Biblioteca Digital
F C E N - U B A

BIBLIOTECA CENTRAL LUIS F LELOIR
BIBLIOTECA CENTRAL LELOIR
FACULTAD DE CIENCIAS EXACTAS Y NATURALES UBA

Tesis Doctoral

Cómputo secuencial eficiente de cotas ajustadas, y su impacto en la performance de los análisis de programas basados en SAT

Ponzio, Pablo Daniel

2014-06-13

Este documento forma parte de la colección de tesis doctorales y de maestría de la Biblioteca Central Dr. Luis Federico Leloir, disponible en digital.bl.fcen.uba.ar. Su utilización debe ser acompañada por la cita bibliográfica con reconocimiento de la fuente.

This document is part of the doctoral theses collection of the Central Library Dr. Luis Federico Leloir, available in digital.bl.fcen.uba.ar. It should be used accompanied by the corresponding citation acknowledging the source.

Cita tipo APA:
Ponzio, Pablo Daniel. (2014-06-13). Cómputo secuencial eficiente de cotas ajustadas, y su impacto en la performance de los análisis de programas basados en SAT. Facultad de Ciencias Exactas y Naturales. Universidad de Buenos Aires.

Cita tipo Chicago:
Ponzio, Pablo Daniel. "Cómputo secuencial eficiente de cotas ajustadas, y su impacto en la performance de los análisis de programas basados en SAT". Facultad de Ciencias Exactas y Naturales. Universidad de Buenos Aires. 2014-06-13.

EXACTAS UBA
Facultad de Ciencias Exactas y Naturales

UBA
Universidad de Buenos Aires

Dirección: Biblioteca Central Dr. Luis F. Leloir, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires. Intendente Güiraldes 2160 - C1428EGA - Tel. (++54 +11) 4789-9293

Contacto: digital@bl.fcen.uba.ar

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

Cómputo secuencial eficiente de cotas ajustadas, y su impacto en la performance de los análisis de programas basados en SAT

Tesis para optar por el título de Doctor de la Universidad de Buenos Aires en el área Ciencias de la Computación

Pablo Daniel Ponzio

Directores:

Dr. Nazareno Matías Aguirre
Dr. Carlos Gustavo López Pombo

Lugar de Trabajo:

Departamento de Computación

Facultad de Ciencias Exactas Físico-Químicas y Naturales
Universidad Nacional de Río Cuarto

Río Cuarto, 2014

Resumen

Dada una descripción formal de los estados válidos del heap –por ejemplo, un invariante de una estructuras de datos– las *cotas ajustadas* para los campos del heap son el conjunto de los valores que pueden tomar los campos en alguna instancia válida. Así, los valores de campos que no pertenecen a las cotas ajustadas pueden ser ignorados por los análisis automáticos (acotados) de código, reduciendo así el espacio de estados a explorar, y mejorando la performance del análisis. Sin embargo, el cómputo de cotas ajustadas es costoso. El único enfoque conocido que computa estas cotas en un tiempo aceptable, TACO+, es intrínsecamente paralelo, y requiere de un cluster con varias computadoras para su ejecución eficiente.

En este trabajo se aborda el problema de computar eficientemente cotas ajustadas en entornos de ejecución secuenciales. Así, se presentan dos nuevos enfoques para resolver este problema. El primero, denominado bottom-up, utiliza un procedimiento de decisión basado en SAT para visitar instancias válidas del heap, y computa cotas ajustadas usando los valores de los campos de estas instancias. bottom-up, al igual que TACO+, requiere de una especificación en algún lenguaje declarativo de alto nivel –como JML para JAVA– de los estados válidos del heap. De esta manera, bottom-up y TACO+ generan cotas ajustadas equivalentes. El segundo, SLBD, se basa en una especificación del heap en términos de un predicado inductivo de la lógica de separación. Este enfoque computa cotas ajustadas mientras realiza (un número finito de) desplegados del predicado inductivo de entrada. En algunos casos SLBD puede producir cotas ajustadas diferentes a las de TACO+ (y bottom-up).

Se evaluaron experimentalmente los enfoques bottom-up, SLBD y la técnica relacionada TACO+, en la generación de cotas ajustadas para varias clases contenedoras JAVA. Los resultados muestran que, en un ambiente de ejecución secuencial, bottom-up es un orden de magnitud más rápido que TACO+ (secuencializado), y que SLBD es dos órdenes de magnitud más rápido que bottom-up. Además, se evaluó el impacto de las cotas ajustadas generadas por nuestros enfoques en la verificación de código y en la generación exhaustiva de estructuras (que satisfacen un predicado) basados en SAT. Los resultados indican que los enfoques que computan cotas ajustadas secuencialmente (vía bottom-up o SLBD), y luego las aprovechan durante el análisis, son significativamente más eficientes que los análisis convencionales, que no explotan la idea de cotas ajustadas. Además, para los análisis basados en SAT mencionados, los resultados muestran que las cotas ajustadas computadas por SLBD tienen un impacto similar a las computadas por TACO+ y bottom-up.

Efficient sequential tight field bounds computation, and their impact on the performance of SAT based program analysis

Abstract

Given a formal description of the feasible heap states –for example, an invariant for a data structure–, *tight field bounds* for heap fields are the set of values that the fields can take in any valid instance. Thus, field values that do not belong to the tight field bounds can be safely ignored by automated (bounded) program analyses, reducing the state space to be explored, and improving the performance of the analysis. However, computing tight field bounds is an expensive process. The only known approach to compute these bounds that performs reasonably, TACO+, is intrinsically parallel, and it requires a cluster of several computers to run efficiently.

In this work we address the problem of computing tight field bounds efficiently under a secuential execution environment. Thus, we introduce two novel approaches to solve this problem. The first, called bottom-up, uses a SAT based decision procedure to traverse valid heap instances, and computes tight field bounds using the field values of the visited instances. bottom-up, like TACO+, requires a description of the feasible heap states in a high level declarative language –like JML for JAVA–. In this way, bottom-up and TACO+ yield equivalent tight field bounds. The second, SLBD, is based on a specification of the valid heap states in terms of a separation logic's inductive predicate. This approach computes tight field bounds during the process of unfolding its input's inductive predicate (a finite number of times). In some cases, the tight field bounds computed by SLBD and TACO+ (and bottom-up) might differ.

We assessed bottom-up, SLBD and the related technique TACO+ experimentally, in generating tight field bounds for several JAVA container classes. The results show that, under a secuential execution environment, bottom-up is an order of magnitude faster than TACO+ (secuentialized), and that SLBD is two orders of magnitude faster than bottom-up. Moreover, we assesed the impact of the tight field bounds produced by our approaches in SAT based code analysis and in exhaustive bounded generation of structures (satisfying a predicate). The results show that the approaches that compute tight field bounds secuentially (via bottom-up or SLBD), and subsecuently exploit them during analysis, are significatively faster than the more conventional analyses that do not take advantage of tight field bounds. Furthermore, for the aformentioned SAT based analyses, the experiments show that the tight field bounds computed by SLBD have a similar impact to those computed by TACO+ and bottom-up.

Agradecimientos

Principalmente quiero agradecer a mi director, Nazareno Aguirre, por ser una fuente de inspiración para mi trabajo, y un modelo de investigador –y persona– a seguir. Sus ideas, sugerencias, las discusiones esclarecedoras, y su constante apoyo durante el transcurso de mi doctorado fueron fundamentales para mejorar este trabajo, y, especialmente, mi formación profesional. También quiero agradecer a Marcelo Frias, Nicolás Rosner y Carlos López Pombo. A Nico y Marcelo por contribuir significativamente a mi formación durante los trabajos en conjunto que realizamos. A Charly por su apoyo constante, sus comentarios y sugerencias sobre este trabajo. Es un honor y un orgullo para mi haber podido trabajar con gente tan profesional y de la excelente calidad humana de Naza, Marcelo, Nico y Charly.

Por último, quiero agradecer al Departamento de Computación de la Universidad de Río Cuarto por darme la posibilidad de finalizar mis estudios de posgrado, y a mis compañeros de trabajo y de oficina del DC de la UNRC: a Renzo Degiovanni, Valeria Bengolea, Germán Regis, Francisco Bavera y Marcelo Uva, con quienes tuve la oportunidad de trabajar en distintas cuestiones relacionadas al doctorado; y a Pablo Castro, Sonia Permigiani, María Marta Novaira, Cecilia Kilmurray, Gastón Scilingo y Marcelo Arroyo, con quienes compartí trabajos docentes/oficina/charlas de investigación. Todos ellos excelentes personas, con un fuerte compromiso por su trabajo.

Dedicatoria

A mi familia: mi hermano, Rodrigo Ponzio, mi mamá Stella Arnaudo, y mi papá, Daniel Ponzio, por el afecto que siempre me brindaron, y por apoyarme para que pueda lograr este objetivo. A mis mejores amigos: Juan Monge, Lucio Colodro y Javier Colodro, por estar siempre.

Índice general

1. Introducción **2**
1.1. Antecedentes 2
1.2. Análisis de código basado en satisfactibilidad booleana (SAT) 3
1.2.1. Cotas ajustadas y rotura de simetrías para un análisis basado en SAT eficiente 5
1.3. Objetivos y Contribuciones 6
1.4. Organización de la tesis 9

2. Preliminares **10**
2.1. Alloy 10
2.1.1. El lenguaje 11
2.1.2. Análisis automático basado en SAT 13
2.2. DynAlloy 15
2.2.1. El lenguaje 15
2.2.2. Análisis automático basado en SAT 17
2.3. Analizando JAVA con DynAlloy 18
2.3.1. Modelo del heap JAVA 18
2.3.2. Traducción de comandos JAVA a DynAlloy 20
2.3.3. Análisis de propiedades del código JAVA 22
2.4. KodKod 23
2.4.1. Representación de campos Alloy 24
2.4.2. Instancias parciales 25
2.5. Análisis de código basado en satisfactibilidad booleana 26
2.5.1. TACO: un análisis de código basado en SAT 27
2.6. Canonización del heap 29
2.6.1. Predicados de rotura de simetría en DynAlloy 30
2.7. Cotas Ajustadas para el Heap 35
2.7.1. Ejemplo motivador 35
2.7.2. Definiciones 37

2.7.3. Cotas ajustadas y rotura de simetrías 38
2.8. TACO+: análisis de código basado en SAT explotando cotas ajustadas 38
2.9. Lógica de Separación 43
2.9.1. Predicados Inductivos 44
2.9.2. Lógica de separación y cotas ajustadas 44
2.9.3. Sintaxis 45
2.9.4. Semántica 48
2.10. El SMT solver Yices 51

3. Cómputo secuencial de cotas ajustadas mediante visita de instancias 55
3.1. Un caso de estudio: AVLs 55
3.2. Intuición y ejemplo motivador 57
3.3. Implementaciones del enfoque 62
3.3.1. Algoritmo estándar 64
3.3.2. Extendiendo todos los conjuntos a la vez 65
3.3.3. Cómputo campo por campo 67
3.4. Cómputo incremental de cotas ajustadas 68
3.5. Interacción entre bottom-up y TACO 70

4. Cómputo secuencial de cotas ajustadas basado en predicados inductivos de la lógica de separación 73
4.1. Fuerza bruta 74
4.1.1. Desplegando predicados inductivos 76
4.1.2. Generando instancias concretas a partir de heaps simbólicos . 81
4.1.3. Codificación de heaps simbólicos en Yices 82
4.1.4. Rotura de simetrías 89
4.1.5. Cotas ajustadas a partir del conjunto completo de instancias del heap 92
4.2. Cómputo de cotas ajustadas al vuelo 93
4.2.1. Campos que apuntan “hacia atrás” 98
4.2.2. Tratando con predicados inductivos más generales 101
4.3. Memorización 103
4.3.1. Forma canónica de los resultados 105
4.3.2. Algoritmo SLBDM 106
4.3.3. Memorización y campos que apuntan hacia atrás 108
4.4. Cotas ajustadas primero en profundidad 109
4.4.1. Predicados de rotura de simetrías primero en profundidad . . 111

4.5. Interacción entre el cómputo de cotas basado en predicados inductivos y TACO 113

5. Evaluación experimental 116

5.1. Evaluación de los diferentes enfoques de cómputo de cotas ajustadas . 117

5.1.1. Evaluación de las distintas versiones de bottom-up 118

5.1.2. Beneficios del cómputo incremental de cotas ajustadas 119

5.1.3. bottom-up vs. TACO+ 121

5.1.4. SLBD vs. bottom-up vs. TACO+ 123

5.2. Impacto de las cotas ajustadas en la verificación de código basado en SAT 125

5.2.1. Rotura de simetrías en la verificación de código basado en SAT 126

5.2.2. Cotas ajustadas en la verificación de código basada en SAT . . 129

5.2.3. Cómputo e instrumentación de la verificación de código basada en SAT con cotas ajustadas 133

5.3. Impacto de las cotas ajustadas en la generación exhaustiva de estructuras basada en SAT 137

5.4. Discusión 140

6. Trabajos relacionados, conclusiones y trabajos futuros 145

6.1. Trabajos Relacionados 145

6.2. Conclusiones 147

6.3. Trabajos futuros 149

Índice de figuras

2.1. Una instancia particular (relacional) de la especificación Alloy de listas enlazadas acíclicas 14
2.2. Representación gráfica de la instancia de la figura 2.1 14
2.3. Fragmento de una implementación JAVA de listas simplemente encadenadas acíclicas. 19
2.4. Especificación DynAlloy para verificar que el método `addFirst` del código JAVA de la figura 2.3 mantiene el invariante `listInvariant`. . 23
2.5. Etapas principales del análisis de código TACO. 27
2.6. Dos listas simplemente encadenadas isomorfas 29
2.7. Valores que puede tomar el campo `next` de listas simplemente encadenadas, con scope 3 para `Node` 36
2.8. Una cota ajustada para el campo `next` de listas, con *scope* tres para `Node`, utilizando el invariante de aciclicidad de la figura 2.4 37
2.9. Esquema del análisis de código TACO+ 39
2.10. Sintaxis de los heaps simbólicos de la lógica de separación 46
2.11. Semántica de los heaps simbólicos de la lógica de separación 49

3.1. Invariante de clase para la estructura AVL en DynAlloy 57
3.2. Dominios de los campos `left`, `right` y `height` de AVL para un máximo de 5 nodos, en el contexto de análisis acotado de código. 58
3.3. Cotas ajustadas para los campos `left`, `right` y `height` de AVL para un máximo de 5 nodos. 58
3.4. Ejemplo de cómputo de cotas ajustadas usando bottom-up: AVLs con 5 nodos 61
3.5. Ejemplo de cómputo incremental de cotas ajustadas usando Bottom-up-inc: AVLs para un máximo de 6 nodos, partiendo de las cotas ajustadas para AVLs y scope 5 (figura 3.3) 68
3.6. Diagrama del análisis de código bottom-up+TACO. 71

4.1. Heaps simbólicos obtenidos de expandir `slist` (izquierda) y su representación gráfica (derecha). La fila i-esima muestra el resultado de expandir `slist` $i-1$ veces. 74
4.2. Función VDECL: retorna un conjunto de declaraciones Yices para las variables del heap simbólico 83
4.3. Función PURE: traduce la parte pura de un heap simbólico a Yices . . 84
4.4. Función DISJ: retorna el conjunto de variables de un heap simbólico que no pueden ser alias 85
4.5. Función RepVars(sh, a): retorna una instancia concreta a partir del heap simbólico sh y la asignación a 88
4.6. Función SBctr: devuelve una conjunción de fórmulas puras que, al añadirlas al heap simbólico de entrada, permiten evitar la generación de instancias simétricas 90
4.7. Árbol binario con sus nodos etiquetados en un recorrido primero en anchura de los campos `left` y `right`. 110
4.8. Árbol binario con sus nodos etiquetados en un recorrido primero en profundidad de los campos `left` y `right`. 111
4.9. Diagrama del análisis de código SLBD+TACO. 114

Índice de cuadros

2.1. Representación la relación `f: A -> (B+null)` como matriz de variables proposicionales 24
2.2. Representación KodKod del campo `next` de la especificación de listas simplemente encadenadas (figura 2.4), con scope 3 para `Node` 25
2.3. Representación KodKod del campo `next` de la especificación de listas simplemente encadenadas (figura 2.4) para scope 3, usando U_{next} como cota superior 26

5.1. Tiempos de cómputo de cotas ajustadas para las distintas implementaciones de bottom-up. 119
5.2. Tiempos de cómputo de cotas ajustadas para bottom-up-f y bottom-up-f-inc (incremental). bottom-up-f(sum) y bottom-up-f-inc(sum) muestran la suma de los tiempos de cómputo para todos los scopes menores o iguales a s (para cada s). 120
5.3. Ganancia en tiempo de usar bottom-up-f en lugar de TACO+(seq) para el cómputo secuencial de cotas ajustadas. 122
5.4. Tiempos de cómputo de cotas ajustadas a partir de especificaciones JML (TACO(seq) y bottom-up-f) y de predicados inductivos (SLBD). 124
5.5. Comparación entre TACO (T), TACO$_{S_{DF}}$ (T$_{S_{DF}}$) y TACO$_{S_{BF}}$ (T$_{S_{BF}}$) en la verificación de código (scopes entre 7 y 12). 127
5.6. Comparación entre TACO (T), TACO$_{S_{DF}}$ (T$_{S_{DF}}$) y TACO$_{S_{BF}}$ (T$_{S_{BF}}$) en la verificación de código (scopes entre 13 y 17). 128
5.7. Tiempos de verificación de código para TACO$_{S_{DF}}$ (T$_{S_{DF}}$), TACO$_{S_{BF}}$ (T$_{S_{BF}}$), TACO$_{SB_{DF}}$ (T$_{SB_{DF}}$) y TACO$_{SB_{BF}}$ (T$_{SB_{BF}}$) (scopes entre 7 y 12). 130
5.8. Tiempos de verificación de código para TACO$_{S_{DF}}$ (T$_{S_{DF}}$), TACO$_{S_{BF}}$ (T$_{S_{BF}}$), TACO$_{SB_{DF}}$ (T$_{SB_{DF}}$) y TACO$_{SB_{BF}}$ (T$_{SB_{BF}}$) (scopes entre 13 y 17). 131

5.9. Tiempos totales de verificación para $\text{TACO}_{S_{DF}}$ ($\text{T}_{S_{DF}}$), SLBD+TACO (SLBD+T) y bottom-up+TACO (BU+T), incluyendo tiempos de cómputo de cotas ajustadas para los últimos dos enfoques (scopes entre 7 y 12). 134
5.10. Tiempos totales de verificación para $\text{TACO}_{S_{DF}}$ ($\text{T}_{S_{DF}}$), SLBD+TACO (SLBD+T) y bottom-up+TACO (BU+T), incluyendo tiempos de cómputo de cotas ajustadas para los últimos dos enfoques (scopes entre 13 y 17). 135
5.11. Tiempos de generación exhaustiva de estructuras para $\text{GEN}_{S_{DF}}$, $\text{GEN}_{S_{BF}}$, $\text{GEN}_{SB_{DF}}$, y $\text{GEN}_{SB_{BF}}$ (scopes entre 7 y 14). 137
5.12. Tiempos de generación exhaustiva de estructuras para los enfoques $\text{GEN}_{SB_{DF}}$ y $\text{GEN}_{SB_{BF}}$, incluyendo tiempos de cómputo de cotas ajustadas. $\text{GEN}_{SB_{BF}}$(a) considera los tiempos de cómputos de cotas ajustadas amortizados entre los experimentos de la sección anterior y el actual. 139

Capítulo 1

Introducción

1.1. Antecedentes

En la década del 40 surgieron las primeras computadoras eléctronicas, cuya utilidad estaba limitada a la realización de cálculos matemáticos "sencillos" (respecto de los efectuados por las computadoras actuales). Además, su costo era elevado, ocupaban una gran cantidad de espacio físico, y su programación requería de intrincados mecanismos – como por ejemplo, el uso de tarjetas perforadas. Debido a estas características, el uso de computadoras en aquella época estaba restringido a ambientes científicos o militares, y sus usuarios eran, generalmente, ciéntificos o personal altamente capacitado.

En la actualidad, en cambio, las computadoras son ubicuas: teléfonos celulares, cajeros automáticos, sistemas de control de vuelo, equipamientos médicos, sistemas de posicionamiento global, y una innumerable lista de dispositivos cotidianos basan su funcionamiento en la ejecución de *programas*. Estos programas realizan tareas de lo más diversas, pueden tener cientos de miles de líneas de código, e interactuar con otros programas –localmente o a través de redes (como internet)– para llevar a cabo su función. Evidentemente, los programas de la actualidad son sustancialmente más complejos que sus pares de antaño.

De esta manera, diversas tareas de la vida cotidiana se han automatizado, incluyendo las denominadas *críticas*: actividades en las que una falla inesperada en el programa de un dispositivo puede resultar en la pérdida de cantidades millonarias de dinero, y/o vidas humanas. Ejemplos famosos –y desafortunados– de este tipo de errores son: la explosión del cohete espacial Arianne 5, a 40 segundos de despegar, debido a un problema en su programa de control de navegación [20], cuyo costo se estima en varios cientos de millones de dólares; y las (al menos) cinco personas fallecidas por haber sido expuestas a una sobredosis de radiación – alrededor de 100 veces

mayor que la recomendada – por un mal cálculo de la máquina que administró la terapia (Therac-25) [35].

Los ejemplos anteriores ilustran la necesidad de desarrollar métodos de construcción de programas confiables. La rama de la ciencia de la computación denominada ingeniería de software fue concebida con el objetivo de abordar el problema de la construcción de programas de calidad. Entre los enfoques propuestos por los científicos de esta área para mejorar la calidad de los programas producidos, en las últimos dos décadas han recibido especial interés los denominados *automáticos*: aquellos capaces de evaluar la corrección de los programas –y, en algunos casos, exponer sus fallas– sin requerir asistencia por parte del usuario. Normalmente, estas técnicas exploran una gran cantidad de estados del programa bajo análisis con el objetivo de exponer errores. Típicamente, los enfoques automáticos sufren de problemas de *escalabilidad*, debido a que la cantidad de estados a explorar crece exponencialmente a medida que crece el tamaño de los programas. Por ejemplo, la cantidad de estados crece exponencialmente con la cantidad de líneas de código.

1.2. Análisis de código basado en satisfactibilidad booleana (SAT)

El análisis de programas basado en satisfactibilidad booleana (análisis basado en SAT para abreviar) es un enfoque automático de análisis de código. Para tratar con el problema de escalabilidad mencionado anteriormente, este enfoque realiza un *análisis acotado y exhaustivo* del código. Por un lado, el análisis se dice *acotado* porque impone límites en la cantidad máxima de elementos de los tipos de datos del programa, y en la cantidad de iteraciones permitidas para los ciclos (a una cantidad finita, y usualmente pequeña en ambos casos). Estos límites son provistos por el usuario del análisis, y se denominan *scopes* en la literatura. Por otro lado, el análisis se dice *exhaustivo* porque recorre el espacio de datos acotado completo en busca de errores.

A nivel de implementación, el análisis basado en SAT codifica el programa a analizar y su especificación en una fórmula de la lógica proposicional. La codificación se realiza de manera tal que una asignación (de valores de verdad a variables) que satisface la fórmula corresponde a una violación de la especificación del programa. Luego, la fórmula que codifica el programa se usa como entrada de un SAT solver: un procedimiento de decisión para la lógica proposicional, que busca exhaustivamente asignaciones que satisfacen la fórmula. En caso de existir una tal asignación, el análisis la transforma en una ejecución real del programa bajo análisis, y la muestra al usuario como *testigo* del error.

Aunque normalmente el análisis basado en SAT es capaz de examinar una cantidad enorme de ejecuciones –en gran parte gracias a la eficiencia de los SAT solvers modernos–, no permite asegurar la ausencia de errores debido a su naturaleza acotada. De todas maneras, La hipótesis del scope pequeño (*small scope hypothesis*) [6] explica la importancia de la técnica en la práctica. Esta hipótesis afirma que una gran proporción de los errores pueden exponerse testeando el programa con todos los valores posibles dentro de un scope pequeño. [6] presenta evidencia empírica sobre la validez de esta hipótesis. Adicionalmente, diversos enfoques de análisis de código basado en SAT se usaron exitosamente para descubrir errores de programas escritos en lenguajes de programación de alto nivel. Por ejemplo, [32, 48] analizan código en el lenguaje C, y [17, 25, 27] código JAVA.

Otra característica importante de las técnicas basadas en SAT es que admiten lenguajes de especificación muy expresivos (respecto de otras técnicas automáticas de análisis), que permiten, por ejemplo, describir propiedades complejas de estructuras de datos dinámicas, como diferentes tipos de listas enlazadas, árboles balanceados (Red-Black Trees, AVL, etc.), etc. [17, 25, 27].

A pesar de ser acotados, los análisis de código basados en SAT presentan importantes problemas de escalabilidad. En primer lugar, es sabido que el problema de encontrar una asignación que satisfaga una fórmula proposicional –conocido como SAT– es NP completo. No obstante, los SAT solvers modernos implementan técnicas de optimización que les permiten resolver este problema para fórmulas de gran tamaño (decenas de miles de variables y millones de cláusulas). En segundo lugar, el tamaño de la fórmula que representa el programa depende de la complejidad del programa: cantidad de líneas de código, tipos de datos utilizados, etc. Por ejemplo, los experimentos de [25] indican que los análisis basados en SAT convencionales de programas que manipulan tipos de datos dinámicos complejos –listas, árboles, etc.– normalmente tienen problemas para lidiar con estructuras dinámicas con 5 o más nodos (scope 5).

Aún cuando la hipótesis del scope pequenño da cuenta de la utilidad de los análisis basados en SAT en la práctica, la utilidad de contar con análisis más eficientes y escalables es evidente: lo segundo permitiría tratar con scopes más grandes, y por lo tanto exponer errores que con scopes más pequeños pasarían desapercibidos; lo primero implicaría que el análisis finalice más rápidamente, proveyendo una mejor experiencia al usuario del mismo.

1.2.1. Cotas ajustadas y rotura de simetrías para un análisis basado en SAT eficiente

En [25, 27] se presenta un enfoque novedoso para mejorar el rendimiento de los análisis de código basados en SAT, para programas que manipulan estructuras de datos dinámicas complejas. Estas estructuras son la base de diversas librerías provistas por lenguajes de programación orientados a objetos contemporáneos, y son reutilizadas una y otra vez por los desarrolladores en la construcción de todo tipo de programas. Por esta razón, ganar confianza sobre la corrección de las librerías, o descubrir errores en una etapa temprana es una tarea de crucial importancia. En particular, en estos artículos se analizan varias clases contenedoras (*container classes*) JAVA.

Por un lado, [25] propone instrumentar el análisis de código con *predicados de rotura de simetrías*. Estos predicados fijan un orden único para la asignación de identificadores a los elementos del heap. Como ejemplo, asumamos un modelo de listas simplemente encadenadas y un máximo de tres nodos para el heap, con identificadores N_0, N_1 y N_2, y predicados de rotura de simetrías que inducen el orden $N_0 < N_1 < N_2$ sobre los identificadores. De esta manera, la única lista válida de tres elementos surge de etiquetar el primer nodo con N_0, el segundo con N_1 y el tercero con N_2. Los etiquetados que no respetan el orden dado (por ejemplo, asignar N_1, N_2 y N_0 al primer, segundo y tercer nodo, respectivamente) generan estructuras isomorfas (equivalentes) a la anterior, y son eliminadas por la rotura de simetrías (sin comprometer la corrección del análisis). Esta reducción en la cantidad de estructuras válidas del heap tiene un impacto sustancial en la eficiencia del análisis. En [25] se presenta un mecanismo para instrumentar el análisis de código basado en SAT con rotura de simetrías de manera completamente automática.

Por otro lado, [25] propone realizar un análisis de la especificación de las entradas del programa –invariantes de representación de las estructuras, precondiciones– para encontrar valores de los campos del heap que están prohibidos por la especificación. Por ejemplo, un invariante de representación para listas acíclicas obliga a que, para cualquier nodo n, $n.\text{next} \neq n$ (el campo *next* de un nodo apunta al siguiente nodo en la lista). El objetivo es computar *cotas ajustadas para los campos del heap* (*tight field bounds*): los conjuntos de valores permitidos para los campos. Luego, las cotas ajustadas computadas se usan para instrumentar el análisis de código. Como en el análisis basado en SAT cada valor posible para un campo se codifica como una variable proposicional en la fórmula que representa el programa, la instrumentación permite eliminar variables *irrelevantes*: aquellas que corresponden a valores de campos prohibidos por la especificación.

Resumiendo, [25] propone dividir el proceso de análisis de código basado en SAT

en dos partes:

1. Cómputo de cotas ajustadas para los campos del heap usando el invariante de la estructura de datos.

2. Instrumentar el análisis de código con las cotas computadas en el paso anterior, eliminando variables proposicionales irrelevantes, y ejecutarlo en busca de errores en el código.

Es importante destacar que ambos pasos del enfoque pueden implementarse de manera completamente automática.

En [25] se presenta una herramienta automática de verificación de propiedades de código JAVA denominada TACO. TACO+ es la versión de TACO que divide el análisis en los dos pasos mencionados anteriormente, y lo instrumenta con predicados de rotura de simetrías. Además, TACO+ implementa un algoritmo paralelo de cómputo de cotas ajustadas (punto 1 anterior). Los resultados experimentales de [25] muestran que el análisis instrumentado con cotas ajustadas y predicados de rotura de simetrías, TACO+, es capaz de verificar código eficientemente, usando scopes significativamente más grandes que los soportados por el análisis convencional –sin cotas ajustadas ni rotura de simetrías. Por ejemplo, TACO+ permitió exponer un error previamente desconocido en una implementación de Binomial Heaps, que sólo aparece en estructuras con más de 12 nodos (el análisis convencional usualmente falla con 5 o más nodos).

Sin embargo, en los experimentos mencionados se asume que se dispone de las cotas ajustadas antes de realizar el análisis de código. Es decir, no se incluyen en los tiempos de análisis el alto costo del cómputo de las cotas ajustadas. Los experimentos de [25] muestran que, usando un cluster de dieciséis computadoras con cuatro núcleos cada una, el algoritmo tarda varios minutos para computar cotas ajustadas en la mayoría de los casos. Así, el punto débil de TACO+ reside en el alto costo asociado al cómputo de cotas ajustadas, y, hasta el momento, no se conocen técnicas alternativas para el cómputo de cotas ajustadas. Este es el problema central que abordaremos en esta tesis.

1.3. Objetivos y Contribuciones

Como se mencionó en la sección anterior, las cotas ajustadas son un mecanismo para mejorar la eficiencia de los análisis de programas, pero el cómputo de estas cotas es costoso. La motivación para abordar el problema de cómputo eficiente de cotas ajustadas es evidente: contar con enfoques que computen cotas ajustadas en tiempos razonables permitiría construir análisis de código que –como el enfoque

discutido en la sección anterior– computen cotas ajustadas y luego las aprovechen durante el análisis, que presenten una eficiencia superior a los análisis convencionales –sin cotas ajustadas. Además, es importante que los enfoques de cómputo de cotas ajustadas sean *secuenciales* –que puedan ejecutarse en una sola computadora–, ya que el requerimiento de un cluster de computadoras puede ser un obstáculo para la adopción de los análisis basados en cotas ajustadas.

De esta manera, las contribuciones principales de este trabajo son dos enfoques secuenciales para el cómputo eficiente de cotas ajustadas:

- bottom-up, que consiste en requerir iterativamente instancias válidas del heap –que satisfacen el invariante de la estructura de datos– a un procedimiento de decisión (SAT solving), y utilizar sus valores de campos como componentes de las cotas ajustadas a generar. Al igual que el enfoque paralelo TACO+ [25], bottom-up requiere de una descripción del invariante en un lenguaje de especificación declarativo (como JML [12] para JAVA). Por lo tanto, para un mismo invariante estos enfoques producen cotas ajustadas equivalentes. Además, bottom-up posibilita el cómputo incremental de cotas ajustadas: aprovechar cotas ajustadas computadas para un scope más pequeño para acelerar el cómputo de cotas ajustadas para un scope mayor. Esto permite amortizar el costo de computar cotas ajustadas para scopes cada vez más grandes (un escenario frecuente cuando se usa un análisis de programa acotado es comenzar con scopes pequeños, e ir incrementándolos a medida que el análisis responde favorablemente).

- SLBD que utiliza los predicados inductivos de la lógica de separación [43] como formalismo para describir invariantes de las estructuras de datos. SLBD computa cotas ajustadas a medida que va desplegando el predicado inductivo que toma como entrada. Este enfoque aprovecha que el operador $*$ de la lógica de separación permite descartar de antemano casos de aliasing entre elementos del heap, para reducir la cantidad de invocaciones a un procedimiento de decisión que efectúa para computar las cotas. Además, se define un mecanismo de rotura de simetría para SLBD, y se lo extiende para soportar la técnica de optimización de programas conocida como memorización. Por estas razones, SLBD es sustancialmente más rápido que las restantes técnicas, aunque en algunos casos genera cotas ajustadas ligeramente menos precisas que TACO+ (y bottom-up).

Es importante subrayar que las técnicas anteriores permiten definir análisis de programas que computan y explotan las cotas ajustadas capaces de ejecutarse por completo en una única computadora, por primera vez.

Las enfoques propuestas se evaluaron experimentalmente en diversos casos de estudio, que abarcan clases contenedoras JAVA (*container classes*) con diferentes grados de complejidad en sus invariantes, tomados de [25]. Las clases se encuentran anotadas con invariantes y pre y postcondiciones JML para los métodos. Por un lado, se evaluó la eficiencia en tiempo de ejecución de bottom-up y SLBD contra el componente de cómputo de cotas ajustadas de TACO+, simulando la ejecución de este último en un entorno de ejecución secuencial. Por otro lado, se evaluó la precisión de las cotas ajustadas generadas por las distintas técnicas (como se mencionón anteriormente, bottom-up TACO+ computan cotas equivalentes, pero SLBD puede producir cotas distintas), y su impacto en dos análisis de programas basados en SAT: la verificación acotada de métodos, y en la generación exhaustiva de estructuras que satisfacen los invariantes de clases de nuestros casos de estudio. Para esto se definieron enfoques que computan cotas ajustadas (vía bottom-up o SLBD), y luego instrumentan los análisis de programa con las cotas ajustadas.

Los resultados de los experimentos permiten concluir que:

- En un entorno de ejecución secuencial bottom-up es un orden de magnitud más rápido que TACO+ (simulando su ejecución secuencial). Además, SLBD es dos órdenes de magnitud más rápido que bottom-up.

- Cuando se computan cotas ajustadas con scopes cada vez más grandes (para cuando se ejecuta el análisis de programa incrementando los scopes cada vez), el enfoque incremental permite amortizar completamente el tiempo requerido para el cómputo de las cotas para los scopes más pequeños (menos el último).

- Los análisis de programa que computan cotas ajustadas (vía bottom-up o SLBD) y luego las aprovechan durante el análisis son significativamente más eficientes que los análisis convencionales –que no explotan la idea de cotas ajustadas.

- En los análisis de programa considerados en nuestros experimentos, las cotas ajustadas computadas por SLBD tienen un impacto similar (son ligeramente menos eficaces) a las computadas por TACO+ (y bottom-up).

Es importante destacar que existen otros análisis automáticos –además de TACO+– que se benefician del uso de cotas ajustadas. La herramienta de generación automática de casos de prueba basada en SAT, FAJITA [5], usa cotas ajustadas para lograr una mejor performance. Diversas extensiones del model checker Symbolic JAVA PathFinder [3] incorporan las cotas ajustadas al mecanismo de ejecución simbólica lazy de esta herramienta [28, 44], también para acelerar la generación automática de casos de prueba. En consecuencia, el cómputo de cotas ajustadas eficiente, facilitado por las técnicas presentadas en este trabajo, es relevante para estas herramientas.

1.4. Organización de la tesis

A continuación se describe el formato de este trabajo. En el capítulo 2 se hará un breve repaso de los conceptos previos requeridos para entender los temas principales de esta tesis. El capítulo 3 introduce los enfoques de cómputo de cotas ajustadas secuenciales bottom-up y bottom-up incremental, y el capítulo 4 presenta el enfoque SLTB. La evaluación experimental de las técnicas se realiza en el capítulo 5. En el capítulo 6.1 se mencionan los trabajos relacionados. Por último, en el capítulo 6.2 se discuten las conclusiones, y posibles trabajos futuros relacionados con esta tesis.

Capítulo 2

Preliminares

2.1. Alloy

Alloy [33] es un lenguaje de especificación declarativo basado en la lógica relacional, una extensión de la lógica de primer orden con operadores como composición y clausura transitiva de relaciones. Alloy es un lenguaje simple, fácil de aprender, y posee una herramienta asociada para verificar propiedades de las especificaciones y exponer errores indeseados. Esta herramienta, llamada Alloy Analyzer, implementa un análisis automático de las especificaciones Alloy, basado en SAT solving, y puede descargarse gratuitamente de [1].

En este trabajo se emplea Alloy como lenguaje intermedio en la traducción de un programa JAVA con anotaciones JML a una fórmula proposicional. El framework de análisis de código usado en este trabajo traduce el código y sus anotaciones a Alloy, y luego realiza un análisis (basado en SAT) de la especificación Alloy obtenida, mediante el Alloy Analyzer. Varios motivos fundamentan la decisión de usar Alloy como lenguaje intermedio en el análisis de código. Primero, el lenguaje posee un gran poder expresivo, lo que permite codificar programas que manipulan estructuras de datos dinámicas con especificaciones complejas, como árboles balanceados, diversos tipos de listas encadenadas, etc. Segundo, el Alloy Analyzer implementa un análisis basado en SAT eficiente [45]. Consecuentemente, es más sencillo codificar el programa en Alloy y analizar la especificación resultante con el Alloy Analyzer que desarrollar un análisis basado en SAT eficiente desde cero. Por último, las especificaciones Alloy pueden instrumentarse con cotas ajustadas, y por lo tanto las técnicas de cómputo de cotas ajustadas presentadas en este trabajo pueden incorporarse en el análisis de código sin mayores complicaciones.

2.1.1. El lenguaje

A continuación discutiremos algunas de las construcciones del lenguaje a través de ejemplos. El lector interesado puede consultar [33] para una descripción más completa.

La parte estructural de las especificaciones Alloy se define mediante *signaturas*. Las signaturas definen conjuntos de elementos no interpretados en las especificaciones Alloy. A menos que se especifique lo contrario (usando la palabra clave `extends`), las signaturas de una misma especificación se asumen disjuntas. Además, las signaturas poseen *campos* que representan relaciones entre la signatura que los define y otras signaturas.

Ejemplo 2.1.1. Consideremos la definición de un modelo de listas simplemente encadenadas en Alloy:

```
one sig null { }

sig List {
    head: one Node+null
}
```

```
sig Node {
    data: one Int,
    next: one Node+null
}
```

La signatura `null` define un conjunto con un único elemento –debido al modificador `one`–, que representa la referencia indefinida, *null*, de los lenguajes de programación. `List` define listas simplemente encadenadas. En este modelo de listas simplemente encadenadas, cada lista posee una referencia al primer nodo de la lista, representado por el campo `head`. Formalmente, $\texttt{head} \subseteq (\texttt{List} \times (\texttt{Node} + \texttt{null}))$, es decir, `head` relaciona elementos de la signatura `List` con elementos del conjunto `Node+null` (`+` denota la unión de conjuntos) –notar que la lista es vacía cuando `head` toma el valor `null`. El modificador `one` obliga a que cada elemento en el dominio de `head` esté relacionado con exactamente uno en su codominio, esto es, `head` debe ser una función total. Por otra parte, la signatura `Node` modela los nodos de las listas. Cada nodo tiene una referencia al nodo que le sigue en la lista (campo `next`) – que puede ser `null`–, y almacena un valor entero (campo `data`). Formalmente, $\texttt{data} \subseteq (\texttt{Node} \times (\texttt{Int}))$ y $\texttt{next} \subseteq (\texttt{Node} \times (\texttt{Node} + \texttt{null}))$. □

En Alloy, los términos se construyen a partir de las signaturas, los campos de las signaturas, y las constantes `univ`, `none` e `iden`. `univ` denota el conjunto de todos los elementos de la especificación, `none` el conjunto vació, e `iden` la identidad sobre `univ`. Si `T` es un término que denota una relación binaria, `~T`, `^T` y `*T` representan la traspuesta, la clausura transitiva y la clausura reflexiva y transitiva `T`, respectivamente. Si $\texttt{T}_1$ y $\texttt{T}_2$ son términos que representan relaciones con la misma aridad, $\texttt{T}_1\texttt{+}\texttt{T}_2$, $\texttt{T}_1\texttt{\&}\texttt{T}_2$ y $\texttt{T}_1\texttt{-}\texttt{T}_2$ representan la union, intersección y diferencia entre las relaciones $\texttt{T}_1$ y $\texttt{T}_2$, respectivamente. Si $\texttt{T}_1$ y $\texttt{T}_2$ son relaciones de cualquier aridad, $\texttt{T}_1\texttt{.}\texttt{T}_2$ denota

la composición relacional de $\mathtt{T}_1$ y $\mathtt{T}_2$. Formalmente, para $\mathtt{T}_1 \subseteq A_1 \times \ldots \times A_n \times B$ y $\mathtt{T}_2 \subseteq B \times C_1 \times \ldots \times C_m$, $\mathtt{T}_1.\mathtt{T}_2 = \{(a_1, \ldots, a_n, c_1, \ldots, c_m) | \exists b \in B, (a_1, \ldots, a_n, b) \in \mathbf{T}_1, (b, c_1, \ldots, c_m) \in \mathbf{T}_2\}$.

Para construir fórmulas a partir de los términos se usan los siguientes operadores. $\mathtt{T}_1$ `in` $\mathtt{T}_2$ indica que la relación $\mathtt{T}_1$ está contenida en $\mathtt{T}_2$, y $\mathtt{T}_1$=$\mathtt{T}_2$ que $\mathtt{T}_1$ y $\mathtt{T}_2$ son la misma relación. Además, están disponibles la negación lógica, `!`, y los operadores binarios: conjunción, `&&`, disyunción, `||`, implicación, `=>`, y equivalencia, `<=>`. Por último, Alloy permite el uso de los cuantificadores existencial, `some`, y universal, `all`, en la definición de fórmulas.

Para introducir axiomas en una especificación Alloy – fórmulas que se asumen válidas en la especificación – se definen *hechos* (con la palabra clave `fact`).

Ejemplo 2.1.2. Para asegurar que las listas del ejemplo 2.1.1 no contienen ciclos se agrega a la especificación Alloy el siguiente hecho:

```
fact acyclic {
    all l: List, all n: Node | n in l.head.*next => !n in n.next.*next
}
```

Alloy es un lenguaje tipado: cada signatura define un tipo. De esta manera, en la declaración de cada variable debe especificarse el tipo de la misma, como en `l: List` y `n: Node` arriba. En lenguaje informal el hecho anterior afirma que, para cualquier lista `l` y nodo `n`, si `n` es alcanzable desde la cabeza – primer elemento– de `l` entonces `n` no puede alcanzarse a partir de si mismo –esto implica que las listas de la especificación deben ser acíclicas.

Notar que, si `x` es una variable de una signatura `S`, y `f` es un campo que representa una función de `S` en `S'`, entonces la composición relacional `x.f` denota la imagen de `x` por `f`. En el ejemplo, `l.head` permite obtener la cabeza de la lista, y `n.next` el nodo que le sigue a `n`. Esta notación es idéntica que la utilizan los lenguajes orientados a objetos para representar dereferencias de campos de objetos.

Observar también que en Alloy se utiliza la clausura reflexo transitiva, `*`, para expresar propiedades de alcanzabilidad. En el hecho anterior, `l.head.*next` denota el conjunto de nodos alcanzables desde la cabeza de la lista, `l.head`, y `n.next.*next` los nodos alcanzables a partir del sucesor de `n`, `n.next`. □

Alloy también provee mecanismos para abstraer partes de un especificación en funciones y predicados parametrizados. Para modelar funciones y/o procedimientos de un lenguaje de programación en Alloy, es necesario modelar los cambios de estado que estos producen. Como Alloy no incorpora la noción de cambio de estado, esta se simula agregando a la definición de las funciones y/o predicados Alloy un conjunto de variables primadas adicional. De esta manera, las variables sin primar representan el estado anterior a la ejecución, y las primadas el estado posterior.

Ejemplo 2.1.3. El predicado `delFirst` a continuación modela una procedimiento que elimina el primer nodo de la lista pasada como parámetro (`l`). `l'` representa el estado de la lista `l` después de la ejecución de `delFirst`.

```
pred delFirst[l,l': List] {
    l.head != null &&
    l'.head = l.head.next
}
```

Notar que la subfórmula `l.head != null` puede pensarse como la precondición de `delFirst`: para poder ejecutar `delFirst` la lista de entrada no debe ser vacía. Por otra parte, `l'.head = l.head.next` describe la postcondición de `delFirst`: el nodo que le sigue a la cabeza de la lista `l` es el primer nodo de `l'`.

Por otro lado, `addFirst` modela la operación de agregar un nuevo nodo con el valor `v` al inicio de la lista `l`:

```
pred addFirst[l,l': List, v: Int] {
    some n: Node |
        n not in l.head.*next &&
        n.next = l.head && n.data = v && l'.head = n
}
```

`addFirst` requiere la existencia de un nodo `n` que no pertenezca a `l` (`n not in l.head.*next`). Entonces, `addFirst` almacena `v` en `n` (`n.data = v`), apunta el campo `next` de `v` al nodo que era anteriormente la cabeza de la lista (`n.next = l.head`), y coloca a `n` como la nueva cabeza (`l'.head = n`). □

2.1.2. Análisis automático basado en SAT

Como se mencionó anteriormente, las especificaciones Alloy pueden ser analizadas automáticamente utilizando el Alloy Analyzer. Para habilitar el análisis el usuario debe proveer límites en la cantidad máxima de elementos de cada signatura (scopes). Usando los scopes el Alloy Analyzer convierte la especificación Alloy en una fórmula proposicional equivalente (KodKod participa como lenguaje intermedio en esta traducción, ver sección 2.4). Luego, el Alloy Analyzer usa esta fórmula como entrada de un SAT solver para averiguar sobre la satisfactibilidad de la fórmula. Si el SAT solver retorna una asignación que hace válida a la fórmula proposicional, el Alloy Analyzer convierte esta asignación en una instancia de la especificación Alloy, y se la muestra al usuario. En caso contrario, el Alloy Analyzer informa al usuario que no existen instancias de la especificación dentro de los scopes provistos.

Alloy posee dos comandos para analizar especificaciones: `run` y `check`. Dados una propiedad y scopes para las signaturas, el primero permite obtener instancias de la especificación (dentro de los scopes) que cumplen con la propiedad, mientras que el segundo permite verificar la especificación acotada por los scopes satisface la propiedad.

Ejemplo 2.1.4. Para obtener una instancia válida de la especificación de listas de esta sección, se agrega a la especificación:

```
pred show[] { }

run show for 1 List, 3 Node
```

Este comando retornará una lista acíclica con a lo sumo tres nodos, dado que show no impone restricciones adicionales sobre el modelo. Una de las posibles instancias (relacional) de esta especificación se muestra en la figura 2.1. La representación gráfica de la misma se exhibe en la figura 2.2. □

$$
\begin{aligned}
\texttt{List} &= \{L_0\} \\
\texttt{Node} &= \{N_0, N_1, N_2\} \\
\texttt{head} &= \{(L_0, N_0)\} \\
\texttt{next} &= \{(N_0, N_1), (N_1, N_2), (N_2, null)\} \\
\texttt{data} &= \{(N_0, I_0), (N_1, I_1), (N_2, I_2)\}
\end{aligned}
$$

Figura 2.1: Una instancia particular (relacional) de la especificación Alloy de listas enlazadas acíclicas

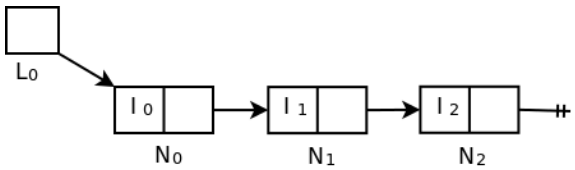


Figura 2.2: Representación gráfica de la instancia de la figura 2.1

Ejemplo 2.1.5. Para verificar (acotadamente) la propiedad `prop1` definida a continuación, se agrega a la especificación Alloy el siguiente fragmento:

```
pred empty[l: List] {
    l.head = Null
}

pred prop1[] {
    all l,l',l'': List, v: Int |
      (empty[l] && addFirst[l,l',v] && delFirst[l',l''])
      => empty[l'']
}

check prop1 for 5
```

`prop1` postula que si se comienza con una lista vacía, a la cual se le inserta un nodo al inicio mediante `addFirst`, y luego se elimina dicho nodo, se obtiene como resultado una lista vacía. Así, ejecutar el Alloy Analyzer con el comando `check prop1 for 5` (scope 5 tanto para List como para Node) da como resultado que no hay contraejemplos de `prop1` dentro de los scopes analizados. Es importante destacar que esto no significa que la propiedad sea válida en general, ya que podrían existir instancias de la especificación con más listas y/o nodos que violen `prop1` (no es el caso para `prop1`). □

2.2. DynAlloy

DynAlloy es una extensión de Alloy para facilitar el modelado del comportamiento dinámico de los programas [22,23]. DynAlloy introduce el concepto de acción atómica: la construcción básica para modelar el cambio de estados intrínseco a los programas imperativos. Además, DynAlloy permite combinar las acciones atomáticas mediante operadores de composición secuencial, elección no determinística e iteración, para modelar programas complejos.

2.2.1. El lenguaje

Una *acción atómica* está compuestas de una precondición y una postcondición definidas en el lenguaje Alloy. Intuitivamente, la precondición explicita las condiciones iniciales requeridas para que la acción pueda ejecutarse exitosamente; la postcondición describe el resultado esperable la ejecución de la acción, si se comienza en un estado que satisface la precondición.

Ejemplo 2.2.1. Para modelar la eliminación e inserción a la cabeza de una lista simplemente encadenada en DynAlloy se definen las acciones:

```
act delFirst[l: List] {
  pre { l.head != null }
  post { l'.head = l.head.next }
}

act addFirst[l: List, v: Int] {
  pre { some n: Node | n not in l.head.*next }
  post { n.data = v && n.next = l.head && l'.head = n }
}
```

Las acciones anteriores son equivalentes a los predicados Alloy homónimos `delFirst` y `addFirst` del ejemplo 2.1.3. La parte estructural de las especificaciones DynAlloy se define en términos de signaturas Alloy. Notar que las variables primadas están definidas implícitamente en las acciones DynAlloy, por lo que no es necesario agregarlas a los parámetros de la acción. En el ejemplo anterior, se asumen las signaturas `List` y `Node` del ejemplo 2.1.1. □

La semántica de una acción atómica se corresponde con (una versión acotada de) la semántica tradicional de las ternas de Hoare [30]: la ejecución de una acción en un estado que satisface su precondición resulta en un estado que satisface la postcondición de la acción.

Por otra parte, como se mencionó anteriormente, DynAlloy provee operadores para combinar acciones atómicas y construir programas. Estos son: la composición secuencial, *;*, la elección no determinística, *+*, y la iteración acotada ***. Además, DynAlloy permite agregar *aserciones* a los programas. En DynAlloy, una aserción `P` es una propiedad que se debe cumplir en cierto punto del programa para que la ejecución continue, y se especifica como `assume P`.

Ejemplo 2.2.2. El predicado Alloy `prop1` del ejemplo 2.1.5 afirma que partiendo de una lista vacía, si se inserta un elemento al inicio de la lista y luego se elimina la cabeza de la lista, se obtiene nuevamente una lista vacía. Una especificación equivalente de `prop1` como un programa DynAlloy se muestra a continuación:

```
assertCorrectness prop1[l: List ,v: Int] {
  pre = { empty[l] }
  program = { addFirst[l,v] ; delFirst[l] }
  post = { empty[l'] }
}
```

Notar que las "aserciones de corrección" (`assertCorrectness`) –como `prop1` anterior– son similares a las ternas de Hoare, ya que se componen de una precondición (`pre`),

un programa (`program`) y una postcondición (`post`). La aserción `prop1` afirma que si se parte de una lista vacía `l`, y se ejecutan secuencialmente las acciones `addFirst` y `delFirst` sobre `l`, se obtiene una lista `l'` vacía. □

Una comparación directa entre las ejemplos 2.1.5 y 2.2.2 ilustra la simplicidad con la que pueden especificarse propiedades de ejecución de programas en DynAlloy, respecto de su contraparte Alloy.

