"JPF–SE: A Symbolic Execution Extension to Java PathFinder".

Tools and Algorithms for the Construction and Analysis of Systems, páginas 134–138. Braga, Portugal, 2007.

[14] Corina S. Pasareanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, y Mark Pape.
"Combining unit-level symbolic execution and system-level concrete execution for testing nasa software".
Proceedings of the International Symposium on Software Testing and Analysis, páginas 15–26, editorial Springer Berlin / Heidelberg. New York, USA, 2008.

[15] N. Tillmann y J. De Halleux.
"Pex: white box test generation for .NET".
Proceedings of the 2nd international conference on Tests and proofs, páginas 134–153, editorial Springer Berlin / Heidelberg. Prato, Italia, 2008.

[16] W. Visser, K. Havelund, G. Brat, S. Park y F. Lerda.
"Model Checking Programs".
Automated Software Engineering Journal, Volumen 10, número 2, April 2003.

[17] Godefroid, P., Klarlund, N., Sen, K.
"DART: directed automated random testing".
Programming Language Design and Implementation, páginas 213–223. ACM (2005)

[18] Wikipedia. Búsqueda en profundidad iterativa.
`http://es.wikipedia.org/wiki/Búsqueda_en_profundidad_iterativa`,
accedido 16/07/2014.

[19] Documentación de Python. Funciones Built-in
`docs.python.org/3/library/functions.html#built-in-funcs`,
accedido 16/07/2014.

[20] Wikipedia. James Whitcomb Riley.
`http://en.wikipedia.org/wiki/James_Whitcomb_Riley`,
accedido 13/08/2014.

[21] Wikipedia. Tipado Dinámico.
http://es.wikipedia.org/wiki/Tipado_din%C3%A1mico,
accedido 13/08/2014.

[22] Wikipedia. CPython.
http://en.wikipedia.org/wiki/CPython
accedido 29/08/2014

[23] Wikipedia. C.
http://en.wikipedia.org/wiki/C_(programming_language)
accedido 29/08/2014

[24] Wikipedia. Pascal.
http://en.wikipedia.org/wiki/Pascal_(programming_language)
accedido 29/08/2014

[25] Wikipedia. Guarded Command Language.
http://en.wikipedia.org/wiki/Guarded_Command_Language
accedido 29/09/2014

[26] Wikipedia. c++.
http://en.wikipedia.org/wiki/C++
accedido 24/09/2014

[27] Wikipedia. .NET Framework.
http://en.wikipedia.org/wiki/.NET_Framework
accedido 12/11/2014

[28] Wikipedia. Bytecode.
http://en.wikipedia.org/wiki/Bytecode
accedido 12/11/2014

[29] Wikipedia. LLVM.
http://en.wikipedia.org/wiki/LLVM
accedido 12/11/2014

[30] HaskellWiki. Haskell.
`http://www.haskell.org/haskellwiki/Haskell`
accedido 24/09/2014

[31] Wikipedia. SMT-Solvers.
`http://en.wikipedia.org/wiki/Satisfiability_Modulo_Theories`
accedido 3/09/2014

[32] Wikipedia. CSP-Solvers.
`http://en.wikipedia.org/wiki/Constraint_satisfaction_problem`
accedido 3/09/2014

[33] Wikipedia. Inspección de Software.
`http://en.wikipedia.org/wiki/Software_inspection`
accedido 3/09/2014

[34] G. Nelson y D. Oppen.
"Simplification by cooperating decision procedures".
ACM Transactions on Programming Languages and Systems, volumen 1, número 2, páginas 245–257, 1979.

[35] G. Nelson y D. Oppen.
"Fast decision procedures based on congruence closure".
Journal of the ACM, volumen 27, número 2, páginas 356–364, 1980.

[36] D. Oppen.
"Complexity, convexity and combinations of theories".
Theoretical computer science, volumen 12, número 3, páginas 291–302, 1980.

[37] D. Oppen.
"Reasoning about recursively defined data structures".
Journal of the ACM (JACM), volumen 27, número 3, paginas 403–411, 1980.

[38] Wikipedia. Problema de Satisfacibilidad Booleano
`http://es.wikipedia.org/wiki/Problema_de_satisfacibilidad_`

`booleana`

Accedido 10/09/2014.

[39] Wikipedia. Automated theorem proving

`http://en.wikipedia.org/wiki/Automated_theorem_proving`

Accedido 10/09/2014.

[40] Wikipedia. Theory (Mathematical Logic)

`http://en.wikipedia.org/wiki/Theory_(mathematical_logic)`

Accedido 10/09/2014.

[41] M. N Velev y R. Bryant.

"Exploiting positive equality and partial non-consistency in the formal verification of pipelined microprocessors".

Design Automation Conference, páginas 397–401. IEEE, 1999.

[42] R. Bryant, S. Lahiri, y S. Seshia.

"Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions".

Computer Aided Verification, páginas 78–92. Springer, 2002.

[43] O. Strichman.

"On solving presburger and linear arithmetic with sat".

Formal Methods in Computer-Aided Design, páginas 160–170. Springer, 2002.

[44] O. Strichman, S. Seshia, y R. Bryant.

"Deciding separation formulas with sat".

Computer Aided Verification, páginas 209–222. Springer, 2002.

[45] Wikipedia. Equisatisfiability

`http://en.wikipedia.org/wiki/Equisatisfiability`

Accedido 11/09/2014.

[46] S. Seshia, S. Lahiri, y R. Bryant.

"A hybrid sat-based decision procedure for separation logic with uninterpreted functions".

Proceedings of the 40th annual Design Automation Conference, páginas 425–430. ACM, 2003.

[47] C. Cadar, D. Dunbar, y D. R. Engler.
"KLEE: Unassisted and au-tomatic generation of high-coverage tests for complex systems programs."
Symp. on Operating Sys. Design and Implem., 2008.

[48] A. Kiezun, P. J. Guo, K. Jayaraman, y M. D. Ernst.
"Automatic creation of SQL injection and cross-site scripting attacks."
Intl.Conf. on Software Engineering, 2009.

[49] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, y P. Saxena.
"Bitblaze: A new approach to computer security via binary analysis."
Intl. Conf. on Information Systems Security, 2008.

[50] V. Chipounov, V. Kuznetsov, y G. Candea.
"S2E: A platform for in-vivo multi-path analysis of software systems.
Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, 2011.

[51] Stefan Bucur, Johannes Kinder, George Candea.
"Prototyping Symbolic Execution Engines for Interpreted Languages."
International Conference on Architectural Support for Programming Languages and Operating Systems, 2014.

[52] Ting Chen, Xiao-song Zhang, Rui-dong Chen, Bo Yang, y Yang Bai.
"Conpy: Concolic Execution Engine for Python Applications."
Springer International Publishing Switzerland, 2014.

[53] S. Wolfman y D. Weld.
"The lpsat engine & its application to resource planning".
En IJCAI, páginas 310–317, 1999.

[54] A. Armando, C. Castellini, y E. Giunchiglia.
"Sat-based procedures for temporal reasoning".
En Recent Advances in AI Planning, páginas 97–108. Springer, 2000.

[55] J. Burch y D. Dill.
"Automatic verification of pipelined microprocessor control".
En Computer Aided Verification, páginas 68–80. Springer, 1994.

[56] D. Déharbe y S. Ranise.
"Light-weight theorem proving for debugging and verifying units of code".
En Proceedings of the First International Conference on Software Engineering and Formal Methods, páginas 220–228. IEEE, 2003.

[57] G. Audemard, A. Cimatti, A. Kornilowicz, y R. Sebastiani.
"Bounded model checking for timed systems".
En Formal Techniques for Networked and Distributed Sytems (FORTE 2002), páginas 243–259. Springer, 2002.

[58] L. De Moura y N. Bjørner.
"Z3: An efficient smt solver".
En Tools and Algorithms for the Construction and Analysis of Systems, páginas 337–340. Springer, 2008.

[59] C. Barrett, C. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, y C. Tinelli.
"Cvc4".
En Computer Aided Verification, páginas 171–177. Springer, 2011.

[60] IEEE Recommended Practice for Software Requirements Specifications.
`http://www.math.uaa.alaska.edu/~afkjm/cs401/IEEE830.pdf`, 1998.

[61] C. A. R. Hoare.
"An axiomatic basis for computer programming".
En Communications of the ACM, volumen 12, páginas 576–580, 1969.

[62] C. B. Jones.
"Development methods for computer programs including a notion of interference".
PhD thesis, Oxford University Computing Laboratory, 1981.

[63] S. Owicki y D. Gries.
"An axiomatic proof tech- nique for parallel programs".
Acta Informatica, volumen 6, páginas 319–340, 1976.

[64] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, Dawn Song.
"Loop-Extended Symbolic Execution on Binary Programs".
International Symposium on Software Testing and Analysis.

[65] Jan Obdrzalek, Marek Trtik.
"Efficient Loop Navigation for Symbolic Execution"
Automated Technology for Verification and Analysis.

[66] Marek Trtík.
"Symbolic Execution and Program Loops".
Ph.D. Thesis in Faculty of Informatics, Masaryk University.

[67] Xusheng Xiao, Sihan Li, Tao Xie y Nikolai Tillmann.
"Characteristic Studies of Loop Problems for Structural Test Generation via Symbolic Execution".
Proc. 28th IEEE/ACM International Conference on Automated Software Engineering (ASE 2013).
`http://www.cs.illinois.edu/homes/taoxie/publications/ase13-loopstudy.pdf`

[68] E. M. Clarke, Orna Grumberg y Doron A. Peled
Model Checking
MIT Press, ISBN: 9780262032704, 1999.

[69] Google. Google Python Style Guide
`http://google-styleguide.googlecode.com/svn/trunk/pyguide.html?`

showone=Comments#Comments
Accedido 15/10/2014

[70] PyCharm. Using Docstrings to Specify Types
http://www.jetbrains.com/pycharm/webhelp/
using-docstrings-to-specify-types.html
Accedido 15/10/2014

[71] Python Enhacement Proposal. PEP 257
http://legacy.python.org/dev/peps/pep-0257/
Accedido 15/10/2014

[72] Python Enhacement Proposal. PEP 287
http://legacy.python.org/dev/peps/pep-0287/
Accedido 15/10/2014

[73] Purdue University. Z3-str: A Z3-Based String Solver for Web Application Analysis
https://www.cs.purdue.edu/homes/zheng16/str/
Accedido 17/10/2014

[74] Unicode Consorcium. The Unicode Consortium
http://www.unicode.org
Accedido 17/10/2014

[75] Wikipedia. ASCII
http://en.wikipedia.org/wiki/ASCII
Accedido 17/10/2014

[76] Aditya Nori, Sriram Rajamani, SaiDeep Tetali y Aditya Thakur.
"The Yogi Project: Software Property Checking via Static Analysis and Testing".
En Tools and Algorithms for the Construction and Analysis of Systems (Lecture Notes in Computer Science Volume 5505), 2009, páginas 178-181.

