Analizando la forma de estructuras de datos no lineales en memoria dinámica con Separation Logic

Lucas Agustín Rearte
Directores: Renato Cherini y Javier Blanco

31 de mayo de 2017

Universidad Nacional de Córdoba
Facultad de Matemática, Astronomía, Física y Computación

Resumen

Presentamos un shape analysis con garantías de terminación para programas que manipulan estructuras de datos no lineales como árboles binarios. El análisis se basa en una ejecución simbólica de los programas sobre estados abstractos compuestos por fórmulas de un conjunto restringido de la Separation Logic. Dada una precondición, calcula automáticamente una postcondición e invariantes concisos para los ciclos del programa. El uso de un nivel de relevancia para las variables según el tipo de manipulación que sufren permite lograr un balance entre precisión y abstracción en las fórmulas. Mostramos los resultados obtenidos por un prototipo que implementa nuestro análisis sobre una variedad de ejemplos.

Abstract

We present a terminating shape analysis for programs manipulating non-linear data structures such as binary trees. The analysis is based on symbolic execution of programs over abstract states composed by formulæ of a restricted subset of Separation Logic. Given a precondition, it automatically calculates concise postconditions and invariants for program loops. The use of relevance level of program variables depending on the manipulations applied on them gives us a balance between precision and abstraction in the formulæ. We report experimental results obtained from running a prototype implementing our analysis on a variety of examples.

Palabras claves: Separation Logic, shape analysis, memoria dinámica, verificación, árboles binarios, fallo de memoria, memory leak, análisis estático.

ACM CCS 2012: Theory of computation∼Program reasoning, Theory of computation∼Separation logic, Theory of computation∼Program verification.

Agradecimientos

A mis padres, Silvia y Juan, y a mis hermanos, Pablo e Ignacio, por el amor y el afecto, por el apoyo constante y la motivación necesaria para poder realizar este trabajo.

A mis directores, Renato y Javier, por permitirme trabajar junto a ellos y por guiarme en este camino. A Renato, por su tiempo y ayuda brindados en estos últimos meses, importantes para que pueda concluir la tesis residiendo a cientos de kilómetros de Córdoba.

A los miembros del tribunal, Pedro y Beta, por la rápida lectura de este trabajo y por sus comentarios.

A los compañeros de la facultad, por aquellos años de cursada y por los momentos compartidos.

A mis jefes, por la flexibilidad otorgada para poder escribir esta tesis simultáneamente con las tareas laborales.

A la familia grande, a los amigos de acá y de allá, a todos los que estuvieron atentos y deseosos de que terminara este trabajo y que, al igual que yo, están felices por este objetivo cumplido.

Índice general

Capítulo 1

Introducción

La obtención de programas correctos, es decir, aquellos que terminan normalmente y realizan la tarea deseada, es de suma importancia en este mundo cada vez más dependiente de los procesos computacionales. Los errores en los programas pueden resultar costosos para sistemas que cada vez tienen un mayor componente de software.

Un enfoque comúnmente utilizado en la industria del software con el objetivo de obtener programas correctos es el uso del *testing*. Que un programa se comporte de acuerdo a lo esperado para un conjunto de casos de prueba incrementa la confianza en su correctitud, pero no permite demostrarla. El *testing* solo puede probar la existencia de errores en un programa, no su ausencia.

Otro enfoque posible para obtener programas correctos (y bastante menos utilizado en la industria del software) consiste en la verificación formal de los mismos, mediante el uso de lógicas. En este trabajo se sigue este enfoque para demostrar, al menos parcialmente, la correctitud de programas sobre un lenguaje imperativo con uso explícito de la memoria dinámica.

La lógica de Hoare [13] es uno de los formalismos más antiguos para el razonamiento y la verificación de programas imperativos. El concepto clave que presenta es la terna de Hoare: $\{p\}\, c\, \{q\}$, que relaciona las aserciones p y q, llamadas precondición y postcondición respectivamente, con un programa c. Las aserciones son fórmulas que describen en lógica de primer orden un estado del programa. Cuando una terna es válida se cumple que siempre que se ejecute el programa c partiendo de un estado que satisface la precondición p, si la ejecución termina, lo hace en un estado que satisface la postcondición q. Esto se llama corrección parcial, ya que no establece nada sobre los programas que no terminan, a diferencia de la corrección total que además exige que el programa termine. En la lógica de Hoare se presentan reglas axiomáticas para la derivación de las ternas, de manera que una terna que se obtenga mediante la aplicación de esas reglas siempre resulta válida.

Originalmente el lenguaje imperativo simple de Hoare no tenía comandos para el uso de la memoria dinámica, sino que toda la memoria que se usaba estaba dada solamente por las variables de programa. Al extender el modelo de estados con el fin de incorporarla, el espacio de memoria queda dividido en dos partes: el *stack*, que contiene registros estáticos asociados a las variables de programa y el *heap*, con una cantidad arbitraria de registros dinámicos que siendo anónimos, pueden ser referenciados indirectamente por variables del *stack*

o por otros registros del *heap*, llamados punteros.

El uso de la memoria dinámica, más allá de las facilidades que otorga al poder representar y manipular estructuras de datos arbitrarias, tiene una dificultad extra para el razonamiento sobre programas, causada por el *aliasing*. Dos punteros que referencian al mismo registro de memoria del *heap* se denominan *alias*. Si uno de los punteros modifica el registro que referencia, el registro referenciado por el otro puntero se modifica igualmente, ya que son el mismo. Esto podría suceder inadvertidamente para el segundo puntero, posiblemente invalidando alguna propiedad expresada en términos de dicho registro.

La lógica de primer orden alcanza un límite práctico en el razonamiento sobre programas que manipulan la memoria dinámica. El problema del *aliasing* evidencia la dificultad de expresar el efecto de comandos simples, como la modificación de un registro en la memoria. Supongamos que el predicado $x \mapsto v$ expresa que en el *heap* existe un registro referenciado por la variable x cuyo valor es v. Supongamos que $[x]$ representa el contenido de este registro de memoria y que $[x] := w$ es un comando de mutación que le asigna el valor w al registro referenciado por x. Es de esperar que la siguiente terna de Hoare sea válida:

$$\{x \mapsto v\}\, [x] := w\, \{x \mapsto w\}$$

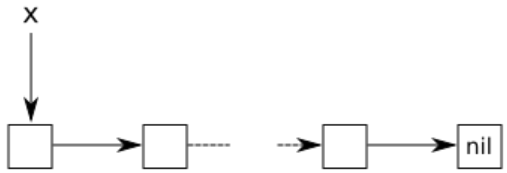
Sin embargo, el efecto del comando de mutación ya no puede expresarse fácilmente en un contexto más global, donde haya otras variables. La terna:

$$\{x \mapsto v \land y \mapsto v\}\, [x] := w\, \{x \mapsto w \land y \mapsto v\}$$

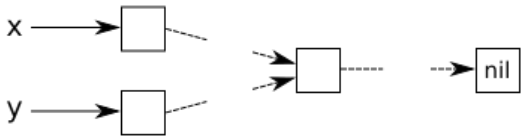
pierde su validez ya que no evita el caso del *aliasing* que se podría dar si x e y referenciaran al mismo registro de memoria. Este caso particular se puede resolver si se fortalece la precondición con el predicado $x \neq y$. A pesar de ello, esta solución no resulta tan sencilla de aplicar en el caso general ya que, además de las múltiples variables de programa que pueda haber, el *aliasing* se puede dar también con los valores de los registros de la memoria, que son *anónimos*. Este último caso se evidencia en el uso de predicados que describen estructuras de datos enlazadas, como listas o árboles. Supongamos por ejemplo que el predicado $\mathbf{list}.x$ establece que hay una lista enlazada que comienza con el registro apuntado por la variable x y termina con un registro con el valor especial **nil**, como en la figura 1.1a.[1] Supongamos que tenemos un programa que necesita combinar de alguna manera dos listas $\mathbf{list}.x$ y $\mathbf{list}.y$, como por ejemplo en el caso típico del *merge* de dos listas ordenadas.[2] Para el correcto funcionamiento del programa, probablemente se debería establecer como precondición que las listas no comparten registros. Necesitaríamos un predicado auxiliar $\mathbf{reach}.i.j$ que

[1] En la representación gráfica de un estado de la memoria usamos rectángulos con uno o más campos para representar registros del *heap*. Los campos pueden contener valores que cuando no nos interesan se omiten. Una variable del *stack* se representa con su nombre. Cuando son punteros distintos de **nil**, tanto las variables como los valores de un registro, se grafican con una flecha que se dirige al registro del *heap* que apuntan o a la nada, si tienen una dirección de memoria que no está definida. Una flecha discontinua representa una cantidad arbitraria de registros enlazados.

[2] Usualmente las listas enlazadas representarían una secuencia de valores abstractos que para el caso de un programa de *merge* estarían ordenados. El algoritmo de *merge* generaría en la memoria una lista enlazada que represente la concatenación de ambas secuencias, pero con sus valores ordenados. Por el momento, no mencionamos estos valores abstractos para no desviar el foco de lo que se pretende justificar. Al final del capítulo 2 volvemos sobre el tema de representación de valores abstractos.



(a) Representación de **list**.x



(b) Caso con *sharing* entre **list**.x y **list**.y

Figura 1.1

establezca que la dirección de memoria j es alcanzable directa o indirectamente por la variable i. Pediríamos entonces como precondición que:

$$\mathbf{list}.x \land \mathbf{list}.y \land (\forall k \ \cdot \ \mathbf{reach}.x.k \land \mathbf{reach}.y.k \Rightarrow k = \mathbf{nil})$$

Esto evitaría que se presenten situaciones como la de la figura 1.1b. Más aún, supongamos que hay alguna otra lista **list**.z que no tiene nada que ver con las dos listas anteriores y por lo tanto debería permanecer invariante durante la ejecución de este programa. Para garantizar que efectivamente no sea afectada, deberíamos establecer como precondición:

$$\begin{aligned}\mathbf{list}.x \land \mathbf{list}.y \land \mathbf{list}.z \land (\forall k \ \cdot \ \mathbf{reach}.x.k \land \mathbf{reach}.y.k \Rightarrow k = \mathbf{nil}) \\ \land (\forall k \ \cdot \ \mathbf{reach}.z.k \land (\mathbf{reach}.x.k \lor \mathbf{reach}.y.k) \Rightarrow k = \mathbf{nil})\end{aligned}$$

Claramente se ve que, aún en situaciones elementales como la anterior donde todo tipo de *sharing* está prohibido, esta forma de razonamiento no escala. Una manera efectiva de solucionar este problema es con la introducción del operador de conjunción de separación $*$ (también llamado conjunción espacial), elemento clave de la *Separation Logic*. El predicado $p * q$ establece que los predicados p y q se satisfacen en porciones disjuntas de la memoria. De esta forma, cualquier celda efectivamente referenciada en p es necesariamente distinta a cualquier celda referenciada en q. Así es como las precondiciones de nuestro hipotético programa sobre las listas referenciadas por x, y (y z en el segundo caso) pueden simplemente expresarse como:

$$\mathbf{list}.x * \mathbf{list}.y \quad \text{y} \quad \mathbf{list}.x * \mathbf{list}.y * \mathbf{list}.z$$

sin necesidad de agregar predicados extra para evitar el *sharing*.

Además, el sistema de prueba de la *Separation Logic* permite verificar una terna $\{p\}\, c\, \{q\}$ mencionando en la precondición p únicamente aquella porción de la memoria manipulada por el comando c, denominada *footprint*. Luego, mediante una importante regla de *frame*, se puede extender su validez a un contexto de memoria más amplio dado por un invariante r, estableciendo la terna:

$$\{p * r\}\, c\, \{q * r\}$$

Esto vuelve completamente innecesario mencionar la lista referenciada por z en el caso anterior. La demostración del programa debería enfocarse solamente en su *footprint*, dado por $\mathbf{list}.x$ y $\mathbf{list}.y$. De esta manera, cuando se necesite mencionar a la lista $\mathbf{list}.z$ en un contexto más general, podremos usar la regla de *frame* para establecer su validez e invarianza durante la ejecución de la parte del programa que no la referencia. De la misma forma, para el ejemplo del comando de mutación mencionado anteriormente, podemos establecer la validez de la siguiente terna:

$$\{x \mapsto v * y \mapsto v\}\,[x] := w\,\{x \mapsto w * y \mapsto v\}$$

donde reemplazamos el operador de conjunción de la lógica clásica $\wedge$ por el operador de conjunción espacial $*$, garantizando ahora que x e y referencian a registros distintos de la memoria.

La posibilidad de que una lógica formal para el razonamiento sobre programas como la *Separation Logic*, aún considerando todos los beneficios expresivos y metodológicos que introduce, escale a sistemas reales, depende de la existencia de herramientas que faciliten su uso en la práctica. Dada la gran complejidad de las verificaciones de los programas que manejan dinámicamente la memoria, resulta conveniente poder llevarlas adelante automáticamente, aunque sea en un marco de aplicación reducido.

Un *shape analysis* es una forma de análisis estático de código para programas imperativos que usan memoria dinámica que trata de verificar automáticamente propiedades sobre estructuras de datos enlazadas, como listas, árboles, etc. Un *shape analysis* tiene como objetivo no solo reportar accesos a direcciones de memoria inválidas o *memory leaks*, sino también analizar la validez de propiedades no triviales sobre la *forma* de las estructuras dinámicas en memoria.

Existen ciertas clases de *shape analysis* que se basan en un tipo de ejecución simbólica del programa sobre estados abstractos especificados por fórmulas de un subconjunto restringido de la *Separation Logic*. Un estado abstracto se compone de un conjunto de fórmulas llamadas *symbolic heaps* y representa todos los estados concretos que satisfacen alguna de ellas. La ejecución simbólica remite a las ternas de Hoare en el sentido de que partiendo de un estado abstracto como precondición, genera un estado abstracto como postcondición.

Los *symbolic heaps* normalmente incluyen predicados para describir estructuras de datos lineales, posiblemente combinadas de formas complejas, como listas enlazadas, listas doblemente enlazadas, etc. En [9], se define un predicado $\mathbf{ls}(E, F)$ para representar caminos acíclicos en la memoria (o segmentos de listas según nuestra terminología de la sección 2.4.1) que comienzan con un registro apuntado por la variable E (que no puede ser **nil**) y terminan en la variable F. Con este predicado, los autores definen un análisis que les permite verificar propiedades sobre la memoria para distintos programas sobre listas, como por ejemplo algoritmos para revertir o destruir una lista.

En este trabajo presentamos una extensión del antes mencionado *shape analisys* a dominios que soportan estructuras de datos no lineales. Utilizamos un modelo de memoria con registros de tres campos, de los cuales dos se usan como punteros a otros registros, para así poder describir estructuras de datos como árboles binarios. Definimos un predicado $\mathbf{trees}.\mathcal{C}.\mathcal{D}$ donde tanto $\mathcal{C}$ como $\mathcal{D}$ son multiconjuntos de expresiones que, en este caso, pueden contener valores **nil**. Este predicado representa estructuras multiligadas con múltiples punteros de

entrada dados por los elementos de $\mathcal{C}$ y punteros de salida dados por los elementos de $\mathcal{D}$. A diferencia del predicado **ls**, que representa caminos acíclicos, nuestro predicado **trees** puede representar en su generalidad grafos dirigidos acíclicos y su importancia radica en que permite especificar las estructuras parciales que ocurren en los puntos intermedios de la ejecución de un programa. Nuestro análisis define una semántica ejecutable y con garantías de terminación que implementamos en Haskell y usamos para verificar, en lo que respecta a la estructura de memoria, una variedad de algoritmos sobre árboles. Nuestro análisis toma una precondición y genera automáticamente invariantes para los ciclos y una postcondición del programa.

El trabajo se organiza de la siguiente manera. En el capítulo 2 presentamos la *Separation Logic*, que usamos como marco teórico para este trabajo, y mostramos algunos ejemplos de verificaciones de programas. En el capítulo 3 presentamos el análisis propuesto que es, salvo diferencias menores, el resultado del trabajo [8], realizado en conjunto con Renato Cherini y Javier Blanco. En el capítulo 4, mostramos los resultados experimentales y discutimos la contribución de nuestro trabajo. Finalmente, en el apéndice mostramos algunas diferencias entre lo presentado aquí y el trabajo originalmente realizado en [8].

Capítulo 2

Separation Logic

En este capítulo se presenta el formalismo de la *Separation Logic* para razonar sobre programas imperativos con un manejo explícito de la memoria dinámica. Nos basamos en el trabajo [22], con la diferencia de que en él, Reynolds usa un modelo de estado de memoria de más bajo nivel, con direcciones de memoria dadas por los números enteros y con la posibilidad de realizar aritmética de punteros. Acá, sin embargo, presentamos una *Separation Logic* clásica, con un modelo de memoria estructurada con registros de diferente tamaño. Se presenta un lenguaje de programación imperativo, con comandos específicos para crear, acceder, modificar y liberar una porción de memoria. Se introduce el lenguaje de fórmulas, con el operador de conjunción espacial $*$ y con un operador de implicación espacial $-\!\!*$, relacionado a aquel. Se presentan reglas para el cálculo de fórmulas y para la verificación de especificaciones, redefiniendo la validez de una terna de Hoare para exigir también que el programa que se ejecuta no termine anormalmente por un fallo de memoria. Finalmente, se muestran algunos ejemplos de aplicación y derivaciones de programas.

2.1. El lenguaje de programación

En la *Separation Logic*, los estados sobre los que corren los programas se componen de dos partes: un *stack* que asigna valores a las variables y un *heap* que asigna a las direcciones de memoria dinámica definidas, un registro de tamaño arbitrario. Los valores de las variables y de los campos de los registros pueden ser valores elementales o direcciones de memoria. Cuando una variable o un campo de un registro tiene un valor que resulta ser una dirección de memoria, decimos que es un *puntero*. Si esa dirección de memoria está definida en el *heap* decimos que es *válida*, y en caso contrario que es *inválida*. El valor distinguido **nil** representa una dirección de memoria inválida y decimos que un puntero con una dirección de memoria inválida que no sea **nil** es *dangling*.

Definición 1 *Dado un conjunto de valores elementales* Atoms*, un conjunto infinito de direcciones de memoria* Locations*, con un elemento particular* **nil** $\in$

Locations, y un conjunto Var de variables, definimos los estados States como:

$$\begin{aligned}
&\mathsf{Values} \doteq \mathsf{Atoms} \cup \mathsf{Locations} \qquad \text{con } \mathsf{Atoms} \cap \mathsf{Locations} = \emptyset \\
&\mathsf{Stacks} \doteq \mathsf{Var} \to \mathsf{Values} \\
&\mathsf{Heaps} \doteq (\mathsf{Locations} - \{\mathbf{nil}\}) \rightharpoonup_f \mathsf{Values}^+ \\
&\mathsf{States} \doteq \mathsf{Stacks} \times \mathsf{Heaps}
\end{aligned}$$

Notación: $A \rightharpoonup_f B$ es el conjunto de funciones parciales de A a B con dominio finito y A^+ es el conjunto de tuplas sobre A de tamaño arbitrario

Definición 2 *La sintaxis de las expresiones Expr y las expresiones booleanas BExp están dadas por las siguientes gramáticas:*

$$\begin{aligned}
\mathsf{Expr} \quad & e ::= x \mid \mathbf{nil} \mid \ldots \\
\mathsf{BExp} \quad & b ::= e = e \mid \neg b \mid b \vee b \mid b \wedge b \mid \ldots
\end{aligned}$$

donde $x \in$ Var

Las expresiones Exp y las expresiones booleanas BExp dependen de los valores elementales Atoms, son completamente estándar, y por lo tanto no damos una sintaxis, ni una semántica completa. Por ejemplo, si el conjunto Atoms fuera el conjunto de los números naturales, en las expresiones se podrían incluir los números 1, 2, etc y operaciones como la suma o la multiplicación de expresiones. En la expresiones booleanas se incluirían relaciones de orden como $<$ o $\leq$ y posiblemente otros operadores de la lógica proposicional. La única restricción es que la semántica sólo dependa del *stack*.

Definición 3 *La semántica de las expresiones Expr y las expresiones booleanas BExp están dadas por las funciones:*

$$\begin{aligned}
&[\![\ ]\!]_e \in \mathsf{Expr} \to \mathsf{Stacks} \to \mathsf{Values} \\
&[\![\ ]\!]_b \in \mathsf{BExp} \to \mathsf{Stacks} \to \{true, false\}
\end{aligned}$$

definidas de la manera estándar:

$$\begin{aligned}
[\![x]\!]_e.s &\doteq s.x \\
[\![\mathbf{nil}]\!]_e.s &\doteq \mathbf{nil} \\
&\ldots \\
[\![e_1 = e_2]\!]_b.s &\doteq [\![e_1]\!]_e.s = [\![e_2]\!]_e.s \\
[\![\neg b_1]\!]_b.s &\doteq \ no\ [\![b_1]\!]_b.s \\
[\![b_1 \vee b_2]\!]_b.s &\doteq [\![b_1]\!]_b.s \ \acute{o}\ [\![b_2]\!]_b.s \\
[\![b_1 \wedge b_2]\!]_b.s &\doteq [\![b_1]\!]_b.s \ y\ [\![b_2]\!]_b.s \\
&\ldots
\end{aligned}$$

donde $x \in$ Var, $e_1, e_2 \in$ Expr, $b_1, b_2 \in$ BExp y $s \in$ Stacks.

Se extiende el lenguaje de programación imperativo simple, originalmente axiomatizado por Hoare [13], con comandos que manipulan la memoria dinámica. Notar que las condiciones en los comandos de alternativa y de ciclo son expresiones booleanas BExp, por lo que no dependen del *heap* y no generan efectos secundarios.

Definición 4 *Gramática de comandos* Comm*:*

$$\begin{array}{lll} \mathsf{Comm} & c ::= \mathbf{skip} \mid x := e \mid c; c \mid \mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else}\ c\ \mathbf{fi} \mid \mathbf{while}\ b\ \mathbf{do}\ c\ \mathbf{od} & \\ & \mid x := \mathbf{new}(\vec{e}) & \textit{(Construcción)} \\ & \mid x := x.i & \textit{(Consulta)} \\ & \mid x.i := e & \textit{(Mutación)} \\ & \mid \mathbf{free}(x) & \textit{(Destrucción)} \end{array}$$

donde $x \in \mathsf{Var}$, $e \in \mathsf{Expr}$, $b \in \mathsf{BExp}$, $\vec{e} \in \mathsf{Expr}^+$, $i \in \mathbb{N}_0$

El comando de construcción $x := \mathbf{new}(e_0, ..., e_{n-1})$ crea un nuevo registro de tamaño n en el *heap* con los valores de los campos dados por las expresiones $e_0, ..., e_{n-1}$ y asigna al puntero x su dirección de memoria. El comando de consulta $x := y.i$ asigna a la variable x el valor dado por el i-ésimo campo del registro apuntado por y. El comando de mutación $x.i := e$ modifica el valor del i-ésimo campo del registro apuntado por x con el valor de la expresión e. Por último, el comando de destrucción $\mathbf{free}(x)$ elimina el registro de memoria apuntado por x y por lo tanto lo convierte en *dangling*.

Para formalizar el significado de los comandos usamos una semántica operacional *small step*, mediante una relación de transición entre configuraciones. Si bien en el lenguaje imperativo simple los comandos siempre se pueden ejecutar exitosamente, los comandos de manipulación del *heap* pueden fallar. Esto sucede cuando se pretende acceder a la dirección de memoria de un puntero inválido o a un campo inexistente de un registro del *heap*. En este caso decimos que el programa termina anormalmente con un fallo de memoria y lo denotamos con la configuración **abort**.

