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

GENERACIÓN DE CASOS DE PRUEBA DE CAJA BLANCA MEDIANTE RESTRICCIONES

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

Septiembre 2018

Autor:
Javier Sagredo Tamayo
Director:
Ricardo Peña Marí

Resumen

La plataforma CAVI-ART para la verificación automática de programas ofrece numerosas herramientas para el análisis de los mismos entre las que encontramos un generador de casos de prueba. El método seguido por el mismo se asemeja bastante a la fuerza bruta lo que desperdicia recursos y tiempo.

El objetivo de este trabajo no es otro que mejorar sustancialmente el generador de casos de prueba. Se persigue conseguir un análisis de caja blanca guiado por el flujo de control del programa que permita generar casos certeros que comprueben hasta cierto punto todos los posibles caminos dentro del programa. Para ello analizaremos los programas según su naturaleza recursiva, desarrollaremos estrategias de análisis de los grafos de flujo de control para obtener caminos dentro de los mismos y traduciremos estos caminos a una serie de restricciones en el lenguaje SMT-LIB para finalmente obtener gracias al resolutor Z3 un modelo para las variables de entrada.

Palabras clave

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

Abstract

The CAVI-ART platform for assisted program validation offers a set of tools for analyzing code among which we find a test-case generator. The actual method followed by this tool is quite similar to bruteforce and so, it wastes resources and time.

The aim of this research is to substantially improve the test case generator. We want to achieve a whitebox analysis guided by the program's control flow which will be used to generate accurate test cases that potentially check all the paths in a program. With this aim in mind, we will first analyze the recursive nature of the program, we will develop strategies for analyzing the control flow graphs in order to find paths through them and we will translate such paths into a sequence of constraints written in the SMT-LIB language in order to finally obtain, by means of the SMT solver Z3, a model for the input variables.

Keywords

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

Índice general

Capítulo 1

Introducción

Castellano

Tras la codificación de un programa, es bien sabido que se deberían realizar pruebas de ejecución. De esta manera, se podría comprobar la corrección del programa y el buen funcionamiento del mismo. Sin embargo, la codificación de las pruebas de ejecución es algo tedioso pues requiere idear casos extremos, comprobar la cobertura del código, realizar numerosas ejecuciones, conocer las respuestas correctas y comprobar las salidas. Es por esto que la automatización del proceso de generación de casos de prueba sería una herramienta muy interesante.

Podemos clasificar las estrategias de pruebas en dos grandes grupos: de caja negra y de caja blanca [1]. El primero de ambos grupos se centra en la generación de casos que consideren la especificación dada por la precondición y la postcondición, obviando el funcionamiento interno del programa. La generación de los casos de caja negra es un tema que se desarrolla ampliamente en [5] Sin embargo, la generación de los casos de prueba considerando solo la precondición no fuerza a que se recorran todos los caminos.

Situado dentro dentro del proyecto CAVI-ART, explicado más adelante, este trabajo pretende trabajar la estrategia de generación de casos de prueba para caja blanca. En trabajos previos sobre esta plataforma [2] [5], se ha desarrollado una herramienta que transforma un aserto como la precondición o la postcondición en una función ejecutable, de forma que dicha función comprueba la validez del aserto en función de los valores de las variables que intervengan en dicho aserto.

Actualmente el sistema seguido para la generación de casos de prueba es similar a la fuerza bruta: de menor a mayor tamaño se generan exhaustivamente todos los posibles casos y se filtran mediante la precondición ejecutable para conservar solo los casos que la cumplen. Tras ello se ejecuta el código del programa para obtener una salida que se comprueba contra la postcondición ejecutable. De este modo se realiza una búsqueda ciega por el espacio de entradas que no hace sino desperdiciar recursos pues numerosos casos han de desecharse.

El objetivo de este trabajo es generar casos de prueba de caja blanca de

manera automática. Generaremos entonces casos que vayan cubriendo todos los posibles caminos hasta un nivel de profundidad dado. Consideramos como *profundidad* el número de iteraciones de los bucles o el número de despliegues de llamadas recursivas. Combinaremos nuestra estrategia con la de [5] para generar casos correctos específicos para cada camino. Utilizaremos, como en el trabajo en el que nos basamos, estructuras de datos con invariantes sofisticados como por ejemplo árboles AVL, o árboles rojinegros.

Realizaremos un estudio de la naturaleza recursiva del programa, buscaremos una estructura similar a un grafo que nos permita representar el flujo del programa y en ella definiremos caminos. Además trabajaremos el concepto de la profundidad de un camino y, de manera específica, la profundidad en un programa recursivo no final.

Así, podemos resumir el objetivo del trabajo en un análisis del programa para generar caminos en el mismo, una traducción de caminos en asertos de Z3 y la combinación de estos con los casos de caja negra para generar casos completos. Para el análisis de los programas y la generación de las restricciones Z3 se ha desarrollado un programa Haskell que utiliza las herramientas IR2Haskell y Prec2Z3 para poder funcionar. Mediante el programa Haskell se genera un archivo SMT-LIB que es procesado manualmente por Z3. En un futuro, el objetivo a largo plazo es que el proceso se realice mediante la API Z3-Haskell[1] lo que permitirá interactuar con los resultados devueltos por Z3.

Podemos distinguir los siguientes bloques en el trabajo. En primer lugar se introducen las herramientas que utilizaremos a lo largo del análisis. Tras ello en el Capítulo 3 se estudia la transformación de los programas en grafos con la exposición de la dificultad de tratamiento de los programas recursivos no finales y la definición del concepto de *grafo modular*. Seguidamente en el Capítulo 4 se estudia la generación de caminos y el funcionamiento del renombramiento de variables. En el Capítulo 5 se expone la transformación entre expresiones del lenguaje y las restricciones de Z3 y la generación de las restricciones del camino. Finalmente, dedicamos el Capítulo 6 a la exposición de experimentos y sus resultados, y exponemos las conclusiones en el Capítulo 7.

[1] http://hackage.haskell.org/package/z3

Inglés

Once finished the coding of a program, it is well known that execution tests should be run. With them, the correctness and well behavior of the program could be checked. It is the case that coding execution tests is a boring and difficult task as the programmer has to think of extreme cases, to check the code coverage, to execute many cases, to know the correct outputs and to check the output given by the program. That's why making this process automatic would be so interesting.

We can classify the testing strategies in two big classes: black-box and white-box [1]. The first of these classes looks forward to generating test cases that satisfy the specification given by the precondition and the postcondition, without looking inside the program at all. The generation of black-box test cases is a topic widely discussed in [5]. However, considering only the precondition doesn't assure that all paths in the program are traversed.

Inside the CAVI-ART project, which we explain later in the text, this research aims to develop the generation of white-box test cases. In previous papers about this platform [2] [5], a tool has been developed that transforms an assertion, such as the precondition or the postcondition, into an executable function so that such function checks the validity of the assertion depending on the values of the variables that appear in such assertion.

At this moment, the method followed for the test case generation is similar to bruteforce: from the smallest cases to the biggest ones, all of them are generated exhaustively and they are filtered by the executable precondition in order to keep only the ones that make it true. After that, the program code is executed in order to get an output that is checked by the executable postcondition. In this way, a blind search is done by the space of inputs that wastes a lot of resources because many generated cases must be thrown away.

The aim of this research is generating white-box test cases in an automated way. We will generate cases that cover all the possible paths within a given recursion level. We will combine our strategy with the one in [5] in order to generate correct specific cases for each path. As in the above cited work, we will deal with data structures with complex invariants such as AVL trees and red-black trees.

We will analyze the recursive nature of the program, find something similar to a graph that can represent the program flow and we will define paths through it. Also, we will generate the concept of path depth and, specifically, the program depth of a non-tail recursive program.

We could summarize the paper aim as a program analysis for generating paths through it, a translation of paths into Z3 assertions, and the combination of this assertions with the black-box generated ones in order to generate complete cases. For performing the program analysis and for generating the Z3 restrictions, a Haskell program has been coded that uses the tools IR2Haskell and Prec2Z3. With this program, a SMT-LIB file is generated and manually fed into Z3. In the future,

this process shall be done by means of the Z3-Haskell API[2] what will make Z3 output accessible and able to be analyzed.

We can distinguish the following parts in the paper. In the first place, we introduce the tools we will use through the analysis. We will analyze in Chapter 3 how to transform a program into a graph discussing the difficulties we found when working with non-tail recursive programs and the definition of *modular graphs*. After that, in Chapter 4 we will analyze how to generate a path and how to rename the variables in order to have fresh ones when doing recursion. In Chapter 5 we show the translation of program expressions into Z3 restrictions, and how to generate the restrictions of a full path. Finally, in Chapter 6 we try to experiment with all this concepts and in Chapter 7 we extract some conclusions.

[2] http://hackage.haskell.org/package/z3

Capítulo 2

Preliminares

2.1. Proyecto CAVI-ART

La plataforma CAVI-ART (*Computer Assisted Validation by Analysis, Transformation and Proofs*) [8] [9] (Figura 2.1) es un proyecto orientado a la validación automática de programas. La clave de este proyecto es lo que se denomina *Intermediate Representation* (en adelante IR) siendo ésta una sintaxis abstracta.

Dado un programa escrito en uno de los lenguajes soportados actualmente por la plataforma (en este momento Java, C, C++ y Haskell), este se transforma en una versión equivalente del programa escrita en la IR. Podemos destacar que la IR se basa en los lenguajes funcionales y por ello su estructura es similar a la de éstos. En la Figura 2.2 puede verse su sintaxis abstracta. Además contamos con una versión textual de esta sintaxis abstracta que se denomina CLIR (Common Lisp IR) y se asemeja en gran medida a la estructura de paréntesis típica de Lisp y sus dialectos.

Tras la primera fase de transformación del programa en su versión en la IR, se simplifican los análisis puesto que se enmascara el lenguaje original lo que nos permite abstraernos de cuál fuera. La plataforma CAVI-ART cuenta con una serie de herramientas que realizan distintos análisis estáticos utilizando técnicas de verificación formal. Son numerosas las posibilidades de análisis de los programas y actualmente están implementadas las pruebas de terminación, inferencia de invariantes, pruebas de condiciones de verificación y generación de casos de prueba automáticos.

Es en este apartado en el que centraremos el objetivo del trabajo. La plataforma actual sigue un método de descarte por el cual se generan una gran cantidad de casos de prueba aleatorios. En primer lugar se filtran dichos casos por la precondición para descartar los que no sean válidos y luego se ejecuta el programa comprobando que la postcondición se satisface con el resultado obtenido. La generación desperdicia numerosos recursos puesto que es una generación ciega que considera solo el tipo de los parámetros de entrada.

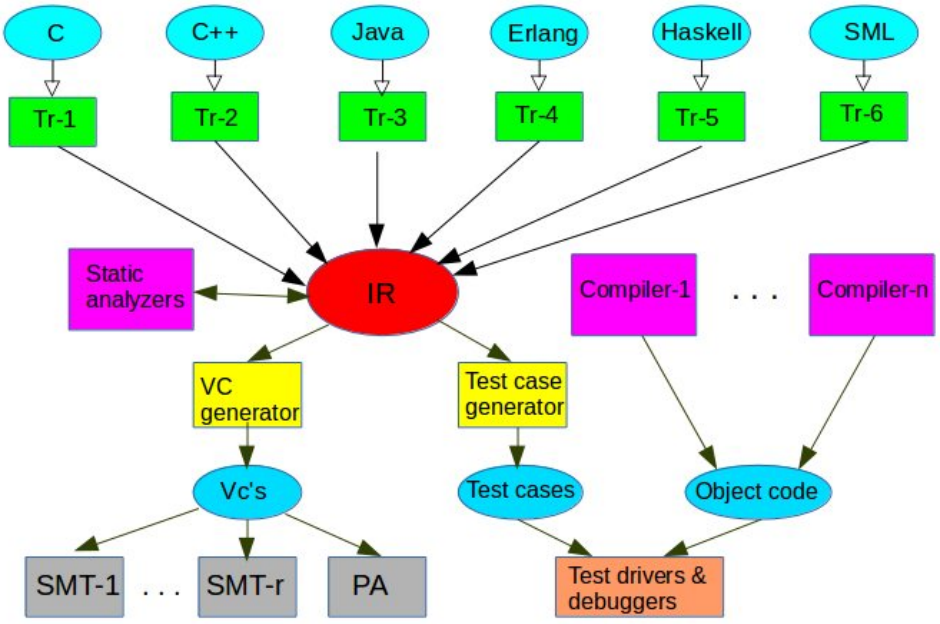

Figura 2.1: Plataforma CAVI-ART

Buscaremos por tanto la manera de optimizar esta generación de casos de prueba atendiendo a la cobertura del código y a los posibles caminos que pueden darse dentro del programa.

2.2. El resolutor SMT Z3

Los resolutores SMT [3] (*Satisfiability Modulo Theories*) son herramientas que resuelven problemas de satisfactibilidad módulo un cierto conjunto de teorías. Así, dado un problema de satisfactibilidad o validez, es decir una fórmula lógica, este es interpretado a la luz de una de las teorías existentes, como pueden ser la teoría de los números enteros o la teoría de los arrays. Así se le da un significado a los símbolos y expresiones que aparecen en la fórmula y se pueden realizar deducciones e inferencias con ellos. Como ejemplo, dada la fórmula lógica $a > b \wedge b > c \wedge c > 0$, Z3 puede inferir nuevas restricciones como por ejemplo $a > 0$ y $b > 0$. Así puede resolver problemas de satisfactibilidad (encontrar ciertos valores que hacen cierta la fórmula) y de validez (demostrar que la fórmula es siempre cierta para cualquier valor de las variables).

Dentro de los resolutores SMT nos encontramos con Z3 [4], un resolutor SMT desarrollado por *Microsoft* y que se encuentra entre los más avanzados. He-

a	$::=$	c	{ constante }
	$\mid$	x	{ variable }
be	$::=$	a	{ expresión atómica }
	$\mid$	$f\ \overline{a_i}$	{ aplicación de función/operador primitivo }
	$\mid$	$\langle \overline{a_i} \rangle$	{ construcción de tuplas }
	$\mid$	$C\ \overline{a_i}$	{ aplicación de constructor }
e	$::=$	be	{ expresión ligada }
	$\mid$	**let** $\langle \overline{x_i :: \tau_i} \rangle = be$ **in** e	{ let secuencial }
	$\mid$	**letfun** $\overline{def_i}$ **in** e	{ let recursivo para definición de funciones }
	$\mid$	**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 }
	$\mid$	$T\ \overline{\tau_i}$	{ aplicación de un constructor de tipo }

Figura 2.2: Sintaxis abstracta de la IR

rramientas avanzadas como el resolutor de restricciones MiniZinc contienen en su interior un resolutor Z3 que le da la capacidad de cómputo necesaria para funcionar. El otro resolutor habitualmente mencionado es CVC4, desarrollado por la *Universidad de Standford*.

El lenguaje aceptado por Z3 es una implementación del estándar SMT-LIB en su última versión 2.6[1]. Dentro del mismo vamos a destacar varios comandos que nos van a ser de gran utilidad a lo largo de este trabajo:

- *check-sat*: comprueba la validez de las restricciones introducidas hasta el momento y devuelve *sat*, *unsat* o *unknown* en caso de que sean satisfactibles, que no lo sean o que no pueda decidirse si lo son o no como en el caso de algunas fórmulas que utilizan cuantificadores.
- *get-model*: una vez que el conjunto de restricciones se determine como *sat*, este comando solicita un modelo, es decir una asignación de valores a las variables, que hacen cierta la fórmula. Cabe destacar que para obtener modelos adicionales la solución más utilizada es la negación reiterada del último modelo obtenido aunque esto puede resultar en unos conjuntos impracticables de restricciones.
- *declare-const*: permite la definición de una variable indicando su tipo.
- *declare-fun*: permite la definición de una función con parámetros.
- *declare-fun-rec*: permite la definición de una función recursiva con parámetros.
- *assert*: introduce en la pila una nueva restricción.

[1] http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2017-07-18.pdf

- *push*: aumenta el contador de la pila y genera un punto de control en la misma.
- *pop*: si el contador de la pila es mayor que cero lo decrementa y desecha las restricciones posteriores al último *push*.

Así, Z3 funciona con una serie de restricciones sobre funciones y variables (siendo estas últimas funciones de aridad 0) que trata de discernir si son satisfactibles obteniendo unos ciertos valores que satisfagan las restricciones o, si es imposible, indicándolo mediante un mensaje de *unsat*.

Cabe destacar que dada una fórmula lógica, si su negación es insatisfactible queda demostrado que la fórmula original es una tautología, es decir, es verdadera para todos los posibles valores que se den.

Englobando esta herramienta en el proceso de investigación que vamos a llevar a cabo, debemos matizar que queremos obtener los modelos de Z3 porque precisamente estos modelos ofrecen unos valores para los parámetros de entrada de la función a probar que son exactamente los casos de prueba de caja blanca que pretendemos generar.