[77] Wolfgang Grieskamp, Xiao Qu, Xiangjun Wei, Nicolas Kicillof y Myra B. Cohen.
"Interaction Coverage Meets Path Coverage by SMT Constraint Solving".
En Testing of Software and Communication Systems, 2009, páginas 99-112, Springer.

[78] Alessandro Cimatti, Sergio Mover y Stefano Tonetta.
"A quantifier-free SMT encoding of non-linear hybrid automata".
En Proceedings of the 12th Conference on Formal Methods in Computer-Aided Design, 2012.

[79] Boyan Yordanov, Christoph M. Wintersteiger, Youssef Hamadi y Hillel Kugler.
"SMT-based analysis of biological computation".
Microsoft Research, 2012.

[80] Grant Olney Passmore, Leonardo de Moura y Paul B. Jackson.
"Gröbner Basis Construction Algorithms Based on Superposition Loops".
En preparation, 2010.

[81] z3Py
`https://z3.codeplex.com/wikipage?title=Z3Py%20on%20Windows&referringTitle=Documentation`
Accedido 05/11/2014

[82] Godefroid, P., Levin, M., Molnar, D.
"Automated whitebox fuzz testing.".
NDSS, 2008, páginas 151–166.

[83] Robert Sedgewick.
"The analysis of quicksort programs."
Acta Informatica 7.4, 1977, páginas 327-355.

Apéndice A

Código fuente de PEF

A.1. IntProxy

```
1  class IntProxy(ProxyObject):
2      """
3      Int Proxy object for variables that behave like ints.
4      """
5      emulated_class = int
6
7      def __init__(self, real=None):
8          """
9          :real: Either symbolic numeric variable in smt solver or real int value.
10         """
11         if isinstance(real, IntProxy):
12             self.real = real.real  # Idempotent creation
13         elif real != None:
14             self.real = real
15         else:
16             self.real = fresh_symbolic_var("int")
17
18     def __abs__(self):
19         """
20         x.__abs__() <==> abs(x)
21         """
22         return self if self > 0 else -self
23
24     @forward_to_rfun()
25     def __add__(self, other):
26         """
27         :types: other: int
28         x.__add__(y) <==> x+y
29         """
30         return IntProxy(self.real + other.real)
31
32     def __bool__(self):
33         """
34         x.__bool__() <==> x != 0
```

```
35          """
36          return bool(self != 0)
37
38      def __deepcopy__(self, memo):
39          """
40          x.__deepcopy__() <==> copy.deepcopy(x)
41          """
42          return self
43
44      @check_equality
45      def __eq__(self, other):
46          """
47          :types: other: int
48          x.__eq__(y) <==> x==y
49          """
50          other = other.real if isinstance(other, IntProxy) else other
51          return BoolProxy(self.real == other)
52
53      @forward_to_rfun()
54      def __floordiv__(self, other):
55          """
56          :types: other: int
57          x.__floordiv__(y) <==> x//y
58          """
59          if other != 0:
60              # Fix z3 bugs
61              # 1) -1 == -1//2 != 1//-2  == 0
62              # 2)  1 ==  1//2 != -1//-2 == 0
63              if other < 0:
64                  return -self//-other
65              else:
66                  return IntProxy(self.real / other.real)
67          else:
68              raise ZeroDivisionError("IntProxy division or modulo by zero")
69
70      @check_comparable_types
71      def __ge__(self, other):
72          """
73          :types: other: int
74          x.__ge__(y) <==> x>=y
75          """
76          other = other.real if isinstance(other, IntProxy) else other
77          return BoolProxy(self.real >= other)
78
79      @check_comparable_types
80      def __gt__(self, other):
81          """
82          :types: other: int
83          x.__gt__(y) <==> x>y
84          """
85          other = other.real if isinstance(other, IntProxy) else other
86          return BoolProxy(self.real > other)
87
88      def __hash__(self):
89          """
90          x.__hash__() <==> hash(x)
```

```
91          """
92          return super().__hash__()
93
94      @check_comparable_types
95      def __le__(self, other):
96          """
97          :types: other: int
98          x.__le__(y) <==> x<=y
99          """
100         other = other.real if isinstance(other, IntProxy) else other
101         return BoolProxy(self.real <= other)
102
103     @check_comparable_types
104     def __lt__(self, other):
105         """
106         :types: other: int
107         x.__lt__(y) <==> x<y
108         """
109         other = other.real if isinstance(other, IntProxy) else other
110         return BoolProxy(self.real < other)
111
112     @forward_to_rfun()
113     def __mod__(self, other):
114         """
115         :types: other: int
116         x.__mod__(y) <==> x%y
117         """
118         if other != 0:
119             return IntProxy(self.real % other.real)
120         else:
121             raise ZeroDivisionError("IntProxy division or modulo by zero")
122
123     @forward_to_rfun()
124     def __mul__(self, other):
125         """
126         :types: other: int
127         x.__mul__(y) <==> x*y
128         """
129         return IntProxy(self.real * other.real)
130
131     def __ne__(self, other):
132         """
133         :types: other: int
134         x.__ne__(y) <==> x!=y
135         """
136         other = other.real if isinstance(other, IntProxy) else other
137         return BoolProxy(self.real != other)
138
139     def __neg__(self):
140         """
141         x.__neg__() <==> -x
142         """
143         return IntProxy(-self.real)
144
145     @check_self_and_other_have_same_type
146     def __radd__(self, other):
```

```
147         """
148         :types: other: int
149         x.__radd__(y) <==> y+x
150         """
151         return self.__add__(other)
152 
153     @check_self_and_other_have_same_type
154     def __req__(self, other):
155         """
156         :types: other: int
157         x.__req__(y) <==> y==x
158         """
159         return self.__eq__(other)
160 
161     @check_self_and_other_have_same_type
162     def __rfloordiv__(self, other):
163         """
164         :types: other: int
165         x.__rfloordiv__(y) <==> y//x
166         """
167         if self != 0:
168             # Fix z3 bugs
169             # 1) -1 == -1//2 != 1//-2  == 0
170             # 2)  1 ==  1//2 != -1//-2 == 0
171             if self < 0:
172                 return -other//-self
173             else:
174                 return IntProxy(other.real / self.real)
175         else:
176             raise ZeroDivisionError("IntProxy division or modulo by zero")
177 
178     @check_self_and_other_have_same_type
179     def __rmod__(self, other):
180         """
181         :types: other: int
182         x.__rmod__(y) <==> y%x
183         """
184         if self != 0:
185             return IntProxy(other.real % self.real)
186         else:
187             raise ZeroDivisionError("IntProxy division or modulo by zero")
188 
189     @check_self_and_other_have_same_type
190     def __rmul__(self, other):
191         """
192         :types: other: int
193         x.__rmul__(y) <==> y*x
194         """
195         return self.__mul__(other)
196 
197     def __repr__(self):
198         """
199         x.__repr__() <==> repr(x)
200         """
201         if not isinstance(self.real, int):
202             s = _smt.simplify(self.real)
```

```
203             if _smt.is_int_value(s):
204                 s = s.as_long()
205         return "IntProxy(%s)" % s
206 
207     @check_self_and_other_have_same_type
208     def __rsub__(self, other):
209         """
210         :types: other: int
211         x.__rsub__(y) <==> y-x
212         """
213         return IntProxy(other.real - self.real)
214 
215     @forward_to_rfun()
216     def __sub__(self, other):
217         """
218         :types: other: int
219         x.__sub__(y) <==> x-y
220         """
221         return IntProxy(self.real - other.real)
222 
223     @forward_to_rfun()
224     def __truediv__(self, other):
225         """
226         :types: other: int
227         x.__truediv__(y) <==> x/y
228         """
229         raise NotImplementedError("FloatProxy not implemented.")
230 
231     def _concretize(self, model):
232         if isinstance(self.real, int):
233             result = self.real
234         else:
235             result = model.evaluate(self.real, model_completion=True).as_long()
236         return result
```

A.2. BoolProxy

```
1  class BoolProxy(ProxyObject):
2      """
3      Bool Proxy object for variables that behave like bools.
4      """
5      emulated_class = bool
6  
7      def __init__(self, formula=None):
8          if formula is None:
9              formula = fresh_symbolic_var("bool")
10         self.formula = formula
11 
12     def __not__(self):
13         return BoolProxy(_smt.Not(self.formula))
14 
15     def __nonzero__(self):
16         return self.__bool__()
```

```
17
18      def __bool__(self):
19          if isinstance(self.formula, bool):
20              return self.formula
21          # If self.formula isn't a builtin bool we have to solve it
22          partial_solve = self._get_partial_solve()
23          if partial_solve != None:
24              return partial_solve
25          if len(self._path) > len(self._pathcondition):
26              branch = self._path[len(self._pathcondition)]
27              if branch:
28                  formula = self.formula
29              else:
30                  formula = _smt.Not(self.formula)
31              self._pathcondition.append(formula)
32              return branch
33          if len(self._path) >= ProxyObject.max_depth:
34              # self._path.pop()  # We go back to the previus branch
35              raise MaxDepthError()
36          self._path.append(True)
37          self._pathcondition.append(self.formula)
38          return True
39
40      def _get_partial_solve(self):
41          """
42          :returns: None if constrains haven't define a concrete value yet,
43                    else returns that concrete value (True or False).
44          It tries to obtain a bool value if possible. Without branching.
45          """
46          conditions = self._pathcondition + self._facts
47          true_cond  = _solve(_smt.And(conditions + [self.formula]))
48          false_cond = _solve(_smt.And(conditions + [_smt.Not(self.formula)]))
49          if true_cond and not false_cond:
50              return True
51          if false_cond and not true_cond:
52              return False
53
54      def __repr__(self):
55          ps = self._get_partial_solve()
56          return "BoolProxy(%s)" % (ps if ps != None else str(self.formula))
57
58      def __deepcopy__(self, memo):
59          return self
60
61      def _concretize(self, model):
62          if isinstance(self.formula, bool):
63              result = self.formula
64          else:
65              result = _smt.is_true(model.evaluate(self.formula,
66                                                   model_completion=True))
67          return result
```