Otra ventaja de DynAlloy es que permite modelar de manera sencilla la ejecución repetida de código (ciclos), gracias al operador `*`. Como el análisis de las especificaciones DynAlloy es acotado (sección 2.2.2), se requiere del usuario una cota superior en la cantidad máxima de ejecuciones de las iteraciones.

Ejemplo 2.2.3. Para postular que cualquier cantidad de ejecuciones de `addFirst` seguidas de un número arbitrario de ejecuciones de `delFirst` mantienen la vacuidad de una lista, se define el siguiente programa DynAlloy:

```
assertCorrectness prop2[l: List ,n: Node] {
  pre = { empty[l] }
  program = { addFirst[l,n]* ; delFirst[l]* }
  post = { empty[l] }
}
```

Analizando la aserción anterior con 5 como scope y cantidad máxima de iteraciones, DynAlloy retorna un contraejemplo en el que `addFirst` se ejecuta dos veces, y `delFirst` ninguna, resultando en una lista con dos elementos en el estado final. □

Por último, es importante destacar que Alloy presenta dificultades para modelar iteración. Para simular la iteración en Alloy es necesario agregar signaturas que definan el concepto de trazas de ejecución manualmente, en cada especificación a analizar [22]. En cambio, DynAlloy soluciona este problema en forma general, simple y concisa, con los conceptos de acciones atómicas y composición de acciones.

2.2.2. Análisis automático basado en SAT

Naturalmente, la semántica de las acciones y aserciones de corrección DynAlloy está basada en ternas de Hoare. [22] detalla el procedimiento de traducción de estas ternas a Alloy, que consiste en utilizar una adaptación del transformador de predicados WLP (Weakest Liberal Precondition) – una versión acotada del famoso WP de Dijsktra [18] – para generar fórmulas relacionales. De esta manera, el proceso de análisis de una especificación DynAlloy comprende una traducción de las extensiones provistas por el lenguaje a fórmulas Alloy equivalentes (usando el WLP anterior).

Por otra parte, como la parte estructural del modelo DynAlloy (signaturas, hechos, predicados, etc.) es esencialmente una especificación Alloy, la traducción realiza una copia del fragmento estructural a la especificación Alloy a generar.

La herramienta implementa la traducción de DynAlloy a Alloy se denomina DynAlloy Translator, y se encuentra disponible para descargar en [2]. Una descripción minuciosa del procedimiento de traducción está fuera de los alcances de esta tesis, por lo que se remite al lector interesado a [22].

Otra ventaja de DynAlloy sobre Alloy es que –como se muestra en los experimentos de [23]– el análisis de especificaciones con propiedades dinámicas en DynAlloy es más eficiente que el análisis de especificaciones Alloy equivalentes.

2.3. Analizando JAVA con DynAlloy

En [24] se propone un método de análisis de programas JAVA que consiste en codificar el programa y su especificación JML en DynAlloy, y verificar su corrección (acotada) a través del DynAlloy Translator. Este método subyace al análisis de código basado en SAT usado en esta tesis. En esta sección, se presentarán los aspectos principales de la traducción utilizando como ejemplo el código JAVA de la figura 2.3.

Ejemplo 2.3.1. El código de la figura 2.3 muestra un fragmento de una implementación JAVA de listas simplemente encadenadas acíclicas. La clase `Node` implementa los nodos de las listas, que poseen dos campos: `data` para almacenar valores, y `next` para referenciar al siguiente nodo en la lista. La clase `List` implementa las listas propiamente dichas, que tienen una referencia al primer nodo de la lista en su campo `head`. El método `addFirst` de `List` crea un nuevo nodo con valor `v` y lo agrega al inicio de la lista. `delFirst` simplemente elimina el primer nodo de la lista. Además, la clase `List` posee un invariante de aciclicidad descrito en el lenguaje JML [12]. El invariante postula que, para todo nodo `n` alcanzable desde el primer nodo de la lista, `n` no puede volver a alcanzarse a partir de si mismo a través del campo `next`. Esto implica que las listas deben ser acíclicas. □

La traducción de JML a Alloy está fuera de los alcances de esta tesis. Una descripción detallada de la misma puede encontrarse en [26].

2.3.1. Modelo del heap JAVA

El heap de JAVA se modela en DynAlloy siguiendo el enfoque de [34]: se define una signatura por cada clase, y una relación binaria por cada uno de los campos de las clases. Así, de la clase `C` a continuación:

```
/*@ invariant
  @ \forall Node n; \reach(head, Node, next).has(n) =>
  @                 ! \reach(head.next, Node, next).has(n);
  @*/
class List {
    Node head;

    void addFirst(v: Int) {
        Node n = new Node();
        n.data = v;
        n.next = this.head;
        this.head = n
    }

    void delFirst() {
        this.head = this.head.next
    }
    ...
}

class Node {
    int data;
    Node next;
    ...
}
```

Figura 2.3: Fragmento de una implementación JAVA de listas simplemente encadenadas acíclicas.

```
class C {
  f_1: T_1
  ...
  f_n: T_n
  ...
}
```

se derivan una signatura `C`, y los campos Alloy:

```
f_1: C -> one T_1
```

```
...
f_n: C -> one T_n
```

Ejemplo 2.3.2. El modelo del heap para el programa JAVA de la figura 2.3 consta de las signaturas:

```
one sig null { }
sig List { }
sig Node { }
```

y los campos:

```
head: List -> one (Node+null)
next: Node -> one (Node+null)
data: Node -> one Int
```

Los campos JAVA son funciones totales. Esto se modela usando el modificador `one` en la definición de los campos de las signaturas. □

Ejemplo 2.3.3. El invariante JML de la clase `List` predica sobre el heap. Su traducción a Alloy resulta en la misma fórmula contenida en el hecho `acyclic` del ejemplo 2.1.2. Sin embargo, es conveniente definirlo en Alloy como predicado en lugar de hecho, por ejemplo, para permitir verificar si los métodos de la clase `List` lo mantienen.

```
pred listInvariant[this: Node,
                   head: List -> one (Node+null),
                   next: Node -> one (Node+null)] {
  all n:Node | n in this.head.*next => n !in n.next.*next
}
```

□

2.3.2. Traducción de comandos JAVA a DynAlloy

En esta sección se presenta la traducción de comandos JAVA a DynAlloy propuesta en [24]. Para la traducción se asume que el programa no tiene errores de tipo (es decir, que fue analizado previamente por un compilador). Se definirá la traducción como una función τ que para cada comando JAVA produce la fórmula DynAlloy correspondiente.

La forma más sencilla del comando de asignación, `v=e`, con `v` una variable y `e` una expresión, se traduce como una invocación a la acción DynAlloy `VarAssign`, que asigna a `v` el resultado de evaluar `e`:

```
act VarAssign[v,e: T] {
    pre {  }
    post { v' = e }
}
```

Es decir, τ(`v=e`)$\doteq$`VarAssign(v,e)`.

La codificación del comando `new T` de JAVA involucra reservar memoria para un nuevo objeto de tipo `T` en el heap. Por lo tanto, para cada clase `T` se define en DynAlloy un conjunto `objs_T`, que almacena los objetos alocados del tipo correspondiente. Los conjuntos `objs_T` se asumen globales, con motivo de simplificar la presentación. Así, τ(`v= new T`)$\doteq$`NewT(v)`, donde:

```
act NewT[v: T] {
    pre {  }
    post {
      some o: T |
        o not in objs_T && v' = o
    }
}
```

Además, por cada campo `f: T -> one T'` se define una acción `Setf`, para modificar el valor de `f` en un objeto particular `o`:

```
act SetF[o: T, e: T', f: T -> one T'] {
    pre { o in objs_T }
    post { f' = f ++ (o->e) }
}
```

En Alloy, `o->e` denota al par ordenado (o, e). Además, si `R` es una relación binaria, el *relational override* `R ++ (o->e)` retornal una relación que es idéntica a `R`, excepto para `o`, que sólo está asociado con `e` en el resultado. De esta manera, la asignación de la expresión `e` al campo `f` del objeto `o` se traduce como: τ(`o.f = e`)$\doteq$`SetF[o,e,f]`.

Por último, la codificación de los comandos JAVA de composición secuencial, alternativa e iteración en DynAlloy es directa, y se muestra a continuación:

```
τ(st1;st2)
       -> τ(st1);τ(st2)
τ(if (P) st1 else st2)
       -> (assume τ(P);τ(st1));(assume !τ(P);τ(st2))
τ(while (P) st)
       -> (assume τ(P);τ(st)*);(assume !τ(P))
```

Ejemplo 2.3.4. Con los conceptos vistos hasta el momento, la traducción del método JAVA `addFirst`, de la figura 2.3 se muestra a continuación:

```
program addFirst[this: Node,
                 head: List -> one (Node+Null),
                 next: Node -> one (Node+Null),
                 data: Node -> one Int,
                 n: Node,
                 v: Int] {
    NewNode[n];
    SetData[n, v];
    SetNext[n, next, this.head];
    SetHead[this, head, n]
}
```

En el programa DynAlloy anterior, los parámetros `this`, `head`, `next` y `data` representan el estado del heap, y los parámetros `n` y `v` el estado de las variables homónimas. □

2.3.3. Análisis de propiedades del código JAVA

Como ejemplo de verificación de un programa JAVA en DynAlloy, construyamos una aserción DynAlloy para averiguar si el método `addFirst` mantiene el invariante de aciclicidad de listas, `listInvariant`:

```
assertCorrectness
    addFirst-mantains-inv[this: Node,
                          head: List -> one (Node+Null),
                          next: Node -> one (Node+Null),
                          data: Node -> one Int,
                          n: Node] {
    pre = { listInvariant[this, head, next] }
    program = { call addFirst[this, head, next, data, n] }
    post = { listInvariant[this', head', next'] }
}
```

La palabra clave `call` permite invocar programas DynAlloy. Su implementación actual realiza una copia del cuerpo del programa con sus parámetros formales reemplazados por los parámetros actuales (proceso que se conoce como *inlining*). Claramente, `addFirst-mantains-inv` es una aserción válida.

La figura 2.4 muestra la traducción completa del código JAVA de la figura 2.3, incluyendo la aserción anterior.

```
one sig null { }
sig List { }
sig Node { }
pred listInvariant[ ... ] {
    all n:Node | n in this.head.*next =>
                   n !in n.next.*next
}
program addFirst[ ... ] {
      NewNode[n];
      SetData[n, v];
      SetNext[n, next, this.head];
      SetHead[this, head, n]
}
program delFirst[ ... ] {
    setHead[this, head, this.head.next]
}
assertCorrectness
    addFirst-mantains-inv[this: Node,
                          head: List -> one (Node+Null),
                          next: Node -> one (Node+Null),
                          data: Node -> one Int,
                          n: Node,
                          v: Int] {
    pre = { listInvariant[this, head, next] }
    program = { call addFirst[this, head, next, data, v] }
    post = { listInvariant[this', head', next'] }
}
```

Figura 2.4: Especificación DynAlloy para verificar que el método `addFirst` del código JAVA de la figura 2.3 mantiene el invariante `listInvariant`.

2.4. KodKod

KodKod [45] es un procedimiento de decisión para la lógica relacional. Como se mencionó en la sección 2.1, está lógica es la base del lenguaje Alloy. Dado un conjunto de fórmulas de la lógica relacional la implementación de KodKod genera una fórmula proposicional equivalente, que luego analiza utilizando un SAT solver externo. Además, KodKod es capaz de convertir las asignaciones (de variables proposicionales a valores de verdad) que satisfacen a la fórmula proposicional –retornadas por el SAT solver– en instancias del conjunto de fórmulas relacionales de origen.

El propósito de KodKod es implementar la traducción descrita anteriormente de manera eficiente, es decir, utilizando la menor cantidad posible de variables proposicionales. Como el problema de buscar una asignación que satisface una fórmula proposicional (SAT para abreviar) es NP-completo [15], es primordial generar representaciones tan compactas como sea posible de las fórmulas relacionales.

La implementación corriente del Alloy Analyzer [1] usa a KodKod como lenguaje intermedio en la traducción de Alloy a una fórmula proposicional analizable vía SAT, por lo que la eficiencia del análisis de las especificaciones Alloy depende fuertemente de KodKod. Para los propósitos de esta tesis KodKod cumple un rol aún más importante: las cotas ajustadas computadas por los algoritmos presentados en este trabajo serán utilizadas por KodKod durante el análisis de código basado en SAT, con el fin de reducir el tamaño de la fórmula proposicional que codifica el programa bajo análisis. Así, las características más importante de KodKod para los propósitos de este trabajo son: la forma de codificar los campos Alloy en fórmulas proposicionales; y la capacidad de KodKod de aprovechar *instancias parciales* de relaciones: el mecanismo que habilita el uso de cotas ajustadas durante el análisis de código basado en SAT.

2.4.1. Representación de campos Alloy

Sean n y m scopes para las signaturas `A` y `B`, es decir, `A`= $\{A_0, \ldots, A_{n-1}\}$ y `B`= $\{B_0, \ldots, B_{m-1}\}$. Durante del proceso de traducción de Alloy a una fórmula proposicional, KodKod representa una relación `f: A -> (B+null)` –derivada de un campo Alloy `f`– como una matriz de variables proposicionales M_f:

M_f	B_0	B_1	...	B_{m-1}	$null$
A_0	p_{A_0,B_0}	p_{A_0,B_1}	...	$p_{A_0,B_{m-1}}$	$p_{A_0,null}$
A_1	p_{A_1,B_0}	p_{A_1,B_1}	...	$p_{A_1,B_{m-1}}$	$p_{A_1,null}$
...	...	...	...	...	...
A_{n-1}	p_{A_{n-1},B_0}	p_{A_{n-1},B_1}	...	$p_{A_{n-1},B_{m-1}}$	$p_{A_{n-1},null}$

Cuadro 2.1: Representación la relación `f: A -> (B+null)` como matriz de variables proposicionales

De esta manera, el hecho de que el valor del campo f del objeto A_i sea B_j ($A_i.f = B_j$) en una instancia particular de la especificación Alloy de origen queda representado en la fórmula proposicional por la variable p_{A_i,B_j}. En otras palabras, $A_i.f = B_j$ en una instancia de la especificación Alloy si y sólo si $p_{A_i,B_j} = true$.

Ejemplo 2.4.1. La matriz correspondiente al campo `next` del modelo DynAlloy de listas simplemente encadenadas (figura 2.4), con scope 3 para `Node` (`Node`= $\{N_0, N_1, N_2\}$), se muestra a continuación:

M_{next}	N_0	N_1	N_2	$null$
N_0	p_{N_0,N_0}	p_{N_0,N_1}	p_{N_0,N_2}	$p_{N_0,null}$
N_1	p_{N_1,N_0}	p_{N_1,N_1}	p_{N_1,N_2}	$p_{N_1,null}$
N_2	p_{N_2,N_0}	p_{N_2,N_1}	p_{N_2,N_2}	$p_{N_2,null}$

Cuadro 2.2: Representación KodKod del campo `next` de la especificación de listas simplemente encadenadas (figura 2.4), con scope 3 para `Node`

Adicionalmente, el campo `next` de la instancia de listas de la figura 2.2 se obtiene asignando el valor verdadero a las variables p_{N_0,N_1},p_{N_1,N_2},$p_{N_2,null}$ de M_{next}, y falso a todas las variables restantes. □

2.4.2. Instancias parciales

Como se mencionó anteriormente, el mecanismo de instancias parciales de KodKod permite el uso de cotas ajustadas durante el análisis de código basado en SAT. KodKod soporta dos tipos diferentes de instancias parciales: cotas superiores (*upper bounds*), y cotas inferiores (*lower bounds*). En esta tesis trabajaremos sólo con las primeras, el lector interesado puede consultar [45] para más información sobre las cotas inferiores. Una *cota superior* U_f para una relación $f : A \times B$ es una relación del mismo tipo de f ($U_f : A \times B$), que KodKod usa para limitar el conjunto de valores que puede tomar f. La idea es que, dados $a \in A$, $b \in B$, si $(a, b) \notin U_f$, entonces $(a, b) \in f$ no puede ocurrir en ninguna instancia del modelo KodKod. Para lograr esto, KodKod elimina de la matriz M_f todas las variables p_{N_i,N_j} que representan pares (N_i, N_j) que no pertenecen a U_f (más precisamente, les asigna el valor falso). Consecuentemente, el uso de cotas superiores permite a KodKod codificar el campo acotado en una fórmula proposicional de menor tamaño.

Ejemplo 2.4.2. Volviendo al ejemplo de listas simplemente encadenadas (figura 2.4) con scope 3 para `Node`. A continuación se muestra una cota ajustada para el campo `next` de listas simplemente encadenadas (ver sección 2.7.1, figura 2.8):

$$U_{next} = \{(N_0, N_1), (N_1, N_2), (N_0, null), (N_1, null), (N_2, null)\}$$

Usando U_{next} como cota superior KodKod la matriz M_{next} de la figura 2.2, ejemplo 2.4.1, se obtiene:

Así, la cota superior (ajustada) U_{next} permite que la codificación del campo `next` requiera de cinco variables proposicionales en lugar de doce (ver figura 2.2). □

M_{next}	N_0	N_1	N_2	$null$
N_0	$false$	p_{N_0,N_1}	$false$	$p_{N_0,null}$
N_1	$false$	$false$	p_{N_1,N_2}	$p_{N_1,null}$
N_2	$false$	$false$	$false$	$p_{N_2,null}$

Cuadro 2.3: Representación KodKod del campo `next` de la especificación de listas simplemente encadenadas (figura 2.4) para scope 3, usando U_{next} como cota superior

2.5. Análisis de código basado en satisfactibilidad booleana

El objetivo principal de esta tesis es definir técnicas para optimizar el análisis acotado y exhaustivo de código, más específicamente, el análisis basado en satisfactibilidad booleana (SAT para abreviar). Los análisis basados en SAT se dicen *acotados* porque requieren de límites superiores en la cantidad de elementos que pueden contener los dominios de datos del programa –denominados *scopes* en la literatura–, y en la cantidad máxima de iteraciones de los ciclos. Además, son *exhaustivos* porque exploran cada una de las posibles ejecuciones del programa enmarcadas dentro de estos límites, con la intención de encontrar errores: violaciones de la especificación del programa, excepciones en tiempo de ejecución, etc.

En general, los análisis basados en SAT convierten el programa acotado y su especificación (si existe) en una fórmula de la lógica proposicional. La conversión se hace de manera que las asignaciones (de valores de verdad a variables) que hacen a la fórmula verdadera representan a errores del programa. Luego, esta fórmula se usa como entrada de un *SAT solver* externo, un programa que busca exhaustivamente asignaciones que satisfacen su fórmula de entrada. Una tal asignación, en caso de existir, es transformada por el análisis en una ejecución concreta del programa, y presentada al usuario como *testigo* del error revelado.

Una de las principales ventajas de los análisis basados en SAT es su automaticidad: no requieren ayuda por parte del usuario para buscar errores en el código. Por el contrario, una de sus desventajas es que sólo pueden asegurar la ausencia de errores dentro de los límites dados. Por ejemplo, un programa que manipula listas podría contener un error detectable con entradas con más de 10 nodos, que pasaría inadvertido si el análisis se ejecuta con un scope para los nodos menor que 10. Resumiendo, los análisis basados en SAT no son capaces de probar la ausencia de errores en los programas. De todos modos, su utilidad práctica reside en su capacidad de detección de errores de manera completamente automática, sobre todo para programas con especificaciones complejas, que otros formalismos no son capaces de manejar. Por

ejemplo, las técnicas de verificación automática tienen problemas para tratar con programas que manipulan estructuras de datos dinámicas, que a su vez requieren de especificaciones intrincadas, como pueden ser los invariantes de clase de los árboles AVLs, árboles rojos y negros, etc. (ver [17, 27]).

2.5.1. TACO: un análisis de código basado en SAT

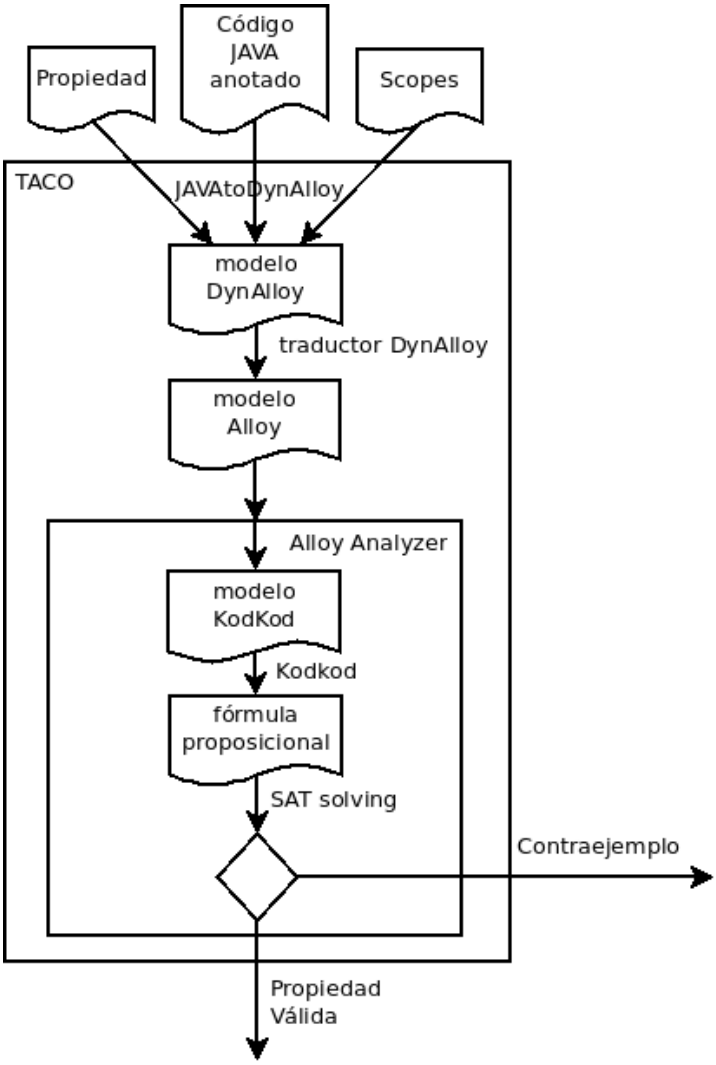


Figura 2.5: Etapas principales del análisis de código TACO.

El punto de partida de esta tesis es el análisis basado en SAT presentado en [27],

denominado TACO. Este análisis permite realizar la verificación *modular* (acotada) de un programa, es decir, verificar que todos los posibles comportamientos (acotados) del código satisfacen su especificación, sin la necesidad de tener en cuenta la forma en que sus clientes lo invocan. Al igual que los análisis basados en SAT descritos en la sección anterior, este análisis es completamente automático. La implementación de este análisis, provista por los autores de [27], soporta código fuente en el lenguaje de programación orientado a objetos JAVA, y JML [12] como lenguaje para expresar las especificaciones. Cuatro razones fundamentan la elección de este análisis: su amplio poder expresivo, lo que nos habilita para especificar propiedades de estructuras de datos complejas; el soporte brindado a un lenguaje de programación moderno, como lo es JAVA; su capacidad de aprovechar cotas ajustadas; y, en conjunción con las cotas ajustadas, se ha mostrado más eficiente que otros análisis competidores [27].

La figura 2.5 exhibe un diagrama con las etapas principales del análisis de TACO: desde la conversión de un programa JAVA a una fórmula proposicional, hasta el análisis de esta última usando un SAT solver. Las entradas de TACO son: el código JAVA a analizar, con especificaciones JML para sus métodos; los límites en la cantidad de objetos por cada clase y el número máximo de ejecuciones de los ciclos (llamados scopes en la figura); y la propiedad a verificar.

Ejemplo 2.5.1. Como ejemplo de entradas de TACO consideremos la figura 2.3 de la sección 2.3. Esta muestra una implementación JAVA de listas simplemente encadenadas, con dos métodos `addFirst` y `delFirst`, y un invariante JML para la clase `List`. El invariante postula que las instancias de `List` son acíclicas. Un ejemplo de propiedad a verificar en este caso consistiría en garantizar que el método `addFirst` mantiene el invariante de aciclicidad. Esto es, cada vez que se ejecuta `addFirst` en un estado inicial que satisface el invariante se concluye en un estado que también cumple con el invariante. □

Como puede verse en la figura 2.5, la primera etapa de TACO involucra en traducir el código a analizar, junto con sus respectivas anotaciones y la propiedad, a DynAlloy (teniendo en cuenta los scopes). Este paso se implementa usando el traductor de DynAlloy, siguiendo las ideas de la sección 2.3.

Ejemplo 2.5.2. Para las entradas del ejemplo 2.5.1, este paso de traducción de JAVA a DynAlloy produce la especificación de la figura 2.4. □

La siguiente etapa consiste en analizar la especificación DynAlloy que codifica el programa JAVA, obtenida en el paso anterior. Como se explicó en la sección 2.2, el análisis de especificaciones DynAlloy involucra traducir las especificificaciones a Alloy. Este procedimiento es implementado por la herramienta DynAlloy translator

[2]. El resultado de esta etapa es una especificación Alloy equivalente al programa JAVA de entrada.

Naturalmente, el paso siguiente es analizar la especificación Alloy obtenida usando el Alloy Analyzer. Los pasos incluídos en esta etapa se grafican en el recuadro interno de la figura 2.5. Como se discutió en la sección 2.4, el análisis implementado por el Alloy Analyzer consiste en producir un conjunto de fórmulas relaciones en lenguaje KodKod, que este último utilizará para generar una fórmula proposicional. En este punto, es importante destacar que es KodKod la herramienta que permite aprovechar las cotas ajustadas computadas para los campos; esto se discute más adelante en la sección 2.4.

El último paso del análisis es utilizar un SAT solver para buscar exhaustivamente asignaciones que satisfagan la fórmula proposicional generada. Una asignación de este tipo, si existe, corresponde a una violación de la especificación del programa JAVA de entrada. En este caso, TACO convierte estas asignaciones en ejecuciones del programa antes de mostrarlas al usuario como contraejemplo de la propiedad dada (esta última conversión se omite en la figura 2.5). En caso contrario, TACO informa al usuario que el programa no tiene errores para los scopes considerados.

Más adelante se discutirá la interacción entre el análisis de TACO y los algoritmos secuenciales de cómputo de cotas ajustadas definidos en este trabajo.

2.6. Canonización del heap

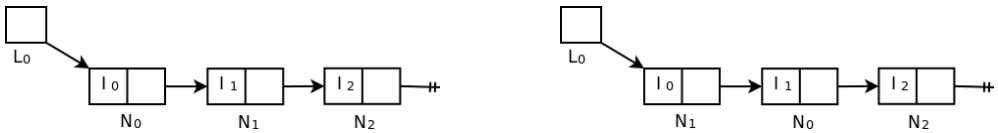


Figura 2.6: Dos listas simplemente encadenadas isomorfas

Un problema del modelo del heap presentado en la sección 2.3 es que admite estructuras *isomorfas*, es decir, que diferentes instancias del heap pueden denotar un mismo elemento de una estructura de datos.

Ejemplo 2.6.1. Consideremos la definición de los campos de un heap para el tipo de datos listas simplemente encadenadas:

```
next: Node -> one (Node+Null)
data: Node -> one Int
```

Asumamos un scope 3, e identificadores de nodos N_0, N_1, N_2. Notar que, para una única asignación de valores a nodos (`data` fijo), todas las listas cuyos nodos conforman una permutación válida de N_0, N_1, N_2 son instancias válidas del heap. A modo ilustrativo, la figura 2.6 exhibe dos de estas instancias. □

En [27] se propone *canonizar* el heap para eliminar instancias isomorfas, instrumentado de manera completamente automática las especificaciones (Dyn)Alloy con *predicados de rotura de simetría*. Intuitivamente, una vez llevada a cabo la instrumentación, la lista que se muestra a la izquierda en la figura 2.6 se convierte en la representación canónica de las listas con tres nodos, y todas las isomorfas a esta (entre las que se encuentra la restante de la figura 2.6) dejan de ser instancias válidas del heap. Los resultados experimentales presentados en [25] indican la canonización del heap vía predicados de rotura de simetría produce una mejora significativa en la eficiencia del análisis de programas JAVA que manipulan estructuras de datos complejas. Los autores del artículo apuntan a la reducción en la cantidad de instancias válidas, producto de la rotura de simetrías, como la razón principal de la mejora en la performance del análisis. Cabe destacar que el Alloy Analyzer implementa una forma de rotura de simetrías más débil, incapaz de eliminar el tipo de estructuras isomorfas que surgen de la traducción de de programas JAVA a (Dyn)Alloy (como las listas de la figura 2.6).

En la sección que sigue se discuten los predicados de rotura de simetría de [27]. Es importante destacar que, además de los beneficios que traen al análisis basado en SAT, el uso de predicados de rotura de simetrías permite la generación de cotas ajustadas más precisas (ver sección 2.7.3).

2.6.1. Predicados de rotura de simetría en DynAlloy

Los *predicados de rotura de simetría* de [27] son predicados DynAlloy que fijan un orden único posible para la asignación de identificadores a los nodos en las instancias válidas de la especificación DynAlloy (que modelan el heap del programa a analizar).

Definiciones preliminares

Explicaremos estos predicados utilizando un modelo DynAlloy de árboles binarios, que se genera a partir de los extractos de clases JAVA a continuación:

```
class Tree {
  root: Node;
  ...
}
```

```
class Node {
  left: Node;
  right: Node;
  ...
}
```

La anterior es una definición estándar JAVA de árboles binarios. La clase `Tree` define árboles binarios con nodo raíz `root`. `Node` define el tipo de los nodos de los árboles, donde cada nodo posee un campo referenciado a su árbol izquierdo y derecho, `left` y `right`, respectivamente. De las clases anteriores surgen las siguientes signaturas y campos DynAlloy:

```
sig Object { }
sig Tree extends Object { }
sig Node extends Object { }
```

```
root: Tree -> one (Node+Null)
left: Node -> one (Node+Null)
right: Node -> one (Node+Null)
```

Notar que nuestro modelo del heap puede pensarse como un grafo $G = \langle N, E, L, R \rangle$. Los nodos del grafo, N, son los objetos alocados en el heap. $E \subseteq N \times N$ es el conjunto de aristas de G. L es una función que etiqueta las aristas en E con nombres de campos. Así, $N_1.f = N_2$ se representa en el grafo como una arista $(N_1, N_2) \in E$ con etiqueta f ($L(N_1, N_2) = f$). Por último, R es el conjunto de nodos raíces del grafo, esto es, los parámetros del método a analizar.

El primer paso de la instrumentación consiste en explicitar nombres para los elementos de cada signatura. Así, asumiendo scope 1 para `Tree` y 5 para `Node` tenemos:

```
one sig Tree0 extends Tree { }
one sig Node0, Node1, Node2, Node3, Node4 extends Node { }
```

De esta manera, `Tree0` es la única instancia para de `Tree`, y `Node0`, `Node1`, `Node2`, `Node3`, `Node4` son los posibles identificadores para los objetos de la clase `Node`. Además, es necesario inducir un orden sobre los identificadores de cada signatura. En nuestro ejemplo, se define la relación DynAlloy `nextNode`, que ordena los identificadores de `Node` de menor a mayor por su índice:

```
fun nextNode[]: Node -> lone Node {
  Node0->Node1 + Node1->Node2
  + Node2->Node3 + Node3->Node4
}
```

Las funciones auxiliares `minNode` y `prevsNode` permiten extraer el mínimo nodo (de acuerdo al orden anterior) de un conjunto dado, y los nodos menores que un nodo dado, respectivamente.

```
fun minNode[ns: set Node]: lone Node {
    ns - ns.^(nextNode[])
}
fun prevsNode[n : Node]: set Node {
    n.^(~nextNode[])
}
```

En la instrumentación, los campos recursivos (por ejemplo, `left` y `right`) se particionan en dos componentes disjuntas: una que apunta "hacia adelante" (*forward*), es decir, que sólo mapea elementos en otros estrictamente más grandes –de acuerdo al orden impuesto en el dominio (por ejemplo, `nextNode`)–; y otra componente que apunta "hacia atrás" (*backward*), que mapea elementos a otros menores o iguales. En nuestro ejemplo, todas las ocurrencias de `left` y `right` en la especificación DynAlloy se reemplazan por `fleft+bleft` y `fright+bright`, respectivamente, donde:

```
fleft: Node -> lone (Node+Null)
bleft: Node -> lone (Node+Null)
fright: Node -> lone (Node+Null)
bright: Node -> lone (Node+Null)
```

El hecho de que los campos *forward* y *backward* forman particiones de los originales se agrega a la especificación DynAlloy incluyendo la fórmula a continuación:

```
fact  no ((fleft.univ) & (bleft.univ)) &&
       Node = fleft.univ + bleft.univ &&
       no ((fright.univ) & (bright.univ)) &&
       Node = fright.univ + bright.univ
```

En la fórmula anterior, por ejemplo, `no ((fleft.univ) & (bleft.univ))` afirma que los dominios de `fleft` y `bleft` son disjuntos, y `Node = fleft.univ + bleft.univ` implica que la unión de los dominios de `fleft` y `bleft` es igual a `Node`.

Con esta nueva caracterización del heap, el conjunto de elementos alcanzables a partir de los nodos raíz del grafo puede especificarse en términos de los campos que apuntan hacia adelante. En nuestro ejemplo, asumiendo que se analiza un método con un único parámetro `this: Tree`, los nodos alcanzables desde la raíz del heap `this` se definen como:

```
fun FReach[]: set Node
{
  this.*(root + fleft + fright) - null
}
```

Notar que la traducción de `FReach` a una fórmula proposicional requiere de menos variables que su contraparte que usa los campos no particionados (`this.*(root + left + right) - null`).

El enfoque de [27] propone ordenar los elementos del heap mirando a sus padres, en la representación del heap como un grafo. En particular, para determinar el orden de un elemento dado sólo se debe tener en cuenta el *mínimo padre* dentro de un orden global de identificadores, `globalMin`. En nuestro ejemplo de árboles binarios, `globalMin` se define como:

```
fun globalMin[s: set Object]: Object {
  s - s.^(Tree0->Node0 + Node0->Node1 +
   Node2->Node3 + Node3->Node4)
}
```

Básicamente, `globalMin` indica que el único objeto de la clase `Tree` es anterior en el orden global a todos los objetos de `Node`, para los que respeta el orden impuesto por `nextNode`. De esta manera, la función `minP`, que devuelve el mínimo padre de un nodo según el orden de `globalMin`, se define como:

```
fun minP[o: Object]: Object {
  globalMin[(root + fleft + fright).o]
}
```

Predicados de rotura de simetría

Con las deficiones anteriores, estamos listos para definir predicados de rotura de simetría para DynAlloy. Estos predicados obligan un orden de asignación de identificadores a los elementos del heap n_1 y n_2 analizando sus padres $minP[n_1]$ y $minP[n_2]$. Para los elementos de una signatura particular T, los predicados de rotura de simetría imponen sobre el heap las restricciones listadas a continuación. Notar que los nodos raíces –los argumentos del método a analizar– no poseen padres, por lo que se ordenan entre sí con independencia del resto de los elementos del heap.

1. Los nodos raíces (no tienen padre) reciben identificadores en el orden de aparicón de los parámetros correspondientes en la definición del método a analizar. Es decir, si n_1 y n_2 son raíces, y n_1 aparece antes que n_2 en la lista de parámetros, n_1 debe tener un identificador menor que el de n_2 (en el orden propio de la signatura).

2. Si n_1 es un nodo raíz y n_2 no lo es, n_1 debe tener un identificador menor que el de n_2.

3. Sean $n_1, \ldots, n_i$ tales que $\mathbf{minP}[n_1] = ... = \mathbf{minP}[n_i] = n$ (no son raíces), entonces existen campos $f_1, ..., f_i$ tal que $n.f_1 = n_1, ..., n.f_i = n_i$. $n_1, \ldots, n_i$ reciben identificadores en el orden en que los campos $f_1, ..., f_i$ fueron declarados en el código JAVA.

4. Sean n_1 y n_2 tales que $\mathbf{minP}[n_1] = n_3$ y $\mathbf{minP}[n_2] = n_4$, con n_3 y n_4 pertenecientes a la misma signatura T', los identificadores de n_1 y n_2 se ordenan de acuerdo al orden de los identificadores de n_3 y n_4 en T'.

5. Sean n_1 y n_2 tales que $\mathbf{minP}[n_1] = n_3$ y $\mathbf{minP}[n_2] = n_4$, con $n_3 \in T'$ y $n_4 \in T''$, $T' \neq T''$, entonces n_1 y n_2 reciben identificadores en el orden en que se definen las clases de las que derivan T' y T'' en el código fuente.

Instrumentando el modelo DynAlloy con hechos que codifiquen las reglas anteriores para cada signatura se logra canonizar el heap. Es fácil ver que la canonización determinada por las reglas anteriores puede implementarse de forma completamente automática.

Ejemplo 2.6.2. Retomemos el ejemplo de árboles binarios. En este caso de estudio, la única raíz es de tipo `Tree` –un conjunto unitario–, por lo que no se introducen nuevos hechos para esta signatura.

Por otra parte, los elementos de la signatura `Node` no pueden ser raíces, por lo que no agregan hechos para los puntos 1 y 2. En cambio, el caso 3 se implementa mediante el siguiente hecho:

```
fact condicionDeOrdenNode3[] {
  all disj n1, n2: Node | (n1+n2 in FReach[] =>
    ((some n: Node | n = minP[n1] && n = minP[n2]
                     && n1 = n.fleft && n2 = n.fright)
      => n2 = n1.nextNode[]))
}
```

esto es, si dos nodos distintos `n1` y `n2` (el modificador `disj` luego del cuantificador `all` asegura que `n1` y `n2` sean diferentes nodos) comparten un mismo "mínimo padre" `n` a través de los campos `left` y `right`, respectivamente, entonces el identificador de `n2` debe ser el que le sigue en el orden `nextNode` al identificador de `n1`. Siguiendo con el punto 4, se deben ordenar los elementos de tipo `Node` con padres en `Node`. Así, si `n1` y `n2` son nodos distintos con padres `n3` y `n4`, respectivamente, si el identificador de `n3` es menor que el de `n4` (`n3 in prevsNode[n4]`), entonces el identificador de `n1` debe ser menor que el de `n2` (`n1 in prevsNode[n2]`). La instrumentación de este punto se muestra abajo:

```
fact condicionDeOrdenNode4[] {
  all disj n1, n2: Node | (n1+n2 in FReach[] =>
    ((some disj n3, n4: Node |
      n3 = minP[n1] && n4 = minP[n2] && n3 in prevsNode[n4])
    => n1 in prevsNode[n2])
}
```

Sólo resta ordenar los elementos de `Node` con padres en `Tree` respecto de elementos de `Node` con padres en la misma signatura. Según la regla 5 listada arriba, los primeros siempre deben recibir identificadores menores que los últimos, de lo que resulta el siguiente hecho:

```
fact condicionDeOrdenNode5[] {
  all disj n1, n2: Node | (n1+n2 in FReach[] =>
    ((some a: Tree, b: Node | a = minP[n1] && b = minP[n2])
    => n1 in prevsNode[n2]))
}
```

□

Además, se instrumenta la especificación DynAlloy con el hecho `compactarNode`, para asegurar que no quedan "huecos" en la asignación de identificadores a nodos. En otras palabras, si se usa un identificador $\mathtt{N}_i$ todos los identificadores menores o iguales que $\mathtt{N}_i$ deben también haber sido usados.

```
fact compactarNode {
  all n: Node | n in FReach[] => prevsNode[n] in FReach[]
}
```

Notar que estos predicados fuerzan la asignación de identificadores a los elementos del heap comenzando por las raíces en un recorrido *primero en anchura* de los campos del heap. Por lo tanto, durante el resto de este trabajo nos referiremos a ellos como predicados de rotura de simetrías primero en anchura (*breadth first*), o *predicados de rotura de simetrías BF* para abreviar.

2.7. Cotas Ajustadas para el Heap

2.7.1. Ejemplo motivador

Más adelante en este capítulo se dará una definición formal del concepto de cotas ajustadas para los campos del heap. En esta sección, se discutirán algunos ejemplos

que explican la intuición detras del concepto de cotas ajustadas, y el beneficio de su uso en el análisis de código.

Retomemos el ejemplo de listas simplemente encadenadas exhibido en la figura 2.4, sección 2.3. En este ejemplo, la aserción a verificar postula que las ejecuciones del método `addFirst` mantienen el invariante de aciclicidad de la clase `List`. Asumamos que se desea verificar la aserción anterior para listas de hasta tres nodos (scope 3 para `Node`). Asumamos también que la especificiación DynAlloy se instrumenta con predicados de rotura de simetría con el método de la sección 2.6.1. De esta manera, el modelo del heap para este ejemplo comprende las signaturas:

```
one sig null { }
sig List { }
sig Node { }
one sig N0,N1,N2 extends Node { }
```

junto con las relaciones:

```
head: List -> one (Node+null)
next: Node -> one (Node+null)
```

El conjunto de todos los valores posibles para el campo `next` dentro de los scopes dados se muestra en la figura 2.7. Sin embargo, la aserción a verificar afirma que sólo las listas acíclicas son entradas válidas del programa a verificar. Por lo tanto, muchos de los valores en `dom_next` de la figura 2.7 nunca ocurren en el estado inicial. Por ejemplo, los pares $(N_i, N_i), 0 \leq i \leq 2$, ya que evidentemente cualquier lista que los posea viola el requerimiento de aciclicidad.

$$\begin{aligned}\texttt{dom_next} = \{&(N_0, null), (N_1, null), (N_2, null),\\ &(N_0, N_0), (N_0, N_1), (N_0, N_2),\\ &(N_1, N_0), (N_1, N_1), (N_1, N_2),\\ &(N_2, N_0), (N_2, N_1), (N_2, N_2)\}\end{aligned}$$

Figura 2.7: Valores que puede tomar el campo `next` de listas simplemente encadenadas, con scope 3 para `Node`

De esta manera, el cómputo de cotas ajustadas consiste en averiguar el conjunto de valores que efectivamente puede tomar un campo en el estado inicial del programa, bajo las restricciones del invariante de clases (o una precondición). Para nuestro ejemplo, los valores permitidos para el campo `next` por el invariante –la cota ajustada `tfb_next` para `next`– se muestran en la figura 2.8 (recordemos que se asume que

el heap está canonizado). Notar que, de doce potenciales valores para `next`, la cota ajustada `tfb_next` admite sólo cinco. Como vimos en la sección 2.4, cada uno de estos posibles valores se representa con una variable proposicional en el análisis de código basado en SAT. Por lo tanto, la cota ajustada permite reducir la cantidad de variables proposicionales requeridas para codificar `next` a menos de la mitad (de 12 a 5). Es importante destacar que el análisis mantiene su corrección y completitud (hasta el scope dado), ya que los valores del campo eliminados por la cota ajustada nunca pueden ocurrir debido a la precondición del programa.

$$\texttt{tfb_next} = \{(N_0, null), (N_1, null), (N_2, null), (N_0, N_1), (N_1, N_2)\}$$

Figura 2.8: Una cota ajustada para el campo `next` de listas, con *scope* tres para `Node`, utilizando el invariante de aciclicidad de la figura 2.4

2.7.2. Definiciones

Hasta el momento hemos trabajado con una noción intuitiva de cotas ajustadas para campos. A continuación se define formalmente este y otros conceptos importantes para el desarrollo de este trabajo.

Definición 2.7.1. Sean $f_1, ..., f_n$ campos del heap, $C(f_1, ...f_n)$ una fórmula lógica que describe las instancias válidas del heap, y s un conjunto de scopes para los tipos de datos del programa. Una *cota ajustada de tamaño s para un campo* (*tight field bound*) $f_i : A \to B$ es una relación $TB_{f_i} \subseteq (A \times B)$ que cumple la siguiente propiedad: si existe al menos una instancia del heap dentro de los scopes que cumple C, en la que $a.f_i = b$, entonces $(a, b) \in TB_{f_i}$.

Notar que, según la definición anterior, una cota ajustada para un campo puede contener valores innecesarios, es decir, valores de campos que no pertenezcan a ninguna instancia válida del heap. Observar que, mientras más precisa la cota ajustada mayor es el beneficio para el análisis basado en SAT, ya que se pueden eliminar más variables proposicionales de la fórmula que codifica el programa a analizar. Consecuentemente, el objetivo de los algoritmos definidos en este trabajo es computar cotas ajustadas tan precisas como sea posible. La noción de cota ajustada más precisa se formaliza a continuación:

Definición 2.7.2. Sean $f_1, ..., f_n$, $C(f_1, ...f_n)$ y s como en la definición anterior. La *cota ajustada más precisa de tamaño s para un campo* (*tightest field bound*)

$f_i : A \to B$ es una relación $TB_{f_i} \subseteq (A \times B)$ tal que: $(a, b) \in TB_{f_i}$ si y sólo si existe al menos una instancia del heap dentro de los scopes que cumple C, para la cual $a.f = b$.

Cuando los scopes y los campos referenciados sean claros del contexto, simplemente hablaremos de *cota ajustada* o *cota ajustada más precisa*. Además, usaremos el plural *cotas ajustadas* para referirnos al conjunto que contiene una cota ajustada para cada uno de los campos que componen el heap:

Definición 2.7.3. Sean $TB_{f_1}, ..., TB_{f_n}$ cotas ajustadas (más precisas) de tamaño s para cada uno de los campos $f_1, ..., f_n$ del heap (acotado por s), respectivamente, se denomina al conjunto $TB_{f_1}, ..., TB_{f_n}$ cotas ajustadas (más precisas) de tamaño s.

2.7.3. Cotas ajustadas y rotura de simetrías

En general, para generar cotas ajustadas más precisas los algoritmos que las computan instrumentan el heap con predicados de rotura de simetrías. Por ejemplo, el enfoque paralelo TACO+ [25] canoniza el heap con los predicados de rotura de simetrías de la sección 2.6.1. Para ver que los predicados de rotura de simetrías influyen en las cotas generadas, consideremos las cotas ajustadas de tamaño 3 para el campo `next` de listas simplemente encadenadas, que se muestra en la figura 2.8. Sin rotura de simetrías, la lista que aparece a la derecha en la figura 2.6 es una instancia válida del heap. Por lo tanto, los pares $(N_1, N_0), (N_0, N_2)$ deberían ser parte de 2.8. Sin embargo, no es necesario que estos pares pertenezcan a las cotas ajustadas, ya que la estructura de la derecha de la figura 2.6 es equivalente a la que aparece a su izquierda –y esta última está permitida por la cota ajustada 2.8.

Recordar de la sección 2.6.1 que los predicados de rotura de simetrías discutidos allí etiquetan los nodos del heap siguiendo un recorrido primero en anchura de los campos. Por lo tanto, las cotas ajustadas computadas para un heap instrumentado con estos predicados se denominarán cotas ajustadas primero en anchura o *cotas ajustadas BF*.

2.8. TACO+: análisis de código basado en SAT explotando cotas ajustadas

Como vimos anteriormente, las cotas ajustadas permiten reducir la cantidad de variables proposicionales en la fórmula que codifica en programa de entrada, lo que produce un impacto positivo en el rendimiento de los análisis de programas basados

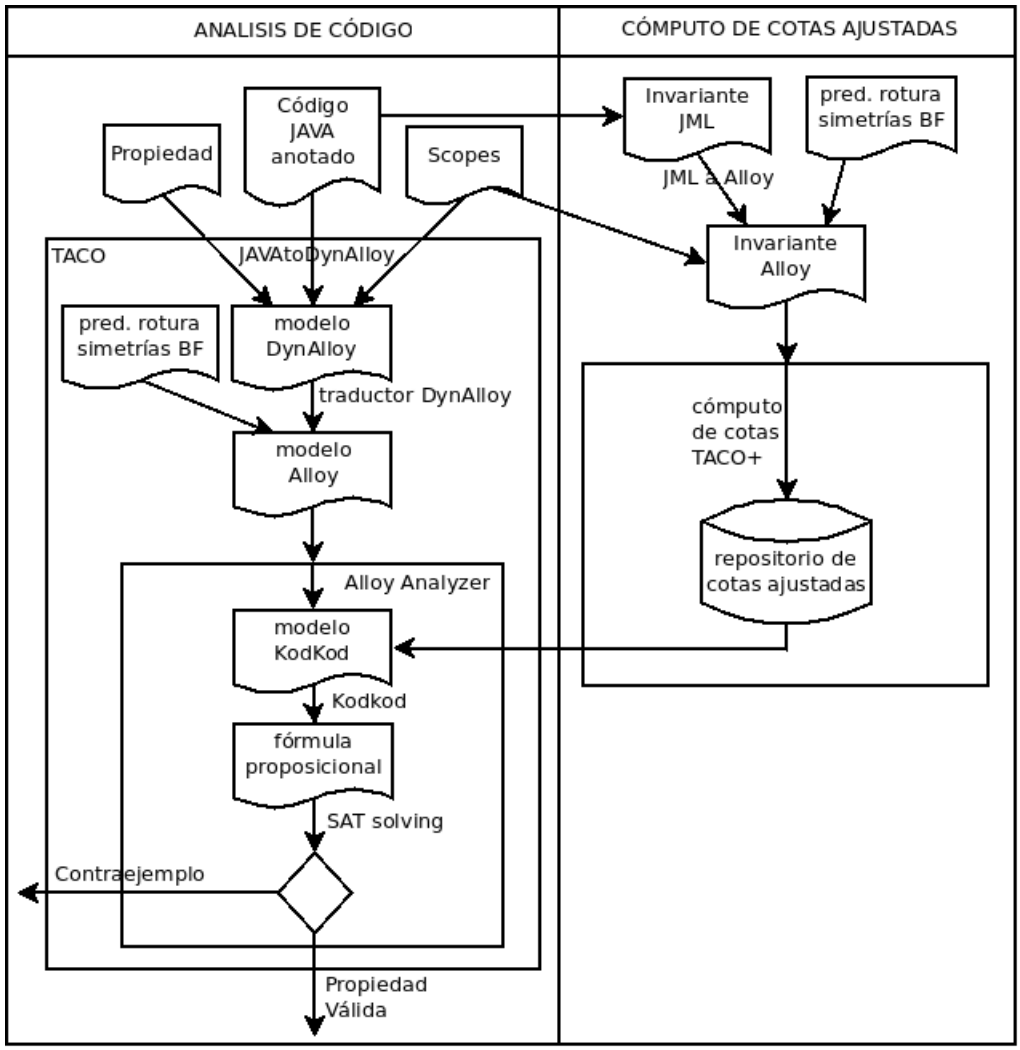


Figura 2.9: Esquema del análisis de código TACO+

en SAT. Recordemos que el análisis de program TACO+ de [25], que computa y explota cotas ajustadas, se divide en dos etapas:

1. Cómputo de cotas ajustadas usando una descripción declarativa de las entradas válidas del programa a analizar (por ejemplo, un invariante de clase).

2. Instrumentación de la verificación de código basado en SAT con las cotas ajustadas producidas en el paso anterior, y ejecución del análisis instrumentado.

La figura 2.9 presenta un esquema con los pasos que lleva a cabo TACO+ para analizar un programa. Básicamente, TACO+ está compuesto por el análisis basado

en SAT TACO (sección 2.5.1) más un componente cuya función es computar cotas ajustadas –implementar la etapa 1 anterior–, que puede verse en la parte derecha de la figura 2.9.

Al igual que TACO, TACO+ toma como entradas el código JAVA a verificar, incluyendo especificaciones de métodos e invariantes en lenguaje JML, un conjunto de scopes y una propiedad a verificar. Sin embargo, como se mencionó anteriormente, el primer paso efectuado por TACO+ es computar cotas ajustadas para el heap. La parte derecha de la figura 2.9 muestra que, en TACO+, el cómputo de cotas requiere de una especificación de las entradas válidas del programa a analizar –por ejemplo, un invariante de clase– en lenguaje Alloy, y que este invariante se genera automáticamente a partir del invariante JML provisto para el código JAVA. Previo al cómputo de las cotas ajustadas, la versión Alloy del invariante se instrumenta con predicados de rotura de simetrías BF (ver sección 2.6.1), para mejorar la precisión de las cotas producidas (como se discutió en la sección anterior). Esto implica que el algoritmo paralelo de TACO+ genera cotas ajustadas BF. Notar que, la generación de los predicados de rotura de simetrías dependen (de los campos) del código JAVA y de los scopes, pero estas dependencias no se muestran en la figura por simplicidad.