Teorías

Z3 permite trabajar con distintas teorías de las cuales destacaremos aquellas que vamos a utilizar a lo largo de este trabajo:

- **Aritmética lineal de enteros y reales:** esta teoría nos permite representar fórmulas lógicas que involucren números enteros y reales y las operaciones habituales de estos incluyendo módulo, resto y división.
- **Lógica proposicional:** gracias al tipo predefinido Bool podemos representar fórmulas lógicas que involucren valores lógicos. Así, podemos razonar con las operaciones lógicas habituales y también con estructuras más complejas como las construcciones si-entonces-sino o la construcción match.
- **Arrays:** los arrays en Z3 son soportados mediante su interpretación como funciones de manera que el modelo de un array nos ofrece una función que hace corresponder el dominio de los índices en el dominio del contenido. Así un array habitual de enteros con índices enteros se representará como una función que recibe un entero y devuelve un entero. La síntesis que realiza Z3 de la función se concreta en estructuras si-entonces-sino anidadas de manera que existe un caso por defecto lo que hace que en la práctica los arrays sean funciones totales lo que en el caso de índices enteros significa que son estructuras infinitas. En el caso de estas estructuras Z3 proporciona dos operaciones (*select* y *store*) que nos permiten acceder o actualizar una posición del array.
- **Tipos algebraicos:** una de las características por las que utilizamos Z3 es por su soporte para tipos algebraicos que pueden ser recursivos. Mediante el

comando *declare-datatypes* podemos definir nuevos tipos de datos que podemos utilizar a lo largo del archivo. Utilizaremos esta teoría para definir un conjunto de estructuras habituales como los árboles rojinegros o los AVL.

- **Cuantificadores:** esta teoría nos permite representar fórmulas lógicas que involucren cuantificadores como son las precondiciones y postciondiciones que suelen utilizar cláusulas de *para todo* o de existencia.

En concreto, para introducir la fórmula lógica $a > b \wedge b > c \wedge c > 0$ en Z3 escribiríamos las siguientes instrucciones:

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

Lo que nos ofrecería la siguiente salida:

```
sat
(model
  (define-fun c () Int
    1)
  (define-fun b () Int
    2)
  (define-fun a () Int
    3)
)
```

Por otro lado, un predicado más complejo como puede ser el que un array esté ordenado crecientemente entre los índices 0 y 5 podría escribirse como:

```
(declare-const a (Array Int Int))
(assert (forall ((i Int) (j Int))
        (=> (and (< (-1) i) (< i 6)
                 (< (-1) j) (< j 6) (< i j))
                 (< (select a i) (select a j)))))
(check-sat)
(get-model)
```

Que nos ofrece una salida bastante más complicada donde podemos ver que el array a se sintetiza como una función y cómo asigna valores para las 5 primeras posiciones:

```
sat
```

```
(model
  (define-fun a () (Array Int Int)
    (_ as-array k!20))
  (define-fun k!20!22 ((x!0 Int)) Int
    (ite (= x!0 3) (- 1)
    (ite (= x!0 2) (- 2)
    (ite (= x!0 1) (- 3)
    (ite (= x!0 0) (- 4)
    (ite (= x!0 4) 0
      7719))))))
  (define-fun k!21 ((x!0 Int)) Int
    (let ((a!1 (ite (<= 3 x!0) (ite (<= 4 x!0)
      (ite (<= 5 x!0) 5 4) 3) 2)))
        (ite (<= 1 x!0) (ite (<= 2 x!0) a!1 1) 0)))
  (define-fun k!20 ((x!0 Int)) Int
    (k!20!22 (k!21 x!0)))
)
```

2.3. Trabajos relacionados

2.3.1. La herramienta IR2Haskell

De las herramientas que se encuentran integradas dentro de la plataforma CAVI-ART cabe destacar para nuestro propósito IR2Haskell desarrollada en [2]. La herramienta consiste inicialmente en un análisis de la sintaxis de la CLIR mediante un analizador de S-expresiones y la producción de un árbol de sintaxis abstracta que representa enteramente la estructura del programa como un conjunto de tipos algebraicos en Haskell. Esta será la materia prima con la que trabajaremos pues en una primera fase analizaremos sintácticamente el archivo con esta herramienta y posteriormente trabajaremos con el AST y sus distintos nodos para realizar los análisis pertinentes.

El resto de la herramienta (que nosotros no utilizaremos) convierte el árbol abstracto del programa y de sus predicados (precondición y postcondición) en funciones Haskell ejecutables.

2.3.2. La herramienta Precd2Z3

Así mismo, en junio de 2018 se presentó la herramienta Precd2Z3 [5] que realizan una generación de casos de caja blanca para la CLIR. El objetivo de esta herramienta es similar al de la que estamos tratando en el presente documento a excepción que solo considera la precondición de cada programa de la CLIR. Así, consigue generar estructuras complejas, como por ejemplo árboles rojinegros, utilizando

Z3 que tienen un cardinal dado y, como es de esperar, se encuentran correctamente coloreados. Además mediante una serie de modificaciones posteriores a su presentación se han mejorado enormemente los tiempos de generación de estructuras. Actualmente la herramienta genera unas guías para que Z3 sea capaz de instanciar las estructuras algebraicas más rápido de lo que lo hacía en su primera versión.

Combinando estas restricciones sobre la estructura de los datos de entrada con las restricciones de los caminos que buscamos obtener conseguimos finalmente algo similar a una ejecución simbólica que nos va a permitir obtener unos valores de los datos de entrada que sean válidos y que además recorran los posibles caminos dentro del programa hasta un nivel de profundidad escogido.

Capítulo 3

Transformación a grafos

Dentro del proceso de análisis que pretendemos establecer, lo primero que realizamos es una transformación del programa de la CLIR en un grafo de control de flujo (*Control Flow Graph* o CFG en adelante). Un CFG pretende representar el flujo de ejecución de un programa y las distintas llamadas o saltos que puede realizar dentro del código. Mediante esta transformación podremos empezar a definir lo que sería un camino completo o parcial y qué implica la definición del mismo en lo que a restricciones se referirá.

En primer lugar, debemos especificar la estructura de un fichero *CLIR*:

1. Un bloque de documentación en el que se especifica aspectos como el lenguaje original del programa o una breve descripción del mismo. Omitiremos este bloque en los archivos que mostremos por carecer de relevancia para nuestro análisis.

2. Una única función *top-level* que llamaremos función externa. Esta función hemos considerado que no es visible desde el interior de su definición, evitando así llamadas recursivas y pudiendo escoger unívocamente un nodo como fuente del flujo de ejecución.

3. Una declaración de la precondición y la postcondición de la función externa. Omitiremos esta declaración en los archivos que mostremos por carecer de relevancia para nuestro análisis.

4. Una única cláusula **letfun** en la que se definen las funciones que serán llamadas durante el programa. Llamaremos a estas funciones *internas*. La cláusula **letfun** finaliza en una expresión que llama a una de las funciones internas.

Aunque pudiera parecer una transformación trivial, lo cierto es que al permitir la existencia de llamadas recursivas no finales no es posible definir un CFG lo suficientemente preciso como para reflejar las llamadas y por ello nos vemos forzados a recurrir a otro tipo de grafos que hemos denominado *grafos modulares* y que explicaremos más adelante. Así mismo, el problema se vuelve más complejo cuando

consideramos programas que tienen partes de ambos paradigmas, recursión final y recursión no final.

Por otro lado nos encontramos con la dificultad de definir qué es un camino de profundidad dada cuando el programa no es recursivo final puesto que una rama puede realizar varias llamadas seguidas a diferencia de los recursivos finales que solo realizan una llamada recursiva. Para poder ejemplificar todo esto vamos a escoger unos programas simples que usaremos como *user running examples* e iremos observando cada una de las transformaciones que sufran. En concreto como programa recursivo final consideraremos la inserción en un array (*insertArray.clir*) cuya IR se muestra en el Código 3.1 y como programa recursivo no final consideraremos la ordenación rápida (*quicksort.clir*) cuya IR se muestra en el Código 3.2.

Nótese la peculiaridad que vamos a obviar de que *quicksort.clir* realiza una llamada a la función partition que no se encuentra entre sus funciones internas. A pesar de que es un aspecto que aún queda abierto a la investigación, hemos mantenido la decisión de utilizar quicksort como ejemplo de programa recursivo no final puesto que nos parece interesante el hecho de que realice varias llamadas en una misma secuencia de instrucciones. Este hecho nos es útil para explicar ciertos conceptos que con ejemplos más sencillos se verían ensombrecidos. Sin embargo, actualmente no somos capaces de procesar las llamadas a funciones de otros archivos sino que utilizaremos en la práctica una versión de quicksort modificada que se presenta en la Sección 3.3.3.

Código 3.1: insertArray.clir

```
(define insert ((x Int) (m Int) (a (Array Int))) ((res (Array Int)))
  (letfun (
    (f2 ((x Int) (m Int) (i Int) (a (Array Int))) ((res2 (Array Int)))
      (let ((b1 Bool)) (@ >= i (the Int 0))
        (case b1 (
          ((@@ false)
            (@ f4 x m i a))
          ((@@ true)
            (let ((e Int)) (@ get-array a i)
              (let ((b2 Bool)) (@ < x e)
                (case b2 (
                  ((@@ true)
                    (let ((u Int)) (@ get-array a i)
                      (let ((i2 Int)) (@ + i (the Int 1))
                        (let ((ap (Array Int))) (@ set-array a i2 u)
                          (let ((i3 Int)) (@ - i (the Int 1))
                            (@ f2 x m i3 ap))))))
                  ((@@ false)
                    (@ f4 x m i a)))))))))))

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

  )
  (let ((i Int)) (@ - m (the Int 1))
```

```
    (@ f2 x m i a))))
```

Código 3.2: quicksort.clir

```
(define quick ((V (Array Int))) ((vres (Array Int)))
  (letfun(
    (f1 ((i Int) (j Int) (V (Array Int))) ((res (Array Int)))
      (let ((p Int) (Vp (Array Int))) (@ partition i j V)
           (let ((p1 Int)) (@ - p (the Int 1))
                (let ((V1 (Array Int))) (@ qsort i p1 Vp)
                  (let ((p2 Int)) (@ + p (the Int 1))
                       (@ qsort p2 j V1))))))

    (qsort ((i Int) (j Int) (V (Array Int))) ((res (Array Int)))
          (let ((b Bool)) (@ <= i j)
            (case b (
              ((@@ true)
                (@ f1 i j v))
              ((@@ false)
                V)))))
  )
  (let ((n Int)) (@ len V)
    (let ((n1 Int)) (@ - n (the Int 1))
         (@ qsort (the Int 0) n1 V)))))
```

3.1. CFG planos y arborescentes

A la hora de generar grafos de los programas hemos considerado unas estructuras típicas de los análisis realizados por los compiladores: los bloques básicos (BB) y los bloques de casos (BC). Un bloque básico se define como un bloque de ejecución que no contiene bifurcaciones de caminos y que acaba en una expresión atómica, una llamada a otra función o un salto a un bloque de casos. Un bloque de casos se define como el conjunto de las ramas en las que se bifurca el flujo de ejecución de un programa o función. Los bloques de casos se generan cuando aparece una expresión **case** en la CLIR.

Cabe destacar que una función siempre empieza con un BB aunque la primera expresión sea un **case** en cuyo caso el bloque básico simplemente contiene el salto al bloque de casos. De igual modo, si se encadenan varias expresiones **case** se introducen bloques básicos (que solo contienen el salto al siguiente BC) entre medias para facilitar el análisis. Podemos ahora pasar a analizar el CFG que se genera en función de si el programa es recursivo final o no.

Cuando consideramos los CFG de los programas, debemos destacar 4 propiedades que aunque puedan parecer elementales sientan las bases para poder realizar los análisis que pretendemos llevar a cabo:

1. Existe un único nodo denominado *fuente* en el que han de comenzar todos los caminos y que solo tiene aristas salientes.

2. Existe un único nodo denominado *sumidero* en el que han de finalizar todos los caminos y que solo tiene aristas entrantes.

3. Existe al menos un camino que conecta la fuente y el sumidero mediante nodos intermedios.

4. Desde la fuente se puede llegar a cualquier nodo del grafo y desde cualquier nodo del grafo se puede llegar al sumidero. No existen por tanto, en terminología de autómatas finitos, ni nodos inalcanzables ni *nodos trampa*.

3.1.1. Programas recursivos finales

En el caso de los programas recursivos finales un CFG describe completamente la estructura de la ejecución de un programa. Esto es debido a que cuando se realiza una llamada recursiva final el flujo de ejecución pasa completamente a la nueva función sin retornar. Así, un programa se representa mediante un primer nodo fuente seguido del bloque básico de la función declarada en la cláusula **define**. Dado que no se permiten llamadas a la función externa, es claro que este bloque básico va a finalizar con una llamada a una de las funciones internas (obviando el caso trivial de que no hubiera funciones internas que carece de interés). Una vez comenzado el grafo, se va siguiendo el flujo de ejecución comenzando bloques básicos en las funciones a las que se llame o tras las construcciones **case** que se encuentren. Finalmente, cuando encontremos que el flujo de ejecución termina en una expresión atómica uniremos el nodo actual con el sumidero completando así el grafo.

Se puede observar el grafo generado por el programa insert.clir en la Figura 3.1 con una estructura reflejada en la Tabla 3.1 donde la expresión BB indica un salto a un bloque básico, BC indica un salto a un bloque de casos y S indica un salto al sumidero devolviendo el valor que se pasa en el primer parámetro.

3.1.2. Programas recursivos no finales

Para los programas recursivos no finales, un CFG plano no describe completamente la estructura de la ejecución de un programa puesto que no tenemos manera de representar el retorno de una función. El flujo de ejecución en este caso se traspasa temporalmente a la función llamada pero al retornar no lo hace al comienzo de la función actual sino que continúa desde un punto intermedio. Por este camino el mejor análisis que podemos realizar es el estudio de las llamadas que se realizan que en el caso de *quicksort.clir* genera una estructura arborescente como podemos observar en la Figura 3.2.

Se puede observar que los arcos entre nodos adquieren unos significados especiales en este contexto pues los arcos horizontales simbolizan que tras un nodo

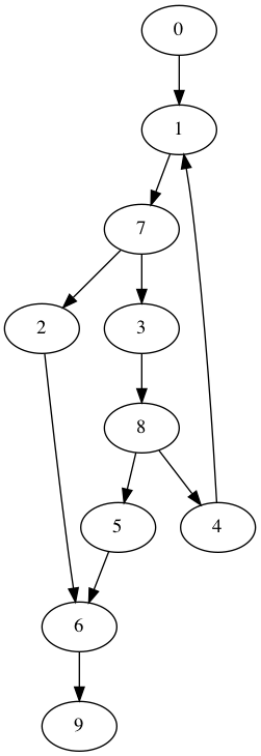

Figura 3.1: CFG de *insertArray.clir*

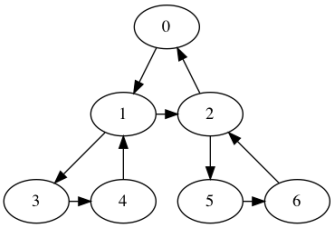

Figura 3.2: CFG arborescente de quicksort.clir

vendrá la ejecución del siguiente quizá con instrucciones intermedias del nodo padre de ambos y las flechas ascendentes simbolizan que se retorna el control del programa al nodo padre. Debido a estos matices, este análisis no nos conduce a ningún resultado válido sobre el flujo de ejecución por lo que para estudiar estos programas utilizaremos lo que denominamos un *grafo modular*.

3.2. Grafos modulares

Debido a las dificultades que se nos presentan a la hora de estudiar los programas recursivos no finales, hemos de buscar otro enfoque para estos. Para obtener un análisis más fino, definimos un concepto nuevo conocido como *grafo modular* en el que se representan cada una de las funciones por separado y en las llamadas sólo especificamos el nombre de la función llamada en vez de unir los

Bloque	**Tipo**	**Función**	**Código**
0	Fuente	insert	(let ((i Int)) (@ - m (the Int 1)) (@ f2 x m i a)))
1	BB	f2	(let ((b1 Bool)) (@ >= i (the Int 0)) (BC 7))
2	BB		(@ f4 x m i a)
3	BB		(let ((e Int)) (@ get-array a i) (let ((b2 Bool)) (@ < x e) (BC 8)))
4	BB		(let ((u Int)) (@ get-array a i) (let ((i2 Int)) (@ + i (the Int 1)) (let ((ap (Array Int))) (@ set-array a i2 u) (let ((i3 Int)) (@ - i (the Int 1)) (@ f2 x m i3 ap)))))
5	BB		(@ f4 x m i a)
6	BB	f4	(let ((i2 Int)) (@ + i (the Int 1)) (let ((ap (Array Int))) (@ set-array a i2 x) (S ap))))
7	BC		(case b1 (((@@ false) (BB 2)) ((@@ true) (BB 3))))
8	BC		(case b2 (((@@ true) (BB 4)) ((@@ false) (BB 5))))
9	Sumidero		

Tabla 3.1: Significado de los bloques de 3.1

vértices con arcos. Un grafo modular entonces constará de nodos de bloques básicos, nodos de bloques de casos y nodos de llamadas además de la fuente y el sumidero.