Definición 5 *Definimos el conjunto de las configuraciones* Conf *como:*

$$\mathsf{Conf} \doteq (\mathsf{States} \times \mathsf{Comm}) \cup \mathsf{States} \cup \{\mathbf{abort}\}$$

Una configuración $\zeta \in \mathsf{Conf}$ *puede ser:*

- *no terminal, cuando* $\zeta \in \mathsf{States} \times \mathsf{Comm}$
- *terminal normal, cuando* $\zeta \in \mathsf{States}$
- *terminal anormal, cuando* $\zeta \in \{\mathbf{abort}\}$

Definición 6 *La semántica operacional de los comandos se define con la relación de transición* $\rightsquigarrow \subseteq \mathsf{Conf} \times \mathsf{Conf}$*, según las siguientes reglas:*

Skip:

$$(s, h),\ \mathbf{skip} \ \rightsquigarrow\ (s, h)$$

Asignación:

$$(s, h),\ x := e_1 \ \rightsquigarrow\ (s|x \to [\![e_1]\!]_e.s, h)$$

Construcción:

$$\frac{l \notin dom.h}{(s,h),\ x := \mathbf{new}(e_0,...,e_{n-1}) \ \rightsquigarrow\ (s|x \to l, h|l \to (\llbracket e_0 \rrbracket_e.s, ..., \llbracket e_{n-1} \rrbracket_e.s))}$$

Notar que l se elige de forma no determinística

Consulta:

$$\frac{\llbracket y \rrbracket_e.s = l \quad h.l = (v_0,...,v_{n-1}) \quad 0 \leq i < n}{(s,h),\ x := y.i \ \rightsquigarrow\ (s|x \to v_i, h)}$$

$$\frac{\llbracket y \rrbracket_e.s = l \quad h.l = (v_0,...,v_{n-1}) \quad i \geq n}{(s,h),\ x := y.i \ \rightsquigarrow\ \mathbf{abort}}$$

$$\frac{\llbracket y \rrbracket_e.s \notin dom.h}{(s,h),\ x := y.i \ \rightsquigarrow\ \mathbf{abort}}$$

Mutación:

$$\frac{\llbracket y \rrbracket_e.s = l \quad \llbracket e_1 \rrbracket_e.s = v \quad h.l = (v_0,...,v_{n-1}) \quad 0 \leq i < n}{(s,h),\ y.i := e_1 \ \rightsquigarrow\ (s, h|l \to (v_0,...,v_{i-1}, v, v_{i+1},...,v_{n-1}))}$$

$$\frac{\llbracket y \rrbracket_e.s = l \quad h.l = (v_0,...,v_{n-1}) \quad i \geq n}{(s,h),\ y.i := e_1 \ \rightsquigarrow\ \mathbf{abort}}$$

$$\frac{\llbracket y \rrbracket_e.s \notin dom.h}{(s,h),\ y.i := e_1 \ \rightsquigarrow\ \mathbf{abort}}$$

Destrucción:

$$\frac{\llbracket x \rrbracket_e.s = l \quad l \in dom.h}{(s,h),\ \mathbf{free}(x) \ \rightsquigarrow\ (s, h - l)}$$

$$\frac{\llbracket x \rrbracket_e.s \notin dom.h}{(s,h),\ \mathbf{free}(x) \ \rightsquigarrow\ \mathbf{abort}}$$

Secuenciación:

$$\frac{(s,h),\ c_1 \ \rightsquigarrow\ (s',h'),\ c_1'}{(s,h),\ c_1; c_2 \ \rightsquigarrow\ (s',h'),\ c_1'; c_2}$$

$$\frac{(s,h),\ c_1 \ \rightsquigarrow\ (s',h')}{(s,h),\ c_1; c_2 \ \rightsquigarrow\ (s',h'),\ c_2}$$

$$\frac{(s,h),\ c_1 \ \rightsquigarrow\ \mathbf{abort}}{(s,h),\ c_1; c_2 \ \rightsquigarrow\ \mathbf{abort}}$$

Alternativa:

$$\frac{\llbracket b_1 \rrbracket_b}{(s,h),\ \mathbf{if}\ b_1\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{fi} \ \rightsquigarrow\ (s,h),\ c_1}$$

$$\frac{no\ \llbracket b_1 \rrbracket_b}{(s,h),\ \mathbf{if}\ b_1\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{fi} \ \rightsquigarrow\ (s,h),\ c_2}$$

Ciclo:

$$\frac{[\![b_1]\!]_b}{(s,h),\ \textbf{while}\ b_1\ \textbf{do}\ c\ \textbf{od}\ \ \rightsquigarrow\ \ (s,h),\ c;\textbf{while}\ b_1\ \textbf{do}\ c\ \textbf{od}}$$

$$\frac{no\ [\![b_1]\!]_b}{(s,h),\ \textbf{while}\ b_1\ \textbf{do}\ c\ \textbf{od}\ \ \rightsquigarrow\ \ (s,h)}$$

Notación: Con $dom.f$ denotamos el dominio de la función f, con $f|x \to y$ denotamos a la función f donde se actualizó el valor de la variable x con el valor y. $f - x$ denota la función f restringida en su dominio a los valores que no son x

Con la relación $\rightsquigarrow^*$ representamos secuencias de una o más transiciones dadas por $\rightsquigarrow$.

Definición 7 *Definimos la relación $\rightsquigarrow^* \subseteq \mathsf{Conf} \times \mathsf{Conf}$ como la clausura transitiva de $\rightsquigarrow$. Es decir, para todos $\zeta_1, \zeta_2, \zeta_3 \in \mathsf{Conf}$:*

$$\frac{\zeta_1\ \rightsquigarrow\ \zeta_2}{\zeta_1\ \rightsquigarrow^*\ \zeta_2}$$

$$\frac{\zeta_1\ \rightsquigarrow^*\ \zeta_2\quad \zeta_2\ \rightsquigarrow^*\ \zeta_3}{\zeta_1\ \rightsquigarrow^*\ \zeta_3}$$

2.2. Fórmulas

El lenguaje de fórmulas resulta de extender las fórmulas usuales de la lógica de primer orden (con expresiones booleanas y cuantificadores) para incluir aserciones referidas a la memoria.

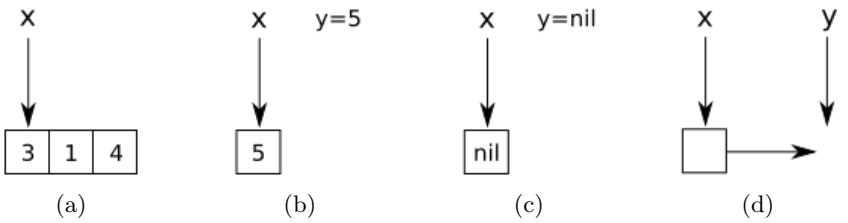
Definición 8 *El conjunto de fórmulas Form está definida por la siguiente gramática:*

$$\begin{array}{lll}
\mathsf{Form} & p ::= b \mid p \vee p \mid \neg p \mid \forall x\ \cdot\ p & \\
& \mid \textbf{emp} & \textit{(Heap vacío)} \\
& \mid x \mapsto \vec{e} & \textit{(Heap} \text{ singleton}\textit{)} \\
& \mid p * p & \textit{(Conjunción espacial)} \\
& \mid p \mathrel{-\!\!*} p & \textit{(Implicación espacial)}
\end{array}$$

donde $b \in \mathsf{BExp}$, $x \in \mathsf{Var}$, $\vec{e} \in \mathsf{Expr}^+$

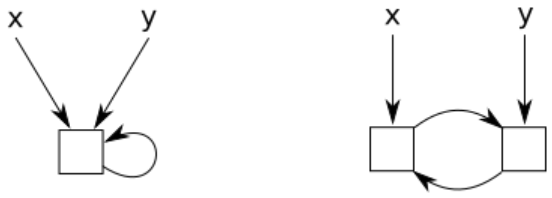
Notar que definimos un conjunto minimal de fórmulas del cálculo de predicados. Los otros operadores y cuantificador ($\wedge$, $\Rightarrow$, $\exists$, etc) se derivan de los primeros. Cuando no nos interese el nombre de una variable cuantificada, usaremos $-$, como en $x \mapsto -$, que significa $\exists y\ \cdot\ x \mapsto y$.

A diferencia de las expresiones y las expresiones booleanas, las fórmulas dependen tanto del *stack* como del *heap*. El predicado **emp** se satisface en los estados con el *heap* vacío. En estos casos, no hay registros en la memoria dinámica y por lo tanto, tampoco direcciones de memoria válidas. Por otro lado, el predicado $x \mapsto \vec{e}$ se satisface en un *heap* que contiene solamente una dirección de memoria válida dada por el valor de x, con un registro con campos dados por los valores de $\vec{e}$, según el *stack*.



(a) (b) (c) (d)

Figura 2.1: Estados de $x \mapsto 3, 1, 4$ (a) y de $x \mapsto y$ (b, c y d)



(a) (b)

Figura 2.2: Estados de $x \mapsto y \wedge y \mapsto x$ (a) y de $x \mapsto y * y \mapsto x$ (b)

Ejemplo 9 *La fórmula* $x \mapsto 3, 1, 4$ *se satisface en un* heap *que contiene un único registro (3,1,4) cuya dirección de memoria no está explícita pero cumple que es igual al valor dado por el* stack *a la variable* x *(fig. 2.1a).*

Ejemplo 10 *La fórmula* $x \mapsto y$ *se satisface cuando el puntero* x *referencia a un registro de un solo campo con el valor dado por el* stack *a la variable* y*. Este valor podría ser un valor elemental (fig. 2.1b) o un puntero, que a su vez podría ser* **nil** *(fig. 2.1c), o un puntero* dangling *(fig. 2.1d) o una dirección de memoria válida. En este último caso, no hay otra opción más que, que tanto* x *como* y *referencien al mismo (y único) registro del* heap *y por lo tanto tengan el mismo valor. De esta forma, el estado también satisfaría la fórmula* $y \mapsto x$ *(fig. 2.2a).*

El operador de conjunción espacial (o conjunción de separación) $*$ permite unir *heaps* disjuntos. El predicado $p * q$ se satisface en un *heap* que se pueda dividir en dos *subheaps* disjuntos tales que p se satisface en uno y q se satisface en el otro.

Ejemplo 11 *La fórmula* $x \mapsto 3, 1, 4 *$ **emp** *se satisface en un estado como el de la figura 2.1a. Notar que* **emp** *es entonces el elemento neutro de la conjunción espacial* $*$*.*

Los operadores de la lógica de primer orden clásica mantienen su semántica estándar. Es interesante comparar el comportamiento de la conjunción de la lógica clásica $\wedge$ con el de la conjunción espacial $*$.

Ejemplo 12 *La fórmula* $x \mapsto 7 * y \mapsto 7$ *se cumple en un* heap *que contiene dos registros (7) en las direcciones de memoria dadas por los valores en el* stack *de* x *e* y*. Notar que, por más que el valor de los registros sea el mismo, la direcciones de memoria de* x *e* y *deben ser necesariamente distintas, ya que el operador* $*$ *exige la separación del* heap *en dos* subheaps *disjuntos.*

Ejemplo 13 *La fórmula $x \mapsto y \wedge z \mapsto w$ se satisface en un modelo que satisfaga tanto $x \mapsto y$ como $z \mapsto w$, por lo que debe tener un* heap *con una única dirección de memoria válida apuntando a un único registro. Entonces los valores dados por el* stack *a x y z deben ser iguales, de la misma manera que los valores de y y w.*

Ejemplo 14 *De los dos ejemplos anteriores podemos inferir que las fórmulas $x \mapsto 7 * x \mapsto 7$ y $x \mapsto 5 \wedge z \mapsto 9$ son ambas insatisfactibles. En la figura 2.2 podemos ver cómo son los estados que satisfacen la conjunción clásica y la conjunción espacial de los predicados $x \mapsto y$ e $y \mapsto x$.*

El operador de implicación espacial $-\!\!*$ tiene una semántica difícil de entender intuitivamente. El predicado $p -\!\!* q$ se satisface en un *heap* siempre que cualquier *heap* disjunto que satisfaga p se pueda extender con el *heap* original de manera que satisfaga q.

Ejemplo 15 *Supongamos que la fórmula p se satisface en un estado cuyo* stack *le asigna a la variable x una dirección de memoria l a la que el* heap *le asigna un registro con un único campo con valor* 11*. Entonces la fórmula $(x \mapsto 11) -\!\!* p$ se satisface en el mismo estado pero con el* heap *que resulta de eliminar l de su dominio. Más aún, la fórmula $x \mapsto 15 * ((x \mapsto 11) -\!\!* p)$ se satisface en el mismo estado pero en cuyo* heap *se cambió de* 11 *a* 15 *el valor del campo del registro asignado a l.*

A continuación damos la semántica formal de las fórmulas, según los estados que la satisfacen.

Definición 16 *La semántica de las fórmulas* Form *está dada por la relación de satisfacción $\models$:*

$$\models \subseteq \mathit{States} \times \mathit{Form}$$

definida de la siguiente manera:

$(s,h) \models b_1$	*si y sólo si* $[\![b_1]\!]_b.s$
$(s,h) \models p \vee q$	*si y sólo si* $(s,h) \models p$ *o* $(s,h) \models q$
$(s,h) \models \neg p$	*si y sólo si no se cumple* $(s,h) \models p$
$(s,h) \models \forall x \cdot p$	*si y sólo si para todo* $v \in \mathit{Values}$, $(s\|x \to v, h) \models p$
$(s,h) \models \mathbf{emp}$	*si y sólo si* $h = \emptyset$
$(s,h) \models x \mapsto e_0, ..., e_{n-1}$	*si y sólo si* $h = \{([\![x]\!]_e.s, ([\![e_0]\!]_e.s, ..., [\![e_{n-1}]\!]_e.s))\}$
$(s,h) \models p * q$	*si y sólo si existen* $h_1, h_2 \in \mathit{Heaps}$ *tales que* $h_1 \perp h_2$, $h = h_1 \cup h_2, (s,h_1) \models p$ *y* $(s,h_2) \models q$
$(s,h) \models p -\!\!* q$	*si y sólo si para todo* $h_1 \in \mathit{Heaps}$ *tal que* $h_1 \perp h$, *vale que* $(s,h_1) \models p$ *implica* $(s,(h \cup h_1)) \models q$

donde $f \perp g$ denota que las funciones f y g tienen dominios disjuntos.

Además, decimos que una fórmula $p \in$ Form es válida *si para todo estado $(s,h) \in$ States se cumple que $(s,h) \models p$. Lo denotamos simplemente con $\models p$ (omitiendo el estado).*

Presentamos reglas de inferencia para el cálculo de fórmulas. Las reglas de la lógica de primer orden clásica permanecen válidas para los operadores heredados. Además:

Lema 17 *Las siguientes son reglas de cálculo válidas de la Separation Logic:*
Conmutatividad:

$$\models p_1 * p_2 \Leftrightarrow p_2 * p_1$$

Asociatividad:

$$\models (p_1 * p_2) * p_3 \Leftrightarrow p_1 * (p_2 * p_3)$$

Elemento neutro:

$$\models p * \mathbf{emp} \Leftrightarrow p$$

Distributividad con $\vee$:

$$\models (p_1 \vee p_2) * q \Leftrightarrow (p_1 * q) \vee (p_2 * q)$$

Semidistributividad con $\wedge$:

$$\models (p_1 \wedge p_2) * q \Rightarrow (p_1 * q) \wedge (p_2 * q)$$

Distributividad con $\exists$:

$$\frac{x \notin FV(q)}{\models (\exists x \ . \ p) * q \Leftrightarrow (\exists x \ . \ p * q)}$$

Semidistributividad con $\forall$:

$$\frac{x \notin FV(q)}{\models (\forall x \ . \ p) * q \Rightarrow (\forall x \ . \ p * q)}$$

Monotonía:

$$\frac{\models p_1 \Rightarrow q_1 \quad \models p_2 \Rightarrow q_2}{\models p_1 * p_2 \Rightarrow q_1 * q_2}$$

Adjuntividad:

$$\models (p_1 \Rightarrow (p_2 \mathrel{-\!\!*} p_3)) \Leftrightarrow (p_1 * p_2 \Rightarrow p_3)$$

donde $p, p_1, p_2, p_3, q, q_1, q_2 \in \mathsf{Form}$. *Con* $FV(q)$ *denotamos el conjunto de las variables libres de* q.

Se pueden validar otras reglas referidas a ciertos tipos de fórmulas especiales. Una fórmula es *pure* si no depende del *heap*. Existe una caracterización sintáctica para las fórmulas *pure*: son aquellas que no contienen **emp**, ni $\mapsto$.

Definición 18 *Decimos que una fórmula* $p \in \mathsf{Form}$ *es* pure *si para todos* $s \in \mathsf{Stacks}, h_1, h_2 \in \mathsf{Heaps}$ *se cumple que:*

$$s, h_1 \models p \text{ si y sólo si } s, h_2 \models p$$

Ejemplo 19 *La fórmula* $x = \mathbf{nil}$ *es* pure. *Es independiente del* heap, *ya que sólo impone una condición sobre el* stack s *que debe cumplir que* $s.x = \mathbf{nil}$.

Lema 20 *Las siguientes son reglas de cálculo válidas de la Separation Logic:*

$$\frac{p_1 \ es \ \text{pure} \ o \ p_2 \ es \ \text{pure}}{\models p_1 \wedge p_2 \Rightarrow p_1 * p_2}$$

$$\frac{p_1 \ y \ p_2 \ son \ \text{pure}}{\models p_1 * p_2 \Rightarrow p_1 \wedge p_2}$$

$$\frac{p_1 \ es \ \text{pure}}{\models (p_1 \wedge p_2) * p_3 \Leftrightarrow p_1 \wedge (p_2 * p_3)}$$

$$\frac{p_1 \ es \ \text{pure}}{\models (p_1 \mathrel{-\!\!*} p_2) \Rightarrow (p_1 \Rightarrow p_2)}$$

$$\frac{p_1 \ y \ p_2 \ son \ \text{pure}}{\models (p_1 \Rightarrow p_2) \Rightarrow (p_1 \mathrel{-\!\!*} p_2)}$$

Otra clase importante de fórmulas son las *domain-exact*, que caracterizan el dominio de los *heaps* que la satisfacen.

Definición 21 *Decimos que una fórmula* $p \in \mathsf{Form}$ *es* domain-exact *si para todos* $s \in \mathsf{Stacks}, h_1, h_2 \in \mathsf{Heaps}$ *se cumple que:*

$$s, h_1 \models p \ y \ s, h_2 \models p \ implican \ que \ dom.h_1 = dom.h_2$$

Ejemplo 22 *La fórmula* $x \mapsto y * y \mapsto \mathbf{nil}$ *es* domain-exact *pues dado un* stack s*, el* heap *queda completamente determinado:* $\{(s.x, (s.y)), (s.y, (\mathbf{nil}))\}$*. Además, la fórmula más débil* $x \mapsto - * y \mapsto -$ *también es* domain-exact *ya que, si bien los valores de los registros del* heap *no están determinados, su dominio sí lo está:* $\{s.x, s.y\}$*.*

Para las fórmulas *domain-exact*, las reglas de semidistributividad del lema 17 se vuelven completas.

Lema 23 *Las siguientes son reglas de cálculo válidas de la Separation Logic:*

Distributividad con $\wedge$:

$$\frac{q \ es \ domain\text{-}exact}{\models (p_1 \wedge p_2) * q \Leftrightarrow (p_1 * q) \wedge (p_2 * q)}$$

Distributividad con $\forall$:

$$\frac{q \ es \ domain\text{-}exact \quad x \notin FV(q)}{\models (\forall x \ . \ p) * q \Leftrightarrow (\forall x \ . \ p * q)}$$

2.3. Especificaciones

Para razonar sobre programas usamos el concepto de *ternas de Hoare* en la especificación de los comandos con la *Separation Logic* como lenguaje de aserciones. Usamos una interpretación de corrección parcial que asegura que los comandos no abortan, pero que no impone requerimientos sobre la terminación de los mismos.

Definición 24 *El conjunto de especificaciones* Spec *está definido por la siguiente gramática:*

$$\mathsf{Spec} \qquad s ::= \vdash \{p\}\, c\, \{p\}$$

donde $p \in \mathsf{Form}$ *y* $c \in \mathsf{Comm}$

Definición 25 *Decimos que una especificación* $\vdash \{p\}\, c\, \{q\}$ *es válida si para todos estados* $(s,h),(s',h') \in \mathsf{States}$ *se cumple que*

$$\begin{array}{ll} s,h \models p \text{ implica que} & \neg((s,h), c \rightsquigarrow^* \mathbf{abort}) \\ & \text{y si } (s,h), c \rightsquigarrow^* (s',h') \text{ entonces } s',h' \models q \end{array}$$

Las fórmulas p *y* q *se denominan* precondición *y* postcondición *del comando* c *respectivamente.*

Notar que las especificaciones están implícitamente cuantificadas sobre los estados y, como el comando de construcción es no determinístico, también sobre todas las posibles ejecuciones. Además, cualquier ejecución que resulte en un fallo de memoria falsifica la validez de la especificación. Por lo tanto, una especificación como $\vdash \{p\}\, c\, \{\mathbf{true}\}$ no es válida en general para cualquier comando c. Pero si logramos establecer su validez, sabremos que el comando c no puede generar fallos de memoria cuando se ejecuta a partir de cualquier estado que satisfaga p.

Las reglas de inferencia de la lógica de *Hoare* para los comandos simples siguen siendo válidas para nuestro marco.

Definición 26 *Reglas de inferencia de especificaciones para los comandos del lenguaje imperativo simple*

Skip:

$$\vdash \{p\}\, \mathbf{skip}\, \{p\}$$

Asignación:

$$\vdash \{p_{/x \leftarrow e}\}\, x := e\, \{p\}$$

Secuenciación:

$$\frac{\vdash \{p\}\, c_1\, \{q\} \quad \vdash \{q\}\, c_2\, \{r\}}{\vdash \{p\}\, c_1; c_2\, \{r\}}$$

Alternativa:

$$\frac{\vdash \{p \wedge b\}\, c_1\, \{q\} \quad \vdash \{p \wedge \neg b\}\, c_2\, \{q\}}{\vdash \{p\}\, \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{fi}\, \{q\}}$$

Ciclo:

$$\frac{\vdash \{p \wedge b\}\, c\, \{p\}}{\vdash \{p\}\, \mathbf{while}\ b\ \mathbf{do}\ c\ \mathbf{od}\ \{p \wedge \neg b\}}$$

donde $p,q,r \in \mathsf{Form}$, $x \in \mathsf{Var}$, $e \in \mathsf{Expr}$, $b \in \mathsf{BExp}$, $c, c_1, c_2 \in \mathsf{Comm}$. *La fórmula* p *en la regla de ciclo se denomina* invariante de ciclo

Notación: $p_{/x \leftarrow e}$ *es* p *con las ocurrencias libres de la variable* x *sustituidas por* e.

Los comandos de manipulación de la memoria pueden caracterizarse por especificaciones *locales*, que incluyen solamente las variables y las partes del *heap* que son creadas, accedidas, mutadas o destruidas por el comando. Esta caracterización mínima se denomina el *footprint* del comando.

Definición 27 *Reglas de inferencia de especificaciones para los comandos de manipulación de la memoria:*

Cronstrucción (local):

$$\vdash \{\mathbf{emp}\}\, x := \mathbf{new}(\vec{e})\, \{\exists x' \;\cdot\; x \mapsto \vec{e}_{/x \leftarrow x'}\}$$

Consulta (local):

$$\frac{\vec{e} = (e_0, ..., e_{n-1}) \qquad 0 \leq i < n}{\vdash \{y \mapsto \vec{e}\}\, x := y.i\, \{\exists x' \;\cdot\; (y \mapsto \vec{e})_{/x \leftarrow x'} \wedge x = e_{i\,/x \leftarrow x'}\}}$$

Mutación (local):

$$\frac{0 \leq i < n}{\vdash \{y \mapsto (e_0, ..., e_{n-1})\}\, y.i := e\, \{y \mapsto (e_0, ..., e_{i-1}, e, e_{i+1}, ..., e_{n-1})\}}$$

Destrucción (local):

$$\vdash \{x \mapsto \vec{e}\}\, \mathbf{free}(x)\, \{\mathbf{emp}\}$$

La mayoría de las reglas estructurales también siguen siendo válidas, con la excepción de la regla de constancia:

$$\frac{\vdash \{p\}\, c\, \{q\} \quad FV(r) \cap Mod(c) = \emptyset}{\vdash \{p \wedge r\}\, c\, \{q \wedge r\}}$$

Notación: $Mod(c)$ es el conjunto de las variables modificadas por c.

En el contexto del lenguaje imperativo simple de *Hoare*, esta regla es importante para la escalabilidad ya que permite extender especificaciones locales sobre un comando, agregando aserciones sobre variables que no se modifican por dicho comando y preservando su *constancia* en la ejecución. Lamentablemente, pierde su validez en el contexto de la Separation Logic ya que no evita la presencia de *aliasing*, como en el siguiente caso:

$$\frac{\vdash \{x \mapsto -\}\, x.0 := 3\, \{x \mapsto 3\}}{\vdash \{x \mapsto - \wedge y \mapsto 7\}\, x.0 := 3\, \{x \mapsto 3 \wedge y \mapsto 7\}}$$

La precondición de la conclusión garantiza que x e y estén apuntando a la misma dirección de memoria y la postcondición es insatisfactible.

La validez de la regla se puede asegurar si pedimos que la aserción r sea pura, aunque su importancia para extender especificaciones locales sea limitada en este nuevo contexto.

