Universidad Complutense de Madrid
Facultad de Informática
Departamento de Sistemas Informáticos y Computación

SISTEMA INTEGRADO DE TESTING BASADO EN ASERTOS

ASSERTION-BASED INTEGRATED TESTING SYSTEM

Trabajo de Fin de Grado
Grado en Ingeniería Informática

Junio 2020

Autor:
Alejandro Cilleros Garrudo
Directores:
Ricardo Peña Marí
Jaime Sánchez Hernández

Agradecimientos

Quiero dar las gracias a mis padres y a mi hermana por apoyarme en todo momento, confiar en mi y por darme fuerzas y ánimos para nunca rendirme. A Elena, por todo el apoyo y cariño que he recibido por su parte. Sin ella, no hubiera logrado estar donde estoy. Al resto de mi familia y amigos, por confiar en mí y ayudarme siempre que lo he necesitado, en especial, a mi abuela Benigna, quién siempre ha estado junto a mi, y vió el comienzo y avances de este trabajo, pero por desgracia no lo ha podido ver terminado. Por último, y no por ello menos importante, quería agradecer a mis directores, Ricardo y Jaime, la ayuda que me han ofrecido, su paciencia y dedicación con este trabajo. Sin ellos esto no hubiera sido posible

Muchas gracias a todos de corazón.

Resumen

La plataforma CAVI-ART para la verificación automática de programas ofrece numerosas herramientas para el análisis de los mismos entre las que encontramos dos generadores de casos de prueba. Ninguno de ellos permite la generación de estos automáticamente, teniendo que interactuar manualmente para obtenerlos.

El objetivo de este trabajo es mejorar las herramientas anteriores, automatizando este proceso gracias al uso de la API de Z3, Z3Py, que es la que nos permitirá poder interactuar con los casos obtenidos. Además, otro de los objetivos establecidos será ampliar estas herramientas, añadiendo una adicional que será la encargada de validar los programas a partir de los casos de prueba generados con las anteriores.

Palabras clave

Pruebas de ejecución, resolutores SMT, generador de restricciones, resolución de restricciones, estructuras de datos, casos de caja blanca, casos de caja negra, validador de casos de prueba.

Abstract

The CAVI-ART platform for assisted program validation offers a set of tools for analyzing them, including two test case generators. None of them allow the generation of these cases automatically, having to interact manually to get them.

The aim of this work is to improve the previous tool, automating this process thanks to te use of the Z3 API, Z3Py, which is what will allow us to interact with the obtained cases. In addition, another stablished objective is to expand these tools, adding an additional one for validating the programs, based on the test cases generated with the previous ones.

Keywords

Testing, SMT solvers, constraints generator, constraints solving, data structures, white box cases, black box cases, test case validator.

Índice general

Capítulo 1

Introducción

Castellano

Una vez terminada la creación de un determinado programa, nos falta la tarea de probar este. A esta tarea se le denomina *testing* y consiste en comprobar si el programa se comporta tal y como se espera, mediante su ejecución con unos casos de prueba y la verificación de los resultados devueltos en cada ejecución. El *testing* es necesario, pero muy costoso: hay que idear esos casos de prueba, escribirlos, ejecutarlos y decidir si los resultados devueltos por el programa bajo prueba son o no correctos. Todo ello, implica un gran esfuerzo humano y muchas posibilidades de cometer errores, como por ejemplo, probar el programa con casos demasiado sencillos, dar por buenos casos fallidos, o peor aún, dar por fallidos casos ejecutados correctamente. Además, esta tarea puede llegar a complicarse aún más si el programa exige precondiciones complejas para sus parámetros como, por ejemplo, que ha de recibir listas o *arrays* ordenados, árboles equilibrados, etc. Ello exigiría escribir código auxiliar para generar dichas estructuras. Debido a todo esto, es por lo que el resultado que se obtiene en la práctica es que se hace menos *testing* del que sería necesario. Por ello, el objetivo que nos planteamos en este trabajo puede resumirse en: realizar muchas pruebas y hacerlo sin intervención humana alguna, es decir, automáticamente.

Tradicionalmente, las estrategias de pruebas se clasifican en dos grandes grupos: de caja negra y de caja blanca [1]. La primera de estas se encarga de generar casos de prueba teniendo en cuenta únicamente la precondición del programa y de validar con la postcondición los resultados obtenidos de ejecutar el programa con cada uno de estos casos. Es decir, con esta estrategía no se tendrá en cuenta el código de nuestro programa, simplemente la especificación. Por el contrario, la segunda de estas, además de tener en cuenta la precondición para generar estos casos, tendrá en cuenta los diferentes caminos que el programa puede presentar, es decir, tendrá en cuenta el código de este. De igual manera, contará con la postcondición para validar los resultados obtenidos de la ejecución del programa con los casos generados.

Para alcanzar nuestro objetivo, partiremos del trabajo previo realizado en el marco del proyecto CAVI-ART, el cuál se ha desarrollado en los últimos años en el

Departamento de Sistemas Informáticos y Computación de la UCM [8,9], ademas de tener en cuenta una serie de propiedades: Los programas de esta plataforma, vienen provistos de una precondición y una postcondición formal. La razón es que el objetivo principal de dicha plataforma es la verificación formal de programas. Además, dentro de este proyecto, se han desarrollado herramientas que convierten asertos formales en restricciones para ser tratadas por resolutores SMT [2] (del inglés, *Satisfiability Modulo Theories*). Dichos resolutores además de admitir la definición de funciones simples, también admiten la definición de funciones recursivas tales como la *altura* de un árbol, el *tamaño* de una lista o el predicado *isBST* que nos dice si un árbol binario es o no de búsqueda. Estos a su vez, son capaces de proporcionar modelos para dichas restricciones. Definiremos un *modelo* como la interpretación que se le da a las variables para que hagan ciertas todas esas restricciones. Por ejemplo, si establecemos la restricción $altura(t) \leq 3 \wedge isBST(t)$, el resolutor nos dice que dicha restricción es satisfactible y además de esto, proporciona una interpretación para la variable t que resulta ser un árbol binario de búsqueda de altura menor o igual que 3. Otras herramientas que encontramos dentro del proyecto CAVI-ART son capaces de calcular todos los caminos de ejecución posibles de un programa y, además recolectar para cada camino, las condiciones booleanas que se cumplen en el mismo.

Este trabajo, pretende completar dichas herramientas y construir un sistema integrado donde el usuario solo tenga que proporcionar el código y la especificación formal de la unidad a probar, y a partir de esto, el resto del proceso se realice automáticamente. Más concretamente nos plantearemos 2 objetivos:

- *Realizar pruebas exhaustivas de tipo caja blanca.* El usuario especificará la *profundidad* que desea para sus caminos y el rango de los valores que los elementos de las estructuras de datos pueden tomar. El sistema sintetizará y ejecutará un caso de prueba para cada camino. La profundidad de un camino se definirá en adelante pero, en esencia, se refiere al número máximo de iteraciones de los bucles y al número máximo de desplegados de las funciones recursivas.

- *Realizar pruebas exhaustivas de tipo caja negra.* El usuario especificará los tamaños máximos que desea para las estructuras de datos, además del rango para los valores de los elementos de estas, y el sistema sintetizará y ejecutará *todos* los casos de prueba que satisfacen la precondición y no exceden dichos tamaños. Por ejemplo, si la unidad a probar es una función *insertArray* que inserta un valor entero x en un array ordenado a, el usuario decide que los arrays serán de tamaño máximo 3 y que los enteros a usar estarán en el rango $1 - 4$, el sistema sintetizará todas las combinaciones de enteros y arrays ordenados dentro de esos límites y luego ejecutará dichos casos.

Otro aspecto relevante del *testing* es la validación de los resultados. Como describíamos al comienzo, este proceso de hacerse manualmente puede llegar a ser muy tedioso y propenso a cometer errores. Dado que, en el contexto de CAVI-ART, disponemos de la postcondición, nos planteamos como último objetivo la *validación automática* de los casos de prueba. Usaremos el resolutor SMT para comprobar

que los valores devueltos por la unidad bajo prueba satisfacen efectivamente la postcondición, previa conversión de esta a un conjunto de restricciones para el SMT.

Estructuraremos el trabajo en los siguientes capítulos: en el primero encontramos la presentación del resolutor SMT que utilizaremos, en este caso Z3 [7], y de su API para Python, Z3Py, junto con sus características y funciones más destacables en este trabajo. Además, en ese primer capítulo, se describirán algunas de las herramientas ya desarrolladas dentro del proyecto CAVI-ART que utilizaremos para conseguir los objetivos planteados. En los capítulos 3 y 4, se explicará en detalle las estrategias seguidas para la obtención de los casos de caja negra y de caja blanca respectivamente. En el capítulo 5, se desarrollará la técnica empleada para probar automáticamente nuestro programa con los casos ya generados. Por último, dedicaremos el capítulo 6 a mostrar resultados de ejecución del sistema completo para diferentes programas y cómo es capaz nuestro sistema de detectar errores en estos.

Inglés

Once the creation of a certain program is finished, it remains the task of testing it. This task is called *testing* and consists of verifying whether the program behaves as we expected by it execution with some test cases and checking the results returned in each execution. Testing is necessary, but it is very expensive: you have to think about those test cases, write them, run them and decide whether the results returned by the program under test are correct or not. All this involves a great deal of human effort and many possibilities of making mistakes, such as testing the program with too simple cases, accepting failed cases, or worse, assuming correctly executed cases which failed. Moreover, this task can be very complicated if the program requires complex preconditions for its parameters, such as sorted lists or *arrays*, balanced trees, etc. This would require writing auxiliary code to generate such structures. Due to this, the result that we obtained in practice that less testing is done than it would be necessary. For that, the objective that we set in this work can be established as: carrying out many tests and doing it without any human intervention, that is, automatically.

Traditionally, testing strategies are classified in two big groups: black box and white box ones [1]. The first one generates test cases taking care only of the program precondition and validating with the postcondition the results obtained by running the program with each one of these cases. That is, with this strategy, the code of our program will not be taken care of our program, only the specification. In contrast to this, the second one, as well as taking care of the precondition to generate these cases, it will take care of the differents paths that the program can follow, in other words, it will take care of its code. Likewise, it will have the postcondition to validate the obtained results from running the program with the generated cases.

To get our objective, we will start from the previous work carried out in the framework of the CAVI-ART project, which has been developed in the last few years in the Software Systems and Computation Department of UCM [8, 9] and we will consider a set of properties: The programs of this platform have formal precondition and postcondition. The reason for that is that the main objective of this platform is the formal verification of programs. As well as, inside this project, we find tools which convert formal assertions into constraints to be treated by SMT [2] solvers (*Satisfiability Modulo Theories*). These solvers, besides supporting the definition of simple functions, also supporting the definition of recursive functions such as the *height* of a tree, the *size* of a list or the *isBST* predicate which tells us if a binary tree is a search tree or not. They are able to provide models for these constraints. We will define a *model* as the interpretation given to the variables to make all these constraints true. For example, if we set the constraint $height(t) \leq 3 \wedge isBST\ (t)$, the solver tells us that this constraint is satisfactory and in addition to this, it provides an interpretation for the variable t which turns out being a search binary tree of height less than or equal to 3. Other tools which we finde inside CAVI-ART project, are able to computing all the possible execution paths of a program and, as well as collecting for each path, the Boolean conditions defining it.

This work aims to complete these tools and to build an integrated system where the user has only to provide the code and the formal specification of the unit to be tested, and from that point, the rest of the process is carried out automatically. More specifically we will consider two objectives:

- *Do exhaustive white box tests.* The user will specify the *depth* which he wants for their paths and the range of values that elements of data structures can take. The system will synthesize and run a test case for each path. The depth of a path will be defined later, but in essence it refers to the maximum number of iterations of the loops and the maximum number of unfoldings of the recursive functions.

- *Do exhaustive black box tests.* The user will specify the maximum sizes which he wants for the data structures, as well as the range for the values of the elements of these, and the system will synthesize and execute all the test cases which satisfy the precondition and do not exceed those sizes. For example, if the unit to test is a function *insertArray* which inserts an integer value x in an sorted array a, the user decides that the arrays will have 3 as maximum size and the integers to use will be in the range $1-4$, the system will synthesize all the combinations of integers and sorted arrays within those limits and then it will execute these cases.

Another relevant aspect of *testing* is the validation of the results. As we described at the beginning, this process of carrying out manually can be very tedious and error prone. Since, in the context of CAVI-ART, we have the postcondition, we aim, as the ultimate goal, the *automatic validation* of test cases. We will use the SMT solver to verify that the returned values by the unit under test effectively satisfies the postcondition, after converting it to a set of constraints for the SMT.

We will structure the work in the following chapters: in the first one we find the presentation of the SMT solver which we will use, in this case Z3 [7], and its API for Python, Z3Py, with its most notable features and functions used in this work. Also, in the first one chapter, some of the tools already developed in the CAVI-ART project that we will use to achieve the stated objectives will be described. In Chapters 3 and 4, the strategies followed to obtain the black box and white box cases, respectively, will be explained in detail. In Chapter 5, the technique used to automatically test our program with the cases already generated will be developed. Lastly, we will dedicate the Chapter 6 to show results of the complete system for different programs and how our system is able to detect errors.

Capítulo 2

Preliminares

2.1. El proyecto CAVI-ART

La plataforma CAVI-ART (del inglés, *Computer Assisted Validation by Analysis, Transformation and Proofs*) [8, 9] (figura 2.1) es un proyecto orientado a la validación automática de programas. Uno de sus aspectos clave es lo que se denomina Representación Intermedia (IR por sus siglas del inglés, *Intermediate Representation*).

Partiendo de un programa escrito en cualquier lenguaje de programación como puede ser *Java*, *C*, *C++*, *Haskell*, *SML* o *Erlang*, este se transforma en una representación equivalente en la, anteriormente mencionada, IR. Esto se hará para obtener una representación abstracta del lenguaje de programación que se esté utilizando, lo cual nos permitirá trabajar de manera independiente a este. En la figura 2.2 puede verse su sintaxis abstracta. Como puede apreciarse, se trata de un lenguaje funcional mínimo, donde la expresión **letfun** permite definir un conjunto de funciones mutuamente recursivas. Además, la IR cuenta con una representación textual de dicha sintaxis abstracta a la cuál denominaremos CLIR (del inglés, *Common Lisp IR*). Cabe destacar que la IR no tiene solo código, sino también las especificaciones formales, debido a que la herramienta CAVI-ART pretende verificar formalmente los programas.

Así mismo, la plataforma CAVI-ART cuenta con numerosas herramientas que se pueden aplicar una vez que el programa se ha transformado, las cuales aportan pruebas de terminación, síntesis de invariantes, pruebas de condiciones de verificación, generación de condiciones de verificación y generación de casos de prueba, entre otras.

Será en esta última parte en la que centraremos este trabajo para generar casos de prueba que satisfagan las especificaciones formales de un programa, y así probar este con estos casos de prueba. Para ello, en el caso de la generación de caja negra los casos se generarán a partir de la precondición del programa, y en el caso de la generación de caja blanca, a partir de la precondición junto a las restricciones de un camino. Una vez generados estos casos, pasaremos a ejecutar el programa con

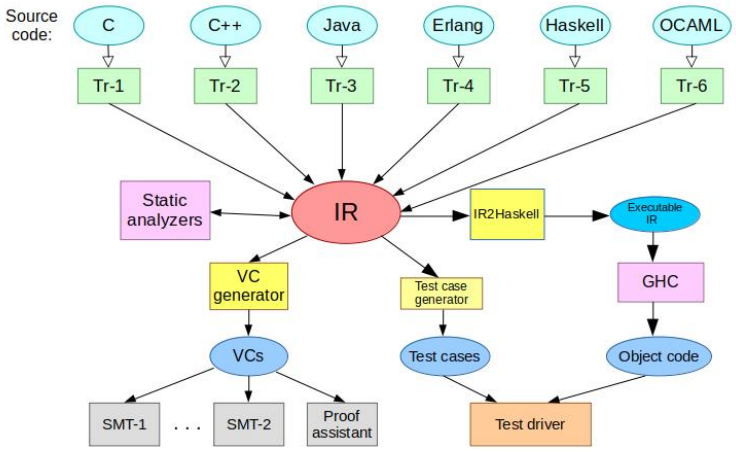

Figura 2.1: Plataforma CAVI-ART

estos, obteniendo unos resultados, que posteriormente se comprobará si son o no los esperados. Con esta herramienta, podremos detectar si el programa dado presenta errores o si, por el contrario, no se ha encontrado ninguno para los casos probados.

2.2. El resolutor Z3 y la API de Python

El resolutor Z3

Los problemas de satisfactibilidad módulo teorías (SMT por sus siglas del inglés, *Satisfiability Module Theories*) son problemas de decisión para diferentes teorías, que se expresaran como fórmulas lógicas de primer orden. Algunas de estas teorías comúnmente utilizadas en el mundo de la computación son: la aritmética lineal con números enteros, la aritmética lineal con números reales, la teoría de tipos algebraicos y la teoría de arrays (entre otras).

Un problema SMT consiste en determinar si una formula lógica de primer órden es satisfactible, es decir, si existen valores para las variables que hagan que esa fórmula lógica se cumpla. Por ejemplo, dada la fórmula lógica $a > b$ (con a y b enteros), se aplicarían las teorías de la aritmética líneal para números enteros, y si la fórmula es satisfactible (que en este caso lo es), entonces podemos obtener un modelo para las variables que haga cierta la fórmula, como podría ser $a = 1$ y $b = 0$.

Es por ello por lo que surgen los resolutores SMT, los cuales serán los encargados de decidir la satisfactibilidad de dichas fórmulas lógicas (como la antes comentada). Dentro de todos ellos, elegiremos el resolutor *Z3* [7]. Se trata de un resolutor SMT creado por Microsoft y que se encuentra entre los más avanzados y eficientes.

Podemos encontrar dos funcionalidades principales dentro de este resolutor. La primera será la mas utilizada en este trabajo, y consistirá en, dadas una serie de declaraciones de variables y/o funciones y una serie de restricciones que se aplicarán a las declaraciones anteriores, determinar su satisfactibilidad, es decir, si

```
a ->                                -- atom
    c                               -- constant
  | x                               -- variable
ae ->                               -- atomic expression
    a                               -- atom
  | f a1 ... an                     -- primitive operator/function application
  | C a1 ... an                     -- constructor application
e ->                                -- structured expression
    ae                              -- atomic expression
  | let p = ae in e                 -- sequential let
  | letfun fd1 ... fdn in e         -- function definition block
  | case a of alt1 ... altn         -- algebraic type or primitive type case
             [ _ -> e]              -- optional default clause
alt ->                              -- case alternative
    C x1 ... xr -> e                -- algebraic type alternative
    c           -> e                -- primitive type alternative
p ->                                -- pattern
    x                               -- variable pattern
  | (x1,...,xn)                     -- tuple pattern
fd -> f x1 ... xn = e               -- function definition. The name f is global
```

Figura 2.2: Sintáxis abstracta de la IR

existe alguna asignación para esas declaraciones, tal que se cumplan las restricciones que se les ha asignado. En el caso de que si lo sea, podremos obtener un modelo, que consistirá en obtener el valor que se le han asignado a estas. En numerosos casos, podemos encontrar que hay muchos modelos posibles que satisfacen el problema, pero Z3 solo nos devolverá uno de ellos. Si queremos obtener todos los posibles, la técnica que proponemos en este trabajo será ir negando todos los modelos que vaya devolviendo para así asegurar que no devuelva el mismo. Esta técnica se desarrollará en la generación de casos de caja negra que se comentará en el capítulo 3. Su segunda funcionalidad será comprobar si una formula lógica es una tautología, es decir, si es cierta para cualquier interpretación de las variables. El problema es el mencionado anteriormente, y es que solo podemos obtener un modelo para estas, por lo que habría que probar si todas las posibles interpretaciones son satisfactibles, opción que no es eficiente si la fórmula que estamos tratando es demasiado compleja. Por ello, el procedimiento que utilizaremos será negar nuestra fórmula lógica y comprobar su satisfactibilidad. Si obtenemos como resultado que la negación de esa fórmula es insatisfactible, podemos asegurar que es una tautología, ya que no existe ninguna interpretación de las variables que hagan cierta esa fórmula, o visto de otra manera, que cualquier interpretación de las variables haría cierta la fórmula original. Por el contrario, si obtenemos que es satisfactible, podemos afirmar que no es una tautología, ya que existe alguna asignación para las variables la cuál hace que la fórmula negada sea satisfactible, es decir, que esa misma asignación haría insatisfactible la fórmula original.