Ejemplo 2.8.1. En el ejemplo 2.5.1 vimos que el código anotado de la figura 2.3 (sección 2.3), junto con la propiedad de que el método `addFirst` mantiene el invariante de aciclicidad, es una entrada válida para TACO. Por lo tanto, también lo es para TACO+. Además, el predicado `listInvariant` de la figura 2.4 es el predicado Alloy obtenido de traducir el invariante JML del código de la figura 2.3. Así, la especificación Alloy que contiene el predicado `listInvariant`, y predicados de rotura de simetrías BF, se usa como entrada del algoritmo paralelo de TACO+ para generar cotas ajustadas. □

El algoritmo de cómputo de cotas de TACO+ consiste en, para cualquier par de elementos o, v con el tipo apropiado, consultar a SAT si $o.f = v$ en alguna instancia del heap. Notar que todas estas consultas individuales son independientes entre sí, y, consecuentemente, pueden efectuarse en paralelo. Así, TACO+ lleva a cabo un proceso iterativo que consiste en intentar resolver –vía SAT– todas las consultas posibles en paralelo, usando todas las computadoras que componen un cluster (y todos sus núcleos). Para cada una de estas consultas TACO+ impone un tiempo límite máximo, e interrumpe aquellas que no pueden resolverse en el tiempo dado. Al final de cada iteración, las consultas pueden particionarse en: aquellas que se respondieron positivamente, las que se respondieron negativamente, y aquellas para las cuales no se ha obtenido respuesta aún. Las consultas negativas se usan en cada paso para refinar la cota ajustada que se está computando, que inicialmente

comienza con todos los valores posibles para cada campo. Además, esta cota se usa (como cota ajustada, aunque no sea la más refinada posible) para acelerar las consultas a SAT. Debido a este proceso de refinamiento y uso de la cota actual, a medida que el algoritmo avanza mejoran las posibilidades de que las consultas a SAT finalicen dentro del tiempo límite. En la próxima iteración, TACO+ intenta resolver paralelamente las consultas para las que aún no obtuvo una respuesta satisfactoria (positiva o negativa). El algoritmo termina cuando este procedimiento converge, es decir, cuando luego de dos iteraciones las consultas sin responder se mantienen iguales. Notar que, dado un tiempo límite lo suficientemente grande, el algoritmo de cómputo de cotas de TACO+ produce las cotas ajustadas más precisas para el heap.

Ejemplo 2.8.2. Supongamos una invocación a TACO+ con las entradas del ejemplo 2.8.1, y scope 3 para la clase `Node`. Usando el invariante de listas `listInvariant` y predicados de rotura de simetrías BF, el algoritmo de cómputo de cotas de TACO+ retorna como resultado la cota ajustada más precisa `tfb_next` para el campo `next` de listas (figura 2.8) de la sección 2.7.1. □

Observar que, si se computan cotas ajustadas para una clase en base a su invariante, las cotas producidas pueden usarse para el análisis de todos los métodos de la clase. Por ejemplo, la cota ajustada `tfb_next` sólo depende de `listInvariant`, y podría usarse para analizar tanto `addFirst` como `delFirst` (figura 2.3). Por lo tanto, los autores de TACO+ proponen almancenar las cotas en un *repositorio de cotas ajustadas* para posteriores usos,, y así evitar el alto costo de recomputar las cotas en el caso mencionado anteriormente. El repositorio se muestran dentro del componente de cómputo de cotas de la figura 2.9. Evidentemente, para mantener la corrección de TACO+ los scopes usados para computar las cotas ajustadas y para efectuar el análisis instrumentado con cotas deben coincidir. Así, el repositorio almacena una cota ajustada por scope y cada clase analizada.

Una vez concluída la etapa de cómputo de cotas ajustadas BF, se debe ejecutar el análisis de código instrumentado con las cotas, como se mencionó en el punto 2 anterior. Observar en la parte izquierda de la figura 2.9 que el análisis instrumentado de TACO+ se desarrolla en gran parte de la misma manera que el análisis sin cotas ajustadas de TACO (figura 2.5). Sin embargo, existen dos diferencias principales entre estos dos análisis: TACO+ se instrumenta con predicados de rotura de simetrías y cotas ajustadas BF, mientras que TACO no. Como se muestra en la figura 2.9, en TACO+ los predicados de rotura de simetrías BF se añaden durante el análisis de código a la especificación Alloy que codifica el programa JAVA de entrada. Como se verá en nuestra evaluación experimental (sección 5), la instrumentación con predicados de rotura de simetrías mejora sustancialmente la velocidad y la escalabi-

lidad del análisis de código. Por otro lado, TACO+ usa las cotas ajustadas BF que computó durante su primera etapa como cotas superiores KodKod (ver sección 2.4) en el análisis de código. Esto se indica en la figura 2.9 con una flecha que relaciona (las cotas almacenadas en) el repositorio del componente de cómputo de cotas y el modelo KodKod generado por el análisis basado en SAT.

Ejemplo 2.8.3. En el ejemplo 2.8.2 se muestra el resultado de computar cotas ajustadas para el campo `next` –`tfb_next`de la figura 2.8– del modelo de listas simplemente encadenadas del ejemplo 2.8.1, para scope 3. En la sección 2.4 se explica como usar `tfb_next` como cota superior KodKod para reducir la cantidad de variables en la matriz que representa el campo `next` (figura 2.2), obteniendo así la matriz simplificada de la figura 2.3. Como se menciona en dicha sección, el uso de `tfb_next` como cota superior KodKod permite reducir de 12 a 5 la cantidad de variables proposicionales necesarias para representar el campo `next`. □

El ejemplo anterior ilustra el objetivo principal del uso de cotas ajustadas: permitir una codificación más compacta de los campos del heap en términos de una fórmula proposicional. Nuestra evaluación experimental muestra que la reducción en la cantidad de variables propiciada por las cotas ajustadas se traduce en la práctica en una mejora significativa del rendimiento de la verificación de código basada en SAT (sección 5.2).

2.9. Lógica de Separación

La lógica de separación (*separation logic*) [43] es una extensión de la lógica de primer orden que permite razonar de manera simple y concisa sobre programas que manipulan estructuras de datos dinámicas –alocadas en el heap–, tales como listas doblemente encadenadas, árboles, etc. En general, en estas estructuras un objeto puede ser referenciado desde diversos puntos del heap, lo que se denomina comúnmente como *aliasing*. Así, probar la corrección de programas que operan sobre estas estructuras requiere de considerar las intrincadas restricciones de *aliasing* que pueden ocurrir en el heap, convirtiendo a los formalismos definidos para este propósito en demasiado engorrosos y de difícil aplicación, inclusive en programas de tamaño moderado [42]. La lógica de separación aborda este problema introduciendo el operador $*$, denominado conjunción separadora (*separating conjunction*). Intuitivamente, $h_1 * h_2$ denota un heap conformado por dos partes disjuntas entre sí, descritas por las fórmulas h_1 y h_2, respectivamente.

Ejemplo 2.9.1. Supongamos un heap que satisface la fórmula $fla \doteq O \mapsto f : E * \texttt{true}$, esto es, el valor del campo f para el objeto O es E, y el objeto O se encuentra separado (sus dominios no se solapan debido a $*$) del subheap restante (que puede tener cualquier forma, ya que satisface `true`). De esta manera, la operación de asignación $O.f = G$ puede ejecutarse localmente sobre la parte izquierda de fla, resultando en la fórmula $fla' \doteq O \mapsto f : G * \texttt{true}$, dado que la ausencia de *aliasing* entre O y el resto del heap se encuentra garantizada por la semántica de $*$. □

La lógica de separación subyace a diversos análisis automáticos que prueban ausencia de violaciones de memoria en programas que operan sobre estructuras dinámicas (propiedad llamada *memory safety* en la literatura) [8, 11, 14, 36, 38, 39]. Las violaciones consideradas por la mayoría de estos análisis son: derreferencias *null* y a objetos no alocados; y "fugas de memoria" (*memory leaks*), es decir, que el programa finalice sin liberar toda la memoria que reservó durante su ejecución. El ejemplo 2.9.1 ilustra la ejecución de una asignación durante la verificación simbólica "hacia adelante" (*forward symbolic execution*) con especificaciones de la lógica de separación, que es la base de los análisis mencionados. Este tipo de razonamiento local, es decir, la capacidad de mutar porciones del heap sin tener en cuenta la forma de las partes restantes –posibilitado por la lógica de separación–, permite que estos enfoques superen ampliamente en escalabilidad a sus predecesores. Estas técnicas han probado ser valiosas para detectar fallas en programas de más de diez mil líneas de código [8], y algunas han escalado incluso hasta los millones de líneas (se aplicaron al núcleo del Sistema Operativo Linux, al famoso servidor web Apache, etc.) [11] –algo impensando para análisis de propiedades similares basados en otros formalismos.

2.9.1. Predicados Inductivos

Para especificar las diferentes estructuras dinámicas que aparecen en el heap durante la ejecución de un programa, en lógica de separación se definen predicados inductivos.

Ejemplo 2.9.2. A continuación, se define un predicado inductivo que caracteriza listas simplemente encadenadas.

$$slist(l) \doteq (l = null) \vee (\exists l_1.l \mapsto n : l_1 * slist(l_1))$$

$slist(l)$ está compuesto de un caso base y un caso inductivo. El primero, $l = null$, representa el caso en que la lista es vacía; el segundo, $\exists l_1.l \mapsto l_1 * slist(l_1)$, describe un nodo referenciado por l, cuyo campo n (siguiente) apunta a una lista simplemente encadenada que comienza en l_1. En este caso, $*$ asegura que el nodo l no forma parte de la lista l_1. □

Los análisis mencionados anteriormente trabajan con *heaps* potencialmente infinitos. Así, su convergencia depende de métodos para abstraer el heap representado por una fórmula de la lógica. Estos consisten en abstaer una cantidad arbitraria de nodos del heap que posean una determinada estructura utilizando predicados inductivos.

Ejemplo 2.9.3. La fórmula a continuación representa un heap con 3 nodos formando lista enlazada acíclica simplemente encadenada:

$$(\exists l_1, l_2.l \mapsto n : l_1 * l_1 \mapsto n : l_2 * l_2 \mapsto n : null)$$

La fórmula anterior puede abstraerse en:

$$slist(l)$$

perdiendo así la información sobre la cantidad de nodos alocados en el heap ($slist(l)$ admite listas de cualquier tamaño). □

2.9.2. Lógica de separación y cotas ajustadas

Uno de los objetivos de este trabajo es evaluar el impacto de utilizar diferentes lenguajes de especificación en el cómputo de cotas ajustadas. De esta manera, presentaremos aquí un uso novedoso de los predicados inductivos de la lógica de separación: dada una especificación de una estructura de datos en términos de un predicado inductivo, presentaremos algoritmos que computan cotas ajustadas para los campos del heap de manera eficiente.

De las definiciones de la sección 2.7.2 se desprende que una forma de computar cotas ajustadas consiste en averiguar cuales son los valores que efectivamente pueden tomar los campos del heap. De hecho, el algoritmo de cómputo de cotas de TACO+ 2.8 realiza una consulta a SAT por cada valor posible en el dominio de cada campo, para concluir sobre su aparición (o no) en la cota resultante. El ejemplo 2.9.4 muestra una intuición de la razón por la cual la forma de las fórmulas consideradas en este trabajo (ver 2.10) permiten responder rápida y eficientemente sobre la no pertenencia de ciertos valores de campos a cotas ajustadas para el heap.

Ejemplo 2.9.4. Consideremos nuevamente la fórmula que representa una lista enlazada acíclica con tres nodos:

$$(\exists l_1, l_2.l_0 \mapsto n : l_1 * l_1 \mapsto n : l_2 * l_2 \mapsto n : null)$$

La presencia del operador $*$ obliga a las variables l_0, l_1 y l_2 a apuntar a diferentes objetos en el heap. Llamemos O_i al objeto apuntado por l_i, $0 \leq i \leq 2$. Por la observación anterior, $O_i.n \neq O_i$, $\forall i, 0 \leq i \leq 2$, y, por lo tanto, dichos valores deben excluirse de la cota ajustada para el campo n. □

Intuitivamente, al eliminar posibilidades de *aliasing*, el operador $*$ permite quitar valores de una cota ajustada sin realizar consultas a un procedimiento de decisión.

2.9.3. Sintaxis

Como se mencionó anteriormente, en este trabajo se define una técnica para el cómputo de cotas ajustadas que requiere de una descripción del heap en términos de predicados inductivos de la lógica de separación. La técnica soporta predicados definidos en un fragmento restringido de la lógica, comúnmente denominado heaps simbólicos (*symbolic heaps*) [19]. Este fragmento de la lógica permite describir un gran cantidad de estructuras de datos frecuentemente usadas en la práctica, y goza de buenas propiedades tanto para el análisis automático de código como para el cómputo de cotas ajustadas (ejemplo 2.9.4).

La figura 2.10 presenta la *sintaxis* de los heaps simbólicos que utiliza nuestra técnica. Se asume un conjunto infinito $x, x_1, x_2, \ldots$ de variables, donde las variables sin subíndices son variables de programa, y las que poseen subíndices denotan variables auxiliares (sólo pueden aparecer en la definición de un predicado inductivo). Además, se asume un conjunto finito de campos del heap $f_1, f_2, \ldots$, un conjunto finito de constantes enteras k (en análisis acotado los scopes limitan la cantidad de constantes enteras disponibles), y la referencia especial *null*. Un *heap simbólico* ($\mathcal{SH}$) es una fórmula: $\exists \vec{x_i}.(\Sigma \wedge \Pi)$. Por un lado, Σ define la forma del heap, por lo

$$
\begin{array}{lll}
null & \in & Atoms \\
x, x_1, x_2, \ldots & \in & Var \\
k & \in & Int \\
f_1, f_2, \ldots & \in & Fields \\
E & := & x \mid null \\
p(x^*) & := & \bigvee_{j=1}^{m} \mathcal{SH}_j \\
\mathcal{SH} & := & \exists \vec{x_i}.(\Sigma \wedge \Pi) \\
\Sigma & := & \texttt{emp} \mid E \mapsto f_1 : E_1, \ldots, f_n : E_n \mid \Sigma_1 * \Sigma_2 \mid p(x^*) \\
\Pi & := & \gamma \wedge \phi \\
\gamma & := & E_1 = E_2 \mid E_1 \neq E_2 \mid \gamma_1 \wedge \gamma_2 \\
\phi & := & b \mid a \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \neg\phi_1 \\
b & := & true \mid false \mid x \mid b_1 = b_2 \\
a & := & s_1 = s_2 \mid s_1 < s_2 \mid s_1 > s_2 \mid s_1 \leq s_2 \mid s_1 \geq s_2 \\
s & := & k \mid x \mid k \times s_1 \mid s_1 + s_2 \mid -s_1 \mid max(s_1, s_2) \mid min(s_1, s_2)
\end{array}
$$

Figura 2.10: Sintaxis de los heaps simbólicos de la lógica de separación

que se denomina *parte espacial* de la fórmula. La parte espacial puede representar: un heap vacío (`emp`); un heap con un único objeto referenciado por E, con valores $E_1, \ldots, E_n$ para sus campos $f_1, \ldots, f_n$, respectivamente ($E \mapsto f_1 : E_1, \ldots, f_n : E_n$); un heap compuesto por dos *subheaps* disjuntos, que satisfacen Σ_1 y Σ_2 ($\Sigma_1 * \Sigma_2$); o un heap que almacena una estructura de datos definida por el predicado inductivo $p(v^*)$. Por otra parte, Π es una fórmula que restringe el conjunto de posibles valores que pueden tomar las variables del heap simbólico. Π es independiente del heap, por lo que usualmente se la denomina *parte pura*. En el lenguaje de la figura 2.10, la parte pura consta de una secuencia de conjunciones γ, con igualdades y desigualdades entre variables (y *null*), y una secuencia de conjunciones de fórmulas ϕ de la aritmética lineal entera libre de cuantificadores. Esto nos permite decidir satisfactibilidad de heaps simbólicos usando el SMT solver Yices (sección 2.10). Por último, los predicados inductivos $p(x^*)$ están compuestos por la disyunción de múltiples heaps simbólicos, cada uno de estos representando un caso base/inductivo particular del predicado.

Ejemplo 2.9.5. Como ejemplos de predicados definidos en el lenguaje de la figura 2.10 considere:

$$
\begin{aligned}
slist(l) &\doteq (l = null) \vee (\exists l_1 . l \mapsto n : l_1 * slist(l_1)) \\
btree(t) &\doteq (t = null) \vee (\exists t_1, t_2 . t \mapsto l : t_1, r : t_2 * tree(t_1) * tree(t_2)) \\
avl(t, h) &\doteq (t = null \wedge h = 0) \vee \\
&\qquad (\exists t_1, t_2, h_1, h_2 . t \mapsto l : t_1, r : t_2, ht : h * avl(t_1, h_1) * avl(t_2, h_2) \wedge \\
&\qquad\qquad h = 1 + max(h_1, h_2) \wedge |h_1 - h_2| \leq 1)
\end{aligned}
$$

slist, discutido anteriormente, modela listas enlazadas simplemente encadenadas. *btree* representa árboles binarios (con punteros). Por último, *avl* extiende el predicado anterior para caracterizar árboles binarios balanceados: las alturas de sus hijos derecho e izquierdo (h_1 y h_2, respectivamente) no pueden diferir en más de 1. □

Los heaps simbólicos presentados en esta sección se definieron originalmente en [39], en donde se utilizaron para verificar propiedades de forma (*shape*; por ejemplo, un programa toma como entrada una lista acíclica (un árbol binario) y retorna una lista acíclica (un árbol binario)) y de tamaño (por ejemplo, un procedimiento que quita un elemento de una lista de longitud n devuelve una lista de longitud $n - 1$) de diversas estructuras de datos. En general, los análisis escalables mencionados anteriormente (ej. [8, 11]) acotan aún más los heaps simbólicos con los que trabajan: sus partes puras sólo incluyen conjuciones de igualdades y desigualdades entre variables y *null* (γ en la Fig. 2.10). Esto reduce su expresividad, limitándolos sólo a la verificación de propiedades de forma (en general, trabajan eficientemente

sobre diversos tipos de listas). En este trabajo se optó por utilizar la forma más expresiva de los heaps simbólicos, con el objetivo de soportar estructuras que han sido objeto de análisis por las técnicas basadas en SAT actuales, como lo son los árboles AVL (notar que el predicado *avl* del ejemplo 2.9.5 incluye fórmulas aritméticas en su parte pura), árboles rojos y negros, etc.

Para finalizar esta sección, cabe mencionar que la lógica de separación también posee el operador denominado implicación separadora $-\!\!*$. $h_1 -\!\!* h_2$ postula que el resultado de extender el heap actual con otro disjunto, caracterizado por h_1, es un heap que satisface h_2. Este operador permite la verificación "hacia atrás" (*backwards reasoning*) [43], al estilo de la lógica de Hoare tradicional [30]. Sin embargo, hasta el momento no se conocen técnicas automáticas de análisis basadas en razonamientos hacia atrás. A nuestro entender esto se debe a la complejidad asociada de razonar con $-\!\!*$ (su definición requiere de cuantificación universal sobre *heaps*). Debido a la falta de utilidad de $-\!\!*$ para los propósitos de este trabajo, no se lo volverá a mencionar en la parte restante del texto.

2.9.4. Semántica

En esta sección se presenta la semántica usual del fragmento de la lógica de separación de la figura 2.10 [43]. En esta semántica, un *modelo* de una fórmula esta conformado por: una asignación de valores a las variables del programa (Store), y un modelo de la parte de la memoria reservada para almacenar los objetos alocados por el programa (Heap). Como el objetivo de este trabajo es el análisis de programas definidos en un subconjunto del lenguaje JAVA (sección 2.5.1), se consideraran sólo los tipos definidos por el usuario, y el tipo entero (Int). Así, para introducir un nuevo tipo T_i, el programador debe definir una clase C_i (con sus correspondientes campos). En nuestro análisis de programas acotado (sección 2.5.1) cada tipo T_i está compuesto por un conjunto acotado de objetos, y se asume que los tipos definidos por el usuario son disjuntos entre si ($\mathrm{T}_i \cap \mathrm{T}_j = \emptyset$ para $i \neq j$). En esta sección se asume un conjunto fijo de clases $\mathrm{C}_1, \mathrm{C}_2, \ldots, \mathrm{C}_k$, que introducen los tipos correspondientes $\mathrm{T}_1, \mathrm{T}_2, \ldots, \mathrm{T}_k$, respectivamente. El conjunto Obj denota la union de todos los objetos disponibles y se define a continuación.

$$\mathrm{Obj} = \mathrm{T}_1 \cup \mathrm{T}_2 \cup \ldots \cup \mathrm{T}_k$$

Una variable de programa puede tomar como valor: un objeto de algún tipo definido por el usuario; *null*, si se trata de una referencia no inicializada; o un número entero. Por lo tanto, el conjunto de valores, Val, se define como sigue:

$$\mathrm{Val} = \mathrm{Obj} \cup \mathrm{Int} \cup \{\mathrm{null}\}$$

$$[\![x]\!]s = s(x) \qquad [\![x_i]\!]s = s(x_i) \qquad [\![nil]\!]s = nil$$

$s \models E_1 = E_2$	sii	$[\![E_1]\!]s = [\![E_2]\!]s$
$s \models E_1 \neq E_2$	sii	$s \not\models E_1 = E_2$
$s \models \phi$	sii	$s \models_A \phi$
$s \models \Pi_1 \wedge \Pi_2$	sii	$s \models \Pi_1$ y $s \models \Pi_2$
$s, h \models \texttt{emp}$	sii	$\mathrm{dom}(h) = \emptyset$
$s, h \models E \mapsto f_1 : E_1, \ldots, f_n : E_n$	sii	$f_i = [[\![E]\!]s \mapsto [\![E_i]\!]s]$, para $i = 1 \ldots n$
$s, h \models \Sigma_1 * \Sigma_2$	sii	existen h_1, h_2 tales que $h_1 * h_2$ y $h = h_1 \cup h_2$ y $s, h_1 \models \Sigma_1$ y $s, h_2 \models \Sigma_2$
$s, h \models \exists \vec{x_i}.(\Sigma \wedge \Pi)$	sii	existen $\vec{v}$ tales que $s(\vec{x_i} \mapsto \vec{v}), h \models \Sigma$ y $s(\vec{x_i} \mapsto \vec{v}) \models \Pi$
$s, h \models \mathcal{SH}_1 \vee \mathcal{SH}_2$	sii	$s, h \models \mathcal{SH}_1$ o $s, h \models \mathcal{SH}_2$

Figura 2.11: Semántica de los heaps simbólicos de la lógica de separación

El formato de las asignaciones de valores a variables (Stores) y de los heaps (Heaps) se muestra a continuación:

$$\text{Stores} : \text{Var} \rightarrow \text{Val} \qquad \text{Heaps} : \text{Fields} \rightarrow (\text{Obj} \rightarrow \text{Val})$$

$s \in$ Stores es simplemente una función de Var en Val. Para entender la definición de Heaps, recordemos que en el análisis TACO (sección 2.5.1) el heap se modela como un conjunto de funciones $f_1, \ldots, f_n$ por cada campo de alguna clase definida por el usuario, donde cada f_i asigna valores al campo homónimo de los objetos (del tipo adecuado) alocados en el heap 2.3.1. En esta sección, se asumen un conjunto fijo de campos para el heap: Fields $= f_1, \ldots, f_n$. Luego, la definición anterior de Heaps mapea cada f_i a la función correspondiente. Adicionalmente, se define la función dom : Heaps $\rightarrow 2^{\text{Obj}}$, que, intuitivamente, retorna los objetos alocados en el heap. Formalmente: $\mathrm{dom}(h) = dom(f_1) \cup \ldots \cup dom(f_n)$ (dom de un campo retorna el dominio del campo).

La semántica de los heaps simbólicos (definidos mediante la gramática de la figura 2.10) se define como una relación Stores $\times$ Heaps $\models \mathcal{SH}$, tal que $s, h \models \mathcal{SH}$ si y sólo si s y h satisfacen $\mathcal{SH}$; y se muestra en la figura 2.11.

La función $[\![]\!]$ define la semántica de las expresiones E, que denotan variables o

la expresión *null* (ver Fig. 2.10), según su valor en el store s. Siguiendo la definición de $[\![]\!]$, el valor de una variable x es $s(x)$.

Por un lado, la validez de las fórmulas puras Π depende exclusivamente del store s (y es independiente del heap h). Así, una igualdad entre expresiones E_1 y E_2 es verdadera siempre y cuando ambas expresiones tienen el mismo valor en s; la desigualdad se cumple si no vale lo anterior. Para evaluar fórmulas aritméticas, ϕ, se utiliza un procedimiento de decisión para la aritmética lineal entera libre de cuantificadores, denotado por $\models_A$ en la figura. La conjunción de dos fórmulas puras es verdadera si y sólo sí ambas fórmulas son verdaderas en el store.

Por otro lado, las partes espaciales, Σ, de una fórmula se evalúan en un store s y un heap h. `emp` es verdadera para h vacío, es decir, cuando no hay objetos alocados en el heap. $E \mapsto f_1 : E_1, \ldots, f_n : E_n$ indica que h esta conformado por un único objeto $[\![E]\!]s$, y los valores de sus campos son $[\![E_1]\!]s, \ldots, [\![E_n]\!]s$. $\Sigma_1 * \Sigma_2$ vale si el heap actual h puede dividirse en dos subheaps disjuntos h_1 y h_2, tales que h_1 satisface Σ_1 y h_2 satisface Σ_2. Formalmente, se pide que $h_1 * h_2$, con $*$ definida por $h_1 * h_2$ sii $\mathrm{dom}(h_1) \cap \mathrm{dom}(h_2) = \emptyset$ y $h = h_1 \cup h_2$, donde:

$$h_1 \cup h_2 = \lambda f, x. \begin{cases} h_1(f)(x) & \text{si } f \in \mathrm{dom}(h_1), x \in \mathrm{dom}(h_1(f)) \\ h_2(f)(x) & \text{si } f \in \mathrm{dom}(h_2), x \in \mathrm{dom}(h_2(f)) \\ \text{indefinido} & \text{otro caso} \end{cases}$$

Como se pide que h_1 y h_2 sean disjuntos ($h_1 * h_2$), no hay problemas de solapamiento entre las primeras dos condiciones. Notar que en la figura 2.11 no se da semántica a los predicados inductivos. La razón es que los predicados inductivos se usarán para: construir un conjunto finito de fórmulas sin llamadas recursivas, a partir de las cuales se computarán cotas ajustadas (sección 4.1); o para computar cotas ajustadas mientras se realizan desplegados de los predicados (sección 4.2).

Intuitivamente, s, h son un modelo de $\exists \vec{x_i}.(\Sigma \wedge \Pi)$ si el store extendido con valores para las variables libres $\vec{x_i}$ del heap simbólico, $s(\vec{x_i} \mapsto \vec{v})$, satisface la parte pura Π, y, junto con h, satisfacen la parte espacial $varSigma$. $s(\vec{x_i} \mapsto \vec{v})$ es una función idéntica a s, salvo para las variables $\vec{x_i}$, para las que devuelve el valor correspondiente en $\vec{v}$. Formalmente:

$$s(x_1, \ldots, x_n \mapsto v_1, \ldots, v_n)(x) = \begin{cases} v_i & \text{si } x = x_i \text{ para algún } i = 1..n \\ s(x) & \text{otro caso} \end{cases}$$

En este texto, las variables del programa aparecen sin subíndices $-l, t, h-$, mientras que las variables auxiliares introducidas por los predicados inductivos llevan subíndices $-x_1, x_2, \ldots$. Por ejemplo, en el predicado slist del ejemplo 2.9.5 l es una variable del programa (apunta a la cabeza de la lista), pero l_1 no lo es. Para simplificar la definición de los predicados inductivos, a partir de ahora la cuantificación existencial

en las variables auxiliares ($\vec{x_i}$) de un heap simbólico se asumirá implícita. Es decir, escribiremos simplemente $\Sigma \wedge \Pi$ para representar el heap simbólico $\exists \vec{x_i}.(\Sigma \wedge \Pi)$.

Ejemplo 2.9.6. Siguiendo esta convención, el predicado btree:

$$btree(t) \doteq (t = null) \vee (\exists t_1, t_2.t \mapsto l : t_1, r : t_2 * btree(t_1) * btree(t_2))$$

se define como:

$$btree(t) \doteq (t = null) \vee (t \mapsto l : t_1, r : t_2 * btree(t_1) * btree(t_2))$$

□

Por último, un par s, h es modelo de la disyunción de dos heaps simbólicos siempre y cuando s, h es modelo de alguno de ellos.

Cabe destacar que la semántica presentada en esta sección difiere de la usual en la definición de los heaps, que se muestra a continuación:

$$\text{Heaps} : \text{Loc} \underset{fin}{\rightharpoonup} (\text{Fields} \rightarrow \text{Val})$$

donde Loc es un conjunto de enteros que representan direcciones de memoria. La definición anterior permite desarrollar análisis que soporten aritmética de enteros. En cambio, el objetivo de este trabajo es definir análisis eficientes para lenguajes orientados de alto nivel (como JAVA), que evitan al programador el manejo de bajo nivel de la memoria. Por lo tanto, se utiliza Obj en lugar de Loc, como el dominio de los heaps. Además, se invierte el orden del primer y segundo argumento en Heaps, usando Fields y Obj como primer y segundo argumento, respectivamente. Esto se debe a que se desea mantener la coherencia entre la semántica dada a las fórmulas de la lógica de separación y la definición de los heaps en el análisis TACO (sección 2.5.1).

2.10. El SMT solver Yices

El SMT solver Yices [21] permite decidir sobre la satisfactibilidad de fórmulas (sin cuantificadores) de la aritmética lineal entera, aritmética lineal real, funciones no interpretadas con igualdad, arreglos y vectores de bits. En este trabajo, se usa Yices para generar instancias concretas de los heaps simbólicos. Para esto es necesario evaluar sus partes puras, es decir, fórmulas de la aritmética lineal entera. A

continuación se discutirán mediante ejemplos las construcciones de Yices más relevantes a nuestros propósitos. Para una descripción más completa, se remite al lector a la documentación disponible en la página web de la herramienta [4], desde donde también puede descargarse una implementación de Yices.

Yices provee los tipos predefinidos `int`, `real` y `bool`, que representan los números enteros, los números reales y valores lógicos, respectivamente. Además, Yices posibilita la definición por parte del usuario de conjuntos finitos de elementos (tuplas), usando la palabra clave `scalar`.

Ejemplo 2.10.1. El tipo `color`, conformado por los elementos `rojo`, `azul` y `verde` se muestra abajo:

```
(define-type color (scalar rojo azul verde))
```

□

También es posible definir constantes tipadas, con la cláusula `define`.

Ejemplo 2.10.2. A continuación se definen las constantes `i` de tipo entero, `r` de tipo real y `c` de tipo color:

```
(define i::int)
(define r::real)
(define c::color)
```

□

Para construir fórmulas aritméticas (recordar que Yices sólo funciona con fórmulas de la aritmética lineal), Yices provee los operadores `+` (suma), `-` (resta), `*` (multiplicación), `/` (división), `div` (división entera) y `mod` (resto de la división entera). Los operadores de igualdad y desigualdad se denotan con `=` y `/=`, respectivamente. Yices no distingue entre términos y fórmulas, por lo que los operadores anteriores pueden aplicarse a fórmulas booleanas, en cuyo caso representan la equivalencia y la disyunción exclusiva. `>` y `<` representan las relaciones de orden usuales para los tipos numéricos `int` y `real`. Además, Yices provee los operadores lógicos tradicionales: `and` (conjunción), `or` (disyunción), `not` (negación) y `=>` (implicación). Yices usa sintaxis prefija para las fórmulas.

Ejemplo 2.10.3. Asumiendo las definiciones:

```
(define i::int)
(define c::color)
```

Son ejemplos de fórmulas (satisfactibles) Yices:

```
(/= rojo azul)
(=> (= c rojo) (/= c azul))
(or (< i 0) (= i 0) (> i 0))
(=> (and (>= i 0) (/= i 0)) (> i 0))
```

La primera afirma que los colores `rojo` y `azul` son distintos, y es válida porque todos los elementos de un tipo escalar (*scalar*) son disjuntos entre sí. La segunda asegura que si la constante `c` es igual a `rojo`, `c` debe ser distinta de `azul`, y es satisfactible por la misma razón que la anterior. La tercera predica que la constante entera `i` es o bien menor a 0, o bien igual a 0, o bien mayor a 0. Finalmente, la última fórmula afirma que si la constante entera `i` es mayor o igual a 0 y distinta de 0 a la vez, entonces `i` debe ser mayor que 0. □

Otra cláusula que utilizaremos en este trabajo es `ite`, que se usa para definir expresiones condicionales. `(ite (cond) e1 e2)` retorna la expresión `e1` si se cumple `cond`, y `e2` en caso contrario.

Ejemplo 2.10.4. La expresión a continuación retorna la mayor de las constantes enteras `x` e `y`:

```
(ite (>= x y) x y)
```

□

Para averiguar sobre la satisfactibilidad de una fórmula (o un conjunto de fórmulas) Yices provee los comandos `assert` y `check`. Por ejemplo, para la fórmula `f` primero se debe ejecutar primero `(assert f)`, y luego `(check)`. Yices retornará `sat` o `unsat` de acuerdo a si `f` es satisfactible o no (respectivamente). Si además se desea que Yices genere un modelo de `f` (en caso de que sea `sat`) se debe ejecutar el comando `(show-model)`.

Ejemplo 2.10.5. Para buscar un modelo de la fórmula `(or (< i 0) (= i 0) (> i 0))` se define la especificación:

```
(define i::int)
(assert (or (< i 0) (= i 0) (> i 0)))
(check)
(show-model)
```

y se invoca a la herramienta Yices con la misma como entrada. El resultado de la ejecución se muestra a continuación:

```
sat
(= i -1)
```

□

Para concluir esta sección, es importante volver a destacar que en este trabajo Yices se usará como procedimiento de decisión para fórmulas de la aritmética lineal entera. Decidir sobre este tipo de fórmulas es una característica estándar de los SMT solvers actuales, implementada por varias herramientas, como Z3 [16], MathSAT [10], etc. La elección de Yices se debe a que posee un herramienta madura que lo implementa, que además fue capaz de responder a todas las consultas realizadas en este trabajo muy eficientemente –en fracciones de segundo–. Como en nuestros experimentos el uso de Yices no limita la performance de los algoritmos que dependen de SMT solving (los enfoques basados en lógica de separación del capítulo 4), no se experimentó con implementaciones alternativas de SMT solvers. De cualquier manera, sería deseable evaluar las diferentes herramientas y seleccionar la más eficiente. La resolución de este problema se deja como trabajo futuro.

Capítulo 3

Cómputo secuencial de cotas ajustadas mediante visita de instancias

En este capítulo presentaremos la primera de las contribuciones principales de esta tesis: un enfoque secuencial para el cómputo de cotas ajustadas basado en la visita de instancias válidas del heap, al que denominamos *bottom-up* por la forma en que lleva a cabo las computaciones –inversa a la de TACO+, que es un algoritmo *top-down* (sección 2.8). El capítulo se organiza como se describe a continuación. En la sección 3.1 se presenta el caso de estudio árboles binarios de búsqueda balanceados, AVL, que se usará en la sección 3.2 para presentar las ideas subyacentes al enfoque bottom-up. Luego, la sección 3.3 discute las distintas variantes de bottom-up implementadas. La sección 3.4 discute la idea de cómputo incremental de cotas ajustadas, es decir, como aprovechar cotas ajustadas para scopes más pequeños en el cómputo de cotas ajustadas para scopes más grandes, con el objetivo de evitar invocaciones al procedimiento de decisión. Por último, en la sección 3.5 se explica como incorporar las cotas ajustadas computadas por bottom-up en el análisis de propiedades de programas TACO 2.5.1).

3.1. Un caso de estudio: AVLs

Un caso de estudio de complejidad media, que permite ilustrar tanto los beneficios del uso de cotas ajustadas en el análisis como el funcionamiento de bottom-up, es la estructura de datos AVL. Los AVLs son arboles binarios de búsqueda balanceados. Los campos de las clases JAVA que definen AVLs y sus nodos se muestran a continuación:

```
class AVL {
  root: Node;
  ...
}
```

```
class Node {
  left: Node;
  right: Node;
  height: Int;
  ...
}
```

Los AVLs tienen una referencia al nodo raíz del árbol, `node`. Los nodos tienen punteros a sus subárboles izquierdo y derecho `left` y `right`, y un atributo para almacenar la altura de cada nodo en el árbol, `height`. Para simplificar la presentación de los ejemplos se omite la definición de los campos de los nodos que almacenan datos (además, las cotas ajustadas para estos campos son en general poco precisas, por lo que no tienen un impacto significativo en el análisis automático). De la traducción de las clases anteriores a DynAlloy resultan los siguientes campos:

```
root: AVL -> one (Node+null),
left: Node -> one (Node+null),
right: Node -> one (Node+null),
height: Node -> one Int
```

La figura 3.1 muestra un invariante declarativo para AVLs, en lenguaje DynAlloy. Es importante recordar que la fórmula `AVL_Invariant` de la figura 3.1 puede generarse automáticamente a partir de una especificación JML del invariante de la clase AVL en el código JAVA. `AVL_Invariant` impone dos tipos de restricciones sobre la estructura. Primero, los AVLs deben ser árboles binarios, esto es, estructuras acíclicas en donde ninguno de sus subárboles comparten nodos. Estas dos restricciones son impuestas por las fórmulas `x !in x.^(left+right)` y `((x.left).*(left+right)) & ((x.right).*(left+right)) in null`, respectivamente –notar que estas restricciones deben cumplirse para todos los nodos `x`. Segundo, los AVLs deben estar balanceados: las alturas de los hijos izquierdo y derecho de cada nodo del árbol no pueden diferir en más de 1. Formalmente, la altura de las hojas es siempre 0 `(x.left=null && x.right=null => x.height=0)`. Si uno de los hijos de un nodo `x` es `null` y el otro no, entonces la altura de `x` debe ser 1, y su hijo debe ser una hoja (ej. `(x.left=null && x.right!=null => x.height=1 && x.right.height=0)`). Por último, si `x` tiene ambos hijos, la altura de `x` se calcula como 1 más el máximo de las alturas de sus hijos izquierdo y derecho (`x.height=max[x.left.height,x.right.height]+1`), y el valor absoluto de la altura de `x` debe ser menor igual a 1 (`abs[x.left.height -x.right.height]<=1`).

Veamos un ejemplo que ilustra la reducción en el número de variables proposicionales en la codificación de los campos de AVL proporcionada por el uso de cotas ajustadas en este caso de estudio.

```
pred AVL_Invariant[root: AVL -> one (Node+null),
                   left: Node -> one (Node+null),
                   right: Node -> one (Node+null),
                   height: Node -> one Int] {
  all x: Node |
      -- los AVLs son arboles binarios
      x !in x.^(left+right) &&
      ((x.left).*(left+right)) & ((x.right).*(left+right)) in null
      -- los AVLs son arboles balanceados
      (x.left=null && x.right=null => x.height=0) &&
      (x.left=null && x.right!=null =>
          x.height=1 && x.right.height=0) &&
      (x.left!=null && x.right=null =>
          x.height=1 && x.left.height=0) &&
      (x.left!=null && x.right!=null =>
          x.height=max[x.left.height,x.right.height]+1 &&
          abs[x.left.height-x.right.height]<=1)
}
```

Figura 3.1: Invariante de clase para la estructura AVL en DynAlloy

Ejemplo 3.1.1. La figura 3.2 exhibe los dominios de los campos `left`, `right` y `height` de AVL, para un máximo de 5 nodos y 5 enteros (0..4) –scope 5 para simplificar. La cantidad total de posibles valores para todos los campos es de 85, y, por lo discutido en la sección 2.4, 85 variables proposicionales son necesarias para codificar los campos en un problema de SAT. Por otro lado, la figura 3.3 muestra las cotas ajustadas más precisas para AVLs, computadas sobre el invariante de la figura 3.1 vía bottom-up. Ahora los posibles valores para los campos se reducen a sólo 29. Es decir, utilizar cotas ajustadas permite reducir sustancialmente – de 85 a 29 – el número de variables proposicionales que representan los campos de AVL. Cabe destacar que esta proporción se mantiene cuando los scopes crecen. □

3.2. Intuición y ejemplo motivador

En enfoque bottom-up surge de observar la definición 2.7.1 de cotas ajustadas de la sección 2.7.2. Intutivamente, la definición afirma que si en alguna instancia válida

$$\texttt{dom_left} = \{(N_0, null), (N_1, null), (N_2, null), (N_3, null), (N_4, null),$$
$$(N_0, N_0), (N_0, N_1), (N_0, N_2), (N_0, N_3), (N_0, N_4),$$
$$(N_1, N_0), (N_1, N_1), (N_1, N_2), (N_1, N_3), (N_1, N_4),$$
$$(N_2, N_0), (N_2, N_1), (N_2, N_2), (N_2, N_3), (N_2, N_4),$$
$$(N_3, N_0), (N_3, N_1), (N_3, N_2), (N_3, N_3), (N_3, N_4),$$
$$(N_4, N_0), (N_4, N_1), (N_4, N_2), (N_4, N_3), (N_4, N_4)\}$$

$$\texttt{dom_right} = \{(N_0, null), (N_1, null), (N_2, null), (N_3, null), (N_4, null),$$
$$(N_0, N_0), (N_0, N_1), (N_0, N_2), (N_0, N_3), (N_0, N_4),$$
$$(N_1, N_0), (N_1, N_1), (N_1, N_2), (N_1, N_3), (N_1, N_4),$$
$$(N_2, N_0), (N_2, N_1), (N_2, N_2), (N_2, N_3), (N_2, N_4),$$
$$(N_3, N_0), (N_3, N_1), (N_3, N_2), (N_3, N_3), (N_3, N_4),$$
$$(N_4, N_0), (N_4, N_1), (N_4, N_2), (N_4, N_3), (N_4, N_4)\}$$

$$\texttt{dom_height} = \{(N_0, 0), (N_1, 0), (N_2, 0), (N_3, 0), (N_4, 0),$$
$$(N_0, 1), (N_1, 1), (N_2, 1), (N_3, 1), (N_4, 1),$$
$$(N_0, 2), (N_1, 2), (N_2, 2), (N_3, 2), (N_4, 2),$$
$$(N_0, 3), (N_1, 3), (N_2, 3), (N_3, 3), (N_4, 3),$$
$$(N_0, 4), (N_1, 4), (N_2, 4), (N_3, 4), (N_4, 4)\}$$

Figura 3.2: Dominios de los campos `left`, `right` y `height` de AVL para un máximo de 5 nodos, en el contexto de análisis acotado de código.

$$\texttt{tb_left} = \{(N_0, null), (N_0, N_1), (N_1, null), (N_2, null), (N_2, N_3),$$
$$(N_3, null), (N_4, null), (N_1, N_3), (N_2, N_4)\ \}$$

$$\texttt{tb_right} = \{(N_0, null), (N_1, null), (N_0, N_1), (N_0, N_2), (N_2, null),$$
$$(N_2, N_4), (N_3, null), (N_4, null), (N_2, N_3), (N_1, N_3), (N_1, N_4)\ \}$$

$$\texttt{tb_height} = \{(N_0, 0), (N_0, 1), (N_1, 0), (N_2, 0), (N_0, 2), (N_2, 1),$$
$$(N_3, 0), (N_4, 0), (N_1, 1)\}$$

Figura 3.3: Cotas ajustadas para los campos `left`, `right` y `height` de AVL para un máximo de 5 nodos.

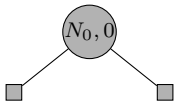
del heap –una instancia que satisfaga las restricciones impuestas sobre el heap, como por ejemplo, un invariante de clase– un campo $o.f$ toma el valor v, entonces el par (o, v) pertenece a la cota ajustada para el campo f. De lo anterior se deriva que una forma posible de computar cotas ajustadas consiste en visitar instancias válidas del heap, e incluir en la cota ajustada que retorna el algoritmo los valores de los campos de cada una de estas instancias. Sin embargo, en general hay un número muy grande de instancias válidas del heap, inclusive para scopes pequeños. Por este motivo, el enfoque bottom-up sólo considera instancias que sirven para agregar nuevos valores a la cota ajustada. Para obtener estas instancias se realizan llamadas a un procedimiento de decisión basado en SAT solving.

Ejemplo 3.2.1. Consideremos el caso de estudio AVL de la sección anterior. La figura 3.4 muestra el proceso de cómputo de cotas ajustadas para este caso de estudio vía bottom-up. bottom-up utiliza un conjunto por cada uno de los campos `left`, `right` y `height`. Los conjuntos son vacíos al inicio del algoritmo, y se van extendiendo con los valores de los campos de las instancias visitadas. bottom-up finaliza cuando no existen más instancias válidas que puedan añadir valores a los conjuntos, momento en el que estos se convierten en cotas ajustadas para sus respectivos campos. Cada iteración de bottom-up involucra:

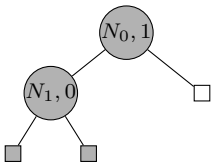
1. Consultar a un procedimiento de decisión por una instancia válida del heap que contenga valores de campos que no pertenezcan a los conjuntos `left`, `right` y `height`.

2. Extender los conjuntos con los valores provistos por la nueva instancia visitada.

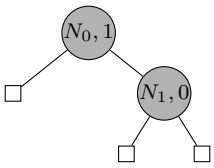
Volviendo a nuestro ejemplo, cada una de las filas de las figuras 3.4a y 3.4b corresponde a una iteración particular de bottom-up. La parte izquierda de una fila muestra la instancia visitada en la iteración correspondiente, y la parte derecha exhibe –en forma tabular– el contenido de los conjuntos `left`, `right` y `height` al final de la iteración. Los valores de campos agregados a la cota en la iteración corriente se resaltan en las instancias como nodos con fondo gris, y en las tablas con fuente en negrita. Por ejemplo, en la primera iteración el algoritmo visita un AVL con un sólo nodo N_0, con altura cero, y sus hijos apuntado a *null*. A la derecha, en la misma fila, puede verse el contenido de los conjuntos `left`, `right` y `height`. Estos eran vacíos al comienzo del algoritmo, por lo que al final de la iteración deben contener los pares $(N_0, null)$, $(N_0, null)$ y $(N_0, 0)$, respectivamente. Por último, cabe destacar que, en la parte derecha de la iteración final de la figura, los conjuntos contienen las cotas ajustadas más precisas para AVLs y un máximo de cinco nodos – ilustradas en la figura 3.3. □



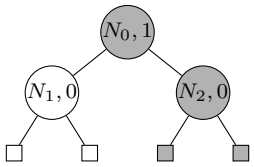
left	right	height
$(\mathbf{N_0}, \mathbf{null})$	$(\mathbf{N_0}, \mathbf{null})$	$(\mathbf{N_0}, \mathbf{0})$



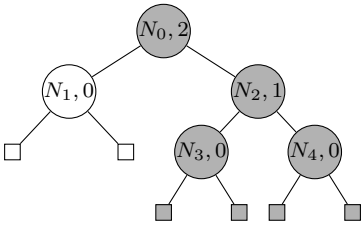
left	right	height
$(N_0, null)$	$(N_0, null)$	$(N_0, 0)$
$(\mathbf{N_0}, \mathbf{N_1})$	$(\mathbf{N_1}, \mathbf{null})$	$(\mathbf{N_0}, \mathbf{1})$
$(\mathbf{N_1}, \mathbf{null})$		$(\mathbf{N_1}, \mathbf{0})$



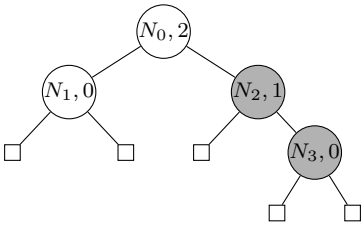
left	right	height
$(N_0, null)$	$(N_0, null)$	$(N_0, 0)$
(N_0, N_1)	$(N_1, null)$	$(N_0, 1)$
$(N_1, null)$	$(\mathbf{N_0}, \mathbf{N_1})$	$(N_1, 0)$



left	right	height
$(N_0, null)$	$(N_0, null)$	$(N_0, 0)$
(N_0, N_1)	$(N_1, null)$	$(N_0, 1)$
$(N_1, null)$	(N_0, N_1)	$(N_1, 0)$
$(\mathbf{N_2}, \mathbf{null})$	$(\mathbf{N_0}, \mathbf{N_2})$	$(\mathbf{N_2}, \mathbf{0})$
	$(\mathbf{N_2}, \mathbf{null})$	

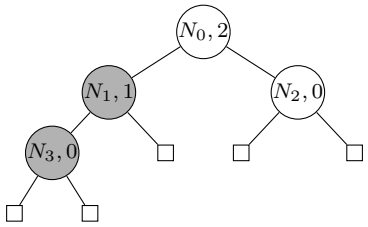


left	right	height
$(N_0, null)$	$(N_0, null)$	$(N_0, 0)$
(N_0, N_1)	$(N_1, null)$	$(N_0, 1)$
$(N_1, null)$	(N_0, N_1)	$(N_1, 0)$
$(N_2, null)$	(N_0, N_2)	$(N_2, 0)$
$(\mathbf{N_2}, \mathbf{N_3})$	$(N_2, null)$	$(\mathbf{N_0}, \mathbf{2})$
$(\mathbf{N_3}, \mathbf{null})$	$(\mathbf{N_2}, \mathbf{N_4})$	$(\mathbf{N_2}, \mathbf{1})$
$(\mathbf{N_4}, \mathbf{null})$	$(\mathbf{N_3}, \mathbf{null})$	$(\mathbf{N_3}, \mathbf{0})$
	$(\mathbf{N_4}, \mathbf{null})$	$(\mathbf{N_4}, \mathbf{0})$

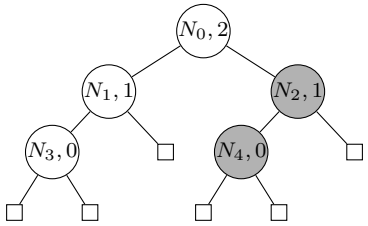


left	right	height
$(N_0, null)$	$(N_0, null)$	$(N_0, 0)$
(N_0, N_1)	$(N_1, null)$	$(N_0, 1)$
$(N_1, null)$	(N_0, N_1)	$(N_1, 0)$
$(N_2, null)$	(N_0, N_2)	$(N_2, 0)$
(N_2, N_3)	$(N_2, null)$	$(N_0, 2)$
$(N_3, null)$	(N_2, N_4)	$(N_2, 1)$
$(N_4, null)$	$(N_3, null)$	$(N_3, 0)$
	$(N_4, null)$	$(N_4, 0)$
	$(\mathbf{N_2}, \mathbf{N_3})$	

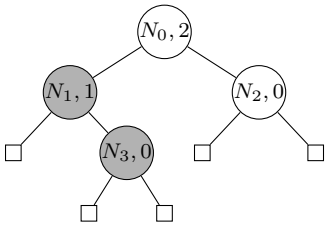
(a) Ejemplo de cómputo de cotas ajustadas usando bottom-up: AVLs con 5 nodos (primera parte)



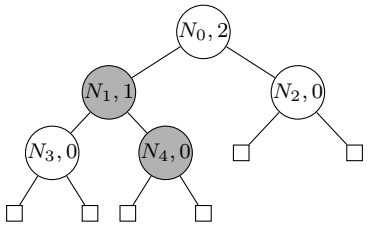
left	right	height
$(N_0, null)$	$(N_0, null)$	$(N_0, 0)$
(N_0, N_1)	$(N_1, null)$	$(N_0, 1)$
$(N_1, null)$	(N_0, N_1)	$(N_1, 0)$
$(N_2, null)$	(N_0, N_2)	$(N_2, 0)$
(N_2, N_3)	$(N_2, null)$	$(N_0, 2)$
$(N_3, null)$	(N_2, N_4)	$(N_2, 1)$
$(N_4, null)$	$(N_3, null)$	$(N_3, 0)$
$(\mathbf{N_1}, \mathbf{N_3})$	$(N_4, null)$	$(N_4, 0)$
	(N_2, N_3)	$(\mathbf{N_1}, \mathbf{1})$



left	right	height
$(N_0, null)$	$(N_0, null)$	$(N_0, 0)$
(N_0, N_1)	$(N_1, null)$	$(N_0, 1)$
$(N_1, null)$	(N_0, N_1)	$(N_1, 0)$
$(N_2, null)$	(N_0, N_2)	$(N_2, 0)$
(N_2, N_3)	$(N_2, null)$	$(N_0, 2)$
$(N_3, null)$	(N_2, N_4)	$(N_2, 1)$
$(N_4, null)$	$(N_3, null)$	$(N_3, 0)$
(N_1, N_3)	$(N_4, null)$	$(N_4, 0)$
$(\mathbf{N_2}, \mathbf{N_4})$	(N_2, N_3)	$(N_1, 1)$



left	right	height
$(N_0, null)$	$(N_0, null)$	$(N_0, 0)$
(N_0, N_1)	$(N_1, null)$	$(N_0, 1)$
$(N_1, null)$	(N_0, N_1)	$(N_1, 0)$
$(N_2, null)$	(N_0, N_2)	$(N_2, 0)$
(N_2, N_3)	$(N_2, null)$	$(N_0, 2)$
$(N_3, null)$	(N_2, N_4)	$(N_2, 1)$
$(N_4, null)$	$(N_3, null)$	$(N_3, 0)$
(N_1, N_3)	$(N_4, null)$	$(N_4, 0)$
(N_2, N_4)	(N_2, N_3)	$(N_1, 1)$
	$(\mathbf{N_1}, \mathbf{N_3})$	



left	right	height
$(N_0, null)$	$(N_0, null)$	$(N_0, 0)$
(N_0, N_1)	$(N_1, null)$	$(N_0, 1)$
$(N_1, null)$	(N_0, N_1)	$(N_1, 0)$
$(N_2, null)$	(N_0, N_2)	$(N_2, 0)$
(N_2, N_3)	$(N_2, null)$	$(N_0, 2)$
$(N_3, null)$	(N_2, N_4)	$(N_2, 1)$
$(N_4, null)$	$(N_3, null)$	$(N_3, 0)$
(N_1, N_3)	$(N_4, null)$	$(N_4, 0)$
(N_2, N_4)	(N_2, N_3)	$(N_1, 1)$
	(N_1, N_3)	
	$(\mathbf{N_1}, \mathbf{N_4})$	

(b) Ejemplo de cómputo de cotas ajustadas usando bottom-up: AVLs con 5 nodos (segunda parte)

Figura 3.4: Ejemplo de cómputo de cotas ajustadas usando bottom-up: AVLs con 5 nodos

bottom-up fue desarrollado con el objetivo de minimizar la cantidad de consultas a SAT requeridas para el cómputo de cotas ajustadas. Esto es importante debido al alto costo de invocar a SAT: el problema de buscar valuaciones que satisfacen una fórmula proposicional es NP-completo. Notar que bottom-up realiza sólo 10 consultas a SAT para computar cotas ajustadas en el ejemplo anterior: una consulta por cada una de las filas de las figuras 3.4a y 3.4b, y una adicional para verificar que los conjuntos no pueden seguir extendiéndose (se convirtieron en cotas ajustadas). Es decir, este ejemplo evidencia que bottom-up realiza una cantidad de consultas a SAT significativamente menor a la cantidad de elementos en las cotas ajustadas (23 en la figura 3.3), y mucho menor que la cantidad total de posibles elementos en los dominios de los campos (85 en la figura 3.2). Esto se debe a tres motivos principales:

1. En estructuras de datos con invariantes con restricciones fuertes, como el caso de los AVLs, nuestros experimentos muestran que la cantidad de elementos en las cotas ajustadas más precisas es mucho menor que la cantidad de valores posibles en los dominios de los campos.