Definición 28 *Reglas estructurales sobre especificaciones:*

Consecuencia:

$$\frac{\models p' \Rightarrow p \quad \vdash \{p\}\, c\, \{q\} \quad \models q \Rightarrow q'}{\vdash \{p'\}\, c\, \{q'\}}$$

Eliminación de variables auxiliares:

$$\frac{\vdash \{p\}\, c\, \{q\} \quad x \notin FV(c)}{\vdash \{\exists x \,.\, p\}\, c\, \{\exists x \,.\, q\}}$$

Sustitución:

$$\frac{\vdash \{p\}\, c\, \{q\} \quad x' \notin FV(p) \cup FV(c) \cup FV(q)}{\vdash (\{p\}\, c\, \{q\})_{/x \leftarrow x'}}$$

Constancia pura:

$$\frac{\vdash \{p\}\, c\, \{q\} \quad r \ es \ \text{pure} \quad FV(r) \cap Mod(c) = \emptyset}{\vdash \{p \wedge r\}\, c\, \{q \wedge r\}}$$

Las especificaciones locales presentadas en la definición 27 cobran sentido cuando podemos extender su aplicación a un contexto más general. La habilidad de extender especificaciones locales se puede recuperar usando la conjunción espacial, en lugar de la conjunción de la lógica de predicados clásica. De esta forma, podemos trasladar el razonamiento sobre un comando enfocado en su *footprint* a estados más generales, agregando predicados sobre variables y partes del *heap* que no son modificadas por el comando.

Definición 29 Regla de *frame*:

$$\frac{\vdash \{p\}\, c\, \{q\} \quad FV(r) \cap Mod(c) = \emptyset}{\vdash \{p * r\}\, c\, \{q * r\}}$$

Eliminando la restricción impuesta por la condición de no interferencia entre las variables modificadas por c y las variables libres en v, podemos obtener la siguiente:

Definición 30 Regla de *frame* interferente:

$$\frac{\vdash \{p\}\, c\, \{q\} \quad \vec{x} = Mod(c) \quad \vec{x}' \cap (FV(c) \cup FV(p) \cup FV(q) \cup FV(r)) = \emptyset}{\vdash \{p * r\}\, c\, \{\exists \vec{x}' \,.\, q * r_{/\vec{x} \leftarrow \vec{x}'}\}}$$

donde, abusando de la notación, vemos a $\vec{x}$ como una lista de variables y como un conjunto. $\vec{x}'$ es una lista de variables frescas y $r_{/\vec{x} \leftarrow \vec{x}'}$ denota la sustitución de cada variable de $\vec{x}$ por su correspondiente en $\vec{x}'$, en la fórmula r.

La regla de *frame* nos permite extender las reglas de especificaciones locales de los comandos de manipulación de la memoria dinámica para obtener especificaciones globales.

Definición 31 *Reglas de inferencia de especificaciones para los comandos de manipulación de la memoria:*

Construcción (global):

$$\vdash \{p\}\, x := \mathbf{new}(\vec{e})\, \{\exists x' \,.\, x \mapsto \vec{e}_{/x \leftarrow x'} * p_{/x \leftarrow x'}\}$$

Consulta (global):

$$\frac{\vec{e} = (e_0, ..., e_{n-1}) \qquad 0 \leq i < n}{\vdash \{y \mapsto \vec{e} * p\}\, x := y.i\, \{\exists x' \,.\, (y \mapsto \vec{e} * p)_{/x \leftarrow x'} \wedge x = e_{i\,/x \leftarrow x'}\}}$$

Mutación (global):

$$\frac{0 \leq i < n}{\vdash \{y \mapsto (e_0, ..., e_{n-1}) * p\}\, y.i := e\, \{y \mapsto (e_0, ..., e_{i-1}, e, e_{i+1}, ..., e_{n-1}) * p\}}$$

Destrucción (global):

$$\vdash \{x \mapsto \vec{e} * p\}\, \mathbf{free}(x)\, \{p\}$$

Ejemplo 32 *Demostraremos un programa que construye un estado como el de la figura 2.2b (pág. 20). De la regla de especificación local del comando de construcción y de la regla de consecuencia para la eliminación del* $\exists$ *en la postcondición, se derivan las siguientes ternas:*

$$\vdash \{\mathbf{emp}\}\, x := \mathbf{new}(\mathbf{nil})\, \{x \mapsto \mathbf{nil}\}$$

$$\vdash \{\mathbf{emp}\}\, y := \mathbf{new}(x)\, \{y \mapsto x\}$$

Por la regla de especificación local del comando de mutación tenemos:

$$\vdash \{x \mapsto \mathbf{nil}\}\, x.0 := y\, \{x \mapsto y\}$$

Aplicando la regla de frame *con* $\mathbf{emp}$, $x \mapsto \mathbf{nil}$ *e* $y \mapsto x$ *respectivamente en las ternas anteriores y usando la regla de consecuencia y elemento neutro para eliminar un* $\mathbf{emp}$ *de la precondición en la primera terna, nos queda:*

$$\vdash \{\mathbf{emp}\}\, x := \mathbf{new}(\mathbf{nil})\, \{x \mapsto \mathbf{nil} * \mathbf{emp}\}$$

$$\vdash \{x \mapsto \mathbf{nil} * \mathbf{emp}\}\, y := \mathbf{new}(x)\, \{x \mapsto \mathbf{nil} * y \mapsto x\}$$

$$\vdash \{x \mapsto \mathbf{nil} * y \mapsto x\}\, x.0 := y\, \{x \mapsto y * y \mapsto x\}$$

Finalmente, aplicando dos veces la regla de secuenciación, obtenemos:

$$\vdash \begin{array}{l} \{\mathbf{emp}\} \\ \quad x := \mathbf{new}(\mathbf{nil}); \\ \quad y := \mathbf{new}(x); \\ \quad x.0 := y \\ \{x \mapsto y * y \mapsto x\} \end{array}$$

Teorema 33 *Si* $\vdash \{p\}\, c\, \{q\}$ *se deduce utilizando las reglas del sistema deductivo dadas por las definiciones 26, 27, 28, 29, 30, 31, entonces* $\models \{p\}\, c\, \{q\}$

2.4. Estructuras de Datos

Los punteros y la memoria dinámica nos permiten definir estructuras de datos mutables, ya sean en los datos que contengan o también en el tamaño o forma de la misma. Para especificar un programa completamente, se necesita no solo describir la forma de sus estructuras, sino también relacionar los estados del programa con los valores abstractos que representan. Para esto, normalmente se definen los valores abstractos de manera algebraica y se los relaciona con nuevos predicados de la *Separation Logic* definidos de manera recursiva sobre aquellos valores abstractos.

2.4.1. Listas

Las listas enlazadas son una de las estructuras de datos más simples y utilizadas y se usan para representar secuencias de valores abstractos.

Definición 34 *El conjunto de expresiones de secuencias de valores* Sequence *se define por la siguiente gramática:*

$$\mathsf{Sequence} \qquad es \doteq [\,] \mid e \triangleright es$$

donde $e \in$ Expr. *Denotaremos con* $[e_1, ..., e_n]$ *a la secuencia* $e_1 \triangleright ... \triangleright e_n \triangleright [\,]$

Para razonar sobre programas con estructuras que representan secuencias de valores es necesario definir ciertas operaciones sobre ellas.

Definición 35 *Las operaciones de inserción al final de una secuencia* $\triangleleft$*, de concatenación de secuencias* $+\!\!+$ *y de borrado de elementos con cierto valor* $\ominus$ *se definen como sigue:*

$$\begin{aligned} \triangleleft &\in [\mathsf{Values}] \to \mathsf{Values} \to [\mathsf{Values}] \\ [\,] \triangleleft v &\doteq [v] \\ (e \triangleright es) \triangleleft v &\doteq e \triangleright (es \triangleleft v) \end{aligned}$$

$$\begin{aligned} +\!\!+ &\in [\mathsf{Values}] \to [\mathsf{Values}] \to [\mathsf{Values}] \\ [\,] +\!\!+ vs &\doteq vs \\ (e \triangleright es) +\!\!+ vs &\doteq e \triangleright (es +\!\!+ vs) \end{aligned}$$

$$\begin{aligned} \ominus &\in [\mathsf{Values}] \to \mathsf{Values} \to [\mathsf{Values}] \\ [\,] \ominus v &\doteq [\,] \\ (e \triangleright es) \ominus v &\doteq \begin{cases} e \triangleright (es \ominus v) & \text{si } e \neq v \\ es \ominus v & \text{en caso contrario} \end{cases} \end{aligned}$$

Una forma típica de representar a una secuencia de valores es mediante una lista enlazada a partir del puntero x. Básicamente, la secuencia vacía se representaría con el puntero $x = \mathbf{nil}$ y una secuencia no vacía se representaría con el puntero x apuntando a un registro de memoria de dos campos, donde el primero se usa para representar el primer valor de la lista y el segundo como un puntero a la representación del resto de la lista.

Definición 36 *Una lista enlazada que comienza en* x *se define con el siguiente predicado recursivo:*

$$\begin{aligned} \mathbf{list}.x.[\,] &\doteq x = \mathbf{nil} \wedge \mathbf{emp} \\ \mathbf{list}.x.(e \triangleright es) &\doteq (\exists y \ \cdot \ x \mapsto e, y * \mathbf{list}.y.es) \end{aligned}$$

Llamaremos nodo *a cada registro referenciado por punteros de la lista enlazado. En el caso de la definición recursiva el par* (e, y) *representa un nodo.*

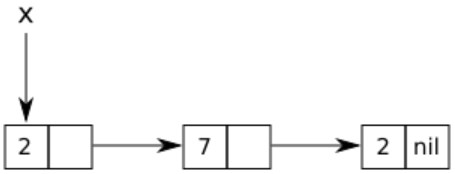


Figura 2.3: Estado de **list**.x.$[2, 7, 2]$

Ejemplo 37 *El predicado* **list**.x.$[2, 7, 2]$ *representa una lista enlazada de tres elementos* 2, 7 *y* 2 *y los estados que los satisfacen son como los representados en la figura 2.3.*

Ejemplo 38 *Para demostrar el uso de este predicado en la verificación de programas, consideremos el siguiente ejemplo sencillo de borrado del primer elemento de una lista enlazada. Como precondición se establece que la secuencia representada no puede ser vacía.*

$$
\begin{array}{l}
\{\mathbf{list}.x.(e \triangleright es)\} \\
\Leftrightarrow (\textit{definición de } \mathbf{list}) \\
\{\exists z' \ \cdot \ x \mapsto e, z' * \mathbf{list}.z'.es\} \\
\quad y := x.1; \\
\{\exists y' \ \cdot \ (\exists z' \ \cdot \ x \mapsto e, z' * \mathbf{list}.z'.es \wedge y = z')\} \\
\Leftrightarrow (\textit{eliminación de los } \exists) \\
\{x \mapsto e, y * \mathbf{list}.y.es\} \\
\quad \mathbf{free}(x); \\
\{\mathbf{list}.y.es\} \\
\quad x := y \\
\{\mathbf{list}.x.es\}
\end{array}
$$

Una desventaja del predicado **list** es que, en muchos casos, no nos permite representar estructuras parciales que ocurren en la ejecución de un programa sobre listas. Tomemos como ejemplo el programa de borrado de todas las ocurrencias de un valor particular en una lista enlazada de la figura 2.4.

Durante la ejecución del ciclo, en la memoria habría una parte de la lista enlazada que ya fue recorrida y de la que ya se eliminaron los valores iguales a v y otra parte que resta recorrer. Esta última parte se puede especificar fácilmente con el predicado de lista que definimos más arriba, pues efectivamente tenemos una lista enlazada que comienza en la variable y y que termina en **nil**. Sin embargo, el segmento de lista ya recorrido no es una lista que termina en **nil**. Entonces necesitamos un predicado más general para poder hablar de estos *segmentos de listas*. Una solución es tomar dos variables para especificarlo, una para determinar el comienzo y la otra para el final del segmento.

Definición 39 *Un segmento enlazado de lista que comienza en x y termina en y se define con el siguiente predicado recursivo:*

$$
\begin{array}{rl}
\mathbf{lseg}.x.y.[\,] & \doteq x = y \wedge \mathbf{emp} \\
\mathbf{lseg}.x.y.(e \triangleright es) & \doteq x \neq y \wedge (\exists z \ \cdot \ x \mapsto e, z * \mathbf{lseg}.z.y.es)
\end{array}
$$

```
{list.x.es}
  y := x;
  while y ≠ nil do
      w := y.0;
      t := y.1;
      if w = v then
          free(y);
          if x = y then
              x := t
          else
              p.1 := t
          fi
      else
          p := y
      fi;
      y := t
  od
{list.x.(es ⊖ v)}
```

Figura 2.4: Algoritmo de borrado de las ocurrencias de un valor en una lista

Una lista típica se podría expresar como un segmento de lista con la segunda variable igual a **nil**, como quedará explicitado en la propiedad 1 del lema 41.

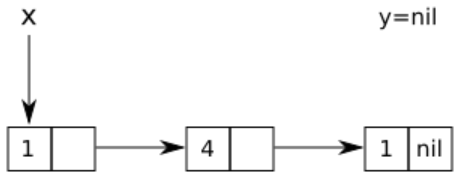
Ejemplo 40 *El predicado* $\mathbf{lseg}.x.y.[1,4,1]$ *representa un segmento de lista enlazada con tres elementos* 1, 4 *y* 1 *y los estados que los satisfacen son como los representados en la figura 2.5. El primero con* $y = nil$ *y el segundo con* y *un puntero* dangling.

Lema 41 *Las siguentes son fórmulas válidas sobre los predicados* **list** *y* **lseg***:*

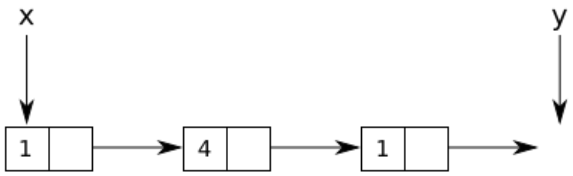
1. $\mathbf{lseg}.x.\mathbf{nil}.es \Leftrightarrow \mathbf{list}.x.es$
2. $\mathbf{list}.x.es \wedge x \neq \mathbf{nil} \Leftrightarrow (\exists e', y', es' \cdot x \mapsto e', y' * \mathbf{list}.y'.es' \wedge es = e' \triangleright es')$
3. $\mathbf{list}.x.es \wedge x = \mathbf{nil} \Rightarrow \mathbf{emp} \wedge es = [\,]$
4. $\mathbf{list}.x.(es \triangleleft e) \Leftrightarrow (\exists y' \cdot \mathbf{lseg}.x.y'.es * y' \mapsto e, \mathbf{nil})$
5. $\mathbf{lseg}.x.y.(es \triangleleft e) * y \mapsto a, z \Leftrightarrow (\exists y' \cdot \mathbf{lseg}.x.y'.es * y' \mapsto e, y * y \mapsto a, z)$

Notar la ocurrencia de predicado $y \mapsto a, z$ en la propiedad 5 del lema anterior. Si no estuviera, solo valdría la implicación $\Rightarrow$ y perderíamos la validez de la vuelta, ya que para obtener un segmento de lista entre x e y, la variable y no puede apuntar a ningún registro del segmento. Con ese predicado aseguramos entonces que la variable y es un puntero que está definido en una porción disjunta del *heap* del segmento de lista y por lo tanto no puede apuntar a ningún nodo del mismo.

Para verificar el programa de borrado hay que tener en cuenta algunos detalles. La variable de puntero p referencia al nodo anterior al que se estará



(a) cuando $y = \mathbf{nil}$



(b) cuando y es un puntero *dangling*

Figura 2.5: Estados que satisfacen $\mathbf{lseg}.x.y.[1, 4, 1]$

recorriendo en el cuerpo del ciclo. Esto es necesario para poder actualizar la referencia al nodo siguiente en la lista en el caso en que el nodo actual sea eliminado de la memoria. Como el primer nodo no tiene nodo anterior, al comienzo y mientras se borren todos los nodos iniciales (por tener un valor igual a v) p en realidad no estará inicializado, pero valdrá que $x = y$. El comando **if** más interno está para separar estos dos casos. Notar entonces que el invariante del ciclo es un poco complejo, ya que debe contemplar estas dos situaciones.

Recordemos la precondición y postcondición del programa de la figura 2.4:

$$\begin{array}{rl} \text{PRE:} & \{\mathbf{list}.x.es\} \\ \text{POST:} & \{\mathbf{list}.x.(es \ominus v)\} \end{array}$$

El invariante del ciclo será:

$$\mathbf{list}.y.es'' * (I_1 \vee I_2)$$

donde I_1 e I_2 son fórmulas para describir los dos casos mencionados antes y es'' es una variable cuantificada existencialmente para la secuencia de valores representada por la lista enlazada que comienza en y. A continuación mostramos la verificación del programa.

Ejemplo 42 *Usaremos L_1, I_1, L_2 e I_2 para denotar predicados según las siguientes ecuaciones:*

$$\begin{array}{l} L_1 \doteq vs' \ominus v = [\,] \wedge es = vs' +\!\!+ es'' \\ I_1 \doteq \mathbf{emp} \wedge x = y \wedge L_1 \\ L_2 \doteq e' \neq v \wedge vs' \ominus v = [\,] \wedge es = es' +\!\!+ [e'] +\!\!+ vs' +\!\!+ es'' \\ I_2 \doteq \mathbf{lseg}.x.p.(es' \ominus v) * p \mapsto e', y \wedge L_2 \end{array}$$

donde las variables primadas o doblemente primadas estarán cuantificadas existencialmente de manera implícita. Los predicados L_1 y L_2 son pure *y hacen referencia a los valores abstractos representados. La variable de secuencia vs'*

representa una secuencia con todos sus valores iguales a v, por eso siempre estará junto con $vs' \ominus v = [\,]$.

Veamos que el invariante presentado $\mathbf{list}.y.es'' * (I_1 \vee I_2)$ *se cumple al comienzo del ciclo:*

$\{\mathbf{list}.x.es\}$
$\Leftrightarrow$
$\{\mathbf{list}.x.es \wedge x = x\}$
$\quad y := x;$
$\{\mathbf{list}.y.es \wedge x = y\}$
$\Leftrightarrow$ (*definición de* $\ominus$ y $+\!\!+$)
$\{\mathbf{list}.y.es \wedge x = y \wedge [\,] \ominus v = [\,] \wedge es = [\,] +\!\!+ es\}$
$\Rightarrow$ (*doble introducción de* $\exists$ *y elemento neutro de* $*$)
$\{\mathbf{list}.y.es'' * \mathbf{emp} \wedge x = y \wedge vs' \ominus v = [\,] \wedge es = vs' +\!\!+ es''\}$
$\Rightarrow$ (*distributividad de* $*$ *sobre* $\vee$)
$\{\mathbf{list}.y.es'' * (I_1 \vee I_2)\}$

Veamos que la conjunción del invariante con la negación de la guarda del ciclo implica la postcondición del programa:

$\{\mathbf{list}.y.es'' * (I_1 \vee I_2) \wedge y = \mathbf{nil}\}$
$\Rightarrow$ (*propiedad 3 lema 41*)
$\{\mathbf{emp} * (I_1 \vee I_2) \wedge es'' = [\,] \wedge y = \mathbf{nil}\}$
$\Rightarrow$ (*distributividad con* $\vee$ *, cálculo elemental de secuencias*)
$\{(\mathbf{emp} \wedge x = \mathbf{nil} \wedge vs' \ominus v = [\,] \wedge es = vs')$
$\vee (\mathbf{lseg}.x.p.(es' \ominus v) * p \mapsto e', \mathbf{nil} \wedge L_2 \wedge es'' = [\,])\}$
$\Rightarrow$ (*definición de* **list**, *propiedad 4 lema 41*)
$\{\mathbf{list}.x.(es \ominus v) \vee (\mathbf{list}.x.((es' \ominus v) \triangleleft e') \wedge L_2 \wedge es'' = [\,])\}$
$\Rightarrow$ (*cálculo sobre secuencias, idempotencia de* $\vee$)
$\{\mathbf{list}.x.(es \ominus v)\}$

Por último, en la figura 2.6, podemos ver que, con la conjunción de la guarda, se mantiene el invariante luego de una ejecución del cuerpo del ciclo. Notar que $\mathbf{list}.y.es'' * I_2 \Rightarrow x \neq y$*, por lo que efectivamente el* **if** *más interno distingue entre I_1 e I_2. Debido a la extensión de la especificación, no se incluyen todos los detalles. En repetidas ocasiones, se aplican varias operaciones por vez.*

2.4.2. Árboles binarios

Dentro de las estructuras de datos no lineales, una de las más simples y de las más utilizadas son los árboles binarios. Como hicimos con las secuencias de valores abstractos y el predicado **list**, presentamos primero los árboles de valores abstractos y luego cómo representarlos en la memoria dinámica.

Definición 43 *El conjunto de expresiones sobre árboles binarios* Tree *se define por la siguiente gramática:*

$$\mathsf{Tree} \qquad t \doteq \langle\,\rangle \mid \langle t, e, t\rangle$$

donde $e \in$ Expr*. Denotaremos con* $\langle e\rangle$ *al árbol binario* $\langle\langle\,\rangle, e, \langle\,\rangle\rangle$ *y en un árbol* $\langle lt, e, rt\rangle$ *diremos que lt es el sub-árbol izquierdo, rt es el sub-árbol derecho y que e representa la raíz del árbol.*

$\{\mathbf{list}.y.es'' * (I_1 \vee I_2) \wedge y \neq \mathbf{nil}\}$
$\Rightarrow$ (propiedad 2 lema 41)
$\{y \mapsto e'', y' * \mathbf{list}.y'.es_1'' \wedge es'' = e'' \triangleright es_1'' * (I_1 \vee I_2)\}$
  $w := y.0;$
  $t := y.1;$
$\{y \mapsto w, t * \mathbf{list}.t.es_1'' \wedge es'' = w \triangleright es_1'' * (I_1 \vee I_2)\}$
  if $w = v$ **then**
    $\{y \mapsto w, t * \mathbf{list}.t.es_1'' \wedge es'' = w \triangleright es_1'' * (I_1 \vee I_2) \wedge w = v\}$
    $\Rightarrow$ (cálculo sobre secuencias, renombre de variables cuantificadas)
    $\{y \mapsto w, t * \mathbf{list}.t.es'' * (I_1 \vee I_2)\}$
      free(y);
    $\{\mathbf{list}.t.es'' * (I_1 \vee I_2)\}$
      if $x = y$ **then**
        $\{\mathbf{list}.t.es'' * I_1\}$
          $x := t$
        $\{\mathbf{list}.t.es'' \wedge L_1 \wedge x = t\}$
      else
        $\{\mathbf{list}.t.es'' * I_2\}$
          $p.1 := t$
        $\{\mathbf{list}.t.es'' * \mathbf{lseg}.x.p.(es' \ominus v) * p \mapsto e', t \wedge L_2\}$
      fi
    $\{(\mathbf{list}.t.es'' \wedge x = t \wedge L_1)$
    $\vee (\mathbf{list}.t.es'' * \mathbf{lseg}.x.p.(es' \ominus v) * p \mapsto e', t \wedge L_2)\}$
  else
    $\{y \mapsto w, t * \mathbf{list}.t.es_1'' \wedge es'' = w \triangleright es_1'' * (I_1 \vee I_2) \wedge w \neq v\}$
    $\Rightarrow$ (cálculo de secuencias, definición de **lseg**, propiedad 5 lema 41)
    $\{y \mapsto w, t * \mathbf{list}.t.es_1'' \wedge w \neq v \wedge vs' \ominus v = [\,] *$
    $((\mathbf{lseg}.x.y.(vs' \ominus v) \wedge es = vs' +\!\!+ [w] +\!\!+ [\,] +\!\!+ es_1'') \vee$
    $(\mathbf{lseg}.x.y.((es' +\!\!+ [e'] +\!\!+ vs') \ominus v) \wedge es = (es' +\!\!+ [e'] +\!\!+ vs') +\!\!+ [w] +\!\!+ [\,] +\!\!+ es_1''))\}$
    $\Rightarrow$ (introducción de $\exists$, idempotencia de $\vee$ y renombre de variable)
    $\{y \mapsto e', t * \mathbf{list}.t.es'' * \mathbf{lseg}.x.y.(es' \ominus v) \wedge L_2\}$
      $p := y$
    $\{p \mapsto e', t * \mathbf{list}.t.es'' * \mathbf{lseg}.x.p.(es' \ominus v) \wedge L_2\}$
  fi;
$\{(\mathbf{list}.t.es'' \wedge x = t \wedge L_1)$
$\vee (\mathbf{list}.t.es'' * \mathbf{lseg}.x.p.(es' \ominus v) * p \mapsto e', t \wedge L_2)\}$
  $y := t$
$\{\mathbf{list}.y.es'' * (I_1 \vee I_2)\}$

Figura 2.6: Verificación del invariante del ciclo para el programa de borrado de todas las ocurrencias de un valor en una lista enlazada (ejemplo 42)