El lenguaje aceptado por Z3 es una implementación del estándar *SMT-LIB*[1]. Este cuenta con una pila interna que será donde se irán almacenando las restricciones que vayamos declarando y que, posteriormente, se valorará su satisfactibilidad obteniendo como posibles resultado: *satisfiable*, *unsatisfiable* o *unknown* (valor que devuelve cuando este no puede determinar si una fórmula es satisfactible

[1] http://smtlib.cs.uiowa.edu/

```
(declare-const a Bool)
(declare-const b Bool)
(assert (and a b))
(check-sat)
(get-model)
```

```
sat
(model
    (define-fun a () Bool)
        true)
    (define-fun a () Bool)
        true)
)
```

Figura 2.3: Ejemplo de lógica proposicional junto con su modelo

o no). Las declaraciones de este lenguaje que más utilizaremos serán:

- `declare-const`: declaración de una variable o constante junto a su tipo.
- `declare-fun`: declaración de una función.
- `declare-fun-rec`: declaración de una función recursiva.
- `assert`: añadir una restricción a la pila interna de Z3.
- `check-sat`: obtener la satisfactibilidad de las restricciones almacenadas en la pila.
- `get-model`: devuelve el modelo en el caso de que la comprobación anterior devuelva *sat*.

Así mismo, como describíamos anteriormente, Z3 permite trabajar con una serie de teorías tales como:

- **Teoría de la lógica proposicional**: el tipo predefinido *Bool* es el tipo de todas las expresiones proposicionales. Las variables de este tipo se declaran mediante los comandos `declare-const()` o `declare-fun()` si estas se tratan de funciones. Las fórmulas pueden contener los operadores booleanos más comunes como *and*, *or*, *xor*, *not*, $\Rightarrow$ (implicación), *ite* (if-then-else), etc. En cuanto a la interpretación, consistirá en asignar a las constantes los valores *true* o *false*. En la figura 2.3 se muestra un ejemplo de programa que utiliza esta teoría junto con el resultado obtenido.
- **Teoría de la aritmética lineal con enteros y reales**: las variables de estos tipos se declaran, al igual que los anteriores, mediante los comandos `declare-const()` o `declare-fun()`. Las fórmulas pueden contener los operadores aritméticos y relacionales tales como: $+$, $-$, $*$, $<$, $\leq$, $>$, $\geq$ (entre otros) y constantes. La interpretación, consistirá en asignar a cada variable un valor entero o real, según el tipo. En la figura 2.4 se muestra un ejemplo en el que se aplica esta teoría para los números enteros junto con el modelo que se obtiene para este.

```
(declare-const a Int)
(declare-const b Int)
(assert (> a 0))
(assert (< b 5))
(assert (> a b))
(check-sat)
(get-model)
```

```
sat
 (model
  (define-fun b () Int)
    0,
  (define-fun a () Int)
    1
 )
```

Figura 2.4: Ejemplo de números enteros junto a su modelo

```
(declare-const a (Array Int Int))
(assert (= (select a 1) 3))
(assert (= (select a 2) 4))
(check-sat)
(get-model)
```

```
sat
 (model
 (define-fun a () (Array Int Int)
   (store ((as const
      (Array Int Int)) 4) 1 3))
 )
```

Figura 2.5: Ejemplo de array junto a su modelo

- **Teoría de Arrays**: las variables de este tipo se declaran, mediante el comando `declare-const()`. La representación interna de estas variables será en forma de una función no interpretada que tendrá en cuenta los índices y los valores. Cabe destacar dos funciones principales para estos:

  - `store a i v`: devuelve un array idéntico a a, pero en la posición i se encuentra el valor v
  - `select a i`: devuelve el valor almacenado en la posición i del array a.

  Z3 usa la constructora `(_as-array f)` para dar la interpretación de estos. De manera que si el modelo del array a es igual a `(_as-array f)`, entonces para todo índice i, la función `(select a i)` es igual que aplicar la función `(f i)`. En el ejemplo de la figura 2.5, Z3 crea una función auxiliar a para asignar una interpretación del array a. De manera que el valor de todas las posiciones es 4, menos en la posición 1 que encontramos un 3 introducido mediante la función *store*.

- **Teoría de tipos algebraicos**: se declaran con el comando `declare-datatypes()`. Pueden ser utilizados para especificar listas finitas, árboles y otras estructuras recursivas que usaremos en este trabajo. Además, dentro de estos tipos, encontramos las tuplas binarias como un caso específico de estos, definiéndose con la palabra reservada *Pair*, seguida de los tipos de los valores de la tupla.

- **Teoría de cuantificadores**: Z3 acepta y puede trabajar con fórmulas que usen cuantificadores, tales como $\forall$ o $\exists$. Es llegado a este punto cuando el resultado de ejecutar el comando *check-sat* pueda ser *unknown*. Por ejemplo, si tenemos un array a y decimos que para todas las posiciones de este, su valor es distinto de 2, según la representación de estos que ya hemos comentado, al tratarse de una función no interpretada, tendría infinitas posiciones y el

resolutor al comprobar su satisfactibilidad, fallaría, obteniendo como resultado *unknown*.

La API Z3Py

Por otro lado, Z3 distribuye diferentes *APIs*[2] para diferentes lenguajes como por ejemplo: *C*, *.Net* y *Python* entre otras. Este trabajo lo centraremos en Z3Py [3], que se corresponde con la API de Z3 para *Python*.

Su funcionamiento, al tratarse de una API, es el mismo que el de Z3, pero en este caso, la sintaxis varía con respecto a la de este, ya que se aprecia mayor parecido con la de Python. Algunas de las funciones que más usaremos en este trabajo serán:

- `solver()`: crear un nuevo resolutor, que será en el que se almacenarán todas las restricciones que vayamos declarando como si fuera una pila.
- `push()`: establece un punto de guardado de la pila. Con esto, podremos volver al estado del resolutor que hemos guardado cuando ejecutemos la función *pop*.
- `pop()`: elimina restricciones de la pila desde la cima hasta el último punto de guardado (realizado con la función `push`).
- `check()`: se corresponde con el `check-sat` de Z3. Resuelve las restricciones insertadas en la pila. Este devolverá como resultado *sat* si se ha encontrado una solución, *unsat* si no existe ninguna solución o *unknown* si el resolutor falla al resolver el sistema de restricciones.
- `model()`: se corresponde con el *get-model* de Z3. Devuelve una interpretación que cumple todas las restricción insertadas.
- `decls()`: permite acceder a la lista de todas las variables o funciones del modelo que se les ha asignado un valor, es decir, aquellas a las que se les ha aplicado una o mas restricciones.
- `from_string()`: analiza sintácticamente las declaraciones y restricciones de Z3 dadas en una cadena de texto en formato SMT-LIB y permite cargar estas dentro del resolutor.
- `add()`: añade una restricción en la cima de la pila del resolutor.
- `set()`: establece el temporizador del resolutor. El tiempo que se establezca es el tiempo que el resolutor puede tardar como máximos en resolver un sistema de restricciones. Cuando este expira, si no ha resuelto dicho sistema, devolverá *unknown*. Por ello es importante ajustar el temporizador a un tiempo no muy

[2]Del inglés, *Application Programmer Interface*, es el conjunto de funciones y procedimientos que cumplen una o muchas funciones con el fin de ser utilizadas por otro software

[3]https://ericpony.github.io/z3py-tutorial/guide-examples.htm

```
a = Bool('a')
b = Bool('b')
s = Solver()
s.add(And(a, b))
check = s.check()
if(str(check) == "sat"):
    m = s.model()
    print m
```

Figura 2.6: Ejemplo de lógica proposicional en Z3Py

```
a = Int('a')
b = Int('b')
s = Solver()
s.add(a > 0)
s.add(b < 5)
s.add(a > b)
check = s.check()
if(str(check) == "sat"):
    m = s.model()
    print m
```

Figura 2.7: Ejemplo de números enteros en Z3Py

alto, para evitar largas esperas, pero tampoco a uno bajo, ya que este podría expirar en la mayoría de las veces.

Así mismo, la sintaxis de las teorías también cambia. Esta vendrá definida de la siguiente manera:

- **Teoría de la lógica proposicional**: Las variables de este tipo se declaran mediante la asignación `p = Bool('x')`, siendo x el nombre que se le asignará dentro del resolutor a la variable p. En la figura 2.6 se desarrolla el mismo ejemplo que el de la figura 2.3 pero con la sintaxis de Z3Py, obteniendo como resultado:

$$[a = True, b = True]$$

- **Teoría de aritmética lineal para enteros y reales**: Las variables de estos tipos se declaran mediante la asignación `x = Int('x')` para los enteros o bien `y = Real('y')` para los reales. En la figura 2.7 se presenta el mismo ejemplo que el de la figura 2.4 pero con la sintaxis de Z3Py. El resultado obtenido será:

$$[b = 0, a = 1]$$

- **Teoría de Arrays**: La forma en la que estas variables se declaran será: `p = Array('A', IntSort(), IntSort())` en el que A será el nombre que se le asociará a la variable dentro del resolutor y lo siguiente denotará un array de entero a entero. Encontramos las funciones *select* y *store*, pero en este caso, la función `Select(a, i)` se puede simplificar con `a[i]`.

```
a = Array('a', IntSort(), IntSort())
s = Solver()
s.add(a[1] == 3)
s.add(a[2] == 4)
check = s.check()
if(str(check) == "sat"):
    m = s.model()
    print m
```

Figura 2.8: Ejemplo de arrays en Z3Py

```
List = Datatype('List')
List.declare('cons', ('car', IntSort()), ('cdr', List))
List.declare('nil')
List = List.create()
```

Figura 2.9: Ejemplo de tipo de datos en Z3Py

Z3Py en este caso, usa la constructora *K(s, v)* para establecer los arrays constantes, la cual devuelve un array en el que todas las posiciones del array v toman el valor s. Sin embargo, si no se cumple lo anterior como pasa en el ejemplo de la figura 2.8, obtendremos como resultado:

$$[a = Store(K(Int, 4), 1, 3)]$$

en el cual tendremos un array constante en el que todas sus posiciones son 4, menos en la posición 1 que se le introduce un 3 mediante la función *store*.

- **Teoría de tipos algebraicos**: La forma en la que se declararán será: `p = Datatype('x')` siendo x el nombre que tomará el tipo internamente en el resolutor. Para declarar las constructoras, se usará la función `declare()`, como se muestra en el ejemplo de la figura 2.9 en el cual se crea un tipo de datos *Lista* que esta formado por dos constructoras. La primera denominada *cons* tendrá un primer parámetro de tipo entero y el segundo parámetro será de tipo *Lista*. La segunda constructora denominada *nil* será una constructora vacía. Finalmente, una vez que tenemos declaradas las constructoras, usaremos la función `create()` para crear el tipo de datos definido.

- **Teoría de cuantificadores**: Al igual que Z3, los cuantificadores que acepta Z3Py serán (entre otros) $\forall$ y $\exists$. Estos pueden ser utilizados en problemas aritméticos, booleanos, de arrays o incluso estructuras de datos complejas. Al igual que en Z3, al usar estos cuantificadores en las fórmulas podemos encontrar que al comprobar la satisfactibilidad de un sistema de restricciones nos devuelva *unknown* porque el resolutor falle.

Como se describía al comienzo de este apartado, con la función `model()` obtenemos el modelo para todas las variables y funciones a las que se le aplica alguna de las restricciones insertadas. Por ello, para obtener el valor de cada variable en el modelo se realizará el método que se muestra en la figura 2.10. En primer lugar,

```
check = s.check()
if(str(check) == "sat"):
    m = s.model()
    for var in m.decls():
        print (var.name(), m[var])
```

Figura 2.10: Obtención del modelo de cada variable

comprobaremos la satisfactibilidad de nuestras restricciones, y en caso de devolver *sat* pediremos un modelo que se almacenará en la variable *m*. Acto seguido nos recorreremos cada variable del modelo, y mediante la expresión $m[var]$ obtendremos el valor asociado a la variable *var* dentro del modelo *m*.

En cuanto a la compatibilidad de la sintaxis de Z3 y Z3Py, mediante la función `from_file()` y `from_string()` se puede introducir al resolutor declarado en Z3Py todas las definiciones de funciones, declaración de variables y restricciones en el formato textual SMT-LIB. Pero, ¿cuál es la diferencia entre estas?. La diferencia es que la primera carga directamente en el resolutor el contenido del archivo (el cual su nombre se le pasa por parámetro a la función). Por otro lado, la segunda función, carga el contenido de una variable (cuyo contenido será un string) dentro del resolutor. Cabe destacar que la declaración de tipos, funciones y variables *no* se almacenan en el resolutor, únicamente las restricciones, pero para que se puedan introducir estas, cada una de ella tiene que ser introducida con sus declaraciones apropiadas.

2.3. La herramienta IR2Haskell

Dentro de todas las herramientas que pertenecen a la plataforma CAVI-ART, una de las que mas utilizaremos en este trabajo será la herramienta IR2Haskell, desarrollada en [6].

Una de las funcionalidades de esta herramienta es que a partir de la CLIR, producirá un árbol de sintaxis abstracta para el programa (AST por sus siglas del inglés, *Abstract Syntax Tree*) que representa la estructura de este y sus predicados (tanto la precondición como la postcondición).

La otra parte de la herramienta, transformará la CLIR en funciones Haskell ejecutables. La razón de utilizarla es que la CLIR no es ejecutable, mientras que Haskell si. Estas funciones se corresponden con las del programa que estamos evaluando. Estas se ejecutarán junto a diferentes casos de prueba, como pueden ser los generados de tipo caja negra o de tipo caja blanca. Posteriormente a su ejecución, comprobaremos si el resultado es el esperado o por el contrario, estas funciones presentan un error.

2.4. La herramienta Precd2Z3

Otra herramienta que utilizaremos será Precd2Z3, desarrollada en [3]. Su objetivo principal será transformar la precondición, que vendrá dada en el formato de la CLIR, a un conjunto de restricciones de Z3, de manera que si estas resultaran satisfactibles, proporcionemos a partir de la API de Z3 para Python modelos que servirán como casos de caja negra. Además, nos proporcionará todos los tipos y predicados auxiliares (que puedan aparecer en las restricciones) en el formato textual SMT-LIB. Esta traducción de la precondición en asertos de Z3 se hará a través de un programa Haskell que producirá como resultado un fichero *.smt* con todas estas restricciones generadas. El lenguaje de especificación utiliza numerosas funciones recursivas que se definen en Z3 y que en este trabajo modificaremos y añadiremos algunas adicionales como se explica en posteriores capítulos.

2.5. La herramienta AST2Z3

Por último, otra herramienta que utilizaremos será AST2Z3, desarrollada en [4]. Ya que en ocasiones las pruebas de caja negra no cubren todos los posibles casos que se pueden dar en un programa, aparecen las pruebas de caja blanca. Estas tienen en cuenta el flujo del programa para detectar los posibles caminos de este, además de su precondición. Para ello, buscará una estructura similar a un grafo que permitirá representar el flujo del programa (dado en la CLIR) sobre la que se definirán los caminos. Con ello, esta herramienta se encarga de generar restricciones para representar los caminos hasta un nivel de profundidad (número de iteraciones de los bucles o de llamas recursivas) dado. Es decir, transformará estos caminos en asertos de Z3 que en este trabajo analizaremos junto a la precondición generada por la herramienta Precd2Z3 para generar estos casos de prueba de caja blanca.

Estas dos últimas herramientas fueron un primer prototipo para generar casos de prueba basados en asertos. Su trabajo termina generando un fichero *.smt* en formato SMT-LIB. El resto del proceso es manual interaccionando con Z3.

Capítulo 3

Generación de casos de caja negra

Como ya hemos mencionado anteriormente, uno de los objetivos principales de este trabajo es generar casos de prueba de caja negra, los cuales son casos que se ajustan a la precondición del programa que se desea probar. Por ello, en primer lugar transformaremos estas precondiciones en asertos de Z3 y en el primer apartado comentaremos la estructura del fichero *.smt* que se genera.

Una vez que tenemos lo anterior, proponemos la funcionalidad auxiliar comentada en el segundo apartado, que consistirá en acotar los valores que se les puede asignar a las llamadas *variables de interés*. Estas variables serán las que se le pasará como argumentos al programa y en las que se devolverá el resultado obtenido. En esta primera parte solo nos interesaremos de las variables de entrada que serán las que deben cumplir la precondición. Por consiguiente, comentaremos la estrategia seguida para la generación exhaustiva de estos casos de prueba. Finalmente, aplicaremos una serie de transformaciones a los resultados obtenidos para convertirlos en términos Haskell.

3.1. Conversión de la precondición en asertos Z3

Como ya se ha introducido en 2.4, la herramienta Precd2Z3 genera un fichero *.smt* que contiene la precondición en forma de aserto de Z3. Usaremos *precd.smt* para referirnos a este. La estructura de la que constaba este fichero en un primer momento era:

1. *Declaración de los tipos de datos y funciones*:
    1.1. *Tipos de datos*: se declaran al comienzo del fichero y estos son: *array*, *LLRB*, *lista*, *BST*, *montículo*, *AVL*, *set* y *multiset*.
    1.2. *Funciones*: en esta sección se declaran, en formato SMT-LIB, todas las funciones auxiliares complementarias para los tipos de datos anteriormente mencionados. Un ejemplo de función será `sortedArr` para determinar si un determinado segmento del array esta ordenado ascendentemente.

```
(define-fun-rec
      sortedArr ((a (Arr Int)) (b1 Int) (b2 Int)) Bool
  (
    ite(>= b1 b2)
        true
        (and (<= (select (first a) b1)
             (select (first a) (+ b1 1)))
             (sortedArr a (+ b1 1) b2))
  )
)
```

Figura 3.1: Definición de la función *sortedArr*

La definición de esta la podemos encontrar en la figura 3.1. La estructura que presentará nuestro tipo de datos array (*Arr*) será un par `(Array a, Int)`, donde el primer valor se corresponde con un array de Z3 y el segundo valor hace referencia al tamaño de este.

2. *Declaración de las variables de interés*: en este punto es cuando se declaran las variables de entrada del programa.

3. *Declaración de la precondición*: es aquí donde se halla traducida la precondición dada en el lenguaje de la IR en forma de aserto de Z3.

4. *Check-sat*: para comprobar la satisfactibilidad de las restricciones anteriormente declaradas.

5. *Get-model*: en caso de ser factible, proporcionar un modelo que cumpla esas restricciones.

Para este trabajo se ha decidido modificar la estructura del fichero anterior. La causa de esto se comentará en detalle en el capítulo 4. Actualmente, contaremos con un fichero auxiliar y el fichero *precd.smt*. El primero se denominará *funciones.smt* y la estructura que le sigue será:

1. *Declaración de los tipos de datos y funciones*:

   1.1. *Tipos de datos*: se mantendrán los mismos a excepción de *set* debido a que Z3 ya cuenta con una definición análoga. Además el nombre de las constructoras se modifica, lo cuál se explicará más en detalle en la sección 3.4.

