UNIVERSIDAD NACIONAL DE CÓRDOBA

FACULTAD DE MATEMÁTICA, ASTRONOMÍA Y FÍSICA

SERIE "A"

TRABAJOS DE INFORMÁTICA

Nº 1/ 08

Razonamiento local para abstracción y Estructuras compartidas

Renato Cherini - Javier O. Blanco

Editores: Pedro R. D'argenio – Gabriel Infante López

CIUDAD UNIVERSITARIA – 5000 CÓRDOBA

REPÚBLICA ARGENTINA

Razonamiento local para abstraccion y estructuras compartidas

Renato Cherini
Fa.M.A.F - U.N.C
cr@hal.famaf.unc.edu.ar

Javier O. Blanco
Fa.M.A.F - U.N.C
blanco@mate.uncor.edu

Resumen

El *razonamiento local* posibilitado por la Separation Logic ha demostrado ser una gran herramienta para la verificación de programas con un manejo complejo de punteros. Sin embargo el poder de esta lógica encuentra su limite en situaciones en las que es necesario especificar diversas estructuras que comparten el *heap*, ya sea por la dificultad de especificarlas de manera disjunta, ya sea por la imposibilidad de hacerlo sin romper las abstracciones que estas estructuras proveen. En el presente articulo presentamos una generalización de la Separation Logic que permite especificar precisamente estructuras complejas en el *heap*, relaciones de *sharing* entre ellas, y un sistema de prueba composicional asociado para verificar programas de forma modular bajo ciertas condiciones, aun cuando no es posible garantizar una completa separación espacial de las estructuras manipuladas.

1. Introducción

La principal ventaja de la Separation Logic ([21, 10, 27]) frente a otros formalismos para el razonamiento sobre programas que manejan dinámicamente la memoria, es que su lenguaje de aserciones junto a su sistema de prueba promueven el *razonamiento local* ([15, 22]). Es posible demostrar un programa C respecto a una precondición P y postcondición Q, denotado por $\{P\}\,C\,\{Q\}$, mencionando única y explícitamente en P y Q la región de memoria manipulada por C, su *footprint*. Luego utilizando la *Regla de Frame* (figura 1), es posible extender la corrección del programa a un contexto de memoria mas amplio especificado por un invariante I, deduciendo $\{P * I\}\,C\,\{Q * I\}$. Aquí la *conjunción espacial* $*$ asegura que la porción de memoria satisfaciendo I es completamente disjunta del footprint de C.

La Separation Logic ha sido utilizada con mucho éxito en la demostración de programas que requieren una manipulación compleja de punteros ([25, 4]), incluso en programas con ciertas formas restringidas de *aliasing*. La introducción del operador $*$ en el lenguaje de aserciones permite codificar la hipótesis generalmente común de inexistencia de alias. A partir de allí el razonamiento local aparece como la forma natural de razonar sobre programas especificados con esta lógica. Sin embargo esta ventaja tiene su costo: la *Regla de Constancia* (figura 1) ya no es válida en este marco. El uso del operador $\wedge$ en una formula como $P \wedge I$ introduce potencialmente múltiples alias, ya que especifica que P y I se satisfacen en el *mismo heap*. De esta manera, mientras que el sistema deductivo de la Separation Logic presenta una clara ventaja para razonar cuando es posible garantizar la separación espacial, no provee herramientas para una deducción sistemática cuando esto no ocurre, obligando a proceder en estas circunstancias de manera *ad hoc*.

$$\frac{\{P\}\,C\,\{Q\}}{\{P * I\}\,C\,\{Q * I\}} \qquad\qquad \frac{\{P\}\,C\,\{Q\}}{\{P \wedge I\}\,C\,\{Q \wedge I\}}$$

cuando ninguna variable libre de I es modificada por C

Figura 1: izq. *Regla de Frame*, der. *Regla de Constancia*

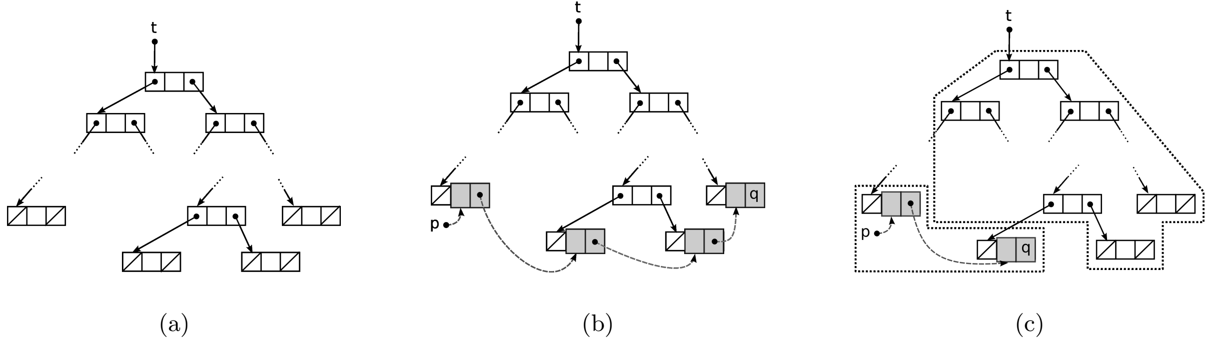

Figura 2: (a) Arbol binario, (b) Arbol binario con fringe, (c) Estructuras en un estado intermedio

La necesidad de especificar múltiples estructuras que comparten el *heap* para su representación se presenta en diversos escenarios. Tomemos por ejemplo el problema del *fringe* presentado en [22]. El *fringe* de un árbol consiste en la lista de las hojas del mismo. El objetivo es construir una lista enlazada que represente el *fringe* utilizando para ello celdas de memoria de los nodos hoja, que no son relevantes para la representación del árbol. En la figura 2 (b) se presenta el resultado esperado de calcular el *fringe* en el árbol de la figura 2 (a).

Sin entrar en detalles, supongamos que el predicado $\mathbf{lseg}.p.q.xs$ representa una segmento de lista enlazada xs entre los punteros p y q. De la misma manera $\mathbf{tree}.t.bt$ representa un árbol binario bt referenciado por t. El objetivo de calcular el fringe de un árbol bt esta especificado originalmente en [22] como:

$$(\mathbf{lseg}.p.q.(fringe.bt) * \mathbf{true}) \wedge \mathbf{tree}.t.bt$$

El patrón de *inclusion sharing* $(P * \mathbf{true}) \wedge Q$ especifica que el *heap* satisfaciendo P se encuentra incluido (o es igual a) el *heap* que satisface Q. A pesar de la simplicidad de esta especificación y del programa recursivo demostrado en [22], el razonamiento local encuentra su limite al intentar demostrar un programa iterativo a partir de esta especificación. Mas aun, dado que se trata de estructuras definidas inductivamente sobre distintos parámetros, no resulta en absoluto trivial especificar de manera disjunta las estructuras parciales que ocurren en los estados intermedios de una computación iterativa.

Un segundo escenario se da en un ámbito que si bien esta mas allá de lo originalmente abarcado por la Separation Logic, ha sido objeto de atención en muchas extensiones propuestas en trabajos recientes ([16, 18, 17, 11, 3]). La modularización y la abstracción son dos formas muy utilizadas para lidiar con la complejidad de programas de tamaño real. Esto se logra normalmente utilizando técnicas de *tipos abstractos de datos* o el paradigma de *orientación a objetos*. En este tipo de programa es común que dos o mas estructuras 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 *representation exposure*. Las distintas estructuras sobre un *heap* representan múltiples vistas que seria deseable poder manipular independientemente.

Imaginemos la situación en la cual disponemos de dos iteradores $\mathbf{iter}.i_1.c.xs$ y $\mathbf{iter}.i_2.c.xs$ sobre una colección mutable especificada por $\mathbf{coll}.c.xs$. Ya que cada iterador necesita memoria extra para su representación, y siguiendo el patrón de *inclusión sharing*, una formula que define esta situación es la siguiente:

$$(\mathbf{iter}.i_1.c.xs * \mathbf{true}) \wedge (\mathbf{coll}.c.xs * \mathbf{true}) \wedge (\mathbf{iter}.i_2.c.xs * \mathbf{true})$$

El principio de *information hiding* nos inhibe de utilizar las definiciones de los predicados para dar otra especificación radicalmente diferente, en términos de formulas disjuntas combinadas con $*$. Pero una especificación como la presentada, no solo resulta inútil para razonar localmente sobre cada unidad de abstracción, sino que permite un *sharing* mucho mas extendido que lo pretendido originalmente. Lo cierto es que a pesar de los avances de las distintas extensiones, resulta imposible especificar en toda su generalidad las relaciones de *sharing* existentes en muchos patrones de programación utilizados comúnmente, como *iterator*, *productor-consumidor*, *visitor*, etc., sin romper las abstracciones que cada una de las múltiples vistas del heap proporciona.

La propuesta que desarrollamos en este articulo tiene como objetivo especifico definir un lenguaje de aserciones que permita la especificación precisa de estructuras complejas en el *heap*, relaciones de *sharing*

$$\begin{aligned}
Values &\triangleq Integers \cup LValues \\
LValues &\triangleq Lists \cup Trees \cup \ldots \\
Atoms \cup Addresses &\triangleq Integers && (Atoms \text{ y } Addresses \text{ disjuntos}) \\
\mathbf{nil} &\in Atoms \\
Heaps &\triangleq Addresses \rightharpoonup_{fin} Integers \\
Stacks_{PVar} &\triangleq PVar \rightarrow Integers && (PVar \text{ conjunto de nombres de variables de programa}) \\
LStacks_{LVar} &\triangleq LVar \rightarrow Values && (LVar \text{ conjunto de nombres de variables de especificación}) \\
States_{Var} &\triangleq Stacks_{PVar} \times LStacks_{LVar} \times Heaps && (Var = PVar \cup LVar)
\end{aligned}$$

Figura 3: Modelo de estados

entre ellas, y un sistema de prueba composicional asociado, que posibilite la verificación de programas de forma modular en el espíritu de la Separation Logic, aun en presencia del fenómeno de *aliasing*. El objetivo general perseguido consiste en definir un marco de trabajo que nos permita precisar las condiciones para razonar modularmente sobre estructuras en el *heap* aun cuando no es posible garantizar la separación espacial de las mismas.

El articulo se organiza de la siguiente manera. En la sección 2 presentamos un nuevo lenguaje de aserciones, propiedades para razonar sobre las formulas, y el sistema de verificación de programas imperativos asociado. La idea principal consiste en una generalización del operador binario de conjunción espacial $*$ con un operador ternario $\langle * : R \rangle$, donde R es una formula de la misma lógica. Informalmente, la aserción $P \ \langle * : R \rangle \ Q$ se satisface en un *heap* si este puede dividirse en dos *subheaps* (en general no disjuntos), de modo que el primero satisfaga P, el segundo Q, y la intersección entre ellos R. Con el uso de una regla de frame aplicable bajo ciertas condiciones, es posible asegurar la corrección de programas, razonando localmente sobre cada una de las múltiples vistas sobre el *heap*.

En la sección 3 extendemos la teoría para soportar un lenguaje con tipos abstractos de datos. Combinando el concepto de *abstract predicate* ([18]) con nuestro concepto de *especificaciones estáticas* somos capaces de caracterizar el comportamiento de instancias de tipos abstractos de datos que comparten memoria para su representación, de manera abstracta y modular.

En la sección 4 concluimos con una comparación con otros trabajos existentes y las posibilidades de trabajo futuro.

2. Razonando con separación espacial y *sharing*

Para presentar el nuevo lenguaje de aserciones y su utilidad para describir patrones de *sharing* nos basamos en la Separation Logic estándar, manteniendo su modelo de estados y lenguaje de programación, y extendiendo su lenguaje de aserciones y reglas de inferencia. En esta sección introducimos los componentes esenciales de la misma. Para una presentación completa se recomienda leer [21].

2.1. El lenguaje de aserciones

La Separation Logic es una extensión de la lógica de Hoare para un lenguaje imperativo simple ([9]). Extiende el modelo de estados, tradicionalmente conformado por un *stack*, con un *heap*, i.e. una función parcial de direcciones de memoria a valores de programa, en este caso enteros. En esta presentación ademas incluimos un *stack* para variables de especificación, con valores sobre tipos abstractos como listas, arboles etc. Los detalles del modelo de estados se presentan en la figura 3. La diferenciación entre *stack* y *heap* permite otorgar un tratamiento especial a las celdas de la memoria dinámica, a través de un nuevo lenguaje de aserciones. La principal novedad de la Separation Logic respecto a la lógica de Hoare consiste en la introducción de operadores espaciales ($*$, $-\!\!*$) que permiten describir patrones de separación entre regiones del *heap*. Informalmente, una aserción $P * Q$ se satisface en un *heap* si puede dividirse en dos *subheaps* disjuntos, lo cuales satisfacen P y Q respectivamente. Utilizando una *forcing relation* $\models$, $h \uplus h'$ para denotar la unión disjunta de los *heaps* h y h', y s e i para los *stack* de variables de

$$
\begin{aligned}
P ::= {} & b \mid lb \mid P \wedge P \mid P \vee P \mid P \Rightarrow P \mid P \equiv P \mid \neg P \mid (\exists v \cdot P) \mid (\forall v \cdot P) \\
& \mid \mathbf{emp} \mid e \mapsto e \mid P \ \langle * : P \rangle \ P \mid P \ \langle -\!\!* : P \rangle \ P
\end{aligned}
$$

$$
\begin{aligned}
b &::= \mathbf{true} \mid \mathbf{false} \mid e = e \mid e < e \\
e &::= x \mid 0 \mid 1 \mid e + e \mid e * e \mid e - e
\end{aligned}
$$

Figura 4: Gramática de formulas y expresiones

programa y de variables de especificación respectivamente, la semántica del operador $*$ se define como:

$$(s, i, h) \models P * Q \ \Leftrightarrow \ \exists h_1, h_2 \cdot h_1 \uplus h_2 = h \wedge (s, i, h_1) \models P \wedge (s, i, h_2) \models Q$$

Nuestra propuesta se centra en la posibilidad de describir *simultáneamente* relaciones de separación espacial (parcial) y relaciones de *sharing* entre distintas regiones del *heap*, a partir de la generalización de los operadores espaciales binarios $*$ y $-\!\!*$ con operadores ternarios $\langle * : R \rangle$ y $\langle -\!\!* : R \rangle$, indexados por una formula R sobre el mismo lenguaje.

