View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Repositorio Digital de la Universidad Nacional de Córdoba

Universidad Nacional de Córdoba
Facultad de Matemática, Astronomía y Física
Doctorado en Ciencias de la Computación

Construcción de programas que manejan dinámicamente la memoria

Renato Cherini

Director: Javier O. Blanco

Tesis presentada ante la Facultad de Matemática, Astronomía y Física, Universidad Nacional de Córdoba, como parte de los requisitos para la obtención del grado de Doctor en Ciencias de la Computación, el 6 de agosto de 2015.

Construcción de programas que manejan dinámicamente la memoria
por Renato Cherini se distribuye bajo una Licencia Creative Commons
Atribución-NoComercial-CompartirIgual 2.5 Argentina.

¡El que abandona, no tiene premio!
¿A quién le importa? ¡toda esta guinda!
¡Si te sofoca! ¿A quién le importa?

Resumen

La obtención de programas correctos, es decir, aquellos que se ejecutan sin errores y computan el resultado esperado, ha sido uno de los principales objetivos de las ciencias de la computación desde sus inicios. Lejos de perder vigencia, este propósito se convierte en una necesidad imperiosa, dada la ubicuidad cada vez mayor de los sistemas informáticos. La tradición que encarna la verificación formal de programas, intenta dar certezas de nivel matemático sobre la corrección de los programas. A pesar de los numerosos marcos de trabajo que dan cuenta de las diferentes construcciones comunes de los lenguajes de programación *reales*, el uso de memoria dinámica continúa siendo un reto difícil de abordar. Esta dificultad representa un problema relevante ya que la memoria dinámica es una de las facilidades más extendidas en el universo de la programación, dado que permite la implementación eficiente de estructuras de datos y da soporte a uno de los paradigmas más utilizados, la programación orientada a objetos.

En este trabajo abordamos diferentes aspectos de la verificación de programas que manejan dinámicamente la memoria, y más en general, al razonamiento formal sobre ellos. No pretendemos, en ningún caso, dar respuestas cabales a las dificultades que surgen en este abordaje, sino más bien, proponer marcos de trabajo que permitan expresar los problemas en términos más adecuados para su análisis y comprensión.

Por un lado, proponemos un marco conceptual para considerar cuestiones ontológicas y epistemológicas de la propia tarea de verificación formal. Aunque no está exenta de críticas dentro del ámbito de las ciencias de la computación, principalmente en cuanto su practicidad, la verificación formal es aceptada como una forma válida de razonamiento sobre los programas. Sin embargo ha sido puesta en duda en ámbitos como la filosofía de la computación. El debate sobre su validez se centra en la presunta naturaleza dual de los programas -su código como entidad abstracta y su ejecución como entidad causal- y una supuesta insalvable distancia entre ellos, que inhabilita cualquier uso de métodos de razonamiento deductivo composicional. En este trabajo proponemos una generalización del concepto de intérprete, como se lo entiende en el ámbito de la teoría de la computación, que nos permite relacionar los aspectos abstractos y concretos de la computación, y así, entre otras cosas, soportar nuestra apuesta de considerar la verificación formal como forma privilegiada para razonar sobre los programas.

En el plano metodológico, la principal dificultad para abordar el razonamiento formal sobre la memoria dinámica deviene de la ineficacia de los marcos teóricos usuales, basados en la lógica de primer orden, para expresar sus propiedades características y el efecto que sobre ella tiene la ejecución de los programas. Se pueden mencionar los problemas de *aliasing* -la existencia de dos o más referencias a una misma porción de memoria-, *dangling pointers* -el acceso indebido a porciones de memoria no definidas- y *memory leaks* -la existencia de porciones de memoria inaccesibles- como los principales exponentes de esta dificultad. La aparición de la *Separation Logic* viene a resolver estos problemas, introduciendo un nuevo formalismo que los trata de manera sintáctica a partir del uso de una novedoso lenguaje de fórmulas y un sistema deductivo, utilizado con mucho éxito en la demostración de programas

paradigmáticos, que realizan una manipulación sutil y compleja de la memoria. Sin embargo, la eficacia de la *Separation Logic* encuentra su límite para razonar sobre programas que manipulan múltiples estructuras de datos que comparten parte de la memoria dinámica para su representación, una situación muy común dentro de las prácticas usuales de la programación. Una contribución de nuestro trabajo es la introducción de la *Sharing Logic*, que permite especificar de forma precisa estructuras dinámicas complejas y las relaciones entre ellas, de manera compatible con los principios de abstracción e *information hiding*. La lógica incluye un sistema de prueba composicional para verificar programas de forma modular bajo ciertas condiciones que pueden ser expresadas y demostradas utilizando el propio lenguaje de fórmulas. Este marco permite caracterizar las condiciones requeridas para razonar modularmente sobre estructuras en la memoria dinámica cuando no es posible garantizar que no comparten parte de su implementación.

En el plano práctico, la posibilidad de que el razonamiento formal escale a sistemas de tamaño real, depende fuertemente de que existan herramientas (semi) automáticas que faciliten su uso. Dada la complejidad intrínseca de los sistemas deductivos formales, es una gran ventaja que existan procedimientos de decisión que resuelvan las condiciones de verificación asociadas a la corrección de un programa. Mayor ventaja representa la existencia de algoritmos que lleven adelante el proceso completo de verificación. El término *shape analysis* es una denominación general para un tipo de tales algoritmos que verifican propiedades asociadas a la *forma* de las estructuras de datos manipuladas por los programas. Por un lado, en el presente trabajo abordamos la decidibilidad del problema de validez de un fragmento de nuestra *Sharing Logic* que permite caracterizar estructuras de datos como listas enlazadas y segmentos de ellas. Como un resultado derivado obtenemos una nueva prueba de la decidibilidad de un fragmento de la *Separation Logic* con características similares. Por otro, presentamos un *shape analysis* con garantías de terminación, que verifica automáticamente programas que manipulan estructuras de datos no lineales, que pueden incluir formas específicas de *sharing* de memoria, como *dags* y grafos generales. Ambas contribuciones ayudan a delimitar de forma más precisa las posibilidades de la automatización de la verificación formal y comprender sus causas y restricciones.

El analisis llevado adelante en los mencionados planos, nos permite comprender de mejor manera la naturaleza de los problemas asociados al razonamiento formal sobre programas que manipulan la memoria dinámica, y de esta forma poder contribuir a salvar las dificultades en el uso de métodos formales para garantizar la fiabilidad, seguridad y corrección de esta clase de programas.

Palabras claves: verificación, memoria dinámica, *separation logic*, análisis estático, intérpretes.

1998 ACM Subject Classification: D.2.4 [Software Engineering]: Software / Program Verification —correctness proofs, formal methods; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs; I.2.2 [Artificial Intelligence]: Automatic Programming —program verification; F.1.m [Computation by Abstract Devices]: Miscellaneous

Agradecimientos

Este trabajo cristaliza el esfuerzo llevado adelante durante un largo tiempo, casi una época, marcada por el intento de hacer aquello que la Ciencia debería hacer más frecuentemente, pero que a duras penas consigue a veces (y que alguno categóricamente afirmó que no hace): pensar. Si tanto esfuerzo rindió algún fruto, con seguridad muy modesto, sólo pudo ser posible gracias a la colaboración y el apoyo de muchas personas que por fortuna se cruzaron en mi camino. A todos ellos quiero agradecerles sinceramente, y a algunos que vienen a mi (mala) memoria ahora, en particular, a riesgo de olvidar injustamente a otros:

A Javier, por no dirigirme sino acompañarme durante todo este tiempo con una generosidad difícil de describir; por enseñarme a ser honesto con mis propias ideas.

A los miembros del tribunal, Damián, Miguel y Germán, por su atenta lectura; sus comentarios y sugerencias ayudaron a mejorar este trabajo.

A mis colegas de la FaMAF, con quienes compartimos innumerables discusiones sobre nuestro quehacer como docentes investigadores; a Javier, Miguel y Ara, por contagiarme el espíritu crítico, por tantos encuentros y algunos desencuentros; gracias Laura, Damián, Frito y Pedro por las buenas palabras, por el genuino interés.

A Pío, Martin, Tomás y Lucas, con quienes tuve la fortuna de trabajar en los temas que componen esta tesis, por el siempre fructífero intercambio de ideas que enriqueció nuestro trabajo común.

A Leti, que fue una gran compañera durante buena parte de este tiempo; de ella aprendí lo que significa ser docente.

A los integrantes del Laboratorio de Comunicaciones Digitales (FCEFyN, UNC), y en particular a Jorge Finochietto; junto a ellos encontré, con alegre sopresa, un espacio para desarrollar mi actividad científico-técnica; me recibieron con mucha generosidad, valorando mi trabajo, y me incentivaron a finalizar la carrera, brindándome su apoyo y las condiciones materiales para lograrlo.

A Miguel (otra vez!); cada *fika* fue una necesaria pausa en tiempos de desasosiego.

A los amores, los que fueron, los que son, los que serán siempre; mi mamá y mi papá, mis hermanos, la bulliciosa familia, mis hermanos-grises, tantos hermosos amigos; todos hacen que el esfuerzo que hoy culmina tenga pleno sentido.

A José, por su invaluable y pacífica compañía, por hacer posible cada mañana. *Si no hay amor, que no haya nada entonces...*

Índice general

Resumen III

Agradecimientos V

1 Introducción **1**
1.1. Verificación formal 1
1.2. Lógicas *a la Hoare* y memoria dinámica 4
1.3. *Separation Logic* 8
1.4. *Sharing Logic* 10
1.5. Automatización y decidibilidad 12
1.6. La organización de este trabajo 14

2 Intérpretes **15**
2.1. Funciones de interpretación y sistemas computacionales 17
2.2. Jerarquías de intérpretes 22
2.3. La verificación formal de programas 27
2.4. El pancomputacionalismo 32
2.5. Discusión 37

3 *Separation logics* **41**
3.1. Fórmulas y modelos 42
3.2. Cálculo de fórmulas 47
3.3. El lenguaje de programación 51
3.4. Especificaciones y razonamiento sobre programas 55
3.5. Estructuras de datos 61
3.6. Una lógica con tipos abstractos de datos 72

4 *Sharing Logic* **87**
4.1. Fórmulas con *specified sharing* 93
4.2. Cálculo de fórmulas 97
4.3. Sistema deductivo y reglas de *frame* 102
4.4. Tipos abstractos de datos con *sharing* 109
4.5. Discusión 116

5 Decidibilidad de la *Sharing Logic* **123**
5.1. Modelos indiscernibles 125
5.2. Modelos estructurales 136
5.3. Procedimiento de decisión 138
5.4. Discusión 139

6 Automatización de *Separation Logic* **143**
6.1. Semántica concreta 145
6.2. Semántica intermedia 147

6.3. Semántica abstracta: el análisis 158
6.4. Analizando grafos generales 167
6.5. Resultados experimentales 170
6.6. Discusión 172

7 Palabras finales **177**

Apéndice **181**
Resultados del capítulo 2 181
Resultados del capítulo 3 181
Resultados del capítulo 4 181
Resultados del capítulo 5 183
Resultados del capítulo 6 193

Bibliografía **195**

Introducción

1

1.1. Verificación formal

Es muy frecuente que los programadores cometan *errores* cuando construyen sistemas informáticos, algunos de los cuales pueden tener consecuencias dramáticas. En la historia reciente existe una gran cantidad de casos desafortunadamente famosos en los que un error de *software* dió lugar a la pérdida de cifras millonarias e incluso vidas [149]. En un orden más cotidiano, se puede arriesgar que cualquier persona ha experimentado alguna falla de un sistema informático, con mayores o menores consecuencias. Dada la ubicuidad cada vez mayor de las computadoras y la creciente dependencia, en muchos casos crítica, de ellas en todos los ámbitos de la vida, las consecuencias de la falta de calidad de los sistemas informáticos, lejos de circunscribirse a casos paradigmáticos, se hacen cada vez más evidententes y cada vez más significativas.

La obtención de programas correctos, es decir, aquellos que se ejecutan sin errores y computan el resultado esperado, ha sido uno de los principales objetivos de las ciencias de la computación desde sus inicios. Lejos de perder vigencia, este propósito se convierte en una necesidad cada vez mas imperiosa. La verificación formal de programas es el intento de dar certezas de nivel matemático de que un programa se comporta como se espera, que cumple su *especificación*, típicamente expresada como una fórmula lógica. Una prueba matemática de la corrección de un programa aporta mayor certeza que otras prácticas comunes dentro la ingeniería de *software*, como son la mera inspección del código fuente (aún llevada a cabo de forma extremadamente cuidadosa) y la ejecución de casos de prueba (*testing*), con un alcance necesariamente limitado. Tal certeza acarrea, por supuesto, un esfuerzo significativo por parte de aquellos encargados de escribir el código del programa, la especificación del mismo y la demostración formal de la adecuación entre ellos. Sin embargo puede pensarse que este esfuerzo se encuentra de alguna manera acotado. La demostración de corrección de un programa no se asemeja a una demostración matemática típica. Usualmente un teorema matemático no ofrece pistas que permitan encontrar una demostración de su corrección. Por el contrario, un programa está escrito por un programador que inevitablemente tiene una comprensión, al menos informal, sobre cómo se comporta y por qué cumple su función. Idealmente, el razonamiento formal debería seguir las intuiciones dadas por el razonamiento informal, ayudando a hacer evidente los errores que puediera contener el programa, o la ausencia de ellos.

La idea de construir programas de forma sistemática en conjunto con la demostración de su corrección utilizando especificaciones formales pertenece a una tradición fundacional dentro de las ciencias de la computación. Los beneficios de adoptar una metodología de esta índole quedan demostrados, entre otros, en los numerosos trabajos de E. W. Dijkstra, C. A. R. Hoare, D. Gries, R. Backhouse, R. J. J. Back, etc. [61, 80, 76, 5, 4].

Lamentablemente la verificación formal no forma parte de las herramientas usuales utilizadas por los programadores en su tarea habitual. Su aplicación se reduce en el mejor de los casos a contextos altamente críticos. Esto se debe a diferentes factores: el esfuerzo que conlleva este tipo de metodologías, la falta de herramientas para su automatización, la poca adecuación de las técnicas a las prácticas comunes de programación, entre otras. Este trabajo se inscribe en la tradición antes mencionada y pretende contribuir a salvar la distancia que impide la utilización de métodos formales en la implementación de sistemas informáticos con manejo dinámico de la memoria y así ayudar a aumentar cualitativamente la fiabilidad sobre su corrección y seguridad

La naturaleza de los programas y la verificación formal

Aunque no está exenta de críticas, el valor epistemológico de la verificación formal se da por descontado dentro de la comunidad de las ciencias de la computación. En todo caso, los cuestionamientos a esta metodología se vinculan a su factibilidad y practicidad frente a otras prácticas de la ingeniería de *software*. Sin embargo, la verificación formal ha sido puesta en cuestión de manera más o menos explícita, por diferentes autores provenientes de la filosofía de la computación [58, 72, 8, 68].

El debate sobre la verificación formal tiene que ver, en parte, con la presunta naturaleza dual de los programas: mientras que el *program-script*, el *texto* de un programa es una entidad abstracta (que caracteriza comportamientos de algún modo abstractos), el *program-process*, el proceso de computación al que da lugar, es una entidad temporal y causal, y si bien no física, depende de la existencia de un sustrato físico que la ejecute. Cuando los programas y computaciones se presentan de esta manera, se genera un *gap* ontológico que tiene consecuencias teóricas: ¿cómo relacionar las entidades abstractas con las físicas? Este es el problema de la implementación, del que se derivan otras cuestiones de índole práctico: ¿qué clase de conocimiento produce el razonamiento formal sobre programas? ¿está justificada nuestra confianza en la verificación formal?

Pese a diversos intentos de dar cuenta de la noción de implementación, este concepto está lejos de estar claramente definido [145]. En [127] Piccinini pone de manifiesto algunas de sus dificultades:

> The problem of computational implementation may be formulated in a couple of different ways. Some people interpret the formalism of computability theory as defining abstract objects. According to this intepretation, Turing machines, algorithms, and the like are abstract objects. But how can a concrete physical system implement an abstract object? Other people treat the formalism of computability theory as abstract computational description. But how can a concrete physical system satisfy an abstract computational description?

Uno de los primeros intentos de solución es el *simple mapping account*, introducido y desarrollado en [130, 139]. Según esta idea, un sistema físico S implementa una computación abstracta C cuando pueden ponerse en correspondencia los estados de S (de acuerdo a alguna descripción física), con los estados abstractos de C, de tal manera que toda transición entre estados físicos de S refleje una transición en C. Dicho de otra manera, para que S implemente C debe ocurrir que si existe una transición del estado c_i al estado c_{i+1} en C, entonces el sistema físco S se mueve del estado s_i, que representa c_i, al estado s_{i+1}, que representa c_{i+1}, en alguna escala de tiempo establecida.

Esta definición de implementación es simple y paradigmática, por una de sus principales consecuencias: bajo este marco es fácil demostrar que cualquier sistema físico con suficientes estados, como una piedra al sol, o un balde de agua sometido a cambios de temperatura, implementa cualquier computación. De esta manera se trivializa el concepto de sistema computacional, y da lugar a lo que se da en llamar el problema del *pan-computacionalismo* [127].

Es claro que en el *simple mapping account* el énfasis está puesto en la computación vista como cambio de estados, olvidando que computar es realizar una serie de operaciones mecánicas dadas por reglas explícitas. En esta solución, tales reglas no aparecen. Pueden inferirse o presuponerse, pero están sólo en un nivel abstracto, sin existencia concreta en el propio sistema. Otras propuestas [56, 125, 126, 45, 138, 53] intentan evitar la trivialización del pan-computacionalismo, introduciendo requisitos más fuertes (como contrafácticos causales, descripciones disposicionales, o caracterizaciones mecanicistas).

Creemos que una característica distintiva que hace *computacional* a un sistema es que sea *programable*: que pueda inducirse al sistema a comportarse de diversas maneras a partir de proveer una codificación, *i.e.* un programa, que prescriba tales comportamientos. Esto presupone los dos *modos de existencia* de un programa: en tanto artefacto abstracto y en tanto realización concreta dentro del sistema.

Como contribución de este trabajo, proponemos que lo que define a un sistema como computacional, es una relación de implementación entre el sistema y una descripción abstracta del mismo (que denominamos *función de interpretación*). Pero tal relación debe entenderse, ya no como la sóla vinculación entre el *script* del programa y el proceso resultante, sino como una relación entre relaciones: la que vincula un

programa (en tanto abstracto) con la caracterización de los comportamientos que prescribe, y la que relaciona una realización concreta de tal programa dentro del sistema con el proceso que induce en él (o el estado final que este produce). Mas aún, la relación de implementación entre una función de interpretación y un sistema *físico* puede estar mediada por otras descripciones computacionales abstractas, lo que permite justificar los múltiples niveles de *abstracción* presentes en los sistemas computacionales paradigmáticos, como las computadoras digitales: electrónica digital, código máquina, sistema operativo, lenguaje de programación de alto nivel, etc.

La implementación comprendida de esta manera da lugar al concepto de *intérprete* y permite lograr una caracterización homogénea de conceptos clave de la teoría de la computación (como computación, programa y sistema computacional), y justificar prácticas comunes en las ciencias de la computación (como la verificación formal y el uso de formas de abstracción en la implementación de sistemas informáticos).

1.2. Lógicas *a la Hoare* y memoria dinámica

La lógica de Hoare [80] es uno de los formalismos más prominentes para razonar sobre programas imperativos. El concepto clave de esta lógica es la terna de Hoare, de la forma $\{p\}\ c\ \{q\}$. Una terna asocia dos *aserciones* p y q, la precondición y la postcondición respectivamente, con un programa c. Normalmente las aserciones son fórmulas de primer orden que describen los estados del programa (el valor de cada una de las variables del mismo). La precondición p describe los estados que se satisfacen antes de la ejecución del programa c, mientras que la postcondición q los estados que se satisfacen luego de su ejecución. La lógica de Hoare incluye un sistema de prueba axiomático que permite establecer la validez de las ternas. Cuando una terna $\{p\}\ c\ \{q\}$ se verifica, significa que si el programa c se inicia en algún estado que satisface la precondición p y termina exitosamente, entonces lo hace culminando en un estado que satisface la postcondición q. Esto se conoce como corrección parcial y no establece ningún compromiso sobre los programas que no terminan. Así mismo, en el trabajo original, Hoare no considera los programas que abortan, lo que dió en llamar *corrección condicional*.

A pesar de los numerosos trabajos posteriores que extendieron el soporte de la lógica de Hoare para dar cuenta de diversas construcciones usuales en los lenguajes de programación *reales*, el uso de memoria dinámica representa un reto difícil de abordar [33, 107, 23]. En palabras de Reynolds [136]:

> Approaches to reasoning about this technique have been studied for three decades, but the result has been methods that suffer from either limited applicability or extreme complexity, and scale poorly to programas of even moderate size.

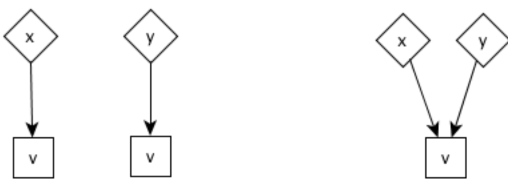


Figura 1.1: Posibles modelos de $x \mapsto v \wedge y \mapsto v$

La memoria dinámica es una de las facilidades más extendidas en el universo de la programación. Permite la construcción de estructuras de datos de tamaño ilimitado, implementación eficiente de una variedad de patrones recurrentes, y da soporte a uno de los paradigmas más utilizados, la programación orientada a objetos. La introducción de memoria dinamica divide el espacio de memoria de un programa en dos: aquellas registros estáticos, definidos en tiempo de compilación y asociados inequívocamente a variables de programa que pertenecen al *stack*; y aquellos registros dinámicos que pertenecen al *heap*, definidos en tiempo de ejecución, anónimos y únicamente referenciados de forma indirecta por variables del *stack* o por otros registros del mismo *heap* (que se denominan punteros o referencias). La principal dificultad para razonar (formalmente) sobre la memoria dinámica es el problema del *aliasing* [95, 83]. Dos o más punteros que referencian al mismo registro de memoria en el *heap* se denominan *alias*. Cuando esto ocurre cualquier modificación a la memoria a través de un puntero pueden repercutir inadvertidamente en la memoria referenciada por otro, y por lo tanto, alterar la validez de cualquier propiedad expresada en términos de este último.

El uso de lógicas de primer orden para el razonamiento formal sobre la memoria dinámica es probadamente ineficaz. La dificultad dada por el *aliasing* se evidencia en la dificultad para expresar el efecto de comandos simples, como la mera modificación de un registro de memoria. Supongamos que la fórmula $x \mapsto v$ expresa que existe un registro en el *heap* apuntado por la variable x cuyo valor es v, y que el comando de mutación $[x] := v'$ modifica el registro referenciado por x asignandole el valor v'. Es esperable que la siguiente terna sea válida:

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

Sin embargo el efecto de la mutación no puede expresarse tan sencillamente en un contexto más amplio, cuando existen otros punteros. La terna

$$\{\, x \mapsto v \wedge y \mapsto v \,\}\; [x] := v' \;\{\, x \mapsto v' \wedge y \mapsto v \,\}$$

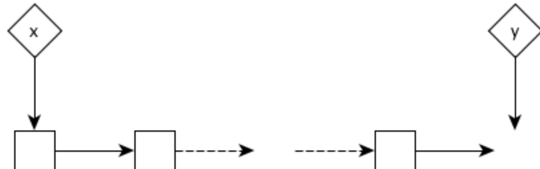


Figura 1.2: Modelo de segmento de lista enlazada.

ya no es válida, porque no hay manera de saber si x e y referencian al mismo registro (fig. 1.1).[1] Frente a esta dificultad, parecería inevitable caer necesariamente en un costoso *razonamiento global*. En este caso particular una precondición como $x \mapsto v \wedge y \mapsto v \wedge x \neq y$ puede ser suficiente para excluir la posibilidad de *aliasing*. Sin embargo expresar la inexistencia de múltiples *alias* sobre un registro de memoria en lógica de primer orden no es sencillo en el caso general, más aún si consideramos que los registros del *heap* no sólo pueden ser referenciados por variables del *stack* sino también por otros registros del mismo *heap*, que son *anónimos*. La cuestión se vuelve más compleja con la utilización de predicados que definen estructuras de datos enlazadas como listas, árboles binários, etc. Supongamos que el predicado $\textbf{lseg}.x.y$ define un segmento de lista enlazada que comienza en el registro apuntado por x y termina en un registro que apunta a la dirección de memoria también referenciada por y. Una representación esquemática se muestra en la figura 1.2. Para poder razonar sobre esta estructura de datos, el predicado debe codificar, entre otras cosas, que los registros que componen la lista son diferentes entre sí y que no existe circularidad (fig. 1.3). Para expresar la existencia de dos listas que no comparten memoria, es necesario recurrir a predicados auxiliares. Una opción es utilizar un fórmula como la siguiente:

$$\textbf{lseg}.x.y. \wedge \textbf{lseg}.w.z \wedge (\forall j,k \cdot \textbf{reachable}.x.j \wedge \textbf{reachable}.w.k \Rightarrow j \neq k)$$

donde $\textbf{reachable}.x.i$ expresa el hecho que la dirección de memoria i es alcanzable directa o transitivamente desde x. El uso de tal predicado inhibe el *sharing* de memoria entre los segmentos de lista, pero también prohibe cualquier vínculo entre ellos. Por ejemplo, especificar que dos segmentos están enlazados conformando un segmento mayor (fig. 1.4) requiere predicados auxiliares aún más sofisticados.

[1] En la graficación de estados de memoria, utilizamos un rombo con un nombre de variable para representar una variable del *stack*, que referencia al registro del *heap* hacia el que se dirije la flecha. No representamos variables del *stack* cuyos valores no sean direcciones de memoria. Las variables del *stack* no se representan exhaustivamente ya que usualmente son infinitas. Utilizamos rectángulos de uno o más campos para representar un registro en el *heap*. Cada campo puede contener un valor particular. Cuando el valor es irrelevante no se representa. Cuando contiene como valor una dirección de memoria, se representa como una flecha que apunta hacia el registro referenciado. Cuando las flechas no apuntan a ningún registro, es porque son direcciones de registros que no se encuentran en el *heap*. No existen en el *heap* más registros que los representados. Utilizamos flechas discontinuas para representar una secuencia de registros enlazados consecutivamente, de longitud arbitraria.

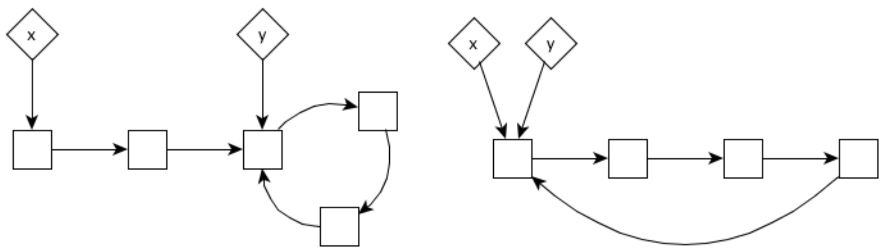


Figura 1.3: Modelos no deseados de segmentos de lista enlazada.

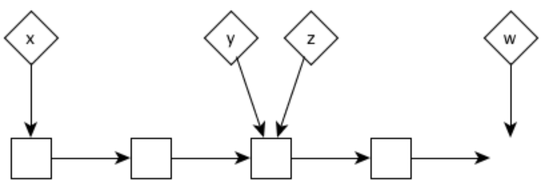


Figura 1.4: Dos segmentos de lista enlazados

La principal falencia de la lógica de primer orden para el razonamiento sobre punteros es que el *aliasing* se da por defecto. Mientras que la existencia de alias en un programa es una situación relativamente rara y que se da de forma controlada, en una fórmula cualquier par de variables puede referenciar al mismo registro de memoria salvo que se exprese explícitamente lo contrario. Así comienza a ser necesario utilizar predicados *ad hoc* que faciliten la tediosa tarea de comprobar que las propiedades de interés se actualizan apropiadamente o se preservan frente a cualquier mutación del *heap*. Este ha sido el camino escogido por una variedad de formalismos [2, 109, 7, 22, 44, 82, 97, 108, 118]. Otros han optado por adoptar simplificaciones sobre el modelo de memoria que otorgan algunas garantías sobre el *aliasing* [14, 34, 70, 105]. En cualquier caso no es de extrañar que estos abordajes no hayan madurado ya que razonar dentro de ellos se vuelve extremadamente complicado y limitado.

A pesar de ser central, el *aliasing* no es la única dificultad que deviene del uso de memoria dinámica. La misma naturaleza de los punteros introduce problemas sutiles. Ejemplo de ello es asegurar el correcto manejo de los punteros *dangling* (aquellos que referencian a registros de memoria inexistentes) para evitar su indebido acceso, o garantizar que no queden definidos registros de memoria que no son alcanzables desde ningún puntero (*memory leak*). Por otro lado, el uso de punteros para definir estructuras de datos enlazadas permite implementaciones eficientes, optimizar el uso de memoria a través del *sharing* en las implementaciones de diferentes instancias, aunque también favorece la aparición de ciclos no deseados. Dar cuenta de estas situaciones es importante no sólo para garantizar la funcionalidad de los programas

y su eficiencia, sino tambien la ausencia de terminaciones inesperadas (como cuando un programa aborta por un error en la gestión de la memoria) o agujeros de seguridad (como los dados por *buffer overflow*).

El razonamiento formal sobre la utilización de memoria dinámica es tan suceptible a errores como la misma programación. Para que efectivamente pueda darse una fructífera interrelación entre la programación y la verificación de su corrección se debe contar con una lógica concisa con gran poder expresivo, que disponga de los recursos necesarios para abordar los potenciales problemas que derivan del uso de punteros, y un sistema de prueba útil que facilite el razonamiento.

1.3. *Separation Logic*

Un marco de trabajo que surge a principios del milenio y se diferencia de los anteriores, es la *Separation Logic* [136, 85, 155, 116]. Este formalismo trata los problemas asociados a los punteros de manera sintáctica a partir del uso de un novedoso lenguaje de fórmulas. Es una extensión de la lógica de Hoare que permite razonar sobre programas imperativos de bajo nivel que utilizan estructuras de datos dinámicas y ha sido utilizada con mucho éxito en la demostración de programas que realizan una manipulación sutil y compleja de punteros [151, 15]. Extiende la lógica de especificación de primer orden introduciendo operadores espaciales ($*$, $-\!\!*$) que permiten describir patrones de separación espacial entre regiones de memoria, estableciendo la ausencia de alias. La principal ventaja frente a otros formalismos es que promueve el *razonamiento local*: los nuevos operadores permiten extender la validez de la corrección de una porción de código a un contexto de memoria general siempre que se garantice su separación espacial.

El sistema de prueba de la *Separation Logic* permite verificar una terna $\{p\}\ c\ \{q\}$ mencionando única y explícitamente en la precondición p la región manipulada por c, denominada *footprint*. Luego es posible extender su validez a cualquier contexto de memoria más amplio especificado por un invariante i, deduciendo la terna

$$\{p * i\}\ c\ \{q * i\}$$

Aquí la conjunción espacial $*$ asegura que la porción de memoria que satisface i es completamente disjunta del *footprint* de c.

El concepto de *footprint* intenta capturar la intuición del programador cuando razona sobre la memoria que está manipulando. Por ejemplo, el cuerpo de un ciclo que recorre una lista enlazada se enfoca únicamente en el elemento actual, el anterior y el posterior, y asume que el resto de la lista permanece inmutable. La *Separation Logic* permite razonar de la misma manera, facilitando el desarrollo de una demostración de corrección que acompaña la escritura del código y las ideas subyacentes. Así con-

trasta fuertemente con los formalismos antes mencionados que obligan a considerar la memoria en su totalidad.

El nuevo lenguaje de fórmulas describe los estados de un programa como una conjunción de registros de memoria dinámica individuales. La lógica garantiza que los registros combinados con $*$ son disjuntos entre sí, permitiendo así describir precisamente los efectos producidos sobre la memoria por cada manipulación del *heap*. Así la terna

$$\{ x \mapsto v \}\ [x] := v'\ \{ x \mapsto v' \}$$

no sólo es válida para el caso particular sino que se convierte en un axioma que caracteriza de forma precisa los efectos de la ejecución del comando de mutación. De la misma manera todos los comandos de manipulación del *heap* se pueden caracterizar utilizando una cantidad mínima de registros. Esta particularidad es la que permite escribir demostraciones modulares que luego pueden ser compuestas para producir una prueba de corrección sobre el total del código. A diferencia de lo que sucede con la lógica de Hoare estándar, la terna

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

es válida ya que la memoria referenciada por x es necesariamente diferente a aquella referenciada por y.

La caracterización de estructuras de datos también se beneficia del lenguaje de fórmulas. Resulta muy sencillo definir el predicado $\mathbf{lseg}.x.y$ con la utilización de $*$ para evitar ciclos, donde y es un puntero *dangling*. Retomando el ejemplo propuesto en la sección anterior, la especificación de un estado con dos segmentos de lista independientes se reduce a la fórmula $\mathbf{lseg}.x.y * \mathbf{lseg}.z.w$. Aquí se puede apreciar además cómo la *Separation Logic* se comporta respecto a los punteros *dangling*. Esta especificación permite que y, la dirección de memoria apuntada por el último registro del segmento sea igual a z, enlazando el primer segmento con el segundo (como en la figura 1.4). Esto muestra la buena interacción entre los punteros *dangling* y el operador $*$, lo que permite describir fácilmente no sólo estructuras de datos en su totalidad, sino también estructuras de datos *parciales* y combinaciones de ellas.

Mencionamos anteriormente el problema de un acceso indebido a un registro inexistente. Un acceso de este estilo ocasiona normalmente que el programa aborte o siga su ejecución de una manera imprevisible. La *Separation Logic* trata con este asunto abandonando la semántica original de *corrección condicional*, y asegurando que un programa demostrado no falla por errores de memoria. Un terna $\{ p \}\ c\ \{ q \}$ se interpreta como sigue: si el programa c comienza en un estado que satisface p, *entonces su ejecución no falla* y si termina, lo hace en un estado que satisface q.

Separación y abstracción

La modularización, abstracción e *information hiding* son nociones muy utilizadas para lidiar con la complejidad de programas de gran tamaño. Esto se logra normalmente utilizando técnicas de tipos abstractos de datos o el paradigma de programación orientada a objetos. A pesar de estar diseñada para un lenguaje de bajo nivel, numerosas extensiones de la *Separation Logic* han abierto la posibilidad de su aplicación a lenguajes orientados a objetos [117, 91, 13, 51, 103], incluso a un subconjunto del lenguaje Java con características avanzadas como herencia múltiple, *dispatch* dinámico, etc. [119, 121, 77]. Estas extensiones permiten la verificación composicional e independiente de los distintos componentes de un sistema de software de manera compatible con las nociones de abstracción e *information hiding*, aunque con ciertas limitaciones.

El sistema deductivo de la *Separation Logic* presenta una clara ventaja para razonar cuando es posible garantizar la separación espacial. Sin embargo no provee herramientas para la descripción y razonamiento sistemático en situaciones donde existen múltiples estructuras en el *heap* que comparten memoria para su representación. Tales estructuras representan múltiples vistas sobre la memoria y es deseable poder razonar independientemente con cada una de ellas de forma acorde a las intuiciones que los programadores puedan tener sobre ellas. Escenarios como estos son muy comunes en sistemas de tamaño real y se encuentran incluso en las librerías estándar de los lenguajes de programación más utilizados. La potencia de la *Separation Logic* encuentra su límite en estas situaciones, obligando a razonar de manera *ad hoc* utilizando sutiles predicados que garanticen de alguna manera la separación espacial de las estructuras involucradas, rompiendo usualmente las abstracciones que las mismas proveen.

1.4. *Sharing Logic*

Una contribución de este trabajo es una generalización de la *Separation Logic* que permite especificar de forma precisa estructuras dinámicas complejas, relaciones de separación y de *sharing* entre ellas, de manera compatible con los principios de abstracción e *information hiding*. La lógica incluye un sistema de prueba composicional para verificar programas de forma modular bajo ciertas condiciones que pueden ser expresadas y demostradas utilizando el propio lenguaje de fórmulas. Este marco permite precisar las condiciones para razonar modularmente sobre estructuras en la memoria dinámica cuando no es posible garantizar su separación espacial.

Mientras que los operadores espaciales de la *Separation Logic* permiten especificar la absoluta separación espacial de regiones del *heap*, nuestra propuesta se centra en la posibilidad de describir simultáneamente relaciones de separación espacial (parcial)

y relaciones de *sharing*, a partir de nuevos operadores. La idea principal consiste en una generalización del operador binario de conjunción espacial $*$ con un operador ternario $\langle *\colon r\rangle$, donde r es una fórmula de la misma lógica. Informalmente, la fórmula $p\ \langle *\colon r\rangle\ q$ especifica dos regiones del *heap*, una que satisface p y otra que satisface q, disjuntas salvo por una subregión común que satisface r. Con nuestro lenguaje de fórmulas y el uso de reglas para extender la validez de una terna de Hoare a contextos de memoria generales, es posible asegurar la corrección de programas, razonando localmente sobre cada una de las múltiples vistas sobre el *heap*.

Nuestro desarrollo utiliza un lenguaje imperativo simple con comandos para la manipulación de la memoria dinámica, la definición de funciones, y el soporte de tipos abstractos de datos. Para razonar sobre los programas utilizamos una especificación de corrección parcial en forma de terna de Hoare, extendida con un contexto formado por definiciones de *predicados abstractos* [120], especificaciones para las funciones y *especificaciones estáticas*.

El concepto de predicado abstracto permite aproximarnos al tipo de razonamiento intuitivo asociado con los tipos abstractos de datos, ya que modela el principio de *information hiding* de manera sencilla y cercana a la concepción del programador. Los *predicados abstractos* así como los tipos abstractos de datos poseen un nombre, una implementación dada por la definición del predicado, y un *scope* dentro del cual se conocen los detalles de la misma. Dentro del *scope* la definición puede intercambiarse libremente con el nombre del predicado, dando la posibilidad al desarrollador de demostrar la corrección de las funciones. Sin embargo fuera del mismo, es decir en el código cliente, el predicado abstracto sólo puede manipularse atómicamente a través de su nombre, con las especificaciones de las funciones y especificaciones estáticas que lo mencionan.

La especificación de una función caracateriza a través de una terna de Hoare la alteración de estado que produce su ejecución desde el punto de vista del cliente. Del mismo modo que ocurre con los comandos elementales, las aserciones sólo mencionan las partes de memoria estrictamente necesarias sin considerar el estado global, incluso cuando pueda existir *sharing* con otras estructuras abstractas.

Las especificaciones estáticas proveen información adicional a las especificaciones de las funciones. Caracterizan de manera sencilla e intuitiva las restricciones en la interacción de diferentes estructuras abstractas y los contextos más generales donde pueden aparecer. Permiten razonar sobre los tipos abstractos de datos utilizando sólo el nombre de los predicados abstractos (y no su implementación) en el código cliente. En particular las especificaciones estáticas son útiles para extender la validez de las ternas de Hoare a contextos de memoria generales frente a la existencia de *sharing*, y permiten razonar sobre instancias de un tipo abstracto de datos sin romper el principio de *information hiding*.

1.5. Automatización y decidibilidad

La verificación automática de algunas propiedades asociadas al uso de memoria dinámica puede abordarse con técnicas específicas que en muchos casos dan buenos resultados. Ejemplo de esto es la verificación de que un programa no accede a punteros *dangling*, que no libera memoria que no se encuentra definida, etc. Sin embargo, con el uso de estructuras de datos dinámicas, incluso las más simples como listas enlazadas, se abre la puerta a un conjunto potencialmente infinito de estados de memoria, y por lo tanto se encuentran los mismos límites teóricos que presentan los sistemas de estados infinitos. Poder establecer automáticamente la correcta manipulación de tales estructuras es aún un problema abierto, difícil, aunque relevante.

La posibilidad de que una lógica para el razonamiento formal sobre programas como la *Separation Logic* o la *Sharing Logic* escale a sistemas reales, depende fuertemente de que existan herramientas (semi) automáticas que faciliten su uso. Dada la complejidad intrínseca de las pruebas de tales lógicas, es una gran ventaja que las mismas puedan ser verificadas automáticamente. Más aún, la posibilidad de que ciertas pasos de prueba elementales puedan ser resueltos por procedimientos de decisión posibilita que el programador se enfoque en los aspectos realmente relevantes.

La decidibilidad del problema de validez de la *Separation Logic*, esto es, la posibilidad de decidir mecánicamente si una fórmula se satisface en cualquier modelo arbitrario, ha sido estudiado inténsamente. En el trabajo fundacional [43] se demuestra la indecidibilidad para la *Separation Logic* sobre modelos de memoria con registros de dos campos, incluso excluyendo los operadores espaciales ($*$, $-\!\!*$) por reducción al problema de satisfactibilidad de la lógica clásica de predicados con una relación binaria. Al mismo tiempo se muestra la decidibilidad para un fragmento proposicional (que excluye todo tipo de cuantificación), estableciendo la propiedad de modelos finitos. En ningún caso se consideran predicados para describir estructuras de datos enlazadas.

Más tarde el problema general fue resuelto en [30], donde se demuestra la indecidibilidad de la *Separation Logic* sobre registros con un único campo, mostrando su reducción a la lógica de segundo orden. En el mismo trabajo se demuestra, por otro lado, la decidibilidad de la *Separation Logic* con cuantificadores de primer orden, pero excluyendo la implicación espacial ($-\!\!*$) mediante una traducción con complejidad LOGSPACE a una lógica monádica de segundo orden. Al incluir cuantificadores, la lógica estudiada permite definir nociones de alcanzabilidad, habilitando la especificación de listas enlazadas.

En el presente trabajo abordamos la decidibilidad del problema de validez de un fragmento proposicional de nuestra *Sharing Logic*, que excluye la implicación espacial, pero incluye un predicado para listas enlazadas. El artefacto teórico utilizado sigue la línea de [43], es decir, la validez de una fórmula se verifica contra un conjunto

finito de modelos que representan al total. Esto nos permite comprender las características de los modelos que habilitan la decidibilidad. Como un resultado derivado obtenemos una nueva prueba de la decidibilidad de un fragmento de la *Separation Logic* con propiedades similares, que si bien está contemplado por los resultados de [30] utiliza medios muy diferentes.

En un intento de abordar la complejidad del problema de decisión, Berdine *et al.* presentan en [10] un procedimiento para decidir la validez de *entailments* de un fragmento reducido de la *Separation Logic* en tiempo polinomial [55]. Este fragmento elimina la mayor parte de los operadores y cuantificadores de la lógica clásica, pero introduce un predicado específico para listas enlazadas. Separa las formulas espaciales de las formulas clásicas inhibiendo especificaciones complejas y cualquier tipo de *sharing*. Aún con una expresividad reducida, permite describir propiedades interesantes en una gran colección de programas que manipulan listas enlazadas. Las fórmulas del fragmento, aunque simples, son compatibles con las precondiciones de las ternas que caracterizan los comandos de manipulación del *heap*. Esto abre las puertas para la utilización de la *Separation Logic* en la definición de diversos *shape analysis*.

El término *shape analysis* denomina a una familia de análisis estáticos que verifica propiedades sobre programas que manipulan estructuras dinámicas. El tipo de propiedades que contempla tiene que ver con la *forma* de las estructuras, enfocandose en los enlaces que generan los punteros e ignorando los datos representados. Por ejemplo, es posible verificar que a lo largo de toda la ejecución de un programa se mantiene una lista enlazada o un árbol binario, que no se generan ciclos dentro de la estructura, que su modificación no genera *memory leaks*, etc. Un *shape analysis* básicamente ejecuta de forma *simbólica* el programa y lleva una representación del estado de la memoria, utilizando para ello diferentes dominios abstractos como *Pointer Assertion Logic* [88], *Three-valued Logic* [98], *Shape graphs* [27], *Separation Logic* [12, 11, 63], etc. La ventaja de la *Separation Logic* frente a otros formalismos está dada no sólo por la concisión de su lenguaje de especificación sino también por la modularidad de su sistema de prueba. Esto permite la definición de una familia de análisis que extienden la expresividad (incluyendo propiedades de tamaño, ordenación, estructuras anidadas, etc.) [112, 9], y la construcción de herramientas aplicables a sistemas grandes como el *kernel* Linux [153, 65, 67, 62, 35].

En los trabajos mencionados, las estructuras de datos soportadas son lineales, posiblemente combinadas de formas intrincadas, como listas doblemente enlazadas, listas de listas, etc. Como última contribución, exploramos la posibilidad de extender el *shape analysis* de [63] para soportar estructuras de datos no lineales. Preferimos montarnos sobre el marco de la *Separation Logic* en lugar de la *Sharing Logic*, por su menor complejidad. Sin embargo incluimos la posibilidad de especificar no sólo estructuras como árboles, sino otras con formas restringidas de *sharing*, como grafos

acíclicos dirigidos y grafos generales. Presentamos un *shape analysis* con garantías de terminación, que dada una precondición y un programa computa automáticamente una postcondición e invariantes para cada ciclo. Introducimos dos predicados linealmente recursivos que permiten la descripción de estructuras de datos no lineales. Además incorporamos un novedoso análisis para ajustar el nivel de abstracción en la computación de invariantes, permitiendo prever los términos que se requieren presentes en las fórmulas para la consecución de la ejecución simbólica. Todo esto permite manejar adecuadamente la complejidad no lineal intrínseca a las estructuras de datos soportadas.

1.6. La organización de este trabajo

La presente tesis se organiza de la siguiente manera. En el capítulo 2 caracterizamos la noción de computación a través del concepto de intérpretes. Es el resultado de diversos trabajos [17, 20, 19] publicados en conjunto con Javier Blanco, Martin Diller y Pío García. En el capítulo 3 introducimos el formalismo de la *Separation Logic*, tomando como referencia [136, 85] principalmente. En el capítulo 4 intoducimos nuestra *Sharing Logic*, trabajo que cuenta con la coautoría de Javier Blanco, publicada originalmente en [49]. En el capítulo 5, presentamos el resultado de decidibilidad de un fragmento de la *Sharing Logic*, y como consecuencia, de un fragmento de la *Separation Logic*. Es el resultado de un trabajo en colaboración con Tomás Cohen Arazi. En el capítulo 6 introducimos un *shape analysis* que utiliza como fundamento teórico la *Separation Logic*, publicado en conjunto con Javier Blanco y Lucas Rearte [50]. En cada uno de los capítulos se presenta una discusión y los trabajos relacionados. Para finalizar, en el capítulo 7 se presenta un cierre de la tesis. En el apéndice se incluyen discusiones sobre los resultados teóricos, referencias, y demostraciones de los más relevantes.

2 Intérpretes

Alrededor de la década de 1930, dos concepciones coherentes sobre lo *efectivamente computable* se anclan en diferentes maneras de plantear este problema. Por un lado, se caracteriza los algoritmos como maneras mecánicas de resolver problemas. Por otro, en el marco de los sistemas axiomáticos, la pregunta se coloca en la noción de calculabilidad en una lógica. Si bien los diferentes intentos de caracterizar lo efectivamente computable se demuestan equivalentes (lo que luego se llama la tesis de Church-Turing), no es sino hasta que Turing explicita restricciones estrictas sobre qué significa un paso mecánico de cálculo, que los participantes de estas discusiones (Church, Kleene, Herbrand, Gödel, entre otros) llegan al consenso que el problema está resuelto.

Para Turing [143] las reglas que determinan el proceso de *computación efectiva* son seguidas por el “computer”, una persona actuando rutinariamente con percepción y memoria limitadas, que se cristalizan en la utilización de papel y lapiz, y operaciones elementales con ellos. Así, computar es seguir reglas prescriptas de manera simple, que en el caso del formalismo propuesto, las máquinas de Turing, son tan elementales que ya no es posible ni necesario descomponerlas. Turing justifica a partir de un análisis conceptual que lo que puede ser calculado de esta manera es lo computable.

Gandy [73] extiende la idea de computación a partir de mecanismos no-humanos, caracterizándolos de forma abstracta y determinando el máximo poder de cómputo que puede obtenerse. Uno de sus objetivos explícitos es independizarse de cualquier formalismo particular, presentando sólo restricciones estructurales que limitan lo computable por esos mecanismos. De alguna manera, lo que se obtiene es un metaformalismo que intenta capturar las posibles maneras de definir mecanismos computacionales y las restricciones para ser considerados tales.

Está generalmente aceptado que la caracterización de Turing, luego consolidada por Gandy, captura la noción de computación. Sin embargo, preguntas como qué es computar, cuándo un sistema computa y en qué sentido es computacional, y cuál es la relación entre los aspectos abstractos y concretos de la computación, no se resuelven definitivamente con su propuesta. Tanto la filosofía de la mente, como la filosofía y las prácticas de las ciencias de la computación son una fuente de problemas para las concepciones de computación.

El problema de la relación entre computación abstracta y concreta incluye la cuestión de la naturaleza dual de los programas, con sus derivas en aspectos metodológicos (como la verificación formal y el testing de programas), la preminencia entre la teoría de la computación y la construcción de computadoras, e incluso la eventual bifurcación de la tesis Church-Turing para sistemas abstractos y físicos [127].

En ocasiones se cuenta la historia de la computación como si la construcción efectiva de una computadora física supusiera la elucidación previa de una noción abstracta de computación. Esta forma de defender la preminencia de una concepción abstracta de computación se enfrenta con la evidencia histórica de la potencia expresiva y la flexibilidad de dispositivos como los de Babbage, diseñados cien años antes que las discusiones de 1930 [101]. Esto sugiere que para construir una computadora no haría falta una noción abstracta previa de computación, pero sí parece necesaria para reconocer un artefacto como computadora y definir sus límites computacionales.

Por otro lado, la división entre aspectos abstractos y concretos de la computación a veces se presenta de tal manera que implica una diferenciación ontológica [68]. Así, un programa debería entenderse como una estructura sintáctica que obedece las leyes de la lógica (*program-script*) o como una estructura causal sujeta a la leyes de la física (*program-process*). La ineludible necesidad de un sustrato material que lleve adelante el proceso de computación otorgaría preminencia a la segunda concepción. Esto, junto a la dificultad para relacionar el *reino* de lo abstacto con el mundo material, parece bregar por soluciones que relativizan cualquier conocimiento que no provenga de una experimentación empírica, lo que choca de lleno con las prácticas actuales en el ámbito de las ciencias de computación.

Respecto a la identificación de cuándo un sistema computa o, de forma más general, cuándo es computacional, se puede señalar al pancomputacionalismo como la concepción que parece trivializar la noción de cómputo. La discusión sobre el pancomputacionalismo comienza en el campo de la filosofía de la mente y luego es tomada por la filosofía de la computación. Un problema que aparece en cualquier definición de computación es saber si puede servir para distinguir sistemas que computan de aquellos que no. Putnam y Searle [130, 139], entre otros, elaboran argumentos que mostrarían la ausencia de rasgos específicos en la idea de computación que se expresa como pancomputacionalismo: "todo computa"; y otras veces como relativismo semántico: "la computación está en el ojo del observador".

Estas críticas toman diferentes formas. Quizá la mejor formulada es la que Putnam expresa en forma de teorema, en el que muestra que cualquier sistema de transición (sin entrada ni salida de datos) es implementado por cualquier sistema físico abierto con suficientes estados. La idea es que los estados abstractos de una computación pueden ser realizados por cualquier (clase de equivalencia de) estado(s) físico(s). Si se consideran sistemas de entrada-salida el teorema se debilita, pero igualmente si-

gue planteando cierta trivialización de la noción de cómputo: *cualquier* sistema de transición finito es implementado por *cualquier* sistema físico abierto que exhibe la relación adecuada de entrada-salida.

En este capítulo proponemos una noción relacional de implementación y computación, a través del concepto de *intérprete*. Computar es un proceso determinado por el vínculo prescriptivo entre un programa y un comportamiento. La implementación que determina un sistema computacional como tal, puede ser comprendida como una relación de realización de una descripción abstracta de este vínculo, que cierra el *gap* ontológico entre los aspectos abstractos y físicos de la computación.

La propiedad de programabilidad, *i.e.* la capacidad de un sistema de recibir una prescripción arbitraria y comportarse de acuerdo a ella, es central. Así, creemos que el concepto de intérprete permite resumir los aspectos centrales que caracterizan la noción de computación. Más aún, permite construir jerarquías de sistemas computacionales que habilitan, ya no a contestar por si o no la pregunta sobre cuándo un sistema es computacional, sino a describir *grados* de computación de acuerdo a la programabilidad de los sistemas.

2.1. Funciones de interpretación y sistemas computacionales

Un intérprete es una noción general que provee un marco uniforme para analizar, caracterizar y relacionar mecanismos físcos, como calculadoras y computadoras; una persona actuando mecánicamente, como el computador de Turing, o haciendo reducciones de un término *lambda*; formalismos matemáticos, como la máquina de Turing universal; o incluso *sistemas computacionales* ideales que están más allá de la computabilidad de Turing, como las computadoras oráculo.

Uno de los aspectos comunes a todos los *sistemas* mencionados es que pueden caracterizarse por sus posibles comportamientos. Por *comportamiento* entendemos una descripción idealizada de las ocurrencias de ciertos eventos considerados relevantes. Las diferentes formas de observar un sistema determinan conjuntos diferentes de comportamientos. La parcialidad en la forma de observar sólo lo considerado relevante, abre la puerta a múltiples *realizaciones* de un mismo comportamiento, *i.e.* diferentes *configuraciones* de un sistema pueden ser vistas como equivalentes y consideradas como el mismo comportamiento. Algunos ejemplos de comportamientos que se consideran en la literatura son la salida de datos, trazas de ejecución, grafos de procesos, conjuntos de fallas, *interleavings* de acciones atómicas, etc. Una definición más precisa de comportamiento sólo tiene sentido en un marco particular. Aquí evitamos hacer específica la definición, ya que nos interesa poder capturar el concepto de sistema computacional en su generalidad y consideramos que nuestros resultados se aplican a todos los comportamientos mencionados. A pesar de esto, vamos a ilustrar nuestros ejemplos con el caso más común: la salida de datos.

La idea central que presentamos en este capítulo es que un intérprete produce comportamientos, a partir de una *codificación* de los mismos que acepta como entrada. La relación entre tal entrada, que llamamos *programa*, y el comportamiento que prescribe, se caracteriza mediante una *función de interpretación*.

Definición 1 (Función de intepretación)**.** *Dado un conjunto B de posibles comportamientos, un conjunto P de* programas, *y un conjunto D de datos, una* función de interpretación *es una función $i \in P \to (D \to B)$,*[1] *que asocia a todo comportamiento $b \in B$ un* programa *$p \in P$ y un dato $d \in D$.*

Usualmente referimos a una función de interpretación $i \in P \to B$, asumiendo que los datos de entrada se encuentran codificados con el mismo programa.

Cuando los elementos de P se construyen utilizando algún lenguaje*, lo llamamos* lenguaje de programación*. Cuando consideramos los comportamientos de salida de datos, suele darse que $B = D$.*

Las nociones de función de interpretación y programa son relacionales. Una función de interpretación es tal cuando prescribe comportamientos a un conjunto dado de programas; es una función de interpretación para tal conjunto. Un programa es tal cuando existe una función de interpretación para él. No hay nada intrínseco en ser un programa más alla de la relación con una función de interpretación. Por ejemplo, los números de Gödel permiten considerar los números como programas [101]. De esta manera, los conceptos de función de interpretación, programa (e incluso lenguaje de programación) son interdefinibles.

Decimos que un sistema I es *computacional* respecto a una función de interpretación i (o equivalentemente I *implementa* i) cuando se comporta de acuerdo a la prescripción establecida por i.

Definición 2 (Sistema computacional)**.** *Dados un sistema I y una función de interpretación $i \in P \to B$, decimos que I* implementa *i si es capaz de recibir una realización del programa p como entrada y* sistemáticamente producir *los eventos relevantes caracterizados por el comportamiento b, tal que $i.p = b$.*

Ni los estados internos, ni la forma particular que tiene I para producir el comportamiento son relevantes para satisfacer el criterio de implementación a excepción, por supuesto, que tales aspectos formen parte de lo considerado relevante. La figura 2.1 muestra la implementación de una función de interpretación como una relación entre relaciones. En la versión "concreta", el comportamiento usualmente toma la forma de un proceso (el llamado *program-process*) o el estado final de un proceso tal, cuya forma particular está establecida por el sistema pero puede ser comprendido en términos de los atributos observables definidos como relevantes. Consideramos

[1] Con $A \to B$ denotamos el conjunto de funciones totales de A en B

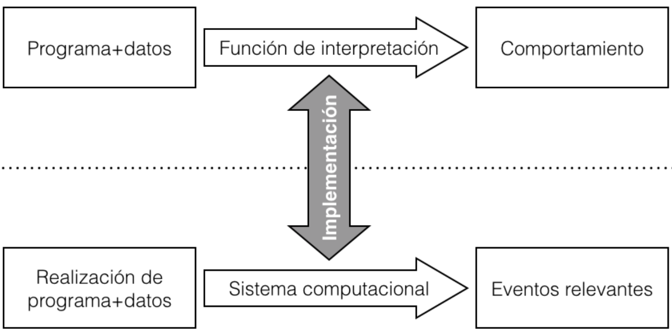


Figura 2.1: Relación de implementación

que el programa mismo (el *program-script*) es parte fundamental del sistema, y como tal, debe ser considerado también en su forma "concreta". Esta idea, que no es nueva como muestra la siguiente cita de [106], generalmente es olvidada en los trabajos que intentan dar cuenta de la noción de computación:

> It is important to remember that computer programs can be understood on the physical level as well as the symbolic level. The programming of early digital computers was commonly done by plugging in wires and throwing switches. Some analogue computers are still programmed in this way. The resulting programs are clearly as physical and as much a part of the computer system as any other part. Today digital machines usually store a program internally to speed up the execution of the program. A program in such a form is certainly physical and part of the computer system.

La presencia de una realización del programa como parte del sistema es quizá la mayor diferencia entre una *descripción computacional de un sistema* (cuando el programa no forma parte del sistema) de un *sistema computacional* (cuando el ahora presente programa actúa como una prescripción). La realización del programa puede darse de diferentes maneras (un texto representado con una configuración electrica de la memoria, la distribución de interruptores en una computadora *hard-programmed*, la tabla de transiciones de una máquina de Turing escrita en papel, etc.).

De esta manera, la principal característica de un sistema computacional es que es *programable*: admite la codificación una variedad de sus comportamientos. El grado de programabilidad del sistema está dado por la variedad de comportamientos que el lenguaje de programación subyacente es capaz de codificar, y es la característica distintiva del sistema computacional como tal. Si consideramos que un sistema es computacional cuando es programable, entonces "ser computacional" es una propiedad que puede ser establecida sólo en relación a un conjunto de comportamientos y una codificación correspondiente. En otras palabras, la propiedad de ser compu-

Figura 2.2: Intérprete

tacional no tiene sentido independientemente de un conjunto de comportamientos y un conjunto de programas.

La distinción entre la prescripción dada por una función de interpretación y su implementación refuerza la idea que "ser un sistema computacional" es principalmente una propiedad relacional abstracta de los sistemas, y por lo tanto, no existe ningún sistema inherentemente computacional. Sin embargo, mientras que es necesaria una contraparte *física* para llevar adelante el proceso de la computación, la propiedad de ser un sistema computacional está determinada por la prescipción que es implementada por el sistema.

Definición 3 (Intérprete y computación)**.** *Utilizamos el término* intérprete *para denotar un sistema computacional en el sentido de la definición 2, reforzando la idea que la prescripción es una parte constitutiva del sistema para que sea considerado computacional. Así comprendemos una* computación *como el resultado producido por un* intérprete *dada cierta prescripción como entrada, pero no simplemente el resultante proceso, o estado final, del sistema subyacente (fig. 2.2).*

La propiedad de "ser un intérprete" para conjuntos dados de programas y comportamientos, puede ser satisfecha por diferentes sistemas. La relación de implementación puede realizarse de diferentes maneras. Por ejemplo, una computadora de ADN o una computadora cúantica no se asemejan en su estructura a una computadora digital tradicional. De manera inversa, un mismo sistema puede implementar diferentes funciones de interpretación. Como mínimo, el subsistema de entrada debe ser modificable para permitir la recepción de un programa (y sus datos de entrada) y el subsistema de salida debe poseer los suficientes estados diferentes como para permitir reconocer la computación realizada. La prueba o justificación de la correspondencia entre la prescripción y la producción de un sistema puede requerir diferentes métodos, ya sean matemáticos o empíricos, dependiendo de la tecnología elegida para la implementación.

Nuesta noción de intérprete generaliza la definición usual en el área de compiladores y teoría de lenguajes, donde un intérprete es un programa de una computadora digital. Esta idea generalizada está implícita en los textos clásicos de computabilidad, por ejemplo, en [90]:

> The "computing agent" then interprets the algorithm; it can be a piece of hardware, or it can be software: an interpreter program written in a lower-level

> programming language. Operationally, an interpreter maintains a pointer to the current instruction within the algorithm's instruction set, together with a representation of that algorithm's current storage state. Larger algorithms correspond to larger interpreted programs, but the interpreter itself remains fixed, either as a machine or as a program."

Tanto en la teoría como en la práctica de las ciencias de la computación, el uso de intérpretes es ubícuo, aunque no siempre son presentados como tales. Los siguientes ejemplos ilustran esta afirmación.

Ejemplo 1. *El formalismo de las máquinas de Turing define elementos estructurales, como una cinta infinita, un cabezal de lectura/escritura, un conjunto finito de* estados mentales*, acciones elementales, como mover la cinta a la izquiera o derecha, leer o escribir un símbolo en la cinta, etc, y sus relaciones; y prescribe su desempeño de acuerdo a una tabla de transiciones. Cada tabla de transiciones (una máquina de Turing particular) es un programa, y su resultado son los valores escritos en la cinta.*

Una posible implementación es un "computer", una persona provista de papel y lápiz, actuando mecánicamente*, obedeciendo las reglas de la tabla de transiciones, utilizando percepción y memoria limitadas. Tanto el programa como el comportamiento resultante se realizan como símbolos en el papel.*

Los comportamientos producidos por las máquinas de Turing son el conjunto de valores resultantes de las funciones computables sobre números (según la tesis de Church-Turing).

Ejemplo 2. *El diseño de una computadora digital moderna establece el conjunto de operaciones elementales del microprocesador sobre valores simples y estructurados, ambos, las operaciones y los valores, codificados como palabras de* bits*, llamadas* código de máquina*, y la forma en que el sistema de memoria los almacena.*

Una computadora particular, construida de acuerdo al diseño anterior, ejecuta el codigo de máquina *de un programa y produce el valor resultante, ambos realizados como patrones eléctricos en los* chips *de memoria. La relación de implementación está garantizada por la teoría detrás de la tecnología electrónica.*

El conjunto de comportamientos puede limitarse a los valores computados (coincidiendo con los comportamientos de las máquinas de Turing), o puede definirse teniendo en cuenta detalles relevantes del proceso de computación, como paralelismo, predicación*, etc.*

Ejemplo 3. *Una computadora* hard-programmable *como la ENIAC, es un intérprete de los mismos comportamientos mencionados anteriormente. En este caso, el programa y los datos se realizan como ciertas configuraciones de* switches *y cables, mientras que el resultado está dado por la configuración del* hardware.

2.2. Jerarquías de intérpretes

En lugar de trazar una línea entre los sistemas que son computacionales y los que no, el concepto de intérprete permite hablar de niveles de computacionalismo, con sistemas que aceptan lenguajes Turing completos en la cima (por ejemplo, las computadoras digitales) y sistemas elementales que implementan intérpretes elementales en lo más bajo (como una piedra, o un balde de agua).

La necesidad de considerar una jerarquía de intérpretes no sólo aparece cuando necesitamos distinguir hasta qué punto un sistema físico computa, sino también en el dominio de formalismos abstractos, como las máquinas de Turing. Por ejemplo, mientras que está fuera de la discusión que el formalismo como un todo, o en otras palabras, el conjunto de todas las máquinas de Turing es un caso paradigmático de un sistema computacional, es menos claro cuándo una máquina de Turing particular computa, o mejor, cuán potente o programable es. Consideremos una máquina de Turing de sumar, que toma la representación unaria de dos números y los concatena. Podríamos afirmar que la máquina computa, o más aún que es un sistema computacional. Algunos autores pueden estar de acuerdo con la primera afirmación pero no con la segunda.

En esta sección, introducimos dos jerarquías diferentes para comparar intérpretes, de acuerdo a dos nociones importantes en el campo de la teoría de la computabilidad: simulación e interpretación. La primera permite relacionar intérpretes de acuerdo a su *poder computacional*, *i.e.* el conjunto de comportamientos que pueden producir. La segunda define una relación más fuerte, de acuerdo a su *programabilidad*, que implica, bajo ciertos requerimientos, a la anterior, y caracteriza además la relación comunmente llamada "implementación".

La primer jerarquía de intérpretes se define en términos de un simple pre-orden entre las funciones de interpretación, teniendo en cuenta el conjunto de comportamientos considerados por cada una de ellas.

Definición 4 (Simulación de comportamientos)**.** *Dadas las funciones de intepretación $i \in P \rightarrow B$ e $i' \in P' \rightarrow B'$, y los intérpretes I e I' respectivamente, decimos que I* se comporta al menos como *I', denotado por $I' \subseteq I$, si $B' \subseteq B$. En el caso que también valga $I \subseteq I'$ decimos que los intérpretes son* comportamentalmente equivalentes.

Usualmente los sistemas computacionales se comparan en términos de los lenguajes de programación que aceptan. En lugar de organizar una jerarquía de intérpretes en términos de los comportamientos que producen, es posible relacionarlos haciendo foco en los lenguajes de programación que soportan.

En [69] se presenta una caracterización de los lenguajes de programación en función de la completitud de Turing, intentando dejar fuera ciertos candidatos anti-intuitivos. Sin embargo esta restricción parece demasiado fuerte, dado que algunos

lenguajes que no son Turing completos pueden ser considerados lenguajes de programación genuinos, por ejemplo: el lenguaje de demostradores de teoremas como Coq e Isabelle, que sólo permiten funciones que terminan; algunos lenguajes derivados de la teoría de tipos; o el lenguaje propuesto en [144] que consta de funciones primitivas recursivas de alto orden. Incluso un lenguaje mucho más simple, como el de una calculadora electrónica, es un lenguaje de programación interesante, aunque poco potente.

Para definir una jerarquía basada en los lenguajes de programación, introducimos la noción de *función de compilación* como una función que traduce un programa p en otro p' *equivalente*, posiblemente escrito en un lenguaje de programación diferente.

Definición 5 (Función de compilación y simulación de lenguajes)**.** *Dadas dos funciones de interpretación* $i \in P \rightarrow B$ *e* $i' \in P' \rightarrow B'$*, decimos que* $c \in P' \rightarrow P$ *es una* función de compilación *entre* P' *y* P *si para cada* $p' \in P'$

$$i.(c.p') = i'.p'$$

En este caso decimos que I simula *a* I'*, denotado por* $I' \sqsubseteq_c I$*. En caso que también valga* $I \sqsubseteq_{c'} I'$ *(para* $c' \in P \rightarrow P'$*) decimos que los intérpretes son* lenguaje-equivalentes.

Como una consecuencia directa de esta definición, el orden $\sqsubseteq$ implica lógicamente el orden $\subseteq$ previamente definido. Lo recíproco también es cierto.

Lema 1. *Dados dos intérpretes* I *e* I'*:*

1. *Si* $I' \subseteq I$*, entonces existe* $c \in P' \rightarrow P$ *tal que* $I' \sqsubseteq_c I$*.*
2. *Si* $I \sqsubseteq_c I'$ *(para alguna* $c \in P \rightarrow P'$*), entonces* $I \subseteq I'$*.*

Ejemplo 4. *Sea* P *un lenguaje de programación que consiste de dos instrucciones:* $sum.x$ *y* $mult.x$ *que producen respectivamente la suma y multiplicación de* x *con el valor almacenado en un acumulador, y almacenan el resultado en el mismo. Sea* I *un intérprete para este lenguaje que acepta secuencias de tales instrucciones como programas. Así* I *se comporta como un evaluador de expresiones algebráicas conformadas por* $+$ *y* $\times$*.*

Sea P' *otro lenguaje de programación que consiste de tres operaciones:* $sum.x$ *definida como la anterior,* A *que accede al valor almacenado en el acumulador, y* $repeat.x.S$ *que repite la secuencia de instrucciones* S *una cantidad* x *de veces. Sea* I' *un intérprete para* P'*.*

Es fácil de demostrar que existe una función de compilación que traduce cada instrucción $mult.x$ *en un programa* $repeat.(x-1).(sum.A)$*. Por lo tanto* I' simula *a* I*. Lo opuesto no es verdadero, ya que* I' *se comporta como un evaluador de expresiones algebráicas conformadas por* $+$, $\times$ *e hiper-exponenciación.*

Mas allá del hecho que los órdenes previamente definidos son equivalentes, el segundo tiene su propia importancia ya que los lenguajes de programación y la traducción entre ellos juegan un rol fundamental en las prácticas de las ciencias de la computación. Ya que son dispositivos diseñados, los intérpretes se construyen usualmente de acuerdo a un lenguaje de programación predefinido en lugar del conjunto de comportamientos subyacentes, que permanece implícito. Más aún, suele suceder que el conjunto de comportamientos no es (o no puede ser) descripto de forma precisa. Como un caso particular, el conjunto de las *funciones efectivamente computables* no puede ser definido excepto por la relación entre diferentes formalismos, conocida como la tesis de Church-Turing.

Lema 2 (Tesis de Church-Turing). *Las funciones recursivas, las máquinas de Turing, y un lenguaje imperativo estándar, etc. vistos como los respectivos intérpretes R, T, I de las "funciones efectivamente computables", son* lenguaje-equivalentes, *y para cualquier otro intérprete I' tal que $R \sqsubseteq I'$ necesariamente vale que $I' \sqsubseteq R$.*

Consideremos dos intérpretes I e I', que implementan las funciones $i \in P \to (D \to B)$ e $i' \in P' \to (D' \to B')$ respectivamente. La jerarquía definida por los órdenes $\sqsubseteq$ y $\subseteq$ especifica que un intérprete, por caso I, puede simular a otro, I', y consecuentemente comportarse como tal, pero no estipula cómo se lleva adelante esa simulación. No es un requerimiento que la función de compilación se pueda ejecutar mecánicamente, ni que sea una función computable. En la literatura se propone el concepto de *compilador* como un procedimiento mecánico para traducir efectivamente un programa escrito en un lenguaje de programación en otro. Si requerimos que la traducción sea ejecutada por I, esto es, existe un compilador $comp \in P$, la condición de la definición 5 se transforma en

$$i.(i.comp.p') = i'.p'$$

Aquí $i.comp.p'$ es la "ejecución" del compilador con el dato de entrada p' que resulta en otro programa, que es la entrada de i. Notar que debe suceder que $P \subseteq D$.

Dado el rol central que tiene la noción de compilador en la teoría de la computación, se puede argumentar que puede reemplazar a la noción de intérprete para determinar cuándo un formalismo sintáctico es un lenguaje de programación, y consecuentemente, cuándo un sistema es computacional. Sin embargo el concepto de intérprete es más primitivo, porque para que un compilador (o más general, cualquier programa) sea "ejecutado", siempre es necesario un intérprete (como se observa a partir de la ecuación anterior). Mientras que el concepto de compilador tiene cierta capacidad explicativa, que permite comprender un lenguaje de programación en términos de otro lenguaje *mas primitivo*, es un intérprete quien finalmente caracteriza los comportamientos que tal lenguaje es capaz de codificar. La ecuación anterior nos

permite decir informalmente, que "la composición de un intérprete con (la ejecución de) un compilador resulta en un intérprete".