Para obtener el grafo modular de un programa procedemos de la siguiente manera:

1. Transcribimos la función externa como una función interna más.

2. Para cada función interna generamos sus bloques básicos y bloques de casos como hemos hecho en los CFG. En caso de encontrar una llamada a otra función interna, anotamos en el nodo cuál es la función a la que se está llamando y continuamos con un nuevo bloque básico.

3. Conectamos la fuente al primer bloque de la función externa.

4. Fusionamos los sumideros de todas las funciones en uno mismo que va a ser el nodo sumidero de todas ellas.

5. Para cada nodo que se conecta al sumidero añadiremos un nuevo nodo intermedio que simbolizará el retorno de la función. De esta manera seremos capaces en todo momento de realizar un seguimiento por la función en la que se encuentra el flujo de ejecución.

Así, obtenemos un grafo que representa el flujo de ejecución para cada función ocultando el flujo interno de las llamadas que haya. De este modo podemos representar y analizar correctamente programas recursivos tanto finales como no finales. A pesar de que en los programas recursivos finales podríamos realizar el análisis utilizando solo el CFG plano, consideramos interesante la unificación del proceso y por tanto realizamos el análisis de los caminos en el grafo modular en ambos casos.

Podemos observar el grafo modular de la inserción en un array en la Figura 3.3 (con su tabla de referencia 3.2) y de la ordenación rápida en la Figura 3.4 (con su tabla de referencia 3.3).

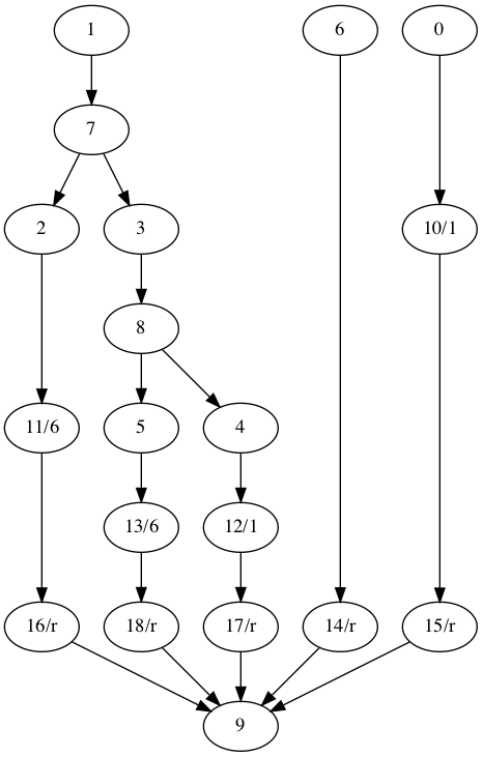

Figura 3.3: Grafo Modular de *insertArray.clir*

Bloque	**Tipo**	**Función**	**Código**
0	Fuente	insert	(let ((i Int)) (@ - m (the Int 1)) (BL 10)))
1	BB	f2	(let ((b1 Bool)) (@ ¿= i (the Int 0)) (BC 7))
2	BB		(BL 11)
3	BB		(let ((e Int)) (@ get-array a i) (let ((b2 Bool)) (@ ¡x e) (BC 8)))
4	BB		(let ((u Int)) (@ get-array a i) (let ((i2 Int)) (@ + i (the Int 1)) (let ((ap (Array Int))) (@ set-array a i2 u) (let ((i3 Int)) (@ - i (the Int 1)) (BL 12)))))
5	BB		(BL 13)
6	BB	f4	(let ((i2 Int)) (@ + i (the Int 1)) (let ((ap (Array Int))) (@ set-array a i2 x) (S ap))))
7	BC		(case b1 (((@@ false) (BB 2)) ((@@ true) (BB 3))))
8	BC		(case b2 (((@@ true) (BB 4)) ((@@ false) (BB 5))))
9	Sumidero		
10	BL		(S (@ f2 x m i a))
11	BL		(S (@ f4 x m i a))
12	BL		(S (@ f2 x m i3 ap))
13	BL		(S (@ f4 x m i a))

Tabla 3.2: Significado de los bloques de 3.3

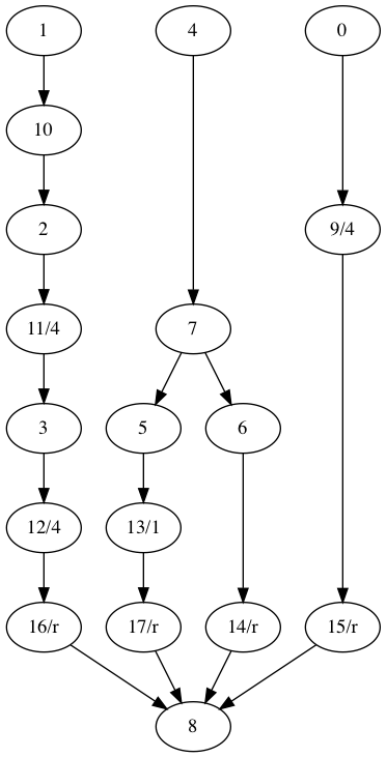

Figura 3.4: Grafo Modular de quicksort.clir

Bloque	Tipo	Función	Código
0	Fuente	quick	(let ((n Int)) (@ len V) (let ((n1 Int)) (@ - n (the Int 1)) (BL 9)))
1	BB	f1	(let ((p Int) (Vp (Array Int))) (BL 10))
2	BB		(let ((p1 Int)) (@ - p (the Int 1)) (let ((V1 (Array Int))) (BL 11)))
3	BB		(let ((p2 Int)) (@ + p (the Int 1)) (BL 12))
4	BB	qsort	(let ((b Bool)) (@ <= i j) (BC 7))
5	BB		(BL 13)
6	BB		(S V)
7	BC		(case b (((@@ true) (BB 5)) ((@@ false) (BB 6))))
8	Sumidero		
9	BL		(S (@ qsort (the Int 0) n1 V))
10	BL		(BB 2 (@ partition i j V))
11	BL		(BB 3 (@ qsort i p1 Vp))
12	BL		(S (@ qsort p2 j V1))
13	BL		(S (@ f1 i j v))

Tabla 3.3: Significado de los bloques de 3.4

3.3. ¿Qué es un camino de profundidad n?

Cuando tratamos de considerar caminos en programas que presentan recursión es bien sabido que esta recursión puede ser infinita. Por ello en el análisis que estamos realizando hemos considerado la inclusión de un *nivel de profundidad máximo* que limite en número de llamadas recursivas que estamos dispuestos a considerar en los caminos que vamos a generar pero dada la ambigüedad del concepto de *profundidad* en un camino creemos necesario especificar a qué nos referimos y cómo se calculará este valor.

Generaremos unas nuevas etiquetas para los nodos que vamos a denominar *nodos de control* para simbolizar los nodos en los cuales consideraremos que al pasar por ellos hemos aumentado en una unidad el nivel de profundidad del camino actual.

3.3.1. Programas recursivos finales

Cuando estamos tratando programas recursivos finales la profundidad es un concepto claro puesto que podemos generar un CFG plano. Hemos visto que en el CFG aparecen componentes fuertemente conexas (en adelante SCC de su denominación en inglés *Strongly Connected Component*), es decir, conjuntos de vértices dentro de los cuales es posible llegar desde un vértice a cualquier otro. Estas SCC vienen generadas por llamadas recursivas que o bien ya lo eran en el programa original o se han generado de la transformación de bucles en los lenguajes imperativos. Cabe destacar que por la transformación que hemos realizado es imposible que dos SCC coincidan en un único punto puesto que los puntos de entrada deben ser comienzos de funciones y si coincidiesen en un único punto y no en dos significaría que existe una bifurcación en un comienzo de función lo que es imposible pues las funciones empiezan siempre por un bloque básico.

Es claro que toda SCC dentro de un CFG tiene que tener un único punto de entrada y puede tener varios puntos de salida. Podemos entonces considerar distintos caminos parciales que pasen por la SCC siendo estos los que pasan por el punto de entrada como primer vértice y no vuelven a pasar por él sino que salen antes por uno de los sumideros. Además estos caminos pueden venir precedidos por tantas vueltas de la SCC como queramos, es por ello que cada una de estas vueltas será considerada como un nivel de profundidad, o lo que es lo mismo, el último nodo antes de volver al punto de entrada de la SCC será considerado el nodo de control. De este modo, el nivel de profundidad coincide con el número de iteraciones al bucle en una CLIR que proviene de un programa imperativo.

En el ejemplo que venimos trabajando (*insert.clir*, Figura 3.1) podemos considerar la SCC formada por los vértices $[1, 7, 3, 8, 4]$ donde tomamos como punto de entrada el vértice 1 y como puntos de salida los vértices $[7, 8]$. Es claro entonces que podemos considerar los caminos $[1, 7, 2]$ y $[1, 7, 3, 8, 5]$. Además podemos escoger que se de una vuelta a la SCC y generar así el camino parcial $[1, 7, 3, 8, 4]$ que podemos continuar consigo mismo tantas veces como queramos o finalizar con uno de los caminos antes mencionados. Así, el número de veces que se incruste al comienzo el camino $[1, 7, 3, 8, 4]$ representa el nivel de profundidad del camino. De este modo, el nodo de control sería el nodo 4.

En el caso de que hubiera más de una SCC existen varias posibilidades aunque todas se resuelven de la misma manera. Dado un programa con varias SCCs, ya sean disjuntas o no, consideraremos que el nivel de profundidad será el número de vueltas que se den a cada una de las SCCs, es decir, el número de veces que se pase por sus nodos de control.

3.3.2. Programas recursivos no finales

En caso de estar tratando programas recursivos no finales, no contamos con el correspondiente CFG y no podemos realizar el análisis de las SCCs. Utilizaremos entonces para el análisis el grafo modular generado por el programa.

En una primera aproximación, se podría tratar de entender el nivel de profundidad como el número de veces que se realiza una llamada recursiva o lo que es lo mismo, el número de veces que se pasa por el primer nodo de la función recursiva. Esto sin embargo nos lleva a resultados erróneos a nuestro parecer pues en el caso de *quicksort.clir*, aquel programa que tenga que particionar el array una vez realizaría un camino de profundidad dos cuando lo lógico sería considerar éste como un camino de profundidad uno.

Así, viendo los grafos arborescentes que hemos comentado que se generan en los programas recursivos no finales, hemos considerado que para contabilizar en nivel de profundidad, éste viene dado por el número de veces que se pasa por el primer nodo de la rama donde se van a realizar las llamadas.

Conceptualmente el proceso que vamos a realizar es el siguiente: para cada llamada no final recorreremos el grafo en dirección contraria hasta encontrar un bloque de casos o en caso de llegar al comienzo de una función repetiremos el proceso desde todos los nodos que representen llamadas a esta función. Así, marcaremos el primer nodo de la rama que va a provocar las llamadas recursivas. Es claro que es necesaria la existencia del mismo siempre y cuando el programa recursivo tenga algún caso base pues esto provoca la existencia de una cláusula **case** y por tanto de un bloque de casos. En caso de no existir dicha cláusula, es trivial ver que el programa entraría en recursión infinita y por tanto estaría mal codificado.

Así, dado el grafo modular de *quicksort.clir* (Figura 3.4), vemos que solo hay una llamada recursiva no final en el nodo 11. Si recorremos el grafo en dirección inversa llegamos al nodo 1 que es el comienzo de una función y por tanto hemos de buscar desde donde se realiza la llamada a dicha función. Localizamos la única llamada en el nodo 13 y recorriendo el grafo en sentido contrario vemos que el primer nodo de la rama del case es el nodo 5. Por tanto, asumiremos que el camino tiene tantos niveles de profundidad como veces pase por el nodo 5, es decir, que el nodo 5 es un nodo de control. Así, el nivel de profundidad en un programa recursivo no final se identifica con la altura de su CFG arborescente.

3.3.3. Programas mixtos

Debido a la existencia de programas que presenten bucles pero también llamadas recursivas no finales nos encontramos que la definición y el proceso que hemos ofrecido no son suficientes para tratar estos casos. Estos casos son los que presentan una mayor complejidad puesto que es necesario realizar un análisis que combine las dos técnicas utilizadas hasta el momento. Así, dado que todos los programas pueden ser expresados en forma de grafo modular, cuando encontremos un programa que no sea solo recursivo final, generaremos en primer lugar su grafo modular y realizaremos el análisis que hemos descrito para los programas recursivos no finales a fin de obtener los puntos de control del nivel de recursión de la parte no final del programa.

Tras ello, pasamos a transformar los nodos que simbolizan llamadas y que

no se encuentran conectados a nodos de control del nivel de profundidad (lo que asegura la consistencia del método pues evitaremos la aparición de falsos bucles que vengan generados por las llamadas recursivas no finales) en arcos hasta el nodo llamado. De este modo destruimos la estructura del grafo modular para pasar a tener una estructura de CFG en la parte recursiva final del programa. Denominaremos esto como *grafo mixto*. En este momento realizaremos el análisis por SCCs que realizabamos en los programas recursivos finales a fin de obtener los nuevos nodos de control y, junto con los que hemos calculado anteriormente, formar la lista completa de nodos de control.

Como ejemplo podemos considerar el programa *quicksortMod.clir* (Código 3.3) que no es sino una versión de *quicksort* que tiene como funciones internas todas las que necesita. Del código proporcionado podemos generar el grafo modular de la Figura 3.5 en el que por el análisis de los programas recursivos no finales llegamos a marcar como nodo de control el nodo 13. Finalmente, en la Figura 3.6 se puede comprobar el resultado de realizar el proceso que hemos detallado en esta sección (donde se puede comprobar que la única llamada que no se ha desarrollado es el nodo "28/9"porque viene precedido por un nodo de control, el nodo 13 y si lo uniésemos daría lugar a bucles por recursiones no finales) y mediante un análisis de SCCs podemos ver fácilmente que los nuevos nodos de control son los nodos 4, 6 y 7.

Podemos entonces desde este punto definir los caminos alrededor de los nodos de control $[4, 6, 7, 13]$ evitando así ciclos infinitos en los análisis al no haber marcado los bucles recursivos finales.