La gramática de las aserciones y expresiones se presenta en la figura 4. Suponemos $v \in Var$, $x \in PVar$. Utilizamos $\overline{v}$ y $\overline{e}$ para denotar secuencias de variables y expresiones. Con FV denotamos el conjunto de variables libres en una formula o expresión. No entraremos en los detalles de las expresiones sobre tipos abstractos, denotadas por bl. Utilizamos $[\,]$, $[x]$ para las listas vacía y unitaria, y $(x \triangleright xs)$ y $(xs \triangleleft x)$ para las listas no vacías con primer y ultimo elemento x. La operación $(xs \uparrow n)$ devuelve la lista de los primeros n valores de xs, y $(xs \downarrow n)$ la lista que resulta de eliminarlos. La evaluación de $xs.n$ resulta en el n-ésimo valor de xs, $xs[x/n]$ en la sustitución del n-ésimo valor de xs por x, y $\#xs$ en la longitud de la lista. Las notaciones $\langle\,\rangle$, $\langle v \rangle$ y $\langle it, v, dt \rangle$ representan los arboles vacío, unitario con nodo v y no vacío con nodo v y subárboles it, dt respectivamente.

Notar que no es posible referenciar al *heap* en las expresiones, por lo tanto estas solo dependen del *stack*. Permitiremos por el momento, como es usual en la literatura, la utilización de predicados recursivos para la especificación de estructuras inductivas. En la sección 3 daremos cuenta formalmente de ellos.

El significado de las formulas heredadas del lenguaje de primer orden es el estándar. La formula **emp** especifica el *heap* vacío, mientras que $e \mapsto e'$ define un *heap* unitario con dirección e y valor e'. La introducción de la formula R en en la conjunción espacial $P \ \langle * : R \rangle \ Q$ nos permite hablar de un *subheap* dado por R compartido entre los *heaps* especificados por P y Q. Intuitivamente, un *heap* h satisface $P \ \langle * : R \rangle \ Q$ si podemos dividirlo en dos *subheaps* h_p y h_q no necesariamente disjuntos, de modo que h_p satisface P, h_q satisface Q y la intersección entre ellos $h_p \cap h_q$ satisface R. La generalización del operador $-\!\!*$ sigue los mismos principios, aunque resulta particularmente difícil dar una interpretación intuitiva correcta. Sin embargo conserva la cualidad de ser el operador adjunto de la conjunción espacial.

Asumiendo la existencia de una función semántica estándar $[\![_]\!]_{exp}$ para expresiones podemos introducir la semántica formal de los nuevos operadores, a través de la siguiente relación:

$$
\begin{array}{lcl}
(s,i,h) \models \mathbf{emp} & \Leftrightarrow & dom_h = \emptyset \\
(s,i,h) \models e \mapsto e' & \Leftrightarrow & dom_h = \{[\![e]\!]_{exp}.s.i\} \wedge h.([\![e]\!]_{exp}.s.i) = [\![e']\!]_{exp}.s.i \\
(s,i,h) \models P \ \langle * : R \rangle \ Q & \Leftrightarrow & \exists h_1, h_2, h_3 \cdot h_1 \uplus h_2 \uplus h_3 = h \wedge (s,i,h_1 \uplus h_3) \models P \wedge (s,i,h_2 \uplus h_3) \models Q \\
& & \quad \wedge (s,i,h_3) \models R \\
(s,i,h) \models Q \ \langle -\!\!* : R \rangle \ P & \Leftrightarrow & \forall h_1 \cdot (\exists h_2 \cdot h_2 \subseteq h \wedge (s,i,h_1 \uplus h_2) \models Q \wedge (s,i,h_2) \models R) \Rightarrow (s,i,h \uplus h_1) \models P
\end{array}
$$

Un operador relacionado que resulta particularmente útil en el sistema deductivo, es la generalización del operador de *septraction* ([23, 7]) definido como el dual de $-\!\!*$:

$$Q \ \langle -\!\!\circledast : R \rangle \ P \triangleq \neg (Q \ \langle -\!\!* : R \rangle \ \neg P)$$

o lo que resulta equivalente según la semántica:

$$Q \ \langle -\!\!\circledast : R \rangle \ P \ \Leftrightarrow \ \exists h_1, h_2 \cdot h_2 \subseteq h \wedge (s, h_1 \uplus h_2) \models Q \wedge (s, h_2) \models R \wedge (s, h \uplus h_1) \models P$$

- **Conmutatividad**

$$P \ \langle * : R \rangle \ Q \ \equiv \ Q \ \langle * : R \rangle \ P$$

- **Elemento neutro**

$$\begin{array}{rcl} P * \mathbf{emp} & \equiv & P \\ P \ \langle * : R_0 * R_1 \rangle \ Q & \Rightarrow & P \ \langle * : R_0 \rangle \ (R_1 \multimap\!\circledast \ Q) \end{array}$$

$$\frac{P \Rightarrow (R_0 * R_1 * \mathbf{true})}{P \ \langle * : R_0 \rangle \ Q \Rightarrow P \ \langle * : R_0 * R_1 \rangle \ (R_1 * Q)}$$

- **Monotonía**

$$\frac{P_0 \Rightarrow P_1 \quad Q_0 \Rightarrow Q_1 \quad R_0 \Rightarrow R_1}{P_0 \ \langle * : R_0 \rangle \ Q_0 \Rightarrow P_1 \ \langle * : R_1 \rangle \ Q_1}$$

- **Adjuntividad**

$$\frac{P_0 \ \langle * : R \rangle \ P_1 \Rightarrow Q}{P_0 \Rightarrow (P_1 \ \langle -\!\!* : R \rangle \ Q)}$$

$$\frac{P \Rightarrow (Q_0 \ \langle -\!\!* : R \rangle \ Q_1)}{P \ \langle * : R \rangle \ Q_0 \Rightarrow Q_1}$$

- **Distributividad**

$$\begin{array}{ll} P \ \langle * : R_0 \vee R_1 \rangle \ Q \equiv (P \ \langle * : R_0 \rangle \ Q) \vee (P \ \langle * : R_1 \rangle \ Q) & \\ (P_0 \vee P_1) \ \langle * : R \rangle \ Q \equiv (P_0 \ \langle * : R \rangle \ Q) \vee (P_1 \ \langle * : R \rangle \ Q) & \\ (P_0 \wedge P_1) \ \langle * : R \rangle \ Q \Rightarrow (P_0 \ \langle * : R \rangle \ Q) \wedge (P_1 \ \langle * : R \rangle \ Q) & \\ (\exists v \cdot P) \ \langle * : R \rangle \ Q \equiv (\exists v \cdot P \ \langle * : R \rangle \ Q) & (v \notin FV.Q) \\ (\forall v \cdot P) \ \langle * : R \rangle \ Q \Rightarrow (\forall v \cdot P \ \langle * : R \rangle \ Q) & (v \notin FV.Q) \end{array}$$

Figura 5: Algunas propiedades de los operadores $\langle * : R \rangle$ y $\langle -\!\!* : R \rangle$

Si h_p, h_q son *heaps* que satisfacen respectivamente las aserciones P y Q, y $h_r \subseteq h_p$ es un *subheap* que satisface R, un *heap* que satisface la aserción $Q \ \langle \multimap\!\circledast : R \rangle \ P$ puede interpretarse como el resultado de sustraerle a h_p el *heap* h_q/h_r. Otras abreviaciones que resultan útiles son:

$$\begin{array}{rll} e \mapsto e_1, e_2, \ldots, e_{n+1} & \triangleq e \mapsto e_1 * (e+1) \mapsto e_2 * \ldots * (e+n) \mapsto e_{n+1} & \\ e \mapsto _ & \triangleq (\exists x \cdot e \mapsto x) & (x \notin FV.e) \\ e \doteq e' & \triangleq (e = e' \wedge \mathbf{emp}) & (\text{idem para } \leq, <, \text{etc}) \end{array}$$

Este nuevo lenguaje de aserciones representa claramente una generalización del lenguaje de la Separation Logic estándar ya que es posible recuperar la semántica original de los operadores espaciales en los siguientes casos particulares[1]:

$$\begin{array}{c} P * Q \equiv P \ \langle * : \mathbf{emp} \rangle \ Q \\ Q -\!\!* P \equiv Q \ \langle -\!\!* : \mathbf{emp} \rangle \ P \end{array}$$

2.2. Clases de aserciones y propiedades

En nuestro marco las reglas del calculo proposicional para los operadores de primer orden permanecen validas, así como las propiedades particulares de la Separation Logic. No vamos a formalizar el sistema deductivo para las formulas de nuestro lenguaje de aserciones sino que presentaremos esquemas de propiedades *validas*. En la figura 5 presentamos algunas propiedades relevantes sobre los operadores espaciales. Tanto como en la Separation Logic usual, no son validas las reglas de *weakening* ni *contraction* para los operadores espaciales. Sin embargo una excepción es el caso particular $P \Rightarrow P \ \langle * : P \rangle \ P$ que se deriva fácilmente de la segunda regla de elemento neutro.

En la literatura ([20, 26]) se definen diferentes clases *semánticas* de formulas de Separation Logic, algunas de las cuales presentamos en la figura 6. Estas clases definen conjuntos de aserciones que satisfacen ciertas propiedades algebraicas entre los operadores. Ejemplo de esto es la reducción de la conjunción e implicación espacial a las correspondientes conjunción e implicación usual para las aserciones *puras*. Para Q una aserción *domain exact* la semidistributividad de $\langle * : R \rangle$ respecto a $\wedge$ se vuelve completa:

$$\begin{array}{ll} (P_0 \wedge P_1) \ \langle * : R \rangle \ Q \Leftarrow (P_0 \ \langle * : R \rangle \ Q) \wedge (P_1 \ \langle * : R \rangle \ Q) & \\ (\forall v \cdot P) \ \langle * : R \rangle \ Q \Leftarrow (\forall v \cdot P \ \langle * : R \rangle \ Q) & (v \notin FV.Q) \end{array}$$

Estas clases desempeñan ademas un papel fundamental en la consistencia de extensiones de la Separation Logic para concurrencia ([14, 23]) y en el soporte de nociones de abstracción ([18, 16]).

[1]Por esta razón utilizaremos $*$, $-\!\!*$ y $\multimap\!\circledast$ para denotar $\langle * : \mathbf{emp} \rangle$, $\langle -\!\!* : \mathbf{emp} \rangle$ y $\langle \multimap\!\circledast : \mathbf{emp} \rangle$ respectivamente

- Una aserción P es *pura* si

$$\forall s,i,h,h' \cdot ((s,i,h) \models P \equiv (s,i,h') \models P)$$

- Una aserción P es *strictly exact* si

$$\forall s,i,h_1,h_2 \cdot ((s,i,h_1) \models P \wedge (s,i,h_2) \models P) \Rightarrow h_1 = h_2$$

- Una aserción P es *domain exact* si

$$\forall s,i,h_1,h_2 \cdot ((s,i,h_1) \models P \wedge (s,i,h_2) \models P) \Rightarrow dom_{h_1} = dom_{h_2}$$

- Una aserción P es *precise* si

$$\forall s,i,h,h_1,h_2 \cdot (h_1 \subseteq h \wedge h_2 \subseteq h \wedge (s,i,h_1) \models P \wedge (s,i,h_2) \models P) \Rightarrow h_1 = h_2$$

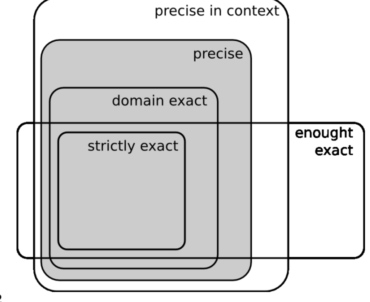

Figura 6: Clases de aserciones y relación de inclusión entre ellas

Lo común entre estas clases es que están definidas independientemente del contexto donde aparezcan las formulas. Una formula R que ocurre en $P \langle op;\ R \rangle Q$ para cualquier operador espacial *op* se encuentra restringida por las formulas P y Q. De esta manera resulta interesante definir clases de aserciones que satisfacen propiedades en relación a otras.