Requerir que una función de compilación pueda ser codificada como un programa para un intérprete dado impone condiciones más fuertes no sólo en los comportamientos que tal intérprete produce, sino tambien en su *programabilidad*. Siguiendo esta dirección, es posible definir un orden lógicamente más fuerte que $\subseteq$ (y que $\sqsubseteq$).

Definición 6 (Interpretación)**.** *Dadas dos funciones de intepretación $i \in P \rightarrow B$ e $i' \in P' \rightarrow B'$, y respectivos intérpretes I e I', e $int \in P$ un programa particular que toma elementos de P' como entrada, decimos que int es un* programa-intérprete *para P' si*

$$i.int = i'$$

En este caso decimos que I interpreta *a I', denotado como $I' \preceq I$. Si además vale que $I \preceq I'$ decimos que I e I' son* inter-interpretables. *Cuando $I \preceq I$ decimos que I es* auto-interpretable, *que es una propiedad bien conocida de los sistemas Turing completos.*

La diferencia entre las jerarquías de simulación e interpretación es sutil. Nuevamente consideremos dos intérpretes I e I', que implementan las funciones $i \in P \rightarrow (D \rightarrow B)$ e $i' \in P' \rightarrow (D' \rightarrow B')$ respectivamente. Mientras que el orden $I' \sqsubseteq I$ determina una relación de *ser parte* entre los comportamientos subyacentes (*i.e.* $B' \subseteq B$), el orden $I' \preceq I$ caracteriza una relación de *ser un elemento* ($i' \in D \rightarrow B$). Como consecuencia, no sólo I puede finalmente comportarse como I', sino que el lenguaje de programación de I es capaz de codificar en un único programa el conjunto completo de comportamientos del intérprete I'.

Para clarificar la distinción, supongamos que $I' \sqsubseteq I$ y $I' \preceq I$. Así, combinando las condiciones de las definiciones 5 y 6 obtenemos

$$i.(c.p') = i.int.p'$$

En el lado izquierdo, $c.p'$ es un programa diferente para cada p'. En el lado derecho, el programa es siempre el mismo, int, mientras que p' es sólo el dato de entrada de int.

La ecuación anterior permite entrever la relación lógica entre $\sqsubseteq$ y $\preceq$. Si $I' \preceq I$ vale, es fácil de imaginar cómo definir una función de compilación de P' a P: asignar a cada $p' \in P'$ un programa como int pero con p' ocupando el lugar del parámetro de entrada.

Lema 3. *Sean $i \in P \rightarrow B$ e $i' \in P' \rightarrow B'$ dos funciones de intepretación, I e I' los intérpretes que implementan i e i' respectivamente. Si $I' \preceq I$, entonces $I' \subseteq I$.*

Interpretación e implementación

Reconsideremos la definición de la relación de interpretación (def. 6). El nombre elegido para referirnos al programa característico *int* puede resultar confuso en

principio, pero no es arbitrario. Previamente introducimos el concepto de intérprete como una generalización del concepto, comúnmente comprendido como un programa que "ejecuta" otros programas. El *programa-intérprete int* es tal programa.

Además, los *programas-intérpretes* juegan un papel fundamental en la relación de implementación. Un *programa-intérprete* es un intérprete de acuerdo a la relación de implementación que presentamos, en el siguiente sentido. Consideremos dos funciones de interpretación $i \in P \to B$ e $i' \in P' \to B'$, e I un sistema computacional respecto a i. Consideremos además que existe un *programa-intérprete* $int \in P$, tal que

$$i.int = i'$$

Es fácil de ver que el intérprete I instanciado con int (*i.e.* fijando int como la entrada de I) satisface el criterio para ser considerado una *implementación* de la función de interpretación i', y por lo tanto *es un intérprete*. Esto nos permite decir informalmente que "la composición de un intérprete con un *programa-intérprete* es un intérprete". Además, para cualquier otro sistema I' que implementa i', se satisface que $I' \leq I$.

Cabe remarcar que no es el *mero* sistema (físico) I quien implementa a i', sino que la función de interpretación i interviene en la relación. Así es que I, considerado como sistema computacional respecto a i y provisto con int, es también computacional respecto a i'. Esto muestra que la relación de implementación es lo suficientemente general como para considerar no sólo sistemas "concretos".

Ejemplo 5. *Según el ejemplo 1 (pág. 21), el "computer" es una implementación del formalismo de las máquinas de Turing (T). Ahora, la Máquina de Turing Universal (utm), "ejecutada" por el* computor, *es un intérprete que toma una codificación de la tabla de transición de una máquina arbitraria junto con su entrada, y se comporta como ella. En otras palabras*

$$T.utm = T$$

Como consecuencia, el formalismo de las máquinas de Turing es auto-interpretable, y el programa particular utm *codifica el poder computacional completo del formalismo.*

Ejemplo 6. *El ejemplo más común de un intérprete es un* programa-intérprete *ejecutándose en una computadora digital, que "ejecuta" un* script *escrito en un lenguaje de programación corriente. El* script, *sus datos y el valor resultante son codificados en alguna estructura de datos del* programa-intérprete. *El conjunto de comportamientos queda definido por la semántica del lenguaje.*

Por ejemplo, los programas `perl` *y* `python` *son* programas-intérpretes *de los lenguajes homónimos, y* `ghci` *para el caso de Haskell.*

Los *programas-intérpretes* se utilizan extensivamente en la teoría de la computación y en la ingeniería de *software* para dar cuenta de la complejidad de los sistemas informáticos. Y las funciones de interpretación juegan un papel importante,

porque son utilizadas profusamente para mediar entre las relaciones de implementación. Es muy común "componer" diversos *programas-intérpretes* en una pila, cada uno de los cuales presenta una "interfaz abstracta" (dada por la respectiva función de interpretación), ocultando las particularidades de su funcionamiento (que podríamos denominar sus "detalles de implementación"), que se basan en la funcionalidad provista por el *programa-intérprete* de un nivel más abajo. En el nivel inferior siempre encontramos un intérprete, encargado de *producir* el comportamiento.

Ejemplo 7. *Una máquina virtual Java es un* programa-intérprete *para el lenguaje de programación Java. A su vez está diseñada para correr sobre un sistema operativo dado, que provee una abstracción de la computadora digital sobre la que el mismo se ejecuta. Esta abstracción es caracterizable por una función de interpretación, y el sistema operativo es luego, en tanto programa, un* programa-intérprete. *Finalmente el sistema operativo se define en términos de la abstracción que provee el lenguaje máquina propio de la computadora, sobre la electrónica que lo implementa.*

La composición de *programas-intérpretes* es muy común incluso en la arquitectura y la organización interna de los sistemas informáticos. La relación de implementación que definimos generaliza la relación más específica, conocida como "implementación" en la comunidad de ciencias de la computación, entre "especificaciones" y "programas" escritos en un lenguaje de programación. Debemos tener en cuenta que cualquier programa puede ser considerado como un *programa-intérprete* elemental de una función de interpretación que relaciona los datos de entrada con la salida de tal programa. Es decir, su "especificación" puede ser vista como una función de interpretación. Y en la tarea de programar una "implementación" de una "especificación" dada, siempre se asume la existencia de un *programa-intérprete* que garantiza la semántica pretendida del lenguaje, y que es el encargado de ejecutar tal programa.

Ejemplo 8. *Consideremos la "especificación" de un tipo abstracto de datos T, y un módulo M programado en el lenguaje Python que la "implementa". El* programa-intérprete `python` *ejecutándose con el módulo M cargado, es ahora un intérprete del lenguaje de programación Python extendido con el tipo abstracto T.*

2.3. La verificación formal de programas

La verificación formal es parte de una tradición que considera los programas como objetos formales, cuyos comportamientos pueden *especificarse* a través de lenguajes lógicos y verificarse por métodos matemáticos, lo que da lugar a una demostración de su corrección. Esta metodología ha sido propuesta como forma privilegiada de garantizar la corrección de programas casi desde los inicios mismos de la práctica

extendida de la programación [102] y defendida por una escuela que tiene como referentes a Hoare [80] y Dijkstra [61] entre otros. A lo largo del tiempo se ha sostenido un debate acerca de su practicidad, su plausibilidad e incluso su validez metodológica con diferentes argumentos. Curiosamente, uno de los primeros, debido a De Millo *et al.* [58], continúa teniendo cierto asidero en el presente.

En el citado trabajo se distigue entre "demostración" y "prueba": la primera consiste en una serie de fórmulas lógicas que se deducen secuencialmente, cuya corroboración es extremadamente tediosa, mientras que la segunda es un esquema incompleto de la primera, que resalta los puntos de mayor interés. Ambas comparten el mismo objetivo: incrementar la confianza en la corrección de un resultado. Al mismo tiempo distinguen entre las supuestas prácticas de las comunidades matemática y de las ciencias de la computación. Los miembros de la primera comparten y discuten las "pruebas" de sus teoremas, lo que les permite ganar confianza en sus resultados y construir un conocimiento general. Los miembros de la segunda se dedican a realizar "demostraciones" que difícilmente resultan interesantes para otros miembros y no aportan a una construcción social de un conocimiento general sobre los programas. Esto se debe a la complejidad de las especificaciones, la "ausencia de continuidad" (una mínima modificación hace de un programa otro radicalmente diferente y su corrección inconmensurable), y la "inevitabilidad de los cambios" (un programa y su especificación cambian constantemente para adecuarse el uno a la otra). De esta manera el objetivo mismo de la verificación es estéril. Como consecuencia, el *testing* de un programa respecto a su especificación es la práctica favorita para incrementar la confianza en su corrección, de la misma manera que las pruebas a las que se somente un artefacto ingenieril aportan confianza sobre su confiabilidad (*reliability*).

Como bien marca Fetzer en su crítica a De Millo *et al.* [72], los argumentos sobre las diferencias en las prácticas sociales de las comunidades matemática y de las ciencias de la computación, y sobre la complejidad intrínseca de los programas (y por lo tanto, la complejidad misma del proceso de verificación) pueden debilitarse con el tiempo:

> ...if program verifiers were to commence collaborating in their endeavors, the principal rationale underlying [De Millo *et al.*] position would tend to disappear.

De hecho, la introducción de mecanismos de modularización y abstracción en los lenguajes de programación, el desarrollo de nuevas y mejores metodologías de razonamiento formal composicional, la aparición de herramientas (semi) automáticas para su implementación, entre otros avances, cambian dramáticamente el contexto de esa discusión. Como hacemos explícito en la introducción de esta tesis, aceptamos que aún queda mucho camino por recorrer para lograr que la verificación formal sea una herramienta habitual en la práctica de la programación. Sin embargo sostenemos que es una metodología confiable y con potencial aplicabilidad general para el análi-

sis y la demostración de la corrección de los programas. Siguiendo a Fetzer, lo que hace que una demostración finalmente sea tal, no es la aceptación de la misma por parte de la comunidad, sino su *validez* (en el sentido formal).

Aunque Fetzer no ahorra críticas, comparte la conclusión principal con De Millo *et al.*, esto es, la imposibilidad de la verificación formal, y para sostenerla propone una distinción ontológica entre las nociones de "algoritmo" y "programa", a las que adscribe diferentes posibilidades de verificación. Un "algoritmo" es una entidad lógico-matemática que se inscribe dentro del dominio de las metodologías deductivas, y como tal, requiere demostraciones. Como ocurre con los teoremas de la matemática, la corrección de un "algoritmo" puede demostrarse a partir de axiomas primitivos, sin requerir ninguna premisa adicional, y por lo tanto es *absolutamente verificable*. Pero los "algoritmos" se diferencian de los teoremas en que tienen significado semántico: las líneas que componen un "algoritmo" refieren a operaciones, en principio, de una máquina *abstracta* ideal, que pueden ser ejecutadas finalmente por una máquina *concreta*, cuya operación no puede garantizarse sino aproximarse por un razonamiento inductivo. Así, la noción de "programa" aparece como la materialización de un "algoritmo" en relación a una máquina concreta. Es una estructura causal que cae dentro de un dominio de pesquisa empírica y por lo tanto requiere garantías inductivas. De esta manera, los "programas" son sujetos, a lo sumo, a una *verificación relativa*: su corrección podría ser demostrada por métodos deductivos a partir de ciertas premisas (que garanticen ciertas propiedades sobre la máquina *concreta*) que deberían poder establecerse inductivamente. Pero esta posibilidad se ve trunca frente a la complejidad causal en la interacción de los programas con otros factores causales (*hardware*, *firmware*, dipositivos de entrada/salida, etc.).

> ...the execution of a program qualifies as causally complex, insofar as even a correct program can produce "widly erratic behaviour ...if only a sigle bit is changed". The reason the result of executing a program cannot provide deductive support for the evaluation of a program ...is that the behavior displayed by a casual system is an effect of the complete set of relevant factors whose presence or absence made a difference to its production.

Así concluye sugiriendo el *testing* de un programa como la vía para obtener evidencia de su confiabilidad, aunque recuerda que "el desempeño operacional ...nunca debería darse por sentado y no puede ser garantizado".

La aceptación del argumento de Fetzer no sólo invalida la noción de verificación formal, sino también la del propio *testing* de programas como se lo entiende en la actualidad: ¿frente a un comportamiento inesperado en una ejecución de un programa, cómo distinguir si el error se encuentra en el programa o en la computadora que lo ejecuta? El "error categorial" de considerar a un programa como un proceso causal

en esos términos es denunciado por Blanco y García [18] y analizado en un marco coherente al propuesto en este trabajo.

En [68] Eden investiga tres *paradigmas* [93] respecto a aspectos ontológicos y epistemológicos sobre los programas. Uno de sus objetivos es responder a la pregunta sobre si el conocimiento acerca del comportamiento de los programas debe proceder por via deductiva o empírica. Los paradigmas se pueden resumir de la siguiente manera:

- el *racionalista*, común entre los científicos teóricos de la computación, que pone los programas a la par con los objetos matemáticos, donde el conocimiento sobre su corrección puede establecerse *a priori* por métodos deductivos;
- el *tecnocráctico*, promulgado por los ingenieros del *software*, donde la confianza en la corrección de los programas se establece *a posteriori* y empíricamente a traves del *testing*;
- el *científico*, que busca un conocimiento *a priori* y *a posteriori* sobre los programas combinando deducción formal y experimentación científica. Un experimento se diferencia de un *test*, en tanto el segundo busca establecer hasta qué punto un programa satisface su especificación, mientras que el primero está diseñado para corroborar o refutar una hipótesis particular (sobre el ámbito en el cual la especificación del programa cobra sentido).

Pero Eden lleva adelante un análisis, según el cual, se distinguen dos nociones diferentes de lo que es un programa:

> We seek to distinguish between two fundamentally distinct senses of the term program in conventional usage: The first is that of a static script, namely a well-formed sequence of symbols in a programming language, to which we shall refer as a program-script, The second sense is that of a process of computation generated by executing a particular program-script, to which we shall refer as a programprocess.

Esta diferenciación, análoga a la Fetzer, caracteriza un "program-script" como una entidad abstracta, y un "program-process" como "temporal, nonphysical, causal, metabolic, contingent upon a physical manifestation, and non-linear". Pero el análisis está sesgado por consideraciones propias de ámbitos específicos como la inteligencia artificial, las simulaciones computacionales, etc., donde la idea de corrección de un programa se disuelve y estos son más bien considerados "modelos" que pueden ir perfeccionandose a vista de los resultados de ciertos experimentos (y su adecuación a los resultados esperados si existieran). No es una sorpresa que partir de esta caracterización ontológica y la consecuente imprevisibilidad de los programas, concluya

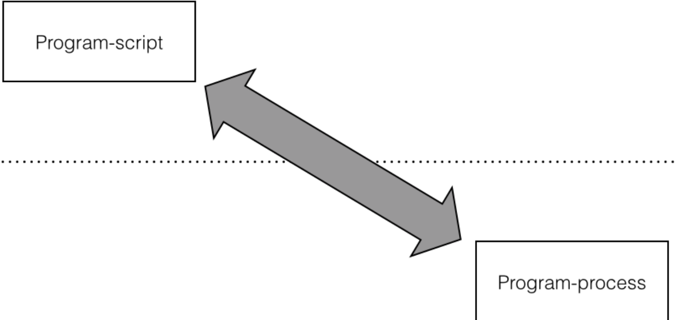


Figura 2.3: Naturaleza dual de los programas.

que el paradigma científico es la forma privilegiada de establecer propiedades sobre los mismos.

Cuando los programas se presentan a la manera de Fetzer y Eden, se genera un *gap* ontológico entre los aspectos abstactos y concretos de la computación. La relación entre tales aspectos permanece oscura y sin explicación, debido, principalmente, a que el propio sistema computacional como tal, y su rol en la generación de una computación a partir de lo prescripto por un programa, están ausente en tales presentaciones. Creemos que la "naturaleza dual" de los programas, y más generalmente de la computación, es una noción clave para entender cuestiones centrales dentro de las ciencias de la computación. En este trabajo rescatamos los aspectos abstractos y concretos tanto de los programas como de los comportamientos que estos producen, las computaciones. Pero sostenemos que esa diferenciación debe comprenderse como una caracterización de los aspectos prescriptivos y descriptivos de un sistema computacional, y no en el plano ontológico.

En la figura 2.3 se muestra un esquema que pone en relación (por su ubicación espacial) la distinción entre "program-script" y "program-process", y los conceptos elementales que introducimos para la caracterización de los sistemas computacionales. De la comparación con las figuras 2.1 y 2.2 (pág. 19) se desprende que la noción de intérprete puede considerarse como el enlace necesario entre algunos aspectos de los "program-scripts" y "program-processes". Sorprendentemente, la distancia entre estas nociones no ha sido estudiada desde esta perspectiva, dejando la conexión entre ellas implícita y desdibujada por las dificultades inherentes a la relación entre los ámbitos fisicos y abstractos.

La introducción del concepto de intérprete permite reformular varias cuestiones concernientes a la naturaleza de los programas y el conocimiento sobre ellos, y proponer preguntas en un contexto más amplio. El *conundrum* que surge a partir de las propiedades contradictorias de los dos significados de un programa, puede comprenderse en términos de la diferencia entre un programa, como lo caracterizamos en la

definición 1 (pág. 18) y la realización del comportamiento que tal programa codifica. Y el enlace entre estos dos aspectos puede ser comprendido en términos del dispositivo particular que produce el comportamiento: el intérprete. Un “program-script” es un objeto abstracto que puede ser considerado como el argumento de un intérprete, mientras que el “program-process” se identifica con el proceso físico (del sistema que forma parte del intérprete) que evidencia el comportamiento prescripto.

Esta propuesta relativiza el problema de la verificación formal. Los “program-process” pueden ser caracterizados en términos de propiedades de alto nivel descriptas por una función de interpretación. Y estas propiedades son invariantes respecto a las potenciales diferentes implementaciones y a las “ejecuciones” particulares de los sistemas concretos subyacentes. Es posible demostrar propiedades sobre los “program-scripts” utilizando métodos deductivos sobre la descripción abstracta de un intérprete. Así, acordamos con Fetzer en que la verificación de un programa es *relativa*, pero nos diferenciamos en que la justificación empírica debe efectuarse para garantizar que un sistema particular efectivamente implementa una función de intepretación (es un inteprete), y no sobre ejecuciones particulares de una concretización singular de un programa. Cómo un sistema físico realiza un intérprete tiene que ver con la teoría física utilizada para describirlo, y su validación empírica. Un intérprete como mediador entre el *script* de un progama y el proceso que prescribe se hace cargo de la complejidad causal del sistema físico que lo realiza. Esto está de acuerdo con las prácticas en todas las ramas de las ingenierías y las descripciones abstractas que generan sobre los sistemas que construyen.

Finalmente, podemos decir que de acuerdo a nuestra propuesta, es debatible si los “program-processes” son como los caracteriza Eden. Las propiedades que les atribuye no son universales; no aplican a cualquier intérprete sino a algunas realizaciones particulares, principalmente a aquellas basadas en una arquitectura Von Neumann (e incluso es debatible si estas propiedades valen para tales realizaciones). Pero incluso para programas *auto-modificables*, *no-lineales* y extremadamente *complejos*, que Eden considera como “modelos” que deben tratarse empíricamente, es posible recuperar una noción de corrección respecto a propiedades mas abstractas de los escenarios que tales programas modelan, y de esta forma, ofrecer ciertas garantías para la interpretación de los experimentos que sobre ellos se lleven a cabo.

2.4. El pancomputacionalismo

El pancomputacionalismo es la tesis que afirma que todo sistema físico abierto realiza cualquier computación. Como consecuencia, la noción de computación se volvería trivial a través del concepto de realización en un sistema físico. El primer argumento riguroso a favor del pancomputacionalismo ilimitado es presentado por Putnam [130], quien sostiene que todo sistema abierto ordinario implementa cual-

quier sistema de transición finito abstracto (sin entradas ni salidas). Primero asume que los campos electromagnéticos y gravitacionales son contínuos y que un sistema físico está en diferentes estados maximales en diferentes momentos. Luego considera un sistema de transición finito arbitrario cuya función de transición da lugar a la secuencia de estados $ABABABA$. Toma un sistema físico arbitrario S en un intervalo de tiempo arbitrario, desde las 12:00 hasta las 12:07, y sostiene que S implementa la secuencia $ABABABA$. Dado que, tanto el sistema de transición como el sistema físico son arbitrarios, el argumento generaliza a cualquier sistema de transición y cualquier conjunto de estados físicos. El núcleo del argumento es el siguiente:

> Let the beginnings of the intervals during which S is to be in one of its stages A or B be $t_1, t_2, \dots, t_n$ (in the example given, $n = 7$, and the times in question are $t_1 = 12{:}00, t_2 = 12{:}01, t_3 = 12{:}02, t_4 = 12{:}03, t_5 = 12{:}04, t_6 = 12{:}05, t_7 = 12{:}06$). The end of the real-time interval during which we wish S to "obey" this table we call t_{n+1} ($t_{n+1} = t_8 = 12{:}07$, in our example). For each of the states t_i to t_{i+1}, $i = 1, 2, \dots, n$, define a (nonmaximal) interval state s_i which is the "region" in phase space consisting of all the maximal states $St(S, t)$ with $t_i \leqslant t < t_{i+1}$ (*i.e.*, S is in s_i just in case S is in one of the maximal states in this "region"). Note that the system S is in s_1 from t_1 to t_2, in s_2 from t_2 to t_3, ..., in s_n from t_n to t_{n+1} (left endpoint included in all cases but not the right - this is a convention to ensure the "machine" is in exactly one of the s_i at a given time.) The disjointness of the states s_i is guaranteed by the Principle of Noncyclical Behavior.
>
> Define $A = s_1$ or s_3 or s_5 or s_7; $B = s_2$ or s_4 or s_6. Then, as is easily checked, S is in state A rom t_1 to t_2, from t_3 to t_4, and from t_5 to t_6, and from t_7 to t_8, and in state B at all other times between t_1 and t_8. So S has the table we specified, with the states A, B we just defined as the realizations of the states A, B described by the table.

En resumen, Putnam toma un sistema físico arbitrario con dinámica contínua, parte su dinámica en intervalos de tiempos discretos, y luego secuencia las partes de manera que se correspondan con una secuencia arbitraria de estados computacionales abstractos. Esto es lo que se conoce como el *simple mapping account*. Así concluye que todo sistema físico implemente cualquier sistema de transición finito.

Tiempo despúes, Searle [139] propone un argumento similar. De acuerdo a este trabajo, si tenemos en cuenta la caracterización de Turing se dan las siguientes consecuencias:

> 1. For any object there is some description of that object such that under that description the object is a digital computer. 2. For any program and for any sifficiently complex object, there is some description of the object under which it is implementing the program.

Existen diferentes intentos de dar cuenta del desafío del pancomputacionalismo. Para algunos investigadores, el principal problema reside en cómo encontrar características o propiedades específicas de la computación para imposibilitar el pancomputacionalismo. Para otros autores, la pregunta es cómo definir la realización de una descripción de un sistema computacional en un sistema físico. En [56] Copeland sugiere el concepto de "arquitectura", junto con la idea de modelos "honestos", como una respuesta. Piccinini [125] presenta la noción de mecanismo como una propiedad específica de la computación. Scheutz trabaja sobre la idea de "realización de una función" y sus condiciones físicas como el concepto clave contra el pancomputacionalismo [138]. Chalmers [45] intenta evitar la trivialización del concepto de computación sosteniendo que el argumento de Putnam no considera una teoría adecuada de implementación.

Copeland analiza la noción de computación, intentando establecer criterios para explicar cómo identificar computaciones en sistemas particulares, y cómo discernir si un sistema está computando o no. Distingue por un lado la "arquitectura" de un sistema, que caracteriza de forma axiomática la programabilidad del mismo en términos de operaciones elementales. Cada axioma es de la forma: si se da la configuración abstracta x, entonces se ejecuta la operación o entre los valores dados por la configuración. Por otro, introduce la noción de "algoritmo", que utiliza tales operaciones para computar, a partir de los valores de entrada de una función dada f, los valores de salida de f. Pero luego, para utilizar una relación respecto a modelos cómo la usual en lógica, los une en una teoría lógica *SPEC* en la cual es imposible distinguirlos. Para Copeland, computar es ejecutar un "algoritmo". De forma más precisa, que una entidad e compute una función f, equivale a que exista una correspondencia entre los estados de e y las configuraciones de *SPEC* de manera que todos los axiomas de la "arquitectura" se validen. Los estados de e se distinguen a partir de un esquema de *labeling* L determinado. Dicho de otra forma, e computa f si (e, L) es un modelo de *SPEC*.

Lamentablemente, y como principal consecuencia de unir un "algoritmo" y la "arquitectura", los argumentos de Putnam y Searle aplican aquí también: es posible especificar un esquema de *labeling* para cada computación. Para esquivar esta dificultad, Copeland requiere que el esquema de *labeling* no sea *ex post facto*, que puede ser un buen requerimiento metodológico, pero es difícil de establecer sobre un sistema dado. Para evitar el pancomputacionalismo, Copeland requiere que los modelos de *SPEC* sean "honestos", donde esto significa que las relaciones lógicas deben interpretarse de la forma que se pretende, satisfaciendo ciertos contrafácticos de forma que las operaciones elementales de la "arquitectura" no puedan interpretarse de forma no estándar. Pero esto último arrastra el mismo problema original: ¿cómo podemos caracterizar que cierto modelo representa el pretendido significado de computación?

Un enfoque interesante, con el cual nuestra propuesta establece un diálogo fértil, es la explicación mecanicista de Piccinini. Su tesis principal es que una computadora es un tipo especial de mecanismo (*i.e.* un sistema sujeto a una descripción mecanicista) cuya función es generar *strings* de salida, a partir de ciertos *strings* de entrada y (posiblemente) los estados internos, de acuerdo a una regla general que describe el comportamiento de la computadora en cuestión. La confiabilidad del mecanismo computacional es capturada por el hecho que es posible reconocer una correspondencia entre ciertas configuraciones temporales y espaciales de entidades físicas o estados del sistema (dígitos) y la operación de concatenación de los símbolos formales (que conforman los *string* arriba mencionados), que da lugar a una relación de realización entre la descripción abstracta del sistema computacional y el mecanismo. Basado en esta caracterización, se propone una taxonomía de diferentes tipos de computadoras [125, 126] de acuerdo a sus componentes y las interacciones entre ellos.

Aunque el enfoque de Piccinini establece una taxonomía bien organizada de diferentes realizaciones de sistemas computacionales, y, más aún, provee una explicación completa de cuándo un sistema es una computadora sin recurrir a ninguna caracterización semántica, no provee un criterio de demarcación claro. Por un lado, el enfoque es demasiado reducido, dado que la caracterización estructural funciona bien con la tecnología actual de implementación de computadoras, pero no necesariamente con tecnologías anteriores (como la máquina analítica de Babagge), no convencionales (como las computadoras de ADN, computadoras cuánticas), o futuras. El solo hecho que artefactos tales puedan ser considerados computadoras requiere que cualquier criterio de demarcación no sea dependiente de una clase particular de realizaciones. Por otro lado, el enfoque es poco preciso, dado que clasifica como equivalentes sistemas computacionales que son claramente diferentes desde el punto de vista computacional (por ejemplo, las maquinas tabuladas de Hollerith y una computadora digital). Este último problema parece salvable sólo a partir de una caracterización gradual. Cualquier clasificación binaria tiene el mismo problema.

Si bien el objetivo principal de nuestro concepto de intérprete es caracterizar la noción de computación, tiene ciertas implicancias en el problema del pancomputacionalismo. En particular, proponemos que un sistema es considerado computacional *en cierto grado* si puede ser programado. Cuanto más grande el conjunto de comportamientos programables, más computacional es el sistema. Una "computadora" que pueda ser programada sólo para un conjunto muy limitado de tareas difícilmente sea considerada un sistema computacional "real". Tal es el caso, por ejemplo, de un termostato o un regador automático. La máquina de sumar de Pascal, puede ser un sistema más interesante, pero aún está muy lejos del poder computacional de una computadora digital. Un intérprete para el cálculo inductivo de construcciones, aunque incompleto (porque sólo permite funciones totales) probablemente sea considerado como un sistema de cómputo útil. Así, creemos que el desafío ontológico

del pancomputacionalismo se disuelve en la pregunta sobre cómo clasificar sistemas computacionales de acuerdo a su programabilidad.

Podemos aceptar las afirmaciones de Putnam y Searle de que, en algún sentido, cualquier sistema puede ser visto como ejecutando una computación particular, sin acordar con, desde nuestra perspectiva, la obviamente falsa afirmación de que cualquier sistema computa en ese mismo sentido (o dicho de otro modo, que cualquier sistema es un sistema computacional en toda su generalidad). La clave de la trivialización de la noción de computación, consecuencia del pancomputacionalismo de Putnam y Searle, es una libertad *ilimitada* en la forma de observación. El *sentido* en que una piedra o una pared computa está dado por el observador. Los estados que describen una computación abstracta pueden ponerse uno a uno con un sistema físico suficientemente complejo. Pero el establecimiento de tal *mapping es* una tarea computacional que debe ser llevada a cabo, en este caso, por el observador. Pero aún dejando de lado la tarea en sí misma, esta forma de ejercer la libertad de observación choca con la programabilidad del sistema. Cada uno de los comportamientos de un sistema computacional está generado por un programa, una prescripción que lo codifica. Un *mapping* para cada computación de cada programa *cambia las reglas del juego* constantemente. Así, se sustituye el proceso generativo que imparte un programa, por un conjunto de observaciones sustancialmente diferentes sobre el sistema. Esto se agrava aun más si consideramos que la presencia de una realización del programa como parte del sistema es lo que distingue a un sistema computacional de una simple descripción computacional de un sistema.

En contraste, nuestra propuesta conserva la libertad en las mútiples formas de realización que determinan la relación de implementación, permitiendo comprender los estados de un sistema de cualquier manera. Esto está de acuerdo con la investigación en el desarrollo de nuevo *hardware* donde se intenta permanentemente explotar nuevas propiedades de los sistemas físicos para mejorar la eficiencia en las implementaciones. Nuestro enfoque acepta la hipotética afirmación de que "todo puede computar"; el desafío es encontrar cómo. Pero la libertad que otorga, de ninguna manera implica la afirmación de que "todo computa". Para mostrar que un sistema lleva adelante ciertas computaciones es necesario indicar cómo se *realiza* el programa, cómo se *observan* los resultados, y más generalmente, cómo los comportamientos se relacionan con los programas, sean cuales sean los aspectos considerados relevantes (*i.e.* cómo se "obedece" la prescripción). Estas condiciones delimitan el problema del pancomputacionalismo.

Nuestro enfoque tiene algunas conexiones con las propuestas que presentamos previamente. Acordamos con Copeland cuando responde a la pregunta sobre cuándo un "dispositivo computa" postulando que "computar es ejecutar un algoritmo" (un programa en nuestro caso). Un "algoritmo" es un procedimiento mecánico, específico a una "arquitectura", que utiliza las operaciones primitivas que esta provee. La

"arquitectura" es lo que determina la semántica funcional del "algoritmo", de forma más específica, pero en el mismo espíritu que lo hace nuestra función de interpretación. Pero la relación de modelización entre una "arquitectura" en conjunto con un "algoritmo" y un sistema, queda mediada por un esquema de *labeling*, que, aunque en menor grado, introduce la libertad que utilizan Putnam y Searle. Así la solución final de Copeland depende de la noción de modelos "honestos". Por el contrario, nuestra noción de implementación, aunque otorga libertad en la interpretación de los estados de un sistema físico subyacente, versa sobre todos los posibles programas que el sistema computacional tiene como entrada. Esta universalidad, obliga a una correspondencia sistemática que limita drásticamente la libertad en la observación.

En el caso de la propuesta de Piccinini, coincidimos en que los estados computacionales no se individuan semánticamente, y consideramos que su descripción mecanicista de computadoras es correcta y arroja luz sobre la naturaleza computacional de los sistemas. Proponemos que nuestro concepto relacional de intérpete (y las jerarquías que se desprenden) establece un criterio de demarcación más general y bien definido sobre cúan computacional es un sistema. Bajo este marco, se puede interpretar el trabajo de Piccinini como una demostración de que una familia de dispositivos con ciertas arquitecturas satisfacen este criterio. Los requerimientos que impone sobre las entidades manipuladas por los sistemas computacionales parecen ser condiciones necesarias para la realizabilidad de los intérpretes. Pero como mencionamos más arriba, el enfoque mecanicista de Piccinini parece estar demasiado sesgado por ejemplos particulares de computadoras.

2.5. Discusión

En este capítulo, nos enfocamos en tres problemas relacionados: qué es computar, cuándo un sistema es computacional, y cuál es la relación entre los aspectos abstractos y concretos de una computación. Los tres problemas están relacionados, y es difícil dar cuenta de uno sin considerar los otros. Pero son problemas diferentes. Ensayamos respuestas que están de acuerdo con las nociones tanto de la teoría de la computación como de las práctica de las ciencias de la computación. Al mismo tiempo, discutimos el problema del pancomputacionalismo, propio del ámbito de la filosofía de la mente. La noción generalizada de intérprete que proponemos permite discutir estos problemas en términos más adecuados.

La principal característica de un intérprete es su habilidad para ser programado para comportarse de diferentes maneras. La variedad de posibles comportamientos que pueden ser codificados determina su nivel de programabilidad. Consideramos que éste es un rasgo distintivo para aceptar un sistema como computacional. Así, ser computacional es relativo a un conjunto dado de comportamientos, una forma de codificar tales comportamientos (los programas, o la sintaxis del lenguaje de progra-

mación, cuando aplica), y la forma de observar el sistema que determina la relación de implementación. En otras palabras, no tiene sentido usar el verbo "computar" de forma intransitiva. Un sistema puede computar un cierto conjunto de comportamientos *via* una codificación y *observado en una forma predeterminada*. Cuando un intérprete está computando, da lugar a un proceso que produce a partir de una prescripción particular (y como tal, abstracta) un comportamiento particular del sistema (y como tal, concreto). La relación de implementación es la que media entre los aspectos abstractos y concretos, y entendida como una relación entre relaciones, da cierta preminencia a los aspectos prescriptivos.

Creemos que el concepto de intérprete es central para una filosofía de la computación. Por un lado, los intérpretes son ubícuos en las ciencias de la computación. Por otro lado, su introducción permite iluminar algunos problemas persistentes. Algunas de las ventajas de su uso se resumen a continuación:

- Algunas cuestiones ontológicas, como qué es un programa, qué es un computación y qué es un sistema computacional pueden ser planteadas en términos más precisos, admitiendo así respuestas más claras.
- Se establecen condiciones mínimas para que un sistema sea computacional en cierto grado. Estas condiciones no dependen de la tecnología actual.
- Las jerarquías que se derivan de la concepción de intérprete pueden ser utilizadas para relacionar y comparar diferentes sistemas. La pregunta acerca de si un sistema es computacional o no, puede ser reformulada en una pregunta más general sobre la variedad de comportamientos que produce un sistema y/o su programabilidad.
- Algunas cuestiones epistemológicas se clarifican, con las consecuencias metodológicas que esto conlleva. El concepto de intérprete media entre los aspectos abstractos y concretos de la computación. La justificación, normalmente empírica, de la implementación que caracteriza un intérprete, valida las premisas fundamentales de la semántica de los lenguajes de programación. Así es posible utilizar técnicas como la verificación formal, e incluso el *testing* de programas, con confianza.
- La caracterización de un sistema como computacional difumina la distinción entre *software* y *hardware*, uno de los mitos denunciado por Moor [106], lo cual es coherente con las prácticas de las ciencias de la computación (programar una máquina "virtual" o "real" es transparente para el programador; operaciones de "bajo nivel" pueden estar implementadas en *hardware*, *firmware* o directamente *software*, etc.).

- La "composición" de intérpretes, compiladores y *programas-intérpretes* permite caracterizar los diferentes "niveles de abstracción" que se suceden hacia el interior de los sistemas informáticos. El concepto de intérprete es el vínculo entre tales niveles.

Separation logics

3

Las *separation logics* conforman una familia de lógicas en las que la sintáxis de una fórmula puede describir la partición del modelo que la satisface en componentes disjuntos. El lenguaje de fórmulas incluye una *conjunción espacial* $*$ (o *conjunción de separación*) que establece que las subfórmulas se satisfacen en partes disjuntas del modelo, y su operador adjunto, la *implicación espacial* $-\!\!*$, que permite razonar sobre posibles extensiones del mismo. El término "*Separation Logic*" refiere tanto a una extensión de la lógica de primer orden con los operadores *espaciales*, así como a la lógica de Hoare que la utiliza como lenguaje de aserciones.

El concepto central que subyace a la conjunción espacial está implícito en el trabajo de Burstall [33]. Alrededor de 1999, Reynolds intenta introducir el operador $*$ de forma explícita en una lógica de Hoare con algunas dificultades técnicas. Poco tiempo después, de forma independiente, tanto Reynolds [135] como Ishtiaq y O'Hearn [85] presentan una lógica *intuicionista* que lo incorpora. El carácter intuicionista impone una propiedad de monotonía, según la cual si un modelo satisface una fórmula cualquier otro modelo que lo incluya también lo hace. Esta característica representa un límite, por ejemplo, para razonar sobre *memory leaks*.

Al tomar conciencia que la lógica es un caso particular de la semantica de recursos de la *Bunched Implications Logic*, desarrollada por Pym y el mismo O'Hearn [115, 131], Ishtiaq y O'Hearn presentan además una versión clásica incluyendo el operador adjunto $-\!\!*$. La versión clásica se libera de la propiedad de monotonía y permite, entre otras cosas, razonar sobre la *eliminación* explícita de memoria. Resulta además más expresiva ya que permite codificar la semántica intuicionista a través de fórmulas con una sintaxis específica.

Originalmente las formulas de cualquier *separation logic* se interpretan respecto a un modelo de memoria que separa las variables del programa (estáticas) de los registros de memoria dinámica de tamaño variable. En un trabajo posterior [136], Reynolds presenta una generalización de la lógica, definiendo su semántica en términos de un modelo de memoria RAM, en el que los punteros referencian a celdas individuales de memoria, permitiendo así el uso irrestricto de expresiones aritméticas sobre punteros.

En esta sección introducimos una *Separation Logic* clásica (que denotamos como SL) respecto a un modelo de memoria estructurada en registros de diferente tamaño. Presentamos el lenguaje de fómulas y su semántica respecto al modelo de memoria

estándar, un lenguaje de programación imperativo con comandos para la manipulación de la memoria dinámica, y un sistema de especificación y razonamiento sobre programas *a la Hoare* (que denotamos SL^H). Además introducimos una extensión para trabajar con tipos abstactos de datos [120]. Además de los trabajos referidos más arriba, se pueden encontrar otras presentaciones de la SL^H y ejemplos de aplicación, desde programas relativamente sencillos a algoritmos extremadamente complejos como el *traversal* Schorr-Waite, en [155, 116, 152, 15, 24, 25, 117].

3.1. Fórmulas y modelos

Los modelos de la SL representan la visión del programador sobre el espacio de memoria de un lenguaje de programación imperativo de bajo nivel *a la C*. Un modelo se asimila a un estado de memoria, y se divide en dos partes: un *stack* que asigna valores a las variables (que luego son las variables de programa), y un *heap* que asigna un registro de memoria de tamaño arbitrario a cada dirección de memoria dinámica que se encuentra definida. Las variables y los campos de los registros pueden tener asignado cualquier valor elemental de un conjunto particular (como los números enteros), o una dirección de memoria. En este caso se denominan *punteros* y decimos que *referencian* o *apuntan* a tal dirección. Una dirección puede ser *válida*, si el *heap* le asigna un registro, o *inválida* en caso contrario. Cuando una variable o registro tiene asignada una dirección inválida lo denominamos puntero *dangling*. El valor distinguido *nil* denota siempre una dirección de memoria inválida.

Definición 7 (Modelo)**.** *Dado un conjunto infinito de valores Loc (interpretados como direcciones de memoria), con un elemento distinguido nil ∈ Loc, un conjunto infinito de valores Atoms (interpretado como los valores elementales), y un conjunto enumerable de variables Var, el conjunto de modelos States está dado por las siguientes ecuaciones:*

$$\begin{aligned} \mathit{Values} &\doteq \mathit{Loc} \cup \mathit{Atoms} && \text{donde } \mathit{Loc} \perp \mathit{Atoms} \\ \mathit{Stacks} &\doteq \mathit{Var} \rightarrow \mathit{Values} \\ \mathit{Heaps} &\doteq (\mathit{Loc} - \{\mathit{nil}\}) \rightharpoonup_{\mathit{fin}} \mathit{Values}^+ \\ \mathit{States} &\doteq \mathit{Stacks} \times \mathit{Heaps} \end{aligned}$$

donde $A \perp B$ indica que los conjuntos A y B son disjuntos, A^+ es el conjunto de tuplas sobre A de orden arbitrario,[1] *y $\rightharpoonup$ ($\rightharpoonup_{fin}$) es el conjunto de funciones parciales (con dominio finito).*

El lenguaje de fórmulas extiende la lógica clásica de primer orden con los operadores espaciales y dos predicados atómicos que caracterizan los *heaps* elementales.

[1] El primer campo de una tupla es el 0-ésimo.

Definición 8 (Sintaxis de SL)**.** *El conjunto de expresiones Expr, relaciones Rel y fórmulas Form sobre Var está dado por la siguiente gramática:*

$$\begin{aligned} Expr \ni e &::= x \mid \mathbf{null} \mid \ldots \\ Rel \ni r &::= e = e \mid \ldots \\ Form \ni p &::= r \mid \mathbf{emp} \mid x \mapsto \vec{e} \mid p \vee p \mid \neg p \mid p * p \mid p \mathrel{-\!\!*} p \mid \forall x \cdot p \end{aligned}$$

donde $x \in Var$ *y* $\vec{e} \in Expr^{+}$.

Como es usual, los restantes operadores lógicos, constantes y cuantificador ($\wedge$, $\Rightarrow$, $\Leftrightarrow$, **true**, **false**, $\exists$) se expresan como notación derivada. Usamos _ para denotar una variable cuantificada cuyo *nombre* es irrelevante. Por ejemplo, la fórmula $x \mapsto _$ denota $\exists y \cdot x \mapsto y$. El operador $\neg$ tiene mayor precedencia. Luego le siguen $*, \wedge$ y $\vee$, a continuación $\mathrel{-\!\!*}, \Rightarrow, \Leftrightarrow$ y finalmente $\forall$ y $\exists$. Como es usual utilizamos los símbolos (y) para demarcar términos dentro de una fórmula.

Notar que no damos una sintaxis completa para las expresiones y relaciones ya que dependen del conjunto de valores elementales *Atoms*. Usualmente las expresiones contienen los numerales $1, 2, \ldots$ y operaciones aritméticas entre ellos, y las relaciones están formadas por las operaciones de comparación $\neq, <, \leqslant, >, \geqslant, \ldots$. Como única restricción, la semántica de las expresiones y relaciones debe depender únicamente del *stack*.

Definición 9 (Semántica de expresiones y relaciones de SL)**.** *Dado* $s \in Stacks$*, la semántica de las expresiones y relaciones está dada por funciones*

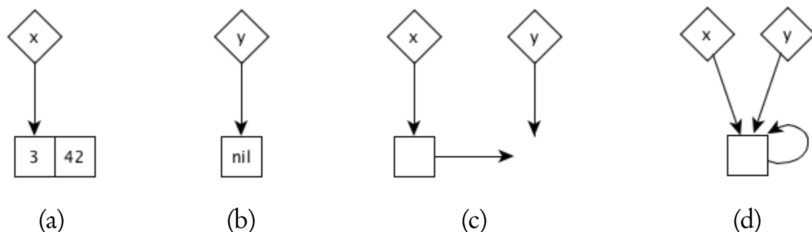
$$\begin{aligned} [\![\ ]\!]_s &\in Expr \to Values \\ \{\![\ ]\!\}_s &\in Rel \to \{true, false\} \end{aligned}$$

que satisfacen las siguientes ecuaciones:

$$\begin{aligned} [\![x]\!]_s &\doteq s.x \\ [\![\mathbf{null}]\!]_s &\doteq nil \\ \{\![e = e']\!\}_s &\doteq [\![e]\!]_s = [\![e']\!]_s \end{aligned}$$

donde $x \in Var$*, y* $e, e' \in Expr$*.*

A diferencia de las expresiones, las fórmulas dependen tanto del *stack* como del *heap*. Los operadores espaciales y los nuevos predicados atómicos refieren principalmente al segundo componente del modelo, aunque en algunos casos establecen condiciones sobre el primero. El predicado **emp** refiere al *heap* vacío. Se satisface únicamente en caso que no exista ningún registro en la memoria dinámica y por lo tanto, ninguna dirección de memoria válida, pero no establece condiciones sobre el



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

Figura 3.1: Modelos de $x \mapsto 3, 42$ (a), $y \mapsto$ **null** (b), y $x \mapsto y$ (c y d).

stack. Por el contrario, el predicado $x \mapsto \vec{e}$ refiere al *heap* unitario. Establece que existe un único registro de memoria dinámica que almacena los valores $\vec{e}$, cuya dirección es asignada por el *stack* a la variable x. Los siguientes ejemplos ayudan a clarificar los alcances y límites de este predicado:

Ejemplo 9. *La fórmula* $x \mapsto 3, 42$ *establece que el* heap *contiene una único registro* $(3, 24)$ *cuya dirección* l *no está explícita, aunque* $s.x = l$ *(fig. 3.1.a).*

Ejemplo 10. *La fórmula* $x \mapsto _$ *simplemente especifica que el puntero* x *es un puntero válido, es decir se encuentra asignado a un registro de memoria del* heap, *y tal registro es el único existente.*

Ejemplo 11. *De manera similar,* $y \mapsto$ **null** *establece que* y *apunta a un registro* (nil) *(fig. 3.1.b). Notar que esto es muy diferente a que* y *sea* dangling, *caracterizado, por ejemplo, por la fórmula* $y =$ **null**. *De hecho, cualquier fórmula de la forma* $x \mapsto \vec{e}$ *establece que* x *es necesariamente un puntero válido.*

Ejemplo 12. *La fórmula* $x \mapsto y$ *se satisface cuando el puntero* x *referencia a un registro* (y), *pero no establece ninguna condición sobre la variable* y. *Esta variable puede contener un valor elemental, o ser un puntero. En el segundo caso, puede suceder que sea* dangling *(fig. 3.1.c) o contenga una dirección válida. Esto último sólo puede darse en el caso particular que* $s.x = s.y$ *(fig. 3.1.d), cuando además el modelo satisface la fórmula equivalente* $y \mapsto x$.

El operador de *conjunción espacial* $*$ compone *heaps* disjuntos. La fórmula $p * q$ se satisface en un modelo cuando el *heap* puede dividirse en dos *subheaps* cuyos dominios son disjuntos, de manera que uno de ellos conforma el modelo que satisface p y el otro el modelo que satisface q.

Ejemplo 13. *Dado que* **emp** *caracteriza el* heap *vacío,* $x \mapsto 3, 42 *$ **emp** *se satisface en el modelo del ejemplo 9 (fig. 3.1.a). Podemos anticipar entonces que* **emp** *es el elemento neutro de* $*$.

Ejemplo 14. *La fórmula* $x \mapsto 3, 42 * y \mapsto$ **null** *se satisface en un modelo que es la unión de los modelos de los ejemplos 9 y 11 (fig. 3.1.a y 3.1.b).*

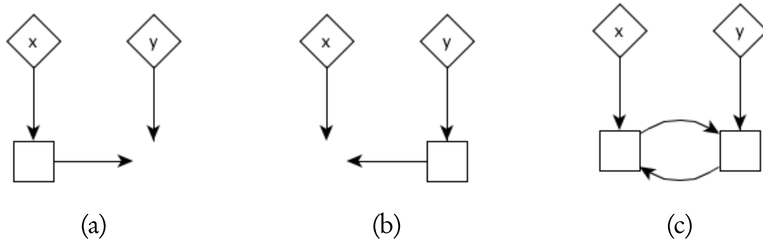


Figura 3.2: Modelos de $x \mapsto y$ (a), $y \mapsto x$ (b), y $x \mapsto y * y \mapsto x$ (c).

Ejemplo 15. *La fórmula* $x \mapsto 3,42 * x \mapsto 7,33$ *es insatisfactible. Esto no está relacionado con los diferentes valores de los registros sino con el uso de un mismo puntero* x *para referenciar a dos registros que deben tener direcciones diferentes. De esta manera, la combinación con* $*$ *de predicados* $x_1 \mapsto \vec{e}_1, \dots, x_n \mapsto \vec{e}_n$ *establece que el* stack *asigna diferentes direcciones a los punteros* $x_1, \dots, x_n$.

Ejemplo 16. *Retomemos la fórmula y modelos del ejemplo 12. La combinación de dos términos* $x \mapsto y * y \mapsto x$*, precluye para ambas subfórmulas los modelos como los de la figura 3.1.d, ya que debe darse* $s.x \neq s.y$*. Los modelos posibles entonces son aquellos en que los registros se apuntan entre sí (fig. 3.2). Notar cómo los punteros* x *e* y *son* dangling *en los modelos de las fórmulas* $y \mapsto x$ *y* $x \mapsto y$ *respectivamente, aunque no lo son para el modelo combinado. El interjuego entre los predicados de la forma* $z \mapsto w$ *donde* w *es un puntero* dangling, *y el operador* $*$ *es crucial para la especificación de estructuras enlazadas.*

El operador $-\!*$ es el adjunto de $*$. Posee una semántica difícil de capturar intuitivamente: una fórmula $p -\!* q$ se satisface en un modelo si cualquier *heap* (disjunto) que sastifaga p extiende el *heap* original de modo que satisface q. Esta idea se sintetiza en la validez de la siguiente propiedad de *modus ponens* (que presentamos en toda su generalidad en la sección siguiente): $p * (p -\!* q) \Rightarrow q$.

Ejemplo 17. *Cuando el dominio de los* heaps *que satisfacen* p *y* p' *está completamente determinado, la fórmula* $p * (p' -\!* q)$ *puede interpretarse como una* actualización *del* heap, *dada por la* eliminación *del* subheap *que satisface* p' *y la extensión con el* heap *que satisface* p*. A este patrón lo llamamos* deletion followed by extension. *De esta manera la fórmula* $x \mapsto 7 * (x \mapsto 4 -\!* q)$ *se satisface en un modelo en que* x *apunta al registro* (7) *de tal manera que si el registro se actualizara con el valor* 4, *entonces el modelo satisfaría* q.

Ejemplo 18. *Lamentablemente una fórmula* $p -\!* q$ *puede satisfacerse* trivialmente *si no se asegura que todo* heap *que satisface* p *es disjunto del* heap *que satisface* q*. En el ejemplo anterior, el término* $x \mapsto 7$ *precluye la posibilidad que un registro cuya dirección es* x *forme parte del* subheap *que satisface* q*. Pero consideremos la fórmula* $x \mapsto 7 -\!* x \mapsto 7$.

Claramente se satisface en el heap *vacío. Pero tambien se satisface en cualquier modelo que contenga un registro referenciado por x, y posiblemente una cantidad arbitraria de otros registros, ya que de este modo no existe ninguna posible extensión (disjunta).*

Los operadores heredados de la lógica de primer orden mantienen su semántica clásica estándar. Puede ser útil comparar el comportamiento del operador $\wedge$ respecto a $*$. Una fómula $p \wedge q$ se satisface en un modelo si tanto p como q se satisfacen para exactamente el mismo *heap*. Así el operador $\wedge$ introduce potenciales alias entre los punteros definidos en las subfórmulas.

Ejemplo 19. *La fórmula $x \mapsto v \wedge y \mapsto w$ se satisface para un único módelo que contiene un único registro. Necesariamente debe valer que $s.x = s.y$ y $s.v = s.w$. Por lo tanto una fórmula como $x \mapsto 3 \wedge y \mapsto 7$ es insatisfactible.*

Ejemplo 20. *Contrastando con el ejemplo 16, la fórmula $x \mapsto y \wedge y \mapsto x$ tiene un único modelo como el de la figura 3.1.d.*

La semántica formal de una fórmula está dada por una *forcing relation* que determina los modelos que la satisfacen.

Definición 10 (Semántica de fórmulas de SL). *Dado un modelo $s, h \in States$, la relación de satisfacción $\models \in States \times Form$ está definida por inducción en las fórmulas de acuerdo a las siguientes ecuaciones:*

$$
\begin{array}{ll}
s,h \models r & \textit{si y sólo si } \{\![r]\!\}_s \\
s,h \models \mathbf{emp} & \textit{si y sólo si } h = \emptyset \\
s,h \models x \mapsto e_0, \dots, e_n & \textit{si y sólo si } h.[\![x]\!]_s = ([\![e_0]\!]_s, \dots, [\![e_n]\!]_s) \textit{ y } |dom.h| = 1 \\
s,h \models p \vee q & \textit{si y sólo si } s,h \models q \textit{ o } s,h \models p \\
s,h \models \neg p & \textit{si y sólo si no vale } s,h \models p \\
s,h \models p * q & \textit{si y sólo si existen } h_1 \textit{ y } h_2 \textit{ tal que } h_1 \perp h_2 \\
 & \quad \textit{y } h_1 \cup h_2 = h \textit{ y } s,h_1 \models p \textit{ y } s,h_2 \models q \\
s,h \models p -\!\!* q & \textit{si y sólo si para todo } h_1 \textit{ tal que } h_1 \perp h, \textit{ si } s,h_1 \models p, \\
 & \quad \textit{entonces } s,h_1 \cup h \models q \\
s,h \models \forall x \cdot p & \textit{si y sólo si para todo } v \in Values, s[x \mapsto v], h \models p
\end{array}
$$

donde $r \in Rel$, $x \in Var$, $e_0, \dots, e_n \in Expr$, y $p, q \in Form$. Con $dom.f$ denotamos el dominio de la función f, con $|A|$ el tamaño del conjunto A, y con $f[x \mapsto v]$ la actualización (o extensión) puntual de la función f, asignando el valor v a $f.x$. Abusando de la notación, escribimos $h \perp h'$ para denotar $dom.h \perp dom.h'$.

3.2. Cálculo de fórmulas

No formalizamos el sistema deductivo de la SL, sino que presentamos esquemas de axiomas como reglas de inferencia sobre fórmulas *válidas*.[2]

Definición 11 (Validez de una fórmula de SL). *Dada una fórmula* $p \in Form$*, decimos que* p *es* válida*, denotado* $\vDash p$*, si para cualquier estado* $s, h \in States$ *se satisface que* $s, h \vDash p$*.*

En la SL las reglas de cálculo de la lógica de primer orden clásica permanecen válidas para los operadores heredados. Se debe notar que la SL es una lógica subestructural, y por lo tanto no existe una regla de contracción ni debilitamiento para $*$. Sin embargo sí se validan las esperables reglas de asociatividad, conmutatividad, elemento neutro y (semi) distributividad.

Lema 4 (Reglas de cálculo de SL). *Sean* $p, q, r \in Form$ *fórmulas cualesquiera, entonces*

Conmutatividad:

$$\vDash p * q \Leftrightarrow q * p$$

Asociatividad:

$$\vDash (p * q) * r \Leftrightarrow p * (q * r)$$

Elemento neutro:

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

Distributividad con $\vee$**:**

$$\vDash (p \vee q) * r \Leftrightarrow (p * r) \vee (q * r)$$

Semidistributividad con $\wedge$**:**

$$\vDash (p \wedge q) * r \Rightarrow (p * r) \wedge (q * r)$$

Distributividad con $\exists$**:**

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

Semistributividad con $\forall$**:**

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

[2] A lo largo de todo el trabajo, utilizamos libremente reglas de inferencia en diferentes contextos, incorporando muchas veces condiciones *de lado* como premisas, para que la presentación sea más compacta. Además cuando no existen premisas, omitimos la típica barra horizontal

donde $FV.p$ denota el conjunto de variables libres de p, definido de manera estándar.

Existen además otras tres reglas de inferencia que caracterizan la monotonía de $*$ y la relación de adjunción entre $*$ y $-\!\!*$.

Lema 5 (Reglas de cálculo de SL (cont.))**.** *Sean $p, q, r, s \in Form$ fórmulas cualesquiera, entonces*

Monotonía:

$$\frac{\models p \Rightarrow r \qquad \models q \Rightarrow s}{\models p * q \Rightarrow r * s}$$

Adjuntividad:

$$\frac{\models p * q \Rightarrow r}{\models p \Rightarrow (q -\!\!* r)} \qquad \frac{\models p \Rightarrow (q -\!\!* r)}{\models p * q \Rightarrow r}$$

Clases de fórmulas

En la literatura [133, 151, 152, 136] se presentan diferentes clases de fórmulas definidas semánticamente. Estas clases reúnen fórmulas que satisfacen reglas de cálculo específicas y desempeñan además un papel fundamental en la consistencia de extensiones de la SL para concurrencia [146, 114], y en el soporte de nociones de abstracción [117, 120].

Una fórmula es *pure* si es independiente del *heap*. Es la única clase que puede caracterizarse además de forma sintáctica: una fórmula es *pure* si y sólo si no contiene **emp** ni $\mapsto$.

Definición 12 (Fórmula *pure*)**.** *Dada $p \in Form$, decimos que p es* pure *si para cualesquiera $s \in Stacks$, $h, h' \in Heaps$ se satisface que*

$$s, h \models p \text{ si y sólo si } s, h' \models p$$

Con Pure denotamos el mayor conjunto $P \subseteq Form$ tal que p es pure *para toda $p \in P$.*

Ejemplo 21. *Una relación como $x = y$ es una fórmula* pure*. Se satisface en cualquier* heap *ya que sólo impone la condición $s.x = s.y$ sobre el* stack *s.*

Los operadores espaciales que ocurren en una fórmula *pure* se reducen a las formas clásicas.

Lema 6 (Reglas de cálculo de SL (cont.))**.** *Sean $p, q, r \in Form$ fórmulas cualesquiera, entonces*[3]

$$\frac{p \in Pure \text{ o } q \in Pure}{\models p \wedge q \Rightarrow p * q}$$

[3]En virtud de la tercer regla, usualmente omitimos paréntesis en una fórmula que involucra $\wedge$ respecto a una fórmula *pure*.

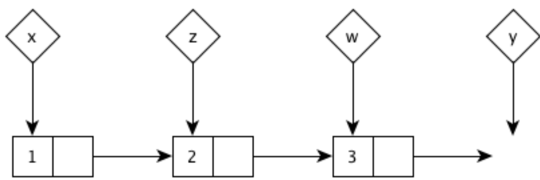


Figura 3.3: Modelo de $x \mapsto 1, z * z \mapsto 2, w * w \mapsto 3, y$

$$\frac{p, q \in Pure}{\vDash p * q \Rightarrow p \wedge q}$$

$$\frac{p \in Pure}{\vDash (p \wedge q) * r \Leftrightarrow p \wedge (q * r)}$$

$$\frac{p \in Pure}{\vDash (p \mathrel{-\!\!*} q) \Rightarrow (p \Rightarrow q)}$$

$$\frac{p, q \in Pure}{\vDash (p \Rightarrow q) \Rightarrow (p \mathrel{-\!\!*} q)}$$

Una fórmula es *strictly exact* cuando determina inequívocamente el *heap* que la satisface. Si una fórmula está compuesta únicamente por **emp**, $\mapsto$ y $*$ entonces es *strictly exact*.

Definición 13 (Fórmula *strictly exact*). *Dada $p \in Form$, decimos que p es* strictly exact *si para cualesquiera $s \in Stacks$, $h, h' \in Heaps$ se satisface que*

$$si \;\; s, h \vDash p \;\; y \;\; s, h' \vDash p \;\; entonces \;\; h = h'$$

Con $Strict$ denotamos al mayor conjunto $S \subseteq Form$ tal que p es strictly exact *para toda $p \in Strict$.*

Este concepto permite introducir la siguiente regla que es útil en contextos donde existe un *sharing* específico, caracterizado con el operador $\wedge$.

Lema 7 (Reglas de cálculo de SL (cont.)). *Sean $p, q \in Form$ fórmulas cualesquiera, entonces*

$$\frac{q \in Strict}{\vDash (q * \mathbf{true}) \wedge p \Leftrightarrow q * (q \mathrel{-\!\!*} p)}$$

Ejemplo 22. *La fórmula* $x \mapsto (1, z) * z \mapsto (2, w) * w \mapsto (3, y)$ *caracteriza tres registros enlazados. Es* strictly exact *porque determina completamente la dirección y contenido de cada registro de la memoria dinámica. Los modelos que la satisfacen, cuando* y *es un puntero, son como los de la figura 3.3.*

Ejemplo 23. *Si relajamos la fórmula del ejemplo anterior cuantificando los valores, la fórmula* $x \mapsto (_, z) * z \mapsto (_, w) * w \mapsto (_, y)$ *aún especifica la existencia de tres registros enlazados, pero que almacenan valores indeterminados. Así la fórmula deja de ser* strictly exact.

Existen dos clases que relajan gradualmente las condiciones de la clase *strictly exact*. La clase *domain exact* agrupa las fórmulas que caracterizan inequivocamente no el *heap* en su totalidad sino su domino. Sintácticamente esto habilita la utilización del cuantificador existencial sobre los valores apuntados en los términos $\mapsto$ (como en el ejemplo anterior). La clase *precise* reune las fórmulas que, para cualquier *heap*, sólo se satisfacen en un único *subheap*.

Definición 14 (Fórmula *domain exact* y *precise*)**.** *Sean* $p \in Form$, $s \in Stacks$ *y* $h, h_1, h_2 \in Heaps$. *Decimos que* p *es* domain exact *si se satisface que*

$$si\ \ s, h \vDash p\ \ y\ \ s, h' \vDash p\ \ entonces\ \ dom.h = dom.h'$$

Con Domain denotamos el mayor conjunto $D \subseteq Form$ *tal que* p *es* domain exact *para toda* $p \in D$.

Por otro lado, decimos que p *es* precise *si se satisface que*

$$si\ \ h_1 \subseteq h\ \ y\ \ h_2 \subseteq h\ \ y\ \ s, h_1 \vDash p\ \ y\ \ s, h_2 \vDash p\ \ entonces\ \ h_1 = h_2$$

Con Precise denotamos el mayor conjunto $P \subseteq Form$ *tal que* p *es* precise *para toda* $p \in P$.

Ejemplo 24. *La fórmula* $x \mapsto (_, z) * z \mapsto (_, w) * w \mapsto (_, y)$ *del ejemplo 23 es* domain exact. *Los modelos que la satisfacen son como los de la figura 3.4.a. (cuando* y *es un puntero). La fórmula* $\exists z, w \cdot x \mapsto (_, z) * z \mapsto (_, w) * w \mapsto (_, y)$, *más débil, no es* domain exact *pero* precise, *porque aunque las direcciones de los registros no están especificadas, quedan determinadas transitivamente a partir del enlace desde la variable* x. *Esta fórmula admite modelos como los anteriores, pero tambien como los de la figura 3.4.b.*

Para fórmulas en las clases anteriores, la semidistributividad de $*$ con $\wedge$ y $\forall$ se vuelven completas.

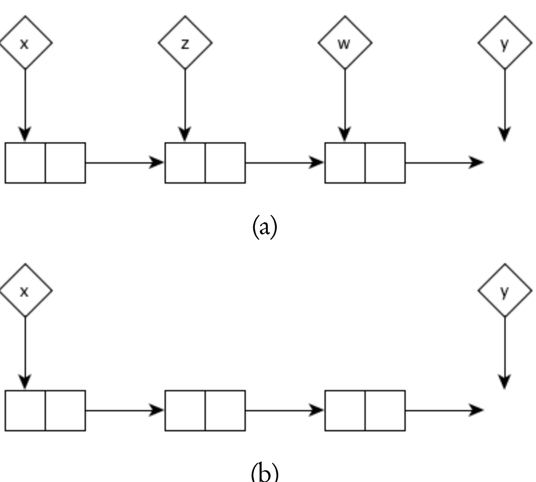


Figura 3.4: (a) Modelo de $x \mapsto (_, z) * z \mapsto (_, w) * w \mapsto (_, y)$, y (b) modelo de $\exists z, w \cdot x \mapsto (_, z) * z \mapsto (_, w) * w \mapsto (_, y)$.

Lema 8 (Reglas de cálculo de SL (cont.)). *Sean $p, q, r \in Form$ fórmulas cualesquiera, entonces*

Distributividad con $\wedge$:

$$\frac{r \in Domain \cup Precise}{\models (p \wedge q) * r \Leftrightarrow (p * r) \wedge (q * r)}$$

Distributividad con $\forall$:

$$\frac{r \in Domain \cup Precise \qquad x \notin FV.r}{\models (\forall x \cdot p) * r \Leftrightarrow \forall x \cdot p * r}$$

Las clases *strictly exact*, *domain exact* y *precise* están incluidas unas en otras.

Lema 9. *$Strict \subset Domain \subset Precise$.*

3.3. El lenguaje de programación

Para describir programas que utilizan memoria dinámica, se extiende el lenguaje de programación imperativo simple, originalmente axiomatizado por Hoare [80], con comandos para su manipulación.

Definición 15 (Comandos de SL^H)**.** *Los conjuntos de expresiones* booleanas *Bool y comandos Comm estan dado por la siguiente gramática:*

$$Bool \ni b ::= r \mid \neg b \mid b \wedge b \mid b \vee b$$

$$Comm \ni c ::= x := e \mid \textbf{skip} \mid c;c \mid \textbf{if } b \textbf{ then } c \textbf{ else } c \textbf{ fi} \mid \textbf{while } b \textbf{ do } c \textbf{ od}$$

$$\mid x := \textbf{cons}(\vec{e}) \qquad \text{(Construcción)}$$

$$\mid x.i := e \qquad \text{(Mutación)}$$

$$\mid x := x.i \qquad \text{(Consulta)}$$

$$\mid \textbf{dispose}(x) \qquad \text{(Destrucción)}$$

donde $x \in Var$, $e \in Expr$, $r \in Rel$, $\vec{e} \in Expr^+$ *y* $i \in \mathbb{N}_0$.

Las expresiones *booleanas Bool* son simples combinaciones proposicionales de las relaciones entre expresiones.

Definición 16 (Semántica de expresiones *booleanas* de SL^H)**.** *Dado* $s \in Stacks$, *la semántica de las expresiones* booleanas *está dada por una función*

$$(\!|\ |\!)_s \in Bool \to \{true, false\}$$

que depende de la semántica de relaciones $\{\!\![\ ]\!\!\}_s$ *y sigue la semántica estándar de la lógica proposicional.*

El comando de construcción $x := \textbf{cons}(e_0, \dots, e_n)$ aloca un nuevo registro de tamaño $n + 1$ en la memoria dinámica, cuyos campos almacenan los valores de las expresiones $e_0, \dots, e_n$, y asigna al puntero x su dirección de memoria. El comando de mutación $x.i := e$ asigna al i-ésimo campo del registro apuntado por x el valor de la expresión e. El comando de consulta $x := y.i$ asigna a la variable x el valor del i-ésimo campo del registro apuntado por y. Finalmente el comando de destrucción **dispose**(x) elimina de la memoria el registro apuntado por x, que se convierte en *dangling*. La semántica del resto de los comandos es estándar.

Cabe remarcar que, aunque los comandos de mutación y consulta utilicen el mismo símbolo que la asignación (:=), son comandos completamente diferentes. Además las expresiones no referencian al *heap*, por lo tanto, la única manera de accederlo es a través de los comandos específicos.

Como es usual, formalizamos los comandos utilizando una semántica operacional *small step*, a través de una relación de transición entre configuraciones. A diferencia del lenguaje imperativo simple, cuyos comandos pueden ejecutarse exitosamente en cualquier estado, los comandos de manipulación del *heap* pueden fallar. Esto sucede cuando se muta o consulta el campo i de un registro de tamaño n con $n < i$, y cuando se accede de alguna manera a un puntero *dangling*. En tal caso decimos que ocurre una falla de memoria y el programa termina anormalmente, lo que denotamos con la configuación **abort**.

Definición 17 (Configuración de SL^H). *El conjunto de configuraciones Conf se define como*

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

Dada $\zeta \in Conf$ una configuración, decimos que es

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

Definición 18 (Semántica de comandos de SL^H). *La relación de transición entre configuraciones $\leadsto \in Conf \times Conf$ queda definida por las siguientes reglas:*

Skip:

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

Asignación:

$$x := e, (s, h) \leadsto (s[x \mapsto [\![e]\!]_s], h)$$

Construcción:

$$\frac{l \in Loc - dom.h}{x := \mathbf{cons}(e_0, \dots, e_n), (s, h) \leadsto (s[x \mapsto l], h[l \mapsto ([\![e_0]\!]_s, \dots [\![e_n]\!]_s)])}$$

Notar que l se elige de forma no determinística.