A.3. ListProxy

```
class ListProxy(ProxyObject):
    """
    List Proxy object for variables that behave like int lists.
    also, isinstance(ListProxy(), Iterable) == True
    """
    emulated_class = list

    def __init__(self, initial=None):
        self._array = fresh_symbolic_var("map(int,int)")
        self._start = 0  # For slice operations
        self._step = 1  # For slice operations
        if isinstance(initial, ListProxy):
            # Idempotent creation
            self._update_self(initial)
        elif isinstance(initial, Iterable):
            self._len = len(initial)
            for i, x in enumerate(initial):
                if not (isinstance(x, int) or isinstance(x, IntProxy)):
                    raise AssertionError("ListProxy can only contain numbers, "+
                            "%s containing %s of type %s given in __init__" % \
                            (str(initial), str(initial[0]), type(initial[0])))
                # Only support for list of ints or IntProxys for now.
                # FIXME: Extend it for at least list of list.
                if isinstance(x, IntProxy):
                    x = x.real
                self._array = _smt.Store(self._array, i, x)
        else:
            self._len = IntProxy()
            boolproxy = self._len >= 0
            self._facts.append(boolproxy.formula)

    def __len__(self):
        """
        x.__len__() <==> len(x)
        """
        return self._len

    def __getitem__(self, i):
        """
        :types: i:[int, slice]
        x.__getitem__(y) <==> x[y]
        """
        if isinstance(i, int) or isinstance(i, IntProxy):
            index = self._get_index(i)
            if index is None:
                raise IndexError("ListProxy index out of range")
            result = IntProxy(_smt.Select(self._array, index.real))
        elif isinstance(i, slice) or isinstance(i, SliceProxy):
            # Transform the slice to a SliceProxy
            i = SliceProxy(i)
            # Calculate the real start/stop/step
            start, stop, step = i.indices(self._len)
            start_i = self._get_index(start)
            if start_i is None:
```

```
55                  # An out of range start means the result list will be []
56                  start, stop, step = 0, 0, 1
57                  start_i = self._start
58              result = copy.deepcopy(self)
59              result._start = start_i
60              result._step = self._step * step
61              # Watch out for negative lengths!
62              fix = 0 if (stop - start) % step == 0 else 1
63              result._len = max(0, (stop - start) // step + fix)
64              return result
65          else:
66              raise TypeError("ListProxy indices must be integers, not %s" \
67                  % type(i).__name__)
68          return result
69
70      def __setitem__(self, i, y):
71          """
72          :types: i:[int, slice],  y:[int, ]
73          x.__setitem__(i, y) <==> x[i]=y
74          """
75          if isinstance(i, int) or isinstance(i, IntProxy):
76              index = self._get_index(i)
77              if index is None:
78                  raise IndexError("ListProxy assignment index out of range.")
79              self._array = _smt.Store(self._array, index.real, y.real)
80          elif isinstance(i, slice) or isinstance(i, SliceProxy):
81              # Transform the slice to a SliceProxy
82              i = SliceProxy(i)
83              # Calculate the real start/stop/step
84              start, stop, step = i.indices(self._len)
85              # If start is out of range, set default values
86              if not 0 <= start < self._len:
87                  start, stop, step = 0, 0, 1
88              # Watch out with negative lengths!
89              fix = 0 if (stop - start) % step == 0 else 1
90              length = max(0, (stop - start) // step + fix)
91              # Get `length` elements from y
92              iterable_y = iter(y)
93              to_assign = []
94              try:
95                  [to_assign.append(next(iterable_y)) for x in range(length)]
96              except StopIteration:
97                  raise ValueError("attempt to assign sequence of size %s to "\
98                       "extended slice of size %s" % (len(to_assign), length))
99              for i in range(start, stop, step):
100                 self[i] = to_assign[i]
101         else:
102             raise TypeError("ListProxy indices must be integers, not %s" \
103                 % type(i).__name__)
104
105     def __iter__(self):
106         """
107         x.__iter__() <==> iter(x)
108         """
109         # FIXME: return list-iterator object instead of generator to match real
110         # lists behaviour
```

```
111         # All the start/stop/step logic is solved in __getattr__ (get_index)
112         i = 0
113         while i < self.__len__():
114             yield self[i]
115             i += 1
116 
117     def __repr__(self):
118         """
119         x.__repr__() <==> repr(x)
120         """
121         if not isinstance(self._len, int):
122             s = "[ ... | length = %s]" % self._len
123         else:
124             l = ["%s" % x for x in self]
125             s = "[%s]" % " ,".join(l)
126         return s
127 
128     def __eq__(self, other):
129         """
130         :types: other: list
131         x.__eq__(y) <==> x == y
132         """
133         result = False
134         if len(self) == len(other):
135             result = True
136             for i in range(len(self)):
137                 if self[i] != other[i]:
138                     result = False
139                     break
140         return result
141 
142     def __gt__(self, other):
143         """
144         :types: other: list
145         x.__ge__(y) <==> x>y
146         """
147         if not self:  # [] < [, ] always
148             return False
149         min_l = min(len(self), len(other))
150         for i in range(min_l):
151             if self[i] < other[i]:
152                 return False
153             if self[i] > other[i]:
154                 return True
155         return len(self) > len(other)
156 
157     def __ge__(self, other):
158         """
159         :types: other: list
160         x.__ge__(y) <==> x>=y
161         """
162         # Equivalent to: return self > other or self == other
163         if not self and other:  # [] < [, ] always
164             return False
165         min_l = min(len(self), len(other))
166         for i in range(min_l):
```

```
167             if self[i] < other[i]:
168                 return False
169             if self[i] > other[i]:
170                 return True
171         return len(self) >= len(other)
172 
173     def __lt__(self, other):
174         """
175         :types: other: list
176         x.__ge__(y) <==> x<y
177         """
178         return not self >= other
179 
180     def __le__(self, other):
181         """
182         :types: other: list
183         x.__ge__(y) <==> x<y
184         """
185         return not self > other
186 
187     def __bool__(self):
188         """
189         x.__bool__() <==> len(self) > 0
190         """
191         # return self._len != 0
192         return bool(self._len > 0)
193 
194     @forward_to_rfun()
195     def __add__(self, other):
196         """
197         :types: other: list
198         x.__add__(y) <==> x+y
199         """
200         return ListProxy([x for x in self] + [x for x in other])
201 
202     def __contains__(self, other):
203         """
204         :types: other: int
205         x.__contains__(y) <==> y in x
206         """
207         for i in self:
208             if i == other:
209                 return True
210         return False
211 
212     def __delitem__(self, other):
213         """
214         :types: other: int
215         x.__delitem__(y) <==> del x[y]
216         """
217         self.pop(other)
218         # Deleting a item in Z3 gets messy, so we just pop the item.
219 
220     def __iadd__(self, other):
221         """
222         :types: other: list
```

```
223         x.__iadd__(y) <==> x+=y
224         """
225         self.extend(other)
226         return self
227 
228     def __imul__(self, n):
229         """
230         :types: n: int
231         x.__imul__(y) <==> x*=y
232         """
233         if n <= 0:
234             self.clear()
235         elif n > 1:
236             #TODO: se puede sin branching? (hace falta una copia inmutable)
237             self_copy = ListProxy([x for x in self])
238             for i in range(n - 1):
239                 self += self_copy
240         return self
241 
242     @forward_to_rfun(int, IntProxy)
243     def __mul__(self, n):
244         """
245         :types: n: int
246         x.__mul__(y) <==> x*y
247         """
248         result = self.copy()
249         result *= n
250         return result
251 
252     def __rmul__(self, n):
253         """
254         :types: n: int
255         x.__rmul__(y) <==> y*x
256         """
257         return self * n
258 
259     def append(self, x):
260         """
261         :types: x: int
262         L.append(object) -> None -- append object to end
263         """
264         index = self.__len__()
265         self._len += 1
266         self[index] = x
267 
268     def clear(self):
269         """
270         L.clear() -> None -- remove all items from L
271         """
272         self._len = 0
273         self._start = 0
274         self._step = 0
275 
276     def copy(self):
277         """
278         L.copy() -> list -- a shallow copy of L
```

```
279         """
280         return ListProxy(self)
281 
282     def count(self, value):
283         """
284         :types: value: int
285         L.count(value) -> integer -- return number of occurrences of value
286         """
287         r = 0
288         for x in self:
289             if x == value:
290                 r += 1
291         return r
292 
293     def extend(self, iterable):
294         """
295         :types: iterable: list
296         L.extend(iterable) -> None -- extend list by appending elements from
297         the iterable
298         """
299         for x in iterable:
300             self.append(x)
301 
302     @handle_varargs_and_defaults(0, None)
303     def index(self, value, *args):
304         """
305         :types: value: int
306         L.index(value, [start, [stop]]) -> integer -- return first index of value.
307         :raises: ValueError if the value is not present.
308         """
309         start, stop = args[0], args[1] if args[1] != None else len(self)
310         result = None
311         try:
312             start = self._get_positive_index(start)
313         except IndexError:
314             raise ValueError("%s not in ListProxy" % (value))
315         for i, x in enumerate(self[start:stop], start):
316             if x == value:
317                 result = i
318                 break
319         if result == None:
320             raise ValueError("%s is not in ProxyList" % (value))
321         else:
322             return result
323 
324     @handle_varargs_and_defaults(-1)
325     def pop(self, *args):
326         """
327         L.pop([index]) -> item -- remove and return item at index (default last).
328         :raises: IndexError if list is empty or index is out of range.
329         """
330         i = self._get_positive_index(args[0])
331         result = self[i]
332         new_self = ListProxy([x for i, x in enumerate(self) if i != i])
333         self._array = new_self._array
334         self._len = self._len - 1  # We do this to keep _len symbolic
```

```
335         return result
336 
337     def reverse(self):
338         """
339         L.reverse() -- reverse *IN PLACE*
340         """
341         # Change direction
342         self._step *= -1
343         # Now we start with the last element
344         self._start = self._get_index(-1)
345 
346     def _sort(self, key=None, reverse=False):
347         """
348         :types: reverse: bool
349         L.sort(key=None, reverse=False) -> None
350         """
351         if not self:
352             return []
353         else:
354             if key == None:
355                 key = lambda x: x
356             pivot = self[0]
357             lesser = ListProxy([])
358             for x in self[1:]:
359                 if key(x) < key(pivot):
360                     lesser.append(x)
361             lesser = lesser._sort()
362             greater = ListProxy([])
363             for x in self[1:]:
364                 if key(x) >= key(pivot):
365                     greater.append(x)
366             greater = greater._sort()
367             result = lesser + [pivot] + greater
368             return result
369 
370     def sort(self, key=None, reverse=False):
371         """
372         L.sort(key=None, reverse=False) -> None -- stable sort *IN PLACE*
373         """
374         result = self._sort(key, reverse)
375         if reverse:
376             result.reverse()
377         self._update_self(ListProxy(result))
378 
379     def insert(self, index, obj):
380         """
381         :types: index: int, obj: [int, ]
382         L.insert(index, object) -> None -- insert object before index
383         """
384         prefix = self[:index]
385         postfix = self[index:]
386         self._update_self(ListProxy(prefix + [obj] + postfix))
387 
388     def remove(self, value):
389         """
390         :types: value: [int,]
```

```
391         L.remove(value) -> None -- remove first occurrence of value.
392         :raises: ValueError if value not in self
393         """
394         i = self.index(value)
395         self.pop(i)
396 
397     def _get_positive_index(self, i):
398         """
399         Given an index ‘i‘ representing a position inside a list,
400         _get_positive_index returns a positive index that is relative to the
401         given ‘i’ position in the array.
402         WARNING: To find the corresponding index in the Z3 array ‘_array’, use
403         _get_index instead.
404         :raises: IndexError if i out of range of self
405         """
406         if 0 <= i < len(self):
407             result = IntProxy(i)  # Positive index in valid range
408         elif -len(self) <= i < 0:
409             result = IntProxy(len(self) + i)  # Negative index in valid range
410         else:
411             raise IndexError("Index out of range")
412         return result
413 
414     def _get_index(self, i):
415         """
416         Given an index ‘i‘ representing a position inside a list, _get_index
417         deals with the start/length/step/‘negative index‘ logic and returns
418         the corresponding index for the private attribute _array or None if out
419         of range.
420         WARNING: Use this to get position in the Z3 array _array.
421         To calculate positive relative index, use _get_positive_index instead.
422         """
423         try:
424             positive_index = self._get_positive_index(i)
425         except IndexError:
426             return None
427         return self._start + self._step * positive_index
428 
429     def __deepcopy__(self, memo):
430         return copy.copy(self)
431 
432     def _update_self(self, other):
433         """
434         copy ‘other‘ attributes into ‘self‘.
435         """
436         self.__dict__ = other.__dict__
437 
438     def _concretize(self, model):
439         self._len = _concretize(self._len, model)
440         if isinstance(self._step, IntProxy):
441             self._step = self._step._concretize(model)
442         self._start = _concretize(self._start, model)
443         return [x._concretize(model) for x in self]
```

