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

GENERACIÓN DE CASOS DE PRUEBA TIPO CAJA NEGRA MEDIANTE RESTRICCIONES

Trabajo de Fin de Grado
Doble Grado en Ingeniería Informática y Matemáticas

Junio 2018

Autor:
Miguel Garrido Canalejas
Director:
Ricardo Peña Marí

Resumen

La plataforma de validación CAVI-ART nos ofrece una representación intermedia de cualquier función escrita en diferentes lenguajes de programación, que incluye su código, su precondición y su postcondición. Sobre dicha función deseamos realizar pruebas de ejecución.

El objetivo de este trabajo reside en crear de manera automática diferentes casos de prueba que cumplan las precondiciones de las funciones que se quieran probar. Para ello, se han estudiado primero los resolutores SMT, en concreto Z3, se han programado en tal resolutor todas las funciones y tipos que pueden interesarnos para las precondiciones, y por último se ha creado, en Haskell, un generador de restricciones que analice las precondiciones de un programa en la IR, gracias a su representación en forma de árbol abstracto, y genere un archivo de restricciones procesable por Z3 con el que obtener los casos de prueba.

Palabras clave

Pruebas de ejecución, resolutores SMT, generador de restricciones, resolución de restricciones, estructuras de datos.

Abstract

The validation platform CAVI-ART offers us an intermediate representation for any function written in different languages, including its precondition, its code and its postcondition. We want to do testing of those functions.

The goal of this work consists of automatically creating different test cases satisfying the preconditions of the functions wanted to be tested. In order to do this, SMT solvers have been studied, specifically Z3, every function and datatype we could be interested in for the preconditions have been programmed, and, finally, a constraints generator has been created in Haskell. It analyzes the preconditions of an IR program, thanks to its abstract syntax tree representation, and generates a constraints file processable by Z3 from which we obtain the test cases.

Keywords

Testing, SMT solvers, constraints generator, constraints solving, data structures.

Índice general

Capítulo 1

Introducción

Castellano

Terminada la creación de un cierto programa, aparece la tarea de hacer pruebas de ejecución. Esta tarea consiste en comprobar si el programa se comporta como debería, mediante la ejecución de una serie de casos de prueba, y la comparación de los resultados obtenidos para dichos casos con las respectivas respuestas esperadas. Normalmente, especificar y preparar manualmente estos casos de prueba resulta tanto costoso como poco fiable. Es costoso porque requiere idear casos que se ajusten a nuestro código, realizar numerosas ejecuciones, determinar cuál debe ser la respuesta correcta en cada caso y comprobar cada uno de los resultados. Por otro lado, la poca fiabilidad viene derivada de no considerar todos los casos necesarios, pues, ¿y si los casos elegidos son pocos o demasiado sencillos y dejan caminos sin explorar? Lo que creemos que está funcionando correctamente resulta tener errores que no hemos detectado. Automatizar todo este proceso parece entonces de sumo interés.

Podemos clasificar las estrategias de pruebas en dos grandes grupos: de caja negra y de caja blanca [1]. La primera de ellas consiste en la generación de casos de prueba basándose exclusivamente en la especificación del programa, es decir, en las precondiciones y postcondiciones, sin importar el código de la función. De este modo, los casos de prueba deberán verificar la precondición, y el resultado obtenido tras la ejecución deberá satisfacer la postcondición. Por su parte, las pruebas de caja blanca tienen en cuenta el código del programa y su objetivo es generar casos que ejecuten los diferentes caminos que puede seguir la ejecución dentro del código.

Enmarcado dentro del proyecto CAVI-ART, explicado más adelante, este trabajo pretende automatizar las pruebas de caja negra mediante la generación de casos de prueba que cumplan las precondiciones requeridas por las distintas funciones del programa. Dicho proyecto está enfocado fundamentalmente a la verificación de programas, y por tanto cada función está provista de precondición y postcondición. En trabajos previos sobre esta plataforma [2], se ha desarrollado una herramienta que transforma un aserto cualquiera en una función ejecutable, de tal forma que se

le dan a dicha función los valores de las variables que intervienen en el aserto, y la función devuelve un valor booleano que indica si tales valores satisfacen o no el aserto.

Hasta ahora, las pruebas de caja negra realizadas en esta plataforma siguen un método de prueba y error: se generan casos de prueba de menor a mayor tamaño, de un modo exhaustivo para cada tamaño, y luego se comprueba, ejecutando el aserto precondición, cuáles cumplen la misma. Los que no la cumplen, se descartan. Este enfoque implica un importante coste, pues se están generando una gran cantidad de casos de prueba para luego quedarse con un porcentaje reducido de ellos.

El objetivo de este trabajo es generar automáticamente casos de prueba que ya satisfagan la precondición, para evitar generar casos que luego son descartados. Las precondiciones que nos interesan son relativamente complejas y pueden involucrar estructuras de datos con invariantes sofisticados. Por ejemplo, árboles, AVL, árboles rojinegros, montículos de diferentes clases, listas y arrays ordenados, etc. Generar directamente estructuras que satisfagan estos invariantes, no parece una tarea sencilla. Por esa razón hemos acotado a unos pocos los tipos de datos que pretendemos tratar, pero cubren sin embargo la mayor parte de las estructuras de datos utilizadas en programación.

El otro objetivo es investigar la potencia de los resolutores SMT [4] (*Satisfiability Modulo Theories*) para realizar esta tarea. Los resolutores SMT permiten resolver problemas de satisfactibilidad. A partir de una serie de restricciones, son capaces de dar valor a las variables que participen en ellas, de forma que tales valores satisfagan las condiciones impuestas por tales restricciones. Uno de los más potentes que podemos encontrarnos en el mercado es Z3[1] [5], que es nuestra elección para desarrollar este trabajo.

El objetivo global puede entonces resumirse en realizar una transformación de un aserto-precondición, dado en la lógica de predicados, a un conjunto de restricciones procesables por Z3, de forma que si estas son satisfactibles, su resolución nos proporcione uno o varios casos de prueba que satisfacen la precondición.

Por un lado, deberemos programar en Z3 todos los tipos y predicados auxiliares que puedan aparecer en las precondiciones, y por otro deberemos desarrollar un generador de restricciones que tome la precondición y la transforme en un conjunto de restricciones para Z3. Para este último generador el lenguaje elegido es Haskell[2]. Dado que el objetivo es comprobar la factibilidad del enfoque, la salida será inicialmente un fichero de restricciones en formato SMT-LIB procesable por Z3. En un futuro, se debería conectar directamente el programa Haskell con Z3 a través de una API que ya existe.

Estructuramos el trabajo en los siguientes bloques: un primer bloque especificará en el lenguaje de Z3 todos los tipos y funciones necesarios; una segunda parte explicará la estrategia que hemos seguido para generar los casos de prueba; y una última parte codificará en Haskell la traducción de asertos en restricciones y

[1] https://rise4fun.com/Z3/tutorial/guide
[2] https://www.haskell.org/

se explicará la obtención de los casos de prueba. El Capítulo 3 se dedica al primer bloque, mientras que el Capítulo 4 explica cómo hemos decidido realizar la generación de casos de prueba y cómo se ha implementado la transformación de asertos en restricciones.

Inglés

Once the creation of a program is finished, it arises the task of testing it. This task consists of checking whether the program works as it should, by running a series of test cases and comparing the results obtained for these programs with those expected. Usually, to specify these test cases is expensive and unreliable. It is expensive because it requires thinking cases that fit our code, doing many executions, deciding what the correct answer should be for each case, and checking every results. On the other hand, being unreliable comes from not considering all the needed cases, since the chosen cases might not be enough or they might be very simple, so some paths might remain unexplored. Then, we thought that the program was working correctly but it turns out to have errors we have not detected. To automate all this process seems to be really interesting.

We can classify the test strategies into two main groups [1]: black-box testing and white-box testing. The first one consists of the generation of test cases based exclusively on the program specification, that is to say, in the preconditions and postconditions, but regardless of the function code. Thereby, the test cases must verify the precondition and, the obtained result after the execution has to satisfy the postcondition. On the other hand, white-box testing considers the program code, and its objective is generating cases that execute all the different paths that the execution can take within the code.

In the framework of the CAVI-ART project, explained later, this work aims at automating the black box testing by generating test cases fitting the preconditions required by the different functions of the program. This project is basically focused on program verification, so every function is provided with a precondition and a postcondition. In previous works on this platform [2], a tool has been developed that transforms an assertion into an executable function, so that values of the variables involved in the assertion are given to the function, which returns a boolean value that indicates whether such values satisfy or not the assertion.

Up to now, black box testing made by this platform follows a trial and error method: test cases are generated from smaller to larger ones, in an exhaustive way for each size, and then it is checked, executing the precondition assertion, which ones satisfy it. Those that do not satisfy it are discarded. This approach implies an important cost, because it generates a huge amount of test cases and then keeps a small percentage of them.

The goal of this work is generating automatically test cases satisfying the precondition, avoiding then generating cases that are later discarded. The preconditions we are interested in are relatively complex and may involve data structures with sophisticated invariants. For exaple, binary search trees, AVL trees, Red-Black trees, different kinds of heaps, sorted lists and arrays, etc. Generating structures directly satisfying these invariants does not seems an easy task. Because of that, we have restricted the datatypes to a few ones, covering however most of the data structures used in programming.

The other goal is investigating the power of the SMT (Satisfiability Module Theories) solvers [4] to perform this task. SMT solvers allow to solve satisfiability problems. From a set of constraints, they are able to give values to the variables involved in them so that these values satisfy the conditions imposed by such constraints. One of the most powerfull we can find in the market is Z3[3] [5], which is our choice to carry out this work.

The overall goal can then be summarized as making a transformation from a precondition-assertion, given in predicate logic, to a set of constraints processable by Z3, so if these are satisfiable, its solution provides us one or more test cases that satisfy the precondition.

On the one hand, we must program in Z3 all the types and functions which coul be usefull for the preconditions, and, on the other hand, a constraints generator that takes the precondition and transforms it into constraints. The chosen language for this generator is Haskell[4]. Since the goal is checking the feasibility of this approach, the outcome will be a *smt* file with constraints executable by Z3. In the future, the Haskell program will be directly connected to Z3 through an existing API.

We can structure this work in the following blocks: a first block in which all the necessary functions in the Z3 language are specified; a second part where we design the strategy to generate the test cases; and a last block in which the translation from predicates to constraints is coded in Haskell, and the obtention of test cases is explained. During Chapter 3 we will discuss about such specifications in Z3, meanwhile in Chapter 4 we will explain the way we have decided to deal with test case generation and how the implementation has been carried out.

[3]https://rise4fun.com/Z3/tutorial/guide
[4]https://www.haskell.org/

Capítulo 2

Preliminares

2.1. Proyecto CAVI-ART

La plataforma CAVI-ART [6] [7] (Figura 2.1) es un proyecto destinado a la validación de programas. La clave de este proyecto es lo que se denomina Representación Intermedia (en adelante IR).

A partir de un programa, escrito en cualquier lenguaje, ya sea imperativo (*Java*, *C* o *C++*) o funcional (*Haskell* o *Erlang*), la plataforma cuenta con una serie de herramientas que transforman tal programa en la mencionada IR. Esta IR es un lenguaje funcional, y nos proporciona una representación abstracta del programa, que nos permite trabajar de manera independiente a cuál fuera el lenguaje de programación original. En la Figura 2.2 puede verse su sintaxis abstracta. La IR cuenta además con una representación textual, CLIR (Common Lisp IR).

Una vez transformado el código en la IR, entran en juego las diferentes herramientas con las que cuenta CAVI-ART, basadas en análisis estático y verificación formal, y que nos aportan pruebas de terminación, inferencia de invariantes o pruebas de condiciones de verificación, entre otras. Además, una de las herramientas con las que cuenta, y la que más interesa para el enfoque de este trabajo es el generador de casos de prueba. Desde el año pasado, la IR pasó de ser una simple sintaxis abstracta a constituir de por sí un código ejecutable. Para ello, se lleva a cabo una transformación desde la IR a un código Haskell, abriendo así la posibilidad de construir herramientas de testeo y posibilitar la ejecución de pruebas, y hacerlo todo ello independientemente del lenguaje de programación del que surgiera la IR.

2.2. Sistema actual de generación de casos de prueba

Hasta ahora, las pruebas de caja negra seguían un método que podemos considerar de prueba y error. En primer lugar, se generaban una gran cantidad de

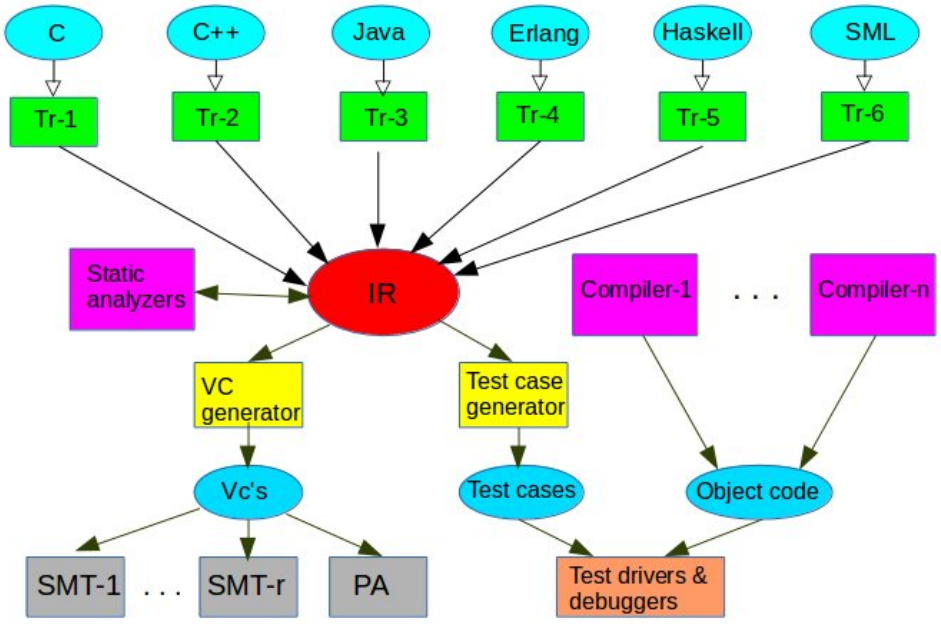


Figura 2.1: Plataforma CAVI-ART

posibles casos de prueba. Tras esto, se filtraban según la precondición, descartando así todos aquellos que no la cumplieran y que, por tanto, no eran válidos para verificar el programa. Este método resulta costoso tanto en tiempo como en recursos de memoria, puesto que estamos generando muchos casos para luego acabar descartando un alto porcentaje de ellos.

El programa actual de generación de casos de caja negra recibe como entrada un programa Haskell que se ajusta a la estructura de la Figura 2.3, con dos funciones booleanas referentes a la precondición y postcondición de la función principal. De este modo, los casos de prueba deben ser valores para los argumentos de la función $x_1 : t_1, \ldots, x_n : t_n$, donde t_i refiere al tipo de la variable x_i, puesto que son la entrada de la función principal. Obviamente, la precondición especificará una serie de restricciones sobre estas variables de entrada. Por ello, los casos de prueba generados son los utilizados en la precondición para ver si se verifica o no. En caso de hacerlo, puede considerarse como un caso de prueba válido y, en caso contrario, es descartado. Los casos válidos se pasan por la función, obteniendo así un resultado final de la misma. Este resultado final, unido a los valores de las variables de entrada, son usados en la postcondición como argumentos para ver si cumplen todos los requisitos de salida necesarios.

La tarea más complicada de este sistema reside en la generación de valores para las variables de entrada de la precondición. Y es que, tales variables pueden ser

a	::=	c	{ constante }
	|	x	{ variable }
be	::=	a	{ expresión atómica }
	|	$f\ \overline{a_i}$	{ aplicación de función/operador primitivo }
	|	$\langle \overline{a_i} \rangle$	{ construcción de tuplas }
	|	$C\ \overline{a_i}$	{ aplicación de constructor }
e	::=	be	{ expresión ligada }
	|	**let** $\langle \overline{x_i :: \tau_i} \rangle = be$ **in** e	{ let secuencial }
	|	**letfun** $\overline{def_i}$ **in** e	{ let recursivo para definición de funciones }
	|	**case** a **of** $\overline{alt_i}[;\ \_ \rightarrow e]$	{ case con rama default opcional }
$tldef$	::=	**define** $\{\psi_1\}\ def\ \{\psi_2\}$	{ definición de función con precondición y postcondición }
def	::=	$f\ (\overline{x_i :: \tau_i}) :: \overline{y_i :: \tau_i} = e$	{ definición de función (las variables de salida tienen nombre) }
alt	::=	$C\ \overline{x_i :: \tau_i} \rightarrow e$	{ rama del case }
τ	::=	α	{ variable de tipo }
	|	$T\ \overline{\tau_i}$	{ aplicación de un constructor de tipo }

Figura 2.2: Sintaxis abstracta de la IR

$$preCD(x_1 : t_1, \ldots, x_n : t_n) : Bool$$
$$fun(x_1 : t_1, \ldots, x_n : t_n) : t$$
$$postCD(x_1 : t_1, \ldots, x_n : t_n, x : t) : Bool$$

Figura 2.3: Estructura del programa Haskell recibido para las pruebas de caja negra

de tipos simples como enteros o booleanos, pero también pueden ser de tipos tan complejos como listas, árboles rojinegros, o aquellos que el usuario desee inventarse. Para conseguir generar casos de prueba de forma correcta, es necesario entonces investigar el tipo de cada una de estas variables, para así saber como asignarles posibles valores. Para ello, se utiliza la herramienta *Template Haskell*. Tal herramienta proporciona la posibilidad de analizar los tipos de cada método, accediendo a su definición para aquellos que son desconocidos (tipos algebraicos definidos en el programa). De este modo, una vez se tienen identificados cada uno de los tipos, el programa de caja negra genera todos los valores posibles hasta un cierto tamaño fijado por el usuario para cada uno de ellos, logrando así todas las asignaciones posibles para las variables que irán pasándose por la precondición, función principal y postcondición.

2.3. La herramienta Z3

El problema de la satisfactibilidad módulo teorías (SMT del término *Satisfiability Modulo Theories* en inglés), es un problema de decisión para fórmulas lógicas de primer orden, con respecto a ciertas teorías subyacentes, como pueden

ser: la teoría de los números enteros, la teoría de los reales, la teoría de los arrays o la de los *bit-vectors*. De este modo, dada una cierta fórmula $\mathcal{F}$, y bajo alguna de las teorías previas, que nos restringen la interpretación de los distintos símbolos que aparecen en nuestra fórmula, el problema consiste en determinar si $\mathcal{F}$ es satisfactible.

En este contexto aparecen lo que se denominan resolutores SMT. Se trata de herramientas encargadas de decidir la satisfactibilidad de una fórmula concreta en su correspondiente teoría. De entre los muchos que pueden encontrarse, uno de los más completos, y el que más se ajusta a nuestras necesidades, es Z3. Se trata de un resolutor SMT creado por *Microsoft Research* para la prueba de teoremas y la comprobación de la satisfactibilidad de fórmulas lógicas sobre una serie de teorías.

Modos de uso

Un resolutor SMT, como es Z3, presenta dos modos de uso: determinar la satisfactibilidad de una serie de restricciones y obtener un modelo que las verifique, o establecer la validez de una fórmula lógica.

En el primer caso, Z3 trabaja con una serie de declaraciones de variables y funciones sobre las cuales se establecen una serie de restricciones para que cumplan ciertos requisitos. Una vez definidas todas estas restricciones mediante asertos, se pide a Z3 que compruebe si se pueden satisfacer, es decir, que busque si existe alguna combinación de valores para las variables de modo que los asertos sean ciertos. En caso de que sí que lo sea, se le puede pedir además que nos devuelva un modelo, es decir, una asignación de valores a cada variable que hace que todas las restricciones se verifiquen. A la hora de pedir el modelo nos encontramos una limitación, y es que únicamente nos devuelve una posible combinación, cuando uno podría estar interesado en obtener más de una.

En el segundo modo de uso, lo que se busca es determinar si una cierta fórmula lógica es válida, es decir, si es siempre cierta para cualquier combinación de valores. Si pidiéramos a Z3 que comprobara la satisfactibilidad de tal fórmula como hacíamos en el primer modo de uso, en caso de obtener que es satisfactible quiere decir que hay al menos una combinación que hace cierta la fórmula. Pero esto no nos es suficiente puesto que nosotros necesitamos que sea cierto para cualquier asignación de valores y no para una particular. De este modo, el procedimiento consiste en negar nuestra fórmula y pedir, ahora sí, que compruebe si se puede satisfacer. Así, si Z3 nos devuelve que esta fórmula negada es insatisfactible, entonces podremos asegurar que la original era válida puesto que se hacía cierta para valores cualesquiera.

Entre los dos modos de uso explicados, nosotros utilizaremos el primero de ellos. Como ya hemos comentado previamente, buscamos generar una serie de casos de prueba para distintas funciones y que pueden tener que cumplir una serie de condiciones que constituyen la precondición. Estas condiciones generarán entonces una serie de restricciones que nosotros necesitamos que se verifiquen. Además, una vez sepamos que pueden satisfacerse, nos interesará obtener un modelo para tales restricciones, que conformará nuestro caso de prueba. Sin embargo, nos encontramos

```
(declare-const a Int)
(declare-fun f (Int) Bool)

(assert (< a 5))
(assert (= (f a) true))

(check-sat)
(get-model)
```

Figura 2.4: Código de un programa básico en Z3

con el problema de que nosotros deseamos obtener varios casos de prueba y, como hemos comentado anteriormente, Z3 solo nos devuelve un modelo. Para solucionar este problema la técnica que proponemos consiste en ir añadiendo sucesivamente nuevas restricciones que nieguen los modelos obtenidos anteriormente. De este modo, los nuevos modelos tendrán que verificar ser distintos a los anteriores, obteniendo así diversos casos de prueba.

Lenguaje

El formato de la entrada en Z3 es una extensión del que se define en el *SMT-LIB 2.0 standard*. Un script de Z3 consta de una secuencia de comandos que se almacenan en una pila que el resolutor gestiona internamente. Entre los comandos podemos encontrar declaraciones de constantes y funciones (*declare-const* y *declare-fun* respectivamente); restricciones (*assert*), que añaden una fórmula en la pila, y que serán las que el verificador trate de hacer ciertas, encontrando una interpretación adecuada para las funciones y constantes declaradas; *check-sat*, que nos servirá para determinar si las fórmulas almacenadas en la pila son satisfactibles o no, devolviendo *sat* o *unsat* según corresponda; y *get-model* que nos devuelve una interpretación para las constantes y funciones en caso de que haya sido resuelto como satisfactible.

En la Figura 2.4 puede verse la sintaxis concreta que tendría un pequeño programa Z3, en el que quedan recogidos todos los comandos mencionados anteriormente. El verificador tratará de dar valores tanto a la constante `a` y la función `f` de forma que ambos asertos se hagan ciertos. En caso de encontrar tales valores, nos devolverá que es satisfactible junto a un modelo concreto, y en caso contrario que es insatisfactible. Queda reflejado en la Figura 2.5 lo obtenido para este sencillo caso de prueba. Se observa que ha encontrado una posible asignación de valores, de modo que el resultado es *sat*, y que dicha asignación nos conforma un modelo en el que se tiene `a=0` y `f(x)=true` si `x=0`, o `f(x)=true` en caso contrario.

```
sat
(model
    (define-fun a () Int
         0)
    (define-fun f ((x!0 Int)) Bool
        (ite (= x!0 0) true
           true))
)
```

Figura 2.5: Resultado y modelo de la ejecución del programa de la Figura 2.4

```
(declare-const a Int)
(declare-const b Int)
(declare-const c Real)
(declare-const d Real)

(assert (< (- a 1) (+ b 2)))
(assert (>= c d))

(check-sat)
(get-model)
```

Figura 2.6: Ejemplo de uso de enteros y reales

Teorías

Z3 cuenta con resolutores para diversas teorías. Contamos a continuación aquellas que usaremos durante el desarrollo de este trabajo.

- **Aritmética lineal de enteros y reales:** declarados mediante el comando *declare-const*, Z3 nos permite representar y utilizar los números enteros y reales matemáticos. Podremos usarlos en nuestras fórmulas junto a diferentes operadores como $+, -, <$, etc. (Figura 2.6). Además, Z3 nos da soporte para realizar la división.

- **Lógica proposicional:** mediante el tipo predefinido *Bool* podemos trabajar con expresiones booleanas en Z3. Soporta los operadores usuales *and, or, xor, not,* $=>$ (para la implicación), *ite* (que representa la estructura *if-then-else*) y $=$ (para la doble implicación). En la Figura 2.7 podemos ver un sencillo ejemplo de uso de booleanos con diferentes operadores. Al pedir que la función `f` aplicada al valor `true` no sea cierta, es decir, que la rama *if* de dicha función sea falsa, obtenemos los valores de `true` y `false` para `p` y `q` respectivamente.

- **Arrays:** Z3 cuenta con una teoría básica de arrays, representados internamente en forma de función no interpretada de índices en valores, en principio de dominio infinito. Quedan caracterizados mediante dos operaciones, *select* y

```
(declare-const p Bool)
(declare-const q Bool)
(declare-const r Bool)

(define-fun f ((t Bool)) Bool
      (
            ite(= t true)
                 (=> p q)
                 (=> q p)
       )
)

(assert (not (f true)))

(check-sat)
(get-model)
```

Figura 2.7: Ejemplo de uso de booleanos

store. De este modo, la operación `(select a i)` nos devuelve el elemento de la posición `i` del array `a`; y la operación `(store a i v)` nos devuelve un array idéntico a `a` salvo en la posición `i` en la que se encuentra el valor `v`. Cuando queramos un modelo de un array, obtendremos una interpretación del mismo en forma de función, mediante el constructor `(- as-array f)`. Si para un cierto array `a` obtenemos tal interpretación, entonces para todo índice `i` se tiene que `(select a i)` es igual a `(f i)`. Observamos en la Figura 2.8 que Z3 crea la función auxiliar `k!0` para dar una interpretación al array `arr`. Esta función nos devuelve: 5 si recibe un 2; 2 si recibe un 1; y 5 en cualquier otro caso. Esto se ajusta a lo requerido en las dos fórmulas `select`.

- **Tipos algebraicos:** encontramos aquí una de las principales ventajas de Z3, que es que nos permitirá especificar algunas de las estructuras de datos más comunes como tuplas, árboles o listas. Además, presenta la opción de declarar tipos recursivos y mutuamente recursivos. Para declarar un tipo algebraico usaremos el comando `declare-datatypes`, indicando después las constructoras del tipo.