   1.2. *Funciones*: además de mantener todas las anteriores, se introducen funciones auxiliares nuevas cuya funcionalidad se explica en 3.2.

Y por otro lado, la estructura del fichero *precd.smt* será la que sigue:

1. *Nombre del programa*: se corresponde con el programa que se está evaluando.

```
;ALTURA AVL
(define-fun-rec
    altA ((t (AVL Int))) Int
    (
        ite(= t LeafA)
        0
        (+ 1 (max (altA (izq t)) (altA (der t))))
    )
)
```

Figura 3.2: Definición de la función altA

2. *Nombre de las variables de interés*: vendrán declaradas en diferentes líneas de la forma `;x` siendo x el nombre de una de las variables. Esta sección terminará cuando se encuentre la línea ********. Los ";" son ignorados por Z3 pero esas declaraciones serán de gran utilidad para nuestro programa Python.

3. *Declaración de las variables de interés.*

4. *Restricciones para limitar el valor de las estructuras y variables.*

5. *Declaración de la precondición.*

3.2. Limitando el tamaño y el contenido de las estructuras de datos

Para la generación de casos de caja negra se tendrán en cuenta una serie de restricciones adicionales. El objetivo de estas es acotar el número de modelos que se pueden obtener para un conjunto de restricciones. Lo que ocurriría si no insertáramos restricciones de este tipo, es que habría infinitos casos de caja negra dadas unas restricciones. Para evitar esto, las que proponemos en este trabajo serán:

- *Restricciones de tamaño en estructuras*: afectan (como su propio nombre indica) al tamaño de las estructuras de datos que pueden aparecer. Para crear estas restricciones podemos usar algunas funciones auxiliares como *altA* (figura 3.2), *long* y *heigthL*. Con la primera función podemos obtener la altura de un *AVL* dado por parámetro, de esta forma se podría limitar la altura de un *AVL* t con altura 4 con la restricción `(assert(= (altA t) 4))`. Con la segunda función, podemos obtener la longitud de una lista que se pasa por parámetro, y de la misma manera que antes, podríamos limitar el tamaño de una lista l con tamaño 4 con la restricción `(assert(= (long l) 4))`. Con la última función podríamos obtener la altura de un *LLRB* (una variante del árbol roji-negro) y mediante el procedimiento anterior, limitar el tamaño de dicha estructura.

- *Restricciones para los valores de los elementos de las estructuras de datos*: con las anteriores restricciones podríamos seguir teniendo infinitos casos, ya que

```
(declare-const a (Arr Int))
(assert (limitArray a 1 3))
(assert (sortedArr a 0 2))

a = mk-pair(Store(Store(K(Int, 1), 2, 3), 1, 2), 3)
```

Figura 3.3: Ejemplo de la función limitArray

```
(define-fun-rec
        limit ((a (Arr Int)) (minArr Int) (maxArr Int) (i Int)) Bool
        (
        ite(= i (second a))
                true
        (and (<= minArr (select(first a) i))
             (<= (select(first a) i) maxArr)
                        (limit a minArr maxArr (+ i 1)))
    )
)

(define-fun-rec
        limitArray ((a (Arr Int)) (minArr Int) (maxArr Int)) Bool
        (
              limit a minArr maxArr 0
        )
)
```

Figura 3.4: Definición de la función *limitArray*

si por ejemplo tenemos un árbol t que cumple la restricción en estructura: *isAVl*, y además cumple la restricción de tamaño en estructuras: `(assert(= (altA t) 2))` para indicar que ese AVL tiene que tener altura 2, seguiríamos teniendo infinitas representaciones de estos, ya que los valores internos de la estructura (valores de los nodos y las hojas) al no estar delimitados podrían tomar cualquier número entero. Es por ello que proponemos estas restricciones para delimitar estos valores entre un rango. Para ello, las funciones que se proponen serán:

- `limitArray a min max`: será la encargada de establecer el rango que pueden tomar los valores pertenecientes al array *a*. Este rango vendrá definido por los parámetro *min* y *max*. En el ejemplo de la figura 3.3, se puede ver que tenemos un array *a* declarado, al que se le aplica una de las restricciones anteriores (*sortedArr*) para establecer una propiedad a los valores del array y la nueva función que estamos describiendo, para limitar estos valores entre 1 y 3. El resultado obtenido en efecto es un array que está ordenado y que sus valores se encuentran dentro del rango establecido. En la figura 3.4 encontramos la definición de esta.
- `limitAVL t min max`: En este caso, la función se encargará de establecer el rango que pueden tomar las hojas y los nodos del AVL *t*. Al igual que antes, dicho rango vendrá definido por los parámetros min y max. En el ejemplo de la figura 3.5 encontramos un AVL t definido, al que se le aplica la función *isAVL* (antes mencionada) y la función que estamos definiendo para limitar los valores entre 1 y 3. Como resultado obtenemos que t cumple efectivamente la propiedad de ser AVL y que los valores de este

```
(declare-const t (AVL Int))
(assert (limitAVL t 1 3))
(assert (isAVL t))

t = NodeA(2, 2, LeafA, NodeA(3, 1, LeafA, LeafA))
```

Figura 3.5: Ejemplo de la función limitAVL

```
(define-fun-rec
    limitAVL ((t (AVL Int)) (minAVL Int) (maxAVL Int)) Bool
    (
        ite(= t LeafA)
            true
            (and (<= minAVL (val t)) (>= maxAVL (val t))
            (limitAVL (izq t) minAVL maxAVL) (limitAVL (der t) minAVL maxAVL))
    )
)
```

Figura 3.6: Definición de la función *limitAVL*

están delimitados por el rango establecido. En la figura 3.6 encontramos la definición de esta función.

- `limitTree t min max`: Utilizada para limitar el rango que pueden tomar las hojas y nodos de t, el cuál podrá ser o bien un BST o bien un montículo. Seguirá el mismo proceso para ello que los anteriores.
- `limitLLRB t min max`: Encargada de limitar esta vez el rango de los valores que pueden tomar los nodos y hojas de t, que en este caso será un $LLRB$.
- `limitLST l min max`: Siendo l una lista, esta función limitará el valor de los elementos que esta puede contener. Al igual que todas las anteriores, el rango de estos vendrá determinado por los parámetros *min* y *max*. En la figura 3.7 se puede ver un ejemplo en el cúal declaramos una lista l a la cuál se le aplica la función que estamos definiendo y, efectivamente, el resultado es el esperado, ya que todos los elementos de la lista, se encuentran entre el rango 1 y 3. En la figura 3.8 encontramos la definición de esta.

Con estas restricciones ya definidas, el usuario podrá generar casos de caja negra más precisos y podremos asegurar que no existirán infinitos casos para un conjunto de restricciones dado.

3.3. Generación exhaustiva de modelos

Una vez que tenemos definidos cuáles ván a ser los ficheros que usaremos para este apartado y cuál será la estructura de estos, podemos comenzar a describir la estrategia que hemos elegido para generar casos de prueba de caja negra.

```
(declare-const l (Lst Int))
(assert (limitLST l 0 3))

l = Cons(1, (Cons (2, (Cons(3, (Cons(2, Nil)))))))
```

Figura 3.7: Ejemplo de la función limitLST

```
(define-fun-rec
      limitLST ((l (Lst Int)) (minLST Int) (maxLST Int)) Bool
      (
            ite(= l Nil)
                true
                (and (<= minLST (hd l)) (>= maxLST (hd l))
                (limitLST (tl l) minLST maxLST))
      )
)
```

Figura 3.8: Definición de la función *limitLST*

En primer lugar, vamos a describir cómo leeremos desde Z3Py las declaraciones, funciones auxiliares y asertos que se encuentran en los ficheros *.smt* mencionados en la sección 3.1. Como se exponía el problema en la sección 2.2, el resolutor no almacena los tipos de datos ni las declaraciones de variables. Es por ello, que cada vez que se introduce una restricción necesita ir acompañada de la declaración de variables que esta usa, y por lo tanto, si alguna de estas es un tipo de datos, a su vez necesita que se introduzca la definición de este junto con la declaración. Por ejemplo, si tenemos el aserto `(assert(isAVL t))` para definir que t cumple la propiedad de ser un AVL, entonces esa variable ha de ir introducida en el resolutor junto a su declaración, es decir, `(declare-const t (AVL Int))`. De esta forma, al decir que t es un AVL, también necesita ser introducido junto a su definición, que sería `(declare-datatypes (T) ((AVL leafA (nodeA (val T) (alt Int) (izq AVL) (der AVL)))))`. Para cargar el contenido de un fichero *.smt* teníamos dos posibilidades como comentábamos en 2.2. Teniendo en cuenta estas posibilidades y la característica anterior, parece evidente que método tenemos que utilizar.

Si utilizáramos el método `from_file()`, el cuál recibe como argumento el fichero *.smt* que queremos cargar en el resolutor, al cargar el archivo *funciones.smt* no habría ningún problema, pero al cargar el fichero *precd.smt*, por la primera característica comentada, esto daría error ya que no tendría las declaraciones de las funciones y de los tipos de datos. Por ello, el método que se usará será `from_string()`. La estrategia que usaremos para ello, será abrir y volcar en dos variables distintas el contenido de los dos ficheros *.smt*. Una vez que los tenemos volcados en las variables, supongamos *fun* y *prog*, el siguiente paso sería invocar el método `from_string()` el cuál recibirá por parámetro la concatenación de las dos variables anteriores, es decir, `fun + prog`. Una vez realizamos esto, tendríamos en el resolutor cargadas todas las restricciones necesarias para generar los casos de caja negra.

La siguiente fase consistirá en conocer cuáles son las variables de interés, como definíamos en el capítulo 3. Es decir, serán las variables de entrada del programa. Esto lo necesitaremos ya que como utilizaremos funciones auxiliares (como

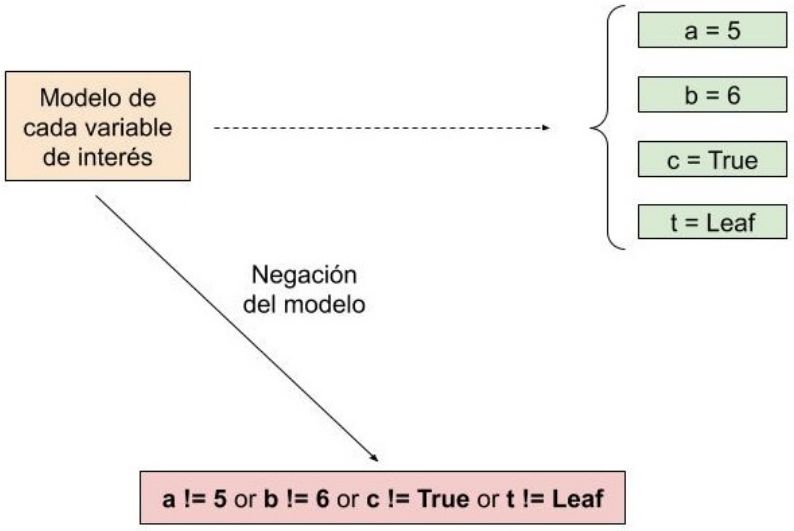

Figura 3.9: Ejemplo de negación del modelo

isAVL), cuando pedimos un modelo, en este aparecen todas las funciones utilizadas durante la resolución de las restricciones, además de posibles variables temporales (como se comentará mas adelante). Es por ello que al comienzo del archivo *precd.smt*, aparece el nombre de estas variables de interés (como se explica en la sección 3.1) en orden según la función espera esos parámetros de entrada. Estos nombres se leerán línea a línea hasta que se encuentra la línea ******, en ese momento, terminamos esta fase y esos nombres quedan almacenados en un array.

Una vez que tenemos cuáles son las variables de interés y tenemos todas las restricciones almacenadas en el resolutor, podemos empezar a generar casos de caja negra. La estrategia que se propone es obtener y guardar el modelo obtenido (se explicará en la siguiente sección) e ir negándolo hasta que, o bien se hayan generado todos los posibles y el resolutor devuelva *unsat*, o bien llegue un punto en el que el resolutor falle, lo que devolverá *unknown*. Para esta técnica, únicamente nos interesará guardar y negar el modelo de las variables de interés, que son las que el programa necesitará. Gracias a la fase anterior, podemos conocer cuáles son estas.

La negación de un modelo consistirá en la disyunción de la negación del modelo de cada variable de interés, es decir, como se expresa en la figura 3.9. Una vez que hemos terminado de negar el modelo de todas, introduciremos en el resolutor la disyunción de esas negaciones con la función `add(Or(block))`, suponiendo que en *block* tenemos todas estas negaciones.

Para cada modelo que nos genere el resolutor, supondremos que este se almacenará en la variable *m*, y es en esta donde encontramos todas las variables y funciones utilizadas junto con su respectivo modelo. Como se menciona en la sección 2.2, con la función `decls()` podemos obtener el nombre de todas estas variables y funciones del modelo, y por lo tanto mediante la estrategia que se propone en la figura

```
(declare-const t (AVL Int))
(declare-const x Int)
(assert (> x 0))
(assert (limitAVL t 1 5))
(assert (isAVL t))

t = NodeA(2, 2, LeafA, NodeA(4, 1, LeafA, LeafA))
x = 1
```

Figura 3.10: Ejemplo de modelo para un AVL y un entero

2.10 podremos obtener el modelo de estas. Si *var* es una variable de interés, entonces procederemos a almacenar su modelo y acto seguido negarlo como se mencionaba anteriormente. El problema surge con el tipo de *var* ya que la técnica para negar el modelo será distinta si esta es un array o si es cualquier otra estructura de datos.

En primer lugar, describiremos la técnica usada para todos los tipos base y estructuras de datos menos los arrays. Como se describía en la sección 2.2, mediante la función `m[var]` podemos obtener el valor asociado a la variable *var* en el modelo *m*. Por lo tanto, para negar un modelo de una variable de este estilo sería tan sencillo como añadir a Z3 la restricción `var() != m[var]`. Con esto, si tenemos algo como en el ejemplo de la figura 3.10 en el que t es de tipo AVL y se le aplican las restricciones *isAVL* y *limitAVL*; x es un entero al que se le aplica la restricción `x` $>$ `0`, un modelo que se obtiene será `t = NodeA(2, 2, LeafA, NodeA(4, 1, LeafA, LeafA))` y `x = 1` por lo tanto, la negación del modelo que añadiríamos al array *block* sería `t != NodeA(2, 2, LeafA, NodeA(4, 1, LeafA, LeafA))` y `x != 1`. De tal manera, que cuando añadamos ese bloque al resolutor mediante la estrategía comentada anteriormente la restricción que añadiremos a este será `t != NodeA(2, 2, LeafA, NodeA(4, 1, LeafA, LeafA)) or x != 1`.

Para los casos anteriores, se puede utilizar dicha técnica ya que los tipos base y tipos de datos que utilizamos son estructuras finitas y bien definidas. El problema, es que los arrays de Z3 no cuentan con esa característica. Como se especificaba en la sección 2.2, su representación interna será como función no interpretada la cuál tendrá un rango infinito de los índices y valores. Por ello, si obtenemos un modelo para un array como el que se muestra en la figura 3.3, lo que nos dice es que en la posición 2 hay almacenado un 3, en la posición 1 un 2 y en cualquier otra posición un 1. Es decir, si preguntamos por el valor que hay en la posición -245 nos dirá que es el mismo que hay almacenado en la posición 354 por ejemplo, ya que al tratarse de una función, no existe la definición de *tamaño del array* ni tampoco cuenta con la característica de que sus índices han de ser positivos. Es por esto que si negamos este modelo de la primera forma comentada, al darse las características anteriores, si intentamos obtener un nuevo modelo con la restricción anterior, el resolutor no es capaz de generar un modelo (ya que como la función puede tomar infinitos índices, no puede asegurar que cada uno de ellos, tenga un valor distinto al anterior) devolviendo *unknown*.

Es por ello, que se requiere otra técnica más sofisticada. Para ello tenemos que tener en cuenta cuál es la estructura de los arrays, la cuál vendrá representada

```
fst = var().sort().accessor(0,0)
Select(fst(var()), i)
```

Figura 3.11: Método para referenciar a la posición i-ésima de un array

en la figura 3.12. Nuestro tipo de datos array, como ya se ha comentado, será un par formado por un array (de Z3) y un entero que hace referencia a su tamaño. La estrategia que proponemos consistirá en negar los valores que se encuentren entre las posiciones desde la 0 hasta la del tamaño de nuestra estructura (sin incluir). Para esto, como se ve en la figura 3.12, si queremos acceder a la primera componente del par (que en este caso será el array de Z3) utilizaremos *first*, de tal manera que *first(a)* nos devolverá una referencia al array de Z3 en ese modelo. Por otro lado si utilizamos *second* accederemos al segundo valor del par, es decir, con *second(a)* obtendríamos una referencia al tamaño de la estructura en ese modelo.

Pero en Z3Py, no es tan sencillo como poner lo anterior. Dentro de la variable *var*, tendremos que buscar estos accesores. Con la función `sort()` accedemos a la constructora de *var*, y mediante la función `accesor()` podemos encontrar lo que estábamos buscando. De manera, que con `var().sort().accessor(0,0)` obtendríamos el accesor *first* y de igual manera, con `var().sort().accessor(0,1)` el accesor *second*.

Una vez que ya podemos acceder al array del modelo con los accesores, veremos cómo acceder a las n posiciones de este y cómo obtener el valor asociado a estas. Como comentábamos anteriormente, gracias a la función `select(a,i)` podemos referirnos a la posición i-ésima del array a, de esta manera, como se muestra en la figura 3.11, haremos referencia a la componente `a[i]`.

Ahora necesitamos el valor asociado a la posición i-ésima antes mencionado. Para ello, describiremos a la función `m.eval(t)`, la cuál dada una expresión t es capaz de evaluarla en el modelo m actual. Esa expresión t será `select(a,i)`, con a el modelo devuelto para el array de Z3. Para acceder a este lo haremos de la siguiente manera: `m[var].arg(0)` y de igual manera, para acceder al tamaño utilizaríamos `m[var].arg(1)`. Es por ello, que combinando todo lo anterior de la siguiente manera:

```
m.eval(Select(modelo[var].arg(0), i))
```

obtendríamos esta funcionalidad.

Con esta información ya podríamos negar el modelo de un array, ya que tenemos la forma de acceder a la variable y a su valor. Por lo que, en vez de la negación del modelo de la figura 3.3 ser `a != mk-pair(Store(Store(K(Int, 1), 2, 3), 1, 2), 3)` será `a[0] != 1 or a[1] != 2 or a[2] != 3 or second(a) != 3`.

En este punto, queda definida cuál será la estrategia que se seguirá para negar modelos y obtener todos los posibles para un determinado programa partiendo de una serie de restricciones iniciales.

```
(declare-datatypes (T1 T2) ((Pair (mk-pair (first T1) (second T2)))))
(define-sort Arr (T) (Pair (Array Int T) Int))
```

Figura 3.12: Declaración del tipo de datos array en Z3

3.4. Convirtiendo modelos Z3 en términos Haskell

Una vez hemos descrito los puntos anteriores, falta saber cómo se guardan esos modelos, y qué serie de transformaciones se les aplica.

El objetivo de transformar estos modelos Z3 en términos Haskell será para probar el programa. Como se describe en la sección 2.3, este será un programa Haskell y se ejecutará con estos casos que hemos generado. Es por ello, que necesitamos que estos casos presenten el modelo que este programa espera, es decir, en formato Haskell.

