UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

Verificación formal de programas con lógicas de separación

TESIS

Que para obtener el título de

Ingeniero en Computación

P R E S E N T A

Diego Roberto Medina Martínez

DIRECTOR DE TESIS

Dr. Ismael Everardo Bárcenas Patiño

Ciudad Universitaria, Cd. Mx., 2021

Agradecimientos

A mis papás y hermanos, por todo el amor, dedicación, apoyo, confianza y sacrificio recibido a lo largo de la carrera con el fin de culminarla exitosamente y alcanzar las metas propuestas durante este proceso de formación.

Al Dr. Everardo Bárcenas, por brindarme la oportunidad, el tiempo y los conocimientos necesarios para guiarme en el desarrollo del presente trabajo de tesis.

Al personal académico de la Facultad de Ingeniería por las valiosas experiencias y conocimiento otorgados.

A los amigos de la Facultad de Ingeniería que tuve el gusto de conocer, por su apoyo durante la estancia en la misma.

Investigación realizada gracias al Programa de Apoyo a Proyectos de Investigación e Innovación Tecnológica (PAPIIT) de la UNAM a través del proyecto IA105420, de nombre "Caracterización modal de lógicas de separación". Agradezco a la DGAPA-UNAM la beca recibida.

Índice general

Índice de figuras

Índice de tablas

Introducción

A lo largo del tiempo, la tecnología se ha integrado a nuestra vida diaria como un medio para automatizar todo tipo de tareas, es así que en algunas de estas se involucran actividades capaces de impactar en la seguridad de las personas. En vista de ello, antes de liberar un sistema de cómputo al público se deben realizar pruebas con el fin de identificar errores potenciales introducidos durante la etapa de desarrollo. Para llevarlo a cabo de manera eficaz, surgen métodos matemáticos que permiten describir y verificar el comportamiento de un sistema de forma precisa, en consecuencia, resulte menos propenso a fallos en comparación con los métodos tradicionales.

Existen trabajos en los que se plantea la verificación formal de sistemas de uso cotidiano, por ejemplo, el análisis formal del kernel de un sistema operativo [20] o una plataforma digital en sistemas financieros [28]. Sin embargo, también se contemplan situaciones críticas como lo es el control de sistemas de trenes [29], sistemas de aviación [32], aeroespaciales [33] y sistemas cruciales en plantas nucleares [34], en donde la vida de las personas está directamente relacionada con su correcta operación. Por otro lado, con la llegada de la cuarta revolución industrial se integra cada vez más infraestructura novedosa que también resulta importante analizar, por ejemplo, el sistema de conducción autónoma de vehículos [18] o el de pilotaje de naves no tripuladas tipo *dron* [17] por mencionar algunos.

Para realizar la verificación de los sistemas existen diversos métodos formales, cada uno provee ventajas sobre otros. En el caso de la lógica de Hoare [14] favorece la creación de especificaciones basadas en aserciones, las cuales a través de un lenguaje de lógica clásica permiten determinar si se procede a la ejecución de un fragmento de código específico y si dicho código se comporta de la manera esperada. Sin embargo, el principal problema se presenta a causa de su lenguaje, ya que es incapaz de especificar adecuadamente el comportamiento de estructuras de datos que manipulan la memoria a través de punteros.

Derivado de ese problema se desarrollan las lógicas de separación como una extensión a la lógica de Hoare [30]. Se incorporan dos nuevos operadores capaces de especificar los estados de la memoria y al mismo tiempo asegurar que se encuentran disjuntos. Como beneficio, se resuelve el problema de especificar estructuras que manipulan apuntadores, debido a que se introduce un modelo descriptivo de la memoria. Además, abre paso al análisis aislado de fragmentos de código, lo que en consecuencia favorece la ejecución del análisis de manera secuencial. Sin em-

bargo, dependiendo de la complejidad del código aún es necesario escribir en casos específicos las aserciones de manera manual.

Inicialmente en [4] se describe Infer como una de las primeras herramientas que implementan las lógicas de separación para la verificación de programas inicialmente con soporte para el lenguaje C. Con el fin de resolver el problema y automatizar la generación de las especificaciones del programa, deciden incorporar un método novedoso conocido como bi-abducción [6]. La bi-abducción es la técnica en la que se infieren las especificaciones del programa y a través de un algoritmo se determina la opción adecuada para determinadas instrucciones de código, proporcionando así un ahorro de tiempo a los desarrolladores que desean verificar un sistema.

En el año 2013, Facebook observa el gran impacto que las lógicas de separación y la bi-abducción poseen, por lo que decide adquirir la empresa Monoidics que desarrolla Infer. Posteriormente, dadas las prioridades de Facebook [5], se incluye soporte para los lenguajes Java, C++ y Objective-C, sin embargo, las lógicas de separación no se limitan a estos lenguajes, sino que es totalmente viable adaptarlo a cualquier otro si así se requiere.

Dos años más tarde, la empresa asigna una licencia de código libre a dicha herramienta, con el fin de que cualquier persona pueda contar con acceso al proyecto y realizar aportaciones. En la presente tesis, la principal motivación consiste en describir las lógicas de separación en contraste al enfoque clásico, así como su aplicación concreta en escenarios reales. Algunos resultados descritos en esta tesis se publicaron primero en [25] y [24].

En el Capítulo 1 se presentan los antecedentes que ejercen de base para la verificación de programas con lógicas de separación. Posteriormente, en el Capítulo 2 se describen las lógicas de separación a mayor detalle, como la sintaxis, semántica y algunas reglas derivadas de ésta, así como los principios de la generación de especificaciones del programa empleando la bi-abducción. Por otro lado, en el Capítulo 3 se aborda el uso de la implementación de las lógicas de separación, incluyendo algunos experimentos aislados, así como la verificación de un sistema, en este caso, la verificación de una sección de código perteneciente a un gestor de bases de datos conocido como MariaDB. Finalmente, en el Capítulo 4 se muestra el análisis y conclusiones de los resultados obtenidos, así como la definición del trabajo a futuro.

Capítulo 1

Antecedentes

La lógica cumple un papel muy importante en la verificación de programas. En este capítulo se abordarán los principios del razonamiento que preceden a las lógicas de separación.

1.1. Lógica de primer orden

Sintaxis

En la lógica de primer orden se emplean las conectivas lógicas clásicas: $\wedge$, $\vee$, $\neg$, $\Rightarrow$ y $\iff$, al igual que en la lógica proposicional, sin embargo, su alfabeto incluye además nuevos símbolos, con el objetivo de definir de manera precisa las sentencias lógicas como se especificará más adelante.

Definición 1.1. El alfabeto de la lógica de primer orden se compone por la tupla $(\mathcal{V}, \mathcal{P}, F)$, el primer argumento $\mathcal{V}$ corresponde a un conjunto de variables, $\mathcal{P}$ a un conjunto de símbolos de predicado y F a un conjunto de símbolos de funciones.

- Se asume que las variables son un conjunto infinito $\mathcal{V} = \{x_0, x_1, x_2, \ldots\}$ y se escriben con letras minúscula.
- Los símbolos de funciones $F = \{f, g, h, \ldots\}$ poseen un número de aridad que corresponde a la cantidad de términos que reciben.
- Los símbolos de predicado $\mathcal{P} = \{P, Q, R, \ldots\}$ poseen un número de aridad relacionado a los términos que reciben.

Definición 1.2. Un conjunto de términos $T = \{t_1, t_2, t_3, \ldots\}$ corresponde a la unión de los conjuntos de variables $\mathcal{V}$ y símbolos de función F. Un símbolo de función de aridad $n > 0$ puede recibir n términos. $f(t_1, t_2, t_3, \ldots, t_n)$ y a su vez se puede convertir en término, como se muestra en la Figura 1.1.

Por otro lado, en el alfabeto también se encuentran símbolos auxiliares como los paréntesis "(", ")" y la coma ",". Además, se definen dos nuevas conectivas lógicas, el cuantificador universal "$\forall$" y el cuantificador existencial "$\exists$".

Definición 1.3. Una fórmula en lógica de primer orden se define de manera inductiva como se muestra en la Figura 1.2.

$$
\begin{array}{ll}
t := x_0 & \text{variable} \\
\mid a & \text{constante (función de aridad cero)} \\
\mid f(t_1, t_2, \ldots, t_n) & \text{función de aridad } n > 0
\end{array}
$$

Donde $x_0 \in V, g \in F$ y $f(t_1, t_2, \ldots, t_n) \in F$.

Figura 1.1: Sintaxis de los términos en lógica de primer orden.