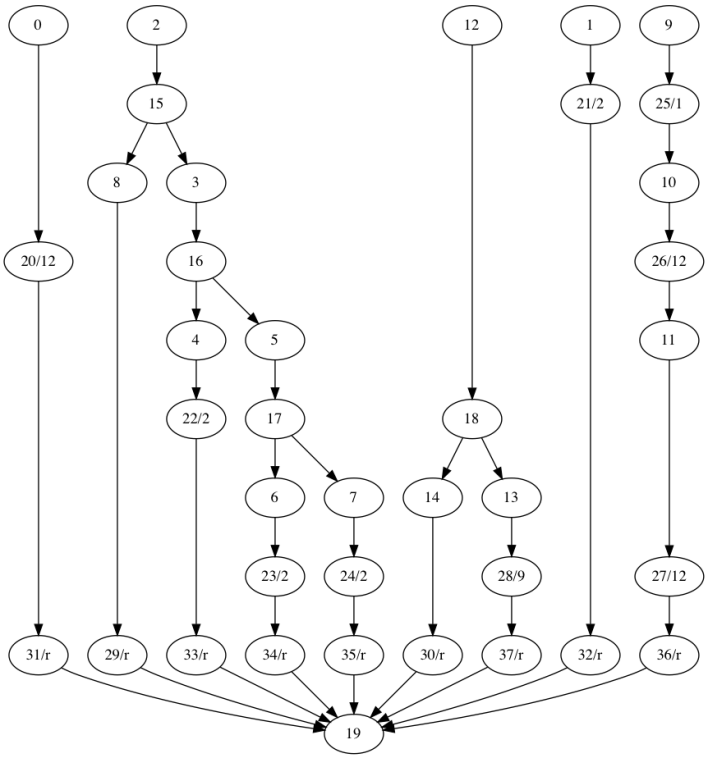

Figura 3.5: Grafo modular de quicksortMod.clir

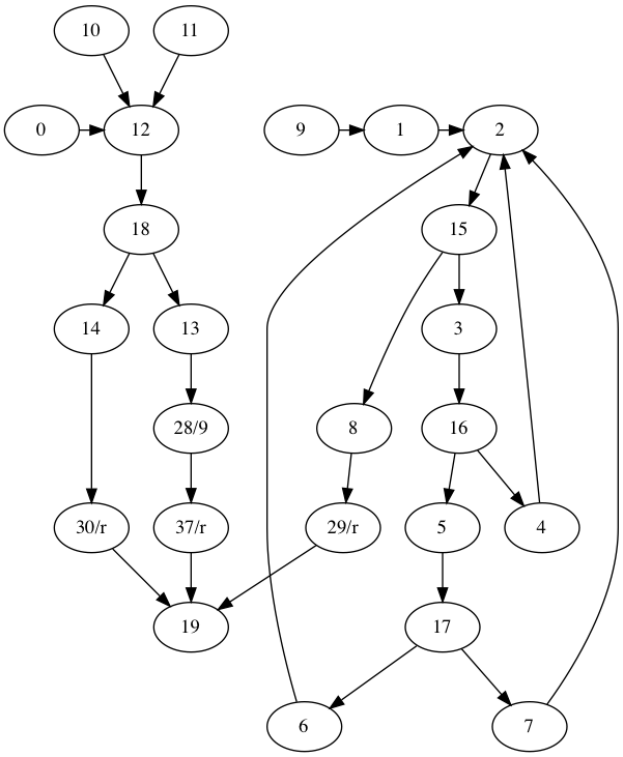

Figura 3.6: Grafo mixto de quicksortMod.clir

Código 3.3: quicksortMod.clir

```
(define quick ((V (Array Int))) ((vres (Array Int)))
   (letfun (
    (partition ((a Int) (b Int) (V (Array Int))) ((res1 Int) (res2 (Array Int)))
      (let ((i Int)) (@ + a (the Int 1))
        (let ((j Int)) b
          (@ f2 a b i j V))))

    (f2 ((u Int) (v Int) (i Int) (j Int) (a (Array Int)))  ((res1 Int) (res2 (Array Int)))
      (let ((b2 Bool)) (@ <= i j)
        (case b2 (
          ((@@ true)
            (let ((ei Int)) (@ get-array a i)
              (let ((eu Int)) (@ get-array a u)
                (let ((ej Int)) (@ get-array a j)
                  (let ((b31 Bool)) (@ < eu ei)
                    (case b31 (
                      ((@@ false)
                        (let ((i1 Int)) (@ + i (the Int 1))
                          (@ f2 u v i1 j a)))
                      ((@@ true)
                        (let ((b32 Bool)) (@ < ej eu)
                          (case b32 (
                            ((@@ true)
                              (let ((a1 (Array Int))) (@ set-array a i ej)
                                (let ((a2 (Array Int))) (@ set-array a1 j ei)
                                  (let ((i1 Int)) (@ + i (the Int 1))
                                    (let ((j1 Int)) (@ - j (the Int 1))
                                      (@ f2 u v i1 j1 a2))))))
                            ((@@ false)
                              (let ((j1 Int)) (@ - j (the Int 1))
                                (@ f2 u v i j1 a))))))))))))))
          ((@@ false)
            (let ((eu Int)) (@ get-array a u)
              (let ((ej Int)) (@ get-array a j)
                (let ((a1 (Array Int))) (@ set-array a u ej)
                  (let ((a2 (Array Int))) (@ set-array a1 j eu)
                    (@ pair j a2)))))))))))

    (f1 ((i Int) (j Int) (V (Array Int))) ((res (Array Int)))
      (let ((p Int) (Vp (Array Int))) (@ partition i j V)
        (let ((p1 Int)) (@ - p (the Int 1))
          (let ((V1 (Array Int))) (@ qsort i p1 Vp)
            (let ((p2 Int)) (@ + p (the Int 1))
              (@ qsort p2 j V1))))))
```

```
  (qsort ((i Int) (j Int) (V (Array Int))) ((res (Array Int)))
    (let ((b Bool)) (@ <= i j)
      (case b (
        ((@@ true)
          (@ f1 i j v))
        ((@@ false)
          V)))))
)
(let ((n Int)) (@ len V)
  (let ((n1 Int)) (@ - n (the Int 1))
    (@ qsort (the Int 0) n1 V)))))
```

Capítulo 4

Generación de los caminos

Una vez contamos con los grafos que representan un programa, podemos utilizar el grafo modular para obtener los distintos caminos hasta una profundidad dada. De hecho, en los programas recursivos finales podemos realizar el análisis en el CFG pero con el objetivo de unificar el proceso, preferimos realizar el análisis en el grafo modular. Sin embargo, esto presenta algunas dificultades prácticas típicas de los programas recursivos como son el renombramiento de las variables para lo cual hemos requerido de la utilización de una mónada transformadora de estados en Haskell.

4.1. Caminos como listas de vértices

En una primera fase, una vez que disponemos del grafo modular de un programa, seguiremos un sencillo algoritmo para obtener los caminos de una profundidad dada:

1. Comenzamos en el nodo 0 como nodo fuente.
2. Conectamos con un camino el nodo fuente y el nodo sumidero.
3. Si encontramos un nodo marcado como nodo de control del nivel de profundidad y éste es mayor que cero, decrementamos el mismo y continuamos el proceso. Si es cero o menor, abortamos el camino.
4. Para los nodos que representen llamadas, repetir el proceso comenzando desde el primer nodo de la función llamada.

Así los caminos quedan representados por una lista de vértices que comienzan en el nodo 0 y finalizan en el nodo sumidero. Conceptualmente esta lista representa los bloques básicos por los que va pasando el camino y las ramas de las cláusulas **case** que se van tomando. Nótese que es posible discernir cuando un camino entra y sale de una función pues sabemos qué nodos simbolizan entradas a

funciones y hemos añadido unos nodos auxiliares para representar el retorno de las funciones que ahora aparecen en la lista.

De este modo, en *insertArray.clir* con un nivel de profundidad de 2 podemos encontrar los caminos mostrados en la Tabla 4.1. Como se puede comprobar, el número de veces que se pasa por el nodo 4 coincide con el nivel de profundidad del camino como veníamos advirtiendo. Por otro lado, en la Tabla 4.2 podemos observar los caminos de *quicksort.clir* con un nivel de profundidad de 2.

Profundidad	**Caminos**
0	[0,1,7,2,6,14,16,15,9] [0,1,7,3,8,5,6,14,16,15,9]
1	[0,1,7,3,8,4,1,7,2,6,14,16,15,9] [0,1,7,3,8,4,1,7,3,8,5,6,14,16,15,9]
2	[0,1,7,3,8,4,1,7,3,8,4,1,7,2,6,14,16,15,9] [0,1,7,3,8,4,1,7,3,8,4,1,7,3,8,5,6,14,16,15,9]

Tabla 4.1: Caminos en insertArray.clir hasta profundidad 2

Profundidad	**Caminos**
0	[0,4,7,6,14,15,8]
1	[0,4,7,5,1,10,2,4,7,6,14,3,4,7,6,14,16,17,15,8]
2	[0,4,7,5,1,10,2,4,7,5,1,10,2,4,7,6,14, 3,4,7,6,14,16,17,3,4,7,6,14,16,17,15,8] [0,4,7,5,1,10,2,4,7,6,14,3,4,7,5,1,10, 2,4,7,6,14,3,4,7,6,14,16,17,16,17,15,8] [0,4,7,5,1,10,2,4,7,5,1,10,2,4,7,6,14,3,4, 7,6,14,16,17,3,4,7,5,1,10,2,4,7,6,14,3,4,7, 6,14,16,17,16,17,15,8]

Tabla 4.2: Caminos en quicksort.clir hasta profundidad 2

4.2. Renombramiento de variables

Dada la inexistencia de bucles en la sintaxis de la IR, la gestión de bloques de código repetitivos se realiza mediante recursión lo que plantea el problema de las referencias a las variables. En los lenguajes de alto nivel, este problema se resuelve por el compilador mediante la utilización de tablas de nombres donde se almacenan las referencias a cada una de las instancias. Sin embargo, dado que nosotros estamos realizando un proceso de mucho menor nivel de abstracción debemos desarrollar una estrategia que nos permita identificar de manera unívoca cada aparición de las variables pero nos vemos limitados a no poder utilizar tablas de nombres pues la interfaz de Z3 no trata referencias sino nombres de variables.

El proceso que definimos se centra en añadir un prefijo que identifique la función a la que pertenece la variable y el nivel de recursión en el que nos encontramos

en esa función. Este es un problema típico de iteración a través de una lista de elementos almacenando un estado. En el caso de nuestra implementación hemos necesitado hacer uso de la mónada transformadora de estados ST de Haskell [7].

Dado que los nodos en los que comienzan las funciones están unívocamente identificados y todas las aristas al sumidero están identificadas mediante un nodo de retorno, somos capaces de identificar cuando el flujo se encuentra dentro de una determinada función. Así, podemos tomar el camino [0, 4, 7, 5, 1, 10, 2, 4, 7, 6, 14, 3, 4, 7, 6, 14, 16, 17, 15, 8] en *quicksort.clir* y analizarlo para ver en qué punto nos encontramos en cada nodo.

En primer lugar, podemos ver que el comienzo [0, y el final 15, 8] corresponden a la función externa *quick*. Para poder integrar nuestra herramienta con Precd2Z3, en el caso de la función externa no vamos a añadir ningún prefijo. Esto no es problemático puesto que hemos restringido la CLIR para que no puedan existir llamadas a la función externa. De este modo, prohibimos la posibilidad de que haya niveles de recursión sobre la función externa y que por tanto no sean necesarios los prefijos.

Tras ello, vemos que la secuencia 4, 7, 5 . . . 17 corresponde a uno de los posibles caminos a través de la función *qsort*. Tenemos por tanto que vamos a necesitar el prefijo "qsort_0_" que dejamos preparado en esos nodos. Por otro lado, la secuencia 1 . . . 2 . . . 3 . . . 16 es el camino que atraviesa la función *f1* por lo que podemos preparar para estos nodos el prefijo "f1_0_".

Puesto que ya hemos comentado que no tratamos las llamadas externas. vamos a obviar el caso del nodo 10 y vamos a apilar de manera artificial el prefijo "partition_0_" con fines meramente ilustrativos. Por último, las dos secuencias 4, 7, 6, 14 se corresponden con caminos dentro de la función *qsort* por lo que podemos asociar los prefijos "qsort_1_" y "qsort_2_" con cada una de las apariciones de dicha secuencia.

De este modo, llegamos a obtener la secuencia de prefijos que se refleja en la Tabla 4.3. Utilizando dichos prefijos, podemos renombrar todas las variables de dichos bloques (teniendo en cuenta que las llamadas hacen referencia a variables de otros bloques) generando así unos nombres de variables únicos para cada nivel de recursión. De este modo, podemos utilizar Z3 para que procese el problema con las variables renombradas, evitando así referencias erróneas a las variables. Nótese que como la generación de los prefijos se realiza siguiendo la estructura de un determinado camino, en el árbol de llamadas la numeración se realiza *en profundidad*.

Nodo	**Prefijo**
0	-
4	qsort_0_
7	qsort_0_
5	qsort_0_
1	f1_0_
10	partition_0_
2	f1_0_
4	qsort_1_
7	qsort_1_
6	qsort_1_
14	qsort_1_
3	f1_0_
4	qsort_2_
7	qsort_2_
6	qsort_2_
14	qsort_2_
16	f1_0_
17	qsort_0_
15	-
8	-

Tabla 4.3: Asociación de prefijos y nodos.

Capítulo 5

Generación de las restricciones

Una vez hemos conseguido obtener los caminos dentro de los grafos y también hemos preparado la estrategia para el renombramiento de variables podemos pasar ya a la última fase antes de la ejecución del archivo SMT. Utilizando la IR (considerando su división en BB) y las listas de nodos, podemos generar una ristra de restricciones para cada uno de los caminos. Necesitamos en este momento establecer una equivalencia entre las expresiones y cláusulas de la IR y restricciones en el lenguaje SMT-LIB. Llegados a este punto consideramos que las variables ya han sido correctamente renombradas en cada BB del camino acorde a las reglas que hemos proporcionado en el anterior capítulo. Si este no fuera el caso, conjuntamente con la traducción habría que realizar el renombramiento de las variables que intervienen en la instrucción que estemos tratando.

5.1. Equivalencias entre las construcciones de la IR y restricciones Z3

Surge ahora en el punto que nos encontramos del proceso de análisis la necesidad de tener una equivalencia entre las construcciones de la sintaxis de la IR, mostradas en la figura 2.2 y las construcciones aceptadas por Z3. Nótese que a pesar de existir construcciones complejas dentro del estándar de Z3 (como la cláusula **match**) hemos tratado de rebajar todo al menor nivel posible, es decir, a meras declaraciones y asignaciones. La equivalencia no se realiza del programa entero sino que se utiliza para transformar un camino concreto en un conjunto de restricciones, obviando así los demás caminos, instrucciones no cubiertas y ramas de cláusulas **case** no utilizadas.

5.1.1. Estructuras de datos

En conjunto con las construcciones de la IR, también encontramos en los programas estructuras de datos que requieren de una representación en Z3. Para ello

vamos a asumir la definición de las mismas que se realiza en [5].

- **Arrays**: dado que los arrays de Z3 son, como hemos comentado, funciones totales, la necesidad de representar arrays de un tamaño dado es solventada creando un nuevo tipo *Arr* que internamente es un par Array-Entero donde el segundo elemento simboliza el tamaño del array. Así, aquellos valores que sean mayores que el segundo elemento pueden ser obviados y los valores que se asignen en la función sintetizada pueden ser desechados. De este modo, los arrays se construyen con *mk-pair* y tienen dos destructoras *first* y *second*.
- **Listas**: las listas presentan una construcción típica de los lenguajes funcionales con dos constructores, *nil* y *cons*, y dos destructoras *hd* y *tl*.
- **Árboles**: en el caso de los árboles binarios, la estructura no difiere de la conocida estructura de árbol con dos constructoras, *leaf* y *node*, y con tres destructoras, *value*, *izq* y *der*.
- **AVL y LLRB**: en el caso de los AVL y de los árboles rojinegros la estructura es idéntica a la de los árboles binarios añadiendo la altura y el color del nodo respectivamente.
- **Set y Multiset**: por último, los TADs conjunto y multiconjuntos de elementos de tipo T son representados por un array de índices T en booleanos y por un array de índices T en enteros respectivamente, simbolizando en el primer caso la existencia o no del elemento en el conjunto y en el segundo caso el número de apariciones del elemento en el multiconjunto.

5.1.2. La cláusula *Let*

Una cláusula **let** en el lenguaje de la IR realiza dos acciones: en primer lugar declara una nueva variable y en segundo lugar le asigna un valor. Para la primera acción podemos realizar una transcripción directa a la sintaxis de Z3:

$$\textbf{let } \langle \overline{x_i :: \tau_i} \rangle = be \textbf{ in } e \Rightarrow \langle (declare - const\ x_i\ \tau_i) \rangle$$

Por otro lado, para la segunda acción podemos encontrarnos con mayores problemas pues esta es una expresión ligada. Esto significa que puede ser un valor constante, una variable, una llamada a una función del sistema, una llamada a una función definida en el programa, una tupla o un constructor. Así, podemos discernir los casos presentados en la Tabla 5.1. Nótese que la última posibilidad se refiere a funciones del sistema en las que solo hace falta aplicar sus argumentos. Dado que se dan por supuesto funciones del sistema que en Z3 no tienen equivalente, hace falta realizar una traducción de la función al formato de Z3. Así, podemos distinguir ciertas funciones del sistema que requieren traducción (ver Tabla 5.2). Podremos entonces tratar las funciones del sistema de una manera muy análoga a las expresiones atómicas pues para nosotros representarán valores y no llamadas a funciones

Expresión ligada	Traducción a Z3
let $x :: \tau = a$ **in** e	$(assert(= \ x \ a))$
let $x :: \tau = C \ \overline{a_i}$ **in** e	$(assert(= x(C \ \overline{a_i})))$
let $\overline{x_i :: \tau_i} = f \ \overline{a_i}$ **in** e	$(assert(= x(z(f) \ \overline{a_i})))$

Tabla 5.1: Traducción a Z3 de la cláusula *let*

ni variaciones en el flujo de control del programa. La función $z(f)$ representa la transformación de la función f a su equivalente en Z3.

Así mismo, actualmente no contamos con capacidad para representar el tipo de datos tupla por lo que se obvia la construcción en el caso del constructor de tuplas. Aún así, sí que existe soporte para devolver múltiples valores desde una función y asignar a múltiples valores haciendo una correspondencia uno a uno.

Función	**Traducción a Z3**
get-array a b	(select (first a) b)
set-array a b c	(mk-pair (store (first a) b c) (second a))
len a	(second a)
semisum a b	(div (+ a b) 2)

Tabla 5.2: Traducción a Z3 algunas funciones comunes

El caso de las funciones definidas en el programa será tratado más adelante puesto que requiere de una mayor precisión.

5.1.3. La cláusula *Case*

Como hemos comentado, esta traducción se centra en un único camino por lo que no nos va a hacer falta generar una estructura muy compleja. Además la expresión atómica se encuentra ya definida antes del **case** por lo que tampoco vamos a necesitar realizar una declaración de la misma.