- **Cuantificadores:** Z3 puede trabajar con fórmulas que utilicen cuantificadores. Para manejar tales fórmulas, utiliza diversos enfoques. Al trabajar con cuantificadores, el comando *check-sat* puede devolvernos un nuevo valor, *unknown*, en caso de no haber podido instanciar las variables cuantificadas, dado que el problema es en general indecidible.

```
(declare-const arr (Array Int Int))

(assert (= (select arr 2) 5))
(assert (= (select arr 1) 2))

(check-sat)
(get-model)


sat
(model
   (define-fun arr () (Array Int Int)
      (_ as-array k!0))
   (define-fun k!0 ((x!0 Int)) Int
      (ite (= x!0 2) 5
      (ite (= x!0 1) 2
          5))
)
```

Figura 2.8: Declaración de array e interpretación en forma de función

Capítulo 3

El lenguaje de asertos

En un aserto de la IR pueden aparecernos expresiones básicas sobre booleanos y enteros, los primeros relacionados entre sí mediante los operadores lógicos *and*, *or* y *not*, y los segundos relacionados con operadores como $<$, $=$, $\leq$, etc. Dentro de estas expresiones básicas podemos encontrarnos también cuantificadores ($\exists$ y $\forall$). Pero además, dentro de un aserto podemos encontrarnos predicados y funciones específicos de ciertos tipos de datos, como podrían ser la longitud de una lista, la ordenación de un array o la altura de un árbol. Todos estos asertos quedan reflejados en la Figura 3.1.

Durante esta sección veremos cómo transformar predicados a restricciones Z3, y sobre todo, analizaremos cómo queda definido cada tipo algebraico en Z3 y todas las operaciones que podremos realizar sobre ellos.

Pero antes de desarrollar todo ello, es conveniente detenernos en comentar dos aspectos de Z3 que resultan de suma importancia para poder implementar tanto los tipos algebraicos como sus diferentes operaciones. Lo primero, como ya comentamos en la Sección 2.3, es la posibilidad de crear tipos recursivos, lo que nos permitirá crear elementos como los árboles o las listas que de otro modo no sería posible. El segundo aspecto importante es que podremos trabajar con funciones recursivas usando el comando `def-fun-rec`. Gracias a esta opción de Z3, podremos recorrer todas las estructuras recursivas de manera cómoda haciendo los cálculos que sean oportunos, que de otro modo, no habría sido posible. Y es que, por ejemplo, algo tan básico como calcular la longitud de una lista requiere el recorrerla recursivamente analizando la constructora y acumulando la longitud.

3.1. Sintaxis básica

Las expresiones más básicas que podemos encontrarnos en un aserto, serán aquellas que realicen sencillas operaciones sobre enteros o booleanos. Para trabajar con ellas en Z3, bastará con crear asertos independientes en los que se vayan plasmando tales operaciones.

$$
\begin{array}{rcll}
\phi & ::= & \mathit{true} \mid \mathit{false} \mid \mathit{id} & \{ \text{ constante booleana o variable } \} \\
& \mid & \mathit{id}\ t_1 \cdots t_n & \{ \text{ aplicación de predicado } \} \\
& \mid & \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi_1 \rightarrow \phi_2 \mid \phi_1 \equiv \phi_2 \mid \neg\phi & \{ \text{ conectivas proposicionales } \} \\
& \mid & \forall\ \overline{\mathit{id}_i : \mathit{type}_i}\ .\ \phi \mid \exists\ \overline{\mathit{id}_i : \mathit{type}_i}\ .\ \phi & \{ \text{ aserto cuantificado } \}
\end{array}
$$

Figura 3.1: Sintaxis abstracta de los asertos

```
(assert (and p q))
(assert (not (and p (or q r))))
(assert (=> p (and q r))
```

Figura 3.2: Ejemplos sobre expresiones booleanas

Al trabajar con expresiones booleanas, podremos encontrarnos distintas variables de tipo *Bool* relacionadas entre sí mediante operadores lógicos tales como *and*, *or*, *not*, implicaciones, etc. Todas estos operadores están a nuestra disposición en la plataforma Z3, por lo que podemos utilizarlos con total normalidad como haríamos en cualquier otro lenguaje. Se muestran en la Figura 3.2 algunos ejemplos sencillos sobre asertos con booleanos, donde las variables `p`, `q` y `r` son de tipo *Bool*.

Por su parte, con los asertos que impliquen enteros podremos encontrarnos operaciones como sumas, restas, multiplicaciones o divisiones, así como relaciones de orden tales como $\leq$, $=$, etc. De nuevo, todos estos operadores están a nuestra disposición en Z3 por lo que nuestros asertos podrán incluirlos sin problema alguno. Algunos ejemplos sencillos pueden verse en la Figura 3.3, donde `a` y `b` son variables enteras.

Además de estas operaciones básicas sobre enteros, podremos encontrarnos, tanto en los asertos como en funciones que iremos exponiendo a lo largo de la sección, otras funciones para las que Z3 no nos da soporte y tenemos que definirlas nosotros. Se trata de operaciones para encontrar el máximo o mínimo de dos números (Figuras 3.4 y 3.5 respectivamente) y para obtener el valor absoluto de un cierto número (Figura 3.6).

3.2. Tipo Lista

Para definir una lista lo haremos de forma recursiva siguiendo la idea de que una lista es un elemento seguido de otra lista. Así, las listas quedan definidas en Z3 como puede verse en la Figura 3.7. Se observa que las constructoras son bien `nil`, para la lista vacía, bien `cons` para una lista no vacía. En este segundo caso, siguiendo la idea comentada previamente, tenemos un elemento de tipo `Int` que será la cabeza de la lista y una cola que será otra lista. Para acceder a estos dos campos contamos con los destructores `hd` y `tl`.

Definido ya el tipo, queda ver cada una de las operaciones que podremos realizar sobre él. En los asertos, entre los predicados que refieren a listas nos en-

```
(assert (= a 5))
(assert (> (+ a b) 7)
```

Figura 3.3: Ejemplos sobre expresiones con enteros

```
(define-fun
      max ((a Int) (b Int)) Int
           (ite(> a b) a b)
 )
```

Figura 3.4: Cálculo del máximo de dos números

contramos: *length*, para calcular la longitud de una lista; *member*, que nos permite saber si un elemento pertenece a una lista; *sortedList*, para saber si una lista está ordenada; y *multiset*, que nos proporciona el multiset de la lista. Veamos entonces como queda implementada cada una de ellas en Z3.

Longitud

La operación *length*, nos devolverá la longitud de una lista dada. El cálculo de dicha longitud se hará de forma recursiva como puede verse en la Figura 3.8. Se observa que la longitud será cero si la lista es vacía o bien uno más la longitud de la lista que conforma la cola.

Lista ordenada

Con esta operación comprobaremos si una cierta lista está ordenada o no, devolviendo *true* o *false* respectivamente. De nuevo, esta función se define recursivamente, de manera que una lista está ordenada si se cumple que el elemento de la cabeza de la lista es menor que el elemento de la cabeza de la cola y, también, que la cola esté ordenada. Además, tanto una lista vacía como una lista con un único elemento están ordenadas. Esta idea queda reflejada en el código de la Figura 3.9, en la que el primer `ite` considera el caso de la lista vacía, el segundo considera la lista de un solo elemento, y la parte del *else* se ocupa del cálculo recursivo.

Pertenencia

Nos encontramos de nuevo con una función recursiva, que nos devolverá *true* en caso de que un elemento dado pertenezca a una lista también dada o *false* en caso contrario. Un elemento pertenecerá a una lista, bien si es la cabeza de la misma o bien si pertenece a la cola. Obviamente, un elemento no puede pertenecer a una lista vacía, por lo que el caso básico de nuestra función en el que la lista sea

```
(define-fun
      min ((a Int) (b Int)) Int
           (ite(< a b) a b)
 )
```

Figura 3.5: Cálculo del mínimo de dos números

```
(define-fun
      absol ((a Int)) Int
           (ite(> a 0) a (* a -1))
 )
```

Figura 3.6: Valor absoluto de un número

vacía devolveremos *false*. En el caso recursivo tendremos que comprobar si la cabeza de la lista es nuestro elemento, o bien si pertenece al resto de la misma y devolver el *or* aplicado a ambas comprobaciones. La función de la Figura 3.10 refleja esta idea recibiendo como parámetros un cierto valor y una lista, y devolviendo la pertenencia de dicho elemento a la lista.

Multiset

Dada una lista, puede resultar interesante obtener su multiset, es decir, una estructura que nos informe sobre cuántas veces aparece un cierto elemento dentro de la lista. En Z3 no existe una estructura propia que nos aporte esta funcionalidad, por lo que tenemos que definirla nosotros. La representación elegida para el tipo *multiset* es en forma de array de tipo T en enteros, siendo T el tipo de los elementos que conforman la lista (Figura 3.11). De este modo, cada posición del array nos dirá cuantas veces aparece un cierto elemento en la lista. Supongamos que la lista en cuestión es de enteros, si $ms[i] = v$, siendo ms un array de tipo *multiset*, entonces tendremos que en nuestra lista el valor i aparece v veces.

Para obtener el multiset de la lista, tendremos que crear un arrray vacío (con todas las posiciones a cero) para el caso de la lista vacía, y sumarle uno a la casilla del elemento de la cabeza de la lista en otro caso. Para ello, necesitaremos una serie de operaciones auxiliares. La función *emptyMs* (Figura 3.12) nos servirá para inicializar un array con todas sus posiciones a cero. Por su parte, la función *unitMs* (Figura 3.13) la usaremos para poner a uno la posición de un cierto elemento y mantener todas las demás a cero. Por último, la función *mset-union* (Figura 3.14) realiza la unión de dos multisets dados. Para ello, gracias a la operación *map* aplicamos el operador + a dos multisets, sumando posición a posición los valores almacenados en cada uno de ellos. Es decir, dados dos multisets *ms1* y *ms2*, con esta función obtenemos un multiset ms tal que $\forall i \in T : ms[i] = ms1[i] + ms2[i]$.

Una vez tenemos ya definido el tipo multiset y las tres operaciones auxiliares que nos hacen falta, procedemos a definir la función principal encargada de

```
(declare-datatypes (T)
        ((Lst nil (cons (hd T) (tl Lst)))))
```

Figura 3.7: Definición del tipo lista

```
(define-fun-rec
        length ((l (Lst Int))) Int
        (
            ite(= l nil)
               0
               (+ 1 (length (tl l)))
        )
  )
```

Figura 3.8: Longitud de una lista

obtener el multiset de una cierta lista dada. Lo haremos de forma recursiva, de modo que en el caso base, cuando la lista sea vacía, devolveremos un array inicializado todo a cero, y en el caso recursivo haremos la unión entre el multiset de la cola de la lista y el multiset unitario con un uno en la posición del elemento de la cabeza de la cola y cero en el resto. Esto queda reflejado en la función de la Figura 3.15.

3.3. Tipo Array

Como ya vimos en la Sección 2.3, la plataforma Z3 nos proporciona una teoría sobre arrays, con una definición propia de los mismos y dos operaciones básicas (*select* y *store*). Haciendo uso únicamente de estos dos operadores, tenemos que definirnos una serie de funciones Z3 con los que cubrir los diferentes predicados que puedan aparecernos en las precondiciones. Estaremos interesados en saber si un array está ordenado y si un cierto array es permutación de otro. Para esta segunda operación, necesitaremos conocer el *set* o el *multiset* de los arrays.

En Z3 los arrays son infinitos, puesto que se tratan como funciones, de modo que para cualquier valor que nos interese pueden devolvernos un resultado. Esto va en contra de la idea clásica de arrays, que comienzan en la posición cero y terminan en alguna posición positiva. Por ello, para caracterizar los arrays en Z3 establecemos por convenio que, para la función que hace las veces de array, solo obtendremos valores a partir de cero. Además, debemos poder conocer en todo momento la longitud del array con el que estemos trabajando. A priori, lo más sencillo parece que es crear una variable con la que referirnos a la longitud. No obstante, esto presenta un problema, y es que en las funciones en las que necesitemos manejar la longitud de un cierto array, pasado como parámetro, no tendremos manera de conocer a qué variable de longitud llamar, pues no conocemos el array que nos están pasando. Otra idea para solucionar este problema sería pasar a nuestras funciones un parámetro adicional que refiera a la longitud, pero algunas de estas funciones se utilizarán como traducciones

```
(define-fun-rec
    sortedList ((l (Lst Int))) Bool
     (
        ite(= l nil)
          true
          (ite(= (tl l) nil)
              true
              (and (< (hd l) (hd (tl l))) (sortedList (tl l))))
    )
)
```

Figura 3.9: Operación *sorted* sobre una lista

```
(define-fun-rec
    member ((i Int) (l (Lst Int))) Bool
    (
        ite(= l nil)
          false
          (or (= i (hd l)) (member i (tl l)))
    )
)
```

Figura 3.10: Pertenencia de un elemento a una lista

directas de predicados de la IR, donde carece de sentido tener un parámetro para la longitud.

Con todo esto, la forma elegida para solventar estos problemas consiste en adaptar el tipo array en nuestros programas. Definimos los arrays como una tupla en la que el primer elemento es un array de Z3 como tal, mientras que el segundo es un entero que refiere a la longitud. De este modo, cuando necesitemos crear una variable de tipo array lo haremos declarándola de este nuevo tipo y cuando conozcamos su longitud le asignaremos tal valor al segundo campo de la pareja. A partir de ahí, podremos trabajar con el array consultando su longitud en cualquier momento. Para trabajar con esta nueva noción de arrays, tenemos entonces que definirnos primero un tipo para representar las parejas y luego definirnos ya el tipo adaptado de los arrays. En la Figura 3.16 pueden verse ambas declaraciones de tipos. De este modo, cuando tengamos una variable de tipo *Arr* y necesitemos acceder a la longitud, usaremos la destructora de los pares *second*, y cuando queramos acceder al array usaremos *first*.

Establecidas entonces estas consideraciones previas, podemos definir cada una de las operaciones que hemos mencionado anteriormente.

```
(define-sort Multiset (T) (Array T Int))
```

Figura 3.11: Definición del tipo *Multiset*

```
(define-fun emptyMs () (Multiset Int)
   ((as const (Array Int Int)) 0))
```

Figura 3.12: Función *empty* para multiset

Longitud

En primer lugar, puede ser interesante contar con una función con la que conocer la longitud de un cierto array. Para ello, bastará con que esta función acceda al segundo campo de la tupla *Arr* que reciba como parámetro, como puede verse en la Figura 3.17.

Array ordenado

Decimos que un array está ordenado si cada elemento de cada posición es menor o igual que los de todas las que le preceden. Gracias al cuantificador *forall* que Z3 pone a nuestra disposición, se puede caracterizar la propiedad de que un array está ordenado del mismo modo que haríamos de forma puramente matemática. A saber, un array está ordenado si $\forall i, j : 0 \leq i \leq j < arr.len \Rightarrow a[i] \leq a[j]$. En lugar de considerar el array completo, nosotros estaremos interesados en poder considerar una parte de éste, de modo que la propiedad quedaría entonces caracterizada por: $\forall i, j : b_1 \leq i \leq j < b_2 \Rightarrow a[i] \leq a[j]$; donde b_1 y b_2 son los límites inferior y superior entre los que comprobar la propiedad de orden. Podemos traducir entonces esta idea directamente a Z3 tal y como puede verse en la Figura 3.18. Observamos que el parámetro de entrada es del tipo *Pair* que nos hemos definido nosotros y que, donde en la fórmula matemática accedemos a posiciones del array a, en nuestro código Z3 cogemos la primera posición de la pareja en la que nos encontramos propiamente el array. Además, recibe como parámetros dos valores enteros que son los límites entre los que comprobar si está ordenado.

Sin embargo, esta caracterización del orden presenta algunos problemas de indecidibilidad. En primer lugar, la representación interna del cuantificador *forall* da problemas si, en lugar de pedir que los elementos sean menores o iguales que los siguientes, pedimos que sean estrictamente menores. En segundo lugar, si dejamos por ejemplo el límite superior en forma de una variable a instanciar por Z3, el comportamiento es también erróneo. Por todo ello, replanteamos la función de orden, haciéndola recursiva, de una forma similar a cómo se hacía con listas. Esta nueva función queda representada en la Figura 3.19.

```
(define-fun unitMs ((i Int)) (Multiset Int)
         (store emptyMs i 1))
```

Figura 3.13: Función *unitMs*

```
(define-fun mset-union ((ms1 (Multiset Int)) (ms2 (Multiset Int))) (Multiset Int)
       ((_ map (+ (Int Int) Int)) ms1 ms2 ))
```

Figura 3.14: Función para realizar la union de dos multisets

Multiset

Del mismo modo que hacíamos con las listas, podemos estar interesados en obtener el *multiset* de un array a fin de conocer los elementos que lo conforman. Para obtenerlo, la idea es exactamente la misma, solo que al no tener constructoras que nos diferencien nuestros casos recursivos de los básicos, necesitaremos trabajar con las longitudes de los arrays. Nos creamos entonces dos funciones, una principal, a la que llamaremos siempre con un único parámetro del tipo *Arr*, como parece lógico, puesto que en una función de cualquier otro lenguaje con este mismo funcionamiento bastaría con tener tal parámetro, y una segunda función auxiliar que utilizaremos de manera interna y a la que pasaremos además la posición por la que vamos recorriendo el array empezando en cero. Ambas funciones aparecen en la Figura 3.20. Vemos además en ella que se utilizan las funciones auxiliares *emptyMs*, *unitMs* y *Mset-union* para realizar los distintos cálculos intermedios, definidas todas ellas en la Sección 3.2 en las Figuras 3.12, 3.13, 3.14, respectivamente.

Permutación

Un array es permutación de otro cuando ambos contienen los mismos elementos aunque ocupen posiciones diferentes. Para calcular si dos arrays son permutación uno de otro, hay que conocer por tanto los elementos que encontramos en cada uno de ellos. Esto se reduce a conocer los *multiset* de cada uno de ellos e igualarlos para ver si coinciden o no. En caso de coincidir ambos multisets podremos afirmar entonces que, efectivamente, los dos arrays eran permutación el uno del otro, y en caso de no hacerlo no lo serán. Recordar que los multisets en Z3 los definimos como un array, que no deja de ser una función que puede recibir infinitos valores de entrada. Podría pensarse que exigir la igualdad de los *multisets*, conlleva que las infinitas posiciones sean iguales, pero afortunadamente basta con que las posiciones para las que está definida sean iguales, ignorando lo que ocurra con el resto. La definición de la función explicada queda recogida en la Figura 3.21.

```
(define-fun-rec
   multisetList ((l (Lst Int))) (Set Int)
   (
      ite(= l nil)
         emptyMs
         (mset-union (multisetList (tl l)) (unitMs (hd l)))
   )
 )
```

Figura 3.15: Cálculo del multiset de una lista

```
(declare-datatypes (T1 T2)
       ((Pair (mk-pair (first T1) (second T2)))))

(define-sort Arr (T) (Pair (Array Int T) Int))
```

Figura 3.16: Definición de tuplas y definición de arrays

3.4. Tipo Árbol Binario

Definimos el tipo algebraico de los árboles binarios de forma recursiva, de modo que un árbol es un nodo raíz que almacena un valor entero y dos hijos, izquierdo y derecho, que son a su vez árboles y que siguen por tanto esta misma estructura. En la Figura 3.22 podemos observar como queda exactamente definido en Z3. Cuenta con dos constructoras, `leaf` para el árbol vacío y `node`. Para la segunda constructora, encontramos el valor almacenado en el nodo, de tipo entero, y dos árboles que son los dos hijos del nodo. Cuando necesitemos acceder a cada uno de estos campos lo haremos usando las destructoras que quedan reflejadas en la definición del tipo: `valor` para acceder al valor almacenado en el nodo; `izq` para acceder al hijo izquierdo; y `der` para el derecho.

Vista ya como queda definido el tipo de los árboles binarios, queda ver cuales son las distintas operaciones que pueden realizarse sobre ellos. De entre las estructuras que pueden implementarse a partir de un árbol binario, las más importantes son los montículos sesgados, los montículos zurdos, los *splay trees* y los BST (*Binary Search Tree*). Por ello, los predicados implementados sobre árboles binarios deben servir, no solo para caracterizar aspectos básicos sobre un árbol, como la altura, sino también para cubrir las condiciones que requieren cada una de estas estructuras. Así, los predicados que analizaremos a continuación serán: *height*, para calcular la altura de un árbol; *minHeight*, para calcular la altura mínima de un árbol y que nos servirá para los montículos; *card*, para saber cuantos elementos tiene el árbol; *set*, con el que obtener el set de un árbol; *isHeap*, que comprobará si un árbol es en particular un montículo; *isLeftist*, para verificar si un montículo es además zurdo; y *isBST*, que nos dirá si un árbol es de búsqueda o no.

```
(define-fun
      lengthArr ((a (Arr Int))) Int
         (second a)
 )
```

Figura 3.17: Longitud de un array

```
(define-fun
    sortedArr ((a (Arr Int)) (b1 Int) (b2 Int)) Bool
    (
         forall ((i Int) (j Int))  (=> (and (<= b1 i) (<= i j) (< j b2))
              (<= (select (first a) i) (select (first a) j)))
    )
)
```

Figura 3.18: Array ordenado

Altura

Para calcular la altura de un árbol lo haremos de forma recursiva, según queda definido en la Figura 3.23. Dado un cierto árbol, la función nos devolverá la altura del mismo. Calcularemos la altura como uno más el máximo de las alturas de los hijos izquierdo y derecho. En caso de tener un árbol vacío, la altura es cero. Así, si por ejemplo tenemos un árbol con un solo nodo, vemos que obtendríamos altura uno $(1 + max(0, 0))$, como cabría esperar.

Para la llamada recursiva, vemos que llamamos a la función *height* pasándole como parámetro el resultado de aplicar la constructora `izq` o `der`, que es de tipo `Tree`, tal y como espera la función.

Altura mínima

La altura mínima de un árbol binario es la altura o profundidad desde la raíz hasta el nodo vacío más cercano. Es decir, es igual que la altura excepto que debemos buscar el mínimo de las alturas de los hijos en vez del máximo. La definición en Z3 (Figura 3.24) es, por tanto, análoga a la función *height*, salvo que en el caso recursivo, en vez de coger el máximo entre las alturas de los hijos, como hacíamos en esa función, aquí tendremos que coger el mínimo.

Cardinal

Cuando hablamos del cardinal de un árbol binario nos referimos al número de elementos que tiene. Es decir, cuantos nodos no vacíos, o lo que es lo mismo, que la constructora no sea `leaf`, nos encontramos. Así, el cardinal de un nodo vacío será

```
(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 3.19: Función recursiva para arrays ordenados

```
   (define-fun
       multisetArr ((a (Arr Int))) (Multiset Int)
          (multisetArrAux 0 a)
   )

   (define-fun-rec
       multisetArrAux ((i Int) (a (Arr Int))) (Multiset Int)
       (
           ite(= i (second a))
              emptyMs
              (Mset-union (multisetArr (+ i 1) a) (unitMs (select (first a) i)))
       )
   )
```

Figura 3.20: Funciones para calcular el multiset de un array

cero y el de uno no vacío será uno. Esto queda reflejado con la función recursiva *card* de la Figura 3.25, en la que en el caso base en el que el árbol es vacío devuelve cero, y en el caso recursivo hace la suma de los cardinales de cada uno de los hijos para calcular el resto de nodos y le suma uno por el nodo actual.

Set

Dado un árbol, podemos estar interesados en obtener una representación del mismo en forma de *Set*. El primer inconveniente que nos encontramos es la representación de sets en Z3, pues no cuenta con un tipo propio. La forma elegida para tratarlos es como un array de valores de tipo T en booleanos, donde T es el mismo tipo que el de los valores del árbol (Figura 3.26), de modo que si un cierto valor v está en el conjunto, $arr[v]$=*true*.

Solucionada la representación del tipo, queda resolver como pasar los valores que nos encontremos en el árbol al array. Lo hacemos recursivamente de modo que el set de un árbol es la unión de los sets obtenidos para sus dos hijos, unido a su vez al conjunto unitario formado por el elemento del nodo en el que estemos. Como caso

```
(define-fun
    permut ((a1 (Arr Int)) (a2 (Arr Int))) Bool
       (= (multisetArr (first a1))
          (multisetArr (first a2)))
)
```

Figura 3.21: Comprobación de si dos arrays son permutación el uno del otro

```
(declare-datatypes (T)
    ((Tree leaf (node (val T) (izq Tree) (der Tree)))))
```

Figura 3.22: Definición del tipo árbol binario

base tendremos el árbol vacío para el cual deberemos devolver un conjunto también vacío. Por tanto, antes de definir la función *Set* nos hace falta definir las funciones auxiliares *empty*, que nos devuelve un set vacío (Figura 3.27); *union*, que hace la unión de dos sets (Figura 3.28); y *unit*, que se encarga de crear un conjunto con un único elemento (Figura 3.29). La primera de ellas simplemente inicializa un array con todas sus posiciones a *false*. La segunda, gracias al *map* aplica la operación *or* a cada par de elementos de dos arrays dados, es decir, $\forall i \in T : set[i] = set1[i] \vee set2[i]$. La tercera pone a *true* la posición dada de un array vacío.

Vistas ya entonces todas las funciones auxiliares, solo queda ver cómo queda la función que obtiene el set de un árbol. Si el árbol es vacío, devolverá un array vacío usando la función *empty*. Si, por el contrario, el árbol no es vacío, y estamos en un nodo con valor v e hijos izquierdo y derecho l y r, respectivamente, devolveremos: $set(l) \cup unit(v) \cup set(r)$. Todo esto queda reflejado en la Figura 3.30.

BST

Un árbol binario es de búsqueda si todos los nodos del hijo izquierdo son menores que la raíz, y ésta a su vez es menor que los nodos del hijo derecho. Además, ambos hijos izquierdo y derecho deben preservar también esta propiedad. Así, para comprobar si un árbol es un BST tendremos que hacerlo de forma recursiva. Comprobar que el orden de los elementos es el adecuado no puede limitarse a comparar la raíz de un árbol con los valores almacenados en los hijos izquierdo y derecho, pues lo que necesitamos es que todos los nodos verifiquen la propiedad. Usamos entonces, para hacer las comprobaciones, el set de los hijos, de modo que todo elemento del set del hijo izquierdo deberá ser menor que la raíz, y todo elemento del set del hijo derecho deberá ser mayor. De este modo, la función haría uso del *forall* de Z3, a fin de comprobar la propiedad mencionada para todos los elementos de cada uno de los *sets*.

No obstante, con esta función esperamos obtener dos funcionalidades. La primera de ellas, comprobar si un árbol dado es BST. Para este caso, la definición aportada para nuestra función es perfectamente válida. La segunda de las funcio-

```
(define-fun-rec
     height ((t (Tree Int))) Int
     (
         ite(= t leaf)
            0
            (+ 1 (max (height (izq t)) (height (der t))))
    )
 )
```

Figura 3.23: Operación *height* sobre un árbol binario

```
(define-fun-rec
     minHeight ((t (Tree Int))) Int
     (
         ite(= t leaf)
            0
            (+ 1 (min (minHeight (izq t)) (minHeight (der t))))
    )
 )
```

Figura 3.24: Altura mínima de un árbol

nalidades que esperamos conseguir, a partir de esta función, es la de rellenar una cierta estructura de árbol con valores cumpliendo que el árbol resultante sea BST. Aquí sin embargo encontramos más problemas con nuestra definición, puesto que al trabajar con cuantificadores Z3 no es capaz de instanciar correctamente las variables de la estructura de manera que respeten las condiciones impuestas. Por ello, debemos redefinir la función sin usar el cuantificador ni *sets*.

Consideremos la estructura general de un árbol binario como en la Figura 3.31. Tal árbol será BST si se verifica que r es mayor que el valor mínimo de t_1 y menor que el máximo de t_2. Por tanto, lo primero que necesitamos es definirnos en Z3 dos funciones que se encarguen de calcular los valores mínimo y máximo de un cierto árbol. Explicamos la función que calcula el mínimo y la encargada de calcular el máximo es totalmente análoga. Dado un árbol, para encontrar el mínimo queremos ir recorriendo recursivamente los hijos izquierdo y derecho y comparando los valores almacenados en los nodos de estos con el valor almacenado en la raíz. De este modo, cuando lleguemos a un nodo vacío le asignaremos un valor suficientemente grande, y en caso de estar en un nodo no vacío devolveremos el mínimo entre el valor almacenado en dicho nodo y el valor mínimo de los árboles que conforman sus dos hijos. Esta idea queda plasmada en la función de la Figura 3.32 y en la función equivalente para calcular el máximo (Figura 3.33). Conocidos ya el mínimo y el máximo de los dos hijos de un nodo, podemos aplicar la idea comentada para averiguar si es BST y descender recursivamente por los hijos comprobando si éstos lo son también. Así, la Figura 3.34 recoge la función encargada de comprobar si un árbol pasado como parámetro es o no de búsqueda.