La precedencia de las conectivas lógicas y los cuantificadores se define en el siguiente orden: $\forall, \exists, \neg, \wedge, \vee, \Rightarrow, \iff$ (estas son asociadas de derecha a izquierda). Por ejemplo, $\phi \wedge \psi \wedge \omega$ es equivalente a $(\phi \wedge (\psi \wedge \omega))$.

Ejemplo 1.1. Si L corresponde a un lenguaje de primer orden correspondiente a nombres de personas relacionadas por un vínculo, $C = \{Fernando,\ Sofía,\ Juan,\ Vicente,\ Pamela\}$, $F = \{padreDe,\ abuelaDe,\ madreDe\}$ y $\mathcal{P} = \{EsHijo,\ EsMadre,\ EsHermana\}$ entonces se pueden formar las siguientes fórmulas:

$$
EsHijo(Fernando, Juan) \quad EsHermana(Vicente, madreDe(Sofía)) \ \wedge \\
EsMadre(Sofía, Pamela)
$$

Un símbolo de predicado generalmente representa hechos, sin embargo, también puede interpretarse como preguntas en el caso que se desconozca el valor de verdad, por ejemplo, *Juan* es hijo de *Fernando* o ¿la madre de *Sofía* es hermana de *Vicente* y *Pamela* es la madre de *Sofía*?.

Semántica

Una estructura de primer orden asigna un significado al lenguaje al momento de interpretar términos o símbolos de predicado, a continuación se describirá de manera precisa.

Definición 1.4. Dada una estructura de primer orden S, ésta se compone de la tupla $S = (U, \mathcal{F}, \mathcal{R})$, donde:

- U es un conjunto de elementos no vacío denominado dominio o universo, puede ser un número finito o infinito de elementos y representa tanto elementos del mundo real como abstractos.
- $\mathcal{F}$ es un conjunto de funciones de aridad $n \geq 0$ que pertenecen a U.
- $\mathcal{R}$ es un conjunto de relaciones de aridad $n > 0$ que pertenecen a U.

Cuando un símbolo de predicado posee una aridad igual a uno se define una propiedad de un elemento del dominio, por ejemplo, $EsMetal(aluminio)$. Cuando la aridad es mayor a uno, se establece y se denomina como una relación entre dos o más elementos, por ejemplo, $EsCapital(Italia, Roma)$. Todo símbolo de función f con aridad $n = 0$ no depende de un elemento del dominio y puede ser equivalente a una constante, pero sigue perteneciendo al conjunto de los símbolos de función $\mathcal{F}$.

$\phi := P(t_1, t_2, \ldots, t_n)$	predicado de aridad n
$\mid \neg\phi$	negación
$\mid \phi \wedge \psi$	conjunción
$\mid \phi \vee \psi$	disyunción
$\mid \phi \Rightarrow \psi$	condicional
$\mid \phi \iff \psi$	bicondicional
$\mid \forall z\ \phi$	cuantificador universal
$\mid \exists z\ \phi$	cuantificador existencial

Donde $P \in \mathcal{P}, z \in V$ y $t_i \in T$ para $i = 1, \ldots, n$.

Figura 1.2: Sintaxis de una fórmula en lógica de primer orden.

Definición 1.5. Una evaluación E de acuerdo a una estructura de primer orden S se define como una función $E : \mathcal{V} \mapsto U$, donde a la variable le es asignada un valor del dominio U.

Definición 1.6. Una interpretación de un término T o de un símbolo de predicado $\mathcal{P}$ requiere de una estructura de primer orden S, y de la función de evaluación. Cada caso se describe de manera recursiva como se muestra a continuación:

- Cuando una variable es interpretada, le es asignado un valor perteneciente a un elemento de U de acuerdo a una estructura S.

$$[\![x]\!]^S_E = E(x)$$

- A un símbolo de función de aridad n le es asignada una función de aridad n de acuerdo a una estructura S, si $n > 0$ cada uno de sus términos son interpretados a elementos de S de manera independiente bajo la misma estructura.

$$[\![a]\!]^S_E = a^S \qquad [\![f(t_1, \ \ldots \ , t_n)]\!]^S_E = f^S\left([\![t_1]\!]^S_E, \ldots, [\![t_n]\!]^S_E\right)$$

  Donde a^S y f^S representan funciones de U sobre una estructura S.

- A un símbolo de predicado de aridad n le es asignado un predicado de aridad n de acuerdo a una estructura S, y cada uno de sus términos son interpretados a elementos de S de manera independiente bajo la misma estructura.

$$[\![P(t_1, \ \ldots \ , t_n)]\!]^S_E = P^S\left([\![t_1]\!]^S_E, \ldots, [\![t_n]\!]^S_E\right)$$

  Donde P^S representa relaciones de U sobre una estructura S.

Ejemplo 1.2. Si se define una estructura de primer orden $S = (U, \mathcal{F}, \mathcal{R})$, tal que $U = \{$Francia, París, Japón, Tokio, Suecia, Estocolmo, $\ldots\}$ es un conjunto de nombres de países y capitales; $\mathcal{F} = \{capitalDe\}$; $\mathcal{R} = \{ColindaCon, EsCapitalDe\}$. Una evaluación E de elementos de U bajo dicha estructura se representa como: $[\![capitalDe(Suecia)]\!]^S_E = Estocolmo$.

$$
\begin{aligned}
[\![P(t_1,\ldots,t_n)]\!]^S_E = 1 \text{ si y solo si } P^M([\![t_1]\!]^S_E,\ldots,[\![t_n]\!]^S_E) \neq \emptyset && \text{predicado de aridad } n>0\\
[\![\neg\phi]\!]^S_E = 1 \text{ si y solo si } [\![\phi]\!]^S_E = 0 && \text{negación}\\
[\![\phi \wedge \psi]\!]^S_E = 1 \text{ si y solo si } [\![\phi]\!]^S_E = 1 \text{ y } [\![\psi]\!]^S_E = 1 && \text{conjunción}\\
[\![\phi \vee \psi]\!]^S_E = 1 \text{ si y solo si } [\![\phi]\!]^S_E = 1 \text{ o } [\![\psi]\!]^S_E = 1 && \text{disyunción}\\
[\![\phi \Rightarrow \psi]\!]^S_E = 1 \text{ si y solo si } [\![\phi]\!]^S_E = 1, \text{ entonces } [\![\psi]\!]^S_E = 1 && \text{condicional}\\
[\![\phi \iff \psi]\!]^S_E = 1 \text{ si y solo si } [\![\phi]\!]^S_E = [\![\psi]\!]^S_E = 1 && \text{bicondicional}\\
[\![\exists x\phi]\!]^S_E = 1 \text{ si y solo si } [\![\phi]\!]^S_{E[^n/_x]} = 1 \text{ para algún } n \in U && \text{cuantificador existencial}\\
[\![\forall x\phi]\!]^S_E = 1 \text{ si y solo si } [\![\phi]\!]^S_{E[^n/_x]} = 1 \text{ para todo } n \in U && \text{cuantificador universal}
\end{aligned}
$$

Donde S es una estructura de primer orden, E una función de evaluación y $E[^n/_x]$ denota que $E(x) = n$.

Figura 1.3: Semántica de las fórmulas en lógica de primer orden.

Definición 1.7. Las fórmulas en lógica de primer orden se pueden interpretar de manera inductiva como se muestra en la Figura 1.3.

Ejemplo 1.3. Tomando la estructura S del Ejemplo 1.2, una fórmula ϕ se puede representar de la siguiente manera: $\phi : EsCapitalDe(Tokio, Japón) \wedge ColindaCon(Suecia, Noruega)$.

Finalmente, las equivalencias lógicas que se emplean en la lógica proposicional continúan siendo válidas en esta lógica. Por ejemplo: $\neg(\phi \Rightarrow \psi)$ *si y solo si* $(\phi \wedge \neg\psi)$, que será empleada como una manera de probar que una fórmula es inválida.

Satisfacción y Validez

Definición 1.8. Se tiene una estructura de primer orden S de un lenguaje en lógica de primer orden L, una fórmula ϕ y una evaluación E. Se dice que S satisface a ϕ si y solo si $[\![\phi]\!]^S_E = 1$. Por lo que S es un modelo de ϕ.

Definición 1.9. Dado un conjunto de fórmulas Γ, se dice que Γ se satisface si y solo si existe una estructura S y una evaluación E tal que $[\![\phi]\!]^S_E = 1$ dado que todas las fórmulas $\phi \in \Gamma$.

Definición 1.10. Para un conjunto de fórmulas Γ, se dice que son válidas si y solo si $[\![\phi]\!]^S_E = 1$ para todas las estructuras S que son modelos de ϕ, tal que $\phi \in \Gamma$. Por lo que una fórmula ϕ es válida si y solo si $\neg\phi$ no se satisface.