Una clase que resulta de particular interés en nuestra lógica es la de aserciones *precise in context* que extiende la noción de *precise* a las aserciones que indexan los operadores espaciales. Decimos que la aserción P es *precise in context* Q si para cualesquiera s, i, h, h_1 y h_2 se satisface que

$$((s,i,h) \models Q \wedge h_1 \subseteq h \wedge h_2 \subseteq h \wedge (s,i,h_1) \models P \wedge (s,i,h_2) \models P) \Rightarrow dom_{h_1} = dom_{h_2}$$

Las aserciones *precise in context* juegan un rol importante porque extienden significativamente la aplicabilidad de distintas propiedades del operador $\langle * : R \rangle$. En particular nos permite establecer la siguiente *regla de intercambio* que subsume a la regla de asociatividad de $*$:

$$\frac{P_1 \Rightarrow (R_1 * \mathbf{true})}{(P_0 \langle * : R_0 \rangle P_1) \langle * : R_1 \rangle P_2 \Rightarrow P_0 \langle * : R_0 \rangle (P_1 \langle * : R_1 \rangle P_2)} \quad (R_1 \text{ es } \textit{precise in context } (P_0 \langle * : R_0 \rangle P_1))$$

Por otro lado decimos que la aserción P es *enough exact* para Q si para cualesquiera s, i, h, h_1 y h_2 se satisface que

$$(s,i,h \uplus h_1) \models Q \wedge (s,i,h_1) \models P \wedge (s,i,h_2) \models P \Rightarrow (s,i,h \uplus h_2) \models Q$$

Como ya adelantamos, el operador $\langle -\circledast : R \rangle$ desempeña un papel fundamental en el sistema de prueba que introduciremos en la próxima sección. Para ciertas aserciones, las obligaciones de prueba generadas pueden descartarse trivialmente dada la fuerte relación de este operador con $\langle -* : R \rangle$. La siguiente regla captura esta idea:

$$Q \langle -\circledast : R \rangle P \Rightarrow Q \langle -* : R \rangle P \qquad (Q \text{ es } \textit{enough exact} \text{ para } P)$$

Otras propiedades del operador $\langle -\circledast : R \rangle$ se presentan en la figura 7.

A excepción de las formulas *puras* ninguna de las clases de formulas puede caracterizarse de manera sintáctica. Sin embargo es posible dar criterios de suficiencia para todas ellas. Los siguientes lemas nos permiten reconocer sintácticamente a que clase pertenece una formula. La relación entre clases se presenta en la figura 6.

Lema 1 *Sean P, Q, R y S formulas cualesquiera y $class_S$ una clase semántica definida anteriormente. Entonces*

1. **emp** *y* $e \mapsto e'$ *son* strictly exact.
2. $e \mapsto _$ *es* domain exact.
3. *Si P y Q pertenecen a $class_S$ entonces $P \langle * : R \rangle Q$ pertenece a $class_S$.*
4. *Si P o Q pertenecen a $class_S$ entonces $P \wedge Q$ pertenece a $class_S$.*
5. P *es* stricty exact $\Rightarrow$ P *es* domain exact $\Rightarrow$ P *es* precise $\Rightarrow$ P *es* precise in context Q.
6. P *es* stricty exact $\Rightarrow$ P *es* enough exact *para* Q

$$\mathbf{emp}\ \langle -\circledast : R\rangle\ Q \equiv (R \equiv \mathbf{emp}) \wedge Q$$
$$P\ \langle -\circledast : R\rangle\ \mathbf{emp} \equiv (P \equiv \mathbf{emp}) \wedge (R \equiv \mathbf{emp})$$
$$(P_0 * P_1) -\circledast (Q_0 * Q_1) \equiv (P_0 -\circledast Q_0) * (P_1 -\circledast Q_1)$$
$$(P_0 * P_1) -\circledast Q \Rightarrow P_0 -\circledast (P_1 -\circledast Q)$$
$$P\ \langle -\circledast : R\rangle\ (P\ \langle * : R\rangle\ Q) \Rightarrow Q \qquad \text{(cuando } P \text{ es } \textit{precise in context } P\ \langle * : R\rangle\ Q)$$

Figura 7: Algunas propiedades del operador $\langle -\circledast : R\rangle$

De la misma manera es posible caracterizar los contextos para las clases *precise in context* y *enough exact* de manera inductiva.

Lema 2 *Sean P, Q_1 y Q_2 aserciones cualesquiera. Entonces*

1. P *es* precise in context $\mathbf{emp}$.
2. P *es* precise in context $e \mapsto _$.
3. *Si* P *es* precise in context Q_1 *y* $P * \mathbf{true} \Rightarrow \neg Q_2$*, entonces* P *es* precise in context $Q_1\ \langle * : R\rangle\ Q_2$.
4. *Si* P *es* precise in context Q_1 *o* precise in context Q_2*, entonces* P *es* precise in context $Q_1 \wedge Q_2$.
5. *Si* P *es* precise in context Q_1 *y* precise in context Q_2*, entonces* P *es* precise in context $Q_1 \vee Q_2$.
6. *Si* P *es* precise in context Q_2 *y* $Q_1 \Rightarrow Q_2$ *entonces* P *es* precise in context Q_1.
7. *Si* $P * \mathbf{true} \Rightarrow \neg Q$ *entonces* P *es* precise in context Q.

Lema 3 *Sean P, Q_1 y Q_2 aserciones cualesquiera. Entonces*

1. P *es* enough exact *para* $\mathbf{emp}$.
2. P *es* enough exact *para* $(\exists x \cdot x \mapsto _)$.
3. *Si* P *es* enough exact *para* Q_1 *y* enough exact *para* Q_2*, entonces* P *es* enough exact *para* $Q_1 * Q_2$*, para* $Q_1 \wedge Q_2$ *y para* $Q_1 \vee Q_2$.
4. *Si* P *es* enough exact *para* Q_1 *y* $Q_1 \Rightarrow Q_2$*, entonces* P *es* enough exact *en* Q_2.
5. *Si* $P * \mathbf{true} \Rightarrow \neg Q$ *entonces* P *es* enough exact *para* Q.

La expresividad dada por los nuevos operadores parece imposible de imitar en su generalidad por la Separation Logic estándar sin utilizar la conjunción $\wedge$ que inhibe el uso de la regla de frame. Sin embargo es posible reducir una formula de nuestra lógica a la Separation Logic, aunque no parece existir una forma de equivalencia general.

Lema 4 *Las siguientes propiedades son validas.*

$$P\ \langle * : R\rangle\ Q \Leftarrow (R -\!\!* P) * R * (P -\!\!* Q)$$
$$P\ \langle * : R\rangle\ Q \Rightarrow (R -\circledast P) * R * (R -\circledast Q)$$
$$Q\ \langle -\!\!* : R\rangle\ P \Leftarrow (R -\circledast Q) -\!\!* P$$

2.3. Ejemplo: refinando la especificación de *fringe*

Retomemos el problema del *fringe* presentado en la introducción. Este problema es particularmente interesante porque refleja las dificultades, muchas veces sutiles, que aparecen al especificar estructuras que comparten el *heap*. En la figura 8 (a) se presentan las definiciones de los predicados **lseg**, **tree** y de $fringe$. Mas allá de los inconvenientes mencionados en la introducción, el problema con la especificación original:

$$(\mathbf{lseg}.p.q.(fringe.bt) * \mathbf{true}) \wedge \mathbf{tree}.t.bt$$

es que resulta demasiado *débil*, permitiendo casos que no se ajustan a la descripción informal del problema. En la figura 9 (b) se presenta el resultado esperado al calcular el *fringe* en (a). El caso (c) representa una

$$
\begin{aligned}
fringe.\langle v\rangle &\triangleq [v] \\
fringe.\langle it, v, dt\rangle &\triangleq fringe.it +\!\!+ fringe.dt \\[1ex]
\mathbf{lseg}.p.q.[\,] &\triangleq p = q \;\wedge\; \mathbf{emp} \\
\mathbf{lseg}.p.q.(x \triangleright xs) &\triangleq (\exists i \;\cdot\; p \mapsto x, i \;*\; \mathbf{list}.i.q.xs) \\[1ex]
\mathbf{tree}.t.\langle v\rangle &\triangleq t \mapsto \mathbf{nil}, v, _ \\
\mathbf{tree}.t.\langle it, v, dt\rangle &\triangleq (\exists i, j \;\cdot\; t \mapsto i, v, j \;*\; \mathbf{tree}.i.it \;*\; \mathbf{tree}.j.dt)
\end{aligned}
$$

(a)

$$
\begin{aligned}
leafs.\langle v\rangle &\triangleq 1 \\
leafs.\langle it, v, dt\rangle &\triangleq leafs.it + leafs.dt \\[1ex]
\mathbf{lseg}'.p.[q].[\,] &\triangleq p = \mathbf{nil} \;\wedge\; q = \mathbf{nil} \;\wedge\; \mathbf{emp} \\
\mathbf{lseg}'.p.[q].[x] &\triangleq p = q \;\wedge\; p \mapsto x, \mathbf{nil} \\
\mathbf{lseg}'.p.((r \triangleright rs) \triangleleft q).(x \triangleright y \triangleright ys) &\triangleq p \mapsto x, r \;*\; \mathbf{lseg}'.r.(rs \triangleleft q).(y \triangleright ys) \\[1ex]
\mathbf{tree}'.p.[r].\langle v\rangle &\triangleq p \mapsto \mathbf{nil}, v, _ \;\wedge\; r = p + 1 \\
\mathbf{tree}'.p.rs.\langle it, v, dt\rangle &\triangleq (\exists i, j \;\cdot\; p \mapsto i, v, j \;*\; \mathbf{tree}'.i.(rs \uparrow leafs.it).it \;* \\
&\qquad \mathbf{tree}'.j.(rs \downarrow leafs.it).dt \\[1ex]
\mathbf{nodes}.[\,] &\triangleq \mathbf{emp} \\
\mathbf{nodes}.(r \triangleright rs) &\triangleq r \mapsto _, _ \;*\; \mathbf{nodes}.rs
\end{aligned}
$$

(b)

Figura 8: Definiciones para especificación de *fringe*

situación patológica permitida por la especificación. El problema reside en que en general las relaciones entre valores abstractos no reflejan las relaciones entre las celdas utilizadas para su representación.

En la figura 8 (b) se encuentran definiciones de predicados alternativos que distinguen las celdas del *heap* que representan los nodo hoja en el árbol. Utilizando estas definiciones, podemos especificar precisamente el *fringe* con la formula:

$$\mathbf{lseg}'.p.rs.(fringe.bt) \;\langle * : \mathbf{nodes}.rs\rangle\; \mathbf{tree}'.t.vs.bt$$

La solución de los problemas de la especificación no son mérito de nuestros operadores espaciales. Los mismos predicados pueden utilizarse para dar una especificación ajustada utilizando el patrón de *inclusión sharing*. Sin embargo dicha especificación conservaría las desventajas discutidas en la introducción. Por otro lado nuestra propuesta se beneficia de las bondades del sistema de verificación que presentamos en la siguiente sección.

2.4. Lenguaje de programación y especificaciones

Para razonar sobre los programas utilizamos, como es habitual, la noción de Terna de Hoare como especificación para los comandos, siguiendo la siguiente gramática:

$$spec ::= \vdash \{P\}\, C\, \{Q\}$$

donde P y Q son formulas, y C un comando. El significado que le asignamos a una especificación es el de *corrección parcial*: suponiendo que el comando C se ejecuta partiendo de un estado que satisface la precondición P y *efectivamente* termina, el estado resultante satisface la postcondición Q.

Para los comandos tomamos por el momento el lenguaje de programación de la Separation Logic, que extiende el lenguaje imperativo simple aportando cuatro comandos específicos para el acceso y manipulación del *heap*. La gramática de los comandos aparece en la figura 10 y sus especificaciones en la figura 11. Utilizamos la notación $p_{/v \leftarrow e'}$ para denotar la sustitución de las ocurrencias libres de la variable v en p por la expresión e'. Extendemos la utilización de FV sobre comandos, y denotamos con $Mod.C$ las variables modificadas por un comando C.

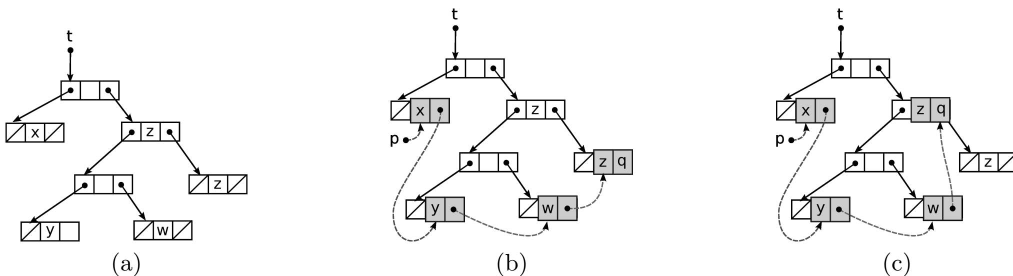

Figura 9: (a) Arbol binario, (b) Arbol binario con fringe, (c) Caso patológico

$$
\begin{aligned}
C ::= x := e \mid \textbf{skip} \mid C;C \ \mid \textbf{if } b \textbf{ then } C \textbf{ else } C \textbf{ fi} \mid \textbf{while } b \textbf{ do } C \textbf{ od} & \\
\mid x := \textbf{cons}(e, \ldots, e) & \qquad \text{(Construcción)} \\
\mid x := [e] & \qquad \text{(Consulta)} \\
\mid [e] := e & \qquad \text{(Modificación)} \\
\mid \textbf{dispose}(e) & \qquad \text{(Destrucción)}
\end{aligned}
$$

Figura 10: Gramática del lenguaje de comandos

Construcción:

$$\vdash \{v = v' \wedge \textbf{emp}\}\, v := \textbf{cons}(e_1, \ldots, e_n)\, \{v \mapsto e_{1/v \leftarrow v'}, \ldots, e_{n/v \leftarrow v'}\}$$

cuando v es distinta de v'.