2. bottom-up guía la búsqueda de instancias válidas de manera que cada una de sus iteraciones agrega al menos un par a la cota ajustada que se está computando.

3. Algunas de las instancias visitadas agregan más de un valor nuevo a la cota resultante, como las de las filas 1, 2, 4, 5 de la figura 3.4a, y la fila 1 de la figura 3.4b.

Por el contrario, el algoritmo de TACO intenta resolver todas las consultas sobre la pertenencia de cada elemento individual en el dominio de un campo a la cota ajustada, en tandas, aprovechando todos los recursos disponibles –las computadoras que componen un cluster. Esto es, TACO se beneficia ampliamente de ambientes de ejecución paralelos, en donde las consultas que concluyen por la negativa pueden usarse (en forma de cotas ajustadas) para mejorar la performance de futuras tandas de invocaciones a SAT. Las observaciones anteriores implican que TACO debe realizar una mayor cantidad de llamadas a SAT que bottom-up. Nuestra evaluación experimental (capítulo 5) muestra que el enfoque bottom-up es más eficiente que TACO en ambientes de ejecución secuenciales, especialmente para estructuras de datos con invariantes restrictivos.

3.3. Implementaciones del enfoque

En esta sección se proveen diferentes versiones de algoritmos basados en el enfoque bottom-up. Estos algoritmos difieren entre sí principalmente en las consultas

que realizan al procedimiento de decisión para obtener instancias válidas del heap, lo que impacta en su performance en el cómputo de cotas ajustadas. Más adelante, en la sección 5.1.1 se evalúa el rendimiento en tiempo de las distintas versiones, para intentar elegir una de ellas como la óptima en cuanto a velocidad de cómputo de cotas.

Si bien nuestra implementación de bottom-up requiere de una especificación de las entradas válidas del programa a analizar –por ejemplo, un invariante de clases o una precondición–, las ideas subyacentes a bottom-up son independientes del lenguaje usado para describir las entradas. Es decir, bottom-up podría utilizar una especificación dada en cualquier lenguaje declarativo con suficiente poder expresivo, susceptible de análisis automático (acotado). Por lo tanto, y con motivo de simplificar la presentación, se usará un lenguaje abstracto para definir los algoritmos derivados de bottom-up en las secciones siguientes.

Durante el resto de este capítulo, se asume el modelo del heap presentado en la sección 2.3, que representa los tipos de datos como conjuntos de identificadores, y los campos como relaciones binarias (con tipos apropiados). Además, se asume un conjunto fijo de dominios de datos $T_1, \ldots, T_k$, una secuencia *scopes* con el número máximo de nodos por cada dominio –$scopes.i = \#T_i, 1 \leq i \leq k$–, y una secuencia fija de campos $f_1 : T_1' \to T_1'', ..., f_n : T_n' \to T_n''$ donde cada uno de los conjuntos con primas y doble primas es alguno de los dominios $T_1, \ldots, T_k$. En la definición de los algoritmos, el predicado que caracteriza las instancias válidas del heap se denomina $I(f_1, ..., f_n)$. Por ejemplo, si se desean computar cotas ajustadas para AVLs (sección 3.1), I sería el predicado `AVL_Invariant` de la figura 3.1. Adicionalmente, se denota $C(f_1, ..., f_n)$ a los predicados de rotura de simetría usados para canonizar el heap. Recordemos que un posible C para (Dyn)Alloy puede generarse automáticamente siguiendo el procedimiento de la sección 2.6. Así, el problema a resolver por los algoritmos de esta sección es producir un conjunto de cotas ajustadas para los campos $f_1, ..., f_n$, con un máximo de *scopes* elementos en los dominios, para un heap caracterizado por I (y C).

Al igual que en la sección anterior, los algoritmos basados en bottom-up llevan un conjunto de pares (del tipo appropiado) por cada campo del heap $f_1, \ldots, f_n$, que se denominarán $S_{f_1}, ..., S_{f_n}$ el pseudocódigo. Recordemos que bottom-up comienza inicializando todos estos conjuntos en vacío. Luego, bottom-up lleva a cabo un proceso iterativo que consiste en: (i) requerir a un procedimiento de decisión una instancia válida del heap que contenga valores de campos "frescos" – que puedan utilizarse para extender los conjuntos $S_{f_1}, ..., S_{f_n}$; (ii) agregar los valores frescos a los conjuntos correspondientes. Este proceso finaliza cuando no existen instancias que permitan extender $S_{f_1}, ..., S_{f_n}$; en este punto $S_{f_1}, ..., S_{f_n}$ son cotas ajustadas para los campos $f_1, \ldots, f_n$.

Algorithm 1 Algoritmo BOTTOM-UP

1: **function** BOTTOM-UP($f_1, ..., f_n, scopes$)
2: $\quad S_{f_1}, ..., S_{f_n} = \emptyset, \ldots, \emptyset$
3: $\quad$ **while** $True$ **do**
4: $\quad\quad \phi = I(f_1, ..., f_n) \wedge C(f_1, ..., f_n) \wedge (\bigvee_{j \in 1,..,n}(f_j \nsubseteq S_{f_j}))$
5: $\quad\quad$ **if** $SAT(\phi, scopes, instance)$ **then**
6: $\quad\quad\quad extend(S_{f_1}, ..., S_{f_n}, instance)$
7: $\quad\quad$ **else**
8: $\quad\quad\quad$ **break**
9: $\quad$ **return** $S_{f_1}, ..., S_{f_n}$

En el pseudocódigo de los algoritmos la etapa (i) se resuelve mediante un procedimiento de decisión basado en SAT solving. De esta manera, una invocación al procedimiento de decisión para averiguar sobre la satisfactibilidad de una fórmula ϕ, dentro de los scopes *scopes*, se denota $SAT(\phi, scopes, instance)$. En caso de que una invocación a SAT retorne verdadero, la asignación que satisface la fórmula ϕ –una instancia válida del heap– se accede a través del último parámetro, *instance*. En caso contrario, el valor de *instance* es indefinido. Para realizar el paso (ii) se asume un procedimiento $extend(S_{f_1}, ..., S_{f_n}, instance)$, cuya responsabilidad es agregar los pares frescos de la instancia válida *instance* a los conjuntos correspondientes $S_{f_1}, ..., S_{f_n}$. Por ejemplo, tomando como punto de partida a los conjuntos $S_{left}, S_{right}, S_{height}$ tabulados a la derecha, en la segunda fila de la figura 3.4a, y suponiendo que *instance* es el AVL a la izquierda de la tercera fila, $extend(S_{left}, S_{right}, S_{height}, instance)$ agrega el par (N_0, N_1) a S_{right}, dando como resultado los conjuntos mostrados a la derecha, en la tercera fila de la figura.

Por último, es importante destacar que todos los algoritmos definidos en esta sección computan las cotas ajustadas más precisas para el heap (ver 2.7.2).

3.3.1. Algoritmo estándar

Como se mencionó anteriormente, la diferencia principal entre las distintas versiones de bottom-up reside en la forma de consultar al procedimiento de decisión por instancias del heap. En el caso de la versión "estándar", denominada como el enfoque: BOTTOM-UP, en cada paso del algoritmo se pide una instancia que extienda *al menos uno* de los conjuntos $S_{f_1}, ..., S_{f_n}$. Un pseudocódigo de BOTTOM-UP se presenta en el algoritmo 1.

La línea 2 de la figura corresponde a la inicialización del algoritmo: por cada campo f_i se inicializa un conjunto S_i en vació. El ciclo principal de la línea 3 im-

plementa el procedimiento iterativo que comprende invocar a SAT para obtener instancias del heap, y extender los conjuntos $S_{f_1}, ..., S_{f_n}$ con los valores frescos de las instancias. El paso esencial del ciclo se muestra en la línea 4: se construye una fórmula ϕ formada por la conjunción de I, C y la fórmula $(\bigvee_{j \in 1,..,n}(f_j \nsubseteq S_{f_j}))$. Los dos primeros conyungendos caracterizan al conjunto de instancias canónicas válidas del heap, mientras que el tercero exige instancias que contengan al menos un valor para extender a alguno de los conjuntos $S_{f_1}, ..., S_{f_n}$. Por ejemplo, para AVL, I es `AVL_Invariant` (figura 3.1), y el tercer conyungendo de ϕ es la fórmula:

$$\texttt{left} \nsubseteq S_{left} \vee \texttt{right} \nsubseteq S_{right} \vee \texttt{height} \nsubseteq S_{height} \tag{3.1}$$

Posteriormente, en la línea 5 se utiliza ϕ como entrada de SAT. Si existe alguna instancia con las características requeridas por ϕ, se añaden los valores correspondientes a $S_{f_1}, ..., S_{f_n}$ en la línea 6, mediante *extend*. En caso contrario, no hay instancias del heap que sirvan para extender $S_{f_1}, ..., S_{f_n}$, y el ciclo principal termina. En este punto, $S_{f_1}, ..., S_{f_n}$ son las cotas ajustadas más precisas para el heap, que se devuelven como resultado en la línea 11.

Nuestra implementación de Bottom-up toma como entradas especificaciones Alloy de I y C. El tercer conyungendo de la fórmula de la línea 4 se codifica directamente en Alloy. Por ejemplo, la traducción de la fórmula 3.1 a Alloy es:

`left not in` S_{left} `|| right not in` S_{right} `|| height not in` S_{height}

Así, SAT se implementa a través del Alloy Analyzer. Por último, nuestra implementación usa las asignaciones que satisfacen la especificación Alloy –retornadas por el Alloy Analyzer– para agregar elementos a los conjuntos $S_{f_1}, ..., S_{f_n}$. Las implementaciones de los restantes algoritmos derivados de bottom-up, presentados en las próximas secciones, también están basadas en especificaciones en el lenguaje Alloy. Consecuentemente, no se discutirán detalles de implementación de los mismos, a menos que las diferencias respecto de lo discutido aquí así lo ameriten.

El ejemplo de la figura 3.4 corresponde a una ejecución real de Bottom-up para la especificación de AVL de la sección 3.1, utilizando un máximo de 5 nodos para los árboles. En particular, la fila i-ésima de la figura muestra la instancia retornada por $SAT(\phi, scopes, instance)$ en la iteración i-ésima del ciclo principal de Bottom-up, y el estado de los conjuntos $S_{f_1}, ..., S_{f_n}$ al final de la iteración.

3.3.2. Extendiendo todos los conjuntos a la vez

El algoritmo presentado en esta sección es una simple modificación de Bottom-up (algoritmo 1). Con la intención de reducir aún más la cantidad de invocaciones a

Algorithm 2 BOTTOM-UP-$\wedge$ algorithm

1: **function** BOTTOM-UP-$\wedge(f_1, ..., f_n, scopes)$
2: $\quad S_{f_1}, ..., S_{f_n} = \emptyset, \ldots, \emptyset$
3: $\quad \oplus = \wedge$
4: $\quad$ **while** $True$ **do**
5: $\quad\quad \phi = I(f_1, ..., f_n) \wedge C(f_1, ..., f_n) \wedge (\bigoplus_{j \in 1,..,n}(f_j \nsubseteq S_{f_j}))$
6: $\quad\quad$ **if** $SAT(\phi, scopes, instance)$ **then**
7: $\quad\quad\quad extend(S_{f_1}, ..., S_{f_n}, instance)$
8: $\quad\quad$ **else**
9: $\quad\quad\quad$ **if** $\oplus = \wedge$ **then**
10: $\quad\quad\quad\quad \oplus = \vee$
11: $\quad\quad\quad$ **else**
12: $\quad\quad\quad\quad$ **break**
13: $\quad$ **return** $S_{f_1}, ..., S_{f_n}$

SAT, se propone modificar BOTTOM-UP para que, en cada paso de su ciclo principal, pida al procedimiento de decisión instancias del heap que sean útiles para extender todos los conjuntos $S_{f_1}, ..., S_{f_n}$ al mismo tiempo. La intuición es que, al agregar más valores a los conjuntos en cada iteración, se deberían reducir la cantidad total de invocaciones a SAT necesarias para computar cotas ajustadas.

El pseudocódigo resultante, BOTTOM-UP-$\wedge$, se muestra en el algoritmo 2. La ejecución de BOTTOM-UP-$\wedge$ se divide en dos etapas. En la primera etapa se piden al procedimiento de decisión instancias que agreguen *al menos un par a cada uno* de los conjuntos $S_{f_1}, ..., S_{f_n}$ – como se mencionó anteriormente. En la segunda etapa, BOTTOM-UP-$\wedge$ se comporta exactamente como BOTTOM-UP, con el objetivo de completar la cota ajustada con los valores de campos de las instancias válidas que sólo pueden sirven para extender uno de los conjuntos $S_{f_1}, ..., S_{f_n}$.

Así, BOTTOM-UP-$\wedge$ construye, en la línea 5, una fórmula ϕ diferente de acuerdo a la etapa en la que se encuentra. Durante la primera etapa, $\oplus$ es el operador lógico $\wedge$, debido a su inicialización en la línea 3. Por lo tanto, el tercer conyungendo de ϕ –en la línea 5– es $(\bigwedge_{j \in 1,..,n}(f_j \nsubseteq S_{f_j}))$, es decir, el algoritmo intenta extender todos los conjuntos a la vez. Para el caso de estudio AVL, el tercer conyungendo de ϕ en la primera etapa de BOTTOM-UP-$\wedge$ es:

$$\texttt{left} \nsubseteq S_{left} \wedge \texttt{right} \nsubseteq S_{right} \wedge \texttt{height} \nsubseteq S_{height}$$

La primera etapa finaliza cuando no existen más instancias que satisfacen la fórmula ϕ anterior –línea 6–. En tal caso, BOTTOM-UP-$\wedge$ cambia el valor de $\oplus$ a $\vee$ en la línea 10, ingresando así en la segunda etapa de su ejecución.

Algorithm 3 BOTTOM-UP-F algorithm

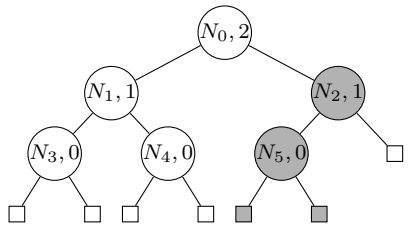
1: **function** BOTTOM-UP-F($f_1, ..., f_n, scopes$)
2: $S_{f_1}, ..., S_{f_n} = \emptyset, \ldots, \emptyset$
3: **for** $i \in 1 \ldots n$ **do**
4: **while** $True$ **do**
5: $\phi = I(f_1, ..., f_n) \wedge C(f_1, ..., f_n) \wedge f_i \nsubseteq S_{f_i}$
6: **if** $SAT_B(\phi, scopes, instance, S_{f_1}, ..., S_{f_{i-1}})$ **then**
7: $extend(S_{f_1}, ..., S_{f_n}, instance)$
8: **else**
9: **break**
10: **return** $S_{f_1}, ..., S_{f_n}$

3.3.3. Cómputo campo por campo

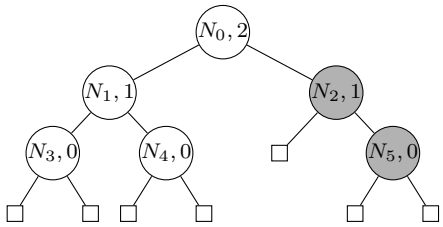
Los algoritmos presentados hasta el momento computan cotas ajustadas, pero, a diferencia del algoritmo paralelo de TACO, no las aprovechan para mejorar la performance de las sucesivas invocaciones a SAT. El tercer algoritmo derivado de bottom-up surge de observar que cuando se completa el cómputo de una cota ajustada para un campo f, esta puede usarse como ajustada para f, y así mejorar la eficiencia de las llamadas posteriores a SAT –para concluir el cómputo de cotas para los campos restantes. La idea es, entonces, computar sucesivamente cotas ajustadas para los campos f_1, f_2, etc., usando durante el cómputo para f_i las cotas ajustadas ya computadas para $f_1, \ldots, f_{i-1}$.

La definición del nuevo algoritmo requiere de un procedimiento de decisión capaz de aprovechar cotas ajustadas para acelerar su propia ejecución. En el pseudocódigo, este procedimiento se denomina $SAT_B(\phi, scopes, instance,\ S_{f_1}, ..., S_{f_{i-1}})$, en donde ϕ, *scopes* e *instance* son las mismas que para SAT, y $S_{f_1}, ..., S_{f_{i-1}}$ son cotas ajustadas para los campos $f_1, \ldots, f_{i-1}$, respectivamente. Recordemos de la sección 3.3.1 que SAT se implementa a través del Alloy Analyzer. Como se discutió en la sección 2.4, el Alloy Analyzer es capaz de explotar cotas ajustadas a nivel de KodKod, y, por lo tanto, también se usa para implementar de SAT_B.

El pseudocódigo de BOTTOM-UP-F –bottom-up por campo– se muestra en el algoritmo 3. En la i-ésima iteración de su ciclo externo –línea 3–, BOTTOM-UP-F computa una cota ajustada para el campo f_i. Para lograr esto, el ciclo interno –línea 4– pide a SAT_B instancias que le permitan completar la cota ajustada para f_i, esto es, instancias con un valor de f_i que sirva para extender $S_{f_{i-1}}$ –$f_i \nsubseteq S_{f_i}$, en el tercer conyungendo de la línea 5 del algoritmo. Para estas consultas se aprovechan las cotas ajustadas ya computadas para los campos $f_1, \ldots, f_{i-1}$, lo que permite mejorar



left	right	height
$(N_0, null)$	$(N_0, null)$	$(N_0, 0)$
(N_0, N_1)	$(N_1, null)$	$(N_0, 1)$
$(N_1, null)$	(N_0, N_1)	$(N_1, 0)$
$(N_2, null)$	(N_0, N_2)	$(N_2, 0)$
(N_2, N_3)	$(N_2, null)$	$(N_0, 2)$
$(N_3, null)$	(N_2, N_4)	$(N_2, 1)$
$(N_4, null)$	$(N_3, null)$	$(N_3, 0)$
(N_1, N_3)	$(N_4, null)$	$(N_4, 0)$
(N_2, N_4)	(N_2, N_3)	$(N_1, 1)$
$(\mathbf{N_2}, \mathbf{N_5})$	(N_1, N_3)	$(\mathbf{N_5}, \mathbf{0})$
$(\mathbf{N_5}, \mathbf{null})$	(N_1, N_4)	
	$(\mathbf{N_5}, \mathbf{null})$	



left	right	height
$(N_0, null)$	$(N_0, null)$	$(N_0, 0)$
(N_0, N_1)	$(N_1, null)$	$(N_0, 1)$
$(N_1, null)$	(N_0, N_1)	$(N_1, 0)$
$(N_2, null)$	(N_0, N_2)	$(N_2, 0)$
(N_2, N_3)	$(N_2, null)$	$(N_0, 2)$
$(N_3, null)$	(N_2, N_4)	$(N_2, 1)$
$(N_4, null)$	$(N_3, null)$	$(N_3, 0)$
(N_1, N_3)	$(N_4, null)$	$(N_4, 0)$
(N_2, N_4)	(N_2, N_3)	$(N_1, 1)$
(N_2, N_5)	(N_1, N_3)	$(N_5, 0)$
$(N_5, null)$	(N_1, N_4)	
	$(N_5, null)$	
	$(\mathbf{N_2}, \mathbf{N_5})$	

Figura 3.5: Ejemplo de cómputo incremental de cotas ajustadas usando Bottom-up-inc: AVLs para un máximo de 6 nodos, partiendo de las cotas ajustadas para AVLs y scope 5 (figura 3.3)

la performance del procedimiento de decisión de SAT_B.

La evaluación experimental de la sección 5.1.1 muestra que esta versión de bottom-up, que aprovecha la idea de cotas ajustadas, requiere de un tiempo de cómputo significativamente menor a las anteriores, sobre todo cuando se consideran scopes medianos o "grandes".

3.4. Cómputo incremental de cotas ajustadas

Otra de las contribuciones importantes de esta tesis, el cómputo incremental de cotas ajustadas, apunta a mitigar el alto costo de computar cotas ajustadas en los casos en que se realiza sucesivos análisis de la misma especificación con scopes crecientes. Este es un escenario típico en el contexto de análisis acotado, y puede resumirse como sigue:

1. Elegir un scope pequeño para comenzar el análisis.

Algorithm 4 Algoritmo BOTTOM-UP-INC

1: **function** BOTTOM-UP-INC($f_1, ..., f_n, scopes, smaller_b$)
2: $S_{f_1}, ..., S_{f_n} = smaller_b$
3: **while** $True$ **do**
4: $\phi = I(f_1, ..., f_n) \wedge C(f_1, ..., f_n) \wedge (\bigvee_{j \in 1,..,n}(f_j \nsubseteq S_{f_j}))$
5: **if** $SAT(\phi, scopes, instance)$ **then**
6: $extend(S_{f_1}, ..., S_{f_n}, instance)$
7: **else**
8: **break**
9: **return** $S_{f_1}, ..., S_{f_n}$

2. Ejecutar el análisis con el scope seleccionado. Si se encuentra una error en la especificación, o el análisis no concluye en un tiempo razonable, terminar. Sino, continuar con el paso siguiente.

3. Elegir un scope más grande que el anterior y volver a 2.

Siguiendo los pasos anteriores, el usuario obtiene un testigo de una violación – que debe corregir si desea reiniciar el análisis –, o adquiere un mayor nivel de confianza en su especificación/software –la máxima que el análisis acotado puede garantizar en un período de tiempo aceptable para el usuario.

El problema es que, para realizar los sucesivos análisis del esquema anterior aprovechando la idea de cotas ajustadas, con las técnicas existentes se debe comenzar el cómputo de las cotas desde cero cada vez, lo que resulta costoso en tiempo. Por lo tanto, en esta sección se propone un enfoque para el *cómputo incremental de cotas ajustadas*, que consiste en utilizar las cotas computadas para algún scope más pequeño al momento de computar cotas para scope mayor, eliminando la necesidad de comenzar el cómputo de cotas desde el principio cada vez. Intuitivamente, la idea subyancente al cómputo incremental basado en bottom-up es que los valores que aparecen en una cota más pequeña deben forsozamente ocurrir en la más grande, dado que las instancias del heap que se visitaron para construir la cota pequeña son también instancias válidas para el heap con scope mayor –los scopes son no estrictos, es decir, un scope k para un dominio implica que se pueden usar hasta k elementos del dominio para generar instancias. El objetivo de este enfoque es reducir la cantidad de llamadas a SAT requeridas para computar las cota ajustadas para el scope mayor.

Notar que bottom-up es una técnica inherentemente incremental, por lo que permite implementar el cómputo incremental de cotas ajustadas de una manera sencilla. Para esto, se propone modificar bottom-up para que, en caso de disponer de una

cota ajustada para un scope más pequeño, la utilice como punto de partida, es decir, como *inicialización* para los conjuntos $S_{f_1}, ..., S_{f_n}$. Notar que, esta modificación sirve para implementar el cómputo incremental en cualquiera de los algoritmos derivados de bottom-up. Por ejemplo, el algoritmo 4, denominado BOTTOM-UP-INC, es la versión incremental de BOTTOM-UP (algoritmo 1). Respecto de BOTTOM-UP, BOTTOM-UP-INC toma un parámetro adicional, la cota ajustada para el scope más pequeño *smaller_b*, y la usa para inicializar los conjuntos $S_{f_1}, ..., S_{f_n}$ en la línea 2. La precondición de BOTTOM-UP-INC requiere que *smaller_b* sea la cota ajustada más precisa para el heap con scopes sc', donde cada valor en sc' debe ser menor que su correspondiente en *scopes* ($sc' = [s'_1, \ldots, s'_k]$, $scopes = [s_1, \ldots, s_k]$ y $s'_1 < s_1, \ldots, s'_k < s_k$). A partir de la línea 3, BOTTOM-UP-INC se comporta exactamente como su contraparte no incremental.

Como ejemplo del enfoque incremental, consideremos extender la cota ajustada de AVLs con scope 5, mostrada en la figura 3.3, para obtener cotas ajustadas para scope 6, utilizando BOTTOM-UP-INC. La figura 3.5 muestra el proceso completo de cómputo efectuado por BOTTOM-UP-INC. En este caso, a partir de las cotas iniciales, BOTTOM-UP-INC sólo visita dos instancias adicionales para obtener las cotas para el scope más grande. En cambio, si se utiliza el algoritmo no incremental, BOTTOM-UP, para computar cotas ajustadas para scope 6 empezando desde cero, se visitan 12 instancias durante el proceso de cómputo. Es decir, se reduce de 13 a 3 la cantidad de invocaciones a SAT al realizar el cómputo de manera incremental. Además, es importante destacar que, en este caso, BOTTOM-UP efectúa la misma cantidad de consultas a SAT para computar cotas de tamaño 6 que BOTTOM-UP-INC para producir secuencialmente cotas de tamaños 5 y 6 (13 en total en ambos casos). Esta tendencia se mantiene en los casos de estudio considerados en la evaluación experimental de la sección 5.1.2, lo que provoca que, en general, BOTTOM-UP-INC sea capaz de computar cotas ajustadas para scopes cada vez mayores hasta tamaño k, en el mismo tiempo en que BOTTOM-UP computa cotas sólo para scope k.

3.5. Interacción entre bottom-up y TACO

En esta sección se discute como incorporar bottom-up como algoritmo de cómputo de cotas ajustadas en el verificador de propiedades de programas basado en SAT TACO, presentado en la sección 2.5.1. Al igual que el algoritmo paralelo de TACO+ (sección 2.8), bottom-up requiere de especificaciones Alloy para computar las cotas ajustadas –que se derivan automáticamente de los invariantes de clase JML. Así, los predicados de rotura de simetrías BF de la sección 2.6.1 son apropiados para instrumentar la especificación Alloy que usa bottom-up para realizar el cómputo

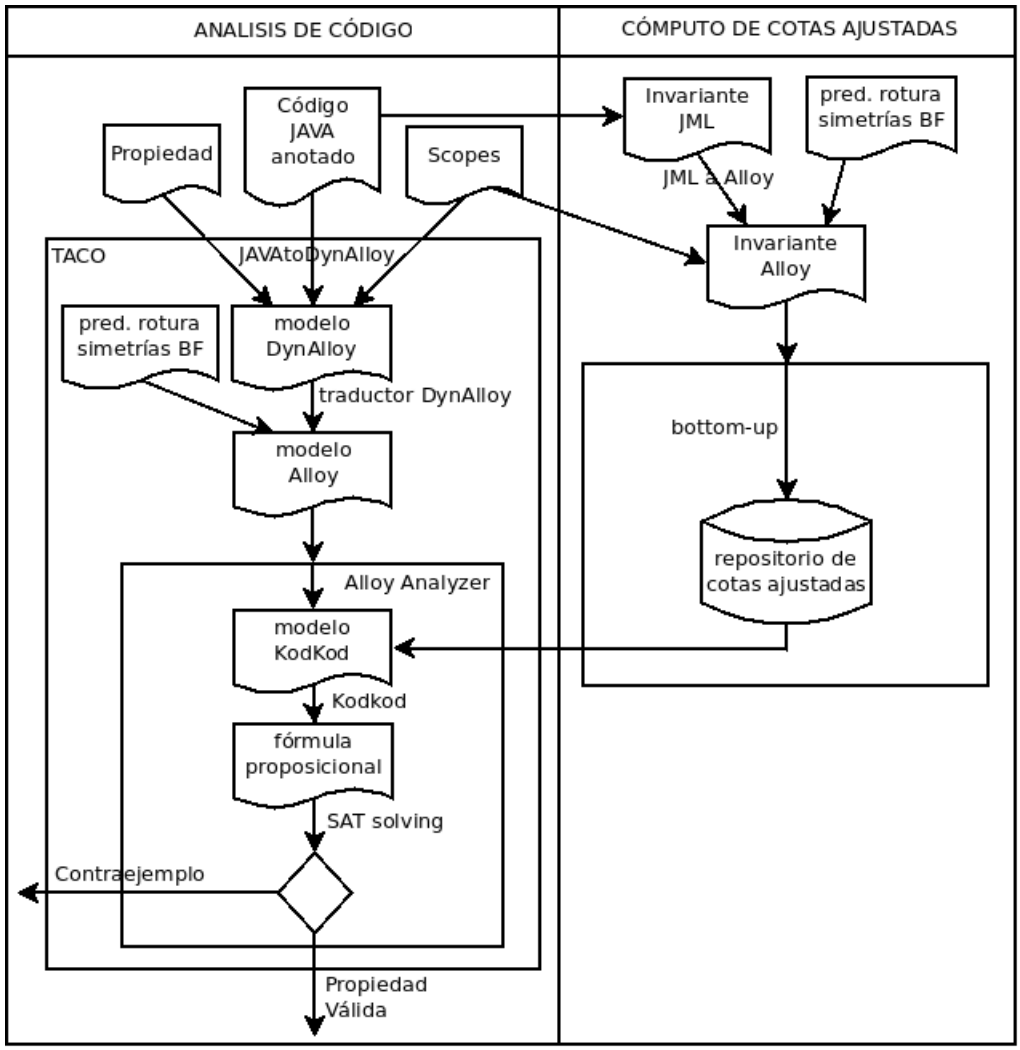


Figura 3.6: Diagrama del análisis de código bottom-up+TACO.

de cotas ajustadas. Usando dichos predicados bottom-up produce cotas ajustadas BF, al igual que TACO+. Es más, para una misma especificación JML (e idénticos scopes), bottom-up y TACO+ generan cotas ajustadas (BF) equivalentes.

De la discusión anterior se desprende que una forma de incorporar bottom-up en TACO consiste en reemplazar en el análisis TACO+ (sección 2.8, figura 2.9) el algoritmo de cómputo de cotas ajustadas paralelo por alguna de las implementaciones de bottom-up. Un esquema del análisis de programas resultante se muestra en la figura 3.6. Llamaremos a esta técnica bottom-up+TACO, ya que comprende el cómputo de cotas ajustadas mediante bottom-up y el posterior análisis de código instrumentado

con las cotas ajustadas. Observar en la figura que, como en TACO+, la verificación de código también se instrumenta con predicados de rotura de simetrías BF, para mejorar la performance del proceso de verificación.

Capítulo 4

Cómputo secuencial de cotas ajustadas basado en predicados inductivos de la lógica de separación

La técnica bottom-up presentada en el capítulo anterior es aplicable cuando se cuenta con predicados declarativos que definen una estructura de datos, como lo son, por ejemplo, las especificaciones JML y Alloy. En este capítulo se consideran especificaciones dadas en términos de predicados inductivos de la lógica de separación (predicados inductivos de ahora en adelante), introducidos en la sección 2.9. Estos predicados tienen características particulares que hacen más eficiente el cómputo de cotas ajustadas. Es importante destacar que este capítulo presenta las ideas publicadas en [41] en mayor detalle. A continuación, se describe la organización del capítulo.

Para facilitar la comprensión de los algoritmos posteriores, en la sección 4.1 se define un enfoque de cómputo de cotas ajustadas –basado en predicados inductivos– por *fuerza bruta*. Este consiste en construir todos los heaps simbólicos que se obtienen de realizar hasta un máximo k de desplegados del predicado inductivo. En las secciones siguientes se presentan dos mejoras a esta técnica inicial. Primero, en la sección 4.2 se presenta el enfoque SLBD (*separation logic bounds*), que computa cotas ajustadas *al vuelo* mientras realiza desplegados del predicado inductivo de entrada, evitando la construcción del conjunto completo de heaps simbólicos mencionado anteriormente. Esto permite disminuir sustancialmente los requerimientos de memoria del enfoque, ya que la cantidad de heaps simbólicos construíbles expandiendo hasta k veces un predicado inductivo puede ser exponencial en k. Segundo,

$\texttt{emp} \wedge l = null$	l
$l \mapsto n : l_1 \wedge l_1 = null$	l l_1
$l \mapsto n : l_1 * l_1 \mapsto n : l_2 \wedge l_2 = null$	l l_1 l_2
$l \mapsto n : l_1 * l_1 \mapsto n : l_2 * l_2 \mapsto n : l_3$ $\wedge\, l_3 = null$	l l_1 l_2 l_3

Figura 4.1: Heaps simbólicos obtenidos de expandir `slist` (izquierda) y su representación gráfica (derecha). La fila i-esima muestra el resultado de expandir `slist` $i-1$ veces.

en 4.3 se adapta el enfoque SLBD para soportar la técnica de optimización de algoritmos conocida como *memorización*. Esta consiste en almacenar las soluciones a subproblemas del problema original, computadas recursivamente, para ser utilizadas posteriormente por llamadas recursivas que resuelven el mismo subproblema. Esta optimización produce una mejora sustancial en el tiempo de ejecución del algoritmo.

SLBD rompe simetrías etiquetando los elementos del heap (desde las raíces) en un recorrido primero en profundidad de los campos (*depth first*), y, por lo tanto, computa cotas ajustadas primero en profundidad. Esto implica que, en algunos casos, las cotas ajustadas generadas por SLBD son diferentes a las producidas por bottom-up o TACO+ (cotas ajustadas BF). La sección 4.4 explica la idea de cotas ajustadas primero en profundidad. Además, introduce los predicados de rotura de simetrías DF (primero en profundidad), compatibles con esta nueva variante de cotas ajustadas. Finalmente, en la sección 4.5 se explica como incorporar SLBD al análisis de código TACO, lo que involucra instrumentar el análisis con cotas ajustadas DF y predicados de rotura de simetrías DF.

4.1. Fuerza bruta

Para ilustrar el cómputo de cotas ajustadas basado en predicados inductivos nos basaremos en los predicados inductivos del ejemplo 2.9.5, que reproducimos a continuación:

Ejemplo 4.1.1. Los predicados `slist`(l), `btree`(t) y `avl`(t, h) definen listas enla-

zadas acíclicas simplemente encadenadas, árboles binarios y árboles binarios balanceados, respectivamente. l y t –para listas y árboles, respectivamente– se consideran variables del programa a analizar que hacen referencia a los nodos raíz de las estructuras.

$$
\begin{aligned}
\texttt{slist}(l) &\doteq (\texttt{emp} \wedge l = null) \vee (l \mapsto n : l_1 * \texttt{slist}(l_1)) \\
\texttt{btree}(t) &\doteq (\texttt{emp} \wedge t = null) \vee (t \mapsto l : t_1, r : t_2 * \texttt{btree}(t_1) * \texttt{btree}(t_2)) \\
\texttt{avl}(t, h) &\doteq (\texttt{emp} \wedge t = null \wedge h = 0) \vee \\
&\qquad (t \mapsto l : t_1, r : t_2, ht : h * \texttt{avl}(t_1, h_1) * \texttt{avl}(t_2, h_2) \wedge \\
&\qquad\qquad h = 1 + max(h_1, h_2) \wedge |h_1 - h_2| \leq 1)
\end{aligned}
$$

□

El primer paso en la definición de nuestro enfoque es observar que el resultado de desplegar un predicado inductivo p una cierta cantidad de veces es un conjunto de heaps simbólicos, donde cada uno de ellos describe un conjunto de instancias de la estructura de datos definida por p. Es importante destacar que los algoritmos desarrollados en este capítulo no computarán cotas ajustadas para los datos almacenados en las estructuras. De cualquier manera, la utilidad de las mismas en la práctica es limitada, debido a que una gran cantidad de estructuras de datos imponen restricciones débiles sobre los datos, resultando en cotas ajustadas poco precisas (que no permiten reducir significativamente la cantidad de posibles valores para los campos). Por esta razón, los campos que almacenan datos no se modelan en los predicados del ejemplo 4.1.1.

Ejemplo 4.1.2. La fila i-ésima de la figura 4.1 muestra el heap simbólico resultante de desplegar el predicado `slist` (ej. 4.1.1) $i - 1$ veces (izquierda); junto con una representación gráfica del heap (derecha). Por ejemplo, la primera fila corresponde a desplegar el predicado 0 veces, es decir, a retornar su caso base, que describe a un heap vacío (con l tomando el valor *null*). En cambio, la tercera fila realiza dos expansiones del caso inductivo del predicado, resultando en un heap con dos nodos, referenciados por l y l_1, con el campo siguiente del último nodo apuntando a *null*. En general, desplegar i veces el predicado `slist` produce un único heap simbólico con exactamente i nodos. □

Del ejemplo anterior se puede inferir una forma sencilla y directa de computar cotas ajustadas, a la que llamaremos enfoque por *fuerza bruta*. La primera etapa de este enfoque consiste en construir todos los heaps posibles, desplegando el predicado de entrada, p, exactamente k veces. Esta etapa se discute en la sección 4.1.1. Cualquiera de los heaps simbólicos en la parte izquierda de la figura 4.1 ejemplifican

Algorithm 5 Algoritmo DESPLEGAR

1: **function** DESPLEGAR(p, k)
2: **if** $k = 0$ **then**
3: ▷ Retornar el heap simbólico correspondiente al caso base de p
4: **let** BASE$(p) = \texttt{emp} \wedge \Pi_b$ **in**
5: **return** $\{\texttt{emp} \wedge \Pi_b\}$
6: ▷ $k > 0$
7: $res = \emptyset$
8: **let** IND$(p(v^*)) = v \mapsto f_{i_1} : v_1, \ldots, f_{i_m} : v_m * p_1(y_1^*) * \ldots * p_i(y_i^*) \wedge \Pi_{ind}$ **in**
9: ▷ Considerar todas las formas posibles de repartir k entre las llamadas recursivas
10: **for** $(k_1, \ldots, k_i) \in$ REPARTIR$(k-1, i)$ **do**
11: $set_1 =$DESPLEGAR$(p_1(y_1^*), k_1)$
12: $\ldots$
13: $set_i =$DESPLEGAR$(p_i(y_i^*), k_i)$
14: **for** $s_1, \ldots, s_i \in set_1 \times \ldots \times set_i$ **do**
15: $res = res \cup \{$COMBINAR$(v \mapsto v_1, \ldots, v_m \wedge \Pi_{ind}, s_1, \ldots, s_i)\}$
16: **return** res

esta etapa. La segunda etapa involucra generar instancias concretas de la estructura definida por p a partir de los heaps simbólicos. El procedimiento se describe en detalle en la sección 4.1.2. En nuestro ejemplo, para cada heap simbólico $\mathcal{SH}$ deberían generarse todas las instancias que respeten la forma del heap que se muestra a la derecha de $\mathcal{SH}$ en la figura 4.1. Finalmente, todos los valores de campos de las instancias generadas conforman cotas ajustadas para el heap. El algoritmo para construir las cotas se define en la sección 4.1.5.

4.1.1. Desplegando predicados inductivos

En esta sección se formaliza el concepto de desplegado de un predicado inductivo p. Durante esta sección se asumen un conjunto fijo de campos $f_1, \ldots, f_n$ para el heap. Para simplificar la presentación de los algoritmos, se asumen predicados inductivos con la forma presentada a continuación:

$$p(v^*) \doteq (\texttt{emp} \wedge \Pi_b) \vee (v \mapsto f_{i_1} : v_1, \ldots, f_{i_m} : v_m * p_1(y_1^*) * \ldots * p_i(y_i^*) \wedge \Pi_{ind}) \quad (4.1)$$

El predicado p posee una lista de parámetros v^* que necesariamente incluye a v como primer parámetro (es decir, la raíz de la estructura es el primer parámetro del predicado), un caso base (que aparece a la izquierda de $\vee$) y un único caso inductivo (a la derecha de $\vee$). En el caso base, las variables libres de Π_b deben aparecer

en v^*. El caso inductivo de p está compuesto por un único objeto apuntado por la variable v (al que denominaremos raíz de la estructura), y llamadas inductivas a los predicados $p_1, \ldots, p_i$, con $i \geq 0$ (donde cualquier p_j puede ser distinto de p). En el objeto referenciado por v, los campos $f_{i_1}, \ldots, f_{i_m}$ son un subconjunto de $f_1, \ldots, f_n$, y las variables $v_1, \ldots, v_m$ están contenidas en la lista v^* o se encuentran existencialmente cuantificadas. Además, tanto los parámetros de las llamadas inductivas $y_1^*, \ldots, y_i^*$ como las variables libres de Π_{ind} son o bien parte de v^*, o son alguna de las variables $v_1, \ldots, v_m$. Sin embargo, v no puede ser el primer parámetro de ninguna de las llamadas inductivas $p_1, \ldots, p_i$, lo que evita predicados que admiten un número infinito de desplegados. Las restricciones anteriores implican que todos los nodos de la estructura que se generan a partir de desplegar el predicado inductivo son alcanzables desde la raíz (v).

Ejemplo 4.1.3. Los predicados `slist`, `btree` y `avl` del ejemplo 4.1.1 satisfacen la forma restringida (4.1). □

Ahora estamos listos para definir el algoritmo Desplegar, que se muestra en la figura 5. Desplegar toma como entradas un predicado inductivo p y un número k indicando la cantidad de veces que se desea desplegar p, y retorna el conjunto de heaps simbólicos resultantes de desplegar p –exactamente– k veces. El algoritmo utiliza las funciones Base e Ind para obtener los heaps simbólicos correspondientes a los casos base e inductivo de p, respectivamente. La idea intuitiva de Desplegar es que, debido a la forma que se asume para p, desplegar k veces p debe retornar sólo heaps simbólicos que representan heaps con exactamante k objetos (esta restricción se deshechará cuando definamos la mejor versión de nuestro algoritmo de cómputo de cotas ajustadas, en la sección 4.2). Así, desplegar 0 veces p ($k = 0$) implica retornar el caso base de p (línea 4 de la fig. 5). Para el caso inductivo (líneas 7-15), deben considerarse todas las formas posibles de repartir los $k - 1$ desplegados (la llamada actual se considera un desplegado) entre las llamadas recursivas, que computarán conjuntos de heaps simbólicos para los predicados $p_1, \ldots, p_i$. Por ejemplo, para el predicado `btree` (ejemplo 4.1.1) y $k - 1 = 1$, deben tenerse en cuenta los casos en que se realizan 0 desplegados del predicado que representa al árbol izquierdo y 1 del correspondiente al árbol derecho, y el caso inverso (0 desplegados del árbol izquierdo y 1 desplegado del árbol derecho). Esta tarea la lleva a cabo la función Repartir (línea 9): Repartir$(k - 1, i)$ retorna todas las formas posibles de dividir los $k - 1$ desplegados entre las i llamadas recursivas. Repartir se define como:

$$\textsc{Repartir}(k, i) = \{(a_1, \ldots, a_i) | a_1 + \ldots + a_i = k, a_j \geq 0 \text{ para } j = 1 \ldots i\}.$$

Una vez fijadas la cantidad de desplegados para cada predicado inductivo $(k_1, \ldots, k_i)$, el algoritmo construye recursivamente para cada p_j ($j = 1 \ldots i$) el conjunto de heaps

simbólicos set_j (líneas 10-12): todos los obtenibles desplegando p_j k_j veces. Finalmente, los heaps simbólicos que forman parte del resultado de DESPLEGAR se obtienen combinando cada tupla de shapes $s_1, \ldots, s_i \in set_1 \times \ldots \times set_i$ con el heap simbólico para la raíz $(v \mapsto f_{i_1} : v_1, \ldots, f_{i_m} : v_m)$ y la parte pura de p (Π_{ind}) (líneas 13-14). Dada una secuencia de heaps simbólicos, la función COMBINAR retorna un único heap simbólico cuya parte espacial es la conjunción vía $*$ de las partes espaciales de las entradas, y cuya parte pura es la conjunción lógica ($\wedge$) de las partes puras de las entradas. Formalmente:

$$\textsc{Combinar}(\Sigma_1 \wedge \Pi_1, \ldots, \Sigma_j \wedge \Pi_j) = (\Sigma_1 * \ldots * \Sigma_j) \wedge (\Pi_1 \wedge \ldots \wedge \Pi_j)$$

Ejemplo 4.1.4. A continuación se describe en detalle la ejecución de DESPLEGAR($\texttt{slist}(l), 2$). De la definición de `slist` (ej. 4.1.1) tenemos que:

$$\textsc{Ind}(\texttt{slist}(l)) = l \mapsto n : l_1 * \texttt{slist}(l_1)$$

Es decir, DESPLEGAR lleva a cabo el desplegado actual, y asigna el restante $(2-1)$ a la invocación recursiva con $\texttt{slist}(l_1)$. El resultado de dicha invocación es:

$$\textsc{Desplegar}(\texttt{slist}(l_1), 1) = \{l_1 \mapsto n : l_2 \wedge l_2 = null\}$$

Luego, DESPLEGAR combina la cabeza de la lista l ($l \mapsto n : l_1$) con el heap simbólico retornado por la llamada recursiva anterior, obteniendo:

$$\begin{aligned}
&\textsc{Combinar}(l \mapsto n : l_1 \wedge \texttt{true}, l_1 \mapsto n : l_2 \wedge l_2 = null) \\
&= \\
&(l \mapsto n : l_1 * l_1 \mapsto n : l_2) \wedge l_2 = null
\end{aligned}$$

que es el único heap simbólico en el resultado de DESPLEGAR($\texttt{slist}(l), 2$), y se corresponde con el mostrado en la segunda fila de la figura 4.1. □

Ejemplo 4.1.5. Durante la ejecución de DESPLEGAR($\texttt{avl}(t, h), 3$) ocurre que:

$$\textsc{Repartir}(2, 2) = \{(0, 2), (1, 1), (2, 0)\}$$

Dado que:

$$\begin{aligned}
\textsc{Ind}(\texttt{avl}(l, h)) = (t \mapsto l : t_1, r : t_2, ht : h * \texttt{avl}(t_1, h_1) * \texttt{avl}(t_2, h_2) \wedge \\
h = 1 + max(h_1, h_2) \wedge |h_1 - h_2| \leq 1)
\end{aligned}$$

el ciclo de la línea 9 del algoritmo (Alg. 5) realizará en sucesivas iteraciones las siguientes llamadas recursivas:

$$\textsc{Desplegar}(\texttt{avl}(t_1, h_1), 0) \text{ y } \textsc{Desplegar}(\texttt{avl}(t_2, h_2), 2)$$
$$\textsc{Desplegar}(\texttt{avl}(t_1, h_1), 1) \text{ y } \textsc{Desplegar}(\texttt{avl}(t_2, h_2), 1)$$
$$\textsc{Desplegar}(\texttt{avl}(t_1, h_1), 2) \text{ y } \textsc{Desplegar}(\texttt{avl}(t_2, h_2), 0)$$

Tomemos el primer caso como ejemplo. Claramente, $\textsc{Desplegar}(\texttt{avl}(t_1, h_1), 0) = \{\texttt{emp} \wedge t_1 = null \wedge h_1 = 0\}$. Por otro lado, el resultado de $\textsc{Desplegar}(\texttt{avl}(t_2, h_2), 2)$ es:

$$\begin{array}{l}
\textsc{Desplegar}(\texttt{avl}(t_2, h_2), 2) \\
= \\
\{ \\
(t_2 \mapsto l : t_3, r : t_4, ht : h_2 * t_3 \mapsto l : t_5, r : t_6, ht : h_3) \wedge \\
\quad (h_2 = 1 + max(h_3, h_4) \wedge |h_3 - h_4| \leq 1 \wedge \\
\quad h_3 = 1 + max(h_5, h_6) \wedge |h_5 - h_6| \leq 1 \wedge \\
\quad t_4 = t_5 = t_6 = null \wedge h_4 = h_5 = h_6 = 0), \\
(t_2 \mapsto l : t_3, r : t_4, ht : h_2 * t_4 \mapsto l : t_5, r : t_6, ht : h_4) \wedge \\
\quad (h_2 = 1 + max(h_3, h_4) \wedge |h_3 - h_4| \leq 1 \wedge \\
\quad h_4 = 1 + max(h_5, h_6) \wedge |h_5 - h_6| \leq 1 \wedge \\
\quad t_3 = t_5 = t_6 = null \wedge h_3 = h_5 = h_6 = 0), \\
\}
\end{array}$$

El primer heap simbólico representa un árbol binario con raíz t_2, hijo izquierdo formado por un único nodo (t_3), e hijo derecho vacío ($null$). En el segundo heap simbólico se mantiene la raíz (t_2), aunque ahora el árbol izquierdo es vacío y el derecho tiene un sólo nodo (t_4). Observar que, al combinar los resultados de la llamada actual con las recursivas $\textsc{Desplegar}(\texttt{avl}(t_1, h_1), 0)$ y $\textsc{Desplegar}(\texttt{avl}(t_2, h_2), 2)$ se obtienen fórmulas inconsistentes. Por ejemplo, combinando el único heap simbólico

en la primera llamada con el primer heap simbólico mostrado arriba nos queda:

$$
\begin{array}{l}
(t \mapsto l : t_1, r : t_2, ht : h * t_2 \mapsto l : t_3, r : t_4, ht : h_2 * t_3 \mapsto l : t_5, r : t_6, ht : h_3) \wedge \\
h = 1 + max(h_1, h_2) \wedge |h_1 - h_2| \leq 1 \wedge \\
\quad t_1 = null \wedge h_1 = 0 \wedge \\
\quad (h_2 = 1 + max(h_3, h_4) \wedge |h_3 - h_4| \leq 1 \wedge \\
\quad h_3 = 1 + max(h_5, h_6) \wedge |h_5 - h_6| \leq 1 \wedge \\
\quad t_4 = t_5 = t_6 = null \wedge h_4 = h_5 = h_6 = 0),
\end{array}
$$

es decir, un heap simbólico cuya parte pura no es satisfactible: h_2 debe ser igual a 2, y como $h_1 = 0$, $|h_1 - h_2| \not\leq 1$. □

El ejemplo anterior muestra que no todos los heaps simbólicos retornados por Desplegar deben ser considerados en el cómputo de cotas ajustadas. De este problema nos encargaremos en la sección siguiente.