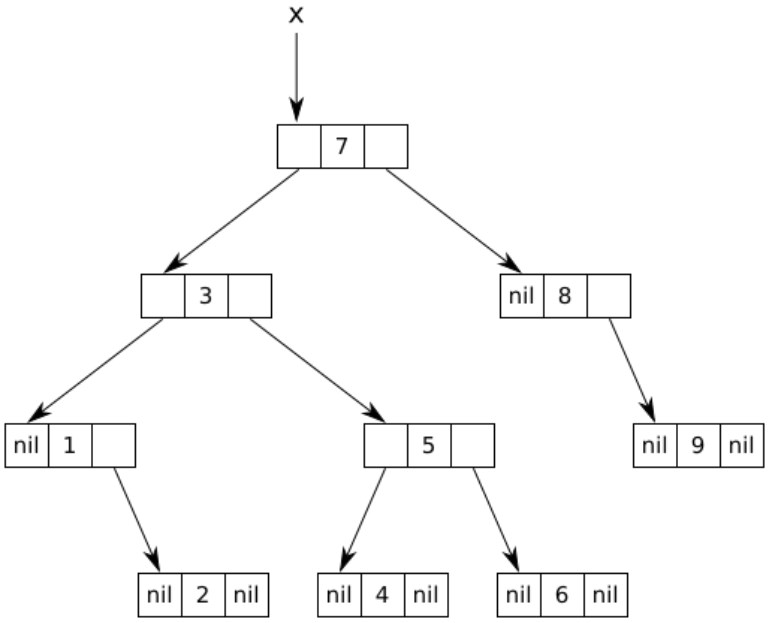


Figura 2.7: Estado de $\mathbf{tree}.x.\langle\langle\langle\,\rangle, 1, \langle 2\rangle\rangle, 3, \langle\langle 4\rangle, 5, \langle 6\rangle\rangle\rangle, 7, \langle\langle\,\rangle, 8, \langle 9\rangle\rangle\rangle$

Definición 44 *Un árbol binario que comienza en x se define recursivamente por las siguientes ecuaciones:*

$$\mathbf{tree}.x.\langle\,\rangle \doteq x = \mathbf{nil} \wedge \mathbf{emp}$$
$$\mathbf{tree}.x.\langle lt, e, rt\rangle \doteq (\exists l, r \;\cdot\; x \mapsto l, e, r * \mathbf{tree}.l.lt * \mathbf{tree}.r.rt)$$

Llamaremos nodo *a los registros referenciados por los punteros que ocurren en el árbol binario. Es el caso de la tripla (l, e, r) en la segunda ecuación de la definición recursiva.*

Ejemplo 45 *La fórmula $\mathbf{tree}.x.\langle\langle\langle\,\rangle, 1, \langle 2\rangle\rangle, 3, \langle\langle 4\rangle, 5, \langle 6\rangle\rangle\rangle, 7, \langle\langle\,\rangle, 8, \langle 9\rangle\rangle\rangle$ representa un árbol binario de búsqueda. Esto significa que para cada sub-árbol no elemental se cumple que los valores del sub-árbol izquierdo son menores al valor de la raíz que a su vez es menor a los valores del sub-árbol derecho. En un árbol con estas características, la búsqueda de un valor particular v se puede realizar comparando v con el valor del nodo raíz y, mientras no sean iguales, siguiendo sucesivamente hacia el nodo del sub-árbol izquierdo o derecho según v sea menor o mayor. En la figura 2.7 se muestra una representación de los estados que satisfacen la fórmula presentada.*

Similarmente a lo que ocurría con el predicado de listas enlazadas **list**, el predicado **tree** encuentra su límite en la representación de estructuras parciales que ocurren durante la ejecución de algoritmos iterativos sobre árboles. Sin embargo, en este caso no resulta tan sencillo extender su definición como lo hicimos con los segmentos de lista **lseg** para el caso de **list**. La principal complicación es el hecho de que estamos ante una estructura de datos no lineal. De todas maneras, en el próximo capítulo presentamos un predicado general que nos permite razonar sobre programas sobre árboles, aunque verificando propiedades sobre la forma de la estructura de memoria, más que sobre los valores que representan.

Capítulo 3

Shape Analysis

En este capítulo se presenta un *shape analysis* que permite verificar la validez de propiedades sobre estructuras de datos no lineales en memoria, como árboles binarios. Es una extensión de análisis de [9] que también se basa en la ejecución simbólica de programas sobre estados abstractos compuestos por fórmulas particulares de la *Separation Logic*, llamadas *symbolic heaps*. En nuestra extensión, que considera estructuras de datos no lineales, podemos prever algunas dificultades:

- El uso de un predicado *naive* para describir las estructuras, como el usual para árboles en Separation Logic (definición 44 de la sección 2.4.2) daría lugar a una proliferación de ocurrencias del predicado por la recursión múltiple inherente a la estructura de datos.
- Los algoritmos sobre árboles (y grafos) suelen ser más complejos que sus pares sobre listas. Usualmente involucran una intensa manipulación de punteros y estructuras de control anidadas. Algoritmos como el Schorr-Waite de marcado de un grafo, suelen proponerse como desafíos para cualquier nueva metodología de verificación de programas con punteros. Esta complejidad impone un fuerte requisito de precisión en los invariantes de ciclo para ser capaz de verificar propiedades interesantes.
- Cada recorrido particular de una estructura suele ser relevante para la validez de propiedades interesantes de un algoritmo. Los múltiples enlaces característicos de los nodos de estructuras no lineales hacen crecer exponencialmente las opciones de recorrido de las mismas. Esto plantea la necesidad de un equilibro entre precisión y abstracción para evitar un crecimiento exagerado en el número de *symbolic heaps* que conforman un estado abstracto.

En este capítulo presentamos un *shape analysis* con garantías de terminación que, dada una precondición sobre un programa, computa automáticamente una postcondición e invariantes para cada ciclo del programa. El análisis adapta para cada variable de programa el nivel de abstracción en el cálculo de invariantes, teniendo en cuenta los requisitos de información según la manipulación que tenga sobre la misma a lo largo del programa. De esta forma los invariantes calculados resultan compactos y lo suficientemente precisos en la mayoría de los casos

prácticos. Como parte del análisis, introducimos un predicado recursivo lineal para la descripción de familias de árboles binarios que nos permite especificar de manera concisa las estructuras de datos parciales que ocurren en puntos intermedios en la ejecución de algoritmos iterativos.

Para explicar el análisis, primero presentamos el marco semántico, introduciendo un modelo de memoria concreto y semántica operacional que define nuestro lenguaje de programación. Es una variante del marco del capítulo anterior para considerar algoritmos que manipulan árboles binarios. Posteriormente presentamos una semántica intermedia ejecutable que considera los programas como transformadores de estados abstractos compuestos por *symbolic heaps*. Esta semántica se extiende luego con la introducción de una fase de abstracción para el cómputo de invariantes de ciclo, definiendo una semántica abstracta ejecutable y con garantías de terminación. Finalmente presentamos resultados experimentales del análisis para una variedad de algoritmos sobre árboles binarios.

3.1. Semántica Concreta

Para la semántica del lenguaje de programación usamos un modelo de memoria similar al de la definición 1, considerando a $\mathbb{Z}_k$, el conjunto de los enteros entre 0 y $k-1$, como el conjunto de valores elementales y limitando la imagen del *heap* a registros de un tamaño fijo de tres campos. Esto nos permite representar la estructura de un árbol binario donde en cada nodo (registro) se guardan un dato y punteros a los subárboles izquierdo y derecho. Para las variables hacemos una distinción entre un conjunto finito de variables de programa Var y un conjunto infinito de variables de especificación $\mathsf{Var'}$ que estarán cuantificadas existencialmente de manera implícita y no podrán aparecen en los comandos.

Definición 46 *Dado un entero positivo k, un conjunto infinito de direcciones de memoria $\mathsf{Locations}$, con un elemento distinguido* **nil** $\in \mathsf{Locations}$*, un conjunto* finito *Var de variables de programa y un conjunto enumerable $\mathsf{Var'}$ de variables lógicas, el conjunto de estados States_a se define como:*

$$
\begin{aligned}
&\mathsf{Values}_a \doteq \mathbb{Z}_k \cup \mathsf{Locations} && \text{con } \mathbb{Z}_k \cap \mathsf{Locations} = \emptyset \\
&\mathsf{Stacks}_a \doteq (\mathsf{Var} \cup \mathsf{Var'}) \to \mathsf{Values}_a && \text{con } \mathsf{Var} \cap \mathsf{Var'} = \emptyset \\
&\mathsf{Heaps}_a \doteq (\mathsf{Locations} - \{\mathbf{nil}\}) \rightharpoonup_f \mathsf{Values}_a \times \mathsf{Values}_a \times \mathsf{Values}_a \\
&\mathsf{States}_a \doteq \mathsf{Stacks}_a \times \mathsf{Heaps}_a
\end{aligned}
$$

Usamos un lenguaje de programación imperativo simple con comandos de manipulación del heap, similar a lo presentado en las definiciones 2 y 4, con la diferencia de que las expresiones booleanas se limitan a disyunciones de conjunciones de relaciones de igualdad o desigualdad (o sea, una Forma Normal Disyuntiva) y el comando de construcción toma tres expresiones (para la creación de un registro de tres campos).

Definición 47 *La sintaxis de las expresiones Expr_a, las relaciones Rel_a, las expresiones booleanas BExp_a y los comandos Comm_a están dadas por la siguientes*

gramáticas:

$$
\begin{array}{rl}
\mathit{Expr}_a & e ::= x \mid \mathbf{nil} \mid 0 \mid 1 \mid \ldots \mid k-1 \\
\mathit{Rel}_a & r ::= e = e \mid e \neq e \\
\mathit{BExp}_a & b ::= (r \wedge \ldots \wedge r) \vee \ldots \vee (r \wedge \ldots \wedge r) \\
\mathit{Comm}_a & c ::= \mathbf{skip} \mid x := e \mid x := \mathbf{new}(e,e,e) \mid x := x.i \mid x.i := e \mid \mathbf{free}(x) \\
 & \quad \mid c;c \mid \mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else}\ c\ \mathbf{fi} \mid \mathbf{while}\ b\ \mathbf{do}\ c\ \mathbf{od}
\end{array}
$$

donde $x \in \mathit{Var}$ *y* $0 \leq i \leq 2$.

La semántica concreta del lenguaje de programación está dada por funciones continuas $[\![p]\!]_c : \mathsf{D}_c \to \mathsf{D}_c$ para cada comando p sobre el reticulado completo D_c, definido a continuación:

Definición 48 *Dado el conjunto de estados* States_a *y un elemento distinguido* $\mathbf{abort} \notin \mathit{States}_a$*, definimos el reticulado* D_c *como:*

$$D_c \doteq (\mathcal{P}(\mathit{States}_a) \cup \{\mathbf{abort}\}, \subseteq)$$

con $\subseteq$ *la relación de inclusión de conjuntos que además cumple que para todo* $d \in D_c$, $d \subseteq \mathbf{abort}$

El elemento distinguido **abort** representará una ejecución que termina anormalmente con un fallo de memoria.

La semántica de cada comando atómico (i.e. skip, asignación, construcción, consulta, mutación y destrucción) se puede definir según la semántica operacional presentada en la definición 6. Notar que, para cualquier comando atómico a, $\rightsquigarrow$ relaciona una configuración $((s,h),a) \in \mathsf{States}_a \times \mathsf{Comm}_a$ con configuraciones terminales $\zeta \in \mathsf{States}_a \cup \{\mathbf{abort}\}$. Entonces, la semántica de cada comando primitivo a puede extenderse fácilmente a una función sobre D_c.

Definición 49 *La función* $a^\uparrow : D_c \to D_c$*, con* $a \in \mathit{Comm}_a$ *un comando atómico se define como:*

$$
a^\uparrow.d \doteq \begin{cases}
\mathbf{abort} & \text{si } d = \mathbf{abort} \text{ o existe } (s,h) \in d \text{ tal que } ((s,h),a) \rightsquigarrow \mathbf{abort} \\
\{(s',h') \in \mathit{States}_a \mid \text{existe } (s,h) \in d \text{ tal que } ((s,h),a) \rightsquigarrow (s',h')\} & \\
\qquad \text{en caso contrario} &
\end{cases}
$$

Si usamos una semántica estándar para las expresiones booleanas $[\![\ ]\!]_b$, que solo dependen de *stack*, como en la definición 3 y usamos $a^\uparrow$ para la semántica de los comandos atómicos a, la semántica denotacional de los comandos compuestos se extiende de forma usual.

Definición 50 *La semántica concreta de los comandos* $[\![\ ]\!]_c \in \mathit{Comm}_a \to D_c \to D_c$*, se define como:*

$$
\begin{array}{rl}
[\![a]\!]_c & \doteq a^\uparrow \\
[\![c_1;c_2]\!]_c & \doteq [\![c_2]\!]_c \circ [\![c_1]\!]_c \\
[\![\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{fi}]\!]_c & \doteq ([\![c_1]\!]_c \circ \mathit{filter}_c.b) \cup_{\mathbf{abort}} ([\![c_2]\!]_c \circ \mathit{filter}_c.\neg b) \\
[\![\mathbf{while}\ b\ \mathbf{do}\ c_1\ \mathbf{od}\ ]\!]_c & \doteq \lambda d \cdot \mathit{filter}_c.\neg b.(\mu d' \cdot d \cup_{\mathbf{abort}} ([\![c_1]\!]_c \circ \mathit{filter}_c.b).d')
\end{array}
$$

donde a representa los comandos atómicos, y denotamos con μ el operador de mínimo punto fijo, con $\circ$ la composición funcional, y con $\neg$ la meta-operación que transforma una expresión booleana en su negación en FND:

$$\neg((r_{1,1} \wedge \ldots \wedge r_{1,k_1}) \vee \ldots \vee (r_{m,1} \wedge \ldots \wedge r_{m,k_m})) \doteq \bigvee_{\substack{1 \leq i_1 \leq k_1 \\ \ldots \\ 1 \leq i_m \leq k_m}} \neg r_{1,i_1} \wedge \ldots \wedge \neg r_{m,i_m}$$

donde $\neg r$ denota la relación de igualdad o desigualdad negada de r. La función $\cup_{\mathbf{abort}} : D_c \times D_c \to D_c$ es la unión con **abort***:*

$$d_1 \cup_{\mathbf{abort}} d_2 \doteq \begin{cases} \mathbf{abort} & \text{si } d_1 = \mathbf{abort} \text{ o } d_2 = \mathbf{abort} \\ d_1 \cup d_2 & \text{en caso contrario} \end{cases}$$

La función $\mathit{filter}_c.b_1 : D_c \to D_c$ remueve los estados que no son consistentes con el valor de verdad de la expresion booleana b_1:

$$\mathit{filter}_c.b_1.d \doteq \begin{cases} \mathbf{abort} & \text{si } d = \mathbf{abort} \\ \{(s,h) \in d \mid [\![b_1]\!]_b.s\} & \text{en caso contrario} \end{cases}$$

3.2. Semántica Intermedia

En nuestro análisis consideramos los programas como transformadores de estados abstractos. En esta sección introducimos una semántica intermedia ejecutable que opera sobre conjuntos de fórmulas restringidas de la *Separation Logic* llamadas *symbolic heaps*. Intuitivamente un estado abstracto representa al conjunto de estados concretos que satisfacen alguna de las fórmulas que lo componen.

Un *symbolic heap* $\Pi \mid \Sigma$ se compone de una fórmula *pure* Π con predicados sobre igualdades y desigualdades de variables, y una fórmula espacial Σ con predicados sobre el *heap*. En este nuevo contexto, dentro de las expresiones tenemos que incluir a las variables primadas, que serán necesarias para las especificaciones de programas.

Definición 51 *Las sintaxis de las expresiones $\mathit{Expr'}_a$, de las relaciones $\mathit{Rel'}_a$, de los multiconjuntos constantes Multiset, de las fórmulas puras Pure, de las fórmulas espaciales Space y de los* symbolic heaps *SH están dadas por las siguientes gramáticas:*

$$\begin{array}{rl} \mathit{Expr'}_a & e ::= x \mid x' \mid \mathbf{nil} \mid 0 \mid 1 \mid \ldots \mid k-1 \\ \mathit{Rel'}_a & r ::= e = e \mid e \neq e \\ \mathit{Multiset} & m ::= \{\!|e, ..., e|\!\} \\ \mathit{Pure} & \Pi ::= \mathbf{true} \mid r \mid \Pi \wedge \Pi \\ \mathit{Space} & \Sigma ::= \mathbf{emp} \mid \mathbf{true} \mid \mathbf{junk} \mid y \mapsto e, e, e \mid \mathbf{trees}.m.m \mid \Sigma * \Sigma \\ \mathit{SH} & sh ::= \Pi \mid \Sigma \end{array}$$

donde $x \in \mathit{Vars}$, $x' \in \mathit{Vars'}$, $y \in \mathit{Vars} \cup \mathit{Vars'}$.

Notación: Denotamos con $\emptyset$ al multiconjunto vacío $\{\!|\,|\!\}$

La semántica de los *symbolic heaps* respecto a los estados concretos sigue la presentada en la definición 16. El predicado **junk** especifica que existe alguna cantidad de celdas asignadas pero inaccesibles. Tiene el propósito de establecer que hay *basura* en el *heap*, tal vez generada por *memory leaks*. El predicado del heap **true** es válido en situaciones en las que **junk** es válido, pero también en las que **emp** es válido, por lo que sólo indica la posibilidad de existencia de *basura* en el *heap*. Definimos a continuación la semántica formal para los *symbolic heaps*, con la excepción del predicado **trees** que se presenta más adelante.

Definición 52 *La semántica de los* symbolic heaps SH *está dada por la relación de satisfacción* $\models$ *:*

$$\models \subseteq \mathsf{States}_a \times (\mathsf{Pure} \cup \mathsf{Space} \cup \mathsf{SH})$$

definida de la siguiente manera:

$(s,h) \models r$	*si y sólo si* $[\![r]\!]_b.s$
$(s,h) \models \mathbf{true}$	*siempre*
$(s,h) \models \Pi \wedge \Pi'$	*si y sólo si* $(s,h) \models \Pi$ *y* $(s,h) \models \Pi'$
$(s,h) \models \mathbf{emp}$	*si y sólo si* $h = \emptyset$
$(s,h) \models \mathbf{junk}$	*si y sólo si* $h \neq \emptyset$
$(s,h) \models x \mapsto l, v, r$	*si y sólo si* $h = \{([\![x]\!]_e.s, ([\![l]\!]_e.s, [\![v]\!]_e.s, [\![r]\!]_e.s))\}$
$(s,h) \models \Sigma * \Sigma'$	*si y sólo si existen* $h_1, h_2 \in \mathsf{Heaps}_a$ *tales que* $h_1 \perp h_2$, $h = h_1 \cup h_2, (s,h_1) \models \Sigma$ *y* $(s,h_2) \models \Sigma'$
$(s,h) \models \Pi \restriction \Sigma$	*si y sólo si existe* $\vec{v}$ *tal que* $((s\|\vec{x}' \to \vec{v}), h) \models \Pi$ *y* $((s\|\vec{x}' \to \vec{v}), h) \models \Sigma$

donde $\vec{x}' \in \mathsf{Var}'^*$ *es la secuencia de variables primadas que ocurren en* $\Pi \restriction \Sigma$*. Extendemos la notación* $f|\vec{x} \to \vec{v}$ *para denotar a la funcion* f *con la actualización del valor de cada campo de* $\vec{x}$ *con el valor del correspondiente campo en* $\vec{v}$*. Notar que definimos la relación* $\models$ *sobre los predicados en* Pure *y en* Space*, pero debido a la presencia de variables primadas, sólo nos interesa la relación sobre* SH*. Si en algún momento hablamos solo de la parte pura o de la parte espacial de una fómula, asumiremos que se trata de un* symbolic heap *en* SH *donde la parte no mencionada es igual a* **true***.*

La intención del predicado **trees** es definir una familia de árboles binarios. En toda su generalidad **trees**.$C.D$ define familias de árboles posiblemente parciales con raíces o punteros de entrada en C y punteros de salida en D. Los punteros de salida pueden referenciar a nodos internos de otros árboles (o del mismo árbol siempre que no se generen ciclos) o ser *dangling*. Muchos árboles pueden tener punteros de salida que apunten al mismo nodo o, que aún siendo *dangling* tengan la misma dirección (inválida) de memoria. Incluso un mismo predicado **trees** podría tener múltiples ocurrencias del mismo puntero de salida. En este sentido, **trees** especifica grafos dirigidos acíclicos (*dags*) con posibilidad de *sharing* que quedará restringida a los registros referenciados por los punteros de salida. Por otro lado, para un predicado **trees**.$C.D$, un puntero distinto de **nil** que no ocurra en D no puede ocurrir múltiples veces en C sin dar lugar a una inconsistencia.

Definición 53 *Dado un estado* $(s,h) \in \mathsf{States}_a$*, decimos que se cumple* $(s,h) \models$ **trees**.$\{|c_1, ..., c_n|\}.\{|d_1, ..., d_m|\}$ *si y sólo si:*

$$
\begin{array}{ll}
(s,h) \models \mathbf{emp} \ y \ [\![d_i]\!]_e.s = \mathbf{nil} \ \textit{para todo} \ 1 \leq i \leq m & \textit{si } n = 0 \\
(s,h) \models \mathbf{trees}.\{\!|c_2,...,c_n|\!\}.\{\!|d_1,...,d_m|\!\} & \textit{si } [\![c_1]\!]_e.s = \mathbf{nil} \\
(s,h) \models \mathbf{trees}.\{\!|c_2,...,c_n|\!\}.\{\!|d_1,...,d_{i-1},d_{i+1},...,d_m|\!\} & \textit{si } [\![c_1]\!]_e.s = [\![d_i]\!]_e.s \\
 & \textit{para algún } 1 \leq i \leq m \\
h.([\![c_1]\!]_e.s) = (v_1,v_2,v_3) \ y & \\
(s',h') \models \mathbf{trees}.\{\!|l',r',c_2,...,c_n|\!\}.\{\!|d_1,...,d_m|\!\} & \\
\quad \textit{donde } s' = s|(l',r') \rightarrow (v_1,v_3), h' = h - [\![c_1]\!]_e.s & \\
\quad \textit{y } l',r' \textit{ son variables frescas} & \textit{en caso contrario}
\end{array}
$$

Se puede ver que la satisfacción de un predicado **trees**.$C.D$ en un estado se mantiene al intercambiar el orden en que ocurren las expresiones en C y en D, por lo que efectivamente podemos considerarlos multiconjuntos, en el sentido de una colección de elementos sin ningún orden particular.

Lema 54 *Sea P_r el conjunto de todas las permutaciones de $\{1,\ldots,r\}$, es decir funciones biyectivas del conjunto $\{1,\ldots,r\}$ en si mismo. Se cumple que:*

$$
\begin{array}{l}
(s,h) \models \mathbf{trees}.\{\!|c_1,\ldots,c_n|\!\}.\{\!|d_1,\ldots,d_m|\!\} \quad \textit{si y solo si} \\
(s,h) \models \mathbf{trees}.\{\!|c_{\pi_1(1)},\ldots,c_{\pi_1(n)}|\!\}.\{\!|d_{\pi_2(1)},\ldots,d_{\pi_2(m)}|\!\}
\end{array}
$$

para todos $(s,h) \in \mathsf{States}_a$, $\pi_1 \in P_n$ y $\pi_2 \in P_m$

En el primer caso de la definición de satisfacción de **trees** tenemos un árbol vacío, es decir sin punteros de entrada. El predicado **trees**.$\emptyset.D$ será válido en un *heap* vacío y cuando todos los punteros de salida válidos hayan sido *alcanzados* y por lo tanto los punteros en D sean todos **nil**. Esto, junto con el segundo caso, nos permite eliminar ocurrencias de punteros **nil** tanto en C como en D, ya que no influyen en la estructura definida. El tercer caso de la definición caracteriza la parcialidad y el *sharing* entre árboles. Para un predicado **trees**.$C.D$, un puntero que ocurra tanto en C como en D representa un corte en la definición recursiva del árbol y una vez alcanzado, se elimina una instancia (por si ocurre múltiples veces) de dicho puntero en ambos multiconjuntos C y D. El registro al que referencie, si fuera un puntero válido, podría quedar definido en otro predicado que se combine con el predicado **trees**.$C.D$ mediante la conjunción espacial $*$. El último caso de la definición permite, para un puntero c_1 distinto de **nil** y que no ocurren en el multiconjunto de salida, especificar el registro de memoria al que apunta y hacer recursión en la parte del *heap* sin este registro, con el predicado **trees** al que se le agregaron al multiconjunto de entrada expresiones que representan los punteros a los sub-árboles izquierdo y derecho del nodo apuntado por c_1. Representa el caso recursivo donde **trees**.$x \oplus C.D$ se puede reemplazar por $x \mapsto l', v', r' * \mathbf{trees}.l' \oplus r' \oplus C.D$ y donde $\oplus$ representa la inserción de un elemento en un multiconjunto.