Consulta:

$$\vdash \{v = v' \wedge e \mapsto e'\}\, v := [e]\, \{v = e'_{/v \leftarrow v'} \wedge e_{/v \leftarrow v'} \mapsto v\}$$

cuando v y v' son distintas.

Modificación:

$$\vdash \{e \mapsto _\}\, [e] := e'\, \{e \mapsto e'\}$$

Destrucción:

$$\vdash \{e \mapsto _\}\, \textbf{dispose}(e)\, \{\textbf{emp}\}$$

Figura 11: Reglas de inferencia para los comandos de manipulación del *heap*

El comando $x := \textbf{cons}(e_1, \ldots, e_n)$ aloca n celdas de memoria inicializadas con los valores $e_1, \ldots, e_n$ y devuelve la dirección de la primer celda en x. Los comandos $x := [e]$ y $[e] := e'$ acceden y modifican la celda del *heap* denotada por e, respectivamente. Finalmente $\textbf{dispose}(e)$ elimina del *heap* la celda e. La ejecución de todos los comandos resulta en una terminación anormal en el caso que la memoria referenciadas no esté definida. Esto se refleja en la forma especifica de la precondición en cada una de las especificaciones. En todos los casos se menciona explícitamente la existencia en el *heap* de las celdas manipuladas. Las especificaciones están tomadas directamente de la Separation Logic y son *locales*, en el sentido que solo mencionan el menor *heap* necesario para su ejecución normal.

Nuestro sistema deductivo para especificaciones consta de las reglas de inferencia especificas de los comandos del lenguaje imperativo simple, así como las reglas estructurales que presentamos en la figura 12. Utilizamos $\models$ para denotar *validez* de una formula.

En general todos los resultados de la lógica de Hoare tradicional siguen siendo validos en nuestro marco, a excepción de la regla de constancia. Para extender la aplicabilidad de las especificaciones de los comandos, la Separation Logic dispone de la regla de frame:

$$\frac{\vdash \{P\}\, C\, \{Q\}}{\vdash \{P * I\}\, C\, \{Q * I\}} \qquad (FV.I \cap Mod.C = \emptyset)$$

que permite extender una especificación local que solo incluye las regiones del *heap* manipuladas por el comando C, con un *invariante* arbitrario I sobre el estado que no esta involucrado en su ejecución.

En nuestra lógica, la introducción de una nueva forma de conjunción espacial nos permite definir 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. Mas aun no es necesario que la formula I sobre el estado global sea invariante si se puede deducir como cambia el estado a lo largo de la ejecución del comando.

De esta manera podemos enuncia la regla de frame general (GFR) como sigue:

$$\frac{\vdash \{P\}\, C\, \{Q\} \quad \models (\exists \overline{v}' \cdot (R_{/\overline{v} \leftarrow \overline{v}'} \multimap\!\!\circledast\, I_{/\overline{v} \leftarrow \overline{v}'})) * R' \Rightarrow I' \quad \models Q \Rightarrow (R' * \textbf{true})}{\vdash \{P \ \langle * : R \rangle\ I\}\, C\, \{Q \ \langle * : R' \rangle\ I'\}}$$

donde $\overline{v}$ representa las variables modificadas por C, y $\overline{v}'$ es una lista de variables frescas

Aquí el *subheap* compartido, que puede cambiar con la ejecución del comando, está dado por las formulas R y R'. La garantía de que la (posible) modificación de la región compartida altera el estado global garantizando I' esta dada por la condición $(\exists\overline{v}' \cdot R_{/\overline{v}\leftarrow\overline{v}'} \multimap\!\circledast\, I_{/\overline{v}\leftarrow\overline{v}'}) * R' \Rightarrow I'$. La condición extra $Q \Rightarrow (R' * \textbf{true})$ excluye el caso inconsistente en el que no existe un *subheap* de Q que satisfaga R'.

Las condiciones para la aplicación de la regla de frame general pueden resultar demasiado fuertes en el caso que las formulas R y R' sean demasiado débiles. Esto no es relevante para los comandos que no modifican el *heap*, por lo tanto podemos enunciar una *Regla de Frame Especial* (SFR) para estos casos:

$$\frac{\vdash \{P\}\, C\, \{Q\}}{\vdash \{P\ \langle * : R\rangle\ I\}\, C\, \{Q\ \langle * : R\rangle\ I\}} \qquad ((FV.R \cup FV.I) \cap Mod.C = \emptyset)$$

2.5. Ejemplo: Inserción en un árbol binario de búsqueda

Para ilustrar el uso de nuestra lógica tomamos el problema de insertar un nodo en un árbol binario de búsqueda. Si bien la especificación misma del problema no involucra estructuras que comparten el *heap*, utilizamos el poder expresivo del operador $\langle * : R\rangle$ para especificar los estados intermedios de la computación, intentado obtener una prueba que se ajuste a las intuiciones que subyacen a la especificación recursiva usual del algoritmo. La especificación del problema se encuentra en la figura 13, donde utilizamos **until** b **do** C **od** para C; **while** b **do** C **od** . La intuición del algoritmo es muy sencilla: el ciclo explora el árbol en busca del punto de inserción; una vez hallado el punto se reemplaza el subárbol vacío por el árbol unitario correspondiente.

En lo que respecta a la utilidad de nuestro sistema de prueba, la porción de código mas interesante es a la salida del ciclo, cuando se encuentra que $x \neq y$. Allí se modifica el *heap* creando un nuevo nodo para el elemento insertado. Un invariante para el ciclo es el siguiente:

$$(\exists pt \cdot (q \mapsto p * q + \overline{x \leq y} \mapsto y * \textbf{tree}.p.pt)\ \langle * : q \mapsto p * q + \overline{x \leq y} \mapsto y * \textbf{tree}.p.pt\rangle\ \textbf{tree}.t.\langle it, v, dt\rangle)$$

donde, para lograr una especificación mas compacta, extendemos las expresiones booleanas con una función $\overline{b}$ cuya semántica se presenta en la figura 13 (b). En la presentación de las pruebas utilizamos la siguiente notación. La aplicación de la regla de consecuencia se presenta como aserciones consecutivas y una justificación, no siempre completa, del paso deductivo utilizado. La aplicación de otras reglas estructurales, normalmente la GFR, se denota con una barra horizontal acompañada del nombre de la regla. La demostración de la porción de código relevante se presenta a continuación:

$\Big\{(\exists pt \cdot (q \mapsto p * q + \overline{x \leq y} \mapsto y * \textbf{tree}.p.pt)\ \langle * : q \mapsto p * q + \overline{x \leq y} \mapsto y * \textbf{tree}.p.pt\rangle\ \textbf{tree}.t.\langle it, v, dt\rangle) * p \doteq \textbf{nil}\Big\}$

$\Rightarrow \langle$ Leibniz, def. **tree** $\rangle$

$\Big\{(\exists pt \cdot (q \mapsto \textbf{nil} * q + \overline{x \leq y} \mapsto y * p \doteq \textbf{nil}))\ \langle * : q \mapsto \textbf{nil} * q + \overline{x \leq y} \mapsto y * p \doteq \textbf{nil}\rangle\ \textbf{tree}.t.\langle it, v, dt\rangle)\Big\}$

$\equiv \langle$ Elemento Neutro, debilitamiento, prop. cuantificadores $\rangle$

$\{(q \mapsto \textbf{nil} * q + \overline{x \leq y} \mapsto y)\ \langle * : q \mapsto \textbf{nil} * q + \overline{x \leq y} \mapsto y\rangle\ \textbf{tree}.t.\langle it, v, dt\rangle\}$

GFR:

> $\{q \mapsto \textbf{nil} * q + \overline{x \leq y} \mapsto y\}$
>
> $p := \textbf{cons}(\textbf{nil}, x, \textbf{nil});$
>
> $\{q \mapsto \textbf{nil} * q + \overline{x \leq y} \mapsto y * p \mapsto \textbf{nil}, x, \textbf{nil}\}$
>
> $\equiv \langle$ Def. **tree** $\rangle$
>
> $\{q \mapsto \textbf{nil} * q + \overline{x \leq y} \mapsto y * \textbf{tree}.p.\langle x\rangle\}$
>
> $[q] := p;$
>
> $\{q \mapsto p * q + \overline{x \leq y} \mapsto y * \textbf{tree}.p.\langle x\rangle\}$
>
> $\equiv \langle$ Def. *ins* $\rangle$
>
> $\{q \mapsto p * q + \overline{x \leq y} \mapsto y * \textbf{tree}.p.(ins.x.\langle\,\rangle)\}$

$\{(q \mapsto p * q + \overline{x \leq y} \mapsto y * \textbf{tree}.p.(ins.x.\langle\,\rangle))\ \langle * : q \mapsto p * q + \overline{x \leq y} \mapsto y * \textbf{tree}.p.(ins.x.\langle\,\rangle)\rangle\ \textbf{tree}.t.(ins.x.\langle it, v, dt\rangle)\}$

$\Rightarrow \langle$ Elemento neutro $\rangle$

$\{(q \mapsto p * q + \overline{x \leq y} \mapsto y * \textbf{tree}.p.(ins.x.\langle\,\rangle)) \multimap\!\circledast (q \mapsto p * q + \overline{x \leq y} \mapsto y * \textbf{tree}.p.(ins.x.\langle\,\rangle)) * \textbf{tree}.t.(ins.x.\langle it, v, dt\rangle)\}$

$\Rightarrow \langle$ Prop. $\multimap\!\circledast$ con **tree** *precise*) $\rangle$

$\{\textbf{tree}.t.(ins.x.\langle it, v, dt\rangle)\}$

Para la aplicación de la GFR, es necesario demostrar la validez de las siguientes formulas:

1. $((q \mapsto p * q + \overline{x \leq y} \mapsto y * \textbf{tree}.p.\langle\,\rangle) \multimap\!\circledast\ \textbf{tree}.t.\langle it, v, dt\rangle)) * (q \mapsto p * q + \overline{x \leq y} \mapsto y * \textbf{tree}.p.(ins.x.\langle\,\rangle) \Rightarrow \textbf{tree}.t.(ins.x.\langle it, v, dt\rangle)$
2. $(q \mapsto p * q + \overline{x \leq y} \mapsto y * \textbf{tree}.p.(ins.x.\langle\,\rangle)) \Rightarrow (q \mapsto p * q + \overline{x \leq y} \mapsto y * \textbf{tree}.p.(ins.x.\langle\,\rangle))$

Consecuencia (CR):

$$\frac{\models P' \Rightarrow P \quad \vdash \{P\}\, C\, \{Q\} \quad \models Q \Rightarrow Q'}{\{P'\}\, C\, \{Q'\}}$$

Eliminación de variables auxiliares (VER):

$$\frac{\vdash \{P\}\, C\, \{Q\}}{\vdash \{\exists v \cdot P\}\, C\, \{\exists v \cdot Q\}}$$

cuando $v \notin FV.C$.

Sustitución (SR):

$$\frac{\vdash \{P\}\, C\, \{Q\}}{\vdash (\{P\}\, C\, \{Q\})/v_1 \leftarrow e_1, \ldots v_n \leftarrow e_n}$$

cuando si $v_i \in Mod.C$ entonces e_i es una variable que no ocurre libre en otra expresión e_j.

Asignación:

$$\vdash \{R_{/v \leftarrow e}\}\, v := e\, \{R\}$$

Secuencia:

$$\frac{\vdash \{P\}\, C\, \{S\} \quad \vdash \{S\}\, C'\, \{Q\}}{\vdash \{P\}\, C; C'\, \{Q'\}}$$

Condicional:

$$\frac{\vdash \{P \wedge B\}\, C\, \{Q\} \quad \vdash \{P \wedge \neg B\}\, C'\, \{Q\}}{\vdash \{P\}\, \mathbf{if}\ B\ \mathbf{then}\ C\ \mathbf{else}\ C'\ \mathbf{fi}\, \{Q\}}$$

Ciclo:

$$\frac{\vdash \{P \wedge B\}\, C\, \{P\}}{\vdash \{P\}\, \mathbf{while}\ B\ \mathbf{do}\ C\mathbf{od}\ \{P \wedge \neg B\}}$$

Figura 12: Reglas de inferencia de comandos y reglas estructurales

La condición (2) se cumple trivialmente. Para satisfacer (1) demostramos la siguiente proposición, que captura la idea intuitiva de que el reemplazo del subárbol vacío apropiado con un subárbol unitario representa la inserción en el árbol completo.