A.4. StringProxy

```
class StringProxy(ProxyObject):
    emulated_class = str

    def __init__(self, initial=None):
        if isinstance(initial, str):
            # Camouflage
            array = fresh_symbolic_var("map(int, char)")
            start, step, length = 0, 1, len(initial)
            for i, e in enumerate(initial):
                array = _smt.Store(array, i, ord(e))
        elif isinstance(initial, StringProxy):
            # Creator idempotence
            array = initial._array
            start, step, length = initial._start, initial._step, initial._len
        elif _smt.is_bv(initial):
            # Special case, a char is also a string
            array = _smt.Store(fresh_symbolic_var("map(int, char)"), 0, initial)
            start, step, length = 0, 1, 1
        elif initial is None:
            array = fresh_symbolic_var("map(int, char)")
            start, step, length = 0, 1, IntProxy()
            boolproxy = length >= 0
            self._facts.append(boolproxy.formula)
        else:
            raise TypeError("type for initial: (%s) not supported" %\
                            (str(type(initial).__name__)))
        self._array = array
        self._start = start
        self._step = step
        self._len = length

    @forward_to_rfun()
    def __add__(self, other):
        """
        :type: other: str
        x.__add__(y) <==> x+y
        """
        result = StringProxy(self)
        for x in other:
            result = result._append(x)
        return result

    def __bool__(self):
        """
        x.__bool__() <==> len(self) > 0
        """
        # return self._len != 0
        return bool(self._len > 0)

    def __contains__(self, elem):
        """
        :type: elem: str
        x.__contains__(y) <==> y in x
        """
```

```
55          return self.find(elem) != -1
56
57      def __copy__(self):
58          return StringProxy(self)
59
60      def __deepcopy__(self, memo):
61          return StringProxy(self)
62
63      @check_equality
64      def __eq__(self, other):
65          """
66          :type: other: str
67          x.__eq__(y) <==> x==y
68          """
69          result = False
70          if len(self) == len(other):
71              result = True
72              for i in range(len(self)):
73                  self_i = self[i]._array[0]
74                  if isinstance(other, str):
75                      other_i = ord(other[i])
76                  else:
77                      other_i = other[i]._array[0]
78                  if BoolProxy(self_i == other_i):
79                      continue
80                  else:
81                      result = False
82                      break
83          return result
84
85      def __format__(self, *args, **kwargs):
86          """
87          S.__format__(format_spec) -> str
88
89          Return a formatted version of S as described by format_spec.
90          """
91          if len(args) == 1 and not args[0]:
92              return str(self)  # Base case
93          else:
94              raise NotImplementedError()
95
96      @check_comparable_types
97      def __ge__(self, other):
98          """
99          :type: other: str
100         x.__ge__(y) <==> x>=y
101         """
102         # Equivalent to: return self > other or self == other
103         if not self and other:  # [] < [, ] always
104             return False
105         min_l = min(len(self), len(other))
106         for i in range(min_l):
107             self_i = self[i]._array[0]
108             if isinstance(other, str):
109                 other_i = ord(other[i])
110             else:
```

```
111                 other_i = other[i]._array[0]
112             if BoolProxy(_smt.ULT(self_i, other_i)):
113                 return False
114             if BoolProxy(_smt.UGT(self_i, other_i)):
115                 return True
116         return len(self) >= len(other)
117 
118     def __getitem__(self, i):
119         """
120         :types: i:[int, slice]
121         x.__getitem__(y) <==> x[y]
122         """
123         if isinstance(i, int) or isinstance(i, IntProxy):
124             index = self._get_index(i)
125             if index is None:
126                 raise IndexError("StringProxy index out of range")
127             result = StringProxy(_smt.Select(self._array, index.real))
128         elif isinstance(i, slice) or isinstance(i, SliceProxy):
129             # Transform the slice to a SliceProxy
130             i = SliceProxy(i)
131             # Calculate the real start/stop/step
132             start, stop, step = i.indices(self._len)
133             start_i = self._get_index(start)
134             if start_i is None:
135                 # An out of range start means the result list will be []
136                 start, stop, step = 0, 0, 1
137                 start_i = self._start
138             result = copy.deepcopy(self)
139             result._start = start_i
140             result._step = self._step * step
141             # Watch out for negative lengths!
142             fix = 0 if (stop - start) % step == 0 else 1
143             result._len = max(0, (stop - start) // step + fix)
144             return result
145         else:
146             raise TypeError("ListProxy indices must be integers, not %s" \
147                             % type(i).__name__)
148         return result
149 
150     @check_comparable_types
151     def __gt__(self, other):
152         """
153         :type: other: str
154         x.__gt__(y) <==> x>y
155         """
156         if not self:  # [] < [, ] always
157             return False
158         min_l = min(len(self), len(other))
159         for i in range(min_l):
160             self_i = self[i]._array[0]
161             if isinstance(other, str):
162                 other_i = ord(other[i])
163             else:
164                 other_i = other[i]._array[0]
165             if BoolProxy(_smt.ULT(self_i, other_i)):
166                 return False
```

```
167             if BoolProxy(_smt.UGT(self_i, other_i)):
168                 return True
169         return len(self) > len(other)
170 
171     def __hash__(self, *args, **kwargs):
172         """
173         x.__hash__() <==> hash(x)
174         """
175         raise NotImplementedError()
176 
177     def __iter__(self):
178         """
179         x.__iter__() <==> iter(x)
180         """
181         # All the start/stop/step logic is solved in __getattr__ (get_index)
182         i = 0
183         while i < self.__len__():
184             yield self[i]
185             i += 1
186 
187     def __le__(self, other):
188         """
189         :type: other: str
190         x.__le__(y) <==> x<=y
191         """
192         return not self > other
193 
194     def __len__(self):
195         """
196         x.__len__() <==> len(x)
197         """
198         return self._len
199 
200     def __lt__(self, other):
201         """
202         :types: other: str
203         x.__ge__(y) <==> x<y
204         """
205         return not self >= other
206 
207     def __mod__(self, *args, **kwargs):
208         """
209         x.__mod__(y) <==> x%y
210         """
211         raise NotImplementedError()
212 
213     @forward_to_rfun(int, IntProxy)
214     def __mul__(self, times):
215         """
216         :type: times: int
217         x.__mul__(n) <==> x*n
218         """
219         result = StringProxy("")
220         times = max(0, times)
221         for x in range(times):
222             result += self
```

```
223         return result
224
225     def __ne__(self, other):
226         """
227         :type: other: str
228         x.__ne__(y) <==> x!=y
229         """
230         return not (self == other)
231
232     @check_self_and_other_have_same_type
233     def __radd__(self, other):
234         """
235         :types: other: str
236         x.__radd__(y) <==> y+x
237         """
238         return self.__add__(other)
239
240     def __repr__(self):
241         """
242         x.__repr__() <==> repr(x)
243         """
244         if not isinstance(self._len, int):
245             return "'...' | length = %s" % self._len
246         # If we know the len, we can return something more representative
247         s = ""
248         for x in self:
249             s_x = _smt.simplify(x._array[0])
250             if _smt.is_bv_value(s_x):
251                 s += chr(s_x.as_long())
252             else:
253                 s += "?"
254         s = "'%s'" % s
255         return s
256
257     def __rmod__(self, *args, **kwargs):
258         """
259         x.__rmod__(y) <==> y%x
260         """
261         raise NotImplementedError()
262
263     def __rmul__(self, times):
264         """
265         :type: times: int
266         x.__rmul__(n) <==> n*x
267         """
268         return self.__mul__(times)
269
270     def __str__(self):
271         """
272         x.__str__() <==> str(x)
273         """
274         return self.__repr__()
275
276     def _append(self, elem):
277         """
278         Given a character ‘elem‘, _append returns a new string equivalent to
```

```
279         self + elem
280         """
281         result = StringProxy(self)
282         to_append = ord(elem) if isinstance(elem, str) else elem._array[0]
283         result._len += 1
284         index = result._get_index(-1).real
285         result._array = _smt.Store(self._array, index, to_append)
286         return result
287 
288     def _concretize(self, model):
289         self._len = _concretize(self._len, model)
290         self._step = _concretize(self._step, model)
291         self._start = _concretize(self._start, model)
292         result = "".join([x._concretize_char(model) for x in self])
293         return result
294 
295     def _concretize_char(self, model):
296         char = model.evaluate(self._array[0], model_completion=True)
297         return chr(char.as_long())
298 
299     def _get_index(self, i):
300         """
301         Given an index `i` representing a position inside a list, _get_index
302         deals with the start/length/step/`negative index` logic and returns
303         the corresponding index for the private attribute _array.
304         """
305         index = None
306         if 0 <= i < self.__len__():
307             index = IntProxy(i)  # Positive index in valid range
308         elif -self.__len__() <= i < 0:
309             # Negative index in valid range
310             index = IntProxy(self.__len__() + i)
311         # If index is something compute the start/step offset
312         return self._start + self._step * index if index != None else index
313 
314     def _prepend(self, elem):
315         """
316         Given a character `elem`, _prepend returns a new string equivalent to
317         elem + self
318         """
319         result = StringProxy(self)
320         to_prepend = ord(elem) if isinstance(elem, str) else elem._array[0]
321         result._len += 1
322         result._start = self._start - self._step
323         result._array = _smt.Store(self._array, result._start.real, to_prepend)
324         return result
325 
326     def capitalize(self):
327         """
328         S.capitalize() -> str
329 
330         Return a capitalized version of S, i.e. make the first character
331         have upper case and the rest lower case.
332         """
333         return self.upper(self[0]) + self[1:] if self else self
334 
```

```
335     def casefold(self):
336         """
337         S.casefold() -> str
338 
339         Return a version of S suitable for caseless comparisons.
340         """
341         # https://mail.python.org/pipermail/python-ideas/2012-January/013293.html
342         # casefold it is not just a call to self.lower
343         raise NotImplementedError("StringProxy.casefold")
344 
345     @handle_varargs_and_defaults(' ')
346     def center(self, width, *args):
347         """
348         :types: width: int, args: str
349 
350         S.center(width[, fillchar]) -> str
351 
352         Return S centered in a string of length width. Padding is
353         done using the specified fill character (default is a space)
354         """
355         fillchar = StringProxy(args[0])
356         if width < len(self):
357             return self
358         else:
359             marg = width - len(self);
360             # Algorithm copied from:
361             #http://svn.python.org/view/python/trunk/Objects/stringobject.c
362             pad_l = marg // 2 + (1 if width % 2 == 1 and len(self) % 2 == 0\
363                                  else 0)
364             pad_r = marg - pad_l
365             return pad_l * fillchar + self + pad_r * fillchar
366 
367     @handle_varargs_and_defaults(None, None)
368     def count(self, sub, *args):
369         """
370         :types: sub: str
371 
372         S.count(sub[, start[, end]]) -> int
373 
374         Return the number of non-overlapping occurrences of substring sub in
375         string S[start:end].  Optional arguments start and end are
376         interpreted as in slice notation.
377         """
378         start, stop = args
379         start = 0 if start is None else start
380         stop = len(self) if stop is None else stop
381         c = 0
382         while True:
383             try:
384                 # Non-overlaping ocurrences
385                 start = self.index(sub, start, stop) + max(len(sub), 1)
386                 c += 1
387             except ValueError:
388                 break
389         return c
390 
```

```
391     def encode(self, encoding='uft-8', errors='strict'):
392         """
393         S.encode(encoding='utf-8', errors='strict') -> bytes
394 
395         Encode S using the codec registered for encoding. Default encoding
396         is 'utf-8'. errors may be given to set a different error
397         handling scheme. Default is 'strict' meaning that encoding errors raise
398         a UnicodeEncodeError. Other possible values are 'ignore', 'replace' and
399         'xmlcharrefreplace' as well as any other name registered with
400         codecs.register_error that can handle UnicodeEncodeErrors.
401         """
402         raise NotImplementedError("StringProxy.encode")
403 
404     @handle_varargs_and_defaults(None, None)
405     def endswith(self, suffix, *args):
406         """
407         :types: suffix: str, args: list
408 
409         S.endswith(suffix[, start[, end]]) -> bool
410 
411         Return True if S ends with the specified suffix, False otherwise.
412         With optional start, test S beginning at that position.
413         With optional end, stop comparing S at that position.
414         suffix can also be a tuple of strings to try.
415         """
416         start, stop = args
417         start = 0 if start is None else start
418         stop = len(self) if stop is None else stop
419         sliced_string = self[start:stop]
420         return sliced_string[len(sliced_string) - len(suffix):] == suffix
421 
422     @handle_varargs_and_defaults(8)
423     def expandtabs(self, *args):
424         """
425         :types: args: list
426 
427         S.expandtabs([tabsize]) -> str
428 
429         Return a copy of S where all tab characters are expanded using spaces.
430         If tabsize is not given, a tab size of 8 characters is assumed.
431         """
432         tabsize = args[0]
433         return self.replace('\t', tabsize * StringProxy(' '))
434 
435     @handle_varargs_and_defaults(None, None)
436     def find(self, sub, *args):
437         """
438         :types: sub: str, args: list
439 
440         S.find(sub[, start[, end]]) -> int
441 
442         Return the lowest index in S where substring sub is found,
443         such that sub is contained within S[start:end].  Optional
444         arguments start and end are interpreted as in slice notation.
445 
446         Return -1 on failure.
```

```
447         """
448         start, stop = args
449         start = 0 if start is None else start
450         stop = len(self) if stop is None else stop
451         start = max(start + len(self), 0) if start < 0 else start
452         if stop < 0:
453             stop = max(stop + len(self), 0)
454         else:
455             stop = min(stop, len(self))
456         if start > len(self):
457             return -1
458         elif stop < start:
459             return -1
460         elif sub == '':
461             return start
462         else:
463             for i in range(start, stop - len(sub) + 1):
464                 if self[i:i + len(sub)] == sub:
465                     return i
466         return -1
467 
468     def format(self, *args, **kwargs):
469         """
470         S.format(*args, **kwargs) -> str
471 
472         Returns formatted version of S, using substitutions from args and kwargs.
473         The substitutions are identified by braces ('{' and '}').
474         """
475         raise NotImplementedError("StringProxy.format")
476 
477     def format_map(self, mapping):
478         """
479         S.format_map(mapping) -> str
480 
481         Returns formatted version of S, using substitutions from mapping.
482         The substitutions are identified by braces ('{' and '}').
483         """
484         raise NotImplementedError()
485 
486     def index(self, sub, *args):
487         """
488         :types: sub: str, args: list
489 
490         S.index(sub[, start[, end]]) -> int
491 
492         Like S.find() but raise ValueError when the substring is not found.
493         """
494         i = self.find(sub, *args)
495         if i == -1:
496             raise ValueError("substring not found")
497         else:
498             return i
499 
500     def isalnum(self):
501         """
502         S.isalnum() -> bool
```

```
503
504         Return True if all characters in S are alphanumeric
505         and there is at least one character in S, False otherwise.
506         """
507         raise NotImplementedError()
508
509     def isalpha(self):
510         """
511         S.isalpha() -> bool
512
513         Return True if all characters in S are alphabetic
514         and there is at least one character in S, False otherwise.
515         """
516         raise NotImplementedError()
517
518     def isdecimal(self):
519         """
520         S.isdecimal() -> bool
521
522         Return True if there are only decimal characters in S,
523         False otherwise.
524         """
525         # TODO: This is not the right implementation.
526         return bool(self) and not [x for x in self if x not in \
527                                    "0123456789".split()]
528
529     def isdigit(self):
530         # TODO: This is not the right implementation.
531         """
532         S.isdigit() -> bool
533
534         Return True if all characters in S are digits
535         and there is at least one character in S, False otherwise.
536         """
537         return bool(self) and not [x for x in self if x not in \
538                                    "0123456789".split()]
539
540     def isidentifier(self):
541         """
542         S.isidentifier() -> bool
543
544         Return True if S is a valid identifier according
545         to the language definition.
546
547         Use keyword.iskeyword() to test for reserved identifiers
548         such as "def" and "class".
549         """
550         raise NotImplementedError()
551
552     def islower(self):
553         """
554         S.islower() -> bool
555
556         Return True if all cased characters in S are lowercase and there is
557         at least one cased character in S, False otherwise.
558         """
```

```
559         raise NotImplementedError()
560
561     def isnumeric(self):
562         """
563         S.isnumeric() -> bool
564
565         Return True if there are only numeric characters in S,
566         False otherwise.
567         """
568         raise NotImplementedError()
569
570     def isprintable(self):
571         """
572         S.isprintable() -> bool
573
574         Return True if all characters in S are considered
575         printable in repr() or S is empty, False otherwise.
576         """
577         raise NotImplementedError()
578
579     def isspace(self):
580         """
581         S.isspace() -> bool
582
583         Return True if all characters in S are whitespace
584         and there is at least one character in S, False otherwise.
585         """
586         return bool(self) and not [x for x in self if x not in \
587                                    StringProxy(" \t\r\n")]
588
589     def istitle(self):
590         """
591         S.istitle() -> bool
592
593         Return True if S is a titlecased string and there is at least one
594         character in S, i.e. upper- and titlecase characters may only
595         follow uncased characters and lowercase characters only cased ones.
596         Return False otherwise.
597         """
598         result = len(self) > 1
599         last_one_was_space = True
600         for i in range(len(self)):
601             if last_one_was_space and self[i].islower():
602                 result = False
603                 break
604             elif not last_one_was_space and self[i].isupper():
605                 result = False
606                 break
607             last_one_was_space = self[i].isspace()
608         return result
609
610     def isupper(self):
611         """
612         S.isupper() -> bool
613
614         Return True if all cased characters in S are uppercase and there is
```

```
615         at least one cased character in S, False otherwise.
616         """
617         raise NotImplementedError()
618 
619     # TODO: Change the type contract to join strings and not only chars
620     def join(self, iterable):
621         """
622         :types: iterable: str
623 
624         S.join(iterable) -> str
625 
626         Return a string which is the concatenation of the strings in the
627         iterable.  The separator between elements is S.
628         """
629         result = ""
630         first_loop = True
631         iterator = iter(iterable)
632         try:
633             while True:
634                 next_string = next(iterator)
635                 if not first_loop:
636                     result += self
637                 else:
638                     first_loop = False
639                 result += next_string
640         except StopIteration:
641             pass
642         return result
643 
644     @handle_varargs_and_defaults(' ')
645     def ljust(self, width, *args):
646         """
647         :types: width: int, args: str
648 
649         S.ljust(width[, fillchar]) -> str
650 
651         Return S left-justified in a Unicode string of length width. Padding is
652         done using the specified fill character (default is a space).
653         """
654         fillchar = StringProxy(args[0])
655         return self + max(0, width - len(self)) * fillchar
656 
657     def lower(self, *args, **kwargs):
658         """
659         S.lower() -> str
660 
661         Return a copy of the string S converted to lowercase.
662         """
663         raise NotImplementedError()
664 
665     # TODO: Change typecontracts behavior! This only test with chars
666     @handle_varargs_and_defaults(None)
667     def lstrip(self, *args):
668         """
669         :types: args: str
670 
```

```
671         S.lstrip([chars]) -> str
672 
673         Return a copy of the string S with leading whitespace removed.
674         If chars is given and not None, remove characters in chars instead.
675         """
676         remove = lambda x: x in StringProxy(args[0]) if args[0] else x.isspace()
677         i = 0
678         while i < len(self):
679             x = self[i]
680             if not remove(x):
681                 break
682             i += 1
683         return self[i:]
684 
685     def maketrans(self, *args, **kwargs):
686         """
687         str.maketrans(x[, y[, z]]) -> dict (static method)
688 
689         Return a translation table usable for str.translate().
690         If there is only one argument, it must be a dictionary mapping Unicode
691         ordinals (integers) or characters to Unicode ordinals, strings or None.
692         Character keys will be then converted to ordinals.
693         If there are two arguments, they must be strings of equal length, and
694         in the resulting dictionary, each character in x will be mapped to the
695         character at the same position in y. If there is a third argument, it
696         must be a string, whose characters will be mapped to None in the result.
697         """
698         raise NotImplementedError()
699 
700     def partition(self, sep):
701         """
702         S.partition(sep) -> (head, sep, tail)
703 
704         Search for the separator sep in S, and return the part before it,
705         the separator itself, and the part after it.  If the separator is not
706         found, return S and two empty strings.
707         """
708         if len(sep) == 0:
709             raise ValueError("empty separator")
710         i = self.find(sep)
711         head = self[:i]
712         if i == -1:
713             sep = self[i:i]
714             tail = self[i:i]
715         else:
716             tail = self[i + len(sep):]
717         return head, sep, tail
718 
719     @handle_varargs_and_defaults(None)
720     def replace(self, old, new, *args):
721         """
722         :types: old: str, new: str, args: list
723 
724         S.replace(old, new[, count]) -> str
725 
726         Return a copy of S with all occurrences of substring
```

```
727         old replaced by new.  If the optional argument count is
728         given, only the first count occurrences are replaced.
729         """
730         start, count = 0, args[0] if args[0] != None else len(self)
731         result, c = StringProxy(""), 0
732         try:
733             while c < count:
734                 index = self.index(old, start)
735                 result += self[start:index] + new
736                 start = index + max(len(old), 1)
737                 c += 1
738         except ValueError:
739             result += self[start:]
740         return result
741 
742     @handle_varargs_and_defaults(None, None)
743     def rfind(self, sub, *args):
744         """
745         :types: sub: str, args: list
746 
747         S.rfind(sub[, start[, end]]) -> int
748 
749         Return the highest index in S where substring sub is found,
750         such that sub is contained within S[start:end].  Optional
751         arguments start and end are interpreted as in slice notation.
752 
753         Return -1 on failure.
754         """
755         start, stop = args
756         start = 0 if start is None else start
757         stop = len(self) if stop is None else stop
758         start = max(start + len(self), 0) if start < 0 else start
759         if stop < 0:
760             stop = max(stop + len(self), 0)
761         else:
762             stop = min(stop, len(self))
763         if start > len(self):
764             return -1
765         elif stop < start:
766             return -1
767         elif sub == '':
768             return stop
769         else:
770             for i in range(stop - len(sub), start - 1, -1):
771                 if self[i:i + len(sub)] == sub:
772                     return i
773         return -1
774 
775     def rindex(self, sub, *args):
776         """
777         :types: sub: str, args: list
778 
779         S.rindex(sub[, start[, end]]) -> int
780 
781         Like S.rfind() but raise ValueError when the substring is not found.
782         """
```

```
783         i = self.rfind(sub, *args)
784         if i == -1:
785             raise ValueError("substring not found")
786         else:
787             return i
788 
789     def rjust(self, width, *args):
790         """
791         :types: width: int, args: str
792 
793         S.rjust(width[, fillchar]) -> str
794 
795         Return S right-justified in a string of length width. Padding is
796         done using the specified fill character (default is a space).
797         """
798         return self[::-1].ljust(width, *args)[::-1]
799 
800     def rpartition(self, sep):
801         """
802         S.rpartition(sep) -> (head, sep, tail)
803 
804         Search for the separator sep in S, starting at the end of S, and return
805         the part before it, the separator itself, and the part after it.  If the
806         separator is not found, return two empty strings and S.
807         """
808         head, sep, tail = self[::-1].partition(sep[::-1])
809         return tail[::-1], sep[::-1], head[::-1]
810 
811     def rsplit(self, sep=None, maxsplit=-1):
812         """
813         S.rsplit(sep=None, maxsplit=-1) -> list of strings
814 
815         Return a list of the words in S, using sep as the
816         delimiter string, starting at the end of the string and
817         working to the front.  If maxsplit is given, at most maxsplit
818         splits are done. If sep is not specified, any whitespace string
819         is a separator.
820         """
821         issep = (lambda x: x in sep) if sep != None else (lambda x: x.isspace())
822         maxsplit = maxsplit if maxsplit >= 0 else len(self) + 1
823         i, splits = len(self) - 1, 0
824         result = []
825         acum = ""
826         while i >= 0:
827             if issep(self[i]) and splits < maxsplit:
828                 result.insert(0, acum)
829                 splits += 1
830                 acum = ""
831             else:
832                 acum = self[i] + acum
833             i -= 1
834         result.insert(0, acum)
835         return result
836 
837     def rstrip(self, *args):
838         """
```

```
839         S.rstrip([chars]) -> str
840 
841         Return a copy of the string S with trailing whitespace removed.
842         If chars is given and not None, remove characters in chars instead.
843         """
844         return self[::-1].strip(*args)[::-1]
845 
846     def split(self, sep=None, maxsplit=-1):
847         """
848         S.split(sep=None, maxsplit=-1) -> list of strings
849 
850         Return a list of the words in S, using sep as the
851         delimiter string.  If maxsplit is given, at most maxsplit
852         splits are done. If sep is not specified or is None, any
853         whitespace string is a separator and empty strings are
854         removed from the result.
855         """
856         issep = (lambda x: x in sep) if sep != None else (lambda x: x.isspace())
857         maxsplit = maxsplit if maxsplit >= 0 else len(self) + 1
858         i, splits = 0, 0
859         result = []
860         acum = ""
861         while i < len(self):
862             if issep(self[i]) and splits < maxsplit:
863                 result.insert(len(self), acum)
864                 splits += 1
865                 acum = ""
866             else:
867                 acum += self[i]
868             i += 1
869         result.insert(len(self), acum)
870         return result
871 
872     @handle_varargs_and_defaults(False)
873     def splitlines(self, *args):
874         """
875         :types: args: list
876         S.splitlines([keepends]) -> list of strings
877 
878         Return a list of the lines in S, breaking at line boundaries.
879         Line breaks are not included in the resulting list unless keepends
880         is given and true.
881         """
882         keepends = args[0]
883         i = 0
884         result = []
885         acum = ""
886         while i < len(self):
887             if self[i] == '\r' or self[i] == '\n':
888                 end = self[i]
889                 if i < len(self) - 1 and self[i:i+2] == '\r\n':
890                     end = '\r\n'
891                     i += 1
892                 if keepends:
893                     acum += end
894                 result.insert(len(self), acum)
```

```
895                 acum = ""
896             else:
897                 acum += self[i]
898             i += 1
899         result.insert(len(self), acum)
900         return result
901 
902     @handle_varargs_and_defaults(None, None)
903     def startswith(self, prefix, *args):
904         """
905         S.startswith(prefix[, start[, end]]) -> bool
906 
907         Return True if S starts with the specified prefix, False otherwise.
908         With optional start, test S beginning at that position.
909         With optional end, stop comparing S at that position.
910         prefix can also be a tuple of strings to try.
911         """
912         start, stop = args
913         return self[start:stop] == prefix
914 
915     def strip(self, *args):
916         """
917         S.strip([chars]) -> str
918 
919         Return a copy of the string S with leading and trailing
920         whitespace removed.
921         If chars is given and not None, remove characters in chars instead.
922         """
923         return self.lstrip(*args).rstip(*args)
924 
925     def swapcase(self):
926         """
927         S.swapcase() -> str
928 
929         Return a copy of S with uppercase characters converted to lowercase
930         and vice versa.
931         """
932         result = ""
933         for c in self:
934             if c.isupper():
935                 result += c.lower()
936             elif x.islower():
937                 result += c.upper()
938             else:
939                 result += c
940         return result
941 
942     def title(self):
943         """
944         S.title() -> str
945 
946         Return a titlecased version of S, i.e. words start with title case
947         characters, all remaining cased characters have lower case.
948         """
949         result = ""
950         last_one_was_space = True
```

```
951         for i in range(len(self)):
952             if last_one_was_space and not self[i].isspace():
953                 result += self[i].upper()
954             elif not last_one_was_space and not self[i].isspace():
955                 result += self[i].lower()
956             else:
957                 result += self[i]
958             last_one_was_space = self[i].isspace()
959         return result
960 
961     def translate(self, *args, **kwargs):
962         """
963         S.translate(table) -> str
964 
965         Return a copy of the string S, where all characters have been mapped
966         through the given translation table, which must be a mapping of
967         Unicode ordinals to Unicode ordinals, strings, or None.
968         Unmapped characters are left untouched. Characters mapped to None
969         are deleted.
970         """
971         raise NotImplementedError()
972 
973     def upper(self, *args, **kwargs):
974         """
975         S.upper() -> str
976 
977         Return a copy of S converted to uppercase.
978         """
979         raise NotImplementedError()
980 
981     def zfill(self, *args, **kwargs):
982         """
983         S.zfill(width) -> str
984 
985         Pad a numeric string S with zeros on the left, to fill a field
986         of the specified width. The string S is never truncated.
987         """
988         raise NotImplementedError()
```