Mutación:

$$\frac{l = [\![x]\!]_s \qquad h.l = (e_0, \dots, e_n) \qquad 0 \leqslant i \leqslant n}{x.i := e, (s, h) \leadsto (s, h[l \mapsto (e_0, \dots, e_n)[i \mapsto e]])}$$

$$\frac{l = [\![x]\!]_s \qquad h.l = (e_0, \dots, e_n) \qquad i > n}{x.i := e, (s, h) \leadsto \mathbf{abort}}$$

$$\frac{[\![x]\!]_s \notin dom.h}{x.i := e, (s, h) \leadsto \mathbf{abort}}$$

Consulta:

$$\frac{l = [\![y]\!]_s \qquad h.l = (e_0, \dots, e_n) \qquad 0 \leqslant i \leqslant n}{x := y.i, (s, h) \leadsto (s[x \mapsto e_i], h)}$$

$$\frac{l = [\![y]\!]_s \qquad h.l = (e_0, \dots, e_n) \qquad i > n}{x := y.i, (s, h) \leadsto \mathbf{abort}}$$

$$\frac{[\![y]\!]_s \notin dom.h}{x := y.i, (s, h) \leadsto \mathbf{abort}}$$

Destrucción:

$$\frac{l = [\![x]\!]_s \qquad l \in dom.h}{\textbf{dispose}(x), (s,h) \rightsquigarrow (s, h|_{dom.h-\{l\}})}$$

$$\frac{[\![x]\!]_s \notin dom.h}{\textbf{dispose}(x), (s,h) \rightsquigarrow \textbf{abort}}$$

Secuenciación:

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

$$\frac{c_1, (s,h) \rightsquigarrow^* \textbf{abort}}{c_1; c_2, (s,h) \rightsquigarrow^* \textbf{abort}}$$

Alternativa:

$$\frac{(\![b]\!)_s \qquad c_1, (s,h) \rightsquigarrow^* \zeta}{\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 \textbf{ fi}, (s,h) \rightsquigarrow^* \zeta}$$

$$\frac{no\ (\![b]\!)_s \qquad c_2, (s,h) \rightsquigarrow^* \zeta}{\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 \textbf{ fi}, (s,h) \rightsquigarrow^* \zeta}$$

Ciclo:

$$\frac{(\![b]\!)_s \qquad c, (s,h) \rightsquigarrow^* (s', h')}{\textbf{while } b \textbf{ do } c \textbf{ od}, (s,h) \rightsquigarrow \textbf{while } b \textbf{ do } c \textbf{ od}, (s', h')}$$

$$\frac{(\![b]\!)_s \qquad c, (s,h) \rightsquigarrow^* \textbf{abort}}{\textbf{while } b \textbf{ do } c \textbf{ od}, (s,h) \rightsquigarrow \textbf{abort}}$$

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

donde $x, y \in Var$, $e, e_0, \ldots, e_n \in Expr$, $b \in Bool$, $i, n \in \mathbb{N}_0$. *Abusando de la notación, utilizamos* $t[i \mapsto v]$ *para denotar la actualización del i-ésimo campo de la tupla* t *con el valor* v*, y con* $f|_A$ *la restricción de la función* f *al dominio* A*. Ademas escribimos* $\zeta \rightsquigarrow^* \zeta'$ *para indicar que existe una secuencia finita de transiciones de* ζ *a* ζ'*.*

El siguiente concepto caracteriza las configuraciones que no fallan. Notar que está formulado con la noción de corrección parcial en mente. En presencia de ciclos, las configuraciones no sólo pueden fallar o terminar exitosamente, sino que pueden diverger infinitamente sin llegar a una configuración terminal.

Definición 19 (Configuración *safe*)**.** *Dada una configuración* $\zeta \in Conf$*, decimos que* ζ *es* safe *si no sucede que* $c \rightsquigarrow^* \textbf{abort}$.

3.4. Especificaciones y razonamiento sobre programas

Para razonar sobre los programas utilizamos el concepto de ternas de Hoare para especificar comandos, utilizando la *Separation Logic* como lenguaje de aserciones. Adoptamos una interpretación de corrección parcial que asegura que los comandos no abortan.

Definición 20 (Especificación de SL^H)**.** *El conjunto Spec de especificaciones de comandos está dado por la siguiente gramática:*

$$Spec \ni s ::= \vdash \{\, p \,\}\; c \;\{\, p \,\}$$

donde $p \in Form$ *y* $c \in Comm$.

Definición 21 (Validez de especificación de SL^H)**.** *Decimos que una especificación* $\vdash \{\, p \,\}\; c \;\{\, q \,\}$ *es válida, denotado* $\vDash \{\, p \,\}\; c \;\{\, q \,\}$*, si para cualesquiera estados* s, h *y* $s', h' \in States$ *se satisface que*

$$\text{si } s, h \vDash p \text{ entonces } c, (s, h) \text{ es safe, y si } c, (s, h) \rightsquigarrow^* (s', h') \text{ entonces } s', h' \vDash q.$$

En tal caso decimos que la especificación demuestra *la corrección de* c *respecto a* p *y* q*. Las* aserciones p *y* q *se denominan* precondición *y* postcondición *del comando* c *respectivamente.*

Notar que las especificaciones están implícitamente cuantificadas sobre los estados, y dado que la construcción de memoria es no determinística, también sobre todas las posibles ejecuciones.

La definición de especificación satisface el lema *"well-specified programs don't go wrong"* [85]. Si se demuestra la especificación de un programa no pueden darse errores de memoria a lo largo de su ejecución, es decir, si el programa termina, lo hace en una configuración *safe*. Como consecuencia, una terna como $\{\, \textbf{true} \,\}\; c \;\{\, \textbf{true} \,\}$ no es válida en general para cualquier comando c. Sin embargo, si logramos establecer $\{\, p \,\}\; c \;\{\, \textbf{true} \,\}$ sabemos a ciencia cierta que el comando c se ejecuta sin fallas desde cualquier estado que satisfaga la precondición p.

El sistema deductivo sobre las especificaciones de SL^H incluye las reglas de inferencia de la lógica de Hoare para los comandos del lenguaje imperativo simple.

Definición 22 (Sistema deductivo de SL^H). *Las siguientes reglas caracterizan los comandos del lenguaje imperativo simple:*

Skip:

$$\vdash \{p\}\ \textbf{skip}\ \{p\}$$

Asignación:

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

Secuenciación:

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

Alternativa:

$$\frac{\vdash \{p \wedge b\}\ c_1\ \{q\} \qquad \vdash \{p \wedge \neg b\}\ c_2\ \{q\}}{\vdash \{p\}\ \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \textbf{fi}\ \{q\}}$$

Ciclo:

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

donde $p, q, r \in Form$, $x \in Var$, $e \in Expr$, $b \in Bool$, $c, c_1, c_2 \in Comm$. Con $f_{/x\leftarrow e}$ denotamos la sustitución sintáctica de las ocurrencias libres de la variable x en f (fórmula, expresión, comando, etc.) por la expresión e.[4] *En la regla de ciclo, la fórmula p se denomina* invariante del ciclo.

La mayor parte de las reglas estructurales de la lógica de Hoare también son válidas en nuestro marco. Una excepción es la llamada regla de constancia que no es válida en general por los potenciales alias que introduce el operador $\wedge$,[5] aunque podemos formular una versión particular para fórmulas *pure*. Para ello es necesario caracterizar las variables que son *modificadas* por un comando.

[4]Notar que no toda sustitución sobre una fórmula/expresión/comando da lugar a una fórmula/expresión/comando bien formado. Esto es más evidente en los comandos, pero también sucede con las fórmulas, ya que en los términos $x \mapsto e$, x sólo puede ser una variable. A lo largo del trabajo, cuando escribimos una sustitución asumimos que tiene como resultado una fórmula/expresión/comando bien formado de acuerdo a las gramáticas respectivas.

[5]Discutimos en mayor detalle esta dificultad en la introducción del capítulo 4

Definición 23 (Variables modificadas de SL^H). *Sea $c \in Comm$, el conjunto de variables modificadas por c se define a través de una función $Mod \in Comm \to Var$, dada por las siguientes ecuaciones:*

$$
\begin{aligned}
Mod.(x := e) &\doteq \{x\} \\
Mod.(\mathbf{skip}) &\doteq \emptyset \\
Mod.(x := \mathbf{cons}(\vec{e})) &\doteq \{x\} \\
Mod.(x.i := e) &\doteq \emptyset \\
Mod.(x := y.i) &\doteq \{x\} \\
Mod.(\mathbf{dispose}(x)) &\doteq \emptyset \\
Mod.(c_1; c_2) &\doteq Mod.c_1 \cup Mod.c_2 \\
Mod.(\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{fi}) &\doteq Mod.c_1 \cup Mod.c_2 \\
Mod.(\mathbf{while}\ b\ \mathbf{do}\ c\ \mathbf{od}) &\doteq Mod.c
\end{aligned}
$$

donde $x, y \in Var$, $e \in Expr$, $\vec{e} \in Expr^+$, $i \in \mathbb{N}_0$, $b \in Bool$, y $c, c_1, c_2 \in Comm$

Definición 24 (Sistema deductivo de SL^H (cont.)). *Las siguientes reglas caracterizan el razonamiento estructural sobre especificaciones:*

Consecuencia:

$$
\frac{\vDash p' \Rightarrow p \qquad \vdash \{\, p \,\}\ c\ \{\, q \,\} \qquad \vDash q \Rightarrow q'}{\vdash \{\, p' \,\}\ c\ \{\, q' \,\}}
$$

Elim. de variables auxiliares:

$$
\frac{\vdash \{\, p \,\}\ c\ \{\, q \,\} \qquad x \notin FV.c}{\vdash \{\, \exists x \cdot p \,\}\ c\ \{\, \exists x \cdot q \,\}}
$$

Sustitución:

$$
\frac{\vdash \{\, p \,\}\ c\ \{\, q \,\} \qquad x' \notin FV.p \cup FV.c \cup FV.q}{\vdash \{\, p \,\}\ c\ \{\, q \,\}_{/x \leftarrow x'}}
$$

Constancia pura:

$$
\frac{\begin{array}{cc} \vdash \{\, p \,\}\ c\ \{\, q \,\} & r \in Pure \\ \{\vec{x}\} = Mod.c & \vec{x}' \notin FV.c \cup FV.p \cup FV.q \cup FV.r \end{array}}{\vdash \{\, p \wedge r \,\}\ c\ \{\, \exists \vec{x}' \cdot q \wedge r_{/\vec{x} \leftarrow \vec{x}'} \,\}}
$$

donde $p, p', q, q', r \in Form$, $c \in Comm$, $x, x' \in Var$ y abusando de la notación utilizamos $FV.c$ para denotar las variables que ocurren en c.

Especificaciones locales

Los comandos de manipulación de la memoria dinámica pueden caracterizarse a través de especificaciones *locales*, que involucran únicamente las variables y las porciones del *heap* que son construidas, accedidas o eliminadas por el comando. Esta caracterización mínima de un comando c se denomina su *footprint*.

Definición 25 (Sistema deductivo de SL^H (cont.)). *Las siguientes reglas caracterizan los comandos de manipulación del* heap*:*

Construcción (local):

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

Mutación (local):

$$\frac{0 \leqslant i < n}{\vdash \{\, x \mapsto (e_0, \dots, e_n) \,\}\ x.i := e\ \{\, x \mapsto (e_0, \dots, e_n)[i \mapsto e] \,\}}$$

Consulta (local):

$$\frac{0 \leqslant i \leqslant n}{\vdash \{\, y \mapsto (e_0, \dots, e_n) \,\}\ x := y.i\ \{\, \exists x' \cdot (y \mapsto (e_0, \dots, e_n))_{/x \leftarrow x'} \wedge x = e_{i_{/x \leftarrow x'}} \,\}}$$

Destrucción (local):

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

donde $x, x', y \in \mathit{Var}$, *$x$ es distinta a x'*, $e, e_0, \dots, e_n \in \mathit{Expr}$, $\vec{e} \in \mathit{Expr}^+$, $i, n \in \mathbb{N}_0$.

En la lógica de Hoare usual, se asume que las pre y postcondiciones establecen *positivamente* los cambios en la memoria, permitiendo modificaciones adicionales sobre las porciones del estado que no se mencionan. La SL^H invierte esta situación, restringiendo los cambios que pueden realizarse a aquellos mencionados explícitamente. Una especificación válida $\{\, p \,\}\ c\ \{\, q \,\}$ es *ajustada* en el sentido que todo registro en el *footprint* debe estar mencionado explícitamente en p o construido por c. Esto impone la necesidad de contar con reglas de inferencia específicas para dar cuenta de las porciones de memoria que permanecen invariantes.

Regla de frame *y especificaciones globales*

Las especificaciones locales introducidas más arriba, sólo tienen sentido si podemos extender su aplicación a un contexto más general. La regla de *frame* permite trasladar el razonamiento sobre un comando c enfocado en su *footprint* a estados del programa más amplios que lo contienen, agregando a la especificación predicados arbitrarios sobre las variables y partes del *heap* que no son modificadas por c.

Definición 26 (Sistema deductivo de SL^H (cont.)).

Regla de *frame*:

$$\frac{\vdash \{p\}\ c\ \{q\} \qquad FV.r \cap Mod.c = \varnothing}{\vdash \{p * r\}\ c\ \{q * r\}}$$

donde $p, q, r \in Form$, y $c \in Comm$.

La regla de *frame* permite que el *razonamiento local* dado por las especificaciones locales sea útil, y en la mayor parte de los casos, suficiente [116]:

> To understand how a program works, it should be possible for reasoning and especification to be confined to the cells that the program actually accesses. The value of any other cell will automatically remain unchanged.

En [152, 155] se muestra que la regla de *frame* es completa en el siguiente sentido: si todo lo que se sabe sobre un comando c está dado por una especificación válida $\{p\}\ c\ \{q\}$, entonces cualquier otra especificación $\{p'\}\ c\ \{q'\}$, cuya validez es una consecuencia semántica de la primera, puede derivarse a partir de ella con la regla de *frame* y las reglas estructurales (def. 24).

La condición sobre la *no interferencia* entre las variables modificadas por el comando y las variables libres de la fórmula que caracteriza el *frame* es necesaria para su validez, pero no representa un verdadero problema en la práctica.

Ejemplo 25. *Supongamos que a partir de $\{p\}\ c\ \{q\}$ queremos derivar $\{p * r\}\ c\ \{q * r\}$, pero $FV.r \cap Mod.c = \{x\}$. Supongamos además que x' es una variable* fresca, *esto es $x' \notin FV.p \cup FV.q \cup FV.r \cup FV.c$. Con el uso de la regla de constancia pura, podemos derivar $\{p \wedge x = x'\}\ c\ \{\exists x'' \cdot q \wedge x'' = x'\}$. Haciendo un reemplazo sintáctico en r, podemos utilizar la regla de* frame, *derivando*

$$\{(p \wedge x = x') * r_{/x \leftarrow x'}\}\ c\ \{\exists x'' \cdot (q \wedge x'' = x') * r_{/x \leftarrow x'}\}$$

Como $x = x'$ es pure, *la precondición equivale a $(p * r) \wedge x = x'$, y la postcondición implica $\exists x'' \cdot q * r_{/x \leftarrow x''}$. Así utilizando la regla de consecuencia, y la regla de eliminación de variables auxiliares y un cálculo elemental para eliminar $x = x'$ de la precondición, obtenemos una versión de la regla de* frame interferente*:*

$$\frac{\{p\}\ c\ \{q\}}{\{p * r\}\ c\ \{\exists x'' \cdot q * r_{/x \leftarrow x''}\}}$$

Con el uso de la regla de *frame* podemos dar especificaciones globales para cada uno de los comandos de manipulación del *heap* a partir de las reglas de especificación locales (def. 25).

Definición 27 (Sistema deductivo de SL^H (cont.)). *Las siguientes reglas caracterizan los comandos de manipulación del* heap*:*

Construcción (global):

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

Mutación (global):

$$\frac{0 \leqslant i < n}{\vdash \{\, x \mapsto (e_0, \dots, e_n) \,*\, p \,\}\; x.i := e\; \{\, x \mapsto (e_0, \dots, e_n)[i \mapsto e] \,*\, p \,\}}$$

Consulta (global):

$$\frac{0 \leqslant i \leqslant n}{\vdash \{\, y \mapsto (e_0, \dots, e_n) \,*\, p \,\}\; x := y.i\; \{\, \exists x' \cdot (y \mapsto (e_0, \dots, e_n) \,*\, p)_{/x \leftarrow x'} \wedge x = e_{i_{/x \leftarrow x'}} \,\}}$$

Destrucción (global y WP):

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

donde $x, x', y \in Var$, *x es distinta a* x', $e, e_0, \dots, e_n \in Expr$, $\vec{e} \in Expr^+$, $i, n \in \mathbb{N}_0$.

Ejemplo 26. *Demostremos un programa que construye un modelo como el del ejemplo 16 (pág. 45). La terna* $\vdash \{\, \mathbf{emp} \,\}\; x := \mathbf{cons}(\mathbf{null})\; \{\, x \mapsto \mathbf{null} \,\}$ *se deriva de la regla de especificación local para el comando de construcción, y la regla de consecuencia, simplificando el* $\exists$ *trivial. Más aún, por las reglas de consecuencia y de elemento neutro es válido*

$$\vdash \{\, \mathbf{emp} \,\}\; x := \mathbf{cons}(\mathbf{null})\; \{\, x \mapsto \mathbf{null} \,*\, \mathbf{emp} \,\}$$

Por un razonamiento análogo vale $\vdash \{\, \mathbf{emp} \,\}\; y := \mathbf{cons}(x)\; \{\, y \mapsto x \,\}$ *y aplicando la regla de* frame *obtenemos*

$$\vdash \{\, x \mapsto \mathbf{null} \,*\, \mathbf{emp} \,\}\; y := \mathbf{cons}(x)\; \{\, x \mapsto \mathbf{null} \,*\, y \mapsto x \,\}$$

Además tenemos que $\vdash \{\, x \mapsto \mathbf{null} \,\}\; x.0 := y\; \{\, x \mapsto y \,\}$ *por la especificación local del comando de mutación. Nuevamente con la regla de* frame *obtenemos*

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

Finalmente, aplicando la regla de de secuenciación, obtenemos

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

Tomando p en las especificaciones globales apropiadamente, de la forma $x \mapsto \vec{e} \mathrel{-\!\!*} p$ es posible derivar reglas para *backward reasoning* que conforman las precondiciones más débiles [85, 152], a excepción del comando de destrucción cuya regla global ya es suficiente.

Definición 28 (Sistema deductivo de SL^H (cont.)). *Las siguientes reglas caracterizan el* backward reasoning *sobre los comandos de manipulación del* heap*:*

Construcción (WP):

$$\frac{x' \notin FV.\vec{e} \cup FV.p}{\vdash \{\, \forall x' \cdot x' \mapsto \vec{e} \mathrel{-\!\!*} p_{/x \leftarrow x'} \,\}\ x := \mathbf{cons}(\vec{e})\ \{\, p \,\}}$$

Mutación (WP):

$$\frac{0 \leqslant i < n}{\vdash \{\, x \mapsto (e_0, \ldots, e_n) * (x \mapsto (e_0, \ldots, e_n)[i \mapsto e] \mathrel{-\!\!*} p) \,\}\ x.i := e\ \{\, p \,\}}$$

Consulta (WP):

$$\frac{0 \leqslant i \leqslant n \qquad\qquad x' \notin FV.p}{\vdash \{\, \exists x' \cdot y \mapsto \vec{e}[i \mapsto x'] * (y \mapsto \vec{e}[i \mapsto x'] \mathrel{-\!\!*} p_{/x \leftarrow x'}) \,\}\ x := y.i\ \{\, p \,\}}$$

donde $x, x', y \in Var$, x, y son distintas a x', $e, e_1, \ldots, e_n \in Expr$, $\vec{e} \in Expr^+$, $i, n \in \mathbb{N}_0$.

Teorema 1 (Consistencia del sistema deductivo de SL^H). *Dados $p, q \in Form$, $c \in Comm$, si $\vdash \{\, p \,\}\ c\ \{\, q \,\}$ se deduce utilizando las reglas del sistema deductivo,*[6] *entonces* $\models \{\, p \,\}\ c\ \{\, q \,\}$.

3.5. Estructuras de datos

Los punteros son utilizados principalmente para definir estructuras de datos enlazadas que puedan mutar, tanto en los valores que contienen como en la cantidad de registros que las conforman. Las estructuras de datos suelen utilizarse para representar tipos de datos *abstractos* que no tienen soporte nativo en el lenguaje de programación. Para especificar tales estructuras utilizando *Separation Logic* es necesario introducir predicados recursivos que vinculen los valores abstractos (usualmente definidos algebraicamente), con el *patrón* de registros de memoria dinámica utilizados para su representación. En esta sección introducimos algunos predicados para caracterizar estructuras de datos comunes como distintas variedades de listas enlazadas y árboles binarios. Ya que la incorporación de valores abstractos en una lógica de Hoare es mayormente estándar, presentamos una tratamiento informal de los mismos. Para un presentación completa formal referimos a [92].

[6] Definiciones 22, 24, 25, 26, 27, 28.

Listas enlazadas

Las listas enlazadas son una de las estructuras de datos mutables más simples y más utilizadas, que sirven para representar secuencias abstractas de valores, definidas inductivamente. Podemos incorporar a nuestro lenguaje expresiones sobre listas *List* que hacen referencia a secuencias abstractas de valores, que denotamos $[Values]$.

Definición 29 (Expresiones sobre listas)**.** *El conjunto List de expresiones sobre listas se define por la siguiente gramática:*

$$List \ni es ::= [\,] \mid e \triangleright es \mid$$

donde $e \in Expr$. *Con* $[e_1, e_2, \ldots, e_n]$ *denotamos la expresión* $e_1 \triangleright e_2 \triangleright \ldots \triangleright e_n \triangleright [\,]$.

Las expresiones $[\,]$ y $e \triangleright es$ refieren a los constructores inductivos de secuencias abstractas; las restantes a las *operaciones* usuales sobre listas: con $es \triangleleft e$ denotamos la operación de insertar el elemento e a la secuencia es como último valor; con $es +\!\!+\ es'$ la secuencia que resulta de concatenar es y es'; con $\#es$ el tamaño de es; $es \downarrow n$ el sufijo de es que resulta de eliminar los primeros n valores; con $es \uparrow n$ el prefijo de es de tamaño n; con $es.n$ el valor de la posición n de es; y con $es[n \mapsto e]$ la secuencia que resulta de reemplazar el valor de la posición n por e. En la figura 3.5 se presenta la semántica pretendida para cada una de estas operaciones. Frecuentemente utilizamos en la especificación de programas otras operaciones sobre expresiones de listas (u otros valores abstractos), presentadas a través de la función que le da semántica. Notar que esto implica una extensión implícita del lenguaje de expresiones.

Las listas enlazadas consisten en una serie de registros encadenados de tal manera que cada uno contiene un puntero al siguiente registro. Para denotar el fin de la lista, el último registro contiene un puntero *nil*. Además, cada registro contiene un campo para almacenar los valores que conforman la secuencia representada.

Definición 30 (Lista enlazada)**.** *El siguiente predicado recursivo caracteriza las listas enlazadas que comienzan en x y terminan en nil:*

$$\mathbf{list}.x.[\,] \doteq x = \mathbf{null} \wedge \mathbf{emp}$$
$$\mathbf{list}.x.(e \triangleright es) \doteq \exists y \cdot x \mapsto e, y \ast \mathbf{list}.y.es$$

donde $x, y \in Var$, $e \in Expr$, $es \in List$.

Ejemplo 27. *El predicado* $\mathbf{list}.x.[1, 2, 3]$ *caracteriza una lista enlazada de tres elementos. Sus modelos son como los de la figura 3.6.*

Con este predicado resulta sencillo verificar algunos programas elementales sobre listas enlazadas.

$$
\begin{aligned}
\lhd &\in \mathit{Values} \to [\mathit{Values}] \to [\mathit{Values}] \\
[\,] \lhd v' &\doteq [v'] \\
(v \rhd vs) \lhd v' &\doteq v \rhd (vs \lhd v') \\
\\
+\!\!\!+ &\in [\mathit{Values}] \to [\mathit{Values}] \to [\mathit{Values}] \\
[\,] +\!\!\!+\ vs' &\doteq vs' \\
(v \rhd vs) +\!\!\!+\ vs' &\doteq v \rhd (vs +\!\!\!+\ vs') \\
\\
\# &\in [\mathit{Values}] \to \mathbb{N}_0 \\
\#[\,] &\doteq 0 \\
\#(v \rhd vs) &\doteq 1 + \#vs \\
\\
\downarrow &\in [\mathit{Values}] \to \mathbb{N}_0 \to [\mathit{Values}] \\
[\,] \downarrow n &\doteq [\,] \\
vs \downarrow 0 &\doteq vs \\
(v \rhd vs) \downarrow (n+1) &\doteq vs \downarrow n \\
\\
\uparrow &\in [\mathit{Values}] \to \mathbb{N}_0 \to [\mathit{Values}] \\
[\,] \uparrow n &\doteq [\,] \\
vs \uparrow 0 &\doteq [\,] \\
(v \rhd vs) \uparrow (n+1) &\doteq v \rhd (vs \uparrow n) \\
\\
. &\in [\mathit{Values}] \rightharpoonup \mathbb{N}_0 \to \mathit{Values} \\
(v \rhd vs).0 &\doteq v \\
(v \rhd vs).(n+1) &\doteq vs.n \\
\\
[\ \mapsto\ ] &\in [\mathit{Values}] \to \mathbb{N}_0 \to \mathit{Values} \to [\mathit{Values}] \\
[\,][n \mapsto v'] &\doteq [\,] \\
(v \rhd vs)[0 \mapsto v'] &\doteq v' \rhd vs \\
(v \rhd vs)[n+1 \mapsto v'] &\doteq v \rhd vs[n \mapsto v']
\end{aligned}
$$

Figura 3.5: Semántica de operaciones sobre listas.

Ejemplo 28. *Consideremos el algoritmo que inserta un valor e como primer elemento de una lista enlazada es. Tal programa puede especificarse, y demostrarse como se muestra en el siguiente esquema de prueba. En los esquema de prueba, utilizamos fórmulas consecutivas para la aplicación de la regla de consecuencia, donde la consecuencia lógica se debe a la aplicación de alguna de las reglas de cálculo de SL presentadas a lo largo del capítulo, o reglas del cálculo de la lógica de primer orden. Además aplicamos implícitamente las reglas estructurales (incluida la regla de* frame*) y las reglas de los comandos compuestos.*

$\{\,\mathbf{list}.x.es\,\}$
 $y := \mathbf{cons}(e, x);$
$\{\, y \mapsto e, x \;*\; \mathbf{list}.x.es \,\}$
$\Rightarrow \langle$ *introducción de* $\exists$ *sobre x* $\rangle$
$\{\, \exists x \cdot y \mapsto e, x \;*\; \mathbf{list}.x.es \,\}$
$\Rightarrow \langle$ *definición de* **list** $\rangle$
$\{\,\mathbf{list}.y.(e \triangleright es)\,\}$
 $x := y$
$\{\,\mathbf{list}.x.(e \triangleright es)\,\}$

Ejemplo 29. *Demostremos ahora el algoritmo que destruye una lista enlazada es dada, donde utilizamos las propiedades* $\mathbf{list}.x.es \wedge x \neq \mathbf{null} \Rightarrow es \neq [\,]$ *(1) y* $\mathbf{list}.x.es \wedge x = \mathbf{null} \Rightarrow \mathbf{emp}$ *(2), que se derivan fácilmente de la definición recursiva de* **list***.*

$\{\,\mathbf{list}.x.es\,\}$
$\Rightarrow \langle$ *introducción de* $\exists$ *sobre es* $\rangle$
$\{\, \exists es \cdot \mathbf{list}.x.es \,\}$
 while $x \neq \mathbf{null}$ **do**
  $\{\, \exists es \cdot \mathbf{list}.x.es \wedge x \neq \mathbf{null} \,\}$
  $\Rightarrow \langle$ *propiedad 1* $\rangle$
  $\{\, \exists es \cdot \mathbf{list}.x.es \wedge x \neq \mathbf{null} \wedge es \neq [\,] \,\}$
  $\Rightarrow \langle$ *caso inductivo de List* $\rangle$
  $\{\, \exists e, es \cdot \mathbf{list}.x.(e \triangleright es) \,\}$
  $\Rightarrow \langle$ *definición de* **list** $\rangle$
  $\{\, \exists e, es, y' \cdot x \mapsto e, y' \;*\; \mathbf{list}.y'.es \,\}$
   $y := x;$
  $\{\, \exists e, es, y' \cdot y \mapsto e, y' \;*\; \mathbf{list}.y'.es \,\}$
   $x := y.1;$
  $\{\, \exists e, es \cdot y \mapsto e, x \;*\; \mathbf{list}.x.es \,\}$
   $\mathbf{dispose}(y)$
  $\{\, \exists es \cdot \mathbf{list}.x.es \,\}$
 od
$\{\, \exists es \cdot \mathbf{list}.x.es \wedge x = \mathbf{null} \,\}$
$\Rightarrow \langle$ *propiedad 2* $\rangle$
$\{\,\mathbf{emp}\,\}$

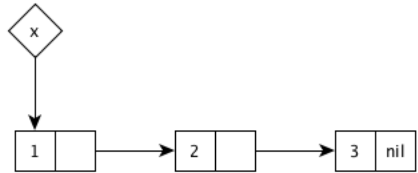


Figura 3.6: Modelo de $\mathbf{list}.x.[1,2,3]$

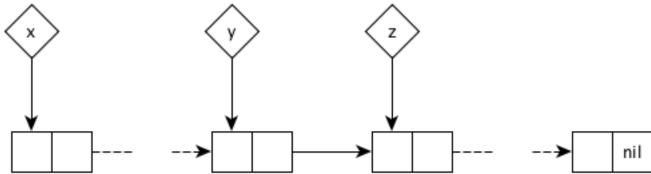


Figura 3.7: Estado de la memoria en plena ejecución del algoritmo de búsqueda.

La utilidad del predicado **list** encuentra rápidamente su límite en la verificación de programas iterativos más complejos. Consideremos el algoritmo que busca un valor dado en una lista enlazada, cuya especificación se muestra en la figura 3.8. Pensemos en una ejecución que se encuentra al comienzo de una iteración arbitraria del ciclo principal. En tal momento, la estructura de datos en memoria y los punteros utilizados para su recorrido se encuentran en un estado como el representado esquemáticamente en la figura 3.7. Si bien el predicado $\mathbf{list}.x.es$ es un invariante del ciclo, tal especificación no es útil para demostrar la validez de las manipulaciones al *heap* a través de los punteros y y z. Para caracterizar el estado como términos disjuntos compuestos con $*$, es necesario disponer de un predicado que especifique la estructura *parcial* que ocurre durante el recorrido iterativo, esto es, ya no una lista enlazada terminada en *nil*, sino un segmento de lista terminado en un puntero arbitrario.

Definición 31 (Segmento)**.** *El siguiente predicado recursivo caracteriza los segmentos enlazados que comienzan en x y terminan en y:*

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

donde $x, y, z \in Var$, $e \in Expr$, $es \in List$.

Ejemplo 30. *El predicado* $\mathbf{lseg}.x.y.[1,2,3]$ *caracteriza un segmento de tres elementos. Sus modelos son como los de la figura 3.9 (cuando y es un puntero). Notar que y es* dangling*. Comparando con el ejemplo 24 (pág. 50) es fácil de ver que esta fórmula es* precise.

$$
\begin{array}{l}
\{\, \textbf{list}.x.(e \triangleright es) \,\} \\
\quad y := x; \\
\quad w := x.0; \\
\quad z := x.1; \\
\quad \textbf{while}\ z \neq \textbf{null} \wedge w \neq v\ \textbf{do} \\
\qquad y := z; \\
\qquad w := y.0; \\
\qquad z := y.1 \\
\quad \textbf{od}; \\
\quad r := w = v \\
\{\, \textbf{list}.x.(e \triangleright es) \wedge r = exists.v.(e \triangleright es) \,\}
\end{array}
$$

$$
\begin{aligned}
& exists \in Values \to [Values] \to \{true, false\} \\
exists.v.[\,] &\doteq false \\
exists.v.(v' \triangleright vs) &\doteq (v = v') \vee exists.v.vs
\end{aligned}
$$

Figura 3.8: Algoritmo de búsqueda dentro de una lista.

Ejemplo 31. *El estado de la memoria de la figura 3.7 puede caracterizarse a través de la fórmula*

$$\exists es', es'' \cdot (\textbf{lseg}.x.y.es' * y \mapsto w, z * \textbf{list}.z.es'') \wedge es = es' + [w] + es''$$

Ejemplo 32. *Una fórmula como* **lseg**.$x.y.es$*, para* es *arbitraria, nunca se satisface con un segmento de lista circular, dado que el caso recursivo de* **lseg** *requiere que* $x \neq y$*. Por lo tanto, esta fórmula nunca admite modelos como los de la figura 3.10.*

Es posible relajar la definición de **lseg** para caracterizar segmentos de listas circulares o con un lazo final.

Definición 32 (Segmento con lazo)**.** *El siguiente predicado recursivo caracteriza los segmentos enlazados que comienzan en* x*, terminan en* y*, donde* y *puede referenciar a un elemento dentro del segmento:*

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

donde $x, y, z \in Var$*,* $e \in Expr$*,* $es \in List$*.*

Ejemplo 33. *La fórmula* **clseg**.$x.y.es$*, para* es *arbitraria, admite modelos como los de la figura 3.10.*

Ejemplo 34. *El modelo de la figura 3.11 satisface la fórmula* **clseg**.$x.x.[1, 2, 3]$*, y por lo tanto también* $\exists es \cdot$ **clseg**.$x.x.es$*. Pero* $\exists es \cdot$ **clseg**.$x.x.es$ *se satisface trivialmente en el* heap *vacío, por lo tanto se deduce que tal fórmula no puede ser* precise*.*

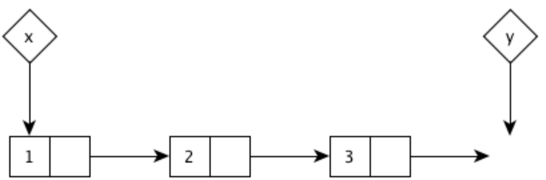


Figura 3.9: Modelo de **lseg**.x.y.[1, 2, 3]

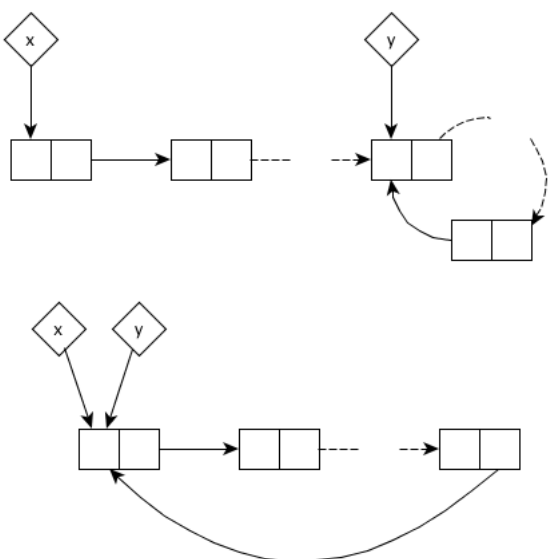


Figura 3.10: Modelos de segmentos con circularidad.

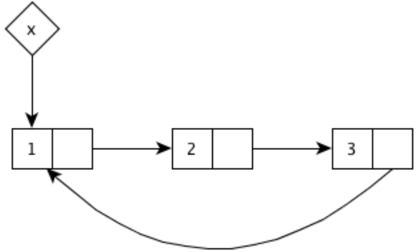


Figura 3.11: Modelo de **clseg**.x.x.[1, 2, 3].

Resulta sencillo demostrar algunas propiedades por inducción estructural.

Lema 10 (Reglas de cálculo de SL (cont.)). *Las siguientes son propiedades de los predicados* **list**, **lseg** *y* **clseg***:*

1. $\models \mathbf{list}.x.es \Leftrightarrow \exists y \cdot \mathbf{lseg}.x.y.es \wedge y = \mathbf{null}$
2. $\models \mathbf{list}.x.es \Leftrightarrow \exists y \cdot \mathbf{clseg}.x.y.es \wedge y = \mathbf{null}$
3. $\models \mathbf{lseg}.x.y.es \Rightarrow \mathbf{clseg}.x.y.es$
4. $\models \mathbf{clseg}.x.y.es * y \mapsto e, z \Leftrightarrow \mathbf{lseg}.x.y.es * y \mapsto e, z$
5. $\models \mathbf{lseg}.x.y.(es \lhd e) \Leftrightarrow \exists z \cdot \mathbf{lseg}.x.z.es * z \mapsto e, y$
6. $\models \mathbf{lseg}.x.y.(es + es') \Leftrightarrow \exists z \cdot \mathbf{lseg}.x.z.es * \mathbf{lseg}.z.y.es'$
7. $\models \mathbf{clseg}.x.y.(es + es') \Leftrightarrow \exists z \cdot \mathbf{clseg}.x.z.es * \mathbf{clseg}.z.y.es'$
8. $\models \mathbf{clseg}.x.y.es \wedge x = \mathbf{null} \Rightarrow y = \mathbf{null} \wedge \mathbf{emp} \wedge es = [\,]$
9. $\models \mathbf{clseg}.x.y.es \wedge x \neq \mathbf{null} \Rightarrow es \neq [\,]$
10. $\models \mathbf{clseg}.x.y.es \wedge x \neq y \Rightarrow es \neq [\,]$
11. $\exists es \cdot \mathbf{lseg}.x.y.es \in Precise$
12. $\mathbf{clseg}.x.y.es \in Precise$, *pero* $\exists es \cdot \mathbf{clseg}.x.y.es \notin Precise$.

donde $x, y, z, \in Var$, $e \in Expr$ *y* $es, es' \in List$.

Ejemplo 35. *Demostremos la corrección del programa especificado en la figura 3.8. Veamos la corrección de la porción inicial del código, cuya última aserción es el invariante:*

$\{\, \mathbf{list}.x.(e \rhd es) \,\}$
$\quad y := x;$
$\{\, \mathbf{list}.x.(e \rhd es) \wedge x = y \,\}$
$\Rightarrow \langle$ *elemento neutro* $\rangle$
$\{\, (\mathbf{list}.y.(e \rhd es) * \mathbf{emp}) \wedge x = y \,\}$
$\Rightarrow \langle$ *distributividad de* $*$ *con* $\wedge$, $x = y \in Pure$ $\rangle$
$\{\, \mathbf{list}.y.(e \rhd es) * (\mathbf{emp} \wedge x = y) \,\}$
$\Rightarrow \langle$ *definición de* **lseg** $\rangle$
$\{\, \mathbf{list}.y.(e \rhd es) * \mathbf{lseg}.x.y.[\,] \,\}$
$\Rightarrow \langle$ *conmutatividad, definición de* **list** $\rangle$
$\{\, \mathbf{lseg}.x.y.[\,] * \exists z \cdot y \mapsto e, z * \mathbf{list}.z.es \,\}$
$\quad w := x.0;$
$\{\, (\mathbf{lseg}.x.y.[\,] * \exists z \cdot y \mapsto w, z * \mathbf{list}.z.es) \wedge w = e \,\}$
$\quad z := x.1;$
$\{\, (\mathbf{lseg}.x.y.[\,] * y \mapsto w, z * \mathbf{list}.z.es) \wedge w = e \,\}$
$\Rightarrow \langle$ *introducción de* $\exists$ *sobre* $[\,]$ *y es, cálculo sobre secuencias abstractas* $\rangle$
$\{\, \exists es', es'' \cdot (\mathbf{lseg}.x.y.es' * y \mapsto w, z * \mathbf{list}.z.es'') \wedge e \rhd es = es' + [w] + es''$
$\wedge\, exists.v.(e \rhd es) = exists.v.(w \rhd es'') \,\}$

*El último paso de la regla de consecuencia se valida por un cálculo elemental sobre la función exists. La demostración del cuerpo del ciclo es similar ya que el código es completamente análogo. Utilizamos la condición $z \neq$ **null** para hacer el* unfolding *de la definición de* **list** *y $w \neq v$ para lo respectivo con exists. Utilizamos la propiedad (5) del lema anterior para combinar el segmento inicial con el registro examinado en el cuerpo del ciclo.*

$$
\begin{array}{l}
\{\, \exists es', es'' \cdot (\mathbf{lseg}.x.y.es' \ * \ y \mapsto w, z \ * \ \mathbf{list}.z.es'') \\
\wedge\ e \triangleright es = es' + [w] + es'' \wedge exists.v.(e \triangleright es) = exists.v.(w \triangleright es'') \\
\wedge\ z \neq \mathbf{null} \wedge w \neq v \,\} \\
\Rightarrow \langle \textit{ definición de exists con } w \neq v \ \rangle \\
\{\, \exists es', es'' \cdot (\mathbf{lseg}.x.y.es' \ * \ y \mapsto w, z \ * \ \mathbf{list}.z.es'') \\
\wedge\ e \triangleright es = es' + [w] + es'' \wedge ins.v.(e \triangleright es) = ins.v.es'' \wedge z \neq \mathbf{null} \,\} \\
\Rightarrow \langle \textit{ propiedad 5 del lema 10 } \rangle \\
\{\, \exists es', es'' \cdot (\mathbf{lseg}.x.z.(es' \triangleleft w) \ * \ \mathbf{list}.z.es'') \wedge e \triangleright es \ = \ es' + [w] + es'' \ \wedge \\
exists.v.(e \triangleright es) = exists.v.es'' \wedge z \neq \mathbf{null} \,\} \\
\Rightarrow \langle \textit{ introducción de } \exists \textit{ sobre } es' \triangleleft w\textit{, cálculo sobre secuencias abstractas } \rangle \\
\{\, \exists es', es'' \cdot (\mathbf{lseg}.x.z.es' \ * \ \mathbf{list}.z.es'') \wedge e \triangleright es = es' + es'' \ \wedge \\
exists.v.(e \triangleright es) = exists.v.es'' \wedge z \neq \mathbf{null} \,\} \\
\quad y := z; \\
\{\, \exists es', es'' \cdot (\mathbf{lseg}.x.y.es' \ * \ \mathbf{list}.y.es'') \wedge e \triangleright es = es' + es'' \ \wedge \\
exists.v.(e \triangleright es) = exists.v.es'' \wedge y \neq \mathbf{null} \,\} \\
\Rightarrow \langle \textit{ definición de } \mathbf{list} \ \rangle \\
\{\, (\exists es', e', es'', z \cdot (\mathbf{lseg}.x.y.es' \ * \ y \mapsto e', z \ * \ \mathbf{list}.z.es'') \ \wedge \\
e \triangleright es = es' + [e'] + es'' \wedge exists.v.(e \triangleright es) = exists.v.(e' \triangleright es'') \,\} \\
\quad w := y.0; \\
\{\, \exists es', es'', z \cdot (\mathbf{lseg}.x.y.es' \ * \ y \mapsto w, z \ * \ \mathbf{list}.z.es'') \ \wedge \\
e \triangleright es = es' + [w] + es'' \wedge exists.v.(e \triangleright es) = exists.v.(w \triangleright es'') \,\} \\
\quad z := y.1; \\
\{\, \exists es', es'' \cdot (\mathbf{lseg}.x.y.es' \ * \ y \mapsto w, z \ * \ \mathbf{list}.z.es'') \\
\wedge\ e \triangleright es = es' + [w] + es'' \wedge exists.v.(e \triangleright es) = exists.v.(w \triangleright es'') \,\}
\end{array}
$$

Finalmente, es fácil que ver que el invariante y la negación de la guarda es precondición suficiente para el último comando. Por una parte, la fórmula

$$\exists es', es'' \cdot (\mathbf{lseg}.x.y.es' \ * \ y \mapsto w, z \ * \ \mathbf{list}.z.es'') \wedge e \triangleright es = es' + [w] + es''$$

implica $\mathbf{list}.x.(e \triangleright es)$ *por las propiedades 1, 5 y 6 del lema anterior y operatoria elemental sobre secuencias abstractas. Por otro lado, si el ciclo termina por $z =$* **null**, *entonces $es'' = [\,]$ y es evidente que $exists.v.(e \triangleright es) \Leftrightarrow exists.v.[w]$. Luego $r \Leftrightarrow w = v$. Si el ciclo termina por $w = v$, entonces directamente vale $r \Leftrightarrow w = v$.*

Arboles binarios y dags

Es sencillo imaginar cómo utilizar ideas análogas a **list** para caracterizar estructuras no lineales. Otra de las estructuras de datos más utilizadas son los árboles binarios (y sus variantes con propiedades de ordenación). A continuación mostramos cómo especificar árboles abstractos de valores, que denotamos $\langle Values \rangle$, a través de expresiones que referencian los constructores inductivos usuales, y cómo representarlos en la memoria dinámica.

Definición 33 (Expresiones sobre árboles binarios)**.** *El conjunto* $Tree$ *de expresiones sobre árboles binarios se define por la siguiente gramática:*

$$Tree \ni et ::= \langle\,\rangle \mid \langle et, e, et \rangle$$

donde $e \in Expr$*. Con* $\langle e \rangle$ *denotamos* $\langle\langle\,\rangle, e, \langle\,\rangle\rangle$*.*

Definición 34 (Arbol binario)**.** *El siguiente predicado recursivo caracteriza los árboles binarios que comienzan en* x*:*

$$\begin{aligned}\mathbf{btree}.x.\langle\,\rangle &\doteq x = \mathbf{null} \wedge \mathbf{emp}\\ \mathbf{btree}.x.\langle lt, e, rt \rangle &\doteq \exists l, r \cdot x \mapsto l, e, r * \mathbf{btree}.l.lt * \mathbf{btree}.r.rt\end{aligned}$$

donde $x, l, r \in Var$*,* $e \in Expr$*,* $lt, rt \in Tree$*.*

Ejemplo 36. *En la figura 3.12 se muestra un esquema de los modelos de la fórmula* $\mathbf{btree}.x.\langle\langle 2 \rangle, 1, \langle\langle\langle 6 \rangle, 4, \langle 5 \rangle\rangle, 3, \langle 5 \rangle\rangle\rangle$*.*

Lema 11 (Reglas de cálculo de SL (cont.))**.** *Las siguientes son reglas son válidas:*

1. $\vDash \mathbf{btree}.x.et \wedge x = \mathbf{null} \Rightarrow \mathbf{emp} \wedge et = \langle\,\rangle$
2. $\vDash \mathbf{btree}.x.et \wedge x \neq \mathbf{null} \Rightarrow et \neq \langle\,\rangle$
3. $\exists et \cdot \mathbf{btree}.x.et \in Precise$

donde $x \in Var$*, y* $et \in Tree$*.*

De forma análoga, es posible caracterizar estructuras de datos que representan árboles abstractos, pero que sus representaciones en memoria conforman gráfos acíclicos dirigidos.

Definición 35 (*Dag*)**.** *El siguiente predicado recursivo caracteriza los grafos acíclicos dirigidos que comienzan en* x*:*

$$\begin{aligned}\mathbf{dag}.x.\langle\,\rangle &\doteq x = \mathbf{null}\\ \mathbf{dag}.x.\langle lt, e, rt \rangle &\doteq \exists l, r \cdot x \mapsto l, e, r * (\mathbf{dag}.l.lt \wedge \mathbf{dag}.r.rt)\end{aligned}$$

donde $x, l, r \in Var$*,* $e \in Expr$*,* $lt, rt \in Tree$*.*

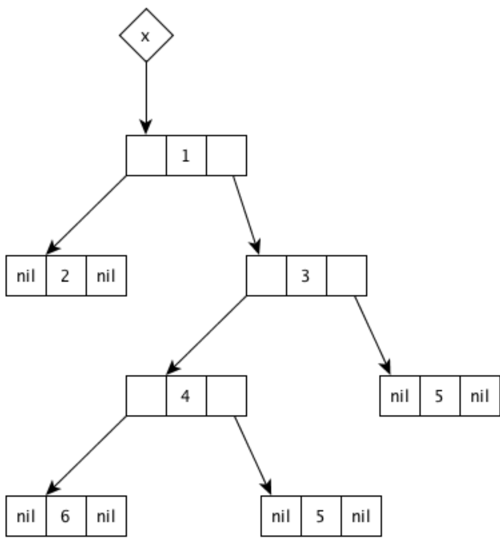


Figura 3.12: Modelo de $\mathbf{btree}.x.\langle\langle 2\rangle, 1, \langle\langle\langle 6\rangle, 4, \langle 5\rangle\rangle, 3, \langle 5\rangle\rangle\rangle$.

Cabe remarcar las diferencias con el predicado **btree**: por un lado no se especifica cómo es el *heap* en el caso de los *dags* vacíos, y por otro se utiliza $\wedge$ en la recursión. En combinación, estas diferencias permiten que un modelo de $\mathbf{dag}.x.\langle lt, e.rt\rangle$ comparta (parte de) su representación para las subestructuras lt y rt, sin obligar a que tales subestructuras sean idénticas. Pero al mismo tiempo, la definición resulta lo suficientemente laxa como para permitir la existencia de registros arbitrarios. El predicado **dag** es *intuicionista* en el sentido dado en la introducción de este capítulo, y por lo tanto no puede ser *precise*.

Lema 12 (Reglas de cálculo de SL (cont.)). *Las siguientes son reglas son válidas:*

1. $\vDash \mathbf{dag}.x.et \wedge x = \mathbf{null} \Rightarrow \mathbf{emp} \wedge et = \langle\ \rangle$
2. $\vDash \mathbf{dag}.x.et \wedge x \neq \mathbf{null} \Rightarrow et \neq \langle\ \rangle$
3. $\vDash \mathbf{dag}.x.et \Leftrightarrow \mathbf{dag}.x.et * \mathbf{true}$, *luego* $\mathbf{dag}.x.et \notin Precise$.

donde $x \in Var$, *y* $et \in Tree$.

Ejemplo 37. *La fórmula* $\mathbf{dag}.x.\langle\langle 2\rangle, 1, \langle\langle\langle 6\rangle, 4, \langle 5\rangle\rangle, 3, \langle 5\rangle\rangle\rangle$, *admite modelos como los de la figura 3.12, pero también aquellos como los de la figura 3.13.*

Los predicados **btree** y **dag** puede utilizarse con algún éxito en la demostración de programas recursivos (algunos ejemplos pueden encontrarse en [136]). Pero, de la misma manera que ocurre con el predicado **list**, no son aptos para la demostración

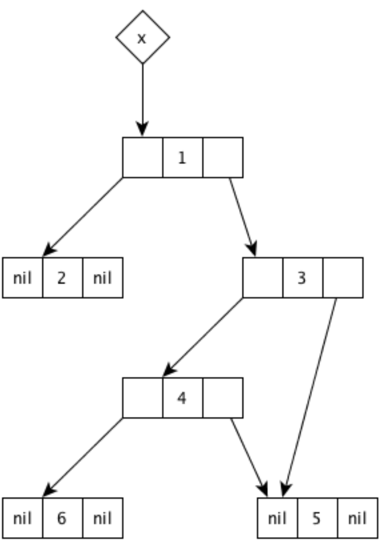


Figura 3.13: Modelo de $\mathbf{dag}.x.\langle\langle 2\rangle, 1, \langle\langle\langle 6\rangle, 4, \langle 5\rangle\rangle, 3, \langle 5\rangle\rangle\rangle$.

de programas iterativos sobre estas estructuras de datos. Aquí también se vuelve necesario disponer de predicados que permitan caracterizar las estructuras parciales que ocurren durante el recorrido de las mismas, análogos a **lseg**. Lamentablemente, no resulta tan sencillo en el caso de las estructuras no lineales, ya que existen múltiples puntos donde podría ser necesario cortar con la recursión. No sólo resulta dificil caracterizar la parcialidad en cuanto a la memoria dinámica, sino también desde el punto de vista de los valores abstractos representados.

En el capítulo 4 presentamos una alternativa que permite demostrar programas iterativos sin necesidad de introducir predicados específicos para dar cuenta de las estructuras parciales. En el capítulo 6 presentamos un predicado general que permite describir árboles y *dags* parciales (también grafos generales), pero que no caracteriza los valores abstractos representados sino únicamente la *forma* de las estructuras en el *heap*. En [48] presentamos un predicado para especificar árboles binarios parciales, caracterizando tanto la estructura en la memoria dinámica, como los datos abstractos, y su utilización para verificar diversos programas iterativos. En [25] se puede encontrar la definición de un predicado para describir *dags* (y grafos) parciales, y su uso en la demostración de algunos programas recursivos.

3.6. Una lógica con tipos abstractos de datos

Los desarrollos originales de la SL^H se enfocan en lenguajes de bajo nivel *a la C*, que no ofrecen ningún soporte para los mecanismos usualmente utilizados para lidiar con la complejidad en la construcción de sistemas grandes: funciones con estado local,

tipos abstractos de datos, clases, etc. En [117] el marco de la SL^H se extiende para dar soporte a módulos *estáticos*, basados en el concepto de *information hiding* de Parnas [123]. Los módulos ocultan los recursos internos utilizados para su representación a los clientes, pero sólo se permite una única instancia de las estructuras de datos ocultas. Así este mecanismo no resulta útil para integrar formas de abstracción como los tipos abstractos de datos, o las clases, donde es usual que se requieran múltiples instancias de los recursos ocultos.

En lo que resta del capítulo, introducimos el concepto de *predicado abstracto* [120, 119], que incorpora la abstracción directamente dentro del marco de trabajo de la *Separation Logic*, dando soporte a tipos abstractos de datos con múltiples instancias y clases, incluyendo las complejidades dadas por los mecanismos de herencia, sobrecarga, resolución de llamadas en tiempo de ejecución, etc. (cuestiones que no tratamos aquí, pero que pueden encontrarse en [121]). Un predicado abstracto tiene un nombre, una definición, y un *scope*, dentro del cual se puede intercambiar libremente el nombre por su definición. Fuera del *scope*, sólo puede utilizarse el nombre del predicado, y el estado puede alterarse utilizando las especificaciones de las funciones que lo manipulan. Los predicados abstractos recuperan la forma recursiva de los predicados que definen estructuras de datos (como los presentados en la sección anterior) pero permiten además encapsular el estado utilizado para la representación.

Utilizando este concepto, introducimos una extensión del lenguaje de fórmulas SL, que denominamos SL_T, y una extensión del sistema deductivo *a la Hoare*, que denominamos SL_T^H, para razonar sobre programas que manipulan múltiples instancias de tipos abstactos de datos.

Fórmulas con predicados abstractos

Para caracterizar los tipos abstractos de datos, extendemos el lenguaje de las fórmulas con predicados atómicos tomados de un conjunto finito de nombres de predicados A, cuyas aridades están dadas por una función $arity \in A \to \mathbb{N}_0$. Además, distinguimos las variables y expresiones de programa respecto a aquellas *lógicas*, que referencian valores *abstractos* como secuencias, árboles, etc., que utilizamos para caracterizar las estructuras de datos.

Definición 36 (Sintaxis de SL_T). *Dados Expr y Rel como en la definición 8, y Var′ un conjunto enumerable de variables* lógicas, *con un elemento distinguido ret, el conjunto de expresiones lógicas Expr′, relaciones lógicas Rel′ y fórmulas Form$_T$ sobre Var y Var′ están dados por la siguiente gramática:*

$$
\begin{aligned}
Expr' \ni e' &::= x' \mid \ldots \\
Rel' \ni r' &::= e' = e' \mid \ldots \\
Form_T \ni p &::= r \mid r' \mid \mathbf{emp} \mid x \mapsto \vec{e} \mid a(\vec{e'}) \mid p \vee p \mid \neg p \mid p * p \mid p \mathrel{-\!\!*} p \mid \forall x'' \cdot p
\end{aligned}
$$

donde $x \in Var$, $x' \in Var'$, $x'' \in Var \cup Var'$, $\vec{e} \in Expr^+$, $\vec{e}' \in (Expr \cup Expr')^n$ *y* $a \in A$ *con* $n = arity.a$.

Como ocurre en la definición de la SL, no damos una sintaxis completa para las expresiones y relaciones lógicas ya que dependen del conjunto de los valores abstractos. Usualmente las expresiones lógicas contienen expresiones sobre listas y árboles, como *List* y *Tree* definidos en la sección anterior. Notar que sólo los predicados atómicos dependen de las expresiones lógicas y por lo tanto, los valores abstractos no pueden ser accedidos desde el programa.

La semántica de las fórmulas está dada ahora por una *forcing relation* respecto a los modelos de la definición 7 en conjunto con un *stack lógico* y un *entorno semántico de predicados*.

Definición 37 (*Stack* lógico y entorno semántico de predicados)**.** *Dado* $Values$ *como en la definición 7, un conjunto de variables lógicas* Var'*, un conjunto de valores abstractos* $Values'$*, un conjunto finito de nombres de predicados* A*, el conjunto de* stacks *lógicos* $Stacks'$ *y el conjunto de entornos semánticos de predicados* $Pred$*, están dados por las siguientes ecuaciones:*

$$
\begin{aligned}
Values' &\doteq Values \cup \ldots \\
Stacks' &\doteq Var' \to Values' \\
Pred &\doteq A \rightharpoonup_{fin} \bigcup_{n \in \mathbb{N}_0} Values'^n \to \mathbb{P}.Heaps
\end{aligned}
$$

donde $\mathbb{P}.A$ *denota el conjunto de partes de* A*.*

Usualmente $Values'$ contiene a $[Values]$, $\langle Values \rangle$, etc. Consideramos únicamente los entornos semánticos de predicados que asignan a cada nombre de predicado una función de la aridad correcta.

Definición 38 (Entorno semántico de predicados bien formado)**.** *Dado* $\alpha \in Pred$*, decimos que está* bien formado *si para todo* $a \in A$, $\alpha.a \in Values'^n \to \mathbb{P}.Heaps$*, donde* $arity.a = n$*.*

Sigue siendo un requisito que la semántica de las expresiones y relaciones sólo dependa del *stack* y, ahora también, el *stack* lógico.

Definición 39 (Semántica de expresiones y relaciones de SL_T)**.** *Dados* $s \in Stacks$ *y* $s' \in Stacks'$ *la semántica de las expresiones y relaciones está dada por funciones:*

$$
\begin{aligned}
[\![\ ]\!]_{s,s'} &\in Expr \cup Expr' \to Values' \\
\{\![\ ]\!]_{s,s'} &\in Rel \cup Rel' \to \{true, false\}
\end{aligned}
$$

que satisfacen las siguientes ecuaciones:

$$\begin{aligned}
[\![x]\!]_{s,s'} &\doteq s.x \\
[\![x']\!]_{s,s'} &\doteq s'.x \\
[\![\mathbf{null}]\!]_{s,s'} &\doteq nil \\
\{\![e_1 = e_2]\!\}_{s,s'} &\doteq [\![e_1]\!]_{s,s'} = [\![e_2]\!]_{s,s'}
\end{aligned}$$

donde $x \in Var$, $x' \in Var'$, *y* $e_1, e_2 \in Expr \cup Expr'$.

La semántica de las fórmulas extiende trivialmente la *forcing relation* de la semántica de SL, teniendo en cuenta el *stack* lógico y el entorno semántico de predicados, y dando cuenta de los nuevos términos.

Definición 40 (Semántica de fórmulas de SL_T). *Dados* $s, h \in States$, $s' \in Stacks'$ *y* $\alpha \in Pred$, *la relación de satisfacción* $\models$ $States \times Stacks' \times Pred \times Form_T$ *se define por inducción en las fórmulas como la relación de la definición 10 (pág. 46) adaptada apropiadamente, y los siguientes casos:*

$$\begin{array}{ll}
s, h, s', \alpha \models r' & \textit{si y sólo si } [\![r']\!]_{s,s'} \\
s, h, s', \alpha \models a(e'_1, \dots, e'_n) & \textit{si y sólo si } a \in dom.\alpha \textit{ y } h \in (\alpha.a).[\![e'_1]\!]_{s,s'} \dots [\![e'_n]\!]_{s,s'}
\end{array}$$

donde $r' \in Rel'$, $a \in A$, $e'_1, \dots, e'_n \in Expr \cup Expr'$, *y* $n = arity.a$.

Lenguaje de programación con funciones

El lenguaje de programación se extiende incorporando variables locales y funciones de primer orden (no recursivas), cuyos nombres son tomados de un conjunto finito de nombres de funciones F, y sus aridades dadas por una función $arity \in F \to \mathbb{N}_0$.

Definición 41 (Comandos de SL_T^H). *El conjunto de comandos* $Comm_T$, *extiende el conjunto Comm presentado en la definición 15 (pág. 51), con los comandos dados por la siguiente gramática:*

$$\begin{array}{lll}
Comm_T \ni c ::= \dots & & \\
& \mid \mathbf{newvar}(x); c & \text{(Variable local)} \\
& \mid \mathbf{return}(e) & \text{(Valor de retorno)} \\
& \mid \mathbf{let}\ f(\vec{x}) = c, \dots, f(\vec{x}) = c\ \mathbf{in}\ c & \text{(Def. de funciones)} \\
& \mid x := f(\vec{e}) & \text{(Llamada a función)}
\end{array}$$

donde $x \in Var$, $e \in Expr$, $\vec{x} \in Var^n$, $\vec{e} \in Expr^n$, $f \in F$ *y* $n = arity.f$.

El comando $\mathbf{newvar}(x); c$ define una nueva variable local en el *scope* del comando c. Asumimos que la variable distinguida *ret* sólo es modificable por el comando

return(e), asignándodole el valor e. El comando **let** $f_1(\vec{x}_1) = c_1 \ldots f_n(\vec{x}_n) = c_n$ **in** c define n funciones f_i con parámetros formales $\vec{x}_i$ y cuerpo de definición c_i en el *scope* del comando c. Finalmente el comando $x := f_i(\vec{e}_i)$ ejecuta el cuerpo de la función f_i con los valores $\vec{e}_i$ como parámetros actuales, y asigna a la variable x el valor de la variable ret. Utilizamos la notación de llamada a *procedimiento* $f(\vec{e})$ como una simple abreviación de $z := f(\vec{e})$ donde z es una variable nunca accedida. Sólo consideramos los programas *bien formados*: aquellos que tienen un comando **return**(e) como último comando en el cuerpo de las funciones, y que sólo definen a través del comando **let** cada nombre de función a lo máximo una vez.

La semántica formal de los nuevos comandos requiere extender el concepto de *configuración* con un *entorno semántico de funciones* que asigna a los nombres de funciones las variables utilizados como parámetros formales y el cuerpo de su definición.

Definición 42 (Configuración de SL_T^H)**.** *Sea*

$$Fun \doteq F \rightharpoonup_{fin} Var^+ \times Comm_T$$

el conjunto de configuraciones $Conf_T$ se define como

$$Conf_T \doteq (Comm_T \times States \times Fun) \cup States \cup \{\mathbf{abort}\}$$

Sea $\zeta \in Conf_T$ una configuración, decimos que es

- no terminal, *cuando $\zeta \in Comm \times States \times Fun$*
- terminal normal, *cuando $\zeta \in States$;*
- terminal anormal, *cuando $\zeta = \mathbf{abort}$.*

Un entorno semántico de funciones se dice bien formado cuando el cuerpo de las funciones sólo accede a los parámetros formales y variables locales, y modifica únicamente las variables locales. Antes utilizamos sin mayores precisiones $FV.c$ para denotar las variables que ocurren en el comando c. Los comandos de variable local y definición de módulo *ligan* nombres de variables lo que obliga a definir $FV.c$ con precisión. Lo mismo ocurre con $Mod.c$.

Definición 43 (Variables libres de comandos de SL_T^H)**.** *Dado $c \in Comm_T$, el conjunto de variables libres de c se define a través de una función $FV \in Comm_T \to Var$, dada por las siguientes ecuaciones:*

$$
\begin{aligned}
FV.(x := e) &\doteq \{x\} \cup FV.e \\
FV.\mathbf{skip} &\doteq \emptyset \\
FV.(x := \mathbf{cons}(\vec{e})) &\doteq \{x\} \cup FV.\vec{e} \\
FV.(x.i := e) &\doteq \{x\} \cup FV.e \\
FV.(x := y.i) &\doteq \{x, y\} \\
FV.(\mathbf{dispose}(x)) &\doteq \{x\} \\
FV.(\mathbf{newvar}(x); c) &\doteq FV.c - \{x\} \\
FV.(\mathbf{return}(e)) &\doteq FV.e \\
FV.(\mathbf{let}\ f_1(\vec{x}_1) = c_1, \dots, f_n(\vec{x}_n) = c_n\ \mathbf{in}\ c) &\doteq FV.c \cup \bigcup_{1 \leqslant u \leqslant n} FV.c_i - \{\vec{x}_i\} \\
FV.(x := f(\vec{e})) &\doteq \{x\} \cup FV.\vec{e} \\
FV.(c_1; c_2) &\doteq FV.c_1 \cup FV.c_2 \\
FV.(\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{fi}) &\doteq FV.b \cup FV.c_1 \cup FV.c_2 \\
FV.(\mathbf{while}\ b\ \mathbf{do}\ c\ \mathbf{od}) &\doteq FV.b \cup FV.c
\end{aligned}
$$

donde $x, y \in Var$, $\vec{x}_1, \dots, \vec{x}_n \in Var^+$, $e \in Expr$, $\vec{e} \in Expr^+$, $i \in \mathbb{N}_0$, $f_1, \dots, f_n \in F$, $c, c_1, c_2, \dots, c_n \in Comm_T$, $b \in Bool$, y asumimos dadas las funciones estándar $FV \in Expr \to Var$ y $FV \in Bool \to Var$.

Definición 44 (Variables modificadas de comandos de SL_T^H)**.** *Sea $c \in Comm_T$, el conjunto de variables modificadas por c se define extendiendo la función de la definición 23 (pág. 57) de la siguiente manera:*

$$
\begin{aligned}
Mod.(\mathbf{newvar}(x); c) &\doteq Mod.c - \{x\} \\
Mod.(\mathbf{return}(e)) &\doteq \emptyset \\
Mod.(\mathbf{let}\ f_1(\vec{x}_1) = c_1, \dots, f_n(\vec{x}_n) = c_n\ \mathbf{in}\ c) &\doteq Mod.c \cup \bigcup_{1 \leqslant u \leqslant n} Mod.c_i - \{\vec{x}_i\} \\
Mod.(x := f(\vec{e})) &\doteq \{x\}
\end{aligned}
$$

donde $x \in Var$, $\vec{x}_1, \dots, \vec{x}_n \in Var^+$, $e \in Expr$, $\vec{e} \in Expr^+$, $c, c_1, c_2, \dots, c_n \in Comm_T$, $n \in \mathbb{N}_0$.

Definición 45 (Entorno semántico de funciones bien formado)**.** *Dada $\phi \in Fun$, decimos que ϕ* está bien formado*, si para todo $(x_1, \dots, x_n, c) \in img.\phi$ vale que $FV.c \subseteq \{x_1, \dots, x_n\}$ y $Mod.c = \emptyset$.*

La semántica de los comandos de SL_T^H extiende trivialmente a las nuevas configuraciones. Los nuevos comandos se definen a continuación.

Definición 46 (Semántica de comandos). *La relación de transición entre configuraciones* $\rightsquigarrow \in Conf_T \times Conf_T$ *se define como la relación de la definición 18 (pág. 53) adaptada apropiadamente, y extendida con las siguientes reglas:*

Variable local:

$$\frac{c,(s,h),\phi \rightsquigarrow^* (s',h')}{\mathbf{newvar}(x); c,(s,h),\phi \rightsquigarrow^* (s'[x \mapsto [\![x]\!]_s],h')}$$

Valor de retorno:

$$\mathbf{return}(e),(s,h),\phi \rightsquigarrow (s[ret \mapsto [\![e]\!]_s],h)$$

Def. de módulo:

$$\frac{c,(s,h),\phi[f_1 \mapsto (\vec{x}_1,c_1)]\dots[f_n \mapsto (\vec{x}_n,c_n)] \rightsquigarrow^* (s',h')}{\mathbf{let}\ f_1(\vec{x}_1) = c_1,\dots,f_n(\vec{x}_n) = c_n\ \mathbf{in}\ c,(s,h),\phi \rightsquigarrow^* (s',h')}$$

Llamada a función:

$$\frac{c,(s[x_1 \mapsto e_1]\dots[x_n \mapsto e_n],h),\phi \rightsquigarrow^* (s',h') \qquad \phi.f = (x_1,\dots,x_n,c)}{x := f(e_1,\dots,e_n),(s,h),\phi \rightsquigarrow^* (s[x \mapsto [\![ret]\!]_{s'}],h')}$$

donde $x, x_1, \dots, x_n \in Var$, $e, e_1, \dots, e_n \in Expr$, $\vec{x}_1, \dots, \vec{x}_n \in Var^+$, $f, f_1, \dots, f_n \in F$, $n \in \mathbb{N}_0$, $c, c_1, \dots, c_n \in Comm$, s, h *y* $s', h' \in States$, *y* $\phi \in Fun$.

Especificación de tipos abstractos de datos

Para razonar sobre los nuevos comandos, extendemos el concepto de especificación incorporando un entorno sintáctico de predicados, con definiciones (posiblemente recursivas) de los predicados A utilizados en las fórmulas, y un entorno sintáctico de funciones, que especifica las pre y postcondiciones de cada función.

Definición 47 (Especificación de SL_T^H). *El conjunto* $Spec_T$ *de especificaciones de comandos está dado por la siguiente gramática:*

$$\begin{aligned} Env_a \ni \Lambda &::= \epsilon \mid a(\vec{x}) \doteq p \mid \Lambda, \Lambda \\ Env_f \ni \Phi &::= \epsilon \mid \{p\}\, f(\vec{x})\, \{p\} \mid \Phi, \Phi \\ Spec_T \ni s &::= \Lambda; \Phi \vdash \{p\}\, c\, \{p\} \end{aligned}$$

donde ϵ *denota la secuencia vacía,* $\vec{x} \in Var^+$, $a \in A$, $f \in F$, $p \in Form_T$ *y* $c \in Comm_T$.

Cada vez que escribimos una especificación asumimos que los predicados y funciones están caracterizados a lo sumo una única vez, y que las variables libres se limitan a los parámetros formales (y eventualmente la variable distinguida *ret*). Sobre los predicados se requiere adicionalmente que las definiciones utilizadas sean *positivas*, *i.e.* que las ocurrencias de los nombres de predicados aparezcan bajo un número par de negaciones, para garantizar una correcta semántica.

Definición 48 (Entorno sintáctico de predicados bien formado). *Dado* $\Lambda \in Env_a$, *decimos que* Λ *está* bien formado *si para cada* $a \in A$*, una definición como* $a(\vec{x}) \doteq p$ *ocurre a lo sumo una vez en* Λ *(para* $p \in Form_T$ *y* $\vec{x} \in (Var \cup Var)^+$*),* $FV.p \subseteq \vec{x}$*, y* a *ocurre en* p *a lo sumo bajo un número par de negaciones.*

Definición 49 (Entorno sintáctico de funciones bien formado). *Dado* $\Phi \in Env_f$*, decimos que* Φ *está* bien formado *si para cada* $f \in F$*, una especificación como* $\{\, p \,\}\ f(\vec{x})\ \{\, q \,\}$ *ocurre a lo sumo una vez en* Φ *(para* $p, q \in Form_T$ *y* $\vec{x} \in Var^+$*), y* $FV.p \cup FV.q \subseteq \vec{x} \cup \{ret\}$.

La validez de una especificación como $\Lambda; \Phi \vdash \{\, p \,\}\ c\ \{\, q \,\}$ se define en términos de los entornos semánticos de predicados y funciones caracterizados por los respectivos entornos sintácticos Λ y Φ. Una vez que contamos con tales entornos semánticos, definir la validez de una terna es elemental.

Definición 50 (Validez respecto a entornos semánticos). *Dados* $\alpha \in Pred$ *y* $\phi \in Fun$*, una terna* $\{\, p \,\}\ c\ \{\, q \,\}$ *es* válida *respecto a* α *y* ϕ*, denotado* $\alpha, \phi \vDash \{\, p \,\}\ c\ \{\, q \,\}$ *si para todos* s, h *y* $s'', h'' \in States$*,* $s' \in Stacks'$

$$\text{si } s, h, s', \alpha \vDash p \text{ entonces } c, (s,h), \phi \text{ es safe, y si } c, (s,h), \phi \leadsto^* (s'', h'') \text{ entonces } s'', h'', s', \alpha \vDash q.$$

Los entornos semánticos de funciones admisibles son aquellos que *satisfacen* las especificaciones dadas.

Definición 51 (Satisfacción de entorno de funciones). *Dado* $\alpha \in Pred$*, un entorno semántico de funciones* $\phi \in Fun$ satisface *un entorno sintático de funciones* $\Phi \in Env_f$*, denotado como* $\alpha, \phi \vDash \Phi$*, si*

$$\alpha, \phi \vDash \{\, p \,\}\ c\ \{\, q \,\}$$

para toda $\{\, p \,\}\ f(\vec{x})\ \{\, q \,\}$ *en* Φ*, con* $\phi.f = (\vec{x}, c)$*.*

Determinar cuáles son los entornos semánticos de predicados es una tarea más difícil. Es necesario garantizar que es posible *construirlos* de acuerdo a las definiciones de Λ. Para ello definimos un reticulado de entornos semánticos.