Ejemplo 55 *El predicado* **trees**.$\{\!|x,y|\!\}.\emptyset$ *se satisface en estados como el de la figura 3.1. Como el multiconjunto de punteros de salida es vacío, no hay posibilidad de* sharing *o de estructuras parciales. Notar la independencia entre árboles referenciados por x e y.*

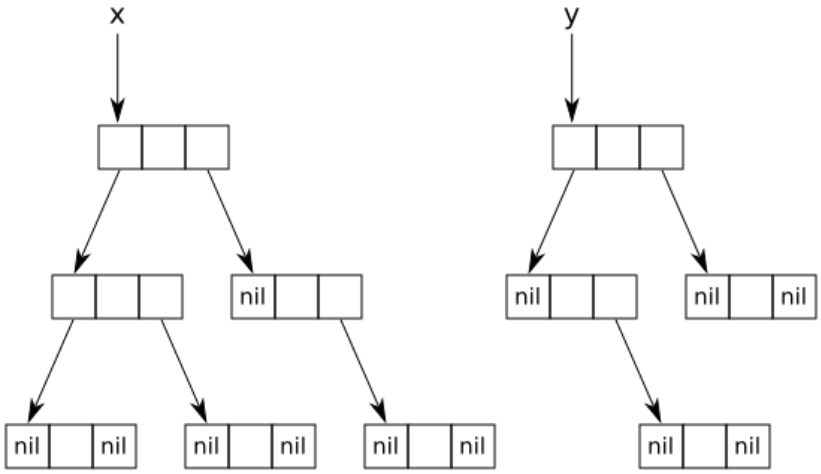


Figura 3.1: Estado de **trees**.$\{\!|x, y|\!\}$.$\emptyset$

Ejemplo 56 *En la figura 3.2 tenemos dos ejemplos de estados que satisfacen la fórmula* **trees**.$\{\!|x|\!\}$.$\{\!|z|\!\}$. *En la figura 3.2a vemos como la variable z es un puntero* dangling *que indica parcialidad. Cuando en el recorrido del árbol referenciado por x llegamos al puntero z simplemente dejamos de avanzar. Sin embargo, podría ocurrir que el puntero z sea alcanzado dos veces por caminos distintos de la estructura con raíz en x. Como z ocurre una sola vez en el multiconjunto de los punteros de salida, en la definición recursiva del predicado* **trees**, *la primera vez que se alcance a z ocurrirá un corte en la recursión al eliminarse de los punteros de salida, pero la segunda vez se continuará libremente determinando el árbol que tiene a z como raíz. Es el caso en el que z referencia un registro válido y se puede observar en la figura 3.2b. Notar que la estructura global definida es un grafo dirigido acíclico.*

Ejemplo 57 *En la figura 3.3a se representa un estado que satisface la fórmula* **trees**.$\{\!|x, y|\!\}$.$\{\!|z|\!\}$. *En este caso, el puntero z indica el* sharing *entre los árboles referenciados por x e y, apuntando a un registro válido que es la raíz de un nuevo árbol. Si consideráramos la fórmula* **trees**.$\{\!|x, y|\!\}$.$\{\!|z, z|\!\}$, *con una ocurrencia extra de la variable z en los punteros de salida, podríamos tener un estado como el de la figura 3.3b. En este caso z se vuelve* dangling *y no tenemos el* sharing *que teníamos antes, sino solo parcialidad en las estructuras.*

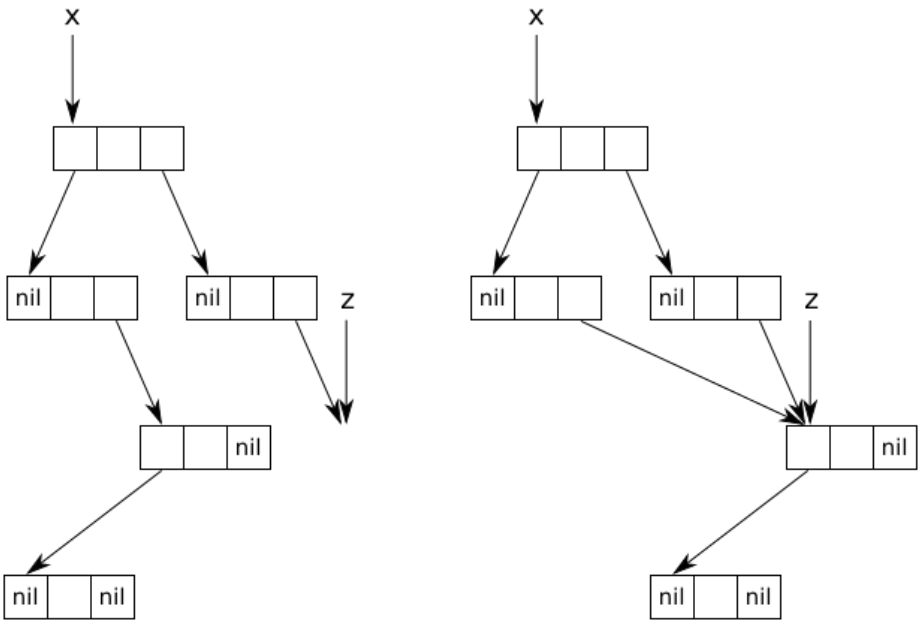
El predicado **trees** resulta útil para especificar las estructuras parciales que ocurren durante la ejecución de los ciclos y tiene buenas propiedades sintácticas. Su importancia es similar a la del predicado **lseg** en la verificación de programas sobre listas enlazadas. La relación con el predicado **tree** definido en la sección 2.4.2 es bastante directa:

Lema 58 *Dado un estado* $(s, h) \in \mathsf{States}$ *se cumple que:*

$$(s, h) \models \mathbf{trees}.\{\!|x_1, \ldots, x_n|\!\}.\emptyset$$

si y sólo si existen $t_1, \ldots, t_n \in \mathsf{Tree}$ *tales que*

$$(s, h) \models \mathbf{tree}.x_1.t_1 * \ldots * \mathbf{tree}.x_n.t_n$$



(a) (b)

Figura 3.2: Estados de **trees**.$\{\!|x|\!\}.\{\!|z|\!\}$

Para dar semántica a los comandos, procedemos de forma similar a la semántica concreta, aunque sobre el dominio de los *symbolic heaps*. La semántica intermedia se define sobre un reticulado completo D_a:

Definición 59 *Dado el conjunto de* symbolic heaps SH *y un elemento distinguido* $\top \notin \mathsf{SH}$*, definimos el reticulado* D_a *como:*

$$\mathsf{D}_a \doteq (\mathcal{P}(\mathsf{SH}) \cup \{\top\}, \subseteq)$$

con $\subseteq$ *la relación de inclusión de conjuntos que además cumple que para todo* $d \in \mathsf{D}_a$*,* $d \subseteq \top$

Definición 60 *Para cada comando atómico se define una relación de transición* $\rightsquigarrow \subseteq \mathsf{SH} \times \mathsf{SH}$ *según las siguientes reglas:*

Skip:

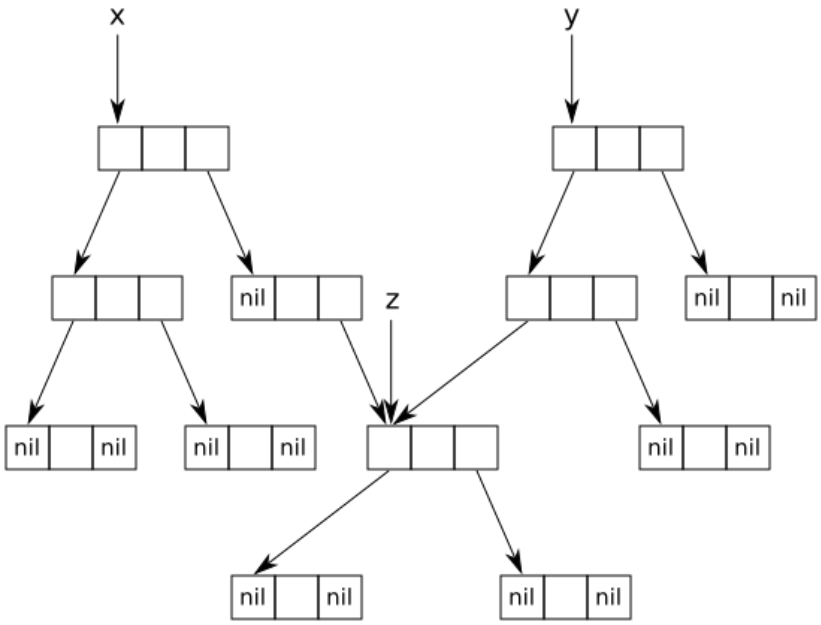
$$\Pi \mid \Sigma,\ \mathbf{skip} \ \rightsquigarrow\ \Pi \mid \Sigma$$

Asignación:

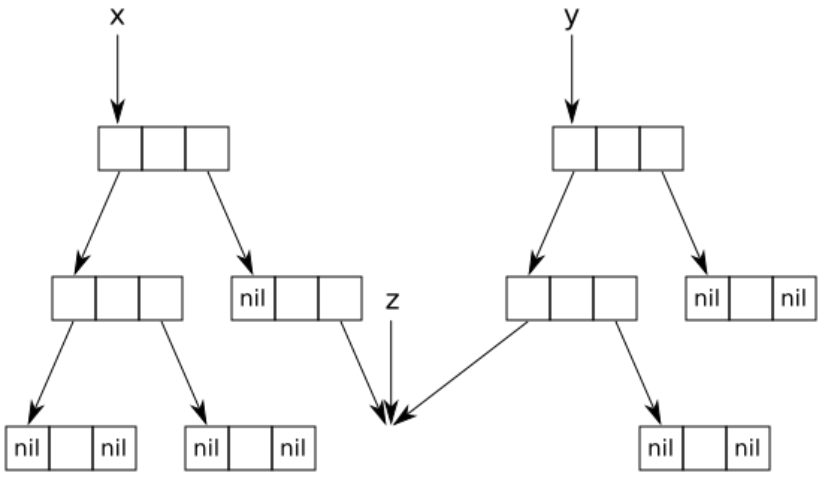
$$\Pi \mid \Sigma,\ x := e \ \rightsquigarrow\ \Pi' \wedge x = e_{/x \leftarrow x'} \mid \Sigma'$$

Construcción:

$$\Pi \mid \Sigma,\ x := \mathbf{new}(\vec{e}) \ \rightsquigarrow\ \Pi' \mid \Sigma' * x \mapsto \vec{e}_{/x \leftarrow x'}$$



(a) Estado de **trees**.$\{\!|x, y|\!\}.\{\!|z|\!\}$



(b) Estado de **trees**.$\{\!|x, y|\!\}.\{\!|z, z|\!\}$

Figura 3.3

Consulta:

$$\Pi \mid \Sigma * y \mapsto e_0, e_1, e_2,\ x := y.i \ \rightsquigarrow\ \Pi' \wedge x = e_{i/x \leftarrow x'} \mid \Sigma' * (y \mapsto e_0, e_1, e_2)_{/x \leftarrow x'}$$

Mutación:

$$\Pi \mid \Sigma * x \mapsto e_0, e_1, e_2,\ x.i := e \ \rightsquigarrow\ \Pi \mid \Sigma * x \mapsto e_0, ..., e_{i-1}, e, e_{i+1}, ..., e_2$$

Destrucción:

$$\Pi \mid \Sigma * x \mapsto \vec{e},\ \mathbf{free}(x) \ \rightsquigarrow\ \Pi \mid \Sigma$$

donde $x' \in \mathsf{Var}'$ *es una variable fresca,* Π' *denota* $\Pi_{/x \leftarrow x'}$ *y* Σ' *denota* $\Sigma_{/x \leftarrow x'}$.

Los comandos de consulta, mutación y destrucción requieren que el preestado explicite la existencia de la celda a ser dereferenciada. Para asegurar esto, definimos una función $\mathsf{rearr}.x$ que dada una variable de interés x aplica reglas de reescritura a cada *symbolic heap* que conforma el estado abstracto, intentando revelar la celda de memoria apuntada por x. En el caso que eso no sea posible, devuelve $\top$. Las reglas de reescritura intentan revelar una celda de memoria a través de igualdad de variables y el *unfolding* del predicado **trees**:

Definición 61 *El sistema de reescritura* $\Longrightarrow$ *se define por las siguientes reglas:*

Eq:

$$\frac{\Pi \mid \Sigma \vdash x = y}{\Pi \mid \Sigma * y \mapsto \vec{e} \implies \Pi \mid \Sigma * x \mapsto \vec{e}}$$

Unfold:

$$\frac{\Pi \mid \Sigma \vdash x = y \wedge y \neq \mathbf{nil} \wedge y \notin \mathcal{D}}{\Pi \mid \Sigma * \mathbf{trees}.y \oplus \mathcal{C}.\mathcal{D} \implies \Pi \mid \Sigma * x \mapsto l', v', r' * \mathbf{trees}.l' \oplus r' \oplus \mathcal{C}.\mathcal{D}}$$

donde l', v', r' *son variables frescas.*

Para dicho sistema de reescritura, la función $\mathsf{rearr}.x \in \mathsf{D}_a \to \mathsf{D}_a$ *se define como:*
$\mathsf{rearr}.x.d$ *es* $\top$ *cuando* $d = \top$ *o cuando existe algún* symbolic heap $sh \in d$ *tal que no es posible aplicar ninguna regla. En caso contrario,* $\mathsf{rearr}.x.d$ *es el conjunto de todos los* $sh \in d$ *a los que se les aplicó alguna de las reglas y por lo tanto quedaron con la variable* x *revelada.*

Las condiciones de aplicación de estas reglas requieren la habilidad de decidir sobre la validez de ciertos predicados, denotado con $\vdash$. A continuación damos una gramática para estas condiciones.

Definición 62 *La sintaxis de los predicados* Cond *está dada por la siguiente gramática:*

$$\mathsf{Cond} \qquad c ::= \ r \mid e \Subset m \mid e \notin m \mid noDangling.e \mid c \wedge c \mid c \vee c$$

donde $r \in \mathsf{Rel'}_a$, $e \in \mathsf{Expr'}_a$, $m \in \mathsf{Multiset}$.

Con la condición $noDangling.x$ se intenta expresar que x es un puntero *correcto*, es decir que o bien tiene asignada una dirección de memoria válida o bien tiene asignado el valor especial **nil**. La semántica del resto de las condiciones es estándar.

Definición 63 *La semántica de* $\models$ *de la definición 52, se extiende para incluir a las condiciones* Cond:[1]

$(s,h) \models e \Subset \{\!|e_1,...,e_n|\!\}$ *si y sólo si* $(s,h) \models e = e_i$, *para algún* $1 \leq i \leq n$
$(s,h) \models e \notin \{\!|e_1,...,e_n|\!\}$ *si y sólo si* $(s,h) \models e \neq e_i$, *para todo* $1 \leq i \leq n$
$(s,h) \models noDangling.e_1$ *si y solo si* $[\![e_1]\!]_e \in dom.h$ *o* $[\![e_1]\!]_e = \mathbf{nil}$
$(s,h) \models c \wedge c'$ *si y sólo si* $(s,h) \models c$ *y* $(s,h) \models c'$
$(s,h) \models c \vee c'$ *si y sólo si* $(s,h) \models c$ *o* $(s,h) \models c'$

A continuación, presentamos un pequeño y simple *theorem prover* sintáctico paras las condiciones Cond.

Definición 64 *El* theorem prover $\vdash$ *se define de la siguiente manera:*

$\Pi \restriction \Sigma \vdash e_1 = e_2$ *si* $e_1 \equiv e_2$ *o* $(\Pi \equiv \Pi' \wedge e_1 = e_3$ *y* $\Pi' \restriction \Sigma \vdash e_3 = e_2)$
$\Pi \restriction \Sigma \vdash e \neq n$ *si* $DF(e,n)$ *o* $\Pi \restriction \Sigma \vdash e = \mathbf{nil}$
o $(\Pi \restriction \Sigma \vdash e = n'$ *y no es* $n \equiv n')$
$\Pi \restriction \Sigma \vdash x \neq \mathbf{nil}$ *si* $DF(x, \mathbf{nil})$ *o* $(\Sigma \equiv \Sigma' * y \mapsto \vec{e}$ *y* $\Pi \restriction \Sigma' \vdash x = y)$
$\Pi \restriction \Sigma \vdash x \neq y$ *si* $DF(x,y)$ *o* $(\Sigma \equiv \Sigma' * \Sigma''$
y $\Pi \restriction \Sigma' \vdash noDangling.x$ *y* $\Pi \restriction \Sigma'' \vdash noDangling.y$
y $\Pi \restriction \Sigma \vdash x \neq \mathbf{nil} \vee y \neq \mathbf{nil})$
$\Pi \restriction \Sigma \vdash e \Subset \{\!|e_1,..,e_n|\!\}$ *si* $\Pi \restriction \Sigma \vdash e = e_i$ *para algún* i *tal que* $1 \leq i \leq n$
$\Pi \restriction \Sigma \vdash e \notin \{\!|e_1,..,e_n|\!\}$ *si* $\Pi \restriction \Sigma \vdash e \neq e_i$ *para todo* i *tal que* $1 \leq i \leq n$
$\Pi \restriction \Sigma \vdash c_1 \wedge c_2$ *si* $\Pi \restriction \Sigma \vdash c_1$ *y* $\Pi \restriction \Sigma \vdash c_2$
$\Pi \restriction \Sigma \vdash c_1 \vee c_2$ *si* $\Pi \restriction \Sigma \vdash c_1$ *o* $\Pi \restriction \Sigma \vdash c_2$
$\Pi \restriction \Sigma \vdash noDangling.e$ *si* $\Pi \restriction \Sigma \vdash e = \mathbf{nil}$ *o* $(\Sigma \equiv \Sigma' * x \mapsto \vec{e}$ *y* $\Pi \restriction \Sigma' \vdash x = e)$
o $(\Sigma \equiv \Sigma' * \mathbf{trees}.C.D$ *y* $\Pi \restriction \Sigma' \vdash e \Subset C \wedge e \notin D)$

donde n *y* n' *representan constantes numéricas,* $DF(x,y)$ *denota*

$$\Pi \equiv \Pi' \wedge x \neq y$$
$$o\ (\Pi \equiv \Pi' \wedge x = z\ y\ \Pi' \restriction \Sigma \vdash y \neq z)$$
$$o\ (\Pi \equiv \Pi' \wedge y = z\ y\ \Pi' \restriction \Sigma \vdash x \neq z)$$

y donde $\equiv$ *es la igualdad sintáctica (salvo conmutatividad en los casos de* Π *o* Σ *y simetría para las expresiones de igualdad y desigualdad).*

Todos los términos *que ocurren libres en la parte de la definición y que no ocurren en lo que se está definiendo están cuantificados existencialmente de manera implícita. Por ejemplo, la segunda parte de la disyunción de la definición de* $\Pi \restriction \Sigma \vdash e_1 = e_2$, *debe interpretarse como: existen* e_3 *y* Π' *tales que* $\Pi \equiv \Pi' \wedge e_1 = e_3$ *y* $\Pi' \restriction \Sigma \vdash e_3 = e_2$.

Se puede ver que el algoritmo definido para $\vdash$ es consistente y que las reglas de reescritura que definen $\mathsf{rearr}.x$ representan implicaciones semánticamente válidas.

Lema 65 (Consistencia de $\Longrightarrow$)

1. *Si* $\Pi \restriction \Sigma \vdash P$, *entonces* $(s,h) \models \Pi \restriction \Sigma$ *implica* $(s,h) \models P$ *para todos* s, h.

[1] Notar que la semántica para las relaciones $\mathsf{Rel'}_a$ ya fue dada y por lo tanto se omite.

2. *Si* $\Pi \mid \Sigma \Longrightarrow \Pi' \mid \Sigma'$, *entonces* $(s,h) \models \Pi \mid \Sigma$ *implica* $(s,h) \models \Pi' \mid \Sigma'$ *para todos* s, h.

Para la definición de la semántica intermedia de los comandos seguimos el mismo esquema que para la semántica concreta, adaptado las definiciones al nuevo dominio de *symbolic heaps*. Para los comandos atómicos de consulta, mutación y destrucción, que son los que podrían fallar en la semántica concreta debido a la manipulación de una dirección inválida dada por el valor de una variable x, ahora solo se ejecutan sobre estados que la función $\mathsf{rearr}.x$ transformó para que quedara explícito que x referencia a una dirección válida, por lo que no fallan en este contexto. Quien podría fallar ahora es $\mathsf{rearr}.x$.

Definición 66 *La función* $a^\Uparrow : D_a \to D_a$, *con* $a \in \mathsf{Comm}_a$ *un comando atómico se define como:*

$$a^\Uparrow.d \doteq \begin{cases} \top & \text{si } d = \top \\ \{sh' \in \mathsf{SH} \mid \text{existe } sh \in d \text{ tal que } (sh, a) \rightsquigarrow sh'\} & \text{en caso contrario} \end{cases}$$

Definición 67 *La semántica intermedia de los comandos* $[\![\ ]\!]_i \in \mathsf{Comm}_a \to D_a \to D_a$, *se define como:*

$$\begin{aligned} [\![a]\!]_i &\doteq a^\Uparrow \\ [\![a_x]\!]_i &\doteq a_x^\Uparrow \circ \mathsf{rearr}.x \\ [\![c_1; c_2]\!]_i &\doteq [\![c_2]\!]_i \circ [\![c_1]\!]_i \\ [\![\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 \textbf{fi}]\!]_i &\doteq ([\![c_1]\!]_i \circ \mathit{filter}_a.b) \cup_\top ([\![c_2]\!]_i \circ \mathit{filter}_a.\neg b) \\ [\![\textbf{while } b \textbf{ do } c_1 \textbf{od}\ ]\!]_i &\doteq \lambda d \cdot \mathit{filter}_a.\neg b.(\mu d' \cdot d \cup_\top ([\![c_1]\!]_i \circ \mathit{filter}_a.b).d') \end{aligned}$$

Donde a *es el comando de skip, asignación o construcción,* a_x *es el comando de consulta, modificación o destrucción de la variable* x, *la función* $\cup_\top : D_a \times D_a \to D_a$ *es la unión con* $\top$ *y se define de manera análoga a* $\cup_{\textbf{abort}}$ *de la semántica concreta.*

La función $\mathit{filter}_a.b : D_a \to D_a$, *con* $b = cnj_1 \vee ... \vee cnj_n$ *donde cada* cnj_i *es una conjunción de relaciones, se define como:*

$$\mathit{filter}_a.b.d \doteq \begin{cases} \top & \text{si } d = \top \\ \bigcup\limits_{1 \le i \le n} \{\Pi \wedge cnj_i \mid \Sigma \mid \Pi \mid \Sigma \in d \text{ y } \Pi \mid \Sigma \not\vdash \neg cnj_i\} & \text{en caso contrario} \end{cases}$$

Si bien establecimos en el lema 65 que el algoritmo para $\vdash$ es consistente, claramente no es completo. Por lo tanto en algunas ocasiones $\mathsf{rearr}.x$ no es capaz de revelar la existencia de una celda de memoria particular con la consecuencia de que ciertas ejecuciones de la semántica intermedia dan como resultado $\top$ aún cuando no existe una violación de memoria según la semántica concreta.

Ejemplo 68 *En el* symbolic heap *sh:*

$$x \neq \textbf{nil} \mid \textbf{trees}.\{\!|x|\!\}.\{\!|y|\!\} * \textbf{trees}.\{\!|y|\!\}.\emptyset$$

$\mathsf{rearr}.x$ *no puede revelar la variable* x, *ya que por un lado no hay ningún predicado* $\mapsto$ *en la parte espacial de sh (i.e. no puede aplicar la regla* Eq*) y por el otro para aplicar la regla* Unfold *necesitaría:*

- *para aplicar en* **trees**.$\{\!|x|\!\}.\{\!|y|\!\}$*, poder probar que* $x \notin \{\!|y|\!\}$*, pero esto no sucede, ya que hay estados que satisfacen sh pero no la condición.*
- *o bien, para aplicar en* **trees**.$\{\!|y|\!\}.\emptyset$*, poder probar que* $x = y$*, que tampoco es cierto.*

En la semántica concreta es claro que una de las dos condiciones anteriores es válida y por lo tanto cualquier estado concreto que satisfaga sh cumple que el puntero x tiene una dirección de memoria válida. Una posibilidad para resolver esta limitación sería enriquecer el estado sh reemplazándolo por el conjunto de dos estados equivalente a sh dado por $\mathsf{filter}_a.(x = y \vee x \neq y).\{sh\}$ *y donde en ambos estados se puede revelar x. Este tipo de predicados aparece recurrentemente en la especificación de los programas y como veremos en la próxima sección, una regla de abstracción puede transformar la parte espacial de sh en* **trees**.$\{\!|x|\!\}.\emptyset$*. Curiosamente éste es un estado abstracto en donde sí se pueden probar las condiciones para la aplicación de la regla de* Unfold *y por lo tanto revelar la variable x. Queda como moraleja que no siempre las fórmulas más débiles resultan en mayor dificultad para garantizar las condiciones de prueba impuestas por* rearr*.*

Para establecer formalmente la relación entre las semánticas concreta e intermedia, definimos una función de concretización.