Para mantener la corrección de Desplegar las variables utilizadas en las llamadas recursivas de las líneas 10-12 no deben solaparse. Una forma de lograr esto, es implementar Ind de modo que reemplace las variables en la definición del predicado inductivo por variables frescas. En nuestra implementación, se utiliza un índice global por cada variable, que Ind incrementa cada vez que es invocado, evitando así repeticiones de variables.

Es importante mencionar que extender el algoritmo Desplegar de la figura 5 para soportar predicados inductivos más generales es sencillo. En primer lugar, la definición de un predicado podría tener más de un objeto (como se discutió arriba, por el momento sólo se acepta una raíz, apuntada por un parámetro del predicado). Desplegar sigue siendo correcto bajo esta nueva hipótesis, pero se pierde la correlación entre la cantidad de desplegados realizados y la cantidad de objetos en la estructura, que es primordial para simplificar la presentación de esta sección y la siguiente. En segundo lugar, se podrían permitir predicados inductivos con más de un caso base/inductivo. Bajo esta asumción, Desplegar simplemente debería retornar el conjunto formado por los heaps simbólicos introducidos por cada uno de los casos bases. Además, debería llevar a cabo las operaciones de las líneas 9-14 para cada uno de los casos inductivos del predicado, por ejemplo, introduciendo un ciclo antes de la línea 9.

4.1.2. Generando instancias concretas a partir de heaps simbólicos

La tarea que nos ocupa en esta sección es la de formalizar la conversión de heaps simbólicos a las instancias concretas del heap que estos representan; una intuición de la misma se presentó en la figura 4.1.

Antes de comenzar, es necesario recordar la definición formal del heap en el análisis de código TACO (sección 2.3.1). Como se analiza (una versión restringida de) el lenguaje JAVA, los únicos tipos disponibles son los definidos por el usuario (mediante clases) y los enteros (Int). Para simplificar la presentación, se asume un scope fijo k, y un único tipo definido por el usuario que representa los nodos de las estructuras de datos:

$$\text{Nodo} = \{N_0, \ldots, N_k\} \tag{4.2}$$

Además, se asume un conjunto fijo de campos:

$$\begin{aligned} &\mathtt{f}_1 : \mathrm{T}_1 \to \mathrm{T}'_1 \\ &\ldots \\ &\mathtt{f}_n : \mathrm{T}_n \to \mathrm{T}'_n \end{aligned}$$

donde cada T_i, T'_i, $1 \leq i \leq n$ es o bien Int, o bien Node.

Ejemplo 4.1.6. A continuación se muestran las definiciones de los campos del heap para las estructuras `slist` y `avl` del ejemplo 4.1.1. Haciendo abuso de notación se utilizará en ambos casos el mismo conjunto Nodo (4.2), aunque en la práctica las clases que definen los nodos de ambas estructuras son diferentes. El único campo en el heap para listas, `n`, es el que relaciona cada nodo con su sucesor en la lista:

$$\mathtt{n} : \text{Nodo} \to (\text{Nodo} \cup \{\mathit{null}\})$$

Para representar AVLs son necesarios los campos que relacionan nodos con sus respectivos árboles izquierdo y derecho (`l` y r, resp.), y su altura (`ht`):

$$\begin{aligned} &\mathtt{l} : \text{Nodo} \to (\text{Nodo} \cup \{\mathit{null}\}) \\ &\mathtt{r} : \text{Nodo} \to (\text{Nodo} \cup \{\mathit{null}\}) \\ &\mathtt{ht} : \text{Nodo} \to (\text{Int}) \end{aligned}$$

□

4.1.3. Codificación de heaps simbólicos en Yices

En este trabajo se propone usar un SMT solver para obtener modelos concretos –con la estructura del heap definida anteriormente– de los heaps simbólicos. En particular, se utilizará aquí el SMT solver *Yices*, presentado en la sección 2.10. La idea es extraer de un heap simbólico un conjunto de restricciones sobre sus variables, en el lenguaje de entrada de Yices, e invocar al solver para obtener asignaciones de valores a variables que satisfagan las restricciones (si existen). Cada una de estas asignaciones se utiliza luego para construir una instancia concreta reemplazando las variables por sus valores correspondientes en la parte espacial del heap simbólico.

La primera parte de la codificación consiste en especificar el tipo Nodo en Yices:

(define-type nodo (scalar n0 n1 . . . nk null))

En la definición anterior el tipo `nodo` está conformado por los valores (diferentes entre sí) n0, . . ., nk, null; representado los elementos respectivos $N_0, \ldots, N_k$ del conjunto Nodo (4.2).

La segunda parte de la codificación requiere la declaración de cada una de las variables componentes del heap simbólico. La notación utilizada en este trabajo para los heaps simbólicos no incluye declaraciones de tipos para las variables, sin embargo, en Yices es necesario asignar un tipo a cada variable declarada. Por lo tanto, se asume que el usuario de nuestro enfoque declara los tipos de todas las variables en la definición de un predicado inductivo. Así, sea x una variable, se asume que $\texttt{type}(x)$ devuelve su tipo. De todas maneras, durante el resto de este trabajo se mantendrá la notación simplificada (sin tipos) por razones de claridad en la presentación. La función VDECL de la figura 4.2 devuelve un conjunto con todas las declaraciones de variables (en notación Yices) que aparecen en un heap simbólico. (Recordar que la sintaxis de los heaps simbólicos se presentó en la figura 2.10, sección 2.9.3). VDECL se define recursivamente de manera estándar, utilizando *pattern matching*. Para facilitar el seguimiento de la función, en algunos casos se agrega a la izquierda del parámetro, seguido por el símbolo :, la fórmula que da origen al parámetro en la gramática de la figura 2.10. Por ejemplo, $\Sigma : emp$ indica que el parámetro, *emp*, pertenece a la parte espacial Σ del heap simbólico.

La tercera parte consiste en convertir la parte pura del heap simbólico a una fórmula Yices. La función PURE de la figura 4.3 lleva a cabo esta tarea. Al igual que VDECL, PURE se define recursivamente vía *pattern matching*. La traducción realizada por PURE es directa: mayormente involucra convertir fórmulas en notación infija a prefija, y cambiar los nombres de los operadores en el lenguaje de los heaps simbólicos por sus contrapartes (obvias) en Yices. Las traducciones más complejas corresponden a los operadores máximo y mínimo, que se basan en la construcción

$$
\begin{array}{lcl}
\textsc{vdecl}(x) & = & \{(\text{define x::}\texttt{type}(\text{x}))\} \\
\textsc{vdecl}(null) & = & \emptyset \\
\textsc{vdecl}(\Sigma \wedge \Pi) & = & \textsc{vdecl}(\Sigma) \cup \textsc{vdecl}(\Pi) \\
\textsc{vdecl}(\Sigma : emp) & = & \emptyset \\
\textsc{vdecl}(\Sigma : E \mapsto f_1 : E_1, \ldots, f_n : E_n) & = & \textsc{vdecl}(E) \cup \textsc{vdecl}(E_1) \cup \ldots \cup \textsc{vdecl}(E_n) \\
\textsc{vdecl}(\Sigma_1 * \Sigma_2) & = & \textsc{vdecl}(\Sigma_1) \cup \textsc{vdecl}(\Sigma_2) \\
\textsc{vdecl}(\Pi : \gamma \wedge \phi) & = & \textsc{vdecl}(\gamma) \cup \textsc{vdecl}(\phi) \\
\textsc{vdecl}(\gamma : E_1 \otimes E_2) & = & \textsc{vdecl}(E_1) \cup \textsc{vdecl}(E_2),\ \otimes \in \{=, \neq\} \\
\textsc{vdecl}(\gamma_1 \wedge \gamma_2) & = & \textsc{vdecl}(\gamma_1) \cup \textsc{vdecl}(\gamma_2) \\
\textsc{vdecl}(\neg\phi_1) & = & \textsc{vdecl}(\phi_1) \\
\textsc{vdecl}(\phi_1 \otimes \phi_2) & = & \textsc{vdecl}(\phi_1) \cup \textsc{vdecl}(\phi_2),\ \otimes \in \{\wedge, \vee\} \\
\textsc{vdecl}(\phi : true) & = & \emptyset \\
\textsc{vdecl}(\phi : false) & = & \emptyset \\
\textsc{vdecl}(\phi : b_1 = b_2) & = & \textsc{vdecl}(b_1) \cup \textsc{vdecl}(b_2) \\
\textsc{vdecl}(\phi : s_1 \otimes s_2) & = & \textsc{vdecl}(s_1) \cup \textsc{vdecl}(s_2),\ \otimes \in \{=, <, >, \leq, \geq\} \\
\textsc{vdecl}(\phi : k) & = & \emptyset \\
\textsc{vdecl}(\phi : k \times s_1) & = & \textsc{vdecl}(s_1) \\
\textsc{vdecl}(\phi : -s_1) & = & \textsc{vdecl}(s_1) \\
\textsc{vdecl}(\phi : s_1 \otimes s_2) & = & \textsc{vdecl}(s_1) \cup \textsc{vdecl}(s_2),\ \otimes \in \{+, max, min\}
\end{array}
$$

Figura 4.2: Función VDECL: retorna un conjunto de declaraciones Yices para las variables del heap simbólico

$$
\begin{array}{lcl}
\textsc{pure}(\Sigma \wedge \Pi) & = & \textsc{pure}(\Pi) \\
\textsc{pure}(x) & = & \text{x} \\
\textsc{pure}(null) & = & \text{null} \\
\textsc{pure}(\Pi : \gamma \wedge \phi) & = & (\text{and } \textsc{pure}(\gamma)\ \textsc{pure}(\phi)) \\
\textsc{pure}(\gamma : E_1 = E_2) & = & (= \textsc{pure}(E_1)\ \textsc{pure}(E_2)) \\
\textsc{pure}(\gamma : E_1 \neq E_2) & = & (/= \textsc{pure}(E_1)\ \textsc{pure}(E_2)) \\
\textsc{pure}(\gamma_1 \wedge \gamma_2) & = & (\text{and } \textsc{pure}(\gamma_1)\ \textsc{pure}(\gamma_2)) \\
\textsc{pure}(\neg\phi_1) & = & (\text{not } \textsc{pure}(\phi_1)) \\
\textsc{pure}(\phi_1 \wedge \phi_2) & = & (\text{and } \textsc{pure}(\phi_1)\ \textsc{pure}(\phi_2)) \\
\textsc{pure}(\phi_1 \vee \phi_2) & = & (\text{or } \textsc{pure}(\phi_1)\ \textsc{pure}(\phi_2)) \\
\textsc{pure}(\phi : true) & = & \text{true} \\
\textsc{pure}(\phi : false) & = & \text{false} \\
\textsc{pure}(\phi : b_1 = b_2) & = & (= \textsc{pure}(b_1)\ \textsc{pure}(b_2)) \\
\textsc{pure}(\phi : s_1 \otimes s_2) & = & (\otimes\ \textsc{pure}(s_1)\ \textsc{pure}(s_2)),\ \otimes \in \{=, <, >\} \\
\textsc{pure}(\phi : s_1 \leq s_2) & = & (<= \textsc{pure}(s_1)\ \textsc{pure}(s_2)) \\
\textsc{pure}(\phi : s_1 \geq s_2) & = & (>= \textsc{pure}(s_1)\ \textsc{pure}(s_2)) \\
\textsc{pure}(\phi : k) & = & \text{k} \\
\textsc{pure}(\phi : k \times s_1) & = & (* \text{ k } \textsc{pure}(s_1)) \\
\textsc{pure}(\phi : -s_1) & = & (- \textsc{pure}(s_1)) \\
\textsc{pure}(\phi : s_1 + s_2) & = & (+ \textsc{pure}(s_1)\ \textsc{pure}(s_2)) \\
\textsc{pure}(\phi : max(s_1, s_2)) & = & (\text{ite } (\textsc{pure}(s_1) > \textsc{pure}(s_2))\ \textsc{pure}(s_1)\ \textsc{pure}(s_2)) \\
\textsc{pure}(\phi : min(s_1, s_2)) & = & (\text{ite } (\textsc{pure}(s_1) < \textsc{pure}(s_2))\ \textsc{pure}(s_1)\ \textsc{pure}(s_2))
\end{array}
$$

Figura 4.3: Función PURE: traduce la parte pura de un heap simbólico a Yices

$$
\begin{array}{lcl}
\textsc{disj}(\Sigma \wedge \Pi) & = & \textsc{disj}(\Sigma) \\
\textsc{disj}(emp) & = & \emptyset \\
\textsc{disj}(E \mapsto f_1 : E_1, \ldots, f_n : E_n) & = & \{E\} \\
\textsc{disj}(\Sigma_1 * \Sigma_2) & = & \textsc{disj}(\Sigma_1) \cup \textsc{disj}(\Sigma_2)
\end{array}
$$

Figura 4.4: Función DISJ: retorna el conjunto de variables de un heap simbólico que no pueden ser alias

if-then-else (`ite`) de Yices: `(ite cond r1 r2)` retorna la expresión r1 si vale el predicado cond, en caso contrario devuelve r2 (sección 2.10).

Por último, es necesario codificar en Yices las restricciones impuestas a las variables por la parte espacial del heap simbólico. Recordemos que, si la parte espacial del heap simbólico es $E_1 \mapsto f_1 : E_{1,1}, \ldots, f_n : E_{1,n} * \ldots * E_m \mapsto f_1 : E_{m,1}, \ldots, f_n : E_{m,n}$, las variables $E_1, \ldots, E_m$ no pueden referenciar al mismo nodo debido a la semántica de $*$ (no pueden ser alias). De esta manera, la función DISJ de la figura 4.4 retorna el conjunto de variables obligadas a tomar valores diferentes debido a la semántica de la parte espacial del heap simbólico. Entonces, sean $d_1, \ldots, d_l$ las variables retornadas por DISJ, por cada par $\mathrm{d}_i, \mathrm{d}_j$ tal que $i \neq j$ $(1 \leq i, j \leq l)$ se debe agregar a la especificación Yices una fórmula que refleje lo anterior:

$$(\neq \mathrm{d}_i\ \mathrm{d}_j)$$

Notar que, si bien las fórmulas anteriores garantizan la disyunción entre los dominios de cada uno de las celdas $(E \mapsto f_1 : E_1, \ldots, f_n : E_n)$ del heap simbólico, podrían existir más nodos –dentro del scope k– que celdas individuales en el heap simbólico, comprometiendo la corrección del algoritmo. Sin embargo, como se discutió anteriormente, en esta sección se concretizarán heaps simbólicos resultantes de realizar tantos desplegados sobre el predicado inductivo de entrada como indique el scope (k). Debido a las restricciones que se impusieron sobre la forma de los predicados inductivos (ver 4.1, sección 4.1), hacer k desplegados de un predicado resulta en heaps simbólicos con exactamente k nodos. Lo anterior implica que la concretización discutida aquí es correcta respecto de la semántica de los heaps simbólicos (sección 2.9.4).

El algoritmo CODIFICAR de la figura 6 resume la codificación presentada anteriormente. CODIFICAR toma como entradas un heap simbólico sh, el límite k en la cantidad de nodos y el nombre del archivo, $outfile$, donde debe escribir la especificación Yices que codifica a sh. CODIFICAR lleva una secuencia de líneas de texto,

Algorithm 6 Algoritmo CODIFICAR

1: **function** CODIFICAR($sh, k, outfile$)
2: ▷ *lines* recolecta la secuencia de líneas de texto de la specificación Yices a retornar
3: $lines = [(\text{define-type nodo (scalar n0 n1} \ldots \text{nk null}))]$
4: $lines = lines + \text{VDECL}(sh)$
5: $lines = lines + [(\text{assert PURE}(sh))]$
6: $lines = lines + [(\text{assert } (\neq \text{d}_i \text{ d}_j))|d_i, d_j \in \text{DISJ}(sh), i \neq j]$
7: ▷ Escribir *lines* al archivo con nombre $outfile$
8: WRITETOFILE($outfile, lines$)

lines, que al final de la ejecución se escriben en un archivo (a través de la función WRITETOFILE) que contiene la especificación Yices que codifica el heap simbólico de entrada (línea 8). La primera línea del archivo generado por CODIFICAR contiene la definición del tipo nodo (línea 3). Luego, se agregan a *lines* las definiciones de variables generadas por VDECL (línea 4). Para agregar líneas de texto al final de *lines*, CODIFICAR hace uso de la función de concatenación de secuencias +. Posteriormente, *lines* se extiende con una aserción que asegura que la restricción de la parte pura del heap simbólico (PURE(sh)) debe cumplirse en la espeicificación Yices (línea 5). Por último, se agregan a *lines* aserciones para asegurar que las variables retornadas por DISJ(sh) no puedan tomar el mismo valor (línea 6).

Ejemplo 4.1.7. Recordemos del ejemplo 4.1.4 que:

$$\begin{array}{l} \text{DESPLEGAR}(\texttt{slist}(l), 2) \\ = \\ \{(l \mapsto n : l_1 * l_1 \mapsto n : l_2) \wedge l_2 = null\} \end{array}$$

Llamemos sh al único heap simbólico en el resultado de DESPLEGAR($\texttt{slist}(l), 2$). Ejecutar CODIFICAR($sh, 2, "output.ys"$) produce la siguiente especificación Yices

Algorithm 7 Algoritmo SatAssigns

1: **function** SatAssigns(*spec*)
2: $res = \emptyset$
3: **while** Yices($spec, a$) = SAT **do**
4: $res = res \cup \{a\}$
5: AddAssert($spec$, Neg(a))
6: **return** res

(las líneas que comienzan con ;; son comentarios agregados por los autores):

```
;; Archivo output.ys
;; Definición de tipos; scope = 2
(define-type nodo (scalar n0 n1 null))
;; Definición de variables
(define l::nodo)
(define l1::nodo)
(define l2::nodo)
;; Fórmula derivada de la parte pura
(assert (= l2 null))
;; Fórmula derivada de la parte espacial
(assert (/= l l1))
```

□

Es importante destacar que la codificación anterior sólo depende de características estándar de los SMT solvers para la aritmética lineal; se eligió Yices principalmente por razones prácticas: una implementación eficiente del mismo puede descargarse gratuitamente desde [4].

El algoritmo SatAssigns de la figura 7 invoca sucesivas veces a Yices para generar todas las posibles asignaciones que satisfagan su especificación de entrada, *spec*. Una invocación a Yices dentro del algoritmo se representa con la función Yices. Yices($spec, a$) retorna SAT si *spec* es satisfactible, y almacena la asignación encontrada en la variable a. En caso contrario, Yices retorna UNSAT y asigna null a a. Para evitar que Yices retorne siempre la misma asignación a, al final de cada iteración del ciclo de la línea 3 (línea 5) se agrega a *spec* una aserción (AddAssert) que asegura que a no puede volver a ocurrir, es decir, se agrega como aserción la negación de a (Neg(a)).

$$
\begin{array}{lcl}
\textsc{RepVars}(\Sigma \wedge \Pi, a) & = & \textsc{RepVars}(\Sigma, a) \\
\textsc{RepVars}(emp, a) & = & \emptyset \\
\textsc{RepVars}(E \mapsto f_1 : E_1, \ldots, f_n : E_n, a) & = & \{\{(a(E), a(E_1))\}, \ldots, \\
 & & \quad \{(a(E), a(E_n))\}\} \\
\textsc{RepVars}(\Sigma_1 * \Sigma_2, a) & = & \{r_1{,}1 \cup r_2{,}1, \ldots, r_1.n \cup r_2.n \mid \\
 & & \quad r_1 = \textsc{RepVars}(\Sigma_1, a), \\
 & & \quad r_2 = \textsc{RepVars}(\Sigma_2, a)\}
\end{array}
$$

Figura 4.5: Función $\textsc{RepVars}(sh, a)$: retorna una instancia concreta a partir del heap simbólico sh y la asignación a

Ejemplo 4.1.8. Invocando a $\textsc{Yices}(spec, a)$, con *spec* la especificación del ejemplo 4.1.7 produce la siguiente asignación:

$$a = [\mathrm{l} \mapsto \mathrm{n0}, \mathrm{l1} \mapsto \mathrm{n1}, \mathrm{l2} \mapsto \mathrm{null}]$$

Además, $\textsc{Neg}(a)$ devuelve la aserción Yices:

(assert (not (and (= l n0) (= l1 n1) (= l2 null))))

que, luego de ser añadida a *spec*, elimina la posibilidad de que a vuelva a ser seleccionada en la siguiente iteración del ciclo principal de SatAssigns. □

Una vez obtenidas las posibles asignaciones de valores a variables, el proceso de generación de estructuras concretas consiste en reemplazar en la parte espacial del heap simbólico las variables por sus valores en una asignación particular. Este procedimiento es implementado por el algoritmo recursivo RepVars, que se muestra en la figura 4.5. Sus entradas son un heap simbólico, y una asignación de valores a variables a. Sean $f_1, \ldots, f_n$ los campos del heap, RepVars genera un conjunto S_i $(1 \leq i \leq n)$ para cada uno de ellos, que contiene los valores del campo S_i. En el algoritmo, $a(E)$ devuelve el valor de la expresión E en la asignación a. Además, se asume que $a(null) = null$. Intuitivamente, para cada objeto $E \mapsto f_1 : E_1, \ldots, f_n : E_n$, RepVars agrega al campo f_i $(1 \leq i \leq n)$ el valor $(a(E), a(E_i))$.

Ejemplo 4.1.9. Sea sh el heap simbólico resultante de ejecutar $\textsc{Unfold}(slist, 2)$ (ejemplo 4.1.7):

$$(l \mapsto n : l_1 * l_1 \mapsto n : l_2) \wedge l_2 = null$$

Algorithm 8 Algoritmo GenInstances

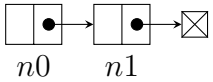
1: **function** GenInstances(p, k)
2: $res = \emptyset$
3: $shs =$ Desplegar(p, k)
4: **for** $sh \in shs$ **do**
5: $spec =$ Codificar(sh, k, "*output.ys*")
6: **for** $a \in$ SatAssigns($spec$) **do**
7: $res = res \cup \{$RepVars(sh, a)$\}$
8: **return** res

y a la asignación que se consigue codificando sh en Yices (ejemplo 4.1.8):

$$a = [\mathrm{l} \mapsto \mathrm{n0}, \mathrm{l1} \mapsto \mathrm{n1}, \mathrm{l2} \mapsto \mathrm{null}]$$

RepVars(sh, a) retorna el heap que se muestra a continuación, en formato relacional y gráfico:

$$next = \{(n0, n1), (n1, null)\}$$



□

El algoritmo GenInstances de la figura 8 formaliza el procedimiento completo de generación de instancias concretas con exactamente k nodos, a partir de un predicado inductivo p. En primer lugar, (línea 3 del algoritmo) se realizan k desplegados de p para generar un conjunto de heaps simbólicos shs. En segundo lugar, cada $sh \in shs$ se codifica en una especificación Yices, $spec$ (líneas 4 y 5), y se generan vía Yices todas las asignaciones a que satisfacen $spec$ (línea 6). Por último, cada una de estas a sirve para construir una instancia concreta del heap simbólico sh, que se agrega al resultado del algoritmo (línea 7). Notar que los heaps simbólicos inconsistentes no poseen asignaciones que los hagan verdaderos, y, por lo tanto, no producen ninguna estructura concreta.

4.1.4. Rotura de simetrías

Una desventaja importante del algoritmo GenInstances (Alg. 8) de la sección anterior es que admite la generación de instancias isomorfas del heap.

Ejemplo 4.1.10. Sea sh el heap simbólico retornado por Unfold($slist, 2$) (ejemplo 4.1.7):

$$(l \mapsto n : l_1 * l_1 \mapsto n : l_2) \wedge l_2 = null$$

$$
\begin{array}{lcl}
\textsc{SBctr}(\Sigma \wedge \Pi, [0,..,k-1]) & = & \textsc{SBctr}(\Sigma) \\
\textsc{SBctr}(emp, []) & = & \emptyset \\
\textsc{SBctr}(E \mapsto f_1 : E_1,..,f_n : E_n * \Sigma, [i,..,k-1]) & = & E = N_i \wedge \\
 & & \textsc{SBctr}(\Sigma, [i+1,..,k-1]) \\
\textsc{SBctr}(E \mapsto f_1 : E_1,..,f_n : E_n, [k-1]) & = & E = N_{k-1}
\end{array}
$$

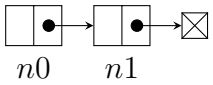
Figura 4.6: Función SBctr: devuelve una conjunción de fórmulas puras que, al añadirlas al heap simbólico de entrada, permiten evitar la generación de instancias simétricas

La codificación en Yices de sh (ejemplo 4.1.7) posee dos asignaciones que la satisfacen, a_1 y a_2:

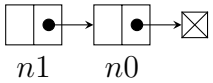
$$
\begin{aligned}
a_1 &= [\mathrm{l} \mapsto \mathrm{n0}, \mathrm{l1} \mapsto \mathrm{n1}, \mathrm{l2} \mapsto \mathrm{null}] \\
a_2 &= [\mathrm{l} \mapsto \mathrm{n1}, \mathrm{l1} \mapsto \mathrm{n0}, \mathrm{l2} \mapsto \mathrm{null}]
\end{aligned}
$$

Entonces, ejecuciones sucesivas de $\textsc{RepVars}(sh, a_1)$ y $\textsc{RepVars}(sh, a_2)$ generan las dos estructuras isomorfas mostradas a continuación:

$next = \{(n0, n1), (n1, null)\}$



$next = \{(n1, n0), (n0, null)\}$



□

En el caso general, para un heap simbólico con k celdas en su parte espacial, $E_1 \mapsto f_1 : E_{1,1},..,f_n : E_{1,n} * \ldots * E_k \mapsto f_1 : E_{k,1},..,f_n : E_{k,n}$, existen $k!$ formas de asignar los k nodos de entrada, $N_0, \ldots, N_k$, a las variables que hacen referencia a cada una de las celdas, $E_1, \ldots, E_k$. Es decir, el algoritmo GenInstances de la figura 8 produce por cada heap simbólico $k!$ instancias concretas isomorfas, lo que es inaceptable desde el punto de vista de la eficiencia.

A continuación se propone una forma sencilla de solucionar el problema anterior. Primero, es necesario inducir un orden sobre los nodos disponibles, por ejemplo, $N_0 < N_1 < \ldots < N_k$. Por lo tanto, podemos pensar al conjunto de nodos de entrada como una *secuencia*: $S = [N_0, \ldots, N_k]$. Intuitivamente, la idea para evitar la generación de estructuras simétricas es asignar nodos de S a las variables $E_1, \ldots, E_k$

en su orden de aparición en un recorrido de la parte espacial del heap simbólico de izquierda a derecha. La función SBctr de la figura 4.6 implementa esta idea. SBctr toma como parámetros un heap simbólico y una secuencia de índices de nodos, y retorna una fórmula pura que fuerza una única posible asignación de nodos a variables según el orden de aparición de las variables (de izquierda a derecha) en el heap simbólico. SBctr tiene como precondición implícita que la cantidad de celdas $E \mapsto f_1 : E_1, .., f_n : E_n$ en la parte espacial del heap simbólico y la cantidad de índices en al secuencia son iguales (recordemos que este es requisito necesario para que el enfoque por fuerza bruta sea correcto).

Ejemplo 4.1.11. Nuevamente, si sh es el heap simbólico retornado por Unfold$(slist, 2)$ (ejemplo 4.1.7):

$$(l \mapsto n : l_1 * l_1 \mapsto n : l_2) \wedge l_2 = null$$

La ejecución de SBctr$(sh, [0, 1])$ retorna la fórmula:

$$l = N_0 \wedge l_1 = N_1$$

que añadida a la parte pura de sh resulta en el heap simbólico sh':

$$(l \mapsto n : l_1 * l_1 \mapsto n : l_2) \wedge (l_2 = null \wedge l = N_0 \wedge l_1 = N_1)$$

Así, existe una sola asignación a que satisface a sh':

$$a = [\mathrm{l} \mapsto \mathrm{n0}, \mathrm{l1} \mapsto \mathrm{n1}, \mathrm{l2} \mapsto \mathrm{null}]$$

eliminando así la posibilidad de generar la instancia simétrica excedente del ejemplo 4.1.10. □

Una versión modificada de GenInstances que incorpora SBctr para romper simetrías se muestra en la figura 9. La diferencia principal entre el algoritmo resultante GenNoSymInstances y el original reside en que el primero agrega restricciones de rotura de simetría a cada uno de los heaps simbólicos producidos por Unfold.

Ejemplo 4.1.12. Veamos una ejecución completa de GenNoSymInstances. GenNoSymInstances$(slist, 2)$ comienza ejecutando Unfold$(slist, 2)$ (línea 3, figura 9), que produce el heap simbólico sh (ver ejemplo 4.1.7):

$$(l \mapsto n : l_1 * l_1 \mapsto n : l_2) \wedge l_2 = null$$

Algorithm 9 Algoritmo GenNoSymInstances

1: **function** GenInstances(p, k)
2: $res = \emptyset$
3: $shs =$ Desplegar(p, k)
4: **for** $sh \in shs$ **do**
5: **let** $sh = \Sigma \wedge \Pi$ **in**
6: $sh' = \Sigma \wedge (\Pi \wedge$ SBctr$(sh, [0..k]))$
7: $spec =$ Codificar$(sh', k, "output.ys")$
8: **for** $a \in$ SatAssigns$(spec)$ **do**
9: $res = res \cup \{$RepVars$(sh', a)\}$
10: **return** res

Al componer la parte pura de sh con el resultado de SBctr$(sh, [0..k])$ (línea 6) se obtiene sh' (ver ejemplo 4.1.11):

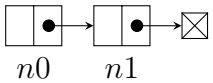
$$(l \mapsto n : l_1 * l_1 \mapsto n : l_2) \wedge (l_2 = null \wedge l = N_0 \wedge l_1 = N_1)$$

Luego, SatAssigns (línea 8) retorna una sóla asignación válida para sh' (ejemplo 4.1.11):

$$a = [\mathrm{l} \mapsto \mathrm{n0}, \mathrm{l1} \mapsto \mathrm{n1}, \mathrm{l2} \mapsto \mathrm{null}]$$

Por último, RepVars(sh', a) (línea 9) produce la única instancia del heap que se muestra abajo:

$$next = \{(n0, n1), (n1, null)\}$$



□

4.1.5. Cotas ajustadas a partir del conjunto completo de instancias del heap

Para completar el enfoque de cómputo de cotas ajustadas por fuerza bruta sólo resta producir una cota ajustada utilizando las instancias retornadas por GenNoSymInstances. Para esto, se define un algoritmo similar a bottom-up (sección 1), pero que toma como entradas el conjunto de todas las instancias (acotadas) del heap, en lugar de obtenerlas a través de un procedimiento de decisión. El algoritmo, denominado GenFieldBounds, se muestra en la figura 10. Su entrada, *all_instances* es el conjunto de instancias producido por GenNoSymInstances. Recordar que, para una instancia particular, *instance*, la función $extend(S_{f_1}, \ldots, S_{f_n}, instance)$ (sección 1) agrega los valores del campo f_i en *instance* al conjunto S_{f_i}, para $1 \leq i \leq n$.

Algorithm 10 Algoritmo GenFieldBounds

1: **function** GenFieldBounds($all_instances$)
2: $S_{f_1}, ..., S_{f_n} = \emptyset, \ldots, \emptyset$
3: **for** $instance \in all_instances$ **do**
4: $extend(S_{f_1}, ..., S_{f_n}, instance)$
5: **return** $S_{f_1}, ..., S_{f_n}$

4.2. Cómputo de cotas ajustadas al vuelo

El algoritmo GenNoSymInstances de la sección anterior (Alg. 9) produce en su peor caso una cantidad exponencial de heaps simbólicos. En esta sección se aborda este problema, y se propone como solución un algoritmo que computa cotas ajustadas "al vuelo", mientras recorre el predicado inductivo de entrada. Este algoritmo evita el problema de tener que almacenar una gran cantidad de heaps simbólicos en memoria, que crecen exponencialmente con la cantidad de desplegados realizados al predicado de entrada.

Dado un predicado inductivo de entrada p (en forma restringida, ver 4.1), nuestro algoritmo, denominado SLBD, computa recursivamente cotas ajustadas $B_1, \ldots, B_i$ para cada uno de los predicados inductivos $p_1, \ldots, p_i$ que aparecen en la definición de p, monitoreando y retornando en cada llamada recursiva los valores asignados a los parámetros V_j de cada p_j ($1 \leq j \leq i$). Luego, los pares que la raíz de p contribuye al resultado se empaquetan en una cota ajustada B_r; esta se computa en base a $V_1, \ldots, V_i$. Finalmente, se realiza la unión de todas las cotas obtenidas en los pasos anteriores ($B_r, B_1, \ldots, B_i$) para producir el resultado final. Veamos un ejemplo intuitivo del funcionamiento del algoritmo SLBD.

Ejemplo 4.2.1. Supongamos que se desean computar cotas ajustadas para un heap caracterizado por el predicado inductivo `slist`, y se dispone de una secuencia de nodos de entrada $S = [N_0, N_1, N_2]$. Recordemos que desplegar el caso inductivo de `slist` resulta en el heap simbólico (ej. 4.1.4):

$$\textsc{Ind}(\texttt{slist}(l)) = l \mapsto n : l_1 * \texttt{slist}(l_1)$$

Notemos que –de manera similar al procedimiento de rotura de simetrías de la sección 4.1.4– es posible recorrer este heap simbólico de izquierda a derecha, asignando el nodo N_0 a la variable l, y utilizar los restantes nodos, N_1 y N_2, en el subheap representado por $\texttt{slist}(l_1)$. Por lo discutido anteriormente, la llamada recursiva del algoritmo con $\texttt{slist}(l_1)$ y la secuencia de nodos $[N_1, N_2]$ retorna una cota ajustada para el heap descripto por $\texttt{slist}(l_1)$ (con exactamente dos nodos), y los posibles

Algorithm 11 SLBD

1: **function** SLBD(p, f, l)
2: $rb := \emptyset, \ldots, \emptyset$
3: $sa := \emptyset$
4: **if** $l - f = 0$ **then**
5: **let** $\textsc{Base}(p(v^*)) = sh = emp \wedge \Pi_b$ **in**
6: $spec := \textsc{Codificar}(sh, f, l, \text{"}output.ys\text{"})$
7: **for** $a \in \textsc{SatAssigns}(spec)$ **do**
8: $sa := sa \cup \{a[v^*]\}$
9: **return** (rb, sa)
10: $\triangleright\ l - f > 0$
11: **let** $\textsc{Ind}(p(v^*)) = v \mapsto f_1 : v_1, \ldots, f_n : v_n * p_1(y_1^*) * \ldots * p_i(y_i^*) \wedge \Pi_{ind}$ **in**
12: $sba := \{v = N_f\}$
13: $\triangleright$ Considerar todas las formas posibles de repartir $l - (f + 1)$ nodos entre las llamadas recursivas
14: **for** $(f_1, l_1), \ldots, (f_i, l_i) \in \textsc{RepartirI}(f + 1, l, i)$ **do**
15: $(b_1, sa_1) =$SLBD$(p_1(y_1^*), f_1, l_1)$
16: $\ldots$
17: $(b_i, sa_i) =$SLBD$(p_i(y_i^*), f_i, l_i)$
18: $sat := false$
19: **for** $a_1, \ldots, a_i \in sa_1 \times \ldots \times sa_i$ **do**
20: $sh := emp \wedge (\Pi_{ind} \wedge \textsc{Conjunciones}(sba \cup a_1 \cup \ldots \cup a_n))$
21: $spec := \textsc{Codificar}(sh, f, l, \text{"}output.ys\text{"})$
22: **for** $a \in \textsc{SatAssigns}(spec)$ **do**
23: $rootb := \textsc{RepVars}(v \mapsto f_1 : v_1, \ldots, f_n : v_n, a)$
24: $rb := rb \cup rootb$
25: $sa := sa \cup \{a[v^*]\}$
26: $sat := true$
27: **if** sat **then**
28: $rb := rb \cup b_1 \cup \ldots \cup b_i$
29: **return** (rb, sa)

valores para l_1 durante la computación. En este caso particular, el resultado es:

$$(tb_i, sa_i) = (\{(N_1, N_2), (N_2, null)\}, \{l_1 = N_1\})$$

donde tb_i es la cota ajustada computada (para el campo siguiente de listas) y sa_i es la una única asignación posible para l_1, en donde l_1 toma el valor N_1. Por otra parte, para la raíz del predicado, $l \mapsto n : l_1$, tenemos:

$$(tbp_r, sa_r) = (\{(l, l_1)\}, \{l = N_0\})$$

es decir, la raíz debe agregar a la cota ajustada un par por cada forma posible de asignar valores a (l, l_1), y la variable l debe tomar el valor N_0 (para romper simetrías, como en la sección 4.1.4). Sin embargo, los valores de l_1 se calculan en la llamada recursiva, y, por lo tanto, no son parte de sa_r. Entonces, el algoritmo combina sa_i y sa_r para obtener:

$$sa = \{l = N_0, l_1 = N_1\}$$

Posteriormente, al reemplazar las variables de tbp_r por sus valores correspondientes en sa se obtiene la cota para la raíz:

$$tb_r = \{(N_0, N_1)\}$$

que se une con tb_i para generar la cota ajustada final:

$$tb = \{(N_0, N_1), (N_1, N_2), (N_2, null)\}$$

□

El algoritmo SLBD se muestra en la figura 11. Toma como entradas un predicado inductivo p, e índices al primer elemento, f, y al último, l, de la secuencia de nodos disponibles (hay $l-f$ nodos). Como se discutió anteriormente, SLBD retorna un par que consta de una cota ajustada rb para el heap, y un conjunto de asignaciones sa para los parámetros de p. Para ilustrar el funcionamiento del algoritmo se utilizará la ejecución del ejemplo 4.2.1: SLBD($\texttt{slist}, 0, 3$).

Si no hay nodos disponibles, $l - f = 0$ en la línea 4, SLBD utiliza el heap simbólico correspondiente al caso base de $p(v^*)$ para construir todas las asignaciones para las variables v^*. Por ejemplo, SLBD($\texttt{slist}(l_3), 3, 3$) utiliza las funciones Codificar y SatAssigns (líneas 6 y 7) para generar la única asignación válida para el caso base de `slist` ($l_3 = null$), $\{l_3 = null\}$. Notar que Codificar se modificó levemente, para tomar como parámetros los índices del primer y del último nodo, f y l, respectivamente. La función $a[v^*]$ de la línea 8 denota la restricción de dominio de la asignación a a las variables v^*. Esta función permite eliminar de las asignaciones a retornar las variables que no son parámetros de p (en el ejemplo no surte ningún efecto). Para concluir con el tratamiento del caso base de SLBD, como el conjunto de nodos de entrada es vacío, la cota ajustada producida por el algoritmo en este caso, rb, es siempre vacía.

Por otra parte, en la línea 12, el tratamiento del caso inductivo de p comienza asignando el primero de los nodos disponibles, N_f, a la variable que hace referencia a la raíz de p, v, con el objetivo de romper simetrías. Esta asignación se guarda en la variable sba. En nuestro ejemplo, esto equivale a asignar N_0 a la variable l, esto

es, $sba = \{(l, N_0)\}$. Luego, la línea 14 considera todas las formas posibles de repartir los restantes $l - (f + 1)$ nodos entre las llamadas recursivas. Así, RepartirI genera todas las posibles particiones en i subintervalos de $[f + 1, \ldots, l)$:

$$\textsc{RepartirI}(f, l, i) = \{(f_1, l_1), \ldots, (f_i, l_i) | f_1 = f \wedge l_i = l \bigwedge_{j=2..i} l_{j-1} = f_j\}$$

Más adelante, en las líneas 15-17, se invoca a SLBD recursivamente para computar cotas (b_j) y asignaciones (sa_j) para cada uno de los predicados inductivos (p_j, $1 \leq j \leq i$), usando como parámetros las particiones retornadas por RepartirI. Notar que las llamadas recursivas se realizan de manera similar a las del algoritmo Desplegar (Alg. 5, sección 4.1.1), intercambiando cantidad de desplegados por intervalos de nodos. En nuestro ejemplo, RepartirI genera un único intervalo, $(1, 3)$, que se usa para la llamada recursiva SLBD($\texttt{slist}(l_1), 1, 3$). Esta invocación retorna:

$$(tb_1, sa_1) = (\{(N_1, N_2), (N_2, null)\}, \{l_1 = N_1\})$$

Notar que, tb_1 y sa_1 son los valores denominados tb_i y sa_i, respectivamente, en el ejemplo 4.2.1.

Posteriormente, en la línea 19 se visita cada tupla posible de asignaciónes, $a_1, \ldots, a_i$, tomadas de los conjuntos de asignaciones computados por las llamadas recursivas, $sa_1 \times \ldots \times sa_i$. Notar que cada a_j, $0 \leq j \leq i$, asigna valores a los parámetros, y_j^*, del predicado p_j. Como se discutió en el ejemplo 4.2.1, cada una de estas asignaciones es útil para computar una cota para la raíz de la estructura. Para lograr esto, SLBD construye el heap simbólico sh, en la línea 20, cuya parte pura está compuesta de Π_{ind}, y una conjunción por cada elemento en las asignaciones sba, $a_1, \ldots, a_i$. Para obtener estas conjunciones, se usa la función Conjunciones, definida a continuación, que convierte una asignación a en una fórmula pura:

$$\textsc{Conjunciones}(a) = \bigwedge \{x = E | x \mapsto E \in a\}$$

En nuestro ejemplo de listas, la parte pura de sh es $l = N_0 \wedge l_1 = N_1$, donde el primer conyungendo viene de sba, y el segundo de a_1; Π_{ind} se omite debido a que la fórmula pura del caso inductivo de `slist` es el valor de verdad *true*. Al asignar el heap *emp* a la parte espacial de sh (línea 20), es posible utilizar Codificar y SatAssigns para buscar asignaciones que satisfacen la parte pura de sh (líneas 21 y 22)–notar que *emp* no agrega restricciones a *spec*. En el ejemplo, SatAssigns (línea 22 del algoritmo) encuentra una única asignación a, que es idéntica a la denominada sa en el ejemplo 4.2.1:

$$a = \{l = N_0, l_1 = N_1\}$$

En la línea 23 se construye una cota para la raíz, *rootb*, reemplazando las variables de $v \mapsto f_1 : v_1, \ldots, f_n : v_n$ por sus valores en a (vía RepVars). En el ejemplo:

$$rootb = \{(N_0, N_1)\}$$

que se corresponde con tb_r del ejemplo 4.2.1. Más adelante, en la línea 24, *rootb* se agrega a la cota resultado rb por medio de la operación de unión de cotas ajustadas, $\cup$. Recordemos que $\cup$ está sobrecargado, y se define como:

$$tb_1, \ldots, tb_n \cup tb'_1, \ldots, tb'_n = tb_1 \cup tb'_1 \ldots tb_n \cup tb'_n$$

En este punto de nuestro ejemplo rb es igual a *rootb*. Como a es una asignación válida de valores a v^*, se agrega $a[v^*]$ al conjunto de asignaciones sa en la línea 25. En el ejemplo, $a[l] = \{l = N_0\}$ es la única asignación válida añadida a sa. Por último, si existe alguna tupla de asignaciones $sa_1 \times \ldots \times sa_i$ satisfactibles en conjunto con sba y Π_{ind} (alguno de los sh de la línea 21 es SAT) –sat recibe el valor $true$ en la línea 26–, se debe extender la cota ajustada resultado, rb, con las cotas obtenidas recursivamente, $b_1, \ldots, b_i$. Esto se lleva a cabo en la línea 28. Observar que este paso podría realizarse dentro del ciclo de la línea 22, pero esto implicaría unir múltiples veces las cotas $b_1, \ldots, b_i$ a rb. En nuestro ejemplo, unir las cotas rb con b_1 –mostradas arriba– resulta en:

$$rb = \{(N_0, N_1), (N_1, N_2), (N_2, null)\}$$

Además, por lo discutido anteriormente, sa contiene sólo la asignación $a[v^*]$ (ver a arriba):

$$sa = \{\{l = N_0\}\}$$

Así:

$$\text{SLBD}(\texttt{slist}, 0, 3) = (\{(N_0, N_1), (N_1, N_2), (N_2, null)\}, \{\{l = N_0\}\})$$

como se discutió informalmente en el ejemplo 4.2.1.

Para concluir esta sección es importante destacar que SLBD computa cotas ajustadas para heaps con exactamente $l - 1$ (suponiendo $f = 0$) nodos, pero los análisis de código basados en SAT usualmente tratan con heaps con cualquier cantidad de nodos entre 0 y $l - 1$. Por lo tanto, para proveer cotas ajustadas para un análisis basado en SAT es necesario invocar sucesivamente a SLBD con 0, 1, 2, ... hasta $l - 1$ nodos, y construir la union de todas las cotas ajustadas resultantes de las ejecuciones anteriores.

Algorithm 12 SLBD

1: **function** SLBD(p, f, l, pa)
2: $rb := \emptyset, \ldots, \emptyset$
3: $sa := \emptyset$
4: **if** $l - f = 0$ **then**
5: **let** $\textsc{Base}(p(v^*)) = sh = emp \wedge \Pi_b \wedge \textsc{Conjunciones}(pa)$ **in**
6: $spec := \textsc{Codificar}(sh, f, l, \text{"}output.ys\text{"})$
7: **for** $a \in \textsc{SatAssigns}(spec)$ **do**
8: $sa := sa \cup \{a[v^*]\}$
9: **return** (rb, sa)
10: $\triangleright\ l - f > 0$
11: **let** $\textsc{Ind}(p(v^*)) = v \mapsto f_1 : v_1, \ldots, f_n : v_n * p_1(y_1^*) * \ldots * p_i(y_i^*) \wedge \Pi_{ind}$ **in**
12: $sba := \{v = N_f\}$
13: $\triangleright$ Considerar todas las formas posibles de repartir $l - (f + 1)$ nodos entre las llamadas recursivas
14: **for** $(f_1, l_1), \ldots, (f_i, l_i) \in \textsc{RepartirI}(f + 1, l, i)$ **do**
15: $sbpa = pa \cup sba$
16: $(b_1, sa_1) =$SLBD$(p_1(y_1^*), f_1, l_1, sbpa[y_1^*])$
17: $\ldots$
18: $(b_i, sa_i) =$SLBD$(p_i(y_i^*), f_i, l_i, sbpa[y_i^*])$
19: $sat := false$
20: **for** $a_1, \ldots, a_i \in sa_1 \times \ldots \times sa_i$ **do**
21: $sh := emp \wedge (\Pi_{ind} \wedge \textsc{Conjunciones}(sbpa \cup a_1 \cup \ldots \cup a_n))$
22: $spec := \textsc{Codificar}(sh, f, l, \text{"}output.ys\text{"})$
23: **for** $a \in \textsc{SatAssigns}(spec)$ **do**
24: $rootb := \textsc{RepVars}(v \mapsto f_1 : v_1, \ldots, f_n : v_n, a)$
25: $rb := rb \cup rootb$
26: $sa := sa \cup \{a[v^*]\}$
27: $sat := true$
28: **if** sat **then**
29: $rb := rb \cup b_1 \cup \ldots \cup b_i$
30: **return** (rb, sa)

4.2.1. Campos que apuntan "hacia atrás"

Generalmente, los campos del heap pueden dividirse en dos categorías: aquellos que permiten recorrer una estructura "hacia adelante" (campos *forward*), como *left* y *right* de árboles, *next* de listas, etc; y los que atraviesan una estructura "hacia atrás" (campos *backward*), como podría ser el campo *parent* de árboles, o el campo *previous* de listas doblemente encadenadas. En esta sección se define una versión de

SLBD que extiende a la anterior para dar un tratamiento especial a los campos que apuntan hacia atrás; el objetivo es mejorar la eficiencia del algoritmo en presencia de estos campos. Para simplificar la explicación de los problemas de SLBD con estos campos se procede a través de ejemplos.

Abajo se muestra la definición del predicado `dlistseg`, que caracteriza segmentos de listas doblemente encadenadas.

$$\begin{aligned}\texttt{dlistseg}(h,p,n,l) \doteq &(h = n \wedge p = l) \vee \\ &(h \mapsto pv : p, nx : h_1 * \texttt{dlistseg}(h_1,h,n,l))\end{aligned}$$

Los nodos de listas doblemente encadenadas tienen dos campos: pv (*previous*), que apunta al nodo anterior en la lista; y nx (*next*), que apunta al nodo siguiente. Intuitivamente, $\texttt{dlistseg}(h,p,n,l)$ denota un segmento de lista doblemente encadenada con primer y último nodo h y l, respectivamente, donde p apunta al nodo anterior a h ($p = h.pv$), y n al posterior a l ($n = l.nx$). Observar que, si $p = null$ y $n = null$, $\texttt{dlistseg}(h,p,n,l)$ caracteriza una lista doblemente encadenada completa (no sólo un segmento), que contiene los nodos comprendidos entre h y l. Por otro lado, si $p = l$ y $h = n$, $\texttt{dlistseg}(h,p,n,l)$ representa una lista circular.

Ejemplo 4.2.2. Consideremos el caso inductivo del predicado que define una lista doblemente encadenada apuntada por h: $\texttt{dlistseg}(h, null, null, l)$.

$$\textsc{Ind}(\texttt{dlistseg}(h, null, null, l)) = (h \mapsto pv : null, nx : h_1 * \texttt{dlistseg}(h_1, h, null, l))$$

Notar que una ejecución de SLBD atravesaría el heap simbólico anterior de izquierda a derecha, asignando un nodo, digamos N_i, a h. Luego, SLBD ejecutaría una llamada recursiva para obtener una asignación para los parámetros de $\texttt{dlistseg}(h_1, h, null, l))$. Sin embargo:

$$\textsc{Base}(\texttt{dlistseg}(h_1, h, null, l)) = h_1 = null \wedge h = l$$

y tanto h como l pueden tomar cualquier valor en su dominio, siempre que sean iguales. No obstante, h recibe el valor N_i antes de la llamada recursiva, lo que obligaría a que h y l sean ambas igual a N_i, pero el algoritmo ignora este hecho. Claramente, esto aumenta la cantidad de invocaciones realizadas al procedimiento de decisión durante la computación de asignaciones para el caso base (y, por lo tanto, la cantidad de asignaciones inválidas retornadas por la llamada recursiva). Observar que el problema descrito anteriormente se debe a que p es un campo que apunta hacia atrás en $\texttt{dlistseg}(h,p,n,l)$ (h reemplaza a p en $\texttt{dlistseg}(h_1, h, null, l)$). □

En general, en presencia de campos que apuntan hacia atrás, las llamadas recursivas dependen de valores asignados a variables antes de ejecutar las mismas. Por lo

tanto, para tratar estos campos se agrega un parámetro adicional, *pa*, al algoritmo SLBD; la nueva versión se muestra en la figura 12. En el caso inductivo de SLBD, se crea en la línea 15 la asignación *sbpa* uniendo el parámetro *pa* y la asignación para romper simetrías *sba*. Intuitivamente, *sbpa* contiene las asignaciones de valores a variables efectuadas previamente a las llamadas recursivas de las líneas 16-18del Alg. 12. Por lo tanto, *sbpa* se usa en las líneas 16-18 para pasar a la llamada recursiva $p_i(y_i^*)$ los valores asignados anteriormente por SLBD a (un subconjunto de) las variables y_i^* (esto es $sbpa[y_i^*]$). Más adelante, en la línea 21, *sbpa* incide en la construcción de asignaciones válidas para los parámetros de la ejecución actual, en combinación con la parte pura del caso inductivo de p, Π_{ind}, y las asignaciones obtenidas recursivamente, $a_1, \ldots, a_n$. Por último, en el caso base de SLBD, *pa* se utiliza como parte del heap simbólico construido para el caso base del predicado inductivo (línea 5), lo que permite evitar el problema del ejemplo 4.2.2.

Veamos dos ejemplos adicionales que ilustran usos concretos de la variable *pa* en el algoritmo SLBD.