Estos modelos se almacenarán en un fichero *.txt* y cada uno de estos, irá en una línea del fichero. La forma en la que se guardará cada modelo será en una tupla *n-aria* de tantas componentes como variables de interés tenga el programa. Es decir, si tenemos 3 variables de interés: x, t y l, lo que encontraremos en cada línea del fichero será algo del estilo `(mod(x), mod(t), mod(l))`, siendo por ejemplo *mod(x)* el modelo para la variable x. Estos modelos deben de cumplir unas ciertas condiciones:

Los modelos de las variables de la tupla *n-aria* deben preservar el orden de los parámetros del programa, es decir, según el ejemplo anterior, el primer parámetro sería x, seguido de t y por último l. Esto lo conseguiremos gracias a que al leer las variables de interés del fichero *precd.smt*, estas vienen en ese orden esperado, y por lo tanto, almacenaremos el nombre de estas en ese orden. En cuanto a los valores de las variables de interés, estos se almacenen en un diccionario (para mejorar la eficiencia a la hora de acceder a estos valores), de manera que cada variable tiene asociado en este su valor. Por ello, una vez que se han inspeccionado todas las variables y se ha almacenado en el diccionario sus respectivos valores, se recorre la estructura donde tenemos almacenados los nombre de estas en orden y se almacena su resultado en el fichero.

La otra condición que se ha de cumplir, es que estos modelos presenten la sintaxis que Haskell espera. Para ello, en primer lugar debemos mencionar las clases *Read* y *Show*. La clase *Show* lo que nos permite es obtener cómo es la representación para una instancia de un tipo de datos definido en Haskell en una cadena de *string*. Por otro lado, la clase *Read* hace todo los contrario, es decir, nos proporciona operaciones para analizar cadenas de *strings* y obtener los valores que estos representan.

Gracias a estas clases, podemos detectar cómo es el formato que Haskell espera. La sintaxis que este presenta para la definición de constructoras y tipos de datos es que estos tienen que empezar por mayúscula, y es por ello que el fichero

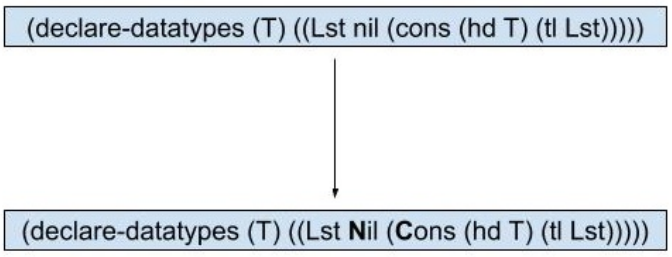

Figura 3.13: Ejemplo de cambio en el fichero *funciones.smt*

```
def imprimeArray(tam, modelo, var):
    final = "(fromList["
    for i in range(tam):
        final += "("+ str(i) +","+
                 str((modelo.eval(Select(modelo[var].arg(0), i))))
                 +")"
        if (i != tam - 1):
            final += ","
    final += "], " + str(tam) + ")"
    return final
```

Figura 3.14: Función para transformar un Array de Z3py en un Array de Haskell

funciones.smt ha sido modificado cambiando el nombre de todas las constructoras que utilizará Haskell para que empiecen por mayúscula como se muestra en la figura 3.13. Una vez que tenemos esto y teniendo en cuenta las clases *Read* y *Show*, definiremos como espera Haskell cada valor de las variables:

- *Arrays*: vendrán definidos por un par, en el que el primer elemento del array es un *Map* que usará la constructora *fromList*, esto es debido a que en Haskell no hay arrays como tales, es por ello que en la herramienta IR2Haskell se simulan con `Map Int a`, es decir, como un diccionario de enteros en valores. El segundo elemento será el tamaño del array. De esta manera, si tenemos el array `[5,8,10]`, el formato que emplearía la clase *Show* y *Read* para arrays sería de la forma `(fromList[(0,5),(1,8),(2,10)], 3)`, cuya interpretación para `(a,b)` será que en la posición *a* se encuentra almacenado el valor *b*. Esta transformación se llevará a cabo con la función de la figura 3.14, la cual recibe como parámetros el tamaño del array, el modelo generado por el resolutor y el nombre de la variable.

- *Tipos base y resto de tipos de datos*: los tipos base según se muestran en el modelo de Z3Py es como los esperan las clases *Show* y *Read* de Haskell. Los tipos de datos, si la constructora no tiene argumentos (es un átomo), al igual que en los tipos base, según se muestran en el modelo de Z3Py es como los esperan estas clases. Pero si estas constructoras tienen parámetros, de la forma en la que estas clases esperan estos será que vayan separados por espacios, en lugar de por "," como muestra el modelo Z3Py. Además, si se trata de una constructora de este tipo, su representación tendrá que venir dada entre

```
def textoRecursivo(estructura):
    final = ""
    if (estructura.num_args() == 0):
        return final + ("(" + str(estructura) + ") ")
    else:
        final += ("(" + str(estructura.decl()) + " ")
        for i in range(estructura.num_args()):
            final += textoRecursivo(estructura.arg(i))
        final += ") "
    return final
```

Figura 3.15: Función para transformar un tipo de datos que no sea array de Z3py a términos Haskell

```
;binSearch

;x
;v
;******

(declare-const x Int)
(declare-const v (Arr Int))
(assert (and (<= 0 x) (<= x 10)))
(assert (limitArray v 0 10))
(assert (<= (second v) 4))
(assert (sortedArr v 0 (second v)))
```

Figura 3.16: Fichero *.smt* de la función *binSearch* para la generación de caja negra

paréntesis. Es por ello, que tanto los tipos base, como las constructoras con argumentos y sin argumentos, se imprimirán entre paréntesis.

Es decir, suponiendo que tenemos el siguiente ejemplo: `NodeA(2, 2, LeafA, NodeA(4, 1, LeafA, LeafA))` la transformación a Haskell quedaría: `(NodeA (2) (2) (LeafA) (NodeA (4) (1) (LeafA) (LeafA)))`. De esto se encargará la función que se presenta en la figura 3.15 la cuál recibe por parámetro el modelo de la variable.

En este punto, ya hemos definido cómo se generan los casos de caja negra para un determinado programa, así como la manera en la que este imprime los resultados para ser procesados en las siguientes fases de este trabajo.

Por último, veremos cómo se comporta esta herramienta con la función *binSearch*, la cuál hace referencia a la búsqueda binaria en un array ordenado. Para esta función, el tamaño de los valores de las variables vendrá limitado por el rango $0 - 10$, ambos incluidos. Además, el tamaño de la estructura se limitará a ser ≤ 4. Con estas restricciones, el fichero *precd.smt* tiene el formato que se ilustra en la figura 3.16. En este podemos encontrar definido al comienzo del fichero el nombre del programa, seguido de las variables de interés, la declaración de estas y las restricciones de tamaño y valores (antes comentadas) seguido por último de la precondición (*sortedArr*).

Una vez que ejecutamos nuestra herramienta con el fichero anterior, generamos todos los casos de prueba que cumplen las restricciones impuestas en la

figura 3.16. Algunos de estos son los que se muestra en la figura 3.17 y, como se puede comprobar, presentan la sintaxis que se describía en esta sección.

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

Figura 3.17: Resultado de la generación de caja negra para las restricciones de la figura 3.16

Capítulo 4

Generación de casos de caja blanca

Otro de los objetivos de este trabajo será la generación de casos de prueba de caja blanca. La finalidad de estos será complementar la generación de casos de caja negra, ya que, aunque en nuestros programas de prueba los casos de caja negra generan todos los posibles casos dadas unas limitaciones, en programas más complejos que cuenten con numerosos condicionales (muchas sentencias `if-then-else`), lo más seguro es que los casos de caja negra no contemplen todos estos casos, y es ahí cuando los casos de caja blanca cubrirían esta carencia. Como se describía en la sección 2.5, lo que se persigue a partir de la herramienta AST2Z3 es teniendo en cuenta el flujo del programa, generar casos que comprueben hasta una profundidad dada, todos los posibles caminos del programa. Estos caminos, se plasmarán en el formato textual SMT-LIB en forma de asertos Z3. Cada camino declara las variables intermedias que aparecen en los `let` y los `case` y además, contiene asertos para los puntos de bifurcación del programa que son las ramas del `case`. Será a partir de estos, cuando en este trabajo, los interpretemos y obtengamos la satisfactibilidad de cada uno de ellos, y en el caso de estar ante un camino válido, generar un modelo para este. Este modelo, nos da el valor que tendrían que tomar los parámetros de entrada para así seguir el flujo de ese camino.

En la primera sección, se describirá el formato del fichero en el que se almacenan estos caminos en forma de aserto de Z3 y el método que usaremos para leer estos caminos. Una vez que tengamos definido lo anterior pasaremos a comentar por qué, de esos caminos, podemos encontrar que númerosos de ellos sean insatisfactibles, es decir, que se tratan de caminos imposibles de ejecutar. Por último, una vez tenemos definido todo lo necesario para la generación de estos, se desarrollará la técnica empleada así como el formato en el que estos resultados se devuelven para siguientes procesamientos.

4.1. Lectura de los caminos en formato SMT-LIB

Como describíamos en el apartado anterior, gracias a la herramienta AST2Z3, podemos obtener los caminos de un programa hasta una cierta profundidad siguien-

do el flujo de este. Estos caminos vendrán representados en forma de aserto de Z3. En un primer momento, la estructura del programa era:

1. *Declaración de tipos de datos.*
2. *Declaración de las funciones auxiliares.*
3. *Declaración de las variables de entrada.*
4. *Restricciones para las variables de interés.*
5. *Para cada camino*:
   - 5.1. *Restricciones del camino.*
   - 5.2. *Precondición.*
   - 5.3. *Check-sat.*
   - 5.4. *Get-model.*
   - 5.5. *Pop.*
   - 5.6. *Push.*

Teniendo en cuenta que tenemos que ir procesando camino a camino para obtener el modelo de las variables para cada uno, este modelo de fichero puede llegar a presentar problemas. La estrategia consistirá en ir leyendo cada línea del camino y cuando hemos terminado de leer un camino entero entonces pasamos a comprobar la satisfactibilidad de este. Cabe recordar como se dijo en la sección 2.2 que cada vez que insertamos en el resolutor una restricción, las funciones y variables a las que se haga referencia en esta deben de ser introducidas junto a su declaración, y en el caso de las últimas ser un tipo de datos, también junto a su definición.

Si tenemos en cuenta lo anterior y la estructura descrita, para comprobar la satisfactibilidad del primer camino tendríamos que leer línea a línea toda la declaración de tipos, seguida de la declaración de funciones y variables y de las restricciones. Una vez leído esto comenzaríamos a leer el primer camino, y cuando tuviéramos esto, probaríamos su satisfactibilidad. Una vez en este punto, tendríamos que procesar el segundo camino y así sucesivamente. El problema es que en leer línea a línea toda la sección previa a los caminos se desperdician muchos recursos y se reduce la eficiencia. Por ello, la opción mas viable es cambiar la estructura del fichero.

En un primer momento, una estructura que se barajó sería similar a la anterior, pero cada camino en vez de ir uno seguido de otro en el fichero, cada uno de estos iría en un fichero separado junto a la parte previa a estos igual que anteriormente (declaraciones de funciones y restricciones de las variables de interés). A la hora de leer cada camino sería inmediato, ya que tendríamos que volcar el fichero entero en el resolutor con la función `from_file()` sin necesidad de ir leyendo línea a línea y sería lo mas rápido y eficiente. El problema es que si un programa genera 10000 caminos como es el caso de la función *insertAVL* con profundidad 2, crearíamos

10000 ficheros y eso puede llegar a ser un problema de espacio en caso de valores mas grandes.

Es por ello, que el fichero que genera AST2Z3 quedará de la siguiente manera:

1. *Para cada camino.*
   1.1. *Número del camino.*
   1.2. *Restricciones del camino.*

En la figura 4.1 se muestra un ejemplo de este. En cuanto a las declaraciones de tipos y funciones, estas se encontrarán en el fichero *funciones.smt* como se explica en la sección 2.4. Por otro lado, la precondición y la declaración de las variables de entrada se encuentran en el fichero que la herramienta Precd2Z3 genera el cuál tendrá la estructura que se define en la sección 2.4. En último lugar, las funciones `check-sat`, `get-model`, `pop` y `push` de cada camino desaparecen, ya que estas funcionalidades se establecerán dentro de la herramienta Z3Py.

Con esta estructura, aunque tengamos que seguir leyendo los caminos línea a línea, la sección de declaración de variables y funciones se puede volcar directamente, factor que mejora la eficiencia de esta herramienta y, además, todos los caminos se encuentran en el mismo archivo, solventando uno de los problemas comentados anteriormente. Es por ello que, para leer un camino completo, tendremos que leer las restricciones del camino, el fichero *funciones.smt* y el fichero *precd.smt* generado por la herramienta *Precd2Z3*.

4.2. Caminos insatisfactibles

Como hemos visto en la anterior sección, en esta parte del trabajo nos centraremos en generar casos de caja blanca, a partir de unos caminos que vendrán dados en formato de aserto. Al procesar un camino completo y comprobar su satisfactibilidad para la obtención de un modelo, podemos obtener 3 resultados: *sat*, *unsat* y *unknown*. El resultado que podríamos esperar es que siempre los caminos que se generen sean satisfactibles, pero veremos la explicación de por qué podemos encontrar *unsat* y *unknown*. En primer lugar, para explicar el primer caso hay que definir lo que llamaremos *camino estático*. Estos caminos se definen a través de un conjunto de funciones recursivas definidas en la UUT (del inglés, *Unit Under Test*) como un camino de ejecución en potencia, comenzando desde la función del nivel superior y terminando cuando esta función produce un resultado. Pero no todas estos caminos estáticos se corresponden con caminos de ejecución reales, ya que algunos de estos caminos estáticos pueden no ser factibles. Estos caminos se generan hasta una profundidad dada, la cuál se definirá como el número máximo de despliegues que una función recursiva puede realizar en un camino. La herramienta AST2Z3 que usamos para generar los caminos, utiliza este tipo de caminos y este concepto de

profundidad. Es por ello, que dado un programa y una profundidad, genera todos los posibles caminos estáticos para esa profundidad, pudiendo ser muchos de ellos insatisfactibles.

También podemos encontrar caminos estáticos que sean satisfactibles, pero debido a la restricción de tamaño que podemos establecer a las variables del programa y estructuras puede ser que esto les convierta en insatisfactibles. Por ejemplo, la función *insertBST* (inserta un valor dentro de un árbol binario de búsqueda) con profundidad 1, generaría dos caminos como se muestra en la figura 4.1. En el primer camino, el árbol de entrada estaría vacío y la función terminaría insertando ese valor en un nodo, estableciendo la altura de este en 1. En el segundo caso, el valor que se intenta insertar, estaría en la raíz del árbol, y la función terminaría sin insertar. Es decir, se tratarían de los casos base del programa. Esto dos caminos son completamente satisfactibles, pero si añadimos como restricción que todas las estructuras tengan tamaño 0 como máximo, entonces el segundo camino, que tendría altura 1 (el tamaño de un BST vendrá definido por la altura), no podría ser satisfactible, ya que no cumpliría esa última restricción. Esto quiere decir que para caminos de la misma profundidad, dependiendo de las restricciones adicionales a los valores de las variables y estructuras podemos obtener que un mismo camino en ocasiones sea satisfactible y en otras ocasiones no, dependiendo únicamente de los valores anteriores.

Otro resultado que podemos obtener es *unknown*. Este resultado podemos encontrarlo incluso si el conjunto de restricciones de un camino son satisfactibles. Esto puede presentarse en programas que evalúan condiciones complejas y el resolutor no puede encontrar un modelo. Es decir, no podremos generar modelos para caminos cuya satisfactibilidad sea *unknown*.

4.3. Generación de casos

Teniendo ya definida la estructura del fichero donde se almacenan los caminos en forma de restricciones, qué partes componen un camino completo y qué clase de caminos nos podemos encontrar en este fichero como describíamos en la sección anterior, estamos preparados para definir cómo se generan estos casos de caja blanca.

Al igual que hacíamos en el capítulo 3, lo primero será detectar cuáles serán los nombres de las variables de interés. Para ello, gracias a que en el archivo *precd.smt* tenemos el nombre de estas, con la misma técnica comentada en la sección 3.3 conseguiremos saber el nombre de estas. Esta funcionalidad cobra más importancia en este punto, ya que a la hora de consultar un modelo para las restricciones insertadas, al declararse muchas variables auxiliares por cada camino, necesitamos poder distinguir sin problema estas variables con las variables de interés.

Seguido de esto, el conjunto de restricciones de cada camino tendrá que ser insertado en el resolutor junto con el contenido del fichero *funciones.smt* que contendrá todas las declaraciones de tipos y funciones necesarias y el fichero *precd.smt*

```
;path 1

(declare-const f1!1!x Int)
(declare-const f1!1!t (Tree Int))
(assert (= x f1!1!x))
(assert (= t f1!1!t))
(declare-const f1!1!res (Tree Int))
(declare-const res (Tree Int))
(assert (= res f1!1!res))
(assert (= f1!1!t Leaf))
(declare-const f1!1!empty_leaf (Tree Int))
(assert (= f1!1!empty_leaf Leaf))
(assert (= f1!1!res (Node f1!1!x f1!1!empty_leaf f1!1!empty_leaf)))

;path 2

(declare-const f1!1!x Int)
(declare-const f1!1!t (Tree Int))
(assert (= x f1!1!x))
(assert (= t f1!1!t))
(declare-const f1!1!res (Tree Int))
(declare-const res (Tree Int))
(assert (= res f1!1!res))
(declare-const f1!1!y Int)
(declare-const f1!1!l (Tree Int))
(declare-const f1!1!r (Tree Int))
(assert (= f1!1!t (Node f1!1!y f1!1!l f1!1!r)))
(declare-const f1!1!b Bool)
(assert (= f1!1!b (< f1!1!x f1!1!y)))
(assert (= f1!1!b false))
(declare-const f1!1!b1 Bool)
(assert (= f1!1!b1 (> f1!1!x f1!1!y)))
(assert (= f1!1!b1 false))
(assert (= f1!1!res f1!1!t))
```

Figura 4.1: Fichero de caminos insertBST

el cuál contendrá la declaración de las variables de entrada y las restricciones para limitar los valores y el tamaño de las estructuras de estas, además de la precondición del programa. Es decir, si suponemos que el fichero *funciones.smt* lo volcamos en la variable *func*, el fichero *precd.smt*, en la variable *prec* y las restricciones de un camino concreto las almacenamos (en forma de cadena de caracteres) en la variable *path*. Mediante la función

```
s.from_string(func + prec + path)
```

conseguiremos insertar todas las restricciones para un camino en el resolutor *s*. Para cada camino, una vez que el resolutor tiene toda esa información podemos pasar a comprobar su satisfactibilidad mediante la función `s.check()`, y en el caso de que este conjunto de restricciones sea *sat* podemos obtener un modelo gracias a la función `s.model()`. A partir del modelo obtenido, tendremos que recorrer sus variables, quedándonos únicamente con las que sean de interés. Cuando detectemos una, procederemos a almacenar su modelo correspondiente en un diccionario.

Al igual que en el capítulo 3, necesitamos que cada modelo tenga un formato *Haskell*. En este momento es cuando por primera vez en esta herramienta tendremos que volver a tener en cuenta la diferencia entre los arrays y el resto de estructuras de datos y tipos base. En el caso de tratarse de los primeros, para ello utilizaremos

la función descrita en la figura 3.14. Con esta, dado un array devuelto por el modelo podemos acceder a todos los valores de este para los índices comprendidos entre 0 y el tamaño del array. Es por ello que teniendo eso, es solo cuestión de hacer un *pretty printing* para crear un *string* que contenga dicha información en el formato que las clases *Show* y *Read* de Haskell lo esperan. Como ejemplo de esta transformación, podemos tomar por referencia un array v, cuyo modelo será `mk-pair(Store (Store (K (Int, 2), 2, 8), 1, 5), 3)`. La interpretación que le daremos a este es que v es un array de tamaño 3, cuyos valores serán: $a[0] = 2$, $a[1] = 5$ y $a[2] = 8$. La transformación que sufrirá para convertirlo en termino Haskell a partir de la función que estamos describiendo produciría como resultado:

```
(fromList[(0,2),(1,5),(2,8)], 3)
```

al igual que se describía en la sección 3.4.