Definición 52 (Orden entre entornos semánticos de predicados). *Dados $\alpha, \alpha' \in$ Pred, definimos el orden $\sqsubseteq \in$ Pred $\times$ Pred y decimos que $\alpha \sqsubseteq \alpha'$ si y sólo si*

$$(\alpha.a).v_1 \dots v_n \subseteq (\alpha'.a).v_1 \dots v_n$$

para todo $a \in A$ tal que $a \in dom.\alpha$, y todos $v_1, \dots, v_n \in Values'$ con $n = arity.a$.

Lema 13 (Reticulado de entornos semánticos). *Los entornos semánticos de predicados bien formados forman un reticulado completo respecto a $\sqsubseteq$. Con $\sqcup$ denotamos el operador de supremo.*

Como un entorno de predicados Λ no necesariamente define cada nombre de predicado, para construir una solución a las definiciones de Λ se vuelve necesario contar con un entorno semántico adicional α que complete lo no definido por Λ. La construcción de la solución para Λ se efectua a través de iteraciones sucesivas de la siguiente función.

Definición 53 (Función *step*). *Dado $\Lambda \in Env_a$ bien formado, y $\alpha \in$ Pred bien formado tal que $dom.\Lambda \cap dom.\alpha = \emptyset$, la función*

$$step_{\alpha,\Lambda} \in Pred \to A \rightharpoonup_{fin} \bigcup_{n \in \mathbb{N}_0} Values'^{n} \to \mathbb{P}.Heaps$$

se define como

$$\begin{aligned} step_{\alpha,\Lambda}.\alpha' \doteq\ & \lambda a \in dom.\Lambda \cdot \lambda \vec{v} \in Values'^{arity.a}. \\ & \{h \in Heaps | s, h, s', (\alpha' \sqcup \alpha) \vDash p_{[\vec{x} \leftarrow \vec{v}]},\ para\ todas\ s, s',\ con\ a(\vec{x}) \doteq p\ en\ \Lambda\} \end{aligned}$$

para todo $\alpha' \in$ Pred bien formado. Abusando de la notación, utilizamos $dom.\Lambda$ para denotar el conjunto de nombres de predicados que se definen en Λ.

La función *step* es mónotona en los entornos de predicados porque todas las definiciones son positivas, y las fórmulas positivas son monótonas respecto a los entornos semánticos de predicados. Así, por el teorema de Tarski y el lema 13, la función *step* tiene mínimo punto fijo.

Lema 14 (Mínimo punto fijo). *Sean $\alpha, \alpha', \alpha'' \in$ Pred, $s, h \in$ States, $s' \in$ Stacks', $p \in Form_T$ y $\Lambda \in Env_a$. Entonces,*

1. *si $\alpha \sqsubseteq \alpha'$ y $s, h, s', \alpha \vDash p$, entonces $s, h, s', \alpha' \vDash p$;*
2. *si $\alpha' \sqsubseteq \alpha''$, entonces $step_{\alpha,\Lambda}.\alpha' \subseteq step_{\alpha,\Lambda}.\alpha''$;*
3. *$step_{\alpha,\Lambda}$ tiene mínimo punto fijo, denotado como $[\![\Lambda]\!]_\alpha$.*

Considerando todas los entornos semánticos de predicados de la forma $\alpha \sqcup [\![\Lambda]\!]_\alpha$, para un α arbitrario que caracteriza los predicados no definidos por Λ, es posible definir la noción de validez de una especificación.

Definición 54. *Dado $\Lambda \in Env_a$, el conjunto de entornos semánticos de predicados respecto a las definiciones de Λ, queda determinado por el conjunto close.Λ definido como*

$$close.\Lambda \doteq \{\alpha \sqcup [\![\Lambda]\!]_\alpha \mid \alpha \in Pred \text{ está bien formado y } dom.\alpha = A - dom.\Lambda\}$$

Definición 55 (Validez de especificación de SL_T^H)**.** *Dada una especificación $\Lambda; \Phi \vdash \{p\}\ c\ \{q\} \in Spec_T$, decimos que es* válida, *denotada $\Lambda; \Phi \vDash \{p\}\ c\ \{q\}$ si para todo $\alpha \in close.\Lambda$ y $\phi \in Fun$ bien formado*

$$\text{si } \alpha, \phi \vDash \Phi \text{ entonces } \alpha, \phi \vDash \{p\}\ c\ \{q\}$$

Razonamiento sobre tipos abstractos de datos

Al comienzo de la sección definimos la satisfacción de una fórmula respecto al modelo de estados y un entorno semántico de predicados (def. 40, pág. 75). Pero para poder razonar sobre aserciones que utilizan predicados abstractos es necesario definir qué significa que una fórmula sea *válida* respecto a un entorno sintáctico de predicados.

Definición 56 (Validez de fórmulas resp. a un entorno de predicados)**.** *Dados $\Lambda \in Env_a$ y $p \in Form$, decimos que p es* válida *respecto a Λ, denotado como $\Lambda \vDash p$, si para cualquier estado $s, h \in States$, cualquier* stack *lógico $s' \in Stacks'$ y toda solución $\alpha \in close.\Lambda$ se satisface que $s, h, s', \alpha \vDash p$.*

Las reglas de cálculo presentadas a lo largo de la sección 3.2 extienden trivialmente a la nueva definición de validez. Ademas incorporamos las siguientes reglas para el uso de los predicados abstractos.

Lema 15 (Reglas de cálculo de SL_T)**.** *Sea $a \in A$ un nombre de predicado con parámetros formales $\vec{x} \in (Var \cup Var')^n$, con $n = arity.a$ y definición $p \in Form$, entonces*

***Fold*:**

$$a(\vec{x}) \doteq p, \Lambda \vDash a(\vec{e}) \Rightarrow p_{/\vec{x} \leftarrow \vec{e}}$$

***Unfold*:**

$$a(\vec{x}) \doteq p, \Lambda \vDash p_{/\vec{x} \leftarrow \vec{e}} \Rightarrow a(\vec{e})$$

donde $\vec{e} \in (Expr \cup Expr')^n$.

La especificación de los comandos de nueva variable, valor de retorno y llamada a procedimiento son mayormente estándar.

Definición 57 (Sistema deductivo de SL_T^H)**.** *Las siguientes reglas caracterizan los nuevos comandos:*

Variable local:

$$\frac{\Lambda;\Phi \vdash \{\, p_{/x\leftarrow y} \wedge x = \mathbf{null} \,\}\ c\ \{\, q_{/x\leftarrow y} \,\} \qquad y \notin FV.p \cup FV.c \cup FV.q}{\Lambda;\Phi \vdash \{\, p \,\}\ \mathbf{newvar}(x); c\ \{\, q \,\}}$$

Valor de retorno:

$$\Lambda;\Phi \vDash \{\, p_{/ret\leftarrow e} \,\}\ \mathbf{return}(e)\ \{\, p \,\}$$

Llamada a función:

$$\frac{\{\, p \,\}\ f(\vec{x})\ \{\, q \,\}}{\Lambda;\Phi \vdash \{\, p_{/\vec{x}\leftarrow\vec{e}} \,\}\ y := f(\vec{e})\ \{\, q_{/\vec{x},ret\leftarrow\vec{e},y} \,\}}$$

donde $x, y \in Var$, $\vec{x} \in Var^n$, $e \in Expr$, $\vec{e} \in Expr^n$, $p, q \in Form_T$, $c \in Comm_T$, *y* $f \in F$ *con* $n = arity.f$.

Los predicados abstractos pueden ser utilizados como tipos abstractos de datos: tienen un nombre, un *scope* y una definición que caracteriza la estructura de datos utilizada para su representación. Dentro de su *scope*, el nombre puede intercambiarse con su definición libremente, pero fuera de él, sólo puede manipularse el tipo abstracto a través de su nombre. Esto motiva la siguiente regla para razonar sobre definiciones de funciones.

Definición 58 (Sistema deductivo de SL_T^H (cont.))**.** *La siguiente regla caracteriza la **definición de funciones:***

$$\frac{\begin{array}{c}\Lambda,\Lambda';\Phi \vdash \{\, p_1 \,\}\ c_1\ \{\, q_1 \,\} \\ \vdots \\ \Lambda,\Lambda';\Phi \vdash \{\, p_n \,\}\ c_n\ \{\, q_n \,\} \\ \Lambda;\{\, p_1 \,\}\ f_1(\vec{x}_1)\ \{\, q_1 \,\}, \dots, \{\, p_n \,\}\ f_n(\vec{x}_n)\ \{\, q_n \,\}, \Phi \vdash \{\, p \,\}\ c\ \{\, q \,\}\end{array}}{\Lambda,\Phi \vdash \{\, p \,\}\ \mathbf{let}\ f_1(\vec{x}_1) = c_1, \dots, f_n(\vec{x}_n) = c_n\ \mathbf{in}\ c\ \{\, q \,\}}$$

cuando

- p, q, Φ, *y* Λ *no contienen nombres de predicados en* $dom.\Lambda'$,
- *para cada* $\{\, p_i \,\}\ c_i\ \{\, q_i \,\}$, *con* $1 \leqslant i \leqslant n$, *se satisface que* $FV.c_i \subseteq \vec{x}_i$ *y* $Mod.c_i = \emptyset$,

y donde $f_1, \dots, f_n \in F$, $\vec{x}_1, \dots, \vec{x}_n \in Var^+$, $p, p_1, \dots, p_n \in Form$, $c, c_1, \dots, c_n \in Comm$, $\Lambda, \Lambda' \in Env_a$ *y* $\Phi \in Env_f$.

Esta regla permite a quien provee las funciones utilizar la definición de un predicado abstracto en Λ', pero al cliente, sólo manipularlo atómicamente a través de las mismas funciones. Los predicados definidos en Λ' están dentro del *scope* de las funciones $fname_i$, y por lo tanto es posible utilizar la definición de los mismos para razonar sobre el código c_i Pero Λ' no está dentro del *scope* del código cliente c, que sólo puede utilizar las especificaciones $\{p_i\}\ c_i\ \{q_i\}$. Las condiciones garantizan los entornos de predicados y de funciones resulten bien formado.

El uso que puede hacerse de la definición de un predicado de abstracción para la verificación del código de una función, se captura a través de la regla de consecuencia en combinación con las propiedades de *folding* y *unfolding* definidas anteriormente.

Definición 59 (Sistema deductivo de SL_T^H (cont.)). *La siguiente regla es la adaptación de la* ***regla de consecuencia*** *a la nueva forma de validez y de especificación:*

$$\frac{\Lambda \vDash p' \Rightarrow p \qquad \Lambda;\Phi \vdash \{p\}\ c\ \{q\} \qquad \Lambda \vDash q \Rightarrow q'}{\Lambda;\Phi \vdash \{p'\}\ c\ \{q'\}}$$

donde $p, p', q, q' \in Form_T$, $c \in Comm_T$, $\Lambda \in Env_a$ *y* $\Phi \in Env_f$.

Las demás reglas que forman parte del sistema deductivo de SL^H, presentadas en la sección 3.4, se adaptan al nuevo tipo de especificación de forma trivial.

Teorema 2 (Consistencia del sistema deductivo de SL_T^H). *Sean* $p, q \in Form_T, c \in Comm_T$, $\Lambda \in Env_a$ *y* $\Phi \in Env_f$. *Si* $\Lambda;\Phi \vdash \{p\}\ c\ \{q\}$ *se deduce utilizando las reglas del sistema deductivo,*[7] *entonces* $\Lambda;\Phi \vDash \{p\}\ c\ \{q\}$.

Ejemplo 38. *Consideremos el caso de un predicado abstracto* $\mathbf{vector}.x.es$ *que define un vector mutable que representa una secuencia abstracta es, y un predicado* $\mathbf{iter}.i.n.es$ *que define un iterador sobre el vector. Supongamos que tales predicados forman parte del entorno sintáctico de predicados* Λ *definido como:*

$$\begin{aligned}
\mathbf{coll}.x.y.ls.es &\doteq (x = y \wedge ls = es = [\,] \wedge \mathbf{emp}) \vee \\
&\quad (\exists ls', e, es', z \cdot (x \mapsto e, z * \mathbf{coll}.z.y.ls'.es') \wedge ls = x \triangleright ls' \wedge es = e \triangleright es') \\
\mathbf{vector}.x.es &\doteq \exists ls \cdot \mathbf{coll}.x.\mathbf{null}.ls.es \\
\mathbf{iter}.i.n.es &\doteq \exists x, j, ls \cdot (i \mapsto x, j * \mathbf{coll}.x.\mathbf{null}.ls.es) \wedge j = ls.n
\end{aligned}$$

El predicado auxiliar $\mathbf{coll}.x.y.ls.es$ *es análogo a* $\mathbf{lseg}.x.y.es$*, donde el parámetro extra ls explicita las direcciones de los registros que conforman la lista enlazada. Esta caracterización es necesaria para poder definir el predicado* **iter***, que contiene un puntero al n-ésimo de estos registros. El predicado* $\mathbf{vector}.x.es$ *resulta análogo al predicado* $\mathbf{list}.x.es$*. El tipo abstracto está conformado por diversas funciones, cuyas especificaciones se muestran a*

[7] Definiciones 22, 24, 25, 26, 27, 28 adaptadas apropiadamente, y definiciones 57, 58, 59.

continuación:

$$
\begin{array}{rcl}
\{\,\mathbf{emp}\,\} & \mathit{new_vec}() & \{\,\mathbf{vector}.\mathit{ret}.[\,]\,\}\\
\{\,\mathbf{vector}.x.\mathit{es}\,\} & \mathit{add}(x,e) & \{\,\mathbf{vector}.x.(e \triangleright \mathit{es})\,\}\\
\{\,\mathbf{vector}.x.(e \triangleright \mathit{es})\,\} & \mathit{del}(x) & \{\,\mathbf{vector}.x.\mathit{es}\,\}\\
\{\,\mathbf{vector}.x.(e \triangleright \mathit{es})\,\} & \mathit{new_iter}(x) & \{\,\mathbf{iter}.\mathit{ret}.0.(e \triangleright \mathit{es})\,\}\\
\{\,\mathbf{vector}.x.\mathit{es}\,\} & \mathit{free_vec}(x) & \{\,\mathbf{emp}\,\}\\
\{\,\mathbf{iter}.i.n.\mathit{es} \wedge n < \#\mathit{es} - 1\,\} & \mathit{next}(i) & \{\,\mathbf{iter}.i.(n+1).\mathit{es}\,\}\\
\{\,\mathbf{iter}.i.n.\mathit{es} \wedge n \leqslant \#\mathit{es} - 1\,\} & \mathit{set}(i,e) & \{\,\mathbf{iter}.i.n.(\mathit{es}[n \mapsto e])\,\}\\
\{\,\mathbf{iter}.i.n.\mathit{es} \wedge n \leqslant \#\mathit{es} - 1\,\} & \mathit{get}(i) & \{\,\mathbf{iter}.i.n.\mathit{es} \wedge \mathit{ret} = \mathit{es}.n\,\}\\
\{\,\mathbf{iter}.i.n.\mathit{es}\,\} & \mathit{end?}(i) & \{\,\mathbf{iter}.i.n.\mathit{es} \wedge \mathit{ret} = (n > \#\mathit{es} - 1)\,\}\\
\{\,\mathbf{iter}.i.n.\mathit{es}\,\} & \mathit{free_iter}(i) & \{\,\mathbf{vector}.\mathit{ret}.\mathit{es}\,\}
\end{array}
$$

Para demostrar las especificaciones de las funciones, podemos utilizar los predicados en Λ*. A continuación, mostramos el esquema de prueba del código que implementa new_iter:*

$\{\,\mathbf{vector}.x.(e \triangleright es)\,\}$

$\Rightarrow \langle$ *definición de* **vector** $\rangle$

$\{\,\exists ls \cdot \mathbf{coll}.x.\mathbf{null}.ls.(e \triangleright es)\,\}$

$\Rightarrow \langle$ *definición de* **coll** $\rangle$

$\{\,\exists ls \cdot \exists ls', z \cdot (x \mapsto e, z \ast \mathbf{coll}.z.\mathbf{null}.ls'.es) \wedge ls = x \triangleright ls'\,\}$

  newvar(i);

$\{\,\exists ls \cdot \exists ls', z \cdot (x \mapsto e, z \ast \mathbf{coll}.z.\mathbf{null}.ls'.es) \wedge ls = x \triangleright ls' \wedge i = \mathbf{null}\,\}$

  $i := \mathbf{cons}(x, x)$;

$\{\,\exists ls \cdot \exists ls', z \cdot (i \mapsto x, x \ast x \mapsto e, z \ast \mathbf{coll}.z.\mathbf{null}.ls'.es) \wedge ls = x \triangleright ls'\,\}$

$\Rightarrow \langle$ *introducción de* $\exists$ *sobre* x $\rangle$

$\{\,\exists j, ls \cdot \exists ls', z \cdot (i \mapsto x, j \ast x \mapsto e, z \ast \mathbf{coll}.z.\mathbf{null}.ls'.es) \wedge ls = x \triangleright ls'$
$\wedge\, j = ls.0\,\}$

$\Rightarrow \langle$ *definición de* **coll** $\rangle$

$\{\,\exists j, ls \cdot (i \mapsto x, j \ast \mathbf{coll}.x.\mathbf{null}.ls.(e \triangleright es)) \wedge j = ls.0\,\}$

$\Rightarrow \langle$ *introducción de* $\exists$ *sobre* x $\rangle$

$\{\,\exists x, j, ls \cdot (i \mapsto x, j \ast \mathbf{coll}.x.\mathbf{null}.ls.(e \triangleright es)) \wedge j = ls.0\,\}$

$\Rightarrow \langle$ *definición de* **iter** $\rangle$

$\{\,\mathbf{iter}.i.0.(e \triangleright es)\,\}$

  return(i)

$\{\,\mathbf{iter}.ret.0.(e \triangleright es)\,\}$

Supongamos que las especificaciones anteriores conforman un entorno de funciones Φ*. Es fácil demostrar la siguiente especificación, sobre una versión* abstracta *del código que verifica la existencia de un valor* v *dentro de un vector* $\mathbf{vector}.x.(e \triangleright es)$*, análogo al código del ejemplo 35 (pág. 68):*

$$
\epsilon;\Phi \vdash
\begin{array}{l}
\{\,\mathbf{vector}.x.(e \triangleright es)\,\}\\
\quad i := new_iter(x);\\
\quad w := get(i);\\
\quad e := end?(i);\\
\quad \mathbf{while}\ \neg e \wedge w \neq v\ \mathbf{do}\\
\qquad next(i);\\
\qquad w := get(i);\\
\qquad e := end?(i)\\
\quad \mathbf{od};\\
\quad x := free_iter(i)\\
\quad r := w = v\\
\{\,\mathbf{vector}.x.(e \triangleright es) \wedge r = exists.v.(e \triangleright es)\,\}
\end{array}
$$

Como el entorno de predicados es vacío, la demostración sólo hace uso de las especificaciones dadas en Φ*, utilizando el siguiente invariante de ciclo:*

$$\mathbf{iter}.i.n.(e \triangleright es) \wedge exists.v.(e \triangleright es) = exists.v.(w \triangleright (es \downarrow n+1))$$

4 *Sharing Logic*

El operador "estrella" del lenguaje de fórmulas de la SL, la conjunción espacial $*$, permite codificar la hipótesis, deseable y usualmente extendida, de inexistencia de alias entre los punteros de un programa. La caracterización del comportamiento de los programas a través de su *footprint*, y el uso de la regla de *frame*, permiten razonar local y efectivamente bajo esta hipótesis. Esta situación es la inversa a la dada en la lógica de Hoare usual, donde el *aliasing* ocurre por defecto. Sin embargo, las ventajas que otorga el nuevo lenguaje de aserciones y el nuevo sistema deductivo tiene su costo: la regla de constancia

$$\frac{\vdash \{p\}\ c\ \{q\} \qquad FV.r \cap Mod.c = \varnothing}{\vdash \{p \wedge r\}\ c\ \{q \wedge r\}}$$

ya no es válida en general. El uso del operador $\wedge$ en una fórmula como $p \wedge r$ *potencialmente* introduce múltiples alias entre los punteros caracterizados por p y r, ya que especifica que ambas fórmulas se satisfacen en el *mismo heap*. Consideremos el siguiente caso particular:

$$\frac{\vdash \{x \mapsto _,_\}\ x.0 := \mathbf{null}\ \{x \mapsto \mathbf{null},_\}}{\vdash \{x \mapsto _,_ \wedge y \mapsto 1,2\}\ x.0 := \mathbf{null}\ \{x \mapsto \mathbf{null},_ \wedge y \mapsto 1,2\}}$$

Aquí la precondición no precluye la posibilidad de que $x = y$, sino que por el contrario *garantiza* que x e y referencian al mismo registro. Sin embargo la postcondición no da cuenta que y referencia a un registro modificado.

Mientras que el sistema deductivo de la SL^H presenta muchas ventajas para razonar cuando es posible garantizar la separación espacial, no provee herramientas para la deducción sistemática cuando esto no ocurre. Esto obliga a proceder en estas circunstancias de manera *ad hoc*, utilizando complejos predicados para caracterizar las porciones de memoria compartida y argumentos *semánticos* para garantizar que la manipulación de la misma preserva las propiedades necesarias. Esta dificultad es un problema abierto que sigue en discusión en la actualidad, como se aprecia en la siguiente cita de [81]:

> The frame rule buys us compositionality in the presence of the heap: we can reason about the effect a program has on the portions of heap it accesses, and reuse that spec in any bigger heap. This has given rise to concise, compositional proofs of programs, even in the presence of some forms of sharing where one

> knows what is shared by whom. Unfortunately, we usually cannot use the frame rule directly when verifying programs that manipulate data structures with unrestricted sharing because such structures cannot easily be massaged into the form $p * q$: for example, the left and right descendants of a dag node are not usually disjoint.

La necesidad de especificar múltiples estructuras que comparten el *heap* para su representación se presenta en diversos escenarios. Las implementaciones de estructuras de datos como *dags* (grafos acíclicos dirigidos) y grafos generales, usualmente involucran una la utilización de alias que determinan subestructuras *compartidas* desde el punto de vista abstracto. Para dar cuenta de ello, Bornat introduce el operador *relevance conjunction*[1] $\uplus$ que relaja la condición de disjuntividad de $*$ permitiendo *sharing* parcial aunque inespecífico entre los *heaps* que satisfacen cada uno de los términos que acompañan al operador.

Definición 60 (Unspecified sharing)**.** *Dado un modelo $s, h \in States$ y $p, q \in Form$, la semántica del operador de* unspecified sharing *está dada por:*

$$s, h \vDash p \uplus q \textit{ si y sólo si existen } h_1, h_2 \textit{ tales que } h_1 \cup h_2 = h \textit{ y } s, h_1 \vDash p \textit{ y } s, h_2 \vDash q$$

Notar que a diferencia del operador $*$, no requiere que $h_1 \perp h_2$ (ver def. 10, pág. 46). Con este operador, es posible dar una definición recursiva *precisa*[2] de *dags*.

Definición 61 (*Dag* preciso)**.** *El siguiente predicado recursivo caracteriza los grafos acíclicos dirigidos que comienzan en x:*

$$\begin{aligned} \mathbf{dag}'.x.\langle\,\rangle &\doteq x = \mathbf{null} \wedge \mathbf{emp} \\ \mathbf{dag}'.x.\langle lt, e, rt\rangle &\doteq \exists l, r \cdot x \mapsto l, e, r * (\mathbf{dag}'.l.lt \uplus \mathbf{dag}'.r.rt) \end{aligned}$$

donde $x, l, r \in Var$, $e \in Expr$, $lt, rt \in Tree$.

Lema 16. $\mathbf{dag}'.x.et \in Precise$, *para cualesquiera $x \in Var, et \in Tree$.*

El operador $\uplus$ tiene buenas propiedades (algunas de las cuales se presentan en la figura 4.1) y permite especificar ciertas formas de *sharing* como la ejemplificada más arriba. Sin embargo esta forma de *sharing* inespecífico dificulta razonar estructuralmente, ya que no parece posible definir una regla de *frame* respecto a este operador.

Diversos algoritmos reutilizan *temporariamente* parte de la estructura de datos que manipulan para otros fines. El ejemplo paradigmático es el algoritmo de marcado de grafos Schorr-Waite que arma la lista de nodos recorridos enlazando los propios

[1] Este operador es propio de las Relevant Logic [] y de allí su nombre.
[2] Y por lo tanto no intuicionista.

$$
\begin{aligned}
&\vDash p \Rightarrow p \mathbin{\cup\!\!\!\!\!\ast} p\\
&\vDash p * q \Rightarrow p \mathbin{\cup\!\!\!\!\!\ast} q\\
&\vDash p \wedge q \Rightarrow p \mathbin{\cup\!\!\!\!\!\ast} q\\
&\vDash p \mathbin{\cup\!\!\!\!\!\ast} q \Leftrightarrow q \mathbin{\cup\!\!\!\!\!\ast} p\\
&\vDash p \mathbin{\cup\!\!\!\!\!\ast}(q \mathbin{\cup\!\!\!\!\!\ast} r) \Leftrightarrow (p \mathbin{\cup\!\!\!\!\!\ast} q) \mathbin{\cup\!\!\!\!\!\ast} r\\
&\vDash p \mathbin{\cup\!\!\!\!\!\ast} \Leftrightarrow p\\
&\vDash (p \vee q) \mathbin{\cup\!\!\!\!\!\ast} r \Leftrightarrow (p \mathbin{\cup\!\!\!\!\!\ast} r) \vee (q \mathbin{\cup\!\!\!\!\!\ast} r)\\
&\vDash (p \wedge q) \mathbin{\cup\!\!\!\!\!\ast} r \Rightarrow (p \mathbin{\cup\!\!\!\!\!\ast} r) \wedge (q \mathbin{\cup\!\!\!\!\!\ast} r)
\end{aligned}
$$

Figura 4.1: Algunas reglas de cálculo del operador $\mathbin{\cup\!\!\!\!\!\ast}$

nodos del grafo. De forma similar, para optimizar algunas operaciones, muchas estructuras de datos implementan diferentes formas abstractas de acceder a los datos utilizando la misma memoria. Por ejemplo, es común que un árbol implemente simultáneamente una lista de nodos para su recorrido lineal (en pre-orden, post-orden, etc.).

Dentro de este escenario se enmarca el problema del *fringe*, presentado en [25]. El *fringe* de un árbol consiste en la lista de las hojas del mismo. El problema es el siguiente: dado un árbol, construir una lista enlazada que represente el *fringe* del mismo, utilizando para ello los propios nodos del árbol. Esto es posible ya que los nodos hoja contienen punteros que no son utilizados para la representación del árbol, dado que no tienen descendientes. El problema está especificado originalmente utilizando el patron de *inclusion sharing* de la siguiente manera:

$$(\mathbf{lseg}'.x.y.(\mathit{fringe}.et) * \mathbf{true}) \wedge \mathbf{btree}'.t.et$$

donde $\mathbf{lseg}'$ y $\mathbf{btree}'$ son ligeras variantes de los predicados presentados en la sección 3.5.[3] La definición de los predicados así como la función *fringe* se presentan en la figura 4.2. En la figura 4.3.b se presenta el árbol binario de la figura 4.3.a con su *fringe*.

El patrón de *inclusion sharing*, de la forma $(p * \mathbf{true}) \wedge q$, especifica que el *heap* que satisface p se encuentra incluído en, o es igual a, el *heap* que satisface q. Sirve para caracterizar de manera sencilla dos estructuras que comparten (parte de) su implementación.

El problema del *fringe* es particularmente interesante porque refleja las dificultades, muchas veces sutiles, que aparecen al describir estructuras que comparten el *heap*. La especificación original presenta diferentes problemas.

Por un lado, resulta demasiado *débil*, porque se satisface en modelos que no se ajustan a la descripción informal del problema. En la figura 4.4.a se presenta una

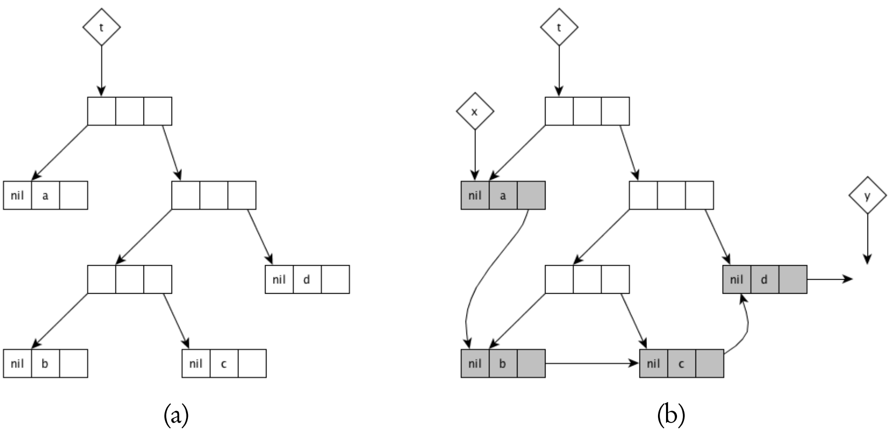
[3] La especificación original se da el marco de una *Separation Logic* con un modelo de celdas individuales y aritmética. Así la lista que representa el *fringe* consta de nodos de dos celdas consecutivas. A pesar de esta diferencia, la especificación es completamente equivalente, incluyendo sus limitaciones.

$$
\begin{aligned}
fringe &\in \langle Values\rangle \to [Values] \\
fringe.\langle e\rangle &\doteq [e] \\
fringe.\langle lt,e,rt\rangle &\doteq fringe.lt +\!\!+\ fringe.rt
\end{aligned}
$$

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

$$
\begin{aligned}
\mathbf{btree}'.x.\langle e\rangle &\doteq x \mapsto \mathbf{null}, e, _ \\
\mathbf{btree}'.x.\langle lt,e,rt\rangle &\doteq \exists l,r \cdot x \mapsto l,e,r * \mathbf{btree}'.l.lt * \mathbf{btree}'.r.rt
\end{aligned}
$$

Figura 4.2: Definiciones de *fringe*, **lseg**′ y **btree**′



(a) (b)

Figura 4.3: (a) Modelo de $\mathbf{btree}'.t.et$, y (b) modelo con *fringe*

.

situación patológica permitida por la especificación. El problema reside en que en general las relaciones entre valores abstractos no reflejan las relaciones entre los registros de memoria utilizados para su representación. Para salvar este problema, parece necesario mencionar explícitamente las direcciones de los registros de memoria que conforman la lista. En la figura 4.5 se presentan variantes de los predicados **lseg**′ y **btree**′ que implementan esta alternativa que, aunque es engorrosa y requiere modificar los predicados usuales, es efectiva. El problema puede especificarse entonces mediante la fórmula

$$\exists zs \cdot (\mathbf{lseg}''.x.y.zs.(fringe.et) * \mathbf{true}) \wedge \mathbf{btree}''.t.zs.et$$

Sin embargo una solución de este estilo es equiparable a las soluciones *ad hoc* de la lógica de Hoare estándar para inhibir el *aliasing*. Mientras que en la lógica estándar es

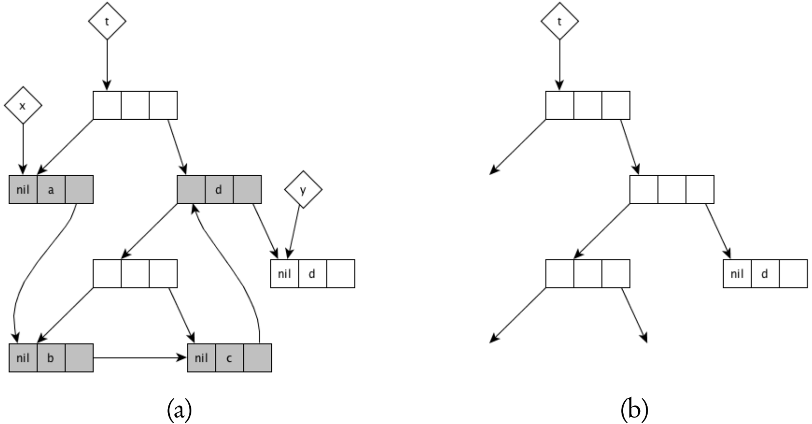


Figura 4.4: (a) Solución patológica del *fringe*, y (b) modelo de un árbol parcial.

$$
\begin{aligned}
leafs &\in \langle Values \rangle \rightarrow \mathbb{N}_0 \\
leafs.\langle v \rangle &\doteq 1 \\
leafs.\langle lt, v, rt \rangle &\doteq leafs.lt + leafs.rt
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{lseg}''.x.y.[\,].[\,] &\doteq x = y \wedge \mathbf{emp} \\
\mathbf{lseg}''.x.y.(z \triangleright zs).(e \triangleright es) &\doteq \exists w \cdot (x \mapsto _, e, w * \mathbf{lseg}''.w.y.zs.es) \wedge x = z
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{btree}''.x.[z].\langle e \rangle &\doteq x \mapsto \mathbf{null}, e, _ \wedge x = z \\
\mathbf{btree}''.x.zs.\langle lt, e, rt \rangle &\doteq \exists l, r \ \cdot \ x \mapsto l, e, r * \mathbf{btree}''.l.(zs \uparrow leafs.it).lt \\
&\qquad * \mathbf{btree}''.r.(zs \downarrow leafs.rt).rt
\end{aligned}
$$

Figura 4.5: Definiciones alternativas **lseg″** y **btree″**, y función auxiliar *leafs*.

necesario explicitar la separación espacial, aquí se vuelve imprescindible caracterizar individualmente los registros compartidos.

Por otro lado, el uso del operador $\wedge$ se salva en la demostración del programa recursivo presentado en [25], pero representa una verdadera traba para razonar localmente en el caso de una versión iterativa del mismo.

Cuando se programa con tipos abstractos de datos, o con el paradigma *orientado a objetos*, es muy común que dos o más instancias del tipo compartan regiones del *heap* para su representación, ya sea por una decisión de diseño, o por un error en la implementación que da lugar a una exposición de la misma. En cualquier caso, las distintas estructuras sobre el *heap* representan diferentes *vistas* o *abstracciones*, que es desable poder manipular de forma independiente.

Imaginemos un caso en un marco similar al del ejemplo 38 del capítulo anterior (pág. 83), en el que disponemos de un iterador $\mathbf{iter}.i.n.es$ sobre una vector mutable especificado por $\mathbf{vector}.x.es$, que representa la lista abstracta es. El iterador precisa memoria extra para su representación y supongamos que, contrario a la definición dada, el vector incluye también registros de memoria además de la colección de datos. El principio de *information hiding* inhibe la utilización de las definiciones de los predicados para dar una especificación como una combinación con $*$ de las partes que integran cada estructura. La situación en que el iterador y el vector comparten la colección de datos subyacente puede especificarse de manera análoga al ejemplo del *fringe*:

$$(\mathbf{iter}.i.n.es * \mathbf{true}) \wedge (\mathbf{vector}.x.es * \mathbf{true})$$

Pero esta especificación no sólo carece de utilidad para razonar localmente sobre cada abstracción, sino que permite un *sharing* mucho más extendido que lo que se pretende. Mas aún, la utilización de **true** que acompaña a cada predicado abre la posibilidad a que exista una cantidad arbitraria de registros en la memoria, que no forman parte de la implementación de **iter** ni de **vector** (la fórmula es *intuicionista*).

Podemos intentar una especificación alternativa utilizando el patrón *deletion followed by extension*. Si $\mathbf{coll}'.y.es$ representa la colección de datos es referenciada por y, una especificación alternativa es

$$(\mathbf{coll}'.y.es \mathrel{-\!\!*} \mathbf{iter}.i.n.es) * \mathbf{coll}'.y.es * (\mathbf{coll}'.y.es \mathrel{-\!\!*} \mathbf{vector}.x.es)$$

Sin embargo, esta especificación resulta demasiado *fuerte*, ya que en una fórmula $p \mathrel{-\!\!*} q$, los *heaps* que satisfarían p están cuantificados universalmente. Rara vez los predicados **iter** o **vector** se validan con *cualquier heap* que satisfaga $\mathbf{coll}'.y.es$. Podemos intentar una variante utilizando el operador de *septraction* [146, 40] definido como el operador dual de $\mathrel{-\!\!*}$.

Definición 62 (Operador *septraction*). *Dadas $p, q \in Form$, el operador de* septraction $\mathrel{-\!\!\circledast}$ *se define como*

$$p \mathrel{-\!\!\circledast} q \doteq \neg(p \mathrel{-\!\!*} \neg q)$$

A diferencia de lo que ocurre con $p \mathrel{-\!\!*} q$, la fórmula $p \mathrel{-\!\!\circledast} q$ cuantifica *existencialmente* sobre los *heaps* que satisfarían p.

Lema 17 (Semántica de *septraction*). *Sean $p, q \in Form$ y $s, h \in States$,*

$$s, h \models p \mathrel{-\!\!\circledast} q \text{ si y sólo si existe } h' \text{ tal que } h' \perp h \text{ y } s, h' \models p \text{ y } s, h \cup h' \models q$$

Así la especificación alternativa es

$$(\mathbf{coll}'.y.es \mathrel{-\!\!\circledast} \mathbf{iter}.i.n.es) * \mathbf{coll}'.y.es * (\mathbf{coll}'.y.es \mathrel{-\!\!\circledast} \mathbf{vector}.x.es)$$

Sin embargo esta versión resulta demasiado débil en general. La porción del *heap* que agrega el término $\mathbf{coll}'.y.es$ no necesariamente es *exactamente* igual a la requerida por los predicados, y “eliminada” a través del operador $-\circledast$.

Para asegurar que cualquiera de las dos variantes especifica de forma precisa la situación, es necesario no sólo explicitar cada una de las direcciones de registros de memoria relevantes (como en el ejemplo del *fringe*) sino también cualquier valor contenido dentro de los mismos que sea requerido para la validez de los predicados **iter** y **vector** (en el caso extremo esto equivale a que el predicado $\mathbf{coll}'.y.es$ sea *strictly exact*). En tal caso, las especificaciones utilizando $-\circledast$ y $-\!\!*$ se vuelven equivalentes.

En este capítulo presentamos una extensión de la SL que llamamos *Sharing Logic* (ShL), cuyo propósito es permitir la especificación de escenarios de *sharing* de una manera homogénea, de forma que nos permita estudiar y caracterizar las condiciones bajo las cuales es posible razonar sistemáticamente con ciertas garantías de localidad. Introducimos un nuevo lenguaje de fórmulas respecto al mismo modelo de memoria de la SL, y reglas de *frame* que permiten razonar manteniendo cierta forma de *localidad* a pesar del *sharing* especificado. La nueva lógica no sólo permite razonar sobre el lenguaje de programación imperativo simple con comandos para la manipulación del *heap*, sino que, agregando un entorno de *especificaciones estáticas*, soporta el lenguaje con tipos abstractos de datos presentado en el capítulo anterior.

4.1. Fórmulas con *specified sharing*

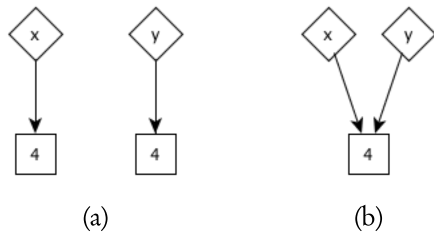
Nuestra propuesta se centra en la posibilidad de describir simultáneamente relaciones de separación espacial parcial y relaciones de *sharing* específico, entre distintas regiones del *heap*, a partir de la generalización de los operadores espaciales $*$ y $-\!\!*$. La ShL utiliza los mismos modelos que la SL (def. 7, pág. 42), pero cambia el lenguaje de fórmulas.

Definición 63 (Sintaxis de ShL). *El conjunto de expresiones Expr, relaciones Rel y fórmulas Form′ sobre Var está dado por las siguientes gramáticas:*

$$\begin{aligned} Expr \ni e &::= x \mid \mathbf{null} \mid \ldots \\ Rel \ni r &::= e = e \mid \ldots \\ Form' \ni p &::= r \mid \mathbf{emp} \mid x \mapsto \vec{e} \mid p \wedge p \mid \neg p \mid p \ \langle *\colon p\rangle\ p \mid p\ \langle -\!\!*\colon p\rangle\ p \mid \forall x \,.\, p \end{aligned}$$

donde $x \in Var$ y $\vec{e} \in Expr^+$.

Los nuevos operadores ternarios $\langle *\colon p\rangle$ y $\langle -\!\!*\colon p\rangle$ utilizan una fórmula p de la misma lógica para caracterizar la porción de *heap* compartido. Más precisamente, $p\ \langle *\colon r\rangle\ q$ se satisface en un modelo s, h si el *heap* h puede divirse en dos *subheaps* h_r y h_q no



(a) (b)

Figura 4.6: Modelos de $x \mapsto 4 \; \langle *\colon \mathbf{emp} \rangle \; y \mapsto 4$ (a), $x \mapsto 4 \; \langle *\colon x \mapsto 4 \rangle \; y \mapsto 4$ (b).

necesariamente disjuntos de manera que s, h_p satisface p, s, h_q satisface q, y la intersección entre ambos $h_p \cap h_q$ satiface r. El operador $\langle -\!*\colon p \rangle$ conserva la propiedad de ser el adjunto de $\langle *\colon p \rangle$.

Definición 64 (Semántica de fórmulas de ShL)**.** *Dado un modelo $s, h \in States$, la relación de satisfacción $\models$ $States \times Form'$ se define por inducción en las fórmulas como la relación de la definición 10 (pág. 46), y los siguientes casos:*

$$\begin{aligned} s, h \models p \; \langle *\colon r \rangle \; q \;\; & \textit{si y sólo si existen } h_1, h_2 \textit{ y } h_3 \textit{ tal que } h_1 \perp h_2 \perp h_3 \\ & \textit{y } h_1 \cup h_2 \cup h_3 = h \textit{ y } s, h_1 \cup h_2 \models p \textit{ y } s, h_2 \cup h_3 \models q \\ & \textit{y } s, h_2 \models r. \\ s, h \models p \; \langle -\!*\colon r \rangle \; q \; & \textit{si y sólo si para todo } h_1 \textit{ tal que } h_1 \perp h \\ & \textit{y existe } h_2 \subseteq h \textit{ tal que si } s, h_1 \cup h_2 \models p \textit{ y } s, h_2 \models r, \\ & \textit{entonces } s, h_1 \cup h \models q. \end{aligned}$$

donde $p, q, r \in Form'$.

El operador de *septraction* también puede generalizarse y resulta útil en nuestro sistema deductivo.

Definición 65 (Operador *septraction*)**.** *Dadas $p, q, r \in Form'$, el operador de* septraction $\langle -\!\circledast\colon \rangle$ *se define como*

$$p \; \langle -\!\circledast\colon r \rangle \; q \doteq \neg(p \; \langle -\!*\colon r \rangle \; \neg q)$$

Lema 18 (Semántica de *septraction*)**.** *Sean $p, q, r \in Form'$ y $s, h \in States$,*

$$\begin{aligned} s, h \models p \; \langle -\!\circledast\colon r \rangle \; q \; & \textit{si y sólo si existen } h_1, h_2 \textit{ tal que } h_1 \perp h \textit{ y } h_2 \subseteq h \\ & \textit{y } s, h_1 \cup h_2 \models p \textit{ y } s, h_2 \models r \textit{ y } s, h \cup h_1 \models q \end{aligned}$$

Ejemplo 39. *Consideremos la fórmula $x \mapsto 4 \; \langle *\colon r \rangle \; y \mapsto 4$. Si r es* **emp**, *entonces x e y referencian diferentes registros (fig. 4.6.a). Si r es $x \mapsto 4$, entonces necesariamente $x \mapsto 4 \; \langle *\colon r \rangle \; y \mapsto 4$ equivale a $x \mapsto 4 \wedge y \mapsto 4$ (fig. 4.6.b). Si r es* **true** *se abre la posibilidad a cualquier tipo de* sharing, *por lo que los modelos pueden ser como los dos mencionados.*

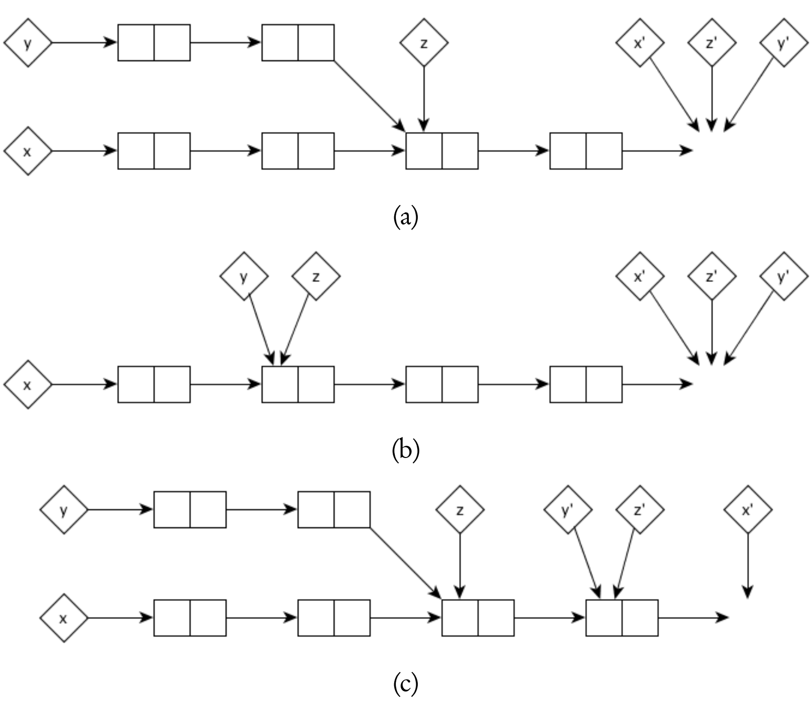


Figura 4.7: Modelos de $\textbf{lseg}.x.x'.xs \; \langle *: \textbf{lseg}.z.z'.zs \rangle \; \textbf{lseg}.y.y'.ys$

Ejemplo 40. *El operador* $\langle *: \rangle$ *introduce muchas sutilezas, porque ahora diferentes fórmulas pueden especificar la misma porción del* heap*. De esta manera, una porción del* heap *puede estar caracterizado en una fórmula, pero subespecificado en las otras. Las siguientes fórmulas son equivalentes y se satisfacen en los modelos como los de la figura 4.6.b):*

- $x \mapsto 4 \; \langle *: x \mapsto _ \rangle \; y \mapsto 4$,
- $x \mapsto 4 \; \langle *: _ \mapsto _ \rangle \; y \mapsto 4$,
- $x \mapsto 4 \; \langle *: y \mapsto _ \rangle \; _ \mapsto _$,
- $x \mapsto 4 \; \langle *: \neg \textbf{emp} \rangle \; y \mapsto _$,
- ...

Ejemplo 41. *Una fórmula como*

$$\textbf{lseg}.x.x'.xs \; \langle *: \textbf{lseg}.z.z'.zs \rangle \; \textbf{lseg}.y.y'.ys$$

caracteriza los modelos que contienen dos segmentos enlazados que comparten un segmento común que comienza en z y termina en z'*. Algunos modelos son como los de la*

figura 4.7. Notar que esta fórmula no es equivalente a

$$\mathbf{lseg}.x.z.xs' * \mathbf{lseg}.y.z.ys' * \mathbf{lseg}.z.z'.zs$$

(para xs' y ys' apropiados), ya que esta última no acepta modelos como los de la figura 4.7.c).

ShL vs. SL

El lenguaje de fórmulas de ShL representa claramente una generalización del lenguaje de SL ya que es posible recuperar la semántica original de los operadores espaciales como casos particulares de los nuevos operadores. Lo mismo ocurre con el operador *relevance conjunction*.

Lema 19. *Para todo modelo $s, h \in States$:*

1. $s, h \models p * q$ *si y sólo si* $s, h \models p \langle *\colon \mathbf{emp} \rangle q$.
2. $s, h \models p -\!\!* q$ *si y sólo si* $s, h \models p \langle -\!\!*\colon \mathbf{emp} \rangle q$.
3. $s, h \models p -\!\!\circledast q$ *si y sólo si* $s, h \models p \langle -\!\!\circledast\colon \mathbf{emp} \rangle q$.
4. $s, h \models p \uplus q$ *si y sólo si* $s, h \models p \langle *\colon \mathbf{true} \rangle q$.

Este resultado permite utilizar de aquí en más la sintaxis de los operadores de la SL como abreviaciones para los casos particulares que representan dentro de la ShL.

Las fórmulas que involucran los nuevos operadores no parecen ser reductibles a fórmulas de la SL (sobre este punto ver más en la sección 4.5). Sin embargo una fórmula de la forma $p \langle *\colon r \rangle q$ puede *aproximarse* utilizando el patrón *deletion followed by extensión* mencionado en la introducción. El siguiente lema precisa en qué sentido utilizar las variantes de este patrón resulta en especificaciones demasiado débiles o demasiado fuertes del *sharing* entre dos fórmulas.

Lema 20 (Reglas de cálculo de ShL)**.** *Sean $p, q, r \in Form'$, entonces*

Reducción débil:

$$\models p \langle *\colon r \rangle q \Rightarrow (r -\!\!\circledast p) * r * (r -\!\!\circledast q)$$

Reducción fuerte:

$$\models p \langle *\colon r \rangle q \Leftarrow (r -\!\!* p) * r * (r -\!\!* q)$$

Ejemplo 42. *Retomemos el ejemplo del* fringe. *Si definimos*

$$\mathbf{leafs}.[\,] \doteq \mathbf{emp}$$
$$\mathbf{leafs}.(x \rhd xs) \doteq x \mapsto _,_,_ * \mathbf{leafs}.xs$$

la especificación del problema se reduce a

$$\exists zs \cdot \mathbf{lseg}'.x.y.(fringe.et)\ \langle *\!: \mathbf{leafs}.zs\rangle\ \mathbf{btree}''.t.zs.et$$

Notar que utilizamos el predicado $\mathbf{lseg}'$ *y no* $\mathbf{lseg}''$ *ya que con la definición de* $\mathbf{btree}''$ *es suficiente para caraterizar precisamente las direcciones de los registros que conforman el conjunto de hojas de et, y* $\mathbf{lseg}'.x.y.(fringe.et)$ *no puede tener registros por fuera de* $\mathbf{leafs}.zs$ *puesto que* $\#fringe.et = \#zs$.

Ejemplo 43. *Si tomamos definiciones de* **iter**, **coll** *y* **vector** *similares a las del ejemplo 38 del capítulo anterior (pág. 83), el hecho que un iterador y un vector compartan la colección subyacente puede especificarse a través de*

$$\exists y \cdot \mathbf{iter}.i.n.es\ \langle *\!: \mathbf{coll}.y.es\rangle\ \mathbf{vector}.x.es$$

Esta fórmula especifica precisamente la situación, independientemente si **iter** *y/o* **vector** *utilizan memoria extra para su representación.*

4.2. Cálculo de fórmulas

De la misma manera que ocurre con la SL, la ShL valida las reglas de la lógica de predicados. En esta sección presentamos esquemas de axiomas para los nuevos operadores que subsumen y extienden aquellas presentadas en la sección 3.2. Como se muestra al final de la sección, el predicado **emp** tiene buenas propiedades que permiten satisfacer trivialmente cualquier condición de aplicación, facilitando de este modo la derivación de reglas para $*$ y $-\!\!*$ a partir de las aquí presentadas.

Lema 21 (Reglas de cálculo de ShL (cont.))**.** *Sean* $p,q,r,s \in Form'$ *fórmulas cualesquiera,* $x \in Var$*, entonces*

Conmutatividad:

$$\vDash p\ \langle *\!: r\rangle\ q \Leftrightarrow q\ \langle *\!: r\rangle\ p$$

Elemento neutro:

$$\vDash p\ \langle *\!: r\rangle\ r \Leftarrow p \wedge (r * \mathbf{true})$$

***Unsharing*:**

$$\vDash p\ \langle *\!: r\ \langle *\!: s\rangle\ t\rangle\ q \Rightarrow (t\ \langle -\!\circledast\!: s\rangle\ p)\ \langle *\!: r\rangle\ q$$

Distributividad con $\vee$:

$$\models (p \vee q)\ \langle *\colon s\rangle\ r \Leftrightarrow (p\ \langle *\colon s\rangle\ r) \vee (q\ \langle *\colon s\rangle\ r)$$

$$\models p\ \langle *\colon r \vee s\rangle\ q \Leftrightarrow (p\ \langle *\colon r\rangle\ q) \vee (p\ \langle *\colon s\rangle\ q)$$

Semidistributividad con $\wedge$:

$$\models (p \wedge q)\ \langle *\colon s\rangle\ r \Rightarrow (p\ \langle *\colon s\rangle\ r) \wedge (q\ \langle *\colon s\rangle\ r)$$

$$\models p\ \langle *\colon r \wedge s\rangle\ q \Rightarrow (p\ \langle *\colon r\rangle\ q) \wedge (p\ \langle *\colon s\rangle\ q)$$

Distributividad con $\exists$:

$$\frac{x \notin FV.r \cup FV.q}{\models (\exists x \cdot p)\ \langle *\colon r\rangle\ q \Leftrightarrow \exists x \cdot p\ \langle *\colon r\rangle\ q}$$

$$\frac{x \notin FV.p \cup FV.q}{\models p\ \langle *\colon \exists x \cdot r\rangle\ q \Leftrightarrow \exists x \cdot p\ \langle *\colon r\rangle\ q}$$

Semistributividad con $\forall$:

$$\frac{x \notin FV.r \cup FV.q}{\models (\forall x \cdot p)\ \langle *\colon r\rangle\ q \Rightarrow \forall x \cdot p\ \langle *\colon r\rangle\ q}$$

$$\frac{x \notin FV.p \cup FV.q}{\models p\ \langle *\colon \forall x \cdot r\rangle\ q \Rightarrow \forall x \cdot p\ \langle *\colon r\rangle\ q}$$

Los operadores espaciales generalizados mantienen la propiedad de monotonía y la relación de adjunción entre ellos.

Lema 22 (Reglas de cálculo de ShL (cont.)). *Sean $p, p', q, q', r, r', s \in Form'$ fórmulas cualesquiera, entonces*

Monotonía:

$$\frac{\models p \Rightarrow p' \qquad \models r \Rightarrow r' \qquad \models q \Rightarrow q'}{\models p\ \langle *\colon r\rangle\ q \Rightarrow p'\ \langle *\colon r'\rangle\ q'}$$

Adjuntividad:

$$\frac{\models p\ \langle *\colon s\rangle\ q \Rightarrow r}{\models p \Rightarrow (q\ \langle -\!\!*\colon s\rangle\ r)} \qquad \frac{\models p \Rightarrow (q\ \langle -\!\!*\colon s\rangle\ r)}{\models p\ \langle *\colon s\rangle\ q \Rightarrow r}$$

Otras clases de fórmulas

En la sección 3.2 introducimos diferentes clases de fórmulas de SL. Estas clases permiten, entre otras cosas, caracterizar aquellas fórmulas sobre las cuales aplican ciertas reglas de inferencia que no son válidas en general. Ejemplo de esto es la reducción de los operadores espaciales $*$ y $-\!\!*$ a la conjunción e implicación usuales ($\wedge$ y $\Rightarrow$) para las fórmulas *pure*, y la distributividad completa de $*$ con $\wedge$ para las fórmulas *precise*. Resultados análogos son válidos en nuestro marco de trabajo.

Lema 23 (Reglas de cálculo de ShL (cont.)). *Dadas $p, q, r, s \in Form'$ fórmulas cualesquiera, $x \in Var$, entonces*

$$\frac{p \in Pure \ o \ q \in Pure}{\models p \wedge q \wedge (r * \mathbf{true}) \Rightarrow p \ \langle *\colon r \rangle \ q}$$

$$\frac{p, q, r \in Pure}{\models p \ \langle *\colon r \rangle \ q \Rightarrow p \wedge q \wedge r}$$

$$\frac{s \in Pure}{\models (s \wedge p) \ \langle *\colon r \rangle \ q \Leftrightarrow s \wedge (p \ \langle *\colon r \rangle \ q)}$$

$$\frac{s \in Pure}{\models p \ \langle *\colon r \wedge s \rangle \ q \Leftrightarrow s \wedge (p \ \langle *\colon r \rangle \ q)}$$

$$\frac{p, r \in Pure}{\models (p \ \langle -\!\!*\colon r \rangle \ q) \Rightarrow ((p \wedge r) \Rightarrow q)}$$

$$\frac{p, r, q \in Pure}{\models ((p \wedge r) \Rightarrow q) \Rightarrow (p \ \langle -\!\!*\colon r \rangle \ q)}$$

$$\frac{r \in Precise}{\models (p \wedge q) \ \langle *\colon s \rangle \ r \Leftrightarrow (p \ \langle *\colon s \rangle \ r) \wedge (q \ \langle *\colon s \rangle \ r)}$$

$$\frac{p, q \in Precise}{\models p \ \langle *\colon r \wedge s \rangle \ q \Leftrightarrow (p \ \langle *\colon r \rangle \ q) \wedge (p \ \langle *\colon s \rangle \ q)}$$

$$\frac{q \in Precise \qquad x \notin FV.r \cup FV.q}{\models (\forall x \cdot p) \ \langle *\colon r \rangle \ q \Leftrightarrow \forall x \cdot p \ \langle *\colon r \rangle \ q}$$

$$\frac{p, q \in Precise \qquad x \notin FV.p \cup FV.q}{\models p \ \langle *\colon \forall x \cdot r \rangle \ q \Leftrightarrow \forall x \cdot p \ \langle *\colon r \rangle \ q}$$

Lo común entre las clases definidas en la sección 3.2 es que caracterizan fórmulas independientemente del contexto donde ellas aparecen. En la ShL, una fórmula r que ocurre en $p \langle *\colon r\rangle\ q$ o $p \langle -\!\!*\colon r\rangle\ q$ se encuentra *restringida* por las fórmulas p y q. En el contexto donde ocurre r, las porciones del *heap* que la satisfacen pueden estar caracterizados por p y/o q de forma más precisa que lo dado por la propia fórmula. De esta manera resulta interesante definir clases de fórmulas que satisfacen propiedades en relación a otras.

Podemos definir una familia de clases *precise in* para una fórmula q que generaliza el concepto de fórmulas *precise*. Intuitivamente p es *precise in* q si el dominio del *heap* que satisface p queda determinado dentro de cualquier *heap* que satisfaga q que lo contenga.

Definición 66 (Formula *precise in*)**.** *Dadas $p, q \in Form$, decimos que p es* precise in q *si para cualesquiera $s \in Stacks$, $h, h_1, h_2 \in Heaps$ tales que $h_1 \subseteq h$ y $h_2 \subseteq h$ se satisface que*

$$\textit{si } s, h \models q \textit{ y } s, h_1 \models p \textit{ y } s, h_2 \models p \textit{ entonces } dom.h_1 = dom.h_2$$

Con $Precise.q$ denotamos el mayor conjunto $P \subseteq Form'$ tal que p es precise in q *para toda $p \in P$.*

Lema 24. *$Precise = Precise.$***true***.*

Utilizando la clase *precise in* es posible definir una *regla de intercambio* que subsume la regla de asociatividad, y "completa" la regla de elemento neutro.

Lema 25 (Reglas de cálculo de ShL (cont.))**.** *Sean $p, q, r, s, t \in Form'$, entonces*

Elemento neutro *:*

$$\frac{r \in Precise.(p \langle *\colon r\rangle\ r)}{\models p \langle *\colon r\rangle\ r \Rightarrow p \wedge (r * \mathbf{true})}$$

Intercambio:

$$\frac{\models q \Rightarrow t * \mathbf{true} \qquad\qquad t \in Precise.(p \langle *\colon s\rangle\ q)}{\models (p \langle *\colon s\rangle\ q) \langle *\colon t\rangle\ r \Rightarrow p \langle *\colon s\rangle (q \langle *\colon t\rangle\ r)}$$

En la sección anterior mencionamos que $p \langle *\colon r\rangle\ q$ puede aproximarse mediante el patrón *deletion followed by extension*. Esta aproximación se convierte en una equivalencia cuando la caracterización de la porción del modelo compartida dada por r es suficientemente específica, de manera que cualquier *heap* que satisfaga r sirve para la satisfacción de p y q. Caracterizamos esta situación con la familia de fórmulas *sufficent for*.

Definición 67 (Fórmula *sufficient for*). *Dadas $p, q \in Form'$, decimos que p es* sufficient for *q si para cualesquiera $s \in Stacks$, $h, h_1, h_2 \in Heaps$ tales que $h \perp h_1$ y $h \perp h_2$ se satisface que*

$$\text{si } s, h \cup h_1 \models q \text{ y } s, h_1 \models p \text{ y } s, h_2 \models p \text{ entonces } s, h \cup h_2 \models q$$

Con $Sufficient.q$ denotamos el mayor conjunto $S \subseteq Form'$ tal que p es sufficient for *q para toda $p \in S$.*

El lema 20 en conjunto con el siguiente caracterizan las condiciones para reducir el operador $\langle *: \rangle$ a una fórmula de SL.

Lema 26 (Reglas de cálculo de ShL (cont.)). *Sean $p, q \in Form'$, entonces*

$$\frac{p \in Sufficient.q}{\models p \ \langle -\!\circledast : r \rangle \ q) \Rightarrow (p \ \langle -\!\!* : r \rangle \ q)}$$

para cualquier $r \in Form'$. Más aún, si $\models p \ \langle -\!\circledast : r \rangle \ q) \Rightarrow (p \ \langle -\!\!* : r \rangle \ q)$ para alguna r, entonces $(p \wedge (r * \mathbf{true})) \in Sufficient.q$.

Corolario 1 (Reducción de ShL a SL). *Sean $p, q, r \in Form'$ tales que $r \in Sufficient.p$ y $r \in Sufficient.q$. Entonces las siguientes son equivalentes:*

- $p \ \langle *: r \rangle \ q$
- $(r -\!\!* \ p) * r * (r -\!\!* \ q)$
- $(r -\!\circledast \ p) * r * (r -\!\circledast \ q)$

A excepción de las fórmulas *pure*, ninguna de las clase puede determinarse completamente de manera sintáctica. Sin embargo es posible dar criterios de suficiencia para todas ellas. Los siguientes lemas nos permiten reconocer en diversos casos la pertenencia de una fórmula a una clase.

Lema 27. *Sean $p, q, r \in Form'$ fórmulas cualesquiera, $e, e' \in Expr$ expresiones cualesquiera, y $\mathbb{C}$ alguno de los conjuntos Pure, Strict, Domain, Precise, Precise.s o Sufficient.s, para $s \in Form'$. Entonces*

1. $\mathbf{emp}, e \mapsto e' \in Strict$.
2. $e \mapsto _ \in Domain$.
3. *Si $p, q \in \mathbb{C}$, entonces $p \ \langle *: r \rangle \ q \in \mathbb{C}$.*
4. *Si $p \in \mathbb{C}$ o $q \in C$, entonces $p \wedge q \in \mathbb{C}$.*
5. *Si $p \in \mathbb{C}$ y $\models q \Rightarrow p$, entonces $q \in \mathbb{C}$.*

6. *Strict* $\subset$ *Domain* $\subset$ *Precise* $\subset$ *Precise.s*, *para cualquier* $s \in Form'$.

7. *Strict* $\subset$ *Sufficient.s*, *para cualquier* $s \in Form'$.

Lema 28. *Sean* $p, q, r, s \in Form'$ *fórmulas cualesquiera,* $e \in Expr$ *una expresión cualquiera. Entonces*

1. $p \in Precise.\mathbf{emp}$.

2. $p \in Precise.(e \mapsto _)$.

3. *Si* $\models p * \mathbf{true} \Rightarrow \neg q$, *entonces* $p \in Precise.q$.

4. *Si* $p \in Precise.q$ *y* $\models p * \mathbf{true} \Rightarrow \neg r$, *entonces* $p \in Precise.(q \langle *: s \rangle\ r)$ *para cualquier* $s \in Form'$.

5. *Si* $p \in Precise.q$ *o* $p \in Precise.r$, *entonces* $p \in Precise.(q \wedge r)$.

6. *Si* $p \in Precise.q$ *y* $p \in Precise.r$, *entonces* $p \in Precise.(q \vee r)$.

7. *Si* $p \in Precise.r$ *y* $\models q \Rightarrow r$, *entonces* $p \in Precise.q$.

Lema 29. *Sean* $p, q, r, s \in Form'$ *fórmulas cualesquiera. Entonces*

1. $p \in Sufficient.\mathbf{emp}$.

2. $p \in Sufficient.(_ \mapsto _)$.

3. *Si* $\models p * \mathbf{true} \Rightarrow \neg q$, *entonces* $p \in Sufficient.q$.

4. *Si* $p \in Sufficient.q$, $p \in Sufficient.r$ *y* $p \in Sufficient.s$, *entonces* $p \in Sufficient.(q \langle *: s \rangle\ r)$, $p \in Sufficient.(q \wedge r)$ *y* $p \in Sufficient.(q \vee r)$.

5. *Si* $p \in Sufficient.q$ *y* $\models q \Rightarrow r$, *entonces* $p \in Sufficient.r$.

4.3. Sistema deductivo y reglas de *frame*

Podemos utilizar la ShL para razonar sobre programas que manipulan la memoria dinámica de la misma manera que la SL, tomando su lenguaje de comandos (def. 15), su forma de especificación (def. 20) y sus reglas deductivas (def. 22, 24, 25, 26, 27 y 28). Denotamos a este sistema deductivo como ShL^H. En particular, en virtud del lema 19 (pág. 96), la regla de *frame* (def. 26, pág. 59) continúa siendo válida en nuestro marco de trabajo. Sin embargo, la introducción del nuevo operador de separación espacial $\langle *: \rangle$ nos obliga a introducir nuevas reglas que permitan razonar de forma composicional estableciendo ciertas garantías sobre cómo las posibles modificaciones al estado compartido afectan al estado global.

En [49] introducimos una regla de *frame* general, motivada por la siguiente idea: es posible garantizar la validez de un invariante i sobre el estado global, aunque dependa del estado modificado por un comando c, siempre y cuando se garantice que la modificación al *heap* compartido no altera el invariante. Más aún, no es necesario que la fórmula i sea invariante, si se puede deducir cómo cambia el estado, ahora especificado por i', a lo largo de la ejecución de c.

Definición 68 (Sistema deductivo de ShL^H)**.**

Regla de *frame* general (GFR):

$$\frac{\vdash \{p\}\ c\ \{q\} \qquad (FV.r \cup FV.i) \cap Mod.c = \emptyset \qquad \models (r \multimap\!\!\circledast\ i) * r' \Rightarrow i' \qquad \models q \Rightarrow r' * \mathbf{true}}{\vdash \{p\ \langle *\colon r\rangle\ i\}\ c\ \{q\ \langle *\colon r'\rangle\ i'\}}$$

donde $p, q, r, r', i, i' \in Form'$*, y* $c \in Comm$*.*

Aquí el *subheap* compartido, que puede cambiar con la ejecución de c, está dado por las fórmulas r y r'. La condición $(r \multimap\!\!\circledast\ i) * r' \Rightarrow i'$, que sigue el patron *deletion followed by extension*, garantiza que la (posible) modificación de la región compartida altera el estado global estableciendo i'. La condición $q \Rightarrow r' * \mathbf{true}$ excluye el caso inconsistente en que no existe un *subheap* de q que sastifaga r'. Es elemental ver que tomando $r = \mathbf{emp}$ se obtiene la regla de *frame* original.

Ejemplo 44. *Consideremos la siguiente terna, justificada por la regla de especificación local para el comando de mutación:*

$$\vdash \begin{array}{l} \{x \mapsto _\} \\ \quad x.0 := \mathbf{null} \\ \{x \mapsto \mathbf{null}\} \end{array}$$

Es sencillo ver que se puede deducir

$$\vdash \begin{array}{l} \{x \mapsto _\ \langle *\colon x \mapsto _\rangle\ y \mapsto _\} \\ \quad x.0 := \mathbf{null} \\ \{x \mapsto \mathbf{null}\ \langle *\colon x \mapsto \mathbf{null}\rangle\ y \mapsto \mathbf{null}\} \end{array}$$

ya que $x \mapsto _ \multimap\!\!\circledast\ y \mapsto _ \Leftrightarrow \mathbf{emp} \wedge x = y$*, luego* $x = y \wedge x \mapsto \mathbf{null} \Rightarrow y \mapsto \mathbf{null}$. *Además la condición* $x \mapsto \mathbf{null} \Rightarrow x \mapsto \mathbf{null} * \mathbf{true}$ *se satiface trivialmente.*

El problema con esta regla es que la caracterización de las modificaciones que ocurren al estado global están dadas en términos de la especificación del estado compartido. Y como vimos en el ejemplo 40 (pág. 95), el estado compartido puede estar subespecificado.

Ejemplo 45. *Si en el ejemplo anterior tomamos r y r' de la GFR como $_ \mapsto _$, lo máximo que podemos deducir es*

$$\begin{array}{ll} & \{\, x \mapsto _ \;\langle *\!: _ \mapsto _ \rangle\; y \mapsto _ \,\} \\ \vdash & \quad x.0 := \mathbf{null} \\ & \{\, x \mapsto \mathbf{null} \;\langle *\!: _ \mapsto _ \rangle\; _ \mapsto _ \,\} \end{array}$$

La precondición es equivalente a la del mencionado ejemplo, donde sabemos que $x = y$, información que se pierde en la postcondición. Peor es el caso en que tomamos r y r' de la GFR como $\neg\mathbf{emp}$, donde obtenemos

$$\begin{array}{ll} & \{\, x \mapsto _ \;\langle *\!: \neg\mathbf{emp} \rangle\; y \mapsto _ \,\} \\ \vdash & \quad x.0 := \mathbf{null} \\ & \{\, x \mapsto \mathbf{null} \;\langle *\!: \neg\mathbf{emp} \rangle\; \neg\mathbf{emp} \,\} \end{array}$$

Por los ejemplos anteriores podemos decir que las condiciones de aplicación de la GFR pueden resultar *débiles* en algunos casos, porque no se utiliza la información contenida en la pre y postcondición sobre el estado global. Sin embargo esto puede llegar a salvarse con una *correcta* elección de las fórmulas que caracterizan el estado compartido (aunque puede no ser posible en todos los casos).

Por otro lado, las condiciones pueden resultar muy *fuertes*, ya que utilizamos el patrón *deletion followed by extension* con el operador $-\!\circledast$, y en situaciones de aplicación simétrica de la GFR, implica la reducción completa del operador $\langle *\!: \rangle$ a una combinación de $-\!\circledast$ y $*$.

Ejemplo 46. *Supongamos que queremos verificar*

$$\begin{array}{ll} & \{\, p \;\langle *\!: r \rangle\; p' \,\} \\ \vdash & \quad c; \\ & \quad c' \\ & \{\, p \;\langle *\!: r \rangle\; p' \,\} \end{array}$$

a partir de $\vdash \{\, p \,\}\; c\; \{\, p \,\}$ y $\vdash \{\, p' \,\}\; c'\; \{\, p' \,\}$. Para simplificar la presentación, supongamos que $Mod.c = Mod.c' = \varnothing$. Para aplicar la GFR en primer lugar, debe suceder que

1. $\models (r -\!\circledast\, p') * r \Rightarrow p'$, *y*

2. $\models p \Rightarrow r * \mathbf{true}$.