Ejemplo 4.2.3. Supongamos que se desean calcular cotas ajustadas para listas doblemente encadenadas, definidas mediante `dlistseg`. Por lo tanto, es necesario un mecanismo para que las variables p y n tomen el valor *null* al inicio de SLBD($\texttt{dlistseg}(h, p, n, l), \ldots$). Para lograr esto, se invoca a SLBD con:

$$pa = \{p = null, n = null\}$$

□

Ejemplo 4.2.4. Veamos los pasos más importantes de la ejecución de SLBD ($\texttt{dlistseg}, 0, 1, pa$), con *pa* como en el ejemplo anterior (4.2.3). Como hay una dirección disponible, la ejecución empieza desplegando el caso inductivo de `dlistseg`:

$$\textsc{Ind}(\texttt{dlistseg}(h, p, n, l)) = (h \mapsto pv : p, nx : h_1 * \texttt{dlistseg}(h_1, h, n, l))$$

Luego, en la línea 12 del algoritmo h recibe el valor N_0:

$$sba = \{h = N_0\}$$

Posteriormente, en la línea 15 se combinan *sba* y *pa* para formar *sbpa*. Intuitivamente, *sbpa* almacena todas las asignaciones de valores a variables de $\textsc{Ind}(\texttt{dlistseg}(h, p, n, l))$ (h, p, n, l, h_1) efectuadas por SLBD con anterioridad a la llamada recursiva ($\texttt{dlistseg}(h_1, h, n, l)$):

$$sbpa = \{p = null, n = null, h = N_0\}$$

Más adelante, en la línea 16 se lleva a cabo la llamada recursiva:

$$\text{SLBD}(\texttt{dlistseg}(h_1, h, n, l), 1, 1, pa')$$

con pa' la restricción de dominio de *sbpa* a las variables h_1, h, n, l, es decir:

$$pa' = \{n = null, h = N_0\}$$

Como:

$$\textsc{Base}(\texttt{dlistseg}(h_1, h, n, l)) = h_1 = n \wedge h = l$$

la llamada recursiva retorna (utilizando los valores de n y h en pa' para construir sa_1):

$$(b_1, sa_1) = (\emptyset, \emptyset, \{h_1 = null, n = null, h = N_0, l = N_0\})$$

Llamemos a_1 a la única asignación en sa_1, en la línea 21 el algoritmo construye la fórmula pura que consta de todas las conjunciones de términos en *sbpa* y a_1:

$$sh = (p = null \wedge n = null \wedge h = N_0) \wedge (h_1 = null \wedge n = null \wedge h = N_0 \wedge l = N_0)$$

La única valuación que satisface esta fórmula es (línea 23):

$$a = \{p = null, n = null, h = N_0, h_1 = null, l = N_0\}$$

que usada para reemplazar las variables de $h \mapsto pv : p, nx : h_1$ resulta en la cota ajustada correspondiente a listas doblemente enlazadas con exactamente un elemento:

$$pv = \{(N_0, null)\} \qquad\qquad nx = \{(N_0, null)\}$$

(Notar que hay sólo una instancia de listas doblemente encadenadas con un nodo N_0, y los campos pv y nx de la misma apuntan a *null*.) □

4.2.2. Tratando con predicados inductivos más generales

Si bien la forma restringida de los predicados considerados hasta el momento permite describir una gran cantidad de estructuras útiles en la práctica –como los diversos tipos de listas y árboles de los ejemplos presentados–, modificar SLBD para soportar predicados más generales es sencillo. De hecho, la forma simplificada de los predicados inductivos consiste en limitar:

1. La cantidad de disyunciones de heaps simbólicos en la definición de los predicados, admitiendo más de un caso base y uno inductivo.

2. El número de celdas $(v \mapsto f_1 : v_1, \ldots, f_n : v_n)$ del heap simbólico que representa el caso inductivo, que podría ser superior a una (a la que frecuentemente llamamos raíz).

El problema 2 es el más simple de solucionar. Primero, por cada celda $v_0 \mapsto f_1 : v_{0,1}, \ldots, f_n : v_{0,n} * \ldots * v_j \mapsto f_1 : v_{j,1}, \ldots, f_n : v_{j,n}$ en el caso inductivo, SLBDGen (la versión más general de SLBD) construye *sba* con las asignaciones $v_0 = N_f$, $v_1 = N_{f+1}$, ..., $v_j = N_{f+j}$ (línea 12), y luego reparte las direcciones restantes $N_{f+j+1}, \ldots, N_{l-1}$ entre las llamadas recursivas (línea 14). Por último, SLBDGen utiliza cada asignación a (línea 23) para generar cotas para todas las celdas $v_1 \mapsto f_1 : v_{1,1}, \ldots, f_n : v_{1,n} * \ldots * v_j \mapsto f_1 : v_{j,1}, \ldots, f_n : v_{j,n}$ (no sólo para la primera). Para solucionar 1 es importante observar que con la definición más amplia de los predicados inductivos no es posible deducir de antemano cuantas direcciones van a requerir los diferentes casos del predicado inductivo p: con la forma restringida de p podíamos asegurar que el caso base no utilizaba direcciones (ver línea 4), y que cada llamada inductiva ocupaba una dirección (ver líneas 10-14), pero esto deja de ser válido en el caso general. Así, SLBDGen cicla sobre todos los posibles heaps simbólicos en la definición de p, computando cotas ajustadas para cada uno de ellos. Para esto, SLBDGen distingue los heaps simbólicos cuya parte espacial es *emp* de los que no. Para los primeros, si $l - f = 0$, SLBDGen ejecuta los pasos en las líneas 5-9 de SLBD original; en caso contrario, es decir, cuando hay nodos del heap que no fueron usados, SLBDGen retorna indefinido ($\perp$). Para los casos inductivos, SLBDGen devuelve $\perp$ cuando los $l - f$ nodos no son suficientes para las celdas $v_0 \mapsto f_0 : v_{1,1}, \ldots, f_n : v_{1,n} * \ldots * v_j \mapsto f_1 : v_{j,1}, \ldots, f_n : v_{j,n}$ en el caso inductivo actual, esto es, cuando $l - f < j$. En caso contrario, SLBDGen ejecuta las líneas 11-30 de SLBD, modificadas para tratar con más de una raíz, como se describió en el párrafo anterior. Para finalizar, SLBDGen retorna la union de todas las cotas ajustadas (y conjuntos de asignaciones) producidas para los heaps simbólicos en la definición de p que pueden completarse con exactamente $l - f$ nodos –aquellos para los que el algoritmo no devuelve $\perp$.

Con estas modificaciones, SLBDGen es capaz de tratar con los predicados inductivos definibles mediante la sintaxis dada la figura 2.10 (sección 2.9.3), es decir, predicados inductivos cuyos diferentes casos se dan en términos de heaps simbólicos, compuestos por una parte espacial (secuencias de fórmulas espaciales unidas por $*$) que define el heap y una parte pura (secuencias de fórmulas puras unidas por $\wedge$) que restringe los valores de las variables.

Es importante destacar que el poder expresivo de estos heaps simbólicos es menor que el del lenguaje JML usado para especificar los invariantes en el enfoque bottom-up del capítulo anterior (y en TACO+). Esto se debe a la dificultad de expresar estructuras con subestructuras compartidas (*sharing*) en nuestra forma restringida de la lógica de separación. Por ejemplo, estructuras como los grafos acíclicos no pueden definirse utilizando secuencias de fórmulas espaciales unidas por el operador $*$ (las únicas disponibles en nuestro lenguaje). Para especificar estas estructuras es necesario introducir operadores adicionales, como el operador **trees** de [13]. Esto significa que tanto bottom-up como TACO+ permiten computar cotas ajustadas para una cantidad mayor variedad de estructuras que SLBDGen. Como trabajo futuro se propone estudiar como extender SLBDGen para soportar fragmentos más expresivos de la lógica de separación.

Por último, se harán algunas consideraciones sobre la relación entre la forma de los predicados inductivos y la eficiencia de SLBDGen. En este sentido, es importante aclarar que la mayoría de los predicados inductivos usados en este trabajo (las definiciones de listas, árboles, y AVLs del ejemplo 4.1.1) se tomaron sin modificaciones de [39]. Por otro lado, se dió una definición estándar de árboles rojos y negros (TreeSet) en lógica de separación. Es decir, no se definieron los predicados inductivos de manera "ad hoc" con la intención de favorecer al algoritmo SLBDGen. La única decisión tomada sobre la forma de los predicados que puede impactar positivamente en el rendimiento de SLBDGen, es la de acomodar en las partes espaciales de las definiciones inductivas las fórmulas del tipo $x \mapsto x_1, \ldots, x_n$ antes que las apariciones inductivas a los predicados. Sin embargo, esta "forma canónica" para las partes espaciales podría obtenerse mediante un preprocesamiento automático de los heaps simbólicos.

4.3. Memorización

El tiempo de ejecución del algoritmo SLBD (Alg. 12) en el peor caso es exponencial. La razón es que cada ejecución de SLBD realiza i llamadas recursivas (líneas 16-18), donde i es la cantidad de predicados inductivos que aparecen en el caso inductivo de p. Para empeorar las cosas, la cantidad de llamadas recursivas se multiplica por el número de posibles particiones de los $l - (f + 1)$ nodos de entrada en i intervalos (línea 14).

Ejemplo 4.3.1. Cada iteración del ciclo de la línea 14 de SLBD($\texttt{btree}(t), 0, 3, \{\}$)

ejecuta:

$$\text{SLBD}(\texttt{btree}(t_1), 1, 1, \{\}) \text{ y } \text{SLBD}(\texttt{btree}(t_2), 1, 3, \{\})$$
$$\text{SLBD}(\texttt{btree}(t_1), 1, 2, \{\}) \text{ y } \text{SLBD}(\texttt{btree}(t_2), 2, 3, \{\})$$
$$\text{SLBD}(\texttt{btree}(t_1), 1, 3, \{\}) \text{ y } \text{SLBD}(\texttt{btree}(t_2), 3, 3, \{\})$$

La primera considera el árbol izquierdo vacío y árboles derechos con dos nodos; la segunda trata el caso en el que tanto el árbol izquierdo como el derecho tienen un único nodo; la tercera tiene en cuenta el caso opuesto al primero, donde el árbol derecho debe ser vacío y los árboles izquierdos tienen dos nodos. □

Sin embargo, notemos que en el ejemplo anterior muchas de las llamadas recursivas resuelven múltiples veces el mismo problema:

- $\text{SLBD}(\texttt{btree}(t_1), 1, 1, \{\})$ y $\text{SLBD}(\texttt{btree}(t_2), 3, 3, \{\})$ computan cotas ajustadas para un heap sin nodos, que representa un árbol vacío.
- $\text{SLBD}(\texttt{btree}(t_1), 1, 2, \{\})$ y $\text{SLBD}(\texttt{btree}(t_2), 2, 3, \{\})$ computan cotas para un heap que denota un árbol binario con un único nodo.
- $\text{SLBD}(\texttt{btree}(t_1), 1, 3, \{\})$ y $\text{SLBD}(\texttt{btree}(t_2), 1, 3, \{\})$ computan cotas para heaps que contienen árboles binarios con exactamente dos nodos.

Por lo tanto, la solución natural para mejorar la eficiencia en tiempo de SLBD es adaptarlo para soportar la técnica denominada *memorización*. Memorización es una técnica de optimización de programas que consiste en almacenar en una estructura de datos apropiada la solución a subproblemas (del problema original) computada por llamadas recursivas finalizadas, de modo que los resultados pueden obtenerse directamente de la estructura en caso de que una llamada recursiva posterior intente resolver el mismo problema, evitando así llevar a cabo computaciones innecesarias. Por ejemplo, en la ejecución descrita en el ejemplo 4.3.1 los resultados de $\text{SLBD}(\texttt{btree}(t_2), 1, 3, \{\})$ (cotas ajustadas para árboles binarios con dos nodos) pueden ser usados posteriormente por $\text{SLBD}(\texttt{btree}(t_1), 1, 3, \{\})$ (aunque los parámetros de `btree` sean diferentes; ver abajo). Las estructuras de datos utilizadas más frecuentemente en la optimización vía memorización son los arreglos y las matrices, debido a que permiten acceso directo y eficiente a las soluciones.

La idea intuitiva de la versión de SLBD basada en memorización –SLBDM a partir de este momento– es almacenar los pares de cotas ajustadas y conjuntos de asignaciones que va computando durante su ejecución en un arreglo m. Así, SLBDM mantiene el siguiente invariante respecto de m: si $m[k] \neq null$, $m[k]$ almacena pares de cotas ajustadas y conjuntos de asignaciones de valores a los parámetros de p para heaps que satisfacen p y poseen exactamente k nodos. Por ejemplo, el resultado de $\text{SLBD}(\texttt{btree}(t), 0, 3, \{\})$ se almacenará en $m[3-0] = m[3]$.

4.3.1. Forma canónica de los resultados

Notar que diferentes ejecuciones de SLBDM pueden computar resultados equivalentes, pero que difieren en los identificadores de nodos y/o nombres de variables utilizados. Por ejemplo:

$$\text{SLBD}(\texttt{btree}(t), 0, 2, \{\}) = \tag{4.3}$$

$$(\{\{(N_0, N_1), (N_1, null)\}, \{(N_0, N_1), (N_1, null)\}\}, \{t = N_0\}) \tag{4.4}$$

$$\text{SLBD}(\texttt{btree}(t_1), 1, 3, \{\}) = \tag{4.5}$$

$$(\{\{(N_1, N_2), (N_2, null)\}, \{(N_1, N_2), (N_2, null)\}\}, \{t_1 = N_1\}) \tag{4.6}$$

Ambas invocaciones retornan cotas ajustadas para árboles binarios con dos nodos (la primera cota corresponde al campo izquierdo y la segunda al campo derecho), aunque en el primer caso los identificadores de nodos usados son N_0 y N_1, y en el segundo N_1 y N_2. Además, aunque los nombres de los parámetros de `btree` son diferentes (t y t_1, respectivamente), las asignaciones también son equivalentes: el valor asignado al parámetro de `btree` es el primer nodo en los dos casos (N_0 y N_1, respectivamente). Observemos que, si en 4.6 cambiamos la variable t_1 por t, y restamos uno a todos los índices de sus identificadores de nodo, obtenemos 4.4.

Por lo anterior, es necesario elegir una *representación canónica* para el almacenamiento de los resultados en m. De esta manera, para k nodos de entrada, se propone usar los identificadores de nodos $N_0, \ldots, N_{k-1}$ en la forma canónica. Además, si $x_0, x_1, \ldots$ son variables especiales, que no pueden aparecer en las definiciones de los predicados inductivos, y si $y_0, y_1, \ldots, y_j$ son los parámetros del predicado de entrada p, la variable x_i $(0 \leq i \leq j)$ se utilizará como marcador de posición para el parámetro y_i en la forma canónica de las asignaciones. Así, los resultados de 4.3 y 4.5 se guardan en $m[2]$ en la forma canónica:

$$(\{\{(N_0, N_1), (N_1, null)\}, \{(N_0, N_1), (N_1, null)\}\}, \{x_0 = N_0\}) \tag{4.7}$$

Para transformar los resultados de SLBD a su forma canónica se definen las funciones auxiliares $\textsc{Desplazar}_B$, $\textsc{Desplazar}_A$ y $[x^*/y^*]$. $\textsc{Desplazar}_B(b, f)$ suma f a los índices de cada uno de los pares en la cota ajustada b:

$$\textsc{Desplazar}_B(b, f) = \{\{((\textsc{sum}_i(f, p_1), \textsc{sum}_i(f, p_2)) | (p_1, p_2) \in b.i\} | i \in 1..n\}$$

En la definición anterior, $b.i$ representa la cota ajustada para el campo i-ésimo (recordar que se asume un conjunto fijo de campos $f_1, \ldots, f_n$), y la función $\textsc{sum}_i$ modifica los índices de los indentificadores de nodos, y retorna la identidad para $null$:

$$\textsc{sum}_i(f, null) = null$$
$$\textsc{sum}_i(f, N_i) = N_{i+f}$$

De manera similar, $\textsc{Desplazar}_A(a, f)$ suma f a los índices de los nodos que aparecen en la asignación a:

$$\textsc{Desplazar}_A(a, f) = \{v = \textsc{sum}_i(n) | v = n \in a\}$$

Por último, dada una asignación a, $a[x^*/y^*]$ denota el resultado de reemplazar cada variable $y_i \in y^*$ por su correspondiente $x_i \in x^*$ en a. Si sa es un conjunto de asignaciones, $sa[x^*/y^*]$ efectúa el reemplazo $[x^*/y^*]$ en cada una de las asignaciones miembros de sa. Formalmente, $sa[x^*/y^*] = \{a[x^*/y^*] | a \in sa\}$.

La conversión de un resultado (rb, sa) a su forma canónica mediante las funciones definidas arriba se muestra a continuación:

$$(\textsc{Desplazar}_B(rb, -f), \textsc{Desplazar}_A(sa, -f)[x^*/y^*])$$

donde f es el menor de los índices de los nodos nodo en rb.

Ejemplo 4.3.2. Sean (rb_1, sa_1) y (rb_2, sa_2) los resultados 4.4 y 4.6, respectivamente, canonizarlos con el método anterior retorna en el par 4.7:

$$\begin{aligned}
&(\textsc{Desplazar}_B(rb_1, -0), \textsc{Desplazar}_A(sa_1, -0)[x_0/t]) \\
&= (\textsc{Desplazar}_B(rb_2, -1), \textsc{Desplazar}_A(sa_2, -1)[x_0/t_1]) \\
&= (\{\{(N_0, N_1), (N_1, null)\}, \{(N_0, N_1), (N_1, null)\}\}, \{x_0 = N_0\})
\end{aligned}$$

□

El procedimiento inverso es también necesario: llevar los resultados almacenados en m en forma canónica a la forma esperada por las llamadas recursivas. Para esto también pueden usarse las funciones $\textsc{Desplazar}_B$, $\textsc{Desplazar}_A$ y $[x^*/y^*]$.

Ejemplo 4.3.3. Si $m[2] \neq null$, la llamada recursiva SLBD($\texttt{btree}(t_1), 1, 3, \{\}$) (4.5) debe desplazar los índices de nodos –para usar N_1 y N_2– y los marcadores de posición –por la variable t_1– en $m[2]$ antes de retornar. Esto se logra invocando a:

$$(\textsc{Desplazar}_B(rb, 1), \textsc{Desplazar}_A(sa, 1)[t/x_0])$$

con $m[2] = (rb, sa)$ (4.7). Observar que, el resultado de esta operación es 4.6. □

4.3.2. Algoritmo SLBDM

La figura 13 presenta el pseudocódigo del algoritmo SLBDM: la versión de SLBD optimizada vía memorización. De la misma manera que en la explicación de SLBD, el tratamiento de campos que apuntan hacia atrás se discutirá en una

Algorithm 13 `SLBDM`

1: **function** SLBDM$(p(v^*), f, l, m)$
2: ▷ Si el subproblema ya fue resuelto anteriormente, retornar los resultados almacenados.
3: **if** $m[l-f] \neq null$ **then**
4: $(rb, sa) := m[l-f]$
5: **return** $(\textsc{Desplazar}_B(rb, f), \textsc{Desplazar}_A(sa, f)[x^*/v^*])$
6: $rb := \emptyset, \ldots, \emptyset$
7: $sa := \emptyset$
8: **if** $l - f = 0$ **then**
9: **let** $\textsc{Base}(p(v^*)) = sh = emp \wedge \Pi_b$ **in**
10: $spec := \textsc{Codificar}(sh, f, l, \text{"}output.ys\text{"})$
11: **for** $a \in \textsc{SatAssigns}(spec)$ **do**
12: $sa := sa \cup \{a[v^*]\}$
13: **else**
14: ▷ $l - f > 0$
15: **let** $\textsc{Ind}(p(v^*)) = v \mapsto f_1 : v_1, \ldots, f_n : v_n * p_1(y_1^*) * \ldots * p_i(y_i^*) \wedge \Pi_{ind}$ **in**
16: $sba := \{v = N_f\}$
17: **for** $(f_1, l_1), \ldots, (f_i, l_i) \in \textsc{RepartirI}(f+1, l, i)$ **do**
18: $(b_1, sa_1) =$SLBDM$(p_1(y_1^*), f_1, l_1, m)$
19: $\ldots$
20: $(b_i, sa_i) =$SLBDM$(p_i(y_i^*), f_i, l_i, m)$
21: $sat := false$
22: **for** $a_1, \ldots, a_i \in sa_1 \times \ldots \times sa_i$ **do**
23: $sh := emp \wedge (\Pi_{ind} \wedge \textsc{Conjunciones}(sba \cup a_1 \cup \ldots \cup a_n))$
24: $spec := \textsc{Codificar}(sh, f, l, \text{"}output.ys\text{"})$
25: **for** $a \in \textsc{SatAssigns}(spec)$ **do**
26: $rootb := \textsc{RepVars}(v \mapsto f_1 : v_1, \ldots, f_n : v_n, a)$
27: $rb := rb \cup rootb$
28: $sa := sa \cup \{a[v^*]\}$
29: $sat := true$
30: **if** sat **then**
31: $rb := rb \cup b_1 \cup \ldots \cup b_i$
32: ▷ Almacenar los resultados computados antes de retornar.
33: $m[l-f] := (\textsc{Desplazar}_B(rb, -f), \textsc{Desplazar}_A(sa, -f)[x^*/v^*])$
34: **return** (rb, sa)

sección posterior (4.3.3). La primera diferencia entre SLBDM y el original (Alg. 12) es el parámetro adicional m: el arreglo donde SLBDM guarda las soluciones a subproblemas que va computando durante su ejecución. La segunda diferencia

reside en que, si existe una solución para el subproblema actual almacenada en m ($m[l-f] \neq null$ en la línea 3 del algoritmo), SLBDM la adecúa a la llamada recursiva actual –las soluciones se guardan en m en forma canónica– y la retorna sin realizar cómputos adicionales (líneas 4 y 5; este procedimiento se discutió en la parte final de la sección 4.3.1). Por último, SLBDM debe guardar la solución al subproblema corriente en m antes de retornar. Así, el algoritmo canoniza (sección 4.3.1)) la solución computada, (rb, sa), y la almacena en $m[l-f]$ (línea 33).

Para concluir esta sección, es importante observar que la técnica de memorización incrementa los requerimientos de memoria principal de los programas a los que se aplica. Claramente, esto se debe a que las soluciones a subproblemas que el algoritmo va computando se mantienen en memoria durante toda su ejecución –en la estructura de datos designada para este propósito. Esto puede traer problemas en casos donde la solución a un subproblema consta de una gran cantidad de datos. Por ejemplo, aplicar memorización al algoritmo por fuerza bruta de la sección 4.1 requiere almacenar una cantidad exponencial (respecto del scope) de heaps simbólicos, pero esto es prohibitivo debido a su alto costo en memoria. SLBDM no sufre de este problema ya que, para los scopes usados en la práctica para los análisis exhaustivos acotados (rara vez exceden los 20 elementos por dominio de datos), las cotas ajustadas para el heap no superan los cientos de pares en cada conjunto –los experimentos realizados como parte de nuestra evaluación experimental (capítulo 5) son evidencia de esto.

4.3.3. Memorización y campos que apuntan hacia atrás

Los campos que apuntan hacia atrás presentan un problema adicional para el algoritmo SLBDM. Para ilustrarlo consideremos el predicado `btreep`, que define árboles binarios cuyos nodos poseen un campo adicional, pr, de manera que $n.pr$ es un puntero al padre del nodo n en el árbol:

$$\begin{aligned}\texttt{btreep}(t,p) \doteq &(\texttt{emp} \wedge t = null \wedge p = null) \vee \\ &(t \mapsto l : t_1, r : t_2, pr : p * \texttt{btreep}(t_1, t) * \texttt{btreep}(t_2, t))\end{aligned}$$

Analicemos los resultados de dos ejecuciones diferentes de SLBD que computan cotas ajustadas para heaps equivalentes: ambos representan un árbol binario con un único nodo:

$$\text{SLBD}(\texttt{btreep}(t,p), 0, 1, \{p = null\}) = \tag{4.8}$$

$$(\{\{(N_0, null)\}, \{(N_0, null)\}, \{(N_0, null)\}\}, \{t = N_0, p = null\}) \tag{4.9}$$

$$\text{SLBD}(\texttt{btreep}(t_1, p_1), 1, 2, \{p_1 = N_0\}) = \tag{4.10}$$

$$(\{\{(N_1, null)\}, \{(N_1, null)\}, \{(N_1, N_0)\}\}, \{t_1 = N_1, p_1 = N_0\}) \tag{4.11}$$

En las cotas ajustadas anteriores el primer conjunto corresponde al campo l, el segundo a r y el tercero a pr. Notar que ambos resultados deberían almacenarse en $m[1]$, sin embargo, esto no es posible debido a que difieren en el valor del campo padre del nodo raíz (N_0 en el primer caso, N_1 en el segundo). Observar además que en ambos casos el valor asignado al campo padre viene determinado por la asignación pa –el cuarto parámetro de SLBD.

Para solucionar este problema se propone insertar marcadores de posición $x'_0, x'_1, \ldots$ en la forma canónica de los resultados, para los valores de los parámetros $y_0, y_1, \ldots$ de p, respectivamente, siempre que su valor venga dado en pa. Por ejemplo, la forma canónica de 4.9 se muestra abajo, en donde x'_1 es el marcador de posición para el valor de la variable p. x'_0 correspondería a t, pero este marcador no se usa porque t no tiene un valor asignado en pa.

$$(\{\{(N_0, null)\}, \{(N_0, null)\}, \{(N_0, x'_1)\}\}, \{x_0 = N_0, x_1 = x'_1\}) \tag{4.12}$$

Notar que la anterior es también la fórma canónica de 4.11.

El procedimiento de construcción de un resultado válido del algoritmo a partir de uno canónico, como 4.12, es sencillo: además de realizar el procedimiento inverso a la canonización discutido en la sección 4.3.1, se deben reemplazar los marcadores de posición $x'_0, x'_1, \ldots$ por sus valores correspondientes en pa. Por ejemplo, si llamamos (rb, sa) a 4.12, el resultado de SLBD($\texttt{btreep}(t_1, p_1), 1, 2, \{p_1 = N_0\}$) se construye como se muestra a continuación:

$$\begin{aligned}&(\textsc{Desplazar}_B(rb, 1)[x'_1/pa(p_1)], \textsc{Desplazar}_A(sa, 1)[x_0/t][x'_1/pa(p_1)])\\ &= (\{\{(N_1, null)\}, \{(N_1, null)\}, \{(N_1, N_0)\}\}, \{t_1 = N_1, p_1 = N_0\})\end{aligned}$$

4.4. Cotas ajustadas primero en profundidad

Los predicados de la sección 2.6.1 rompen simetrías asignando identificadores a los elementos del heap durante un recorrido primero en anchura de los campos, por lo que los denominamos predicados de rotura de simetrías BF.

Ejemplo 4.4.1. El árbol binario de la figura 4.7 satisface los predicados de rotura de simetría BF. Veamos algunas de las razones del etiquetado del árbol de la figura. En el árbol, T_0 es el único objeto de tipo `Tree` –y la única raíz del heap–, y $N_0, \ldots, N_4$ son objetos de tipo `Node`. El nodo raíz del árbol es N_0 por ser el único nodo cuyo padre es una raíz del heap (ver regla 1 de la sección 2.6.1). Por la regla 4 de 2.6.1, los hijos izquierdo y derecho de N_0 –asumiendo que el campo `left` se declara en el código antes que el campo `right`, y que no quedan identificadores "intermedios" sin asignar– deben tener los identificadores N_1 y N_2, respectivamente. Un razonamiento similar justifica la asignación de los identificadores N_3 y N_4 a los nodos correspondientes. □

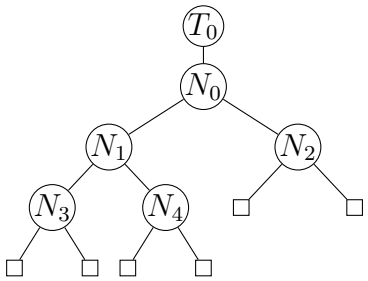


Figura 4.7: Árbol binario con sus nodos etiquetados en un recorrido primero en anchura de los campos `left` y `right`.

Sin embargo, el algoritmo SLBD computa cotas ajustadas en un recorrido *primero en profundidad* (*depth first*) de los campos del heap. Como ejemplo, analicemos el caso inductivo de SLBD (Alg. 11) cuando se invoca con el predicado `btree` (ver ejemplo 4.1.1) como entrada. El primer paso del algoritmo consiste en asignar el primer identificador N_f a la raíz (línea 12). Luego, por cada partición $[N_{f+1}, \ldots, N_k], [N_{k+1}, \ldots, N_{l-1}]$ de la secuencia ordenada de nodos de entrada $[N_f, \ldots, N_{l-1}]$ (línea 14), SLBD recorre recursivamente todos los posibles árboles izquierdos con los identificadores $[N_{f+1}, \ldots, N_k]$ (línea 15), para después recorrer recursivamente los árboles derechos con los restantes $[N_{k+1}, \ldots, N_{l-1}]$. Claramente, este es un recorrido primero en profundidad de las estructuras definidas por `btree`. Por este motivo, las cotas producidas por SLBD se denominan cotas ajustadas primero en profundidad o *cotas ajustadas DF* (*depth first*). El ejemplo a continuación muestra que las cotas ajustadas DF son incompatibles con los predicados de rotura de simetría BF.

Ejemplo 4.4.2. Las cotas ajustadas DF generadas por SLBD para el predicado `btree` con hasta 5 nodos son:

$$\begin{aligned} left =& \{(N_0, null), (\mathbf{N_1}, \mathbf{null}), (N_0, N_1), (N_2, null), (\mathbf{N_1}, \mathbf{N_2}), (N_3, null), \\ & (N_2, N_3), (N_4, null), (N_3, N_4)\} \\ right =& \{(N_0, null), (N_0, N_1), (N_1, null), (N_1, N_2), (N_2, null), (N_0, N_2), \\ & (N_2, N_3), (N_3, null), (N_1, N_3), (N_0, N_3), (N_3, N_4), (N_4, null), \\ & (N_2, N_4), (N_1, N_4), (N_0, N_4)\} \end{aligned}$$

Observar que usando las cotas anteriores el nodo izquierdo de N_1 puede ser o bien *null* o bien N_2 (ver los pares en negrita). Por lo tanto, las cotas no permiten generar el árbol (válido) de la figura 4.7, etiquetado en orden primero en anchura. El mismo árbol pero con sus nodos etiquetados en orden primero en profundidad, y,

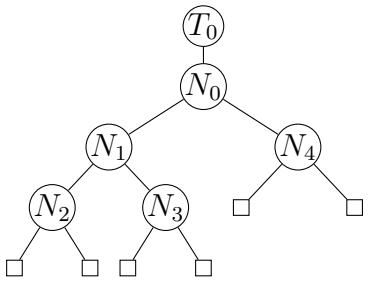


Figura 4.8: Árbol binario con sus nodos etiquetados en un recorrido primero en profundidad de los campos `left` y `right`.

por lo tanto, compatible con las cotas ajustadas DF, se muestra en la figura 4.8. Sin embargo, este árbol no está permitido por los predicados de rotura de simetrías BF (entre otras cosas, porque el hijo derecho de N_0 debería ser N_2). □

Recordemos que uno de nuestros objetivos es incorporar las cotas ajustadas computadas por SLBD en el análisis basado en SAT TACO (sección 2.5.1). El ejemplo anterior revela que los predicados de rotura de simetrías BF definidos para TACO son incompatibles con las cotas ajustadas DF producidas por SLBD: las cotas eliminan instancias que son válidas según los predicados, y viceversa. Por lo tanto, para cumplir con nuestro propósito tenemos dos opciones: desactivar rotura de simetrías, con su consecuente impacto negativo en el rendimiento del análisis; o definir predicados de rotura de simetrías compatibles con las nuevas cotas (DF). Este es el propósito de la próxima sección.

4.4.1. Predicados de rotura de simetrías primero en profundidad

Para romper simetrías en un recorrido primero en profundidad de los campos del heap, se mantiene la forma de nombrar los elementos de las signaturas, y las divisiones de los campos en *forward* y *backward* comentadas en la sección 2.6.1. Además, se define la función auxiliar `FReachN`, similar a `FReach`, pero que permite obtener el conjunto de elementos del heap alcanzables a partir un elemento dado, realizando cero o más dereferencias de campos que apuntan hacia adelante. Retomando el ejemplo de árboles binarios de la sección 2.6.1, a continuación se define `FReachN` para este caso de estudio:

```
fun FReachN[n: Node]: set Node
{
  n.*(fleft + fright) - null
```

```
}
```

Por otro lado, `FReachNP` retorna el conjunto de elementos alcanzables a partir de un elemento efectuando al menos una dereferencia de campos *forward*:

```
fun FReachNP[n: Node]: set Node
{
  n.^(fleft + fright) - null
}
```

Como en la sección 2.6.1, se darán a continuación una serie de reglas cuya función es ordenar los nodos de tipo T en el heap. Al leer las definiciones, tener en cuenta que la noción de alcanzabilidad está definida en base a los campos *forward*.

1. Sean r_1 y r_2 son raíces, tales que el tipo de r_1 es anterior al de r_2 en la lista de parámetros del método a analizar y r_2 no es alcanzable desde r_1. Entonces, sean n_1 y n_2 nodos distintos de tipo T, tal que n_1 es alcanzable desde r_1, n_2 es alcanzable desde r_2 y n_2 no es alcanzable desde r_1, entonces n_1 es menor que n_2. Intuitivamente, esto significa que todos los nodos alcanzables desde r_2 que no son alcanzables desde r_1 deben ser menores que todos los alcanzables a partir de r_1 (para r_1 definido antes que r_2 en el código; r_2 no alcanzable a partir de r_1).

2. Si n_1 y n_2 son nodos distintos de tipo T tal que n_2 es alcanzable a partir de n_1 dereferenciando al menos un campo (`FReachNP`), entonces el identificador de n_1 tiene que ser menor al de n_2.

3. Sean n_1 y n_2 nodos distintos que comparten el mismo padre n, es decir, existen dos campos distintos f_1 y f_2 tales que $n.f_1 = n_1$ y $n.f_2 = n_2$. Supongamos que f_1 se declara en el código antes que f_2. Entonces, todos los nodos n_2' de tipo T alcanzables a partir de n_2 que no son alcanzables a partir de n_1 deben ser menores a todos los n_1' de tipo T alcanzables desde n_1.

Así, la especificación Alloy que codifica el programa a analizar debe instrumentarse con un hecho para cada una de las reglas anteriores, por cada signatura T de la misma.

Ejemplo 4.4.3. Retomemos el ejemplo de árboles binarios de la sección 2.6.1. Recordemos que en este caso de estudio hay una única raíz, de tipo `Tree`. Consecuentemente, no se introducen hechos para la signatura `Tree`, ni para el punto 1 anterior.

Para hacer valer la regla 2 para la signatura `Node` se introduce el siguiente hecho:

```
fact condicionDeOrdenNode2[] {
  all disj n1, n2: FReach[] |
    n2 in FReachNP[n1] => n1 in node_prevs[n2]
}
```

Esto es, para todo nodo `n2` alcanzable a partir de `n1` en al menos un paso, `n1` debe recibir un identificador menor a `n2`.

El hecho a continuación asegura el cumplimiento de la regla 3:

```
fact condicionDeOrdenNode3[] {
  all n: FReach[] | all n2: FreachN[n.fright] |
    n2 not in FreachN[n.fleft] => FReach[n.fleft] in node_prevs[n2]
}
```

Este hecho afirma que, debido a que `left` se declara antes que `right`, entonces los nodos alcanzables a partir del hijo izquierdo de `n` (`n.left`) son todos menores que los (`n2`) alcanzables a desde el hijo izquierdo de `n` que no son alcanzables desde `n.left`.

Por último, es importante recordar que la instrumentación debe incluir el hecho `compactarNode`, para no permitir "huecos" en la asignación de identificadores a nodos.

```
fact compactarNode {
  all n: Node | n in FReach[] => prevsNode[n] in FReach[]
}
```

`compactarNode` asegura que, si se usa un identificador `n`, entonces todos los `m` menores que `n` debe haber sido usados. □

Ejemplo 4.4.4. El árbol binario de la figura 4.8 satisface las condiciones 1-3 anteriores. Para todo nodo del árbol se cumple que su padre tiene un identificador menor, es decir, vale la regla 2. Con respecto a la regla 3, observemos que los identificadores de los nodos alcanzables desde el hijo izquierdo de N_0 (N_1, N_2, N_3) son todos menores que los alcanzables desde el hijo derecho de N_0 (N_4) (y el hijo derecho de N_0 no es alcanzable a partir de su hijo izquierdo). Un razonamiento similar permite concluir que los hijos izquierdo y derecho de N_1 están etiquetados correctamente. □

4.5. Interacción entre el cómputo de cotas basado en predicados inductivos y TACO

En esta sección se explica como incorporar el algoritmo de cómputo de cotas ajustadas basado en predicados inductivos de la lógica de separación, SLBD, al análisis

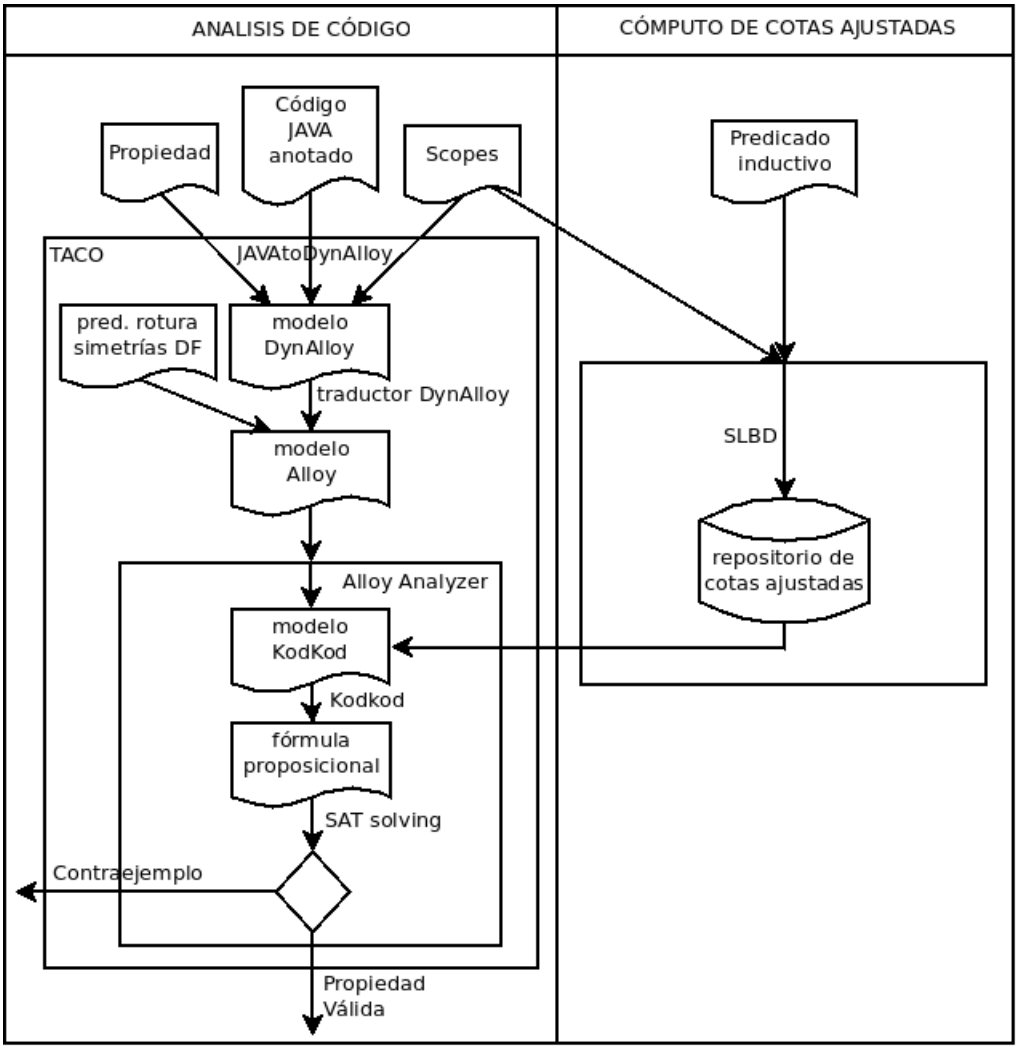


Figura 4.9: Diagrama del análisis de código SLBD+TACO.

de código TACO (sección 2.5.1). Al igual que con las técnicas bottom-up+TACO (seccción 3.5) y TACO+ (sección 2.8), la idea es computar cotas ajustadas mediante el enfoque SLBD y luego ejectuar el verificador de código TACO instrumentado con las cotas producidas. Llamaremos a este análisis SLBD+TACO. Un diagrama de SLBD+TACO se muestra en la figura 4.9. La diferencia principal entre los análisis que usan invariantes declarativos (bottom-up+TACO y TACO+) y SLBD+TACO es que, en este último, el usuario debe proveer una especificación de la clase a analizar en términos de un predicado inductivo de la lógica de separación. El predicado inductivo se usará únicamente para computar cotas ajustadas mediante SLBD (ver

la parte derecha de la figura 4.9). Las precondiciones, postcondiciones e invariantes del método JAVA a verificar (con TACO) deben proveerse por separado (ver las entradas del análisis de código en la parte derecha de la figura 4.9). Otra diferencia es que, dado que SLBD produce cotas ajustadas DF, se debe instrumentar TACO con predicados de rotura de simetrías compatibles con estas cotas (sección 4.4), es decir, predicados de rotura de simetrías DF (en lugar de los predicados de rotura de simetrías BF que se usan en bottom-up+TACO y TACO+).

Desafortunadamente, no existen algoritmos para traducir especificaciones declarativas (por ejemplo, JML) en predicados inductivos de la lógica de separación. Por lo tanto, los predicados inductivos que se usan como entrada de SLBD+TACO en nuestra evaluación experimental deben definirse manualmente. Por otra parte, tampoco se conocen formas eficientes de codificar predicados inductivos en lenguajes declarativos, lo que permitiría usar los mismos predicados inductivos como anotaciones en el código JAVA. Este es un problema a investigar en el futuro.

Capítulo 5

Evaluación experimental

En este capítulo se evalúan experimentalmente las técnicas bottom-up y SLBD. Los experimentos se dividen en dos partes. En primer lugar, en la sección 5.1 se comparan los tiempos de cómputo de cotas ajustadas para cada enfoque, en varios casos de estudio, en un entorno de ejecución secuencial (una PC, un núcleo). La intención de este experimento es concluir sobre la eficiencia de los distintas técnicas. En segundo lugar, en la sección 5.2 se evalúa el impacto de las cotas ajustadas DF, producidas por SLBD, en el análisis de código basado en SAT TACO, respecto de las cotas ajustadas computadas BF por TACO+ (y bottom-up), y del análisis TACO sin usar cotas ajustadas. Además, se comparan las cotas ajustadas DF y BF en la generación exhaustiva acotada (basada en SAT) de estructuras que satisfacen un predicado (invariantes de clase en nuestro caso), basada en SAT, en la sección 5.3. Para esta comparación, en primer lugar se ejecuta la generación de estructuras sin cotas ajustadas, y luego la generación instrumentada con las cotas ajustadas DF y BF. El objetivo de esta segunda tanda de experimentos es medir la utilidad de los distintos tipos de cotas ajustadas en los análisis basados en SAT considerados.

Para la evaluación experimental se consideran como casos de estudio las siguientes clases contenedoras (*container classes*) JAVA, listadas según la complejidad de sus invariantes de clase (de menor a mayor):

LList: una implementación de secuencias basada en listas simplemente encadenadas.

AList: la implementación `AbstractLinkedList` de la interface `List` del paquete commons.collections de Apache, basada en listas circulares doblemente encadenadas.

CList: la implementación `NodeCachingLinkedList` de la interface `List`, también del paquete commons.collections de Apache.

BSTree: una implementación de árboles binarios de búsqueda usada como parte de la evaluación experimental de [47]

TreeSet: la clase `TreeSet` del paquete java.util, que implementa árboles rojos y negros (*Red-Black Trees*).

AVL: una implementación de árboles `AVL` utilizada como caso de estudio en [7].

BHeap: una implementación de binomial heaps extraída de los experimentos realizados en [47].

Estos casos de estudio se tomaron de la evaluación experimental de [25]. Las clases poseen anotaciones –pre y postcondiciones para métodos, invariantes– en lenguaje JML.

El componente de cómputo de cotas de TACO+ es el único enfoque paralelo en la evaluación experimental. Los tiempos de cómputo de cotas mediante TACO+ se tomaron de [25], en donde el algoritmo se ejecutó en un cluster de 16 computadoras, cada una con dos procesadores Intel Dual Core Xeon (para un total de 64 núcleos). La velocidad de cada núcleo es de 2.66GHz, con 2MB por núcleo de cache (8MB por procesador), y 2GB de memoria principal por computadora. Todos los algoritmos restantes son secuenciales. Estos se ejecutaron en una computadora con procesador Intel Core i5-750 de 2.66Ghz, con 3.2Ghz de velocidad máxima (turbo), 8MB de cache y 4GB de memoria principal. Para simular el comportamiento del algoritmo de cómputo de cotas ajustadas de TACO+ esta estación de trabajo se tomaron los tiempos de reloj de [25] (*wall time*), se multiplicaron por la cantidad de núcleos (64) y por el coeficiente 0,831 para compensar por la diferencia entre los relojes de la computadora individual (3.2Ghz) y las del cluster (2.66Ghz).

5.1. Evaluación de los diferentes enfoques de cómputo de cotas ajustadas

En esta sección se miden los tiempos de cómputo de cotas ajustadas para los algoritmos bottom-up, SLBD y TACO+. Para comenzar, en 5.1.1 se comparan las distintos algoritmos derivados de bottom-up entre sí, para seleccionar al mejor de ellos para posteriores comparaciones con los enfoques relacionados. En 5.1.2 se discuten las ventajas de usar cómputo incremental de cotas ajustadas vía bottom-up respecto del cómputo tradicional (desde cero), en el caso en que se desean computar cotas ajustadas para scopes cada vez más grandes (ver sección 3.4). Luego, en 5.1.3 se compara la mejor versión de bottom-up contra el módulo de cómputo de cotas

de TACO+, simulando la ejecución de este último en un entorno secuencial (como se mencionó en el último párrafo de la sección previa). Es importante recordar que estos dos algoritmos son directamente comparables, ya que ambos utilizan una descripción del invariante de clase en Alloy, derivada automáticamente del invariante JML de la clase correspondiente. Por último, en 5.1.4 se extiende la comparación anterior para incluir al algoritmo de cómputo de cotas ajustadas basado en predicados inductivos de la lógica de separación, SLBD. Los predicados inductivos requeridos por el enfoque se definieron traduciendo manualmente los invariantes JML de las clases consideradas.

5.1.1. Evaluación de las distintas versiones de bottom-up

El objetivo de esta primera ronda de experimentos es comparar los tiempos de ejecución de las diferentes versiones de bottom-up en el cómputo de cotas ajustadas, en un entorno de ejecución secuencial (la PC usada se describió en la sección 5). Se impuso un límite máximo de 3 horas para la ejecución de los algoritmos. Las ejecuciones que superan este límite se reportan como TO (*timeout*). La tabla 5.1 muestra los resultados de este experimento. En la tabla se reportan los tiempos de ejecución de cada algoritmo, en formato `hh:mm:ss` (horas, minutos y segundos), para todos los casos de estudio mencionados en la sección 5, y múltiples scopes (entre 5 y 17).

Por un lado, los resultados muestran que bottom-up-$\wedge$ es sólo 17 % más rápido en promedio que bottom-up. No se observa en los experimentos una reducción significativa en la cantidad de invocaciones a SAT respecto de bottom-up. En cambio, bottom-up-f es 129 % más rápido que bottom-up. Esto indica que el aprovechamiento de cotas ajustadas por parte de bottom-up-f es relevante para su eficiencia. En vista de los resultados, bottom-up-f puede considerarse como la *mejor versión* derivada del enfoque bottom-up.

Algunas observaciones sobre las razones de la diferencia de rendimiento de los algoritmos en los distintos casos de estudio son apropiadas. Las cotas ajustadas para estructuras del tipo lista se computan muy rápidamente en todos los casos, debido a que estas poseen menos campos que las estructuras más complejas, y el espacio de estados que debe recorrer el SAT solver para retornar instancias válidas del heap es más pequeño. Las estructuras con invariantes fuertes –muy restrictivos–, como los binomial heaps, resultan en cotas ajustadas con pocos elementos (respecto del total de sus campos). Así, los procedimientos basados en bottom-up convergen más rápidamente, y muestran una mejor performance en tiempo en estos casos.

LList	**S5**	**S7**	**S10**	**S12**	**S15**	**S17**
bottom-up	00:08	00:14	00:29	00:42	01:02	01:30
bottom-up-∧	00:10	00:16	00:31	00:45	01:05	01:35
bottom-up-f	00:08	00:14	00:29	00:42	01:02	01:30
AList	**S5**	**S7**	**S10**	**S12**	**S15**	**S17**
bottom-up	00:09	00:17	00:35	00:46	01:38	06:28
bottom-up-∧	00:11	00:20	00:38	00:50	01:19	02:55
bottom-up-f	00:12	00:21	00:41	00:53	01:53	05:42
CList	**S5**	**S7**	**S10**	**S12**	**S15**	**S17**
bottom-up	01:42	02:14	02:41	03:13	05:50	21:59
bottom-up-∧	01:46	02:15	02:41	03:16	05:46	20:38
bottom-up-f	01:55	02:00	02:40	03:09	04:59	14:17
BSTree	**S5**	**S7**	**S10**	**S12**	**S15**	**S17**
bottom-up	00:33	01:31	05:35	11:45	47:26	TO
bottom-up∧	00:23	00:50	02:47	06:29	37:41	2:41:20
bottom-up-f	00:21	00:38	02:29	05:24	27:33	2:26:10
AVL	**S5**	**S7**	**S10**	**S12**	**S15**	**S17**
bottom-up	00:22	01:25	05:01	12:07	1:16:36	TO
bottom-up-∧	00:19	00:56	03:44	08:43	1:02:22	TO
bottom-up-f	00:22	00:51	02:42	05:25	30:25	TO
TSet	**S5**	**S7**	**S10**	**S12**	**S15**	**S17**
bottom-up	00:36	01:51	05:35	12:31	46:32	TO
bottom-up-∧	00:23	01:16	03:59	08:39	34:43	2:43:03
bottom-up-f	00:25	01:07	03:12	05:34	18:47	1:00:18
BHeap	**S5**	**S7**	**S10**	**S12**	**S15**	**S17**
bottom-up	00:43	01:54	05:55	12:12	41:00	2:07:54
bottom-up-∧	00:40	01:42	05:41	11:34	39:14	1:59:54
bottom-up-f	00:38	01:29	03:48	06:48	14:47	29:48

Cuadro 5.1: Tiempos de cómputo de cotas ajustadas para las distintas implementaciones de bottom-up.

5.1.2. Beneficios del cómputo incremental de cotas ajustadas

En esta sección se evalúan las ventajas de computar cotas ajustadas incrementalmente (sección 3.4). Recordemos que el objetivo del cómputo incremental de cotas ajustadas es mitigar el (elevado) costo de computar sucesivamente cotas ajustadas para scopes cada vez más grandes. En este experimento sólo se considera la versión incremental de bottom-up-f (llamada bottom-up-f-inc), debido a que es la versión derivada de bottom-up más eficiente (ver la sección anterior).