La traducción a Z3 simplemente consiste en generar un aserto de igualdad entre la variable y el patrón de la rama escogida. Esto presenta algunas dificultades en casos concretos. En primer lugar, mientras realizamos el análisis y la deconstrucción en BB y BC del programa, hemos de localizar el tipo de la variable observada en el **case** puesto que el tipo no aparece en la cláusula. Así, en caso de encontrarnos constructores en los patrones, podemos inferir fácilmente de qué tipo son sus variables y declararlas de manera que podamos utilizarlas dentro de la expresión de la rama escogida.

Por otro lado, se presenta también una sutileza cuando encontramos una rama por defecto. La restricción que se plantea en este caso es precisamente la conjunción de las negaciones de todos los patrones de las demás ramas o, equivalentemente, la negación de la disyunción de todos los patrones de las demás ramas.

Así, la traducción de la cláusula **case** puede resumirse en la Tabla 5.3.

Rama escogida	**Traducción a Z3**
C	$(assert(= \ a \ C))$
$C \ \overline{a_i}$	$\overline{(declare - const \ a_i \ \tau_i)}$ $(assert(= \ a \ (C \ \overline{a_i})))$
$\_$	$\overline{(declare - const \ a_i \ \tau_i)}$ $(assert(not(or(= \ a \ alt_1) \ldots (= \ a \ alt_m))))$

Tabla 5.3: Traducción de la expresión **case** a **of** $\overline{alt_i}[; \ \_ \rightarrow e]$ a Z3

5.1.4. Llamadas a funciones definidas en el programa

Dentro del camino que estamos considerando podemos encontrar también llamadas a funciones definidas en el propio programa, ya sea recursivas finales, recursivas no finales o no recursivas. En todos los casos el procedimiento que hemos seguido es el mismo.

En primer lugar, declaramos los parámetros de la función destino con sus correspondientes tipos. Tras ello asertamos la igualdad de cada parámetro con la variable o valor que le corresponda en la llamada. Finalmente, dado que la llamada se realiza en una cláusula **let** o como última expresión de una función, deberemos declarar la variable que especifica la función como variable de retorno e igualarla en función del caso a las variables que queremos ligar con el **let** o a la variable de retorno de la función origen.

Así, podríamos resumir la traducción de una llamada a la función (suponiendo las variables de la cláusula **let** o en su defecto las variables de retorno de la función origen ya declaradas) con las reglas de la Tabla 5.4.

Llamada	**Traducción a Z3**
let $\overline{z_i :: \tau_i} = f \ \overline{a_i}$ **in** e	$\overline{(declare - const \ x_i \ \tau_i)}$ $\overline{(assert(= \ x_i \ a_i))}$ $\overline{(declare - const \ y_i \ \tau_i)}$ $\overline{(assert(= \ z_i \ y_i))}$
$f \ (\overline{a_i})$ es la última expresión de la función g	$\overline{(declare - const \ x_i \ \tau_i)}$ $\overline{(assert(= \ x_i \ a_i))}$ $\overline{(declare - const \ y_i \ \tau_i)}$ $\overline{(assert(= \ v_j \ y_i))}$

Tabla 5.4: Traducción de la llamada a la función $f \ (\overline{x_i :: \tau_i}) :: \overline{y_i :: \tau_i}$ desde la función $g \ (\overline{u_j :: \tau_j}) :: \overline{v_j :: \tau_j}$ a Z3

5.1.5. Retornos

Por último, cuando un camino llega a su fin dentro de una función nos encontramos con una expresión que hemos de resolver para obtener el valor de

retorno de una función. Dado que este valor es devuelto mediante el parámetro de retorno de la función y este ya se encuentra declarado desde la llamada a la misma, la traducción a Z3 es simplemente la igualación del valor de retorno a la variable de retorno. Así dada la función $f\ (\overline{x_i :: \tau_i}) :: \overline{y_i :: \tau_i}$, la traducción a Z3 de la expresión de retorno sería la siguiente:

$$(f(\ldots)((y\ \tau))(\ldots a)) \Rightarrow (assert(=\ a\ y))$$
$$(f(\ldots)(\overline{(y_i\ \tau_i)})(\ldots tuple\ \overline{a_i})) \Rightarrow \overline{(assert(=\ a_i\ y_i))}$$

5.2. Obtención de las restricciones de un camino

Supongamos ahora conocidas las tablas de traducción de las expresiones de la CLIR al lenguaje de Z3, podemos aventurarnos a utilizarlas en los caminos que hemos obtenido previamente. Así, dado un camino que representa una secuencia de bloques y dada la lista de prefijos generados para que todo sean variables frescas, podemos traducir dicho camino al lenguaje de Z3.

La traducción es un proceso bastante directo por lo que nos guiaremos con un ejemplo para la explicación de la misma. Enumeraremos el proceso para su mejor seguimiento:

1. Tomemos entonces el camino en *insertArray.clir* que discurre por los nodos $[0, 1, 7, 3, 8, 4, 1, 7, 3, 8, 5, 6, 14, 16, 15, 9]$.
2. Generaremos los prefijos del camino para el renombramiento de variables obteniendo la lista [*“”, “f2_0_”, “f2_1_”, “f4_0_”, “f2_1_”, “f2_0_”, “”*] mediante el proceso descrito en la Sección 4.2.
3. Utilizando los prefijos, renombramos las variables de cada bloque para obtener variables frescas en todos ellos.
4. Traducimos ahora la secuencia de bloques renombrados a expresiones Z3.

El resultado de la traducción puede consultarse en la Tabla 5.5. Nótese que a fin de permitir una mejor comprensión del proceso, hemos obviado los nodos de retorno puesto que no representan traducción alguna, y tampoco se han explicitado las declaraciones de los parámetros de la función externa. Consideramos estos parámetros otorgados por la herramienta Precd2Z3, detalles que explicaremos en la siguiente sección.

5.3. Combinación de las restricciones con Precd2Z3

La herramienta Precd2Z3 [5] genera un archivo *.smt* con el objetivo de ejecutar manualmente después Z3 dándole el archivo generado. Así, se obtienen

Bloque	Prefijo	Código CLIR	Código Z3
0	“”	(let ((i Int)) (@ - m (the Int 1)) (@ f2 x m i a)))	(declare-const res (Arr Int)) (declare-const i Int) (assert (= i (- m 1))) (declare-const f2_0_x Int) (declare-const f2_0_m Int) (declare-const f2_0_i Int) (declare-const f2_0_a (Arr Int)) (assert (= x f2_0_x)) (assert (= i f2_0_i)) (assert (= m f2_0_m)) (assert (= a f2_0_a)) (declare-const f2_0_res2 (Arr Int)) (assert (= res f2_0_res2))
1	“f2_0_”	(let ((b1 Bool)) (@ >= i (the Int 0)) (BC 7))	(declare-const f2_0_b1 Bool) (assert (= f2_0_b1 (>= f2_0_i 0)))
7	“f2_0_”	(case b1 (((@@ false) (BB 2)) ((@@ true) (BB 3))))	(assert (= f2_0_b1 true))
3	“f2_0_”	(let ((e Int)) (@ get-array a i) (let ((b2 Bool)) (@ < x e) (BC 8)))	(declare-const f2_0_e Int) (assert (= f2_0_e (select (first f2_0_a) f2_0_i))) (declare-const f2_0_b2 Bool) (assert (= f2_0_b2 (< f2_0_x f2_0_e)))
8	“f2_0_”	(case b2 (((@@ true) (BB 4)) ((@@ false) (BB 5))))	(assert (= f2_0_b2 true))
4	“f2_0_”	(let ((u Int)) (@ get-array a i) (let ((i2 Int)) (@ + i (the Int 1)) (let ((ap (Array Int))) (@ set-array a i2 u) (let ((i3 Int)) (@ - i (the Int 1)) (@ f2 x m i3 ap)))))	(declare-const f2_0_u Int) (assert (= f2_0_u (select (first f2_0_a) f2_0_i))) (declare-const f2_0_i2 Int) (assert (= f2_0_i2 (+ f2_0_i 1))) (declare-const f2_0_ap (Arr Int)) (assert (= f2_0_ap (mk-pair (store (first f2_0_a) f2_0_i2 f2_0_u) (second f2_0_i2)))) (declare-const f2_0_i3 Int) (assert (= f2_0_i3 (- f2_0_i 1))) (declare-const f2_1_x Int) (declare-const f2_1_m Int) (declare-const f2_1_i Int) (declare-const f2_1_a (Arr Int)) (assert (= f2_0_x f2_1_x)) (assert (= f2_0_m f2_1_m)) (assert (= f2_0_i3 f2_1_i)) (assert (= f2_0_ap f2_1_a)) (declare-const f2_1_res2 (Arr Int)) (assert (= f2_1_res2 f2_0_res2))
1	“f2_1_”	(let ((b1 Bool)) (@ >= i (the Int 0)) (BC 7))	(declare-const f2_1_b1 Bool) (assert (= f2_1_b1 (>= f2_1_i 0)))
7	“f2_1_”	(case b1 (((@@ false) (BB 2)) ((@@ true) (BB 3))))	(assert (= f2_1_b1 true))
3	“f2_1_”	(let ((e Int)) (@ get-array a i) (let ((b2 Bool)) (@ ¡x e) (BC 8)))	(declare-const f2_1_e Int) (assert (= f2_1_e (select (first f2_1_a) f2_1_i))) (declare-const f2_1_b2 Bool) (assert (= f2_1_b2 (< f2_1_x f2_1_e)))
8	“f2_1_”	(case b2 (((@@ true) (BB 4)) ((@@ false) (BB 5))))	(assert (= f2_1_b2 false))
5	“f2_1_”	(@ f4 x m i a)	(declare-const f4_0_x Int) (declare-const f4_0_m Int) (declare-const f4_0_i Int) (declare-const f4_0_a (Arr Int)) (assert (= f2_1_x f4_0_x)) (assert (= f2_1_m f4_0_m)) (assert (= f2_1_i f4_0_i)) (assert (= f2_1_a f4_0_a)) (declare-const f4_0_res4 (Arr Int)) (assert (= f2_1_res2 f4_0_res4))
6	“f4_0_”	(let ((i2 Int)) (@ + i (the Int 1)) (let ((ap (Array Int))) (@ set-array a i2 x) (S ap))))	(declare-const f4_0_i2 Int) (assert (= f4_0_i2 (+ f4_0_i 1))) (declare-const f4_0_ap (Arr Int)) (assert (= f4_0_ap (mk-pair (store (first f4_0_a) f4_0_i2 f4_0_x) (second f4_0_a)))) (assert (= f4_0_ap f4_0_res4))

Tabla 5.5: Traducción del camino de ejemplo

modelos, es decir casos de prueba, que cumplen la precondición o en caso de ser insatisfactible, la confirmación de que así es.

El archivo generado por Precd2Z3 sigue una estructura definida separada en varias secciones:

1. *Declaraciones de los tipos de datos admitidos*: en concreto cuenta con la definición de los TADs lista, árbol binario, árbol AVL, LLRB, array, conjunto y multiconjunto.
2. *Declaraciones de las funciones asociadas a los tipos de datos*: entre las que podemos encontrar por ejemplo la altura de un árbol o el predicado que especifica que una lista esté ordenada.
3. *Declaración de las variables de entrada del programa.*
4. *Restricciones sobre las variables de entrada*: trabajaremos con plantillas con un tamaño máximo predeterminado. Así, las listas que generaremos serán como mucho de longitud 6, los arrays tendrán también a lo sumo longitud 6 y todos los tipos de árboles tendrán a lo sumo altura 6. En este apartado se instancian todas y cada una de las variables de la estructura de datos y se generan guías para Z3 de manera que sea capaz de poblar la estructura.
5. *Precondición*: se incluye una traducción de las restricciones de la precondición al lenguaje de Z3.

Reutilizando gran parte de la estructura del archivo de salida de Precd2Z3, generaremos nuestro propio archivo combinando las restricciones de la precondición con las restricciones de los caminos. El objetivo es generar casos que a la vez satisfagan la precondición y ejecuten un cierto camino. Nuestro archivo constará entonces de los siguientes apartados:

1. *Declaraciones de los tipos de datos admitidos.*
2. *Declaraciones de las funciones asociadas a los tipos de datos.*
3. *Declaración de las variables de entrada del programa.*
4. *Restricciones sobre las variables de entrada.*
5. *Push*: para inicializar la pila de Z3.
6. Para cada camino:
   - 6.1. *Restricciones del camino*: generadas mediante el proceso especificado en 5.2.
   - 6.2. *Precondición.*
   - 6.3. *Check-sat*: para comprobar la satisfactibilidad del camino.

6.4. *Get-model*: en caso de ser satisfactible, imprime un modelo que hace ciertas las restricciones.

6.5. *Pop*: para desapilar el camino actual.

6.6. *Push*: para reinicializar la pila de Z3.

De este modo, Z3 apila en un primer momento las declaraciones de los tipos de datos y sus funciones asociadas, las variables de entrada del programa y sus restricciones puesto que son instrucciones comunes a todos los caminos. Tras ello, se va presentando secuencialmente cada camino tras lo que se solicita su resolución y modelo. Una vez resuelto se desapila y se repite el proceso con el camino siguiente.

Obtenemos así una secuencia de ejecuciones en las que en cada una toma relevancia un único camino y las restricciones sobre la entrada.

Nótese que las restricciones dadas por la precondición se encuentran justo antes de cada comprobación de la satisfactibilidad. El motivo de este emplazamiento es el no-determinismo de la ejecución de Z3 puesto que cuando la precondición es un predicado de cierta complejidad, si este se introduce en la pila antes que el camino, Z3 no finaliza la ejecución, mientras que si se introduce como última restricción, Z3 resuelve el caso instantáneamente. Desconocemos el porqué de este no-determinismo aunque creemos que está relacionado con la síntesis de variables intermedias que en un caso las sintetiza sobre valores ya instanciados y en otro caso las sintetiza antes y posteriormente encuentra problemas para cuadrarlas con los valores que se van introduciendo.

Capítulo 6

Experimentos

Vamos ahora a realizar experimentos para intentar comprobar las hipótesis en las que nos aventuramos al comenzar esta investigación. Así, vamos a tratar de dar respuesta a las siguientes preguntas:

1. ¿Es posible generar un grafo utilizando el proceso descrito en el Capítulo 3?
2. ¿Es posible generar caminos en el grafo utilizando el proceso descrito en el Capítulo 4?
3. ¿Se cubren potencialmente todos los caminos del programa?
4. ¿Es posible transformar esos caminos a restricciones en el lenguaje Z3?
5. ¿Queda la estructura de los caminos completamente descrita mediante las restricciones generadas?
6. ¿Son resolubles por Z3 los caminos generados?
7. ¿Los modelos generados por Z3 son correctos?
8. ¿Se realiza el proceso en un tiempo razonable?

Debido a que queremos cubrir varias estructuras de datos, programas mixtos, recursión final y no final y otras peculiaridades, hemos reunido una serie de programas CLIR que son una buena representación estos aspectos. Así, vamos a utilizar los programas reflejados en la Tabla 6.1. Sus nombres expresan la función que representan. El código de cada uno de ellos puede ser consultado en A.

Realizaremos el seguimiento completo de uno de ellos, en concreto de *insertList.clir*, comentando el proceso que vamos realizando y en cada pregunta responderemos con los resultados de todos ellos, remarcando si en alguno ha habido algún detalle especial o si la experimentación va según lo previsto.

TAD	Programa	Recursión
Arrays	insertArray	Final
	quicksortMod	Mixto
	binSearch	Final
	linearSearch	Final
	dutchNationalFlag	Final
List	insertList	No final
	deleteList	No final
BST	insertBST	No final
	searchBST	Final
AVL	insertAVL	No final
	searchAVL	Final
Montículos	unionLeftist	No final
	insertLeftist	No final
LLRB	searchLLRB	Final

Tabla 6.1: Programas CLIR utilizados en la fase de experimentos.

Generación de un grafo

En primer lugar, a partir del el código fuente realizamos la separación en bloques básicos y bloques de salto generando en el ejemplo que estamos trabajando los bloques que aparecen en la Tabla 6.2.

Una vez generada la lista de bloques, podemos con una inspección rápida ver que no todas las llamadas son llamadas finales y por tanto descartamos la generación de un CFG. Nos aventuramos entonces a generar un grafo modular considerando los nodos de llamadas y los nodos de retorno como hemos comentado a lo largo del texto. Así, generamos el grafo modular de la Figura 6.1.

Podemos ahora dar respuesta a la primera pregunta de nuestra investigación de manera afirmativa. Hemos sido capaces de generar una estructura que refleja fielmente el flujo de control del programa. Para el resto de programas, la generación de los grafos ha sido exitosa en todos los casos generando, dependiendo de la naturaleza recursiva del programa, CFGs o grafos modulares y mixtos.

Generación de caminos

Una vez que contamos con el grafo del programa, del tipo que sea, podemos pasar a realizar un análisis para obtener los nodos de control del mismo. Dependiendo del tipo de grafo realizaremos el análisis de las SCCs (descrito en la Sección 3.3.1) en el caso de los CFG o el análisis de los programas recursivos no finales (descrito en la Sección 3.3.2) si no contamos con el CFG.