A.5. SliceProxy

```
1  class SliceProxy(ProxyObject):
2      emulated_class = slice
3  
4      def __init__(self, initial=None):
5          if isinstance(initial, slice) or isinstance(initial, SliceProxy):
6              self.start = initial.start
7              self.stop = initial.stop
8              self.step = initial.step
9          elif initial is None:
10             nargs = IntProxy()
11             self.start = None
12             self.step = None
```

```
13              if nargs == 1:
14                  self.stop = IntProxy()
15              elif nargs == 2:
16                  self.start = IntProxy()
17                  self.stop = IntProxy()
18              else:
19                  self.start = IntProxy()
20                  self.stop = IntProxy()
21                  self.step = IntProxy()
22                  self._facts.append((self.step != 0).formula)
23
24          else:
25              raise TypeError("type for initial: (%s) not supported" %\
26                              (str(type(initial).__name__)))
27
28      def indices(self, len):
29          """
30          :types: len: int
31
32          Had to read Python slice.indices() source code to find out the right way.
33          """
34          start = self.start
35          stop = self.stop
36          step = self.step if self.step != None else 1
37          if start is None:
38              start = len-1 if step < 0 else 0
39          start = start + len if start < 0 else start
40          if start < 0:
41              start = -1 if step < 0 else 0
42          if start >= len:
43              start = len -1 if step < 0 else len
44          if stop is None:
45              stop = -1 if step < 0 else len
46          stop = stop + len if stop < 0 else stop
47          if stop < 0:
48              stop = -1 if step < 0 else 0
49          if stop >= len:
50              stop = len -1 if step < 0 else len
51          if step == 0:
52              raise ValueError("SliceProxy step cannot be zero")
53          return (start, stop, step)
54
55      def __repr__(self):
56          return "SliceProxy(%s, %s, %s)" % (self.start, self.stop, self.step)
57
58      def _concretize(self, model):
59          start = _concretize(self.start, model)
60          stop = _concretize(self.stop, model)
61          step = _concretize(self.step, model)
62          return slice(start, stop, step)
```