Proposicion 1 *Para cualquier árbol binario de búsqueda $\langle it, v, dt\rangle$ se satisface*

$$((q \mapsto p * q + \overline{x \leq y} \mapsto y * \mathbf{tree}.p.pt) -\!\circledast\ \mathbf{tree}.t.\langle it, v, dt\rangle)) * (q \mapsto p * q + \overline{x \leq y} \mapsto y * \mathbf{tree}.p.(ins.x.pt) \Rightarrow \mathbf{tree}.t.(ins.x.\langle it, v, dt\rangle))$$

2.6. Semántica

2.6.1. El lenguaje de programación

Para definir el significado de los comandos del lenguaje de programación utilizamos una semántica *small-step*, a través de una relación de transición $\rightsquigarrow$ entre configuraciones, que pueden ser

- *no terminales*: un par comando-estado, $\langle C, (s, h)\rangle$ tal que $FV.C \subseteq dom_s$
- *terminales*: un estado (s, h), o una terminación anormal denotada por **abort**.

Obviamos la mención del *stack* i de variables de especificación, ya que la semántica de los comandos y de las expresiones involucradas no dependen del mismo. Presentamos en la figura 14 la semántica de los comandos de manipulación del *heap*. Utilizamos la notación $[f \mid x : a]$ para representar la función que mapea x en a y todo otro argumento $y \in dom_f$ en $f.y$; y $f|_D$ para denotar la restricción de la función f al dominio D. Con A/B denotamos la sustracción del conjunto B al conjunto A. Escribimos $\gamma \rightsquigarrow^* \gamma'$

$$ins : Int \rightarrow \langle Int\rangle \rightarrow \langle Int\rangle$$

$$\begin{aligned} ins.x.\langle\ \rangle &\triangleq \langle x\rangle \\ ins.x.\langle it, y, dt\rangle &\triangleq (x < y \rightarrow \langle ins.x.it, y, dt\rangle \\ &\qquad x = y \rightarrow \langle it, y, dt\rangle \\ &\qquad x > y \rightarrow \langle it, y, ins.x.dt\rangle \\ &\qquad) \end{aligned}$$

(a)

$$\begin{aligned} \mathbf{tree}.t.\langle\ \rangle &\triangleq t = \mathbf{nil} \wedge \mathbf{emp} \\ \mathbf{tree}.t.\langle it, v, dt\rangle &\triangleq (\exists i, j \cdot t \mapsto i, v, j * \\ &\qquad \mathbf{tree}.i.it * \mathbf{tree}.j.dt) \end{aligned}$$

(b)

$$\begin{aligned} [\overline{b}]_{bool}.s.i \triangleq\ & ([b]_{bool}.s.i \rightarrow 1 \\ & \square\ \neg[b]_{bool}.s.i \rightarrow -1 \\ &) \end{aligned}$$

(c)

```
{ tree.t.⟨it, v, dt⟩ }
  p := t;
  until p ≠ nil ∧ x ≠ y do
      y := [p + 1];
      if x ≤ y then q := p
      else q := p + 2 fi;
      p := [q];

  od
  if x ≠ y then
     p := cons(nil, x, nil);
     [q] := p;
  fi
{ tree.t.(ins.x.⟨it, v, dt⟩) }
```

(d)

Figura 13: Inserción en un árbol binario de búsqueda

- **Construcción**:

$$\frac{(\forall i \in \{1,\ldots,n\} \;\cdot\; a_i \in Address/dom_h)}{\langle v := \mathbf{cons}(e_1,...,e_n),(s,h)\rangle \rightsquigarrow ([s \mid v : a_1],[h \mid a_1 : [e_1]_{exp}.s \mid \ldots \mid a_n : [e_n]_{exp}.s])}$$

- **Consulta:**

$$\frac{[e]_{exp}.s \in dom_h}{\langle v := [e],(s,h)\rangle \rightsquigarrow ([s \mid v : h([e]_{exp}.s)],h)} \qquad \frac{[e]_{exp}.s \notin dom_h}{\langle v := [e],(s,h)\rangle \rightsquigarrow \mathbf{abort}}$$

- **Modificación:**

$$\frac{[e]_{exp}.s \in dom_h}{\langle [e] := e',(s,h)\rangle \rightsquigarrow (s,[h \mid [e]_{exp}.s : [e']_{exp}.s])} \qquad \frac{[e]_{exp}.s \notin dom_h}{\langle [e] := e',(s,h)\rangle \rightsquigarrow \mathbf{abort}}$$

- **Destrucción:**

$$\frac{[e]_{exp}.s \in dom_h}{\langle \mathbf{dispose}(e),(s,h)\rangle \rightsquigarrow (s,h|_{dom_h/\{[e]_{exp}.s\}})} \qquad \frac{[e]_{exp}.s \notin dom_h}{\langle \mathbf{dispose}(e),(s,h)\rangle \rightsquigarrow \mathbf{abort}}$$

Figura 14: Semántica operacional de los comandos

para indicar que existe una secuencia finita de transiciones de γ a γ', y $\gamma \uparrow$ para indicar que existe una secuencia infinita comenzando desde γ.

Enunciamos a continuación dos propiedades fundamentales de los comandos, que resultan imprescindibles a la hora de demostrar la consistencia del sistema de prueba. Por una demostración revisar [27]. Decimos que la configuración $\langle C,(s,h)\rangle$ es *safe* si $\langle C,(s,h)\rangle \not\rightsquigarrow^* \mathbf{abort}$, y que $\langle C,(s,h)\rangle$ *termina normalmente* si $\langle C,(s,h)\rangle$ es *safe* y $\langle C,(s,h)\rangle \not\uparrow$. Con $h \perp h'$ denotamos la disjuntividad entre h y h'.

Lema 5 ***(Safety y Termination Monotonicity)***
1. Si $\langle C,(s,h)\rangle$ *es* safe *y* $h \perp h'$*, entonces* $\langle C,(s,h \uplus h')\rangle$ *es* safe.
2. Si $\langle C,(s,h)\rangle$ termina normalmente *y* $h \perp h'$*, entonces* $\langle C,(s,h \uplus h')\rangle$ termina normalmente.

Lema 6 ***(Propiedad de Frame)***
Supongamos que $\langle C,(s,h_0)\rangle$ *es* safe *y* $\langle C,(s,h_0 \uplus h_1)\rangle \rightsquigarrow^* \langle C,(s,h')\rangle$*. Entonces existe* h'_0 *tal que* $\langle C,(s,h_0)\rangle \rightsquigarrow^* \langle C,(s,h'_0)\rangle$ *y* $h' = h'_0 \cup h_1$*.*

2.6.2. Especificaciones

Para terminar, estamos en posición de presentar la semántica formal de las especificaciones. Decimos que una especificación es *valida* denotado por $\models \{P\}\,C\,\{Q\}$ si

$$\forall s,i,h,s',h' \cdot (s,i,h) \models P \;\Rightarrow\; \langle C,(s,h)\rangle \text{ es } \mathit{safe} \wedge (\langle C,(s,h)\rangle \rightsquigarrow^* (s',h') \;\Rightarrow\; (s',i,h') \models Q)$$

Con esta definición podemos presentar el siguiente teorema de consistencia del sistema de prueba sobre las especificaciones

Teorema 7 ***(Soundness)***
Si $\vdash \{P\}\,C\,\{Q\}$ *se deduce aplicando las reglas del sistema deductivo entonces se satisface* $\models \{P\}\,C\,\{Q\}$*.*

Demostración: Verificaremos únicamente la validez de las reglas de frame.

- Para el caso *General*:

  Supongamos que las premisas de la regla se cumplen. Supongamos ademas que $(s,h_p \uplus h_r) \models P$, $(s,h_r) \models R$ y $(s,h_i \uplus h_r) \models I$. La especificación $\vdash \{P\}\,C\,\{Q\}$ nos dice que $\langle C,(s,h_p \uplus h_r)\rangle$ es *safe* y por *Safety Monotonicity* la configuración $\langle C,(s,h_p \uplus h_r \uplus h_i\rangle$ es *safe*. Por otro lado si $\langle C,(s,h_p \uplus h_r \uplus h_i)\rangle \rightsquigarrow^* (s',h')$ la *Propiedad de Frame* nos dice que existe un h'_q tal que $\langle C,(s,h_p \uplus h_r)\rangle \rightsquigarrow^* (s',h'_q)$ y $h' = h'_q \uplus h_i$. Así por la validez de la especificación tenemos que $(s',h'_q) \models Q$. La la premisa $Q \Rightarrow (R' * \mathbf{true})$ nos asegura que existe $h'_r \subseteq h'_q$ tal que $(s',h'_r) \models R'$.

  Para concluir que $(s',h'_q \uplus h_i) \models Q \;\langle * : R'\rangle\; I'$ es necesario demostrar que $(s',h'_r \uplus h_i) \models I'$. Para ello utilizaremos la hipótesis $(\exists v' \cdot R_{/v \leftarrow v'} \;-\!\circledast\; I_{/v \leftarrow v'}) * R' \Rightarrow I'$. Sabemos ya que $(s',h'_r) \models R'$,

por lo tanto es suficiente ver que $(s', h_i) \models \exists v' \bullet R_{/v \leftarrow v'} \multimap\!\circledast\ I_{/v \leftarrow v'}$. Por nuestra hipótesis inicial, tenemos que $(s, h_i) \models (R \multimap\!\circledast\ I)$ y para toda $v \notin Mod.C$, se cumple que que $s.v = s'.v$. Luego es evidente que $([s' \mid v : s.v], h_i) \models (R \multimap\!\circledast\ Q)$ y por lo tanto $(s', h_i) \models (\exists v' \cdot R_{/v \leftarrow v'} \multimap\!\circledast\ I_{/v \leftarrow v'})$.

□

- Para el caso *Especial*:

  Supongamos que $(s, h_p \uplus h_r) \models P$, $(s, h_r) \models R$, $(s, h_i \uplus h_r) \models I$. La especificación $\{P\}\, C\, \{Q\}$ nos dice que $\langle C, (s, h_p \uplus h_r)\rangle$ es *safe* y por *Safety Monotonicity* la configuración $\langle C, (s, h_p \uplus h_r \uplus h_i\rangle$ es *safe*. Por otro lado si $\langle C, (s, h_p \uplus h_r \uplus h_i)\rangle \rightsquigarrow^* (s', h')$ la *Propiedad de Frame* nos dice que existe un h'_q tal que $\langle C, (s, h_p \uplus h_r)\rangle \rightsquigarrow^* (s', h'_q)$ y $h' = h'_q \uplus h_i$. Así por la definición de especificación tenemos que $(s', h'_q) \models Q$.

  Como C no modifica el *heap*, entonces $h'_q = h_p \uplus h_r$, luego $(s', h_p \uplus h_r) \models Q$. Por otro lado como C no modifica variables que ocurren en R o I tenemos que $(s', h_r) \models R$ y $(s', h_i \uplus h_r) \models I$. Asia finalmente concluimos que $(s', h'_q \uplus h_i) \models Q\ \langle * : R\rangle\ I$.

□

3. Un lenguaje con módulos

En esta sección presentamos una extensión al lenguaje de programación para soportar módulos que implementan tipos abstractos de datos. Para dar cuenta de ello en las aserciones y el sistema de prueba asociado, utilizaremos el concepto de *abstract predicate* ([18, 17]). Lo referido a ese concepto es tomado directamente de [18] y se recomienda su lectura para encontrar los detalles. Así mismo extendemos el *framework* introduciendo la noción de *especificaciones estáticas*[2], que nos permite razonar modularmente en presencia de un (posible) *sharing* en la implementación de los tipos abstractos de datos.

3.1. Lenguaje de programación y especificaciones

Extendemos el lenguaje de programación introduciendo comandos específicos para la definición y llamada a procedimientos. La sintaxis de los nuevos comandos se presentan en la figura 15. Restringimos nuestra consideración a programas *bien formados*: suponemos existe una variable distinguida *ret* que no es modificable excepto a través del comando **return**; los procedimientos solo tienen un comando **return** como ultimo comando; y un comando **let** define cada nombre de procedimiento a lo sumo una vez.

Para razonar sobre los nuevos comandos extendemos la noción de especificación incluyendo un contexto formado por: una secuencia de definiciones de predicados de abstracción Λ; una secuencia de especificaciones para los procedimientos Γ; y una secuencia especificaciones estáticas Σ.

$$spec ::= \Lambda; \Gamma; \Sigma \vdash \{P\}\, C\, \{Q\}$$

$$\begin{aligned}
\Lambda &::= \epsilon \mid \alpha(\overline{x}) \triangleq P, \Lambda \\
\Gamma &::= \epsilon \mid \{P\}k(\overline{x})\{Q\}, \Gamma \\
\Sigma &::= \epsilon \mid [P_{\overline{x}} \Rightarrow Q_{\overline{x}}], \Sigma
\end{aligned}$$

Utilizamos $P_{\overline{x}}$ para denotar que la formula P tiene sus variables libres contenidas en $\overline{x}$. Denotamos con ϵ la secuencia vacía. Al igual que los programas solo consideraremos contextos *bien definidos*: cada nombre de procedimiento k solo ocurre una vez y las variables libres en la especificación están contenidas en los parámetros $\overline{x}$ y *ret*. En cuanto a las definiciones de los predicados de abstracción, α solo ocurre una vez, las variables libres de P ocurren en los argumentos $\overline{x}$ y P es una formula positiva (i.e. una formula donde los nombres de predicados ocurren solo bajo un numero par de negaciones).