En el caso del ejemplo que estamos utilizando, podemos buscar el primer nodo de la rama que genera los despliegues de las llamadas recursivas. En este caso

Número	Nombre	Función	Tipo	Código
0	insertList_0	insertList	BB	(@ f1 x l)
1	f1_0	f1	BB	(BC f1_1)
2	f1_1_1_0		BB	(let ((vacia (Lst Int)) (@@ nil) (@@ cons y vacia))
3	f1_1_2_0		BB	(let ((b Bool)) (@ <= y z) (BC f1_1_2_1))
4	f1_1_2_1_1_0			(@@ cons y ys)
5	f1_1_2_1_2_0		BB	(let ((zz (Lst Int))) (@ f1 y zs) (BB f1_1_2_1_2_1))
6	f1_1_2_1_2_1		BB	(@@ cons z zz)
7	f1_1		BC	(case ys (((@@ nil) (BB f1_1_1_0)) ((@@ cons z zs) (BB f1_1_2_0))))
8	f1_1_2_1		BC	(case b (((@@ true) (BB f1_1_2_1_1_0)) (default (BB f1_1_2_1_2_0))))

Tabla 6.2: Bloques de insertList.clir

es trivial reconocer que la única llamada recursiva que existe es la presente en el nodo 11 y que el nodo que comienza la rama es el nodo 5. Así, podemos marcar el nodo 5 como nodo de control de este programa.

Podemos ahora utilizar la estrategia descrita en la Sección 4.1 para generar los caminos hasta un nivel de profundidad pedido. Incrementando el nivel de profundidad podemos obtener la respuesta para la tercera pregunta de nuestra investigación. Así, para un nivel de profundidad 0 encontramos 2 posibles caminos ([0,1,7,2,12,15] y [0,1,7,3,8,4,13,15]), para un nivel de profundidad 1 encontramos 4 posibles caminos (es decir, 2 nuevos caminos que son [0,1,7,3,8,5,1,7,2,12,6,14,15] y [0,1,7,3,8,5,1,7,3,8,4,13,6,14,15]), para un nivel de profundidad 2 encontramos 6 posibles caminos (es decir, 2 nuevos que son [0,1,7,3,8,5,1,7,3,8,5,1,7,2,12,6,14,6,14,15] y [0,1,7,3,8,5,1,7,3,8,5,1,7,3,8,4,13,6,14,6,14,15]) y así sucesivamente. Es claro ver que el valor del nivel de profundidad pedido se corresponde con las veces que aparece el nodo de control 5 en el camino. Nótese que cada ejecución con un nivel de profundidad devuelve también los caminos con un nivel de profundidad menor.

En este momento nos encontramos en posición de poder dar respuesta a la segunda pregunta de nuestra investigación de manera afirmativa. Utilizando el grafo del programa, somos capaces de extraer caminos del mismo. Así mismo, para el resto de programas podemos generar caminos a partir de sus grafos. Nos encontramos también con capacidad para responder a la tercera pregunta haciendo una

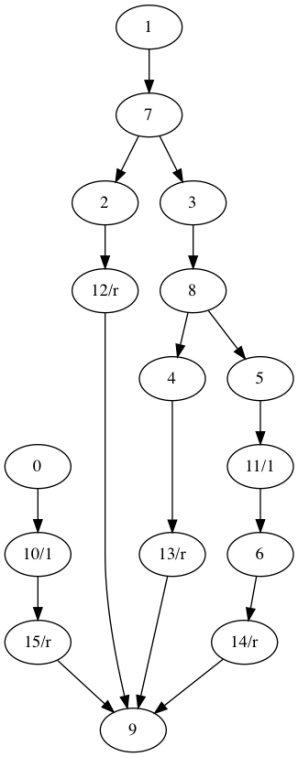

Figura 6.1: Grafo modular de insertList.clir

inferencia a partir del proceso de generación y con los datos de la Tabla 6.3. Podemos afirmar que todos los caminos son potencialmente recorridos pues comenzamos desde los casos base y vamos incrementando la complejidad de los caminos según vamos aumentando el nivel de recursión.

Cabe destacar el caso de la inserción en un AVL puesto que el número de caminos generados crece de manera explosiva según aumentamos en nivel de recursión. Examinando manualmente los caminos generados, aparentemente se debe a la suma complejidad de sus funciones intermedias que realizan un proceso recursivo que podíamos decir está *maquillado* o *enmascarado* y a pesar de estar recorriendo un bucle nuestro sistema no lo trata como tal sino que considera cada vuelta como un camino distinto. Es sin embargo interesante hacer notar que esto no se debe a una errónea identificación de los puntos de control puesto que si existiera algún bucle real que no hubieramos marcado el proceso entraría en una búsqueda infinita al poder generar caminos sin verse detenido por el nivel de recursión y por tanto no llegaríamos a obtener ninguna salida del programa.

Traducción de los caminos a Z3

Una vez que contamos con los caminos generados en el grafo, podemos pasar a transformarlos a ristras de restricciones en el lenguaje SMT-LIB para poder después procesarlas con Z3. En el caso del ejemplo que estamos trabajando con-

	0	**1**	**2**	**3**	**4**
insertArray	2	4	6	8	10
quicksortMod	1	2	7	33	173
binSearch	1	3	7	15	31
linearSearch	2	4	6	8	10
dutchFlag	1	4	13	40	121
deleteList	3	6	9	12	15
insertBST	2	6	14	30	62
searchBST	2	6	14	30	62
insertAVL	2	2	66	10306	12354
searchAVL	2	6	14	30	62
unionLeftist	2	10	58	354	2194
insertLeftist	2	10	58	354	2194
searchLLRB	2	10	14	30	62

Tabla 6.3: Número de caminos generados en cada programa en función del nivel de recursión solicitado

sideraremos los caminos hasta profundidad 3. Utilizaremos las tablas definidas en la Sección 5.1 para dirigir la traducción. Así, podemos ver para cada camino las siguientes ristras de restricciones:

Camino 1: [0,1,7,2,12,15]

```
(declare-const res (Lst Int))
(declare-const f1_1_y Int)
(declare-const f1_1_ys (Lst Int))
(assert (= x f1_1_y))
(assert (= l f1_1_ys))
(declare-const f1_1_ls (Lst Int))
(assert (= res f1_1_ls))
(assert (= f1_1_ys nil))
(declare-const f1_1_vacia (Lst Int))
(assert (= f1_1_vacia nil))
(assert (= f1_1_ls (cons f1_1_y f1_1_vacia)))
(assert (sortedList l))
(check-sat)
(get-model)
(pop)
(push)
```

Camino 2: [0,1,7,3,8,4,13,15]

```
(declare-const res (Lst Int))
(declare-const f1_1_y Int)
(declare-const f1_1_ys (Lst Int))
(assert (= x f1_1_y))
(assert (= l f1_1_ys))
(declare-const f1_1_ls (Lst Int))
(assert (= res f1_1_ls))
(declare-const f1_1_z Int)
(declare-const f1_1_zs (Lst Int))
(assert (= f1_1_ys (cons f1_1_z f1_1_zs)))
(declare-const f1_1_b Bool)
(assert (= f1_1_b (<= f1_1_y f1_1_z)))
(assert (= f1_1_b true))
(assert (= f1_1_ls (cons f1_1_y f1_1_ys)))
(assert (sortedList l))
(check-sat)
(get-model)
(pop)
(push)
```

Camino 3: [0,1,7,3,8,5,1,7,2,12,6,14,15]

```
(declare-const res (Lst Int))
(declare-const f1_1_y Int)
(declare-const f1_1_ys (Lst Int))
(assert (= x f1_1_y))
(assert (= l f1_1_ys))
(declare-const f1_1_ls (Lst Int))
(assert (= res f1_1_ls))
(declare-const f1_1_z Int)
(declare-const f1_1_zs (Lst Int))
(assert (= f1_1_ys (cons f1_1_z f1_1_zs)))
(declare-const f1_1_b Bool)
(assert (= f1_1_b (<= f1_1_y f1_1_z)))
(assert (not (or (= f1_1_b true))))
(declare-const f1_2_y Int)
(declare-const f1_2_ys (Lst Int))
(assert (= f1_1_y f1_2_y))
(assert (= f1_1_zs f1_2_ys))
(declare-const f1_2_ls (Lst Int))
(declare-const f1_1_zz (Lst Int))
(assert (= f1_1_zz f1_2_ls))
(assert (= f1_2_ys nil))
(declare-const f1_2_vacia (Lst Int))
```

```
(assert (= f1_2_vacia nil))
(assert (= f1_2_ls (cons f1_2_y f1_2_vacia)))
(assert (= f1_1_ls (cons f1_1_z f1_1_zz)))
(assert (sortedList l))
(check-sat)
(get-model)
(pop)
(push)
```

Camino 4: [0,1,7,3,8,5,1,7,3,8,4,13,6,14,15]

```
(declare-const res (Lst Int))
(declare-const f1_1_y Int)
(declare-const f1_1_ys (Lst Int))
(assert (= x f1_1_y))
(assert (= l f1_1_ys))
(declare-const f1_1_ls (Lst Int))
(assert (= res f1_1_ls))
(declare-const f1_1_z Int)
(declare-const f1_1_zs (Lst Int))
(assert (= f1_1_ys (cons f1_1_z f1_1_zs)))
(declare-const f1_1_b Bool)
(assert (= f1_1_b (<= f1_1_y f1_1_z)))
(assert (not (or (= f1_1_b true))))
(declare-const f1_2_y Int)
(declare-const f1_2_ys (Lst Int))
(assert (= f1_1_y f1_2_y))
(assert (= f1_1_zs f1_2_ys))
(declare-const f1_2_ls (Lst Int))
(declare-const f1_1_zz (Lst Int))
(assert (= f1_1_zz f1_2_ls))
(declare-const f1_2_z Int)
(declare-const f1_2_zs (Lst Int))
(assert (= f1_2_ys (cons f1_2_z f1_2_zs)))
(declare-const f1_2_b Bool)
(assert (= f1_2_b (<= f1_2_y f1_2_z)))
(assert (= f1_2_b true))
(assert (= f1_2_ls (cons f1_2_y f1_2_ys)))
(assert (= f1_1_ls (cons f1_1_z f1_1_zz)))
(assert (sortedList l))
(check-sat)
(get-model)
(pop)
(push)
```

Camino 5: [0,1,7,3,8,5,1,7,3,8,5,1,7,2,12,6,14,6,14,15]

```
(declare-const res (Lst Int))
(declare-const f1_1_y Int)
(declare-const f1_1_ys (Lst Int))
(assert (= x f1_1_y))
(assert (= l f1_1_ys))
(declare-const f1_1_ls (Lst Int))
(assert (= res f1_1_ls))
(declare-const f1_1_z Int)
(declare-const f1_1_zs (Lst Int))
(assert (= f1_1_ys (cons f1_1_z f1_1_zs)))
(declare-const f1_1_b Bool)
(assert (= f1_1_b (<= f1_1_y f1_1_z)))
(assert (not (or (= f1_1_b true))))
(declare-const f1_2_y Int)
(declare-const f1_2_ys (Lst Int))
(assert (= f1_1_y f1_2_y))
(assert (= f1_1_zs f1_2_ys))
(declare-const f1_2_ls (Lst Int))
(declare-const f1_1_zz (Lst Int))
(assert (= f1_1_zz f1_2_ls))
(declare-const f1_2_z Int)
(declare-const f1_2_zs (Lst Int))
(assert (= f1_2_ys (cons f1_2_z f1_2_zs)))
(declare-const f1_2_b Bool)
(assert (= f1_2_b (<= f1_2_y f1_2_z)))
(assert (not (or (= f1_2_b true))))
(declare-const f1_3_y Int)
(declare-const f1_3_ys (Lst Int))
(assert (= f1_2_y f1_3_y))
(assert (= f1_2_zs f1_3_ys))
(declare-const f1_3_ls (Lst Int))
(declare-const f1_2_zz (Lst Int))
(assert (= f1_2_zz f1_3_ls))
(assert (= f1_3_ys nil))
(declare-const f1_3_vacia (Lst Int))
(assert (= f1_3_vacia nil))
(assert (= f1_3_ls (cons f1_3_y f1_3_vacia)))
(assert (= f1_2_ls (cons f1_2_z f1_2_zz)))
(assert (= f1_1_ls (cons f1_1_z f1_1_zz)))
(assert (sortedList l))
(check-sat)
(get-model)
(pop)
(push)
```

Camino 6: [0,1,7,3,8,5,1,7,3,8,5,1,7,3,8,4,13,6,14,6,14,15]

```
(declare-const res (Lst Int))
(declare-const f1_1_y Int)
(declare-const f1_1_ys (Lst Int))
(assert (= x f1_1_y))
(assert (= l f1_1_ys))
(declare-const f1_1_ls (Lst Int))
(assert (= res f1_1_ls))
(declare-const f1_1_z Int)
(declare-const f1_1_zs (Lst Int))
(assert (= f1_1_ys (cons f1_1_z f1_1_zs)))
(declare-const f1_1_b Bool)
(assert (= f1_1_b (<= f1_1_y f1_1_z)))
(assert (not (or (= f1_1_b true))))
(declare-const f1_2_y Int)
(declare-const f1_2_ys (Lst Int))
(assert (= f1_1_y f1_2_y))
(assert (= f1_1_zs f1_2_ys))
(declare-const f1_2_ls (Lst Int))
(declare-const f1_1_zz (Lst Int))
(assert (= f1_1_zz f1_2_ls))
(declare-const f1_2_z Int)
(declare-const f1_2_zs (Lst Int))
(assert (= f1_2_ys (cons f1_2_z f1_2_zs)))
(declare-const f1_2_b Bool)
(assert (= f1_2_b (<= f1_2_y f1_2_z)))
(assert (not (or (= f1_2_b true))))
(declare-const f1_3_y Int)
(declare-const f1_3_ys (Lst Int))
(assert (= f1_2_y f1_3_y))
(assert (= f1_2_zs f1_3_ys))
(declare-const f1_3_ls (Lst Int))
(declare-const f1_2_zz (Lst Int))
(assert (= f1_2_zz f1_3_ls))
(declare-const f1_3_z Int)
(declare-const f1_3_zs (Lst Int))
(assert (= f1_3_ys (cons f1_3_z f1_3_zs)))
(declare-const f1_3_b Bool)
(assert (= f1_3_b (<= f1_3_y f1_3_z)))
(assert (= f1_3_b true))
(assert (= f1_3_ls (cons f1_3_y f1_3_ys)))
(assert (= f1_2_ls (cons f1_2_z f1_2_zz)))
(assert (= f1_1_ls (cons f1_1_z f1_1_zz)))
(assert (sortedList l))
(check-sat)
```

```
(get-model)
(pop)
(push)
```

Podemos ver en todos ellos como se comienza declarando el parámetro de retorno de la función y después se va recorriendo la lista de nodos, consultando sus bloques asociados y traduciendo las expresiones al lenguaje de Z3. Podemos ver también el correcto renombramiento de las variables en cada llamada a la función y la consistencia en la manera de utilizar las variables dentro de una misma llamada. Así mismo, podemos observar cómo todos los caminos terminan con la precondición *(assert (sortedList l))* seguida de las instrucciones de comprobación de la satisfactibilidad, la obtención del modelo y la gestión de la pila de Z3.

Podemos ahora dar respuesta a la cuarta y quinta preguntas de nuestra investigación pues sí que es posible al traducción al lenguaje de Z3 y la estructura de los caminos queda completamente descrita mediante estas ristras de restricciones. En el caso de los demás programas, la traducción de los caminos es exitosa en todos los casos, lo que corrobora nuestras hipótesis. Aún así, como en toda ejecución simbólica, siempre habrá programas cuyas condiciones de bifurcación sean muy complejas y no puedan ser traducidas a restricciones solubles por un resolutor SMT.

Ejecución de Z3

En esta última fase del proceso, vamos a proceder a ejecutar Z3 con los archivos generados y estudiar las preguntas de investigación sexta y séptima. Considerando el caso que nos ocupa y manteniendo el nivel de profundidad en 2, podemos procesar el archivo generado en la fase anterior a través de Z3. Obtenemos así los siguientes valores para los parámetros de entrada de la función:

- Camino 1: $x = 1; l = nil$.
- Camino 2: $x = (-8365); l = (cons\ 0\ nil)$.
- Camino 3: $x = 1; l = (cons\ 0\ nil)$.
- Camino 4: $x = (-2); l = (cons\ (-3)\ (cons\ 0\ nil))$.
- Camino 5: $x = 0; l = (cons\ (-2)\ (cons\ (-1)\ nil))$.
- Camino 6: $x = 2; l = (cons\ 0\ (cons\ 1\ (cons\ 2\ nil)))$.