A.6. proxify

```
def proxify(obj):
    """
    Transforms object :obj: into a proxy class object
    """
    if isinstance(obj, ProxyObject) or isinstance(obj, type(None)):
        return obj
    if isinstance(obj, type) and issubclass(obj, ProxyObject):
        return obj()
    # Dynamic generation of this table allow adding new proxy classes at run time
    real_to_proxy = {x.emulated_class: x for x in ProxyObject.__subclasses__()}
    instance = None if isinstance(obj, type) else obj
    instance_type = type(obj) if instance is not None else obj
    assert(hasattr(instance_type, '__name__'))
    if instance_type.__name__ in __builtins__:
        # instance_type is builtin, we just return the corresponding Proxy
        # Object instance
        if instance is not None:
            return real_to_proxy[instance_type](instance)
        else:
            return real_to_proxy[instance_type]()
    elif instance_type == type(None):
        return None
    else:
        # instance_type is a user_defined class
        tc = contracts.TypeContract.parse(instance_type.__init__)
        args_count = obj.__init__.__code__.co_argcount
        if not tc and args_count > 1:
            raise contracts.ContractError("Arguments without contracts declared.")
        elif args_count == 1:
            args, kwargs = [], {}
        else:
            args, kwargs = tc.types['args'], tc.types['kwargs']
            args = args[1:] # Removing the self reference.
        for i, arg in enumerate(args):
            assert(arg)
            # Get `randomly` some type, and try them all!
            args[i] = proxify(_branch_iterate(arg))
        for k, v in kwargs.items():
            kwargs[k] = proxify(_branch_iterate(v))
        result = instance_type(*args, **kwargs)
        result._concretize = _types.MethodType(_concretize, result)
        return result
    #TODO: soporte para kwargs
```