```
(define-fun-rec
     card ((t (Tree Int))) Int
     (
         ite(= t leaf)
              0
              (+ 1 (+ (card (izq t)) (card (der t))))
    )
 )
```

Figura 3.25: Cardinal de un árbol

```
(define-sort Set (T) (Array T Bool))
```

Figura 3.26: Definición del tipo *Set*

Montículo

Un montículo es una particularización de los árboles binarios, donde el problema que pretende resolverse es el de encontrar el elemento mínimo o máximo, según corresponda, y no un elemento cualquiera. Según el elemento que desee encontrarse tenemos los *minHeaps*, en los que hay acceso directo al elemento mínimo, y los *maxHeaps*, en los que lo hay al máximo. Nosotros trabajaremos con los primeros.

En un *minHeap*, la característica fundamental es que todos los hijos de un nodo son mayores o iguales que éste, pero luego entre ellos no hay ninguna restricción adicional. Esto queda reflejado en la función de la Figura 3.35, en la que se comprueba recursivamente que se verifique esta propiedad. En el primer *if* consideramos el caso en el que el árbol sea vacío y en el segundo tenemos que el árbol está formado por un único nodo, cumpliéndose en ambos casos que el árbol t es un montículo y devolviendo por tanto *true*. En el tercer *if* consideramos que solo uno de los hijos sea vacío, concretamente el izquierdo, de modo que en esa rama solo tenemos que comprobar los valores de la raíz y del hijo derecho, mientras que en el cuarto hacemos lo mismo pero considerando esta vez que el hijo vacío es el derecho. Por último, en la rama del *else* final encontramos el caso en que ambos hijos sean no vacíos, para el cual debemos hacer todas las comprobaciones entre el valor de la raíz y los de los hijos.

Por otro lado, un montículo sesgado requiere haber sido formado mediante dos montículos sesgados previos usando *mezcla sesgada*. Esta propiedad resulta imposible de especificar en Z3, y tampoco nos resulta de gran interés, por ello, consideraremos, como única restricción de este tipo de montículos, la que refiere al orden de sus elementos como en el resto de montículos. Por tanto, nos bastará con comprobar la función de la Figura 3.35 para afirmar si un montículo es sesgado.

```
(define-fun empty () (Set Int)
     ((as const (Array Int Bool)) false))
```

Figura 3.27: Función *empty*

```
(define-fun set-union ((s1 (Set Int)) (s2 (Set Int))) (Set Int)
          ((_ map or) s1 s2 )  )
```

Figura 3.28: Función *union*

Zurdo

Uno de los tipos de montículos más interesantes que nos encontramos son los montículos zurdos. Para la definición de un montículo zurdo es necesario usar la altura mínima definida previamente en esta sección. Un montículo es zurdo entonces si es vacío, o si ambos hijos son zurdos y además la altura mínima del hijo izquierdo es mayor o igual que la del hijo derecho. Esta idea queda reflejada en Z3 en la función recursiva de la Figura 3.36. Por tanto, un montículo debe cumplir, para ser zurdo, tanto la función anterior *isHeap*, como esta, *zurdo*.

3.5. Tipo Árbol AVL

Un árbol AVL es un tipo especial de árbol binario en el que en cada nodo, además de almacenar un cierto valor, guardamos también la altura a la que se encuentra dicho nodo en el árbol. Por ello, la representación en Z3 es totalmente idéntica a la de los árboles binarios, pero añadiendo simplemente un campo más a la constructora *node* que refiera a la altura del nodo (Figura 3.37). Además, los árboles AVL son árboles de búsqueda, por lo que los elementos deberán mantener un cierto orden, pero cuentan con una característica adicional, que es que la altura debe mantener el árbol equilibrado, esto es, la diferencia entre las alturas de los dos hijos de un nodo cualquiera no puede ser mayor que uno.

La mayoría de los cómputos que necesitemos realizar sobre un AVL serán los mismos que hacíamos con los árboles binarios: calcular la altura mediante una función *height*; calcular el número de elementos o cardinal; y averiguar su set. Además de estas funciones ya conocidas, que deberán ser redefinidas para el tipo AVL, aparece una nueva función, *isAVL*, encargada de comprobar que se verifiquen todas las condiciones necesarias para que un árbol binario sea un AVL.

```
(define-fun unit ((i Int)) (Set Int)
      (store empty i true))
```

Figura 3.29: Función *unit*

```
(define-fun-rec
   set ((t (Tree Int))) (Set Int)
   (
      ite(= t leaf)
        empty
        (set-union (set-union (set (izq t)) (set (der t))) (unit (val t)))
   )
)
```

Figura 3.30: Función que calcula el set de un árbol binario

Altura

Calcular la altura de un árbol AVL respeta la idea seguida con los árboles binarios. A saber, la altura de un nodo será uno más el máximo de las alturas de sus dos hijos. Por ello, la definición de la función queda exactamente igual que estaba para tales árboles, con la única salvedad de que hay que renombrar las constructoras. En la Figura 3.38 queda reflejada esta función.

Cardinal

Del mismo modo que con la altura, calcular el cardinal es igual para los árboles AVL que para los binarios. Por ello, la función de la Figura 3.39, encargada de hacer tal cálculo, queda definida del mismo modo que la de la Figura 3.25 renombrando las constructoras que aparezcan.

Set

Una vez más, nos encontramos con una función que no presenta diferencias con la utilizada para árboles binarios, más allá de los renombramietos oportunos. Calcular el *set* de un árbol AVL se hace del mismo modo que con árboles binarios, es decir, si tenemos un árbol vacío el conjunto devuelto es también vacío, y para un nodo no vacío el conjunto se obtiene de realizar la unión entre los *sets* de los dos hijos y el conjunto unitario formado por el valor almacenado en dicho nodo. Por ello, se hará uso aquí también de las funciones auxiliares definidas para los árboles binarios, a saber, *empty* (Figura 3.27), *unit* (Figura 3.29), y *set-union* (Figura 3.28). La función general que calcula el *set* del árbol queda definida en la Figura 3.40.

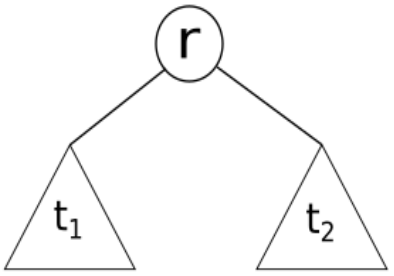


Figura 3.31: Estructura general de un árbol binario

```
(define-fun-rec
       minT ((t (Tree Int))) Int
       (
           ite(= t leaf)
              2000
              (min (val t) (min (minT (izq t)) (minT (der t))))
       )
)
```

Figura 3.32: Cálculo del valor mínimo de un árbol

BST

Para que un árbol sea AVL tendrá que cumplir que sea de búsqueda. Por ello, necesitamos redefinir el predicado *isBST* de la Figura 3.34. La idea seguida es la misma, pues los elementos deben mantener el mismo orden que imponíamos en tal función. Por tanto, únicamente debemos preocuparnos de renombrar las funciones auxiliares (Figura 3.41 y Figura 3.42) y modificar, tanto en ellas como en la principal (Figura 3.43), las diferentes apariciones de las constructoras de nuestro tipo de datos.

AVL

Como comentábamos al principio de la sección, un árbol AVL no deja de ser un árbol binario de búsqueda, con la única particularidad de que tiene una cierta imposición sobre las alturas. Serán entonces estas las restricciones que debamos comprobar a la hora de determinar si un cierto árbol es o no un AVL.

En primer lugar, por ser un árbol de búsqueda tendremos que recurrir al predicado *isBST* que nos determina si el orden de los elementos en los nodos es el adecuado. En segundo lugar, hay que comprobar que la diferencia de alturas entre ambos hijos no sea mayor que uno. Por último, para que un cierto árbol sea

```
(define-fun-rec
       maxT ((t (Tree Int))) Int
       (
           ite(= t leaf)
              -2000
              (max (val t) (max (maxT (izq t)) (maxT (der t))))
       )
)
```

Figura 3.33: Máximo de un árbol

```
(define-fun-rec
   isBST ((t (Tree Int))) Bool
   (
      ite(= t leaf)
        true
        (ite (and (= (izq t) leaf) (= (der t) leaf))
           true
           (ite (= (izq t) leaf)
               (and (isBST (der t)) (< (val t) (minT (der t))))
               (ite (= (der t) leaf)
                   (and (isBST (izq t)) (< (maxT (izq t)) (val t)))
                   (and (and (isBST (izq t)) (isBST (der t)))
                   (and (< (maxT (izq t)) (val t)) (< (val t) (minT (der t)))))
                )
           )
        )
   )
)
```

Figura 3.34: Función que comprueba si un árbol es BST

AVL, además de cumplir estas dos condiciones anteriores, tiene que cumplir también que sus dos hijos sean también AVL. Por ello, la comprobación deberá hacerse de manera recursiva, considerando como caso base el árbol vacío, el cual sí es AVL. La función de la Figura 3.44 plasma todas estas ideas. Observamos que en la rama *if* comprobamos si el árbol es vacío, devolviendo *true* en tal caso, y que en la rama del *else* hacemos las llamadas recursivas con los hijos izquierdo y derecho como parámetros, además de comprobar que sea BST y que se cumpla la diferencia de alturas requerida, siendo *absol* la función valor absoluto definida como en la Sección 3.1 de este capítulo. Además de todo esto, tenemos que añadir una condición especial. Si recordamos la definición de un árbol AVL, teníamos un campo reservado en cada nodo para almacenar su altura. Pues bien, este valor debe ser correcto, es decir, debe ser efectivamente la altura de dicho nodo. Por ello, al comprobar que un árbol sea AVL haremos también tal comprobación, lo que se reduce a ver si el valor guardado en el nodo es igual a uno más el máximo de las alturas de sus hijos.

```
(define-fun-rec
 isHeap ((t (Tree Int))) Bool
 (
  ite(= t leaf)
   true
   (ite (and (= (izq t) leaf) (= (der t) leaf))
     true
     (ite(= (izq t) leaf)
       (and (< (value t) (minT (der t))) (isHeap (der t)))
       (ite(= (der t) leaf)
           (and (< (value t) (minT (izq t))) (isHeap (izq t)))
           (and (and (< (val t) (val (izq t))) (< (val t) (val (der t))))
                 (and (isHeap (izq t)) (isHeap (der t))))
       )
     )
)))
```

Figura 3.35: Función *isHeap*

```
(define-fun-rec
    isLeftist ((t (Tree Int))) Bool
    (
        ite (= t leaf)
           true
           (and (and (isLeftist (izq t)) (isLeftist (der t)))
                   (>= (minHeight (izq t)) (minHeight (der t))))
       )
)
```

Figura 3.36: Función que comprueba si un árbol binario es zurdo

Esta función es la principal de los árboles AVL y nos proporcionará dos usos fundamentales. El primero y más básico es comprobar si el árbol que recibe como parámetro verifica todas las condiciones que debe cumplir para ser AVL. No obstante, el uso más interesante es rellenar una cierta estructura de forma que el árbol resultante sea AVL. De este modo, un cierto árbol con estructura de AVL, pero con variables desconocidas como campos de cada uno de los nodos, puede pasarse a la función como parámetro y que ésta nos de valores a cada una de las variables, tanto las de valor como las de altura, cumpliendo todas ellas todos los requisitos necesarios.

```
(declare-datatypes (T)
     ((AVL leafA (nodeA (val T) (alt Int)
          (izq AVL) (der AVL)))))
```

Figura 3.37: Definición del tipo AVL

```
(define-fun-rec
     heightA ((t (AVL Int))) Int
     (
         ite(= t leafA)
            0
            (+ 1 (max (heightA (izq t)) (heightA (der t))))
    )
)
```

Figura 3.38: Altura de un AVL

3.6. Tipo Árbol Rojinegro

Los árboles rojinegros son una variante de los árboles binarios en los que añadimos a las aristas un color. Alternativamente, podemos asignar un color a un nodo suponiendo que dicho color es el de la arista que le unía con su padre. Nosotros adoptaremos esta segunda idea, creando un tipo de datos análogo al de los árboles binarios pero con un campo adicional referente al color. Por otro lado, dentro de los árboles rojinegros podemos encontrar diferentes variantes. Nosotros utilizaremos la variante *Left-Leaning Red-Black Tree* [3] (en adelante LLRB). De este modo, encontramos un tipo definido recursivamente en el que tendremos la constructora `leafL` para el árbol vacío y la constructora `nodeL` para el resto. Para esta segunda constructora, tendremos que indicar el valor, y los hijos izquierdo y derecho, ambos a su vez árboles rojinegros, como ya hacíamos para el tipo `Tree`, pero indicaremos también el color del nodo en un nuevo campo, como puede verse en la Figura 3.45. Además, para representar el color tendremos otro tipo algebraico nuevo con dos constructoras para sus dos posibles valores, *Rojo* o *Negro* (Figura 3.46).

Para trabajar con árboles rojinegros necesitamos los siguientes predicados: *heightL*, para calcular la altura; *cardL*, que nos dará el número de elementos que tiene; *blackHeight*, con el que saber la altura negra; *getColor*, que nos devolverá el color de un nodo; *goodColor* para saber si el coloreado de los nodos es correcto; e *isLlrb* para verificar si un cierto árbol es rojinegro o no.

```
(define-fun-rec
     cardA ((t (AVL Int))) Int
     (
         ite(= t leafA)
              0
              (+ 1 (+ (cardA (izq t)) (cardA (der t))))
    )
 )
```

Figura 3.39: Cardinal de un árbol AVL

```
(define-fun-rec
   setA ((t (AVL Int))) (Set Int)
   (
      ite(= t leafA)
        empty
        (set-union (set-union (setA (izq t)) (setA (der t))) (unit (val t)))
   )
)
```

Figura 3.40: Función que calcula el set de un árbol AVL

Altura

El método para calcular la altura de un árbol rojinegro es totalmente análogo al explicado en la Sección 3.4 para un árbol binario. Redefinimos la función para trabajar con árboles rojinegros pero manteniendo la misma estructura recursiva. Podemos ver en la Figura 3.47 como queda definida.

Altura negra

Al trabajar ahora con colores en los nodos, nos aparece una nueva altura, la altura negra. Ésta se define como el número de nodos negros que han de atravesarse desde la raíz hasta un nodo vacío. La idea para calcular esta altura se apoya en la que seguíamos en el cálculo de la altura normal, a excepción de que solo sumaremos cuando encontremos un nodo de color negro. Así, el cálculo de la altura negra se realiza de manera recursiva, tomando el máximo de las alturas negras de los hijos izquierdo y derecho, y sumándole uno en caso de que el nodo en el que nos encontremos sea negro. En caso de tratar con un nodo vacío la altura devuelta será cero. Estas ideas quedan reflejadas en la definición de la función *blackHeight* de la Figura 3.48.

```
(define-fun-rec
       minTA ((t (AVL Int))) Int
       (
           ite(= t leafA)
              2000
              (min (val t) (min (minTA (izq t)) (minTA (der t))))
       )
)
```

Figura 3.41: Cálculo del valor mínimo de un árbol

```
(define-fun-rec
       maxTA ((t (AVL Int))) Int
       (
           ite(= t leafA)
              -2000
              (max (val t) (max (maxTA (izq t)) (maxTA (der t))))
       )
)
```

Figura 3.42: Máximo de un árbol

Cardinal

Como ya ocurría con la altura, esta función es igual que la definida en la Sección 3.4 para árboles binarios, modificando simplemente el tipo del árbol que se le pasa como entrada. En la Figura 3.49 encontramos la definición exacta de la función que calcula el cardinal de un árbol rojinegro.

Set

Una vez más, podemos estar interesados en obtener el set, en este caso, de un árbol rojinegro. El método seguido es el mismo que en secciones previas en las que veíamos como calcular el *set* de árboles binarios o AVL. Requiere por tanto de las mismas funciones auxiliares: *empty*, *unit* y *union* (Figuras 3.27, 3.29 y 3.28 respectivamente). Por su parte, la función principal queda igual que en los casos previos, modificando los nombres de las constructoras cuando sea oportuno (Figura 3.50).

```
(define-fun-rec
   isBSTA ((t (AVL Int))) Bool
   (
      ite(= t leafA)
        true
        (ite (and (= (izq t) leafA) (= (der t) leafA))
           true
           (ite (= (izq t) leafA)
               (and (isBSTA (der t)) (< (val t) (minTA (der t))))
               (ite (= (der t) leafA)
                   (and (isBSTA (izq t)) (< (maxTA (izq t)) (val t)))
                   (and (and (isBSTA (izq t)) (isBSTA (der t)))
                   (and (< (maxTA (izq t)) (val t)) (< (val t) (minTA (der t)))))
                )
           )
        )
   )
)
```

Figura 3.43: Función que comprueba si un árbol de tipo AVL es BST

```
  (define-fun-rec
      isAVL ((t (AVL Int))) Bool
      (
          ite(= t leaf)
             true
             (and (isBSTA t) (and (and (and (avl (izq t)) (avl (der t)))
               (<= (absol (- (alt (izq t)) (alt (der t)))) 1))
               (= (alt t) (+ 1 (max (heightA (izq t)) (heightA (der t))))))))
      )
  )
```

Figura 3.44: Comprobación de si un árbol es AVL

Good Color

La función *goodColor* será la encargada de comprobar que los nodos estén bien coloreados.

Antes de entrar en el código en sí de la función, resulta conveniente comentar qué restricciones han de cumplirse para considerar que el color de un nodo es adecuado o no. En primer lugar, un nodo vacío, por defecto, está bien coloreado. Por otro lado, si tenemos un nodo no vacío, deberán cumplirse diferentes condiciones sobre los hijos según nuestro nodo sea de color negro o rojo. En caso de tratar con un nodo negro, si el hijo derecho es rojo, entonces el izquierdo deberá ser rojo también (de ahí su nombre "escorado a la izquierda"). Si en cambio estamos tratando con un nodo rojo, deberá cumplirse que ambos hijos sean de color negro, puesto que en

```
(declare-datatypes (T)
     ((LLRB leafL (nodeL (val T) (color Color)
          (izq LLRB) (der LLRB)))))
```

Figura 3.45: Definición de los árboles rojinegros

```
(declare-datatypes ()
     ((Color Rojo (Negro))))
```

Figura 3.46: Definición del tipo color

un árbol rojinegro no podemos encontrar dos nodos rojos seguidos.

Todo esto que acabamos de comentar queda reflejado en la función de la Figura 3.51, en la que nos encontramos un primer *if* para considerar el caso del árbol vacío o no, y un segundo *if* para distinguir entre un color u otro de nuestro nodo, haciendo lo que corresponda en cada una de las ramas.

Get Color

Como puede verse en la Figura 3.51, hacemos uso de una cierta función *getColor* para acceder al color de los hijos. Para obtener el color de un nodo, podría usarse la destructora *color*, pero para las hojas no podríamos obtener entonces ningún color. Se trata entonces de una función auxiliar que nos devuelve el color de un nodo, con la particularidad de que si el nodo es vacío nos devuelve negro por defecto. Esta función queda definida en la Figura 3.52

BST

Las condiciones que un árbol rojinegro debe cumplir para ser un árbol de búsqueda, no difieren respecto a ninguno de los tipos de árboles vistos hasta ahora. Por ello, la definición de esta función es totalmente análoga a las anteriores. Únicamente requiere renombrarla para saber que estamos trabajando con este nuevo tipo de datos y considerar que las constructoras son diferentes. Así, la definición de la función queda como puede verse en la Figura 3.53.

LLRB

La mayoría de las funciones vistas hasta ahora durante esta sección, nos servirán para comprobar si un cierto árbol es o no rojinegro. Las condiciones que debe cumplir un árbol binario para serlo son: que sea BST, que la altura negra del hijo izquierdo sea igual que la del hijo derecho, que esté bien coloreado y, por último, que sus dos hijos sean también rojinegros. Estamos, por tanto, ante una

```
(define-fun-rec
     heightL ((t (LLRB Int))) Int
     (
         ite(= t leafL)
            0
            (+ 1 (max (heightL (izq t)) (heightL (der t))))
    )
 )
```

Figura 3.47: Altura de un árbol rojinegro

```
(define-fun-rec
   blackHeight ((t (LLRB Int))) Int
   (
        ite(= t leafL)
           0
           (ite(= (getColor t) Negro)
              (+ 1 (max (blackHeight (izq t)) (blackHeight (der t))))
              (max (blackHeight (izq t)) (blackHeight (der t)))
           )
   )
 )
```

Figura 3.48: Altura negra de un árbol rojinegro

función recursiva, en la que como caso base tenemos el árbol vacío, para el cual se devuelve *true*, y en el caso recursivo haremos las comprobaciones de las condiciones previas usando las funciones *blackHeight*, *isBSTL* y *goodColor*, además de realizar las llamadas recursivas con ambos hijos del nodo. Todo esto queda recogido en la función de la Figura 3.54.

Como puede verse, esta función recibe el nombre de *isLlrbAux*. Y es que hay una condición más que nos interesa que cumplan los árboles rojinegros, sin ser una restricción propia de este tipo de datos como tal. Se trata de tener la raíz coloreada como negra. Por ello, la función principal a la que llamaremos para comprobar si un árbol es LLRB, mirará primero si la raíz es de color negro y después llamará a la función previa *isLlrbAux* para que haga el resto de comprobaciones. En la Figura 3.55 queda recogida esta función.

Esta función nos aporta dos usos fundamentales. El primero de ellos es comprobar si un cierto árbol dado es o no rojinegro. El segundo, mucho más interesante, nos permitirá rellenar un árbol dado de modo que cumpla ser un árbol rojinegro. De este modo, si para un cierto árbol no conocemos los valores de los nodos ni el color de estos, podremos llamar a esta función con tales campos sin completar y Z3 nos los rellenará respetando todas los requisitos que debe cumplir para ser rojinegro, es decir, ordenará los valores para que sea BST y colocará los colores para que se verifique la igualdad sobre las alturas negras y las condiciones

```
(define-fun-rec
     cardL ((t (LLRB Int))) Int
     (
         ite(= t leafL)
              0
              (+ 1 (+ (cardL (izq t)) (cardL (der t))))
    )
 )
```

Figura 3.49: Cardinal de un árbol rojinegro

```
(define-fun-rec
   setL ((t (LLRB Int))) (Set Int)
   (
      ite(= t leafL)
        empty
        (set-union (set-union (setL (izq t)) (setL (der t))) (unit (val t)))
   )
)
```

Figura 3.50: Obtención del set de un LLRB

de la función *goodColor*.

3.7. Doble uso de los predicados

Durante la explicación de algunos de los métodos ya hemos ido introduciendo la idea de que presentaban dos modos de uso. Este planteamiento requiere abordarlo con algo más de detenimiento pues es lo más innovador que nos aporta este trabajo. Si recordamos, en capítulos previos decíamos que hasta ahora la generación de casos de prueba consistía en generar diferentes casos y filtrarlos descartando aquellos que no eran válidos, y que nosotros trataríamos de generarlos siendo directamente buenos. Pues bien, es precisamente este doble uso de los predicados definidos el que nos permite hacer esto sin perder la capacidad de filtrar en caso de necesitarlo.

En primer lugar, el uso más básico que podemos hacer es el de filtrar de manera clásica. Dado un cierto valor para una variable, cualquiera que sea su tipo, podremos llamar a una cierta función que nosotros tengamos definida que realice una serie de comprobaciones sobre ella y nos diga si cumple una serie de condiciones o no. Hasta aquí, no encontramos nada nuevo, puesto que esto puede hacerse con cualquier función que nos definamos en cualquier lenguaje de programación.

En segundo lugar, podemos usar los predicados de Z3 para asignar valores a variables cumpliendo una serie de condiciones. Y es aquí donde se nos presentan las opciones más novedosas e interesantes, pues, estamos entonces en condiciones

```
(define-fun
   goodColor ((t (LLRB Int))) Bool
   (
      ite(= t leafL)
        true
        (ite(= (color t) Negro)
            (=> (= (getColor (der t)) Rojo) (= (getColor (izq t)) Rojo))
            (and (= (getColor (izq t)) Negro) (= (getColor (der t)) Negro))
        )
   )
)
```

Figura 3.51: Función *goodColor* para árboles rojinegros

```
(define-fun
      getColor ((t (LLRB Int))) Bool
      (
          ite(= t leafL)
              Negro
              (color t)
      )
)
```

Figura 3.52: Función *getColor* para árboles rojinegros

de darle directamente valores buenos a una variable, en lugar de perder recursos en asignarle valores aleatorios y luego comprobar si nos valen o no. Por ejemplo, de manera muy sencilla, si una cierta variable *var* necesitamos que sea menor que tres, con el primer enfoque tendríamos que darle valores aleatorios y luego descartar mientras que con el segundo podemos conseguir que automáticamente tome un valor que nos interese. Esto que podría parecer irrelevante con variables y restricciones tan sencillas, puede aplicarse a cualquier variable de cualquier tipo, y es ahí donde obtenemos la verdadera potencia de este método de utilización de nuestros predicados.

Consideremos por ejemplo una cierta variable de tipo *Tree*. Esta variable tiene cinco nodos para los que necesitamos obtener unos valores que nos verifiquen una serie de restricciones de orden, por ejemplo, que estén ordenadas de manera que el árbol sea de búsqueda. Si solo contáramos con el primer método de uso comentado, tendríamos que limitarnos a crear árboles de cinco nodos con valores aleatorios en cada uno de ellos, y pasarlos por nuestro método correspondiente para comprobar si están ordenados como deseábamos. No obstante, con la segunda posibilidad podemos almacenar en cada nodo una cierta variable, de modo que al pedirle a Z3 que verifique la función correspondiente, él solo sea capaz de darnos valores buenos para cada uno de estos nodos ahorrándonos tanto la generación automática como el filtrado.

No obstante, se encuentran algunas limitaciones en este modo de uso. Y es

```
(define-fun-rec
   isBSTL ((t (LLRB Int))) Bool
   (
      ite(= t leafL)
        true
        (ite (and (= (izq t) leafL) (= (der t) leafL))
           true
           (ite (= (izq t) leafL)
               (and (isBSTL (der t)) (< (val t) (minTL (der t))))
               (ite (= (der t) leafL)
                   (and (isBSTL (izq t)) (< (maxTL (izq t)) (val t)))
                   (and (and (isBSTL (izq t)) (isBSTL (der t)))
                   (and (< (maxTL (izq t)) (val t)) (< (val t) (minTL (der t)))))
                )
           )
        )
   )
)
```

Figura 3.53: Función que comprueba si un árbol rojinegro es de búsqueda

```
 (define-fun-rec
    isLlrbAux ((t (LLRB Int))) Bool
    (
       ite(= t leafL)
          true
          (and (isBSTL t) (and (and (and (isLlrbAux (izq t)) (isLlrbAux (der t)))
             (goodColor t)) (eq (blackHeight (izq t)) (blackHeight (der t)))))
    )
 )
```

Figura 3.54: Función *isLlrbAux*

que, a la hora de crear variables de un tipo algebraico perdemos esta potencia. Por ejemplo, tal y como hemos visto, funciones como la encargada de calcular el cardinal de un árbol o la altura del mismo, están definidas de manera recursiva. Por ello, si nosotros pidiéramos a Z3 que intentara darle valor a una variable t de tipo *Tree*, que es un tipo algebraico definido también recursivamente, cumpliendo que tuviera un cierto cardinal o una cierta altura, Z3 es incapaz de encontrar una asignación para nuestra variable t. Por ello, para estos tipos de restricciones no nos quedará más remedio que tratarlas con el primero de los enfoques comentados.

Será durante el capítulo próximo cuando entremos en detalle a comentar como hemos decidido resolver estas limitaciones, intentando exprimir la potencia de Z3 al máximo cuando sea posible, clasificando las restricciones en diferentes tipos. Comentaremos también a qué tipo de restricciones pertenecen cada una de las funciones vistas a lo largo de este capítulo y con ello el enfoque de uso que les damos.