Ejemplo 1.4. Del ejemplo escrito en el libro [2] página 45, se obtiene el siguiente ejemplo. Se tiene una fórmula $\phi : (\forall x.\ P(x,x)) \Rightarrow (\exists x.\ \forall y.\ P(x,y))$ y se busca probar su invalidez.

Para mostrar que es inválida se debe encontrar una evaluación E tal que: $[\![\neg\phi]\!]^S_E = 1$.
Por lo que: $[\![\neg(\forall x.\ P(x,x)) \Rightarrow (\exists x.\ \forall y.\ P(x,y)))]\!]^S_E = 1$.

De acuerdo con la semántica del operador condicional, la fórmula anterior se reescribe como: $[\![(\forall x.\ P(x,x)) \wedge \neg(\exists x.\ \forall y.\ P(x,y))]\!]^S_E = 1$. Si se selecciona una estructura S cuyo dominio $U =$

$\{0,1\}$, $R = \{P\}$, $F = \{\emptyset\}$ y $P = \{(0,0),(1,1)\}$. Se dice que una relación $P(a,b)$ es verdadera si y solo si $(a,b) \in P$.

Por lo que si $P(x,x)$ se evalúa como $P(0,0)$ y $P(x,y)$ como $P(1,0)$, entonces se traduce como $[(\forall x.\ P(x,x)) \wedge \neg(\exists x.\ \forall y.\ P(x,y))]_E^S = 1$ y por lo tanto, se ha demostrado que la fórmula ϕ es inválida.

1.2. Teoría de las estructuras de datos recursivas

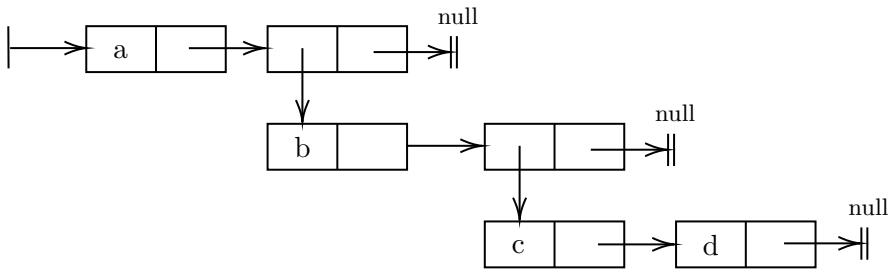
Una manera de razonar sobre las estructuras de datos es empleando la lógica de primer orden. Diversas teorías se han derivado, por ejemplo, la teoría de las listas acíclicas, la teoría de los arreglos, de las estructuras de datos acíclicas, entre otras. La teoría de las estructuras de datos recursivas permite describir listas, pilas, colas y árboles de todo tipo. El caso ideal es emplear múltiples teorías para definir las especificaciones de los programas. En esta sección se describirá la teoría de las listas simples.

Teoría de listas simples

Para definirla, se empleará una lista ligada similar a la sintaxis del lenguaje de programación LISP. La estructura S de este lenguaje está compuesta por los siguientes elementos: $F : \{lista,\ getd,\ geti,\ a,\ b,\ c, \ldots\}$, $P : \{Elemento,\ =\}$ y $V : \{x,\ y,\ z,\ \ldots\}$. El símbolo de función *lista* recibe dos elementos que representan el contenido de la lista, sin embargo, es posible expandirla a un número mayor de elementos. El símbolo de función *getd* de aridad uno obtiene el conjunto elementos derecho de una lista y *geti* el izquierdo. Por otro lado, las funciones de aridad cero $\{a,\ b,\ c,\ \ldots\}$ son constantes. Finalmente, el símbolo de predicado *Elemento* es de aridad uno, que resulta verdadero si el término corresponde a un único elemento de una lista simple.

Derivado de la teoría de la igualdad el símbolo de predicado "=" es verdadero si dos elementos son iguales. Su alfabeto se define como $Pred : \{=,\ p,\ q,\ r,\ \ldots\}$ correspondiente a símbolos de predicado, $Func : \{f,\ g,\ h,\ \ldots\}$ como símbolos de funciones y $Var : \{x_0,\ x_1,\ \ldots,\ y_0,\ y_1,\ \ldots\}$. El conjunto de axiomas se describe en la Figura 1.4.

Una vez que se ha definido el alfabeto es posible construir fórmulas de primer orden a partir de ello. Por ejemplo, para construir una lista con dos elementos se representa mediante $lista(x,\ y)$ tal que x es el primer elemento y y el segundo. Para una lista con más de dos elementos se puede escribir como: $lista(a,\ lista(b,\ lista(c,\ d)))$. A continuación se muestra de manera gráfica este último ejemplo.



$\forall x.\ x = x$ reflexividad

$\forall x, y.\ x = y \Rightarrow y = x$ simetría

$\forall x, y, z.\ x = y \wedge y = z \Rightarrow x = z$ transitividad

$\forall n. \mid n > 0 \in$ números enteros y cada símbolo de función, se tiene que:

$\forall x_1, \ldots, x_n, y_1, \ldots, y_n \bigwedge_{i=1}^{n} x_i = y_i \rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$ congruencia de la función

$\forall n. \mid n > 0 \in$ números enteros y cada símbolo de predicado, se tiene que:

$\forall x_1, \ldots, x_n, y_1, \ldots, y_n \bigwedge_{i=1}^{n} x_i = y_i \rightarrow (P(x_1, \ldots, x_n) \leftrightarrow P(y_1, \ldots, y_n))$ congruencia del predicado

Figura 1.4: Axiomas de la teoría de la igualdad.

Para obtener el elemento b de la lista anterior primero se emplea el símbolo de función *getd*, y como resultado devuelve una sublista $lista(b,\ lista(c,\ d))$. Posteriormente, con *geti* se obtiene el elemento de la izquierda, es decir, el elemento b. La fórmula final se escribe de la siguiente manera: $geti(getd(\ lista(a,\ lista(b,\ lista(c,\ d)))\ ))$.

Ejemplo 1.5. Dada la siguiente fórmula basada en la teoría de estructuras de datos recursivas verificar si es válida: $\phi : lista(a,\ b) \wedge getd(a) = getd(b) \wedge geti(a) = geti(b) \Rightarrow \neg Elemento(b)$.

Por el método de la contradicción, se asume que existe una evaluación E tal que $[\![\phi]\!]_E^S = 0$, donde S corresponde a la firma definida anteriormente para las estructuras de datos recursivas. De lo anterior se obtiene que $[\![lista(a,\ b) \wedge getd(a) = getd(b) \wedge geti(a) = geti(b)]\!]_E^S = 1$ y $[\![Elemento(b)]\!]_E^S = 0$.

Posteriormente, derivado de la congruencia de una función representada en la Figura 1.5, se afirma que: $[\![lista(getd(a),\ geti(a)) = lista(getd(b),\ geti(b))]\!]_E^S = 1$. Así, del axioma constructor de esa misma Figura, se obtiene que: $[\![\neg Elemento(b)]\!]_E^S = 1$ y esto se deriva en una contradicción, por lo que se afirma que la fórmula ϕ es válida bajo la estructura S.

1.3. Lógica de Hoare

La lógica de Hoare es un sistema formal propuesto en [14] para especificar programas de cómputo por medio de aserciones. Su sintaxis y semántica estan basadas en la lógica clásica para definir el comportamiento esperado de un programa y así asegurar el correcto funcionamiento. Parte de este sistema fue inspirado gracias al trabajo de Floyd [11], el cual define un método inductivo para probar la corrección mediante ejecución simbólica y diagramas de flujo.

Desde esa época, cuando los programas de cómputo se desarrollaban en lenguajes de programación de alto nivel (como Fortran y COBOL), se empleaban estructuras de datos complejas que incluían apuntadores para la manipulación de la memoria a bajo nivel. Sin embargo, no

$\forall x,\, y.\ getd(lista(x,\ y)) = x$	proyección derecha
$\forall x,\, y.\ geti(lista(x,\ y)) = y$	proyección izquierda
$\forall x,\, y.\ \neg Elemento(lista(x,\ y))$	átomo
$\forall x. \neg Elemento(x) \Rightarrow lista(getd(x),\ geti(x)) = x$	constructor
$\forall x_1,\, x_2,\, y_1,\, y_2.\ x_1 = x_2\ \wedge\ y_1 = y_2 \Rightarrow lista(x_1,\ y_1) = lista(x_2,\ y_2)$	
$\forall x,\, y.\ x = y \Rightarrow getd(x) = getd(y)$	
$\forall x,\, y.\ x = y \Rightarrow geti(x) = geti(y)$	congruencia de función

Figura 1.5: Axiomas de la teoría de listas simples.

existía en ese momento algún método formal que permitiera describirlas de forma precisa. Esto se debe a que el lenguaje de la lógica de Hoare no es lo suficientemente descriptivo. Por otro lado, si se deseaba verificar grandes cantidades de código, era prácticamente inviable, puesto que cada vez que se deseaba realizar un análisis al programa, después de alguna modificación al código, se requería ejecutar el análisis en su totalidad y esto consumía demasiado tiempo de procesamiento.