Definición 69 *La función de concretización* $\gamma : D_a \to D_c$ *se define como:*

$$\gamma.d \doteq \begin{cases} \textbf{abort} & \text{si } d = \top \\ \{(s,h) \mid \exists\, \Pi \,|\, \Sigma \in d \;\cdot\; (s,h) \models \Pi \,|\, \Sigma\} & \text{en caso contrario} \end{cases}$$

Teorema 70 *La semántica intermedia es una sobre-aproximación consistente de la semántica concreta:*

$$[\![p]\!]_c.(\gamma.d) \subseteq \gamma.([\![p]\!]_i.d)$$

para todo $p \in \mathsf{Comm}_a, d \in D_a$*.*

3.3. Semántica Abstracta

La semántica intermedia es ejecutable pero no posee ningún mecanismo que facilite el cálculo de invariantes de ciclos y garantice la terminación. En general, la ejecución de un ciclo que dereferencia una variable x genera estados abstractos que contienen una cantidad arbitraria de *symbolic heaps* con un número cada vez mayor de términos de la forma $x' \mapsto y', z', w'$.

Ejemplo 71 *En un algoritmo que iterativamente recorre el vínculo izquierdo de los nodos de un árbol binario de búsqueda con el objetivo de encontrar su mínimo valor:*

$p := x;$
while $p \neq$ **nil do**
  $m := p.1;$
  $p := p.0$
od

la ejecución de la semántica intermedia, partiendo de la precondición esperada $\{\mathbf{true} \,|\, \mathbf{trees}.\{\!|x|\!\}.\emptyset\}$*, genera estados de la forma:*[2]

$$\begin{array}{l}\{p = x \,|\, \mathbf{trees}.\{\!|x|\!\}.\emptyset,\\ \quad \mathbf{true} \,|\, x \mapsto p, m, r_1' * \mathbf{trees}.\{\!|p, r_1'|\!\}.\emptyset,\\ \quad \mathbf{true} \,|\, x \mapsto l_1', v_1', r_1' * l_1' \mapsto p, m, r_2' * \mathbf{trees}.\{\!|p, r_1', r_2'|\!\}.\emptyset,\\ \quad \mathbf{true} \,|\, x \mapsto l_1', v_1', r_1' * l_1' \mapsto l_2', v_2', r_2' * l_2' \mapsto p, m, r_3' * \mathbf{trees}.\{\!|p, r_1', r_2', r_3'|\!\}.\emptyset,\\ \quad \ldots\end{array}$$

causando la divergencia en su cómputo.

Para manejar esta situación, definimos una función de abstracción, $\mathsf{abs} : \mathsf{D}_a \to \mathsf{D}_a$, que intenta simplificar las fórmulas que componen un estado, reemplazando información concreta sobre la forma del heap, por otra más abstracta pero también útil. El objetivo de esta función es facilitar la convergencia del cálculo de punto fijo que representa la semántica de un ciclo. Nuestra semántica abstracta $[\![\ ]\!]_a$ entonces, aplica la función abs a la entrada y luego de cada iteración de un ciclo.

Definición 72 *La semántica abstracta de los comandos* $[\![\ ]\!]_a \in \mathsf{Comm}_a \to \mathsf{D}_a \to \mathsf{D}_a$*, se define como:*

$$\begin{array}{rl}[\![p]\!]_a & \doteq [\![p]\!]_i\\ [\![\mathbf{while}\ b\ \mathbf{do}\ c\ \mathbf{od}\ ]\!]_a & \doteq (\lambda d \cdot \mathsf{filter}_a.\neg b.(\mu d' \cdot \mathsf{abs}(d \cup_\top ([\![c]\!]_a \circ \mathsf{filter}_a.b).d'))) \circ \mathsf{abs}\end{array}$$

donde p representa a los comandos atómicos, la secuenciación y la alternativa.

De esta forma, aunque el dominio de la semántica abstracta sigue siendo D_a, la semántica de un ciclo se calcula sobre un subconjunto de estados con características que, como veremos más adelante, aseguran la convergencia del cálculo de punto fijo.

La función abs depende de la aplicación, sobre cada *symbolic heap* que compone un estado abstracto, de un conjunto de reglas de reescritura organizadas en tres etapas.

Definición 73 *Las reglas de reescritura de la primera etapa son:*

ReduceNull1:

$$\frac{\Pi \,|\, \Sigma \vdash e = \mathbf{nil}}{\Pi \,|\, \Sigma * \mathbf{trees}.(e \oplus \mathcal{C}).\mathcal{D} \implies \Pi \,|\, \Sigma * \mathbf{trees}.\mathcal{C}.\mathcal{D}}$$

ReduceNull2:

$$\frac{\Pi \,|\, \Sigma \vdash e = \mathbf{nil}}{\Pi \,|\, \Sigma * \mathbf{trees}.\mathcal{C}.(e \oplus \mathcal{D}) \implies \Pi \,|\, \Sigma * \mathbf{trees}.\mathcal{C}.\mathcal{D}}$$

[2] En rigor de verdad, la parte pura de los estados contendrían igualdades entre las variables x, p y m con variables primadas, así como otras igualdades y desigualdades entre **nil** y variables primadas que representaban relaciones sobre p y m, anteriores a que le fueran asignados nuevos valores. La parte espacial contendría, en el lugar de las variables p y m, sus variables primadas relacionadas por igualdad en la parte pura. Para simplificar la exposición, se eliminaron variables primadas irrelevantes y se sustituyeron por p y m las de la parte espacial. De hecho, este procedimiento es uno de los pasos que contiene la definición de abs que veremos más adelante.

ReduceTree:

$$\frac{\Pi \mid \Sigma \vdash e = e'}{\Pi \mid \Sigma * \mathbf{trees}.(e \oplus \mathcal{C}).(e' \oplus \mathcal{D}) \implies \Pi \mid \Sigma * \mathbf{trees}.\mathcal{C}.\mathcal{D}}$$

DivideTree:

$$\frac{\textit{cuando } \mathcal{E} \neq \emptyset}{\Pi \mid \Sigma * \mathbf{trees}.(e \oplus \mathcal{E}).\emptyset \implies \Pi \mid \Sigma * \mathbf{trees}.\{\!|e|\!\}.\emptyset * \mathbf{trees}.\mathcal{E}.\emptyset}$$

AbsPointer:

$$\frac{\Sigma \doteq \Sigma' * x \mapsto l, v, r \qquad \Pi \mid \Sigma \vdash x \neq l \wedge x \neq r}{\Pi \mid \Sigma \implies \Pi \mid \Sigma' * \mathbf{trees}.\{\!|x|\!\}.\{\!|l, r|\!\}}$$

AbsTree:

$$\frac{\Pi \mid \Sigma \vdash e = e' \qquad \Pi \mid \Sigma \vdash \mathit{noDangling}.e'' \quad \textit{para todo } e'' \textit{ en } \mathcal{F}}{\Pi \mid \Sigma * \mathbf{trees}.\mathcal{C}.(e \oplus \mathcal{D}) * \mathbf{trees}.(e' \oplus \mathcal{E}).\mathcal{F} \implies \Pi \mid \Sigma * \mathbf{trees}.(\mathcal{C} \uplus \mathcal{E}).(\mathcal{D} \uplus \mathcal{F})}$$

Las reglas ReduceNull1-2 y ReduceTree eliminan puntos de entrada y salida que no aportan información sobre el *heap*. La regla DivideTree *normaliza* un árbol con varios puntos de entrada pero sin puntos de salida, en varios árboles con un solo punto de entrada. Estas primeras reglas no pierden información, o sea, representan equivalencias, más que implicaciones semánticas. La regla AbsPointer abstrae predicados $\mapsto$ en predicados **trees**, olvidando así el número de nodos que conforman la estructura arbórea. La regla AbsTree combina árboles olvidando puntos intermedios.

Ejemplo 74 *Retomando el ejemplo 71, si aplicamos las reglas AbsPointer, DivideTree y AbsTree, en la segunda iteración del cálculo del punto fijo obtenemos el invariante:*

$$\{p = x \mid \mathbf{trees}.\{\!|x|\!\}.\emptyset, \quad \mathbf{true} \mid \mathbf{trees}.\{\!|x|\!\}.\{\!|p|\!\} * \mathbf{trees}.\{\!|p|\!\}.\emptyset\}$$

Las condiciones de aplicación de las reglas impiden la formación de ciclos dentro de la estructura, preservando la característica de árbol. En el caso de AbsPointer esto es trivial ya que se exige que $\Pi \mid \Sigma \vdash x \neq l \wedge x \neq r$. En el caso de AbsTree, la condición sobre las expresiones e'' de $\mathcal{F}$ es que en la parte espacial que no incluye a los predicados **trees** que se van a combinar, se cumpla que e'' representa un puntero con una dirección válida o que sea **nil**. Esto asegura que todos los punteros de la estructura arbórea generada a partir de $\mathcal{C}$ en el predicado **trees** son distintos a los elementos de $\mathcal{F}$, pues aquéllos referencian registros en un *heap* disjunto al *heap* en que éstos referencian registros o son **nil**. Notar que la condición no excluye la posibilidad de que, además de los **nil**s, haya elementos con valores repetidos en $\mathcal{F}$.

3.3.1. Relevancia de variables y abstracción

Comencemos definiendo una terminología. Usamos $T.\mathcal{C}.\mathcal{D}$ para referirnos a un predicado $\mathbf{trees}.\mathcal{C}.\mathcal{D}$ o a un predicado $x \mapsto l, v, r$ con $\mathcal{C} = \{\!|x|\!\}$ y $\mathcal{D} = \{\!|l, r|\!\}$. Decimos que dos términos $T_1.\mathcal{C}.\mathcal{D}$ y $T_2.\mathcal{E}.\mathcal{F}$ de un *symbolic heap* forman un *enlace de cadena* si existe $x \in \mathcal{D}$ tal que $x \in \mathcal{E}$. Una sucesión de términos

$T_1, T_2, \ldots, T_n$ forman una *cadena* si T_i y T_{i+1} forman un enlace de cadena, para todo $0 \leq i < n$.

La aplicación de las dos últimas reglas de abstracción presentadas en la definición 73 implican pérdida de información sobre el *heap* de dos formas distintas: por un lado la información concreta sobre una celda referenciada por cierta variable (en el caso de la variable x en la regla AbsPointer), y por otro, la información sobre una variable que hace de nexo en un enlace de cadena (en el caso de la expresión e en la regla AbsTree). Esta pérdida de información puede resultar en la imposibilidad de continuar con el análisis si la variable abstraída es dereferenciada en un punto de ejecución posterior, ya sea porque no es posible garantizar las condiciones de las reglas de rearr, o porque se pierde todo rastro de la variable como punto intermedio de la estructura.

En general, en los *shape analysis* derivados de la *Separation Logic*, las reglas de abstracción se aplican sobre una variable siempre que esté cuantificada existencialmente, pero un criterio análogo resulta inadecuado en nuestro caso. Por un lado puede resultar muy fuerte, causando demasiada pérdida de información. Por otro lado, puede resultar muy débil, dando lugar a fórmulas innecesariamente precisas y por lo tanto, invariantes grandes. Llevar la cuenta de una cadena con dos, tres o más enlaces de cadena no parece un problema si se trata de estructuras lineales. Pero en estructuras multiligadas, dadas las múltiples combinaciones posibles, el número de fórmulas necesarias para describir esta cadena crece exponencialmente. Aunque desde el punto de vista de la corrección ésto no es un problema, mantener estados reducidos agiliza el análisis y permite una mejor comprensión sobre los invariantes y postcondiciones obtenidos.

En nuestro análisis, la aplicación de las reglas de abstracción son relativas a un *nivel de relevancia* asignado a cada variable. El nivel de una variable en cierto punto de ejecución depende del tipo de comandos que la incluyen y que van a ser ejecutados a partir de este punto. Se lleva a cabo un un análisis estático simple relev sobre un programa que devuelve una función $f : \mathsf{Var} \to \mathbb{Z}$ que asigna a cada variable de programa un valor que representa la información que se va a necesitar de ella. Si el primer comando en el que ocurre una variable y es, por ejemplo, el de consulta $x := y.i$, la necesidad de información sobre y es alta, ya que se requiere poder acceder a un registro de la memoria apuntado por y. Sin embargo, si el comando anterior es el primero en el que ocurre la variable x, la necesidad de información sobre x es nula, ya que el valor que tuviera antes se *pisará* con uno nuevo después de este comando.

Definición 75 *Pensando un programa como una secuencia de comandos, la función* $\mathsf{relev} \in [\mathsf{Comm}_a] \to (\mathsf{Var} \to \mathbb{Z}) \to (\mathsf{Var} \to \mathbb{Z})$ *actualiza una función inicial* f*, de la siguiente manera:*

$$
\begin{aligned}
\mathsf{relev}.(skip; cs).f &\doteq \mathsf{relev}.cs.fs \\
\mathsf{relev}.(x := e; cs).f &\doteq (\mathsf{relev}.cs.fs \,|\, x \to 0)\!\uparrow\! y \to 2 \quad \textit{para todo } y \in e \\
\mathsf{relev}.(x := y.i; cs).f &\doteq (\mathsf{relev}.cs.f \,|\, x \to 0)\!\uparrow\! y \to 3 \\
\mathsf{relev}.(x.i := e; cs).f &\doteq (\mathsf{relev}.cs.f\!\uparrow\! x \to 4)\!\uparrow\! y \to 2 \quad \textit{para todo } y \in e \\
\mathsf{relev}.(\mathbf{free}(x); cs).f &\doteq \mathsf{relev}.cs.f\!\uparrow\! x \to 3 \\
\mathsf{relev}.(x := \mathbf{new}(\vec{e}); cs).f &\doteq (\mathsf{relev}.cs.f \,|\, x \to 0)\!\uparrow\! y \to 2 \quad \textit{para todo } y \in \vec{e} \\
\mathsf{relev}.(\mathbf{if}\ b\ \mathbf{then}\ cs_1 \mathbf{else}\ cs_2 \mathbf{fi}; cs).f &\doteq \\
(\mathsf{relev}.(cs_1 +\!\!+ cs).f \ \text{máx}\ \mathsf{relev}.(cs_2 +\!\!+ cs).f)\!\uparrow\! y \to 2 &\quad \textit{para todo } y \in b \\
\mathsf{relev}.(\mathbf{while}\ b\ \mathbf{do}\ cs_1 \mathbf{od}\ ; cs).f &\doteq \\
(\mathsf{relev}.(cs_1 +\!\!+ cs).f \ \text{máx}\ \mathsf{relev}.cs.f)\!\uparrow\! y \to 2 &\quad \textit{para todo } y \in b \\
\mathsf{relev}.\epsilon.f &\doteq f
\end{aligned}
$$

donde ϵ *es la secuencia vacía,* $+\!\!+$ *es concatenación, y los operadores* $\uparrow$ *y* máx *se definen como:*

$$(f{\uparrow}y \to n).x \doteq \begin{cases} f.x & si\ x \neq y\ o\ f.x \geq n \\ n & en\ caso\ contrario \end{cases}$$

$$(f \operatorname{máx} g).x \doteq \begin{cases} f.x & si\ f.x \geq g.x \\ g.x & en\ caso\ contrario \end{cases}$$

Entonces, el *nivel de relevancia* de x para $\Pi \upharpoonright \Sigma$ está dado por el mayor valor de la función de relevancia aplicada a las variables de programa de la clase de equivalencia de x, dada por las relaciones de igualdad en Π. El nivel 1 se reserva para las variables de la precondición y las variables cuantificadas tendrían un nivel de -1, siempre que no estén en la misma clase de equivalencia que alguna variable de programa con valor mayor.

Definición 76 *Dado un programa con su precondición* $PRE \in \mathsf{SH}$*. Sea* $\mathsf{V}_{PRE} \subseteq \mathsf{Var}$ *el conjunto de variables (no primadas) libres de la precondición del programa. La función* $g \in \mathsf{Var} \to \mathbb{Z}$ *se define como:*

$$g.x\ es\ 1\ si\ x \in \mathsf{V}_{PRE},\ y\ es\ 0\ en\ caso\ contrario$$

Dados un symbolic heap $sh \in \mathsf{SH}$*, una secuencia de comandos* $cs \in [\mathsf{Comm}_a]$*, una variable* $x \in \mathsf{Var} \cup \mathsf{Var'}$*, una expresión* $e \in \mathsf{Expr'}_a$*, el* nivel de relevancia individual *(denotado como n.r.i.) de* e *en* cs *se define como:*

$$n.r.i.(e)\ es\ \mathsf{relev}.cs.g.e\ si\ e \in \mathsf{Var}\ y\ es\ -1\ en\ caso\ contrario$$

el nivel de relevancia *de* x *en* cs *para* sh *se define como:*

$$\operatorname{máx}(\{n.r.i.(z) \mid\ existe\ z\ que\ cumple\ sh \vdash x = z\})$$

Antes de aplicar las reglas de abstracción sobre cada *symbolic heap* que conforma un estado abstracto, se calcula el nivel de relevancia de cada variable respecto a los comandos que resten ejecutar. En nuestro análisis, el nivel de relevancia requerido por las reglas queda parametrizado, permitiendo graduar el nivel de abstracción. En nuestra implementación, para la regla AbsPointer, el parámetro p establece el nivel de relevancia de la variable x a partir del que la regla deja de aplicarse. Con niveles de relevancia menores a ese límite p sí se aplica la regla. Para AbsTree, el parámetro t indica que la regla se aplica solamente cuando todas las variables que podrían definir un enlace de cadena entre los dos predicados **trees** a abstraer tienen un nivel de relevancia menor a t. En los experimentos obtuvimos los mejores resultados con un valor de 2 para ambos parámetros, aunque algunos algoritmos permiten límites más altos, logrando invariantes más compactos y menores tiempos de ejecución.

3.3.2. Terminación

Las variables cuantificadas tienen un rol determinante en la terminación del análisis inducido por $[\![\ ]\!]_a$, ya que permiten generar un conjunto infinito de fórmulas como imagen de abs. El predicado **trees** también podría ser responsable de una cantidad arbitraria de fórmulas, aún solo con una cantidad finita de

variables de programa, debido a que los multiconjuntos pueden contener ocurrencias múltiples de una misma expresión. En circunstancias donde $img.\mathsf{abs}$ es infinito, el mínimo punto fijo que representa la semántica de un ciclo no siempre puede computarse efectivamente.

Sin embargo, abs consta de ciertas reglas, en la segunda y tercera etapa, que acotan el número de variables cuantificadas y garantizan la finitud de su imagen, y por lo tanto la convergencia del análisis.

Definición 77 *Las reglas de reescritura de la segunda etapa son:*

EqElimination1:

$$\frac{\textit{cuando } n.r.i.(x) < 1}{\Pi \wedge x = e \mid \Sigma \implies (\Pi \mid \Sigma)_{/x \leftarrow e}}$$

EqElimination2:

$$\frac{\textit{cuando no ocurren variables en } e_1, e_2}{\Pi \wedge e_1 = e_2 \mid \Sigma \implies \Pi \mid \Sigma}$$

NeqElimination:

$$\frac{\textit{cuando } n.r.i.(e_1) < 1 \textit{ y } n.r.i.(e_2) < 1 \textit{ o alguna es cuantificada}}{\Pi \wedge e_1 \neq e_2 \mid \Sigma \implies \Pi \mid \Sigma}$$

Garbage1:

$$\Pi \mid \Sigma * x' \mapsto l, v, r \implies \Pi \mid \Sigma * \mathbf{junk}$$

Garbage2:

$$\frac{\textit{cuando } x' \textit{ no ocurre en predicados de } \mapsto \textit{ en } \Sigma}{\Pi \mid \Sigma * \mathbf{trees}.(x' \oplus \mathcal{C}).\mathcal{D} \implies \Pi \mid \Sigma * \mathbf{true}}$$

Garbage3:

$$\frac{\textit{cuando } x' \textit{ no ocurre en predicados de } \mapsto \textit{ en } \Sigma}{\Pi \mid \Sigma * \mathbf{trees}.\mathcal{C}.(x' \oplus \mathcal{D}) \implies \Pi \mid \Sigma * \mathbf{true}}$$

Garbage4:

$$\Pi \mid \Sigma * \mathbf{trees}.\mathcal{C}.(e \oplus e \oplus \mathcal{D}) \implies \Pi \mid \Sigma * \mathbf{true}$$

Garbage5:

$$\Pi \mid \Sigma * \mathbf{trees}.(e \oplus e \oplus \mathcal{C}).\mathcal{D} \implies \Pi \mid \Sigma * \mathbf{true}$$

donde $x \in \mathsf{Var} \cup \mathsf{Var'}$ *y como es usual* $x' \in \mathsf{Var'}$.

La regla EqElimination1 substituye en un *symbolic heap*, variables cuantificadas o variables de programa para las que no hay necesidades de información en el programa que resta ejecutar, por su igual según alguna relación de igualdad en la parte pura. Notar que luego de la aplicación exhaustiva de esta regla, no deberían quedar variables cuantificadas en relaciones de igualdad de la parte pura de un *symbolic heap*. La regla EqElimination1 simplemente elimina igualdades triviales entre constantes y la regla NeqElimination elimina desigualdades. Es importante que esta última regla se aplique después de EqElimination1, ya

que de esta manera las variables eliminadas en las desigualdades no ocurren en ninguna igualdad de la parte pura y por lo tanto no perdemos información importante.

Como las reglas de aplican exhaustivamente por etapas, las reglas de la segunda etapa Garbage1-3 eliminan los punteros y predicados **trees** que involucran variables cuantificadas y que quedaron sin ser abstraídas por las reglas de la primera etapa, por la imposibilidad de garantizar una estructura de árbol. De la misma misma manera, las reglas Garbage4-5 eliminan predicados **trees** que contienen múltiples ocurrencias de expresiones, ya sea entre los punteros de entrada o los punteros de salida. Que no se hayan abstraído en la primera etapa implica que no pudieron probarse iguales a **nil** o a alguna otra expresión del otro multiconjunto del predicado. En el caso de las ocurrencias múltiples entre los punteros de salida, esto podría indicar que el predicado **trees** representa en realidad un *dag* y nuestro análisis encuentra un límite en el manejo de este tipo de estructuras. Por otro lado, en el caso de las ocurrencias múltiples de una expresión e entre los punteros de entrada, para que la fórmula sea consistente debería suceder que e sea **nil** o que sea igual a alguna expresión entre los punteros de salida, pero no pudo probarse ninguna de las dos opciones en la etapa anterior, por lo que no hay mucho más para hacer. Estas reglas generan predicados **junk** o **true** indicando la posible presencia de *basura* en el *heap*

Finalmente, en la tercera etapa se aplican reglas que acotan el tamaño de los *symbolic heaps*, eliminando términos repetidos por propiedades de elemento neutro, nilpontencia o idempotencia. Se acumula toda la *basura* en un solo predicado **junk** o **true** y se limita la cantidad de términos puros. También se eliminan múltiples ocurrencias de $\mathbf{trees}.\mathcal{C}.\mathcal{D}$ ya que la única circunstancia en que ello no incurre en una inconsistencia es cuando, ignorando los valores **nil**, los valores representados por los multiconjuntos $\mathcal{C}$ y $\mathcal{D}$ son los mismos y por lo tanto $\mathbf{trees}.\mathcal{C}.\mathcal{D} = \mathbf{emp}$.

Definición 78 *Las reglas de reescritura de la tercera etapa son:*

AndZero:

$$\Pi \wedge \mathbf{true} \mid \Sigma \implies \Pi \mid \Sigma$$

SpatialZero:

$$\Pi \mid \Sigma * \mathbf{emp} \implies \Pi \mid \Sigma$$

TreesNilpotence:

$$\Pi \mid \Sigma * \mathbf{trees}.\mathcal{C}.\mathcal{D} * \mathbf{trees}.\mathcal{C}.\mathcal{D} \implies \Pi \mid \Sigma$$

AndIdempotence:

$$\Pi \wedge r \wedge r \mid \Sigma \implies \Pi \wedge r \mid \Sigma$$

SpatialIdempotence1:

$$\Pi \mid \Sigma * \mathbf{true} * \mathbf{true} \implies \Pi \mid \Sigma * \mathbf{true}$$

SpatialIdempotence2:

$$\Pi \mid \Sigma * \mathbf{junk} * \mathbf{junk/true} \implies \Pi \mid \Sigma * \mathbf{junk}$$

Como últimos pasos de abs, se unifican los *symbolic heaps* que solo difieren en el nombre de las variables cuantificadas y se eliminan los que se pueden probar inconsistentes. Si bien en términos de correctitud esto no es necesario ya que un *symbolic heap* inconsistente no pude ser satisfecho por ningún estado concreto, su eliminación nos permite obtener una semántica más ajustada y asegurar la finitud de la imagen de abs.