Para la segunda aplicación de la GFR, debe suceder que

3. $\models (r -\!\circledast\, p) * r \Rightarrow p$, *y*

4. $\models p' \Rightarrow r * \mathbf{true}$.

A partir de las condiciones 1 *y* 3*, aplicando la regla de adjuntividad, se deduce que* $r \in Sufficient.p$ *y* $r \in Sufficient.q$*. Por el corolario 1, esto significa que sólo podemos aplicar la GFR de forma tán simétrica cuando la fórmula que caracteriza el estado global es reductible a la SL.*

Aquí hemos llevado al extremo las especificaciones de c *y* c' *utilizando* p *y* p' *invariantes, de forma que sea evidente la forma de reducir* $\langle *\colon \rangle$ *a la SL estándar. Sin embargo el resultado sigue siendo válido para casos particulares de especificaciones no invariantes.*

Para evitar estos problemas podemos formular una nueva regla de *frame*, que caracteriza la forma en que cambia el estado global, ya no a través de la especificación del estado compartido, sino del mismo estado global.

Definición 69 (Sistema deductivo de ShL^H (cont.))**.**

Regla de *specified frame* (SFR):

$$\frac{\vdash \{p\}\ c\ \{q\} \qquad \{\vec{x}\} = Mod.c \qquad \vec{x}' \notin (FV.p \cup FV.r \cup FV.i) \qquad \models (\exists \vec{x}' \cdot (p \multimap\!\circledast (p\ \langle *\colon r\rangle\ i))_{/\vec{x}\leftarrow\vec{x}'}) * q \Rightarrow q\ \langle *\colon r'\rangle\ i'}{\vdash \{p\ \langle *\colon r\rangle\ i\}\ c\ \{q\ \langle *\colon r'\rangle\ i'\}}$$

donde $\vec{x}, \vec{x}' \in Var^+$, $p, q, r, r', i, i' \in Form'$, *y* $c \in Comm$.

La regla sigue una idea análoga a la GFR. Pero el uso del patrón *deletion followed by extension* con p y q evita el potencial problema de subespecificación del estado compartido por r. Esto permite, por un lado, demostrar postcondiciones más fuertes, y por otro, aplicar la SFR de fórma simétrica sin obligar a que r sea tan *suficientemente específica* como para que la fórmula sea expresable en SL (como en el ejemplo 46).

Lema 30. *Sean* $p, q, r, i \in Form'$ *y* $c \in Comm$ *tales que* $\vdash \{p\}\ c\ \{q\}$, *con* $\{\vec{x}\} = Mod.c$. *Si* $p \in Precise.(p\ \langle *\colon r\rangle\ q)$ *y* $\vec{x}' \notin (FV.r \cup FV.i)$, *entonces*

$$(\exists \vec{x}' \cdot (p \multimap\!\circledast (p\ \langle *\colon r\rangle\ i))_{/\vec{x}\leftarrow\vec{x}'}) * q$$

es la postcondición mas fuerte *para* c *respecto a la precondición* $p\ \langle *\colon r\rangle\ q$.

Las siguientes reglas de inferencia facilitan el cálculo de la condición de aplicación de la SFR.

Lema 31 (Reglas de cálculo de ShL (cont.))**.** *Sean* $p, p', q, r \in Form'$, *entonces*

Absorbente a izquierda:

$$\models \mathbf{emp}\ \langle \multimap\!\circledast\colon r\rangle\ q \Leftrightarrow (r \Leftrightarrow \mathbf{emp}) \wedge q$$

Absorbente a derecha:

$$\models p\ \langle \multimap\!\circledast\colon r\rangle\ \mathbf{emp} \Leftrightarrow (p \Leftrightarrow \mathbf{emp}) \wedge (r \Leftrightarrow \mathbf{emp})$$

Currificación:

$$\vDash (p \ \langle *\colon r\rangle \ p') \ \langle -\!\circledast\colon r\rangle \ q \Rightarrow p \ \langle -\!\circledast\colon r\rangle \ (p' \ \langle -\!\circledast\colon r\rangle \ q)$$

Eliminación precisa:

$$\frac{p \in Precise.(p \ \langle *\colon r\rangle \ q)}{p \ \langle -\!\circledast\colon r\rangle \ (p \ \langle *\colon r\rangle \ q) \Rightarrow q}$$

Las condiciones de aplicación de la SFR (y la GFR) pueden resultar demasiado estrictas cuando la precondición no es *precise*, particularmente respecto a comandos que no modifican el *heap*. En el caso de la asignación esto no es un problema práctico, ya que su regla es global. Para el caso del comando de consulta, podemos utilizar la siguiente regla.

Definición 70 (Sistema deductivo de ShL^H (cont.))**.**

Regla de *invariant frame* (IFR):

$$\frac{\vdash \{p\} \ c \ \{q\} \qquad (FV.r \cup FV.i) \cap Mod.c = \varnothing}{\vdash \{p \ \langle *\colon r\rangle \ i\} \ c \ \{q \ \langle *\colon r\rangle \ i\}}$$

donde $p, q, r, r', i, i' \in Form'$, y $c \in Comm$ no incluye los comandos de construcción, mutación ni destrucción.

Teorema 3 (Consistencia del sistema deductivo de ShL^H)**.** *Dados $p, q \in Form'$, $c \in Comm$, si $\vdash \{p\} \ c \ \{q\}$ se deduce utilizando las reglas del sistema deductivo,*[4] *entonces $\vDash \{p\} \ c \ \{q\}$.*

Ejemplo 47. *Retomemos los ejemplos 44 y 45. A partir de la especificación*

$$\vdash \begin{array}{l} \{x \mapsto _\} \\ \quad x.0 := \mathbf{null} \\ \{x \mapsto \mathbf{null}\} \end{array}$$

y el uso de la SFR es posible derivar

$$\vdash \begin{array}{l} \{x \mapsto _ \ \langle *\colon \neg\mathbf{emp}\rangle \ y \mapsto _\} \\ \quad x.0 := \mathbf{null} \\ \{x \mapsto \mathbf{null} \ \langle *\colon \neg\mathbf{emp}\rangle \ y \mapsto \mathbf{null}\} \end{array}$$

ya que la postcondición es una consecuencia lógica de

$$(x \mapsto _ \ -\!\circledast \ (x \mapsto _ \ \langle *\colon \neg\mathbf{emp}\rangle \ y \mapsto _)) * x \mapsto \mathbf{null}$$

puesto que $\vDash (x \mapsto _ \ -\!\circledast \ (x \mapsto _ \ \langle *\colon \neg\mathbf{emp}\rangle \ y \mapsto _)) \Leftrightarrow \mathbf{emp} \wedge x = y$.

```
   { btree.x.⟨lt,e,rt⟩ ∧ bst.⟨lt,e,rt⟩ }
     y := x;
     until y ≠ null ∧ w ≠ v do
         w := y.1;
         z := y;
         if v ⩽ w then
             y := y.0;
         else
             y := y.2;
         fi
⊢    od;
     if v < w then
         y := cons(null, v, null);
         z.0 := y
     fi;
     if v > w then
         y := cons(null, v, null);
         z.2 := y
     fi;
   { btree.x.(ins.v.⟨lt,e,rt⟩) ∧ bst.(ins.v.⟨lt,e,rt⟩) }
```

$$ins \in Values \to \langle Values \rangle \to \langle Values \rangle$$

$$ins.v.\langle\ \rangle \doteq \langle v \rangle$$

$$ins.v.\langle lt, v', rt \rangle \doteq \begin{cases} \langle ins.v.lt, v', rt \rangle & \text{si } v < v' \\ \langle lt, v', rt \rangle & \text{si } v = v' \\ \langle lt, v', ins.v.rt \rangle & \text{si } v > v' \end{cases}$$

$$gbst \in Values \to (Values \to Values \to \{true, false\}) \to \langle Values \rangle \to \{true, false\}$$

$$gbst.v.rel.\langle\ \rangle \doteq true$$

$$gbst.v.rel.\langle lt, e, rt \rangle \doteq rel.v.e \wedge gbst.v.rel.lt \wedge gbst.v.rel.rt$$

$$bst \in \langle Values \rangle \to \{true, false\}$$

$$bst.\langle\ \rangle \doteq true$$

$$bst.\langle lt, v, rt \rangle \doteq gbst.v.(<).lt \wedge gbst.v.(>).rt \wedge bst.lt \wedge bst.rt$$

Figura 4.8: Algoritmo de inserción en un árbol binario de búsqueda.

Ejemplo 48. *Tomemos el problema de insertar un nodo en un árbol binario de búsqueda. Si bien la especificación misma del problema no involucra estructuras de datos que comparten el* heap, *utilizamos el poder expresivo de* $\langle *\colon\ \rangle$ *para especificar los estados intermedios de la computación, intentando obtener una prueba que se ajuste a las intuiciones que subyacen a la especificación recursiva del algoritmo. De esta manera evitamos la necesidad, que impondría el estilo de especificación de la SL (en término de partes siempre disjuntas), de definir un predicado que caracterice un árbol parcial, a la manera del predicado* **lseg**.

Retomemos el predicado **btree** *(def. 34, pág. 70). La especificación del problema se encuentra en la figura 4.8, donde utilizamos* **until** b **do** c **od** *como* syntactic sugar *de* c; **while** b **do** c **od**, *y la función bst para definir recursivamente la propiedad de ordenación característica. La idea del algoritmo es muy sencilla: el ciclo explora el árbol en busca del punto de inserción; una vez hallado, se reemplaza el subárbol vacío por el árbol unitario correspondiente. Un invariante para el ciclo es el siguiente:*

$$\begin{aligned}&\exists t, yt \cdot (((v < w \wedge z \mapsto y, w, t \,*\, \mathbf{btree}.y.yt)\ \langle *\colon z \mapsto _,_,_ \,*\, \mathbf{btree}.y.yt\rangle\ \mathbf{btree}.x.\langle lt, e, rt\rangle)\\&\quad \vee ((v > w \wedge z \mapsto t, w, y \,*\, \mathbf{btree}.y.yt)\ \langle *\colon z \mapsto _,_,_ \,*\, \mathbf{btree}.y.yt\rangle\ \mathbf{btree}.x.\langle lt, e, rt\rangle))\\&\quad \wedge bst.\langle lt, e, rt\rangle\end{aligned}$$

La demostración del cuerpo del ciclo se puede llevar adelante sobre la base de un razonamiento local sobre los términos $z \mapsto \ldots \,*\, \mathbf{btree}.y.yt$ *con el* unfolding *de* **btree**, *aplicando la IFR (evitando la condición sobre las variables modificadas de forma similar al ejemplo 25 (pág. 59)), y simplicando la postcondición con la regla del elemento neutro.*

En lo que respecta a la utilidad de nuestro sistema de prueba, la porción de código más interesante es la posterior al ciclo, cuando vale que $w \neq v$. *Allí se modifica el* heap *creando un nuevo nodo para el elemento a insertar. En la figura 4.9 presentamos el esquema de prueba para el caso* $v < w$, *siendo el caso* $v > w$ *completamente simétrico. Denotamos la aplicación explícita de alguna regla de* frame *con una barra vertical. La aplicación de la SFR requiere que la siguiente fórmula sea válida:*

$$\begin{aligned}&\quad ((v < w \wedge z \mapsto \mathbf{null}, w, t)\\&\quad -\!\circledast ((v < w \wedge z \mapsto \mathbf{null}, w, t)\ \langle *\colon z \mapsto _,_,_\rangle \,*\, \mathbf{btree}.x.\langle lt, e, rt\rangle) \wedge bstr.\langle lt, e, rt\rangle)\\&\quad * (v < w \wedge z \mapsto y, w, t \,*\, \mathbf{btree}.y.(ins.v.\langle\ \rangle))\\&\Rightarrow ((v < w \wedge z \mapsto y, w, t \,*\, \mathbf{btree}.y.(ins.v.\langle\ \rangle))\\&\quad \langle *\colon z \mapsto _,_,_ \,*\, \mathbf{btree}.y.(ins.v.\langle\ \rangle)\rangle\ \mathbf{btree}.x.(ins.v.\langle lt, e, rt\rangle))\\&\quad \wedge bst.(ins.v.\langle lt, e, rt\rangle)\end{aligned}$$

Esto es un caso particular del siguiente resultado, que se demuestra semánticamente por inducción:

[4]Definiciones 22, 24, 25, 26, 27, 28, 68, 69 y 70.

Proposición 1. *Sean $xt, lt, rt \in Tree$, $e, v, w, l, r \in Expr$ y $x, y \in Var$. Entonces*

$$\begin{aligned} \vDash ((y \mapsto l, w, r \,*\, \mathbf{btree}.l.\langle lt, e, rt\rangle) \wedge v < w \wedge bst.\langle lt, e, rt\rangle \multimap\!\!\circledast\; \mathbf{btree}.x.xt \wedge bstr.xt) \\ *\; y \mapsto l, w, r \,*\, \mathbf{btree}.y.(ins.v.\langle lt, e, rt\rangle) \;\Rightarrow\; \mathbf{btree}.x.(ins.v.xt) \wedge bst.(ins.v.xt) \end{aligned}$$

4.4. Tipos abstractos de datos con *sharing*

La extensión de la SL presentada en la sección 3.6 soporta tipos abstractos de datos, incorporando funciones y predicados de abstracción para dar cuenta del principio de *information hiding*. Los detalles de implementación dados por la definición de un predicado de abstracción sólo están disponibles para la verificación de las funciones. Para cualquier demostración sobre el código cliente, que sólo puede manipular las instancias de un tipo abstracto de datos a partir de las funciones provistas, unicamente pueden utilizarse las especificaciones de las mismas. Consideremos un sistema deductivo como el de SL_T^H, pero que use el lenguaje de fórmulas de la ShL, permitiendo la especificación de *sharing* entre instancias de tipos abstractos de datos. La verificación de un código cliente podría requerir, como es usual, el uso intensivo de las reglas de *frame*, que dependen de la validicación de ciertas condiciones que típicamente involucrarían los predicados de abstracción. Pero tales obligaciones no podrían ser demostradas en el contexto del código cliente, donde no se cuenta con sus definiciones. Para lidiar con este problema es necesario caracterizar un tipo abstracto, no sólo con la especificación de sus funciones, sino también con ciertos *axiomas* que permitan resolver las condiciones de aplicación de las reglas de *frame*.

En esta sección presentamos una extensión a la ShL^H completamente análoga a la SL_T^H, que denominamos ShL_T^H, que incorpora predicados de abstracción a las fórmulas de ShL (con una sintaxis y semántica análoga a la de la sección 3.6, que denominamos ShL_T[5]), utiliza el mismo lenguaje de programación (def. 41), y un sistema deductivo similar basado en una extensión del concepto de especificación (def. 47) para razonar modularmente sobre instancias de tipos abstractos de datos en presencia de un (posible) *sharing* en su implementación.

Especificación y razonamiento sobre TAD's con sharing

Extendemos el concepto de especificación agregando un entorno de *especificaciones estáticas*, formado por fórmulas de la misma lógica que usualmente proveen una caracterización adicional de los predicados de abstracción y su interacción.

[5]Al conjunto de fórmulas lo denominamos $Form_T'$

$\{\, ((v < w \land z \mapsto y, w, t \, * \, \mathbf{btree}.y.yt)$
$\langle *\!: z \mapsto _,_,_ \, * \, \mathbf{btree}.y.yt\rangle \; \mathbf{btree}.x.\langle lt, e, rt\rangle) \land bst.\langle lt, e, rt\rangle \land y = \mathbf{null} \,\}$
$\Rightarrow \langle$ distributividad con $y = \mathbf{null} \in Pure$ $\rangle$
$\{\, (((v < w \land y = \mathbf{null} \land z \mapsto y, w, t) \, * \, (\mathbf{btree}.y.yt \land y = \mathbf{null}))$
$\langle *\!: z \mapsto _,_,_ \, * \, (\mathbf{btree}.y.yt \land y = \mathbf{null})\rangle \; \mathbf{btree}.x.\langle lt, e, rt\rangle)$
$\land \, bst.\langle lt, e, rt\rangle \,\}$
$\Rightarrow \langle$ caso base de definición de **btree**, reemplazo de y por **null** $\rangle$
$\{\, ((v < w \land z \mapsto \mathbf{null}, w, t \, * \, \mathbf{emp}) \; \langle *\!: z \mapsto _,_,_ \, * \, \mathbf{emp}\rangle \; \mathbf{btree}.x.\langle lt, e, rt\rangle)$
$\land \, bst.\langle lt, e, rt\rangle \,\}$
$\Rightarrow \langle$ elemento neutro $\rangle$
$\{\, ((v < w \land z \mapsto \mathbf{null}, w, t) \; \langle *\!: z \mapsto _,_,_\rangle \; \mathbf{btree}.x.\langle lt, e, rt\rangle)$
$\land \, bst.\langle lt, e, rt\rangle \,\}$

SFR:
| $\{\, v < w \land z \mapsto \mathbf{null}, w, t \,\}$
| $y := \mathbf{cons}(\mathbf{null}, v, \mathbf{null});$
| $\{\, v < w \land z \mapsto \mathbf{null}, w, t \, * \, y \mapsto \mathbf{null}, v, \mathbf{null} \,\}$
| $z.0 := y;$
| $\{\, v < w \land z \mapsto y, w, t \, * \, y \mapsto \mathbf{null}, v, \mathbf{null} \,\}$
| $\Rightarrow \langle$ definición de **btree** $\rangle$
| $\{\, v < w \land z \mapsto y, w, t \, * \, \mathbf{btree}.y.\langle v\rangle \,\}$
| $\Rightarrow \langle$ definición de ins; $\rangle$
| $\{\, v < w \land z \mapsto y, w, t \, * \, \mathbf{btree}.y.(ins.v.\langle\, \rangle) \,\}$

$\{\, ((v < w \land z \mapsto y, w, t \, * \, \mathbf{btree}.y.(ins.v.\langle\, \rangle))$
$\langle *\!: z \mapsto _,_,_ \, * \, \mathbf{btree}.y.(ins.v.\langle\, \rangle)\rangle \; \mathbf{btree}.x.(ins.v.\langle lt, e, rt\rangle))$
$\land \, bst.(ins.v.\langle lt, e, rt\rangle) \,\}$
$\Rightarrow \langle$ introducción de $\exists$ sobre y y w, debilitamiento $\rangle$
$\{\, ((z \mapsto _,_,_ \, * \, \mathbf{btree}.y.(ins.v.\langle\, \rangle))$
$\langle *\!: z \mapsto _,_,_ \, * \, \mathbf{btree}.y.(ins.v.\langle\, \rangle)\rangle \; \mathbf{btree}.x.(ins.v.\langle lt, e, rt\rangle))$
$\land \, bst.(ins.v.\langle lt, e, rt\rangle) \,\}$
$\Rightarrow \langle$ elemento neutro $\rangle$
$\{\, (z \mapsto _,_,_ \, * \, \mathbf{btree}.y.(ins.v.\langle\, \rangle) \, * \, \mathbf{true}) \land \mathbf{btree}.x.(ins.v.\langle lt, e, rt\rangle)$
$\land \, bst.(ins.v.\langle lt, e, rt\rangle) \,\}$
$\Rightarrow \langle$ debilitamiento $\rangle$
$\{\, \mathbf{btree}.x.(ins.v.\langle lt, e, rt\rangle) \land bst.(ins.v.\langle lt, e, rt\rangle) \,\}$

Figura 4.9: Demostración (parcial) del algoritmo de inserción en un árbol binario de búsqueda

Definición 71 (Especificación de ShL_T^H). *El conjunto $Spec'_T$ de especificaciones de comandos está dado por la siguiente gramática:*

$$\begin{aligned} Env_a \ni \Lambda &::= \epsilon \mid a(\vec{x}) \doteq p \mid \Lambda, \Lambda \\ Env_f \ni \Phi &::= \epsilon \mid \{p\}\, f(\vec{x})\, \{p\} \mid \Phi, \Phi \\ Env_s \ni \Sigma &::= \epsilon \mid p \mid \Sigma, \Sigma \\ Spec'_T \ni s &::= \ \Lambda; \Phi; \Sigma \vdash \{p\}\, c\, \{p\} \end{aligned}$$

donde ϵ denota la secuencia vacía, $\vec{x} \in Var^+$, $a \in A$, $f \in F$, $p \in Form_T{}'$ y $c \in Comm_T$.

Adoptamos todo el marco de trabajo presentado en la sección 3.6: el lenguaje de fórmulas extendido con predicados de abstracción y su semántica, el lenguaje de programación y su semántica, y la semántica de los entornos de predicados y de funciones. Pero la introducción de las especificaciones estáticas requiere modificar la noción de validez de una especificación apropiadadmente.

Definición 72 (Validez de especificación de ShL_T). *Dada una especificación $\Lambda; \Phi; \Sigma \vdash \{p\}\, c\, \{q\} \in Spec'_T$, decimos que es* valida*, denotada como $\Lambda; \Phi; \Sigma \vDash \{p\}\, c\, \{q\}$ si para todo $\alpha \in close.\Lambda$ y $\phi \in Fun$ bien formado*

$$si\ \alpha, \phi \vDash \Phi\ y\ \alpha \vDash r\ para\ toda\ r \in \Sigma,\ entonces\ \alpha, \phi \vDash \{p\}\, c\, \{q\}$$

Las especificaciones estáticas permiten realizar deducciones válidas en el código cliente sin conocer los detalles de la implementación de los predicados abstractos. Se espera, en particular, que sean útiles para resolver las hipótesis de las reglas de *frame* (GFR, SFR) frente a la existencia de *sharing*. Cabe remarcar que las especificaciones estáticas abren la posibilidad de exponer la representación del tipo abstracto, rompiendo la barrera de la abstracción. Esto es un error de diseño que efectivamente también puede suceder con las especificaciones de las funciones. Sin embargo, bien utilizadas, posibilitan el razonamiento sobre las instancias de los tipos abstractos de datos sin romper el principio de *information hiding*.

Para formalizar las posibilidades deductivas es necesario modificar la noción de *validez* de una fórmula p, respecto a un entorno de predicados Λ y un entorno de especificaciones estáticas Σ. La intuición es que p debe satisfacerse en cualquier estado y cualquier solución a las definiciones dadas en Λ, suponiendo válidas todas las especificaciones de Σ.

Definición 73 (Validez respecto a un entorno de predicados y de especificaciones estáticas). *Dados $\Lambda \in Env_a$, $\Sigma \in Env_s$ y $p \in Form_T{}'$, decimos que p es* válida *respecto a Λ y Σ, denotado como $\Lambda; \Sigma \vDash p$, cuando*

$$si\ \Lambda \vDash q\ para\ toda\ q \in \Sigma, entonces\ s, h, s', \alpha \vDash p$$

para todo estado $s, h \in States$, todo stack *lógico $s' \in Stacks'$ y toda solución $\alpha \in close.\Lambda$.*

Como sucede con la SL_T, todas las reglas de cálculo de ShL extienden trivialmente a la nueva definición de validez. Lo mismo ocurre con las regla de *fold* y *unfold* sobre los predicados de abstracción (def. 15, pág. 81). Pero la nueva forma de validez permite introducir la siguiente regla que caracteriza cómo utilizar una especificación estática en el razonamiento sobre programas.

Lema 32 (Reglas de cálculo de ShL_T). *Sea $p \in Form_T{}'$ una fórmula con variables libres $\vec{x} \in (Var \cup Var')^+$, entonces*

$$\Lambda; p, \Sigma \vDash p_{/\vec{x} \leftarrow \vec{e}}$$

donde $\vec{e} \in (Expr \cup Expr')^+$.

Su ámbito de aplicación usual es la regla de consecuencia, que debe ser reformulada de forma acorde.

Definición 74 (Sistema deductivo de ShL_T^H). *La siguiente es la adaptación de la* ***regla de consecuencia*** *a la nueva forma de validez y de especificación:*

$$\frac{\Lambda; \Sigma \vDash p' \Rightarrow p \qquad \Lambda; \Phi; \Sigma \vdash \{\,p\,\}\; c\; \{\,q\,\} \qquad \Lambda; \Sigma \vDash q \Rightarrow q'}{\Lambda; \Phi; \Sigma \vdash \{\,p'\,\}\; c\; \{\,q'\,\}}$$

donde $p, p', q, q' \in Form_T{}'$, $c \in Comm_T$, $\Lambda \in Env_a$, $\Phi \in Env_f$ y $\Sigma \in Env_s$.

La regla de definición de funciones, que captura la noción de tipo abstracto de datos, debe adaptarse para incorporar el uso de las especificaciones estáticas.

Definición 75 (Sistema deductivo de ShL_T^H (cont.)). *La siguiente regla es la adaptación de la regla de* ***definición de funciones*** *a la nueva forma de especificación:*

$$\frac{\begin{array}{c} \Lambda, \Lambda'; \Sigma \vDash r \textit{ para toda } r \in \Sigma' \\ \Lambda, \Lambda'; \Phi; \Sigma, \Sigma' \vdash \{\,p_1\,\}\; c_1\; \{\,q_1\,\} \\ \vdots \\ \Lambda, \Lambda'; \Phi; \Sigma, \Sigma' \vdash \{\,p_n\,\}\; c_n\; \{\,q_n\,\} \\ \Lambda; \{\,p_1\,\}\; f_1(\vec{x}_1)\; \{\,q_1\,\}, \ldots, \{\,p_n(\vec{x}_n)\,\}\; f_n\; \{\,q_n\,\}, \Phi; \Sigma; \Sigma' \vdash \{\,p\,\}\; c\; \{\,q\,\} \end{array}}{\Lambda; \Phi; \Sigma \vdash \{\,p\,\}\; \mathbf{let}\; f_1(\vec{x}_1) = c_1, \ldots f_n(\vec{x}_n) = c_n\; \mathbf{in}\; c\; \{\,q\,\}}$$

cuando

- *p, q, Φ, Λ y Σ no contienen nombres de predicados en $dom.\Lambda'$,*
- *para cada $\{\,p_i\,\}\; c_i\; \{\,q_i\,\}$, con $1 \leqslant i \leqslant n$, se satisface que $FV.c_i \subseteq \vec{x}_i$ y $Mod.c_i = \varnothing$,*

y donde $p, p_1, \ldots, p_n \in Form_T{}'$, $c, c_1, \ldots, c_n \in Comm_T$, $\Lambda, \Lambda' \in Env_a$, $\Phi \in Env_f$ y $\Sigma, \Sigma' \in Env_s$.

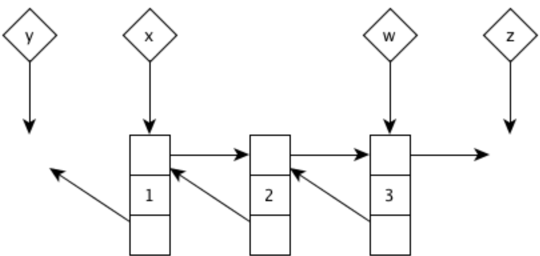


Figura 4.10: Modelo de **dlseg**.$x.y.z.w.[1, 2, 3]$.

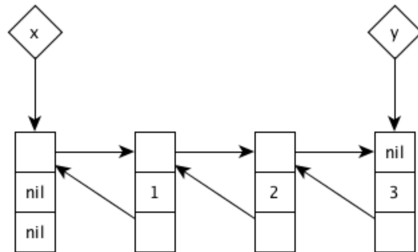


Figura 4.11: Modelo de **buff**.$x.y.[1, 2, 3]$.

La primer premisa garantiza que las especificaciones estáticas que conforman Σ', que posiblemente involucran predicados abstractos de Λ', son consistentes. Estas especificaciones pueden utilizarse en la demostración de las funciones, pero principalmente sirven para demostrar el código cliente c, aún cuando Λ' no se encuentra en su *scope*, y sus definiciones ya no están disponibles.

Las restantes reglas se adaptan al nuevo tipo de especificación de forma trivial.

Teorema 4 (Consistencia del sistema deductivo de ShL_T^H). *Sean $p, q \in Form_T'$, $c \in Comm_T$, $\Lambda \in Env_a$, $\Phi \in Env_f$ y $\Sigma \in Env_s$. Si $\Lambda; \Phi; \Sigma \vdash \{\, p \,\}\ c\ \{\, q \,\}$ se deduce utilizando las reglas del sistema deductivo,[6] entonces $\Lambda; \Phi; \Sigma \models \{\, p \,\}\ c\ \{\, q \,\}$.*

Ejemplo 49. *El siguiente caso está inspirado en un ejemplo de [7]. Supongamos que disponemos de dos tipos abstractos de datos que se comunican de manera asíncrona a través de un* buffer *de datos compartido con una política FIFO. Dicho escenario es muy común, por ejemplo, en la implementación de la pila de un protocolo de comunicación. Supongamo que el predicado* **prod**.$p.l.es$ *caracteriza el tipo que produce la información, manipulable a través del puntero p, y* **cons**.$c.l.es$ *el tipo que consume la información, manipulable a través del puntero c. En ambos casos, es representa la lista de valores compartidos, que especificamos con el predicado* **data**.es. *La situación en que una instancia*

[6] Definiciones 22, 24, 25, 26, 27, 28, 57, 68, 69, 70 adaptadas apropiadamente, definiciones 74 y 75.

de cada tipo comparten un buffer *puede especificarse como*

$$\exists l, l' \cdot \mathbf{prod}.p.l.es \ \langle *\colon \mathbf{data}.es \rangle \ \mathbf{cons}.c.l'.es$$

A continuación se presentan las definiciones de estos predicados y otros auxiliares, que podemos suponer forman parte de un entorno sintáctico de predicados Λ*:*

$$\begin{aligned}
\mathbf{dlseg}.x.y.z.w.es &\doteq (x = z \wedge y = w \wedge \mathbf{emp}) \vee \\
&\qquad (\exists y', e', es' \cdot (x \mapsto y', e', y * \mathbf{dlseg}.y'.y.z.w.es') \wedge es = e' \triangleright es') \\
\mathbf{buff}.x.y.es &\doteq (x \mapsto \mathbf{null}, \mathbf{null}, \mathbf{null} \wedge x = y) \vee \\
&\qquad (\exists x', e', es' \cdot (x \mapsto x', \mathbf{null}, \mathbf{null} * \mathbf{dlseg}.x'.x.\mathbf{null}.y.es) \wedge es = e' \triangleright es') \\
\mathbf{data}.es &\doteq \exists x, y \cdot \mathbf{buff}.x.y.es \\
\mathbf{prod}.p.l.es &\doteq \exists x \cdot p \mapsto x, \ldots * \mathbf{buff}.x.l.es \\
\mathbf{cons}.c.l.es &\doteq \exists y \cdot c \mapsto y, \ldots * \mathbf{buff}.l.y.es
\end{aligned}$$

El predicado $\mathbf{dlseg}.x.y.z.w.es$ *caracteriza una lista doblemente ligada que almacena los valores de la secuencia abstracta* es*, cuyo primer elemento está referenciado por* x *y el último por* w*. En la figura 4.10 se puede apreciar cómo son los modelos de una fórmula como* $\mathbf{dlseg}.x.y.z.w.[1,2,3]$.

El predicado $\mathbf{buff}.x.y.es$ *caracteriza escencialmente una lista doblemente ligada, que cuenta con un registro* centinela*. En la figura 4.11 se muestra un modelo de la fórmula* $\mathbf{buff}.x.y.[1,2,3]$.

El predicado **data** *no involucra variables (de programa), por lo que sólo sirve para caracterizar el estado compartido. Los predicados* $\mathbf{prod}.p.l.es$ *y* $\mathbf{cons}.c.l.es$ *incluyen en su implementación un* buffer *que representa la secuencia* es *y posiblemente otros registros. En ambos casos, la variable* l *existe para poder asegurar la invarianza de la dirección de cierto registro: en el caso de* **prod***, la dirección del último registro; en el caso de* **cons***, la dirección del primer registro. De esta manera es posible especificar, por ejemplo, que una función asociada a* **prod** *garantiza la consistencia de la implementación de* **cons***, y viceversa.*

Veamos, por caso, la verificación del código de la función $produce(e)$ *que "produce" un dato con valor* e*, lo almacena en el* buffer *y garantiza que la dirección del último registro de la lista permanece inmutable. El esquema de prueba de la terna*

$$\Lambda; \epsilon; \epsilon \vdash \{\, \mathbf{prod}.p.l.es \,\}\ produce(p, e)\ \{\, \mathbf{prod}.p.l.(e \triangleright es) \,\}$$

se muestra a continuación, donde utilizamos FR para denotar la regla de frame*:*

$\{\, \mathbf{prod}.p.l.es \,\}$

$\Rightarrow \langle$ *definición de* **prod** $\rangle$

$\{\, \exists x \cdot p \mapsto x, \ldots \ast \mathbf{buff}.x.l.es \,\}$

$\quad \mathbf{newvar}(x);;$

$\{\, (\exists x \cdot p \mapsto x, \ldots \ast \mathbf{buff}.x.l.es) \wedge x = \mathbf{null} \,\}$

$\quad x := p.0$

$\{\, p \mapsto x, \ldots \ast \mathbf{buff}.x.l.es \,\}$

$\Rightarrow \langle$ *introducción de* $\exists$ *sobre* x $\rangle$

$\{\, \exists x' \cdot p \mapsto x', \ldots \ast \mathbf{buff}.x.l.es \,\}$

FR:

> $\{\, \mathbf{buff}.x.l.es \,\}$
>
> $\Rightarrow \langle$ *definición de* **buff** $\rangle$
>
> $\{\, (x \mapsto \mathbf{null}, \mathbf{null}, \mathbf{null} \wedge x = l) \vee$
> $(\exists x', e', es' \cdot (x \mapsto x', \mathbf{null}, \mathbf{null} \ast \mathbf{dlseg}.x'.x.\mathbf{null}.l.es) \wedge es = e' \triangleright es') \,\}$
>
> $\quad \mathbf{newvar}(y);;$
>
> $\{\, (x \mapsto \mathbf{null}, \mathbf{null}, \mathbf{null} \wedge x = l \wedge y = \mathbf{null}) \vee$
> $(\exists x', e', es' \cdot (x \mapsto x', \mathbf{null}, \mathbf{null} \ast \mathbf{dlseg}.x'.x.\mathbf{null}.l.es) \wedge es = e' \triangleright es'$
> $\wedge\, y = \mathbf{null}) \,\}$
>
> $\quad y := \mathbf{cons}(x, \mathbf{null}, \mathbf{null});$
>
> $\{\, (y \mapsto x, \mathbf{null}, \mathbf{null} \ast x \mapsto \mathbf{null}, \mathbf{null}, \mathbf{null} \wedge x = l) \vee$
> $(\exists x', e', es' \cdot (y \mapsto x, \mathbf{null}, \mathbf{null} \ast x \mapsto x', \mathbf{null}, \mathbf{null} \ast \mathbf{dlseg}.x'.x.\mathbf{null}.l.es)$
> $\wedge\, es = e' \triangleright es') \,\}$
>
> $\quad x.1 := e;$
>
> $\quad x.2 := y;$
>
> $\{\, (y \mapsto x, \mathbf{null}, \mathbf{null} \ast x \mapsto \mathbf{null}, e, y \wedge x = l) \vee$
> $(\exists x', e', es' \cdot (y \mapsto x, \mathbf{null}, \mathbf{null} \ast x \mapsto x', e, y \ast \mathbf{dlseg}.x'.x.\mathbf{null}.l.es)$
> $\wedge\, es = e' \triangleright es') \,\}$
>
> $\Rightarrow \langle$ *definición de* **dlseg** $\rangle$
>
> $\{\, (y \mapsto x, \mathbf{null}, \mathbf{null} \ast \mathbf{dlseg}.x.y.\mathbf{null}.l.[e]) \vee$
> $(\exists e', es' \cdot (y \mapsto x, \mathbf{null}, \mathbf{null} \ast \mathbf{dlseg}.x.y.\mathbf{null}.l.(e \triangleright es)) \wedge es = e' \triangleright es') \,\}$
>
> $\quad x := y;$
>
> $\{\, (\exists x' \cdot x \mapsto x', \mathbf{null}, \mathbf{null} \ast \mathbf{dlseg}.x'.x.\mathbf{null}.l.[e]) \vee$
> $(\exists x', e', es' \cdot (x \mapsto x', \mathbf{null}, \mathbf{null} \ast \mathbf{dlseg}.x'.x.\mathbf{null}.l.(e \triangleright es)) \wedge es = e' \triangleright es') \,\}$
>
> $\Rightarrow \langle$ *introducción de* $\exists$*, debilitamiento, idempotencia de* $\vee$ $\rangle$
>
> $\{\, \exists x' \cdot x \mapsto x', \mathbf{null}, \mathbf{null} \ast \mathbf{dlseg}.x'.x.\mathbf{null}.l.(e \triangleright es) \,\}$
>
> $\Rightarrow \langle$ *definición de* **buff** $\rangle$
>
> $\{\, \mathbf{buff}.x.l.(e \triangleright es) \,\}$

$\{\, \exists x' \cdot p \mapsto x', \ldots \ast \mathbf{buff}.x.l.(e \triangleright es) \,\}$

$\quad p.0 := x;$

$\{\, p \mapsto x, \ldots \ast \mathbf{buff}.x.l.(e \triangleright es) \,\}$

$\Rightarrow \langle$ *introducción de* $\exists$ *sobre* x $\rangle$

$\{\, \exists x \cdot p \mapsto x, \ldots \ast \mathbf{buff}.x.l.(e \triangleright es) \,\}$

$\quad \mathbf{return}(\mathbf{null});$

$\{\, \exists x \cdot p \mapsto x, \ldots \ast \mathbf{buff}.x.l.(e \triangleright es) \,\}$

$\Rightarrow \langle$ *definición de* **prod** $\rangle$

$\{\, \mathbf{prod}.p.l.(e \triangleright es) \,\}$

Supongamos que un entorno sintáctico de funciones Φ incluye las siguientes ternas:

$$\{\,\mathbf{prod}.p.l.es\,\}\ produce(p,e)\ \{\,\mathbf{prod}.p.l.(e \triangleright es)\,\}$$
$$\{\,\mathbf{cons}.c.l.(es \triangleleft e)\,\}\ \ consume(c)\ \ \{\,\mathbf{cons}.c.l.es\,\}$$

y un entorno de especificaciones estáticas Σ, las siguientes fórmulas cuya demostración por vía semántica es sencilla:

$$\begin{aligned}(\mathbf{prod}.p.l.es \ -\!\circledast\ (\mathbf{prod}.p.l.es\ \langle *\colon \mathbf{data}.es\rangle\ \mathbf{cons}.c.l'.es))\ *\ \mathbf{prod}.p.l.(e \triangleright es)\\ \Rightarrow \mathbf{prod}.p.l.(e \triangleright es)\ \langle *\colon \mathbf{data}.(e \triangleright es)\rangle\ \mathbf{cons}.c.l'.(e \triangleright es)\end{aligned}$$

$$\begin{aligned}(\mathbf{cons}.c.l.(es \triangleleft e)\ -\!\circledast\ (\mathbf{prod}.p.l'.(es \triangleleft e)\ \langle *\colon \mathbf{data}.(es \triangleleft e)\rangle\ \mathbf{cons}.c.l.(es \triangleleft e)))\ *\ \mathbf{cons}.p.l.es\\ \Rightarrow \mathbf{prod}.p.l'.es\ \langle *\colon \mathbf{data}.es\rangle\ \mathbf{cons}.c.l.es\end{aligned}$$

Con Φ y Σ podemos demostrar un código cliente que alterna un número arbitrario de llamadas a produce y consume:

$$\{\,\exists l,l' \cdot \mathbf{prod}.p.l.es\ \langle *\colon \mathbf{data}.es\rangle\ \mathbf{cons}.c.l'.es\,\}$$

$$\text{SFR:}\ \left|\begin{array}{l}\{\,\mathbf{prod}.p.l.es\,\}\\ \quad produce(p,e);\\ \{\,\mathbf{prod}.p.l.(e \triangleright es)\,\}\end{array}\right.$$

$$\{\,\exists l,l' \cdot \mathbf{prod}.p.l.(e \triangleright es)\ \langle *\colon \mathbf{data}.(e \triangleright es)\rangle\ \mathbf{cons}.c.l'.(e \triangleright es)\,\}$$

$\Rightarrow$ ⟨ *cálculo sobre secuencias abstractas* ⟩

$$\{\,\exists l,l',e',es' \cdot (\mathbf{prod}.p.l.(es' \triangleleft e')\ \langle *\colon \mathbf{data}.(es' \triangleleft e')\rangle\ \mathbf{cons}.c.l'.(es' \triangleleft e'))$$
$$\wedge\ (e \triangleright es) = (es' \triangleright e')\,\}$$

$$\text{SFR:}\ \left|\begin{array}{l}\{\,\mathbf{cons}.c.l'.(es' \triangleleft e')\,\}\\ \quad consume(c)\\ \{\,\mathbf{cons}.c.l'.es'\,\}\end{array}\right.$$

$$\{\,\exists l,l',e',es' \cdot (\mathbf{prod}.p.l.es'\ \langle *\colon \mathbf{data}.es'\rangle\ \mathbf{cons}.c.l'.es') \wedge (e \triangleright es) = (es' \triangleright e')\,\}$$

Las condiciones de aplicación de la SFR se resuelven directamente por las especificaciones en Σ.

4.5. Discusión

Completitud de ShL y reductibilidad a SL

La *incompletitud* de un sistema de prueba lógico en relación a un modelo particular es inevitable en sistemas lógicos suficientemente expresivos. El caso paradigmatico, demostrado por Gödel, es la aritmética de Peano que contiene afirmaciones que

son válidas en los números naturales, pero no en cualquier otro modelo, y por tanto no son demostrables a partir de sus axiomas. Como mencionamos en la introducción, la *Separation Logic* respecto al modelo de memoria estándar (def. 7, pág. 42), restringiendo los valores a *Loc* y eliminando los conectivos espaciales $*$ y $-\!\!*$, es indecidible. Mas específicamente, decidir la validez de una fórmula de la lógica clásica de primer orden extendida con el predicado $\mapsto$ no es recursivamente enumerable. Por lo tanto, el predicado $\mapsto$ no puede ser axiomatizado de forma completa. Evidentemente esto tira por la borda también cualquier esperanza de axiomatizar la *Sharing Logic*.

Sin embargo, la parte proposicional de la *Separation Logic*, esto es, el fragmento que contiene combinaciones de proposiciones arbitrarias con los operadores espaciales y los operadores lógicos clásicos, se corresponde con la lógica *Boolean BI*, cuyos modelos son monoides conmutativos relacionales. Y su sistema deductivo, respecto al modelo estándar, es decir, cuando las proposiciones se interpretan como *heap* finitos, es completo [131]. Así, es posible lograr una caracterización completa de los operadores del fragmento proposicional, que se traduce en las propiedades "algebraicas" presentadas en la sección 3.2. Pero suele suceder que muchos sistemas lógicos completos no tienen la suficiente expresividad como para capturar propiedades interesantes de los modelos pretendidos. Los modelos de la *Separation Logic* son clases particulares de monoides conmutativos relacionales, basados en la composición de *heaps* (denominados *separation algebras*) [41]. Este tipo de modelo exhibe propiedades que no se validan por cualquier monoide conmutativo relacional, y por lo tanto no son capturadas por la demostrabilidad de la *Boolean BI*.

En [32] se demuestra que la *Boolean BI* no tiene el suficiente poder expresivo para axiomatizar los modelos usuales de las diferentes versiones de la *Separation Logic*. Se presenta una extensión *hibrida* de la *Boolean BI*, denominada HyBBI,[7] que incorpora la posibilidad de individualizar estados en un modelo de una fórmula (que en el modelo estandar se traduce en porciones de un *heap*), y vincularlos con proposiciones. El sistema deductivo de la HyBBI, que resulta ser una extensión conservativa de la *Boolean BI*, se demuestra completo.

Lo particular de esta lógica es que, con el poder expresivo de sus operadores, es capaz de caracterizar la semántica del operador $\uplus$ y de nuestro operador $\langle *\colon \rangle$. Es decir, los operadores de *sharing* pueden presentarse como simple notación derivada. Los autores sugieren que tal codificación no es posible en la *Boolean BI*. Esto apoya nuestra intuición que la *Sharing Logic* no es reducible en toda su generalidad a la *Separation Logic*. Además nos brinda un marco teórico donde evaluar nuestra teoría, y un sistema deductivo completo del cual derivar esquemas de axiomas para fortalecer nuestra presentación, que dista de ser completa.

[7] La HyBBI se relaciona con la *Boolean BI* de la misma manera que la lógica hibrida normal se relaciona con la lógica modal normal [16].

El sentido preciso de lo que significa la imposibilidad de reducir la *Sharing Logic* no es tan sencillo de capturar, menos aún de demostrar. La *Separation Logic* tiene el suficiente poder expresivo para caracterizar de forma completa un *heap*, a través del predicado $\mapsto$ y expresiones que denotan los valores atómicos y direcciones de memoria. La *Sharing Logic* viene a introducir una forma concisa de caracterizar un conjunto amplio de posibles formas de *sharing* entre los *heaps* que satisfacen dos fórmulas arbitrarias. La cantidad de tales formas *diferentes* de *sharing* pueden ser infinitas, si las fórmulas admiten infinitos *heaps* de diferente tamaño. Podemos conjeturar entonces que los potenciales infinitos escenarios que caracteriza $\langle *: \rangle$ no son definibles con un conjunto finito de fórmulas de *Separation Logic*.

Conjetura 1 (Irreductibilidad de la ShL a la SL)**.** *Existe $p \in Form'$ para la cual no existe un conjunto finito de fórmulas $p_1, \dots, p_n \in Form$ (para cualquier $n \in \mathbb{N}_0$) tales que*

$$s, h \vDash p \textit{ si y sólo si } s, h \vDash p_1 \vee \dots \vee p_n$$

para todo $s, h \in States$.

Trabajos relacionados

Existen diversas extensiones de la *Separation Logic* que intentan modelar diferentes formas abstracción, desde el soporte a simples o múltiples instancias de módulos imperativos [117, 120, 13] a complejos modelos de lenguajes orientados a objetos con soporte para herencia, *dispatch* dinámico, etc. [122, 51, 103]. Estas extensiones permiten en mayor o menor medida la verificación composicional e independiente de los módulos y el código cliente que los utiliza. Sin embargo estas propuestas se basan en el lenguaje de la *Separation Logic*, y por lo tanto no logran dar cuenta de la interacción entre múltiples instancias de diferentes tipos abstractos de datos cuando comparten la memoria para su representación concreta.

En [119] se combinan la noción de *abstract predicate* y la extensión de la *Separation Logic* con *permisos* [29, 26] dentro de un marco de trabajo que da cuenta de un subconjunto del núcleo de Java. Esto da lugar a la posibilidad de especificar instancias de clases que comparten memoria para su representación. Sin embargo esta extensión requiere introducir un modelo de estados relativamente complejo para lidiar con el problema de que la conjunción espacial $*$ ya no expresa disjuntividad, y por otro lado la expresividad lograda es de *read sharing*, restringiendo cualquier posible modificación del estado compartido.

En [140] siguen un camino similar, decorando cada campo de los registros con modificadores que estipulan su *status* (*ausente*, modificable, o invariable) logrando una forma de caracterización parcial del *heap*. Con la introducción de un nuevo operador de conjunción *superpuesta* $\mathbin{\text{⩓}}$, son capaces de describir estructuras de datos superpuestas (como el ejemplo del *fringe*) combinando con $\mathbin{\text{⩓}}$ diferentes predicados

recursivos que caracterizan parcialmente la memoria. La capacidad expresiva de esta propuesta es una especie limitada de noción de *permisos*. Sin embargo su simplicidad permite definir un *shape analysis* (ver capítulo 6) que soporta la verificación de programas que manipulan este tipo de estructuras de datos.

En el trabajo no publicado [91] tratan el problema de especificar abstracta y modularmente el comportamiento de *sharing* de módulos imperativos. Utilizan una versión de *Separation Logic* de alto orden sobre un lenguaje funcional tipado de alto orden con referencias mutables. El soporte de la abstracción de datos está dado por predicados cuantificados existencialmente, y el comportamiento de *sharing* de los módulos por su propio concepto de "especificaciones estáticas". A pesar de la distancia en el aspecto técnico, se presentan grandes similitudes en el tratamiento del problema al que nos abocamos, aunque con algunas diferencias remarcables. Mientras que nuestro lenguaje permite especificar simultáneamente múltiples vistas sobre el *heap*, la técnica presentada en el citado artículo obliga a intercambiar permanentemente entre las distintas abstracciones, restringiendo el razonamiento sobre una abstracción por vez en cada punto del programa. El intercambio esta justificado por las "especificaciones estáticas" del modulo, que son tautologías que vinculan las diferentes abstracciones que comparten parte de su representación concreta. Las "especificaciones estáticas" suelen utilizar un patrón del tipo *deletion followed by extension* combinado con fórmulas arbitrarias cuantificadas, lo que permite caracterizar de forma precisa las formas de *sharing*, a costa de utilizar una lógica de alto orden y la complejidad que esto acarrea.

En [146] se extiende la *Separation Logic* a un entorno concurrente combinándola con la metodología Rely/Guarantee [150]. Aunque los objetivos son completamente diferentes, el lenguaje de fórmulas propuesto presenta grandes similitudes con el nuestro. Allí se utiliza la notación $\boxed{p}$ para distinguir una fórmula p sobre el *heap* compartido, en ese caso entre los diferentes *threads*. Para dar cuenta de ello, es necesario modificar el estado distinguiendo entre el *heap* compartido y el local. Nuestro lenguaje de aserciones por un lado ofrece una mayor expresividad ya que el heap compartido puede ser especificado por diferentes formulas en cada una de las múltiples vistas, y por otro lado descansa sobre un modelo de estados simple. Esta coincidencia junto a la similitud entre su noción de *estabilidad* sobre el heap compartido y las condiciones en las reglas de *frame*, nos hacen pensar en la posibilidad de definir un marco unificado para el razonamiento modular y abstracto sobre estructuras tanto en un entorno secuencial como concurrente.

En [81] se presentan dos reglas para razonar *globalmente* en el contexto de cualquier *Separation Logic*:

Regla de *backward ramify*:

$$\frac{\vdash \{\, p \,\}\; c\; \{\, q \,\} \qquad \vDash r \Rightarrow p * (q \mathrel{-\!\!*} r') \qquad FV.(q \mathrel{-\!\!*} r') \cap Mod.c = \varnothing}{\vdash \{\, r \,\}\; c\; \{\, r' \,\}}$$

Regla de *forward ramify*:

$$\frac{\vdash \{\, p \,\}\; c\; \{\, q \,\} \qquad \vDash (p \mathrel{-\!\!\circledast} r) * q \Rightarrow r' \qquad \vDash r \Rightarrow p * \mathbf{true} \qquad FV.(p \mathrel{-\!\!\circledast} r) \cap Mod.c = \varnothing}{\vdash \{\, r \,\}\; c\; \{\, r' \,\}}$$

La regla de *forward ramify* sigue la misma forma de razonamiento que nuestras GFR y SFR. De hecho la SFR es un caso particular de esta regla, donde introducimos una cuantificación existencial para lidiar con la molesta, en este caso prácticamente prohibitiva, condición sobre las variables modificadas. Con el uso de estas reglas, demuestran una variedad de programas recursivos sobre estructuras de datos con *sharing* inespecífico, como grafos y *dags*, que caracterizan utilizando los operadores $\uplus^{*}$ y $\wedge$. Proponen el desarrollo de una biblioteca de resultados que permitan resolver las condiciones de aplicación de las reglas de *ramify* y razonar en presencia del tipo de *sharing* permitido, similar en su espíritu a nuestra idea de especificaciones estáticas (aunque no proveen soporte para tipos abstractos de datos). A pesar de las similitudes técnicas y en el enfoque, no introducen ninguna facilidad para caracterizar formas de *sharing* específicas y complejas.

Contribuciones

En este capítulo presentamos una generalización de la *Separation Logic* para razonar de forma relativamente local sobre diversas estructuras complejas, incluyendo relaciones de *sharing* entre ellas, de manera compatible con los principios de abstracción e *information hiding*. Con el uso de los nuevos operadores resulta particularmente sencillo especificar simultáneamente relaciones de separación espacial y *sharing*, brindando soporte a múltiples vistas sobre el estado de la memoria. Este poder expresivo también puede utilizarse para distinguir subestructuras y así evitar la necesidad de definir complejos predicados para caraterizar estructuras de datos *parciales*

Las reglas de *frame* permiten razonar independientemente con estas vistas de manera elegante. Las hipótesis necesarias para su aplicación, aunque de cierta complejidad, caracterizan con fórmulas dentro de la misma lógica las condiciones mínimas para garantizar la consistencia de las estructuras que comparten el *heap* frente a la modificación de la memoria. Resulta una grata sorpresa haber encontrado los errores en la especificación del problema del *fringe* a partir del intento de demostrar las

condiciones de la GFR. Creemos que esto sugiere la necesidad de prestar una mayor atención a los problemas que surgen en presencia de *sharing* entre distintas estructuras, que muchas veces son extremadamente sutiles.

Para soportar tipos abstractos de datos, optamos por el concepto de predicado abstracto ya que modela esta noción de manera sencilla y cercana a la intuición del programador. Ademas su aplicación se extiende a un lenguaje orientado a objetos con características avanzadas, lo que permitiría desarrollar nuestras ideas en ese contexto. La introducción del concepto de especificación estática permite caracterizar de manera igualmente sencilla e intuitiva las posibilidades y restricciones en la manipulación de las estructuras abstractas más allá de la caracterización *dinámica* que proveen las especificaciones de las funciones. La caracterización provista por las especificaciones estáticas es de algún modo *axiomática* y parece necesaria para poder razonar en general en presencia de formas de abstracción. Esto se evidencia, por ejemplo, en [147] donde se utiliza una forma de caracterización similar para lidiar con las relaciones de herencia entre estructuras abstractas. La combinación de las reglas de *frame* con las especificaciones de las funciones y las especificaciones estáticas permiten extender las posibilidades del razonamiento local, modular y abstracto más allá de todas las extensiones que conocemos de la *Separation Logic*.

Sin embargo no desconocemos los límites de nuestra propuesta. Por un lado, razonar en presencia del operador $\langle *: \rangle$ puede resultar díficil, y muchas veces requiere argumentos semánticos. La complejidad para manipularlo es equiparable a la que surge del razonamiento sobre el operador $-\!\!*$, manifiesta en la mayor parte de los trabajos de la *Separation Logic*. Esto no es sorpresivo, ya que ambos operadores tienen una expresividad comparable, que hemos caracterizado precisamente a lo largo del capítulo.

Por otro lado, las condiciones para poder extender el razonamiento local a un contexto más general, cristalizadas como las hipótesis de las reglas de *frame*, vuelven necesaria en algunas ocasiones una caracterización del estado compartido tan *suficientemente específica* que el escenario global puede especificarse con alguno de los patrones *deletion followed by extension* en *Separation Logic*. Tal es el caso del ejemplo 48 (pág. 108) que puede llevarse adelante en esos términos, aunque tal caracterización no provee ningún beneficio en el desarrollo de la demostración.

La necesidad de poder determinar de forma precisa ciertas porciones del *heap* para razonar en presencia de *sharing* es una conjetura manifiesta de Reynolds [134]. En [66, 54] responden a esta necesidad introduciendo explícitamente formas de nombrar *heaps* en las fórmulas, cuantificar sobre ellos, etc. Esto permite definir estructuras de datos recursivamente, identificando de forma precisa subestructuras relevantes (que pueden estar compartidas). Como ya lo mencionamos, la misma idea está presente en la HyBBI. El marco de trabajo que hemos desarrollado soporta la conjetura de Reynolds, ya que cuanto más especificado está el estado compartido, más sencillo es

razonar sobre él, y más posibilidades existen de extender el razonamiento a contextos de memoria más amplios. En el ejemplo 49 (pág. 113), la variable de especificación l en los predicados **prod**.$p.l.es$ y **cons**.$c.l.es$ sirve justamente para identificar los registros cuya invarianza frente a la modificación de la memoria, es necesaria para garantizar el estado global. Sin un recurso de este tipo no sería posible demostrar la validez de las condiciones de aplicación de las reglas de *frame*. Sin embargo, este ejemplo muestra al mismo tiempo que no es necesario que *toda* la memoria compartida esté completamente especificada. En ese sentido, este ejemplo no es expresable, tal como está presentado, utilizando la *Separation Logic*. Nuestro lenguaje de formulas permite variar la especificidad de la porción del *heap* compartido a través de la fórmula elegida para su caracterización.

La propuesta que desarrollamos ofrece un marco *uniforme y general* para analizar y caracterizar las condiciones necesarias para poder extender un razonamiento local a un contexto más general, cuando pueden existir múltiples predicados sobre la misma porción de memoria. Esto nos permite, incluso, reconocer diferentes patrones que son útiles para caracterizar, bajo ciertas circunstancias particulares precisadas por la clase *Sufficient*, el *sharing* de memoria con fórmulas de *Separation Logic*. En tales casos, las condiciones para aplicar las reglas de *frame* pueden verse simplificadas. Hasta donde tenemos conocimiento, la utilidad general de tales patrones para este fin no ha sido reconocida en la literatura.

Las posibilidades de trabajo futuro se presentan en variadas direcciones. Como mencionamos anteriormente, una extensión inmediata es la adaptación de nuestra teoría a un lenguaje orientado a objetos más cercano a los utilizados en la practica cotidiana. Igualmente interesante resulta explorar las ventajas que nuestro lenguaje de aserciones podría brindar en un marco concurrente en combinación con la metodología Rely/Guarantee. Finalmente consideramos importante estudiar la teoría HyBBI de [32] para analizar la posibilidad de dar un caracterización algebráica de los operadores $\langle *: \rangle$ y $\langle -\!\!*: \rangle$ más completa.

Decidibilidad de la *Sharing Logic*

5

La *decidibilidad* de una lógica se refiere a la existencia de un *procedimiento efectivo* para determinar cuándo una fórmula arbitraria es *válida*, es decir, cuando se satisface en cualquier modelo posible. Un problema relacionado es la decisión sobre la *satisfactibilidad* (finita) de una fórmula, esto es, determinar si existe algún modelo (finito) para la misma. Para las lógicas cerradas por negación los problemas de validez y satisfactibilidad son equivalentes. Determinar la decidibilidad de la *Sharing Logic* se reduce entonces a demostrar si es posible contestar la pregunta sobre la validez de una fórmula de nuestra lógica de manera algorítmica. En este capítulo, nos proponemos analizar la decidibilidad de un fragmento que incluye un predicado para caracterizar segmentos de listas enlazadas. Como punto de partida, repasamos la (ind)decidibilidad de la *Separation Logic*, que ha sido estudiada profusamente. Cabe remarcar que, en tanto el poder expresivo de la *Separation Logic* está incluido en la *Sharing Logic*, la indecidibilidad de la primera implica la indecidibilidad de la segunda. Recordemos la sintaxis de una de sus formulaciones, que reduce las expresiones y relaciones al mínimo.

Definición 76 (Sintaxis alternativa de SL)**.** *El conjunto de expresiones Expr y fórmulas Form sobre Var está dado por la siguiente gramática:*

$$Expr \ni e ::= x \mid \mathbf{null}$$
$$Form \ni p ::= e = e \mid \mathbf{emp} \mid x \mapsto \vec{e} \mid p \vee p \mid \neg p \mid p * p \mid p \mathrel{-\!\!*} p \mid \forall x \,.\, p$$

donde $x \in Var$ y $\vec{e} \in Expr^+$.

Para poder establecer de forma precisa los resultados que se encuentran en la literatura, introducimos la siguiente notación. Sea $\star$ una metavariable sobre el conjunto de operadores y cuantificadores. Con k-SL* denotamos el conjunto de fórmulas restringidas, que excluyen cualquier ocurrencia del operador $\star$ y refieren únicamente, a través del predicado $\mapsto$, a registros de tamaño k. Utilizamos libremente esta notación con más de un operador.

En [43] se demuestra por primera vez la indecidibilidad de 2-SL$^{*,-\!*}$, reduciendo el problema a la satisfactibilidad finita de la lógica de predicados clásica con una relación binaria (para lo cual se utiliza fuertemente que el modelo de memoria consta de registros de tamaño dos). Al mismo tiempo se prueba la decidibilidad del fragmento 2-SL$^\forall$ estableciendo una propiedad de modelos acotados. En [39] se presenta

otra prueba del mismo hecho, traduciendo 2-SL$^{\forall}$ a un fragmento de lógica de primer orden con operadores proposicionales, cuantificación sobre números naturales e igualdad.

Mas tarde, en [94] se establece la indecidibilidad de k-SL^{-*} (para k arbitrario) demostrando su equivalencia expresiva[1] con la lógica de segundo orden. Finalmente este resultado se generaliza en [30] estableciendo la igualdad expresiva entre 1-SL, 1-SL* y la lógica de segundo orden. En el mismo trabajo se muestra que 1-SL^{-*} es decidible reduciendo este fragmento, via una traducción con complejidad LOGSPACE, a la lógica de segundo orden *monádica*. Sin bien la lógica analizada no incluye predicados particulares para especificar estructuras de datos, se muestra cómo codificar un predicado sobre segmentos de listas enlazadas a través de cuantificadores y $\mapsto$. En [6] los mismos autores demuestran la decidibilidad para 1-SL^{-*} extendida con un predicado que permite comparar los valores de dos datos sucesivos en una lista, incluyendo así la noción de ordenación.

En [60] se demuestra la decidibilidad del problema de satisfactibilidad finita de un fragmento de 1-SL con una única variable cuantificada, que incluye la posibilidad de predicar sobre listas enlazadas, reduciendo una fórmula arbitraria a una combinación proposicional de predicados atómicos elementales. Este es el mayor fragmento que incluye $-*$ y cuantificación, ya que en [59] se muestra que 1-SL sin variables libres y sólo dos variables cuantificadas se vuelve indecidible.

En [10] se sigue un enfoque diferente, mostrando cómo decidir la validez de *entailments* entre fórmulas, llamadas *symbolic heaps*, de un sub-fragmento restringido de 2-SL$^{\forall,-*}$, que incluye un predicado para especificar segmentos de listas enlazadas.

Definición 77 (Syntaxis de *symbolic heaps* de [10]). *El conjunto de* symbolic heaps, *denotado por SH, se define por la siguiente gramática:*

$$\begin{aligned}\Pi \ni \pi &::= e = e \mid e \neq e \mid \mathbf{true} \mid \pi \wedge \pi \\ \Sigma \ni \sigma &::= \mathbf{emp} \mid \mathbf{true} \mid x \mapsto e \mid \mathbf{lseg}.e.e \mid \sigma * \sigma \\ SH \ni sh &::= \pi \wedge \sigma\end{aligned}$$

donde $x \in Var$ y $e \in Expr$.

Este fragmento elimina la mayor parte de los conectivos lógicos, separando las fórmulas espaciales de las proposicionales, deshabilitando así cualquier tipo de *sharing* y especificaciones complejas. Las razones para restringir tan fuertemente las fórmulas, no son tanto teóricas sino prácticas: por un lado, resulta sencillo definir análisis estáticos basados en la ejecución simbólica de programas utilizando *symbolic*

[1]Dos lógicas A y B son equivalentemente expresivas si para toda fórmula p de A, existe una fórmula p' de B, y viceversa, tal que para cualquier modelo μ, μ satisface p si y sólo si μ satisface p'.

heaps como abstracciones de los estados (asunto en el que profundizamos en el capítulo siguiente); por otro, la complejidad del mecanismo de decisión puede reducirse hasta tiempo polinomial [55, 78, 132].

En esta misma dirección, en [128, 124] se muestra cómo reducir los *symbolic heaps* a lógicas que caen dentro del marco de satisfactibilidad módulo teorías (SMT). Este enfoque permite integrar este fragmento a herramientas existentes (SMT *solvers*), resolviendo los problemas de satisfactibilidad y decisión de *entailments*, entre otros, y combinarlo de manera sencilla con otras teorías de primer orden decidibles.

Otros trabajos posteriores extienden el fragmento de *symbolic heaps*, manteniendo la decidibilidad de *entailments*, con definiciones recursivas parametrizadas, que permiten especificar diferentes estructuras de datos como listas, listas doblemente ligadas, árboles, etc, [31, 84] (aunque la incorporación de predicados recursivos generales introduce indecidibilidad [3]); e incluso predicados para caracterizar los datos representados en su tamaño, ordenación, etc. [28, 52, 129].

5.1. Modelos indiscernibles

Consideremos el siguiente fragmento decidible de la *Sharing Logic*, que denominamos ShL_D.

Definición 78 (Sintaxis de ShL_D)**.** *El conjunto de expresiones $Expr_D$ y fórmulas $Form_D$ sobre un conjunto enumerable de variables Var, está dado por las siguientes gramáticas:*

$$Expr_D \ni e ::= x \mid \mathbf{null}$$
$$Form_D \ni p ::= e = e \mid \mathbf{emp} \mid x \mapsto e \mid \mathbf{lseg}.x.x \mid p \vee p \mid \neg p \mid p \ \langle *\colon p\rangle \ p$$

donde $x \in Var$ y $e \in Expr_D$.

Los restantes operadores proposicionales y constantes ($\wedge$, $\Rightarrow$, **true**, **false**) se introducen como notación derivada. El conjunto de modelos es similar al presentado en la definición 7 (pág. 42), pero restringiendo el tamaño de los registro a uno y el conjunto de valores a las direcciones de memoria.

Definición 79 (Modelo de ShL_D)**.** *Dado un conjunto infinito de valores Loc (interpretados como direcciones de memoria), con un elemento distinguido $nil \in Loc$, y un conjunto enumerable de variables Var, el conjunto de modelos $States_D$ está dado por las siguientes ecuaciones:*

$$Stacks_D \doteq Var \to Loc$$
$$Heaps_D \doteq (Loc - \{nil\}) \rightharpoonup_{fin} Loc$$
$$States_D \doteq Stacks_D \times Heaps_D$$

La semántica de este fragmento, exceptuando el predicado **lseg**, es completamente análoga a la introducida en la definición 64 (pág. 94). Para el predicado **lseg** damos una definición semántica equivalente a la definición recursiva introducida anteriormente (def. 31, pág. 65), que encadena registros que sólo contienen un puntero al siguiente registro.

Definición 80 (Semántica de **lseg**). *Dado un modelo* $s, h \in States_D$*, la semántica de* **lseg** *está dada por las siguientes ecuaciones:*

$$
\begin{array}{ll}
s,h \models \mathbf{lseg}^0.l_1.l_2 & \textit{si y sólo si } l_1 = l_2 \textit{ y } h = \emptyset \\
s,h \models \mathbf{lseg}^{n+1}.l_1.l_2 & \textit{si y sólo si } l_1 \neq l_2 \textit{ y existen } l \in Loc, h' \in Heaps_D \textit{ tales que} \\
& l_1 \notin dom.h', h = \{(l_1,l)\} \cup h' \textit{ y } s,h' \models \mathbf{lseg}^n.l.l_2 \\
s,h \models \mathbf{lseg}.x.y & \textit{si y sólo si existe } n \textit{ tal que } s,h \models \mathbf{lseg}^n.[\![x]\!]_s.[\![y]\!]_s
\end{array}
$$

donde $l_1, l_2, l \in Loc$*,* $n \in \mathbb{N}_0$*, y* $x, y \in Var$*.*

De acuerdo a la semántica de este fragmento, podemos observar intuitivamente que la satisfacción de las fórmulas no puede distinguir entre:

- las direcciones específicas de los registros definidos en la memoria y los punteros *dangling*, y por lo tanto, los valores de las variables. Cualquier acceso a la memoria es indirecto, a través de una variable, y así la dirección particular de los registros de memoria es irrelevante. Siempre que se preserve la relación entre las variables del *stack* y el *heap*, y entre los mismos registros del *heap*, las direcciones particulares pueden tomarse arbitrariamente;
- la cantidad de *basura* (por sobre cierto límite), *i.e.* los registros que no son referenciados directa o indirectamente por ninguna variable. Cuando es necesario que exista *basura* para la satisfacción de una fórmula, sólo importa su existencia, pero no es relevante ninguna otra propiedad específica, como los valores que almacenan o su interrelación. La basura sólo puede estar presente en la satisfacción de fórmulas puras o fórmulas negadas, ninguna de las cuales distingue de forma precisa la cantidad de memoria asignada, aunque la interacción con $\langle * : \rangle$ puede imponer la necesidad de una cierta cantidad mínima;
- la longitud de registros encadenados que conforman segmentos de listas (nuevamente por sobre cierto límite). El *tamaño* de un modelo está básicamente determinado por los operadores $\mapsto$ y $\langle * : \rangle$. Un término **lseg** no puede diferenciar entre cadenas de registros de diferente longitud (excepto para la fórmula trivial **lseg**.$x.x$). Sin embargo el *sharing* introducido por $\wedge$ impone un limite minimal al tamaño de los modelos.

A continuación, en línea con las ideas presentadas en [43], explotamos estas propiedades para demostrar la decidibilidad del problema de validez para la ShL_D. Defi-

nimos una relación de equivalencia para cada uno de los puntos anteriores y demostramos que la satisfacción de fórmulas no puede discernir entre los modelos relacionados, y por lo tanto, se preserva para las clases de equivalencia de modelos inducidas. Combinando tales relaciones podemos establecer que la validez de las fórmulas puede decidirse a partir de un conjunto finito de clases de modelos indiscernibles.

Indiscernibilidad de direcciones

Comencemos definiendo una relación que captura la indiscernibilidad de las direcciones de los registros a través de un isomorfismo entre modelos.

Definición 81 (Relación $\sim$)**.** *Dado un conjunto de variables* $X \subseteq Var$*, y dos modelos* s, h *y* $s', h' \in States_D$*, definimos la relación* $\sim^X \in States_D \times States_D$*, y decimos que* $s, h \sim^X s', h'$ *si y sólo si existen funciones* $f \in Loc \to Loc$ *y* $g \in Loc \to Loc$ *tales que:*

1. $f.nil = g.nil = nil$
2. $f \bullet g = id = g \bullet f$
3. $f \bullet h = h' \bullet f$
4. $(f \bullet s).x = s'.x$*, para toda* $x \in X$

donde id es la función identidad, y $\bullet$ *denota la composición de funciones.*

Lema 33. *Dado* $X \subseteq Var$*,* $\sim^X$ *es una relación de equivalencia.*

La condición 1 asegura el rol de *nil* como dirección inválida. La condición 3 garantiza que h' preserva las relaciones entre direcciones dadas por h, *i.e.* su *estructura*, y la condición 4 la consistencia de las referencias al *heap* desde el *stack*. Finalmente la condición 2 asegura que tal preservación se da en ambos sentidos.

La fortaleza de la *Sharing Logic*, heredada de la *Separation Logic*, es la capacidad de razonar localmente habilitada por el uso del operador $\langle *: \rangle$. La semántica de este operador impone sobre la relación anterior (y sobre cualquier otra que pretenda preservar la satisfacción de fórmulas) el requisito de exhibir cierta forma de *localidad*, *i.e.* la relación debe ser preservada frente a la partición y composición de *heaps*.

Lema 34 (Propiedad de partición de $\sim$)**.** *Sean* $X \subseteq Var$*,* s, h *y* $s', h' \in States_D$ *tales que* $s, h \sim^X s', h'$ *y* $h = h_1 \cup h_2$ *para algunos* $h_1, h_2 \in Heaps$ *con* $h_1 \perp h_2$*. Entonces existen* $h'_1, h'_2 \in Heaps_D$ *tales que:*

1. $s, h_1 \sim^X s', h'_1$,
2. $s, h_2 \sim^X s', h'_2$,
3. $h'_1 \perp h'_2$ *y* $h' = h'_1 \cup h'_2$.