La tabla 5.2 presenta los resultados de los experimentos. Para cada caso de

LList	**S5**	**S7**	**S10**	**S12**	**S15**	**S17**
bottom-up-f	00:08	00:14	00:29	00:42	01:02	01:30
bottom-up-f-inc	00:08	00:05	00:10	00:09	00:15	00:15
bottom-up-f(sum)	00:08	00:22	00:51	01:33	02:35	04:05
bottom-up-f-inc(sum)	00:08	00:13	00:23	00:32	00:47	01:02
AList	**S5**	**S7**	**S10**	**S12**	**S15**	**S17**
bottom-up-f	00:12	00:21	00:41	00:53	01:53	05:42
bottom-up-f-inc	00:12	00:10	00:18	00:17	00:36	01:34
bottom-up-f(sum)	00:12	00:33	01:14	02:07	04:00	09:42
bottom-up-f-inc(sum)	00:12	00:22	00:40	00:57	01:33	03:07
CList	**S5**	**S7**	**S10**	**S12**	**S15**	**S17**
bottom-up-f	01:55	02:00	02:40	03:09	04:59	14:17
bottom-up-f-inc	01:55	00:37	00:56	00:57	02:10	04:22
bottom-up-f(sum)	01:55	03:55	06:35	09:44	14:43	29:00
bottom-up-f-inc(sum)	01:55	02:32	03:28	04:25	06:35	10:57
BSTree	**S5**	**S7**	**S10**	**S12**	**S15**	**S17**
bottom-up-f	00:21	00:38	02:29	05:24	27:33	2:26:10
bottom-up-f-inc	00:21	00:31	01:49	02:49	22:54	1:50:21
bottom-up-f(sum)	00:21	00:59	03:28	08:52	36:25	3:02:35
bottom-up-f-inc(sum)	00:21	00:52	02:41	05:30	28:24	2:18:45
AVL	**S5**	**S7**	**S10**	**S12**	**S15**	**S17**
bottom-up-f	00:22	00:51	02:42	05:25	30:25	TO
bottom-up-f-inc	00:22	00:35	01:49	03:12	20:31	2:29:30
bottom-up-f(sum)	00:22	01:13	03:55	09:20	39:45	NA
bottom-up-f-inc(sum)	00:22	00:57	02:46	05:58	26:29	2:55:59
TreeSet	**S5**	**S7**	**S10**	**S12**	**S15**	**S17**
bottom-up-f	00:25	01:07	03:12	05:34	18:47	1:00:18
bottom-up-f-inc	00:25	00:45	02:03	03:22	11:35	33:21
bottom-up-f(sum)	00:25	01:32	04:44	10:18	29:05	1:29:23
bottom-up-f-inc(sum)	00:25	01:10	03:13	06:35	18:10	51:31
BHeap	**S5**	**S7**	**S10**	**S12**	**S15**	**S17**
bottom-up-f	00:38	01:29	03:48	06:48	14:47	29:48
bottom-up-f-inc	00:38	00:46	02:27	02:53	07:41	10:51
bottom-up-f(sum)	00:38	02:07	05:55	12:43	27:30	57:18
bottom-up-f-inc(sum)	00:38	01:24	03:51	06:44	14:25	25:16

Cuadro 5.2: Tiempos de cómputo de cotas ajustadas para bottom-up-f y bottom-up-f-inc (incremental). bottom-up-f(sum) y bottom-up-f-inc(sum) muestran la suma de los tiempos de cómputo para todos los scopes menores o iguales a s (para cada s).

estudio, la fila correspondiente a bottom-up-f muestra el tiempo requerido por el algoritmo para computar cotas ajustadas desde cero, para cada uno de los múltiples

scopes considerados (entre 5 y 17). En cambio, para bottom-up-f-inc se muestran los tiempos de computar cotas ajustadas incrementalmente para el scope actual, es decir, asumiendo que se dispone de cotas para el scope previo. Por último, las filas nombradas bottom-up-f(sum) y bottom-up-f-inc(sum) muestran, por cada scope s, la suma de los tiempos requeridos por los algoritmos correspondientes para computar cotas ajustadas para todos los scopes menores o iguales a s.

En los resultados de la tabla 5.2 pueden verse dos tendencias. En primer lugar, bottom-up-f-inc es significativamente más rápido que bottom-up-f para computar cotas ajustadas cada vez más grandes, hasta un cierto scope (la diferencia empieza a hacerse visible a partir de scopes medianos). Segundo, bottom-up-f-inc permite amortizar totalmente el costo de computar las cotas ajustadas para todos los scopes más pequeños. En otras palabras, en el mismo tiempo que bottom-up-f computa cotas ajustadas para un sólo scope, bottom-up-f-inc produce también cotas ajustadas para todos los más pequeños.

5.1.3. bottom-up vs. TACO+

En esta sección se comparan la mejor versión de bottom-up (bottom-up-f) contra el módulo de cómputo de cotas ajustadas de TACO+ (al que llamaremos TACO+ para abreviar). El objetivo de la comparación es medir la ganancia en tiempo de usar bottom-up-f en lugar de TACO+ en un ambiente de ejecución secuencial. Como se mencionó en la sección 5, los tiempos de reloj de cómputo de cotas de TACO+ (TACO+(wall)) se tomaron de [25], en donde se usó un cluster de 16 computadoras con 4 núcleos cada una, con una velocidad de reloj de 2.66Ghz. En cambio, bottom-up-f se ejecutó en un sólo núcleo de 3.2Ghz. Entonces, para simular la ejecución secuencial de TACO+ (TACO+(seq)) se multiplicaron los tiempos de reloj (TACO+(wall)) por la cantidad total de núcleos (64) y luego por 0,831 para compensar por la diferencia entre los relojes de las diferentes computadoras. Nuevamente, el formato de los resultados es `hh:mm:ss` (horas, minutos y segundos).

La tabla 5.3 muestra los tiempos de ejecución de TACO+ paralelo (TACO+(wall)), secuencial (TACO+(seq)) y de bottom-up-f para nuestros casos de estudio (y múltiples scopes). Las filas denominadas `Speedup` muestran la ganancia en tiempo de usar bottom-up-f en lugar de TACO+(seq), para cada caso de estudio y scope. En este experimento, bottom-up-f es un orden de magnitud más rápido que TACO+(seq) en promedio: 15,2 veces para ser exactos. Es interesante destacar que TACO+ paralelo (TACO(wall)) es, en promedio, 3,5 veces más rápido que bottom-up-f, aunque la diferencia entre la cantidad de núcleos de procesamiento usados por los enfoques es de 63.

Es importante destacar una vez más que (el módulo de cómputo de cotas de) TA-

LList	**S5**	**S7**	**S10**	**S12**	**S15**	**S17**
TACO+(wall)	00:11	00:11	00:15	00:24	00:47	01:04
TACO+(seq)	09:45	09:45	13:17	21:16	41:39	56:43
bottom-up-f	00:08	00:14	00:29	00:42	01:02	01:30
`Speedup`	73.1	41.8	27.5	30.4	40.3	37.8
AList	**S5**	**S7**	**S10**	**S12**	**S15**	**S17**
TACO+(wall)	00:11	00:14	00:25	00:33	00:55	03:15
TACO+(seq)	09:45	12:24	22:00	29:15	48:45	2:52:50
bottom-up-f	00:12	00:21	00:41	00:53	01:53	05:42
`Speedup`	48.8	35.4	31.6	33.1	25.9	30.3
CList	**S5**	**S7**	**S10**	**S12**	**S15**	**S17**
TACO+(wall)	00:35	00:46	01:11	01:38	01:46	05:16
TACO+(seq)	31:01	40:46	1:02:56	1:26:52	1:33:57	4:40:06
bottom-up-f	01:55	02:00	02:40	03:09	04:59	14:17
`Speedup`	16.2	20.4	23.6	27.6	18.9	19.6
BSTree	**S5**	**S7**	**S10**	**S12**	**S15**	**S17**
TACO+(wall)	00:11	00:11	00:16	00:38	01:56	04:05
TACO+(seq)	09:45	09:45	14:10	33:40	1:42:49	3:37:10
bottom-up-f	00:21	00:38	02:29	05:24	27:33	2:26:10
`Speedup`	27.9	15.4	5.7	6.2	3.7	1.5
AVL	**S5**	**S7**	**S10**	**S12**	**S15**	**S17**
TACO+(wall)	00:17	00:32	01:55	03:46	10:36	47:25
TACO+(seq)	15:04	28:21	1:41:56	3:20:19	9:23:45	42:01:48
bottom-up-f	00:22	00:51	02:42	05:25	30:25	3:14:06
`Speedup`	41.1	33.4	37.8	37.0	18.5	13.0
TSet	**S5**	**S7**	**S10**	**S12**	**S15**	**S17**
TACO+(wall)	00:16	00:30	01:44	02:51	05:19	16:42
TACO+(seq)	14:10	26:35	1:32:11	2:31:34	4:42:45	14:48:10
bottom-up-f	00:25	01:07	03:12	05:34	18:47	1:00:18
`Speedup`	34.0	23.8	28.8	27.2	15.1	14.7
BHeap	**S5**	**S7**	**S10**	**S12**	**S15**	**S17**
TACO+(wall)	00:22	01:12	02:46	04:41	10:32	34:50
TACO+(seq)	19:30	1:03:49	2:27:08	4:09:04	9:20:12	30:52:34
bottom-up-f	00:38	01:29	03:48	06:48	14:47	29:48
`Speedup`	30.8	43.0	38.7	36.6	37.9	62.2

Cuadro 5.3: Ganancia en tiempo de usar bottom-up-f en lugar de TACO+(seq) para el cómputo secuencial de cotas ajustadas.

CO+ y bottom-up-f son intercambiables: para un caso de estudio particular ambos toman como entrada la misma especificación Alloy, derivada automáticamente a partir del invariante JML de la clase considerada (ver sección 3.5). Por esta razón, para una misma estructura de datos ambos enfoques producen cotas ajustadas idénticas.

La discusión anterior permite concluir que bottom-up-f es un buen candidato para reemplazar al módulo de cómputo de cotas de TACO+ en un entorno de ejecución secuencial.

Por último, es importante considerar una posible desventaja del enfoque bottom-up respecto de los enfoques *top-down* (TACO+). En caso de que el cómputo de cotas ajustadas exceda el tiempo límite dado, los enfoques top-down pueden retornar las cotas ajustadas computadas hasta el momento, aunque estas no sean tan precisas como sea posible. Por el contrario, bottom-up no puede retornar hasta no construir las cotas ajustadas más precisas, es decir, en caso de superar el límite de tiempo bottom-up no provee ninguna información útil. Se deja como trabajo futuro la evaluación de la utilidad de las cotas ajustadas menos precisas en diferentes tipos de análisis de código.

5.1.4. SLBD vs. bottom-up vs. TACO+

Hasta el momento, la evaluación experimental se centró en las técnicas de cómputo de cotas ajustadas, bottom-up y TACO+, que requieren de una especificación declarativa (JML) del invariante de la estructura de datos. En esta sección, se agrega a la comparación la técnica SLBD (capítulo 4), que se basa en una descripción de los invariantes de clases en términos de predicados inductivos de la lógica de separación. A diferencia de las técnicas anteriores –que derivan automáticamente su especificación Alloy de entrada a partir del invariante JML de la clase–, para cada caso de estudio se definió manualmente un predicado inductivo representando el invariante de la clase correspondiente, que se usó luego como entrada de SLBD.

La tabla 5.4 muestra los tiempos (en formato `hh:mm:ss`) que tardan los enfoques TACO+(seq), bottom-up-f y SLBD para computar cotas ajustadas, en un subconjunto significativo de nuestros casos de estudio (para varios scopes). Las filas etiquetadas `Speedup` muestra la ganancia en tiempo de usar SLBD en lugar de bottom-up-f. Los resultados del experimento son contundentes: computar cotas ajustadas a partir de predicados inductivos de la lógica de separación es varios órdenes de magnitud más rápido que hacerlo usando especificaciones declarativas. En promedio SLBD es 831 veces más rápido que bottom-up-f (que a su vez es más rápido que TACO+(seq)); si consideramos sólo el mayor scope (17), SLBD es 1677 veces más rápido que bottom-up-f. Atribuimos la ventaja de SLBD por sobre bottom-up-f a dos razones principales: (i) el operador $*$ de la lógica de separación que permite eliminar casos de aliasing y reducir la cantidad de invocaciones al procedimiento de decisión; (ii) la forma de los predicados inductivos admite la posibilidad de optimizar el cómputo de cotas mediante la técnica de memorización –evitando realizar una cantidad exponencial de llamadas recursivas para las estructuras de tipo árbol.

LList	**S5**	**S7**	**S10**	**S12**	**S15**	**S17**
TACO+(wall)	00:11	00:11	00:15	00:24	00:47	01:04
TACO+(seq)	09:45	09:45	13:17	21:16	41:39	56:43
bottom-up-f	00:08	00:14	00:29	00:42	01:02	01:30
SLBD	00:00.06	00:00.09	00:00.09	00:00.11	00:00.12	00:00.14
`Speedup`	133.3	155.6	322.2	381.8	516.7	642.9
BSTree	**S5**	**S7**	**S10**	**S12**	**S15**	**S17**
TACO+(wall)	00:11	00:11	00:16	00:38	01:56	04:05
TACO+(seq)	09:45	09:45	14:10	33:40	1:42:49	3:37:10
bottom-up-f	00:21	00:38	02:29	05:24	27:33	2:26:10
SLBD	00:00.26	00:00.20	00:00.50	00:00.63	00:01.03	00:01.38
`Speedup`	80.8	190.0	298.0	514.3	1604.9	6355.1
AVL	**S5**	**S7**	**S10**	**S12**	**S15**	**S17**
TACO+(wall)	00:17	00:32	01:55	03:46	10:36	47:25
TACO+(seq)	15:04	28:21	1:41:56	3:20:19	9:23:45	42:01:48
bottom-up-f	00:22	00:51	02:42	05:25	30:25	3:14:06
SLBD	00:00.12	00:00.19	00:00.40	00:00.56	00:00.94	00:01.38
`Speedup`	183.3	268.4	405.0	580.4	1941.5	8439.1
TSet	**S5**	**S7**	**S10**	**S12**	**S15**	**S17**
TACO+(wall)	00:16	00:30	01:44	02:51	05:19	16:42
TACO+(seq)	14:10	26:35	1:32:11	2:31:34	4:42:45	14:48:10
bottom-up-f	00:25	01:07	03:12	05:34	18:47	1:00:18
SLBD	00:00.66	00:01.15	00:02.69	00:04.35	00:08.26	00:11.48
`Speedup`	37.9	58.3	71.4	76.8	136.4	315.2

Cuadro 5.4: Tiempos de cómputo de cotas ajustadas a partir de especificaciones JML (TACO(seq) y bottom-up-f) y de predicados inductivos (SLBD).

Es importante recordar que SLBD produce cotas ajustadas DF, que son diferentes para las estructuras de tipo árbol a las cotas ajustadas BF generadas por bottom-upu o TACO+. En la sección siguiente se realiza una comparación de los beneficios que traen ambos tipos de cotas ajustadas para el análisis de código basado en SAT. Luego, en la sección 5.3 se comparan las ventajas de las cotas ajustadas producidas por los diferentes enfoques para la generación exhaustiva acotada (basada en SAT) de estructuras que satisfacen los invariantes de clase de nuestros casos de estudio.

5.2. Impacto de las cotas ajustadas en la verificación de código basado en SAT

Sabemos que SLBD produce cotas ajustadas DF, y que bottom-up y TACO+ generan cotas ajustadas BF. Como se mencionó en la sección 4.4, para mantener la corrección del verificador de código TACO, este debe instrumentarse con predicados de rotura de simetrías y cotas ajustadas compatibles. Así, para los experimentos de esta sección se tendrán en cuenta los siguientes enfoques (secuenciales) de verificación de programas:

- TACO: El verificador de código sin rotura de simetrías ni cotas ajustadas presentado en la sección 2.5.1.
- $\text{TACO}_{S_{BF}}$: TACO instrumentado con rotura de simetrías BF.
- $\text{TACO}_{S_{DF}}$: TACO instrumentado con rotura de simetrías DF.
- $\text{TACO}_{SB_{BF}}$: TACO instrumentado con rotura de simetrías y cotas ajustadas BF. Se asume que se dispone de las cotas ajustadas a utilizar durante el análisis.
- $\text{TACO}_{SB_{DF}}$: TACO instrumentado con rotura de simetrías y cotas ajustadas DF. Se asume que se dispone de las cotas ajustadas.
- bottom-up+TACO: El análisis de la sección 3.5, que computa cotas ajustadas vía bottom-up y luego las utiliza para instrumentar la verificación de código mediante TACO. Recordar que este enfoque instrumenta el análisis TACO con rotura de simetrías BF.
- SLBD+TACO: El análisis de la sección 4.5, que computa cotas ajustadas vía SLBD y luego las utiliza para instrumentar la verificación de código mediante TACO. Recordar que este enfoque instrumenta el análisis TACO con rotura de simetrías DF.

Notar que, los tiempos de ejecución de bottom-up+TACO (resp. SLBD+TACO) son iguales a la suma del tiempo que tarda bottom-up (SLBD) para computar cotas ajustadas y el tiempo de verificación de $\text{TACO}_{SB_{BF}}$ ($\text{TACO}_{S_{DF}}$). En otras palabras, los resultados para el análisis $\text{TACO}_{SB_{BF}}$ (resp. $\text{TACO}_{SB_{DF}}$) no incluyen el tiempo de cómputo de cotas ajustadas, que si se considera para bottom-up+TACO (SLBD+TACO). Consecuentemente, bottom-up+TACO y SLBD+TACO son los

primeros enfoques de verificación de código que explotan la idea de cotas ajustadas y se ejecutan por completo en una sola computadora –no dependen de cotas ajustadas computadas previamente.

Los experimentos que se realizarán en esta sección consisten en analizar la corrección (dentro de los límites impuestos por los scopes) de algunos métodos de nuestros casos de estudio, respecto de sus especificaciones JML. Todos los métodos elegidos satisfacen su especificación. Esta elección es intencional: estos métodos representan el peor caso para el análisis basado en SAT, ya que se debe examinar su espacio de estados completo antes de poder concluir el análisis. Cada método fue analizado para diversos scopes (entre 5 y 17). Al igual que en las secciones anteriores, se interrumpieron los procesos que no lograron finalizar antes del límite de tiempo (3 horas). En el código, los ciclos se expandieron hasta un máximo de 10 veces. Las invocaciones a métodos se reemplazaron por su código correspondiente, cambiando los parámetros formales por los actuales (técnica conocida como *inlining*).

Esta sección se organiza como sigue. En 5.2.1 se evalúan los beneficios de utilizar predicados de rotura de simetrías DF y BF en el análisis de código, comparando los enfoques TACO, $\text{TACO}_{S_{BF}}$ y $\text{TACO}_{S_{DF}}$. Luego, en 5.2.2 se mide el impacto de las cotas ajustadas (BF Y DF) en el análisis de código, evaluando los enfoques $\text{TACO}_{SB_{BF}}$ y $\text{TACO}_{SB_{DF}}$ respecto de $\text{TACO}_{S_{BF}}$ y $\text{TACO}_{S_{DF}}$. Por último, en 5.2.3 se incluye en la comparación anterior el tiempo de cómputo de cotas ajustadas, es decir, se miden los tiempos totales de ejecución de los análisis de código bottom-up+TACO y SLBD+TACO.

5.2.1. Rotura de simetrías en la verificación de código basado en SAT

En esta sección se evalúan los beneficios de usar los predicados de rotura de simetrías BF y DF en el análisis de código TACO. Para esto se comparan los enfoques TACO, $\text{TACO}_{S_{DF}}$ (TACO con rotura de simetrías DF) y $\text{TACO}_{S_{BF}}$ (TACO con rotura de simetrías BF) en la verificación de varios métodos de nuestros casos de estudio, para múltiples scopes. Cabe aclarar que en ninguno de los casos se usaron cotas ajustadas.

Los resultados de este experimento se exhiben en las tablas 5.5 (scopes de 7 a 12) y 5.6 (scopes de 13 a 17). Para abreviar, en las tablas TACO, $\text{TACO}_{S_{DF}}$ y $\text{TACO}_{S_{BF}}$ se denominan T, $\text{T}_{S_{DF}}$ y $\text{T}_{S_{BF}}$, respectivamente. Los análisis instrumentados con rotura de simetrías, $\text{TACO}_{S_{DF}}$ y $\text{TACO}_{S_{BF}}$, superan ampliamente en escalabilidad y velocidad de ejecución a TACO. Este último rara vez concluye exitosamente con scope 7, mientras que $\text{TACO}_{S_{DF}}$ y $\text{TACO}_{S_{BF}}$ posibilitan el análisis con scopes entre 12 y 15 en la mayoría de los casos, llegando a alcanzar el scope máximo considerado

			S7	S8	S9	S10	S11	S12
LList	Contains	T	TO	TO	TO	TO	TO	TO
		$T_{S_{XF}}$	0:00:03	0:00:03	0:00:04	0:00:04	0:00:06	0:00:06
	Insert	T	0:04:04	TO	TO	TO	TO	TO
		$T_{S_{XF}}$	0:00:02	0:00:03	0:00:03	0:00:04	0:00:04	0:00:05
	Remove	T	TO	TO	TO	TO	TO	TO
		$T_{S_{XF}}$	0:00:03	0:00:03	0:00:04	0:00:05	0:00:05	0:00:07
BSTree	Find	T	TO	TO	TO	TO	TO	TO
		$T_{S_{DF}}$	0:00:25	0:01:20	0:07:34	1:03:25	TO	TO
		$T_{S_{BF}}$	0:00:29	0:01:36	0:07:38	0:46:48	TO	TO
	Remove	T	TO	TO	TO	TO	TO	TO
		$T_{S_{DF}}$	0:00:10	0:00:40	0:02:54	0:11:08	0:42:38	2:58:12
		$T_{S_{BF}}$	0:00:14	0:00:56	0:04:15	0:13:56	0:55:05	2:55:22
	Insert	T	TO	TO	TO	TO	TO	TO
		$T_{S_{DF}}$	0:49:04	TO	TO	TO	TO	TO
		$T_{S_{BF}}$	0:40:50	TO	TO	TO	TO	TO
AVL	Find	T	TO	TO	TO	TO	TO	TO
		$T_{S_{DF}}$	0:00:07	0:00:17	0:00:37	0:01:09	0:03:12	0:05:50
		$T_{S_{BF}}$	0:00:07	0:00:18	0:00:47	0:01:21	0:02:51	0:06:28
	FindMax	T	TO	TO	TO	TO	TO	TO
		$T_{S_{DF}}$	0:00:02	0:00:03	0:00:05	0:00:08	0:00:18	0:00:30
		$T_{S_{BF}}$	0:00:01	0:00:02	0:00:03	0:00:07	0:00:15	0:00:19
	Insert	T	TO	TO	TO	TO	TO	TO
		$T_{S_{DF}}$	0:01:09	0:01:44	0:02:47	0:05:52	0:11:09	0:20:34
		$T_{S_{BF}}$	0:01:12	0:01:57	0:03:01	0:05:35	0:11:53	0:18:00
TreeSet	Find	T	TO	TO	TO	TO	TO	TO
		$T_{S_{DF}}$	0:00:10	0:00:28	0:00:55	0:02:09	0:06:03	0:07:19
		$T_{S_{BF}}$	0:00:12	0:00:22	0:00:50	0:01:19	0:03:43	0:06:54
	Insert	T	TO	TO	TO	TO	TO	TO
		$T_{S_{DF}}$	0:02:41	0:37:47	TO	TO	TO	TO
		$T_{S_{BF}}$	0:02:45	0:32:04	TO	TO	TO	TO

Cuadro 5.5: Comparación entre TACO (T), $\text{TACO}_{S_{DF}}$ ($T_{S_{DF}}$) y $\text{TACO}_{S_{BF}}$ ($T_{S_{BF}}$) en la verificación de código (scopes entre 7 y 12).

(17) en algunos casos. Comparando los análisis con diferentes tipos de rotura de simetrías, $\text{TACO}_{S_{BF}}$ y $\text{TACO}_{S_{DF}}$, es posible observar que en la mayoría de los casos ambos alcanzan los mismos scopes, y que, en general, $\text{TACO}_{S_{BF}}$ es ligeramente más rápido que $\text{TACO}_{S_{DF}}$: 20 % en promedio (considerando sólo los scopes que ambas técnicas son capaces de analizar exitosamente). Las excepciones son los métodos Find de AVL y Find de Treeset. En el primer caso, la diferencia en tiempo a favor de $\text{TACO}_{S_{BF}}$ comienza a ser significativa a partir del scope 14, lo que le permite a $\text{TACO}_{S_{BF}}$ alcanzar un scope más que $\text{TACO}_{S_{DF}}$ (16 en lugar de 15). En el segundo

			S13	S14	S15	S16	S17
LList	Contains	T	TO	TO	TO	TO	TO
		$T_{S_{XF}}$	0:00:07	0:00:09	0:00:09	0:00:11	0:00:15
	Insert	T	TO	TO	TO	TO	TO
		$T_{S_{XF}}$	0:00:06	0:00:07	0:00:08	0:00:09	0:00:16
	Remove	T	TO	TO	TO	TO	TO
		$T_{S_{XF}}$	0:00:07	0:00:10	0:00:10	0:00:13	0:00:27
BSTree	Find	T	TO	TO	TO	TO	TO
		$T_{S_{DF}}$	TO	TO	TO	TO	TO
		$T_{S_{BF}}$	TO	TO	TO	TO	TO
	Remove	T	TO	TO	TO	TO	TO
		$T_{S_{DF}}$	TO	TO	TO	TO	TO
		$T_{S_{BF}}$	TO	TO	TO	TO	TO
	Insert	T	TO	TO	TO	TO	TO
		$T_{S_{DF}}$	TO	TO	TO	TO	TO
		$T_{S_{BF}}$	TO	TO	TO	TO	TO
AVL	Find	T	TO	TO	TO	TO	TO
		$T_{S_{DF}}$	0:13:17	0:38:35	1:02:12	TO	TO
		$T_{S_{BF}}$	0:11:22	0:21:33	0:28:56	1:53:45	TO
	FindMax	T	TO	TO	TO	TO	TO
		$T_{S_{DF}}$	0:01:04	0:02:01	0:04:10	0:15:35	0:31:06
		$T_{S_{BF}}$	0:00:38	0:01:15	0:02:12	0:13:19	0:24:08
	Insert	T	TO	TO	TO	TO	TO
		$T_{S_{DF}}$	0:37:28	1:32:23	2:39:44	TO	TO
		$T_{S_{BF}}$	0:36:09	0:57:20	2:38:32	TO	TO
TreeSet	Find	T	TO	TO	TO	TO	TO
		$T_{S_{DF}}$	0:19:49	0:34:43	1:25:40	TO	TO
		$T_{S_{BF}}$	0:08:28	0:24:03	0:48:43	1:27:50	TO
	Insert	T	TO	TO	TO	TO	TO
		$T_{S_{DF}}$	TO	TO	TO	TO	TO
		$T_{S_{BF}}$	TO	TO	TO	TO	TO

Cuadro 5.6: Comparación entre TACO (T), $TACO_{S_{DF}}$ ($T_{S_{DF}}$) y $TACO_{S_{BF}}$ ($T_{S_{BF}}$) en la verificación de código (scopes entre 13 y 17).

caso, $TACO_{S_{BF}}$ supera significativamente a $TACO_{S_{DF}}$ desde el scope 13, y también es capaz de analizar un scope más que este último (16 contra 15). Notar que, como ambos enfoques de rotura de simetrías son equivalentes para el caso de listas (etiquetar nodos en un recorrido BF o DF del heap resulta en la misma canonización debido a que hay un sólo campo: `next`), sus resultados se muestran una sola vez en las tablas agrupados en las filas denominadas $T_{S_{XF}}$.

Para entender por qué los predicados de rotura de simetrías BF tienen un mayor impacto que los DF en TACO, comparemos sus respectivas definiciones (secciones

2.6.1 y 4.4.1). Notemos que para etiquetar dos nodos en orden BF sólo es necesario tener en cuenta el etiquetado de los padres directos de dichos nodos. En cambio, para etiquetar dos nodos en orden DF es necesario determinar si uno de ellos es alcanzable a partir del otro, en caso negativo comparar el identificador del segundo nodo con todos los nodos alcanzables a partir del primero, etc. Entonces, para definir predicados DF en Alloy se debe utilizar el operador de clausura reflexo transitiva `*` (ver sección 4.4.1), que no es necesario en el caso de los predicados BF. Por lo tanto, creemos que la diferencia en eficiencia del análisis usando los diferentes tipos de predicados se debe a que la codificación de los predicados DF en Alloy es más compleja.

Dado que la instrumentación de TACO con predicados de rotura de simetrías BF es completamente automática (sección 2.6.1), de los resultados anteriores se desprende que $\text{TACO}_{S_{BF}}$ es el análisis de código basado en SAT más eficiente de los evaluados hasta el momento. En las próximas secciones veremos como influyen las cotas ajustadas en el rendimiento de TACO.

Es importante explicar la razón de la ineficiencia de los análisis en el caso de estudio BSTree. Si bien los árboles binarios son estructuras más simples que los AVLs y árboles rojos y negros (TreeSet) (tienen menos campos, y sus métodos son signficativamente más sencillos), los experimentos muestran que sus métodos son más difíciles de analizar. Esto se debe a que, en el caso de estudio BSTree los datos almacenados en los árboles se representan como enteros Alloy (`int`), y en todos los restantes casos se representan con una signatura `Data` (definida por los autores de [25]). El tipo predefinido `int` de Alloy no permite elegir la cantidad exacta de elementos en la signatura, únicamente es posible seleccionar el "ancho de bits" (*bitwidth*). Por ejemplo, si se desean usar enteros entre 0 y 6 (scope 7) se debe elegir un ancho de bits de 4, lo que implica definir enteros entre -8 y 7, aún cuando los enteros negativos no se encuentran dentro del rango de enteros a analizar. Claramente, esto hace que la fórmula proposicional crezca innecesariamente, y, por consiguiente, que el análisis sea más lento. En cambio, `Data` se define de manera que sea posible representar exactamente el rango de enteros deseados en cada scope.

5.2.2. Cotas ajustadas en la verificación de código basada en SAT

En esta sección se evalúa el impacto de usar cotas ajustadas BF y DF en la verificación de código basada en SAT. Para esto, se instrumenta el análisis TACO con rotura de simetrías DF ($\text{TACO}_{S_{DF}}$) con cotas ajustadas DF (producidas por SLBD), y TACO con rotura de simetrías BF ($\text{T}_{S_{BF}}$) con cotas ajustadas BF (generadas por bottom-up o TACO+). De las instrumentaciones anteriores surgen los análisis

			S7	S8	S9	S10	S11	S12
LList	Contains	T	0:00:03	0:00:03	0:00:04	0:00:04	0:00:06	0:00:06
		$T_{SB_{XF}}$	0:00:02	0:00:03	0:00:03	0:00:03	0:00:04	0:00:04
	Insert	T	0:00:02	0:00:03	0:00:03	0:00:04	0:00:04	0:00:05
		$T_{SB_{XF}}$	0:00:02	0:00:03	0:00:03	0:00:03	0:00:03	0:00:04
	Remove	T	0:00:03	0:00:03	0:00:04	0:00:05	0:00:05	0:00:07
		$T_{SB_{XF}}$	0:00:02	0:00:03	0:00:03	0:00:04	0:00:05	0:00:06
BSTree	Find	$T_{S_{DF}}$	0:00:25	0:01:20	0:07:34	1:03:25	TO	TO
		$T_{S_{BF}}$	0:00:29	0:01:36	0:07:38	0:46:48	TO	TO
		$T_{SB_{DF}}$	0:00:16	0:01:03	0:05:09	0:24:20	1:46:43	2:48:21
		$T_{SB_{BF}}$	0:00:12	0:00:44	0:03:11	0:26:15	1:34:12	TO
	Remove	$T_{S_{DF}}$	0:00:10	0:00:40	0:02:54	0:11:08	0:42:38	2:58:12
		$T_{S_{BF}}$	0:00:14	0:00:56	0:04:15	0:13:56	0:55:05	2:55:22
		$T_{SB_{DF}}$	0:00:08	0:00:25	0:01:57	0:05:43	0:23:07	1:39:50
		$T_{SB_{BF}}$	0:00:09	0:00:30	0:02:43	0:08:35	0:26:46	2:06:14
	Insert	$T_{S_{DF}}$	0:49:04	TO	TO	TO	TO	TO
		$T_{S_{BF}}$	0:40:50	TO	TO	TO	TO	TO
		$T_{SB_{DF}}$	0:33:15	2:33:12	TO	TO	TO	TO
		$T_{SB_{BF}}$	0:29:22	1:47:57	TO	TO	TO	TO
AVL	Find	$T_{S_{DF}}$	0:00:07	0:00:17	0:00:37	0:01:09	0:03:12	0:05:50
		$T_{S_{BF}}$	0:00:07	0:00:18	0:00:47	0:01:21	0:02:51	0:06:28
		$T_{SB_{DF}}$	0:00:05	0:00:10	0:00:14	0:00:31	0:00:56	0:01:09
		$T_{SB_{BF}}$	0:00:05	0:00:07	0:00:09	0:00:16	0:00:22	0:00:36
	FindMax	$T_{S_{DF}}$	0:00:02	0:00:03	0:00:05	0:00:08	0:00:18	0:00:30
		$T_{S_{BF}}$	0:00:01	0:00:02	0:00:03	0:00:07	0:00:15	0:00:19
		$T_{SB_{DF}}$	0:00:01	0:00:01	0:00:02	0:00:02	0:00:03	0:00:04
		$T_{SB_{BF}}$	0:00:01	0:00:01	0:00:02	0:00:02	0:00:02	0:00:03
	Insert	$T_{S_{DF}}$	0:01:09	0:01:44	0:02:47	0:05:52	0:11:09	0:20:34
		$T_{S_{BF}}$	0:01:12	0:01:57	0:03:01	0:05:35	0:11:53	0:18:00
		$T_{SB_{DF}}$	0:01:05	0:01:38	0:02:31	0:04:25	0:07:39	0:14:12
		$T_{SB_{BF}}$	0:01:03	0:01:28	0:02:14	0:03:42	0:06:12	0:11:48
TreeSet	Find	$T_{S_{DF}}$	0:00:10	0:00:28	0:00:55	0:02:09	0:06:03	0:07:19
		$T_{S_{BF}}$	0:00:12	0:00:22	0:00:50	0:01:19	0:03:43	0:06:54
		$T_{SB_{DF}}$	0:00:07	0:00:19	0:00:27	0:00:48	0:01:44	0:03:47
		$T_{SB_{BF}}$	0:00:05	0:00:09	0:00:19	0:00:47	0:01:14	0:02:19
	Insert	$T_{S_{DF}}$	0:02:41	0:37:47	TO	TO	TO	TO
		$T_{S_{BF}}$	0:02:45	0:32:04	TO	TO	TO	TO
		$T_{SB_{DF}}$	0:02:34	0:22:38	TO	TO	TO	TO
		$T_{SB_{BF}}$	0:02:54	0:17:15	TO	TO	TO	TO

Cuadro 5.7: Tiempos de verificación de código para $TACO_{S_{DF}}$ ($T_{S_{DF}}$), $TACO_{S_{BF}}$ ($T_{S_{BF}}$), $TACO_{SB_{DF}}$ ($T_{SB_{DF}}$) y $TACO_{SB_{BF}}$ ($T_{SB_{BF}}$) (scopes entre 7 y 12).

$TACO_{SB_{DF}}$ y $TACO_{SB_{BF}}$, respectivamente. Estos análisis se usan, en conjunción con $TACO_{S_{DF}}$ y $TACO_{S_{BF}}$ de la sección anterior, en la verificación de la corrección

			S13	S14	S15	S16	S17
LList	Contains	T	0:00:07	0:00:09	0:00:09	0:00:11	0:00:15
		$T_{SB_{XF}}$	0:00:04	0:00:04	0:00:05	0:00:07	0:00:07
	Insert	T	0:00:06	0:00:07	0:00:08	0:00:09	0:00:16
		$T_{SB_{XF}}$	0:00:04	0:00:05	0:00:05	0:00:07	0:00:09
	Remove	T	0:00:07	0:00:10	0:00:10	0:00:13	0:00:27
		$T_{SB_{XF}}$	0:00:06	0:00:07	0:00:07	0:00:11	0:00:13
BSTree	Find	$T_{S_{DF}}$	TO	TO	TO	TO	TO
		$T_{S_{BF}}$	TO	TO	TO	TO	TO
		$T_{SB_{DF}}$	TO	TO	TO	TO	TO
		$T_{SB_{BF}}$	TO	TO	TO	TO	TO
	Remove	$T_{S_{DF}}$	TO	TO	TO	TO	TO
		$T_{S_{BF}}$	TO	TO	TO	TO	TO
		$T_{SB_{DF}}$	TO	TO	TO	TO	TO
		$T_{SB_{BF}}$	TO	TO	TO	TO	TO
	Insert	$T_{S_{DF}}$	TO	TO	TO	TO	TO
		$T_{S_{BF}}$	TO	TO	TO	TO	TO
		$T_{SB_{DF}}$	TO	TO	TO	TO	TO
		$T_{SB_{BF}}$	TO	TO	TO	TO	TO
AVL	Find	$T_{S_{DF}}$	0:13:17	0:38:35	1:02:12	TO	TO
		$T_{S_{BF}}$	0:11:22	0:21:33	0:28:56	1:53:45	TO
		$T_{SB_{DF}}$	0:01:49	0:04:57	0:15:43	0:16:03	0:35:27
		$T_{SB_{BF}}$	0:01:22	0:01:54	0:02:44	0:05:24	0:12:36
	FindMax	$T_{S_{DF}}$	0:01:04	0:02:01	0:04:10	0:15:35	0:31:06
		$T_{S_{BF}}$	0:00:38	0:01:15	0:02:12	0:13:19	0:24:08
		$T_{SB_{DF}}$	0:00:05	0:00:06	0:00:09	0:00:19	0:00:31
		$T_{SB_{BF}}$	0:00:03	0:00:04	0:00:05	0:00:06	0:00:08
	Insert	$T_{S_{DF}}$	0:37:28	1:32:23	2:39:44	TO	TO
		$T_{S_{BF}}$	0:36:09	0:57:20	2:38:32	TO	TO
		$T_{SB_{DF}}$	0:23:27	0:44:24	1:12:32	TO	TO
		$T_{SB_{BF}}$	0:22:22	0:36:42	1:15:15	TO	TO
TreeSet	Find	$T_{S_{DF}}$	0:19:49	0:34:43	1:25:40	TO	TO
		$T_{S_{BF}}$	0:08:28	0:24:03	0:48:43	1:27:50	TO
		$T_{SB_{DF}}$	0:06:42	0:16:13	0:34:04	1:20:35	1:31:50
		$T_{SB_{BF}}$	0:05:26	0:06:32	0:16:45	0:35:39	1:15:05
	Insert	$T_{S_{DF}}$	TO	TO	TO	TO	TO
		$T_{S_{BF}}$	TO	TO	TO	TO	TO
		$T_{SB_{DF}}$	TO	TO	TO	TO	TO
		$T_{SB_{BF}}$	TO	TO	TO	TO	TO

Cuadro 5.8: Tiempos de verificación de código para $TACO_{S_{DF}}$ ($T_{S_{DF}}$), $TACO_{S_{BF}}$ ($T_{S_{BF}}$), $TACO_{SB_{DF}}$ ($T_{SB_{DF}}$) y $TACO_{SB_{BF}}$ ($T_{SB_{BF}}$) (scopes entre 13 y 17).

de algunos métodos de nuestros casos de estudio. El objetivo de este experimento es medir los beneficios proporcionados por los diferentes tipos de cotas ajustadas al

análisis de código, asumiendo que se dispone de cotas ajustadas. Es decir, en este experimento no se tienen en cuenta los tiempos de cómputo de las cotas.

Los tiempos de ejecución de los análisis $\text{TACO}_{S_{DF}}$, $\text{TACO}_{S_{BF}}$, $\text{TACO}_{SB_{DF}}$ y $\text{TACO}_{SB_{BF}}$, para varios métodos de nuestros casos de estudio, se muestran en las tablas 5.7 (para scopes entre 7 y 12) y 5.8 (para scopes entre 13 y 17). Los resultados muestran que los análisis que explotan cotas ajustadas, $\text{TACO}_{SB_{DF}}$ y $\text{TACO}_{SB_{BF}}$, son 120 % y 122 % más rápidos en promedio que $\text{TACO}_{S_{DF}}$ y que $\text{TACO}_{S_{BF}}$, respectivamente (teniendo en cuenta sólo los casos en que las técnicas comparadas terminan correctamente). Es decir, los tiempos de verificación se reducen, en promedio, 54,7 % y 55,1 % usando cotas ajustadas DF y BF, respectivamente. Además, el aprovechamiento de ambos tipos de cotas ajustadas permite extender los límites del análisis en 1 scope para los métodos Find e Insert de BSTree, Find de AVL y Find de TreeSet (4 de 11 analizados, notar que este número podría ser mayor ya que en algunos casos no se superan los límites de los análisis que no explotan las cotas ajustadas). Una comparación entre los análisis que usan cotas ajustadas, $\text{TACO}_{SB_{DF}}$ y $\text{TACO}_{SB_{BF}}$, indica que el segundo es 24 % más rápido que el primero en promedio. Sin embargo, en la sección anterior se concluyó que $\text{TACO}_{S_{BF}}$ es un 20 % más rápido que $\text{TACO}_{S_{DF}}$. Por lo tanto, estos números indican que las cotas ajustadas DF tienen un impacto similar a las cotas ajustadas BF en el análisis de código basado en SAT. Por último, de los resultados se desprende que el aprovechamiento de cotas ajustadas DF le permite a $\text{TACO}_{SB_{DF}}$ ser un 87 % más rápido en promedio que $\text{TACO}_{S_{BF}}$ (reducir los tiempos de verificación en un 46,7 %) –a pesar de que $\text{TACO}_{S_{BF}}$ es 20 % más rápido que $\text{TACO}_{S_{DF}}$, sobre el cual se basa $\text{TACO}_{SB_{DF}}$. Esto es importante porque las cotas ajustadas DF pueden computarse en unos pocos segundos, y varios órdenes de magnitud más rápido que las BF (sección 5.1.4). En la sección siguiente se comparan los enfoques de análisis de código de esta sección incluyendo los tiempos requeridos para computar cotas ajustadas, con el objetivo de encontrar el análisis de código más eficiente de los evaluados en este trabajo.

Las razones de la (mínima) diferencia de performance de TACO usando cotas ajustadas DF y BF son todavía desconocidas. Por ejemplo, para nuestras estructuras de árboles binarios, la cantidad de valores en las cotas ajustadas DF para el campo left es usualmente menor que en las cotas BF, pero lo contrario ocurre si se observa el campo right. De esta manera, si se incluyen todos los campos, lo más frecuente es que las cotas DF y BF posean una cantidad similar de valores (en algunos casos particulares las cotas DF son ligeramente más compactas, y en otros lo son las cotas BF), lo que en teoría debería resultar en tiempos de análisis similares para los diferentes tipos de cotas. Sería deseable realizar experimentos adicionales en un futuro, que permitan explicar mejor las diferencias entre las cotas BF y DF.

Para concluir, es importante hacer algunas observaciones sobre los resultados presentados en estas últimas dos secciones. Primero, que los predicados de rotura de simetrías aumentan la eficiencia y escalabilidad del análisis basado en SAT sustancialmente, siendo estos –y no las cotas ajustadas– la principal razón de la mayor escalabilidad del análisis basado en SAT TACO de [25], respecto de enfoques relacionados más antiguos (JForge [17]). Segundo, si bien las cotas ajustadas reducen los tiempos de ejecución significativamente, estas no son tan importantes para TACO como los predicados de rotura de simetrías. Estos resultados son inesperados, ya que no se mencionan explícitamente en los artículos que introducen la idea de cotas ajustadas [25, 27]. De cualquier manera, esto no implica que los resultados anteriores sean el techo de lo que puede lograrse utilizando cotas ajustadas. De hecho, TACO es sólo el primer enfoque de análisis de código capaz de explotar estas cotas, y su uso durante el análisis es directo y poco óptimo: sólo se usan para eliminar variables proposicionales en el estado inicial del programa, pero no tienen incidencia directa sobre los (muchos) estados restantes (aunque indirectamente el SAT solver pueda resolver cláusulas que codifican estos estados debido a las variables proposicionales eliminadas en el estado inicial). Sería deseable definir métodos de propagación de las cotas ajustadas a los restantes estados que permitan obtener una mayor eficiencia en el análisis ([40] implementa esta idea, pero sólo logra una mejora moderada del rendimiento de TACO+). En la sección 5.4 se discuten técnicas novedosas que introducen formas más sofisticadas de aprovechar las cotas ajustadas (como las computadas por nuestros enfoques), y así obtener mayores beneficios durante el análisis de código. Estas técnicas se desarrollaron en paralelo a los enfoques de cómputo de cotas ajustadas de este trabajo, razón por la cual no se incluyeron en esta comparación experimental.

5.2.3. Cómputo e instrumentación de la verificación de código basada en SAT con cotas ajustadas

En la sección anterior se concluyó que las cotas ajustadas DF y BF permiten, para los métodos de nuestros casos de estudio, reducir los tiempos de análisis un 54,7 % y 55,1 %, tomando como base a los análisis instrumentados con rotura de simetrías BF y DF, respectivamente. Además, en varios casos (4 de 11) las cotas ajustadas (DF y BF) aumentan la escalabilidad del análisis en un scope. Además, se mencionó que el análisis con cotas ajustadas y predicados de rotura de simetría BF es 24 % más rápido que su contraparte con cotas ajustadas y rotura de simetrías DF. Sin embargo, estos resultados no tienen en cuenta el tiempo que se necesita para computar las cotas ajustadas.

En esta sección se comparan los enfoques bottom-up+TACO (sección 3.5) y

			S7	S8	S9	S10	S11	S12
LList	Contains	$T_{S_{BF}}$	0:00:03	0:00:03	0:00:04	0:00:04	0:00:06	0:00:06
		SLBD+T	0:00:02	0:00:03	0:00:03	0:00:03	0:00:04	0:00:04
		BU+T	0:00:06	0:00:09	0:00:11	0:00:12	0:00:15	0:00:17
	Insert	$T_{S_{BF}}$	0:00:02	0:00:03	0:00:03	0:00:04	0:00:04	0:00:05
		SLBD+T	0:00:02	0:00:03	0:00:03	0:00:03	0:00:03	0:00:04
		BU+T	0:00:06	0:00:09	0:00:11	0:00:12	0:00:14	0:00:17
	Remove	$T_{S_{BF}}$	0:00:03	0:00:03	0:00:04	0:00:05	0:00:05	0:00:07
		SLBD+T	0:00:02	0:00:03	0:00:03	0:00:04	0:00:05	0:00:06
		BU+T	0:00:06	0:00:09	0:00:11	0:00:13	0:00:16	0:00:19
BSTree	Find	$T_{S_{BF}}$	0:00:29	0:01:36	0:07:38	0:46:48	TO	TO
		SLBD+T	0:00:16	0:01:03	0:05:09	0:24:20	1:46:43	2:48:21
		BU+T	0:00:24	0:01:08	0:03:44	0:27:04	1:35:19	TO
	Remove	$T_{S_{BF}}$	0:00:14	0:00:56	0:04:15	0:13:56	0:55:05	2:55:22
		SLBD+T	0:00:08	0:00:25	0:01:57	0:05:43	0:23:07	1:39:50
		BU+T	0:00:21	0:00:54	0:03:16	0:09:24	0:27:53	2:08:00
	Insert	$T_{S_{BF}}$	0:40:50	TO	TO	TO	TO	TO
		SLBD+T	0:33:15	2:33:12	TO	TO	TO	TO
		BU+T	0:29:34	1:48:21	TO	TO	TO	TO
AVL	Find	$T_{S_{BF}}$	0:00:07	0:00:18	0:00:47	0:01:21	0:02:51	0:06:28
		SLBD+T	0:00:05	0:00:10	0:00:14	0:00:31	0:00:56	0:01:09
		BU+T	0:00:21	0:00:37	0:00:50	0:01:09	0:01:27	0:02:23
	FindMax	$T_{S_{BF}}$	0:00:01	0:00:02	0:00:03	0:00:07	0:00:15	0:00:19
		SLBD+T	0:00:01	0:00:01	0:00:02	0:00:02	0:00:03	0:00:04
		BU+T	0:00:17	0:00:31	0:00:43	0:00:55	0:01:07	0:01:50
	Insert	$T_{S_{BF}}$	0:01:12	0:01:57	0:03:01	0:05:35	0:11:53	0:18:00
		SLBD+T	0:01:05	0:01:38	0:02:31	0:04:25	0:07:39	0:14:12
		BU+T	0:01:19	0:01:58	0:02:55	0:04:35	0:07:17	0:13:35
TreeSet	Find	$T_{S_{BF}}$	0:00:12	0:00:22	0:00:50	0:01:19	0:03:43	0:06:54
		SLBD+T	0:00:07	0:00:19	0:00:28	0:00:49	0:01:45	0:03:49
		BU+T	0:00:38	0:00:59	0:01:26	0:02:23	0:03:23	0:05:06
	Insert	$T_{S_{BF}}$	0:02:45	0:32:04	TO	TO	TO	TO
		SLBD+T	0:02:34	0:22:38	TO	TO	TO	TO
		BU+T	0:03:27	0:18:05	TO	TO	TO	TO

Cuadro 5.9: Tiempos totales de verificación para $TACO_{S_{DF}}$ ($T_{S_{DF}}$), SLBD+TACO (SLBD+T) y bottom-up+TACO (BU+T), incluyendo tiempos de cómputo de cotas ajustadas para los últimos dos enfoques (scopes entre 7 y 12).

SLBD+TACO (sección 4.5) –que computan cotas ajustadas y luego las usan para instrumentar el análisis de código TACO– contra el análisis de código sin cotas ajustadas más eficiente, $TACO_{S_{BF}}$, en la verificación acotada de los métodos de nuestros casos de estudio (los mismos que en dos secciones previas). Para los enfoques que usan cotas ajustadas se considera la suma de los tiempos de cómputo de cotas ajus-

			S13	S14	S15	S16	S17
LList	Contains	$T_{S_{BF}}$	0:00:07	0:00:09	0:00:09	0:00:11	0:00:15
		SLBD+T	0:00:04	0:00:04	0:00:05	0:00:07	0:00:07
		BU+T	0:00:19	0:00:21	0:00:25	0:00:34	0:00:36
	Insert	$T_{S_{BF}}$	0:00:06	0:00:07	0:00:08	0:00:09	0:00:16
		SLBD+T	0:00:04	0:00:05	0:00:05	0:00:07	0:00:09
		BU+T	0:00:19	0:00:22	0:00:25	0:00:34	0:00:38
	Remove	$T_{S_{BF}}$	0:00:07	0:00:10	0:00:10	0:00:13	0:00:27
		SLBD+T	0:00:06	0:00:07	0:00:07	0:00:11	0:00:13
		BU+T	0:00:21	0:00:24	0:00:27	0:00:38	0:00:42
BSTree	Find	$T_{S_{BF}}$	TO	TO	TO	TO	TO
		SLBD+T	TO	TO	TO	TO	TO
		BU+T	TO	TO	TO	TO	TO
	Remove	$T_{S_{BF}}$	TO	TO	TO	TO	TO
		SLBD+T	TO	TO	TO	TO	TO
		BU+T	TO	TO	TO	TO	TO
	Insert	$T_{S_{BF}}$	TO	TO	TO	TO	TO
		SLBD+T	TO	TO	TO	TO	TO
		BU+T	TO	TO	TO	TO	TO
AVL	Find	$T_{S_{BF}}$	0:11:22	0:21:33	0:28:56	1:53:45	TO
		SLBD+T	0:01:49	0:04:57	0:15:43	0:16:03	0:35:27
		BU+T	0:04:21	0:07:08	0:12:46	0:45:14	1:16:39
	FindMax	$T_{S_{BF}}$	0:00:38	0:01:15	0:02:12	0:13:19	0:24:08
		SLBD+T	0:00:05	0:00:06	0:00:09	0:00:19	0:00:31
		BU+T	0:03:02	0:05:18	0:10:07	0:39:56	1:04:11
	Insert	$T_{S_{BF}}$	0:36:09	0:57:20	2:38:32	TO	TO
		SLBD+T	0:23:27	0:44:24	1:12:32	TO	TO
		BU+T	0:25:21	0:41:56	1:25:17	TO	TO
TreeSet	Find	$T_{S_{BF}}$	0:08:28	0:24:03	0:48:43	1:27:50	TO
		SLBD+T	0:06:44	0:16:16	0:34:08	1:20:39	1:31:55
		BU+T	0:09:54	0:13:19	0:26:08	0:54:41	1:45:14
	Insert	$T_{S_{BF}}$	TO	TO	TO	TO	TO
		SLBD+T	TO	TO	TO	TO	TO
		BU+T	TO	TO	TO	TO	TO

Cuadro 5.10: Tiempos totales de verificación para $TACO_{S_{DF}}$ ($T_{S_{DF}}$), SLBD+TACO (SLBD+T) y bottom-up+TACO (BU+T), incluyendo tiempos de cómputo de cotas ajustadas para los últimos dos enfoques (scopes entre 13 y 17).

tadas divido por la cantidad de veces que se reusan las cotas, con el tiempo de análisis (instrumentado con cotas ajustadas) de cada método. Por ejemplo, computar cotas vía bottom-up para el caso de estudio TreeSet con scope 17 tarda 1 hora y 18 segundos, pero como se analizan dos métodos de TreeSet usando las mismas cotas ajustadas el tiempo de cómputo de cotas se amortiza entre los métodos, agregando

30 minutos y 9 segundos al análisis de cada uno. Notar que, para cada caso de estudio y scope particular, el tiempo total de análisis de bottom-up+TACO es equivalente al tiempo de cómputo de cotas (amortizado) vía bottom-up más los tiempos de la verificación acotada $\text{TACO}_{SB_{BF}}$ de la sección anterior (respectivamente, SLBD+TACO con $\text{TACO}_{SB_{DF}}$).