Se puede comprobar fácilmente con un seguimiento manual que estos valores efectivamente realizan 2 despliegues recursivos o menos y que son correctos. Además, se ve fácilmente que también las listas de entrada están ordenadas como solicita la precondición por lo que podemos dar estos casos como correctos y precisos, respondiendo en este caso a la pregunta 6 con un afirmación pues Z3 ha generado

	Sat	**Unsat**	**Unknown**
insertArray	8	0	0
quicksortMod	4	29	0
binSearch	13	2	0
linearSearch	8	0	0
dutchFlag	40	0	0
deleteList	40	0	0
insertAVL	3	56	7
searchAVL	2	0	28
insertBST	28	0	2
searchBST	24	0	6
unionLeftist	8	346	0
insertLeftist	6	348	0
searchLLRB	12	0	18

Tabla 6.4: Salidas de los distintos programas con un nivel de profundidad 3.

una salida y a la pregunta 7 también de manera afirmativa pues hemos comprobado que son modelos correctos.

Sin embargo, las respuestas que extraemos estudiando los demás ejemplos contradicen en parte lo que hemos afirmado hace un momento. En la Tabla 6.4 se pueden consultar las salidas de los distintos programas ejecutados todos con un nivel de profundidad 3 excepto *insertAVL* que ha usado un nivel de profundidad de 2 (puesto que el nivel de profundidad 3 generaba un archivo de 1987987 líneas). Así, se puede ver cómo abundan los resultados insatisfactibles y conforme vamos complicando la estructura que utilizamos aparecen los resultados desconocidos. Siendo tan bueno el desempeño en algunos casos y tan malo en otros, nos hace pensar que Z3 tiene problemas para sintetizar estructuras complejas. Al dar a Z3 el archivo de restricciones de, por ejemplo, *searchAVL*, menos en las dos salidas *sat*, en el resto se queda bloqueado hasta que abortamos la resolución y le forzamos a pasar al siguiente caso.

Además, estos datos no parecen ser consistentes pues en algunos casos aparecen más salidas *unknown* con los mismos datos de entrada. Creemos que esto puede deberse a que las fórmulas generadas son de gran complejidad y las heurísticas de Z3 no son suficientemente potentes. En definitiva, tropezamos aquí con la indecibilidad en general del problema de satisfactibilidad de la lógica de primer orden.

Sí es importante comentar que en todos los programas ha devuelto alguna salida satisfactible y el correspondiente modelo es correcto y se ajusta a las restricciones que hemos introducido. Además estas salidas satisfactibles aunque concentradas en los primeros casos (los caminos menos complejos), también aparecen de repente en casos bastante complejos por lo que no parece que estén condicionados por la cantidad de restricciones que conforman el camino.

Es interesante también notar como *insertLeftist* es un mero envoltorio sobre *unionLeftist* pero en las salidas aparece un mayor número de caminos insatisfactibles

	AST2Z3	**Z3**
insertArray	0.22	0.26
quicksortMod	0.4	0.22
binSearch	0.23	0.31
linearSearch	0.21	0.17
dutchFlag	0.35	0.94
insertList	0.23	0.18
deleteList	0.20	0.25
insertAVL	0.41 - 101.90	-
searchAVL	0.27	-
insertBST	0.25	-
searchBST	0.22	-
unionLeftist	1.34	0.69
insertLeftist	1.50	0.38
searchLLRB	0.26	-

Tabla 6.5: Tiempos de ejecución de los distintos programas.

lo que no hace sino apoyar nuestra teoría de la estrategia no determinista.

Tiempos de ejecución

Por último, vamos a realizar una reseña de los tiempos que toman las dos fases de esta investigación. Por un lado estudiaremos el tiempo que tarda el análisis del programa CLIR mediante nuestra implementación de la herramienta en Haskell y después consideraremos el tiempo que tarda Z3 en procesar los archivos generados.

En la Tabla 6.5 estan reflejados los tiempos, medidos en segundos, obtenidos en cada fase por cada uno de los programas. Nótese que en los programas que ofrecen salidas *unknown* se ha obviado el tiempo de Z3 por ser este dependiente de cuando corte el usuario el bloqueo en los casos que devuelven *unknown* y por tanto no es un tiempo real de procesamiento. En el caso de *insertAVL* se ha marcado también el tiempo que tarda AST2Z3 en generar el archivo en el caso de que solicitemos un nivel de profundidad de 3.

Como se puede comprobar los tiempos son bastante bajos en la herramienta AST2Z3 menos en el caso de *insertAVL* con profundidad 3 que el tiempo se dispara y en el caso de los montículos. Suponemos que el incremento de tiempo en los programas de los montículos se debe a que cada camino puede pasar por numerosas funciones de una complejidad no trivial, por lo que es normal que el programa tarde más en resolverlos.

También podemos remarcar que los tiempos de Z3 cuando no aparecen casos irresolubles son bastante adecuados procesando por ejemplo en los montículos 354 caminos (archivos de aproximadamente 50000 líneas de restricciones) en apenas 0.7 segundos encontrando la insatisfactibilidad de la mayoría de los casos. Es im-

portante notar que habitualmente es más costoso probar la insatisfactibilidad que la satisfactibilidad, pues la segunda requiere solo encontrar un caso que satisfaga las restricciones, mientras que la primera requiere comprobar que ninguno de todos los posibles casos la satisfacen.

Podemos entonces dar una respuesta a la última pregunta de investigación de manera afirmativa por parte de la herramienta AST2Z3, mientras que por la parte de Z3 la pregunta queda abierta a una mayor investigación sobre el no determinismo de esta herramienta que permita aclarar las causas de los caminos irresolubles y permita entonces hacer un análisis completo del tiempo que tarda Z3 en procesar los archivos.

Capítulo 7

Conclusiones

Castellano

Llegados a este punto, podemos dar por concluida nuestra investigación y afirmar que, en su mayoría, los objetivos que se marcaron al comienzo del trabajo han sido cumplidos.

En primer lugar, hemos estudiado la naturaleza de los programas que genera la CLIR, buscando una manera de representar su flujo, generando unas reglas consistentes y fundamentadas para la transformación de los programas, es decir, un *método*.

Por otro lado, una vez generadas las estructuras, hemos generado también unas reglas de análisis para generar los caminos evitando los bucles infinitos y pudiendo seguir un cierto orden (de los caminos más simples hacia los más complejos) que permite un buen análisis de los casos.

Finalmente, se ha elaborado una traducción de los caminos a Z3 y se ha conseguido en mayor o menor medida dependiendo de la complejidad del programa, la obtención de unos modelos que sirven como casos de prueba. Así pues, hemos conseguido generar de manera automática casos de prueba de tipo caja blanca, además en un tiempo razonable.

Tras esta investigación queda abierta ante todo una cuestión para investigaciones futuras: ¿Cómo se puede ayudar a Z3 a resolver los casos que actualmente no puede resolver?

Por último, de cara a futuro sería interesante la utilización de la API Z3-Haskell, pues esto permitiría interactuar con Z3 y variar los datos de entrada durante el programa en vez de tener que utilizar un archivo *.smt* intermedio y realizar una inspección manual del mismo.

Inglés

At this stage, we can conclude our research and affirm that, in the majority of cases, the goals that were set at the beginning of this research have been achieved.

In the first place, we have studied the nature of the programs generated by the CLIR, looking for a way of representing its flow, developing consistent and well-founded rules for transforming the programs, in fact, we have developed a *method*.

On the other hand, once the graph structures have been generated, we have also developed some analysis rules in order to generate the paths, avoiding infinite loops and achieving some kind of order (from the most basic to the complex paths) that allows us to do a good analysis of the cases.

Finally, we have developed a translation of the program paths into Z3 restrictions and depending on the program complexity, we have been able of generating models that can be used as test cases. That means that we have been able to generate white-box test cases automatically, and all this in a reasonable time.

After this research, there is one more question to research: How can we help Z3 in solving the cases it can't solve at the moment?

Finally, it would be interesting to use the Z3-Haskell API, because it would permit to interact with Z3 and change the parameters through the program instead of having to use a temporary *.smt* file and having to process it by hand.

Apéndice A

Programas CLIR

A lo largo de esta sección, se presentan los distintos códigos CLIR que se han utilizado para la fase de experimentos.

A.1. Arrays

A.1.1. insertArray.clir

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

(define insert ((x Int) (m Int) (a (Array Int))) ((res (Array Int)))
  (declare (assertion
    (precd
      (and (@ <= (the Int 0) m) (@ < m (@ len a)) (forall ((i Int) (j Int))
        (-> (@ <= (the Int 0) i)
          (@ <= i j)
          (@ < j m)
          (@ <= (@ get-array a i) (@ get-array a j))))))
    (postcd
      (forall ((i Int) (j Int))
        (-> (@ <= (the Int 0) i)
          (@ <= i j)
          (@ <= j m)
          (@ <= (@ get-array res i) (@ get-array res j)))))))

  (letfun (
    (f2 ((x Int) (m Int) (i Int) (a (Array Int))) ((res2 (Array Int)))
      (let ((b1 Bool)) (@ >= i (the Int 0))
        (case b1 (
          ((@@ false)
            (@ f4 x m i a))
          ((@@ true)
            (let ((e Int)) (@ get-array a i)
              (let ((b2 Bool)) (@ < x e)
                (case b2 (
                  ((@@ true)
                    (let ((u Int)) (@ get-array a i)
                      (let ((i2 Int)) (@ + i (the Int 1))
                        (let ((ap (Array Int))) (@ set-array a i2 u)
                          (let ((i3 Int)) (@ - i (the Int 1))
                            (@ f2 x m i3 ap))))))
                  ((@@ false)
                    (@ f4 x m i a)))))))))))

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

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

A.1.2. quicksortMod.clir

```
(verification-unit "QSORT"
           :sources "(((:lang :clir) (:module :self)))"
           :uses "(:ir)"
           :documentation "This is the qksort function on the CLIR.")

(define quick ((V (Array Int))) ((vres (Array Int)))
  (declare (assertion
    (precd
      (and (@ <= (the Int 0) i)
        (@ <= i (@ len V))
        (@ <= (the Int -1) j)
        (@ < j (@ len V))))
    (postcd
      (forall ((j1 Int) (j2 Int))
        (-> (@ <= i j1)
        (-> (@ <= j1 j2)
        (-> (@ <= j2 j)
          (@ <= (@ get-array res j1) (@ get-array res j2)))))))))

  (letfun (
    (partition ((a Int) (b Int) (V (Array Int))) ((res1 Int) (res2 (Array Int)))
      (let ((i Int)) (@ + a (the Int 1))
        (let ((j Int)) b
          (@ f2 a b i j V))))

    (f2 ((u Int) (v Int) (i Int) (j Int) (a (Array Int)))  ((res1 Int) (res2 (Array Int)))
      (let ((b2 Bool)) (@ <= i j)
        (case b2 (
          ((@@ true)
            (let ((ei Int)) (@ get-array a i)
              (let ((eu Int)) (@ get-array a u)
                (let ((ej Int)) (@ get-array a j)
                  (let ((b31 Bool)) (@ < eu ei)
                    (case b31 (
                      ((@@ false)
                        (let ((i1 Int)) (@ + i (the Int 1))
                          (@ f2 u v i1 j a)))
                      ((@@ true)
                        (let ((b32 Bool)) (@ < ej eu)
                          (case b32 (
                            ((@@ true)
                              (let ((a1 (Array Int))) (@ set-array a i ej)
                                (let ((a2 (Array Int))) (@ set-array a1 j ei)
                                  (let ((i1 Int)) (@ + i (the Int 1))
                                    (let ((j1 Int)) (@ - j (the Int 1))
                                      (@ f2 u v i1 j1 a2))))))
                            ((@@ false)
                              (let ((j1 Int)) (@ - j (the Int 1))
                                (@ f2 u v i j1 a))))))))))))))
          ((@@ false)
            (let ((eu Int)) (@ get-array a u)
              (let ((ej Int)) (@ get-array a j)
                (let ((a1 (Array Int))) (@ set-array a u ej)
                  (let ((a2 (Array Int))) (@ set-array a1 j eu)
                    (@ pair j a2)))))))))))

    (f1 ((i Int) (j Int) (V (Array Int))) ((res (Array Int)))
      (let ((p Int) (Vp (Array Int))) (@ partition i j V)
        (let ((p1 Int)) (@ - p (the Int 1))
          (let ((V1 (Array Int))) (@ qsort i p1 Vp)
            (let ((p2 Int)) (@ + p (the Int 1))
              (@ qsort p2 j V1)))))))

    (qsort ((i Int) (j Int) (V (Array Int))) ((res (Array Int)))
      (let ((b Bool)) (@ <= i j)
        (case b (
          ((@@ true)
            (@ f1 i j v))
          ((@@ false)
            V)))))
  )
  (let ((n Int)) (@ len V)
    (let ((n1 Int)) (@ - n (the Int 1))
      (@ qsort (the Int 0) n1 V)))))
```

A.1.3. binSearch.clir

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

(define binSearch ((x Int)  (v (Array Int))) ((p Int))
  (declare (assertion
    (precd
```

```
      (forall ((i Int) (j Int))
        (-> (@ <= (the Int 0) i)
        (-> (@ <= i j)
        (-> (@ < j (@ len v))
          (@ <= (@ get-array v i) (@ get-array v j)))))))
    (postcd
      (and (@ <= (the Int 0) p)
        (@ <= p (@ len res))
        (forall ((j Int))
          (-> (@ <= (the Int 0) j)
            (@ < j p)
            (@ < (@ get-array res j) x)) )
        (forall ((j Int))
          (-> (@ <= p j)
            (@ < j (@ len res))
            (@ <= x (@ get-array res j))) )
        (forall ((i Int) (j Int))
          (-> (@ <= (the Int 0) i)
          (-> (@ <= i j)
          (-> (@ < j (@ len res))
            (@ <= (@ get-array res i) (@ get-array res j))))))))))
  (letfun (
    (bin ((a Int) (b Int) (x Int) (v (Array Int))) ((p2 Int))
      (let ((b1 Bool)) (@ > a b)
        (case b1 (
          ((@@ true) a)
          ((@@ false)
            (let ((m Int)) (@ semisum a b)
              (let ((y Int)) (@ get-array v m)
                (let ((b2 Bool)) (@ > x y)
                  (case b2 (
                    ((@@ true)
                      (let ((m2 Int)) (@ + m (the Int 1))
                        (@ bin m2 b x v) ))
                    ((@@ false)
                      (let ((m3 Int)) (@ - m (the Int 1))
                        (@ bin a m3 x v))))))))))))
  )
  (let ((a Int)) (the Int 0)
    (let ((l Int)) (@ len v)
      (let ((b Int)) (@ - l (the Int 1))
        (@ bin a b x v))))))
```

A.1.4. linearSearch.clir

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

(define search ((e Int)(V (Array Int))) ((res1 Int))
  (declare (assertion
    (precd
      (@ <= (the Int 0) (@ len V)))
    (postcd
      (and (@ <= (the Int 0) res1)
        (@ <= res1 (@ len res2))
        (forall ((j Int))
          (-> (@ <= (the Int 0) j )
          (-> (@ < j res1)
          (-> (@ <> (@ get-array res2 j) e)))))
        (-> (@ < res1 (@ len res2))(@ = (@ get-array res2 res1) e))
        (@ = (@ len V) (@ len res2))
        (forall ((j Int))
          (-> (@ <= (the Int 0) j)
          (-> (@ < j (@ len res2))
          (-> (@ = (@ get-array V j) (@ get-array res2 j))))))))))

  (letfun (
    (f ((i Int) (e Int) (V (Array Int))) ((res1 Int))
      (let ((l Int)) (@ len V)
        (let ((b1 Bool)) (@ < i l)
          (case b1 (
            ((@@ false)
              i)
            ((@@ true)
              (let ((vi Int)) (@ get-array V i)
                (let ((b2 Bool)) (@ == vi e)
                  (case b2 (
                    ((@@ true)
                      i)
                    ((@@ false)
                      (let ((i1 Int)) (@ + i (the Int 1))
                        (@ f i1 e V))))))))))))
  )
  (let ((i Int)) (the Int 0)
    (@ f i e V))))
```