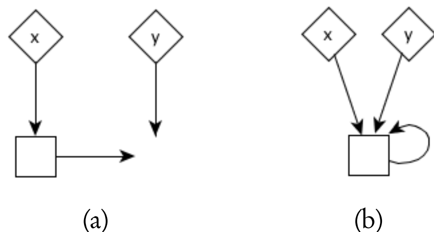


Figura 5.1: Modelos de $x \mapsto y$ cuando $x \neq y$ (a) y $x = y$ (b).

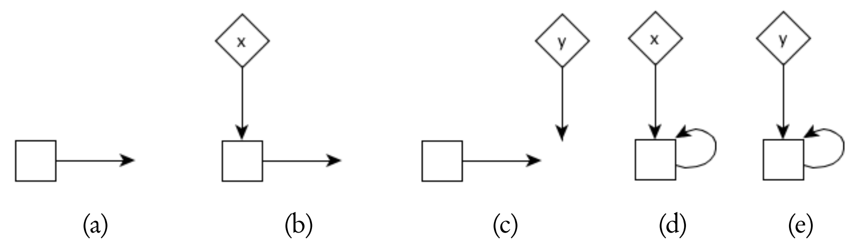


Figura 5.2: Modelos de tamaño 1 que no satisfacen $x \mapsto y$

Lema 35 (Propiedad de unión de $\sim$). *Sean $X \subseteq Var$ y $s, s' \in Stacks_D$, $h_1, h'_1, h_2, h'_2 \in Heaps_D$ tales que $s, h_1 \sim^X s', h'_1$ y $s, h_2 \sim^X s', h'_2$, donde $h_1 \perp h_2$ y $h'_1 \perp h'_2$. Entonces, si tomamos $h = h_1 \cup h_2$ y $h' = h'_1 \cup h'_2$*

$$s, h \sim^X s', h'$$

Los modelos relacionados por $\sim$ son isomorfos en el sentido que comparten todas las propiedades que puede ser expresadas por una fórmula.

Lema 36 (Preservación de satisfacción de $\sim$). *Sean $p \in Form_D$, $X \supseteq FV.p$, s, h y $s', h' \in States_D$, tales que $s, h \sim^X s', h'$. Entonces:*

$$s, h \models p \text{ si y sólo si } s', h' \models p$$

Ejemplo 50. *Consideremos la fórmula $x \mapsto y$ y tomemos $X = \{x, y\}$. Esta fórmula se satisface únicamente en la clase de modelos s, h que cumplen que $h = \{(s.x, s.y)\}$, independientemente de los valores particulares $s.x$ y $s.y$ (fig. 5.1), y que por lo tanto están relacionados por $\sim^X$. Los modelos para los cuales $h = \{(l, l')\}$ con $s.x \neq l$ o $s.y \neq l'$ nunca la satisfacen, no importa cuáles sean las direcciones l y l' (fig. 5.2).[2] Lo mismo ocurre con aquellos modelos en los que $h = \emptyset$ o $|dom.h| \geqslant 2$ sin importar cúales son las direcciones de $img.s$, $dom.h$ e $img.h$.*

[2]En este capítulo, en las representaciones esquemáticas de los modelos se grafican explícitamente todas las referencias desde el *stack* a registros en la memoria, o a direcciones referenciadas por ellos, considerando un conjunto de variables de interés que suelen ser las variables libres de la fórmula bajo consideración.

Indiscernibilidad de la basura

La *basura* se define tradicionalmente como la porción del *heap* que no es alcanzable directamente desde una variable del *stack* o transitivamente desde otro registro del mismo *heap* que es a su vez alcanzable.

Definición 82. *Dados $h \in Heaps_D$ y $L \subseteq Loc$, el conjunto de direcciones de memoria de h alcanzables desde L, denotado como $reach_L.h$, se define como*

$$reach_L.h \doteq (dom.h \cap L) \cup \{l \in dom.h \mid \text{existen } l' \in Loc, n \in \mathbb{N}_0 \text{ tales que } l = h^n.l'\}$$

Una consecuencia obvia de esta definición, es que la basura no es un concepto local sino global: un registro alcanzable en un *heap* puede no serlo si sólo consideramos una porción de ese *heap*. Además la basura es siempre en relación a un conjunto de direcciones *iniciales*. Cuando hablamos de basura respecto a una fórmula, normalmente consideramos como direcciones iniciales las contenidas en las variables libres que ocurren en la misma.

La próxima relación iguala *heaps* que contienen basura por sobre cierto límite. Debajo de él, la cantidad de basura debe ser la misma para ambos *heaps*. Por encima de él, la cantidad de basura puede variar. Respecto a los registros que no son basura, los *heaps* resultan idénticos. Para lidiar con la globalidad del concepto de basura, introducimos *heaps* auxiliares para su determinación.

Definición 83 (Relación $\approx$)**.** *Dados $L \subseteq Loc$, $n \in \mathbb{N}_0$ y $h, c, h', c' \in Heaps_D$, tales que $h \perp c$ y $h' \perp c'$, definimos la relación $\approx_n^L$ y decimos que $h, c \approx_n^L h', c'$ si:*

1. $dom.h \cap reach_L.(h \cup c) = dom.h' \cap reach_L.(h' \cup c')$;
2. $h.l = h'.l$ *para todo* $l \in dom.h \cap reach_L.(h \cup c)$;
3. *si* $|dom.h - reach_L.(h \cup c)| < n$, *entonces* $|dom.h' - reach_L.(h' \cup c')| = |dom.h - reach_L.(h \cup c)|$;
4. *si* $|dom.h - reach_L.(h \cup c)| \geqslant n$, *entonces* $|dom.h' - reach_L.(h' \cup c')| \geqslant n$.

Lema 37. *Dados $L \subseteq Loc$ y $n \in \mathbb{N}_0$, $\approx_n^L$ es una relación de equivalencia.*

La condición 1 establece que las porciones alcanzables de ambos *heaps* son las mismas, y la 2 que ambos *heaps* son iguales para tal porción. Las condiciones 3 y 4 establecen la relación de la basura en ambos *heaps* respecto al límite dado por n. Notar cómo en la definición de $\approx$ la distinción entre lo que es basura y lo que es alcanzable se realiza utilizando los *heaps* auxiliares. Esto permite que la relación se comporte bien frente a las operaciones de partición y unión.

Lema 38 (Propiedad de partición de $\approx$)**.** *Sean* $n = n_1 + n_2$, $h, h' \in Heaps_D$ *tales que* $h, c \approx^L_n h', c'$ *para algunos* $c, c' \in Heaps_D$, *y* $h = h_1 \cup h_2$ *para algunos* $h_1, h_2 \in Heaps$ *con* $h_1 \perp h_2$. *Entonces existen* $h'_1, h'_2 \in Heaps_D$ *tales que:*

1. $h_1, h_2 \cup c \approx^L_{n_1} h'_1, h'_2 \cup c'$,
2. $h_2, h_1 \cup c \approx^L_{n_2} h'_2, h'_1 \cup c'$,
3. $h'_1 \perp h'_2$ *y* $h' = h'_1 \cup h'_2$.

Lema 39 (Propiedad de unión de $\approx$)**.** *Sean* $n_1, n_2 \in \mathbb{N}_0$, $h_1, h'_1, h_2, h'_2 \in Heaps_D$ *tales que* $h_1, h_2 \cup c \approx^L_{n_1} h'_1, h'_2 \cup c'$ *y* $h_2, h_1 \cup c \approx^L_{n_1} h'_2, h'_1 \cup c'$, *para algunos* $c, c' \in Heaps_D$ *donde* $h_1 \perp h_2$ *y* $h'_1 \perp h'_2$. *Entonces si tomamos* $h = h_1 \cup h_2$ *y* $h' = h'_1 \cup h'_2$

$$h, c \approx^L_{\text{mín}.n_1.n_2} h, c'$$

Para una fórmula particular p, es posible acotar de forma segura la cantidad de basura que puede cambiar el *status* de su satisfactibilidad, a través de una función *tamaño*, denotada $|p|$ y definida a partir de su sintáxis. Cuando la basura varía en una cantidad igual o mayor a $|p|$, la satisfactibilidad de p es invariante (ya sea que valga o no). El *tamaño* de una fórmula, introducida en primera instancia en [43] y aquí adaptada para nuestra lógica, se define como sigue.

Definición 84 (Tamaño de fórmula (provisoria))**.** *Dada una fórmula* $p \in Form_D$, *la función* tamaño $|\ | \in Form_D \to \mathbb{N}_0$, *se define recursivamente en la estructura de* p *de acuerdo a las siguientes ecuaciones:*

$$\begin{aligned} |e_1 = e_2| &\doteq 0 & |\neg p| &\doteq |p| \\ |\mathbf{emp}| &\doteq 1 & |p \vee q| &\doteq max.|p|.|q| \\ |x \mapsto e| &\doteq 1 & |p\ \langle *\!: r\rangle\ q| &\doteq |p| + |q| + max.\{|p|, |q|, |r|\} \\ |\mathbf{lseg}.e_1.e_2| &\doteq 1 \end{aligned}$$

donde $e_1, e_2, e \in Expr_D$, $x \in Var$ *y* $p, q, r \in Form_D$.

Las fórmulas *pure* del tipo $e_1 = e_2$ son indiferentes al *heap* y por lo tanto a la basura que pueda contener. Las fórmulas atómicas **emp**, $x \mapsto e$ y $\mathbf{lseg}.e_1.e_2$ no admiten basura (son *precise*), y por lo tanto hay una diferencia significativa entre la inexistencia de basura y la presencia de cualquier cantidad de ella. La negación puede relajar el límite pero no puede fortalecerlo. Por ejemplo $\neg x \mapsto e$ se satisface en un *heap* consistente de una cantidad de basura mayor o igual a cero, pero esto no es cierto para $\neg\mathbf{lseg}.x.x$, que requiere que el *heap* sea no vacío. Para el operador $\vee$, la única opción segura es fortalecer el límite para cubrir los requerimientos de ambas sub-fórmulas. Finalmente, como $\langle *\!: \rangle$ impone la separación (parcial) incluso de la basura, se deben tener en cuenta los requerimientos que impone cada una de las subfórmulas por

separado. Por ejemplo, la fórmula $\neg\mathbf{emp} * \neg\mathbf{emp}$ sólo es satisfecha por *heaps* con una cantidad de basura mayor o igual a dos. Sin embargo, la interacción de $\langle *\colon \rangle$ con la basura es sutil. Una fórmula como $\neg\mathbf{emp}\ \langle *\colon \neg\mathbf{emp}\rangle\ \neg\mathbf{emp}$ no puede distinguir entre *heaps* con una cantidad de basura mayor o igual a uno.

El problema para aproximar $|p\ \langle *\colon r\rangle\ q|$ es que la basura puede estar subespecificada en p (y q), pero determinada en r. Ejemplo de ello es la fórmula $\mathbf{true}\ \langle *\colon \neg\mathbf{emp}\rangle\ q$, donde el *subheap* que satisface la parte izquierda de la fórmula debe contener al menos un registro basura. Más aún, la basura requerida por p puede estar determinada por q (y viceversa). Esto ocurre por ejemplo en $\mathbf{true}\ \langle *\colon \mathbf{true}\rangle\ \neg\mathbf{emp}$. Así la unica opción segura para acotar la basura de p es tomar $|p|$ en conjunto con la basura común, que necesariamente queda limitada por $max.\{|q|, |r|\}$. Se puede seguir un razonamiento simétrico para q, lo que motiva la definición dada.

Finalmente la posibilidad de acotar la cantidad de basura sin alterar la condición de satisfactibilidad de una fórmula, se expresa formalmente como sigue.

Lema 40 (Preservación de satisfacción de $\approx$)**.** *Sean $p \in Form_D$, $X \supseteq FV.p$, $n \in \mathbb{N}_0$, $L \subseteq Loc$, s, h y $s, h' \in States_D$, tales que $L \supseteq s^{\upharpoonright}.X$, $n \geqslant |p|$, y $h, c \approx_n^L h', c'$, para algunos $c, c' \in Heaps_D$ donde $h \perp c$ y $h' \perp c'$. Entonces*

$$s, h \vDash p \ \text{ si y sólo si } \ s, h' \vDash p$$

donde $s^{\upharpoonright}.X$ denota el lifting *de s al conjunto de variables X.*

Ejemplo 51. *Consideremos en detalle la fórmula $\neg x \mapsto y$, para la cual tenemos $X = \{x, y\}$ y $n = |\neg x \mapsto y| = 1$. Los únicos modelos s, h que no la satisfacen son aquellos mencionados en el ejemplo 50, que cumplen $h = \{(s.x, s.y)\}$ para cualquier combinación de valores de $s.x$ y $s.y$ (fig. 5.1). Tales modelos contienen 0 registros basura y cada* heap *está relacionado por $\approx_n^X$ únicamente consigo mismo (independientemente del* heap *auxiliar que se tome). De este razonamiento se sigue que cualquier otro modelo con registros basura satisface la fórmula. Cada* heap *de estos modelos, en función de su parte alcanzable, pertenece a una clase de equivalencia dada por $\approx_n^X$. Los* heaps *que únicamente contienen basura están todos relacionados entre sí; los* heaps *que tienen alguna parte alcanzable, con aquellos con quienes comparten tales registros. Finalmente, hay modelos s, h que no contienen basura pero que aún así satisfacen la fórmula. Son aquellos para los cuales $h = \emptyset$, $|dom.h| \geqslant 2$, o $h = \{(l, l')\}$ con $s.x \neq l$ o $s.y \neq l'$ (fig. 5.2). En todos los casos, los* heaps *están relacionados sólo consigo mismo.*

Indiscernibilidad de la longitud de segmentos

Las siguientes definiciones permiten caracterizar la *estructura fundamental* de un *heap*: una dirección *dangling* es una dirección mencionada como el "sucesor" de otra, pero que no está definida en el modelo; una dirección *de entrada* no tiene "antecesor";

una dirección *no bastarda* tiene al menos dos "antecesores" diferentes; una dirección *nombrada* es aquella mencionada explícitamente en un conjunto L dado.

Definición 85 (Dirección *dangling*). *Dado $h \in Heaps_D$, una dirección $l \in img.h - \{nil\}$ es* dangling *si $l \notin dom.h$. Denotamos con $dang.h$ el mayor conjunto $D \subseteq img.h$ tal que para cada $l \in D$, l es una dirección* dangling.

Definición 86 (Dirección de entrada). *Dado $h \in Heaps_D$, una dirección $l \in dom.h$ es una dirección* de entrada *si $l \notin img.h$. Denotamos con $entry.h$ el mayor conjunto $E \subseteq dom.h$ tal que para cada $l \in E$, l es una dirección de entrada.*

Definición 87 (Dirección no bastarda). *Dado $h \in Heaps_D$, una dirección $l \in dom.h$ es* no bastarda *si existen $l_1, l_2 \in dom.h$ tales que $l_1 \neq l_2$ y $h.l_1 = h.l_2 = l$. Denotamos con $nbastard.h$ el mayor conjunto $B \subseteq dom.h$ tal que para cada $l \in B$, l es una dirección no bastarda.*

Definición 88 (Dirección nombrada). *Dados $h \in Heaps_D$ y $L \subseteq Loc$, una dirección $l \in dom.h$ es* nombrada$_L$ *si $l \in L$. Denotamos con $named_L.h$ el conjunto $L \cap dom.h$.*

El conjunto de direcciones *relevantes* de un *heap* está compuesto por las direcciones que caen en alguna de las categorías definidas anteriormente y que pertenecen al dominio del *heap* (y por lo tanto no son direcciones *dangling*).

Definición 89 (Dirección relevante). *Dados h y $L \subseteq Loc$, una dirección $l \in dom.h$ es* relevante$_L$ *si $l \in entry.h$, $l \in nbastard.h$ o $l \in named_L.h$. Denotamos con $rel_L.h$ el mayor conjunto $R \subseteq dom(h)$ tal que para cada $l \in R$, l es relevante$_L$.*

Cualquier registro cuya dirección no es relevante es necesariamente un registro *interno* de un *segmento*. Más precisamente, un segmento es una secuencia de registros que forman una cadena enlazada tal que la dirección del primer registro es relevante, y la dirección del último es relevante o *dangling*.

Definición 90 (Segmento). *Dados un* heap *$h \in Heaps_D$ y un conjunto finito $L \subseteq Loc$, decimos que $\varsigma \subseteq h$ es un* segmento$_L$ *comenzando en l_b y terminando en l_e, denotado $\varsigma = l_b \triangleright_L l_e$, si existe una secuencia de direcciones $l_0, l_1, \ldots, l_n \in Loc$ tal que:*

1. *$|dom.\varsigma| = n$;*
2. *$l_b = l_0$ y $l_e = l_n$;*
3. *$l_i \in dom.\varsigma$ para toda l_i tal que $0 \leqslant i < n$;*
4. *$h(l_i) = l_{i+1}$ para toda l_i tal que $0 \leqslant i < n$;*
5. *$l_0, l_k \in rel_L.h \cup dang.h$; y*
6. *$l_i \notin rel_L.h$ para toda l_i tal que $1 \leqslant i < n$.*

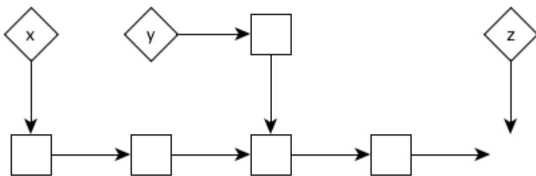


Figura 5.3: Dos segmentos con sufijo común.

Cabe remarcar que una dirección puede ser al mismo tiempo, el extremo final de dos segementos (dado que puede ser la imagen por h de diferentes direcciones), pero sólo puede ser el extremo inicial de un único segmento. Por definición los segmentos son mutuamente excluyentes y cada registro pertenece a un único segmento (ya sea que su dirección es relevante o no).

Lema 41. *Dados $h \in Heaps_D$ y $L \subseteq Loc$, para toda $l \in dom.h$ existe un único segmento$_L$ $\varsigma \subseteq h$ tal que $l \in dom.h'$.*

Las direcciones relevantes son clave para la satisfacción de las fórmulas, ya que caracterizan la *estructura fundamental* del grafo subyacente al *heap*. Las direcciones nombradas capturan los registros referenciados desde el *stack* y que son, por lo tanto, indispensables para la satisfacción de los términos $\mapsto$ y **lseg**, y las (des)igualdades entre variables. Las direcciones no bastardas son importantes en situaciones particulares de *sharing* entre estructuras abstractas. Por ejemplo, consideremos la fórmula **lseg**.$x.z$ $\langle *$: **true**$\rangle$ **lseg**.$y.z$. Algunos de sus modelos consisten en dos segmentos que comparten un sufijo común, como se grafica en la figura 5.3. El sufijo común comienza necesariamente en un registro cuya dirección es no bastarda. Esta característica es clave y distingue esta clase de modelos de otros donde uno de los segmentos está incluido completamente en otro, o donde ambos segmentos son iguales. Finalmente, las direcciones de entrada y *dangling* son indispensables por el uso de $\langle *{:}\rangle$ para la composición de modelos en términos de sus partes. Ya hemos discutido con anterioridad la interacción entre los operadores espaciales y las direcciones *dangling*.

La última relación de equivalencia que introducimos iguala *heaps* que preservan la estructura dada por los registros relevantes y *dangling*, pero permite el aumento o disminución de la longitud de los segmentos. La variación de la longitud de un segmento sólo se permite cuando la cantidad de registros irrelevantes que lo conforman está por encima de cierto límite. Debe notarse que el concepto de segmento es relativo a un *heap* específico. Agregar o eliminar un registro del *heap* puede no solamente alterar el segmento al que el registro pertenece, sino también la calificación de otros *subheaps* como segmentos, ya que las direcciones no bastardas o *dangling* podrían cambiar. Como hicimos anteriormente con la determinación de la basura,

utilizamos *heaps* auxiliares para dar cuenta de los registros que son significativos para la determinación de los segmentos.

Definición 91 (Relación $\approx$). *Dados $L \subseteq Loc$, $n \in \mathbb{N}_0$, y $h, h', c, c' \in Heaps_D$, tales que $h \perp c$ y $h' \perp c'$, definimos la relación $\approx_n^L \in (Heaps_D \times Heaps_D) \times (Heaps_D \times Heaps_D)$ y decimos que $h, c \approx_n^L h', c'$ si para todo segmento$_L$ $\varsigma \subseteq h \cup c$, con $\varsigma = l \triangleright_L l'$ para $l, l' \in Loc$, existe un segmento$_L$ $\varsigma' \subseteq h' \cup c'$ con $\varsigma' = l \triangleright_L l'$ (y viceversa) tales que:*

1. *$dom.\varsigma \cap dom.c = \emptyset$ si y sólo si $dom.\varsigma' \cap dom.c' = \emptyset$;*

2. *si $|dom.\varsigma \cap dom.h| < n$, entonces $|dom.\varsigma' \cap dom.h'| = |dom.\varsigma \cap dom.h|$;*

3. *si $|dom.\varsigma \cap dom.h| \geqslant n$, entonces $|dom.\varsigma' \cap dom.h'| \geqslant n$;*

Lema 42. *Dados $L \subseteq Loc$ y $n \in \mathbb{N}_0$, $\approx_n^L$ es una relación de equivalencia.*

Por el cuidado tomado en su definición, no es sorprendente que $\approx$ se comporte bien frente a las operaciones de partición y unión, ya que los *heaps* auxiliares permiten preservar los segmentos del *heap* original.

Lema 43 (Propiedad de partición de $\approx$). *Sean $n = n_1 + n_2$ y $h, h \in Heaps_D$, tales que $h, c \approx_n^L h', c'$ para algunos $c, c' \in Heaps_D$, y $h = h_1 \cup h_2$ para algunos $h_1, h_2 \in Heaps$ con $h_1 \perp h_2$. Entonces existen $h'_1, h'_2 \in Heaps_D$ tales que:*

1. *$h_1, h_2 \cup c \approx_{n_1}^L h'_1, h'_2 \cup c'$,*

2. *$h_2, h_1 \cup c \approx_{n_2}^L h'_2, h'_1 \cup c'$,*

3. *$h'_1 \perp h'_2$ y $h' = h'_1 \cup h'_2$.*

Lema 44 (Propiedad de unión de $\approx$). *Sean $n_1, n_2 \in \mathbb{N}_0$, $h_1, h'_1, h_2, h'_2 \in Heaps_D$ tales que $h_1, h_2 \cup c \approx_{n_1}^L h'_1, h'_2 \cup c'$ y $h_2, h_1 \cup c \approx_{n_1}^L h'_2, h'_1 \cup c'$, para algunos $c, c' \in Heaps_D$ donde $h_1 \perp h_2$ y $h'_1 \perp h'_2$. Entonces si $h = h_1 \cup h_2$ y $h' = h'_1 \cup h'_2$*

$$h, c \approx_{\text{mín}.n_1.n_2}^L h, c'$$

No es posible igualar indiscriminadamente *heaps* que sólo difieren en la longitud de los segmentos sin ningún tipo de restricción, y esperar que se preserve la satifacción de cualquier fórmula. Pero, así como ocurre con la basura, existe un límite sobre el cual el tamaño de los segmentos puede variar sin afectar el *status* de satisfactibilidad. Siguiendo un razonamiento análogo podemos pensar al *tamaño* de una fórmula como una cota para la longitud de los segmentos. Por ejemplo, **lseg**.$x.y$ puede ser satisfecha por un modelo s, h donde o bien h es vacío, o bien h contiene un segmento que comienza en $s.x$ y finaliza en $s.y$ de cualquier longitud. Sin embargo, $x \mapsto y$ también se satisface en un *heap* con un segmento similar, pero únicamente

de tamaño igual a uno. Esto nos obliga a modificar la función de cota para este caso particular, incrementando su valor. Sin embargo, ya que se trata de una cota superior, *tamaño* sigue cumpliendo su función respecto a la cantidad de basura (aunque de forma menos ajustada).

Definición 92 (Tamaño de fórmula (definitiva)). *Dada una fórmula $p \in Form_D$, la función* tamaño $|\,| \in Form_D \to \mathbb{N}_0$, *se define recursivamente en la estructura de p de acuerdo a las siguientes ecuaciones:*

$$\begin{aligned} |e_1 = e_2| &\doteq 0 & |\neg p| &\doteq |p| \\ |\mathbf{emp}| &\doteq 1 & |p \vee q| &\doteq max.|p|.|q| \\ |x \mapsto e| &\doteq 2 & |p \ \langle *\!: r\rangle \ q| &\doteq |p| + |q| + max.\{|p|, |q|, |r|\} \\ |\mathbf{lseg}.e_1.e_2| &\doteq 1 \end{aligned}$$

donde $e_1, e_2, e \in Expr_D$, $x \in Var$ y $p, q, r \in Form_D$.

Por separado, los predicados atómicos imponen restricciones mínimas sobre la longitud de los segmentos. Sin embargo, el *sharing* introducido por $\wedge$ impone condiciones sutiles. Por ejemplo, en la fórmula $\mathbf{lseg}.x.y \wedge x \mapsto y$, el término derecho restringe al *heap* de una manera más fuerte que lo estrictamente necesario determinado por el término izquierdo. Como el *sharing* al que está sujeto un término es impredecible desde el punto de vista del propio término, todos los segmentos de un modelo deben obedecer una única restricción global. Cuando esto ocurre, la satisfacción de las formulas se preserva.

Lema 45 (Preservación de satifacción de $\approx$). *Sean $p \in Form_D$, $X \supseteq FV.p$, $n \in \mathbb{N}_0$, $L \in Loc$, s, h y $s, h' \in States_D$, tales que $L \supseteq s^{\upharpoonright}.X$, $n \geqslant |p|$ y $h, c \approx^L_n h', c'$, para algunos $c, c' \in Heaps_D$ donde $h \perp c$ y $h' \perp c'$. Entonces*

$$s, h \vDash p \ \textit{ si y sólo si } \ s, h' \vDash p$$

Ejemplo 52. *Consideremos la fórmula* $\mathbf{lseg}.x.y$ *con $X = \{x, y\}$ y $n = |\mathbf{lseg}.x.y| = 1$. Si $x = y$, entonces lós únicos modelos que la satisfacen son aquellos cuyos* heaps *son vacíos, y están relacionados por $\approx^X_n$ consigo mismo. Los modelos s, h que no la satisfacen son aquellos para los cuales $|dom.h| \geqslant 1$, que, dependiendo de su estructura fundamental, están relacionados con otros que satisfacen la misma propiedad.*

Si $x \neq y$, los modelos s, h que la satisfacen constan de una serie de registros enlazados, el primero de los cuales tiene dirección $s.x$ y el último referencia a $s.y$. Tal secuencia de registros conforma un segmento $s.x \rhd_{s^{\upharpoonright}.X} s.y$ (fig. 5.1). Cada uno de los heaps *está relacionado por $\approx^X_n$ con otros con que contienen un segmento con las mismas direcciones inicial y final, aunque posiblemente de diferente longitud, lo que mantiene la satisfacción.*

Entre los modelos s, h que no satisfacen la fórmula, se encuentran aquellos para los que $h = \emptyset$, x (o y) no referencian a los registros apropiados, o h contiene dos o más segmentos no encadenados (que pueden o no ser basura). La variación sobre tales heaps *que*

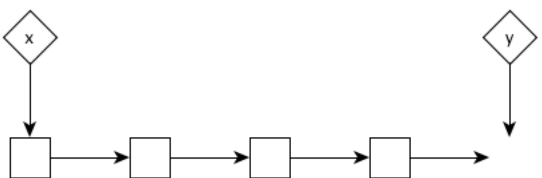


Figura 5.4: Modelo de **lseg**.$x.y$

permite $\approx_n^X$ *no altera estas características, y por lo tanto los modelos con* heaps *relacionados preservan la no satifacción.*

5.2. Modelos estructurales

Para ser capaces de decidir la validez de una fórmula, tenemos que lidiar con modelos que tienen diferencias *insignificantes*, y tratarlos como un único modelo. Para ello, definimos el concepto de *modelo estructural*. La idea clave es que cada modelo estructural *nuclea* modelos indiscernibles de acuerdo a los tres puntos discutidos anteriormente: la particular elección de las direcciones de memoria, la cantidad de registros basura, y la longitud de los segmentos.

Las relaciones de equivalencia definidas previamente, $\sim, \approx$ y $\approxeq$, pueden combinarse en una única relación de equivalencia.

Definición 93 (Relación $\cong$). *Dados* $X \subseteq Var$, $n \in \mathbb{N}_0$, *y* $s,h,\ s',h' \in States_D$, *definimos la relación* $\cong_n^X \in States_D \times States_D$ *y decimos que* $s,h \cong_n^X s',h'$ *si existen* $h_1, h_2 \in Heaps$ *tales que:*

- $s,h \sim^X s',h_1$,
- $h_1,\varnothing \approxeq_n^L h_2,\varnothing$, *y*
- $h_2,\varnothing \approx_n^L h',\varnothing$.

donde $L \doteq s'^{\upharpoonright}.X$.

Lema 46. *Dados* $X \subseteq Var$ *y* $n \in \mathbb{N}_0$, $\cong_n^X$ *es una relación de equivalencia.*

En este contexto, un *modelo estructural* es un conjunto de modelos relacionados por $\cong$.

Definición 94 (Modelo estructural). *Dados* $X \subseteq Var$ *y* $n \in \mathbb{N}_0$, *definimos el conjunto de modelos estructurales respecto a* X *y* n, *denotado como* SM_n^X, *como el conjunto cociente* $States_D/\cong_n^X$.

En virtud de los lemas 36, 40 y 45 es directo ver que $\cong$ preserva la satisfacción de fórmulas entre modelos relacionados. Como consecuencia los modelos estructurales tienen asociada una buena noción de satisfacción.

Corolario 2 (Preservación de satisfacción de $\cong$)**.** *Sean $p \in Form_D$, $X \supseteq FV.p$, s, h y $s', h' \in States_D$, y $n \geqslant |p|$ tales que $s, h \cong_n^X s', h'$. Entonces*

$$s, h \vDash p \ \text{ si y sólo si } \ s', h' \vDash p$$

Definición 95 (Satisfacción de fórmula resp. a modelo estructural)**.** *Dada una fórmula $p \in Form_D$ tal que $X \supseteq FV.p$ y $n \geqslant |p|$, un modelo estructural $\mu \in SM_n^X$ satisface p, denotado como $\mu \vDash p$, si $s, h \vDash p$ para algún $s, h \in \mu$.*

De aquí en más, el problema de la validez de una fórmula de ShL_D (respecto al modelo de memoria) puede reducirse a verificar la validez de la misma respecto a los modelos estructurales. Este problema puede decidirse, ya que el número total de modelos estructurales es finito (cuando consideramos un conjunto finito de variables). Veamos en primer lugar que cada modelo estructural puede caracterizarse a través de un modelo en *forma normal* (en el sentido débil del concepto) que garantiza la minimización de registros basura y registros irrelevantes, y que la cantidad de tales modelos es finita.

Definición 96 (Forma normal de modelo)**.** *Dados $s, h \in States_D$, $X \subseteq Var$ un conjunto finito, $n \in \mathbb{N}_0$, y $Ord \in Loc \rightarrow \mathbb{N}_0$ biyectiva tal que $Ord.nil = 0$, decimos que s, h está en* forma normal *respecto a X, n y Ord si:*

1. *$|dom.h - reach_L.h| \leqslant n$;*
2. *todo segmento$_L$ $\varsigma \subseteq h$, cumple que $|dom.\varsigma| \leqslant n$;*
3. *$s.x = nil$ para toda $x \notin X$;*
4. *$L \cup dom.h \cup img.h \subseteq \{l \in Loc \mid Ord.l \leqslant m\}$, con $m = |L \cup dom.h \cup img.h|$;*

donde $L \doteq s^{\restriction}.X$. Denotamos con $Norm_{Ord}^{X,n}$ el mayor conjunto $S \subseteq States_D$ tal que para todo $s, h \in S$, s, h está en forma normal respecto a X, n y Ord.

Es fácil de ver que todo modelo estructural contiene al menos un modelo en forma normal. Consideremos un modelo s, h: o bien s, h está en forma normal, o bien puede ser *transformado* en otro modelo s', h' equivalente a s, h, a partir de reducir la cantidad de basura y la longitud de cada segmento al valor límite n (en caso que lo sobrepasaran), elegir las direcciones de memoria entre las primeras dadas por el orden total fijado, y asignar a cualquier variable irrelevante el valor *nil*. Tal estado s', h' está en forma normal y obviamente relacionado a s, h por $\approx$, $\approxeq$ y $\sim$.

Notar que existen diversos modelos en forma normal que son equivalentes por $\sim$, dados por las diferentes permutaciones de las direcciones de memoria. Sin embargo, la cantidad de modelos en forma normal está acotada principalmente por las condiciones 1 y 2 de la definición 96, que determinan un límite para la cantidad de basura y la longitud de los segmentos, y en conjunto, un límite para el tamaño del dominio del *heap*. Como el número de permutaciones en la asignación de direcciones a los registros de la memoria dinámica y las variables de *stack* también está limitado por las condiciones 3 y 4, encontramos que existe un número finito de modelos en forma normal, y por lo tanto, un numero finito de modelos estructurales.

Teorema 5. *Sean $X \subseteq Var$ un conjunto finito, $n \in \mathbb{N}_0$ y $Ord \in Loc \to \mathbb{N}_0$ biyectiva tal que $Ord.nil = 0$:*

1. *Para cada $s, h \in StatesD$ existe $s', h' \in Norm^{X,n}_{Ord}$ tal que $s, h \cong^X_n s', h'$.*
2. *Si $s, h \in Norm^{X,n}_{Ord}$, entonces $|dom.h| \leqslant 2 * |X| * n$.*
3. *$|Norm^{X,n}_{Ord}| \leqslant (N * (N - 1)/2)^{|X|}$, donde $N = 2 * |X| * n$.*

Corolario 3. *Dados $X \subseteq Var$ un conjunto finito y $n \in \mathbb{N}_0$, $|States_D/ \cong^X_n|$ es finito.*

5.3. Procedimiento de decisión

Veamos ahora que es posible representar SM^X_n con el conjunto finito de modelos s, h que utilizan las primeras m direcciones de memoria, donde m es una cota para $|s^{\upharpoonright}.X| + |dom.h| + |img.h|$, con $|dom.h| < 2 * |X| * n$, y que por lo tanto incluye todos los modelos en forma normal.

Definición 97 (Representación de SM^X_n)**.** *Dados $X \subseteq Var$ un conjunto finito, $n \in \mathbb{N}_0$ y $Ord \in Loc \to \mathbb{N}_0$ biyectiva tal que $Ord.nil = 0$, y sean*

- $m = (4 * n + 1)|X|$,
- $L = \{l \in Loc \mid 0 < Ord.l \leqslant m\}$,
- $S_X = X \to (L \cup nil)$,
- $H = \{h \in L \rightharpoonup_{fin} (L \cup nil) \mid |dom.h| \leqslant 2 * |X| * n\}$.

Entonces

$$SM^{X,n}_{Ord} = \{s', h \mid h \in H \text{ y } s' = s \cup \{(x, nil) | \ x \notin dom.s\}, \text{para } s \in S_X\}$$

Notar que siempre es posible construir el conjunto finito de funciones (parciales) de A en B, cuando A y B son finitos.

Teorema 6. *Sean $X \subseteq Var$ un conjunto finito, $n \in \mathbb{N}_0$ y $Ord \in Loc \to \mathbb{N}_0$ biyectiva tal que $Ord.nil = 0$. Entonces:*

1. *es posible generar efectivamente $\mathtt{SM}^{X,n}_{Ord}$;*

2. *$Norm^{X,n}_{Ord} \subseteq \mathtt{SM}^{X,n}_{Ord}$; luego, para todo $\mu \in SM^X_n$ existe $s, h \in \mathtt{SM}^{X,n}_{Ord}$ tal que $s, h \in \mu$.*

Finalmente, como la satisfacción de una fórmula p respecto a un modelo es decidible, podemos *decidir efectivamente* la validez de p, verificando su satisfacción respecto a la representación de SM^X_n.

Lema 47. *Dados $p \in Form$, $s, h \in States$, $s, h \vDash p$ es decidible.*

Corolario 4 (Procedimiento de decisión)**.** *Sean $p \in Form_D$, $X = FV.p$, $n = |p|$ y $Ord \in Loc \to \mathbb{N}_0$ biyectiva tal que $Ord.nil = 0$. Entonces p es* válida *si $s, h \vDash p$ para todo $s, h \in \mathtt{SM}^{X,n}_{Ord}$.*

Ejemplo 53. *Consideremos la fórmula $p \doteq x \mapsto y \wedge x \neq y \Rightarrow$ **lseg**$.x.y$, donde $FV.p = \{x, y\}$ y $|p| = 2$. Definamos $q \doteq x \mapsto y \wedge x \neq y$, $r \doteq$ **lseg**$.x.y$ Para decidir la validez de p es necesario revisar los modelos s, h cuyos* heaps *tienen a lo sumo 8 registros. Los modelos con $h = \emptyset$ satisfacen p porque no satisfacen q. De entre los modelos con $|dom.h| = 1$, sólo aquellos como el de la figura 5.1.a (pág. 128) satisfacen q (y ya hemos visto que tales modelos también satisfacen r). Los restantes modelos, como los de la figura 5.1.b y 5.2 (pág. 128) no satisfacen q y por lo tanto satisfacen p. Finalmente, ningún modelo tal que $|dom.h| \geqslant 2$ satisface q, por lo tanto, todos satisfacen p, y así se verifica su validez.*

5.4. Discusión

En este capítulo presentamos un procedimiento efectivo para decidir la validez de fórmulas de un fragmento de la *Sharing Logic*, sin cuantificadores, pero que incluye un predicado específico para caracterizar segmentos de listas. El procedimiento presentado es impracticable (a primera vista tiene complejidad EXPTIME) aunque está en línea con los resultados sobre la *Separation Logic*: la validez de 2-SL$^{\forall}$ es PSPACE [43], la complejidad de 1-SL* hereda la de la lógica de segundo orden monádica (NELEMENTARY) [30], incluso la extensión del lenguage de *symbolic heaps* con ciertas formas de predicados recursivos es EXPTIME [3].

Ya que la *Sharing Logic* subsume la *Separation Logic*, nuestro resultado abarca la decidibilidad de 1-SL$^{\forall,*}$ extendida con el predicado **lseg**. Pero su utilidad no se limita sólo a ello, ya que en la mayoría de los fragmentos decidibiles de la *Separation Logic* el operador $-\!\!*$ está ausente. En tales contextos, el poder expresivo de la *Sharing Logic* ni siquiera puede ser *aproximado* (ver sección 4.1).

Aunque el resultado de decidibilidad de 1-SL* está cubierto por [30], los medios para su demostración son muy diferentes. En el mencionado trabajo el problema de la decidibilidad se reduce a un problema equivalente de la lógica de segundo orden monádica. La mayor parte de los trabajos citados al comienzo del capítulo siguen una metodología similar que, aunque resulta práctica porque permite reutilizar resultados conocidos de otras lógicas, tiene sus desventajas.

Por un lado, resulta complejo recorrer el camino inverso para comprender cuáles son las características de la semántica de la *Separation Logic* respecto a su modelo estándar que habilitan la decidibilidad, y cuales son aquellas otras que pueden limitarla para algunas extensiones y/o fragmentos de la lógica. Por el contrario, nuestra demostración sigue ideas intuitivas comunes sobre la equivalencia de modelos y permite ganar comprensión sobre la naturaleza de los mismos. Da un marco que soporta el razonamiento informal sobre las clases de modelos que satisfacen una fórmula, al mismo tiempo que resalta cuestiones que no deberían pasarse por alto, como por ejemplo, la interacción de $\langle *\colon \rangle$ con los registros basura, la restricciones a la longitud de los segmentos que impone $\wedge$, etc.

Por otro lado, la mayor parte de los resultados conocidos, salvando algunas de las recientes extensiones sobre *symbolic heaps*, no pueden extenderse para soportar otras estructuras de datos más allá de segmentos de listas, por limitaciones propias a las lógicas utilizadas. En nuestro caso no resulta demasiado difícil prever cómo caracterizar estructuras de registros dentro del *heap*, de manera análoga a nuestro concepto de *segmento*, para dar cuenta de otras estructuras de datos como, por ejemplo, árboles binarios; ni cómo, de forma análoga a $\approx$, caracterizar la equivalencia entre modelos que no distinguen el *tamaño* de tales estructuras.

Finalmente, podemos agregar que no parece haber ninguna dificultad para extender trivialemente nuestro resultado a un fragmento que incluya cuantificadores, lo que es coherente con el resultado de [30]. Originalmente restringimos esta posibilidad para intentar incluir el operador $\langle -\!\!*\colon \rangle$ (o al menos $-\!\!*$), lo que es incompatible con la presencia de cuantificadores. Sin embargo vimos limitada nuestra pretensión original por la mala interacción de este operador con los registros basura, ya que su semántica versa sobre potenciales extensiónes al *heap* que pueden cambiar el *status* de alcanzabilidad de los registros. En el fragmento decidibile 2-SL$^{\forall}$ esto no es un problema ya que todo registro relevante está referenciado por una variable en el *stack*. Pero cuando se incluye un predicado sobre segmentos de listas se vuelve difícil diferenciar entre la basura y una cadena de registros que, correctamente extendida, sea la porción del modelo que satisface un término **lseg**. Sorprendentemente el fragmento estudiado en el reciente artículo [60] muestra la compatibilidad entre los segmentos de listas y $-\!\!*$ que nos hace descartar nuestra conjetura sobre su incompatibilidad intrínseca y nos obliga a seguir trabajando para comprender los alcances de este resultado y analizar cómo sortear los impedimentos técnicos de nuestra de-

mostración. Poder avanzar en esta dirección es interesante porque $\twoheadrightarrow$ juega un papel muy importante en el sistema deductivo de ShL^H.

Automatización de *Separation Logic*

6

Shape analysis es una forma de análisis estático [113] de código para programas imperativos que utilizan memoria dinámica, que intenta descubrir y verificar propiedades sobre estructuras de datos enlazadas como listas, árboles, etc. El propósito de un análisis de este tipo no sólo es reportar accesos a punteros *dangling*, descubrir *memory leaks*, o recolectar información sobre *aliasing* entre variables. Su objetivo es, además, analizar en profundidad la memoria dinámica para validar propiedades no triviales sobre la *forma* de las estructuras que se manipulan durante la ejecución de un programa.

En la literatura se describen una variedad de *shape analysis* basados en diferentes teorías, por ejemplo: *3-valued Logic* [99, 137], lógica monádica de segundo orden [104, 89], y *Separation Logic* [64, 11, 111, 47]. En particular, esta última ha recibido gran atención debido a las ventajas del razonamiento local que habilita, lo que da lugar a análisis que pueden ser extendidos de forma relativamente sencilla para soportar programas de gran escala [37, 9, 36, 154, 74], programas concurrentes [75, 148, 42, 38], y programas orientados a objetos [65, 142].

Los *shape analysis* basados en *Separation Logic* consideran los programas como transformadores de estados abstractos. Se basan en la ejecución simbólica del programa sobre un estado caracterizado como un conjunto de fórmulas restringidas, los *symbolic heaps* mencionados en el capítulo anterior (pág. 124). Un estado abstracto representa todos los estados (concretos) que satisfacen alguno de los *symbolic heaps* que lo componen.

La ejecución simbólica de un programa se inspira en las ternas de Hoare que caracterizan los comandos de manipulación del *heap*. La idea básica puede graficarse a través de la especificación global del comando de mutación (def. 27, pág. 60):

$$\{ x \mapsto \vec{e} * p \} \; x.i := e' \; \{ x \mapsto \vec{e}[i \mapsto e'] * p \}$$

donde la postcondición se actualiza *localmente*, de manera que refleja de forma precisa la actualización del estado concreto que ocurre con la ejecución del comando. El uso de $*$ hace innecesaria la verificación del impacto que esta modificación pueda tener en el estado global. Como hemos visto, todos los comandos de manipulación del *heap* pueden caracterizarse de igual manera, lo que permite pensar la ejecución de un programa como sucesivas transformaciones *sintácticas* de estados abstractos. Sin embargo para que esto pueda darse, se requiere que la precondición tenga una forma particular, en la que el puntero manipulado esté mencionado explícitamente. Para

garantizarlo, es que los *symbolic heaps* tienen la forma $\pi \wedge \sigma$, donde π es está compuesto únicamente por términos *pure* y σ es una combinación con $*$ de términos sobre el *heap*.

Los mencionados *shape analysis* utilizan un fragmento de la *Separation Logic* que incluye predicados espaciales que caracterizan estructuras de datos lineales, posiblemente combinadas de formas intrincadas, como listas ligadas, listas doblemente ligadas, listas de listas, etc. Si consideramos la posibilidad de extender el *shape analysis* de [64] para dar soporte a estructuras de datos no lineales, como árboles y grafos generales, podemos prever algunas dificultades:

- El uso de un predicado naif para describir las estructuras puede dar lugar a una inmanejable proliferación de ocurrencias de los predicados en los *symbolic heaps*, dada la recursión múltiple inherente. En el caso de grafos, no existe en la literatura ningún predicado bien establecido que sirva para especificarlos adecuadamente.

- Los algoritmos sobre árboles y grafos son, con frecuencia, más complejos que sus contrapartes sobre listas. Usualmente involucran una manipulación intensiva de punteros y estructuras de control anidadas. Un ejemplo de ello es el algoritmo de marcado de Schorr-Waite, que suele proponerse como "el" desafío de cualquier nueva metodología de verificación sobre punteros. La complejidad de estos algoritmos imponen fuertes requisitos de precisión al momento de calcular los invariantes de ciclos, necesarios para la verificación de propiedades interesantes.

- Normalmente cada camino particular en el recorrido de una estructura no lineal es relevante para la validez de las propiedades satisfechas por un algoritmo. Sin embargo, los múltiples enlaces en este tipo de estructuras dan lugar a un número exponencial de caminos que las recorren. Esto impone la necesidad de encontrar un balance entre precisión y abstracción para prevenir un crecimiento excesivo en el número de *symbolic heaps* que caracaterizan un estado abstracto.

En este capítulo presentamos un *shape analysis*, que denominamos $\mathrm{SL}_{\mathscr{A}}$, con garantía de terminación que, dada una precondición para un programa que manipula estructuras de datos no lineales, computa automáticamente una postcondición e invariantes para cada ciclo dentro del mismo. El análisis ajusta el nivel de abstracción en la computación de invariantes teniendo en cuenta la información sobre cada variable que es necesario mantener a lo largo de la ejecución, para dar cuenta de las ulteriores manipulaciones a las mismas. Así, los invariantes calculados resultan ser compactos y lo suficientemente precisos en la mayor parte de los casos prácticos.

Para definir nuestro análisis introducimos un predicado recursivo lineal que describe (familias de) árboles binarios, dando lugar a especificaciones compactas de las estructuras de datos "parciales" que ocurren en los puntos intermedios de la ejecución de los algoritmos iterativos. Este predicado puede ser adaptado fácilmente para soportar otras estructuras de datos como árboles balanceados, *threaded trees*, e incluso estructuras con formas particulares de *sharing* como *dags*, grafos, etc.

Introducimos el análisis en tres etapas. Primero presentamos el modelo de estados y semántica *concretos* que definen el lenguaje de programación para algoritmos que manipulan árboles binarios. Luego presentamos una semántica *intermedia* ejecutable en la que los estados (abstractos) de programa están dados por conjuntos de *symbolic heaps*. Esta semántica no alcanza para definir un análisis por la posibilidad que los estados *crezcan* indefinidamente en la ejecución de un ciclo impidiendo alcanzar un punto fijo (invariante). Finalmente refinamos esta semántica introduciendo una fase de abstracción (o *widening*) que transforma cada *symbolic heap* en una *forma canónica*. Ya que existe una cantidad finita de tales formas, la semántica permite encontrar invariantes para los ciclos. Así terminamos definiendo una semántica abstracta ejecutable y con garantías de terminación.

6.1. Semántica concreta

Para dar semántica al lenguaje de programación utilizamos una restricción del modelo de memoria de la definición 7 (pág. 42). Fijamos como valores atómicos el conjunto de números naturales y limitamos los registros de memoria dinámica a tuplas de tres campos. Esto permite representar estructuras cuyos nodos poseen dos punteros a las subestructras y un valor, como es el caso de los árboles y grafos binarios. Sin embargo ninguno de nuestros resultados se basa en esta limitación, pudiendo ser extendidos fácilmente para considerar estructuras con nodos con un número mayor de punteros. Además distinguimos dos conjuntos de variables: *Var* para las variables de programa y *Var*′ para variables lógicas (que están cuantificadas implícitamente en la semántica de las fórmulas y no son accesibles desde los programas).

Definición 98 (Estados concretos de $SL_{\mathscr{A}}$). *Dado un conjunto infinito de valores Loc (interpretado como direcciónes de memoria), con un elemento distinguido nil* $\in$ *Loc, el conjunto* $\mathbb{N}_0$ *de números naturales, y sendos conjuntos enumerables de variables Var y Var′, el conjunto de estados concretos* $States_A$ *está dado por las siguientes ecuaciones:*

$$
\begin{aligned}
Values_A &\doteq Loc \cup \mathbb{N}_0 \qquad\qquad donde\ Loc \perp \mathbb{N}_0 \\
Stacks_A &\doteq Var \cup Var' \to Values_A \\
Heaps_A &\doteq (Loc - \{nil\}) \rightharpoonup_{fin} Values_A \times Values_A \times Values_A \\
States_A &\doteq Stacks_A \times Heaps_A
\end{aligned}
$$

El lenguaje de programación es similar al de SL^H, presentado en la definición 15 (pág. 51), con algunas restricciones. Las expresiones se fijan para $\mathbb{N}_0$, las relaciones se limitan a igualdades y desigualdades, y las expresiones *booleanas* se restringen a formas disjuntiva normales.

Definición 99 (Comandos de $SL_{\mathscr{A}}$). *Los conjunto de expresiones* $Expr_A$*, relaciones* Rel_A*, expresiones* booleaneas $Bool_A$ *y comandos* $Comm_A$ *estan dados por la siguiente gramática:*

$$
\begin{aligned}
Expr_A \ni e &::= x \mid \mathbf{null} \mid 0 \mid 1 \mid 2 \mid \ldots \\
Rel_A \ni r &::= e = e \mid e \neq e \\
Bool_A \ni b &::= (r \wedge \ldots \wedge r) \vee \ldots \vee (r \wedge \ldots \wedge r) \\
Comm_A \ni c &::= x := e \mid x := \mathbf{cons}(e,e,e) \mid x.i := e \mid x := x.i \mid \mathbf{dispose}(x) \\
& \quad c;c \mid \mathbf{if}\ b\ \mathbf{then}\ c \mathbf{else}\ c\ \mathbf{fi} \mid \mathbf{while}\ b\ \mathbf{do}\ c \mathbf{od}
\end{aligned}
$$

donde $i \in \{0,1,2\}$.

Siguiendo el marco de trabajo de *abstract interpretation* [57], la semántica concreta del lenguaje de programación está dada por funciones continuas $[\![c]\!]_c \in D_c \to D_c$ para cada comando c sobre el reticulado completo D_c. D_c se define como el conjunto de partes de $States_A$, con un elemento distinguido **abort** que representa una ejecución que termina anormalmente por un error de memoria.

Definición 100. *Dado el conjunto* $States_A$ *de estados y* **abort** *un elemento distinguido, definimos el reticulado* $D_c \doteq (\mathbb{P}.States_A \cup \{\mathbf{abort}\}, \subseteq)$*, donde* $\subseteq$ *es la relación de inclusión de conjuntos usual que además cumple que:*

1. *para todos* $d, d' \in D_c$ *si* $\mathbf{abort} \in d$ *y* $\mathbf{abort} \in d'$ *entonces* $d = d'$*;*

2. *para todo* $d \in D_c$, $d \subseteq \{\mathbf{abort}\}$.

Lema 48. *El reticulado* D_c *es completo.*

La semántica de los comandos atómicos (asignación, construcción, consulta, mutación y destrucción) puede definirse a partir de la semántica operacional dada en la definición 18 (pág. 53). Para cada comando atómico, $\rightsquigarrow$ relaciona un estado con otro estado, o con **abort** cuando se da un fallo de memoria. Podemos entonces intepretar la relación como una función $\rightsquigarrow \in States_A \to (States_A \cup \{\mathbf{abort}\})$ para cada comando atómico. Esta función puede extenderse fácilmente a una función sobre el dominio D_c.

Definición 101. *Dado $a \in Comm_A$ un comando atómico, la función $a^{\upharpoonright} \in D_c \to D_c$ se define como:*

$$a^{\upharpoonright}.d \doteq \begin{cases} \{\mathbf{abort}\} & \textit{si } \mathbf{abort} \in d \\ \{s', h' \in States_A \mid \textit{existe } s, h \in d \textit{ tal que } a, (s, h) \rightsquigarrow s', h'\} & \textit{en c.c.} \end{cases}$$

Si mantenemos la semántica de las expresiones *booleanas* $(\!|\ |\!)_s$ respecto a un *stack* s como la de SL^H (def. 16, pág. 52), y utilizamos las funciones $a^{\upharpoonright}$ como semántica para los comandos atómicos, el lenguaje de programación se completa con un semántica denotacional estándar.

Definición 102 (Semántica concreta de $\mathrm{SL}_{\mathscr{A}}$)**.** *La semántica concreta de comandos $[\![\ ]\!]_c \in Comm_A \to D_c \to D_c$ se define como:*

$$\begin{aligned} [\![a]\!]_c &\doteq a^{\upharpoonright} \\ [\![c_1; c_2]\!]_c &\doteq [\![c_1]\!]_c \bullet [\![c_2]\!]_c \\ [\![\mathbf{if}\ b\ \mathbf{then}\ c_1 \mathbf{else}\ c_2\ \mathbf{fi}]\!]_c &\doteq ([\![c_1]\!]_c \bullet \mathit{filter}_c.b) \cup ([\![c_2]\!]_c \bullet \mathit{filter}_c.\neg b) \\ [\![\mathbf{while}\ b\ \mathbf{do}\ c_1 \mathbf{od}]\!]_c &\doteq \lambda d \cdot \mathit{filter}_c.\neg b.(\mu d' \cdot d \cup ([\![c_1]\!]_c \bullet \mathit{filter}_c.b).d') \end{aligned}$$

donde $a \in Comm_A$ es atómico, $c_1, c_2 \in Comm_A$; $b \in Bool_A$, μ denota el operador de mínimo punto fijo, y $\neg b$ la meta-operación que transforma la expresión booleana *b en su negación en forma disjuntiva normal. La función $\mathit{filter}_c.b$ remueve los estados que son inconsistentes con el valor de verdad de la expresión* booleana *b:*

$$\mathit{filter}_c \in Bool_A \to D_c \to D_c$$

$$\mathit{filter}_c.b.d \doteq \begin{cases} \{\mathbf{abort}\} & \textit{si } \mathbf{abort} \in d \\ \{s, h \in d \mid \textit{si } (\!|b|\!)_s\} & \textit{en c.c.} \end{cases}$$

6.2. Semántica intermedia

Nuesto análisis considera los programas como transformadores de estados abstractos. La semántica intermedia opera sobre conjuntos de fórmulas de un fragmento restringido de la SL, los llamados *symbolic heaps*. Intuitivamente un estado abstracto representa a todos aquellos estados concretos que satisfacen alguna de las fórmulas que lo conforman.

Estados abstractos

Un *symbolic heap*, denotado como $\pi \,|\, \sigma$, está conformado por una fórmula *pure* π que combina igualdades y desigualdades, y una fórmula *espacial* σ que caracteriza la *forma* de las estructuras en la memoria.

Definición 103 (Sintáxis de *symbolic heaps*). *Los conjuntos de expresiones $Expr'_A$ que incluyen variables en Var', las relaciones entre ellas Rel'_A, las expresiones de multiconjuntos $Multiset$, las fórmulas puras Π, las fórmulas espaciales Σ y los* symbolic heaps *SH se definen por la siguiente gramática:*

$$
\begin{aligned}
Expr'_A \ni e &::= x \mid x' \mid \mathbf{null} \mid 0 \mid 1 \mid 2 \mid \dots \\
Rel'_A \ni r &::= e = e \mid e \neq e \\
Multiset \ni s &::= \emptyset \mid \{\!\![e, e, \dots, e]\!\!\} \\
\Pi \ni \pi &::= \mathbf{true} \mid r \mid \pi \wedge \pi \\
\Sigma \ni \sigma &::= \mathbf{emp} \mid \mathbf{true} \mid \mathbf{junk} \mid y \mapsto e, e, e \mid \mathbf{trees}.s.s \mid \sigma * \sigma \\
SH \ni sh &::= \pi \wedge \sigma
\end{aligned}
$$

donde $x \in Var$, $x' \in Var'$, e $y \in Var \cup Var'$. Al referirnos a un symbolic heap *$\pi \wedge \sigma$ preferimos utilizar la notación $\pi \,|\, \sigma$ para mayor claridad.*

En adelante utilizamos las metavariables $C, D, E, F, \dots$ para denotar expresiones de multiconjuntos, $\uplus$ como la operación sintáctica de suma de dos expresiones de multiconjuntos, y $\oplus$ como la operación sintáctica de inserción de una instancia de un elemento en una expresión de multiconjunto, que usaremos principalmente para distinguir una ocurrencia de una expresión particular.

La semántica de los *symbolic heaps* sigue la presentada en la definición 10 (pág. 46). El propósito de las variables en Var' es representar valores irrelevantes y por lo tanto están cuantificadas existencialmente en la semántica de los *symbolic heaps*. El predicado **junk** tiene el propósito de especificar que existe *basura* en el *heap*, es decir, alguna cantidad de registros no accesibles. Se diferencia de **true** en que éste abre la posibilidad que el *heap* sea vacío, y por lo tanto sólo denota la *posible* existencia de basura. La semántica formal se presenta a continuación, a excepción del predicado **trees** que se discute más adelante.

Definición 104 (Semántica de *symbolic heaps*). *Dado un modelo* $s, h \in States_A$ *las relación de satisfacción* $\models \in States_A \times (\Pi \cup \Sigma \cup SH)$ *está definida por inducción de acuerdo a las siguientes ecuaciones:*

$s, h \models r$	*si y sólo si* $\{\![r]\!\}_s$
$s, h \models \mathbf{true}$	*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* $dom.h = \emptyset$
$s, h \models \mathbf{junk}$	*si y sólo si* $dom.h \neq \emptyset$
$s, h \models x \mapsto e_1, e_2, e_3$	*si y sólo si* $h.[\![x]\!]_s = ([\![e_1]\!]_s, [\![e_2]\!]_s, [\![e_3]\!]_s)$ *y* $\lvert dom.h \rvert = 1$
$s, h \models \sigma * \sigma'$	*si y sólo si existen* h_1 *y* h_2 *tal que* $h_1 \perp h_2$ *y* $h_1 \cup h_2 = h$ *y* $s, h_1 \models \sigma$ *y* $s, h_2 \models \sigma'$
$s, h \models \pi \mid \sigma$	*si y sólo si existe* $\vec{v}$ *tal que* $s[\vec{x}' \mapsto \vec{v}], h \models \pi$ *y* $s[\vec{x}' \mapsto \vec{v}], h \models \sigma$

donde $r \in Rel'_A$, $x \in Var \cup Var'$, $e_1, e_2, e_3 \in Expr'_A$, $\pi, \pi' \in \Pi$, $\sigma, \sigma \in \Sigma$, $\vec{x}' \in Var^+$ *es la secuencia de variables que ocurren en* $\pi \mid \sigma$, *y* $\vec{v} \in Values^+$ *es una secuencia de valores.*

La intención del predicado **trees** es definir una familia de árboles binarios. De forma más general, **trees**.$C.D$ define una familia de árboles posiblemente parciales con raíces en C (a las que llamamos punteros *de entrada*), y punteros *de salida* en D. Por un lado un puntero *de salida* de un árbol parcial puede referenciar un nodo interno de otro árbol. Además otros árboles pueden tener punteros *de salida* hacia el mismo nodo. Por otro lado, un puntero *de salida* puede ser *dangling*. Así **trees**.$C.D$ define en realidad una grafo acíclico dirigido de nodos binarios, donde el *sharing*, si existe, queda restringido a los registros mencionados explícitamente en D. Un puntero no puede aparecer más de una vez en C sin dar lugar a una inconsistencia. Pero si puede aparecer múltiples veces en D. La cantidad de ocurrencias indica la cantidad de referencias a la dirección dada por el valor del puntero, sin considerar la referencia desde el árbol que define el registro con tal dirección, si existiera.

Definición 105 (Semántica de **trees**). *Dado* $s, h \in States_A$, *definimos que* $s, h \models$ **trees**.$C.D$ *si y sólo si:*

1. $s, h \models \mathbf{emp}$, *si* $C \approx \emptyset$ *y* $D \approx \emptyset$; *o*
2. $s, h \models e = \mathbf{null} \mid \mathbf{trees}.C'.D$, *si existen* e *y* C' *tales que* $C \approx e \oplus C'$; *o*
3. $s, h \models e = \mathbf{null} \mid \mathbf{trees}.C.D'$, *si existen* e *y* D' *tales que* $D \approx e \oplus D'$; *o*
4. $s, h \models e = e' \mid \mathbf{trees}.C'.D'$, *si existen* e *y* C' *tales que* $C \approx e \oplus C'$ *y* e' *y* D' *tales que* $D \approx e' \oplus D'$; *o*
5. $s, h \models x \neq e' \wedge x \neq \mathbf{null} \mid x \mapsto l', v', r' * \mathbf{trees}.(l' \oplus r' \oplus C').D$, *si existen* x *y* C' *tales que* $C \approx x \oplus C'$ *y para todos* e' *y* D' *tales que* $D \approx e' \oplus D'$.

donde $x \in Var \cup Var'$, $l', v', r' \in Var'$, $e, e' \in Expr'_A$, $C, C', D, D' \in Multiset$, *y* $\approx$ *denota la igualdad sintáctica.*

El caso 1 representa el árbol vacío. Notar que se requiere que $D \approx \emptyset$ y por lo tanto todos los punteros de salida mencionados deben ser alcanzados. Los casos 2 y 3 especifican que los punteros *nil* no influyen en la estructura definida. El caso 4 caracteriza la parcialidad y el *sharing* entre árboles. Para una fórmula **trees**.$C.D$ un puntero que se encuentra tanto en C como en D, representa un *corte* en la definición recursiva del árbol al cual pertenece, y una vez alcanzado deja de ser considerado, y es eliminado de C y D. Luego el registro referenciado puede ser definido a través de otro puntero. Finalmente el caso 5 es análogo a la definición recursiva usual, donde se explicita la existencia del registro referenciado, y se agregan los punteros de los subárboles como punteros de entrada en la recursión.

Ejemplo 54. *La fórmula* **trees**.$\{\!\{x, y\}\!\}.\emptyset$ *se satiface en modelos como los de la figura 6.1. Como no existen punteros de salida, no hay posibilidad de parcialidad o subestructuras compartidas. Podemos afirmar entonces que cuando el valor de un puntero de entrada no ocurre dentro de los punteros de salida, entonces referencia necesariamente un registro existente en la memoria dinámica.*

Ejemplo 55. *La fórmula* **trees**.$\{\!\{x, y\}\!\}.\{\!\{z\}\!\}$ *se satisface en diferentes clases de modelos (fig. 6.2). Aquí z puede indicar parcialidad de uno de los árboles, o* sharing *entre ellos. Podemos afirmar entonces que un puntero de salida no necesariamente es* dangling.

Cabe remarcar, aunque pueda resultar una obviedad, que tales situaciones pueden darse dentro *de un mismo árbol. La fórmula podria ocurrir en un contexto más amplio como* $t \mapsto x, v, y \ast$ **trees**.$\{\!\{x, y\}\!\}.\{\!\{z\}\!\}$.

Ejemplo 56. *La fórmula* **trees**.$\{\!\{x, y\}\!\}.\{\!\{z, z\}\!\}$ *se satisface en modelos como los de la figura 6.3, entre otros, pero no en los modelos del ejemplo anterior. Aquí la* cantidad *de valores z indica la cantidad de árboles que* comparten *el nodo referenciado. Tal nodo podría estar definido por un tercer árbol que lo alcance.*

Ejemplo 57. *En la figura 6.4 se presenta un modelo de* **trees**.$\{\!\{x, y\}\!\}.\{\!\{z, w, v, v\}\!\}$. *Notar cómo w es un puntero* dangling *del árbol que comienza en x, z es un puntero al subárbol compartido entre los árboles que comienzan en x e y, y v es un puntero* dangling *compartido entre ambos.*

El predicado **trees** es útil para especificar las estructuras parciales que ocurren durante la iteración de los ciclos, y tiene buenas propiedades sintácticas. Su necesidad y utilidad son análogas a las del predicado **lseg** en la demostración de programas sobre listas. El potencial *sharing* que habilita es manejable dada la clase de fórmulas que usualmente especifican las precondiciones de interés, donde el multiconjunto de

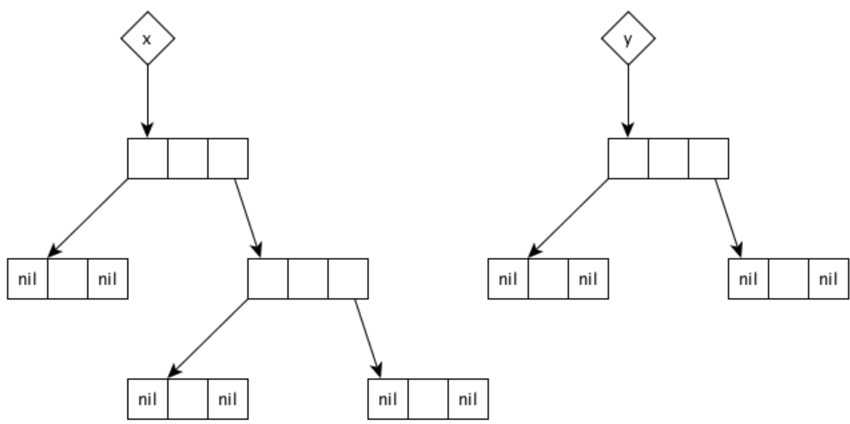


Figura 6.1: Modelo de **trees**.$\{\![x, y]\!\}.\emptyset$.

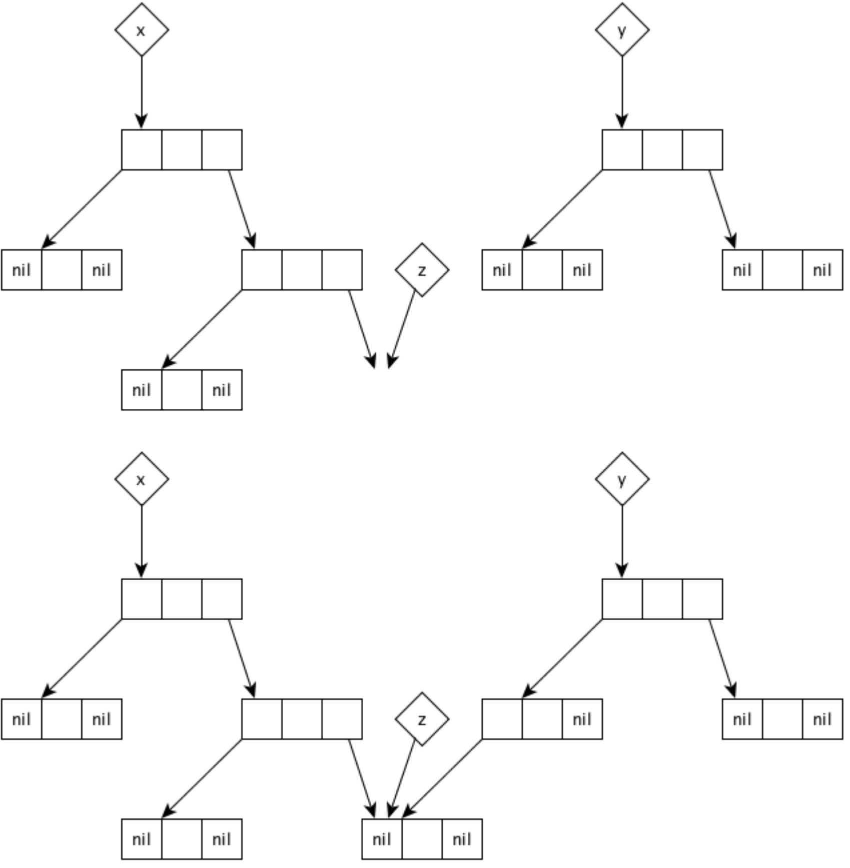


Figura 6.2: Modelos de **trees**.$\{\![x, y]\!\}.\{\![z]\!\}$.

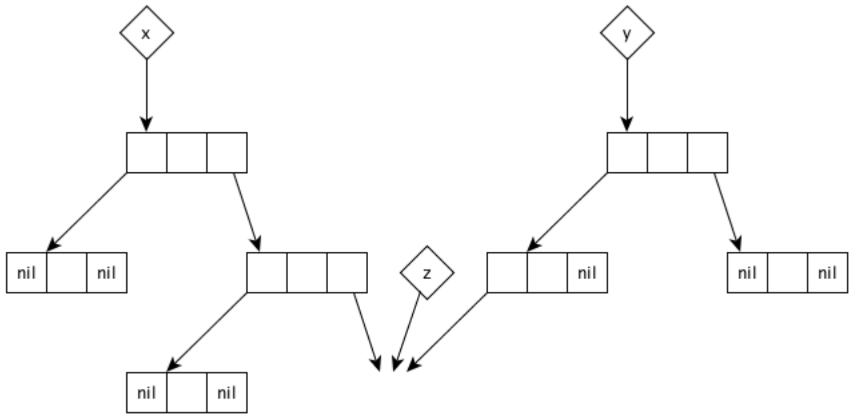


Figura 6.3: Modelo de **trees**.$\{\![x, y]\!\}.\{\![z, z]\!\}$.

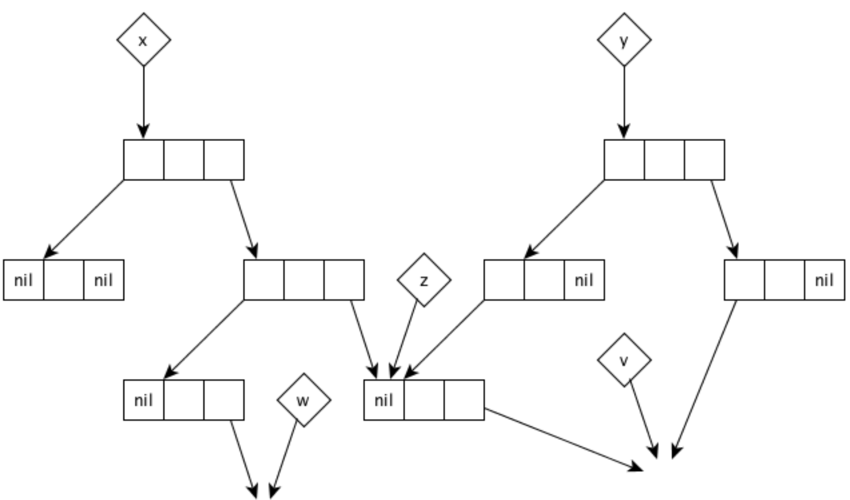


Figura 6.4: Modelo de **trees**.$\{\![x, y]\!\}.\{\![z, w, v, v]\!\}$.

punteros de salida es inicialmente vacío. La relación con el predicado **btree** introducido en la sección 3.5 es directa.