Sin embargo, si nos centramos en el resto de estructuras de datos y tipos básicos, el modelo que devuelve Z3Py, en el caso de los tipos básicos, es el que espera Haskell, y en cuanto a las estructuras de datos, como comentábamos en 3.4, habría que cambiar las *","* por espacios en blanco. De esta transformación se encarga la función que se describe en la figura 3.15. Mediante esta, lo que conseguiremos será en el caso de los átomos imprimirlo tal cuál entre paréntesis, es decir, si tenemos el átomo `Leaf`, la transformación devolverá `(Leaf)`. Si por el contrario, no es un átomo, tendremos que introducir entre paréntesis el resultado de recorrer los argumentos de este recursivamente hasta llegar a un átomo. De esta manera, si obtenemos que para un árbol AVL su modelo es `NodeA(2, 2, LeafA, NodeA(3, 1, LeafA, LeafA)`, el resultado que obtendremos tras aplicar la transformación será:

```
(NodeA (2) (2) (LeafA) (NodeA (3) (1) (LeafA) (LeafA)))
```

Una vez que tenemos estos resultados almacenados en el formato Haskell y se han recorrido todas las variables del resolutor, procedemos a imprimir estos resultados en un fichero *.txt*. Estos resultados vendrán representados en una línea del fichero, es decir, cada línea del archivo será un modelo para un camino satisfactible. Al igual que se define en la sección 3.4, para cada camino, los resultados de las variables de interés se representarán como una tupla n-ária, en el que los valores de esta preservan el orden de los parámetros de la función. Además contaremos con un fichero auxiliar *.txt* en el que para cada camino, imprimiremos en una línea el número del camino que se trata y su satisfactibilidad. De tal manera que si para el camino 4, el resolutor nos devuelve *sat*, en el fichero encontraremos escrito

$$Path: \ 4 \ = \ sat$$

Por último, cuando hemos terminado de evaluar la satisfactibilidad de todos los caminos, al final del fichero anterior se informará del análisis obtenido para estos caminos. El formato que tendrá esta última parte vendrá dado por:

1. *Total de caminos evaluados*

```
((0)  , (fromList[] , 0))
((1)  , (fromList[(0 ,0)] , 1))
((0)  , (fromList[(0 ,0)] , 1))
```

Figura 4.2: Modelo para los caminos de la función binSearch

```
Path: 1 = sat
Path: 2 = sat
Path: 3 = sat

Total paths: 3
Satisfiable: 3
Unsatisfiable: 0
Unknown: 0
```

Figura 4.3: Análisis del resultado obtenido para los caminos de la función binSearch

2. *Total de caminos satisfactibles*
3. *Total de caminos insatisfactibles*
4. *Total de caminos unknowns*

Una vez que hemos descrito como se realizará la evaluación de los caminos, y que ficheros devolveremos junto con su contenido, pasamos a ver, al igual que en la sección 3.4, como se comporta esta herramienta con la función *binSearch* al generar casos de caja blanca. En primer lugar, en cuanto a los límites de las variables, serán los mismos que comentábamos entonces, es decir, el tamaño de las estructuras será ≤ 4, y los valores que podrán tomar todas estas variables estarán en el rango $0 - 10$. Es por ello, que utilizaremos el mismo fichero que en la figura 3.16 para cargar estas restricciones y la precondición. En cuanto a los caminos, al ejecutar la herramienta AST2Z3 con profundidad 2, genera 3 caminos como se muestra en la figura 4.4.

Una vez que evaluamos cada camino junto a las restricciones antes mencionadas, obtenemos dos ficheros. El primero contendrá el modelo para aquellos caminos que su evaluación sea satisfactible, en este caso, los 3 caminos lo son y por lo tanto encontramos 3 modelos como se muestra en la figura 4.2. El segundo fichero contendrá el análisis de estos caminos. El formato que presenta es el de la figura 4.3 que se adapta al descrito anteriormente.

```
;path 1

(declare-const a Int)
(assert (= a 0))
(declare-const l Int)
(assert (= l (second v)))
(declare-const b Int)
(assert (= b (- l 1)))
(declare-const bin!1!a Int)
(declare-const bin!1!b Int)
(declare-const bin!1!x Int)
(declare-const bin!1!v (Arr Int))
(assert (= a bin!1!a))
(assert (= b bin!1!b))
(assert (= x bin!1!x))
(assert (= v bin!1!v))
(declare-const bin!1!p2 Int)
(declare-const p Int)
(assert (= p bin!1!p2))
(declare-const bin!1!b1 Bool)
(assert (= bin!1!b1 (> bin!1!a bin!1!b)))
(assert (= bin!1!b1 true))
(assert (= bin!1!p2 bin!1!a))

;path 2

(declare-const a Int)
(assert (= a 0))
(declare-const l Int)
(assert (= l (second v)))
(declare-const b Int)
(assert (= b (- l 1)))
(declare-const bin!1!a Int)
(declare-const bin!1!b Int)
(declare-const bin!1!x Int)
(declare-const bin!1!v (Arr Int))
(assert (= a bin!1!a))
(assert (= b bin!1!b))
(assert (= x bin!1!x))
(assert (= v bin!1!v))
...
```

Figura 4.4: Caminos generados para la función binSearch

Capítulo 5

Ejecutor-validador de casos

En este momento se trata de interpretar la CLIR como un programa y correrlo sobre cada uno de los casos de prueba generados antes para comprobar sí los resultados para cada uno de estos, verifican la postcondición. Para ello, debido a que, como ya se ha dicho, la CLIR no es ejecutable, se transforma esta a un programa Haskell que sí es ejecutable, a partir de la herramienta IR2Haskell, que será el que finalmente se ejecutará con esos casos.

Para explicar el procedimiento que seguiremos para ello, en la primera sección trataremos la librería *Template Haskell*, gracias a esta podremos analizar en tiempo de compilación los tipos de los argumentos y resultados del programa a evaluar. En la segunda sección, mostraremos cómo será la interacción con la UUT desde un programa en Haskell. La verificación de los resultados a partir de la postcondición se hará al igual que en las secciones anteriores, con la herramienta Z3Py. Es por ello que necesitamos en este caso una conversión a la inversa que la comentada en las secciones 3.4 y 4.3, del formato Haskell, al formato textual SMT-LIB, es decir, a asertos Z3. En la tercera sección, se explicará la transformación que estos resultados sufren. En la última sección, mostraremos un ejemplo de ejecución de esta herramienta tanto para un programa en el que no se detectan errores, como para uno en el que sí se detectan.

5.1. La herramienta Template Haskell

Template Haskell (en adelante, TH) [10] es una API del compilador GHC[1] [5] de Haskell para realizar metaprogramación, con un estilo muy similar al de las plantillas de C++. Esta herramienta, nos permite acceder en tiempo de compilación a módulos previamente compilados y obtener información sobre ellos. También permite construir código en tiempo de compilación adaptado a la información obtenida, el cuál será posteriormente compilado. Sin embargo, a diferencia de las anteriormente mencionadas plantillas de C++, el lenguaje para programar con TH no es un lenguaje distinto del lenguaje generado, sino que será el propio Haskell.

[1] *Glasgow Haskell Compiler*

Para este trabajo, utilizamos TH en el ejecutor de casos de prueba. Las razones para ello son que esta herramienta ha de estar preparada para llamar a la función principal de la UUT, y que estas funciones pueden presentar nombres, número de parámetros y tipos de estos diferentes en cada caso.

A su vez, el ejecutor también tiene que leer los casos de prueba en un formato que difiere según el tipo de estos y ha de convertir los resultados devueltos por la UUT a términos y asertos Z3. De nuevo, el nombre y el tipo de dichos resultados variará de una UUT a otra.

Para facilitar la tarea, hemos establecido los siguientes convenios:

- La versión Haskell de la UUT estará contenida en un único módulo Haskell llamado `UUT.hs`, el cual será importado desde el ejecutor de casos. La estructura que presentará este fichero será el descrito en la siguiente sección.

- En dicho módulo, estará definido el nombre exportado `uutMethod` de tipo `String` cuyo valor será el nombre de la función visible de la UUT, es decir, la que se desea probar.

Con esa información, el ejecutor accede en tiempo de compilación al tipo de la función principal de la UUT e investiga el número y el tipo de sus parámetros y resultados. Las funciones más importantes usadas por el ejecutor son:

- `uutTopName`: devuelve el nombre de la función principal de la UUT.

- `uutNargs`: devuelve el número de argumentos de entrada de la función `uutTopName`.

- `uutNres`: devuelve el número de resultados de la función `uutTopName`.

- `uutType`: devuelve una tupla n-aria que almacena los tipos de todos los argumentos de entrada de la función `uutTopName`. Gracias a esta tupla, se leerán los casos de prueba utilizando la función `read` de la clase `Read`. Dicha función analiza sintácticamente un texto dado acorde con el tipo tupla antes mencionado.

- `untuple`: esta función recibe una tupla n-aria de valores de distintos tipos y devuelve una lista del mismo tamaño con la representación en forma cadena de caracteres de cada uno de ellos. Se utilizará para separar los valores de cada caso de prueba y para separar los resultados devueltos por la UUT, cuando estos son más de uno, como se describe en la siguiente sección.

A partir de la definición de estas funciones, podemos pasar a describir en detalle cuál será el proceso mediante el que interaccionaremos con la UUT.

```
get_array::Array a -> Int -> a
get_array a i = if isNothing e
    then throw (IndexOutOfRange i) else fromJust e
    where e = M.lookup i (fst a)
```

Figura 5.1: Definición función get_array

5.2. Interfaz con la UUT

En primer lugar describiremos el formato de dicha interfaz, ya que como hemos dicho, se tratará de un fichero Haskell. La estructura vendrá dada por:

- `import Arrays`: con esto se importará el fichero `Arrays.hs`, el cuál contiene la declaración del tipo de datos array y todas las funciones auxiliares necesarias para estos en formato Haskell. Debido a que en Haskell no existen arrays como tal, la estructura con la que se han definido en dicho módulo es como una tupla cuyo primer elemento es `Map Int a` y el segundo elemento es un entero que simboliza el tamaño. Algunos ejemplos de estas funciones serían: `len` para obtener el tamaño de un array, `get_array` (cuya definición se muestra en la figura 5.1) para obtener el elemento de una posición y `set_array` para actualizar un elemento de una posición.
- `import SupportedTypes`: en este caso se importará el fichero `SupportedTypes.hs` que contendrá todas las declaraciones en formato Haskell de las estructuras de datos que utilizaremos para este trabajo y que se muestran en la figura 5.2.
- *constante* `uutMethod`: esta constante devolverá el nombre de la función principal de la UUT a evaluar, es decir, si esta fuera *insertAVL*, esa función devolverá esa cadena de caracteres.
- *Función principal del programa*: esta será la función que se desea probar, es decir, la anteriormente mencionada `uutTopName`. El nombre de esta coincidirá por lo tanto con la cadena de caractéres que devuelve la anterior. Contendrá el código Haskell equivalente al código de la CLIR de la cual procede.

Una vez conocida la estructura de este fichero, procedemos a describir cómo será el ejecutor de casos. Constará de dos partes, una parte Haskell que interaccionará con este fichero y un programa Python que interaccionará con Z3 a través de su API, Z3Py, para validar los resultados obtenidos. La función principal de la parte Haskell recibirá tres argumentos:

1. *Fichero de la postcondición*: en este argumento encontramos la ruta exacta del fichero *poscd.smt*, que contendrá una estructura similar al fichero *precd.smt*, descrito en la sección 3.1. Como variables de interés, en este caso, también se tiene en cuenta las variables de retorno. En lugar de la precondición, aparece la postcondición y las restricciones de tamaño y valores desaparecen. Para la generación de este fichero se ha modificado la herramienta AST2Z3 para que además de generar caminos, devuelva este fichero.

```
data Color   = Rojo | Negro
             deriving (Read, Show)

data Lst a   = Nil
             | Cons a (Lst a)
             deriving (Read, Show)

data Tree a  = Leaf
             | Node  a (Tree a) (Tree a)
             deriving (Read, Show)

data LLRB a  = LeafL
             | NodeL  a Color (LLRB a) (LLRB a)
             deriving (Read, Show)

data AVL a   = LeafA
             | NodeA  a Int (AVL a) (AVL a)
             deriving (Read, Show)
```

Figura 5.2: Contenido del fichero SupportedTypes.hs

2. *Fichero con los casos de prueba*: la ruta exacta del fichero *.txt* donde se encuentran todos los casos de prueba en el formato que las clases *Show* y *Read* de Haskell lo esperan para convertirlos en valores Haskell del tipo apropiado.

3. *Fichero para los resultados*: contendrá la ruta del fichero en la que se imprimirá el resultado obtenido de ejecutar y validar el programa con los casos de prueba anteriores.

Al obtener los casos de prueba con los que probaremos el programa, estos casos se analizan sintácticamente teniendo en cuenta el tipo de los parámetros de la función. Esto se conseguirá gracias a la función `uutType` cuya definición se describía en la sección 5.1 y que utilizará el nombre del programa. Cada caso de prueba dado como una tupla n-aria en forma de cadena de caracteres, es analizado por la función `read` según el tipo del valor que se espera que la ocupe. De esta manera tendremos almacenado todos los casos de prueba, analizados sintácticamente, en una lista de tuplas n-arias tipadas.

El ejecutor necesita conocer cuál es la función principal, que será la que deseamos probar con los casos anteriores. Para ello, leeremos el nombre y el tipo de esta función desde un módulo que importa la UUT que se supone ya compilada. Crearemos una función `f (x1,...,xn)` en tiempo de compilación la cual se comporta como la función de la UUT con la diferencia de que la nueva función creada recibirá los argumentos en una tupla. Es decir, si suponemos que tenemos una función *test* que recibe 2 parámetros enteros de entrada la forma para invocar esta función en la UUT sería:

```
test 2 3
```

y trás la creación de la nueva función f quedaría:

```
f (2, 3)
```

Generalizando, la transformación que sufría la función `uutTopName` sería:

```
f (x1,...,xn) = uutTopName x1 ... xn
```

Una vez que tenemos los casos de prueba analizados sintácticamente y la función ya definida, podemos pasar a ejecutarla con esos casos. Para ello, aplicaremos la función f a cada caso de prueba. Esto producirá un resultado para cada uno de estos, el cuál se almacenará en una lista de tuplas n-arias. Llegados a este punto, tendremos almacenados todos los casos que hemos probado y los resultados obtenidos al aplicar la función f a cada uno de ellos.

Por último, nos faltaría validar cada uno de estos utilizando la postcondición. Este procedimiento se explicará en la sección 5.4. La validación de estos se realizará en un programa auxiliar Python (llamado desde la parte Haskell) mediante la interacción con Z3 a través de su API, Z3Py. Necesitaremos una traducción inversa de los valores en Haskell a un formato de aserto de Z3. En la siguiente sección se explicará la técnica usada para esto.

5.3. Convirtiendo términos Haskell en asertos Z3

Necesitamos transformar los valores en asertos de Z3. Esta transformación se desarrollará en Z3Py. La idea es que estos asertos se inserten en el resolutor como cadenas de caracteres. En el caso de que el valor a transformar en aserto sea un array, previo a esto habrá que transformar el array al formato SMT-LIB. Para ello, partiendo del array v, la idea será añadir una instrucción de *store* en la nueva cadena de caracteres por cada posición del array, es decir, si tenemos un array de tamaño 3, en nuestra transformación aparecerían 3 llamadas a *store*.

La sintaxis que seguirá a cada *store* será la ya descrita en la sección 2.2. Es por ello que recorreremos todas las posiciones del array, y si estamos en la posición 0 del array lo que añadiremos a nuestra cadena *cad* será `(store (first v) 0 valor)`. Pero si por el contrario estamos en cualquier otra posición entre la 1 y el tamaño del array (este último sin incluir), habría que añadir `(store cad posición valor)` y así sucesivamente. Por último, cuando hemos recorrido todas las posiciones, tan solo habría que crear una nueva cadena de caracteres de la forma `(mk-pair cad tamaño)`. De manera que si partimos del array `v = (fromList[(0,1),(1,4),(2,5)], 3)` en formato Haskell, la transformación al lenguaje SMT-LIB quedaría de la forma:

```
(mk-pair (store (store (store (first v) 0 1) 1 4) 2 5), 3)
```

Si por el contrario, nos encontramos ante un array vacío (tamaño 0), la transformación quedaría de la forma:

```
(mk-pair (store (first v) -1 -1) 0)
```

Lo que nos querrá decir es que en la posición -1 almacenaremos un -1. Esto no será relevante a la hora de la validación, pero si para poder tener una representación del array vacío en el resolutor. Decimos que no será relevante porque al consultar los valores en la posiciones de este, de la forma que hacíamos en los capítulos 3 y

4, comprobaríamos desde la posición 0 hasta la del tamaño sin incluir, es decir, no accederíamos a ninguna posición en este caso.

Si, por el contrario, el valor es un tipo base o un átomo, el valor tal cual lo genera Haskell, es cómo se utilizará para crear el aserto. De otra manera, si se trata de cualquier otra constructora con argumentos, la transformación consistirá en añadir el valor entre paréntesis, es decir, si tenemos el valor en formato Haskell `NodeA 1 2 LeafA LeafA` la transformación quedaría de la forma:

```
(NodeA 1 2 LeafA LeafA)
```

Una vez que hemos descrito qué transformaciones van a sufrir todos los posibles valores, solo nos quedaría crear el aserto con el nombre de la variable y el valor transformado.

Para ello, al igual que hacíamos en los capítulos 3 y 4, detectaremos cuáles son el nombre de estas variables de interés. En este caso, las encontraremos al comienzo de fichero *poscd.smt*. De manera que una vez que tenemos el nombre de la variable y el valor de esta, tan solo nos quedaría asignarle a esta variable ese valor que hemos obtenido. Para conseguir esto bastaría con introducir el aserto:

```
(assert(= nombre valor))
```

5.4. Validación y refutación del caso de prueba

Una vez explicado cómo conseguimos interactuar con la UUT, y la serie de transformaciones que sufren estos valores en formato Haskell para ser convertidos en asertos de Z3, podemos pasar a ver cómo se comporta esta herramienta con cada caso de prueba.

Partiendo de donde nos quedamos en la sección 5.2, una vez que tenemos los resultados y los casos de prueba almacenados, pasamos a evaluarlos uno a uno. De esto se encargará el programa Python que se describía en la sección 5.3. Para cada caso, se invoca a este desde el programa Haskell con una serie de parámetros de entrada:

- *Ruta del fichero .smt que contiene la postcondición*: que lo obtendremos por parámetro cuando se ejecuta este programa Haskell.
- *Caso de prueba*: según la estructura que hemos descrito para estos, el formato en el que los encontraremos será como una tupla n-aria tipada. Para facilitar el manejo de estos valores, haremos uso de la función `untuple`, descrita en la sección 5.1. De esta manera, tendremos todos los parámetros del programa como cadena de caracteres en una lista.
- *Resultado*: al igual que los anteriores, se le aplicará al resultado obtenido la función `untuple` para transformar esa tupla n-aria en una lista del mismo tamaño que almacenará el contenido de esta en formato de cadena de caracteres.