```
(define-fun
     isLlrb ((t (LLRB Int))) Bool
     (
           and (= (getColor t) Negro) (isLlrbAux t)
     )
)
```

Figura 3.55: Función *isLlrb*

A modo de conclusión, se incluye un pequeño ejemplo con el que ilustrar estas ideas que acabamos de comentar. Supongamos que tenemos una variable t que es un árbol binario (tipo *Tree*). Abordemos primero las limitaciones. Si pedimos a Z3 que trate de verificar una restricción del estilo `(assert (= (card t) 4))`, en la que se requiere que el cardinal de t sea cuatro, nos encontramos que Z3 se queda bloqueado, incapaz de resolver este problema. Comentamos ahora los diferentes modos de uso. Mantengamos la misma variable t, pero esta vez con una estructura ya definida, por ejemplo `(node x1 (node x2 leaf leaf) (node x3 leaf leaf))`, donde cada variable x_i refiere al valor almacenado en cada nodo. Si le pedimos a Z3 que verifique la restricción `(assert (= (card t) 3))`, en la que se le pide que el cardinal sea tres, actuará en modo de filtro calculando el cardinal de nuestro árbol y devolviéndonos *sat* o *unsat* según cumpla la restricción o no. Por otro lado, si utilizamos una restricción del tipo `(assert (isBST t))`, estamos pidiendo a Z3 que verifique que el árbol es de búsqueda. Pues bien, es aquí donde entra en juego el segundo modo de uso, pues Z3 no solo nos devuelve *sat* en caso de que pueda rellenarse, sino que nos devuelve también valores para cada variable x_i, a saber, $x_1 = 1999, x_2 = -1999$ y $x_3 = 2001$. Estos valores son aleatorios pero cumplen el orden requerido.

Capítulo 4

Estrategia de generación de casos

Como ya hemos mencionado, este trabajo tiene por objetivo la generación de casos de prueba que se ajusten automáticamente a las precondiciones del programa que deseemos probar. Para ello, resulta conveniente transformar tales precondiciones en una secuencia de restricciones. Así, con estas restricciones y gracias a la potencia de Z3, podremos obtener un modelo que cumpla todas ellas, es decir, un caso de prueba de nuestro programa totalmente válido.

Para poder generar los casos de prueba correctamente, en primer lugar hemos tenido que clasificar las restricciones que puede interesarnos tratar en diferentes grupos, como veremos en la primera sección de este capítulo. Una vez determinadas las diferentes restricciones y elegido el método seguido para tratar cada una de ellas, solo queda transformar la IR a un conjunto de restricciones procesables por Z3, creando para ello un archivo de formato *smt*. Comentaremos, por tanto, en la segunda sección de este capítulo la estrategia seguida para llevar este proceso a cabo. Hablaremos también de las limitaciones encontradas y como las hemos ido solventando.

4.1. Tipos de restricciones

Como ya se introdujo en la Sección 3.7, las diferentes funciones de Z3 podían ser utilizadas de diversas maneras, encontrándonos problemas cuando queríamos construir un elemento de tipo algebraico a partir de una altura o un número de elementos. Estas funciones se utilizarán por precondiciones que puedan requerirlas. Por tanto, estas diferencias entre métodos de utilización de unas funciones y otras deriva en que debemos tratar de manera diferente los asertos que podamos encontrarnos en las precondiciones. Cada aserto al final impone un restricción sobre una cierta variable, por lo que lo que haremos es clasificar las restricciones en tres grandes grupos: restricciones de tamaño, restricciones de estructura y restricciones de contenido.

4.1.1. Restricciones de tamaño

Tal y como su nombre indica, hacen referencia al tamaño de las diferentes estructuras que puedan aparecernos. Serán entonces la función *lengthArr* para arrays, *length* para listas y *card* para árboles. Para arrays, si recordamos, la longitud no era más que un campo de una tupla, por lo que no habría más que crearnos una restricción para asignar el valor deseado a tal campo y tendríamos ya la longitud del array. No obstante, para listas y árboles, que son tipos algebraicos, nos encontramos el problema mencionado en la Sección 3.7, a saber, Z3 no puede crear una estructura a partir de un cierto tamaño. Por ello, estas funciones serán tratadas como meros elementos de verificación y no para crear variables. De esta manera, podremos utilizarlas para comprobar si el tamaño de una estructura es el requerido, en caso de que la precondición nos lo pida.

4.1.2. Restricciones de estructura

Aplicadas exclusivamente a árboles, trabajarán sobre la estructura de los mismos, es decir, sobre aspectos como la altura, la altura mínima, los colores, la altura negra...

Con estas restricciones desearíamos obtener árboles vacíos, es decir, que no tengan ningún dato almacenado, y que se ajusten a la estructura pedida. Por ejemplo, supongamos una cierta precondición que nos pide que la diferencia de alturas entre los hijos izquierdo y derecho de un árbol no sea mayor que uno (condición interna de ser AVL), nos gustaría que Z3 nos construyera un árbol adecuado a esto. Es decir, pedirle a Z3 que nos resuelva una fórmula del estilo

```
(assert (= (difHeight t) true)),
```

siendo *difHeight* una supuesta función que compruebe la condición sobre las alturas mencionada anteriormente, y t una constante del tipo *Tree* definido como en la Sección 3.4, y que nos construya directamente t como nosotros queremos. Pues bien, nos enfrentamos de nuevo a la limitación comentada de Z3, puesto que no es capaz de realizar tal tarea.

Vistos entonces los problemas encontrados con este tipo de asertos, veamos cómo tratarlos para salvar estas dificultades. Como ya hacíamos con las restricciones de tamaño, nos limitaremos a tratar estas restricciones como elementos de verificación. De este modo, sí podremos verificar que un árbol verifique que cumple alguna cierta propiedad estructural.

Las restricciones estructurales acostumbran a venir incluidas dentro de predicados más generales, por ejemplo, un árbol AVL debe cumplir ciertas condiciones sobre las alturas de los hijos o un árbol rojinegro debe cumplir que las alturas negras de cada hijo sean iguales. Sin embargo, otros predicados son puramente estructurales, como es el caso de que un montículo sea zurdo. Para los primeros, las llamadas

a estas funciones nos verificarán internamente que se cumpla la restricción estructural requerida, devolviendo *unsat* en caso de no hacerlo. Si lo verifica, proseguirá con el resto de comprobaciones aunque podría fallar por otro lado por alguna otra restricción. Por su parte, para los segundos, la llamada comprobará exclusivamente la restricción estructural devolviéndonos directamente si se cumple o no. Ilustramos estas ideas con dos ejemplos. Si yo pido a Z3 que verifique la siguiente restricción

```
(assert (isAVL t)),
```

con t un árbol AVL y la función *isAVL* definida como en la Figura 3.44, comprobará aspectos sobre las alturas de los hijos, entre otras consideraciones. Sin embargo, si yo pido verificar una restricción como

```
(assert (isLeftist t)),
```

donde t es un árbol binario e *isLeftist* corresponde a la función definida en la Figura 3.36, estamos tratando con una función puramente estructural que solo hace comprobaciones sobre las alturas mínimas.

Concluimos las restricciones de estructura con algunos casos particulares, que son la altura negra de los árboles rojinegros y la altura de los árboles AVL. Al tratarse de alturas, estamos evidentemente ante restricciones estructurales, con las mismas limitaciones que el resto, es decir, no podemos construir un árbol desde cero a partir de una serie de aspectos sobre la altura negra. De este modo, por ejemplo para la primera, tendremos que utilizarla para verificar si una estructura de árbol rojinegro tiene una altura negra u otra y ver si se adapta a lo necesitado. Sin embargo, la altura negra, entre otras cosas analiza los colores de los nodos. Tales colores, si recordamos, eran un campo más de un árbol rojinegro, al que esperamos darle un valor. Es decir, los colores son parte del contenido del árbol. Por ello,estas restricciones referidas a la altura negra, si bien no pueden construir un árbol, podrán servirnos como restricciones de contenido, tal y como explicaremos a continuación, para dar valores a los colores. Lo mismo ocurre entonces con el campo de altura almacenado en un árbol AVL y las restricciones sobre la altura de estos árboles.

4.1.3. Restricciones de contenido

Trabajarán sobre el contenido de nuestra estructura, es decir, sobre los datos que la compongan. Estas restricciones, al contrario que las anteriores, nos permiten los dos métodos de usos comentados en la Sección 3.7. Podemos, por un lado, comprobar si una variable con ciertos valores guardados cumple una restricción y, por otro lado, rellenar una estructura concreta. El enfoque verdaderamente interesante y novedoso es el segundo, y es el que aporta la potencia a este trabajo, pues nos servirá para crear todos los valores sin necesidad de filtrarlos después. Nos centraremos entonces en éste, analizando como rellenar estructuras. Comentaremos primero los casos elementales de contenido, para luego analizar las particularidades introducidas en las restricciones estructurales.

```
(declare-const u (Arr Int))
(declare-const v (Arr Int))

(assert (= (second u) 4))
(assert (= (second v) 4))

(assert (permut u v))
```

Figura 4.1: Comprobación de la permutación de dos arrays

```
(define-fun k!19 ((x!0 Int)) Int
    (ite (= x!0 2) 11
    (ite (= x!0 3) 7
    (ite (= x!0 1) 7
    (ite (= x!0 0) 7
        5)))))

(define-fun k!20 ((x!0 Int)) Int
    (ite (= x!0 2) 7
    (ite (= x!0 3) 7
    (ite (= x!0 1) 11
    (ite (= x!0 0) 7
        6)))))
```

Figura 4.2: Modelo obtenido tras ejecutar la función *permut*

Cuando hablamos de contenido, hablamos de los valores almacenados en arrays, listas o árboles. Distinguimos la forma de trabajar con arrays respecto a las otras dos. Y es que, al tener Z3 soporte para arrays, la forma de crear contenido puede hacerse de forma más directa que en los otros casos. Comentamos cada perspectiva.

Al trabajar con arrays, Z3 nos proporciona una representación interna en forma de función. Las funciones *sortedArr* y *permut* trabajan sobre el contenido de los arrays. Pues bien, simplemente llamando a la primera con un array de una longitud determinada, Z3 nos creará directamente los valores almacenados en cada posición del array verificando la propiedad de orden. Por su parte, para la segunda, si le pasamos dos arrays con una cierta longitud fijada para cada uno de ellos (misma longitud pues tiene que haber el mismo número de elementos), de nuevo Z3 crea los dos arrays correctamente. Pero además, no solo es capaz de crear ambos arrays de cero, sino que, en caso de que uno de los dos tenga ya valores y el otro no, puede rellenar las distintas posiciones del segundo con los elementos que conforman el primero, verificando que el array resultante sea permutación del de partida. Vemos en la Figura 4.1 qué ocurre al llamar a la función permutación. Los dos arrays se crean vacíos, y únicamente se les asigna una longitud a cada uno de ellos. El modelo obtenido (Figura 4.2) verifica que el array *u*, definido como la función *k!19*, tiene los mismos elementos que *v*, que corresponde a la función *k!20*, aunque en distinto orden, es decir, efectivamente uno es permutación del otro.

Por su parte, al trabajar con listas o árboles la tarea se vuelve algo más compleja. Al contrario que para los arrays, la definición de estos tipos no es interna de Z3 sino que nos la hemos creado nosotros con nuestros tipos algebraicos correspondientes. Por ello, para trabajar con cualquier elemento de uno de estos tipos necesitamos tener especificada previamente una estructura concreta. Esta estructura vendrá vacía, es decir, con los nodos sin ningún valor almacenado en ellos, sino con variables del tipo que corresponda según el campo, que son las que esperamos que Z3 nos instancie con valores adecuados. Por tanto, al contrario que con los arrays, en los que nos bastaba definir la variable y darle una longitud, aquí tendremos que darle a la variable concreta una cierta estructura.

Comentamos, por último, esa dualidad para algunas de las restricciones de estructura. Como hemos mencionado, aspectos como la altura de un AVL o la altura negra de un árbol rojinegro refieren a campos del propio tipo de datos. Por ello, podremos usarlas también para rellenar estos parámetros. De este modo, si a una cierta estructura de árbol de tipo LLRB le pedimos que compruebe que sea efectivamente rojinegro, es decir, llamamos a la función *isLlrb* de la Figura 3.55, entre las comprobaciones que realiza encontramos algunas que se refieren a los colores, y entre ellas la de las alturas negras de los hijos. Así, todas estas restricciones en conjunto nos devolverían valores para cada uno de los colores de los nodos. En la Figura 4.3, vemos que aparece declarado un árbol rojinegro, además de una serie de variables enteras y de color para construir la estructura del árbol. Pues bien, al pedir que compruebe la función *isLlrb*, obtenemos el modelo de la Figura 4.4, en el que podemos ver que el árbol ha quedado completamente instanciado, con todas las variables de color tomando un valor concreto. En la Figura 4.5 podemos ver la representación del modelo en forma de árbol, a fin de hacer más cómoda la interpretación de los valores obtenidos. El árbol obtenido es efectivamente un árbol rojinegro correcto.

4.2. Estrategia

Hasta ahora hemos visto cómo trabajar con cada tipo de restricción. A continuación, veremos cómo esta distinción nos facilitará la generación de casos gracias a la estrategia seguida. Podemos distinguir cuatro etapas: fijar un tamaño para los casos de prueba, generar estructuras de ese tamaño, aplicar las restricciones de estructura correspondientes a las estructuras generadas y por último popular tales estructuras. Con estas etapas lo que se pretende es salvar las limitaciones de Z3 a la hora de trabajar con las restricciones de tamaño y estructura y aprovechar después su potencia al tratar las de contenido. Como Z3 no es capaz de generar estructuras dado un cierto tamaño, las dos primeras etapas se realizan desde Haskell. Después, las estructuras creadas se pasan a Z3, donde se resuelven las siguientes dos etapas, consiguiendo poblar tales estructuras y obteniendo, con ello, nuestro caso de prueba final.

```
(declare-const t (LLRB Int))

(declare-const v1 Int)
(declare-const v2 Int)
(declare-const v3 Int)
(declare-const v4 Int)
(declare-const v5 Int)

(declare-const c1 Color)
(declare-const c2 Color)
(declare-const c3 Color)
(declare-const c4 Color)
(declare-const c5 Color)

(assert (= t (nodeL v1 c1
               (nodeL v2 c2 (nodeL v3 c3 leafL leafL)
                            (nodeL v4 c4 leafL leafL))
               (nodeL v5 c5 leafL leafL))))))

(assert (isLlrb t))
```

Figura 4.3: Rellenar un árbol rojinegro

La primera de las etapas es la más sencilla, pues solo requiere determinar el tamaño que tendrán cada una de nuestras posibles estructuras. Para ello, nuestro programa Haskell interactúa con el usuario pidiéndole que nos proporcione los tamaños que él desea para cada tipo de datos.

La etapa de generación de las estructuras resulta algo más compleja. En primer lugar requiere analizar las variables que participan en nuestra precondición para saber qué casos debemos generar. Para ello, primero deben extraerse todas las variables que recibe como parámetros de entrada nuestra función, puesto que estas serán las que puedan aparecer en los diferentes asertos de la precondición. Una vez extraídas todas las variables, solo nos interesa quedarnos con aquellas para las que tendremos que generar una cierta estructura. Es decir, descartamos todas las variables de tipo entero o *booleano*.

Conocidas ya las variables que nos interesan, procedemos a generar todas las estructuras del tipo correspondiente y con el tamaño adecuado. Generar *arrays* es sencillo puesto que solo requiere guardar su tamaño. Por su parte, generar listas sigue siendo relativamente fácil haciéndolo de forma recursiva, de modo que, si la longitud de la lista a crear es cero construimos la lista *nil*, mientras que si no es cero, creamos la lista mediante la constructora *cons*, y seguimos creando la cola recursivamente restándole uno a la longitud. Sin duda, la estructura más compleja que necesitamos crear es la de los árboles, ya sean binarios, AVL o rojinegros. Para crearlos, la idea seguida es la siguiente: dado el cardinal del árbol, tendremos que crear todos los posibles hijos izquierdos y derechos tales que sus respectivos cardinales sumen el

```
(define-fun v5 () Int
   2001)
(define-fun c2 () Color
   Negro)
(define-fun c3 () Color
   Rojo)
(define-fun c4 () Color
   Rojo)
(define-fun c5 () Color
   Negro)
(define-fun v2 () Int
   (- 1999))
(define-fun v4 () Int
   (- 1998))
(define-fun v3 () Int
   (- 2001))
(define-fun c1 () Color
   Negro)
(define-fun v1 () Int
   1999)
```

Figura 4.4: Modelo devuelto por Z3

número total de nodos esperado menos uno (menos uno, ya que uno de los nodos ya lo consumimos en la raíz). Por ejemplo, si nos piden generar árboles de cardinal cuatro, tendremos que generar hijos izquierdos de cardinales cero, uno, dos y tres, y del mismo modo hijos derechos de cardinales cuatro, tres, dos y uno, respectivamente. A partir de ahí, el resto se reduce a ir aplicando esta idea recursivamente hasta llegar a crear todo el árbol entero, de modo que cuando pidamos crear un árbol de cardinal cero nos encontramos con el caso base de nuestro problema y creamos un árbol vacío *leaf*. Para ilustrar esto con un ejemplo, en la Figura 4.6 podemos ver la lista resultante de crear árboles binarios de cardinal dos. Observamos que respeta la idea comentada previamente puesto que nos crea dos árboles, uno de ellos con hijo izquierdo vacío y un solo nodo en el hijo derecho, y otro al revés, con el hijo derecho vacío y el izquierdo con un único nodo.

Así, tendríamos casi creadas todas las estructuras. No obstante, si recordamos la definición de los tipos algebraicos vistos en el Capítulo 3, cada uno de ellos contaba con un campo para el valor en cada nodo de la lista o árbol, además de un campo para la altura o el color en caso de que se trate de árboles AVL o rojinegros respectivamente. Por ello, al generar nuestras estructuras también debemos tener en cuenta la creación de cada uno de estos campos. Para evitar que se produzcan coincidencias con los nombres de cada uno de estos parámetros de las estructuras, las crearemos a partir del nombre de la variable para la cual estamos generando la estructura, que debe ser único, añadiéndole un subíndice o etiqueta según la profundidad de nuestro nodo.

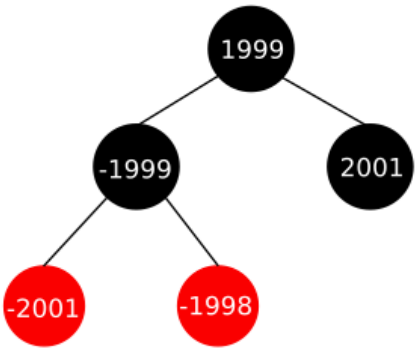


Figura 4.5: Modelo en forma de árbol

```
[node a leaf (node b leaf leaf),
 node a (node b leaf leaf) leaf]
```

Figura 4.6: Creación de árboles de cardinal dos

Con esto quedan resueltas las dos primeras etapas, en las que hemos obtenido los tamaños de nuestras variables y después hemos generado estructuras para todas ellas. A partir de aquí, el resto del trabajo es tarea de Z3. Para ello, es necesario volcar todas estas estructuras generadas en un fichero de tipo *smt*, en el que incluyamos también las restricciones de las precondiciones a verificar para después ejecutarlo y obtener los modelos a analizar. El trabajo de crear el fichero y escribir en él todas las funciones, las declaraciones de tipos y las restricciones se realiza también desde el propio programa Haskell. Antes de entrar en las siguientes dos etapas, conviene analizar entonces cómo escribir tal fichero, pues es la tarea más compleja que queda por completar.

Teniendo las estructuras creadas y almacenadas en diferentes listas del programa Haskell, podríamos limitarnos a recorrerlas e ir escribiéndolas en el fichero de salida, haciendo las asignaciones a las variables según corresponda. Sin embargo, los distintos casos a ejecutar no salen de recorrer linealmente todas las estructuras, sino que deben surgir de combinar todas las listas con todas nuestras estructuras. Por ejemplo, supongamos que tenemos por un lado una lista con la siguiente estructura de lista: `[(cons l1 (cons l2 (cons l3 nil))]`; y que por otro lado tenemos la lista de árboles `[(node a (node b leaf leaf) leaf), (node a leaf (node b leaf leaf))]`. Entonces, las estructuras a probar surgen de combinar ambas listas. De este modo, la lista resultante sería `[[(cons l1 (cons l2 (cons l3 nil))), (node a (node b leaf leaf) leaf)], [(cons l1 (cons l2 (cons l3 nil))), (node a leaf (node b leaf leaf))]]`, obteniendo así dos posibles asignaciones para las variables de nuestro programa. Gracias a la función de la Figura 4.7 podemos levar a cabo la combinación de todas las listas obtenidas previamente, formando nuevas listas con las estructuras de cada una de las variables ya mezcladas.