Apéndice B

Código fuente de Implementation Test

B.1. Implementation Test

```
#! /usr/bin/env python3
#coding=utf-8

import proxy
import sys
from types import FunctionType
import contracts
from contracts import TypeContract
from capture import Capturing
import z3 as _z3
import traceback
try:
    from clint.textui import colored
    # ('red', 'green', 'yellow', 'blue', 'black', 'magenta', 'cyan', 'white')
except ImportError:
    print("`clint` module not found, please run `pip3 install clint`.")
    sys.exit(-1)


def test_class(cls, method=None, show_progress=True, show_statistics=True,
               start_from=None, debug=False):
    passed_tests, failed_tests = 0, 0

    if show_progress or show_statistics:
        print(" > Running automated tests cases for class %s" \
            % colored.blue(cls.__name__))

    # Get list of methods to test
    proxy_class_methods = [x for x in dir(cls) if \
                           type(getattr(cls, x)) == FunctionType]
```

```
31      not_relevant_methods = ["__init__", "__repr__", "__str__", "__hash__"]
32      real_class_methods = [x for x in dir(cls.emulated_class)]
33      if not method:
34          methods_list = [getattr(cls, x) for x in proxy_class_methods\
35                      if x in real_class_methods and not x in not_relevant_methods]
36      else:
37          methods_list = [getattr(cls, method)]
38      # Test each method
39      skip = False if not start_from else True
40      for method in methods_list:
41          # Print method name
42          if skip and method.__name__ == start_from:
43              skip = False
44          if skip:
45              continue
46          try:
47              r, e, paths = test_method(cls, method)
48          except contracts.ContractError as e:
49              failed_tests += 1
50              print(colored.red("\n    ContractError: ") + str(e))
51              print(status_string(method, False, 0))
52              continue
53          passed_tests += 1 if r else 0
54          failed_tests += 0 if r else 1
55          # Print exit status
56          if show_progress:
57              print(status_string(method, r, paths))
58              if not r: print(error_string(e, indent="    ", debug=debug))
59
60      # Statistics
61      if show_statistics:
62          print(" < Runned %s tests: %s/%s [passed/failed]\n" %\
63              (passed_tests + failed_tests, passed_tests, failed_tests))
64      return passed_tests, failed_tests
65
66
67  def test_method(cls, method):
68      """
69      r: if excecuted withot errors
70      e: extra info for errors
71      """
72      r = True
73      e = {}
74      paths = 0
75      real_class = cls.emulated_class
76      executions = proxy.explore(method)
77      for result in executions:
78          args = result['args']
79          kwargs = result['kwargs']
80          returnv = result['result']
81          exception = result['exception']
82          outputs = result['outputs']
83          paths += 1
84          if isinstance(exception, proxy.MaxDepthError):
85              continue
86          if isinstance(exception, _z3.Z3Exception):
```

```
87              # It's definitely an internal error, so let it propogate in this case
88              raise exception
89          real_exception = None
90          real_result = None
91          try:
92              with Capturing() as output:
93                  method_name = method.__name__
94                  real_result = getattr(real_class, method_name)(*args, **kwargs)
95          except Exception as err:
96              real_exception = err
97          if real_result != returnv or type(real_exception) != type(exception):
98              r = False
99              e = {"args": (args, kwargs),
100                  "real": (real_result, real_exception),
101                  "proxy": (returnv, exception),
102                 }
103             break
104     return (r, e, paths)
105
106
107 # Pretty Formatted strings for status and errrors
108 def error_string(e, indent="    ", debug=False):
109     real_exception = e["real"][1] if e["real"][1] else None
110     real_exception_tname = type(real_exception).__name__ if real_exception \
111                                                          else None
112     exception = e["proxy"][1] if e["proxy"][1] else None
113     exception_tname = type(exception).__name__ if exception else None
114     result = e["proxy"][0]
115     real_result = e["real"][0]
116     args = e["args"][0]
117     kwargs = e["args"][1]
118     # Error found! Pretty-Print error
119     params = [repr(x) for x in args] + ["%s=%s" %(k, v) for (k,v) \
120                                        in kwargs.items()]
121     error_str = colored.red("ERROR FOUND")
122     error_params = error_str + ": running with params %s" % (", ".join(params))
123     # Get the right error description
124     if real_exception and not exception:
125         error_desc = "Expected exception: %s but no exception was raised" \
126             %(real_exception_tname)
127     elif real_exception and exception:
128         error_desc = "Expected exception: %s but ProxyObject raised: %s" \
129             %(real_exception_tname, exception_tname)
130     elif exception:
131         error_desc = "Unexpected exception: %s (No exeception should be raised)"\
132             %(colored.red(exception_tname))
133     else:
134         error_desc = "Expected result: %s, but ProxyObject returned: %s" \
135             %(repr(real_result), repr(result))
136     if debug and exception:
137         error_desc += "\n\n" + repr(exception)
138         traceback = e["proxy"][1].__traceback__)
139         error_desc += "\n"+"\n".join(traceback.format_tb(traceback)
140     return (indent + error_params + '\n' + indent + \
141         error_desc).replace('\n', '\n' + indent)
142
```

```
143
144 def status_string(method, r, p, indent="    "):
145     symbol = colored.green("✓") if r else colored.red("✗")
146     return indent + symbol + "  " + method.__name__ + "\t[paths: %s]" % p
147
148
149 if __name__ == '__main__':
150     import argparse
151     parser = argparse.ArgumentParser(add_help=True)
152     parser.add_argument("cls", action="store", nargs="?",
153                         help="Run tests for class name 'cls'")
154     parser.add_argument("-m", "--method", action="store",
155                         help="Run tests for specific method in class 'cls'")
156     parser.add_argument("-d", "--debug", action="store_true",
157                         help="Show more debug info for exceptions raised")
158     parser.add_argument("-f", "--fromm", action="store",
159                         help="skip tests until method with given name is found")
160
161     args = parser.parse_args()
162     classes = [x.__name__ for x in proxy.ProxyObject.__subclasses__()] if\
163                                                 not args.cls else [args.cls]
164     passed_tests, failed_tests = 0, 0
165     for c in classes:
166         cls = getattr(proxy, c, None)
167         if not cls:
168             msg = ": class %s not found\n" % c
169             print(colored.red("\nWARNING") + msg)
170         elif proxy.ProxyObject not in cls.__bases__:
171             msg = ": class %s does not inherit from ProxyObject\n" % c
172             print(colored.red("\nWARNING") + msg)
173         else:
174             try:
175                 p, f = test_class(cls, method=args.method, debug=args.debug,
176                                   start_from=args.fromm)
177                 passed_tests += p
178                 failed_tests += f
179             except Exception as e:
180                 print(colored.red("\nINTERNAL ERROR") + ": %s" % e)
181                 raise e
```