Tripleta de Hoare

La piedra angular para representar aserciones lógicas es la tripleta de Hoare, que define dos estados en el que se encuentra la memoria de un programa, una precondición y una post-condición. Un estado representa un conjunto de variables que, cuando son interpretadas, se obtienen los valores almacenados en memoria para ese mismo instante de tiempo. Se describe de la siguiente manera:

$$\{P\}\, C\, \{Q\} \tag{1.1}$$

La precondición P deberá ser verdadera antes de la ejecución de un comando C. Posteriormente, una post-condición Q será válida si después de ejecutar C, es verdadera. Un ejemplo sencillo se muestra a continuación:

$$\{y \mapsto 3\}\, x := y + 1\, \{x > 3\} \tag{1.2}$$

En este caso, el código a ejecutar C consiste en incrementar el valor de la variable y y almacenarlo en x. El estado inicial P en el que se encuentra la memoria antes de ejecutarlo indica que la variable y contiene el valor tres. Si se ejecuta dicho código, la post-condición Q se mantiene verdadera en cuanto al valor de la variable, puesto que el valor será mayor a tres.

Ese conjunto de pre- y post-condiciones pueden ser escritas en un programa para definir, por ejemplo, el comportamiento esperado de una función. Las condiciones que son escritas entre líneas de código son llamadas anotaciones o aserciones, y en conjunto se les conoce como especificaciones de programa o especificaciones funcionales. Pueden ser empleadas para verificar errores que se presentan en tiempo de ejecución, como lo son divisiones entre cero, indirecciones nulas, intentar acceder a un arreglo en una localidad de memoria inválida, entre algunas más que son descritas en el Capítulo 3.

```
while
    @F
    (⟨condición⟩) {
        ⟨cuerpo⟩
    }
```

(a) Ciclo while.

```
<inicialización>
while
    @F
    (⟨condición⟩) {
        ⟨cuerpo⟩
        ⟨incremento⟩
    }
```

(b) Equivalencia al ciclo for.

Figura 1.6: Anotaciones invariantes

Por otro lado, existe un tipo específico de anotaciones llamadas invariantes de ciclo (*loop invariants*). Estas se colocan antes de evaluar una condición de salida, ya sea en `while` o `for`, y se cumplen sin importar el número de iteraciones, de allí su nombre. En el libro [2] emplean el lenguaje `pi` para ejemplificar las especificaciones de un programa, las anotaciones son expresadas por el símbolo @ y es similar al lenguaje C, con algunos pequeños cambios. Para los siguientes ejemplos, se empleará la misma sintaxis y semántica definida en dicho trabajo, como se muestra en la Figura 1.6.

Así, $@F$ representa la invariante del ciclo, y esa anotación será válida al comienzo de cada iteración y al final de la ejecución de dicho bucle. El caso ideal, es que las aserciones se puedan generar de manera automática para poder razonar sobre cualquier código que pudiera ser escrito, sin embargo, en la lógica clásica es poco lo que se puede deducir, es por ello que se buscan extensiones o métodos alternos que ayuden a especificar un programa de manera aún más automatizada. A continuación se muestra una función escrita en C que posteriormente se especificará de manera manual.

Ejemplo 1.6. Con base en el código mostrado en la parte inferior del párrafo, se definirán las anotaciones pertinentes. Primeramente, la variable `x` recibida por parámetros de la función puede tomar cualquier valor entero. El ciclo `while` se ejecuta hasta que el índice `i` sea mayor o igual a diez. Finalmente si el ciclo no permanece ejecutándose de manera infinita, la función regresará un valor. En el código, a simple vista se pueden encontrar casos en que el valor recibido provoque un error como una división entre cero, es por ello que las anotaciones deben definirse, por ejemplo, para delimitar el valor que `x` puede tomar.

```c
int operation(int x) {
    int i = 0;
    while(i < 10) {
        i = i + (x / (x + i));
    }
    return i;
}
```

Como precondición, el valor de `x` debe ser mayor o igual a uno para que la función pueda operar en un tiempo finito, debido a que se desea que el valor de `i` vaya en incremento.

Por otro lado, la postcondición se cumplirá cuando la función retorne el valor de `i` si y solo si es mayor o igual a diez. Además, se debe definir una invariante de ciclo $@L$ que permita asegurar que el valor de `i` toma únicamente valores positivos, de acuerdo con la precondición dada. Finalmente, se debe incluir una aserción de tiempo de ejecución denotada por a $(x+i) \neq 0$, con el fin de evitar que el programa termine inesperadamente por una división entre cero. A continuación se muestra el código anotado:

```
@pre x ≥ 1
@post rv ↔ i ≥ 10
int operation(int x) {
    int i = 0;
    while
        @L : ∃i. 0 ≤ i < 10
        (i < 10) {
           @ (x + i) ≠ 0
           i = i + ( x / (x + i));
    }
    return i;
}
```

Ejemplo 1.7. En este otro ejemplo se muestra un programa previamente anotado. La función recibe el parámetro `n` y calcula la cantidad de números de la serie Fibonacci. Posteriormente, lo almacena en el arreglo `arr` y una vez que finaliza devuelve el resultado de la suma de dichos números de la serie.

```
@pre |arr| = n
@post rv > 0 ↔ n > 1 ∧ rv = 0 ↔ n ≤ 1
int fibonacci(int n, int arr[])
{
        int i, j, k, suma = 0;
        for
        @L : (∃i. 0 ≤ i < n) ∧ j < k
        (i = j = 0, k = 1; i < n; i++)
        {
                if(j != 0)
                        suma += j;
                @L : i < n
                arr[i] = j;
                j = k;
                @L : i < n
                k += arr[i];
        }
        return suma;
}
```

Como precondición, el número de elementos en la serie debe ser igual al tamaño del la variable `arr`, en caso contrario se accedería a una parte de la memoria fuera del arreglo y posiblemente

terminaría el programa inesperadamente. Por otra parte, la postcondición asegura que se regresará un valor mayor a cero si y solo si `n` es mayor a uno, de cualquier otro modo retornará cero. En este caso, la invariante del ciclo $@L : (\exists i.\ 0 \leq i < n) \wedge j < k$ protege el acceso a `arr`, además conociendo que la variable `j` siempre será menor que `k` inferido de que `k + = arr[i];` y que `arr[i] = j;` se incluye a $j < k$ en la invariante del ciclo. Finalmente, existen dos anotaciones que impiden el acceso a una localidad fuera del arreglo `arr`, que pueden resultar útiles si en el código dentro del ciclo se modificara el valor de `i`.

En resumen, la precondición de una función incluye únicamente los parámetros formales que recibe y se especifican de acuerdo al comportamiento que se espera. En cambio, la postcondición de una función relaciona el valor de retorno con variables libres y parámetros formales. Cuando dichas condiciones son fórmulas distintas de $\top$ se les conoce como pre- o post-condiciones no triviales, pero no son aceptables a menos que se trate de un caso muy específico, como cuando en el que el lenguaje de programación esté definido el concepto de métodos privados, como es el caso del lenguaje Java.

Existe un conjunto de axiomas de acuerdo con las instrucciones de código más comunes de los lenguajes imperativos. Por ejemplo, para definir la instrucción `skip`, la asignación $x := E$, donde E corresponde a una expresión, controles de flujo como `if-else` y ciclos como `while`. Cada una de estas se escribe como una tripleta de Hoare, como se muestran a continuación:

$$\frac{}{\{P\}\ skip\ \{P\}} \text{ Axioma de sentencia vacía} \qquad \frac{}{\{P[E/x]\}x := E\{P\}} \text{ Axioma de asignación} \tag{1.3}$$

$$\frac{\{P\}\ C_1\ \{Q\}\quad \{Q\}\ C_2\ \{R\}}{\{P\}\ C_1; C_2\ \{R\}} \text{ Regla de composición} \tag{1.4}$$

$$\frac{\{P \wedge A\}\ C_1\ \{Q\}\quad \{P \wedge \neg A\}\ C_2\ \{Q\}}{\{P\}\quad if(A)\ C_1\ else\ C_2\quad \{Q\}} \text{ Regla del condicional} \tag{1.5}$$

$$\frac{\{P \wedge S\}\ C\ \{P\}}{\{P\}\quad While(A)\ C\quad \{P \wedge \neg A\}} \text{ Regla del while} \tag{1.6}$$

Corrección de programas

Una vez que se generan las condiciones de verificación, se pueden comprobar dos tipos de corrección:

- **Corrección parcial**: Se prueban las especificaciones sin necesidad de que el programa termine su ejecución.
- **Corrección total**: Además de verificar que las especificaciones son válidas, se asegura que el programa termina.