Es decir, si suponemos que estamos evaluando la función *binSearch* y, por lo tanto, su postcondición se encuentra en el fichero *binSearchPoscd.smt*; la instancia de la clase `Show`, para el caso de prueba, es de la forma `(3, (fromList[(0,1)], 1))` el cuál tras aplicarle la función `untuple` producirá como resultado: `["3", "(fromList[(0,1)], 1)"]`; y la instancia de la clase `Show` para el resultado es `1`, el cuál tras aplicarle la misma transformación produciría como resultado: `["1"]`, la lista de parámetros que recibirá nuestro programa Python será:

```
[binSearchPoscd.smt, 3, (fromList[(0,1)], 1), 1]
```

En este momento, en nuestro programa Python tenemos casi todo lo necesario para validar este caso de prueba. Únicamente nos faltaría saber cuál es el nombre de las variables de interés y que valor, de los que hemos pasado por parámetro, le corresponde a cada una de estas. Para averiguar esto, utilizaremos el fichero *binSearchPoscd.smt* en este caso. Como ya hemos comentado, al comienzo de este fichero vendrán declaradas estas variables, tanto las de entrada como las de salida. La forma de obtener esta información del fichero se hará de la misma manera que se explicaba en la sección 3.3.

Una vez que tenemos cuáles son las variables, faltaría por último saber qué valor de los que recibimos por parámetro se le asigna a que variable. Esta asignación será inmediata, puesto que los valores que se reciben por parámetro están en el mismo orden que la declaración de las variables. Es decir, si partimos del fichero de la imagen 5.3, a la variable x le correspondería el valor 3, a la variable v el valor `(fromList[(0,1)], 1)` y a la variable p el valor 1. De esta manera, una vez que sabemos qué valor le corresponde a qué variable, podemos proceder a transformar cada una de estas igualdades a un formato de aserto de Z3, siguiendo el procedimiento descrito en la sección 5.3. De esta manera, los asertos resultantes quedarían de la forma:

```
(assert(= x 3))
(assert(= v (mk-pair((store (first v) 0 1) 1))))
(assert(= p 1))
```

Ya introducidos estos asertos en el resolutor, pasamos a comprobar su satisfactibilidad para devolver este dato al programa Haskell principal. Es en este punto donde, si el resultado devuelto por el programa Python es *"sat"*, diremos que ese caso ha pasado la prueba satisfactoriamente y daremos paso al siguiente caso. Si, por el contrario, el resultado devuelto es *"unsat"*, significa que nuestro caso no ha pasado la prueba, es decir, que hemos encontrado un error en el programa a evaluar debido a que el resultado obtenido no cumple la postcondición de este. En este caso, anotaremos cuáles eran los parámetros de entrada, y cuál ha sido el resultado obtenido. De igual manera, procederíamos a evaluar el siguiente caso.

Este procedimiento, se realizará para cada caso de prueba. Una vez que hemos terminado de evaluar todos, el programa Haskell devolverá un fichero *.txt* cuyo formato será:

```
;binSearch

;x
;v
;p
;******

(declare-const x Int)
(declare-const v (Arr Int))
(declare-const p Int)
(assert (and (<= 0 p) (and (<= p (second v)) (and
        (forall ((j Int) ) (=> (<= 0 j) (=> (< j p)
        (< (get_array v j) x)))) (forall ((j Int) )
        (=> (<= p j) (=> (< j (second v))
        (<= x (get_array v j)))))))))
```

Figura 5.3: Contenido del fichero binSearchPoscd.smt

```
Total examples: 1500
Total successful: 1500
Total errors: 0

All test cases were successful !
```

Figura 5.4: Resultado obtenido tras la prueba exitosa de la función binSearch

1. *Total de casos probados*
2. *Total de casos que pasaron la prueba con éxito*
3. *Total de casos que fallaron al pasar la prueba*

Seguido de esto, si no ha fallado ningún caso aparecerá un mensaje de éxito, pero si, por el contrario, al menos uno de los casos ha fallado la prueba, se imprimirá para cada uno de estos, qué parámetros presentaba ese caso y qué resultado se ha obtenido de ejecutar la función con esos parámetros.

A continuación ilustramos cómo se comporta esta herramienta con la función *binSearch*. Los casos con los que se probará esta función serán de caja negra, generados como se describe en el capítulo 3. Algunos de estos serán los que se muestran en la figura 3.17. El fichero que contendrá la postcondición para esta función será el mismo que se muestra en la figura 5.3. El resultado de ejecutar esta herramienta con esa información producirá como resultado el que se muestra en la figura 5.4. Como se puede apreciar, los 1500 casos han pasado la prueba satisfactoriamente, por lo que se imprime un mensaje de éxito.

Sin embargo, ahora probaremos cómo se comporta esta herramienta ante esta misma función pero introduciendo en el código algunos errores. Como se puede ver en la figura 5.5, de los 1500 casos probados, hemos encontrado que 446 fallan al pasar esta prueba y, por lo tanto, la función que estamos probando contiene errores. Como muestra de esto, encontramos en esa misma figura algunos de estos casos que han fallado. Con la función *binSearch* lo que se obtiene como resultado es un valor p en el que a la izquierda de este, todos son menores que el elemento buscado x, y

```
Total examples: 1500
Total successful: 1054
Total errors: 446

ERRORS FOUND:
"Args: (9,(fromList [(0,4),(1,8),(2,9)],3)), Result: 3"
"Args: (9,(fromList [(0,7),(1,8),(2,9)],3)), Result: 3"
"Args: (9,(fromList [(0,2),(1,3),(2,9)],3)), Result: 3"
"Args: (9,(fromList [(0,3),(1,3),(2,9)],3)), Result: 3"
"Args: (9,(fromList [(0,2),(1,2),(2,9)],3)), Result: 3"
"Args: (9,(fromList [(0,0),(1,0),(2,9)],3)), Result: 3"
"Args: (9,(fromList [(0,0),(1,1),(2,9)],3)), Result: 3"
"Args: (9,(fromList [(0,0),(1,2),(2,9)],3)), Result: 3"
"Args: (9,(fromList [(0,0),(1,3),(2,9)],3)), Result: 3"
"Args: (9,(fromList [(0,0),(1,4),(2,9)],3)), Result: 3"
"Args: (9,(fromList [(0,0),(1,7),(2,9)],3)), Result: 3"
"Args: (9,(fromList [(0,0),(1,8),(2,9)],3)), Result: 3"
"Args: (9,(fromList [(0,0),(1,9),(2,9)],3)), Result: 2"
"Args: (9,(fromList [(0,0),(1,6),(2,9)],3)), Result: 3"
...
```

Figura 5.5: Resultado obtenido tras la prueba de la función errónea binSearch

desde p hasta el tamaño del array serán $\geq x$. Es decir, si aparece x, su aparición más a la izquierda (en el caso que salga repetido) será p, y si no aparece, p es la posición donde habría que insertarlo.

Ya con la definición de la función, si nos fijamos en el caso:

```
Args: (9,(fromList [(0,4),(1,8),(2,9)],3)), Result: 3
```

Podemos comprobar que lo que debería devolver la función es 2 ya que es en la posición donde se encuentra el valor que estamos buscando, en este caso 9, pero en su lugar devuelve un 3.

Capítulo 6

Experimentos

Una vez descritas las herramientas principales del sistema que hemos implementado en este trabajo, pasaremos a describir en la primera sección de este capítulo cuál es la estructura completa de este, además de cómo el usuario interacciona con él. En la segunda sección, mostraremos una serie de experimentos realizados para un conjunto de funciones con diferentes restricciones en sus estructuras de entrada. Por último, mostraremos casos reales en los que algunas de las funciones que probamos presentan errores y, cómo el sistema es capaz de detectarlo e informar de ello.

6.1. Estructura del sistema completo

En esta sección, describiremos cuál es la estructura de nuestro sistema, así como la manera en la que el usuario interacciona con este. La estructura del sistema se muestra en la figura 6.1. En primer lugar, el usuario interacciona con la herramienta *caminos*. Esta se corresponde con las herramientas AST2Z3 y Precd2Z3, ya comentadas en las secciónes 2.5 y 2.4 respectivamente. En este trabajo, se ha decidido combinar estas dos en una, que llamamos *caminos*. Con ella, el usuario establece una serie de restricciones: el tamaño máximo de las estructuras, el rango de los valores contenidos en las estructuras y la profundidad máxima a la hora de generar caminos. Esta leerá el programa descrito en el lenguaje de la CLIR, *uut.clir* y junto a las restricciones anteriores generará los archivos: *precd.smt*, *paths.smt* y *poscd.smt*.

Por otro lado, gracias a la herramienta IR2Haskell, descrita en la sección 2.3, podremos convertir dicho programa, dado en el lenguaje de la CLIR, a un conjunto de funciones Haskell ejecutables, obteniendo como resultado el fichero Haskell `UUT.hs`. Este fichero generado se almacenerá junto a lo que se denomina *"fuentes del ejecutor"*, que corresponden a los ficheros que constituyen el ejecutor más algunos ficheros auxiliares que son importados por `UUT.hs`.

Retomando los ficheros devueltos por *caminos*, el ejecutable *cajaNegra* generará a partir del fichero *precd.smt* y la interacción con Z3, los casos de caja negra que cumplen las restricciones que se especifican en ese fichero, produciendo como

resultado el fichero *blackBox.txt* que contendrá los casos obtenidos. De igual manera, el ejecutable *cajaBlanca*, generará a partir del fichero *precd.smt*, *paths.smt* y la interacción con Z3, los casos de prueba de caja blanca. Esto producirá como resultado dos ficheros: *whiteBox.txt* y *SAT.txt* donde encontraremos los casos generados y el análisis obtenido de los caminos, respectivamente.

En este punto, procedemos a probar el programa con la herramienta *ejecutor*, que primero habrá de ser compilada junto con `UUT.hs`. Esta recibirá por un lado el programa en formato Haskell junto con las *fuentes del ejecutor*, el fichero con los casos que se desea probar nuestro programa, en este caso, *blackBox.txt* o *whiteBox.txt*, y el fichero *postcd.smt*. De esta manera, la herramienta interaccionará con Z3 a través de la API, Z3Py, para probar el programa con estos casos. Por último, devolverá un fichero *blackBoxRes.txt* en el que encontramos el resultado de esta prueba para los casos de caja negra, y de igual manera, un fichero *whiteBoxRes.txt* que contendrá el resultado de esta prueba pero para los casos de caja blanca. Con este último fichero, el usuario comprobará si se han encontrado o no fallos en el programa con los casos que se han probado, y en el caso de que estos existan, notificarle cuáles han sido.

Para facilitar al usuario el uso de este sistema, se ha desarrollado una pequeña GUI para automatizar el proceso anterior. La forma que esta presenta es el mostrado en la figura 6.2 y las distintas opciones que ofrece son:

- *Opción 1*: en este desplegable aparecerán todas las CLIR localizadas en las carpetas del sistema.

- *Opción 2*: con esta opción, dada una CLIR podremos generar los ficheros *precd.smt*, *paths.smt* y *poscd.smt* para dicha CLIR con las restricciones de tamaño que se requieran, las cuales se establecen en los cuadros de texto que aparecen. El primer cuadro se corresponderá con el tamaño máximo de las estructuras, el segundo y el tercero con el valor mínimo y máximo, respectivamente, para el contenido de estas, y el cuarto con la profundidad máxima para generar los caminos.

- *Opción 3*: genera el fichero Haskell ejecutable, `UUT.hs`, a partir de la CLIR seleccionada.

- *Opción 4*: genera los casos de caja negra teniendo en cuenta las restricciones almacenadas en el fichero *precd.smt*, el cual tendrá que haber sido generado previamente con la opción 2.

- *Opción 5*: genera los casos de caja blanca teniendo en cuenta las restricciones almacenadas en los fichero *precd.smt* y *paths.smt*, que previamente deben de haber sido generados con la opción 2.

- *Opción 6*: prueba el programa Haskell obtenido por la opción 3 con los casos de prueba de caja negra, los cuáles deben haber sido generados por la opción 4.

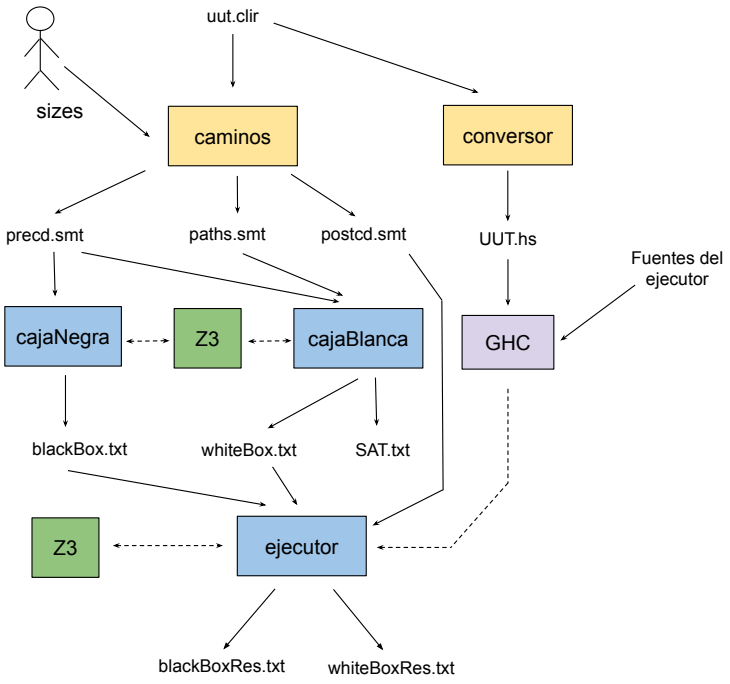

Figura 6.1: Plataforma CAVI-TEST

- *Opción 7*: prueba el programa Haskell obtenido por la opción 3 con los casos de prueba de caja blanca, que previamente deben haber sido generados por la opción 5.

De esta manera, si el usuario desea generar los casos de caja negra para la CLIR *insertAVL* con: tamaño máximo de las estructuras = 2, valor mínimo = 1, valor máximo = 3 y profundidad máxima para los caminos = 2, entonces deberá seleccionar la CLIR en la opción 1, seguido de esto, marcar la función 2 y establecer los parámetros descritos anteriormente y, por último, la opción 4 para generar estos. Si además se quiere validar esa CLIR con los casos de caja negra, habría que seleccionar la opción 3 para transformar esa CLIR a un formato Haskell ejecutable y la opción 6 para realizar esa validación.

6.2. Ejemplos probados

En esta sección iremos mostrando resultados obtenidos de nuestro sistema al ejecutarlo con diferentes CLIRS. Las que utilizaremos serán:

- `insertList x l`: inserta el elemento x en la lista ordenada l.
- `deleteList x l`: elimina el elemento x de la lista ordenada l.

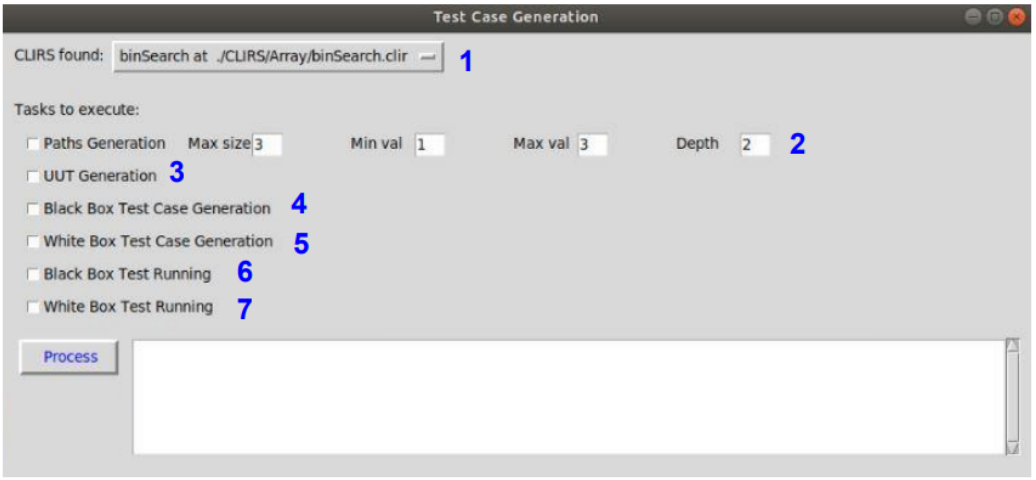

Figura 6.2: Formato de la GUI para la herramienta CAVI-TEST

- `searchLLRB x t`: busca el elemento x en el árbol LLRB (del inglés, *Left-Leaning Red-Black*) t.
- `insertLeftist x t`: inserta el elemento x en el montículo zurdo t.
- `unionLeftist t1 t2`: realiza la unión entre dos montículos zurdos $t1$ y $t2$.
- `insertBST x t`: inserta el elemento x en el árbol BST (del inglés, *Binary Search Tree*) t.
- `searchBST x t`: busca el elemento x en el árbol BST t.
- `insertAVL x t`: inserta el elemento x en el árbol AVL t.
- `searchAVL x t`: busca el elemento x en el árbol AVL t.
- `binSearch x v`: busca el elemento x, mediante el método de la búsqueda binaria, en el array v.
- `dutchNationalFlag v`: resuelve el problema *Dutch national flag* para el array ordenado v. Este problema consiste en, dados unos valores que representan los colores de la bandera Holandesa (rojo, blanco y azul), organizar estos valores de forma que todos los que sean iguales estén juntos formando un grupo y, que cada uno de estos, este en el orden correcto.
- `fill x v`: inserta el elemento x en todas las posiciones del array v.
- `insertArray x v`: inserta el elemento x en el array v ordenado ascendentemente.
- `linearSearch x v`: busca el elemento x, mediante el método de la búsqueda lineal, en el array v.
- `qSortMod v`: devuelve el array v, ordenado ascendentemente, mediante el método del *quick sort*.

- `selSort`: devuelve el array *v*, ordenado ascendentemente, mediante el método del *select sort*.

- `insSort`: devuelve el array *v*, ordenado ascendentemente, mediante el método del *insertion sort*.

En este momento, pasamos a ejecutar el sistema completo para todas las funciones anteriores con las restricciones de tamaño: *tamaño máximo de las estructuras* = 2, *valores mínimo y máximo* = 1 y 3 respectivamente, y *profundidad máxima de los caminos* = 1. El resultado obtenido es el que se muestra en la figura 6.3. En esta tabla podremos encontrar, para cada función, el número de casos de caja negra generados, *blackBox*; la evaluación obtenida para los caminos, *whiteBox*, que contendrá los casos de caja blanca generados en el valor *"sat"*; el resultado de probar la función con los casos de caja negra en *blackBoxResul* y, por último, el resultado de probarla con los casos de caja blanca en *whiteBoxResul*.

Lo primero que se puede observar es que, si establecemos la profundidad máxima de los caminos a 1, en la función que más caminos se generan es en *deleteList*, que generará 3. De igual manera, a excepción de la función *unionLeftist*, obtenemos en general muy pocos casos de caja negra, cuyo motivo será el rango tan pequeño que hemos establecido para los tamaños. Otro factor importante que se puede apreciar, es que si nos fijamos en la función *insertLeftist*, al probar esta función con los casos de caja negra, obtenemos que 19 de los 40 generados, no han pasado la prueba satisfactoriamente, es decir, hemos encontrado un error en la función. Sin embargo, si hubiéramos probado esta misma función con el caso de caja blanca, habríamos obtenido que este habría pasado la prueba sin problemas. Con este ejemplo, se refuerza más la idea de que los casos de caja blanca y caja negra se complementan entre sí.

En segundo lugar, pasamos a ver cómo se comporta el sistema si ampliamos el rango del valor mínimo y máximo a 1 y 5, respectivamente. Además, también se aumenta el tamaño máximo de la profundidad de los caminos a 2. Esto produce el resultado que se muestra en la figura 6.4. Como se puede apreciar, al aumentar el tamaño máximo de la profundidad de los caminos, la herramienta AST2Z3 puede producir más de estos. Aún así, la cantidad que se genera de estos es muy similar a la cantidad descrita anteriormente, a excepción de la función *insertAVL* que genera 10306 caminos, de los cuáles, tan solo 1 de estos es satisfactible. Si por el contrario, nos fijamos en los casos generados de caja negra, en la mayoría de funciones, estos aumentan considerablemente respecto a los anteriores. Esto hace que en este caso la diferencia entre los casos generados de caja negra y caja blanca crezca considerablemente respecto al caso anterior. Si nos volvemos a fijar en la función *unionLeftist*, de los 6 casos de caja blanca, todos pasan la prueba sin problema, pero sin embargo, de los 1500 casos de caja negra, 552 casos presentan problemas en la función.