Hasta aquí hemos utilizado predicados recursivos para especificar estructuras inductivas de manera informal, como es costumbre en la literatura. Sin embargo para que nuestro nuevo *framework* sea

[2]Ver la discusión en la sección 4 en relación al concepto homónimo de [11]

$$
\begin{aligned}
C ::= \ & \ldots \\
& \mid \textbf{newvar}(v); C && \text{(Variable Local)} \\
& \mid \textbf{return}(e) && \text{(Valor retorno)} \\
& \mid \textbf{let}\ k\ \overline{x} = C \ldots, k\ \overline{x} = C\ \textbf{in}\ c && \text{(Definición de modulo)} \\
& \mid v := k(\overline{x}) && \text{(Llamada a procedimiento)}
\end{aligned}
$$

Figura 15: Gramática de comandos extendidos

consistente es necesario ser mas preciso a la hora de intercambiar un predicado de abstracción por su definición. contexto. El contexto introducido en las especificaciones nos permite aproximarnos al tipo de razonamiento intuitivo asociado con los tipos abstractos de datos. Los predicados de abstracción tanto como los tipos abstractos de datos poseen un nombre, una *implementación* dada por la definición del predicado, y un *scope* definido dentro del cual se conocen los detalles de la misma. Dentro del *scope* la definición puede intercambiarse libremente con el nombre del predicado. Sin embargo fuera del mismo, es decir en el código cliente, el predicado de abstracción solo puede manipularse atómicamente a través de su nombre, con las especificaciones de los procedimientos y especificaciones estáticas que lo mencionan.

Las especificaciones estáticas proveen información adicional a las especificaciones de los procedimientos. Permiten realizar deducciones validas en el código cliente sin conocer los detalles de la implementación. En particular se espera que sean útiles para resolver las hipótesis de las reglas de frame frente a la existencia de *sharing*. Un punto importante a tener en cuenta, es que las especificaciones estáticas abren la posibilidad de exponer la representación del tipo, rompiendo la abstracción. Este es un error de diseño que efectivamente también puede suceder en las especificaciones de los procedimientos. Sin embargo en muchos casos abren la posibilidad de razonar sobre instancias del tipo abstracto de datos sin romper el principio de ocultamiento de información.

Para formalizar las posibilidades deductivas de los predicados de abstracción y especificaciones estáticas es necesario modificar la noción de *validez* de una formula respecto a un contexto, denotado por $\Lambda; \Sigma \models P$. La intuición es que la formula P es verdadera para cualquier estado suponiendo correctas las definiciones de predicados en Λ y validas las implicaciones lógicas en las especificaciones estáticas Σ. Los siguientes axiomas formalizan las intuiciones desarrolladas mas arriba:

$$
\begin{aligned}
\alpha(\overline{x}) \triangleq P, \Lambda; \Sigma \models \alpha(\overline{e}) \ \equiv \ P_{/\overline{x} \leftarrow \overline{e}} \\
\Lambda; [P_{\overline{x}} \ \Rightarrow \ Q_{\overline{x}}], \Sigma \models P_{/\overline{x} \leftarrow \overline{e}} \ \Rightarrow \ Q_{/\overline{x} \leftarrow \overline{e}}
\end{aligned}
$$

La especificaciones de los nuevos comandos se presentan en la figura 16. Quizás la regla que merezca mayor atención es la de definición de procedimiento. Abusamos de la notación escribiendo $\Lambda, \Lambda' \models \Sigma$ para denotar que $\Lambda, \Lambda' \models P \Rightarrow Q$ para toda especificación $[P_{\overline{x}} \Rightarrow Q_{\overline{x}}] \in \Sigma$. Este regla permite al desarrollador del modulo utilizar la definición del predicado de abstracción (en Λ') para el desarrollo de los procedimientos $k_1, \ldots, k_n$ del modulo, y demostrar la validez de las especificaciones estáticas. Por otro lado el cliente solo puede manipular el tipo abstracto a través de las especificaciones de los procedimientos, y realizar pasos deductivos con las especificaciones estáticas. Las condiciones de lado principalmente aseguran que los predicados de abstracción no se escapen del *scope*.

La extensión de las reglas originales de la sección 2.3 a la nueva forma de especificación es trivial. Notar que es necesario modificar la regla estructural de consecuencia para la nueva forma de validez.

3.2. Ejemplos

3.2.1. Iteradores

El patrón de iterador es común en cualquier lenguaje de programación actual. Un iterador es un tipo abstracto de datos sobre una colección mutable de elementos (lista enlazada, vector, etc.) que permite recorrerla, accediendo y modificando sus valores. La colección de datos puede incluir métodos que la modifiquen *estructuralmente*, como agregar o quitar un elemento, y otros que no, como observar la cantidad de elementos. De la misma manera el iterador posee métodos para recorrer la colección subyacente, que no la modifican, y otros que permiten alterar los valores. Este ultimo tipo de modificación es *no*

Variable local:

$$\frac{\Lambda, \Sigma, \Gamma \vdash \{P_{/x \leftarrow y} \wedge x = \mathbf{nil}\}\, C\, \{Q_{/x \leftarrow y}\}}{\Lambda, \Sigma, \Gamma \vdash \{P\}\, \mathbf{newvar}(x); C\, \{Q\}}$$

cuando $y \notin FV.P \cup FV.C \cup FV.Q$.

Valor retorno:

$$\Lambda; \Gamma; \Sigma \vdash \{P_{/ret \leftarrow x}\}\, \mathbf{return}(x)\, \{P\}$$

Definición de modulo:

$$\frac{\begin{array}{c} \Lambda, \Lambda'; \Sigma \models \Sigma' \\ \Lambda, \Lambda'; \Gamma; \Sigma, \Sigma' \vdash \{P_1\}\, C_1\, \{Q_1\} \\ \vdots \\ \Lambda, \Lambda'; \Gamma; \Sigma, \Sigma' \vdash \{P_n\}\, C_n\, \{Q_n\} \\ \Lambda, \Gamma, \{P_1\}k_1(\overline{x_1})\{Q_1\}, \ldots, \{P_n\}k_n(\overline{x_n})\{Q_n\}; \Sigma, \Sigma' \vdash \{P\}C\{Q\} \end{array}}{\Lambda; \Gamma; \Sigma \vdash \{P\}\mathbf{let}\ k_1\ \overline{x_1} \triangleq C_1, \ldots, k_n\ \overline{x_n} \triangleq C_n\ \mathbf{in}\ C\{Q\}}$$

cuando

- P,Q,Γ, Λ y Σ no contienen nombres de predicados en $dom(\Lambda')$,
- $dom(\Lambda')$ y $dom(\Lambda)$ son disjuntos, y
- los procedimientos solo modifican variables locales.

Llamada a procedimiento:

$$\frac{\Lambda, \Sigma, \Gamma \vdash \{P\}\, k(\overline{x})\, \{Q\} \in \Gamma}{\Lambda; \Gamma; \Sigma \vdash \{P_{/\overline{x} \leftarrow \overline{y}}\}\, y = k(\overline{y})\, \{Q_{/\overline{x}, ret \leftarrow \overline{y}, y}\}}$$

Figura 16: Especificación de los comandos de módulos

estructural. La diferencia entre ambos tipos de modificación es inapreciable desde el punto de vista del código cliente, dada la capa de abstracción provista por el tipo de datos. Sin embargo esta distinción es comúnmente utilizada en la descripción o especificación informal de este tipo de módulos, que suelen incluir restricciones como la siguiente: para que la semántica de uno o mas iteradores sobre una colección este bien definida no se permite la ejecución de métodos que la modifiquen estructuralmente; sin embargo esta permitida la ejecución de métodos de observación o modificación no estructural. Esta restricción se debe a que en general cualquier implementación eficiente de un iterador incluye un puntero al interior de la estructura que implementa la colección. Es imposible para los métodos que modifican estructuralmente la colección mantener la consistencia con ese puntero.

En la figura 17 se presenta parte de la especificación de un modulo que implementa colecciones mutables e iteradores. El predicado $\mathbf{coll}.c.ls.xs$ define una posible implementación de una colección, donde c es una variable de programa y ls y xs son listas de expresiones de especificación. La utilización de la variable de especificación ls es imprescindible para dar cuenta de las modificaciones estructurales. Notar que esto no representa una exposición de la implementación, ya que ls no incluye variables de programa y la definición de **coll** permanece *oculta* fuera del *scope* del modulo.

El predicado $\mathbf{iter}.i.n.c.ls.xs$ define la parte relevante de la implementación de un iterador. La variable de programa i es utilizada para referenciar la instancia del tipo abstracto de dato. La variable de especificación n representa el indice actual del recorrido sobre la colección subyacente representada por c, ls y xs.

Se presentan una lista (no completa) de procedimientos sobre los tipos de datos y sus especificaciones. Notar la utilización de la variable de especificación ls para denotar cambios estructurales en los procedimientos *add* y *del*. De la misma manera, las variables xs y n reflejan los cambios producidos solo en los *valores*, en los procedimientos *next* y *set*. Nuevamente no es posible una distinción por parte del cliente de estos tipos de modificaciones, sin embargo, las especificaciones de procedimientos junto a las especificaciones estáticas reflejan estas diferencias. Las especificaciones ς_2 y ς_3 permiten deducir los cambios en las múltiples vistas posibles sobre la colección al producirse un cambio en los valores xs. Finalmente la especificación ς_1 permite recuperar la información de la colección subyacente a un iterador.

El la figura 18 se presentan ejemplos de código que, aunque no realistas, permiten dar una idea de como combinar las especificaciones de procedimientos, especificaciones estáticas y la GFR para razonar

$$\Lambda: \quad \begin{aligned} \mathbf{coll}.c.[\,].[\,] &\triangleq c = \mathbf{nil} \wedge \mathbf{emp} \\ \mathbf{coll}.c.(l \triangleright ls).(x \triangleright xs) &\triangleq c \mapsto x, l \,*\, \mathbf{coll}.l.ls.xs \\ \mathbf{iter}.i.n.c.ls.xs &\triangleq \exists k \cdot i \mapsto c, k \,*\, \ldots \,*\, (\mathbf{coll}.c.ls.xs \wedge (n < \#xs \Rightarrow k = ps.n)) \end{aligned}$$

$$\Gamma: \quad \begin{array}{rcl} \{\mathbf{emp}\} & new_coll() & \{\mathbf{coll}.ret.[\,].[\,]\} \\ \{\mathbf{coll}.c.ls.xs\} & add(c,x) & \{(\exists ls \cdot \mathbf{coll}.c.ls.(x \triangleright xs))\} \\ \{\mathbf{coll}.c.ls.(x \triangleright xs)\} & del(c) & \{(\exists ls \cdot \mathbf{coll}.c.ls.xs)\} \\ \{\mathbf{coll}.c.ls.xs\} & size(c) & \{\mathbf{coll}.c.ls.xs \wedge ret = \#xs\} \\ \{\mathbf{coll}.c.ls.xs\} & new_iter(c) & \{\mathbf{iter}.ret,0.c.ls.xs\} \\ \{\mathbf{iter}.i.n.c.ls.xs \wedge n < \#xs - 1\} & next(i) & \{\mathbf{iter}.i.(n+1).c.ls.xs \wedge ret = xs.n\} \\ \{\mathbf{iter}.i.n.c.ls.xs \wedge n < \#xs\} & set(i,x) & \{\mathbf{iter}.i.n.c.ls.(xs[x/n])\} \\ \{\mathbf{iter}.i.n.c.ls.xs\} & index(i) & \{\mathbf{iter}.i.n.c.ls.xs \wedge ret = n\} \end{array}$$

$$\Sigma: \quad \begin{array}{l} \varsigma_1 : [\mathbf{iter}.i.n.c.ls.xs \Rightarrow \mathbf{coll}.c.ls.xs \,*\, \mathbf{true}] \\ \varsigma_2 : [(\mathbf{coll}.c.ls.xs \multimap\!\!\circledast\; \mathbf{iter}.i.n.c.ls.xs) \,*\, \mathbf{coll}.c.ls.xs' \Rightarrow \mathbf{iter}.i.n.c.ls.xs'] \\ \varsigma_3 : [(\mathbf{coll}.c.ls.xs \multimap\!\!\circledast\; \mathbf{coll}.c.ls.xs) \,*\, \mathbf{coll}.c.ls.xs' \Rightarrow \mathbf{coll}.c.ls.xs'] \end{array}$$

Figura 17: Modulo Iterador

con múltiples vistas sobre un *heap* compartido que sufre modificaciones. En el ejemplo (a), la primera aplicación de la GFR tiene como hipótesis:

- $(\mathbf{coll}.c.ls.[x_1, x_0] \multimap\!\!\circledast\; \mathbf{coll}.c.ls.[x_1, x_0]) \,*\, \mathbf{coll}.c.ls.[x_1, x_0] \Rightarrow \mathbf{coll}.c.ls.[x_1, x_0]$
- $\mathbf{iter}.i,1.c.ls.[x_1, x_0] \Rightarrow (\mathbf{coll}.c.ls.[x_1, x_0] \,*\, \mathbf{true})$

que son satisfechas fácilmente a partir de las especificaciones ς_1 y ς_3. Por otro lado la segunda aplicación requiere una hipótesis de la forma:

- $(\mathbf{coll}.c.ls.[x_1, x_0] \multimap\!\!\circledast\; \mathbf{iter}.i,1.c.ls.[x_1, x_0]) \,*\, (\exists ls \cdot \mathbf{coll}.c.ls.[x_2, x_1, x_0]) \Rightarrow P$

para alguna formula P. Esta hipótesis no se puede satisfacer de ninguna manera ya que el *heap* compartido es modificado estructuralmente por el comando *add*. Por lo tanto no hay otra alternativa que escoger P como **true**. Esto refleja la perdida de información sobre el iterador por la inconsistencia introducida con la modificación.