```
combinarListas:: [[a]] -> [[a]]
combinarListas [] = [[]]
combinarListas (x:xs) = [x:y | x<-x, y<-(combinarListas xs)]
```

Figura 4.7: Combinación de los elementos de una serie de listas

```
(assert (= t (node a (node b leaf leaf) leaf)))
(assert (= t (node a leaf (node b leaf leaf))))
(assert (isAVL t))
```

Figura 4.8: Asignación incorrecta de las estructuras a su variable

Una vez tenemos esta lista final, ya sí podemos proceder a escribir todo en nuestro fichero Z3. En primer lugar, se declaran las variables que se utilizan en nuestra precondición y a las que después asignaremos cada estructura. Además de estas variables, hay que declarar todas aquellas que hayamos utilizado para la generación de cada una de las estructuras, pues son, al fin y al cabo, a las que después Z3 tendrá que dar un valor. Tras esto, toca asignar a cada una de las variables las estructuras que hemos creado para ellas. Si tuviéramos por ejemplo una variable `t` de tipo árbol, con las dos siguientes posibles estructuras `(node a (node b leaf leaf) leaf)` y `(node a leaf (node b leaf leaf))`, hacer una secuencia de restricciones del estilo de la Figura 4.8 nos haría imposible obtener casos de prueba puesto que es imposible satisfacer que ambas estructuras sean a la vez AVL. Por ello, tenemos que ser capaces de considerar cada una de las estructuras por separado sin que las anteriores condicionen los resultados de la actual. Para ello, usamos la funcionalidad que Z3 pone a nuestra disposición para apilar y desapilar restricciones mediante las instrucciones *(push)* y *(pop)*, respectivamente. Así, la forma adecuada de plantear las restricciones puede verse en la Figura 4.9. Mediante esta distribución de las restricciones, Z3 primero trataría de verificar que la primera estructura es AVL, devolviéndonos unos posibles valores de las variables a y b, y luego haría lo propio con la segunda pero sin que tengan ninguna influencia los valores obtenidos anteriormente. Por último, haciendo un análisis del árbol abstracto de nuestro programa a verificar, podemos obtener de manera sencilla las precondiciones y pasarlas a nuestro fichero en forma de restricciones Z3. De nuevo, como se referirán a una serie de variables que pueden tomar varios valores, deberemos incluir estas restricciones también en los bloques *push-pop* como ocurre en la Figura 4.9 con la restricción *isAVL*. Este análisis del árbol abstracto solo requiere ir analizando cada uno de los elementos que podamos encontrarnos, escribiéndolos en un *string* que se ajuste al estándar SMT-LIB.

Hemos conseguido entonces generar el fichero *smt* con todas las estructuras consideradas y todas las restricciones requeridas. A partir de aquí, podemos proceder a realizar las dos últimas etapas de nuestra estrategia. Tales etapas consistían en verificar las estructuras y poblarlas después. Aunque las presentamos como fases diferentes, pues, a priori, refieren a estructuras de tipos distintos, estructurales las de la primera de las etapas y de contenido las de la segunda, tal y como vimos

```
(push)
(assert (= t (node a (node b leaf leaf) leaf)))
(assert (isAVL t))
(push)

(push)
(assert (= t (node a leaf (node b leaf leaf))))
(assert (isAVL t))
(pop)
```

Figura 4.9: Uso de *push* y *pop* para asignar las estructuras a las variables

cuando comentábamos las estructuras hay muchos predicados que están a caballo entre unas y otras. Por tanto, estas dos fases se resuelven a la par ejecutando en Z3 el fichero *smt* obtenido. Con la ejecución del fichero, lo que estamos pidiendo a Z3 es que trate de verificar las diferentes restricciones, devolviéndonos *sat* en caso de lograrlo y *unsat* en caso contrario. Si recibimos como respuesta a la ejecución que es insatisfactible, querrá decir que alguna de las restricciones estructurales ha fallado, o que, al intentar rellenar el contenido de algunas variables no ha sido capaz. Este segundo caso se dará con variables como las de color de los árboles rojinegros, que al fin y al cabo, aunque las manejamos como si fueran elementos de contenido, hacen referencia a la estructura del árbol, de modo que, de nuevo, ser insatisfactible por este motivo nos lleva a que la estructura era inadecuada. Por su parte, si el resultado obtenido es *sat*, querrá decir que la estructura era correcta y ha podido encontrar un modelo que verifique todas las restricciones. Pidiendo entonces a Z3 que nos proporcione el modelo obtenido, conseguimos las distintas asignaciones para cada una de nuestras variables, es decir, poblamos todas nuestras estructuras.

En resumen, partiendo de una serie de tamaños indicados por el usuario, somos capaces de generar una serie de estructuras para las variables que nos interesan, volcar toda esta información en un fichero *smt* que se ejecuta mediante Z3, obteniendo así la satisfactibilidad de nuestras estructuras y los modelos que las conforman en caso de que sean satisfactibles. Analizando estos modelos, conseguimos directamente cada uno de nuestros casos de prueba, que sabemos que se ajustan necesariamente a nuestras precondiciones y que son, por tanto, casos de prueba válidos para la función que deseábamos verificar.

Capítulo 5

Experimentos

Hasta ahora, nos hemos dedicado a explicar, primero la implementación en Z3 de las diferentes funciones que pueden resultar interesantes a la hora de tratar los asertos, y después la estrategia seguida para transformar la precondición de una cierta función, dada en su representación intermedia, en un fichero *smt* para ejecutarlo y obtener a partir de él nuestros casos de prueba. A lo largo de este capítulo, cubriremos diferentes funciones que afecten a todas las estructuras estudiadas en este trabajo, mostraremos los casos de prueba obtenidos para cada una de ellas, y analizaremos tales resultados a fin de demostrar la eficacia de nuestro sistema.

Las diferentes estructuras de datos comentadas a lo largo del Capítulo 3 recordamos que eran arrays, listas, árboles binarios, árboles AVL y árboles rojinegros. Dentro de cada una de ellas, las funciones principales eran: *sortedArr* y *permutArr* para arrays; *sortedList* y *member* para listas; *isHeap*, *isLeftist* e *isBST* para árboles binarios, siendo las dos primeras para trabajar con montículos y la tercera con árboles de búsqueda; *isAVL* para los árboles AVL; e *isLLRB* para los rojinegros. Por ello, las precondiciones que hemos puesto a prueba para analizar los resultados incluyen todos estos predicados.

Si recordamos la idea comentada en el Capítulo 4, lo que se busca es tratar en Z3 las restricciones de contenido con la intención de poblar las diferentes estructuras. De este modo, Z3 puede tomar cualquier valor entero para dar valor a las diferentes variables de contenido de cada estructura. Para limitar un poco este rango y que no puedan tomar infinitos valores, establecemos por defecto que cada variable debe estar contenida en el intervalo $(-5, 5)$.

5.1. Listas

Como ya hemos mencionado, los predicados más importantes que hemos implementado en Z3, referentes a las listas, son *sorted*, para listas ordenadas y *member* para conocer si un elemento pertenece a una lista. Ambos predicados constituyen las precondiciones de las dos funciones que hemos probado sobre listas, reflejadas en la Tabla 5.1. Analizaremos por separado cada una de las funciones, comentando en

Función	Entrada	Precondición	Descripción
InsertList	$x : Int, l : Lst$	$\{sortedList(l)\}$	Inserta el elemento x ordenadamente en l
DeleteList	$x : Int, l : Lst$	$\{member(x, l)\}$	Elimina el elemento x de la lista l

Tabla 5.1: Funciones para listas

cada caso los resultados obtenidos para diferentes tamaños de lista.

Sorted

Nuestra intención aquí es poblar diferentes estructuras de listas con una longitud dada, cumpliendo que la lista resultante esté ordenada. Con el fin de aportar una cantidad de ejemplos suficientes, trabajaremos listas de cardinales desde dos hasta seis. Enumeramos a continuación todos los resultados obtenidos para cada uno de esos tamaños:

- `(cons 0 (cons 1 nil))`
- `(cons (- 1) (cons 0 (cons 1 nil)))`
- `(cons 0 (cons 1 (cons 2 (cons 3 nil))))`
- `(cons 0 (cons 1 (cons 2 (cons 3 (cons 4 nil)))))`
- `(cons (- 1) (cons 0 (cons 1 (cons 2 (cons 3 (cons 4 nil))))))`

Puede verse que en todos ellos, la lista generada está ordenada, de modo que Z3 está poblando correctamente nuestras estructuras de lista.

Member

Si en el caso anterior buscábamos listas ordenadas, aquí el orden de los elementos no nos interesa, lo único importante es que el elemento x pertenezca a la lista. Por tanto, Z3 no solo tiene la tarea de generar la lista, sino que además deberá instanciar esa variable x con un cierto valor y asegurar que éste sea uno de los elementos que la conforman. De nuevo, estudiaremos las listas generadas según los tamaños elegidos. Quedan enumeradas a continuación todas estas listas junto al valor asignado a la variable x:

Función	Entrada	Precondición	Descripción
InsertA	$x : Int, m : Int, a : Array$	$\{0 \leq m < length(a) \wedge sortedArr(a, 0, m)\}$	Inserta el elemento x en el array a

Tabla 5.2: Funciones para arrays

- `x=0: (cons 0 (cons (- 2) nil))`
- `x=0: (cons 0 (cons (- 2) (cons 2 nil)))`
- `x=0: (cons 0 (cons (- 2) (cons 2 (cons 3 nil))))`
- `x=0: (cons 0 (cons (- 2) (cons 2 (cons 3 (cons 3 nil)))))`
- `x=0: (cons 0 (cons (- 2) (cons 2 (cons 3 (cons 3 (cons 4 nil))))))`

En todos los casos encontramos que el valor asignado a x es cero, y que este valor se encuentra en todas las listas. Por ello, el modelo proporcionado por Z3 para cada tamaño es también correcto en este caso.

A fin de probar algún resultado diferente, añadimos manualmente una restricción en la que forcemos a la x a tomar un valor diferente de cero o uno. El modelo que proporciona entonces Z3, para una lista de cinco elementos, es:

- `x=2: (cons 2 (cons 0 (cons (-2) (cons 2 (cons 3 nil)))))`

5.2. Arrays

El predicado más importante que nos encontramos para arrays es *sortedArr*, que comprueba si un cierto array está ordenado entre dos posiciones dadas. Para probar tal predicado, hemos usado la función de inserción que aparece en la Tabla 5.2. Vemos que la función recibe como parámetro un array, el elemento a insertar en él y la posición en la que hacerlo (m). La precondición requiere que la posición en la que insertar esté dentro de los límites del array, y que éste esté ordenado entre el principio y dicha posición. De este modo, dado que la precondición afecta tanto al array como al parámetro de entrada m, Z3 deberá encontrar valores adecuados para ambos. Comentaremos algunos resultados obtenidos para diferentes longitudes del array de entrada.

En primer lugar consideramos como longitud del array cuatro. El modelo devuelto por Z3 puede verse en la Figura 5.1. Vemos que el array viene definido a partir de la función *k!0*. Esta función devuelve: -2 si recibe un 0; 2 si recibe un 1; y 3 en cualquier otro caso. Además, la variable m ha tomado el valor cero. Esto quiere decir que el array debe estar ordenado entre las posiciones cero y cero, lo que es trivial y se cumple sea cual sea el valor que le haya dado al array, así que, en particular, nuestros valores lo cumplen.

```
(define-fun a () (Pair (Array Int Int) Int)
    (mk-pair (_ as-array k!0) 4))
(define-fun m () Int
    0)

(define-fun k!0 ((x!0 Int)) Int
    (ite (= x!0 1) 2
    (ite (= x!0 0) (- 2)
        3)))
```

Figura 5.1: Modelo para array de longitud cuatro

```
(define-fun a () (Pair (Array Int Int) Int)
    (mk-pair (_ as-array k!0) 5))
(define-fun m () Int
    0)

(define-fun k!0 ((x!0 Int)) Int
    (ite (= x!0 1) 2
    (ite (= x!0 0) (- 2)
    (ite (= x!0 4) 4
        3)))
```

Figura 5.2: Modelo para array de longitud cinco

Vamos a considerar ahora que la longitud del array es cinco. En este caso, el modelo que nos proporciona Z3 es el de la Figura 5.2. De nuevo, la m toma el valor cero, así que razonando del mismo modo que hemos hecho antes, podemos afirmar que el array proporcionado por Z3 es correcto.

Que Z3 instancie la m siempre a cero es un caso que repite cualquiera que sea la longitud que le pongamos al array. Estos casos, como ya hemos visto son triviales así que vamos a forzar a buscar algún caso algo más complicado. Para ello, tomamos arrays de longitud seis y, añadiremos manualmente dos restricciones con las que prohibir a Z3 que le de a la variable m los valores cero o uno. Con todo esto, el modelo que nos proporciona el resolutor es el que aparece en la Figura 5.3. En este caso, Z3 ha elegido cinco como valor para la m. Por tanto, la precondición pide en este caso que el array completo esté ordenado. Si nos fijamos en la función *k!0* que es la que define a nuestro array, vemos que devuelve -1 en caso de recibir como entrada un cero, un 3 en caso de recibir un 5, y un 0 en cualquier otro caso. Si representamos esta función en forma de array como en la Figura 5.4, se ve fácilmente que el array cumple que está ordenado.

```
(define-fun a () (Pair (Array Int Int) Int)
    (mk-pair (_ as-array k!0) 6))
(define-fun m () Int
    5)