Los resultados del experimento se exhiben en las tablas 5.9 (scopes entre 7 y 12) y 5.10 (scopes entre 13 y 17). Estos muestran que, por un lado, tener en cuenta los tiempos de cómputo de cotas ajustadas, vía bottom-up o SLBD, no reduce la escalabilidad de los análisis de código. En otras palabras, SLBD+TACO y bottom-up+TACO tienen idéntica escalabilidad que $\text{TACO}_{SB_{DF}}$ y $\text{TACO}_{SB_{BF}}$ de la sección anterior, respectivamente. De esta manera, tanto SLBD+TACO como bottom-up+TACO presentan una escalabilidad levemente mejor que $\text{TACO}_{S_{BF}}$ (1 scope más en 4 de 11 casos). Por otro lado, con respecto a la eficiencia de los diferentes análisis podemos observar que SLBD+TACO es tan eficiente como $\text{TACO}_{SB_{DF}}$ –debido a la gran velocidad de cómputo de cotas de SLBD el tiempo de cómputo de cotas sólo suma algunos segundos al total. Así, SLBD+TACO es un análisis de código 87 % más rápido que $\text{TACO}_{S_{BF}}$ en promedio, lo que corresponde a una reducción en los tiempos de ejecución de $\text{TACO}_{S_{BF}}$ de 46,7 %. En cambio, bottom-up+TACO es sólo 41 % más rápido que $\text{TACO}_{S_{BF}}$ en promedio (29,5 % de reducción en tiempos de análisis), aún cuando $\text{TACO}_{SB_{BF}}$ es 122 % más rápido que $\text{TACO}_{S_{BF}}$ (promedio). Esto implica que el tiempo que tarda bottom-up para cómputar cotas ajustadas BF contribuye significativamente a los tiempos totales de análisis de bottom-up+TACO, y que en el caso de SLBD+TACO el tiempo de cómputo de cotas es despreciable. De todas maneras, el peso relativo del tiempo de cómputo de cotas ajustadas en el análisis de código disminuye si se analizan más métodos de cada clase, lo que es factible si se desea verificar la corrección de una clase completa. En tal caso, bottom-up+TACO podría llegar a ser preferible a SLBD+TACO (87 % es el porcentaje máximo de ganancia de $\text{TACO}_{SB_{DF}}$ contra $\text{TACO}_{S_{BF}}$, pero para $\text{TACO}_{SB_{BF}}$ es de hasta 122 %). Por otra parte, para el caso de estudio Linked List y para los el análisis de árboles con scopes pequeños $\text{TACO}_{S_{BF}}$ supera a bottom-up+TACO. Es decir, cuando el espacio de estados a verificar es simple, el impacto insignificante de las cotas ajustadas en el análisis no logra amortizar el costo de cómputo de cotas.

De los resultados podemos concluir que ambos enfoques que explotan la idea de cotas ajustadas son significativamente más eficientes y ligeramente más escalables que el análisis que no utilizan estas cotas. Sin embargo, bottom-up+TACO puede no ser recomendable si se desea analizar un sólo método de una clase, o scopes pequeños. En cambio, la única desventaja de SLBD+TACO respecto de los análisis que no usan cotas ajustadas es la necesidad de contar con un predicado inductivo de la lógica de separación como invariante de clase. En dicho caso, el uso de SLBD+TACO

es preferible a todos los enfoques considerados en esta evaluación experimental. La excepción sería el caso en el que la cantidad de métodos a analizar para una misma clase sea grande, de modo que permita amortizar el costo de computar cotas ajustadas BF, aunque esto no puede saberse con anterioridad a la ejecución del análisis ni del cómputo de las cotas.

5.3. Impacto de las cotas ajustadas en la generación exhaustiva de estructuras basada en SAT

BSTree	**S7**	**S8**	**S9**	**S10**	**S11**	**S12**	**S13**	**S14**
$\text{GEN}_{S_{DF}}$	0:00:07	0:00:31	0:03:19	0:23:29	TO	TO	TO	TO
$\text{GEN}_{S_{BF}}$	0:00:08	0:00:36	0:03:50	0:24:35	TO	TO	TO	TO
$\text{GEN}_{SB_{DF}}$	0:00:06	0:00:20	0:01:59	0:14:57	2:28:34	TO	TO	TO
$\text{GEN}_{SB_{BF}}$	0:00:06	0:00:23	0:02:12	0:17:03	TO	TO	TO	TO
TreeSet	**S7**	**S8**	**S9**	**S10**	**S11**	**S12**	**S13**	**S14**
$\text{GEN}_{S_{DF}}$	0:00:04	0:00:13	0:00:47	0:02:53	0:10:30	0:37:24	2:24:48	TO
$\text{GEN}_{S_{BF}}$	0:00:04	0:00:14	0:00:48	0:02:55	0:10:24	0:33:28	2:02:15	TO
$\text{GEN}_{SB_{DF}}$	0:00:05	0:00:13	0:00:29	0:01:38	0:04:19	0:21:41	1:00:41	TO
$\text{GEN}_{SB_{BF}}$	0:00:05	0:00:07	0:00:20	0:01:12	0:04:13	0:14:47	0:54:01	TO
AVL	**S7**	**S8**	**S9**	**S10**	**S11**	**S12**	**S13**	**S14**
$\text{GEN}_{S_{DF}}$	0:00:04	0:00:10	0:00:26	0:01:28	0:04:35	0:15:48	0:48:34	2:36:10
$\text{GEN}_{S_{BF}}$	0:00:03	0:00:10	0:00:30	0:01:28	0:04:44	0:15:57	0:49:21	2:30:56
$\text{GEN}_{SB_{DF}}$	0:00:03	0:00:05	0:00:09	0:00:22	0:01:17	0:04:10	0:13:57	0:49:50
$\text{GEN}_{SB_{BF}}$	0:00:03	0:00:05	0:00:08	0:00:19	0:00:53	0:03:36	0:12:08	0:40:23

Cuadro 5.11: Tiempos de generación exhaustiva de estructuras para $\text{GEN}_{S_{DF}}$, $\text{GEN}_{S_{BF}}$, $\text{GEN}_{SB_{DF}}$, y $\text{GEN}_{SB_{BF}}$ (scopes entre 7 y 14).

Como experimento adicional para evaluar la utilidad de los distintos tipos de cotas ajustadas se considera la generación exhaustiva de estructuras basada en SAT, usando los invariantes de clase, para algunos de nuestros casos de estudio. El procedimiento de generación de estructuras usado es sencillo: para cada caso de estudio se traduce automáticamente el invariante de clase JML a una especificación Alloy equivalente, que puede ser instrumentada con los distintos tipos de predicados de rotura de simetrías y cotas ajustadas (la especificación se instrumenta con cotas ajustadas de la misma manera que en la sección 2.8). La especificación Alloy se traduce en una fórmula proposicional, que luego se pasa como entrada a un SAT solver para generar todas las instancias posibles de la especificación Alloy, es decir, todas las estructuras

que satisfacen el invariante JML. Como en todo análisis basado en SAT, para que el proceso anterior pueda llevarse a cabo el usuario debe proveer scopes para cada clase. Es importante mencionar que el objetivo de esta evaluación experimental es añadir evidencia de que las cotas ajustadas DF proporcionan beneficios similares a las BF en otro tipo diferente de análisis basado en SAT. La herramienta más eficiente para la generación exhaustiva acotada de estructuras es Korat [9]. El rendimiento de nuestros enfoques (básicos) de generación de estructuras que usa cotas ajustadas es inferior al de Korat.

De esta manera, se definen los siguientes enfoques de generación automática de estructuras:

- $\text{GEN}_{S_{DF}}$: Generación a partir de la especificación Alloy instrumentada con predicados de rotura de simetrías DF.
- $\text{GEN}_{S_{BF}}$: Generación a partir de la especificación Alloy instrumentada con predicados de rotura de simetrías BF.
- $\text{GEN}_{SB_{DF}}$: Generación a partir de la especificación Alloy instrumentada con predicados de rotura de simetrías y cotas ajustadas DF.
- $\text{GEN}_{SB_{BF}}$: Generación a partir de la especificación Alloy instrumentada con predicados de rotura de simetrías y cotas ajustadas BF.

Recordar que las cotas ajustadas BF se computan usando el enfoque bottom-up, y las cotas ajustadas DF son generables mediante SLBD.

Los tiempos de generación exhaustiva de estructuras para los casos de estudio BSTree, AVL y TreeSet para los enfoques anteriores se resumen en la tabla 5.11. En esta tabla se asume que se dispone de las cotas ajustadas para cada caso de estudio (no se contabiliza el tiempo de cómputo de las mismas). Los resultados de este experimento muestran que las diferencias entre los enfoques con diferentes tipos de rotura de simetrías, $\text{GEN}_{S_{DF}}$ y $\text{GEN}_{S_{BF}}$, son levemente menores que para el análisis de código: $\text{GEN}_{S_{BF}}$ aventaja a $\text{GEN}_{S_{DF}}$ en sólo 6 % ($\text{TACO}_{S_{BF}}$ es 20 % más rápido que $\text{TACO}_{S_{DF}}$). Por otra parte, el uso de cotas ajustadas DF y BF permite acelerar la velocidad del análisis en 155 % y 177 %, en promedio, tomando como base a los enfoques con rotura de simetrías DF y BF, respectivamente. Es decir, el impacto de las cotas ajustadas DF y BF es ligeramente mayor que en el análisis de código, en donde proporcionaban una mejora promedio de 120 % y 122 %, respectivamente. Atribuímos este hecho a que las cotas ajustadas se usan sólo en el estado inicial del programa en nuestro enfoque de análisis de código, y no influyen directamente sobre los estados restantes del programa. La diferencia entre los análisis que usan cotas ajustadas es de un 15 % en favor de $\text{GEN}_{SB_{BF}}$, aunque un 6 % de dicha diferencia se

explica por la mayor eficiencia de la rotura de simetrías BF. En vista de la discusión anterior, es posible concluir que las cotas ajustadas DF tienen, también, un impacto similar a las BF en este tipo de análisis automático.

BSTree	**S7**	**S8**	**S9**	**S10**	**S11**	**S12**	**S13**	**S14**
$\text{GEN}_{S_{BF}}$	0:00:08	0:00:36	0:03:50	0:24:35	TO	TO	TO	TO
$\text{GEN}_{SB_{DF}}$	0:00:06	0:00:20	0:01:59	0:14:57	2:28:34	TO	TO	TO
$\text{GEN}_{SB_{BF}}$	0:00:44	0:01:38	0:03:52	0:19:32	TO	TO	TO	TO
$\text{GEN}_{SB_{BF}}$(a)	0:00:15	0:00:41	0:02:37	0:17:40	TO	TO	TO	TO
TreeSet	**S7**	**S8**	**S9**	**S10**	**S11**	**S12**	**S13**	**S14**
$\text{GEN}_{S_{BF}}$	0:00:04	0:00:14	0:00:48	0:02:55	0:10:24	0:33:28	2:02:15	TO
$\text{GEN}_{SB_{DF}}$	0:00:06	0:00:14	0:00:30	0:01:39	0:04:21	0:21:44	1:00:46	TO
$\text{GEN}_{SB_{BF}}$	0:01:12	0:01:48	0:02:34	0:04:24	0:08:31	0:20:21	1:02:57	TO
$\text{GEN}_{SB_{BF}}$(a)	0:00:27	0:00:40	0:01:04	0:02:15	0:05:38	0:16:37	0:56:57	TO
AVL	**S7**	**S8**	**S9**	**S10**	**S11**	**S12**	**S13**	**S14**
$\text{GEN}_{S_{BF}}$	0:00:03	0:00:10	0:00:30	0:01:28	0:04:44	0:15:57	0:49:21	2:30:56
$\text{GEN}_{SB_{DF}}$	0:00:03	0:00:05	0:00:09	0:00:22	0:01:17	0:04:10	0:13:57	0:49:50
$\text{GEN}_{SB_{BF}}$	0:00:54	0:01:36	0:02:13	0:03:01	0:04:12	0:09:01	0:21:11	0:56:17
$\text{GEN}_{SB_{BF}}$(a)	0:00:15	0:00:27	0:00:39	0:00:59	0:01:42	0:04:57	0:14:23	0:44:21

Cuadro 5.12: Tiempos de generación exhaustiva de estructuras para los enfoques $\text{GEN}_{SB_{DF}}$ y $\text{GEN}_{SB_{BF}}$, incluyendo tiempos de cómputo de cotas ajustadas. $\text{GEN}_{SB_{BF}}$(a) considera los tiempos de cómputos de cotas ajustadas amortizados entre los experimentos de la sección anterior y el actual.

En los resultados que se muestran en la tabla 5.12 se tiene en cuenta el tiempo de cómputo de cotas ajustadas más el tiempo de generación de estructuras para los análisis $\text{GEN}_{SB_{DF}}$ y $\text{GEN}_{SB_{BF}}$. Las filas etiquetadas $\text{GEN}_{SB_{BF}}$(a) muestran el tiempo de cómputo de cotas ajustadas BF amortizado entre los casos de estudio de la sección anterior y la actual (ya que las cotas computadas para las clases BSTree, AVL y TreeSet se pueden reusar también para la generación de estructuras), más el tiempo de generación de estructuras. Los tiempos de $\text{GEN}_{SB_{DF}}$ no se amortizan –aunque esto es técnicamente posible– debido a que el tiempo de cómputo de cotas es mínimo y casi despreciable con respecto al tiempo de generación de estructuras. En estas circunstancias, $\text{GEN}_{SB_{DF}}$ es preferible a $\text{GEN}_{SB_{BF}}$: $\text{GEN}_{SB_{DF}}$ y $\text{GEN}_{SB_{BF}}$ son 139 % y 86 % más rápidos en promedio que $\text{GEN}_{S_{DF}}$ (el enfoque más eficiente que no usa cotas ajustadas). Es decir, $\text{GEN}_{SB_{DF}}$ es 27 % más rápido en promedio que $\text{GEN}_{SB_{BF}}$, a pesar de que la rotura de simetrías BF es 6 % más rápida que la DF. Esto indica que el tiempo de cómputo de cotas ajustadas BF contribuye significativamente en el tiempo total de $\text{GEN}_{SB_{BF}}$ (sin cosiderar estos tiempos $\text{GEN}_{SB_{BF}}$ es 177 % más rápido que $\text{GEN}_{S_{BF}}$), y que en el caso de $\text{GEN}_{SB_{DF}}$ el tiempo de cómputo de cotas ajustadas DF es despreciable. Sin embargo, las cosas cambian si se amortizan los

tiempos de cómputo de cotas ajustadas BF entre los experimentos de esta sección y los de la anterior. En tal caso, $\mathrm{GEN}_{SB_{BF}}$(a) es 144 % más rápido que $\mathrm{GEN}_{S_{BF}}$, y comparable a $\mathrm{GEN}_{SB_{DF}}$. Es decir, el experimento muestra que un enfoque que usa cotas ajustadas BF ($\mathrm{GEN}_{SB_{BF}}$) es preferible si el costo de cómputo de las cotas puede amortizarse entre varios análisis que las utilicen, y si los scopes considerados no son demasiado pequeños (para scopes pequeños $\mathrm{GEN}_{SB_{BF}}$(a) es más lento que $\mathrm{GEN}_{S_{BF}}$). En todos los restantes casos, el enfoque recomendado es $\mathrm{GEN}_{SB_{DF}}$ (recordar que este enfoque requiere de un predicado inductivo de la lógica de separación para generar las cotas). Notar que, de todas maneras, los enfoques que usan cotas ajustadas son significativamente más eficientes que $\mathrm{GEN}_{S_{BF}}$.

Por último, cabe destacar que el mecanismo de aprovechamiento de cotas ajustadas por parte de $\mathrm{GEN}_{SB_{BF}}$ y $\mathrm{GEN}_{SB_{DF}}$ es el más simple posible: las cotas se usan para eliminar variables proposicionales de la fórmula proposicional que se analiza luego a través de un SAT solver. Otros mecanismos que podrían mejorar los beneficios que traen las cotas ajustadas a la generación exhaustiva de estructuras, como el refinamiento de cotas ajustadas, no se implementan en $\mathrm{GEN}_{SB_{BF}}$ y $\mathrm{GEN}_{SB_{DF}}$ (son posteriores a la realización de este trabajo [44], ver sección 5.4).

5.4. Discusión

En esta sección se resumen los resultados de la evaluación experimental realizada en este capítulo, y se realizan algunas observaciones relevantes sobre los resultados y los enfoques evaluados. La primera parte de la experimentación (sección 5.1) compara los tiempos requeridos por los enfoques SLBD, bottom-up –presentados en este trabajo–, y la técnica relacionada TACO+, para computar cotas ajustadas (DF para el primer enfoque, y BF para los últimos). Los resultados son contundentes. Por un lado, bottom-up es un orden de magnitud (15,2 veces) más rápido que TACO+ (simulando su ejecución secuencializada). Por otro lado, el enfoque SLBD basado en predicados inductivos de la lógica de separación es dos órdenes de magnitud (831 veces) más rápido que bottom-up, o tres órdenes de magnitud (1677 veces) si se considera el scope más grande. Esto último se debe a la técnica de memorización, que le permite a SLBD conseguir una escalabilidad superior a la de sus competidores.

Más adelante, en las secciones 5.2.2 y 5.3, se evalúa la utilidad de las cotas ajustadas DF y BF en dos tipos de análisis automático basado en SAT diferentes: verificación acotada de código y generación exhaustiva acotada de estructuras. Las conclusiones de estos experimentos se resumen a continuación.

En la sección 5.2.1 se explicó que los predicados de rotura de simetrías son la principal razón de la escalabilidad y eficiencia del análisis TACO+ de [25]. Compa-

rados con el análisis sin rotura de simetrías, ambos tipos de rotura de simetrías (BF y DF) permiten hasta triplicar los scopes soportados por el verificador de código TACO, y mejorar sustancialmente su eficiencia. Con respecto a la eficiencia, los análisis instrumentados con rotura de simetrías BF son ligeramente más rápidos que los que usan rotura de simetrías DF (en promedio, 20 % para la verificación de código y 6 % para la generación acotada de estructuras). Como se mencionó en 5.2.1, el motivo de la diferencia de eficiencia entre los distintos tipos de rotura de simetrías reside en la mayor complejidad de las fórmulas requeridas para expresar los predicados de rotura de simetrías DF (debido al uso de clausuras reflexo transitivas). No se tuvo en cuenta la generación de estructuras sin rotura de simetrías, ya que algoritmo con tales características generaría una cantidad enorme de estructuras isomorfas, por lo que, además de ser ineficiente, su salida no tendría demasiada utilidad en la práctica.

A pesar de no tener un impacto tan grande en TACO como los predicados de rotura de simetrías, las cotas ajustadas permiten acelerar significativamente los análisis basados en SAT considerados. El uso de cotas ajustadas BF y DF en la verificación de código permite acelerar en 122 % y 120 %, en promedio, la ejecución de los análisis respecto de la verificación con rotura de simetrías DF y BF, respectivamante (sección 5.2.2). Para la generación exhaustiva acotada de estructuras, estos números se incrementan ligeramente, a 177 % y 155 % promedio, para las cotas ajustadas BF y DF, respecto de los análisis con las roturas de simetrías correspondientes (sección 5.3). Además, los análisis que usan conjuntamente rotura de simetrías y cotas ajustadas BF son ligeramente más rápidos que los instrumentados con rotura de simetrías y cotas ajustadas DF, en un 24 % para verificación y un 15 % para generación exhaustiva –aunque una buena parte de esta diferencia se explica por la diferencia en eficiencia de las distintas roturas de simetrías (20 % y 6 % respectivamente). De esta manera, podemos concluir que las cotas ajustadas DF, generadas por el enfoque basado en lógica de separación introducido en este trabajo, proporcionan beneficios comparables a las cotas ajustadas BF –que computa el enfoque relacionado TACO+ (y bottom-up)– al análisis automático basado en SAT. Recordemos que, en la sección 5.2.2, se explicó que las razones de la (mínima) diferencia de eficacia de las cotas BF y DF en TACO no son directamente deducibles de los experimentos llevados a cabo en este trabajo, y que se requiere de experimentación adicional para arribar a conclusiones sobre este asunto.

Al considerar la suma de los tiempos de cómputo de cotas y los tiempos de verificación exhaustiva de código, es decir, los tiempos totales de los análisis bottom-up+TACO y SLBD+TACO, resulta que SLBD+TACO y bottom-up+TACO son 87 % y 41 % más rápidos en promedio que TACO con rotura de simetrías BF ($\text{TACO}_{S_{BF}}$) –el análisis basado en SAT más eficiente sin cotas ajustadas–, respectivamente. Esto significa que el tiempo de cómputo de cotas DF vía SLBD es des-

preciable, ya que se mantiene la ventaja en velocidad de 187 % entre el análisis con rotura de simetrías y cotas ajustadas DF y el análisis con rotura de simetrías BF únicamente. En cambio, el cómputo de cotas ajustadas usando bottom-up tiene un impacto significativo en los tiempos totales de análisis. Si se ignoran los tiempos de cómputo de cotas, la verificación con cotas ajustadas y rotura de simetrías BF supera en eficiencia a la verificación con cotas ajustadas y rotura de simetrías DF. Esto último también es cierto para la generación exhaustiva de estructuras, pero si se consideran los tiempos de cómputo de cotas ajustadas entonces la generación con cotas ajustadas BF también es más lenta que la generación con cotas ajustadas DF: la primera es 86 % más rápida que la generación sin cotas y con rotura de simetrías BF, siendo la segunda 139 % más rápida que esta última. En resumen, de los análisis automáticos evaluados, los que explotan cotas ajustadas (incluyendo el cómputo de las mismas) son significativamente más rápidos que los que no las usan. Es decir, es recomendable computar cotas ajustadas mediante de alguno de los enfoques secuenciales presentados aquí para luego hacer algún tipo de análisis explotando las cotas, en lugar de ejecutar el análisis directamente, sin cotas ajustadas. La única excepción es la combinación de cómputo de cotas ajustadas BF y scopes pequeños, donde el reducido impacto de las cotas ajustadas en el análisis posterior no es suficiente para amortizar el tiempo requerido para computar las cotas. De cualquier manera, el tiempo total de análisis en estos casos es relativamente pequeño. En cuanto a la conveniencia de usar cotas ajustadas DF o BF, se recomiendan las primeras si las cotas se van a reutilizar unas pocas veces –debido a que el costo de su cómputo es despreciable–. En caso contrario, es decir, cuando las mismas cotas se puedan usar en varios análisis distintos, el significativo tiempo de cómputo de cotas ajustadas BF se amortiza entre los diferentes experimentos, y estas cotas (junto con la rotura de simetría correspondiente) pueden ofrecer una mejora en performance ligeramente mayor que las cotas DF.

Con respecto al enfoque SLBD, es importante subrayar que sólo pueden aprovecharse sus beneficios si se cuenta con predicados inductivos de la lógica de separación que describen los invariantes de las clases a analizar. Esto significa que el usuario del análisis debe definir invariantes en este formalismo. Así, una ventaja del lenguaje usado por bottom-up y TACO+, JML, es que tiene un mayor grado de adopción entre los programadores, lo que facilita la aplicación de los análisis de programas basados en estas técnicas en la práctica. Otra ventaja de JML sobre el subconjunto de la lógica de separación usado en este trabajo –y por lo tanto de bottom-up y TACO+ respecto de SLBD– es su mayor poder expresivo. Como se discutió en la sección 4.2.2, expresar estructuras que poseen subestructuras compartidas, por ejemplo grafos, no es posible en nuestro subconjunto de la lógica de separación.

Una potencial ventaja del enfoque *top-down* TACO+ (que comienza con todos

los pares en la cota ajustada, y los va refinando iterativamente), respecto de los enfoques bottom-up y SLBD (que empiezan con una cota vacía, que van extendiendo a medida que avanza su ejecución), es que, por sus características, el primero puede interrumpirse en cualquier momento, y aún así retornar una cota ajustada (aunque no sea la más precisa). En cambio, los restantes enfoques no pueden retornar una cota ajustada hasta que su ejecución no finaliza correctamente (en tal caso siempre computan la cota ajustada más precisa). Sería interesante evaluar en el futuro el costo de computar cotas ajustadas menos precisas usando algún enfoque *top-down*, y su impacto en el análisis de programas acotado. Si bien creemos que es muy difícil que dicho enfoque supere la velocidad de cómputo de cotas de SLBD (y la precisión de las mismas), existe la posibilidad de que este compromiso entre precisión y tiempo de cómputo de las cotas ajustadas pueda resultar en un enfoque competitivo o superior a bottom-up+TACO. Por otro lado, cabe destacar que las estructuras consideradas en nuestra evaluación experimental poseen invariantes fuertes. Consecuentemente, las cotas ajustadas para estas estructuras contienen una proporción pequeña de valores, respecto del conjunto de todos los posibles valores para los campos. Esto beneficia claramente a los enfoques *bottom-up*, ya que necesitan realizar una cantidad relativamente pequeña de invocaciones a un procedimiento de decisión para completar el cómputo de las cotas ajustadas. Por lo tanto, otro experimento interesante a realizar en el futuro es una comparación de los enfoques de cómputo de cotas ajustadas en estructuras con invariantes permisivos, como por ejemplo, grafos.

Para concluir, es importante mencionar los últimos avances en enfoques de análisis de código que aprovechan cotas ajustadas. El caso más notable es el de BLISS [44], que explota las cotas ajustadas BF (computadas por bottom-up y TACO+) para eliminar condiciones de camino inválidas (debido a las cotas) durante la ejecución simbólica del programa bajo análisis. BLISS es capaz de usar las cotas ajustadas de manera más inteligente que TACO: en BLISS las cotas se usan durante toda la ejecución del análisis, no sólo en el estado inicial, como es el caso de TACO. Adicionalmente, BLISS introduce la noción de refinamiento de cotas ajustadas. Esto significa que, en cada paso de la ejecución simbólica BLISS puede eliminar valores de las cotas ajustadas, observando a la parte explorada de la estructura corriente. Por último, BLISS efectúa consultas a SAT que, en conjunción con las cotas ajustadas (que pueden ser más refinadas que las computadas para el estado inicial), permiten detectar condiciones de camino inválidas tan pronto como sea posible en la ejecución simbólica. Debido a estos mecanismos más sofisticados de uso de cotas ajustadas, BLISS es más de 5 veces más rápido que el enfoque original (ejecución simbólica sin cotas ajustadas), en la mayoría de los casos de estudio de [44], y de hasta 4 órdenes de magnitud en casos "difíciles", superando ampliamente las ganancias que se obtuvieron de explotar cotas ajustadas en los análisis de programas considerados

en nuestros experimentos. El trabajo de incorporación de las cotas ajustadas DF en BLISS –que suponemos debería ser directo– se llevará a cabo en el futuro. En vista de los resultados experimentales discutidos arriba, creemos que las cotas ajustadas DF deberían tener un impacto similar a las BF en el rendimiento de BLISS.

Capítulo 6

Trabajos relacionados, conclusiones y trabajos futuros

6.1. Trabajos Relacionados

El enfoque de cómputo de cotas ajustadas TACO+, presentado en [25, 27], es el trabajo más estrechamente relacionado con esta tesis. Por un lado, porque introducen el concepto de cotas ajustadas, y el primer algoritmo eficiente (paralelo) para computarlas. Por otro lado, porque en este trabajo se usa el análisis de código TACO, también presentado en dichos artículos, para evaluar el impacto de las cotas producidas por los enfoques bottom-up y SLBD. Recordemos que tanto TACO+ como bottom-up computan cotas ajustadas en base a invariantes de clases descritos en lenguajes declarativos, por lo que ambas técnicas son intercambiables. En particular, las implementaciones de las técnicas soportan invariantes JML. En cambio, SLBD se basa en predicados inductivos de la lógica de separación. Su ventaja principal es una amplia ventaja en rendimiento respecto de las técnicas bottom-up y TACO+. Sin embargo, las especificaciones en términos de predicados inductivos pueden ser más complicados de definir que sus contrapartes declarativas. Además, los lenguajes declarativos como JML son más conocidos por los desarrolladores de software que la lógica de separación, lo que puede dificulta la adopción de este último formalismo en la práctica. Por último, a diferencia de bottom-up y TACO+, nuestra implementación actual de SLBD no computa cotas ajustadas para los campos que almacenan la información de los nodos en las estructuras. De todos modos, para las estructuras consideradas las restricciones para estos campos son débiles, por lo que las cotas resultantes son "poco ajustadas" (contienen casi todos los pares posibles). Los experimentos de las secciones 5.2.2 y 5.3 muestran que la incidencia de las cotas ajustadas para estos campos en el comportamiento de los análisis basados en SAT

considerados no es demasiado significativa. Las comparaciones experimentales entre TACO+, bottom-up y SLBD y sus conclusiones se presentaron en las secciones 5.1.3 y 5.1.3.

JForge [17] es otra herramienta de verificación acotada basada en SAT para código JAVA con especificaciones JML. Sin embargo, JForge no utiliza cotas ajustadas. Los experimentos de [27] muestran que JForge es superada ampliamente en eficiencia y escalabilidad por TACO+ –debido al uso de cotas ajustadas y rotura de simetrías–, y que exhibe una performance similar a la de TACO. Como JForge no ha sufrido mejoras recientes que la hagan más competitiva, se optó por excluirla del análisis experimental. Otros ejemplos de análisis de código basados en SAT se introducen en [32, 48], pero estos están diseñados específicamente para el lenguaje C, y tampoco utilizan cotas ajustadas.

Además de TACO, existen otros enfoques que se benefician del uso de cotas ajustadas. FAJITA [5] es una adaptación de TACO+, cuyo propósito es la generación automática de casos de prueba para código JAVA. Al igual que TACO, la eficiencia y escalabilidad respecto a los scopes de FAJITA depende fuertemente del uso de cotas ajustadas. En [28] se propone una adaptación del mecanismo de inicialización perezosa (*lazy initialization*) del model checker Symbolic JAVA PathFinder [3, 46] para aprovechar cotas ajustadas. Esto permite mejorar la performance del model checker en la generación automática de casos de prueba para código JAVA. En [44] se incorpora al enfoque de [28] el refinamiento de cotas ajustadas basado en SAT solving, y consultas a SAT para eliminar condiciones de camino tan pronto como sea posible en la ejecución simbólica. La técnica presentada en [44] produce mejoras sustanciales en la performance de Symbolic PathFinder, debido a los mecanismos sofisticados de aprovechamiento de cotas ajustadas que implementa. Sin embargo, no pudo incluirse esta técnica en las comparaciones experimentales de este trabajo, ya que se encontraba en etapa de desarrollo al momento de escribir esta tesis. [40] presenta un análisis basado en *dataflow* para propagar las cotas ajustadas computadas para el estado inicial –como las computadas por los enfoques presentados en este trabajo– al resto de los estados del programa. Como resultado de la propagación se obtiene una mejora en la eficiencia de los análisis basados en SAT que usan cotas ajustadas sólo para el estado inicial (TACO+, bottom-up+TACO, SLBD+TACO). Es importante destacar que ninguno de los enfoques mencionados en este párrafo proponen técnicas alternativas de cómputo de cotas ajustadas, y que todos ellos pueden beneficiarse de las técnicas presentadas aquí.

Respecto a los trabajos relacionados en el área de análisis automáticos basados en la lógica de separación, se utiliza aquí el fragmento de la lógica de separación de [39], para definir los predicados inductivos que el enfoque SLBD usa como entrada. En [39] los predicados inductivos se usan para probar propiedades de “forma”

(ej. invariantes de representación de estructuras dinámicas como listas, árboles, etc.) y de "tamaño" (ej. propiedades sobre la cantidad de elementos en las estructuras dinámicas, de balanceo de árboles, etc.), de programas que manipulan estructuras de datos dinámicas. Los artículos [14, 38] extienden el trabajo de [39], mejorando el procedimiento de verificación a través de diferentes mecanismos. Otros ejemplos exitosos de verificación automática basados en la lógica de separación se presentan en [8, 11, 36], cuyo objetivo es el de probar que los programas no cometen violaciones de memoria, como por ejemplo, dereferenciar variables no inicializadas, alocar memoria y no liberarla antes de finalizar la ejecución, etc. Los trabajos mencionados anteriormente utilizan los predicados de forma y algún cálculo para los mismos, e intentan probar propiedades haciendo ejecución simbólica con los predicados. Nuestro enfoque es diferente: los predicados inductivos se usan para computar cotas ajustadas mediante SLBD, que luego pueden ser usadas por varios tipos de análisis exhaustivos acotados, como los de verificación de propiedades y de generación automática de casos de prueba mencionados en el párrafo anterior.

Por último, es importante mencionar que en la literatura pueden encontrarse numerosos ejemplos de análisis automáticos que utilizan rotura de simetrías, y que, consecuentemente, pueden beneficiarse de una cantidad reducida de instancias isomorfas durante el análisis. Por ejemplo, [31, 37] canonizan el heap en el contexto de model checking. Por otro lado, diversos enfoques rompen simetrías para acelerar la generación automática de casos de prueba, tanto de caja negra [9], como de caja blanca [29, 46].

6.2. Conclusiones

Las cotas ajustadas fueron concebidas como un mecanismo para mejorar la eficiencia de diversos tipos de análisis acotado (automático) de código. Los ejemplos más destacados de análisis de código automático que usan cotas ajustadas son el verificador de propiedades TACO [25, 27] (basado en SAT), y los generadores de casos de prueba FAJITA [5] (basado en SAT) y Symbolic JAVA PathFinder [28, 44] (basado en model checking). Sin embargo, hasta el momento el único enfoque conocido que presenta una performance razonable, TACO+, es intrínsecamente paralelo, y requiere de un cluster con varias computadoras para computar cotas ajustadas en un tiempo aceptable.

En esta tesis se provee evidencia de que las cotas ajustadas pueden computarse eficentemente en entornos de ejecución secuenciales –en donde computar cotas ajustadas vía TACO+ puede resultar en tiempos de ejecución inaceptables. En particular, en nuestros experimentos el enfoque bottom-up es un orden de magnitud

más rápido que el algoritmo de cómputo de cotas ajustadas de TACO+, simulando su ejecución en una sola computadora. Además, bottom-up genera cotas ajustadas equivalentes a TACO+ (cotas ajustadas BF). Consecuentemente, cuando se dispone de invariantes de clases especificados en lenguajes declarativos, como JML, y un entorno de ejecución secuencial, bottom-up es un buen sustituto para el algoritmo de cómputo de cotas de TACO+. Por otra parte, SLBD es el enfoque secuencial de cómputo de cotas ajustadas más eficiente: dos o tres órdenes de magnitud más rápido que bottom-up. A diferencia de bottom-up y TACO+, SLBD genera cotas ajustadas DF. En nuestros experimentos, las cotas ajustadas DF probaron tener un impacto similar en la eficiencia de los análisis considerados a las cotas ajustadas BF, aún cuando su tiempo de cómputo es despreciable. Sin embargo, SLBD requiere invariantes de clase especificados como predicados inductivos de la lógica de separación, lo que puede ser considerado una desventaja, debido a que la lógica de separación es un formalismo menos difundido que JML. De esta manera, para el programador promedio trabajar con predicados inductivos de esta lógica puede resultar más difícil y trabajoso que con especificaciones JML.

Un resultado importante de este trabajo es que los enfoques presentados habilitan la definición de análisis automáticos (verificación de código y generación exhaustiva de estructuras) que computan y explotan cotas ajustadas, y que pueden ejecutarse íntegramente en una única PC. Es más, estos análisis automáticos son, en general, más rápidos en tiempo que sus contrapartes que no usan cotas ajustadas (con la única excepción de computar cotas ajustadas BF y realizar luego un análisis con scopes pequeños). Computar cotas ajustadas DF y luego utilizarlas en alguno de los análisis considerados mejora, en promedio, un 117 % los tiempos respecto del mejor análisis que no usa cotas ajustadas. En cambio, para las cotas ajustadas BF la ganancia es del 63 %. Si bien el impacto de las cotas ajustadas BF en los tiempos de análisis es ligeramente superior al de las DF, la ventaja en favor de los análisis que usan cotas ajustadas DF viene dada por la gran diferencia en velocidad entre los enfoques de cómputo de cotas SLBD y bottom-up. Sin embargo, cuando se realizan varios experimentos con las mismas cotas ajustadas (por ejemplo, cuando se analizan múltiples métodos de una clase), los tiempos de cómputo de cotas se amortizan entre los diferentes experimentos. En tal caso, el impacto de las cotas ajustadas BF en el análisis puede llegar a ser ligeramente superior al de las DF. De lo anterior se desprende que en la mayoría de los casos (salvo la excepción mencionada arriba) para los análisis considerados es preferible computar cotas ajustadas –mediante alguno de los enfoques presentados en este trabajo– y luego aprovecharlas durante el análisis, que ejecutar el análisis sin cotas ajustadas.

Por último, es importante mencionar que los resultados de nuestra evaluación experimental no representan los máximos beneficios que las cotas ajustadas pue-

den brindar a los análisis de programas. De hecho, los enfoques considerados en las secciones 5.2 y 5.3 implementan un método simple y directo de uso de las cotas ajustadas: las cotas ajustadas sólo les permiten quitar variables proposicionales de la representación del estado inicial. En cambio, nuevas técnicas de análisis son capaces de aprovechar más inteligentemente las cotas ajustadas, como por ejemplo el generador automático de casos de prueba de caja blanca para programas JAVA denominado BLISS [44]. BLISS combina el mecanismo de ejecución simbólica con inicialización lazy del model checker Java PathFinder, aprovechamiento de cotas ajustadas (BF en su versión actual), refinamiento de las cotas ajustadas a medida que avanza la ejecución, e invocaciones a SAT (que usan las cotas ajustadas); para descartar condiciones de camino (*path conditions*) inválidas tan pronto como sea posible en la ejecución simbólica del programa de entrada. Este uso "inteligente" de las cotas ajustadas por parte de BLISS lo convierte en más de 5 veces más rápido que el algoritmo original de inicialización lazy de Java PathFinder (sin cotas) en el 60 % de los casos de estudio de [44]. Además, para programas complejos BLISS obtiene una mejora en tiempo de 4 órdenes de magnitud por sobre Java PathFinder. Estos números superan sustancialmente a los de las técnicas consideradas en este trabajo, y son una muestra del potencial de las cotas ajustadas en el análisis automático de programas.

6.3. Trabajos futuros

Como trabajos futuros se piensan estudiar los beneficios provistos por las cotas ajustadas en otras clases de análisis acotados. Por ejemplo, como parte de un trabajo en curso se está investigando el impacto de utilizar cotas ajustadas en la generación exhaustiva de estructuras (FAJITA y Symbolic JAVA PathFinder son enfoques de generación de casos de prueba de caja blanca), a través de la herramienta Korat [9]. Resultados preliminares indican que la performance de Korat mejora si se evita la exploración de las partes del espacio de estados que son inconsistentes con las cotas ajustadas. Además, sería interesante medir el impacto de las cotas ajustadas DF en la herramienta de generación automática de casos de prueba de caja blanca BLISS [44], respecto de las cotas ajustadas BF, para sumar evidencia que sostenga nuestra hipótesis de que las primeras son tan útiles como las segundas.

Por otra parte, sería deseable implementar una variante del enfoque bottom-up que aproveche las capacidades de los SAT solvers incrementales, es decir, que no tenga que empezar el análisis basado en SAT desde cero cada vez que hace una consulta para requerir una nueva instancia válida del heap. Dado que bottom-up es un enfoque inherentemente incremental, creemos que el desarrollo de esta técnica

no debería presentar mayores problemas. Esta nueva versión de bottom-up debería presentar mejoras significativas de rendimiento respecto a los derivados de bottom-up definidos en este trabajo.

Por último, nos gustaría explorar más en profundidad la relación entre los análisis acotados basados en SAT y la lógica de separación. Por un lado, sería deseable encontrar una codificación eficiente para los predicados inductivos de la lógica en algún lenguaje declarativo, como Alloy. Esto eliminaría la necesidad de usar dos especificaciones diferentes de la misma estructura de datos en el enfoque SLBD+TACO. Por otro lado, la separación del heap en subheaps disjuntos permitida por el operador $*$ podría ser útil para mejorar la eficiencia de la generación exhaustiva de estructuras basada en SAT. La intuición es que, para generar estructuras con k elementos que satisfagan $f_1 * f_2$, se podrían construir independientemente todas las estructuras con k' elementos que satisfagan f_1, y las que cumplan con f_2 y posean k'' elementos, con $k = k' + k''$. Luego, los resultados se combinarían para formar el conjunto de estructuras que satisfacen $f_1 * f_2$. La idea es dividir la generación de estructuras basada en SAT de una fórmula compleja para un scope grande en subproblemas más pequeños e independientes, para los que se puedan usar scopes menores, de modo que la resolución de estos subproblemas sea más simple y rápida.

Bibliografía

[1] *Alloy homepage*, http://alloy.mit.edu/alloy/. 2.1, 2.4

[2] *Dynalloy translator homepage*, http://www.dc.uba.ar/inv/grupos/rfm_folder/dynalloy. 2.2.2, 2.5.1

[3] *Java pathfinder model checker homepage.*, http://babelfish.arc.nasa.gov/trac/jpf. 1.3, 6.1

[4] *Yices homepage*, http://yices.csl.sri.com/index.shtml. 2.10, 4.1.3

[5] Pablo Abad, Nazareno Aguirre, Valeria Bengolea, Daniel Ciolek, Marcelo F. Frias, Juan Galeotti, Tom Maibaum, Mariano Moscato, Nicolas Rosner, and Ignacio Vissani, *Improving test generation under rich contracts by tight bounds and incremental sat solving*, 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, ICST'13, IEEE Computer Society, 2013, pp. 21–30. 1.3, 6.1, 6.2

[6] Alexandr Andoni, Dumitru Daniliuc, and Sarfraz Khurshid, *Evaluating the "small scope hypothesis"*, Tech. report, MIT-LCS-TR-921, MIT CSAIL, 2003. 1.2

[7] Jason Belt, Robby, and Xianghua Deng, *Sireum/topi ldp: a lightweight semi-decision procedure for optimizing symbolic execution-based analyses*, Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, ESEC/FSE '09, ACM, 2009, pp. 355–364. 5

[8] Josh Berdine, Byron Cook, and Samin Ishtiaq, *Slayer: memory safety for systems-level code*, Proceedings of the 23rd international conference on Computer aided verification, CAV'11, Springer-Verlag, 2011, pp. 178–183. 2.9, 2.9.3, 6.1

[9] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov, *Korat: automated testing based on java predicates*, Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis, ISSTA '02, ACM, 2002, pp. 123–133. 5.3, 6.1, 6.3

[10] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani, *The mathsat 4 smt solver*, Proceedings of the 20th International Conference on Computer Aided Verification, CAV '08, Springer-Verlag, 2008, pp. 299–303. 2.10

[11] Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang, *Compositional shape analysis by means of bi-abduction*, Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '09, ACM, 2009, pp. 289–300. 2.9, 2.9.3, 6.1

[12] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll, *Beyond assertions: advanced specification and verification with jml and esc/java2*, Proceedings of the 4th international conference on Formal Methods for Components and Objects, FMCO'05, Springer-Verlag, 2006, pp. 342–363. 1.3, 2.3.1, 2.5.1

[13] Renato Cherini, Lucas Rearte, and Javier Blanco, *A shape analysis for non-linear data structures*, Proceedings of the 17th International Conference on Static Analysis, SAS'10, Springer-Verlag, 2010, pp. 201–217. 4.2.2

[14] Wei-Ngan Chin, Cristian Gherghina, Răzvan Voicu, Quang Loc Le, Florin Craciun, and Shengchao Qin, *A specialization calculus for pruning disjunctive predicates to support verification*, Proceedings of the 23rd international conference on Computer aided verification, CAV'11, Springer-Verlag, 2011, pp. 293–309. 2.9, 6.1

[15] Stephen A. Cook, *The complexity of theorem-proving procedures*, Proceedings of the third annual ACM symposium on Theory of computing, STOC '71, ACM, 1971, pp. 151–158. 2.4

[16] Leonardo De Moura and Nikolaj Bjørner, *Z3: An efficient smt solver*, Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'08/ETAPS'08, Springer-Verlag, 2008, pp. 337–340. 2.10

[17] Greg Dennis, Felix Sheng-Ho Chang, and Daniel Jackson, *Modular verification of code with sat*, Proceedings of the 2006 international symposium on Software testing and analysis, ISSTA '06, ACM, 2006, pp. 109–120. 1.2, 2.5, 5.2.2, 6.1

[18] Edsger W. Dijkstra and Carel S. Scholten, *Predicate calculus and program semantics*, Springer-Verlag New York, Inc., 1990. 2.2.2

[19] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang, *A local shape analysis based on separation logic*, Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'06, Springer-Verlag, 2006, pp. 287–302. 2.9.3

[20] Mark Dowson, *The ariane 5 software failure*, SIGSOFT Softw. Eng. Notes **22** (1997), no. 2, 84–. 1.1

[21] Bruno Dutertre and Leonardo de Moura, *A fast linear-arithmetic solver for dpll(t)*, Proceedings of the 18th International Conference on Computer Aided Verification, CAV'06, Springer-Verlag, 2006, pp. 81–94. 2.10

[22] Marcelo F. Frias, Juan P. Galeotti, Carlos G. López Pombo, and Nazareno M. Aguirre, *Dynalloy: upgrading alloy with actions*, Proceedings of the 27th international conference on Software engineering, ICSE '05, ACM, 2005, pp. 442–451. 2.2, 2.2.1, 2.2.2

[23] Marcelo F. Frias, Carlos G. Lopez Pombo, Juan P. Galeotti, and Nazareno M. Aguirre, *Efficient analysis of dynalloy specifications*, ACM Trans. Softw. Eng. Methodol. **17** (2007), no. 1, 4:1–4:34. 2.2, 2.2.2

[24] Juan P. Galeotti and Marcelo F. Frias, *Dynalloy as a formal method for the analysis of java programs*, Software Engineering Techniques: Design for Quality, SET 2006, October 17-20, 2006, Warsaw, Poland (Krzysztof Sacha, ed.), IFIP, vol. 227, Springer, 2006, pp. 249–260. 2.3, 2.3.2

[25] Juan P. Galeotti, Nicolas Rosner, Carlos G. Lopez Pombo, and Marcelo F. Frias, *Taco: Efficient sat-based bounded verification using symmetry breaking and tight bounds*, IEEE Transactions on Software Engineering **99** (2013), no. PrePrints, 1. 1.2, 1.2.1, 1.2.1, 1.3, 2.6, 2.7.3, 2.8, 5, 5.1.3, 5.2.1, 5.2.2, 5.4, 6.1, 6.2

[26] Juan Pablo Galeotti, *Verificación de software usando Alloy*, Ph.D. thesis, Universidad de Buenos Aires. Argentina, 2010. 2.3

[27] Juan Pablo Galeotti, Nicolás Rosner, Carlos Gustavo López Pombo, and Marcelo Fabian Frias, *Analysis of invariants for efficient bounded verification*, Proceedings of the 19th international symposium on Software testing and analysis, ISSTA '10, ACM, 2010, pp. 25–36. 1.2, 1.2.1, 2.5, 2.5.1, 2.6, 2.6.1, 2.6.1, 5.2.2, 6.1, 6.2

[28] Jaco Geldenhuys, Nazareno Aguirre, Marcelo F. Frias, and Willem Visser, *Bounded lazy initialization*, Proceedings of the 5th NASA Formal Methods Symposium, NFM'13, Springer, 2013, pp. 229–243. 1.3, 6.1, 6.2

[29] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov, *Test generation through programming in udita*, Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE '10, ACM, 2010, pp. 225–234. 6.1

[30] C. A. R. Hoare, *An axiomatic basis for computer programming*, Communications of the ACM **12** (1969), no. 10, 576–580. 2.2.1, 2.9.3

[31] Radu Iosif, *Symmetry reduction criteria for software model checking*, Proceedings of the 9th International SPIN Workshop on Model Checking of Software, Springer-Verlag, 2002, pp. 22–41. 6.1

[32] Franco Ivanăić, Zijiang Yang, Malay K. Ganai, Aarti Gupta, Ilya Shlyakhter, and Pranav Ashar, *F-soft: software verification platform*, Proceedings of the 17th international conference on Computer Aided Verification, CAV'05, Springer-Verlag, 2005, pp. 301–306. 1.2, 6.1

[33] Daniel Jackson, *Software abstractions - logic, language, and analysis*, MIT Press, 2006. 2.1, 2.1.1

[34] Daniel Jackson and Mandana Vaziri, *Finding bugs with a constraint solver*, Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis, ISSTA '00, ACM, 2000, pp. 14–25. 2.3.1

[35] Nancy Leveson and Clark Turner, *An investigation of the therac-25 accidents*, Computer **26** (1993), no. 7, 18–41. 1.1

[36] Stephen Magill, Ming-Hsien Tsai, Peter Lee, and Yih-Kuen Tsay, *Automatic numeric abstractions for heap-manipulating programs*, Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '10, ACM, 2010, pp. 211–222. 2.9, 6.1

[37] Madanlal Musuvathi and David L. Dill, *An incremental heap canonicalization algorithm*, Proceedings of the 12th international conference on Model Checking Software, SPIN'05, Springer-Verlag, 2005, pp. 28–42. 6.1

[38] Huu Hai Nguyen and Wei-Ngan Chin, *Enhancing program verification with lemmas*, Proceedings of the 20th international conference on Computer Aided Verification, CAV '08, Springer-Verlag, 2008, pp. 355–369. 2.9, 6.1

[39] Huu Hai Nguyen, Cristina David, Shengchao Qin, and Wei-Ngan Chin, *Automated verification of shape and size properties via separation logic*, Proceedings of the 8th international conference on Verification, model checking, and abstract interpretation, VMCAI'07, Springer-Verlag, 2007, pp. 251–266. 2.9, 2.9.3, 4.2.2, 6.1

[40] Bruno Cuervo Parrino, Juan Pablo Galeotti, Diego Garbervetsky, and Marcelo F. Frias, *A dataflow analysis to improve sat-based bounded program verification*, Proceedings of the 9th international conference on Software engineering and formal methods, SEFM'11, Springer-Verlag, 2011, pp. 138–154. 5.2.2, 6.1

[41] Pablo Ponzio, Nicolás Rosner, Nazareno Aguirre, and Marcelo Frias, *Efficient tight field bounds computation based on shape predicates*, FM 2014: Formal Methods (Cliff Jones, Pekka Pihlajasaari, and Jun Sun, eds.), Lecture Notes in Computer Science, vol. 8442, Springer International Publishing, 2014, pp. 531–546 (English). 4

[42] John C. Reynolds, *Intuitionistic reasoning about shared mutable data structure*, Millennial Perspectives in Computer Science, Palgrave, 2000, pp. 303–321. 2.9

[43] John C. Reynolds, *Separation logic: A logic for shared mutable data structures*, Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science, LICS '02, IEEE Computer Society, 2002, pp. 55–74. 1.3, 2.9, 2.9.3, 2.9.4

[44] Nicolás Rosner, Jaco Geldenhuys, Nazareno M. Aguirre, Willem Visser, and Marcelo F. Frias, *Bliss: Bounded lazy initialization with sat support*. 1.3, 5.3, 5.4, 6.1, 6.2, 6.3

[45] Emina Torlak and Daniel Jackson, *Kodkod: a relational model finder*, Proceedings of the 13th international conference on Tools and algorithms for the construction and analysis of systems, TACAS'07, Springer-Verlag, 2007, pp. 632–647. 2.1, 2.4, 2.4.2

[46] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid, *Test input generation with java pathfinder*, Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis, ISSTA '04, ACM, 2004, pp. 97–107. 6.1

[47] Willem Visser, Corina S. Păsăreanu, and Radek Pelánek, *Test input generation for java containers using state matching*, Proceedings of the 2006 international

symposium on Software testing and analysis, ISSTA '06, ACM, 2006, pp. 37–48. 5

[48] Yichen Xie and Alex Aiken, *Saturn: A scalable framework for error detection using boolean satisfiability*, ACM Trans. Program. Lang. Syst. **29** (2007), no. 3. 1.2, 6.1