A.1.5. dutchNationalFlag.clir

```
(verification-unit "dutchNationalFlag"
          :sources "((:lang :unknown)  (:module :unknown)))"
          :uses "(:ir)"
          :documentation "Dutch National Flag algorithm")

(define DutchNationalFlag ((v (Array Int))) ((p Int) (q Int) (res (Array Int)))
  (declare (assertion
    (precd
      (forall ((j Int))
        (-> (@ <= (the Int 0) j)
          (@ <  j  (@ len v))
          (or (@ = (@ get-array v j) (the Int 0))
            (@ = (@ get-array v j) (the Int 1))
            (@ = (@ get-array v j) (the Int 2))))))
    (postcd
      (and (@ = (@ len res) (@ len v))
        (@ <= (the Int 0) p)
        (@ <= p q)
        (@ <= q (@ len res))
        (forall ((j Int)) (-> (@ <= (the Int 0) j)
          (@ < j p)
          (@ = (@ get-array res j) (the Int 0))))
        (forall ((j Int)) (-> (@ <= p j)
          (@ <  j q)
          (@ = (@ get-array res j) (the Int 1))))
        (forall ((j Int)) (-> (@ <= q j)
          (@ < j (@ len res))
          (@ = (@ get-array res j) (the Int 2))))))))

  (letfun (

    (flag ((a Int) (b Int) (c Int) (v (Array Int))) ((p2 Int) (q2 Int) (r2 (Array Int)))
      (let ((b0 Bool)) (@ <= b c)
        (case b0 (
          ((@@ true)
            (let ((x Int)) (@ get-array v b)
              (let ((b1 Bool)) (@ == x (the Int 0))
                (case b1 (
                  ((@@ true)
                    (let ((y Int)) (@ get-array v a)
                      (let ((v1 (Array Int))) (@ set-array v a x)
                        (let ((v2 (Array Int))) (@ set-array v1 b y)
                          (let ((a2 Int)) (@ + a (the Int 1))
                            (let ((b2 Int)) (@ + b (the Int 1))
                              (@ flag a2 b2 c v2)))))))
                  ((@@ false)
                    (let ((b2 Bool)) (@ == x (the Int 1))
                      (case b2 (
                        ((@@ true)
                          (let ((m2 Int)) (@ + b (the Int 1))
                            (@ flag a m2 c v) ))
                        ((@@ false)
                          (let ((y Int)) (@ get-array v c)
                            (let ((v1 (Array Int))) (@ set-array v c x)
                              (let ((v2 (Array Int))) (@ set-array v1 b y)
                                (let  ((c2 Int)) (@ - c (the Int 1))
                                  (@ flag a b c2 v2)))))))))))))
          ((@@ false)
            (@ triple a b v))))))

    )
    (let ((a Int)) (the Int 0)
      (let ((b Int)) (the Int 0)
        (let ((l Int)) (@ len v)
          (let ((c Int)) (@ - l (the Int 1))
            (@ flag a b c v)))))))
```

A.2. Listas

A.2.1. insertList.clir

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

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

```
      (and (@ sortedList res)
        (@ = (@ multiset res)
          (let ((m2 (Multiset Int))) (@ multiset l)
            (let ((m3 (Multiset Int))) (@ unitMs x)
              (@ Mset-union m2 m3)))))))

  (letfun (
    (f1 ((y Int) (ys (Lst Int))) ((ls (Lst Int)))
      (case ys (
        ((@@ nil)
          (let ((vacia (Lst Int))) (@@ nil)
            (@@ cons y vacia)))
        ((@@ cons z zs)
          (let ((b Bool)) (@ <= y z)
            (case b (
              ((@@ true)
                (@@ cons y ys))
              (default
                (let ((zz (Lst Int))) (@ f1 y zs)
                  (@@ cons z zz)))))))))

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

A.2.2. deleteList.clir

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

(define deleteList ((x Int) (l (Lst Int))) ((res (Lst Int)))
  (declare (assertion
    (precd
      (@ sortedList l))
    (postcd
      true)))

  (letfun (
    (delete ((x Int) (l (Lst Int))) ((res (Lst Int)))
      (case l (
        ((@@ nil) (@@ nil))
        ((@@ cons y ys)
          (let ((b1 Bool)) (@ < x y)
            (case b1 (
              ((@@ true)
                l)
              ((@@ false)
                (let ((b2 Bool)) (@ > x y)
                  (case b2 (
                    ((@@ true)
                      (let ((lp (Lst Int))) (@ deleteList x ys)
                        (@@ cons y lp)))
                    ((@@ false) ys)))))))))))

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

A.3. BST

A.3.1. insertBST.clir

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

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

  (letfun (
```

```
    (f1 ((x Int) (t (Tree Int))) ((res (Tree Int)))
          (case t (
                  ((@@ leaf)
          (let ((empty-leaf (Tree Int))) (@@ leaf)
            (@@ node x empty-leaf empty-leaf)))
        ((@@ node y l r)
          (let ((b Bool)) (@ < x y)
            (case b (
              ((@@ true)
                (let ((z (Tree Int))) (@ f1 x l)
                  (@@ node y z r)))
              ((@@ false)
                (let ((b1 Bool)) (@ > x y)
                  (case b1 (
                    ((@@ true)
                      (let ((z (Tree Int))) (@ f1 x r)
                        (@@ node y l z)))
                    ((@@ false)
                      t))))))))))))
  )
  (@ f1 x t)))
```

A.3.2. searchBST.clir

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

(define searchBST ((x Int) (t (Tree Int))) ((res Bool))
  (declare (assertion
    (precd
      (@ isBST t))
    (postcd
      true)))

  (letfun (
    (search ((x Int) (t (Tree Int))) ((res Bool))
      (case t (
        ((@@ leaf)
          (the Bool false))
        ((@@ node y l r)
          (let ((b1 Bool)) (@ < x y)
            (case b1 (
                ((@@ true) (@ search x l))
                ((@@ false) (let ((b2 Bool)) (@ > x y)
                    (case b2 (
                        ((@@ true) (@ search x r))
                        ((@@ false) (the Bool true))))))))))))
  )
  (@ search x t)))
```

A.4. AVL

A.4.1. insertAVL.clir

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

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

  (letfun (
    (f1 ((x Int) (t (AVL Int))) ((res (AVL Int)))
      (case t(
        ((@@ leafA)
          (@@ nodeA x (the Int 0) leafA leafA))
        ((@@ nodeA y h l r)
```

```
          (let ((b1 Bool)) (@ < x y)
            (case b1 (
              ((@@ true)
                (let ((ia (AVL Int))) (@ f1 x l)
                  (@ equil ia y r)))
              ((@@ false)
                (let ((b2 Bool)) (@ > x y)
                  (case b2 (
                    ((@@ true)
                      (let ((ia (AVL Int))) (@ f1 x r)
                        (@ equil l y ia)))
                    ((@@ false)
                      t)))))))))))

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

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

    (height ((t (AVL Int))) ((hres Int))
      (case t (
        ((@@ leafA)
          (the Int 0))
        ((@@ nodeA x h l r)
          h))))

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

    (rightBalance ((l (AVL Int)) (x Int) (r (AVL Int))) ((res (AVL Int)))
      (case r (
        ((@@ nodeA rx rh rl rr)
          (let ((rlh Int)) (@ height rl)
            (let ((rrh Int)) (@ height rr)
              (let ((b Bool)) (@ >= rrh rlh)
                (case b (
                  ((@@ true)
                    (let ((tx (AVL Int))) (@ compose l x rl)
                      (@ compose tx rx rr)))
                  ((@@ false)
                    (case rl (
                      ((@@ nodeA rlx rlh rll rlr)
                        (let ((cp1 (AVL Int))) (@ compose l x rll)
                          (let ((cp2 (AVL Int))) (@ compose rlr rx rr)
                            (@ compose cp1 rlx cp2)))))))))))))))
    )
    (@ f1 x t)))
```

A.4.2. searchAVL.clir

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

```
(define searchAVL ((x Int) (t (AVL Int))) ((res Bool))
  (declare (assertion
    (precd
      (@ isAVL t))
    (postcd
      true)))

  (letfun (
    (search ((x Int) (t (AVL Int))) ((res Bool))
      (case t (
        ((@@ leafA)
          (the Bool false))
        ((@@ nodeA y h l r)
          (let ((b1 Bool)) (@ < x y)
            (case b1 (
                ((@@ true) (@ search x l))
                ((@@ false) (let ((b2 Bool)) (@ > x y)
                    (case b2 (
                        ((@@ true) (@ search x r))
                        ((@@ false) (the Bool true))))))))))))
  )
  (@ search x t)))
```

A.5. Montículo

A.5.1. unionLeftist.clir

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

(define unionLeftist ((t1 (Tree Int)) (t2 (Tree Int))) ((res (Tree Int)))
  (declare (assertion
    (precd
      (and (@ isLeftist t1)
        (@ isHeap t1)
        (@ isLeftist t2)
        (@ isHeap t2)))
    (postcd
      (and (@ isLeftist res)
        (@ isHeap res)
        (@ = (@ multiset res)
          (let ((m1 (Multiset Int))) (@ multiset t1)
            (let ((m2 (Multiset Int))) (@ multiset t2)
              (@ Mset-union m1 m2)))))))))

  (letfun (
    (f1 ((t1 (Tree Int)) (t2 (Tree Int))) ((res (Tree Int)))
      (case t1 (
          ((@@ leaf)
        t2)
          ((@@ node x1 l1 r1)
        (case t2 (
                  ((@@ leaf)
            t1)
                      ((@@ node x2 l2 r2)
                        (let ((b Bool)) (@ <= x1 x2)
                          (case b (
                                  ((@@ true)
                  (let ((ul (Tree Int))) (@ f1 r1 t2)
                                  (@ equil x1 l1 ul)))
                                  ((@@ false)
                  (let ((ul (Tree Int))) (@ f1 t1 r2)
                                  (@ equil x2 l2 ul))))))))))))

    (equil ((x Int) (l (Tree Int)) (r (Tree Int))) ((res (Tree Int)))
          (let ((hl Int)) (@ hmin l)
            (let ((hr Int)) (@ hmin r)
                  (let ((b Bool)) (@ >= hl hr)
                    (case b (
                          ((@@ true)
                (@@ node x l r))
                          ((@@ false)
                (@@ node x r l)))))))))

    (hmin ((t (Tree Int))) ((h Int))
          (case t (
            ((@@ leaf)
          (the Int 0))
                ((@@ node x l r)
          (let ((hl Int)) (@ hmin l)
                (let ((hr Int)) (@ hmin r)
                          (let ((m Int)) (@ min hl hr)
```

```
                              (@ + (the Int 1) m))))))))
  ) (@ f1 t1 t2)))
```

A.5.2. insertLeftist.clir

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

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

  (letfun (

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

    (f1 ((t1 (Tree Int)) (t2 (Tree Int))) ((res (Tree Int)))
      (case t1 (
            ((@@ leaf)
          t2)
            ((@@ node x1 l1 r1)
          (case t2 (
                    ((@@ leaf)
              t1)
                        ((@@ node x2 l2 r2)
                          (let ((b Bool)) (@ <= x1 x2)
                            (case b (
                                    ((@@ true)
                          (let ((ul (Tree Int))) (@ f1 r1 t2)
                                      (@ equil x1 l1 ul)))
                                    ((@@ false)
                          (let ((ul (Tree Int))) (@ f1 t1 r2)
                                      (@ equil x2 l2 ul)))))))))))))

    (equil ((x Int) (l (Tree Int)) (r (Tree Int))) ((res (Tree Int)))
          (let ((hl Int)) (@ hmin l)
            (let ((hr Int)) (@ hmin r)
                  (let ((b Bool)) (@ >= hl hr)
                    (case b (
                            ((@@ true)
                (@@ node x l r))
                            ((@@ false)
                (@@ node x r l)))))))))

    (hmin ((t (Tree Int))) ((h Int))
          (case t (
            ((@@ leaf)
          (the Int 0))
                ((@@ node x l r)
          (let ((hl Int)) (@ hmin l)
                (let ((hr Int)) (@ hmin r)
                          (let ((m Int)) (@ min hl hr)
                            (@ + (the Int 1) m)))))))))

  ) (let ((h1 (Tree Int))) (@@ leaf)
      (let ((h2 (Tree Int))) (@@ leaf)
        (let ((h3 (Tree Int))) (@@ node x h1 h2)
          (@ unionLeftist h3 t))))))
```

A.6. LLRB

A.6.1. searchLLRB.clir

```
(verification-unit "searchLLRB"
                   :sources "(((:lang :clir) (:module :self)))"
```

```
                :uses "(:ir)"
                :documentation "Search on an LLRB")

(define searchLLRB ((x Int) (t (LLRB Int))) ((res Bool))
  (declare (assertion
    (precd
      (@ isLLRB t))
    (postcd
      true)))

  (letfun (
    (search ((x Int) (t (LLRB Int))) ((res Bool))
      (case t (
        ((@@ leafL)
          (the Bool false))
        ((@@ nodeL y c l r)
          (let ((b1 Bool)) (@ < x y)
            (case b1 (
                ((@@ true) (@ search x l))
                ((@@ false) (let ((b2 Bool)) (@ > x y)
                    (case b2 (
                        ((@@ true) (@ search x r))
                        ((@@ false) (the Bool true))))))))))))

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

Apéndice B

Notas sobre la implementación

A fin de dar una visión global de la investigación realizada, vemos necesario dar unas notas sobre la implementación que se ha realizado de la herramienta en el lenguaje Haskell. Un planteamiento general del funcionamiento de la misma puede verse en la Figura B.1.

Así, la herramienta tiene como entradas el nivel de recursión que el usuario escoja, el árbol de sintaxis abstracta que genera la herramienta IR2Haskell, las restricciones que genera la herramienta Precd2Z3 y los ficheros con los tipos abstractos de datos y sus funciones asociadas. De este modo, termina generando un archivo .SMT que será procesado por Z3 para la obtención de los modelos. Cada modelo representa un caso de prueba tipo caja blanca que ejecuta un camino de la función a probar. Comentaremos ahora brevemente el significado de cada una de las fases por las que pasa nuestra implementación de la herramienta.

Fase 1

En la primera fase, se realiza el proceso de deconstrucción del AST para dividirlo en las unidades mencionadas en la Sección 3.1. De esta primera fase obtenemos los bloques básicos y los bloques de casos que conforman el programa.

Fase 2

Tomando como entrada los bloques básicos y de casos del programa, en esta fase se genera si es posible el CFG plano del programa en caso de que sea recursivo final. Tras ello se genera el grafo modular y se realiza un análisis de los grafos que se hayan generado a fin de encontrar los nodos de control. Si se ha generado el CFG se realizará el análisis de las SCCs descrito en la Sección 3.3.1. En caso contrario, se realizará el análisis de los programas recursivos no finales descrito en la Sección 3.3.2 seguido del análisis de los programas mixtos descrito en la Sección 3.3.3.

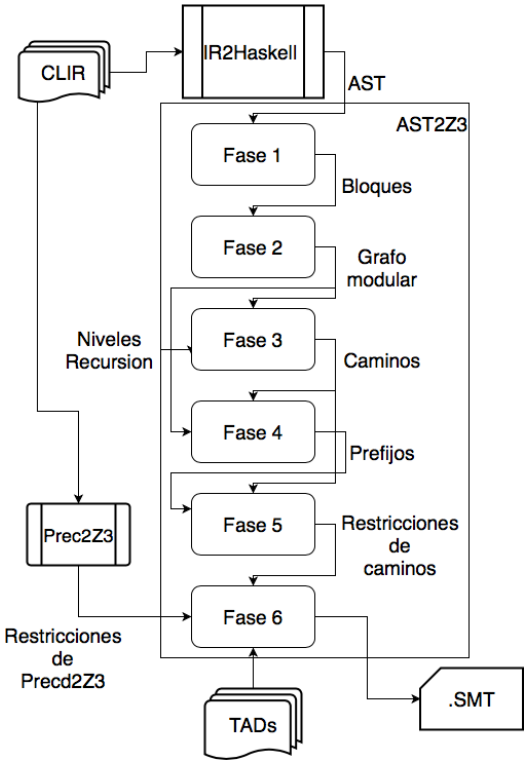

Figura B.1: Funcionamiento general de la herramienta AST2Z3.

En esta fase hemos hecho uso de la biblioteca *Data.Graph* [6] de Haskell que proporciona métodos para generar grafos y estudiar las componentes fuertemente conexas en ellos.

Fase 3

Mediante un sencillo método de conexión de dos nodos en un grafo con un camino y considerando los nodos de control se generan sin mayor dificultad los caminos del programa hasta un nivel de recursión dado como se indica en la Sección 4.1

Fase 4

Utilizando el grafo y los caminos por el mismo, podemos generar mediante el proceso descrito en la Sección 4.2 la lista de prefijos de los distintos bloques que conforman el camino. Como ya hemos comentado, en esta fase hemos tenido que utilizar una mónada transformadora de estados para consumir los caminos y generar en el estado la lista de prefijos.

Fase 5

Una vez que contamos con los caminos y los prefijos para cada bloque en cada uno de ellos, los transformamos a secuencias de restricciones en el lenguaje de Z3 mediante las tablas descritas en la Sección 5.1.

Fase 6

En esta última fase, se realiza la combinación de las restricciones ofrecidas por la herramienta Precd2Z3 y las que estamos generando. Así, se imprime secuencialmente en un archivo *.smt* la secuencia que se describe en la Sección 5.3.

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] 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.

[4] 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.

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

[6] David King and John Launchbury. Lazy depth-first search and linear graph algorithms in haskell. Technical report, Glasgow Workshop on Functional Programming, 1994.

[7] John Launchbury and Simon L. Peyton Jones. Lazy functional state threads. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 24–35, New York, NY, USA, 1994. ACM.

[8] Manuel Montenegro, Susana Nieva, Ricardo Peña, and Clara Segura. Liquid types for array invariant synthesis. In Deepak D'Souza and K. Narayan Kumar, editors, *Automated Technology for 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.

[9] 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.