Lema 49. *Sean* $s, h \in States_A$, *también considerado dentro de* $States$, $s' \in Stacks'$ *arbitraria*, $\alpha \in Pred$ *que satisface las ecuaciones de la definición 34 (pág. 70), y* $e_0, \ldots, e_n \in Expr'_A$. *Entonces*

$$s, h \models \mathbf{trees}.\{\!\![e_0 \ldots, e_n]\!\!\}.\varnothing$$

si y sólo si

$$s, h, s', \alpha \models \exists x'_0, \ldots, x'_m, t_0, \ldots, t_n \cdot \mathbf{btree}.e_0.t_0 * \ldots * \mathbf{btree}.e_n.t_n$$

donde $\{x'_0, \ldots, x'_m\} = (FV.e_0 \cup \ldots \cup FV.e_n) \cap Var'$.

Semántica de los comandos

La semántica intermedia se define sobre un reticulado D_a análogo al reticulado D_c de la semántica concreta.

Definición 106. *Dado el conjunto* SH *de* symbolic heaps *y* **abort** *un elemento distinguido, definimos el reticulado* $D_a \doteq (\mathbb{P}.SH \cup \{\mathbf{abort}\}, \subseteq)$, *donde* $\subseteq$ *es la relación de inclusión de conjuntos usual que además cumple que:*

1. *para todos* $d, d' \in D_a$ *si* **abort** $\in d$ *y* **abort** $\in d'$ *entonces* $d = d'$;

2. *para todo* $d \in D_a$, $d \subseteq \{\mathbf{abort}\}$.

Lema 50. *El reticulado* D_a *es completo.*

Para dar semántica al lenguaje de programación respecto a D_a, comenzamos definiendo una relación $\leadsto \in SH \times SH \cup \{\mathbf{abort}\}$ para cada comando atómico.

Definición 107 (Relación $\leadsto$). *Para cada comando atómico* $a \in Comm_A$, *se define la relación* $\leadsto \in SH \times SH \cup \{\mathbf{abort}\}$ *según las siguientes reglas:*

Asignación

$$x := e, \pi \,|\, \sigma \;\leadsto\; \pi' \wedge x = e_{/x \leftarrow x'} \,|\, \sigma'$$

Construcción:

$$x := \mathbf{cons}(\vec{e}), \pi \,|\, \sigma \;\leadsto\; \pi' \,|\, \sigma' * x \mapsto \vec{e}_{/x \leftarrow x'}$$

Consulta:

$$x := y.i, \pi \,|\, \sigma * y \mapsto e_1, e_2, e_3 \;\leadsto\; \pi' \wedge x = e.i_{/x \leftarrow x'} \,|\, \sigma' * (y \mapsto e_1, e_2, e_3)_{/x \leftarrow x'}$$

Mutación:

$$x.i := e', \pi \,|\, \sigma * x \mapsto \vec{e} \;\leadsto\; \pi \,|\, \sigma * x \mapsto \vec{e}[i \mapsto e']$$

Destrucción:

$$\mathbf{dispose}(x), \pi \,|\, \sigma \,*\, x \mapsto \vec{e} \;\rightsquigarrow\; \pi \,|\, \sigma$$

donde $x, y \in Var$; $x' \in Var'$ *es fresca;* $e \in Expr_A$; $\vec{e} \in {Expr_A}^3$; $i \in \{0,1,2\}$; $\pi \,|\, \sigma, \pi' \,|\, \sigma' \in SH$ *tales que* $\pi' \doteq \pi_{/x \leftarrow x'}$ *y* $\sigma' \doteq \sigma_{/x \leftarrow x'}$.

Los comandos de mutación, consulta y destrucción requieren que el *symbolic heap* sobre el que se ejecutan indique explícitamente la existencia del puntero que acceden. Para asegurar esto, definimos una función *rearr* que, dada una variable de interés x, intenta revelar el registro de memoria referenciado por x en cada *symbolic heap* que conforma un estado abstracto. La función *rearr* aplica reglas de reescritura sobre cada *symbolic heap*, sustituyendo sintácticamente cualquier variable cuando se puede deducir que es igual a x, y aplicando *unfold* de los predicados $\mathbf{trees}.C.D$ cuando se puede deducir que x ocurre en C (pero no en D). Más adelante discutimos cómo llevar adelante tales deducciones, denotadas por $\vdash$.

Definición 108 (Función *rearr*)**.** *Dados* $x \in Var$ *y* $d \in D_a$, *la función* $rearr \in Var \to D_a \to D_a$ *es tal que* $rearr.x.d$ *reescribe todo* $\pi \,|\, \sigma \in d$ *hasta que* $x \mapsto \vec{e}$ *ocurre en* $\pi \,|\, \sigma$ *(para algún* $\vec{e} \in {Expr'_A}^3$*), o bien devuelve* $\{\mathbf{abort}\}$ *si no es posible aplicar ninguna regla. El sistema de reescritura* $\Longrightarrow$ *está dado por las siguientes reglas:*

Eq:

$$\frac{\pi \,|\, \sigma \vdash x = y}{\pi \,|\, \sigma \,*\, y \mapsto \vec{e} \;\Longrightarrow\; \pi \,|\, \sigma \,*\, x \mapsto \vec{e}}$$

Unfold:

$$\frac{\pi \,|\, \sigma \vdash x = z \wedge z \neq \mathbf{null} \wedge z \notin D}{\pi \,|\, \sigma \,*\, \mathbf{trees}.(z \oplus C).D \;\Longrightarrow\; \pi \,|\, \sigma \,*\, x \mapsto l', v', r' \,*\, \mathbf{trees}.(l' \oplus r' \oplus C).D}$$

donde $x \in Var$, $y, z \in Var \cup Var'$ *tales que* x *e* y *son distintas;* $l', v', r' \in Var'$, $\vec{e} \in {Expr'_A}^3$, $C, D \in Multiset$, *y* $\pi \,|\, \sigma \in SH$.

Lema 51. *El conjunto de reglas* ***Eq*** *y* ***Unfold*** *es fuertemente normalizante,* i.e. *toda secuencia de aplicación de las reglas termina.*

Para la aplicación de las reglas de reescritura se requiere la validez de ciertas condiciones, que incluyen igualdad y desigualdad de expresiones, pertenencia a un multiconjunto, etc. Al conjunto de estas condiciones las denominamos *Cond*. Su semántica respecto a los modelos concretos es la intuitiva.

Definición 109 (Sintaxis de *Cond*). *El conjunto de predicados Cond está dado por la siguiente gramática:*

$$Cond \ni c ::= e = e \mid e \neq e \mid e \in s \mid e \notin s \mid alloc.e \mid c \wedge c \mid c \vee c$$

donde $e \in Expr'_A$, *y* $s \in Multiset$.

Definición 110 (Semántica de *Cond*). *Dado* $s, h \in States_A$, *la semántica de Cond está dada por la relación* $\vDash\ \in States_A \times Cond$, *definida por las siguientes reglas:*

$s,h \vDash e = e'$	*si y sólo si* $s,h \vDash e = e'$
$s,h \vDash e \neq e'$	*si y sólo si* $s,h \vDash e \neq e'$
$s,h \vDash e \in \varnothing$	*nunca*
$s,h \vDash e \in \{\!\{e_1, \dots, e_n\}\!\}$	*si y sólo si* $s,h \vDash e = e_i$ *para algún* e_i *con* $1 \leqslant i \leqslant n$
$s,h \vDash e \notin \varnothing$	*siempre*
$s,h \vDash e \notin \{\!\{e_1, \dots, e_n\}\!\}$	*si y sólo si* $s,h \vDash e \neq e_i$ *para todo* e_i *con* $1 \leqslant i \leqslant n$
$s,h \vDash alloc.e$	*si y sólo si* $[\![e]\!]_s \in dom.h$
$s,h \vDash c_1 \wedge c_2$	*si y sólo si* $s,h \vDash c_1$ *y* $s,h \vDash c_2$
$s,h \vDash c_1 \vee c_2$	*si y sólo si* $s,h \vDash c_1$ *o* $s,h \vDash c_2$

donde $e, e', e_1, \dots, e_n \in Expr'_A$ *y* $c_1, c_2 \in Cond$.

Presentamos a continuación un pequeño *theorem prover* sintáctico para *Cond*.

Definición 111 (*Theorem prover* $\vdash$). *Dada una condición* $c \in Cond$, *la función* $\vdash\ \in States_A \times Cond \to \{true, false\}$ *se define recursivamente de la siguiente manera:*

$\pi \mid \sigma \vdash e = e_1$	*si* $e \approx e_1$ *o* $\pi \approx \pi' \wedge e = e_2$ *y* $\pi' \mid \sigma \vdash e_1 = e_2$
$\pi \mid \sigma \vdash e \neq n$	*si* $\pi \mid \sigma \vdash e = \mathbf{null}$
	o $\pi \mid \sigma \vdash e = n'$ *y no vale que* $n' \approx n$
	o $\pi \approx \pi' \wedge e = e'$ *y* $\pi' \mid \sigma \vdash e' \neq n$
	o $\pi \approx \pi' \wedge e \neq e'$ *y* $\pi' \mid \sigma \vdash e' = n$
$\pi \mid \sigma \vdash x \neq \mathbf{null}$	*si* $\pi \approx \pi' \wedge e \neq \mathbf{null}$ *y* $\pi' \mid \sigma \vdash x = e$
	o $\pi \approx \pi' \wedge e = \mathbf{null}$ *y* $\pi' \mid \sigma \vdash x \neq e$
	o $\sigma \approx \sigma' * y \mapsto \vec{e}$ *y* $\pi \mid \sigma' \vdash x = y$
$\pi \mid \sigma \vdash x \neq y$	*si* $\pi \approx \pi' \wedge e \neq y$ *y* $\pi' \mid \sigma \vdash x = e$
	o $\pi \approx \pi' \wedge e = y$ *y* $\pi' \mid \sigma \vdash x \neq e$
	o $\sigma \approx \sigma' * \sigma''$ *y* $\pi \mid \sigma' \vdash alloc.x$ *y* $\pi \mid \sigma'' \vdash alloc.y$
$\pi \mid \sigma \vdash e \in \varnothing$	*nunca*
$\pi \mid \sigma \vdash e \in \{\!\{e_1, \dots, e_n\}\!\}$	*si* $\pi \mid \sigma \vdash e = e_i$ *para algún* e_i *con* $1 \leqslant i \leqslant n$
$\pi \mid \sigma \vdash e \notin \varnothing$	*siempre*
$\pi \mid \sigma \vdash e \notin \{\!\{e_1, \dots, e_n\}\!\}$	*si* $\pi \mid \sigma \vdash e \neq e_i$ *para todo* e_i *con* $1 \leqslant i \leqslant n$
$\pi \mid \sigma \vdash alloc.e$	*si* $\sigma \approx \sigma' * x \mapsto \vec{e}$ *y* $\pi \mid \sigma' \vdash e = x$
	o $\sigma \approx \sigma' * \mathbf{trees}.C.D$ *y* $\pi \mid \sigma' \vdash e \in C \wedge e \notin D \wedge e \neq \mathbf{null}$
$\pi \mid \sigma \vdash c_1 \wedge c_2$	*si* $\pi \mid \sigma \vdash c_1$ *y* $\pi \mid \sigma \vdash c_2$
$\pi \mid \sigma \vdash c_1 \vee c_2$	*si* $\pi \mid \sigma \vdash c_1$ *o* $\pi \mid \sigma \vdash c_2$

donde ≈ denota la igualdad sintáctica (módulo conmutatividad), $e, e_1, \ldots, e_n \in Expr'_A$, $\vec{e} \in {Expr'_A}^3$, $n, n' \in \mathbb{N}_0$, $x, y \in Var \cup Var'$, *y* $c_1, c_2 \in Cond$.

El *theorem prover* $\vdash$ permite derivar en muchas circunstancias la validez de una condición en *Cond* a partir de un *symbolic heap*, sin necesidad de recurrir a su semántica respecto a los modelos concretos. Aunque claramente $\vdash$ no es completo, es consistente, *i.e.* las condiciones derivadas desde un *symbolic heap* $\pi \,|\, \sigma$ pueden deducirse semánticamente a partir de los estados caracterizados por $\pi \,|\, \sigma$. Además las reglas de reescritura que definen *rearr* son implicaciones semánticas válidas.

Lema 52 (Consistencia de $\vdash$ y $\implies$). *Sean* $\pi \,|\, \sigma, \pi' \,|\, \sigma' \in SH$ *y* $c \in Cond$*:*

1. *si* $\pi \,|\, \sigma \vdash c$, *entonces para todo* $s, h \in States_A$, $s, h \models \pi \,|\, \sigma$ *implica que* $s, h \models c$.
2. *si* $\pi \,|\, \sigma \implies \pi' \,|\, \sigma'$, *entonces para todo* $s, h \in States_A$, $s, h \models \pi \,|\, \sigma$ *implica que* $s, h \models \pi' \,|\, \sigma'$.

Con el aparato formal desarrollado hasta aquí, es posible extender la relación $\rightsquigarrow$ entre *symbolic heaps* dada en la definición 107 para dar una semántica *ejecutable* de todos comandos sobre el dominio D_a, similiar a la semántica concreta, pero con algunos cambios. Los comandos atómicos que modifican un registro utilizan *rearr* para revelar el puntero que lo referencia. Los comandos compuestos siguen el patrón de la anterior semántica, pero es necesario definir una nueva función $filter_a$ que opere sobre *symbolic heaps* en lugar de estados concretos. Retomemos la sintáxis de las expresiones *booleanas Bool*, que sirven como *guardas* en los comandos de alternativa y ciclo. Una guarda b es de la forma $eqs_1 \vee \ldots \vee eqs_n$ donde cada término eqs_i es una conjunción de (des)igualdades. La función $filter_a$ combina cada *symbolic heap* con los términos eqs_i de manera que el resultado es *consistente*: no existen contradicciones, y no se utilizan valores atómicos como punteros. Esto no es estrictamente necesario, dado que un *symbolic heap* inconsistente no puede ser satisfecho por ningún estado concreto, pero permite que la semántica sea más ajustada. Definimos la consistencia de un *symbolic heap* en términos de verificaciones sintácticas utilizando $\vdash$.

Definición 112. *Dado* $\pi \,|\, \sigma \in SH$, *decimos que* $\pi \,|\, \sigma$ *es* es consistente *si:*

1. *para todos* $e, e' \in Expr$ *que* ocurren *en* $\pi \,|\, \sigma$, *no sucede que* $\pi \,|\, \sigma \vdash e = e'$ *y al mismo tiempo* $\pi \,|\, \sigma \vdash e \neq e'$;
2. *para ningún* $n \in \mathbb{N}_0$ *vale* $\pi \,|\, \sigma \vdash alloc.n$;
3. *para ningún* $n \in \mathbb{N}_0$ *vale* $\pi \,|\, \sigma \vdash n \in C$, *donde* $C \in Multiset$ *ocurre en* $\pi \,|\, \sigma$.

Definición 113 (Función $filter_a$). *Dado $b \in Bool_A$, donde la expresión b es de la forma $eqs_1 \vee \ldots \vee eqs_n$, la función $filter_a \in Bool_A \to D_a \to D_a$ se define como:*

$$filter_c.b.d \doteq \begin{cases} \{\mathbf{abort}\} & si\ \mathbf{abort} \in d \\ \bigcup_{1 \leqslant i \leqslant n} \{\pi \wedge eqs_i \,|\, \sigma \mid existe\ \pi \,|\, \sigma \in d\ y \\ \qquad\qquad \pi \wedge eqs_i \,|\, \sigma\ es\ consistente\} & en\ c.c. \end{cases}$$

Definición 114 (Semántica intermedia de $SL_{\mathscr{A}}$). *La semántica intermedia de comandos $[\![\ ]\!]_i \in Comm \to D_a \to D_a$ se define como:*

$$\begin{aligned} [\![a]\!]_i &\doteq a^{\upharpoonright} \\ [\![a_x]\!]_i &\doteq a_x^{\upharpoonright} \bullet rearr.x \\ [\![c_1 ; c_2]\!]_i &\doteq [\![c_1]\!]_i \bullet [\![c_2]\!]_i \\ [\![\mathbf{if}\ b\ \mathbf{then}\ c_1 \mathbf{else}\ c_2\ \mathbf{fi}]\!]_i &\doteq ([\![c_1]\!]_i \bullet filter_a.b) \cup ([\![c_2]\!]_i \bullet filter_a.\neg b) \\ [\![\mathbf{while}\ b\ \mathbf{do}\ c_1 \mathbf{od}]\!]_i &\doteq \lambda d \cdot filter_a.\neg b.(\mu d' \cdot d \cup ([\![c_1]\!]_i \bullet filter_a.b.).d') \end{aligned}$$

donde $x \in Var$, a es comando de asignación o construcción, a_x es el comando de mutación, consulta o destrucción sobre la variable x, $c_1, c_2 \in Comm_A$, y $b \in Bool_A$.

Dada la incompletitud de $\vdash$, en algunas circunstancias *rearr* no es capaz de revelar la existencia del registro referenciado por un puntero particular. Como consecuencia, ciertas ejecuciones de la semántica intermedia terminan en {**abort**} aún cuando no existe una violación de memoria de acuerdo con la semántica concreta. Para establecer esta propiedad precisamente, definimos la relación que existe entre los estados abstractos y los concretos, a través de una función de concretización. A partir de esta función se puede establecer el tipo de relación que existe entre la semántica intermedia y la semántica concreta.

Definición 115 (Concretización). *Dado $d \in D_a$, la función de concretización $\gamma \in D_a \to D_c$ se define como*

$$\gamma.d \doteq \begin{cases} \{\mathbf{abort}\} & si\ \mathbf{abort} \in d \\ \{s, h \in States_A \mid existe\ \pi \,|\, \sigma \in d\ tal\ que\ s, h \vDash \pi \,|\, \sigma\} & en\ c.c. \end{cases}$$

Teorema 7. *Dado $c \in Comm_A$, la semántica intermedia de c es una sobre-aproximación consistente de la semántica concreta de c:*

$$[\![c]\!]_c \bullet \gamma \subseteq \gamma \bullet [\![c]\!]_i$$

6.3. Semántica abstracta: el análisis

La semántica intermedia es ejecutable pero no ofrece ningún mecanismo para facilitar la computación de invariantes para los ciclos, y de esa manera asegurar la terminación. En general, un ciclo que utiliza un puntero x para recorrer una estructura arbórea genera un estado abstracto con un número arbitrario de *symbolic heaps*, cada uno de los cuales contiene una cantidad cada vez mayor de términos de la forma $x' \mapsto l', v', r'$.

Ejemplo 58. *A continuación se presenta el código fuente del algoritmo que computa en r el valor mínimo de un árbol binario de búsqueda (BST):*

```
p := x;
while p ≠ null do
    r := p.1;
    p := p.0;
od
```

Partiendo de la precondición $\{\mathbf{true} \,|\, \mathbf{trees}.\{\![x]\!\}.\emptyset\}$, *la semántica intermedia utiliza la variable p para explorar las ramas izquierdas de los subárboles que encuentra al hacer el* unfolding *del predicado* **trees**. *Así, iteración tras iteración genera un estado de la forma*

$$
\begin{aligned}
&\{p = x \,|\, \mathbf{trees}.\{\![x]\!\}.\emptyset, \\
&\quad r = v'_1 \,|\, x \mapsto p, v'_1, r'_1 \,*\, \mathbf{trees}.\{\![p, r'_1]\!\}.\emptyset, \\
&\quad r = v'_2 \,|\, x \mapsto l'_1, v'_1, r'_1 \,*\, l'_1 \mapsto p, v'_2, r'_2 \,*\, \mathbf{trees}.\{\![p, r'_1, r'_2]\!\}.\emptyset, \\
&\quad r = v'_3 \,|\, x \mapsto l'_1, v'_1, r'_1 \,*\, l'_1 \mapsto l'_2, v'_2, r'_2 \,*\, l'_2 \mapsto p, v'_3, r'_3 \,*\, \mathbf{trees}.\{\![p, r'_1, r'_2, r'_3]\!\}.\emptyset, \ldots
\end{aligned}
$$

causando la divergencia de la ejecución.

Para manejar situaciones de este tipo, definimos una función de abstracción $abs \in D_a \to D_a$, cuyo propósito es simplificar los *symbolic heaps* que conforman el estado abstracto, reemplazando información precisa sobre la forma de la memoria dinámica, por otra más abstracta pero suficientemente útil. El uso de esta función pretende facilitar la convergencia de la computación de punto fijo que representa la semántica de un ciclo. Definimos una semántica abstracta $[\![\ ]\!]_a$, análoga a $[\![\ ]\!]_i$, que aplica la función *abs* a la entrada de un ciclo y luego de cada iteración.

Definición 116 (Semántica abstracta). *La semántica abstracta de comandos* $[\![\ ]\!]_a \in Comm_A \to D_a \to D_a$ *se define como:*

$$
\begin{aligned}
[\![a]\!]_a &\doteq a^{\upharpoonright} \\
[\![a_x]\!]_a &\doteq a_x^{\upharpoonright} \bullet rearr.x \\
[\![c_1; c_2]\!]_a &\doteq [\![c_1]\!]_a \bullet [\![c_2]\!]_a \\
[\![\textbf{if } b \textbf{ then } c_1 \textbf{else } c_2 \textbf{ fi}]\!]_a &\doteq ([\![c_1]\!]_a \bullet filter_a.b) \cup ([\![c_2]\!]_a \bullet filter_a.\neg b) \\
[\![\textbf{while } b \textbf{ do } c_1 \textbf{ od}]\!]_a &\doteq (\lambda d \cdot filter_a.\neg b.(\mu d' \cdot abs.(d \cup ([\![c_1]\!]_a \bullet filter_a.b).d'))) \bullet abs
\end{aligned}
$$

donde $x \in Var$, *a es comando de asignación o construcción,* a_x *es el comando de mutación, consulta o destrucción sobre la variable* x, $c_1, c_2 \in Comm_A$, *y* $b \in Bool_A$.

De esta forma, aunque el dominio de la semántica abstracta sigue siendo el mismo que el de la semántica intermedia, la semántica de un ciclo se calcula sobre un subconjunto de los estados que, como mostramos más adelante, asegura la convergencia del cómputo del punto fijo.

La función *abs* consiste en la eliminación de *symbolic heaps* inconsistentes, la aplicación de un conjunto de reglas de reescritura, y la igualación de los *symbolic heaps* que sólo difieren en el nombre de las variables cuantificadas. El sistema de reescritura que define se organiza en tres etapas, cada una de las cuales incluye la aplicación exhaustiva de un subconjunto de tales reglas.

Definición 117 (Función *abs* (primera etapa)). *Dado* $d \in D_a$, *la función* $abs \in D_a \to D_a$ *aplica a cada* symbolic heap *en* d, *reglas de reescritura hasta que resulta imposible continuar aplicando alguna de ellas. La reglas se organizan en tres conjuntos, cada una de los cuales es agotado sucesivamente. Las reglas de la primera etapa se listan a continuación. Utilizamos* $T.C.D$ *para denotar o bien un predicado* $\textbf{trees}.C.D$, *o bien* $x \mapsto l, v, r$ *con* $C = \{\![x]\!\}$ *y* $D = \{\![l, r]\!\}$.

AbsTrees1:

$$
\frac{\sigma \approx \sigma' * \textbf{trees}.C.(e \oplus D) * \textbf{trees}.(e' \oplus E).\varnothing \qquad \pi \,|\, \sigma \vdash e = e'}{\pi \,|\, \sigma \implies \pi \,|\, \sigma' * \textbf{trees}.C.D * \textbf{trees}.E.\varnothing}
$$

AbsTrees2:

$$
\frac{\begin{array}{c}\sigma \approx \sigma' * \textbf{trees}.C.(e \oplus D) * T.(e' \oplus E).F \\ \pi \,|\, \sigma \vdash e = e' \qquad \pi \,|\, \sigma \vdash alloc.e'' \vee e'' = \textbf{null} \ \textit{para todo } e'' \textit{ en } F\end{array}}{\pi \,|\, \sigma \implies \pi \,|\, \sigma * \textbf{trees}.(C \uplus E).(D \uplus F)}
$$

AbsNull1:

$$
\frac{\pi \,|\, \sigma \vdash e = \textbf{null}}{\pi \,|\, \sigma * \textbf{trees}.(e \oplus C).D \implies \pi \,|\, \sigma * \textbf{trees}.C.D}
$$

AbsNull2:

$$\frac{\pi \,|\, \sigma \vdash e = \textbf{null}}{\pi \,|\, \sigma * \textbf{trees}.C.(e \oplus D) \implies \pi \,|\, \sigma * \textbf{trees}.C.D}$$

AbsPartial:

$$\frac{\pi \,|\, \sigma \vdash e = e'}{\pi \,|\, \sigma * \textbf{trees}.(e \oplus C).(e' \oplus D) \implies \pi \,|\, \sigma * \textbf{trees}.C.D}$$

AbsArrowLeft1:

$$\frac{\sigma \approx \sigma' * x \mapsto l,v,r * \textbf{trees}.(e \oplus C).\varnothing \qquad \pi \,|\, \sigma \vdash l = e \wedge x \neq r}{\pi \,|\, \sigma \implies \pi \,|\, \sigma' * \textbf{trees}.\{\!\{x\}\!\}.\{\!\{r\}\!\} * \textbf{trees}.C.\varnothing}$$

AbsArrowLeft2:

$$\frac{\sigma \approx \sigma' * x \mapsto l,v,r * T.(e \oplus C).D \qquad \pi \,|\, \sigma \vdash l = e \wedge x \notin (r \oplus D) \wedge e \notin D}{\pi \,|\, \sigma \implies \pi \,|\, \sigma' * \textbf{trees}.(x \oplus C).(r \oplus D)}$$

AbsArrowRight1:

$$\frac{\sigma \approx \sigma' * x \mapsto l,v,r * \textbf{trees}.(e \oplus C).\varnothing \qquad \pi \,|\, \sigma \vdash r = e \wedge x \neq l}{\pi \,|\, \sigma \implies \pi \,|\, \sigma' * \textbf{trees}.\{\!\{x\}\!\}.\{\!\{l\}\!\} * \textbf{trees}.C.\varnothing}$$

AbsArrowRight2:

$$\frac{\sigma \approx \sigma' * x \mapsto l,v,r * T.(e \oplus C).D \qquad \pi \,|\, \sigma \vdash r = e \wedge x \notin (l \oplus D) \wedge e \notin D}{\pi \,|\, \sigma \implies \pi \,|\, \sigma' * \textbf{trees}.(x \oplus C).(l \oplus D)}$$

donde $e, e', e'', l, v, r \in Expr'_A$, $x \in Var \cup Var'$, $C, D, E, F \in Multiset$, $\pi \in \Pi$ *y* $\sigma, \sigma' \in \Sigma$.

Las reglas **AbsTrees1** y **AbsTrees2** combinan estructuras arbóreas *olvidando* los enlaces entre ellas. En ambos casos se asegura que el enlace a abstraer referencie efectivamente a un registro, ya sea porque no hay punteros de salida (**AbsTrees1**), o ya sea porque los punteros de salida referencian otros registros definidos en σ' (**AbsTrees2**). Así las condiciones de aplicación previenen la formación de ciclos dentro de la estructura resultante. Las reglas **AbsNull** y **AbsPartial** remueven punteros de entrada y de salida que no proveen información relevante sobre la memoria dinámica. Las reglas **AbsArrow** abstraen predicados $\mapsto$ en predicados **trees**, *olvidando* el número de registros que forman la estructura arbórea. Aquí tambien las condiciones de aplicación inhiben la conformación de ciclos.

Ejemplo 59. *Retomemos el ejemplo 58. Si aplicamos las reglas **AbsArrowLeft1**, **AbsTrees2** y **AbsPartial**, en la segunda iteración del cálculo del punto fijo obtenemos el invariante*

$$\{p = x \,|\, \textbf{trees}.\{\!\{x\}\!\}.\varnothing,\;\; r = v_2' \,|\, \textbf{trees}.\{\!\{x\}\!\}.\{\!\{p\}\!\} * \textbf{trees}.\{\!\{p\}\!\}.\varnothing\}$$

Relevancia de las variables en la abstracción

Comencemos por definir cierta terminología. Utilizamos nuevamente $T.C.D$ para denotar o bien un predicado **trees**.$C.D$, o bien $x \mapsto l, v, r$ con $C = \{\!\![x]\!\!\}$ y $D = \{\!\![l, r]\!\!\}$. Decimos que dos términos $T_1.C.D$ y $T_2.E.F$ de un *symbolic heap* forman un *enlace de cadena* si existe una expresión e en D tal que $e \in E$. Una secuencia 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 todos los i con $1 \leqslant i < n$.

La aplicación de las regla de abstracción significa la pérdida de dos tipos de información sobre el *heap*: primero, los valores específicos del registro referenciado por cierta variable; y segundo, la información sobre la expresión que define un enlace de cadena entre dos términos. Por ejemplo, ambos casos ocurren en la regla **AbsArrowLeft1** para x y e respectivamente. Cuando una variable es abstraída como resultado de la aplicación de una regla, la perdida de información puede ocasionar la imposibilidad de analizar un programa que intente accederla, o que dependa de la información que provee de forma más indirecta. Esto se debe a que o bien no es posible revelar el registro referenciado por la variable, dado que las condiciones para la aplicación de *rearr* no están ya garantizadas, o bien porque la información de que la variable conformaba un enlace entre las estructuras arbóreas se pierde.

En general, los *shape analysis* basados en *Separation Logic* aplican reglas de abstracción siempre que la variable que define el enlace de cadena esté cuantificada. Pero un enfoque de este estilo no es adecuado en nuestro caso. Por un lado, puede ocasionar demasiada pérdida de información. Por otro lado puede dar lugar a formulas innecesariamente precisas y por lo tanto, invariantes más grandes. Llevar cuenta de una cadena de dos, tres o más enlaces no es un problema en el caso de las estructuras lineales. Pero en el tratamiento de estructuras con múltiples enlaces, dada las numerosas combinaciones posibles, el número cadenas crece exponencialmente. Aunque desde el punto de vista de la corrección esto no es un problema, mantener estados abstractos reducidos agiliza el análisis y permite una mejor comprensión de los invariantes y las postcondiciónes obtenidas.

La aplicación de las reglas de abstracción en *abs* es relativa al *nivel de relevancia* asignado a cada variable. El nivel de relevancia de una variable en cierto punto de ejecución depende del tipo de comandos que la utilizan y que se ejecutan luego de ese punto.

Definimos un análisis muy simple, *relev*, que se ejecuta sobre un programa y devuelve una función de tipo $Var \rightarrow \mathbb{N}_0$ que codifica los accesos y/o modificaciones que sufre cada variable, y por lo tanto, la información que un estado abstracto debe preservar sobre ella. Ya que el uso de una variable puede cambiar a lo largo del programa el análisis se ejecuta de atras para adelante. Por ejemplo, la información que se pueda tener sobre una variable x es irrelevante luego de un comando como $x := e$.

El análisis *relev* se define entonces sobre los programas considerados como secuencias de comandos separados por ";", partiendo de una función que asigna 0 a todas las variables de programa, a excepción de las variables que ocurren originalmente en la precondición del programa bajo análisis, a las que le asigna el valor 1.

Definición 118 (Función *relev*)**.**

$$
\begin{array}{rcll}
relev & \in & (Var \to \mathbb{N}_0) \to [Comm_A] \to (Var \to \mathbb{N}_0) & \\
relev.f.(x := e; cs) & = & (relev.f.cs[x \mapsto 0])\lfloor y \mapsto 2 \rfloor & para\ y\ en\ e \\
relev.f.(x := \mathbf{cons}(\vec{e}); cs) & = & (relev.f.cs[x \mapsto 0])\lfloor y \mapsto 2 \rfloor & para\ y\ en\ \vec{e} \\
relev.f.(x.i := e; cs) & = & (relev.f.cs.[x \mapsto 4])\lfloor y \mapsto 2 \rfloor & para\ y\ en\ e \\
relev.f.(x := y.i; cs) & = & (relev.f.cs[x \mapsto 0])\lfloor y \mapsto 3 \rfloor & \\
relev.f.(\mathbf{dispose}(x); cs) & = & relev.f.cs[x \mapsto 3] & \\
relev.f.(\mathbf{if}\ b\ \mathbf{then}\ cs_1 \mathbf{else}\ cs_2\ \mathbf{fi}; cs) & = & (relev.f.(cs_1 + cs)\ max\ relev.f.(cs_2 + cs))\lfloor y \mapsto 2 \rfloor & para\ y\ en\ b \\
relev.f.(\mathbf{while}\ b\ \mathbf{then}\ cs_1\ \mathbf{od}; cs) & = & (relev.f.(cs_1 + cs)\ max\ relev.f.cs)\lfloor y \mapsto 2 \rfloor & para\ y\ en\ b \\
relev.f.(\epsilon) & = & f &
\end{array}
$$

donde $x, y \in Var$, $e \in Expr'_A$, $\vec{e} \in {Expr'_A}^3$, $i \in \{0, 1, 2\}$, *cs es un secuencia de comando,* $b \in Bool_A$, ϵ *denota la secuencia vacía,* $+$ *denota la concatenación de secuencias, y la actualización de función* $\lfloor\ \rfloor$ *y la función max se definen como:*

$$
\lfloor\ \rfloor \in (Var \to \mathbb{N}_0) \to (Var \to \mathbb{N}_0)
$$

$$
(f\lfloor y \mapsto n \rfloor).x \doteq \begin{cases} x & si\ x \neq y\ o\ f.x > n \\ n & en\ c.c. \end{cases}
$$

$$
max \in (Var \to \mathbb{N}_0) \to (Var \to \mathbb{N}_0) \to (Var \to \mathbb{N}_0)
$$

$$
(max.f.g).x \doteq \begin{cases} f.x & si\ f.x > g.x \\ g.x & en\ c.c. \end{cases}
$$

donde $n \in \mathbb{N}_0$.

El *nivel de relevancia* de una variable x en $\pi \mid \sigma$ está dado por el mayor valor de acuerdo con f dentro de la clase de equivalencia inducida por las igualdades que ocurren en π.

Definición 119 (Nivel de relevancia)**.** *Dados* $\pi \mid \sigma \in SH$, $c \in Comm_A$, $x \in Var$, $V \subseteq Var$ *un conjunto de variables de referencia,* $f \in Var \to \mathbb{N}_0$ *tal que* $f.y = 0$ *para toda* $y \in Var - V$ *y* $f.y = 1$ *para toda* $y \in V$, *el* nivel de relevancia *de* x *en* c *para* $\pi \mid \sigma$ *se define como*

$$
max.\{v \in \mathbb{N}_0 \mid v = (relev.f.c).y\ para\ toda\ y \in Class.x.(\pi \mid \sigma)\}
$$

donde

$$Class.x.(\pi\,|\,\sigma) \doteq \{y \in Var \ \mid \ \pi\,|\,\sigma \vdash x = y\}$$

En nuestro análisis, antes de aplicar las reglas de abstracción sobre cada uno de los *symbolic heaps* que conforman un estado abstracto, se calcula el nivel de relevancia de cada variable respecto al programa que resta ejecutar. En todos los casos, el conjunto de variables de referencia para este cálculo es el conjunto de variables libres de la precondición original del programa a analizar.

En cada regla, las variables de programa que ocurren en los términos que se abstraen, tienen asociado un valor límite que condiciona su aplicación. El valor límite depende del rol de la variable, por ejemplo, si define un *enlace de cadena* u ocurre a la izquierda en un término $\mapsto$. La regla se aplica sólo si todas las variables intervinientes tienen un nivel de relevancia calculado menor a ese límite. El límite de cada variable en una regla está parametrizado, permitiendonos ajustar el nivel de abstracción en diferentes ejecuciones. Según nuestros experimentos imponer como límite un nivel de relevancia 2 para cualquier variable resulta una elección segura, que permite evitar una rápida convergencia a {**abort**}. Algunos programas admiten un límite mayor, lo que da lugar a invariantes más compactos, y menores tiempos de ejecución.

Variables cuantificadas y terminación

La no terminación de la semántica abstracta está relacionada con la posibilidad de que la ejecución de un ciclo haga crecer ilimitadamente el estado abstracto incorporando un número cada vez mayor de *symbolic heaps* "diferentes". Ya que el número de variables de programa está siempre acotado, la generación de una cantidad ilimitada de *symbolic heaps* sólo puede darse por dos vías: la repetición de términos, o la inclusión de nuevos términos que involucran variables cuantificadas y/o valóres atómicos.

En el caso de los términos *pure*, su múltiplicación puede acotarse por la propiedad de idempotencia de $\wedge$ y la posibilidad de descartar términos que involucran variables cuantificadas y valores atómicos, sin demasiado impacto en la consecución del análisis.

En el caso de los términos espaciales, la función *rearr* es la principal responsable de generar nuevos términos $\mapsto$ que forman cadenas de términos arbitrariamente largas, que en un principio inician en un término que involucra una variable de programa pero que, por las posteriores modificaciones a las variables, pueden quedar "huerfanas". Se vuelve necesario entonces, por un lado, limitar la longitud de las cadenas que comienzan con una variable de programa, y por otro, descartar las cadenas que comienzan con variables cuantificadas. La limitación a los términos espaciales repetidos y espúreos se lleva adelante a través de la verificación de consistencia que incluye *abs*.

Como mencionamos anteriormente, la reescritura producida por *abs* se organiza en tres etapas, cada una de las cuales consiste en la aplicación exhaustiva de un conjunto de reglas. La primera etapa es presentada en la definición 117 (pág. 159). La aplicación de sus reglas eliminan una cantidad significativa de variables quantificadas cuando es posible asegurar la ausencia de ciclos dentro de las estructuras, pero esto no es suficiente. La segunda y tercer etapas de *abs* constan de un número de reglas que aseguran la finitud de su imagen, y por lo tanto, la convergencia del cálculo de punto fijo.

El objetivo principal de las reglas de la segunda etapa es eliminar las variables cuantificadas que forman enlaces de cadena en situaciones en que no es posible asegurar una estructura arbórea, y por lo tanto, en las que las reglas de la primera etapa no aplican.

Definición 120 (Función *abs* (segunda etapa))**.** *Las reglas de la segunda etapa son:*

EqElimination1:

$$\pi \wedge x = n \,|\, \sigma \implies (\pi \,|\, \sigma)_{/x \leftarrow n}$$

EqElimination2:

$$\pi \wedge n = n \,|\, \sigma \implies \pi \,|\, \sigma$$

EqElimination3:

$$\pi \wedge x' = e \,|\, \sigma \implies (\pi \,|\, \sigma)_{/x' \leftarrow e}$$

NeqElimination1:

$$\pi \wedge n \neq e \,|\, \sigma \implies \pi \,|\, \sigma$$

NeqElimination2:

$$\pi \wedge x' \neq e \,|\, \sigma \implies \pi \,|\, \sigma$$

Garbage1:

$$\frac{x' \textit{ no ocurre en } \pi \,|\, \sigma}{\pi \,|\, \sigma * \mathbf{trees}.(x' \oplus C).\varnothing \implies \pi \,|\, \sigma * \mathbf{trees}.C.\varnothing * \mathbf{true}}$$

Garbage2:

$$\frac{x' \textit{ no ocurre en } \pi \,|\, \sigma}{\pi \,|\, \sigma * x' \mapsto l, v, r \implies \pi \,|\, \sigma * \mathbf{junk}}$$

Garbage3:

$$\frac{x' \textit{ no ocurre en } \pi \,|\, \sigma}{\pi \,|\, \sigma * \mathbf{trees}.(x' \oplus C).D \implies \pi \,|\, \sigma * \mathbf{true}}$$

Garbage3:

$$\frac{x' \text{ no ocurre en } \pi \mid \sigma}{\pi \mid \sigma * \mathbf{trees}.C.(x' \oplus D) \implies \pi \mid \sigma * \mathbf{true}}$$

TreesIdempotence:

$$\pi \mid \sigma * \mathbf{trees}.C.D * \mathbf{trees}.C.D \implies \pi \mid \sigma$$

ChainBreak1:

$$\pi \mid \sigma * T.C.(x' \oplus D) * x' \mapsto l, v, r \implies \pi \mid \sigma * T.C.(x' \oplus D) * \mathbf{junk}$$

ChainBreak2:

$$\pi \mid \sigma * T.C.(x' \oplus D) * \mathbf{trees}.(x' \oplus E).\varnothing \implies \\ \pi \mid \sigma * T.C.(x' \oplus D) * \mathbf{trees}.E.\varnothing * \mathbf{true}$$

ChainBreak3:

$$\pi \mid \sigma * T.C.(x' \oplus D) * \mathbf{trees}.(x' \oplus E).F \implies \pi \mid \sigma * T.C.(x' \oplus D) * \mathbf{true}$$

donde $x' \in Var'$, $e, l, v, r \in Expr'$, $n \in \mathbb{N}_0$, $C, D, E, F \in Multiset$, *y* $\pi \mid \sigma \in SH$.

Las reglas **EqElimination** y **NeqElimination** eliminan las relaciones que involucran variables cuantificadas y valores atómicos. Las reglas **Garbage1**, **Garbage2** y **Garbage3** eliminan las cadenas que no comienzan con variables de programa. Las reglas **Garbage4** y **TreesIdempotence** limitan las cadenas que comienzan con un término **trees**.$C.D$, ya sea, eliminando los terminos que tienen variables cuantificadas como punteros de salida, o eliminando múltiples ocurrencias de un mismo término, dado que no dan lugar a una inconsistencia sólo si $C = D$ y por lo tanto **trees**.$C.D$ es equivalente a **emp**. Las reglas **ChainBreak** reducen la longitud de las cadenas, removiendo los enlaces formados por variables cuantificadas. Son análogas a las reglas **AbsTrees** y **AbsArrow** de la primera etapa, y aplican cuando aquellas no han podido ser utilizadas por la invalidez de sus condiciones de aplicación.

El propósito de la tercera etapa es acotar el tamaño de los *symbolic heaps* reduciendo el número de términos repetidos, aplicando diversas propiedades de elemento neutro e idempotencia. Estas reglas reúnen la *basura* en únicos predicados **junk** y/o **true** y delimitan el número de términos *pure* repetidos, complementando las reglas **EqElimination** y **NeqElimination**.

Definición 121 (Función *abs* (tercera etapa))**.** *Las reglas de la tercera etapa de son:*

AndZero:

$$\pi \wedge \mathbf{true} \mid \sigma \implies \pi \mid \sigma$$

SpatialZero:

$$\pi \,|\, \sigma * \mathbf{emp} \implies \pi \,|\, \sigma$$

AndIdempotence:

$$\pi \wedge r \wedge r \,|\, \sigma \implies \pi \wedge r \,|\, \sigma$$

SpatialIdempotence1:

$$\pi \,|\, \sigma * \mathbf{true} * \mathbf{true} \implies \pi \,|\, \sigma * \mathbf{true}$$

SpatialIdempotence2:

$$\pi \,|\, \sigma * \mathbf{junk} * \mathbf{junk} \implies \pi \,|\, \sigma * \mathbf{junk}$$

SpatialIdempotence3:

$$\pi \,|\, \sigma * \mathbf{junk} * \mathbf{true} \implies \pi \,|\, \sigma * \mathbf{junk}$$

donde $r \in Rel'$, *y* $\pi \,|\, \sigma \in SH$.

En la eliminación de términos espaciales repetidos juegan un rol importante la regla **TreesIdempotence** de la segunta etapa, y la verificación de consistencia inicial, que elimina la posibilidad de que existan múltiples términos $x \mapsto _$ que compartan la variable x. Esta verificación además inhibe la presencia de términos $n \mapsto _$ con $n \in \mathbb{N}_0$. Finalmente, considerar como equivalentes cualesquiera dos *symbolic heaps* que sólo difieren en el nombre de sus variables cuantificadas, permite acotar el número de *symbolic heaps* que pueden ser el resultado de aplicar *abs*.

Lema 53 (Terminación de *abs*)**.**

1. *img.abs es finita.*
2. *El conjunto de reglas de reescritura que conforman* abs *es fuertemente normalizante.*

Las reglas de abstracción que conforman *abs* no son confluentes, *i.e.* la forma normal final obtenida luego de una secuencia de reescritruas no es única. El orden de aplicación de las reglas de la primera etapa es relevante, dado que reglas como **AbsArrow** pierden información que podría ser necesaria para derivar condiciones de aplicación de otras reglas. Sin embargo, en los casos prácticos, no observamos diferencias significativas al cambiar el orden de aplicación.

Como ocurre con las reglas de reescritura de *rearr*, las reglas de abstracción son consecuencias elementales de la definición de **trees** y de propiedades básicas de la *Separation Logic*. El resultado 2 del lema 52 (pág. 156) puede extenderse a todas las reglas de reescritura presentadas en esta sección.

Lema 54 (Consistencia de $\Longrightarrow$). *Sean* $\pi \,|\, \sigma, \pi' \,|\, \sigma' \in SH$, *si* $\pi \,|\, \sigma \Longrightarrow \pi' \,|\, \sigma'$, *entonces para todo* $s, h \in States_A$, $s, h \vDash \pi \,|\, \sigma$ *implica que* $s, h \vDash \pi' \,|\, \sigma'$.

En virtud de los lemas anteriores podemos garantizar que el análisis dado por la ejecución de la semántica abstracta es consistente y siempre termina.

Teorema 8. *Dado* $c \in Comm$, *la semántica abstracta de* c *es una sobre-aproximación consistente de la semántica concreta de* c*:*

$$[\![c]\!]_c \bullet \gamma \subseteq \gamma \bullet [\![c]\!]_a$$

Más aún, el algoritmo definido por $[\![\ ]\!]_a$ *termina.*

6.4. Analizando grafos generales

Es posible utilizar ideas similares a las subyacentes en la definición del predicado **trees** para describir estructuras de grafos generales que incluyen ciclos y *sharing* irrestricto. Definimos entonces un nuevo predicado **graph**.$C.D$ que especifica un grafo general, y determina con C los punteros de entrada al grafo, y con D los punteros que *pueden* referenciar nodos *compartidos* dentro de la estructura. Así, los parámetros del predicado **graph** reflejan las ideas normalmente utilizadas para razonar sobre los algoritmos de grafos: mientras que C representa los puntos de entrada de los caminos a recorrer, D representa los nodos *ya visitados*. Para un grafo con nodos binarios, la semántica formal de **graph** se define a continuación.

Definición 122 (Semántica de **graph**). *Dado* $s, h \in States_A$, *definimos que* $s, h \vDash$ **graph**.$C.D$ *si y sólo si:*

1. $s, h \vDash$ **emp**, *si* $C \approx \emptyset$*; o*
2. $s, h \vDash e =$ **null** $|$ **graph**.$C'.D$, *si existen* e *y* C' *tales que* $C \approx e \oplus C'$*; o*
3. $s, h \vDash e =$ **null** $|$ **graph**.$C.D'$, *si existen* e *y* D' *tales que* $D \approx e \oplus D'$*; o*
4. $s, h \vDash e = e'$ $|$ **graph**.$C'.D$, *si existen* e *y* C' *tales que* $C \approx e \oplus C'$ *y* e' *y* D *tales que* $D \approx e' \oplus D'$*; o*
5. $s, h \vDash e \neq e' \wedge e \neq$ **null** $|$ $e \mapsto l', v', r' \,*$ **graph**.$(l' \oplus r' \oplus C').(x \oplus D)$, *si existen* e *y* C' *tales que* $C \approx e \oplus C$ *y para tods* e' *y* D' *tales que* $D = e' \oplus D'$.

donde $x \in Var \cup Var'$, $l', v', r' \in Var'$, $e, e' \in Rel'_A$ *y* $C, C', D, D' \in Multiset$.

El caso 1 representa el grafo vacío. Notar que aquí ya no es necesario que $D = \emptyset$, y por lo tanto los punteros de salida en D pueden o no ser alcanzados. Así, representan *potenciales* puntos de *sharing*. Los casos 2 y 3 son análogos a aquellos de la definición

de **trees**, y especifican que los punteros *nil* no influyen en la estructura definida. El caso 4 caracteriza la parcialidad y el potencial *sharing*. El puntero que es tanto puntero de entrada como de salida no define ningún nuevo registro. Pero a diferencia de **trees**, tal puntero no es eliminado del conjunto de punteros de salida. En el caso 5, se define el registro apuntado y se agregan los punteros a los subgrafos como nuevos punteros de entrada. Además, el puntero recién definido se agrega a los potenciales puntos de *sharing*. Asi se abre la posibilidad de que existan múltiples punteros que "compartan" el registro referenciado por x, dando lugar a la existencia de ciclos.

Adaptar el análisis para soportar grafos en lugar de árboles, requiere modificar algunas reglas de reescritura, mientras que otras sufren cambios elementales (el reemplazo de **trees** por **graph**) y aún otras permanecen intactas.

La regla **Unfold** de *rearr* debe dar cuenta de las diferencias en la definición del predicado **graph** respecto a **trees**.

Definición 123 (Función *rearr* para **graph**). *Dados $x \in Var$ y $d \in D_a$, la función $rearr \in Var \to D_a \to D_a$ se define como en la definición 108 (pág. 154) reemplazando la regla de reescritura* ***Unfold*** *por la siguiente:*

$$\frac{\pi \,|\, \sigma \vdash x = y \wedge y \neq \mathbf{null} \wedge y \notin D}{\pi \,|\, \sigma * \mathbf{graph}.(y \oplus C).D \implies \pi \,|\, \sigma * x \mapsto l', v', r' * \mathbf{graph}.(l' \oplus r' \oplus C).(x \oplus D)}$$

donde $x \in Var$, $y \in Var \cup Var'$ tales que x e y son distintas, $l', v', r' \in Var'$, $C, D \in Multiset$, y $\pi \,|\, \sigma \in SH$.

Dado el uso que se hace del conjunto de punteros de salida por la posible existencia de ciclos, una condición como $y \notin D$ puede resultar difícil de derivar en una gran cantidad de casos. Para evitar la convergencia a $\{\mathbf{abort}\}$ cuando $rearr.x.d$ falla para un $\pi \,|\, \sigma \in d$, se lo reemplaza por el conjunto de *symbolic heaps* que resulta de agregar la hipótesis $y \in D \vee y \notin D$. Más precisamente, $\pi \,|\, \sigma$ es reemplazado por el conjunto dado por $filter_a.b.\pi \,|\, \sigma$ donde

$$b \doteq (y \neq e_1 \wedge \ldots \wedge y \neq e_n) \vee y = e_1 \vee \ldots \vee y = e_n$$

para $D \approx \{\!\![e_1, \ldots, e_n]\!\!\}$. Es trivial comprobar que este reemplazo da lugar a un estado abstracto semánticamente equivalente. De esta manera, para el *symbolic heap* $\pi \wedge (y \neq e_1 \wedge \ldots \wedge y \neq e_n) | \sigma$ se puede resolver la aplicación de **Unfold**, y las igualdades introducidas aquellos de la forma $\pi \wedge y = e_i | \sigma$ abren la posibilidad de revelar el registro referenciado por x a partir de otros términos.

En cuanto a *abs*, las reglas que sufren cambios importantes son aquellas de la primera etapa, que deben tener en cuenta la posible existencia de ciclos y *sharing*, y el hecho de que un puntero de salida en un término no necesariamente es alcanzable desde la estructura que define.

Definición 124 (Función *abs* para **graph** (primera etapa)). *Las reglas de la primera etapa de abs del análisis que soporta el predicado* **graph** + *se listan a continuación:*

AbsNull1:

$$\frac{\pi \,|\, \sigma \vdash e = \mathbf{null}}{\pi \,|\, \sigma * \mathbf{graph}.(e \oplus C).D \implies \pi \,|\, \sigma * \mathbf{graph}.C.D}$$

AbsNull2:

$$\frac{\pi \,|\, \sigma \vdash e = \mathbf{null}}{\pi \,|\, \sigma * \mathbf{graph}.C.(e \oplus D) \implies \pi \,|\, \sigma * \mathbf{graph}.C.D}$$

AbsPartial:

$$\frac{\pi \,|\, \sigma \vdash e = e'}{\pi \,|\, \sigma * \mathbf{graph}.(e \oplus C).(e' \oplus D) \implies \pi \,|\, \sigma * \mathbf{graph}.C.D}$$

AbsArrowLeft:

$$\frac{\begin{array}{c} \sigma \approx \sigma' * x \mapsto l, v, r * \mathbf{graph}.(e \oplus C).D \qquad \pi \,|\, \sigma \vdash l = e \wedge (r = x \vee r = \mathbf{null}) \\ D \approx D' \uplus D'' \textit{ tal que } \pi \,|\, \sigma \vdash x \notin D' \textit{ y } \pi \,|\, \sigma \vdash x = e_i \textit{ para toda } e_i \textit{ en } D'' \end{array}}{\pi \,|\, \sigma \implies \pi \,|\, \sigma' * \mathbf{graph}.(x \oplus C).D'}$$

AbsArrowRight:

$$\frac{\begin{array}{c} \sigma \approx \sigma' * x \mapsto l, v, r * \mathbf{graph}.(e \oplus C).D \qquad \pi \,|\, \sigma \vdash r = e \wedge (l = x \vee l = \mathbf{null}) \\ D \approx D' \uplus D'' \textit{ tal que } \pi \,|\, \sigma \vdash x \notin D' \textit{ y } \pi \,|\, \sigma \vdash x = e_i \textit{ para toda } e_i \textit{ en } D'' \end{array}}{\pi \,|\, \sigma \implies \pi \,|\, \sigma' * \mathbf{graph}.(x \oplus C).D'}$$

AbsArrows1:

$$\frac{\begin{array}{c} \sigma \approx \sigma' * x \mapsto l, v, r * y \mapsto l', v', r' \qquad \pi | \sigma \vdash l = y \vee r = y \\ \{\!\{l, r\}\!\} \approx D' \uplus D'' \textit{ tal que } \pi \,|\, \sigma \vdash y \notin D' \textit{ y } \pi \,|\, \sigma \vdash y = e_i \textit{ para toda } e_i \textit{ en } D'' \end{array}}{\pi \,|\, \sigma \implies \pi \,|\, \sigma' * x \mapsto l, v, r * \mathbf{graph}.\{\!\{y\}\!\}.D'}$$

AbsArrows2:

$$\frac{\begin{array}{c} \sigma \approx \sigma' * x \mapsto l, v, r * T.C.D * T.E.F \\ \pi \,|\, \sigma \vdash l = e_i \textit{ para algún } e_i \textit{ en } C \qquad \pi \,|\, \sigma \vdash r = e_i \textit{ para algún } e_i \textit{ en } E \\ D \uplus F \approx G' \uplus G'' \textit{ tal que } \pi \,|\, \sigma \vdash e \notin G' \textit{ y } \pi \,|\, \sigma \vdash e = e_i \textit{ para toda } e_i \textit{ en } G'', \\ \textit{para toda } e \textit{ en } C \uplus D \end{array}}{\pi \,|\, \sigma \implies \pi \,|\, \sigma' * x \mapsto l, v, r * \mathbf{graph}.(C \uplus E).G'}$$

AbsArrows3:

$$\frac{\begin{array}{c} \sigma \approx \sigma' * x \mapsto l, v, r * \mathbf{graph}.(e \oplus e' \oplus C).D \qquad \pi \,|\, \sigma \vdash l = e \wedge r = e' \\ D \approx D' \uplus D'' \textit{ tal que } \pi \,|\, \sigma \vdash x \notin D' \textit{ y } \pi \,|\, \sigma \vdash x = e_i \textit{ para todo } e_i \textit{ en } D'' \end{array}}{\pi \,|\, \sigma \implies \pi \,|\, \sigma' * \mathbf{graph}.(x \oplus C).D'}$$

donde $e, e', e_1, \dots, e_n, l, v, r, l', v', r' \in \mathit{Expr}'$, $x, y \in \mathit{Var} \cup \mathit{Var}'$, $\sigma, \sigma' \in \Sigma$, $\pi \in \Pi$, *y* $C, D, D', D'', E, F, G', G'' \in \mathit{Multiset}$.

```
p := null;
if r ≠ null then m := r.1 else m := 1 fi;
while p ≠ null ∨ (r ≠ null ∧ m = 0) do
    if r = null ∨ m ≠ 0 then
        c := p.1;
        if c = 2 then
            q := r;
            r := p;
            p := p.2;
            r.2 := q
        else
            q := r;
            r := p.2;
            p.2 := p.0;
            p.0 := q;
            p.1 := 2
        fi
    else
        q := p;
        p := r;
        r := r.0;
        p.0 := q;
        p.1 := 1
    fi;
    if r ≠ null then m := r.1 else m := 1 fi
od
```

Figura 6.5: Adaptación del algoritmo Schorr-Waite.

6.5. Resultados experimentales

Lejos de pretender desarrollar una herramienta de verificación, implementamos nuestro análisis como una forma de validarlo. Con esta implementación condujimos experimentos sobre una variedad de algoritmos iterativos sobre árboles binarios de búsqueda, adaptados de la GNU LibAVL [1] y una adaptación del algoritmo de marcado Schorr-Waite de [151]. Nuestra implementación aplica una fase de abstracción sobre el estado final para hacer la postcondición más compacta, tratando cada variable que no ocurre en la precondición como una variable cuantificada. Presentamos a continuación dos casos en detalle para mostrar el nivel de complejidad en la manipulación de punteros que nuestro analisis es capaz de soportar.

En la figura 6.5 mostramos el código adaptado del algoritmo Schorr-Waite, en el cual combinamos los campos *mark* y *check* en el campo valor de los registros para que se adecue a nuestro modelo de memoria. Ejecutando el análisis desde la precondición

$$\{r = x \mid \mathbf{trees}.\{\![x]\!\}.\varnothing\}$$

obtenemos como postcondición

$$\{\mathbf{true} \,|\, \mathbf{emp},\ x \neq \mathbf{null} \,|\, \mathbf{trees}.\{\!\![x]\!\!\}.\varnothing,\ r \neq \mathbf{null} \,|\, \mathbf{trees}.\{\!\![r]\!\!\}.\varnothing\}$$

El invariante computado es conciso. Consta de ocho *symbolic heaps* cuya parte espacial es alguna de las fórmulas **emp**, $\mathbf{trees}.\{\!\![r]\!\!\}.\varnothing$, $\mathbf{trees}.\{\!\![p]\!\!\}.\varnothing$ o $\mathbf{trees}.\{\!\![r,p]\!\!\}.\varnothing$. Este es un buen ejemplo de la precisión que nuestro análisis es capaz de lograr. Si únicamente aplicamos las reglas de abstracción sobre variables cuantificadas, sin tener en cuenta el nivel de relevancia de las variables, obtenemos una postcondición equivalente, pero con un invariante que consta de más de 180 *symbolic heaps*. Utilizando el predicado **graph** en lugar de **trees** obtenemos resultados similares.

En la figura 6.6 presentamos el código fuente del algoritmo de borrado de un elemento dentro de un árbol binário de búsqueda, tomado de la GNU LibAVL. La guarda "*undef*" denota una condición que no podemos representar en nuestro lenguaje. Así mismo los valores *booleanos* son codificados con valores enteros. Fuera de estas excepciones, respetamos el estilo de programación del código original. Este algoritmo representa un gran desafío ya que manipula fuertemente la estructura de datos luego del ciclo principal, y por lo tanto requiere un invariante lo suficientemente preciso. Ejecutando nuestro análisis desde la precondición

$$\{\mathbf{true} \,|\, x \mapsto x', 0, 0 * \mathbf{trees}.\{\!\![x']\!\!\}.\varnothing\}$$

obtenemos la postcondición

$$\{\mathbf{true} \,|\, x \mapsto \mathbf{null}, 0, 0,\ \mathbf{true} \,|\, x \mapsto x', 0, 0 * \mathbf{trees}.\{\!\![x']\!\!\}.\varnothing\}$$

En ambos casos el puntero x representa el centinela del árbol. El invariante computado está conformado por once *symbolic heaps*, cuya parte espacial es esencialmente de la forma

$$x \mapsto x'_1, 0, 0 * \mathbf{trees}.\{\!\![x'_1]\!\!\}.\{\!\![q]\!\!\} * q \mapsto p, x'_2, x'_3 * \mathbf{trees}.\{\!\![p, x'_2]\!\!\}.\varnothing$$

con variaciones en la *ubicación* de los punteros p y x'_3 en el registro referenciado por q.

En el cuadro 6.1 resumimos los resultados de los experimentos llevados a cabo. Las columnas Link y Arrow representan los límites impuestos sobre el nivel de relevancia para abstraer una variable que forma una cadena de enlaces, y una variable que ocurre como el puntero en un término $\mapsto$ solitario respectivamente. La columna Inv. contiene el número de *symbolic heaps* que conforman el invariante del ciclo principal, y la columna Iter. el número de iteraciones de la fase de abstracción necesarias hasta alcanzar el punto fijo. En todos los casos el tiempo de ejecución del análisis, así como la memoria consumida corriéndolo en una estación de trabajo estándar son despreciables.

```
q := x;
p := x.0;
d := 0;
a := 0;
while p ≠ null ∧ a = 0 do
    b := p.1;
    if b = v then
        a := 1
    else
        q := p;
        if undef then
            d := 0;
            p := p.0
        else
            d := 1;
            p := p.2
        fi
    fi
od;
if p ≠ null then
    e := p.2;
    if e = null then
        if d = 0 then q.0 := p.0 else q.2 := p.0 fi
    else
        g := e.0;
        if g = null then
            e.0 := p.0;
            if d = 0 then q.0 := e else q.2 := e fi
        else
            k := g.0;
            while k ≠ null do e := g; g := k; k := k.0 od
            e.0 := g.2;
            g.0 := p.0;
            g.2 := p.2
            if d = 0 then q.0 := g else q.2 := g fi
        fi
    fi;
    dispose(p)
fi
```

Figura 6.6: Adaptación del algoritmo de borrado de un elemento dentro de un BST.

6.6. Discusión

Contribuciones

La semántica abstracta presentada define implícitamente un análisis que es capaz de verificar automáticamente propiedades interesantes sobre la forma de las estructuras de datos manipuladas por los programas sobre árboles binários y grafos. A partir

Algoritmo	Precondición / Postcondición	Link	Arrow	Inv.	Iter.
`min/max`	$\mathbf{true} \mid \mathbf{trees}.\{\![x]\!\}.\emptyset$ ——— $x = \mathbf{null} \mid \mathbf{emp}$ $x \neq \mathbf{null} \mid \mathbf{trees}.\{\![x]\!\}.\emptyset$	2	4	2	2
`destroy`	$\mathbf{true} \mid \mathbf{trees}.\{\![x]\!\}.\emptyset$ ——— $x = \mathbf{null} \mid \mathbf{emp}$ $x \neq \mathbf{null} \mid \mathbf{emp}$	4	4	4	3
`search`	$\mathbf{true} \mid \mathbf{trees}.\{\![x]\!\}.\emptyset$ ——— $x = \mathbf{null} \mid \mathbf{emp}$ $x \neq \mathbf{null} \mid \mathbf{trees}.\{\![x]\!\}.\emptyset$	2	4	4	3
`insert`	$\mathbf{true} \mid x \mapsto x', 0, 0 * \mathbf{trees}.\{\![x']\!\}.\emptyset$ ——— $\mathbf{true} \mid x \mapsto x'_1, 0, 0 * x'_1 \mapsto \mathbf{null}, x'_2, \mathbf{null}$ $\mathbf{true} \mid x \mapsto x', 0, 0 * \mathbf{trees}.\{\![x']\!\}.\emptyset$	3	3	12	4
`toVine`	$\mathbf{true} \mid x \mapsto x', 0, 0 * \mathbf{trees}.\{\![x']\!\}.\emptyset$ ——— $\mathbf{true} \mid x \mapsto \mathbf{null}, 0, 0$ $\mathbf{true} \mid x \mapsto x'_1, 0, 0 * x'_1 \mapsto \mathbf{null}, x'_2, \mathbf{null}$ $\mathbf{true} \mid x \mapsto x', 0, 0 * \mathbf{trees}.\{\![x']\!\}.\emptyset$	3	4	8	6
`delete`	$\mathbf{true} \mid x \mapsto x', 0, 0 * \mathbf{trees}.\{\![x']\!\}.\emptyset$ ——— $\mathbf{true} \mid x \mapsto \mathbf{null}, 0, 0$ $\mathbf{true} \mid x \mapsto x', 0, 0 * \mathbf{trees}.\{\![x']\!\}.\emptyset$	3	2	14	5
`Schorr-Waite`	$r = x \mid \mathbf{true}.\{\![x]\!\}.\emptyset$ ——— $\mathbf{true} \mid \mathbf{emp}$ $x \neq \mathbf{null} \mid \mathbf{trees}.\{\![x]\!\}.\emptyset$ $r \neq \mathbf{null} \mid \mathbf{trees}.\{\![r]\!\}.\emptyset$	4	4	8	4
`Schorr-Waite`	$r = x \mid \mathbf{graph}.\{\![x]\!\}.\emptyset$ ——— $\mathbf{true} \mid \mathbf{emp}$ $x \neq \mathbf{null} \mid \mathbf{graph}.\{\![x]\!\}.\emptyset$ $r \neq \mathbf{null} \mid \mathbf{graph}.\{\![r]\!\}.\emptyset$	4	4	17	4

Cuadro 6.1: Resultados de nuestro *shape analysis* sobre diferentes ejemplos.

de una precondición, el análisis calcula invariantes para cada ciclo y una postcondición. Esta semántica es una sobre-aproximación de la semantica operacional sobre el modelo de memoria estándar. Por lo tanto, puede reportar falsos errores de acceso a memoria y falsos *memory leaks*. Sin embargo, los experimentos muestran un buen ajuste entre los estados abstractos computados y aquellos esperables según la semántica concreta.

Para definir el análisis introducimos novedosos predicados lineales, **trees** y **graph** para describir estructuras no lineales con múltiples puntos de entrada y de salida. Las buenas propiedades sintácticas que poseen permiten una caracterización simple de la fase de abstracción presente en el análisis. Una investigación no concluída, basada en variaciones de estos predicados y de las reglas de abstracción, parece prometedora en la posibilidad de verificar algoritmos sobre árboles binarios de búsqueda y árboles AVL que requieren ciertas propiedades sobre balanceo, y sobre arreglos, así como el algoritmo de *garbage collection* por copia de Cheney [15].

El novedoso análisis del nivel de relevancia de las variables de programa, aunque es una idea muy sencilla, redunda en una mejora significativa en la compacidad de

los invariantes, sin perjudicar la necesaria precisión para obtener postcondiciones relevantes. En algunos de los experimentos llevados a cabo, se consigue una reducción dramática en la cantidad de fórmulas que componen los estados abstractos. Esto hace más rápido el proceso de análisis y ayuda en la computación de invariantes correctos e intuitivamente comprensibles. Además nos habilita a prever un buen comportamiento del analisis sobre programas de mayor escala. Para permitir el análisis sobre este tipo de programas sería interesante extender el concepto de Bi-Abducción [37] a nuestro dominio para soportar especificaciones incompletas y llamadas a procedimientos.

Para finalizar podemos decir que, ya que nuestro análisis se basa en aquel de [64], heredamos todas las ventajas de la *Separation Logic*. Nuestras reglas de reescritura son implicaciones válidas cuya verificación semántica es muy sencilla, mientras que la ejecución simbólica se deriva directamente de las ternas de Hoare que caracterizan los comandos atómicos. Así, nuestro análisis es intuitivo y su corrección es fácilmente verificable.

Trabajos relacionados

Ya que nuestro trabajo es una extensión de [64], los trabajos más cercanos son aquellos también derivados del mismo, como [37, 9, 154]. Estas extensiones están motivadas por la intención de verificar sistemas de código de gran escala y son implementadas en la herramienta SpaceInvader. Permiten la verificación de estructuras lineales posiblemente combinadas de diferentes maneras. Nuestro trabajo extiende el dominio de aplicabilidad de esos métodos dando soporte a estructuras no lineales. El trabajo [11] es predecesor de [64] e introduce la herramienta Smallfoot, basada en un tipo diferente de ejecución simbólica, aunque soporta árboles binarios. Esta herramienta toma un programa anotado con pre y postcondiciones y reduce las condiciones de verificación a implicaciones lógicas entre los estados abstractos, que son resueltas por un procedimiento de decisión completo. La herramienta no soporta ciclos y por lo tanto no computa invariantes. Sin embargo permite la definición de programas recursivos, para cuya verificación el predicado usual de árboles binarios (**btree**) parece adecuado. En [111, 74] se presentan análisis similares al de [64], tanto en los aspectos técnicos como en sus limitaciones.

La herramienta Xisa [47] también se basa en la *Separation Logic* para describir estados abstractos, pero en lugar de especificar estructuras con predicados fijos, utiliza predicados inductivos generalizados. Estos predicados son provistos por el usuario en forma de verificadores de invariantes sobre las estructuras de datos. Las operaciones de *fold* y *unfold* de los predicados guían las estrategias automáticas de materialización (*rearr* en nuestro trabajo) y abstracción. La extensión de [46] agrega poder expresivo para especificar relaciones entre regiones del *heap*, como invariantes de

ordenación, punteros *back* y *cross*, etc. Se presenta como caso de estudio la inserción de un elemento en un árbol *Red-Black*.

Otra herramienta que basa su análisis en *Separation Logic* es Verifast [86]. Permite la verificación de ternas de Hoare que utilizan predicados inductivos para especificar las estructuras y funciones recursivas puras sobre esos tipos. Genera las condiciones de verificación que son resueltas por un *SMT solver*. Deben proveerse invariantes para los ciclos, e instrucciones que guíen las operaciones de *fold* y *unfold* de los predicados. Es capaz de verificar una implementación recursiva de una biblioteca de programas sobre árboles binarios, y el patrón *composite*, con su estructura de grafo subyacente [87].

El sistema PALE [104, 89] permite la verificación de programas especificados con fórmulas en *Weak Monadic Second-order Logic of Graph Types*. Los ciclos deben estar anotados con invariantes y las condiciones de verificación son descargadas en la herramienta MONA [79]. La extensión [71] hace más eficiente la verificación de algoritmos sobre estructuas con forma arbórea.

El análisis de [96] utiliza *shape graphs* y *grammar annotations* para especificar estructuras de datos acíclicas, y descubre automaticamente descripciones de las estructuras que ocurren en los puntos intermedios de la ejecución de los programas. Este análisis verifica los algoritmos de Schorr-Waite, de destrucción de árboles y de construcción de un *heap* binomial.

El *shape analysis* paramétrico de [137, 21] basado en *3-valued Logic*, en conjunto con su implementación TVLA [99], conforman el marco de trabajo más general, poderoso y utilizado en la verificación de propiedades sobre la fórma de estructuras dinámicas en programas que involucran complejas manipulaciones. Este marco de trabajo debe ser instanciado para cada caso particular a través de predicados de instrumentación definidos por el usuario que especifican el tipo de estructuras soportadas y las formas de abstracción. El análisis presentado en [100] permite la verificación de corrección parcial (como en nuestro caso) y también la corrección total (terminación) del algoritmo de Schorr-Waite sobre árboles. Nuestra propuesta es menos ambiciosa, ya nuestro análisis está especializado en árboles y grafos. Sin embargo nuestro dominio abstracto permite una alta eficiencia y precisión en una clase bastante amplia de algoritmos. Además nuestro marco de trabajo soporta el razonamiento local que ha demostrado gran potencial para escalar los análisis a programas de tamaño real, como queda demostrado en [37, 154].