Corrección parcial

Con el fin de asegurar la corrección de un programa se deben satisfacer las especificaciones que aseguren su correcto funcionamiento, sin importar si termina o no la ejecución. Dichas especificaciones también reciben el nombre de propiedades de correción parcial o de seguridad. Para realizar la corrección parcial existen distintos métodos formales, uno de ellos es el método de aserción inductiva (*inductive assertion method*) basado en el principio de la inducción matemática.

Para ese método, el código es separado en funciones individuales, luego cada función se divide en un conjunto de rutas básicas. Una ruta básica es una secuencia de código que puede iniciar con una precondición o una invariante de ciclo y terminar con una aserción, una postcondición de la función u otra invariante de ciclo. La regla principal es que sólo puede existir una invariante de ciclo por ruta básica, cada ruta genera sus propias condiciones de verificación basadas en las anotaciones para su posterior validación.

Ejemplo 1.8. Se posee una función `sumatoria` escrita en código C. Como parámetros recibe un arreglo `x` que contiene números enteros y el tamaño `tam` de dicho arreglo. Posteriormente se ejecuta un ciclo `for` el cual se encargará de almacenar en la variable `suma` cada uno de los números positivos enteros del arreglo `x`. Por último, retorna el resultado `vr` (valor de retorno). A continuación se muestra el fragmento de código anotado del cual se generarán las rutas básicas:

```
@Pre 0 ≤ tam < |x|
@Post vr ≥ 0
int sumatoria(int x[], int tam) {
    int i, suma;
    for
    @L : ∃i. 0 ≤ i < tam
    (i = tam, suma = 0; i > 0; i--)
        if(x[i] >= 0)
            suma += x[i];
    return suma;
}
```

La primer ruta básica, comienza por la precondición de la función y termina con la invariante de ciclo, las instrucciones a lo largo de este camino inicializan las variables, como se muestra:

$$\begin{aligned} &\textbf{@Pre } 0 \leq tam < |x| \\ &i = tam; \\ &suma = 0; \\ &\textbf{@L: } \exists i.\ 0 \leq i < tam \end{aligned} \tag{1.7}$$

La segunda ruta básica asume que el ciclo `for` continua ejecutándose por más de una iteración y que la condición `if` es verdadera, por lo que se realiza la operación de suma a la variable `suma`.

La aserción $i \geq 0$ corresponde al caso en el que la condición del ciclo `for` se cumple y la siguiente aserción al condicional `if` cuando un número es positivo o igual a cero.

$$\begin{aligned}
&\textbf{@L:}\ \exists i.\ 0 \leq i < tam \\
&\textbf{asumir}\ i > 0; \\
&\textbf{asumir}\ x[i] \geq 0; \\
&suma = suma + x[i]; \\
&i = i - 1 \\
&\textbf{@L:}\ \exists i.\ 0 \leq i < tam
\end{aligned} \tag{1.8}$$

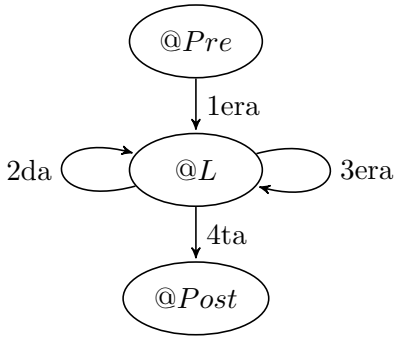
La tercer ruta básica consiste en que el ciclo `for` continúa, pero la condición `if` no se cumple, por lo que simplemente decrementa el valor de la variable `i`. La aserción $i > 0$ corresponde al caso en el que la condición del ciclo `for` es verdadera. Por otra parte, la aserción $x[i] < 0$ pertenece al condicional `if` cuando la condición resulta ser falsa, es decir, que el número `x[i]` es negativo.

$$\begin{aligned}
&\textbf{@L:}\ \exists i.\ 0 \leq i < tam \\
&\textbf{asumir}\ i > 0; \\
&\textbf{asumir}\ x[i] < 0; \\
&i = i - 1; \\
&\textbf{@L:}\ \exists i.\ 0 \leq i < tam
\end{aligned} \tag{1.9}$$

Por último, la cuarta ruta básica representa el momento en el que el ciclo finaliza o cuando la condición del ciclo no se cumple debido al valor de `tam`, y como valor de retorno `rv` devuelve el contenido de `suma`.

$$\begin{aligned}
&\textbf{@L:}\ \exists i.\ 0 \leq i < tam \\
&\textbf{asumir}\ i \leq 0; \\
&\textbf{vr}\ = suma; \\
&\textbf{@Post}\ vr \geq 0
\end{aligned} \tag{1.10}$$

Las cuatro rutas básicas se muestran gráficamente de la siguiente manera:



@P
C_1;
C_2;
...
C_n;
@Q

Figura 1.7: Fragmento de instrucciones anotadas.

Condiciones de verificación. En la semántica definida por Disjkstra [8] para la verificación de programas, que extiende la lógica de Hoare, se define un transformador de predicado conocido como precondición más débil (abreviado *wp* por sus siglas en inglés). Un transformador de predicado p es una función que toma una fórmula de primer orden ϕ y una o más instrucciones de programa C con el objetivo de encontrar una fórmula ψ tal que cumpla:

$$p : \phi \ \times \{C_n, \ \ldots, \ C_1, \ C_0\} \Rightarrow \psi$$

El objetivo principal de la precondición más débil es convertir las rutas básicas obtenidas del código de un programa en condiciones de verificación. Una condición de verificación es una fórmula lógica que ha sufrido un procesamiento basado en las precondiciones, postcondiciones y anotaciones descritas en una ruta básica, con el fin de ser utilizadas en un probador automático de teoremas y así sea posible verificar la corrección del programa.

Definición 1.11. La precondición más débil se describe de la siguiente manera:

$$[wp(\phi, \ C)]^S_{E[e]} = 1$$

donde ϕ corresponde a una fórmula de primer orden, generalmente una post-condición o una invariante de ciclo. C corresponde a un conjunto no vacío de instrucciones de programa, S a una estructura definida en alguna teoría de lógica de primer orden, y $E[e]$ a la evaluación de la fórmula bajo un estado e.

Como se mencionó en la definición de la tripleta de Hoare, un estado e es el instante en el que las variables, al ser interpretadas, poseen un valor determinado en memoria. Por ejemplo, para establecer que el programa se encuentra ejecutando alguna sentencia de código específica, el contador de programa *pc* se podría definir que apunta a una dirección de memoria M_1, la cual posee la siguiente instrucción a ejecutar: $e_1 : \{pc \mapsto M_1\}$.

Definición 1.12. La representación de la tripleta de Hoare $\{P\}\ C_1; C_2; \ldots; C_n\ \{Q\}$ como una condición de verificación de la ruta básica mostrada en la Figura 1.7 se define como:

$$P \Rightarrow wp(Q, \ C_1; \ C_2; \ldots; C_n)$$

dado que la precondición P debe ser válida antes de ejecutar el conjunto de instrucciones C_i y durante la ejecución, Q será válida.