El ejemplo (b), donde suponemos $\#xs > N$ para N una constante positiva, muestra casos *exitosos* de la aplicación de GFR incluso bajo modificaciones de la estructura compartida. Las hipótesis de la GFR se resuelven con las especificaciones estáticas. Cabe llamar la atención sobre la facilidad para introducir múltiples vistas sobre el *heap* compartido, y la simpleza con la que la GFR junto a las especificaciones estáticas codifican las modificaciones permitidas al mismo, y el cambio en el estado global que esto produce.

3.2.2. *Shared Buffer*

El siguiente caso esta inspirado en un ejemplo de [1]. Supongamos que disponemos de dos módulos que se comunican de manera asíncrona a través de un *buffer* compartido. Dicho escenario es muy común, por ejemplo, en la implementación de la pila de un protocolo de comunicación. Denotemos con $\mathbf{prod}.p.b.xs$ al modulo que produce la información, y $\mathbf{cons}.c.b.xs$ al modulo que la consume, donde xs representa los valores del *buffer*. Una especificación común de la interacción entre estos dos módulos permite que ambos modifiquen el buffer compartido, agregando o quitando elementos. Dependiendo de la implementación del *buffer*, esta modificación puede ser estructural o no.

En las figuras 19 (a) y (b) se presentan las definiciones esquemáticas de ambos módulos. Para nuestra elección de **buff** las modificaciones en *produce* y *consume* son estructurales. Notar que aunque comparten la información sobre la definición del predicado de abstracción **buff** los módulos no necesita conocer los detalles de la implementación del otro. Un esquema de cliente de dichos módulos se presenta en la figura 19 (c). Nuevamente las condiciones de aplicación de la GFR se resuelven fácilmente con las especificaciones estáticas.

(a)

$\{\,\mathbf{emp}\,\}$
$\quad c := new_coll();$
$\{\,\mathbf{coll}.c.[\,].[\,]\,\}$
$\quad add(c, x_0);$
$\quad add(c, x_1);$
$\{\,(\exists ls \;\cdot\; \mathbf{coll}.c.ls.[x_1, x_0])\,\}$
$\Rightarrow \langle$ Elemento neutro $\rangle$
$\{\,(\exists ls \;\cdot\; \mathbf{coll}.c.ls.[x_1, x_0] \;\langle * : \mathbf{coll}.c.ls.[x_1, x_0]\rangle\; \mathbf{coll}.c.ls.[x_1, x_0])\,\}$
VAR:
$\quad \{\,\mathbf{coll}.c.ls.[x_1, x_0] \;\langle * : \mathbf{coll}.c.ls.[x_1, x_0]\rangle\; \mathbf{coll}.c.ls.[x_1, x_0]\,\}$
$\quad$ GFR:
$\quad\quad \{\,\mathbf{coll}.c.ls.[x_1, x_0]\,\}$
$\quad\quad\quad i := new_iter(c);$
$\quad\quad \{\,\mathbf{iter}.i,0.c.ls.[x_1, x_0]\,\}$
$\quad\quad\quad next(i);$
$\quad\quad \{\,\mathbf{iter}.i,1.c.ls.[x_1, x_0]\,\}$
$\quad \{\,(\mathbf{iter}.i,1.c.ls.[x_1, x_0]) \;\langle * : \mathbf{coll}.c.ls.[x_1, x_0]\rangle\; \mathbf{coll}.c.ls.[x_1, x_0]\,\}$
$\quad$ GFR:
$\quad\quad \{\,\mathbf{coll}.c.ls.[x_1, x_0]\,\}$
$\quad\quad\quad add(c, x_2);$
$\quad\quad \{\,(\exists ls \;\cdot\; \mathbf{coll}.c.ls.[x_2, x_1, x_0])\,\}$
$\quad \{\,\mathbf{true} * (\exists ls \;\cdot\; \mathbf{coll}.c.ls.[x_2, x_1, x_0])\,\}$
$\{\,(\exists ls \;\cdot\; \mathbf{true} * (\exists ls \;\cdot\; \mathbf{coll}.c.ls.[x_2, x_1, x_0]))\,\}$
$\Rightarrow \langle$ calculo de predicados $\rangle$
$\{\,(\exists ls \;\cdot\; \mathbf{true} * \mathbf{coll}.c.ls.[x_2, x_1, x_0])\,\}$

(b)

$\{\,\mathbf{coll}.c.ls.xs \;\langle * : \mathbf{coll}.c.ls.xs\rangle\; \mathbf{iter}.i_1,0.c.ls.xs\,\}$
GFR:
$\quad \{\,\mathbf{coll}.c.ls.xs\,\}$
$\quad\quad i_0 := new_iter(c);$
$\quad \{\,\mathbf{iter}.i_0,0.c.ls.xs\,\}$
$\quad\quad next(i_0);$
$\quad \{\,\mathbf{iter}.i_0,1.c.ls.xs\,\}$
$\{\,\mathbf{iter}.i_0,1.c.ls.xs \;\langle * : \mathbf{coll}.c.ls.xs\rangle\; \mathbf{iter}.i_1,0.c.ls.xs\,\}$
GFR:
$\quad \{\,\mathbf{iter}.i_1,0.c.ls.xs\,\}$
$\quad\quad next(i_1);$
$\quad \{\,\mathbf{iter}.i_1,1.c.ls.xs\,\}$
$\quad\quad x := next(i_1);$
$\quad \{\,\mathbf{iter}.i_1,2.c.ls.xs \wedge x = xs{,}1\,\}$
$\quad\quad set(i_1, x);$
$\quad \{\,\mathbf{iter}.i_1,2.c.ls.(xs[xs,1/2])\,\}$
$\left\{\begin{array}{l}\mathbf{iter}.i_0,1.c.ls.(xs[xs,1/2]) \;\langle * : \mathbf{coll}.c.ls.(xs[xs,1/2])\rangle \\ \mathbf{iter}.i_1,2.c.ls.(xs[xs,1/2])\end{array}\right\}$
GFR:
$\quad \{\,\mathbf{iter}.i_0,1.c.ls.(xs[xs,1/2])\,\}$
$\quad\quad y := next(i_0);$
$\quad \{\,\mathbf{iter}.i_0,2.c.ls.(xs[xs,1/2]) * y \doteq xs{,}1\,\}$
$\left\{\begin{array}{l}(\mathbf{iter}.i_0,2.c.ls.xs[xs,1/2] * y \doteq xs{,}1) \;\langle * : \mathbf{coll}.c.ls.(xs[xs,1/2])\rangle \\ \mathbf{iter}.i_1,2.c.ls.(xs[xs,1/2])\end{array}\right\}$

Figura 18: Código cliente del modulo iterador

(a)

Λ:
$$\mathbf{buff}.b.[\,] \triangleq b = \mathbf{nil} \wedge \mathbf{emp}$$
$$\mathbf{buff}.b.(x \triangleright xs) \triangleq (\exists i \;\cdot\; b \mapsto x, i * \mathbf{buff}.i.xs)$$
$$\mathbf{prod}.p.b.xs \triangleq p \mapsto b, \ldots * \mathbf{buff}.b.xs$$

Γ: $\{\mathbf{prod}.p.b.xs\}\; produce(p)\; \{\mathbf{prod}.p.b.(x \triangleright xs)\}$

Σ:
$\varsigma_1 : [(\mathbf{buff}.b.xs \mathrel{-\!\circledast} \mathbf{prod}.p.b.xs) * \mathbf{buff}.b.xs' \Rightarrow \mathbf{prod}.p.b.xs']$
$\varsigma_2 : [\mathbf{prod}.p.b.xs \Rightarrow (\mathbf{buff}.b.xs * \mathbf{true})]$

(b)

Λ:
$$\mathbf{buff}.b.[\,] \triangleq b = \mathbf{nil} \wedge \mathbf{emp}$$
$$\mathbf{buff}.b.(x \triangleright xs) \triangleq (\exists i \;\cdot\; b \mapsto x, i * \mathbf{buff}.i.xs)$$
$$\mathbf{cons}.c.b.xs \triangleq c \mapsto b, \ldots * \mathbf{buff}.b.xs$$

Γ: $\{\mathbf{cons}.c.b.(xs \triangleright x)\}\; consume(c)\; \{\mathbf{cons}.c.b.xs\}$

Σ:
$\varsigma_3 : [(\mathbf{buff}.b.xs \mathrel{-\!\circledast} \mathbf{cons}.p.b.xs) * \mathbf{buff}.b.xs' \Rightarrow \mathbf{cons}.p.b.xs']$
$\varsigma_4 : [\mathbf{cons}.p.b.xs \Rightarrow (\mathbf{buff}.b.xs * \mathbf{true})]$

(c)

$\{\,\mathbf{prod}.p.b.xs \;\langle * : \mathbf{buff}.b.xs\rangle\; \mathbf{cons}.c.b.xs\,\}$
GFR:
$\quad \{\,\mathbf{prod}.p.b.xs\,\}$
$\quad\quad produce(p, x);$
$\quad \{\,\mathbf{prod}.p.b.(x : xs)\,\}$
$\{\,\mathbf{prod}.p.b.(x \triangleright xs) \;\langle * : \mathbf{buff}.b.(x \triangleright xs)\rangle\; \mathbf{cons}.c.b.(x \triangleright xs)\,\}$
$\vdots$
$\{\,\mathbf{prod}.p.b.(y \triangleright ys) \;\langle * : \mathbf{buff}.b.(y \triangleright ys)\rangle\; \mathbf{cons}.c.b.(y \triangleright ys)\,\}$
GFR:
$\quad \{\,\mathbf{cons}.c.b.(y : ys)\,\}$
$\quad\quad consume(c);$
$\quad \{\,\mathbf{cons}.c.b.ys\,\}$
$\{\,\mathbf{prod}.p.b.ys \;\langle * : \mathbf{buff}.b.ys\rangle\; \mathbf{cons}.c.b.ys\,\}$

Figura 19: Esquema productor-consumidor con buffer compartido

3.3. Semántica

3.3.1. Predicados de Abstracción y Validez de formulas

Para dar semántica a las formulas que involucran predicados de abstracción es necesario extender la relación de semántica $\models$ respecto a un *entorno semántico de predicados* Δ. Dado L un conjunto de nombres de predicados, un entorno semántico de predicados se define como el siguiente producto indexado:

$$\Delta : \prod \alpha \in L \cdot (Values^{arity(\alpha)} \rightarrow \mathbb{P}(Heaps))$$

que asigna a cada predicado una función de la aridad correcta. De esta manera podemos definir la semántica de un predicado de abstracción como:

$$(s, i, h) \models_\Delta \alpha(\overline{e}) \;\Leftrightarrow\; \alpha \in dom_\Delta \wedge h \in \Delta_\alpha.(\overline{[\![e]\!]_{s,i}})$$

La semántica del resto de las formulas se extiende.

Demostramos brevemente a continuación que es posible construir un entorno semántico de predicados Δ a partir de una secuencia de definiciones Λ siempre que se cumpla que para todo $\alpha(\overline{x}) = P$ en Λ, P es una formula positiva. Para presentar el procedimiento de construcción es necesario introducir

distintos resultados. Sin embargo, dado que esto no es parte de la contribución del presente articulo, solo mencionaremos las nociones estrictamente necesarias. El lector puede encontrar los detalles en [17] y [18].

Respecto a los entornos semánticos de predicados es posible definir una relación de orden $\sqsubseteq$ como la extensión puntual de la relación de inclusión $\subseteq$ de *heaps* para cada predicado α:

$$\Delta \sqsubseteq \Delta' \triangleq \forall \alpha, \overline{e} \in Values^{arity(\alpha)} \cdot \Delta_\alpha \neq \bot \;\Rightarrow\; \Delta_\alpha.\overline{e} \subseteq \Delta'_\alpha.\overline{e}$$

De esta manera una formula positiva P es monótona respecto a entornos semánticos de predicados:

$$\Delta \sqsubseteq \Delta' \wedge (s,i,h) \models_\Delta P \;\Rightarrow\; (s,i,h) \models_{\Delta'} P$$

Los entornos semánticos forman un reticulado completo respecto a $\sqsubseteq$ cuya supremo denotamos con $\sqcup$.