La última prueba consistirá en aumentar todos los rangos, incluidos esta vez, el tamaño máximo de las estructuras. Los nuevos tamaños que asignaremos a nuestro sistema serán: *tamaño máximo de las estructuras* = 3, *valores mínimo y máximo* = 1 y 8 respectivamente, y *profundidad máxima de los caminos* = 3. Los

Función	blackBox	whiteBox	blackBoxResul	whiteBoxResul
insertList	21	sat = 2 unsat = 0 unknown = 0	ok = 12 error = 9	ok = 1 error = 1
deleteList	21	sat = 3 unsat = 0 unknown = 0	ok = 21 error = 0	ok = 3 error = 0
searchLLRB	27	sat = 2 unsat = 0 unknown = 0	ok = 27 error = 0	ok = 2 error = 0
insertLeftist	40	sat = 1 unsat = 1 unknown = 0	ok = 21 error = 19	ok = 1 error = 0
unionLeftist	168	sat = 2 unsat = 0 unknown = 0	ok = 74 error = 94	ok = 2 error = 0
insertBST	42	sat = 2 unsat = 0 unknown = 0	ok = 42 error = 0	ok = 2 error = 0
searchBST	42	sat = 2 unsat = 0 unknown = 0	ok = 42 error = 0	ok = 2 error = 0
insertAVL	33	sat = 2 unsat = 0 unknown = 0	ok = 33 error = 0	ok = 2 error = 0
searchAVL	33	sat = 2 unsat = 0 unknown = 0	ok = 33 error = 0	ok = 2 error = 0
binSearch	30	sat = 1 unsat = 0 unknown = 0	ok = 19 error = 11	ok = 1 error = 0
dutchNationalFlag	7	sat = 1 unsat = 0 unknown = 0	ok = 7 error = 0	ok = 1 error = 0
fill	39	sat = 1 unsat = 0 unknown = 0	ok = 39 error = 0	ok = 1 error = 0
insertArray	27	sat = 1 unsat = 1 unknown = 0	ok = 27 error = 0	ok = 1 error = 0
linearSearch	39	sat = 2 unsat = 0 unknown = 0	ok = 39 error = 0	ok = 2 error = 0
qSortMod	13	sat = 1 unsat = 0 unknown = 0	ok = 13 error = 0	ok = 1 error = 0
selSort	13	sat = 1 unsat = 0 unknown = 0	ok = 10 error = 3	ok = 1 error = 0
insSort	13	sat = 1 unsat = 0 unknown = 0	ok = 13 error = 0	ok = 1 error = 0

Figura 6.3: Resultados obtenidos con tamaños 2, 1, 3 y profundidad 1.

Función	blackBox	whiteBox	blackBoxResul	whiteBoxResul
insertList	80	sat = 4 unsat = 0 unknown = 0	ok = 55 error = 25	ok = 2 error = 2
deleteList	80	sat = 6 unsat = 0 unknown = 0	ok = 80 error = 0	ok = 6 error = 0
searchLLRB	205	sat = 6 unsat = 0 unknown = 0	ok = 205 error = 0	ok = 6 error = 0
insertLeftist	234	sat = 4 unsat = 30 unknown = 0	ok = 158 error = 76	ok = 3 error = 1
unionLeftist	1500	sat = 6 unsat = 28 unknown = 0	ok = 948 error = 552	ok = 6 error = 0
insertBST	317	sat = 6 unsat = 0 unknown = 0	ok = 317 error = 0	ok = 6 error = 0
searchBST	317	sat = 6 unsat = 0 unknown = 0	ok = 317 error = 0	ok = 6 error = 0
insertAVL	180	sat = 1 unsat = 10198 unknown = 107	ok = 180 error = 0	ok = 1 error = 0
searchAVL	180	sat = 6 unsat = 0 unknown = 0	ok = 180 error = 0	ok = 6 error = 0
binSearch	105	sat = 3 unsat = 0 unknown = 0	ok = 55 error = 50	ok = 2 error = 1
dutchNationalFlag	7	sat = 3 unsat = 1 unknown = 0	ok = 7 error = 0	ok = 3 error = 0
fill	155	sat = 2 unsat = 0 unknown = 0	ok = 155 error = 0	ok = 2 error = 0
insertArray	125	sat = 2 unsat = 2 unknown = 0	ok = 125 error = 0	ok = 2 error = 0
linearSearch	155	sat = 4 unsat = 0 unknown = 0	ok = 155 error = 0	ok = 4 error = 0
qSortMod	31	sat = 3 unsat = 2 unknown = 0	ok = 31 error = 0	ok = 3 error = 0
selSort	31	sat = 2 unsat = 2 unknown = 0	ok = 21 error = 10	ok = 2 error = 0
insSort	31	sat = 3 unsat = 4 unknown = 0	ok = 31 error = 0	ok = 3 error = 0

Figura 6.4: Resultados obtenidos con tamaños 2, 1, 5 y profundidad 2.

resultados obtenido se pueden ver en la figura 6.5. Como era de esperar, al aumentar el rango de todos los tamaños, obtenemos más casos de prueba para cada una. Los casos de caja blanca, crecen respecto al anterior pero sin gran diferencia a excepción de las funciones *insertLeftist* y *unionLeftist* que generan 20202 caminos cada una, aunque de estos, tan solo 4 y 3 son satisfactibles, respectivamente. En cuanto a los casos de caja negra, la mayoría de estas funciones generan 1500. Esto es debido al tamaño máximo fijado para estas pruebas, por lo que realmente, se podrían generar más de estos. Por último, si hacemos referencia a la función *insertAVL*, podemos comprobar que nuestro sistema no ha sido capaz de generar casos de caja blanca, ni por lo tanto, probar el programa con estos. Esto es debido a que la herramienta AST2Z3 falla a la hora de generar caminos con profundidad 3 por la complejidad que estos presentan.

Por último, otro resultado que hemos obtenido gracias a este sistema es el problema del uso de cuantificadores en las fórmulas. En un comienzo, la mayoría de las precondiciones de las funciones, como por ejemplo *insertArray*, hacían uso de los cuantificadores. Esto provocaba que la generación de caja negra devolviera como resultado 1 o ningún caso de prueba en este caso, ya que el resolutor fallaba al comprobar estas restricciones. Es por ello que se decidió transformar estas definiciones a funciones recursivas que sustituyeran el uso de los cuantificadores por la recursividad. De esta manera, se consigue pasar de generar estos casos, a los mostrados en las figuras 6.3, 6.4 y 6.5.

6.3. Capacidad del sistema para la detección de errores

Como hemos podido observar en la sección anterior, el sistema es capaz de detectar errores bien en el código, o bien en su especificación formal. En las figuras 6.3, 6.4 y 6.5 podemos comprobar esto para las funciones *insertList*, *insertLeftist*, *unionLeftist*, *binSearch* y *selSort*. Estas CLIRS fueron realizadas en un primer momento a mano por lo que simularían perfectamente un caso real de prueba del sistema. Además, la capacidad del sistema para informar sobre cuáles son los casos que han fallado, son los que nos permitieron averiguar dónde se encontraban los fallos de estas. Las funciones *binSearch* y *selSort* presentaban problemas en su código. Las funciones *unionLeftist* y *insertLeftist* presentaba problemas en su especificación al definir incorrectamente el predicado *isHeap* para establecer que un árbol es un montículo. Por último, la función *insertList* presentaba problemas al definir el predicado *sortedList* para indicar que la lista se trata de una lista ordenada ascendentemente. Esto provocaba que la función *deleteList* generara menos casos de caja negra de los que debería generar, ya que su precondición hace uso de este predicado. Tras solucionar estos errores, se decide ejecutar de nuevo nuestro sistema para validar estos cambios. Los tamaños con los que se ejecutan serán los mismos que los de la figura 6.4, y los resultados obtenidos se muestran en la figura 6.6.

En este caso, no se han encontrado errores en las funciones para los casos

probados, pero debido a que este sistema es una herramienta de *testing*, que no hayamos encontrado errores para las funciones dados estos, no quiere decir que esta no los presente. Esto lo encontramos en algunos de los casos de caja blanca mostrados anteriormente, que no presentan errores, pero con los de caja negra detectamos que si los hay.

Función	blackBox	whiteBox	blackBoxResul	whiteBoxResul
insertList	744	sat = 6 unsat = 0 unknown = 0	ok = 568 error = 176	ok = 4 error = 2
deleteList	744	sat = 9 unsat = 0 unknown = 0	ok = 744 error = 0	ok = 9 error = 0
searchLLRB	1500	sat = 14 unsat = 0 unknown = 0	ok = 1500 error = 0	ok = 14 error = 0
insertLeftist	1500	sat = 4 unsat = 20149 unknown = 49	ok = 1172 error = 328	ok = 2 error = 2
unionLeftist	1500	sat = 3 unsat = 19973 unknown = 226	ok = 1224 error = 276	ok = 3 error = 0
insertBST	1500	sat = 14 unsat = 0 unknown = 0	ok = 1500 error = 0	ok = 14 error = 0
searchBST	1500	sat = 14 unsat = 0 unknown = 0	ok = 1500 error = 0	ok = 14 error = 0
insertAVL	1500	-	ok = 1500 error = 0	-
searchAVL	1500	sat = 14 unsat = 0 unknown = 0	ok = 1500 error = 0	ok = 14 error = 0
binSearch	1320	sat = 7 unsat = 0 unknown = 0	ok = 494 error = 826	ok = 3 error = 4
dutchNationalFlag	15	sat = 7 unsat = 6 unknown = 0	ok = 15 error = 0	ok = 7 error = 0
fill	1500	sat = 3 unsat = 0 unknown = 0	ok = 1500 error = 0	ok = 3 error = 0
insertArray	1500	sat = 4 unsat = 2 unknown = 0	ok = 1500 error = 0	ok = 4 error = 0
linearSearch	1500	sat = 6 unsat = 0 unknown = 0	ok = 1500 error = 0	ok = 6 error = 0
qSortMod	585	sat = 9 unsat = 2540 unknown = 0	ok = 585 error = 0	ok = 9 error = 0
selSort	585	sat = 4 unsat = 53 unknown = 0	ok = 165 error = 420	ok = 3 error = 1
insSort	585	sat = 9 unsat = 202 unknown = 0	ok = 585 error = 0	ok = 9 error = 0

Figura 6.5: Resultados obtenidos con tamaños 3, 1, 8 y profundidad 3.

Función	blackBox	whiteBox	blackBoxResul	whiteBoxResul
insertList	1320	sat = 6 unsat = 0 unknown = 0	ok = 1320 error = 0	ok = 6 error = 0
deleteList	1320	sat = 9 unsat = 0 unknown = 0	ok = 1320 error = 0	ok = 9 error = 0
insertLeftist	1500	sat = 4 unsat = 20149 unknown = 49	ok = 1500 error = 0	ok = 4 error = 0
unionLeftist	1500	sat = 3 unsat = 19987 unknown = 212	ok = 1500 error = 0	ok = 3 error = 0
binSearch	1320	sat = 7 unsat = 0 unknown = 0	ok = 1320 error = 0	ok = 7 error = 0
selSort	585	sat = 10 unsat = 47 unknown = 49	ok = 585 error = 0	ok = 10 error = 0

Figura 6.6: Resultados obtenidos tras la corrección de las funciones.

Capítulo 7

Conclusiones

Castellano

Llegados a este punto, podemos dar por concluido nuestro trabajo y afirmar que los objetivos que se marcaron al comienzo de este, en el capítulo 1, han sido cumplidos.

Como primer objetivo, nos planteamos automatizar la generación de casos de caja negra a partir de la precondición dada como un aserto en el lenguaje SMT-LIB. Con un programa Python y la interacción con el resolutor SMT, Z3, podemos a partir de la precondición del programa y el rango de los valores de los elementos que las estructuras de datos pueden tomar establecidos por el usuario, en forma de aserto, generar todos los posibles casos que cumplan estas restricciones hasta llegar a un punto en el que se hayan generado todos los posibles o, simplemente, el resolutor falle debido a la complejidad de estas. Además, podemos transformar estos casos generados a un formato Haskell que será el esperado por sus clases *Show* y *Read* para posteriormente probar el programa con estos.

En el segundo objetivo, nos planteamos automatizar la generación de casos de caja blanca. Con un programa Python y la interacción con Z3, conseguimos a partir de los caminos del programa, la precondición y las ya mencionadas restricciones de tamaño impuestas por el usuario, todo ello dado en forma de aserto en el lenguaje SMT-LIB, generar estos casos. Además, podemos transformar cada uno de estos al formato que las clases *Read* y *Show* de Haskell los esperan para interaccionar con estos. Por último, obtenemos un análisis de todos los caminos que hemos probado, obteniendo como resultado cuantos de ellos eran satisfactibles, cuantos insatisfactibles, y cuantos no ha sido capaz el resolutor de determinar su satisfactibilidad.

Como último objetivo, nos planteamos automatizar la ejecución y prueba del programa a partir de cualquiera de los casos de prueba anteriormente mencionados. A partir de un programa Haskell, y un programa Python que interacciona con Z3, conseguimos a través del primero, leer los casos de prueba, analizarlos sintácticamente y ejecutar el programa, definido como una función Haskell ejecutable, con

los casos de prueba. Con esto, obtendremos unos resultados que se le pasarán al programa Python junto a la postcondición del programa y, será este, el encargado de evaluar estos resultados con la postcondicion. Finalmente, esta herramienta, devolverá el análisis obtenido de estos, mostrando al usuario si el programa ha pasado la prueba para todos los casos, y en caso contrario, informará de cuáles han fallado y qué resultado se ha obtenido.

En resumen, hemos conseguido, tal como decíamos en el capítulo de introducción, un sistema integrado de *testing* con el que los usuarios pueden realizar cientos de casos de prueba de forma totalmente automática. La clave de este resultado es que los programas a probar vienen provistos de una especificación formal. Además, gracias a este trabajo, hemos podido comprobar algunos de los limites que el resolutor Z3 presenta, como por ejemplo, el uso del cuantificador universal $\forall$ en restricciones complejas.

Inglés

Reached this point, we can conclude our work and affirm that the objetives which were set at the beggining, in chapter 1, have been accomplished.

As a first objective, we plan to automate the generation of black box cases based on the precondition given as an assert in the SMTLIB language. With a Python program and the interacction with SMT solver, Z3, we can from, the precondition of the program and the range of the values of the elements which the data structures can taken as an assert by the user, generating all the possible cases which accomplish the previous asserts, until reaching a point at which all possible cases have been generated or simply the resolver fails due to the complexity of these. As well as, thanks to this program we can transform these generated cases into a Haskell format which will be the one expected by the *Show* y *Read* classes, to later testing the program with them.

In the second objective, we plan to automate the generation of white box cases. With a Python program and the interaction with Z3, we get from the program paths, the precondition and the size restrictions established by the user, all of this given as assert in the SMTLIB language, to generate these cases. Moreover, we can transform each of these into the format that Haskell's *Read* and *Show* classes expected to interact with. Finally, we obtain an analysis of all the paths which we have tried, determining how many of them were satisfiable, how many unsatisfiable, and how many the solver was not able to evaluate.

The last objective, we plan to automate the execution and testing of the program from any of the previous test cases. Starting from a Haskell program, and a Python program which interacts with Z3, thanks to the first one, we can read the test cases, analize them syntactically and run the program defined as an executable Haskell function with test cases. Thanks to this, we will obtain some results which will be passed to the Python program with the program's postcondition and it will evaluate these results with the postcondition. Finally, this tool, will return the analysis obtained from these, showing to the user if the program has passed the test for all cases, and on the contrary case, it will report which ones have failed and what result has been obtained.

To sum up, we have achieved, as we said in the introduction chapter, an integrated *testing* system with which users can carry out hundreds of test cases automatically. The key to this result is that the programs to be tested come with a formal specification. Besides thanks to this work, we have been able to check some of the limits that the Z3 solver presents, as for example, the use of the universal quantifier $\forall$ in complex constraints.

Apéndice A

Programas CLIR

En esta sección se muestran algunos de los códigos CLIR a los cuáles se hace referencia en este trabajo.

A.1. Arrays

A.1.1. binSearch.clir

```
(verification-unit "binSearch"
       :sources "(((:lang :unknown) (:module :unknown)))"
       :uses "(:ir)"
       :documentation "Binary Search")

(define binSearch ((x Int)  (v (Array Int))) ((p Int))
(declare (assertion
  (precd (@ sortedArr v (the Int 0) (@ len v))
    )
  (postcd
    (and (@ <= (the Int 0) p)
      (@ <= p (@ len v))
      (forall ((j Int))
        (-> (@ <= (the Int 0) j)
          (@ < j p)
          (@ < (@ get_array v j) x)) )
      (forall ((j Int))
        (-> (@ <= p j)
          (@ < j (@ len v))
          (@ <= x (@ get_array v j))) )
    ))))
(letfun (
  (bin ((a Int) (b Int) (x Int) (v (Array Int))) ((p2 Int))
    (let ((b1 Bool)) (@ > a b)
      (case b1 (
        ((@@ True) a)
        ((@@ False)
          (let ((m Int)) (@ + a b)
            (let ((n Int)) (@ div m (the Int 2))
            (let ((y Int)) (@ get_array v n)
              (let ((b2 Bool)) (@ > x y)
                (case b2 (
                  ((@@ True)
                    (let ((m2 Int)) (@ + n (the Int 1))
                      (@ bin m2 b x v) ))
```

```
                  ((@@ False)
                    (let ((m3 Int)) (@ - n (the Int 1))
                      (@ bin a m3 x v))))))))))))))
)
(let ((a Int)) (the Int 0)
  (let ((l Int)) (@ len v)
    (let ((b Int)) (@ - l (the Int 1))
      (@ bin a b x v))))))
```

A.1.2. insertArray.clir

```
(verification-unit "insert"
        :sources "(((:lang :erlang)  (:module /insert.erl)))"
        :uses "(:ir)"
        :documentation "Insert in a sorted array")

(define insertArray ((x Int) (m Int) (a (Array Int))) ((res (Array Int)))
  (declare (assertion
    (precd (and (@ <= (the Int 0) m) (@ < m (@ len a))
                (@ sortedArr a (the Int 0) (@ - m (the Int 1))))
    )
    (postcd (@ sortedArr res (the Int 0) m)
    ))
  )
  (letfun (
    (f2 ((x Int) (m Int) (i Int) (a (Array Int))) ((res2 (Array Int)))
      (let ((b1 Bool)) (@ >= i (the Int 0))
        (case b1 (
          ((@@ False)
            (@ f4 x m i a))
          ((@@ True)
            (let ((e Int)) (@ get_array a i)
              (let ((b2 Bool)) (@ < x e)
                (case b2 (
                  ((@@ True)
                    (let ((u Int)) (@ get_array a i)
                      (let ((i2 Int)) (@ + i (the Int 1))
                        (let ((ap (Array Int))) (@ set_array a i2 u)
                          (let ((i3 Int)) (@ - i (the Int 1))
                            (@ f2 x m i3 ap))))))
                  ((@@ False)
                    (@ f4 x m i a)))))))))))

    (f4 ((x Int) (m Int) (i Int) (a (Array Int))) ((res4 (Array Int)))
      (let ((i2 Int)) (@ + i (the Int 1))
        (let ((ap (Array Int))) (@ set_array a i2 x)
          ap)))

  )
  (let ((i Int)) (@ - m (the Int 1))
    (@ f2 x m i a))))
```