(define-fun k!0 ((x!0 Int)) Int
    (ite (= x!0 5) 3
    (ite (= x!0 0) (- 1)
        0)))
```

Figura 5.3: Modelo para array de longitud seis con m distinto de cero o uno

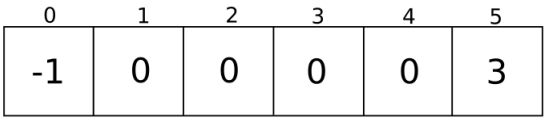


Figura 5.4: Modelo de array de cardinal seis

5.3. Árboles Binarios

Los árboles binarios sirven para representar diferentes estructuras. De entre las más interesantes, las que decidimos tratar en el Capítulo 3 fueron los montículos, concretamente los zurdos, y los árboles de búsqueda (BST). Por ello, las funciones estudiadas a lo largo de esta sección referirán a ambos tipos.

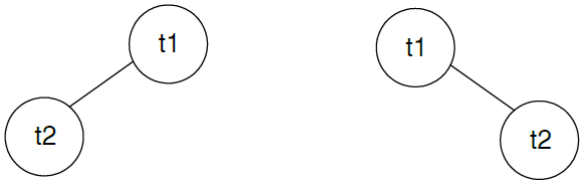
5.3.1. Montículos Zurdos

Dentro de los distintos tipos de montículos, uno de los más interesantes a probar son los zurdos, puesto que nos aportan una clara separación entre propiedades estructurales, referidas a las alturas mínimas, y de contenido, sobre el orden de los elementos dentro del árbol. De este modo, los predicados más importantes a la hora de trabajar con estos montículos son *isLeftist*, encargado de comprobar si la estructura es adecuada, e *isHeap*, que hará lo propio con el contenido. Por tanto, las funciones que vamos a analizar incluyen en sus precondiciones tales predicados, como puede verse en la Tabla 5.3.

Dado que los predicados de ambas precondiciones son los mismos, utilizaremos la primera para analizar los resultados obtenidos para diferentes cardinales, y después mostraremos algunos de los modelos proporcionados para la segunda, a fin de ver el comportamiento de Z3 al trabajar con dos estructuras a la vez. Así, para la primera función comentaremos primero los modelos para cardinales dos y tres, analizaremos algunos de cardinales superiores y veremos al final una estadística de casos aceptados y rechazados. Por su parte, para la segunda, aplicaremos a las dos estructuras que participan los distintos razonamientos expuestos para la anterior función, viendo algunos resultados para cardinales diferentes.

Función	Entrada	Precondición	Descripción
InsertLeft	$x : Int, t : Tree$	$\{isLeftist(t) \wedge isHeap(t)\}$	Inserta x en el montículo t
UnionLeft	$t1 : Tree, t2 : Tree$	$\{(isLeftist(t1) \wedge isHeap(t1)) \wedge (isLeftist(t2) \wedge isHeap(t2))\}$	Une dos montículos zurdos

Tabla 5.3: Funciones para montículos zurdos



(a) (b)

Figura 5.5: Estructuras de un árbol binario de dos nodos

Antes de entrar a comentar los resultados, conviene recordar las dos restricciones que deben cumplirse. En primer lugar, debe ocurrir que la altura mínima del hijo izquierdo debe ser mayor o igual que la del hijo derecho y, en segundo lugar, el valor almacenado en la raíz debe ser menor o igual que el de sus hijos.

Cuando trabajamos con árboles de cardinal dos, únicamente podemos encontrarnos dos estructuras diferentes, que aparecen en la Figura 5.5. Resulta sencillo ver que la primera de las estructuras sí que cumple la restricción estructural requerida, puesto que la altura mínima del hijo izquierdo es uno mientras que la del derecho es cero, pero que, por el contrario, la segunda la incumple ya que la altura mínima del hijo derecho es uno, que es mayor que la del hijo izquierdo que es cero. Como era de esperar, al ejecutar nuestras restricciones en Z3 obtenemos que efectivamente el caso correspondiente a la segunda estructura es insatisfactible. Por su parte, para el primero obtenemos el modelo `(node 0 (node 3 leaf leaf) leaf)`, que cumple que la raíz es menor que el hijo izquierdo, como deseábamos. Puede verse la representación del modelo obtenido en forma de árbol en la Figura 5.6.

Al trabajar con cardinal tres, el número de estructuras posibles crece a cinco, representadas todas ellas en la Figura 5.7. Dado que la condición estructural que tiene que cumplirse requiere que la altura mínima del hijo izquierdo sea mayor o igual que la del derecho, vemos rápidamente que las estructuras de las Figuras 5.7d y 5.7e no podrán satisfacer nuestra precondición. Pero además, la propiedad para ser zurdo era recursiva, de modo que los hijos también deben cumplirla. Si nos fijamos en la estructura de la Figura 5.7b, vemos que para el hijo izquierdo, sus respectivos hijos incumplen la propiedad sobre las alturas mínimas. Pues bien, al pasar nuestro fichero de restricciones por Z3, obtenemos *unsat* como resultado para estas tres estructuras y *sat* para el resto. Así, para estas estructuras satisfactibles obtenemos

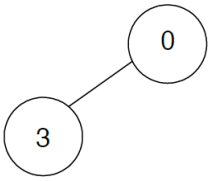


Figura 5.6: Modelo para montículo zurdo de cardinal dos

además los modelos `(node (-3) (node (-3) (node 0 leaf leaf) leaf) leaf)` para la Figura 5.7a y `(node 0 (node 2 leaf leaf) (node 4 leaf leaf))` para la Figura 5.7c. Ambos modelos obtenidos aparecen reflejados en la Figura 5.8

A partir de aquí veremos solo algunos ejemplos interesantes de modelos obtenidos para cardinales superiores. En la Figura 5.9a se muestra un modelo devuelto por Z3 para una estructura de cardinal cuatro. En primer lugar, verifica evidentemente la propiedad estructural, puesto que si no fuera así no habría podido resolverla. Es una estructura interesante puesto que, al tener dos nodos en el hijo derecho y solo uno en el izquierdo podría llevarnos a engaño. Sin embargo, la altura mínima del hijo derecho es uno, igual que la del izquierdo. En segundo lugar, vemos que los valores asignados a cada nodo cumplen ser menores o iguales que los de sus respectivos hijos. Con todo ello, el modelo obtenido es correcto. Por su parte, en las Figuras 5.9b y 5.9c aparecen dos nuevos modelos, uno de cardinal cinco y otro de cardinal seis. De nuevo, si nos fijamos en los valores proporcionados por Z3 para rellenar cada uno de los nodos, vemos que ambos modelos vuelven a ser correctos puesto que respetan el orden requerido, al cumplirse siempre que la raíz es menor o igual que sus hijos.

Para la segunda función, la única diferencia es que en vez de trabajar sobre una única estructura lo haremos sobre dos, y en consecuencia, la precondición afecta a ambas, comprobando que sean montículos zurdos. Así, las condiciones a cumplir son exactamente las mismas, por lo que los razonamientos a la hora de identificar estructuras válidas son totalmente análogos a los realizados hasta ahora con la primera función. De este modo, si por ejemplo, tenemos un caso en el que se combinan las estructuras de las Figuras 5.7c y 5.5b, la primera cumpliría las restricciones pero la segunda no, de manera que la restricción derivada de la precondición sería insatisfactible. Por el contrario, si por ejemplo se han combinado las estructuras de las Figuras 5.7a y 5.7c, al ser ambas válidas, como ya vimos previamente, la restricción en este caso sí sería satisfactible, y Z3 devolvería los modelos apropiados. A fin de mostrar un ejemplo con el que ilustrar que efectivamente rellena correctamente dos estructuras, en la Figura 5.10 puede verse el modelo obtenido para dos árboles, el primero de cardinal tres y el segundo de cardinal cuatro, ambos correctos tanto estructuralmente como en lo que refiere al orden de sus elementos.

Por último, en la Tabla 5.4 se muestran el número de casos satisfactibles e insatisfactibles obtenidos para cardinales cuatro, cinco y seis.

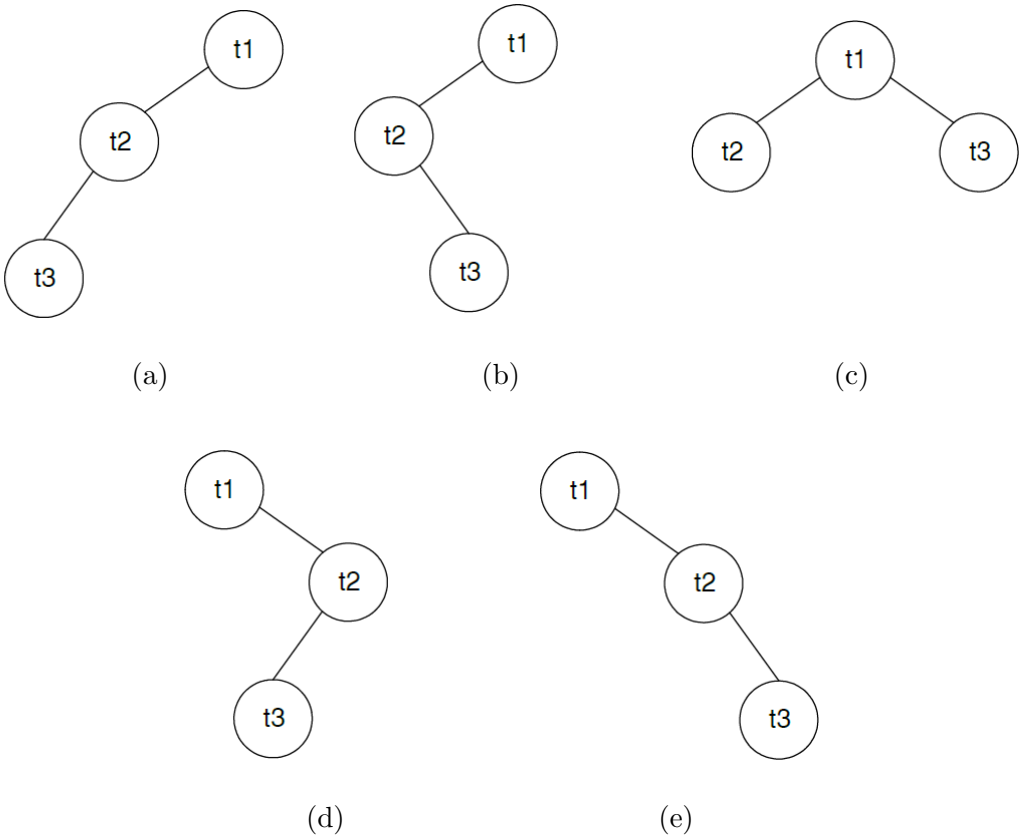
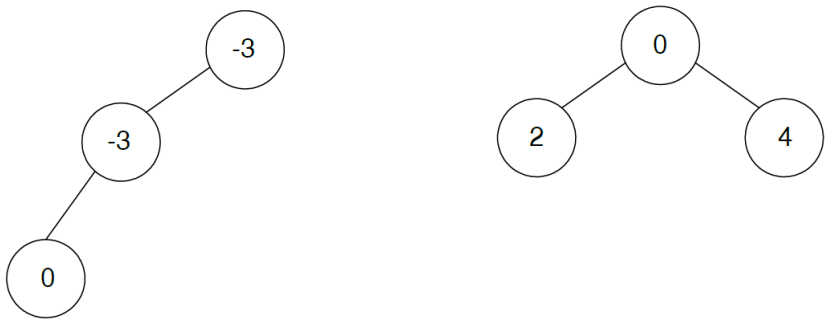


Figura 5.7: Estructuras de un árbol binario de tres nodos

5.3.2. Árboles de Búsqueda

Las funciones utilizadas para probar los árboles de búsqueda aparecen descritas en la Tabla 5.5. Todas ellas reciben como parámetro de entrada una variable t de tipo *Tree*, y comparten como precondición la llamada al predicado *isBST*, encargado de comprobar si un cierto árbol es o no de búsqueda. Para analizar los resultados obtenidos, mostraremos los modelos obtenidos para estructuras de cardinal dos y tres, luego comentaremos algunos modelos interesantes de tamaño cuatro o cinco. Si recordamos la función *isBST* de Z3, expuesta en el Capítulo 3, no presentaba ninguna restricción de carácter estructural, y únicamente se limitaba a comprobar que los elementos estuvieran ordenados de manera adecuada. Por ello, para todas las estructuras obtendremos que la restricción *isBST* es satisfactible, junto a un modelo adecuado para cada una de ellas. Por tanto, no entraremos a analizar la posible satisfactibilidad de las diferentes estructuras, puesto que todas van a serlo, sino que únicamente estudiaremos los resultados obtenidos a fin de ver si cumplen lo esperado.

Para árboles binarios de cardinal dos, encontramos únicamente dos posibles estructuras. Al ejecutar Z3, los modelos obtenidos son `(node 1 (node 0 leaf leaf) leaf)` para la estructura de la Figura 5.5a y `(node (- 1) leaf (node 0 leaf leaf))` para la estructura de la Figura 5.5b. En ambos casos, los valores asig-



(a) (b)

Figura 5.8: Montículos zurdos de tres nodos

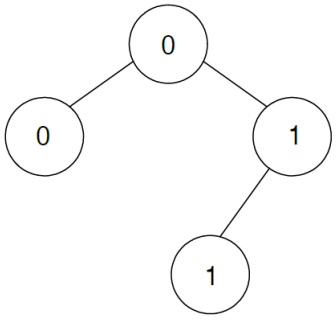
Cardinal	Casos Satisfactibles	Casos Insatisfactibles
Cuatro	4	10
Cinco	8	34
Seis	17	115

Tabla 5.4: Resultados obtenidos para montículos zurdos

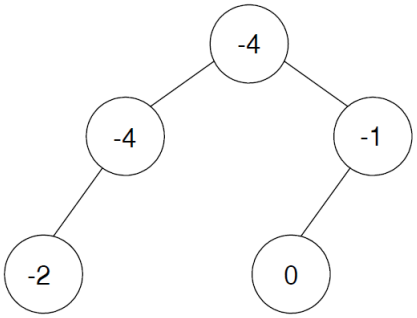
nados a los nodos son correctos puesto que, en el primero, la raíz es mayor que el hijo izquierdo, y en el segundo, la raíz es menor que el hijo derecho. Tales modelos pueden verse representados en forma de árbol en la Figura 5.11.

Cuando el cardinal es tres, pasamos de tener dos posibles estructuras a cinco (Figura 5.7), todas ellas satisfactibles como comentábamos previamente. Así, el resultado de ejecutar Z3 sobre nuestras restricciones, son cinco modelos diferentes para las cinco estructuras. Estos modelos quedan reflejados en la Figura 5.12, en la que podemos ver todas las estructuras de cardinal tres pobladas con diferentes valores. Vemos que en cada una de ellas, si descendemos por los diferentes subárboles, siempre se cumple que el hijo izquierdo sea menor que la raíz, y ésta a su vez sea menor que el hijo derecho. Por tanto, todos los modelos proporcionados por Z3 son correctos.

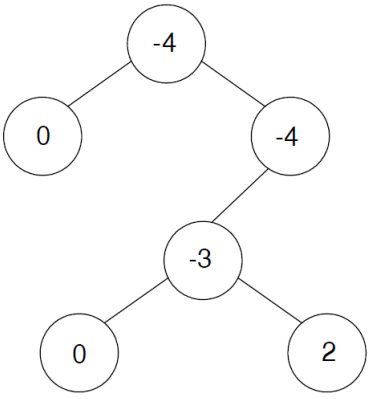
A partir de aquí, el número de estructuras generadas crece notablemente, por lo que solo mostraremos algunos ejemplos de cardinales cuatro y cinco con los que terminar de confirmar que Z3 está poblando correctamente nuestros árboles. En la Figura 5.13 encontramos tres posibles modelos para cardinal cuatro, mientras que en la Figura 5.14 aparecen dos alternativas para cardinal cinco. En todas ellas, de nuevo, se verifica que el orden de los elementos es correcto.



(a) Cardinal cuatro



(b) Cardinal cinco



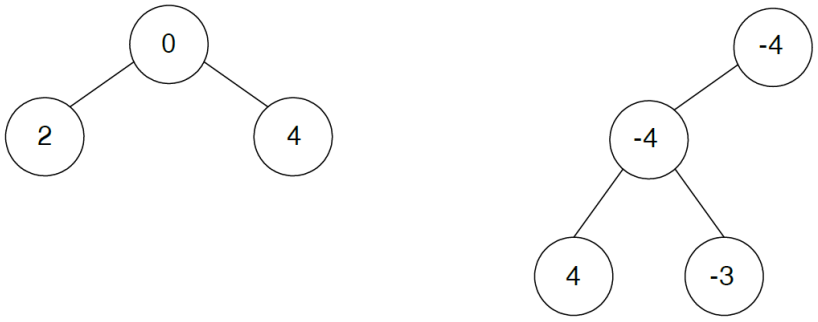
(c) Cardinal seis

Figura 5.9: Montículos zurdos de cardinales cuatro, cinco y seis

5.4. Árboles AVL

Para probar los árboles AVL hemos utilizado las funciones de inserción de un elemento, búsqueda de un elemento, y borrado de un elemento. En la Tabla 5.6 quedan recogidas todas estas funciones con sus respectivas precondiciones, junto a algunos datos adicionales. Puede verse que en todas ellas se recibe como entrada de la función un árbol de tipo AVL, sobre el que actúan todas las precondiciones requiriendo que sea, efectivamente, un AVL. Dado que todas las precondiciones son entonces iguales, no separaremos por casos sino que analizaremos todas por igual, comentando los resultados obtenidos para diferentes tamaños de entrada. Comenzaremos analizando los resultados para tamaños dos y tres, comentaremos algunos casos representativos para tamaños mayores y, por último, para tamaños mayores analizaremos el número de estructuras rechazadas y aceptadas, a fin de saber cuantas se pueden poblar de todas las que se generan.

Los árboles AVL contaban con un campo adicional para almacenar la altura de cada nodo, por lo que, para ver que el árbol creado por Z3 es correcto, tendremos que fijarnos tanto en el orden de los elementos como en este dato. Además, conviene recordar que la propiedad estructural que debe cumplir un árbol para ser AVL es



(a) (b)

Figura 5.10: Montículos zurdos de cardinales tres y cuatro para la función de unión

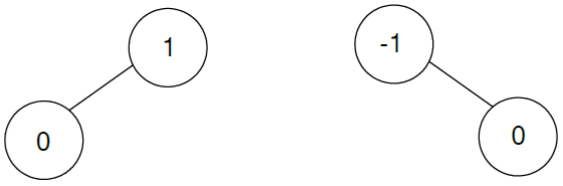
Función	Entrada	Precondición	Descripción
InsertBST	$x : Int, t : Tree$	$\{isBST(t)\}$	Inserta el elemento x en el árbol t
SearchBST	$x : Int, t : Tree$	$\{isBST(t)\}$	Busca el elemento x en el árbol t
DeleteBST	$x : Int, t : Tree$	$\{isBST(t)\}$	Elimina el elemento x del árbol t

Tabla 5.5: Funciones para árboles binarios de búsqueda

que la diferencia entre las alturas de los hijos no sea mayor que uno. A lo largo de la sección, cuando comentemos los casos descartados nos referiremos a esta propiedad.

Si tenemos árboles binarios de cardinal dos, únicamente podemos encontrarnos dos estructuras, una con la raíz y un nodo en el hijo izquierdo, y la otra con la raíz y un nodo en el derecho. Ambas estructuras verifican ser AVL, puesto que en los dos casos la diferencia de alturas de los hijos no es mayor que uno. Por ello, cabría esperar que, al pedir a Z3 que resuelva nuestras restricciones, obtuviéramos que todos los casos son satisfactibles, y nos proporcionara, por tanto, un modelo para nuestras estructuras verificando tales restricciones. En efecto, Z3 resuelve los dos casos como *sat* y devuelve el modelo `(nodeA 3 2 (nodeA 0 1 leafA leafA) leafA)` para el árbol con hijo derecho vacío (Figura 5.5a), y `(nodeA (- 1) 2 leafA (nodeA 0 1 leafA leafA))` para el de hijo izquierdo vacío (Figura 5.5b). En ambos casos, en cada nodo, el campo reservado para la altura toma los valores esperados (dos para la raíz y uno para el hijo), mientras que los valores almacenados respetan el orden requerido para ser árbol de búsqueda, puesto que, en el primer caso, la raíz es mayor que el hijo izquierdo, y en el segundo, la raíz es menor que el hijo derecho. En la Figura 5.15 se muestran ambos modelos en forma de árbol.

Con los árboles de cardinal tres empiezan a crecer el número de estructuras generadas, recogidas todas ellas en la Figura 5.7. De entre las cinco posibles estructuras, únicamente aquella con un nodo como hijo izquierdo y otro como hijo



(a) (b)

Figura 5.11: Modelos de árboles de búsqueda de cardinal dos

Función	Entrada	Precondición	Descripción
InsertAVL	$x : Int, t : AVL$	$\{isAVL(t)\}$	Inserta el elemento x en el árbol t
SearchAVL	$x : Int, t : AVL$	$\{isAVL(t)\}$	Busca el elemento x en el árbol t
DeleteAVL	$x : Int, t : AVL$	$\{isAVL(t)\}$	Elimina el elemento x del árbol t

Tabla 5.6: Funciones para árboles AVL

derecho (Figura 5.7c) cumple la restricción estructural requerida para los árboles AVL, puesto que ambos hijos tienen la misma altura. Sin embargo, en el resto de estructuras la diferencia entre las alturas de los hijos es siempre dos, incumpliendo así la restricción necesaria. Por ello, sería esperable que al ejecutar Z3 obtuviéramos que cuatro de los casos son insatisfactibles, y únicamente para el caso restante nos devolviera que es satisfactible, junto a un posible modelo. En la Figura 5.16 se muestra que, efectivamente, en la salida producida por Z3, para cuatro de los casos a verificar devuelve *unsat*, mientras que para el quinto devuelve *sat*. Por tanto, visto que descarta correctamente las estructuras, solo queda ver si el modelo devuelto es correcto. Tal modelo es `(nodeA 2 2 (nodeA 0 1 leafA leafA) (nodeA 3 1 leafA leafA))` (Figura 5.17). En él puede verse, primero, que las alturas asignadas a cada nodo son correctas, y segundo, que los valores almacenados son también correctos, ya que la raíz cumple ser mayor que el hijo izquierdo pero menor que el derecho.

Al trabajar con árboles de cardinal cuatro o cinco, el número de estructuras crece drásticamente. Por ello, en lugar de analizar todas ellas, mostraremos solo algunos ejemplos relevantes con los que terminar de demostrar que, efectivamente, Z3 está poblando bien nuestros árboles.

Para cardinal cuatro, encontramos entre los modelos devueltos el siguiente: `(nodeA 3 3 (nodeA 0 2 (nodeA (- 1) 1 leafA leafA) leafA) (nodeA 4 1 leafA leafA))`. En primer lugar, vemos que los valores de cada nodo están correctamente elegidos, ya que siempre se tiene que la raíz es mayor que el hijo izquierdo y menor que el derecho. En la Figura 5.18a puede verse el modelo en forma de árbol, con las correspondientes alturas asignadas a cada nodo. Comprobamos que la altura

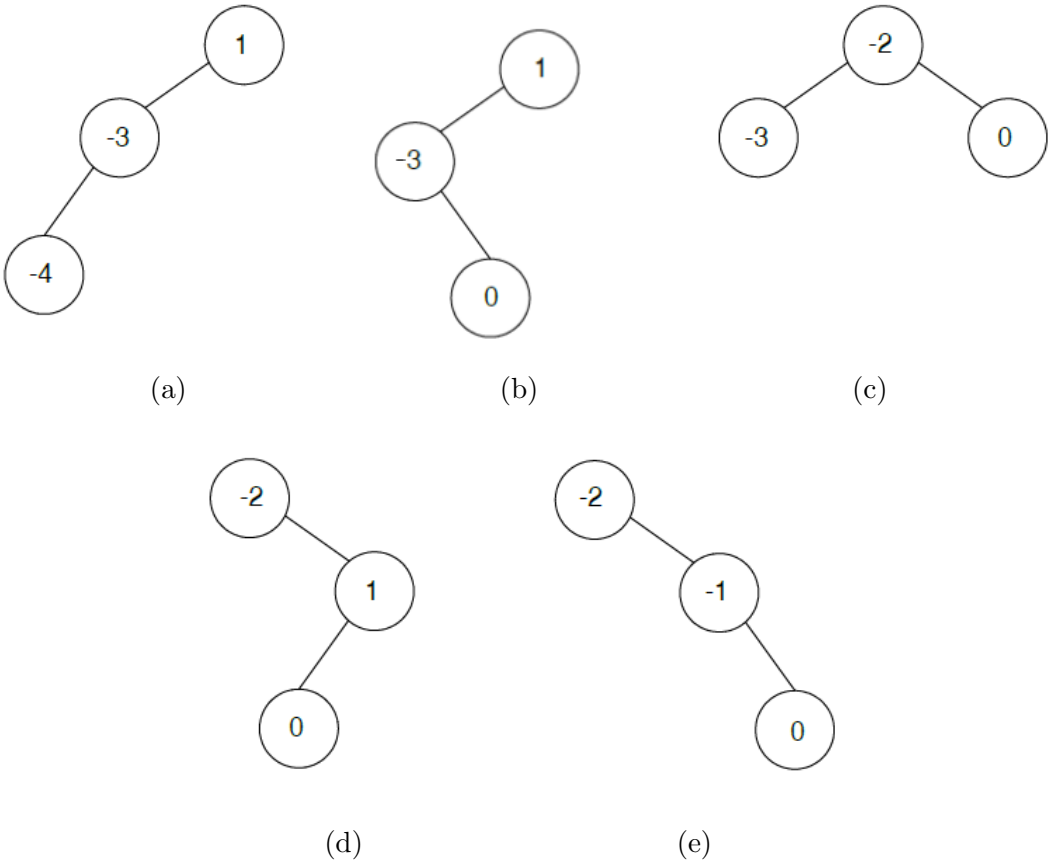


Figura 5.12: Modelos BST para cardinal tres

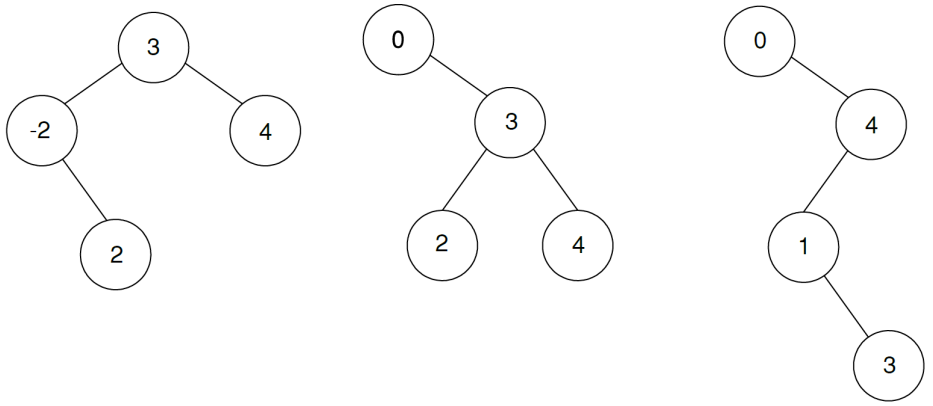
de cada nodo se corresponde con la asignada por Z3. En la Figura 5.18b se puede ver otro árbol diferente de cardinal cuatro, sobre el que es también sencillo comprobar que todos los parámetros rellenados son correctos.

Por otro lado, para cardinal cinco obtenemos como uno de los posibles modelos el representado en la Figura 5.18c. En él podemos ver que, tanto los valores de cada nodo, que respetan el orden requerido, como las alturas de éstos, son correctos. Otro posible modelo es el de la figura 5.18d, correcto también. Así, los modelos proporcionados por Z3 vuelven a ser acertados.

Hasta aquí hemos comentado entonces algunas de las estructuras para las que Z3 nos ha devuelto que el predicado *isAVL*, aplicado a ellas, es satisfactible. Veamos para terminar que aquellas para las que no ha devuelto tal resultado ha sido porque, efectivamente, incumplían alguna de las restricciones implícitas en dicho predicado. Consideremos una posible estructura de cardinal cinco, como por ejemplo:

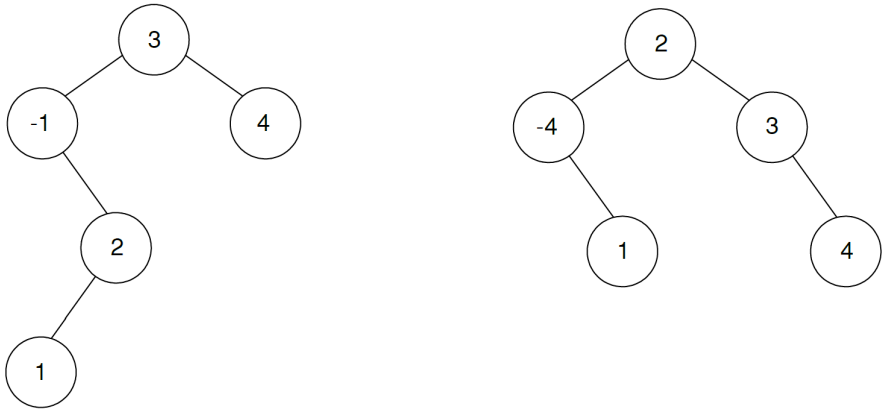
```
(node t1 th1 (node t2 th2 (node t3 th3 leafA (nodeA t4 th4 leafA
    leafA)) leafA) (nodeA t5 th5 leafA leafA)).
```

Pues bien, el hijo izquierdo, tiene altura tres, mientras que el derecho tiene únicamente altura uno, de modo que la diferencia de alturas entre ambos es mayor que



(a) (b) (c)

Figura 5.13: Modelos de árboles BST para cardinal cuatro

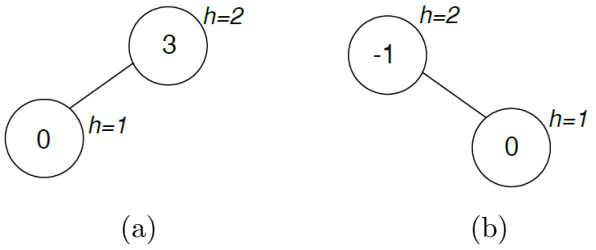


(a) (b)

Figura 5.14: Modelos de cardinal cinco para BST

uno. Además, el hijo izquierdo presenta a su vez una diferencia de alturas entre sus dos hijos mayor que uno. En conclusión, la estructura fue correctamente descartada. Razonando, de manera análoga, con cada una de las estructuras rechazadas, puede verse que se hizo correctamente.

Para concluir la sección, en la Tabla 5.7 se muestran los resultados obtenidos para cardinales cuatro, cinco y seis, en términos del número de casos satisfactibles e insatisfactibles. Llama la atención que disminuya el número de casos satisfactibles para cardinal seis, respecto a los de cardinal cinco. No obstante, al colocar un nuevo nodo puede ocurrir que estructuras que eran buenas dejen de serlo, puesto que, aunque tal vez el árbol en conjunto mantenga las mismas alturas en los hijos izquierdo y derecho, como el predicado *isAVL* requiere que los distintos subárboles también sean AVL, es en estos subárboles donde pasa a fallar la propiedad estructural sobre la diferencia de alturas. Además, puede verse que el número de casos satisfactibles es notablemente reducido. Esto parece razonable, puesto que todas las estructuras que combinen todos los nodos a la izquierda o todos a la de-



(a) (b)

Figura 5.15: Modelos para árboles AVL de cardinal dos