Definición 79 *Decimos que un* symbolic heap $\Pi \mid \Sigma \in \mathit{SH}$ *es inconsistente si sucede alguna de las siguientes opciones:*

1. *Existen* $e_1, e_2 \in \mathit{Expr'}_a$ *tales que* $\Pi \mid \Sigma \vdash e_1 = e_2 \wedge e_1 \neq e_2$
2. *Existe* n *constante numérica tal que* $\Pi \mid \Sigma \vdash \mathit{noDangling}.n$

Notar que el primer punto de la definición anterior garantiza que no pueden haber múltiples ocurrencias de términos $x \mapsto -, -, -$ con la misma variable x en la parte espacial del *symbolic heap*.

Con todas las reglas de abstracción de cada etapa presentadas, podemos finalmente definir la función abs.

Definición 80 *La función de abstracción* $\mathsf{abs} \in D_a \to D_a$ *se define como:*

$$\mathsf{abs}(d) \doteq \begin{cases} \top & \text{si } d = \top \\ \mathsf{rmEq}(\{\, \Pi' \mid \Sigma' \mid \exists\, \Pi \mid \Sigma \in d \;\cdot\; \Pi \mid \Sigma \Longrightarrow^* \Pi' \mid \Sigma' & \\ \qquad y\ \Pi' \mid \Sigma' \text{ no es inconsistente } \}) & \text{en caso contrario} \end{cases}$$

donde $\Longrightarrow^*$ *consiste en la aplicación de cero o más reglas de reescritura* $\Longrightarrow$ *presentadas en las definiciones 73, 77 y 78, de manera exhaustiva en cada etapa*[3] *antes de continuar con la siguiente, y donde* rmEq *es una función que para un conjunto de* symbolic heaps $S \subseteq \mathit{SH}$ *devuelve un conjunto con un representante de cada clase de equivalencia dada por renombre de variables cuantificadas de los elementos en* S*, donde en el representante se hizo un renombre de manera que sólo ocurren las primeras* j *variables primadas* $x'_1, \ldots, x'_j$ *de una enumeración fija de* $\mathit{Var'}$*:* $x'_1, \ldots, x'_j, x'_{j+1}, \ldots$

Vimos que en la parte pura de un *symbolic heap* no pueden quedar variables cuantificadas luego de la aplicación de las reglas EqElimination1 y NeqElimination de la segunda etapa. Junto a la aplicación de la regla AndIdempotence de la tercera etapa y debido a que tenemos una cantidad finita de variables de programa y de constantes numéricas, la cantidad de posibles partes puras que pueden tener los *symbolic heaps* que queden luego de la aplicación de abs está acotada. Por otro lado, la regla Garbage1 de la segunda etapa garantiza que no aparecen variables cuantificadas como puntero de términos de $\mapsto$, por lo que, junto con el chequeo de no inconsistencia según la definición 79, tenemos una cantidad acotada de términos $\mapsto$ en la parte espacial de cualquier *symbolic heap* sh de la imagen de abs. Si $E_{\mapsto}$ es el conjunto de expresiones que ocurren en términos de $\mapsto$ de sh, las reglas Garbage2-3 aseguran que no ocurren variables

[3] Y de manera determinística también. Aunque no se haya explicitado el orden y la forma exacta de aplicar las reglas, el algoritmo que implementa las reescrituras es determinístico, de manera que $\Longrightarrow^*$ es, además de una relación, una función. Enseguida se discute respecto al orden de aplicación de las reglas.

cuantificadas en predicados **trees** que no estén en $E_{\mapsto}$, por lo que la cantidad de variables cuantificadas que ocurren en la parte espacial de sh está acotada. Luego, debido a aplicación de la función rmEq, el conjunto de expresiones de $\mathsf{Expr'}_a$ que pueden ocurrir en sh es finito y por consecuencia el conjunto de términos de $\mapsto$ también. Las reglas Garbage4-5 garantizan que no aparezcan expresiones repetidas en los multiconjuntos de un predicado **trees** y por tanto que el conjunto de términos **trees** que pueden ocurrir en sh es finito. Finalmente, junto con las reglas TreesNilpotence y SpatialIdempotence1-2 se evita la presencia de términos repetidos en la parte espacial y, con lo que vimos de la parte pura, demostramos la finitud de la imagen de abs.

Lema 81 (Terminación de abs**)**

1. *img.*abs *es finita.*
2. *El conjunto de reglas de abstracción es fuertemente normalizante, es decir, cualquier secuencia de reescrituras eventualmente termina.*

Las reglas de abstracción no son confluentes, i.e. la forma normal obtenida después de una secuencia de reescrituras no es única. El orden de aplicación de las reglas de la primera etapa es relevante pues la aplicación de la regla AbsTree podría causar mayor o menor pérdida de información cuando se aplica antes o después de la regla DivideTree y esto podría conllevar a que no se cumplan las condiciones necesarias para la aplicación de otras reglas. Además tiene sentido que se aplique AbsTree después de las reglas ReduceNull1-2 y ReduceTree para que la expresion e (o e') no se pueda haber simplificado de manera trivial y forme realmente un enlace de cadena de los predicados **trees** a abstraer. En la segunda etapa es importante la aplicación de la regla EqElimination1 antes que las demás, en especial NeqElimination y Garbage1-3, que se refieren a variables cuantificadas. En conclusión, nuestro análisis sigue el orden presentado, pero, más allá de la importancia de aplicar la regla AbsTree al final de la primera etapa y la regla EqElimination1 al comienzo de la segunda, no notamos diferencias significativas al cambiar el orden de las otras reglas dentro de cada etapa.

La consistencia de las reglas de reescritura de la sección previa se pueden extender para incluir las reglas de abstracción:

Lema 82 (Consistencia de $\Longrightarrow$)
Si $\Pi \mid \Sigma \implies \Pi' \mid \Sigma'$, *entonces* $(s,h) \models \Pi \mid \Sigma$ *implica* $(s,h) \models \Pi' \mid \Sigma'$ *para todos* s, h

Estos resultados garantizan que el analisis dado por la ejecución de la semántica abstracta es consistente y siempre termina:

Teorema 83 *La semántica abstracta es una sobre-aproximación consistente de la semántica concreta:*

$$[\![p]\!]_c.(\gamma.d) \subseteq \gamma.([\![p]\!]_a.d)$$

para todo $p \in \mathsf{Comm}_a, d \in \mathsf{D}_a$. *Más aún, el algoritmo definido por* $[\![\ ]\!]_a$ *termina.*

Capítulo 4

Discusión

En este capítulo se muestran los resultados obtenidos de la implementación del análisis para distintos algoritmos, se mencionan los trabajos relacionados al nuestro y las contribuciones del mismo a modo de resumen final.

4.1. Resultados experimentales

Implementamos nuestro análisis en Haskell como una forma de validarlo. Con esta implementación realizamos experimentos sobre una variedad de algoritmos iterativos sobre árboles binarios de búsqueda adaptados de la GNU libavl [1], y sobre una adaptación del algoritmo de marcado de Schorr-Waite tomado de [24]. Nuestra implementación aplica una fase de abstracción sobre el estado final para obtener una postcondición más compacta, tratando a las variables que no ocurren en le precondición somo si fueran cuantificadas.

Recordemos del final de la sección 3.3.1 los parámetros p y t para decidir si se aplican las reglas AbsPointer y AbsTree según el nivel de relevancia de la variable de puntero en un término $\mapsto$ o de las expresiones que forman enlaces de cadena. Las reglas se aplican cuando el nivel de relevancia de las variables a abstraer es menor al límite indicado en el parámetro. Los posibles niveles de relevancia van de -1 (para variables cuantificadas) a 4. En los ejemplos presentados se corrió el análisis para todos los posibles valores de p y t desde 0 hasta 5 y se eligieron los valores que no daban como resultado $\top$ (posiblemente por un exceso de abstracción) y que menor cantidad de y más compactos *symbolic heaps* generaban en menor cantidad de iteraciones de los ciclos. En la figura 4.1 se presenta un resumen de los resultados obtenidos. Las columnas p y t tienen los valores de los parámetros que aseguran mejores resultados. Cuando se presenta un rango significa que se obtuvo el mismo resultado del análisis para todos los valores en el rango. La columna Inv. expone la cantidad de *symbolic heaps* que conforman el invariante de los ciclos y la columna It. el número de iteraciones necesarias para alcanzar el punto fijo.

El algoritmo del mínimo (análogo al máximo) fue analizado en el ejemplo 74, para la obtención del invariante del ciclo. En el algoritmo de búsqueda `search` se agregó a la precondición la variable f con el solo objetivo de diferenciar en la postcondición los casos en que se encontró el valor buscado ($f = 1$) de los que no ($f = 0$). Claramente el caso que no se puede dar es $f = 1 \wedge x = \mathbf{nil}$. Los

Algoritmo	Precondición / Postcondición	p	t	Inv.	It.
`min/max`	$\mathbf{true} \mid \mathbf{trees}.\{\!\lvert x \rvert\!\}.\emptyset$	2-5	0-3	2	2
	$x = \mathbf{nil} \mid \mathbf{emp}$				
	$x \neq \mathbf{nil} \mid \mathbf{trees}.\{\!\lvert x \rvert\!\}.\emptyset$				
`search`	$f = 0 \mid \mathbf{trees}.\{\!\lvert x \rvert\!\}.\emptyset$	4-5	0-3	4	3
	$f = 0 \wedge x = \mathbf{nil} \mid \mathbf{emp}$				
	$f = 0 \wedge x \neq \mathbf{nil} \mid \mathbf{trees}.\{\!\lvert x \rvert\!\}.\emptyset$				
	$f = 1 \wedge x \neq \mathbf{nil} \mid \mathbf{trees}.\{\!\lvert x \rvert\!\}.\emptyset$				
`insert`	$f = 0 \wedge n = \mathbf{nil} \mid t \mapsto x, 0, 0 * \mathbf{trees}.\{\!\lvert x \rvert\!\}.\emptyset$	4	2-5	10	4
	$f = 0 \wedge x = \mathbf{nil} \mid t \mapsto n, 0, 0 * \mathbf{trees}.\{\!\lvert n \rvert\!\}.\emptyset$				
	$f = 0 \wedge x \neq \mathbf{nil} \mid t \mapsto x, 0, 0 * \mathbf{trees}.\{\!\lvert x \rvert\!\}.\emptyset$				
	$f = 1 \wedge x \neq \mathbf{nil} \mid t \mapsto x, 0, 0 * \mathbf{trees}.\{\!\lvert x \rvert\!\}.\emptyset$				
`delete`	$\mathbf{true} \mid t \mapsto x', 0, 0 * \mathbf{trees}.\{\!\lvert x' \rvert\!\}.\emptyset$	0-3	0-5	10	4
	$\mathbf{true} \mid t \mapsto \mathbf{nil}, 0, 0$			10	2
	$\mathbf{true} \mid t \mapsto x', 0, 0 * \mathbf{trees}.\{\!\lvert x' \rvert\!\}.\emptyset$				
`destroy`	$\mathbf{true} \mid \mathbf{trees}.\{\!\lvert x \rvert\!\}.\emptyset$	5	2-5	3	2
	$x = \mathbf{nil} \mid \mathbf{emp}$				
	$x \neq \mathbf{nil} \mid \mathbf{emp}$				
`tree to vine`	$\mathbf{true} \mid t \mapsto x', 0, 0 * \mathbf{trees}.\{\!\lvert x' \rvert\!\}.\emptyset$	0-4	1-5	6	4
	$\mathbf{true} \mid t \mapsto \mathbf{nil}, 0, 0$				
	$\mathbf{true} \mid t \mapsto x', 0, 0 * \mathbf{trees}.\{\!\lvert x' \rvert\!\}.\emptyset$				
`Schorr-Waite`	$r = x \mid \mathbf{trees}.\{\!\lvert x \rvert\!\}.\emptyset$	5	2-5	13	4
	$x = \mathbf{nil} \wedge r = \mathbf{nil} \mid \mathbf{emp}$				
	$x \neq \mathbf{nil} \mid \mathbf{trees}.\{\!\lvert x \rvert\!\}.\emptyset$				
	$r \neq \mathbf{nil} \mid \mathbf{trees}.\{\!\lvert r \rvert\!\}.\emptyset$				

Figura 4.1: Resultados del *shape analysis* sobre diferentes ejemplos

mismo se hizo con el algoritmo de inserción `insert`: se agregaron las variables f, que se usa en el programa para marcar que se encontró el valor a insertar en el árbol y por lo tanto ya no es necesaria ninguna operación, y n, que representa el nodo que se creará con el valor a insertar en el árbol. Además, la variable del predicado **trees** no está cuantificada. Esto nos permite que aparezcan en la postcondición y podamos ver en el primer *symbolic heap* que, cuando el árbol original estaba vacío, se tuvo que crear el nuevo nodo a insertar y es ahora el nodo raíz del árbol (el del primer campo del registro referenciado por t). Los otros dos *symbolic heaps* son similares y muestran el caso en que el árbol no era vacío ($x \neq \mathbf{nil}$) y se encontró o no un nodo con el valor a insertar. Si no nos interesara el valor de estas variables de programa, podríamos haber ejecutado el análisis con la precondición

$$t \mapsto x', 0, 0 * \mathbf{trees}.\{\!\lvert x' \rvert\!\}.\emptyset$$

y habríamos obtenido como postcondición un único *symbolic heap* con exactamente la misma fórmula.

Notar que los algoritmos `insert`, `delete` y `tree to vine` tienen en la precondición un puntero t a un registro que, en el primer campo, tiene la variable puntero a la raíz del árbol. Esto es así debido a la estructura de árbol de la GNU libavl y tiene sentido ya que de esta forma la referencia t no cambia nunca de valor y se mantiene aún cuando el árbol queda vacío. Si bien la estructura es la

misma para los algoritmos sobre BSTs, los otros no usan la variable t en sus comandos, por lo que se puede obviar fácilmente, resultando en una presentación más clara.

En la figura 4.2 se muestra la adaptación del algoritmo de `delete`. La guarda $p \neq \mathbf{nil}$ en el primer ciclo y en el **if** posterior se usa para simular un **return** que aparecen en el cuerpo del ciclo en el código original. La guarda *undef* denota una condición que no podemos representar en nuestro lenguaje y no agrega información a ninguna de las ramas de **if** . Se utilizan variables temporales *pval*, *pleft*, y las *tmp*s para codificar operaciones más complejas de lo que permite nuestro lenguaje, como los comandos de consulta y mutación juntos en uno solo, o la presencia de referencias a registros del *heap* en las guardas de los comandos **if** o **while** . Además, el valor *booleano* representado por *found* se codifica con enteros y los **if**s que tienen como guarda $dir = 0$ representan lo que en el código original eran comandos de mutación del campo número *dir* de un registro, que en nuestro lenguaje tiene que ser una constante. Más allá de estas adaptaciones, se conserva el estilo original del código, incluso con los mismos nombres de variables.

Similarmente al algoritmo de inserción, que tiene comandos posteriores a su ciclo, el algoritmo de borrado representa un gran desafío ya que manipula fuertemente la estructura de datos luego del primer ciclo y por lo tanto requiere un invariante lo suficientemente preciso. En la figura 4.1 podemos ver que el primer ciclo alcanza su punto fijo luego de cuatro iteraciones y que su invariante se compone de diez *symbolic heaps*, mientras que para el segundo ciclo solo se necesitan dos iteraciones para llegar a un invariante de diez *symbolic heaps* también. Un caso interesante al correr el análisis con un valor del parámetro `p` de 4 es que el resultado es $\top$ debido a que `rearr`.p no puede revelar la variable p en un estado en que la parte espacial contiene, entre otras cosas, los términos $\mathbf{trees}.\{\!|p|\!\}.\{\!|r|\!\} * r \mapsto tmpi, x'_1, x'_2$ y en la parte pura aparece $p \neq \mathbf{nil}$. Como recordaremos del ejemplo 68, enriqueciendo el estado con los casos $p = r$ y $p \neq r$ se podría, en cada uno por separado, revelar la variable p como puntero a un registro del *heap*. Una forma sencilla de lograr esto sin cambiar la implementación ni la definición de `rearr`.p, es agregando después del segundo ciclo (que es donde se produce la falla al querer ejecutar $tmpj := p.0$) el comando:

$$\textbf{if } p = r \textbf{ then skip else skip fi}$$

Entonces, corriendo de esta manera con el parámetro `p` con valor 4 y cualquier valor de 0 a 5 para `t` se obtienen invariantes más compactos. De hecho, en el segundo ciclo se alcanza el punto fijo con solo una iteración y el invariante se compone de cinco *symbolic heaps*. El invariante del ciclo principal sigue teniendo diez *symbolic heaps* cuya parte espacial es de la forma:

$$t \mapsto p, 0, 0 * \mathbf{trees}.\{\!|p|\!\}.\emptyset \quad \text{ó}$$
$$t \mapsto q, 0, 0 * q \mapsto p, x'_1, x'_2 * \mathbf{trees}.\{\!|p|\!\}.\emptyset * \mathbf{trees}.\{\!|x'_2|\!\}.\emptyset \quad \text{ó}$$
$$t \mapsto x'_3, 0, 0 * \mathbf{trees}.\{\!|x'_3|\!\}.\{\!|q|\!\} * q \mapsto p, x'_1, x'_2 * \mathbf{trees}.\{\!|p|\!\}.\emptyset * \mathbf{trees}.\{\!|x'_2|\!\}.\emptyset$$

con variaciones intercambiando los valores de p y x'_2 en el registro apuntado por q en los últimos dos casos. Más allá de los invariantes un poco más concisos por el mayor nivel de abstracción, la postcondición sigue siendo la misma claramente.

El algoritmo `destroy` realiza la liberación de toda la memoria ocupada por un árbol de manera iterativa mediante *rotaciones*. En este caso, las rotaciones

```
q := t;
p := t.0;
found := 0;
dir := 0;
while p ≠ nil ∧ found = 0 do
    pval := p.1;
    if pval = v then
        found := 1
    else
        q := p;
        if undef then
            dir := 0;   p := p.0
        else
            dir := 2;   p := p.2
        fi
    fi
od ;
if p ≠ nil then
    r := p.2;
    pleft := p.0;
    if r = nil then
        if dir = 0 then q.0 := pleft else q.2 := pleft fi
    else
        s := r.0;
        if s = nil then
            r.0 := pleft;
            if dir = 0 then q.0 := r else q.2 := r fi
        else
            tmp := s.0;
            while tmp ≠ nil do
                r := s;
                s := r.0;
                tmp := s.0
            od ;
            tmpi := s.2;   r.0 := tmpi;
            tmpj := p.0;   s.0 := tmpj;
            tmpk := p.2;   s.2 := tmpk;
            if dir = 0 then q.0 := s else q.2 := s fi
        fi
    fi;
    free(p);
else skip fi
```

Figura 4.2: Adaptación del algoritmo de borrado de un elemento de un BST

```
q := t;
p := t.0;
while p ≠ nil do
    pright := p.2;
    if pright = nil then
        q := p;
        p := p.0
    else
        r := p.2;
        tmp := r.0;   p.2 := tmp;
        r.0 := p;
        p := r;
        q.0 := r
    fi
od
```

Figura 4.3: Adaptación del algoritmo de transformación de árbol a *vine*

consisten en mover nodos desde el subárbol izquierdo de la raíz hacia el subárbol derecho. Eventualmente la raíz queda con un subárbol izquierdo vacío y por lo tanto se puede liberar el nodo raíz para luego proceder al subárbol derecho. Partiendo de un árbol referenciado por la variable x, la postcondición que se obtiene es la esperable:

$$\{x = \mathbf{nil} \,|\, \mathbf{emp}, \qquad x \neq= \mathbf{nil} \,|\, \mathbf{emp}\}$$

El algoritmo `tree to vine` usa un método similar al `destroy` para, mediante rotaciones, transformar un árbol binario de búsqueda arbitrario en un *vine*. Un *vine* es un árbol binario degenerado parecido a una lista, donde todos los nodos tienen a lo sumo un solo subárbol no vacío. En este caso se obtiene un *vine* con nodos que no tienen subárbol derecho (i.e. que está vacío). Este algoritmo se usa para el balanceo de árboles binarios de búsqueda: luego de obtener el *vine* se realizan otras transformaciones que lo van comprimiendo hasta obtener un árbol binario de altura mínima. En la figura 4.3 se muestra el algoritmo de transformación a *vine*, en donde la única adaptación necesaria fue el uso de las variables auxiliares *pright* y *tmp*. Se obtiene un invariante, luego de cuatro iteraciones, con seis *symbolic heaps* cuya parte espacial resulta de combinar con el operados de conjunción $*$ de todas las formas posibles los elementos de los siguientes conjuntos:

$$\{t \mapsto p, 0, 0, \qquad t \mapsto q, 0, 0 * q \mapsto p, x'_4, \mathbf{nil},$$
$$t \mapsto x'_5, 0, 0 * \mathbf{trees}.\{\!|x'_5|\!\}.\{\!|q|\!\} * q \mapsto p, x'_4, \mathbf{nil}\}$$

$$\{\mathbf{trees}.\{\!|p|\!\}.\emptyset, \qquad p \mapsto x'_1, x'_2, x'_3 * \mathbf{trees}.\{\!|x'_1|\!\}.\emptyset * \mathbf{trees}.\{\!|x'_3|\!\}.\emptyset\}$$

Finalmente, presentamos el algoritmo de marcado de `Schorr-Waite` en la figura 4.4. Este algoritmo realiza el recorrido de un grafo de manera iterativa sin el uso extra de estructuras de datos más complejas como *stacks* o *queues*, sino simplemente con los registros de la misma estructura de grafo. Cada registro tiene, además de los dos punteros que representan aristas a otros nodos de la

```
p := nil;
if r ≠ nil then
    rval := r.1;   if rval = 1 ∨ rval = 3 then rm := 1 else rm := 0 fi
else rm := 1 fi
while p ≠ nil ∨ (r ≠ nil ∧ rm = 0) do
    if r = nil ∨ rm = 1 then
        pval := p.1;   if pval = 2 ∨ pval = 3 then pc := 1 else pc := 0 fi
        if pc = 1 then
            q := r;
            r := p;
            p := p.2;
            r.2 := q
        else
            q := r;
            r := p.2;
            pl := p.0;
            p.2 := pl;
            p.0 := q;
            if pval = 0 then pval := 2 else pval := 3 fi;
            p.1 := pval
        fi
    else
        q := p;
        p := r;
        r := r.0;
        p.0 := q;
        p.1 := 1;
    fi;
    if r ≠ nil then
        rval := r.1;   if rval = 1 ∨ rval = 3 then rm := 1 else rm := 0 fi
    else rm := 1 fi
od
```

Figura 4.4: Adaptación del algoritmo Schorr-Waite

estructura, dos campos booleanos: *check* y *mark*. Al comienzo el campo *mark* está en **false** para todos los nodos y al terminar quedan marcados con **true**. El campo *check*, junto con la reutilización de los dos punteros del registro es lo que permite hacer el recorrido. La única adaptación del código tomado de [24] consiste en combinar y codificar los campos *check* y *mark* en nuestro campo de valor del registro para adecuarlo a nuestro modelo de memoria. De esta forma, cuando *check* sea **true** el valor de nuestro registro será 2 o 3, y cuando *mark* sea **true** será 1 o 3. De ahí la necesidad de agregar varios **if**s extra, que se presentan en una sola linea para no ensuciar el resto del código. Si bien el algoritmo funciona para grafos en general, lo probamos con árboles partiendo de la precondición:

$$r = x \,¦\, \mathbf{trees}.\{\!|x|\!\}.\emptyset$$

La postcondición obtenida es:

$$\{x = \mathbf{nil} \wedge r = \mathbf{nil} \,¦\, \mathbf{emp}, \quad x \neq \mathbf{nil} \,¦\, \mathbf{trees}.\{\!|x|\!\}.\emptyset, \quad r \neq \mathbf{nil} \,¦\, \mathbf{trees}.\{\!|r|\!\}.\emptyset\}$$

Se puede apreciar que se termina con un árbol vacío y sin nada en la memoria (predicado **emp**) cuando $x = \mathbf{nil}$ o con un árbol no vacío que comienza en el registro referenciado por x o por r. Sin embargo, si bien la variable r está en la precondición, es una variable que se usa en el programa y cuyo valor puede cambiar, a diferencia de la variable x que no se modifica en el algoritmo. Debido a las varias verificaciones formales del algoritmo de `Schorr-Waite`, una postcondición válida es que se mantiene el mismo árbol de la precondición y referenciado por la misma variable x que no fue modificada. Nuestro análisis encuentra un límite al no poder establecer $x = r$ en el segundo *symbolic heap*, por más que se cumpla en cualquier ejecución de la semántica concreta. Más allá de esto, sí logramos establecer como postcondición que en la memoria hay una estructura de árbol, como al comienzo. El invariante se alcanza en cuatro iteraciones y se obtienen trece *symbolic heaps* cuya parte espacial es esencialmente:

$$\mathbf{trees}.\{\!|p|\!\}.\emptyset * \mathbf{trees}.\{\!|r|\!\}.\emptyset$$

combinada con casos donde $p = \mathbf{nil}$ y/o $r = \mathbf{nil}$. Esto resulta en que, además de la fórmula anterior, la parte espacial puede ser alguna de las fórmulas **emp**, $\mathbf{trees}.\{\!|p|\!\}.\emptyset$ o $\mathbf{trees}.\{\!|r|\!\}.\emptyset$. Este es un buen ejemplo de la precisión que puede alcanzar nuestro análisis. Si usáramos por ejemplo un valor de 2 para el parámetro `p`, en vez de 5, obtendríamos una postcondición equivalente, pero con un invariante alcanzado luego de diez iteraciones y compuesto por 118 *symbolic heaps*.