La precondición más débil para una aserción P establece que:

$$wp(P, asumir\ \phi) \text{ si y solo si } \phi \Rightarrow P$$

Por otro lado, para obtener la precondición más débil de un número n de instrucciones $C_1; C_2; \ldots; C_n$, se define:

$$wp(\phi,\ C_1;\ C_2;\ \ldots,\ C_n) \text{ si y solo si } wp(wp(\phi,\ C_n);\ C_1;\ C_2;\ \ldots;\ C_{n-1})$$

Finalmente, para la instrucción de asignación $x := e$ donde x es una variable y $e \in U$ bajo una estructura S, se dice que la precondición más débil es:

$$wp(P(x),\ x := e) \text{ si y solo si } P(e)$$

Ejemplo 1.9. Determinar la condición de verificación del siguiente fragmento de código.

$$\begin{aligned} &\textbf{@Pre} \quad y \geq 5 \\ &\quad x := y + 3; \\ &\textbf{@Post} \quad x \geq 8 \end{aligned}$$

Como primer paso se debe calcular la precondición más débil wp dada la siguiente fórmula ϕ:

$$\phi : y \geq 5 \Rightarrow wp(x \geq 8,\ x := y + 3) \tag{1.11}$$

Para realizar el calculo, se substituye el valor asignado a la variable x en la fórmula $x \geq 8$. El resultado es $y + 3 \geq 8$ y puede simplificarse en $y \geq 5$. Al no existir más instrucciones de código, el procedimiento termina y la fórmula ϕ resultante corresponde a la condición de verificación. Debido a que es una operación básica, la fórmula es válida a simple vista, sin embargo, en casos reales se emplea un demostrador automático de teoremas.

$$\begin{aligned} &wp(x \geq 8,\ x := y + 3) \\ &\iff x \geq 8 \\ &\iff y + 3 \geq 8 \\ &\iff y \geq 5 \\ &\phi : y \geq 5 \Rightarrow y \geq 5 \end{aligned}$$

Ejemplo 1.10. Del ejemplo 1.8 se calculará la condición de verificación de la ruta básica número dos. Se tiene que $\phi : L \Rightarrow wp(L,\ C_1;\ C_2;\ C_3;\ C_4)$, donde L corresponde a la invariante de ciclo anotada en el programa y C_n al conjunto de instrucciones de programa y aserciones estipuladas en la ruta básica. La aserción L debe ser verdadera antes de ejecutar C y posteriormente después de su ejecución, dicha aserción debe continuar siendo verdadera. Para convertir la precondición más débil `wp` a una fórmula se realiza el siguiente procedimiento:

$$
\begin{aligned}
& wp(L,\ C_1;\ C_2;\ C_3;\ C_4) \\
& \iff wp(wp(\exists i.\ 0 \leq i < tam,\ i = i - 1),\ C_1,\ C_2,\ C_3) \\
& \iff wp(\exists i.\ 0 \leq (i-1) < tam,\ C_1,\ C_2,\ C_3) \\
& \iff wp(wp(\exists i.\ 0 \leq (i-1) < tam,\ suma = suma + x[i]),\ C_1,\ C_2) \\
& \iff wp(\exists i.\ 0 \leq (i-1) < tam,\ C_1,\ C_2) \\
& \iff wp(wp(\exists i.\ 0 \leq (i-1) < tam,\ \textbf{asumir}\ x[i] \geq 0),\ C_1) \\
& \iff wp(x[i] \geq 0 \Rightarrow \exists i.\ 0 \leq (i-1) < tam,\ C_1) \\
& \iff wp(x[i] \geq 0 \Rightarrow \exists i.\ 0 \leq (i-1) < tam,\ \textbf{asumir}\ i > 0) \\
& \iff i > 0 \Rightarrow x[i] \geq 0 \Rightarrow \exists i.\ 0 \leq (i-1) < tam
\end{aligned}
$$

Por lo que substituyendo $wp(L,\ C_1;\ C_2;\ C_3;\ C_4)$ y L en ϕ se obtiene la condición de verificación: $\exists i.\ 0 \leq i < tam \Rightarrow i > 0 \Rightarrow x[i] \geq 0 \Rightarrow \exists i.\ 0 \leq (i-1) < tam$. Así, de acuerdo a la regla de equivalencia:

$$\phi_1 \wedge \phi_2 \Rightarrow (\phi_3 \Rightarrow (\phi_4 \Rightarrow \phi_5)) \iff (\phi_1 \wedge \phi_2 \wedge \phi_3 \wedge \phi_4) \Rightarrow \phi_5$$

Se tiene como resultado la condición de verificación ψ escrita de la siguiente manera:

$$\psi : \exists i.\ 0 \leq i < tam \wedge i > 0 \wedge x[i] \geq 0 \Rightarrow \exists i.\ 0 \leq (i-1) < tam$$

Finalmente, ahora es posible validar ψ a través de un demostrador automático de teoremas y verificar si esa sección del código se comporta de la manera esperada. Sin embargo, como se mencionó anteriormente la corrección parcial únicamente comprueba si un fragmento de código es correcto o no. En el caso que un ciclo por alguna razón quedara ejecutándose indefinidamente o alguna sección recursiva, el programa no terminaría, para ello se requiere probar la corrección total.

Corrección total

La corrección total en contraste con la parcial asegura que el programa termina su ejecución o que la función a validar regresa un valor esperado en un tiempo finito. Esta distinción es importante, puesto que la lógica de Hoare por sí sola no permite verificar la corrección total, sino que se requieren de métodos adicionales, por ejemplo, el uso del método de función de clasificación (*ranking function method*) [2] para asegurar que un programa no queda ejecutándose indefinidamente. Es por ello que la dificultad para validar una corrección total resulta mayor, y debe tenerse en consideración dependiendo de las necesidades del problema a solucionar.

Para probar la terminación de un programa, se requiere del uso de una relación bien fundada, la cual es aquella estructura S en la que existe una secuencia finita tal que el elemento sucesor sea menor que su antecesor $\{e_1 \succ e_2 \succ e_3 \succ \ldots\ e_n\}$. Los símbolos $\succ$ y $\prec$ resultan análogos a los predicados utilizados con los números naturales $>$ y $<$, no obstante dependen del contexto en que se utilicen.

El método por función de clasificación indica que se debe encontrar una función en lógica de primer orden que asocie los estados del programa a elementos de una estructura S y además cada uno cumpla la condición de una relación bien fundada a lo largo de una ruta básica. En

$$
\begin{aligned}
&\textbf{@P}\\
&\downarrow \lambda[\overline{x}]\\
&\qquad C_1;\\
&\qquad C_2;\\
&\qquad \ldots\\
&\qquad C_n;\\
&\downarrow \varrho[\overline{x}]
\end{aligned}
$$

Figura 1.8: Anotaciones con funciones de clasificación.

esta técnica se emplean un tipo especial de anotaciones conocidas como funciones de clasificación que serán denotadas con el símbolo $\downarrow$.

Por ejemplo, en la Figura 1.8 se muestra dos funciones de clasificación λ y ϱ que se deben ser verdaderas antes y después de la ejecución de las instrucciones respectivamente. Sin embargo, una vez que se define la ruta básica deben ser interpretadas para obtener las condiciones de verificación. Para ello se utiliza el símbolo de predicado de la precondición más débil *wp* al igual que en la corrección parcial definido de la siguiente manera:

$$F \Rightarrow wp(\varrho \prec \lambda[\overline{x_0}],\ C_1;\ C_2;\ \ldots;\ C_n)\{\overline{x_0} \mapsto \overline{x}\} \tag{1.12}$$

Donde $C_1;\ C_2;\ \ldots;\ C_n$ corresponde a un número n de instrucciones de programa y aserciones y la fórmula F es la precondición. En este método se busca asegurar que $\varrho \prec \lambda[\overline{x_0}]$. Para el caso de la función de clasificación $\lambda[\overline{x_0}]$ debe asignarse un nombre temporal a cada variable en el cálculo de la precondición más débil y posteriormente se realiza la sustitución mediante $\{\overline{x_0} \mapsto \overline{x}\}$.

Ejemplo 1.11. Para ejemplificar la corrección total, se toma un ejemplo del libro [2], pero adaptado ligeramente a la sintaxis del lenguaje C y descrito de manera más detallada. En la Figura 1.9 se muestra el algoritmo *bubble sort* previamente anotado.

La función de clasificación binaria $\downarrow (i+1,\ i+1)$ escrita en el primer ciclo se escoge debido a que la variable `i` podría tomar un valor negativo cuando el tamaño del arreglo `arr` es de cero. Sin embargo, debido a que se busca probar que la condición de verificación es válida bajo la teoría de los número naturales, se debe asegurar que no tome valores negativos. La segunda función de clasificación $\downarrow (i+1,\ i-j)$ fue escogida porque la variable i decrementa a lo largo de la ejecución del programa, por ejemplo, para el caso en el que j toma el valor de cero se tiene que $i+1 > i$, resultando verdadera.

El primer paso para obtener la condición de verificación es obtener las rutas básicas. Para este ejemplo, únicamente se escoge una de ellas, sin embargo, el procedimiento que se realiza a continuación se debe llevar a cabo para cada una de las rutas básicas.

```
@Pre ⊤
@Post ⊤
int[] BubbleSort(int arr[]) {
    int[] a = arr;
    int i, j;
    for
    @L1 : i + 1 ≥ 0
    ↓ (i + 1, i + 1)
    (i = len(a) - 1; i > 0; i = i - 1) {
        for
        @L2 : i + 1 ≥ 0 ∧ i − j ≥ 0
        ↓ (i + 1, i − j)
        (j = 0; j < i; j = j + 1) {
            if(a[j] > a[j+1]) {
                int t = a[j];
                a[j] = a[j+1];
                a[j+1] = t;
            }
        }
    }
    return a;
}
```

Figura 1.9: Programa anotado con funciones de clasificación.