Palabras finales

7

Al comienzo de este trabajo introducimos la crítica que hacen De Millo *et al.* al programa de la verificación formal y mencionamos que tal crítica tiene aún asidero en la actualidad. Según estos autores la tarea misma de la verificación formal es fútil; es un juego matemático tedioso que no aplica a la tarea real de la programación por la escala y la permanente evolución de los sistemas informáticos, y que no colabora en la producción de un conocimiento comunitario dentro del ámbito de las ciencias de la computación. Estos argumentos de índole pragmático subsisten hasta el día de hoy dentro del seno mismo de la comunidad, a pesar de los muchos avances que se han logrado en la automatización del proceso de verificación, a través de herramientas como demostradores de teoremas, análisis estáticos (dentro de los cuales entran los *shape analysis* como los tratados en el capítulo 6), etc.

Mucho tiempo ha pasado desde la aparición de la crítica de De Millo *et al.* y efectivamente las prácticas que se han dado dentro de la comunidad de las ciencias de la computación a lo largo de ese tiempo relativizan sus argumentos. Pero más allá del *revisionismo histórico* que podamos hacer, hay un aspecto central en su crítica, y que se preserva entre quienes mantienen hoy una postura similar: la verificación formal se reduce a una concepción *instrumental*, esto es, se la entiende como un mero *método* para resolver *un* problema particular -la corrección de un programa respecto a su especificación formal-.

En este trabajo, introducimos el concepto de intéprete, que permite caracterizar nociones claves como sistema computacional, programa, computación, etc, y discutir una variedad de problemas, como la validez metodológica de la verificación formal y el pancomputacionalismo. El concepto de intérprete es muy *abstracto*, tanto que permite además caracterizar la organización interna de los sistemas informáticos. Podemos pensar la tarea de creación de sistemas computacionales como la construcción de múltiples niveles de abstracción que permiten ir acortando la distancia entre los dispositivos que efectivamente realizan una computación (*chips* y sus configuraciones eléctricas) y la funcionalidad de alto nivel que el sistema provee (en términos de conceptos abstractos del dominio del problema que el sistema intenta resolver). Los intérpretes ocurren a lo largo de esa cadena de abstracciones, proveyendo las garantías necesarias para que tenga sentido. Garantizar que un compilador obedece la semántica formal de un lenguaje de programación, es verificar la implementación de un intérprete; verificar la operación de una computadora digital respecto a las propiedades caracterizadas por su diseño, también lo es. Y no parece estar en duda

la necesidad y utilidad de llevar adelante estas tareas. La verificación formal es el ámbito donde llevar adelante una tarea semejante respecto a los programas. Podemos arriesgar que la concepción instrumental sobre la misma es la que promueve que este último paso es irrealizable o carece de sentido.

Creemos que la verificación formal, o más en general el razonamiento formal sobre programas, no sólo abre la posibilidad de conseguir certeza matemática sobre la corrección de los programas, sino que provee un conjunto de herramientas conceptuales y metodológicas para el análisis sobre los mismos, que enriquecen el dominio de la programación y posibilitan encontrar (mejores) soluciones. Con estas herramientas, el interés no está puesto únicamente en la tarea de demostración rigurosa de un programa, sino en un abordaje más amplio de la tarea de programación, que puede incluirla, pero que no se reduce a ella.

A partir de la introducción a la *Separation Logic* provista, deberían ser evidentes las ventajas que un formalismo tal provee para el razonamiento sobre la memoria dinámica, y al mismo tiempo la complejidad de su uso. Pero tal complejidad es inherente al problema mismo; lo que hace el formalismo es simplemente evidenciarla. Consideremos, por ejemplo, el código del algoritmo de marcado de Schorr-Waite (fig. 6.5, pág. 170). Difícilmente un programador bien entrenado sea capaz de comprender y/o explicar cómo se ejecuta tal programa, aún conociendo las ideas subyacentes. El problema es que tales ideas están expresadas en término de conceptos abstractos (árbol, grafo, lista de nodos recorridos, etc); y el código del programa en término de los punteros que *representan* tales conceptos. Existe una distancia entre el dominio del problema y el dominio de la representación que sólo puede ser salvado con herramientas analíticas, como las que provee un sistema formal como la *Separation Logic*.

A pesar de su utilidad, la *Separation Logic* no carece de limitaciones, en particular, en escenarios donde el *sharing* de memoria es la regla y no la excepción. Lamentablemente esta situación ocurre frecuentemente, ya sea en la representación misma de una estructura de datos, ya sea entre instancias de diferentes estructuras. En este trabajo presentamos nuestra *Sharing Logic* que permite analizar este problema, caracterizarlo de forma precisa y establecer las condiciones mínimas que deben garantizarse para poder razonar de forma relativamente local. Esta lógica provee un lenguaje rico para la especificación de complejas situaciones de *sharing*, un cálculo para razonar sobre tales especificaciones y un sistema de prueba composicional. Analizamos además el problema de la decidibilidad de un fragmento de la lógica, lo que nos permite comprender en profundidad la naturaleza de los modelos y sus caracteristicas, que habilitan la decisión mecánica pero que también subyacen a formas de razonamiento más *intuitivo*. Tal fragmento posibilita además automatizar parte de las tareas asociadas a la verificación de los programas. Finalmente estudiamos la posibilidad de automatizar un proceso de verificación formal en forma de *shape*

analysis, ya no considerando el problema del *sharing* en general, sino de una fórma específica, que ocurre en cierto tipo de estructuras de datos no lineales como árboles, *dags* y grafos generales. Esta herramienta permite verificar propiedades interesantes como la preservación de las estructuras de datos, la ausencia de *memory leaks* y errores de acceso a memoria; analizar programas en busca de invariantes, y, en general, colaborar en la construcción de un conocimiento más detallado sobre los mismos.

Creemos que las contribuciones que hemos presentados colaboran a construir un conocimiento mas profundo sobre la naturaleza de los problema que aparecen con las diferentes formas de utilizar la memoria dinámica en la programación. Tenemos confianza en que contar con herramientas conceptuales y metodológicas para la caracterización, análisis y abordaje de estos problemas, posibilitan la construcción de programas que manejan dinámicamente la memoria con mayor certeza sobre su corrección y seguridad.

Apéndice

Resultados del capítulo 2

La validez del lema 1 se desprende directamente de las definiciones involucradas. Consideramos que los comentarios incluidos en el cuerpo del trabajo son suficientes entrever la validez del lema 3. Finalmente, es ampliamente conocido que el lema 2 es una conjetura aún no demostrada (ni refutada).

Resultados del capítulo 3

Existen mútliples referencias en la literatura sobre los resultados del capítulo 3. Las reglas de cálculo de la *Separation Logic* (lemas 4, 5, 6, 7, 8) se demuestran fácilmente a partir de la semántica de las fórmulas. Algunas son consecuencia directa de reglas del sistema deductivo de la *Bunched Implications Logic* [131]. Respecto a la consistencia del sistema deductivo de la SL^H (teorema 1), podemos referenciar [92] como una presentación comprehensiva de las reglas de la lógica de Hoare. La validez de las reglas que especifican los comandos de manipulación del *heap* y la regla de *frame* se demuestra en [152]. En este trabajo también se puede encontrar una discusión de las clases de fórmulas (lema 9). En [134] se encuentran referencias a las clases de fórmulas, y demostraciones de algunas propiedades sobre estructuras de datos (lemas 10, 11, 12).

Todo lo referido al soporte para tipos abstractos de datos de la SL_T y SL_T^H, acerca de su semántica (lemas 13, 14), reglas de cálculo (lema 15) y la consistencia del sistema de prueba a la Hoare (teorema 2) se encuentra en [119].

Resultados del capítulo 4

La demostración del lema 16 puede encontrarse en [134]. Las semánticas de las variantes del operador de *septraction* (lemas 17, 18) se deducen directamente de las semánticas de las respectivas implicaciones espaciales. La generalización que la ShL representa (lema 19) se demuestra por un cálculo elemental sobre la semántica de los operadores. Lo mismo ocurre con las reglas del cálculo de ShL (lemas 20, 21, 22, 23, 25, 26, 31, corolario 1). Los resultados sobre las clases de fórmulas son mayormente estándar (lemas 24, 27, 28, 29).

En cuanto a la consistencia del sistema deductivo de ShL^H (teorema 3), presentamos más adelante los casos relevantes de su demostración (la validez de las reglas de *frame*). Lo referido a la extensión con tipos abstractos de datos (lema 32, teorema 4) no aporta dificultades mas alla de los resultados respectivos sobre la *Separation Logic*.

Teorema 3 (Consistencia del sistema deductivo de ShL^H). *Dados $p, q \in Form'$, $c \in Comm$, si $\vdash \{p\}\ c\ \{q\}$ se deduce utilizando las reglas del sistema deductivo, entonces $\models \{p\}\ c\ \{q\}$.*

Demostración. Utilizamos los siguientes resultados, cuya demostración puede encontrarse en [152].

- **Safety y Termination Monotonicity:** Sean $s, h \in States$, $h \in Heaps$, y $c \in Comm$.

  1. Si $c, (s, h)$ es *safe* y $h \perp h'$, entonces $c, (s, h \cup h')$ es *safe*.
  2. Si $c, (s, h)$ *termina normalmente* y $h \perp h'$, entonces $c, (s, h \cup h')$ *termina normalmente*.

- **Propiedad de Frame:** Sean $s, h \in States$, $h', h_0, h_1, h'_0 \in Heaps$, y $c \in Comm$. Supongamos que $c, (s, h_0)$ es *safe* y $c, (s, h_0 \cup h_1) \rightsquigarrow^* c, (s, h')$, para algún $h_1 \perp h_0$. Entonces existe h'_0 tal que $h'_0 \perp h_1$, $c, (s, h_0) \rightsquigarrow^* c, (s, h'_0)$ y $h' = h'_0 \cup h_1$.

Veamos en detalle la validez de las reglas de *frame*. Supongamos que las premisas de la regla se cumplen. Supongamos además que $s, h_p \cup h_r \models p$, $s, h_r \models r$ y $s, h_i \cup h_r \models i$, para algunos $h_p, h_r, h_i \in Heaps$, tales que $h_p \perp h_r \perp h_i$. La especificación $\vdash \{p\}\ c\ \{q\}$ dice que $c, (s, h_p \cup h_r)$ es *safe* y por *Safety Monotonicity* la configuración $c, (s, h_p \cup h_r \cup h_i)$ es *safe*. Por otro lado si $c, (s, h_p \cup h_r \cup h_i) \rightsquigarrow^* (s', h')$, la *Propiedad de Frame* dice que existe un h'_q tal que $h'_q \perp h_i$, $c, (s, h_p \uplus h_r) \rightsquigarrow^* (s', h'_q)$ y $h' = h'_q \cup h_i$. Así por la validez de la especificación tenemos que $s', h'_q \models q$.

- Para el caso de la GFR:

  La premisa $\models q \Rightarrow (r' * \mathbf{true})$ nos asegura que existe $h'_r \subseteq h'_q$ tal que $(s', h'_r) \models r'$. Para concluir que $s', h'_q \cup h_i \models q\ \langle *\!: r' \rangle\ i'$ es necesario demostrar que $s', h'_r \cup h_i) \models i'$. Para ello utilizamos la hipótesis $\models (r \multimap\!\!\circledast\ i) * r' \Rightarrow i'$. Sabemos ya que $s', h'_r \models r'$, por lo tanto es suficiente ver que $(s', h_i) \models r \multimap\!\!\circledast\ i$. Por nuestra hipótesis inicial, tenemos que $s, h_i \cup h_r \models i$, luego se deduce $s, h_i \models (r \multimap\!\!\circledast\ i)$ y como $(FV.r \cup FV.i) \cap Mod.c = \emptyset$, se cumple que $s', h_i \models (r \multimap\!\!\circledast\ i)$.

- Para el caso de la SFR:

  Por la premisa $\models (\exists \vec{x}' \cdot (p \multimap\!\!\circledast (p\ \langle *\!: r \rangle\ i))_{/\vec{x} \leftarrow \vec{x}'}) * q \Rightarrow q\ \langle *\!: r' \rangle\ i'$ alcanza con ver que $s', h'_q \cup h_i \models (\exists \vec{x}' \cdot (p \multimap\!\!\circledast (p\ \langle *\!: r \rangle\ i))_{/\vec{x} \leftarrow \vec{x}'}) * q$, o equivalentemente,

$s', h_i \models \exists \vec{x}' \cdot (p \mathrel{-\!\circledast} (p \ \langle *\colon r \rangle \ i))_{/\vec{x} \leftarrow \vec{x}'}$. Como $\{\vec{x}\} = Mod.c$ y $\vec{x}' \notin (FV.p \cup FV.r \cup FV.i)$, esto equivale a demostrar que $s, h_i \models p \mathrel{-\!\circledast} (p \ \langle *\colon r \rangle \ i)$, que se deduce directamente de la semántica de $\mathrel{-\!\circledast}$ y las suposiciones iniciales.

- Para el caso de la IFR:

  Como c no modifica el *heap*, entonces $h'_q = h_p \cup h_r$, luego $s', h_p \cup h_r \models q$. Por otro lado como c no modifica variables que ocurren en r o i tenemos que $s', h_r \models r$ y $s', h_i \cup h_r \models i$. Así finalmente concluimos que $s', h'_q \cup h_i \models q \ \langle *\colon r \rangle \ i$.

□

Resultados del capítulo 5

Los resultados más relevantes son las propiedades de partición y unión de las tres relaciones (lemas 34 35, 38, 39, 43, 44), y sus consecuencias en la preservación de satisfacción de las fórmulas (lemas 36, 40, 45). Demostrar que cada una de las relaciones es relación de equivalencia (33, 37, 42) es elemental. El hecho que la relación combinada es relación de equivalencia (lema 46) es una consecuencia directa de los lemas anteriores; de igual manera, la preservación de satisfacción del corolario 2 es consecuencia directa de los lemas 33, 37 y 42.

Los lemas 34, 35 y 36 son directos. La demostración de las propiedades de partición y unión de los lemas 38, 39, 43 y 44 se demuestran completamente más adelante. Además se demuestran en detalle los casos mas importantes de los resultados de preservación de satisfacción de los lemas 40 y 45.

Presentamos algunos comentarios sobre la demostración del teorema 5, que caracteriza los modelos en forma normal. El corolario 3 es una consecuencia directa. La posibilidad de decidir la satisfacción de una fórmula respecto a un modelo (lema 47) se deduce a partir de la semántica de fórmulas y resultados básicos de teoría de la computabilidad, lo mismo que el teorema 6. El corolario 4 es consecuencia directa de estos resultados.

Lema 38 (Propiedad de partición de $\approx$)**.** *Sean $n = n_1 + n_2$, $h, h' \in Heaps_D$ tales que $h, c \approx^L_n h', c'$ para algunos $c, c' \in Heaps_D$, y $h = h_1 \cup h_2$ para algunos $h_1, h_2 \in Heaps$ con $h_1 \perp h_2$. Entonces existen $h'_1, h'_2 \in Heaps_D$ tales que:*

1. $h_1, h_2 \cup c \approx^L_{n_1} h'_1, h'_2 \cup c'$,
2. $h_2, h_1 \cup c \approx^L_{n_2} h'_2, h'_1 \cup c'$,
3. $h'_1 \perp h'_2$ *y* $h' = h'_1 \cup h'_2$.

Demostración. Tengamos presente que $h, c \approx_n^L h', c'$ y por lo tanto se dan las condiciones 1, 2, 3 y 4. Sea $\{A, B\}$ una partición de $dom.h' - reach_L.(h' \cup c')$ de modo que se da alguna de las tres siguientes situaciones:

(i) Cuando $|dom.h - reach_L.(h \cup c)| < n$, $|A| = |dom.h_1 - reach_L.(h \cup c)|$ y $|B| = |dom.h_2 - reach_L.(h \cup c)|$. Esto es posible ya que por la condición 3

$$\begin{aligned} |dom.h' - reach_L.(h' \cup c')| &= |dom.h - reach_L.(h \cup c)| \\ &= |dom.h_1 - reach_L.(h \cup c)| + |dom.h_2 - reach_L.(h \cup c)| \end{aligned}$$

(ii) Cuando $|dom.h - reach_L.(h \cup c)| \geqslant n$ y $|dom.h_2 - reach_L.(h \cup c)| \geqslant n_2$, $|A| = \text{mín}\,.|dom.h_1 - reach_L.(h \cup c)|.n_1$. Esto es posible ya que por la condición 4

$$\begin{aligned} |dom.h' - reach_L.(h' \cup c')| &\geqslant n \\ &\geqslant n_1 \\ &\geqslant \text{mín}\,.|dom.h_1 - reach_L.(h \cup c)|.n_1 \end{aligned}$$

(iii) Cuando $|dom.h - reach_L.(h \cup c)| \geqslant n$ y $|dom.h_2 - reach_L.(h \cup c)| < n_2$, $|B| = |dom.h_2 - reach_L.(h \cup c)|$. Esto es posible ya que por 4

$$\begin{aligned} |dom.h' - reach_L.(h' \cup c')| &\geqslant n \\ &\geqslant n_2 \end{aligned}$$

Tomemos $h'_A \doteq h'|_{A \cup (dom.h_1 \cap reach_L.(h \cup c))}$ y $h'_B \doteq h'|_{B \cup (dom.h_2 \cap reach_L.(h \cup c))}$ como candidatos a h'_1 y h'_2 respectivamente. Veamos que vale $h' = h'_A \uplus h'_B$.[1] Por un lado tenemos que la intersección de los dominios de h'_A y h'_B es vacía:

$$\begin{aligned} dom.h'_A \cap dom.h'_B &= (A \cup (dom.h_1 \cap reach_L.(h \cup c))) \cap (B \cup (dom.h_2 \cap reach_L.(h \cup c)) \\ &= (A \cap dom.h_2 \cap reach_L.(h \cup c)) \cup (B \cap dom.h_1 \cap reach_L.(h \cup c))) \cup \\ &\quad (A \cap B) \cup (dom.h_1 \cap reach_L.(h \cup c) \cap dom.h_2 \cap reach_L.(h \cup c)) \end{aligned}$$

y como $\{A, B\}$ es una partición y $h_1 \perp h_2$

$$dom.h'_A \cap dom.h'_B = (A \cap dom.h_2 \cap reach_L.(h \cup c)) \cup (B \cap dom.h_1 \cap reach_L.(h \cup c))$$

y como $dom.h_2 \cap reach_L.(h \cup c) \subseteq dom.h' \cap reach_L.(h' \cup c')$ por la condición 1 y además $A \subseteq dom.h' - reach_L.(h' \cup c')$

$$dom.h'_A \cap dom.h'_B = B \cap dom.h_1 \cap reach_L.(h \cup c)$$

y análogamente para h_1 y B

$$dom.h'_A \cap dom.h'_B = \emptyset$$

[1]Con $A \uplus B$ denotamos $A \cup B$ cuando además se da que $A \perp B$

Por otro lado, la unión de los dominios de h'_A y h'_B es igual al dominio de h':

$$\begin{aligned} dom.h'_A \cup dom.h'_B &= A \cup (dom.h_1 \cap reach_L.(h \cup c)) \cup B \cup (dom.h_2 \cap reach_L.(h \cup c) \\ &= A \cup B \cup (dom.h \cap reach_L.(h \cup c)) \\ &= (dom.h' - reach_L.(h' \cup c')) \cup (dom.h \cap reach_L.(h \cup c)) \end{aligned}$$

y por 1

$$\begin{aligned} dom.h'_A \cup dom.h'_B &= (dom.h' - reach_L.(h' \cup c')) \cup (dom.h' \cap reach_L.(h' \cup c')) \\ &= dom.h' \end{aligned}$$

Veamos que vale $h_1, c \cup h_2 \approx^L_{n_1} h'_1, c' \cup h'_2$ demostrando cada condición.

- Veamos que $dom.h_1 \cap reach_L.(h_1 \cup h_2 \cup c) = dom.h'_A \cap reach_L.(h'_A \cup h'_B \cup c')$.

  $$\begin{aligned} dom.h'_A \cap reach_L.(h'_A \cup h'_B \cup c') &= dom.h'_A \cap reach_L.(h' \cup c') \\ &= (A \cup (dom.h_1 \cap reach_L.(h \cup c))) \cap reach_L.(h' \cup c') \\ &= (A \cap reach_L.(h' \cup c')) \\ &\quad \cup (dom.h_1 \cap reach_L.(h \cup c) \cap reach_L.(h' \cup c')) \end{aligned}$$

  como $A \subseteq dom.h' - reach_L.(h' \cup c')$

  $$dom.h'_A \cap reach_L.(h'_A \cup h'_B \cup c') = dom.h_1 \cap reach_L.(h \cup c) \cap reach_L.(h' \cup c')$$

  como $dom.h_1 \cap reach_L.(h \cup c) \subseteq dom.h' \cap reach_L.(h' \cup c')$

  $$\begin{aligned} dom.h'_A \cap reach_L.(h'_A \cup h'_B \cup c') &= dom.h_1 \cap reach_L.(h \cup c) \\ &= dom.h_1 \cap reach_L.(h_1 \cup h_2 \cup c) \end{aligned}$$

- La condición que $h_1.l = h'_A.l$ para todo $l \in dom.h_1 \cap reach_L.(h \cup c)$ vale puesto que vale 2 para h y h', y h_1 y h'_A son restricciones de h y h' respectivamente.

- Tengamos presente de ahora en más que, dada la definición de h'_A y el hecho que $dom.h_1 \cap reach_L.(h \cup c) \subseteq reach_L.(h' \cup c')$ por la condición 1 de la hipótesis, tenemos que

  $$A = dom.h'_A - reach_L.(h' \cup c')$$

  Supongamos que se da la situación (i), entonces

  $$\begin{aligned} |dom.h_1 - reach_L.(h_1 \cup h_2 \cup c)| &= |dom.h_1 - reach_L.(h \cup c)| \\ &= |A| \\ &= |dom.h'_A - reach_L.(h' \cup c')| \\ &= |dom.h'_A - reach_L.(h'_A \cup h'_B \cup c')| \end{aligned}$$

Luego se cumple 3 o 4.

Supongamos que se da la situación (ii). Si $|dom.h_1 - reach_L.(h_1 \cup h_2 \cup c)| \geqslant n_1$, entonces $|A| = n_1$ y vale 4. En caso contrario vale 3 puesto que $|A| = |dom.h_1 - reach_L.(h_1 \cup h_2 \cup c)|$.

Finalmente supongamos que se da la situación (iii). Entonces necesariamente vale que $|dom.h_1 - reach_L.(h_1 \cup h_2 \cup c)| \geqslant n_1$, y como $|B| < n_2$ y por 1 de la hipótesis $|dom.h' - reach_L.(h' \cup c')| \geqslant n$, necesariamente $|A| \geqslant n_1$ y por lo tanto vale 4.

Para terminar, veamos que vale $h_2, c \cup h_1 \approx^L_{n_1} h'_2, c' \cup h'_1$. La demostración de las condiciones 1 y 2 son análogas a las anteriores, lo mismo que si se da la situación (i). Además siguiendo un razonamiento análogo tenemos que

$$B = dom.h'_B - reach_L.(h' \cup c')$$

Supongamos que se da la situación (ii). Como $|A| \leqslant n_1$ y por 1 de la hipótesis $|dom.h' - reach_L.(h' \cup c')| \geqslant n$, necesariamente $|B| \geqslant n_2$ y vale 4. Si se da la situación (iii), vale directamente 3. □

Lema 39 (Propiedad de unión de $\approx$). *Sean $n_1, n_2 \in \mathbb{N}_0$, $h_1, h'_1, h_2, h'_2 \in Heaps_D$ tales que $h_1, h_2 \cup c \approx^L_{n_1} h'_1, h'_2 \cup c'$ y $h_2, h_1 \cup c \approx^L_{n_1} h'_2, h'_1 \cup c'$, para algunos $c, c' \in Heaps_D$ donde $h_1 \perp h_2$ y $h'_1 \perp h'_2$. Entonces si tomamos $h = h_1 \cup h_2$ y $h' = h'_1 \cup h'_2$*

$$h, c \approx^L_{\text{mín}\,.n_1.n_2} h, c'$$

Demostración. Veamos que $dom.h \cap reach_L.(h \cup c) = dom.h' \cap reach_L.(h' \cup c')$.

$$\begin{aligned} dom.h \cap reach_L.(h \cup c) &= dom.h_1 \cup h_2 \cap reach_L.(h_1 \cup h_2 \cup c) \\ &= (dom.h_1 \cap reach_L.(h_1 \cup h_2 \cup c)) \cup (dom.h_2 \cap reach_L.(h_1 \cup h_2 \cup c)) \\ &= (dom.h'_1 \cap reach_L.(h'_1 \cup h'_2 \cup c)) \cup (dom.h'_2 \cap reach_L.(h'_1 \cup h'_2 \cup c)) \\ &= dom.h'_1 \cup h'_2 \cap reach_L.(h'_1 \cup h'_2 \cup c) \\ &= dom.h' \cap reach_L.(h' \cup c') \end{aligned}$$

La condición que $h.l = h'.l$ para toda $l \in dom.h \cap reach_L.(h \cup c)$ value pues por hipótesis $h_1.l = h'_1.l$ y $h_2.l = h'_2.l$ ya sea que $l \in dom.h_1 \cap reach_L.(h_1 \cup h_2 \cup c)$ o $l \in dom.h_2 \cap reach_L.(h_2 \cup h_1 \cup c)$.
Ahora sin pérdida de generalidad supongamos que $n_1 = \text{mín}\,.n_1.n_2$. El caso en que $n_2 = \text{mín}\,.n_1.n_2$ sigue un razonamiento simétrico. Supongamos que $|dom.h -$

$reach_L.(h \cup c)| < n_1$, luego $|dom.h_1 - reach_L.(h \cup c)| < n_1$ y $|dom.h_2 - reach_L.(h \cup c)| < n_1 \leqslant n_2$. Por lo tanto

$$|dom.h'_1 - reach_L.(h' \cup c)| = |dom.h_1 - reach_L.(h \cup c)| \text{ y}$$
$$|dom.h'_2 - reach_L.(h' \cup c)| = |dom.h_2 - reach_L.(h \cup c)|$$

luego

$$|dom.h' - reach_L.(h' \cup c)| = |dom.h - reach_L.(h \cup c)|$$

Finalmente, supongamos que $|dom.h - reach_L.(h \cup c)| \geqslant n_1$. Puede valer que $|dom.h_1 - reach_L.(h \cup c)| < n_1$ o $|dom.h_1 - reach_L.(h \cup c)| \geqslant n_1$. Lo mismo ocurre con h_2. En cualquier caso sabemos que

$$|dom.h'_1 - reach_L.(h' \cup c)| \geqslant \text{mín} .|dom.h_1 - reach_L.(h \cup c)|.n_1 \text{ y}$$
$$|dom.h'_2 - reach_L.(h' \cup c)| \geqslant \text{mín} .|dom.h_2 - reach_L.(h \cup c)|.n_2$$

con $n_2 \geqslant n_1$ y $|dom.h_1 - reach_L.(h \cup c)| + |dom.h_2 - reach_L.(h \cup c)| \geqslant n_1$. Luego

$$|dom.h'_1 - reach_L.(h' \cup c)| + |dom.h'_2 - reach_L.(h' \cup c)| \geqslant n_1$$
$$|dom.h' - reach_L.(h' \cup c)| \geqslant n_1$$

□

Lema 40 (Preservación de satisfacción de $\approx$)**.** *Sean $p \in Form_D$, $X \supseteq FV.p$, $n \in \mathbb{N}_0$, $L \subseteq Loc$, s, h y $s, h' \in States_D$, tales que $L \supseteq s^{\upharpoonright}.X$, $n \geqslant |p|$, y $h, c \approx^L_n h', c'$, para algunos $c, c' \in Heaps_D$ donde $h \perp c$ y $h' \perp c'$. Entonces*

$$s, h \vDash p \;\; \textit{si y sólo si} \;\; s, h' \vDash p$$

donde $s^{\upharpoonright}.X$ denota el lifting *de s al conjunto de variables X.*

Demostración. La demostración es por inducción en la fórmula p. Para una fórmula de la forma $e_1 = e_2$, el lema es trivial ya que se satisface en cualquier *heap*. Cuando consideramos fórmulas como **emp**, $x \mapsto y$ o $\mathbf{lseg}.x.y$, tenemos que $dom.h - reach_L.(h \cup c) = \varnothing$, y luego $h = h'$. Para los casos de la forma $p \vee q$ y $\neg p$, la demostración sale directamente por hipótesis inductiva.

Consideremos en detalle una fórmula de la forma $p \langle *\colon r \rangle\, q$. Si $s, h \vDash p \langle *\colon r \rangle\, q$ existen $h_1, h_2, h_3 \in Heaps$ tales que $h = h_1 \uplus h_2 \uplus h_3$ y

1. $s, h_1 \cup h_2 \vDash p$
2. $s, h_3 \cup h_2 \vDash q$
3. $s, h_2 \vDash r$

Además $n \geqslant |p| + |q| + \text{máx}.\{|p|, |q|, |r|\}$. Sean $n_1, n_2, n_3 \in \mathbb{N}_0$ tales que $n = n_1 + n_2 + n_3$ y $n_1 \geqslant |p|$, $n_3 \geqslant |q|$ y $n_2 \geqslant \text{máx}.\{|p|, |q|, |r|\}$. Tomando $h_{23} = h_2 \uplus h_3$, por el lema 38 existen $h'_1, h'_{23} \in \mathit{Heaps}$ tales que

$$h_1, c \cup h_{23} \approx^L_{n_1} h'_1, c' \cup h'_{23} \text{ y} \quad (7.1)$$
$$h_{23}, c \cup h_1 \approx^L_{n_2+n_3} h'_{23}, c' \cup h'_1$$

Nuevamente por el lema 38, existen $h'_2, h'_3 \in \mathit{Heaps}$ tales que

$$h_2, c \cup h_1 \cup h_3 \approx^L_{n_2} h'_2, c' \cup h'_1 \cup h'_3 \text{ y} \quad (7.2)$$
$$h_3, c \cup h_1 \cup h_2 \approx^L_{n_3} h'_3, c' \cup h'_1 \cup h'_2 \quad (7.3)$$

Ahora aplicando el lema 39 entre 7.1 y 7.2, y entre 7.3 y 7.2, obtenemos que:

$$h_1 \cup h_2, c \cup h_3 \approx^L_{\text{mín}.n_1.n_2} h'_1 \cup h'_2, c' \cup h'_3 \text{ y}$$
$$h_3 \cup h_2, c \cup h_1 \approx^L_{\text{mín}.n_3.n_2} h'_3 \cup h'_2, c' \cup h'_1$$

Por hipotesis inductiva, ya que $\text{mín}.n_1.n_2 \geqslant |p|$, $\text{mín}.n_3.n_2 \geqslant |q|$ y $n_2 \geqslant |r|$, tenemos que

1. $s, h'_1 \cup h'_2 \vDash p$
2. $s, h'_3 \cup h'_2 \vDash q$
3. $s, h'_2 \vDash r$

y $h' = h'_1 \uplus h'_2 \uplus h'_3$, y por lo tanto $s, h' \vDash p \ \langle *\colon r \rangle\ q$.

□

Lema 43 (Propiedad de partición de $\approxeq$). *Sean $n = n_1 + n_2$ y $h, h \in \mathit{Heaps}_D$, tales que $h, c \approxeq^L_n h', c'$ para algunos $c, c' \in \mathit{Heaps}_D$, y $h = h_1 \cup h_2$ para algunos $h_1, h_2 \in \mathit{Heaps}$ con $h_1 \perp h_2$. Entonces existen $h'_1, h'_2 \in \mathit{Heaps}_D$ tales que:*

1. $h_1, h_2 \cup c \approxeq^L_{n_1} h'_1, h'_2 \cup c'$,
2. $h_2, h_1 \cup c \approxeq^L_{n_2} h'_2, h'_1 \cup c'$,
3. $h'_1 \perp h'_2$ *y* $h' = h'_1 \cup h'_2$.

Demostración. Construyamos h'_1 y h'_2 garantizando que se cumplen las condiciones 1 y 2, considerando uno a uno los segmentos de h y h'. Notar que los segmentos en $h \cup c$, son los mismos que aquellos en $h_1 \cup h_2 \cup c$, y lo mismo ocurre con h', h'_1 y h'_2. Como los segmentos son disjuntos y cada registro de memoria pertenece a un segmento (lema 41), por construcción obtenemos que $h' = h'_1 \uplus h'_2$.

Sea $s \subseteq h_1 \cup h_2 \cup c$ un *segmento*$_L$ cualquiera y $s' \subseteq h' \cup c'$ el correspondiente segmento según $\approxeq^L_n$. Sabemos que $|\mathit{dom}.s' \cap \mathit{dom}.h'| \geqslant \text{mín}.|\mathit{dom}.s \cap \mathit{dom}.h|.n$ (*).

Tomemos $\{s'_1, s'_2\}$ una partición de s' y hagamos que $s'_1|_{\mathit{dom}.h'} \subseteq h'_1$ y $s'_2|_{\mathit{dom}.h'} \subseteq h_2$. La partición es tal que sucede alguna de las siguientes situaciones:

(i) Si $|dom.s \cap dom.h| < n$, tomamos

$$|dom.s'_1 \cap dom.h'_1| = |dom.s \cap dom.h_1| \text{ y}$$
$$|dom.s'_2 \cap dom.h'_2| = |dom.s \cap dom.h_2|$$

Esto es posible porque $|dom.s' \cap dom.h| = |dom.s \cap dom.h|$ dada la relación entre h y h'.

(ii) Si $|dom.s \cap dom.h| \geqslant n$ y $|dom.s \cap dom.h_2| \geqslant n_2$, tomamos

$$|dom.s'_1 \cap dom.h'_1| = \text{mín}\,.|dom.s \cap dom.h_1|.n_1$$

Esto es posible por (*) y el hecho que $|dom.s \cap h| \geqslant |dom.s \cap h_1|$ y $n \geqslant n_1$.

(iii) Si $|dom.s \cap dom.h| \geqslant n$ y $|dom.s \cap dom.h_2| < n_2$, tomamos

$$|dom.s'_2 \cap dom.h'_2| = |dom.s \cap dom.h_2|$$

Esto es posible porque $|dom.s \cap h| \geqslant n \geqslant n_2 > |dom.s \cap dom.h_2|$.

Veamos que vale $h_1, h_2 \cup c \approx^L_{n_1} h'_1, h'_2 \cup c'$ en lo que respecta a s y s'. Si se da la situación (i):

- Si $dom.s \cap dom.h_2 \cup c = \emptyset$, entonces $|dom.s \cap dom.h_1| = |dom.s \cap dom.h|$, luego

$$\begin{aligned} |dom.s'_1 \cap dom.h'_1| &= |dom.s \cap dom.h_1| \\ &= |dom.s \cap dom.h| \\ &= |dom.s' \cap dom.h'| \end{aligned}$$

  Luego debe valer $dom.s' \cap dom.h'_2 \cup c' = \emptyset$. Lo recíproco vale por un razonamiento similar.

- Si $|dom.s \cap dom.h_1| < n_1$, por construcción $|dom.s'_1 \cap dom.h'_1| = |dom.s \cap dom.h_1|$.

- Si $|dom.s \cap dom.h_1| \geqslant n_1$, por construcción $|dom.s'_1 \cap dom.h'_1| \geqslant n_1$.

Si se da la situación (ii):

- Consideremos que $dom.s \cap (h_2 \cup c) \neq \emptyset$. Si $dom.s \cap c \neq \emptyset$ entonces vale $dom.s' \cap c' \neq \emptyset$ por la hipótesis. Si $dom.s \cap h_2 \neq \emptyset$, entonces $|dom.s \cap dom.h_2| > 0$, debe valer entonces $n_2 > 0$. Como $|dom.s' \cap dom.h'| \geqslant n_1 + n_2$ y $|dom.s' \cap dom.h'_1| \leqslant n_1$, entonces $|dom.s' \cap dom.h'_2| \geqslant n_2 > 0$. Luego $dom.s \cap h_2 \neq \emptyset$.

- Si $|dom.s \cap dom.h_1| < n_1$, entonces $|dom.s' \cap dom.h'_1| = |dom.s \cap dom.h_1|$.

- Si $|dom.s \cap dom.h_1| \geqslant n_1$, entonces $|dom.s' \cap dom.h_1'| = n1$.

Si se da la situación (iii):

- Sólo tiene sentido considerar el caso que $dom.s \cap h_2\ \emptyset$, puesto que $dom.s \cap c = \emptyset$ si y sólo si $dom.s \cap c = \emptyset$ por hipótesis, y si ese fuera el caso trivialmente se cumple la condición. Si $dom.s \cap dom.h_2 = \emptyset$ entonces $|dom.s \cap dom.h_2| = 0$. Pero $|dom.s \cap dom.h_2| = |dom.s' \cap dom.h_2'|$ y por lo tanto $dom.s' \cap dom.h_2' = \emptyset$. Lo recíproco vale por un razonamiento similar.

- Como $|dom.s \cap dom.h_2| < n_2$ pero $|dom.s \cap dom.h| \geqslant n_1 + n_2$, necesariamente debe valer que $|dom.s \cap dom.h_1| \geqslant n_1$. Siguiendo un razonamiento similar, $|dom.s' \cap dom.h_1'| \geqslant n_1$.

Ahora veamos que vale $h_2, h_1 \cup c \approx^L_{n_1} h_2', h_1' \cup c'$. Si se da la situación (i) la demostración es completamente análoga a la demostración de la situación (i) anterior. Si se da la situación (ii):

- Consideremos si $dom.s \cap dom.h_1 = \emptyset$. Si esto sucede, necesariamente $|dom.s \cap dom.h_1| = 0$, luego $|dom.s' \cap dom.h_1'| = 0$, y por lo tanto $dom.s' \cap dom.h_1' = \emptyset$.

- Sucede que $|dom.s' \cap dom.h_1'| \leqslant n_1$. Como $|dom.s' \cap dom.h'| \geqslant n_1 + n_2$ debe suceder que $|dom.s' \cap dom.h_2'| \geqslant n_2$.

Si se da la situación (iii):

- Si $dom.s \cap dom.h_1 \cup c = \emptyset$, entonces $|dom.s \cap dom.h_2| = |dom.s \cap dom.h|$, luego

$$\begin{aligned} |dom.s_2' \cap dom.h_2'| &= |dom.s \cap dom.h_2| \\ &= |dom.s \cap dom.h| \\ &= |dom.s' \cap dom.h'| \end{aligned}$$

  Luego debe valer $dom.s' \cap dom.h_1' \cup c' = \emptyset$. Lo recíproco vale por un razonamiento similar.

- Sucede $|dom.s \cap h_1| < n_2$, y $|dom.s_2' \cap dom.h'| = |dom.s \cap dom.h_2|$.

□

Lema 44 (Propiedad de unión de $\approx$). *Sean $n_1, n_2 \in \mathbb{N}_0$, $h_1, h_1', h_2, h_2' \in Heaps_D$ tales que $h_1, h_2 \cup c \approx^L_{n_1} h_1', h_2' \cup c'$ y $h_2, h_1 \cup c \approx^L_{n_1} h_2', h_1' \cup c'$, para algunos $c, c' \in Heaps_D$ donde $h_1 \perp h_2$ y $h_1' \perp h_2'$. Entonces si $h = h_1 \cup h_2$ y $h' = h_1' \cup h_2'$*

$$h, c \approx^L_{\text{mín}\, n_1 . n_2} h, c'$$

Demostración. Sea $s \subseteq h_1 \cup h_2 \cup c$ un segmento$_L$, con $s = l \rhd_L l'$. Sea $s' \subseteq h'_1 \cup h'_2 \cup c'$ el segmento correspondiente dado por la hipótesis $h_1, c \cup h_2 \approx^L_{n_1} h'_1, c' \cup h'_2$. Notar que $l \in s'$, y por el lema 41 existe un único segmento en $h'_1 \cup h'_2 \cup c'$ al que l puede pertenecer. Entonces s' es tambien el segmento correspondiente a s según la hipótesis $h_2, c \cup h_1 \approx^L_{n_1} h'_2, c' \cup h'_1$.
Supongamos sin pérdida de generalidad que $n_1 = \text{mín} .n_1.n_2$. Si fuera el otro caso, la demostración es completamente simétrica.

- Si $dom.s \cap dom.c = \emptyset$, entonces debe valer $dom.s \cap dom.h_1 \cup c = \emptyset$. Luego por hipótesis $dom.s' \cap dom.h'_1 \cup c' = \emptyset$, y entonces $dom.s' \cap dom.c' = \emptyset$. Lo recíproco se demuestra análogamente.

- Supongamos $|dom.s \cap dom.h_1 \cup h_2| < n_1 < n_2$. Luego necesariamente $|dom.s \cap dom.h_1| < n_1$ y $|dom.s \cap dom.h_2| < n_2$, y por lo tanto por hipótesis $|dom.s' \cap dom.h'_1| = |dom.s \cap dom.h_1|$ y $|dom.s' \cap dom.h'_2| = |dom.s \cap dom.h_2|$.

$$\begin{aligned} |dom.s \cap dom.h_1 \cup h_2| &= |dom.s \cap dom.h_1| + |dom.s \cap dom.h_2| \\ &= |dom.s' \cap dom.h'_1| + |dom.s' \cap dom.h'_2| \\ &= |dom.s' \cap dom.h'_1 \cup h'_2| \end{aligned}$$

- Supongamos $|dom.s \cap dom.h_1 \cup h_2| \geqslant n_1$. Puede darse que $|dom.s \cap dom.h_1| < n_1$ o $|dom.s \cap dom.h_1| \geqslant n_1$, y otro tanto para h_2. Pero sabemos que

$$\begin{aligned} &|dom.s' \cap dom.h'_1| \geqslant \text{mín} .|dom.s \cap dom.h_1|.n_1 \text{ y} \\ &|dom.s' \cap dom.h'_2| \geqslant \text{mín} .|dom.s \cap dom.h_2|.n_1 \end{aligned}$$

En cualquiera de los casos posibles

$$|dom.s' \cap dom.h'_1| + |dom.s' \cap dom.h'_2| \geqslant n_1$$

□

Lema 45 (Preservación de satifacción de $\approx$). *Sean $p \in Form_D$, $X \supseteq FV.p$, $n \in \mathbb{N}_0$, $L \in Loc$, s, h y $s, h' \in States_D$, tales que $L \supseteq s^{\upharpoonright}.X$, $n \geqslant |p|$ y $h, c \approx^L_n h', c'$, para algunos $c, c' \in Heaps_D$ donde $h \perp c$ y $h' \perp c'$. Entonces*

$$s, h \vDash p \text{ si y sólo si } s, h' \vDash p$$

Demostración. La demostración es por inducción en la fórmula p. Para una fórmula de la forma $e_1 = e_2$, el lema es trivial ya que se satisface en cualquier *heap*. Lo mismo ocurre con una fórmula como **emp**, ya que $h = h' = \emptyset$. Para $x \mapsto y$, ocurre que $h = \{(s.x, s.y)\}$ y el único registro es el único segmento$_L$. Como $|h| < n$, entonces debe valer $h = h'$ y el lema se satiface. El caso interesante son las fórmulas del tipo

lseg.$x.y$, que son satisfechas cuando h contiene un único segmento$_l$ $s.x \rhd_L s.y$ de longitud arbitraria. Es elemental ver que cualquier otro $s.x \rhd_L s.y$ satisface la fórmula. Para los casos de la forma $p \vee q$ y $\neg p$, la demostración sale directamente por hipótesis inductiva. La demostración del caso de la forma $p \ \langle *\!: r \rangle \ q$ es completamente análoga a aquella del lema 40. □

Teorema 5. *Sean* $X \subseteq Var$ *un conjunto finito,* $n \in \mathbb{N}_0$ *y* $Ord \in Loc \to \mathbb{N}_0$ *biyectiva tal que* $Ord.nil = 0$*:*

1. *Para cada* $s, h \in States_D$ *existe* $s', h' \in Norm^{X,n}_{Ord}$ *tal que* $s, h \cong^X_n s', h'$.
2. *Si* $s, h \in Norm^{X,n}_{Ord}$*, entonces* $|dom.h| \leqslant 2 * |X| * n$.
3. $|Norm^{X,n}_{Ord}| \leqslant (N * (N-1)/2)^{|X|}$*, donde* $N = 2 * |X| * n$.

Demostración.

1. Supongamos que $s, h \notin Norm^{X,n}_{Ord}$. Supongamos además que esto se debe a que $|dom.h - reach_L.h| > n$. Es fácil ver que podemos construir un h' similar a h pero con una menor cantidad de registros basura, tal que $s, h \approx^X_n s, h'$.

   Supongamos ahora que algún segmento$_L$ $\varsigma \subseteq h$, no cumple que $|dom.\varsigma| \leqslant n$; como en el caso anterior, es fácil construir un h' similar a h' pero reemplazando ς por ς' que cumple $|dom.\varsigma'| \leqslant n$. Así, trivialmente vale que $h \approx^X_n h'$.

   Finalmente, supongamos que $s.x \neq nil$ para alguna $x \notin X$, o bien $L \cup dom.h \cup img.h \nsubseteq \{l \in Loc \mid Ord.l \leqslant m\}$ (donde $m = |L \cup dom.h \cup img.h|$). La relación $\sim^X$ nos permite *elegir* arbitrariamente los valores que asigna s a cualquier variable que no ocurra en X, y las direcciones de memoria de h, obteniendo s', h' tales que $s, h \sim^X s', h'$.

2. Consideremos la porción de h *alcanzable* desde X. La cantidad de segmentos que tal porción incluye, en el peor de los casos, es igual a la cantidad de nodos de un árbol binario con $|X|$ hojas, esto es, $2 * |X| - 1$. Como cada segmento puede tener hasta n registros, el número de registros total de la parte alcanzable es igual a $2 * |X| * n - n$. Pero además, pueden existir hasta n registros *basura* en la porción no alcanzable. Por lo tanto, $|dom.h| \leqslant 2 * |X| * n$.

3. La cota para el tamaño del conjunto de modelos normales se deduce a partir de una combinatoria elemental de la cantidad de *heaps* con hasta $2 * |X| * n$ registros en conjunto con la cantidad de *stacks* que satisfacen las condiciones de *normalidad*.

□

Resultados del capítulo 6

La completitud de los reticulados (lemas 48 y 50) es directa ya que en ambos casos se definen en términos del conjunto de partes.

Resulta sencillo ver que las reglas de reescritura son consistentes (lemas 52 y 54) ya que todas son implicaciones elementales de fórmulas de *Separation logic*, cuya validez se desprende directamente de su semántica. Lo mismo aplica para el *theorem prover* sintáctico (lema 54). Es fácil ver que la aplicación iterativa de estas reglas es normalizante (lema 51 y teorema 53) dado que el resultado es *más débil* semánticamente, lo que elimina progresivamente la validez de las condiciones para de aplicación de cada regla.

Los aspectos más relevantes sobre la terminación del análisis se discuten en el cuerpo del trabajo (teorema 53). Teniendo en cuenta este resultado, y la consistencia de las reglas de reescritura, la consistencia entre las diferentes semánticas es directa (teoremas 7 y 8).

Bibliografía

[1] GNU LibAVL, `http://www.stanford.edu/~blp/avl/`.

[2] Martín Abadi and K. Rustan M. Leino. A logic of object-oriented programs. In Nachum Dershowitz, editor, *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, pages 11–41. Springer, 2003.

[3] Timos Antonopoulos, Nikos Gorogiannis, Christoph Haase, Max Kanovich, and Joël Ouaknine. Foundations for decision problems in separation logic with general inductive predicates. In Anca Muscholl, editor, *Foundations of Software Science and Computation Structures*, volume 8412 of *Lecture Notes in Computer Science*, pages 411–425. Springer Berlin Heidelberg, 2014.

[4] Ralph-Johan J. Back, Abo Akademi, and J. Von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1998.

[5] Roland C. Backhouse. *Program construction and verification*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

[6] Kshitij Bansal, Rémi Brochenin, and Étienne Lozes. Beyond shapes: Lists with ordered data. In Luca de Alfaro, editor, *Foundations of Software Science and Computational Structures, 12th International Conference, FOSSACS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5504 of *Lecture Notes in Computer Science*, pages 425–439. Springer, 2009.

[7] Michael Barnett and David A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *MPC*, pages 54–84, 2004.

[8] Jon Barwise. Mathematical Proofs of Computer System Correctness. *Notices of the American Mathematical Society*, 36(7):844–851, 1989.

[9] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O'Hearn, Thomas Wies, and Hongseok Yang. Shape analysis for composite data structures. In *Proceedings of the 19th International Conference on Computer Aided Verification*, CAV'07, pages 178–192, Berlin, Heidelberg, 2007. Springer-Verlag.

[10] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. A decidable fragment of separation logic. In *Proceedings of the 24th International Conference on Foundations of Software Technology and Theoretical Computer Science*, FSTTCS'04, pages 97–109, Berlin, Heidelberg, 2004. Springer-Verlag.

[11] Josh Berdine, Cristiano Calcagno, and PeterW. O'Hearn. Symbolic execution with separation logic. In Kwangkeun Yi, editor, *Programming Languages and Systems*, volume 3780 of *Lecture Notes in Computer Science*, pages 52–68. Springer Berlin Heidelberg, 2005.

[12] Josh Berdine, Cristiano Calcagno, and PeterW. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In FrankS. de Boer, MarcelloM. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer Berlin Heidelberg, 2006.

[13] Bodil Biering, Lars Birkedal, and Noah Torp-Smith. Bi-hyperdoctrines, higher-order separation logic, and abstraction. *ACM Trans. Program. Lang. Syst.*, 29(5):24, 2007.

[14] Richard S. Bird. Unfolding pointer algorithms. *J. Funct. Program.*, 11(3):347–358, 2001.

[15] Lars Birkedal, Noah Torp-Smith, and John C. Reynolds. Local reasoning about a copying garbage collector. *SIGPLAN Not.*, 39(1):220–231, 2004.

[16] Patrick Blackburn and Balder Ten Cate. Pure extensions, proof rules and hybrid axiomatics. In *Preliminary proceedings of Advances in Modal Logic (AiML 2004*, pages 277–322, 2004.

[17] Javier Blanco, Renato Cherini, Martin Diller, and Pio Garcia. Intepreters as computational mechanisms. In *8th Conference on Computing and Philosophy (ECAP)*, october 2010.

[18] Javier Blanco and Pio Garcia. A categorial mistake in the formal verification debate. In *European Conference on Computing and Philosophy (ECAP)*, June 2008.

[19] Javier Blanco, Pío García, Renato Cherini, and Martin Diller. A behavioral characterization of computational systems. In Charles Ess and Ruth Hagengruber, editors, *Proceedings IACAP 2011 - First International Conference of IACAP: The Computational Turn: Past, Presents, Futures? 4 – 6 July, 2011, Aarhus University*, MV-Wissenschaft. Verlagshaus Monsenstein und Vannerdat OHG, 2011.

[20] Javier Blanco, Pío Garcíaa, and Renato Cherini. Convergencias y divergencias en la noción de computación. *Revista iberoamericana de ciencia tecnologíaa y sociedad*, 7:111 – 121, 12 2012.

[21] 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.

[22] Paulo Borba, Augusto Sampaio, and Márcio Cornélio. A refinement algebra for object-oriented programming. In Luca Cardelli, editor, *ECOOP*, volume 2743 of *Lecture Notes in Computer Science*, pages 457–482. Springer, 2003.

[23] Richard Bornat. Proving pointer programs in hoare logic. In Roland Carl Backhouse and José Nuno Oliveira, editors, *MPC*, volume 1837 of *Lecture Notes in Computer Science*, pages 102–126. Springer, 2000.

[24] Richard Bornat. List reversal: back into the frying pan, 2006.

[25] Richard Bornat and Cristiano Calcagno. Local reasoning, separation, and aliasing. In *In Proc. 2nd workshop on Semantics, Program Analysis and Computing Environments for Memory Management*, 2004.

[26] Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. Permission accounting in separation logic. In Jens Palsberg and Martín Abadi, editors, *POPL*, pages 259–270. ACM, 2005.

[27] Ahmed Bouajjani, Marius Bozga, Peter Habermehl, Radu Iosif, Pierre Moro, and Tomáš Vojnar. Programs with lists are counter automata. In Thomas Ball and RobertB. Jones, editors, *Computer Aided Verification*, volume 4144 of *Lecture Notes in Computer Science*, pages 517–531. Springer Berlin Heidelberg, 2006.

[28] Ahmed Bouajjani, Cezara Dr goi, Constantin Enea, and Mihaela Sighireanu. Accurate invariant checking for programs manipulating lists and arrays with infinite data. In Supratik Chakraborty and Madhavan Mukund, editors, *Automated Technology for Verification and Analysis*, Lecture Notes in Computer Science, pages 167–182. Springer Berlin Heidelberg, 2012.

[29] John Boyland. Checking interference with fractional permissions. In Radhia Cousot, editor, *SAS*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer, 2003.

[30] Rémi Brochenin, Stéphane Demri, and Etienne Lozes. On the almighty wand. *Inf. Comput.*, 211:106–137, February 2012.

[31] James Brotherston, Carsten Fuhs, Juan A. Navarro Pérez, and Nikos Gorogiannis. A decision procedure for satisfiability in separation logic with inductive predicates. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, CSL-LICS '14, pages 25:1–25:10, New York, NY, USA, 2014. ACM.

[32] James Brotherston and Jules Villard. Parametric completeness for separation theories. *SIGPLAN Not.*, 49(1):453–464, January 2014.

[33] R. M. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, (7):23–50, 1972.

[34] Michael J. Butler. Calculational derivation of pointer algorithms from tree operations. *Sci. Comput. Program.*, 33(3):221–260, 1999.

[35] Cristiano Calcagno and Dino Distefano. Infer: An automatic program verifier for memory safety of c programs. In Mihaela Bobaru, Klaus Havelund, GerardJ. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 459–465. Springer Berlin Heidelberg, 2011.

[36] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In Kwangkeun Yi, editor, *SAS*, volume 4134 of *LNCS*, pages 182–203. Springer, 2006.

[37] 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.

[38] Cristiano Calcagno, Dino Distefano, and Viktor Vafeiadis. Bi-abductive resource invariant synthesis. In Zhenjiang Hu, editor, *APLAS*, volume 5904 of *LNCS*, pages 259–274. Springer, 2009.

[39] Cristiano Calcagno, Philippa Gardner, and Matthew Hague. From separation logic to first-order logic. In Vladimiro Sassone, editor, *FoSSaCS*, volume 3441 of *Lecture Notes in Computer Science*, pages 395–409. Springer, 2005.

[40] Cristiano Calcagno, Philippa Gardner, and Uri Zarfaty. Context logic as modal logic: completeness and parametric inexpressivity. In Martin Hofmann and Matthias Felleisen, editors, *POPL*, pages 123–134. ACM, 2007.

[41] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *IN PROC. 22ND ANNUAL IEEE SYMPOSIUM ON LOGIC IN COMPUTER SCIENCE (LICS'07*, pages 366–378, 2007.

[42] Cristiano Calcagno, Matthew J. Parkinson, and Viktor Vafeiadis. Modular safety checking for fine-grained concurrency. In Hanne Riis Nielson and Gilberto Filé, editors, *SAS*, volume 4634 of *LNCS*, pages 233–248. Springer, 2007.

[43] Cristiano Calcagno, Hongseok Yang, and PeterW. O'Hearn. Computability and complexity results for a spatial assertion language for data structures. In Ramesh Hariharan, V. Vinay, and Madhavan Mukund, editors, *FST TCS 2001: Foundations of Software Technology and Theoretical Computer Science*, volume 2245 of *Lecture Notes in Computer Science*, pages 108–119. Springer Berlin Heidelberg, 2001.

[44] Ana Cavalcanti and David A. Naumann. A weakest precondition semantics for refinement of object-oriented programs. *IEEE Trans. Software Eng.*, 26(8):713–728, 2000.

[45] D. Chalmers. Does a rock implement every finite-state automaton, 1996.

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

[47] 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.

[48] Renato Cherini. Construcción de pogramas que manejan dinámicamente la memoria. Facultad de Matemática, Astronomía y Física, Universidad Nacional de Córdoba, 2006. Trabajo final de grado.

[49] Renato Cherini and Javier O. Blanco. Local reasoning for abstraction and sharing. In Sung Y. Shin and Sascha Ossowski, editors, *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC), Honolulu, Hawaii, USA, March 9-12, 2009*, pages 552–557. ACM, 2009.

[50] Renato Cherini, Lucas Rearte, and Javier O. Blanco. A shape analysis for non-linear data structures. In Radhia Cousot and Matthieu Martel, editors, *Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings*, volume 6337 of *Lecture Notes in Computer Science*, pages 201–217. Springer, 2010.

[51] Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Enhancing modular oo verification with separation logic. *SIGPLAN Not.*, 43(1):87–99, 2008.

[52] Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Sci. Comput. Program.*, 77(9):1006–1036, August 2012.

[53] Ronald L. Chrisley. Why everything doesn't realize every computation. *Minds and Machines*, 4(4):403–20, 1994.

[54] Duc-Hiep Chu and Joxan Jaffar. Separation logic with first-class heaps and a new frame rule (draft).

[55] Byron Cook, Christoph Haase, Joël Ouaknine, Matthew Parkinson, and James Worrell. Tractable reasoning in a fragment of separation logic. In Joost-Pieter Katoen and Barbara König, editors, *CONCUR 2011 – Concurrency Theory*, volume 6901 of *Lecture Notes in Computer Science*, pages 235–249. Springer Berlin Heidelberg, 2011.

[56] B. Jack Copeland. What is computation? *Synthese*, 108(3):335–59, 1996.

[57] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[58] Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Commun. ACM*, 22(5):271–280, May 1979.

[59] Stéphane Demri and Morgan Deters. Two-variable separation logic and its inner circle. *ACM Transactions on Computational Logic*, 2015. To appear.

[60] Stéphane Demri, Didier Galmiche, Dominique Larchey-Wendling, and Daniel Méry. Separation logic with one quantified variable. In Jean-Éric Pin, editor, *Proceedings of the 9th International Computer Science Symposium in Russia (CSR'14)*, volume 8476 of *Lecture Notes in Computer Science*, pages 125–138, Moscow, Russia, June 2014. Springer.

[61] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[62] Dino Distefano. Attacking large industrial code with bi-abductive inference. In María Alpuente, Byron Cook, and Christophe Joubert, editors, *Formal Methods for Industrial Critical Systems*, volume 5825 of *Lecture Notes in Computer Science*, pages 1–8. Springer Berlin Heidelberg, 2009.

[63] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'06, pages 287–302, Berlin, Heidelberg, 2006. Springer-Verlag.

[64] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In Holger Hermanns and Jens Palsberg, editors, *TACAS*, volume 3920 of *LNCS*, pages 287–302. Springer, 2006.

[65] Dino Distefano and Matthew J. Parkinson J. jstar: Towards practical verification for java. *SIGPLAN Not.*, 43(10):213–226, October 2008.

[66] GregoryJ. Duck, Joxan Jaffar, and NicolasC.H. Koh. Constraint-based program reasoning with heaps and separation. In Christian Schulte, editor, *Principles and Practice of Constraint Programming*, volume 8124 of *Lecture Notes in Computer Science*, pages 282–298. Springer Berlin Heidelberg, 2013.

[67] Kamil Dudka, Petr Peringer, and Tomáš Vojnar. Predator: A practical tool for checking manipulation of dynamic data structures using separation logic. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 372–378. Springer Berlin Heidelberg, 2011.

[68] Amnon H. Eden. Three paradigms of computer science. *Minds and Machines*, 17(2):135–167, 2007.

[69] Amnon H. Eden and Raymond Turner. Problems in the ontology of computer programs. *Applied Ontology*, 2(1):13–36, 2007.

[70] Thorsten Ehm. Transformational construction of correct pointer algorithms. In Dines Bjørner, Manfred Broy, and Alexandre V. Zamulin, editors, *Ershov Memorial Conference*, volume 2244 of *Lecture Notes in Computer Science*, pages 116–130. Springer, 2001.

[71] 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.

[72] James H. Fetzer. Program verification: the very idea. *Commun. ACM*, 31(9):1048–1063, 1988.

[73] R. Gandy. Church's thesis and principles for mechanisms. In K. J. Barwise, H. J. Keisler, and K. Kunen, editors, *The Kleene Symposium*, volume 101, pages 123–148, 1978.

[74] 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.

[75] Alexey Gotsman, Josh Berdine, Byron Cook, and Mooly Sagiv. Thread-modular shape analysis. In Jeanne Ferrante and Kathryn S. McKinley, editors, *ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 266–277. ACM, 2007.

[76] David Gries. *The Science of Programming*. Springer, 1981.

[77] Christian Haack and Clément Hurlin. Separation logic contracts for a Java-like language with fork/join. In Jose Meseguer and Grigore Rosu, editors, *International Conference on Algebraic Methodology and Software Technology (AMAST)*, number 5140 in Lecture Notes in Computer Science, pages 199–215, Urbana, Illinois, USA, July 2008. Springer-Verlag.

[78] Christoph Haase, Samin Ishtiaq, Joël Ouaknine, and Matthew J. Parkinson. Seloger: A tool for graph-based reasoning in separation logic. In Sharygina and Veith [141], pages 790–795.

[79] 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.

[80] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[81] Aquinas Hobor and Jules Villard. The ramifications of sharing in data structures. *SIGPLAN Not.*, 48(1):523–536, January 2013.

[82] John Hogg. Islands: aliasing protection in object-oriented languages. *SIGPLAN Not.*, 26(11):271–285, 1991.

[83] John Hogg, Doug Lea, Alan Wills, Dennis deChampeaux, and Richard Holt. The geneva convention on the treatment of object aliasing. *SIGPLAN OOPS Mess.*, 3(2):11–16, 1992.

[84] Radu Iosif, Adam Rogalewicz, and Jiri Simacek. The tree width of separation logic with recursive definitions. In MariaPaola Bonacina, editor, *Automated Deduction – CADE-24*, volume 7898 of *Lecture Notes in Computer Science*, pages 21–38. Springer Berlin Heidelberg, 2013.

[85] Samin S. Ishtiaq and Peter W. O'Hearn. Bi as an assertion language for mutable data structures. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 14–26, New York, NY, USA, 2001. ACM Press.

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

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

[88] Jakob L. Jensen, Michael E. Jørgensen, Michael I. Schwartzbach, and Nils Klarlund. Automatic verification of pointer programs using monadic second-order logic. *SIGPLAN Not.*, 32(5):226–234, May 1997.

[89] 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.

[90] Neil D. Jones. *Computability and complexity: from a programming perspective*. MIT Press, Cambridge, MA, USA, 1997.

[91] N. Krishnaswami, L. Birkedal, J. Aldrich, and J. Reynolds. Idealized ml and its separation logic, 2006.

[92] R. Apt Krzysztof and Ernst-Rüdiger Olderog. *Verification of Sequential and Concurrent Programs*. Springer-Verlag New York, Inc., New York, NY, USA, 1991.

[93] Thomas S. Kuhn. *The structure of scientific revolutions*. University of Chicago Press, Chicago, 1970.

[94] Viktor Kuncak and Martin C. Rinard. On spatial conjunction as second-order logic. *CoRR*, cs.LO/0410073, 2004.

[95] William Landi and Barbara G. Ryder. Pointer-induced aliasing: a problem taxonomy. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 93–103, New York, NY, USA, 1991. ACM.

[96] 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.

[97] K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using data groups to specify and check side effects. In *PLDI*, pages 246–257, 2002.

[98] Tal Lev-Ami and Mooly Sagiv. Tvla: A system for implementing static analyses. In Jens Palsberg, editor, *Static Analysis*, volume 1824 of *Lecture Notes in Computer Science*, pages 280–301. Springer Berlin Heidelberg, 2000.

[99] 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.

[100] 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.

[101] Davis Martin and Weyuker Elaine J. *Computability, complexity, and languages : fundamentals of theoretical computer science*. Academic Press, New York, 1983.

[102] J. Mccarthy. Towards a mathematical science of computation. In *In IFIP Congress*, pages 21–28. North-Holland, 1962.

[103] Ronald Middelkoop, Kees Huizing, and Ruurd Kuiper. A separation logic proof system for a class-based language. In *Proceedings of the Workshop on Logics for Resources, Processes and Programs (LRPP)*, 2004.

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

[105] Bernhard Möller. Calculating with pointer structures. In *Proceedings of the IFIP TC 2 WG 2.1 international workshop on Algorithmic languages and calculi*, pages 24–48, London, UK, UK, 1997. Chapman & Hall, Ltd.

[106] James H. Moor. Three myths of computer science. *British Journal for the Philosophy of Science*, 29(3):213–222, 1978.

[107] J. M. Morris. A general axiom of assignment. assignment and linked data structure. a proof of the schorr-waite algorithm. In M. Broy and G. Schmidt, editors, *Theoretical Foundations of Programming Methodology*, pages 25–51, 1981.

[108] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular invariants for layered object structures. *Sci. Comput. Program.*, 62(3):253–286, 2006.

[109] David A. Naumann and Michael Barnett. Towards imperative modules: Reasoning about invariants and sharing of mutable state. *Theor. Comput. Sci.*, 365(1-2):143–168, 2006.

[110] George C. Necula and Philip Wadler, editors. *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*. ACM, 2008.

[111] 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.

[112] HuuHai 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, *Verification, Model Checking, and Abstract Interpretation*, volume 4349 of *Lecture Notes in Computer Science*, pages 251–266. Springer Berlin Heidelberg, 2007.

[113] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[114] Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.

[115] Peter W. O'Hearn and David J. Pym. The logic of bunched implications. *BULLETIN OF SYMBOLIC LOGIC*, 5(2):215–244, 1999.

[116] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In Laurent Fribourg, editor, *CSL*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001.

[117] Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. In Neil D. Jones and Xavier Leroy, editors, *POPL*, pages 268–280. ACM, 2004.

[118] Richard F. Paige and Jonathan S. Ostroff. Erc - an object-oriented refinement calculus for eiffel. *Formal Asp. Comput.*, 16(1):51–79, 2004.

[119] Matthew J. Parkinson. Local reasoning for Java. Technical Report UCAM-CL-TR-654, University of Cambridge, Computer Laboratory, November 2005.

[120] Matthew J. Parkinson and Gavin M. Bierman. Separation logic and abstraction. In Jens Palsberg and Martín Abadi, editors, *POPL*, pages 247–258. ACM, 2005.

[121] Matthew J. Parkinson and Gavin M. Bierman. Separation logic, abstraction and inheritance. In Necula and Wadler [110], pages 75–86.

[122] Matthew J. Parkinson and Gavin M. Bierman. Separation logic, abstraction and inheritance. In Necula and Wadler [110], pages 75–86.

[123] David L. Parnas. Software pioneers. chapter The Secret History of Information Hiding, pages 399–409. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[124] Juan Antonio Navarro Pérez and Andrey Rybalchenko. Separation logic modulo theories. In Chung-chieh Shan, editor, *Programming Languages and Systems - 11th Asian Symposium, APLAS 2013, Melbourne, VIC, Australia, December 9-11, 2013. Proceedings*, volume 8301 of *Lecture Notes in Computer Science*, pages 90–106. Springer, 2013.

[125] Gualtiero Piccinini. Computing mechanisms. *Philosophy of Science*, 74(4), 2007.

[126] Gualtiero Piccinini. Computers. *Pacific Philosophical Quarterly*, 89(1):32–73, 2008.

[127] Gualtiero Piccinini. Computation in physical systems. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Summer 2015 edition, 2015.

[128] Ruzica Piskac, Thomas Wies, and Damien Zufferey. Automating separation logic using SMT. In Sharygina and Veith [141], pages 773–789.

[129] Ruzica Piskac, Thomas Wies, and Damien Zufferey. Automating separation logic with trees and data. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, volume 8559 of *Lecture Notes in Computer Science*, pages 711–728. Springer International Publishing, 2014.

[130] Hilary Putnam. *Representation and Reality*. MIT Press, Cambridge, MA, USA, 1988.

[131] D.J. Pym. *The Semantics and Proof Theory of the Logic of Bunched Implications*. Applied Logic Series. Springer, 2002.

[132] Juan Antonio Navarro Pérez and Andrey Rybalchenko. Separation logic + superposition calculus = heap theorem prover. In Mary W. Hall and David A. Padua, editors, *PLDI*, pages 556–566. ACM, 2011.

[133] J. Reynolds. Precise, intuitionistic, and supported assertions in separation logic, 2005.

[134] John C. Reynolds. A short course in separation logic.

[135] John C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Millennial Perspectives in Computer Science*, pages 303–321. Palgrave, 2000.

[136] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.

[137] 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.

[138] Matthias Scheutz. When physical systems realize functions. *Minds and Machines*, 9(2):161–196, 1999.

[139] John R. Searle. Is the brain a digital computer? *Proceedings and Addresses of the American Philosophical Association*, 64(November):21–37, 1990.

[140] Asankhaya Sharma, Aquinas Hobor, and Wei-Ngan Chin. Specifying Compatible Sharing in Data Structures. Technical report, National University of Singapore, Department of Computer Science.

[141] Natasha Sharygina and Helmut Veith, editors. *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*. Springer, 2013.

[142] Jan Smans, Bart Jacobs, and Fran Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In Sophia Drossopoulou, editor, *ECOOP*, volume 5653 of *LNCS*. Springer-Verlag, 2009.

[143] Alan Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(1):230–265, 1936.

[144] David Turner. Elementary strong functional programming. In Pieter H. Hartel and Marinus J. Plasmeijer, editors, *Functional Programming Languages in Education, First International Symposium, FPLE'95, Nijmegen, The Netherlands, December 4-6, 1995, Proceedings*, volume 1022 of *Lecture Notes in Computer Science*. Springer, 1995.

[145] Raymond Turner. The philosophy of computer science. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Winter 2014 edition, 2014.

[146] Viktor Vafeiadis and Matthew J. Parkinson. A marriage of rely/guarantee and separation logic. In Luís Caires and Vasco Thudichum Vasconcelos, editors, *CONCUR*, volume 4703 of *Lecture Notes in Computer Science*, pages 256–271. Springer, 2007.

[147] Stephan van Staden and Cristiano Calcagno. Reasoning about multiple related abstractions with multistar. *SIGPLAN Not.*, 45(10):504–519, October 2010.

[148] Jules Villard, Étienne Lozes, and Cristiano Calcagno. Proving copyless message passing. In Zhenjiang Hu, editor, *APLAS*, volume 5904 of *LNCS*. Springer.

[149] Wikipedia. List of software bugs — wikipedia, the free encyclopedia, 2015.

[150] Qiwen Xu, Willem P. de Roever, and Jifeng He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Asp. Comput.*, 9(2):149–174, 1997.

[151] H. Yang. An example of local reasoning in bi pointer logic: the schorr-waite graph marking algorithm, 2000.

[152] Hongseok Yang. *Local reasoning for stateful programs*. PhD thesis, Champaign, IL, USA, 2001. Adviser-Uday S. Reddy.

[153] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter O'Hearn. Scalable shape analysis for systems code. In *Proceedings of the 20th International Conference on Computer Aided Verification*, CAV '08, pages 385–398, Berlin, Heidelberg, 2008. Springer-Verlag.

[154] 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.

[155] Hongseok Yang and Peter W. O'Hearn. A semantic basis for local reasoning. In Mogens Nielsen and Uffe Engberg, editors, *FoSSaCS*, volume 2303 of *Lecture Notes in Computer Science*, pages 402–416. Springer, 2002.