4.2. Trabajos Relacionados

Los principales trabajos relacionados son [5, 2, 25], que al igual que el nuestro, derivan de [9]. Estas extensiones están implementadas en la herramienta SpaceInvader y buscan verificar sistemas de código real, soportando estructuras lineales combinadas de maneras complejas. Nuestro trabajo extiende el dominio de aplicación de estos métodos para soportar estructuras de datos no lineales. El trabajo [3] es predecesor de [9] y se basa en un tipo diferente de ejecución simbólica aunque soporta árboles binarios. Introduce la herramienta Smallfoot que reduce la verificación de ternas de Hoare a implicaciones lógicas entre los

symbolic heaps, pero no soporta ciclos. El predicado de árbol usual (**tree**) parece adecuado para los programas recursivos contemplados en este análisis. En [21, 11] se presentan análisis similares al de [9] tanto en los aspectos técnicos como en sus limitaciones.

La herramienta Xisa [7] también se basa en Separation Logic para describir estados abstractos, pero usa predicados inductivos generalizados en la forma de verificadores de invariantes de estructuras provistos por el usuario. Las operaciones de *fold* y *unfold* de los predicados guían las estrategias automáticas de materialización (rearr en nuestro trabajo) y abstracción. La extensión [6] agrega poder expresivo para especificar relaciones entre regiones del *heap*, como invariantes de ordenación, punteros *back* y *cross*, etc. y presenta como caso de estudio la inserción de un elemento en un árbol *Red-Black*.

La herramienta Verifast [14] también se basa en Separation Logic y permite la verificación de ternas de Hoare que utilizan predicados inductivos para especificar las estructuras y funciones recursivas puras sobre esos tipos. Genera condiciones de verificación resueltas por un *SMT solver*. No se proveen mecanismos de abstracción, sino que deben anotarse manualmente invariantes e instrucciones que guíen el *fold* y *unfold* de los predicados. Es capaz de verificar una implementación recursiva de una librería sobre árboles binarios y el patrón *composite* (con su estructura de grafo subyacente) [15].

El sistema PALE [20, 16] permite la verificación de programas especificados con aserciones en *Weak Monadic Second-order Logic of Graph Types*. Los ciclos deben anotarse con invariantes y las condiciones de verificación son descargadas en la herramienta MONA [12]. La extensión [10] hace más eficiente la verificación de algoritmos sobre estructuras de árboles.

El análisis de [17] usa *shape graphs* y *grammar annotations* para especificar estructuras de datos acíclicas y permite descubrir automáticamente descripciones para las estructuras que ocurren en los puntos intermedios de la ejecución de programas. Este análisis verifica los algoritmos de Schorr-Waite, de destrucción de un árbol y de construcción de un *heap* binomial.

El *shape analysis* paramétrico [23, 4] basado en *3-valued Logic*, junto con su implementación TVLA [18], conforman el marco de trabajo más general, poderoso y utilizado en la verificación de propiedades sobre la forma de estructuras dinámicas en programas que involucran complejas manipulaciones de las mismas. Este marco de trabajo debe ser instanciado para cada caso particular a través de predicados de instrumentación provistos por el usuario para definir el tipo de estructuras soportadas y las formas de abstracción. En [19] se presenta una instancia que permite la verificación de corrección total (i.e. con terminación) del algoritmo de Schorr-Waite sobre árboles.

4.3. Palabras finales sobre el *shape analysis*

El análisis definido por la semántica abstracta presentada en la sección 3.3 permite verificar propiedades interesantes sobre la forma de las estructuras de datos manipuladas por programas sobre árboles binarios. Para un programa con su precondición, el análisis calcula automáticamente invariantes para cada ciclo y una postcondición. Esta semántica es una sobreaproximación de la semántica operacional sobre un modelo de memoria estándar, por lo que puede reportar falsos fallos de memoria y *memory leaks*. Sin embargo, los experimentos mues-

tran un buen nivel de ajuste entre los estados abstractos computados y aquellos esperados según la semántica concreta.

Para definir el análisis, introdujimos un novedoso predicado lineal **trees** que describe estructuras de datos no lineales con múltiples puntos de entrada y de salida. Este predicado nos da la posibilidad de expresar de forma sencilla las complejas estructuras que normalmente aparecen en los puntos intermedios durante la ejecución de programas. Las buenas propiedades sintácticas que posee permiten una caracterización simple de la fase de abstracción del análisis.

La utilización del análisis estático que determina el nivel de relevancia de las variables para la fase de abstracción, aunque es una idea muy sencilla, introduce una mejora significativa en cuanto a invariantes de ciclo más compactos, sin perjudicar la precisión necesaria para obtener postcondiciones relevantes. En algunos casos se consigue una reducción dramática en la cantidad de fórmulas que componen un estado abstracto, agilizando el proceso completo del análisis, y ayudando a la computación de invariantes comprensibles e intuitivamente correctos. Nos permite además prever un buen comportamiento del análisis sobre programas de mayor tamaño.

Como consideración final, al basar nuestro análisis en el trabajo [9], heredamos todas las ventajas de la *Separation Logic*. Nuestras reglas de reescrituras son implicaciones válidas cuya verificación semántica es muy sencilla, mientras que las reglas de ejecución simbólica se derivan directamente de las ternas de Hoare. De esta manera el análisis resulta intuitivo y su corrección fácilmente comprobable.

Apéndice

La realización de este trabajo se llevó a cabo en dos etapas, en momentos diferentes. En la primera etapa hubo un proceso de aprendizaje sobre *Separation Logic* y lectura de diferentes artículos basados en ella referidos a distintos *shape analysis*, ejecución simbólica, etc. Con este conocimiento se emprendió un camino de investigación con la intención de poder describir estructuras de datos no lineales y verificar propiedades sobre algoritmos que las manipulan. Este proceso dio como frutos la realización del artículo [8], conjuntamente con Renato Cherini y Javier Blanco, y del que se basa la presentación del capítulo 3. Finalmente, en la segunda etapa se realizó una revisión de ese artículo y la escritura final de este trabajo. En la actual presentación, se optó por cambiar algunas de las reglas de la fase de abstracción y se corrigió un error referido a la terminación del análisis descubierto en la revisión del artículo. Estas diferencias se presentan a continuación.

Para empezar, se reemplazaron algunas reglas de la primera etapa de la función de abstracción abs por dos reglas más simples, AbsPointer y AbsTree, junto con la regla de normalización de árboles sin punteros de salida DivideTree, presentadas en la definición 73. Las otras reglas de esa etapa, las de eliminación de punteros **nil** y de expresiones iguales en los multiconjuntos de entrada y salida, están de igual manera en el trabajo original.

La mayoría de las reglas sustituidas se pueden expresar con la aplicación de las reglas actuales. La regla AbsTree2 del trabajo original:

$$\frac{\Pi \mid \Sigma \vdash x = y \qquad \Pi \mid \Sigma \vdash Cell(e) \vee e = \mathbf{nil} \quad \text{para todo } e \text{ en } \mathcal{F}}{\Pi \mid \Sigma * \mathbf{trees}.\mathcal{C}.(x \oplus \mathcal{D}) * T.(y \oplus \mathcal{E}).\mathcal{F} \implies \Pi \mid \Sigma * \mathbf{trees}.(\mathcal{C} \uplus \mathcal{E}).(\mathcal{D} \uplus \mathcal{F})}$$

es igual a nuestra nueva regla AbsTree para el caso en que T represente un predicado **trees**, teniendo en cuenta que la condición $Cell(e)$ es igual a nuestra condición $noDangling.e$, solo que esta última también incluye el caso $e = \mathbf{nil}$, por lo que no hace falta explicitarlo aparte. Cuando T representa un predicado de $\mapsto$, esta regla se puede expresar usando en primer lugar la regla AbsPointer para abstraer el predicado de $\mapsto$ en un predicado **trees**, que gracias a la condición de que todas las expresiones de $\mathcal{F}$ representan punteros a registros válidos en un *heap* disjunto al del predicado T o son **nil**, se puede establecer su condición de que y sea distinto a los elementos de $\mathcal{F}$, y en segundo lugar directamente con la regla AbsTree.

La regla original AbsTree1 se puede obtener componiendo las nuevas reglas DivideTree para separar la variable y del resto de los punteros de entrada de $\mathcal{E}$ en su propio predicado **trees** y la regla AbsTree para unir este predicado **trees** resultante con $\mathbf{trees}.\mathcal{C}.x \oplus \mathcal{D}$.

$$\frac{\Sigma \doteq \Sigma' * \mathbf{trees}.\mathcal{C}.x \oplus \mathcal{D} * \mathbf{trees}.y \oplus \mathcal{E}.\emptyset \qquad \Pi \mid \Sigma \vdash x = y}{\Pi \mid \Sigma \implies \Pi \mid \Sigma' * \mathbf{trees}.\mathcal{C}.\mathcal{D} * \mathbf{trees}.\mathcal{E}.\emptyset}$$

De la misma forma, la regla original AbsArrow1, que no presentamos acá, tanto para el caso de abstracción del *link* izquierdo como para el análogo caso del derecho, se puede expresar con la combinación de las tres nuevas reglas AbsPointer, DivideTree y AbsTree.

En general, las reglas en el trabajo original son más complejas porque se pretendía asegurar que con su aplicación se redujeran de alguna manera las expresiones involucradas en los predicados de $\mapsto$ o **trees**. Así es que en esas reglas siempre hay un enlace de cadena que se abstrae y desaparece. Un posible problema del nuevo conjunto de reglas es que se podría aplicar la regla AbsPointer sin que ello conlleve a una aplicación de la regla AbsTree. Esta última regla es la de mayor importancia en la abstracción ya que permite unir estructuras parciales entre sí, olvidando información concreta innecesaria. En el mencionado caso, la abstracción de un predicado de $\mapsto$ habría sido en vano y además podría generar inconvenientes para continuar con el análisis en caso de que se intente acceder a la variable asbtraida. Afortunadamente, esto no sucede gracias al cálculo del nivel de relevancia de las variables y a la posibilidad de abstraerlas cuando no se va a requerir un dereferenciamiento de las mismas en los comandos posteriores.

La única regla del trabajo original que no se puede expresar con el conjunto de reglas nuevas es AbsArrow2, que presentamos a continuación para el caso izquierdo, ya que el derecho es análogo:

$$\frac{\Sigma \doteq \Sigma' * x \mapsto l, v, r * T.y \oplus \mathcal{C}.\mathcal{D} \qquad \Pi \mid \Sigma \vdash l = y \wedge x \notin r \oplus \mathcal{D} \wedge y \notin \mathcal{D}}{\Pi \mid \Sigma \implies \Pi \mid \Sigma' * \mathbf{trees}.x \oplus \mathcal{C}.r \oplus \mathcal{D}}$$

El caso en que T represente un predicado de $\mapsto$, se puede abstraer con la regla AbsPointer en un predicado **trees** ya que tenemos la condición de aplicación asegurada con $y \notin \mathcal{D}$. Por lo tanto, asumiremos que T representa un predicado **trees**. En el conjunto de reglas nuevas la combinación de estructuras se da siempre sobre predicados **trees**, por lo que podríamos intentar abstraer el predicado de puntero $x \mapsto l, v, r$ en un predicado **trees**. La condición $y \notin \mathcal{D}$, junto con $T.y \oplus \mathcal{C}.\mathcal{D}$ implica que podemos probar que $x \neq l$ (l es igual a y que es un puntero válido o es **nil**). También podemos probar $x \neq r$ (es implicado por $x \notin r \oplus \mathcal{D}$) por lo que efectivamente podemos abstraer el predicado de $\mapsto$ a $\mathbf{trees}.\{\!|x|\!\}.\{\!|l, r|\!\}$. Ahora es cuando se plantea el problema de tener condiciones de aplicación de la nueva regla AbsTree más fuertes de lo que exige la regla original AbsArrow2. La condición $x \notin \mathcal{D}$ es realmente necesaria para mantener la estructura de árbol (sin ciclos) si se pretende combinar los dos predicados **trees**, pero no es suficiente. Al perder la información de que el árbol $\mathbf{trees}.\{\!|x|\!\}.\{\!|l, r|\!\}$ en realidad representaba un solo registro apuntado por x, la simple condición $x \notin \mathcal{D}$ ya no alcanza para asegurar que ningún elemento de $\mathcal{D}$ referencie a algún registro del otro árbol. Sin embargo, la manera en que usualmente se establece la validez de una condición como $x \notin \mathcal{D}$, sabiendo que x referencia un registro válido, es con los elementos de $\mathcal{D}$ referenciando otros registros o siendo iguales a **nil**, es decir, siendo *noDangling*, que es precisamente la condición de aplicación de la regla AbsTree. Entonces, en los casos prácticos generalmente se puede lograr la abstracción y así es como con el nuevo conjunto de reglas se pudieron verificar los mismos algoritmos analizados en el trabajo original.

En la revisión del artículo original para la escritura de este trabajo se descubrió un error referido a la terminación del análisis. Para asegurar la terminación en el cómputo del punto fijo de los invariantes, se usaba (y se sigue usando) que la imagen de la función abs era finita. Sin embargo, esto no era cierto. Había algunos detalles menores, como considerar al conjunto de los números naturales como los valores elementales en la definición del estado y como constantes del conjunto de expresiones. Entonces, las reglas de abstracción de la segunda y tercera etapa no prohibían que términos de la forma $x \neq n$ con x variable de programa con un nivel de relevancia mayor a 0 y n cualquier constante numérica entre los naturales ocurran en los *symbolic heaps* de la imagen de abs, por lo que dicha imagen no podía ser finita. Por otro lado, las condiciones de aplicación de las reglas que eliminaban variables cuantificadas de la parte espacial de un *symbolic heap* impedían que fórmulas como:

$$\mathbf{trees}.\{\!|x'_1|\!\}.\{\!|y'_1|\!\} * \mathbf{trees}.\{\!|x'_1|\!\}.\{\!|y'_2|\!\} * \mathbf{trees}.\{\!|x'_2|\!\}.\{\!|y'_1|\!\} * \mathbf{trees}.\{\!|x'_2|\!\}.\{\!|y'_2|\!\}$$

pudieran ser abstraídas, para cualesquiera variables cuantificadas x'_1, x'_2, y'_1, y'_2, causando infinitas fórmulas posibles en la imagen de abs. Esta fórmula, si bien debe necesariamente ser equivalente a **emp** o a una fórmula de la forma $\mathbf{trees}.\{\!|x'|\!\}.\{\!|y'|\!\} * \mathbf{trees}.\{\!|y'|\!\}.\{\!|x'|\!\}$ para que sea válida, cumple con su objetivo de refutación.

Los casos anteriores motivaron que en este trabajo, por un lado se use el conjunto $\mathbb{Z}_k$ de los números enteros entre 0 y $k-1$ como valores elementales y constantes de las expresiones, y por el otro se usen las reglas Garbage1-3 de la segunda etapa (definición 77) como forma efectiva para limitar la proliferación de variables cuantificadas.

Si bien los casos anteriores son medio rebuscados y difícilmente se puedan obtener como resultados de la ejecución de un programa real, un tema importante es que en los multiconjuntos de los predicados **trees** podrían ocurrir expresiones una cantidad arbitraria de veces. Ya no hablamos de variables cuantificadas, sino de una misma variable de programa. Más aún, el siguiente algoritmo muestra una construcción donde se consigue generar un estado abstracto con una cantidad no acotada de *symbolic heaps* en la ejecución del ciclo, usando las reglas del trabajo original, de manera que no solo se evidencia la no finitud de la imagen de abs, sino que representa un ejemplo para el que el análisis no termina:

$$\begin{array}{l}\textbf{while } x \neq p \textbf{ do}\\ \quad k := \mathbf{new}(z, 0, d);\\ \quad z.2 := p;\\ \quad x := z;\\ \quad z := k\\ \textbf{od}\end{array}$$

El programa anterior, partiendo de la precondición:

$$\mathbf{true} \,|\, p \mapsto p, 0, p * z \mapsto x, 0, d * \mathbf{trees}.\{\!|x|\!\}.\emptyset$$

genera, luego de la ejecución del cuerpo del ciclo, fórmulas como las siguientes:

$$\begin{array}{l}\mathbf{true} \,|\, p \mapsto p, 0, p * z \mapsto x, 0, d * \mathbf{trees}.\{\!|x|\!\}.\{\!|p|\!\}\\ \mathbf{true} \,|\, p \mapsto p, 0, p * z \mapsto x, 0, d * \mathbf{trees}.\{\!|x|\!\}.\{\!|p, p|\!\}\\ \mathbf{true} \,|\, p \mapsto p, 0, p * z \mapsto x, 0, d * \mathbf{trees}.\{\!|x|\!\}.\{\!|p, p, p|\!\}\end{array}$$

donde cada vez hay una mayor cantidad de expresiones p en el multiconjunto de los punteros de salida del último predicado **trees**.

Vemos a continuación cómo se van modificando los *symbolic heaps* en el cuerpo del ciclo, partiendo del segundo ejemplo de arriba con la guarda $x \neq p$

$$
\begin{array}{l}
\{x \neq p \mid p \mapsto p, 0, p * z \mapsto x, 0, d * \mathbf{trees}.\{\!|x|\!\}.\{\!|p, p|\!\}\} \\
\quad k := \mathbf{new}(z, 0, d); \\
\{x \neq p \mid p \mapsto p, 0, p * k \mapsto z, 0, d * z \mapsto x, 0, d * \mathbf{trees}.\{\!|x|\!\}.\{\!|p, p|\!\}\} \\
\quad z.2 := p; \\
\{x \neq p \mid p \mapsto p, 0, p * k \mapsto z, 0, d * z \mapsto x, 0, p * \mathbf{trees}.\{\!|x|\!\}.\{\!|p, p|\!\}\} \\
\quad x := z; \\
\{x' \neq p \mid p \mapsto p, 0, p * k \mapsto x, 0, d * x \mapsto x', 0, p * \mathbf{trees}.\{\!|x'|\!\}.\{\!|p, p|\!\}\} \\
\quad z := k \\
\{x' \neq p \mid p \mapsto p, 0, p * z \mapsto x, 0, d * x \mapsto x', 0, p * \mathbf{trees}.\{\!|x'|\!\}.\{\!|p, p|\!\}\}
\end{array}
$$

En el último *symbolic heap* podemos aplicar la regla del trabajo original AbsArrow2 (usando que $x' \neq p$ para establecer $y \notin \mathcal{D}$) o las nuevas reglas AbsPointer (usando que $x' \neq p$ para demostrar que $x \neq x'$) y AbsTree para combinar los últimos dos términos en $\mathbf{trees}.\{\!|x|\!\}.\{\!|p, p, p|\!\}$.

Notar que el ejemplo es de un ciclo infinito que construye *dags* cada vez mayores. Como no es la intención del trabajo especificar *dags*, se incluyen en la segunda etapa las reglas Garbage4-5 que evitan que ocurran situaciones así. Además en el trabajo se presenta una justificación detallada de la finitud de la imagen de abs.

Bibliografía

[1] GNU libavl. `http://adtinfo.org/index.html`.

[2] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Shape analysis for composite data structures. In Werner Damm and Holger Hermanns, editors, *CAV*, volume 4590 of *LNCS*, pages 178–192. Springer, 2007.

[3] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In *APLAS*, volume 3780 of *LNCS*, pages 52–68. Springer, 2005.

[4] Igor Bogudlov, Tal Lev-Ami, Thomas W. Reps, and Mooly Sagiv. Revamping TVLA: Making parametric shape analysis competitive. In Werner Damm and Holger Hermanns, editors, *CAV*, volume 4590 of *LNCS*, pages 221–225. Springer, 2007.

[5] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In Zhong Shao and Benjamin C. Pierce, editors, *ACM SIGPLAN-SIGACT 2009 Symposium on Principles of Programming Languages*, pages 289–300. ACM, 2009.

[6] Bor-Yuh Evan Chang and Xavier Rival. Relational inductive shape analysis. In *ACM SIGPLAN-SIGACT 2008 Symposium on Principles of Programming Languages*, pages 247–260. ACM, 2008.

[7] Bor-Yuh Evan Chang, Xavier Rival, and George C. Necula. Shape analysis with structural invariant checkers. In Hanne Riis Nielson and Gilberto Filé, editors, *SAS*, volume 4634 of *LNCS*, pages 384–401. Springer, 2007.

[8] Renato Cherini, Lucas Rearte, and Javier Blanco. A shape analysis for non-linear data structures. In *International Static Analysis Symposium*, pages 201–217. Springer, 2010.

[9] Dino Distefano, Peter W O'hearn, and Hongseok Yang. A local shape analysis based on separation logic. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 287–302. Springer, 2006.

[10] Jacob Elgaard, Anders Møller, and Michael I. Schwartzbach. Compile-time debugging of C programs working on trees. In Gert Smolka, editor, *ESOP*, volume 1782 of *LNCS*, pages 182–194. Springer-Verlag, 2000.

[11] Alexey Gotsman, Josh Berdine, and Byron Cook. Interprocedural shape analysis with separated heap abstractions. In Kwangkeun Yi, editor, *SAS*, volume 4134 of *LNCS*, pages 240–260. Springer, 2006.

[12] Jesper G. Henriksen, Jakob L. Jensen, Michael E. Jørgensen, Nils Klarlund, Robert Paige, Theis Rauhe, and Anders Sandholm. Mona: Monadic second-order logic in practice. In Ed Brinksma, Rance Cleaveland, Kim Guldstrand Larsen, Tiziana Margaria, and Bernhard Steffen, editors, *TACAS*, volume 1019 of *LNCS*, pages 89–110. Springer, 1995.

[13] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[14] Bart Jacobs and Frank Piessens. The verifast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, August 2008.

[15] Bart Jacobs, Jan Smans, and Frank Piessens. Verifying the composite pattern using separation logic. In *SAVCBS Composite pattern challenge track*, 2008.

[16] Jakob L. Jensen, Michael E. Jørgensen, Michael I. Schwartzbach, and Nils Klarlund. Automatic verification of pointer programs using monadic second-order logic. In *ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, pages 226–236. ACM, 1997.

[17] Oukseh Lee, Hongseok Yang, and Kwangkeun Yi. Automatic verification of pointer programs using grammar-based shape analysis. In Shmuel Sagiv, editor, *ESOP*, volume 3444 of *LNCS*, pages 124–140. Springer, 2005.

[18] Tal Lev-Ami and Shmuel Sagiv. TVLA: A system for implementing static analyses. In Jens Palsberg, editor, *SAS*, volume 1824 of *LNCS*, pages 280–301. Springer, 2000.

[19] Alexey Loginov, Thomas W. Reps, and Mooly Sagiv. Automated verification of the deutsch-schorr-waite tree-traversal algorithm. In Kwangkeun Yi, editor, *SAS*, volume 4134 of *LNCS*, pages 261–279. Springer, 2006.

[20] Anders Møller and Michael I. Schwartzbach. The pointer assertion logic engine. In *ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*. ACM, 2001.

[21] Huu Hai Nguyen, Cristina David, Shengchao Qin, and Wei-Ngan Chin. Automated verification of shape and size properties via separation logic. In Byron Cook and Andreas Podelski, editors, *VMCAI*, volume 4349 of *LNCS*, pages 251–266. Springer, 2007.

[22] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE Computer Society, 2002.

[23] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.

[24] Hongseok Yang. *Local reasoning for stateful programs.* PhD thesis, University of Illinois at Urbana-Champaign, 2001.

[25] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Scalable shape analysis for systems code. In Aarti Gupta and Sharad Malik, editors, *CAV*, volume 5123 of *LNCS*, pages 385–398. Springer, 2008.