@L_2: $i + 1 \geq 0 \ \wedge \ i - j \geq 0$
$\downarrow L_2 : (i + 1, \ i - j)$
asumir $j < i$;
asumir $a[j] \leq a[j + 1]$;
$j = j + 1$;
@L_2: $(i + 1, \ i - j)$

La ruta básica mostrada corresponde al caso en el que se ejecuta el segundo ciclo `for` del algoritmo y cuando la condición `if` no es verdadera, por lo que únicamente realiza un incremento a la variable `j`. La condición de verificación se escribe como una fórmula ϕ de la siguiente manera:

$$\phi : L_2 \Rightarrow wp((i + 1, \ i - j) \ \prec \ (i + 1, \ i - j)[\overline{x_0}], \ C_1; \ C_2; \ C_3;)\{\overline{x_0} \mapsto \overline{x}\}$$

Para transformar la precondición más débil `wp` a una fórmula se debe realizar el siguiente procedimiento:

$$
\begin{aligned}
& wp((i+1,\ i-j) \prec (i+1,\ i-j)[\overline{x_0}],\ C_1;\ C_2;\ C_3)\{\overline{x_0} \mapsto \overline{x}\} \\
& \iff wp((i+1,\ i-j) \prec (i_0+1,\ i_0-j_0),\ C_1;\ C_2;\ C_3)\{\overline{x_0} \mapsto \overline{x}\} \\
& \iff wp(wp((i+1,\ i-j) \prec (i_0+1,\ i_0-j_0),\ j=j+1),\ C_1;\ C_2)\{\overline{x_0} \mapsto \overline{x}\} \\
& \iff wp((i+1,\ i-(j+1)) \prec (i_0+1,\ i_0-j_0),\ C_1;\ C_2)\{\overline{x_0} \mapsto \overline{x}\} \\
& \iff wp(wp((i+1,\ i-(j+1)) \prec (i_0+1,\ i_0-j_0),\ \textbf{asumir}\ a[j] \leq a[j+1]),\ C_1)\{\overline{x_0} \mapsto \overline{x}\} \\
& \iff wp(a[j] \leq a[j+1] \Rightarrow (i+1,\ i-(j+1)) \prec (i_0+1,\ i_0-j_0),\ C_1)\{\overline{x_0} \mapsto \overline{x}\} \\
& \iff wp(a[j] \leq a[j+1] \Rightarrow (i+1,\ i-(j+1)) \prec (i_0+1,\ i_0-j_0),\ \textbf{asumir}\ j<i)\{\overline{x_0} \mapsto \overline{x}\} \\
& \iff j<i \Rightarrow a[j] \leq a[j+1] \Rightarrow (i+1,\ i-(j+1)) \prec (i_0+1,\ i_0-j_0)\{\overline{x_0} \mapsto \overline{x}\} \\
& \iff j<i \Rightarrow a[j] \leq a[j+1] \Rightarrow (i+1,\ i-(j+1)) \prec (i+1,\ i-j)
\end{aligned}
$$

Por lo que substituyendo $wp(L,\ C_1;\ C_2;\ C_3;\ C_4)$ y L_2 en ϕ se obtiene la condición de verificación:

$$\phi : i+1 \geq 0 \ \wedge\ i-j \geq 0 \Rightarrow j<i \Rightarrow a[j] \leq a[j+1] \Rightarrow (i+1,\ i-(j+1)) \prec (i+1,\ i-j)$$

que resulta equivalente a:

$$\phi : i+1 \geq 0 \ \wedge\ i-j \geq 0 \wedge j<i \wedge a[j] \leq a[j+1] \Rightarrow (i+1,\ i-(j+1)) \prec (i+1,\ i-j)$$

Finalmente, esta condición de verificación es posible validarla para comprobar si el programa termina realmente. En la fórmula ϕ, las invariantes de ciclo y las anotaciones permiten verificar la corrección parcial del programa. En cambio, las funciones de clasificación complementan la fórmula para probar que el programa finaliza en tiempo finito.

Capítulo 2

Lógicas de separación

Las lógicas de separación son una extensión a la lógica de Hoare. Surgen para resolver el problema que presenta la lógica clásica con programas que manipulan memoria dinámica y hacen uso de estructuras de datos que utilizan punteros, ya que proveen un lenguaje capaz de representar aserciones a nivel de la memoria RAM. Uno de los primeros trabajos enfocados en especificar ese tipo de programas [3] toma como referencia el método inductivo de Floyd. Su objetivo es especificar operaciones de listas ligadas y árboles basado en los lenguajes imperativos de esa época, sin embargo, el lenguaje lógico empleado aún poseía limitaciones.

Varios años más tarde, en [26] se propone una lógica desde el punto de vista de las implicaciones agrupadas *(bunched implications en inglés)* con tal de modificar las conectivas tradicionales para especificar estructuras de datos mutables. Poco después, dichos trabajos sirven de inspiración para que Reynolds en el año 2000 [31] defina los principios de unos operadores capaces de especificar la memoria tomando de base la lógica de Hoare.

Posteriormente, en [16] diversos autores introducen el concepto de la regla del marco que permite una verificación de programas incremental, al enfocarse únicamente en los recursos de la memoria que intervienen en la ejecución de un programa, probandoo así su viabilidad. Finalmente, en [27, 30] se define formalmente lo que se conoce como las lógicas de separación. Gracias a dichos trabajos, ahora es posible especificar operaciones de manipulación de la memoria, como son: reservar un espacio, la des-asignación de dicho espacio reservado, la mutación del contenido en esas localidades, y obtener el contenido de alguna localidad específica.

2.1. Sintaxis

El lenguaje empleado en las lógicas de separación, al ser una extensión de la lógica de Hoare, es muy similar a éste. Se define a través de la siguiente tupla: $(Vals, \mathcal{V}, \mathcal{F}, \mathcal{P})$. Donde $Vals = \{C \cup Dirs \cup \mathbb{N}\}$ corresponde a constantes $C = \{null, -, a, b, c, ...\}$, direcciones de memoria $Dirs = \{d_0, d_1, d_2, ...\}$ o números enteros. Por otro lado, $\mathcal{V}$ es un conjunto contable finito de valores que engloba variables $\mathcal{V} = \{x_0, y_0, z_0, ...\}$, mientras que $\mathcal{F}$ corresponde a símbolos de funciones que se definirán más adelante como el montículo y la pila. Finalmente, $\mathcal{P}$ es un conjunto de símbolos de predicado que incluye el símbolo $=$ de la teoría de la igualdad y el símbolo $\mapsto$.

Se incluyen las conectivas lógicas tradicional como $\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\iff$ y los cuantificadores $\forall$, $\exists$, sin embargo, se incorpora la conjunción de separación que es denotada por el símbolo $*$ y la implicación de separación $-\!\!*$.

Una expresión $\mathcal{E}$ será definida como una variable o un valor: $\mathcal{E} = \{\mathcal{V} \cup Vals\}$. A continuación, se define un conjunto de aserciones para realizar la construcción de las fórmulas en las lógicas de separación empleando expresiones:

$$Ase ::= e_1 = e_2 \mid e \mapsto e_1 \mid e \mapsto (e_1, e_2, ...) \mid e \mapsto - \mid emp$$

tal que, $e,\ e_1,\ e_2, emp \in \mathcal{E}$.

De manera inductiva una fórmula en lógicas de separación ϕ se representa de acuerdo con la siguiente gramática:

$$\phi ::= \neg\phi \mid \phi \wedge \psi \mid \phi \vee \psi \mid \phi \Rightarrow \psi \mid \phi \iff \psi \mid \exists x.\ \phi \mid \forall x.\ \phi \mid \phi * \psi \mid \phi -\!\!* \psi \mid Ase$$

2.2. Semántica

Para poder interpretar una fórmula en lógicas de separación, se definen los estados de memoria [15, 12]. De manera formal: $Estado \stackrel{\text{def}}{=} m \times s$. Estos se dividen en montículo m (*heap*) y pila s (*stack*).

El montículo es una función parcial finita, que interpreta direcciones de memoria hacia un conjunto de valores. De manera formal:

$$m \stackrel{\text{def}}{=} Dirs \mapsto Vals \times Vals$$

donde *dir* corresponde a un conjunto contable finito de direcciones de memoria y *vals* a un conjunto de valores enteros, direcciones de memoria o átomos. En la Figura 2.1 se muestra una representación del montículo de manera gráfica.