```
unsat
   (model is not available)
unsat
   (model is not available)
sat
   (model ...)
unsat
   (model is not available)
unsat
   (model is not available)
```

Figura 5.16: Satisfactibilidad devuelta por Z3 para árboles AVL de cardinal tres

recha son malas, y esto se repite recursivamente, según descendemos por los hijos, siempre que el número de nodos restantes a combinar sea mayor que dos.

5.5. Árboles Rojinegros

Igual que para árboles BST y AVL, las funciones utilizadas para poner a prueba la generación de casos para árboles rojinegros son insertar, borrar o buscar un cierto elemento. Observando la Tabla 5.8, en la que están recogidas las características de estas tres funciones, vemos que todas ellas comparten la misma precondición, a saber, que el árbol que reciben las funciones como entrada sea LLRB. El predicado *isLLRB* se encarga de comprobar tanto el coloreado del árbol, como el orden de los elementos almacenados. Para que el orden sea correcto, se requiere, de nuevo, que el árbol sea de búsqueda, por lo que la característica nueva a analizar para estos árboles es la del color de los nodos. Del mismo modo que en los casos anteriores, analizaremos todas las estructuras generadas para cardinales dos y tres, mostraremos y comentaremos algunas relevantes de cardinales superiores y, por último, presentaremos a modo de estadística el número de casos satisfactibles e insatisfactibles para cardinales cuatro, cinco y seis, a fin de ver cuantas estructuras son aceptadas o rechazadas.

Como ya hemos dicho, lo nuevo será analizar cómo Z3 colorea nuestros nodos respetando las condiciones requeridas por los árboles rojinegros para estar

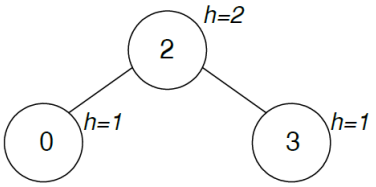


Figura 5.17: Modelo AVL de cardinal tres

Cardinal	Casos Satisfactibles	Casos Insatisfactibles
Cuatro	4	10
Cinco	6	36
Seis	4	128

Tabla 5.7: Resultados obtenidos para árboles AVL

correctamente coloreados. Por ello, antes de entrar en comentar los diferentes casos generados, parece conveniente detenernos en enumerar brevemente dichas condiciones, pues nos referiremos a ellas durante el análisis de los resultados obtenidos. Recordemos que para que un árbol esté correctamente coloreado tiene que darse que la raíz sea de color negro, que la altura negra de ambos hijos sea igual, que no haya dos nodos rojos seguidos, y que si el hijo derecho es rojo, entonces el izquierdo ha de ser rojo también. Además, recordar también que el color de un nodo vacío, *leafL*, es por defecto negro.

De nuevo, para cardinal dos, nos encontramos únicamente dos posibles estructuras (Figura 5.5). Cabría esperar que Z3 nos devolviera que la restricción para la estructura de la Figura 5.5a es satisfactible, mientras que para la de la Figura 5.5b es, por el contrario, insatisfactible. Y es que, para esta segunda, la única forma de conseguir que la altura negra sea igual para todos los hijos sería coloreando el hijo derecho de rojo, pero, sin embargo, esto incumple la última de las condiciones mencionadas previamente, puesto que un nodo de color rojo en el hijo derecho requiere que el hijo izquierdo sea rojo también, aspecto imposible de cumplir en este caso. Pues bien, Z3, efectivamente, nos devuelve que el predicado *isLLRB* aplicado al segundo árbol es insatisfactible, mientras que para el primero nos proporciona el modelo `(nodeL 1 Negro (nodeL 0 Rojo leafL leafL) leafL))`, recogido en forma de árbol en la Figura 5.19.

La Figura 5.7 recoge las cinco estructuras que podemos construir a partir de tres nodos. Al ejecutar nuestras restricciones con Z3, obtenemos la insatisfactibilidad de cuatro de las estructuras, siendo satisfactible únicamente el árbol de la Figura 5.7c. El modelo obtenido para dicho árbol es `(nodeL 1 Negro (nodeL 0 Negro leafL leafL) (nodeL 2 Negro leafL leafL)))` (Figura 5.20). Es un modelo correcto ya que, por un lado, los valores están correctamente ordenados dentro del árbol y, por otro lado, respeta todas las condiciones referidas a los colores, pues-

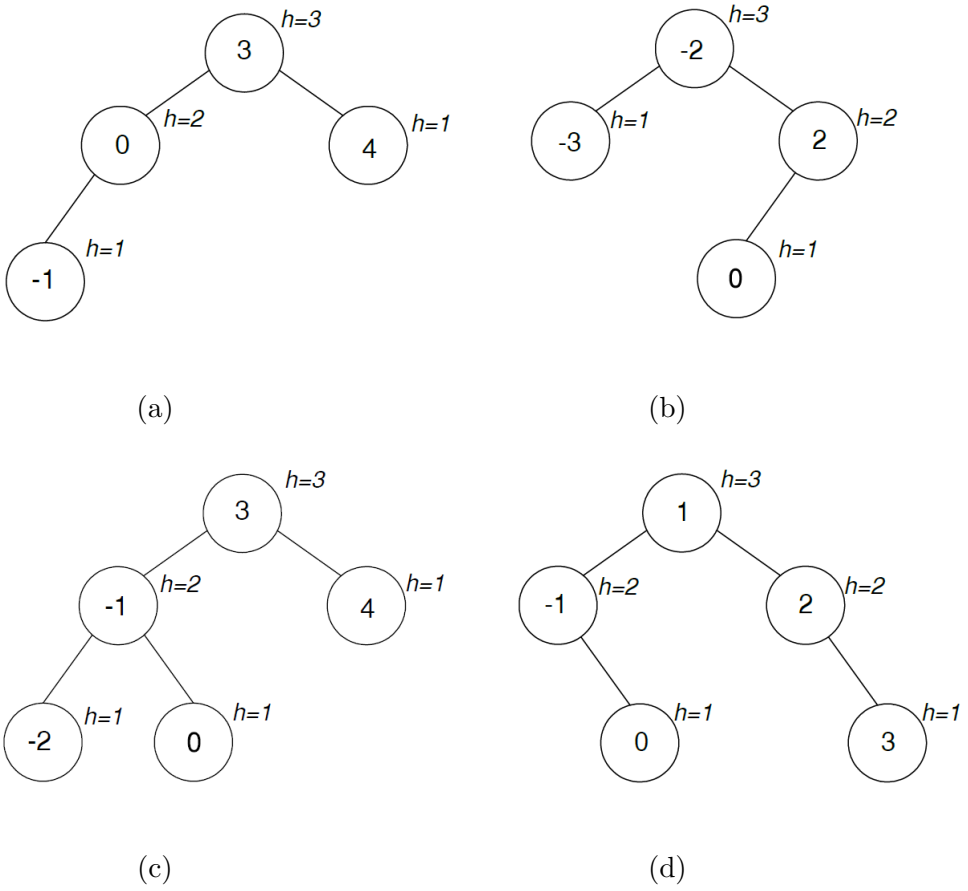


Figura 5.18: Modelos para árboles AVL de cardinales cuatro y cinco

to que las alturas negras son iguales y, al no haber ningún nodo rojo, no puede incumplir ninguna de los demás requisitos. Queda ver, por tanto, si para el resto de estructuras es correcto el resultado obtenido. Consideremos, por ejemplo, la estructura de la Figura 5.7a. Al tener todos los nodos a la izquierda de la raíz, en cuanto tengamos uno de ellos de color negro incumpliremos que las alturas negras de los dos hijos sean iguales. Por otro lado, si intenta respetarse esta condición, entonces necesariamente los dos nodos deberían colorearse de rojo, incumpliendo así la condición que nos dice que no puede haber dos nodos rojos seguidos. Por tanto, es correcto que Z3 nos haya devuelto, para este caso, que es insatisfactible. Para las otras tres estructuras para las que obteníamos el mismo resultado, el razonamiento es totalmente análogo.

Para concluir con el análisis de los árboles rojinegros, presentamos algunos casos interesantes con cardinales cuatro, cinco o seis, con los que ver como se comporta Z3 al tener que colorear estructuras más complejas. Del mismo modo, tomaremos algunas estructuras para las que obtenemos que el predicado a verificar es insatisfactible a fin de ver si, en efecto, es así.

En la Figura 5.21a podemos observar la representación en forma de árbol del modelo obtenido para una de las estructuras de cardinal cuatro. En primer lugar, se cumple que el valor almacenado en cada nodo es siempre mayor que los alma-

Función	Entrada	Precondición	Descripción
InsertLLRB	$x : Int, t : LLRB$	$\{isLLRB(t)\}$	Inserta el elemento x en el árbol t
SearchLLRB	$x : Int, t : LLRB$	$\{isLLRB(t)\}$	Busca el elemento x en el árbol t
DeleteLLRBÏ	$x : Int, t : LLRB$	$\{isLLRB(t)\}$	Elimina el elemento x del árbol t

Tabla 5.8: Funciones para árboles rojinegros

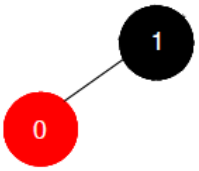


Figura 5.19: Modelo para un árbol de dos nodos

cenados en el subárbol izquierdo, y menor que aquellos almacenados en el derecho, cumpliendo así que sea BST. Por otro lado, respecto a los colores, respetan que las alturas negras son iguales para cualquier camino desde la raíz a un nodo vacío y, además, el único nodo rojo que nos aparece está a la izquierda. Por todo ello, el modelo proporcionado es totalmente correcto.

Por otro lado, un posible modelo para árboles de cardinal cinco es el reflejado en la Figura 5.21b. En él vemos que por cualquier camino que elijamos la altura negra es siempre dos, por lo que se respeta la primera de las condiciones para que esté correctamente coloreado. Además, los dos nodos rojos que nos encontramos no están consecutivos y, aunque hay uno rojo como hijo derecho de un nodo, el correspondiente hijo izquierdo es también rojo. Por todo ello, los colores proporcionados por Z3 son correctos. Respecto a los valores que encontramos, también respetan el orden requerido, por lo que, en conjunto, todo el modelo vuelve a ser correcto.

Por último, para cardinal seis comenzamos a obtener modelos más interesantes con los que mostrar la potencia de Z3, pues empieza a tener que alternar colores en los diferentes caminos. Uno de los modelos obtenidos es el de la Figura 5.21c. En él podemos ver como, a lo largo del hijo izquierdo, va alternando nodos de color negro y rojo, a fin de conseguir respetar todas las condiciones requeridas. Vemos que la primera de ellas, referida a las alturas negras de los hijos, se cumple, puesto que por todos los posibles caminos desde la raíz a un nodo vacío encontramos la misma altura negra. Por su parte, las referidas a los nodos rojos también se cumplen, puesto que no encontramos ni dos nodos rojos seguidos ni un nodo rojo como hijo derecho siendo el hijo izquierdo negro. Por tanto, todos los requisitos sobre el coloreado del árbol se satisfacen. Por su parte, en lo que refiere al orden de los

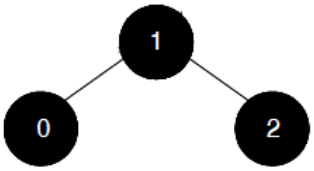


Figura 5.20: Modelo para un árbol de cardinal 3

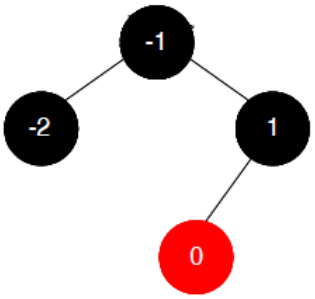
Cardinal	Casos Satisfactibles	Casos Insatisfactibles
Cuatro	2	12
Cinco	3	39
Seis	4	128

Tabla 5.9: Resultados obtenidos para árboles LLRB

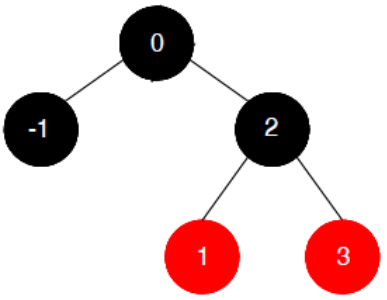
valores almacenados, también se cumple la condición impuesta para ser BST. Con todo ello, el modelo analizado es correcto.

Para concluir, en la Tabla 5.9 quedan recogidos los datos sobre el número de casos satisfactibles e insatisfactibles según el cardinal del árbol. Podemos ver que son muy pocas las estructuras que Z3 puede rellenar, puesto que la mayoría de los casos son insatisfactibles. Si bien de primeras puede resultar chocante que sean tantas las estructuras descartadas, todas las restricciones de color son, en conjunto, tan restrictivas, que esto puede ocurrir. Y es que, en primer lugar, en todas aquellas estructuras en las que no haya un cierto equilibrio entre los nodos que encontramos a la izquierda y la derecha de la raíz, resultará imposible conseguir que las alturas negras sean iguales. Descartadas todas estas, aún teniendo ese cierto equilibrio, hay muchas en las que el hijo izquierdo de un nodo es vacío mientras que el derecho no lo es, y a éste le correspondería el color rojo, situación prohibida también. En conclusión, resulta muy complicado verificar todas las restricciones involucradas en el predicado *isLLRB*, haciendo así que muchas de las estructuras generadas sean descartadas.

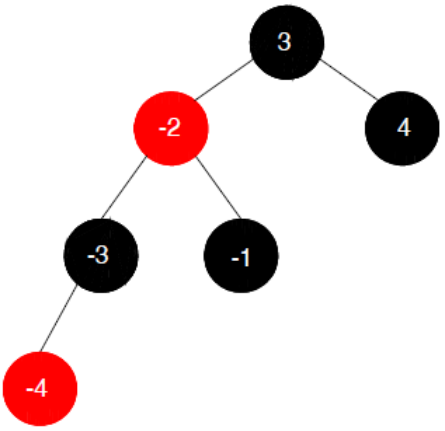
Viendo que son tantas las estructuras rechazadas, podría pensarse que nuestro sistema no está aportando ninguna mejoría. Si consideramos la generación de casos de prueba anterior, en la que se generaba aleatoriamente la estructura, la probabilidad de que fuera mala, y por tanto descartada, era igual que en nuestro caso, puesto que las comprobaciones a realizar son exactamente las mismas. Sin embargo, además de generarse aleatoriamente la estructura, se generaban también los valores almacenados en ella y los colores de cada nodo. Estos valores debían pasar el correspondiente filtro, haciendo que, aunque la estructura fuera correcta, hubiera que descartarla por incumplir tales valores alguna propiedad. Esto en cambio no ocurre en nuestro sistema, puesto que una vez tenemos las estructuras buenas, es imposible que los valores que las rellenen sean incorrectos.



(a) Cardinal 4



(b) Cardinal 5



(c) Cardinal 6

Figura 5.21: Modelos para cardinales cuatro, cinco y seis

Para concluir la sección, vamos a analizar, en términos de eficiencia, el comportamiento de nuestro sistema. Lo hacemos para árboles rojinegros ya que son los que tienen que hacer mayor número de comprobaciones y generar los modelos más complejos, siendo por tanto en los que peores resultados pueden obtenerse. En primer lugar, mencionar que al proporcionarnos Z3 un modelo, ésto no solo consiste en dar valor a las variables sino que todas las funciones declaradas forman parte del modelo. Por tanto, parte del tiempo lo ocupa en escribir todas estas funciones para cada modelo y, del mismo modo, la mayor parte de las líneas del fichero de salida se corresponden con todas ellas. Por ello, los resultados mostrados a continuación son, primero contando con la escritura de todas estas funciones para cada modelo, y después teniendo solo en cuenta la resolución de la satisfactibilidad de cada posible estructura. En la Tabla 5.10 se muestran los tiempos de ejecución para generar todos los modelos de diferentes cardinales, primero teniendo que escribir todo el modelo y después teniendo en cuenta solo la resolución de la satisfactibilidad de cada caso. Vemos que hasta cardinal seis los tiempos son bastante bajos, pero a partir de cardinal ocho se disparan. Esto parece lógico pues tenemos hasta 1430 estructuras

Cardinal	Tiempo modelo completo	Tiempo satisfactibilidad
Tres	$0.134s$	$0.128s$
Cuatro	$0.222s$	$0.163s$
Cinco	$0.481s$	$0.369s$
Seis	$1.149s$	$1.018s$
Ocho	$12.075s$	$11.645s$

Tabla 5.10: Tiempos de ejecución

diferentes. Otro aspecto que se observa es que difieren poco los tiempos con modelo de los de sin modelo. El número de estructuras rechazadas para estos árboles era muy elevado, por lo que son pocos los modelos que tiene que escribir en el primero de los casos. Por último, respecto al número de líneas del fichero final de salida, mostramos solo algún dato con el que hacernos a la idea de lo que ocupa cada cosa. Si tomamos por ejemplo estructuras de cardinal cinco, el fichero obtenido consta de 871 líneas. Sin embargo, si cogemos uno de los modelos proporcionados, resulta que solo 25 son de las variables que nos interesan, mientras que 240 son del resto de funciones. Por tanto, será principalmente esto último lo que haga que el número de líneas se eleve cuando tengamos más casos satisfactibles y, por tanto, más modelos.

Capítulo 6

Conclusiones

Castellano

Llegados hasta este punto, podemos afirmar que los objetivos que se marcaron al comienzo del trabajo han sido cumplidos.

Como primer gran objetivo, nos planteamos ser capaces de transformar un aserto-precondición en un conjunto de restricciones. Gracias al programa Haskell creado, expuesto en el Apéndice A, podemos transformar la precondición de una función, dada en su representación intermedia, en una serie de restricciones procesables por Z3 con las que generar automáticamente casos de prueba a partir de su satisfactibilidad

Por otro lado, se planteó como segundo reto la investigación de los resolutores SMT, concretamente de Z3, con la intención de codificar en esta plataforma todos los posibles predicados involucrados en las distintas precondiciones de algunas de las funciones más interesantes sobre las estructuras de datos tratadas. Durante el Capítulo 3, abordamos la explicación de la transformación de los diferentes asertos en funciones de Z3, mostrando además la potencia de esta herramienta para dar valor a las diferentes variables involucradas en las restricciones planteadas. Los resultados desprendidos de esta investigación fueron dispares. Y es que, si bien se ha descubierto que es muy potente a la hora de dar valor a variables de tipos sencillos como enteros o arrays, cuando tiene que sintetizar estructuras algebraicas como árboles o listas, es incapaz de hacerlo. Sin embargo, gracias a la estrategia planteada sobre separar las restricciones en diferentes tipos, hemos logrado salvar esta dificultad de Z3, consiguiendo que nuestro sistema sea bastante eficaz.

Con todo esto, hemos logrado cumplir la tarea de generar casos de prueba que satisfagan una cierta precondición, partiendo únicamente de la especificación de la misma.

Respecto a las líneas de trabajo a seguir en un futuro, la principal ampliación del proyecto reside en hacer uso de la API de Z3 para Haskell. Mediante su utilización, serán dos las principales ventajas que se puedan obtener. En primer lugar, la automatización de todo el proceso de generación de casos. Con nuestro tra-

bajo, generamos un fichero *smt* a partir del cuál, al ejecutarlo en Z3, obtenemos los diferentes casos de prueba buscados. Utilizando la API, podremos saltarnos el paso de crear dicho fichero obteniendo directamente los modelos que conforman los casos de prueba. En segundo lugar, la ventaja más importante es la posibilidad de generar un mayor número de casos de prueba. Hasta ahora, con el fichero *smt* podemos dar un único valor a cada una de nuestras estructuras. Gracias a la API de Z3, al poder tener acceso directo a los modelos proporcionados, podríamos lograr este objetivo. Para ello, habría que analizar tales modelos para crear nuevas restricciones en las que éstos sean negados, y volver a ejecutar después consiguiendo entonces un modelo diferente al anterior. Iterando este proceso mientras queden combinaciones de valores a elegir, se podría entonces conseguir un mayor número de casos de prueba para una misma estructura.

Inglés

At this stage, we can confirm that the stated goals at the beginning of this work have been accomplished.

As the first main goal, we considered being able to transform a precondition-assertion into a set of restrictions. Thanks to the Haskell program we have created, displayed in the Appendix A, we can transform any function precondition, given in its intermediate representation, to a series of constraints processable by Z3, by which to automatically generate test cases from its satisfiability.

On the other hand, the second challenge was the investigation of SMT solvers, Z3 in particular, with the intent of codifying in this platform all the predicates involved in the preconditions of some of the most interesting functions about the data structures we have considered. During Chapter 3, we explained the transformation of all the assertions into Z3 functions, showing, in addition, the power of this tool in giving values to the variables involved in the raised constraints. The results gathered from this investigation were disparate. On the one hand, while we have discovered Z3 is very powerful in giving values to simple type variables, such as integers or arrays, on the other hand it is unable to synthesize algebraic structures such as trees or lists. However, thanks to the proposed strategy of separating the constraints in different kinds, we have achieved to overcome this Z3 difficulty, so our system is quite effective.

Following this, we have managed to complete the task of generating test cases satisfying a certain precondition, only from its specification.

About the future work, the main extension of the project resides in using the Z3 API for Haskell. By using it, two advantages will be obtained. First, the automation of the entire case generation process. With our work, we generate a *smt* file from which, when executing it in Z3, we obtain the test cases we looked forward. Using the API, we may skip the step of creating such a file, directly obtaining the models which shape the test cases. Second, the most important advantage is the possibility of generating a larger number of test cases. Up to now, with the *smt* file we can give only one value to each of our structures. Thanks to the Z3 API, as we have direct access to the provided models, we could achieve this goal. In order to do that, those models should be analyzed in order to create new constraints which negate the model. Then, we will execute again the constraints obtaining a model different from the previous one. Iterating this process while it remains to exist combinations of the values, a larger number of test cases could be generated.

Apéndice A

Programa Haskell

A lo largo de este apéndice se muestra el código Haskell implementado encargado de traducir los archivos CLIR en ficheros *smt*. La función principal se encuentra en un módulo *Main*, que hace uso de una serie de funciones definidas en el módulo *Ast2Smt*. El módulo principal se encarga de comprobar que los tipos de las variables son correctos, lee los datos de las funciones y las declaraciones de tipos creados en Z3, y escribe el fichero de salida. El módulo secundario es el encargado de transformar el árbol abstracto de la función dada en una secuencia de cadenas en las que se declaren todas las variables necesarias y los asertos derivados de la precondición de dicha función. Además, este módulo es el encargado de generar las distintas estructuras para ciertos tipos de datos.

Dado que todo el trabajo se engloba dentro un proyecto más grande, el código que se muestra a continuación queda incluido dentro del proyecto Haskell *ir2Haskell*. Por ello, algunas de las llamadas a funciones que pueden encontrarse, o alguno de los módulos importados, no son propios de este trabajo sino que corresponden a trabajos previos sobre el proyecto global.

A.1. Main

```
module Main where
import Ir2Haskell (parseAST, haskellcode)
import Ast2Smt (filterVarsList, analyzeVars, allVars2String,
                 paramsCases2String, obtainCasesList, getAssertion, casesList2String)
import qualified Text.PrettyPrint.Mainland as D
import Language.Clir
import Data.List
import qualified Data.Text as T

-------------------------------------------------
--TIPOS CORRECTOS (ESTRUCTURAS ALMACENAN INT)
-------------------------------------------------
```

```
rightTypes:: [TopLevelDef] -> Either String TopLevelDef
rightTypes xs
        | (rightTypesAux (head . tail $ xs)) == True = Right (head . tail $ xs)
        | otherwise = Left "Tipos no soportados"

rightTypesAux:: TopLevelDef -> Bool
rightTypesAux (TopFunDef _ xs _ _ _) = rightTypesVarList xs

rightTypesVarList:: [TypedVar] -> Bool
rightTypesVarList [] = True
rightTypesVarList (x:xs) = rightTypesSingleVar x && rightTypesVarList xs

rightTypesSingleVar:: TypedVar -> Bool
rightTypesSingleVar (TypedVar _ t) = case t of
        UnitType -> True
        SimpleType _ -> True
        TypeVar _ -> True
        CompoundType ts -> rightTypesCTList ts

rightTypesCTList:: [ClirType] -> Bool
rightTypesCTList xs = rightSize && rightFirstType && rightSecondType
        where rightSize = (length xs == 2)
              rightFirstType = checkFirstType (head xs)
              rightSecondType = checkSecondType (last xs)

checkFirstType:: ClirType -> Bool
checkFirstType (SimpleType n)
        | t == "Arr" = True
        | t == "Lst" = True
        | t == "Tree" = True
        | t == "AVL" = True
        | t == "LLRB" = True
        | otherwise = False
        where t = T.unpack n
checkFirstType (CompoundType _) = False
checkFirstType (TypeVar _) = False
checkFirsType (UnitType) = False

checkSecondType:: ClirType -> Bool
checkSecondType (SimpleType n)
        | t == "Int" = True
        | otherwise = False
        where t = T.unpack n
checkSecondType (CompoundType _) = False
checkSecondType (TypeVar _) = False
checkSecondType (UnitType) = False
```

```
-------------------------------------------------
--LECTURA DE DATOS
-------------------------------------------------
readSizes:: [(String, String)] -> IO [Int]
readSizes [] = return ([]::[Int])
readSizes (x:xs) = case (snd x) of
        "Array" -> do
                   s <- readArraySize (fst x)
                   sl <- readSizes xs
                   return (s:sl::[Int])
        "Lst" -> do
                   s <- readListSize (fst x)
                   sl <- readSizes xs
                   return (s:sl::[Int])
        "Tree" -> do
                   s <- readTreeSize (fst x)
                   sl <- readSizes xs
                   return (s:sl::[Int])
        "AVL" -> do
                   s <- readAVLSize (fst x)
                   sl <- readSizes xs
                   return (s:sl::[Int])
        "LLRB" -> do
                   s <- readLLRBSize (fst x)
                   sl <- readSizes xs
                   return (s:sl::[Int])

readArraySize:: String -> IO Int
readArraySize n = do
        putStrLn ("Size of array " ++ n ++ "(>0): ")
        size <- getLine
        return (read size::Int)

readListSize:: String -> IO Int
readListSize n = do
        putStrLn ("Size of list " ++ n ++ "(>0): ")
        size <- getLine
        return (read size::Int)

readTreeSize:: String -> IO Int
readTreeSize n = do
        putStrLn ("Size of binary tree " ++ n ++ "(>0): ")
        size <- getLine
        return (read size::Int)
```

```
readAVLSize:: String -> IO Int
readAVLSize n = do
        putStrLn ("Size of AVL tree " ++ n ++ "(>0): ")
        size <- getLine
        return (read size::Int)

readLLRBSize:: String -> IO Int
readLLRBSize n = do
        putStrLn ("Size of LLRB tree " ++ n ++ "(>0): ")
        size <- getLine
        return (read size::Int)


-------------------------------------------------
--MAIN
-------------------------------------------------
outFileName:: TopLevelDef -> String
outFileName (TopFunDef x _ _ _ _) = x++".smt"

main :: IO ()
main = do parsing <- parseAST
          case parsing of
            Left  error -> putStrLn error
            Right defs  -> do
                let rTypes = rightTypes defs
                case rTypes of
                    Left error -> putStrLn error
                    Right def -> do
                       lSize <- readSizes lVars
                       cs1 <- readFile "tipos.smt"
                       cs2 <- readFile "funciones.smt"
                       writeFile name (cs1 ++ "\n\n" ++ cs2 ++ "\n\n" ++ vars ++
                          "\n\n" ++ (paramsCases2String lVars lSize) ++ "\n\n" ++
                          (casesList2String lAss (obtainCasesList def lSize) lVars))
                       where lVars = filterVarsList (analyzeVars def)
                             name = outFileName def
                             vars = allVars2String def
                             lAss = getAssertion def
```

A.2. Ast2Smt

```
module Ast2Smt where

import qualified Data.Text as T
```

```
import Language.Clir
import qualified Data.List as L

type PredName = String

-------------------------------------------------
--DECLARACIÓN DE LOS TIPOS ALGEBRAICOS
-------------------------------------------------
data Lst = Nil
          | Cons String Lst
          deriving Show

data Tree = Leaf
          | Node String Tree Tree
          deriving Show

data LLRB = LeafL
          | NodeL String String LLRB LLRB


data AVL = LeafA
          | NodeA String String AVL AVL


-------------------------------------------------
--PARSING DE LOS ASERTOS
-------------------------------------------------

getAssertion:: TopLevelDef -> [Assertion]
getAssertion (TopFunDef _ _ _ xs _) = getAssertionContr xs

getAssertionContr:: [Contract] -> [Assertion]
getAssertionContr [] = []
getAssertionContr ((SpecContract xs):ys) = getAssertionSpec xs ++ getAssertionContr ys

getAssertionSpec:: [Spec] -> [Assertion]
getAssertionSpec [] = []
getAssertionSpec (x:xs) = case x of
        PreCD ass -> [ass] ++ getAssertionSpec xs
        PostCD _ -> []
        OtherAssertion _ -> []

assList2String:: [Assertion] -> String
assList2String [] = "\n(check-sat)\n(get-model)\n"
assList2String (x:xs) = "(assert " ++ ass2String x ++ ")\n" ++ assList2String xs

ass2String:: Assertion -> String
ass2String ass = case ass of
        ATrue -> "true"
```

```
        AFalse -> "false"
        Not x -> notAss2String x
        And xs -> andAss2String xs
        Or xs -> orAss2String xs
        Imp xs -> impAss2String xs
        Equiv xs -> equivAss2String xs
        Aplic n xs -> aplicAss2String n xs
        Forall xs a -> forall2String xs a
        Exists xs a -> exists2String xs a

notAss2String:: Assertion -> String
notAss2String x = "(not " ++ ass2String x ++ ")"

andAss2String:: [Assertion] -> String
andAss2String [] = ")"
andAss2String [x] = ass2String x
andAss2String (x:xs) = "(and " ++ ass2String x ++ " " ++ andAss2String xs ++ ")"

orAss2String:: [Assertion] -> String
orAss2String [] = ")"
orAss2String [x] = ass2String x
orAss2String (x:xs) = "(or " ++ ass2String x ++ " " ++ orAss2String xs ++ ")"

impAss2String:: [Assertion] -> String
impAss2String [] = ")"
impAss2String [x] = ass2String x
impAss2String (x:xs) = "(-> " ++ ass2String x ++ " " ++ impAss2String xs ++ ")"

equivAss2String:: [Assertion] -> String
equivAss2String [] = ")"
equivAss2String [x] = ass2String x
equivAss2String (x:xs) = "(= " ++ ass2String x ++ " " ++ equivAss2String xs ++ ")"

forall2String:: [TypedVar] -> Assertion -> String
forall2String xs a = "(forall (" ++ varsFaEx2String xs ++ ") (=>" ++
                     ass2String a ++ "))"

exists2String:: [TypedVar] -> Assertion -> String
exists2String xs a = "(exists (" ++ varsFaEx2String xs ++ ") (=>" ++
                     ass2String a ++ "))"

varsFaEx2String:: [TypedVar] -> String
varsFaEx2String [] = ""
varsFaEx2String ((TypedVar n t):xs) = "(" ++ n ++ " " ++ type2String t ++
                                        ") " ++ varsFaEx2String xs

aplicAss2String:: PredName -> [BindingExpression] -> String
aplicAss2String n xs = "(" ++ n ++ beList2String xs
```

```
beList2String:: [BindingExpression] -> String
beList2String [] = ")"
beList2String (x:xs) = " " ++ be2String x ++ beList2String xs

be2String:: BindingExpression -> String
be2String be = case be of
      AtomE ae -> ae2String ae
      FunA n ae -> fun2String n ae
      ConstrA n ae -> constr2String n ae

aeList2String:: [AtomicExpression] -> String
aeList2String [] = ")"
aeList2String (x:xs) = " " ++ ae2String x ++ aeList2String xs

ae2String:: AtomicExpression -> String
ae2String ae = case ae of
      Var n -> n
      Const v _ -> constVal2String v

constVal2String:: ConstValue -> String
constVal2String (ConstInt x) = show x
constVal2String (ConstBool x)
        | x==True = "true"
        | otherwise = "false"

fun2String:: FunName -> [AtomicExpression] -> String
fun2String n xs = "(" ++ n ++ aeList2String xs

constr2String:: DataConstructor -> [AtomicExpression] -> String
constr2String n xs = "(" ++ n ++ " " ++ aeList2String xs


-------------------------------------------------
--ANÁLISIS DE LAS VARIABLES
-------------------------------------------------

analyzeVars:: TopLevelDef -> [(String, String)]
analyzeVars (TopFunDef _ xs _ _ _) = analyzeVarsList xs

analyzeVarsList:: [TypedVar] -> [(String, String)]
analyzeVarsList [] = []
analyzeVarsList (x:xs) = (analyzeSingleVar x):(analyzeVarsList xs)

analyzeSingleVar:: TypedVar -> (String, String)
analyzeSingleVar (TypedVar n x) = (n, analyzeType x)

analyzeTypeList:: [ClirType] -> String
analyzeTypeList [] = ""
analyzeTypeList (x:xs) = analyzeType x
```

```
analyzeType:: ClirType -> String
analyzeType x = case x of
        SimpleType n -> T.unpack n
        TypeVar n -> T.unpack n
        CompoundType xs -> analyzeTypeList xs

filterVarsList:: [(String, String)] -> [(String, String)]
filterVarsList xs = [(x,y) | (x,y)<-xs, (not (elem y ["Int", "Bool"]))]


------------------------------------------------
--GENERADOR DE CASOS
------------------------------------------------

generateCases:: [(String, String)] -> [Int] -> [[String]]
generateCases [x] [y] = case (snd x) of
        "Array" -> []
        "Tree" ->  [loadTrees (fst x) y]
        "Lst" ->  [loadList (fst x) y]
        "LLRB" -> [loadLLRBs (fst x) y]
        "AVL" -> [loadAVLs (fst x) y]
generateCases (x:xs) (y:ys) = case (snd x) of
        "Array" -> generateCases xs ys
        "Tree" -> (loadTrees (fst x) y):(generateCases xs ys)
        "Lst" -> (loadList (fst x) y):(generateCases xs ys)
        "LLRB" -> (loadLLRBs (fst x) y):(generateCases xs ys)
        "AVL" -> (loadAVLs (fst x) y):(generateCases xs ys)

combineLists:: [[a]] -> [[a]]
combineLists [] = [[]]
combineLists (x:xs) = [x:y | x<-x, y<-(combineLists xs)]

splits:: [a] -> [([a], [a])]
splits xs = L.zip (L.inits xs) (L.tails xs)

---------------GENERADOR DE LISTAS---------------

loadList:: String -> Int -> [String]
loadList x s = lists2String (generateList s x)

lists2String:: [Lst] -> [String]
lists2String [] = []
lists2String (x:xs) = oneList2String x:lists2String xs

oneList2String:: Lst -> String
oneList2String x = case x of
        Nil -> "nil"
        Cons c l -> "(cons " ++ c ++ " " ++ oneList2String l ++ ")"
```

```
generateListAux:: String -> [Char] -> Lst
generateListAux _ [] = Nil
generateListAux n (x:xs) = Cons (n ++ [x]) (generateListAux n xs)

generateList:: Int -> String -> [Lst]
generateList n x = [generateListAux x (L.take n ['1'..])]

--------------GENERADOR DE ÁRBOLES---------------

loadTrees:: String -> Int -> [String]
loadTrees x s = trees2String (generateTree s x)

trees2String:: [Tree] -> [String]
trees2String [] = []
trees2String (x:xs) = oneTree2String x:trees2String xs

oneTree2String:: Tree -> String
oneTree2String x = case x of
        Leaf -> "leaf"
        Node c l r -> "(node " ++ c ++ " " ++ oneTree2String l ++ " " ++
           oneTree2String r ++ ")"

generateTreeAux :: String -> [Char] -> [Tree]
generateTreeAux _ [] = return Leaf
generateTreeAux n (x:xs) = do
  (left, right) <- splits xs
  Node <$> pure (n++[x]) <*> generateTreeAux n left <*>
     generateTreeAux n right

generateTree:: Int -> String -> [Tree]
generateTree n x = generateTreeAux x (L.take n ['1'..])