Un entorno abstracto Λ no necesariamente incluye una definición para cada predicado. Por lo tanto, para construir un entorno semántico Δ a partir de Λ puede ser necesario contar con definiciones semánticas adicionales Δ'. La construcción del entorno semántico se realiza a través de iteraciones sucesivas Δ^n de la siguiente función:

$$step_{\Delta',\Lambda}(\Delta^n) \triangleq \lambda(\alpha(\overline{x}) \triangleq P) \in \Lambda \cdot \lambda\overline{e} \in Values^{arity(\alpha)} \cdot \{h \mid (s,i,h) \models_{\Delta^n \sqcup \Delta'} P_{/\overline{x} \leftarrow \overline{e}}\}$$

Esta función es monótona en el reticulado de los entornos semánticos, por lo tanto tiene un mínimo punto fijo que denotamos con $[\![\Lambda]\!]_\Delta$. Así tomamos como solución de un entorno abstracto Λ a cada entorno semántico en el siguiente conjunto:

$$close(\Lambda) \triangleq \{\Delta \cup [\![\Lambda]\!]_\Delta \mid dom_\Delta = L/dom_\Lambda\}$$

De esta manera podemos definir formalmente la validez de una formula P respecto a un entorno de predicados abstracto Λ y un conjunto de especificaciones estáticas Σ denotado por $\Lambda;\Sigma \models P$ como:

$$(\forall s,i,h,\Delta \in close(\Lambda), [\varsigma] \in \Sigma \cdot (s,h,i) \models_\Delta \varsigma) \;\Rightarrow\; (\forall s,i,h,\Delta \in close(\Lambda) \cdot (s,h,i) \models P)$$

Lema 8 *Los siguientes axiomas son validos*

$$\alpha(\overline{x}) \triangleq P, \Lambda;\Sigma \models \alpha(\overline{e}) \;\equiv\; P_{/\overline{e} \rightarrow \overline{x}}$$
$$\Lambda;[P \;\Rightarrow\; Q],\Sigma \models P_{/\overline{e} \rightarrow \overline{x}} \;\Rightarrow\; Q_{/\overline{e} \rightarrow \overline{x}}$$

3.3.2. Lenguaje de programación y especificaciones

Para dar cuenta de los procedimientos en la semántica de los nuevos comandos es necesario extender la noción de configuración introducida en la sección 2.6.1. Sea K un conjunto de nombre de procedimientos, definimos un *entorno de procedimientos* $\Pi : K \rightharpoonup (\overline{PVar}, C)$ como una función parcial de nombre de procedimientos a un par de nombres de variables y comando. Decimos que un entorno esta bien formado, denotado por Π **ok**, si el cuerpo de cada procedimiento solo modifica variables locales. Ahora extendemos las configuraciones incluyendo un entorno de procedimientos bien definido. Para los comandos heredados, la semántica operacional se extiende de forma trivial; para los nuevos, se presenta en la figura 20.

Lema 9 *Las resultados de* Safety *y* Termination Monotonicity *y* Propiedad de Frame *son validos respecto a la semántica operacional extendida.*

Utilizamos la notación $\Lambda;\Gamma;\Sigma \models \{P\}\,C\,\{Q\}$ para denotar que bajo las hipótesis de que para cada entorno semántico de predicados solución de Λ, cada especificación de procedimiento es valida en el entorno de procedimientos; y cada especificación estática es valida, entonces la especificación $\{P\}C\{Q\}$ es verdadera en el sentido de *corrección parcial*:

$$\forall \Delta \in close(\Lambda), \Pi, [\varsigma] \in \Sigma \;\cdot\; \Pi\,\mathbf{ok} \wedge (\Delta \models_\Pi \Gamma) \wedge (\Delta \models [\varsigma]) \;\Rightarrow\; (\Delta \models_\Pi \{P\}\,C\,\{Q\})$$

donde

$$\begin{aligned}
\Delta \models_\Pi \Gamma \;&\Leftrightarrow\; \forall \{P\}\,k\,\{Q\} \in \Gamma \cdot \Pi.k = (\overline{x}, C) \;\Rightarrow\; \Delta \models_\Pi \{P\}\,C\,\{Q\} \\
\Delta \models [\varsigma] \;&\Leftrightarrow\; \forall s,i,h \cdot (s,i,h) \models_\Delta \varsigma \\
\Delta \models_\Pi \{P\}\,C\,\{Q\} \;&\Leftrightarrow\; \forall s,i,h,s',h' \cdot (s,i,h) \models_\Delta P \;\Rightarrow\; \langle c,(s,h),\Pi\rangle \text{ es } safe \;\wedge \\
&\qquad (\langle c,(s,h),\Pi\rangle \leadsto^* (s',h') \;\Rightarrow\; (s',i,h') \models_\Delta Q)
\end{aligned}$$

Teorema 10 ***(Soundness)***
Si $\Lambda;\Sigma;\Gamma \vdash \{P\}\,C\,\{Q\}$ *se deduce aplicando las reglas del sistema deductivo entonces se satisface* $\Lambda,\Sigma,\Gamma \models \{P\}\,C\,\{Q\}$.

- **Variable local**:

$$\frac{\langle c, ([s \mid x : \mathbf{nil}], h), \Pi \rangle \rightsquigarrow^* (s', h')}{\langle \mathbf{newvar}(x); c, (s, h), \Pi \rangle \rightsquigarrow^* ([s' \mid x : [x]_{exp}.s], h')}$$

- **Valor retorno:**

$$\langle \mathbf{return}(e), (s, h), \Pi \rangle \rightsquigarrow^* ([s \mid ret : [e]_{exp}.s], h)$$

- **Definición de modulo:**

$$\frac{\langle c, (s, h), [\Pi \mid k_1 : (\overline{x_1}, c_1) \mid \ldots \mid k_n : (\overline{x_n}, c_n)] \rangle \rightsquigarrow^* (s', h')}{\langle \mathbf{let}\ k_1\ \overline{x_1} = c_1, \ldots, k_n\ \overline{x_n} = c_n\ \mathbf{in}\ c, (s, h), \Pi \rangle \rightsquigarrow^* (s', h')}$$

- **Llamada a procedimiento:**

$$\frac{\langle c, ([s \mid \overline{x} : \overline{[y]_{exp}.s}], h), \Pi \rangle \rightsquigarrow^* (s', h') \quad \Pi.k = (\overline{x}, c)}{\langle x = k(\overline{y}), (s, h), \Pi \rangle \rightsquigarrow^* ([s \mid x : [ret]_{exp}.s'], h')}$$

Figura 20: Semántica operacional de los comandos

4. Conclusiones

4.1. Trabajos relacionados

Existen diversas extensiones de Separation Logic que intentar modelar diferentes formas abstracción, desde el soporte de simples o múltiples instancias de módulos imperativos ([16, 18, 3]) a complejos modelos de lenguajes orientados a objetos con soporte para herencia, *dispatch* dinámico, etc. ([19, 8, 13]). 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 ninguna de estas propuestas logra dar cuenta de la interacción entre múltiples instancias de diferentes tipos abstractos de datos cuando comparten la memoria para su representación concreta.

En [17] Parkinson combina la noción de *abstract predicate* y la extensión de Separation Logic con *permisos* ([6, 5]) dentro de un marco de trabajo que da cuenta de un core subset 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.

En un trabajo no publicado [11] Krishnaswami *et al.* tratan el problema de especificar abstracta y modularmente el comportamiento de *sharing* de módulos imperativos. Utilizan una versión de Separation Logic de orden superior sobre un lenguaje funcional tipado de orden superior con referencias mutables. El soporte de la abstracción de datos esta dado por predicados cuantificados existencialmente, y el comportamiento de *sharing* de los módulos esta dado por su propio concepto de *static specification*. A pesar de la distancia en el aspecto técnico, se presentan grandes similitudes en el tratamiento de nuestro objetivo, aunque con algunas diferencias remarcables. Mientras que nuestro lenguaje nos permite especificar simultáneamente múltiples vistas sobre el *heap*, la técnica presentada en el citado articulo 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. Esto obliga a que los tipos abstractos de datos conozcan todos los detalles de la implementación mutuamente. Por el contrario en nuestra propuesta solo es necesario que los diversos módulos conozcan los detalles de la porción de representación compartida. En este sentido no parece posible especificar en su sistema nuestro ejemplo de *shared buffer* sin que el productor y consumidor conozcan todos los detalles de implementación.

En [23] Vafeiadis *et al.* extienden la Separation Logic a un entorno concurrente combinándola con la metodología Rely/Guarantee [24]. Aunque los objetivos son completamente diferentes, el lenguaje de aserciones propuesto presenta grandes similitudes con el nuestro. Allí se utiliza la notación $\boxed{P}$ para distinguir una aserción 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 propuesto 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 de nuestra regla 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.

4.2. Contribuciones y trabajo futuro

En este trabajo presentamos una extensión a la Separtion Logic para razonar de manera modular sobre la interacción de estructuras abstractas con *sharing* en su representación. Para lidiar con la noción de abstracción optamos por concepto de abstract predicate 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, y está en nuestros planes desarrollar nuestras ideas en este camino.

De forma análoga creemos que nuestro concepto de especificación estática caracteriza de manera igualmente sencilla e intuitiva las posibilidades y restricciones en la manipulación de las estructuras abstractas mas allá de las especificaciones *dinamicas* de los procedimientos. Esto abre las posibilidades de *sharing* entre las estructuras hasta limite de flexibilidad que el desarrollador del modulo este dispuesto a brindar a través de la elección de los predicados de abstracción utilizados en las especificaciones.

El lenguaje de aserciones nos permite especificar de manera precisa diversas estructuras complejas en el *heap*, incluyendo relaciones de *sharing* entre ellas, de manera compatible con los principios de abstracción y ocultamiento de información. Resulta particularmente sencillo especificar simultáneamente las múltiples vistas sobre el estado global y la relaciones entre ellas.

La Regla de Frame General nos permite razonar independientemente con estas estructuras de manera elegante. Las hipótesis necesarias para su aplicación, aunque complejas, caracterizan con una formula dentro de la misma lógica las condiciones mínimas para garantizar la consistencia de las estructuras que comparten el *heap* para su representación. Resulto una grata sorpresa encontrar los errores en la especificación del problema del fringe a partir del intento de demostrar las condiciones de la GFR para una especificación similar a la original. 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. La combinación de la GFR junto a las especificaciones *locales* de los procedimientos y las especificaciones estáticas nos permiten extender las posibilidades del razonamiento local, modular y abstracto mas allá de todas las extensiones actuales de la Separation Logic.

Las posibilidades de trabajo futuro se presentan en variadas direcciones. Por un lado seria interesante explorar los aspectos teóricos de nuestra lógica en el marco de trabajo de las *BI-Hyperdoctrines*, y analizar los aspectos de completitud del sistema de prueba.

Como mencionamos anteriormente, una extensión inmediata es la adaptación de nuestra teoría a un lenguaje orientado a objetos mas 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 esta dentro de nuestros planes implementar nuestro marco de trabajo en alguna de las herramientas (semi)automáticas existentes ([2, 12]).

Referencias

[1] M. Barnett and D. A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *MPC*, pages 54–84, 2004.

[2] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 2005.

[3] B. Biering, L. Birkedal, and N. Torp-Smith. Bi-hyperdoctrines, higher-order separation logic, and abstraction. *ACM Trans. Program. Lang. Syst.*, 29(5):24, 2007.

[4] L. Birkedal, N. Torp-Smith, and J. C. Reynolds. Local reasoning about a copying garbage collector. *SIGPLAN Not.*, 39(1):220–231, 2004.

[5] R. Bornat, C. Calcagno, P. W. O'Hearn, and M. J. Parkinson. Permission accounting in separation logic. In J. Palsberg and M. Abadi, editors, *POPL*, pages 259–270. ACM, 2005.

[6] J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *SAS*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer, 2003.

[7] C. Calcagno, P. Gardner, and U. Zarfaty. Context logic as modal logic: completeness and parametric inexpressivity. In M. Hofmann and M. Felleisen, editors, *POPL*, pages 123–134. ACM, 2007.

[8] W.-N. Chin, C. David, H. H. Nguyen, and S. Qin. Enhancing modular oo verification with separation logic. *SIGPLAN Not.*, 43(1):87–99, 2008.

[9] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[10] S. S. Ishtiaq and P. 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.

[11] N. Krishnaswami, L. Birkedal, J. Aldrich, and J. Reynolds. Idealized ml and its separation logic, 2006.

[12] N. Marti and R. Affeldt. A certified verifier for a fragment of separation logic. *Computer Software*, 2008.

[13] R. Middelkoop, K. Huizing, and R. 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.

[14] P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.

[15] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In L. Fribourg, editor, *CSL*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001.

[16] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In N. D. Jones and X. Leroy, editors, *POPL*, pages 268–280. ACM, 2004.

[17] M. J. Parkinson. Local reasoning for Java. Technical Report UCAM-CL-TR-654, University of Cambridge, Computer Laboratory, Nov. 2005.

[18] M. J. Parkinson and G. M. Bierman. Separation logic and abstraction. In J. Palsberg and M. Abadi, editors, *POPL*, pages 247–258. ACM, 2005.

[19] M. J. Parkinson and G. M. Bierman. Separation logic, abstraction and inheritance. In G. C. Necula and P. Wadler, editors, *POPL*, pages 75–86. ACM, 2008.

[20] J. Reynolds. Precise, intuitionistic, and supported assertions in separation logic, 2005.

[21] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.

[22] C. C. Richard Bornat and P. O'Hearn. Local reasoning, separation and aliasing.

[23] V. Vafeiadis and M. J. Parkinson. A marriage of rely/guarantee and separation logic. In L. Caires and V. T. Vasconcelos, editors, *CONCUR*, volume 4703 of *Lecture Notes in Computer Science*, pages 256–271. Springer, 2007.

[24] Q. Xu, W. P. de Roever, and J. He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Asp. Comput.*, 9(2):149–174, 1997.

[25] H. Yang. An example of local reasoning in bi pointer logic: the schorr-waite graph marking algorithm, 2000.

[26] H. Yang. *Local reasoning for stateful programs.* PhD thesis, Champaign, IL, USA, 2001. Adviser-Uday S. Reddy.

[27] H. Yang and P. W. O'Hearn. A semantic basis for local reasoning. In M.Ñielsen and U. Engberg, editors, *FoSSaCS*, volume 2303 of *Lecture Notes in Computer Science*, pages 402–416. Springer, 2002.