Apéndice C

Funciones de ejemplo

C.1. Funciones de ejemplo para probar PEF

```
1  import builtins
2
3  def simple(i):
4      """
5      :type: i: builtins.int
6      :assume: i > 0
7      """
8      if i:
9          print("i es diferente a 0")
10     else:
11         print("i es igual a 0")
12
13
14 def simple_bool(a, b):
15     """
16     :type: a: NoneType, b: bool
17     """
18     return a and b
19
20
21 def fun(a):
22     """
23     :type: a: int
24     """
25     a = a + 3
26     if a:
27         raise AssertionError('Error')
28     else:
29         return a
30
31
32 def funab(a, b):
33     """
34     :type: a: int, b: int
```

```
35      """
36      a = a + 3
37      if a == b:
38          if a != a:
39              raise AssertionError('Esto no debería pasar')
40          raise AssertionError('Esto es posible')
41      else:
42          return a
43
44
45  def funfor(a):
46      """
47      :type: a: int
48      """
49      a = a + 3
50      for i in range(a):
51          if a == 2:
52              raise AssertionError('Error')
53      return a
54
55
56  def funlist_in(l, i, j):
57      """
58      :type: l: list, i: int, j: int
59      """
60      if len(l) == i:
61          if l[j] == i:
62              print("Éxito")
63          else:
64              print("Casi")
65      else:
66          print("No se ingreso al if")
67
68
69  def funlist_iter(l): # FIXME: Performance issue
70      """
71      :type: l: list
72      """
73      for i, x in enumerate(l):
74          print(x)
75
76
77  def funlist_slice(l, start, stop, step):
78      """
79      :type: l: list, start: int, stop: int, step: int
80      """
81      return l[start:stop:step]
82
83
84  def assert_not_divisible(x, y):
85      """
86      :type: x: int, y: int
87      """
88      if(x > y):
89          r = divisible(x, y)
90          if r:
```

```
91              raise Exception("x no tiene que ser divisible por y")
92          else:
93              print("Correcto!")
94      else:
95          raise Exception("x tiene que ser mas grande que y")
96
97
98  def divisible(x, y):
99      """
100     :type: x: int, y: int
101     :assume: x > y
102     """
103     if x > y:
104         return divisible(x - y, y)
105     elif x == y or x == 0:
106         return True
107     else:
108         return False
109
110
111 class Test():
112     counter = 0
113     def __init__(self, a, l):
114         """
115         :types: a: int, l:list
116         """
117         self.a = a
118         self.l = l
119
120     def recalculate(self):
121         self.counter += 1
122         self.a = self.l.__len__()
123
124     def __str__(self):
125         return "Test <%s, %s>" % (self.a, self.l)
126
127
128 def test_Test(t):
129     """
130     :types: t:Test
131     """
132     if t.a != t.l.__len__():
133         t.recalculate()
134     else:
135         t.l[0] = 5
136     return t.l[0]
137
138
139 def test_mock_Test(a, l):
140     """
141     :types: a:int, l: list
142     """
143     if a != l.__len__():
144         a = l.__len__()
145     else:
146         l[0] = 5
```

```
147     return l[0]
148
149
150 def test_gt(a, b):
151     """
152     :types: a: int, b: int
153     """
154     if a > b:
155         return b - a
156     else:
157         return a - b
158
159
160 def test_kw_args(a=1, b=2):
161     """
162     :types: a: int, b: int
163     """
164     if a > b:
165         return True
166     else:
167         return False
168
169
170 class Simple():
171     def __init__(self, a, b):
172         """
173         :types: a: int, b: int
174         """
175         if a > b:
176             print ("Primero mayor!")
177         else:
178             print ("Primero NO mayor!")
179
180
181 def simple_test(s):
182     """
183     :types: s: Simple
184     """
185     return s
186
187
188 class Queue(object):
189     def __init__(self, initial):
190         """
191         :types: initial: list
192         """
193         self.queue = []
194         if initial:  # TODO: ver que sea iterable
195             self.queue = initial #[x for x in initial]
196
197     def push(self, item):
198         self.queue.append(item)
199
200     def pop(self):
201         try:
202             return self.queue.pop()
```

```
203         except IndexError:
204             raise IndexError("Pop from empty Queue")
205 
206     def __len__(self):
207         return len(self.queue)
208 
209     def __repr__(self):
210         return self.queue.__repr__()
211 
212 
213 def example00(pushes, pops):
214     """
215     :types: pushes: int, pops: int
216     """
217     q = Queue([])
218     i = 0; j = 0
219 
220     while i < pushes:
221         q.push(i)
222         i += 1
223     while j < pops:
224         q.pop()
225         j += 1
226     return q.__len__() > 0  # Tiene elementos?
227 
228 
229 def example01(queue, pushes, pops):
230     """
231     :types: queue: Queue, pushes: int, pops: int
232     """
233     i = 0; j = 0
234     while i < pushes:
235         queue.push(i)
236         i += 1
237     while j < pops:
238         queue.pop()
239         j += 1
240     result = len(queue) > 0
241     return result  # Tiene elementos?
242 
243 
244 def example02(a, b):
245     """
246     :types: a:int, b: int
247     """
248     return a//b
249 
250 
251 def multitypes(a, b):
252     """
253     :types: a: [int, list], b: [int, list]
254     """
255     a_m = "a es int" if isinstance(a, int) else "a es list"
256     b_m = "b es int" if isinstance(b, int) else "b es list"
257     print ("%s y %s" % (a_m, b_m))
258 
```

```
259
260 def test_strings1(s1, s2):
261     """
262     :types: s1: str, s2: [str]
263     """
264     result = ""
265     if s1 == s2:
266         tmp = s1 + s2
267         result += tmp
268     elif len(s1) < len(s2):
269         if s1 > s2:
270             result = "%s es mas corto que %s pero mayor lexicográficamente." %\
271                      (s1, s2)
272         else:
273             result = "%s mayor que %s en todo sentido." % (s2, s1)
274     else:
275         if s1[0] == s1[0]:
276             result = "{0} y {1} coinciden en el primer caracter:{2}"\
277                                     .format(s1, s2, s1[0])
278         elif s1.isdecimal():
279             result = "s1 es un número! (y nada que ver con s2)."
280     return result
281
282
283 import types
284
285
286 def print_species(tree):
287     """
288     :type: tree: list
289     """
290     if type(tree) == list:
291         for child in tree:
292             print_species(child)
293     else:
294         print(tree)
295
296
297 class Lista:
298     def __init__(self, value, sig):
299         """
300         :type: value: [int], sig: [Lista, NoneType]
301         """
302         self.value = value
303         self.sig = sig
304
305     def __repr__(self):
306         if self.sig == None:
307             return repr(self.value)
308         else:
309             return repr(self.value) + " -> " + repr(self.sig)
310
311
312
313 ls = Lista(1,Lista(2,None))
314
```

```
315
316 def sum_lista(ls, ka=None):
317     """
318     :type: ls: Lista, ka: int
319     """
320     if (ls.sig == None):
321         return ls.value
322     else:
323         return ls.value + sum_lista(ls.sig)
324
325
326 class BinaryTree:
327     def __init__(self, rootObj, leftChild=None, rightChild=None):
328         """
329         :type: rootObj: int, leftChild:[BinaryTree, None], \
330                                 rightChild: [BinaryTree, None]
331         """
332         self.key = rootObj
333         self.leftChild = leftChild
334         self.rightChild = rightChild
335
336     def __repr__(self):
337         return "<" + repr(self.leftChild)  + " ,(" + repr(self.key) + "), " +\
338                      repr(self.rightChild) + ">"
339
340
341 def insertLeft(t, newNode):
342     """
343     :type: t: BinaryTree, newNode: int
344     """
345     if t.leftChild == None:
346         t.leftChild = BinaryTree(newNode)
347     else:
348         tn = BinaryTree(newNode, leftChild=t.leftChild)
349         t.leftChild = tn
350     return t
351
352
353 import math
354
355 def sum_fac(n):
356     """
357     :type: n: int
358     """
359     if n < 0:
360         raise ValueError
361     facs = [math.factorial(i) for i in range(n+1)]
362     return sum(facs)
363
364
365 def sum_fac2(n):
366     """
367     :type: n: int
368     """
369     if n < 0:
370         raise ValueError
```

```
371     elif n == 0:
372         return 1
373     elif n == 1:
374         return 2
375     elif n == 2:
376         return 4
377     elif n >= 3:
378         return sum_fac2(n-1) * n - sum_fac2(n-3) * (n-1)
379 
380 def test_sum_fac(n):
381     """
382     :type: n: int
383     """
384     return sum_fac2(n) == sum_fac(n)
385 
386 def fact_spec(x):
387     """
388     :type: x: int
389     """
390     n = 1
391     while x > 0:
392         n *= x
393         x -= 1
394     return n
395 
396 def faulty_fact(x):
397     """
398     :type: x: int
399     :ensure: returnv == fact_spec(x)
400     """
401     if x == 40:
402         return 123456789
403     n = 1
404     while x > 0:
405         n *= x
406         x -= 1
407     return n
408 
409 
410 def suma_con_bug(a, b):
411     """
412     :types: a: int, b: int
413     """
414     result = a + b
415     if result == 42:
416         print("La gran respuesta.")
417     if a - b == 42:
418         raise Exception("Bug")
419     return result
420 
421 def f_l_j(l, j):
422     """
423     :types: l: list, j: int
424     """
425     l += [1,2,3]
426     return l[j]
```

```
427
428 # examples paper king
429 def sum1(a,b,c):
430     """
431     :type: a: int, b: int, c: int
432     """
433     x = a + b
434     y = b + c
435     z = x + y - b
436     return z
437
438 def power(x,y):
439     """
440     :type: x: int, y: int
441     """
442     z = 1
443     j = 1
444     while y >= j:
445         z = z * x
446         j = j + 1
447     return z
448
449 # examples paper bruni
450
451 def abs (x):
452     """
453     :type: x: int
454     """
455     if x >= 0:
456         return x
457     else:
458         return -x
```