----------------GENERADOR DE AVL-----------------

loadAVLs:: String -> Int -> [String]
loadAVLs x s = avls2String (generateAVL s x)

avls2String:: [AVL] -> [String]
avls2String [] = []
avls2String (x:xs) = oneAvl2String x:avls2String xs

oneAvl2String:: AVL -> String
oneAvl2String x = case x of
        LeafA -> "leafA"
        NodeA v h l r -> "(nodeA " ++ v ++ " " ++ h ++ " " ++
               oneAvl2String l ++ " " ++ oneAvl2String r ++ ")"
```

```
generateAVLAux :: String -> [Char] -> [AVL]
generateAVLAux _ [] = return LeafA
generateAVLAux n (x:xs) = do
  (left, right) <- splits xs
  NodeA <$> pure (n++[x]) <*> pure (n++"h"++[x]) <*> generateAVLAux n left <*>
      generateAVLAux n right

generateAVL:: Int -> String -> [AVL]
generateAVL n x = generateAVLAux x (L.take n ['1'..])

---------------GENERADOR DE LLRB-----------------

loadLLRBs:: String -> Int -> [String]
loadLLRBs x s = llrbs2String (generateLLRB s x)

llrbs2String:: [LLRB] -> [String]
llrbs2String [] = []
llrbs2String (x:xs) = oneLlrb2String x:llrbs2String xs

oneLlrb2String:: LLRB -> String
oneLlrb2String x = case x of
        LeafL -> "leafL"
        NodeL v c l r -> "(nodeL " ++ v ++ " " ++ c ++ " " ++
               oneLlrb2String l ++ " " ++ oneLlrb2String r ++ ")"

generateLLRBAux :: String -> [Char] -> [LLRB]
generateLLRBAux _ [] = return LeafL
generateLLRBAux n (x:xs) = do
  (left, right) <- splits xs
  NodeL <$> pure (n++[x]) <*> pure (n++"c"++[x]) <*> generateLLRBAux n left <*>
      generateLLRBAux n right

generateLLRB:: Int -> String -> [LLRB]
generateLLRB n x = generateLLRBAux x (L.take n ['1'..])


-------------------------------------------------
--PARSING DE LOS CASOS
-------------------------------------------------

obtainCasesList:: TopLevelDef -> [Int] -> [[String]]
obtainCasesList x sizes = combineLists
                       (generateCases (filterVarsList (analyzeVars x)) sizes)
```

```
casesList2String:: [Assertion] -> [[String]] -> [(String, String)] -> String
casesList2String a [x] y = "(push)\n" ++ cases2String a x y
casesList2String a (x:xs) y = "(push)\n" ++ cases2String a x y ++
                                casesList2String a xs y

cases2String:: [Assertion] -> [String] -> [(String, String)] -> String
cases2String a [] [] = "\n" ++ assList2String a ++ "(pop)\n\n"
cases2String a [] [(_, "Array")] = "\n" ++ assList2String a ++ "(pop)\n\n"
cases2String a (x:xs) (y:ys)
        | (snd y) == "Array" = "" ++ cases2String a (x:xs) ys
        | otherwise = "(assert (= " ++ fst y ++ " " ++ x ++ "))\n" ++
                        cases2String a xs ys


-------------------------------------------------
--PARSING DE LOS PARÁMETROS DE LOS CASOS
-------------------------------------------------

paramsCases2String:: [(String, String)] -> [Int] -> String
paramsCases2String [x] [y] = case (snd x) of
    "Array" -> sizeArr2String (fst x) y ++ paramsArr2String (fst x) (L.take y ['0'..])
    "Lst" -> paramsList2String (fst x) (L.take y ['1'..])
    "Tree" -> paramsTree2String (fst x) (L.take y ['1'..])
    "LLRB" -> paramsLLRB2String (fst x) (L.take y ['1'..])
    "AVL" -> paramsAVL2String (fst x) (L.take y ['1'..])
paramsCases2String (x:xs) (y:ys) = case (snd x) of
    "Array" -> sizeArr2String (fst x) y ++ paramsCases2String xs ys
    "Lst" -> paramsList2String (fst x) (L.take y ['1'..]) ++ paramsCases2String xs ys
    "Tree" -> paramsTree2String (fst x) (L.take y ['1'..]) ++ paramsCases2String xs ys
    "LLRB" -> paramsLLRB2String (fst x) (L.take y ['1'..]) ++ paramsCases2String xs ys
    "AVL" -> paramsAVL2String (fst x) (L.take y ['1'..]) ++ paramsCases2String xs ys

sizeArr2String:: String -> Int -> String
sizeArr2String n x = "(assert (= (second " ++ n ++ ") " ++ (show x) ++ "))\n\n"

paramsArr2String:: String -> [Char] -> String
paramsArr2String _ [] = "\n"
paramsArr2String n (x:xs) =
                  "(assert (> (select (first " ++ n ++ ") " ++ [x] ++ ") -5))\n" ++
                  "(assert (< (select (first " ++ n ++ ") " ++ [x] ++ ") 5))\n" ++
                   paramsArr2String n xs
```

```
paramsList2String:: String -> [Char] -> String
paramsList2String _ [] = "\n"
paramsList2String n (x:xs) = "(declare-const " ++ n ++ [x] ++ " Int)\n" ++
                               "(assert (> " ++ n ++ [x] ++ " -5))\n" ++
                               "(assert (< " ++ n ++ [x] ++ " 5))\n" ++
                                paramsList2String n xs

paramsTree2String:: String -> [Char] -> String
paramsTree2String _ [] = "\n"
paramsTree2String n (x:xs) = "(declare-const " ++ n ++ [x] ++ " Int)\n" ++
                               "(assert (> " ++ n ++ [x] ++ " -5))\n" ++
                               "(assert (< " ++ n ++ [x] ++ " 5))\n" ++
                                paramsTree2String n xs

paramsLLRB2String:: String -> [Char] -> String
paramsLLRB2String n xs = paramsLLRBVal2String n xs ++ paramsLLRBColor2String n xs

paramsLLRBVal2String:: String -> [Char] -> String
paramsLLRBVal2String n [x] = "(declare-const " ++ n ++ [x] ++ " Int)\n" ++
                             "(assert (> " ++ n ++ [x] ++ " -5))\n" ++
                             "(assert (< " ++ n ++ [x] ++ " 5))\n"
paramsLLRBVal2String n (x:xs) = "(declare-const " ++ n ++ [x] ++ " Int)\n" ++
                                  "(assert (> " ++ n ++ [x] ++ " -5))\n" ++
                                  "(assert (< " ++ n ++ [x] ++ " 5))\n" ++
                                   paramsLLRBVal2String n xs

paramsLLRBColor2String:: String -> [Char] -> String
paramsLLRBColor2String n [x] = "(declare-const " ++ n ++ "c" ++ [x] ++ " Color)\n"
paramsLLRBColor2String n (x:xs) = "(declare-const " ++ n ++ "c" ++ [x] ++
                                    " Color)\n" ++ paramsLLRBColor2String n xs

paramsAVL2String:: String -> [Char] -> String
paramsAVL2String n xs = paramsAVLVal2String n xs ++ paramsAVLHeight2String n xs

paramsAVLVal2String:: String -> [Char] -> String
paramsAVLVal2String n [x] = "(declare-const " ++ n ++ [x] ++ " Int)\n" ++
                            "(assert (> " ++ n ++ [x] ++ " -5))\n" ++
                            "(assert (< " ++ n ++ [x] ++ " 5))\n"
paramsAVLVal2String n (x:xs) = "(declare-const " ++ n ++ [x] ++ " Int)\n" ++
                                 "(assert (> " ++ n ++ [x] ++ " -5))\n" ++
                                 "(assert (< " ++ n ++ [x] ++ " 5))\n" ++
                                  paramsAVLVal2String n xs

paramsAVLHeight2String:: String -> [Char] -> String
paramsAVLHeight2String n [x] = "(declare-const " ++ n ++ "h" ++ [x] ++ " Int)\n"
paramsAVLHeight2String n (x:xs) = "(declare-const " ++ n ++ "h" ++ [x] ++
                                    " Int)\n" ++ paramsAVLHeight2String n xs
```

```
-------------------------------------------------
--PARSING DE LAS VARIABLES
-------------------------------------------------

allVars2String:: TopLevelDef -> String
allVars2String (TopFunDef _ xs _ _ _) = varList2String xs

varList2String:: [TypedVar] -> String
varList2String [] = "\n"
varList2String (x:xs) = var2String x ++ varList2String xs

var2String:: TypedVar -> String
var2String (TypedVar n x) = "(declare-const " ++ n ++ " " ++ type2String x ++ ")\n"

typesList2String:: [ClirType] -> String
typesList2String [] = ""
typesList2String (x:xs) = type2String x ++ " " ++ typesList2String xs

type2String:: ClirType -> String
type2String (SimpleType n)
        | t =="Array" = "Arr"
        | otherwise = t
        where t= T.unpack n
type2String (TypeVar n) = T.unpack n
type2String (CompoundType xs) = "(" ++ typesList2String xs ++ ")"
```

Apéndice B

Resultados de Ejecución

Cuando ejecutamos nuestro programa Haskell sobre un fichero CLIR, obtenemos como salida un fichero *smt* con todas las restricciones necesarias, que es el que pasamos a Z3 para obtener así los diferentes modelos. En este apéndice se muestra tal fichero, a fin de poder comprobar que la traducción que realiza nuestro programa es correcta. La cabecera de este fichero es muy extensa y únicamente consta de las declaraciones de los tipos y las funciones de que se explicaron en el Capítulo *3*. Por ello, aquí lo único que se muestra es la parte del fichero estrictamente perteneciente a la declaración de variables y la creación de restricciones a partir del fichero CLIR. El archivo mostrado corresponde a la traducción de la función *insertLLRB* para árboles de cardinal tres. Además, al ejecutar Z3 los modelos se vuelcan sobre un fichero *txt* desde el que poder analizar los resultados. Se muestra también en este apéndice dicho fichero para la misma función anterior.

B.1. Fichero Smt

```
(declare-const t (LLRB Int ))
(declare-const x Int)

(declare-const t1 Int)
(assert (> t1 -5))
(assert (< t1 5))
(declare-const t2 Int)
(assert (> t2 -5))
(assert (< t2 5))
(declare-const t3 Int)
(assert (> t3 -5))
(assert (< t3 5))
(declare-const tc1 Color)
(declare-const tc2 Color)
(declare-const tc3 Color)
```

```
(push)
(assert (= t (nodeL t1 tc1 leafL (nodeL t2 tc2 leafL (nodeL t3 tc3 leafL leafL)))))

(assert (isLLRB t))

(check-sat)
(get-model)
(pop)

(push)
(assert (= t (nodeL t1 tc1 leafL (nodeL t2 tc2 (nodeL t3 tc3 leafL leafL) leafL))))

(assert (isLLRB t))

(check-sat)
(get-model)
(pop)

(push)
(assert (= t (nodeL t1 tc1 (nodeL t2 tc2 leafL leafL) (nodeL t3 tc3 leafL leafL))))

(assert (isLLRB t))

(check-sat)
(get-model)
(pop)

(push)
(assert (= t (nodeL t1 tc1 (nodeL t2 tc2 leafL (nodeL t3 tc3 leafL leafL)) leafL)))

(assert (isLLRB t))

(check-sat)
(get-model)
(pop)

(push)
(assert (= t (nodeL t1 tc1 (nodeL t2 tc2 (nodeL t3 tc3 leafL leafL) leafL) leafL)))

(assert (isLLRB t))

(check-sat)
(get-model)
(pop)
```

B.2. Fichero Txt

```
unsat
(error "line 477 column 10: model is not available")
unsat
(error "line 486 column 10: model is not available")
sat
(model
  (define-fun tc1 () Color
    Negro)
  (define-fun t () (LLRB Int)
    (nodeL 1 Negro (nodeL 0 Negro leafL leafL) (nodeL 2 Negro leafL leafL)))
  (define-fun t3 () Int
    2)
  (define-fun t2 () Int
    0)
  (define-fun tc2 () Color
    Negro)
  (define-fun t1 () Int
    1)
  (define-fun tc3 () Color
    Negro)
  (define-fun minTL ((x!0 (LLRB Int))) Int
    (ite (= x!0 leafL) 2000
      (let ((a!1 (+ (minTL (izq x!0)) (* (- 1) (minTL (der x!0))))))
      (let ((a!2 (ite (>= a!1 0) (minTL (der x!0)) (minTL (izq x!0)))))
        (ite (>= (+ (val x!0) (* (- 1) a!2)) 0) a!2 (val x!0))))))
  (define-fun maxT ((x!0 (Tree Int))) Int
    (ite (= x!0 leaf) (- 2000)
      (let ((a!1 (+ (maxT (izq x!0)) (* (- 1) (maxT (der x!0))))))
      (let ((a!2 (ite (<= a!1 0) (maxT (der x!0)) (maxT (izq x!0)))))
        (ite (<= (+ (value x!0) (* (- 1) a!2)) 0) a!2 (value x!0))))))
  (define-fun multisetArr ((x!0 Int) (x!1 (Array Int Int)) (x!2 Int)) (Array Int
         Int)
    (ite (= x!0 x!2)
         ((as const (Array Int Int)) 0)
         ((_ map (+ (Int Int) Int))
           (multisetArr (+ 1 x!0) x!1 x!2)
           (store ((as const (Array Int Int)) 0) (select x!1 x!0) 1))))
  (define-fun cardA ((x!0 (AVL Int))) Int
    (ite (= x!0 leafA) 0
      (+ 1 (cardA (izq x!0)) (cardA (der x!0)))))
  (define-fun k!0 ((x!0 Int)) Int
    5)
  (define-fun k!29 ((x!0 (LLRB Int))) (LLRB Int)
    (ite (= x!0 (nodeL 0 Negro leafL leafL)) (nodeL 0 Negro leafL leafL)
      leafL))
  (define-fun minHeight ((x!0 (Tree Int))) Int
```

```
  (ite (= x!0 leaf) 0
    (let ((a!1 (+ (minHeight (izq x!0)) (* (- 1) (minHeight (der x!0))))))
      (+ 1 (ite (>= a!1 0) (minHeight (der x!0)) (minHeight (izq x!0)))))))
(define-fun sortedList ((x!0 (Lst Int))) Bool
  (let ((a!1 (or (= x!0 nil) (not (= (tl x!0) nil))))
        (a!2 (+ (hd x!0) (* (- 1) (hd (tl x!0))))))
  (let ((a!3 (or (>= a!2 0) (not (sortedList (tl x!0))))))
    (or (= x!0 nil) (not a!1) (not a!3)))))
(define-fun maxTL ((x!0 (LLRB Int))) Int
  (ite (= x!0 leafL) (- 2000)
    (let ((a!1 (+ (maxTL (izq x!0)) (* (- 1) (maxTL (der x!0))))))
    (let ((a!2 (ite (<= a!1 0) (maxTL (der x!0)) (maxTL (izq x!0)))))
      (ite (<= (+ (val x!0) (* (- 1) a!2)) 0) a!2 (val x!0))))))
(define-fun k!28 ((x!0 (LLRB Int))) (LLRB Int)
  (ite (= x!0 (nodeL 2 Negro leafL leafL)) (nodeL 2 Negro leafL leafL)
    leafL))
(define-fun k!27 ((x!0 (LLRB Int))) (LLRB Int)
  (ite (= x!0 (nodeL 0 Negro leafL leafL)) (nodeL 0 Negro leafL leafL)
  (ite (= x!0 (nodeL 2 Negro leafL leafL)) (nodeL 2 Negro leafL leafL)
    leafL)))
(define-fun altA ((x!0 (AVL Int))) Int
  (ite (= x!0 leafA) 0
    (let ((a!1 (+ (altA (izq x!0)) (* (- 1) (altA (der x!0))))))
      (+ 1 (ite (<= a!1 0) (altA (der x!0)) (altA (izq x!0)))))))
(define-fun setA ((x!0 (AVL Int))) (Array Int Bool)
  (ite (= x!0 leafA) ((as const (Array Int Bool)) false)
    ((_ map (or (Bool Bool) Bool))
      (setA (izq x!0))
      (setA (der x!0))
      (store ((as const (Array Int Bool)) false) (val x!0) true))))
(define-fun minT ((x!0 (Tree Int))) Int
  (ite (= x!0 leaf) 2000
    (let ((a!1 (+ (minT (izq x!0)) (* (- 1) (minT (der x!0))))))
    (let ((a!2 (ite (>= a!1 0) (minT (der x!0)) (minT (izq x!0)))))
      (ite (>= (+ (value x!0) (* (- 1) a!2)) 0) a!2 (value x!0))))))
(define-fun setL ((x!0 (LLRB Int))) (Array Int Bool)
  (ite (= x!0 leafL) ((as const (Array Int Bool)) false)
    ((_ map (or (Bool Bool) Bool))
      (setL (izq x!0))
      (setL (der x!0))
      (store ((as const (Array Int Bool)) false) (val x!0) true))))
(define-fun alt ((x!0 (Tree Int))) Int
  (ite (= x!0 leaf) 0
    (let ((a!1 (+ (alt (izq x!0)) (* (- 1) (alt (der x!0))))))
      (+ 1 (ite (<= a!1 0) (alt (der x!0)) (alt (izq x!0)))))))
(define-fun maxTA ((x!0 (AVL Int))) Int
  (ite (= x!0 leafA) (- 2000)
    (let ((a!1 (+ (maxTA (izq x!0)) (* (- 1) (maxTA (der x!0))))))
    (let ((a!2 (ite (<= a!1 0) (maxTA (der x!0)) (maxTA (izq x!0))))
```

```
      (ite (<= (+ (val x!0) (* (- 1) a!2)) 0) a!2 (val x!0))))))
(define-fun isHeap ((x!0 (Tree Int))) Bool
  (let ((a!1 (or (= x!0 leaf)
                 (not (= (izq x!0) leaf))
                 (not (= (der x!0) leaf))))
        (a!2 (or (= x!0 leaf) (not (= (der x!0) leaf))))
        (a!3 (+ (value x!0) (* (- 1) (value (izq x!0)))))
        (a!4 (+ (value x!0) (* (- 1) (value (der x!0))))))
  (let ((a!5 (or (>= a!3 0)
                 (>= a!4 0)
                 (not (isHeap (izq x!0)))
                 (not (isHeap (der x!0))))))
    (or (= x!0 leaf) (not a!1) (ite a!2 (not a!5) (not (>= a!3 0)))))))
(define-fun minTL!32 ((x!0 (LLRB Int))) Int
  (ite (= x!0 leafL) 2000
    2))
(define-fun heightL ((x!0 (LLRB Int))) Int
  (ite (= x!0 leafL) 0
    (let ((a!1 (+ (heightL (izq x!0)) (* (- 1) (heightL (der x!0))))))
      (+ 1 (ite (<= a!1 0) (heightL (der x!0)) (heightL (izq x!0))))))))
(define-fun k!34 ((x!0 Int)) Bool
  false)
(define-fun maxTL!33 ((x!0 (LLRB Int))) Int
  (ite (= x!0 leafL) (- 2000)
    0))
(define-fun long ((x!0 (Lst Int))) Int
  (ite (= x!0 nil) 0
    (+ 1 (long (tl x!0)))))
(define-fun member ((x!0 Int) (x!1 (Lst Int))) Bool
  (let ((a!1 (not (or (= x!0 (hd x!1)) (member x!0 (tl x!1))))))
    (not (or (= x!1 nil) a!1))))
(define-fun isBSTL ((x!0 (LLRB Int))) Bool
  (let ((a!1 (or (not (= (izq x!0) leafL)) (not (= (der x!0) leafL))))
        (a!2 (+ (val x!0) (* (- 1) (minTL (der x!0)))))
        (a!4 (+ (val x!0) (* (- 1) (maxTL (izq x!0))))))
  (let ((a!3 (or (not (isBSTL (der x!0))) (>= a!2 0)))
        (a!5 (or (not (isBSTL (izq x!0))) (<= a!4 0)))
        (a!6 (or (not (isBSTL (izq x!0)))
                 (not (isBSTL (der x!0)))
                 (<= a!4 0)
                 (>= a!2 0))))
  (let ((a!7 (ite (= (izq x!0) leafL)
                  (not a!3)
                  (ite (= (der x!0) leafL) (not a!5) (not a!6)))))
    (or (= x!0 leafL) (not a!1) a!7)))))
(define-fun minTA ((x!0 (AVL Int))) Int
  (ite (= x!0 leafA) 2000
    (let ((a!1 (+ (minTA (izq x!0)) (* (- 1) (minTA (der x!0))))))
    (let ((a!2 (ite (>= a!1 0) (minTA (der x!0)) (minTA (izq x!0)))))
```

```
      (ite (>= (+ (val x!0) (* (- 1) a!2)) 0) a!2 (val x!0))))))
(define-fun isBSTA ((x!0 (AVL Int))) Bool
  (let ((a!1 (or (not (= (izq x!0) leafA)) (not (= (der x!0) leafA))))
        (a!2 (+ (val x!0) (* (- 1) (minTA (der x!0)))))
        (a!4 (+ (val x!0) (* (- 1) (maxTA (izq x!0))))))
  (let ((a!3 (or (not (isBSTA (der x!0))) (>= a!2 0)))
        (a!5 (or (not (isBSTA (izq x!0))) (<= a!4 0)))
        (a!6 (or (not (isBSTA (izq x!0)))
                 (not (isBSTA (der x!0)))
                 (<= a!4 0)
                 (>= a!2 0))))
  (let ((a!7 (ite (= (izq x!0) leafA)
                  (not a!3)
                  (ite (= (der x!0) leafA) (not a!5) (not a!6)))))
    (or (= x!0 leafA) (not a!1) a!7)))))
(define-fun set ((x!0 (Tree Int))) (Array Int Bool)
  (ite (= x!0 leaf) ((as const (Array Int Bool)) false)
    ((_ map (or (Bool Bool) Bool))
      (set (izq x!0))
      (set (der x!0))
      (store ((as const (Array Int Bool)) false) (value x!0) true))))
(define-fun isLeftist ((x!0 (Tree Int))) Bool
  (let ((a!1 (+ (minHeight (izq x!0)) (* (- 1) (minHeight (der x!0))))))
  (let ((a!2 (or (not (isHeap (izq x!0)))
                 (not (isHeap (der x!0)))
                 (not (>= a!1 0)))))
    (or (= x!0 leaf) (not a!2)))))
(define-fun isAVL ((x!0 (AVL Int))) Bool
  (let ((a!1 (+ (altA (izq x!0)) (* (- 1) (altA (der x!0)))))
        (a!2 (+ (* (- 1) (altA (izq x!0))) (altA (der x!0)))))
  (let ((a!3 (not (<= (ite (<= a!1 0) a!2 a!1) 1)))
        (a!4 (* (- 1) (ite (>= a!1 0) (altA (izq x!0)) (altA (der x!0))))))
  (let ((a!5 (not (= (+ (alt x!0) a!4) 1))))
  (let ((a!6 (or (not (isBSTA x!0))
                 (not (isAVL (izq x!0)))
                 (not (isAVL (der x!0)))
                 a!3
                 a!5)))
    (or (= x!0 leafA) (not a!6)))))))
(define-fun isBST ((x!0 (Tree Int))) Bool
  (let ((a!1 (or (not (= (izq x!0) leaf)) (not (= (der x!0) leaf))))
        (a!2 (+ (value x!0) (* (- 1) (minT (der x!0)))))
        (a!4 (+ (value x!0) (* (- 1) (maxT (izq x!0))))))
  (let ((a!3 (or (not (isBST (der x!0))) (>= a!2 0)))
        (a!5 (or (not (isBST (izq x!0))) (<= a!4 0)))
        (a!6 (or (not (isBST (izq x!0)))
                 (not (isBST (der x!0)))
                 (<= a!4 0)
                 (>= a!2 0))))
```

```
    (let ((a!7 (ite (= (izq x!0) leaf)
                    (not a!3)
                    (ite (= (der x!0) leaf) (not a!5) (not a!6)))))
      (or (= x!0 leaf) (not a!1) a!7)))))
  (define-fun blackHeight ((x!0 (LLRB Int))) Int
    (ite (= x!0 leafL) 0
      (let ((a!1 (+ (blackHeight (izq x!0)) (* (- 1) (blackHeight (der x!0))))))
      (let ((a!2 (ite (<= a!1 0)
                      (blackHeight (der x!0))
                      (blackHeight (izq x!0)))))
        (ite (= (ite (= x!0 leafL) Negro (color x!0)) Negro) (+ 1 a!2) a!2)))))
  (define-fun isLlrbAux ((x!0 (LLRB Int))) Bool
    (let ((a!1 (= (ite (= (der x!0) leafL) Negro (color (der x!0))) Rojo))
          (a!2 (= (ite (= (izq x!0) leafL) Negro (color (izq x!0))) Rojo))
          (a!3 (= (ite (= (izq x!0) leafL) Negro (color (izq x!0))) Negro))
          (a!4 (= (ite (= (der x!0) leafL) Negro (color (der x!0))) Negro))
          (a!6 (not (= (blackHeight (izq x!0)) (blackHeight (der x!0))))))
    (let ((a!5 (ite (= (color x!0) Negro)
                    (or (not a!1) a!2)
                    (not (or (not a!3) (not a!4))))))
    (let ((a!7 (or (not (isBSTL x!0))
                   (not (isLlrbAux (izq x!0)))
                   (not (isLlrbAux (der x!0)))
                   (not (or (= x!0 leafL) a!5))
                   a!6)))
      (or (= x!0 leafL) (not a!7))))))
  (define-fun blackHeight!35 ((x!0 (LLRB Int))) Int
    (ite (= x!0 leafL) 0
      1))
  (define-fun multiset ((x!0 (Lst Int))) (Array Int Int)
    (ite (= x!0 nil) ((as const (Array Int Int)) 0)
      ((_ map (+ (Int Int) Int))
        (multiset (tl x!0))
        (store ((as const (Array Int Int)) 0) (hd x!0) 1))))
  (define-fun card ((x!0 (Tree Int))) Int
    (ite (= x!0 leaf) 0
      (+ 1 (card (izq x!0)) (card (der x!0)))))
  (define-fun cardL ((x!0 (LLRB Int))) Int
    (ite (= x!0 leafL) 0
      (+ 1 (cardL (izq x!0)) (cardL (der x!0)))))
)
unsat
(error "line 504 column 10: model is not available")
unsat
(error "line 513 column 10: model is not available")
```

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

[3] Lars Arge, Robert Sedgewick, and Raimund Seidel. 08081 abstracts collection - data structures. In Lars Arge, Robert Sedgewick, and Raimund Seidel, editors, *Data Structures, 17.02. - 22.02.2008*, volume 08081 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2008.

[4] 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.

[5] 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.

[6] 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 Verification 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.

[7] Manuel Montenegro, Ricardo Peña, and Jaime Sánchez-Hernández. A generic intermediate representation for verification 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.