A.1.3. linearSearch.clir

```
(verification-unit "LinearSearch"
        :sources "(((:lang :handmade-clir) (:module :self)))"
        :uses "(:ir)"
        :documentation "Linear search")

(define linearSearch ((e Int)(v (Array Int))) ((res Int))
  (declare (assertion
```

```
    (precd
      True)
    (postcd
        (and (forall ((j Int))
                     (-> (@ <= (the Int 0) j)
                         (@ < j res)
                         (not (@ = (@ get_array v j) e))))
             (-> (@ < res (@ len v))(@ = (@ get_array v res) e))) )
  ))
  (letfun (
    (f ((i Int) (e Int) (v (Array Int))) ((res1 Int))
      (let ((l Int)) (@ len v)
        (let ((b1 Bool)) (@ < i l)
          (case b1 (
            ((@@ False)
              i)
            ((@@ True)
              (let ((vi Int)) (@ get_array v i)
           (let ((b2 Bool)) (@ == vi e)
                  (case b2 (
                 ((@@ True)
                      i)
                  ((@@ False)
                      (let ((i1 Int)) (@ + i (the Int 1))
                        (@ f i1 e v))))))))))))
    )
  (let ((i Int)) (the Int 0)
    (@ f i e v))))
```

A.1.4. selSort.clir

```
(verification-unit "selSort"
        :sources "(((:lang :unknown) (:module :unknown)))"
        :uses "(:ir)"
        :documentation "Selection Sort")

(define selSort ((v (Array Int))) ((vres0 (Array Int)))
(declare (assertion
 (precd
  True)
 (postcd
  (@ sortedArr vres0 (the Int 0) (let ((l1 Int)) (@ len v) (@ - l1 (the Int 1)))))))

(letfun (

  (selSort_wh1 ((i Int) (n Int) (v (Array Int))) ((vres1 Int) (vres2 (Array Int)))
   (let ((v_3_25 Int)) (@ - n (the Int 1))
    (let ((v_4_26 bool)) (@ < i v_3_25)
     (case v_4_26 (
      ((@@ True)
       (let ((min_27 Int)) i
        (let ((j_28 Int)) (@ + i (the Int 1))
         (let ((j_29 Int) (min_30 Int) (v_31 (Array Int))) (@ selSort_wh1_wh1 n j_28 v min_27)
          (let ((tmp_32 Int)) (@ get_array v_31 i)
           (let ((v_8_33 Int)) (@ get_array v_31 min_30)
            (let ((v_34 (Array Int))) (@ set_array v_31 i v_8_33)
             (let ((v_35 (Array Int))) (@ set_array v_34 min_30 tmp_32)
              (let ((i_36 Int)) (@ + i (the Int 1))
               (@ selSort_wh1 i_36 n v_35)))))))))
      ((@@ False)
       (tuple i v)))))))

  (selSort_wh1_wh1 ((n Int) (j Int) (v (Array Int)) (min Int)) ((vres3 Int)
                   (vres1 Int) (vres2 (Array Int)))
   (let ((v_13_37 bool)) (@ < j n)
    (case v_13_37 (
```

```
      ((@@ True)
       (let ((v_14_38 Int)) (@ get_array v j)
        (let ((v_15_39 Int)) (@ get_array v min)
         (let ((v_16_40 bool)) (@ < v_14_38 v_15_39)
          (let ((min_41 Int)) (@ selSort_wh1_wh1_if1 v_16_40 j min)
           (let ((j_42 Int)) (@ + j (the Int 1))
            (@ selSort_wh1_wh1 n j_42 v min_41)))))))
      ((@@ False)
       (tuple j min v))))))

  (selSort_wh1_wh1_if1 ((v_16 bool) (j Int) (min Int)) ((vres4 Int))
   (case v_16 (
    ((@@ True)
     (let ((min_43 Int)) j
      min_43))
    ((@@ False)
     min)))))

(let ((n_21 Int)) (@ len v)
 (let ((i_22 Int)) (the Int 0)
  (let ((i_23 Int) (v_24 (Array Int))) (@ selSort_wh1 i_22 n_21 v)
   v_24)))))
```

A.2. AVL

A.2.1. insertAVL.clir

```
(verification-unit "insertAVL"
                :sources "(((:lang :clir) (:module :self)))"
                :uses "(:ir)"
                :documentation "Insertion on an AVL")

(define insertAVL ((x Int) (t (AVL Int))) ((res (AVL Int)))
  (declare (assertion
    (precd
      (@ isAVL t))
    (postcd
      (and (@ isAVL res)
        (@ = (@ setA res)
          (let ((s1 (Set Int))) (@ setA t)
            (let ((s2 (Set Int))) (@ unit x)
              (@ union s1 s2)))))))))

  (letfun (
    (ins ((x Int) (t (AVL Int))) ((res (AVL Int)))
      (case t (
        ((@@ LeafA)
          (@@ NodeA x (the Int 1) t t))
        ((@@ NodeA y h l r)
          (let ((b1 Bool)) (@ < x y)
            (case b1 (
              ((@@ True)
                (let ((ia (AVL Int))) (@ ins x l)
                  (@ equil ia y r)))
              ((@@ False)
                (let ((b2 Bool)) (@ > x y)
                  (case b2 (
                    ((@@ True)
                      (let ((ia (AVL Int))) (@ ins x r)
                        (@ equil l y ia)))
                    ((@@ False)
                      t)))))))))))
```

```
(equil ((l (AVL Int)) (x Int) (r (AVL Int))) ((res (AVL Int)))
  (let ((hl Int)) (@ height l)
    (let ((hr Int)) (@ height r)
      (let ((hr2 Int)) (@ + hr (the Int 2))
        (let ((b Bool)) (@ == hl hr2)
          (case b (
            ((@@ True)
              (@ leftBalance l x r))
            ((@@ False)
              (let ((hl2 Int)) (@ + hl (the Int 2))
                (let ((b2 Bool)) (@ == hr hl2)
                  (case b2 (
                    ((@@ True)
                      (@ rightBalance l x r))
                    ((@@ False)
                      (@ compose l x r)))))))))))))

(compose ((l (AVL Int)) (x Int) (r (AVL Int))) ((res (AVL Int)))
  (let ((hl Int)) (@ height l)
    (let ((hr Int)) (@ height r)
      (let ((mx Int)) (@ max hl hr)
        (let ((h Int)) (@ + (the Int 1) mx)
          (@@ NodeA x h l r))))))

(height ((t (AVL Int))) ((hres Int))
  (case t (
    ((@@ LeafA)
      (the Int 0))
    ((@@ NodeA x h l r)
      h))))

(leftBalance ((l (AVL Int)) (x Int) (r (AVL Int))) ((res (AVL Int)))
  (case l (
    ((@@ NodeA lx lh ll lr)
      (let ((llh Int)) (@ height ll)
        (let ((lrh Int)) (@ height lr)
          (let ((b Bool)) (@ >= llh lrh)
            (case b (
              ((@@ True)
                (let ((tx (AVL Int))) (@ compose lr x r)
                  (@ compose ll lx tx)))
              ((@@ False)
                (case lr (
                   ((@@ NodeA lrx lrh2 lrl lrr)
                      (let ((cp1 (AVL Int))) (@ compose ll lx lrl)
                        (let ((cp2 (AVL Int))) (@ compose lrr x r)
                          (@ compose cp1 lrx cp2))))))))))))))

(rightBalance ((l (AVL Int)) (x Int) (r (AVL Int))) ((res (AVL Int)))
  (case r (
    ((@@ NodeA rx rh rl rr)
      (let ((rlh Int)) (@ height rl)
        (let ((rrh Int)) (@ height rr)
          (let ((b Bool)) (@ >= rrh rlh)
            (case b (
              ((@@ True)
                (let ((tx (AVL Int))) (@ compose l x rl)
                  (@ compose tx rx rr)))
              ((@@ False)
                (case rl (
                  ((@@ NodeA rlx rlh2 rll rlr)
                    (let ((cp1 (AVL Int))) (@ compose l x rll)
                      (let ((cp2 (AVL Int))) (@ compose rlr rx rr)
                        (@ compose cp1 rlx cp2))))))))))))))
)
(@ ins x t)))
```

A.3. BST

A.3.1. insertBST.clir

```
(verification-unit "insertBST"
                :sources "(((:lang :clir) (:module :self)))"
                :uses "(:ir)"
                :documentation "Insert on an BST")

(define insertBST ((x Int) (t (Tree Int))) ((res (Tree Int)))
  (declare (assertion
    (precd
      (@ isBST t))
    (postcd
      (and (@ isBST res)
        (@ = (@ set res)
          (let ((s1 (Set Int))) (@ set t)
            (let ((s2 (Set Int))) (@ unit x)
              (@ union s1 s2)))))))))

  (letfun (
    (f1 ((x Int) (t (Tree Int))) ((res (Tree Int)))
     (case t (
    ((@@ Leaf)
          (let ((empty_leaf (Tree Int))) (@@ Leaf)
            (@@ Node x empty_leaf empty_leaf)))
        ((@@ Node y l r)
          (let ((b Bool)) (@ < x y)
            (case b (
              ((@@ True)
                (let ((z (Tree Int))) (@ f1 x l)
                  (@@ Node y z r)))
              ((@@ False)
                (let ((b1 Bool)) (@ > x y)
                  (case b1 (
                    ((@@ True)
                      (let ((z (Tree Int))) (@ f1 x r)
                        (@@ Node y l z)))
                    ((@@ False)
                      t)))))))))))
  )
  (@ f1 x t)))
```

A.4. Listas

A.4.1. insertList.clir

```
(verification-unit "insertList"
                :sources "(((:lang :clir) (:module :self)))"
                :uses "(:ir)"
                :documentation "Insertion in a sorted List")

(define insertList ((x Int) (l (Lst Int))) ((res (Lst Int)))
  (declare (assertion
    (precd
      (@ sortedList l))
    (postcd
      (and (@ sortedList res)
        (@ = (@ multiset res)
```

```
          (let ((m2 (Multiset Int))) (@ multiset l)
            (let ((m3 (Multiset Int))) (@ unitMs x)
              (@ mset-union m2 m3))))))))

  (letfun (
    (f1 ((y Int) (ys (Lst Int))) ((ls (Lst Int)))
      (case ys (
        ((@@ Nil)
          (let ((vacia (Lst Int))) (@@ Nil)
         (@@ Cons y vacia)))
        ((@@ Cons z zs)
          (let ((b Bool)) (@ <= y z)
            (case b (
              ((@@ True)
                (@@ Cons y ys))
              ((@@ False)
                (let ((zz (Lst Int))) (@ f1 y zs)
                  (@@ Cons z zz)))))))))

  ) (@ f1 x l)))
```

A.4.2. deleteList.clir

```
(verification-unit "deleteList"
               :sources "(((:lang :clir) (:module :self)))"
               :uses "(:ir)"
               :documentation "Deletion in a sorted List")

(define deleteList ((x Int) (l (Lst Int))) ((res (Lst Int)))
  (declare (assertion
    (precd
      (@ sortedList l))
    (postcd
      (@ = (@ multiset res) (let ((xs (Multiset Int))) (@ unitMs x)
                              (let ((ls (Multiset Int))) (@ multiset l)
                                 (@ mset-diff ls xs))))
    )))

  (letfun (
    (delete ((x Int) (l (Lst Int))) ((res2 (Lst Int)))
      (case l (
        ((@@ Nil) (@@ Nil))
        ((@@ Cons y ys)
          (let ((b1 Bool)) (@ < x y)
            (case b1 (
              ((@@ True)
                l)
              ((@@ False)
                (let ((b2 Bool)) (@ > x y)
                  (case b2 (
                    ((@@ True)
                      (let ((lp (Lst Int))) (@ delete x ys)
                        (@@ Cons y lp)))
                    ((@@ False) ys)))))))))))

  )
  (@ delete x l)))
```

A.5. LLRB

A.5.1. searchLLRB.clir

```
(verification-unit "searchLLRB"
            :sources "(((:lang :clir) (:module :self)))"
            :uses "(:ir)"
            :documentation "Search on an LLRB")

(define searchLLRB ((x Int) (t (LLRB Int))) ((res Bool))
  (declare (assertion
    (precd
      (@ isLLRB t))
    (postcd
      (@ = res (let ((s (Set Int))) (@ setL t)
                 (let ((b Bool)) (@ belongs x s) b)))
    )))

  (letfun (
    (search ((x Int) (t (LLRB Int))) ((res Bool))
      (case t (
        ((@@ LeafL)
          (the Bool False))
        ((@@ NodeL y c l r)
          (let ((b1 Bool)) (@ < x y)
            (case b1 (
                ((@@ True) (@ search x l))
                ((@@ False) (let ((b2 Bool)) (@ > x y)
                    (case b2 (
                        ((@@ True) (@ search x r))
                        ((@@ False) (the Bool True))))))))))))

  )
  (@ search x t)))
```

A.6. Montículo

A.6.1. insertLeftist.clir

```
(verification-unit "insertLeftist"
                :sources "(((:lang :clir) (:module :self)))"
                :uses "(:ir)"
                :documentation "Insertion on a leftist heap")

(define insertLeftist ((x Int) (t (Tree Int))) ((res (Tree Int)))
  (declare (assertion
    (precd
        (and (@ isLeftist t)
           (@ isHeap t)))
    (postcd
      (and (@ isLeftist res)
        (@ isHeap res)
        (@ = (@ msetT res)
          (let ((m1 (Multiset Int))) (@ msetT t)
            (let ((m2 (Multiset Int))) (@ unitMs x)
              (@ mset-union m1 m2))))))))

    (letfun (
```

```
        (unionLeftist ((t1 (Tree Int)) (t2 (Tree Int))) ((res (Tree Int)))
          (@ f1 t1 t2))

        (f1 ((t1 (Tree Int)) (t2 (Tree Int))) ((res (Tree Int)))
         (case t1 (
            ((@@ Leaf)
             t2)
            ((@@ Node x1 l1 r1)
               (case t2 (
                  ((@@ Leaf)
                    t1)
                 ((@@ Node x2 l2 r2)
                   (let ((b Bool)) (@ <= x1 x2)
                     (case b (
                       ((@@ True)
                         (let ((ul (Tree Int))) (@ f1 r1 t2)
                           (@ equil x1 l1 ul)))
                       ((@@ False)
                          (let ((ul (Tree Int))) (@ f1 t1 r2)
                            (@ equil x2 l2 ul))))))))))))
        (equil ((x Int) (l (Tree Int)) (r (Tree Int))) ((res (Tree Int)))
          (let ((hl Int)) (@ hmin l)
            (let ((hr Int)) (@ hmin r)
              (let ((b Bool)) (@ >= hl hr)
                (case b (
                  ((@@ True)
                    (@@ Node x l r))
                  ((@@ False)
                    (@@ Node x r l))))))))
        (hmin ((t (Tree Int))) ((h Int))
          (case t (
            ((@@ Leaf)
              (the Int 0))
            ((@@ Node x l r)
              (let ((hl Int)) (@ hmin l)
                (let ((hr Int)) (@ hmin r)
                  (let ((m Int)) (@ min hl hr)
                    (@ + (the Int 1) m))))))))
      ) (let ((h1 (Tree Int))) (@@ Leaf)
          (let ((h2 (Tree Int))) (@@ Leaf)
            (let ((h3 (Tree Int))) (@@ Node x h1 h2)
              (@ unionLeftist h3 t))))))
```

A.6.2. unionLeftist.clir

```
(verification-unit "unionLeftist"
                :sources "(((:lang :clir) (:module :self)))"
                :uses "(:ir)"
                :documentation "Union of leftist heaps")

(define unionLeftist ((t1 (Tree Int)) (t2 (Tree Int))) ((res (Tree Int)))
  (declare (assertion
    (precd
      (and (@ isLeftist t1)
        (@ isHeap t1)
        (@ isLeftist t2)
        (@ isHeap t2)))
    (postcd
      (and (@ isLeftist res)
        (@ isHeap res)
        (@ = (@ msetT res)
          (let ((m1 (Multiset Int))) (@ msetT t1)
            (let ((m2 (Multiset Int))) (@ msetT t2)
              (@ mset-union m1 m2))))))))
    (letfun (
      (f1 ((t1 (Tree Int)) (t2 (Tree Int))) ((res (Tree Int)))
```

```
      (case t1 (
        ((@@ Leaf)
          t2)
        ((@@ Node x1 l1 r1)
          (case t2 (
            ((@@ Leaf)
              t1)
            ((@@ Node x2 l2 r2)
              (let ((b Bool)) (@ <= x1 x2)
                (case b (
                  ((@@ True)
                    (let ((ul (Tree Int))) (@ f1 r1 t2)
                      (@ equil x1 l1 ul)))
                  ((@@ False)
                    (let ((ul (Tree Int))) (@ f1 t1 r2)
                      (@ equil x2 l2 ul))))))))))))
  (equil ((x Int) (l (Tree Int)) (r (Tree Int))) ((res (Tree Int)))
    (let ((hl Int)) (@ hmin l)
      (let ((hr Int)) (@ hmin r)
        (let ((b Bool)) (@ >= hl hr)
          (case b (
            ((@@ True)
              (@@ Node x l r))
            ((@@ False)
              (@@ Node x r l))))))))
  (hmin ((t (Tree Int))) ((h Int))
    (case t (
      ((@@ Leaf)
        (the Int 0))
      ((@@ Node x l r)
        (let ((hl Int)) (@ hmin l)
          (let ((hr Int)) (@ hmin r)
            (let ((m Int)) (@ min hl hr)
                (@ + (the Int 1) m))))))))
) (@ f1 t1 t2)))
```

Bibliografía

[1] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John A. Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil McMinn. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978-2001, 2013.

[2] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825-885. IOS Press, 2009.

[3] Miguel Garrido. Generación de casos de prueba de caja negra mediante restricciones. Trabajo de Fin de Grado. Doble Grado en Ingeniería Informática y Matemáticas. Facultad de Informática. Universidad Complutense de Madrid, Junio 2018.

[4] Javier Sagredo. Generación de casos de prueba de caja blanca mediante restricciones. Trabajo de Fin de Grado. Doble Grado en Ingeniería Informática y Matemáticas. Facultad de Informática. Universidad Complutense de Madrid, Septiembre 2018.

[5] GHC Team. Glasgow Haskell Compiler core Language. `https://ghc.haskell.org/trac/ghc/wiki/Commentary/Compiler/CoreSynType`.

[6] Marta Aracil, Pedro García, and Ricardo Peña. A tool for black-box testing in a multilanguage verication platform. In *Proceedings of XVII Jornadas sobre Programación y Lenguajes, PROLE 2017, Tenerife, Spain, September 2017*, pages 1-15, 2017.

[7] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. *In Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 337-340, 2008.

[8] Manuel Montenegro, Susana Nieva, Ricardo Peña, and Clara Segura. Liquid types for array invariant synthesis. In Deepak D'Souza and K. Narayan Kumar, editors, *Automated Technology for Verication and Analysis - 15th International Symposium, ATVA 2017, Pune, India, October 3-6, 2017, Proceedings, volume 10482 of Lecture Notes in Computer Science*, pages 289-306. Springer, 2017.

[9] Manuel Montenegro, Ricardo Peña, and Jaime Sánchez-Hernández. A generic intermediate representation for verication condition generation. In Moreno Falaschi, editor, *Logic-Based Program Synthesis and Transformation - 25th International Symposium, LOPSTR 2015, Siena, Italy, July 13-15, 2015. Revised Selected Papers*, volume 9527 of Lecture Notes in Computer Science, pages 227-243. Springer, 2015.

[10] Template Meta-programming for Haskell. Tim Sheard and Simon Peyton Jones. 2002. October. `https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/meta-haskell.pdf`