Por otro lado, la pila es una función parcial que asocia un conjunto finito de variables $Vars$ hacia un conjunto de valores $Vals$ como valores enteros, direcciones de memoria o átomos:

$$s \stackrel{\text{def}}{=} Vars \mapsto Vals$$

Se dice que dos montículos m_1 *y* m_2 se encuentran disjuntos si y solo si el $dominio(m_1) \cap dominio(m_2) = \emptyset$. A partir de este punto será denotado mediante el símbolo $\perp$, de la siguiente manera: $m_1 \perp m_2$. A continuación, se describe a detalle la semántica de las fórmulas de lógicas de separación. Para comenzar, la sentencia *emp* representa el montículo vacío, en los lenguajes de programación equivaldría a declarar una variable o reservar un espacio de memoria, pero sin inicializar.

$$[emp]^s_m = 1 \text{ si y solo si } dominio(m) = \emptyset$$

Existe una forma de definir cuando una celda en el montículo permanece reservada, pero con un valor nulo si y solo si dada una variable x interpretada bajo una pila s, dicho valor pertenece al dominio de un montículo m:

Localidad	Contenido
0xFF00	0x0007
0xFF01	23
0xFF02	17
0xFF03	8
0xFF04	35
0xFF05	0xFF78
0xFF06	0xFF96
0xFF07	7
0xFF08	10

Figura 2.1: Representación del montículo de manera gráfica.

$$[e \mapsto -]^s_m = 1 \text{ si y solo si } \mathit{dominio}(m) = \{[e]_s\} \mid e \in \mathcal{E}$$

El símbolo de predicado $\mapsto$ permite establecer el contenido del montículo en un momento determinado. Requiere de especificar una dirección de memoria y el valor a almacenar. Se le conoce también como montículo unitario, porque corresponde a la unidad más pequeña que puede representarse en la memoria.

$$[e \mapsto e_1]^s_m = 1 \text{ si y solo si } [e]_s \in \mathit{dominio}(m) \text{ y } [[e]_s]_h = [e_1]_s \mid e, e_1 \in \mathcal{E}$$

En caso de emplear dicho símbolo para establecer más de un valor comenzando desde la localidad de memoria e, se puede representar de la siguiente manera:

$$[e \mapsto (e_1, e_2, ..., e_n)]^s_m = 1 \text{ si y solo si}$$
$$[e]_s \in \mathit{dominio}(m) \text{ y } [[e]_s]_h = ([e_1]_s, [e_2]_s, ..., [e_n]_s) \mid e, ..., e_n \in \mathcal{E}$$

Por otra parte, un montículo puede satisfacer $\phi * \psi$ si y solo si puede descomponerse en dos montículos separados m_1 y m_2 tal que satisfagan a ϕ y ψ respectivamente y a su vez, pueda formarse un montículo h tal que $m = m_1 \uplus m_2$, donde $m_1 \uplus m_2$ representa la unión de dos montículos disjuntos m_1 y m_2.

$$[\phi * \psi]^s_m = 1 \text{ si y solo si } \exists\, m_1, m_2.\ m = m_1 \uplus h_2,\ [\phi]^s_{m_1} = 1 \text{ y } [\psi]^s_{m_2} = 1$$

La conjunción de separación establece que dado una fórmula $\phi \mathrel{-\!\!*} \psi$ se generan nuevos montículos tal que para todo m_1 disjunto de h hace verdadero una fórmula ϕ, e implica que la unión de m_1 y h forman un nuevo montículo que hace verdadero ψ.

$$[\phi \mathrel{-\!\!*} \psi]^s_m = 1 \text{ si y solo si } \forall\, m_1.\ (m_1 \perp m \text{ y } [\phi]^s_{m_1} = 1) \text{ implica } [\psi]^s_{m \uplus m_1} = 1$$

En el caso del cuantificador existencial se dice que, dada una fórmula ϕ existe una variable x bajo un montículo m y pila s que es verdadera si y solo si existe un valor $v \in Vals$, tal que $x \mapsto v$.

$$[\![\exists x.\ \phi]\!]^s_m = 1 \text{ si y solo si } \exists v. \in Vals.\ [\![\phi]\!]^{s\,|\,x\mapsto v}_m = 1$$

Finalmente, para el cuantificador universal se dice que dada una fórmula ϕ existe una variable x bajo un montículo m y una pila s que es verdadera si y solo si es verdadero para todos los valores de $v \in Vals$, tal que $x \mapsto v$.

$$[\![\forall x.\ \phi]\!]^s_m = 1 \text{ si y solo si } \forall v. \in Vals.\ [\![\phi]\!]^{s\,|\,x\mapsto v}_m = 1$$

Dados los modelos de los montículos m y las pilas s, se dice que una fórmula ϕ es válida si y solo si para todos los estados de m y s se tiene que $[\![\phi]\!]^m_s = 1$. En cambio, se dice que una fórmula ϕ se satisface si y solo si para algún estado de m y s se cumple que $[\![\phi]\!]^m_s = 1$.

En el trabajo de Reynolds se encuentran definidos comandos o instrucciones de programa que describen el comportamiento de la memoria dinámica. Corresponde a la sección central en las tripletas de Hoare:

$$\begin{array}{lll}
\langle instrucciones\rangle ::= ... & & \\
\qquad \langle variable\rangle := \textbf{reservar}\,(e_1,\, e_2,\, ...) & \text{asignación} \\
\qquad \langle variable\rangle := [\,e\,] & \text{obtención} \\
\qquad [\,e\,] := e_1 & \text{mutación} \\
\qquad \textbf{liberar}\,(e) & \text{liberación}
\end{array}$$

donde $e, e_1, e_2 \in \mathcal{E}$. Dichos comandos permiten especificar operaciones a realizar sobre el montículo, como lo es la asignación de un valor constante a través de expresiones, la obtención, alteración del valor alojado en una dirección de memoria y su liberación. El operador de asignación := accede a la dirección de memoria indicada en el lado izquierdo de éste y almacena el valor expresado en el lado derecho, de esta manera se indica una asignación explicita en memoria.

En el caso de **reservar** $(e_1,\, e_2,\, ..., e_n)$ se inicializa n direcciones de memoria en el montículo, en programación esto se expresa como la declaración e inicialización de un arreglo. En cambio, **liberar** (e) elimina del montículo una expresión e. Por último, el operador $[\,e\,]$ define explícitamente la dirección de memoria a la que se va a acceder. Para la búsqueda se realiza una operación de lectura sobre esa dirección, y para la mutación se realiza una operación de escritura.

Ejemplo 2.1. En el contexto de una lista ligada como estructura de datos, suponga que los nodos se representan como $\{n_0, n_1, n_2\}$ y cada uno almacena tanto la dirección al siguiente nodo como su contenido, en este ejemplo el contenido de los nodos corresponderá con su índice. Si se desea ejecutar una función que elimine el nodo n_2, primero se debe liberar la memoria para ese nodo:

$$\{n_0 \mapsto (0,\, n_1) * n1 \mapsto (1,\, n_2) * \mathbf{n_2} \mapsto (\mathbf{2},\, \mathbf{emp})\} \quad liberar(n_2) \quad \{n_0 \mapsto (0,\, n_1) * n_1 \mapsto (1,\, n_2) * \mathbf{emp}\}$$

Posteriormente, la dirección de memoria a la que apunta n_2 correspondería a una dirección de memoria inválida, puesto que n_3 no existe más en el montículo. Esta última operación se expresa de la siguiente manera:

$$\{n_0 \mapsto (0,\, n_1)\ *\ n1 \mapsto (1,\, \mathbf{n_2})\} \quad [n1] := (1, null) \quad \{n_0 \mapsto (0,\, n_1)\ *\ n_1 \mapsto (1,\, -)\}$$

$$
\begin{aligned}
P * S &\iff S * P \\
P * emp &\iff P \\
(P * S) * T &\iff P * (S * T) \\
(P \vee Q) * T &\iff (P * T) \vee (P * T) \\
(P \wedge Q) * T &\iff (P * T) \wedge (P * T) \\
(\exists x.\ P) * Q &\iff \exists x.\ (P * Q) \\
(\forall x.\ P) * Q &\iff \forall x.\ (P * Q)
\end{aligned}
$$

Figura 2.2: Reglas de inferencia derivadas de los nuevos operadores en las lógicas de separación.

La precondición indica a través del operador $\mapsto$ el valor que posee en el montículo, cada uno compuesto por la tupla $(contenido, sig\_nodo)$. El nodo n_1 establece que el valor al siguiente nodo se encuentra indefinido debido a la operación de borrar el n_2, pero aún se encuentra el espacio de memoria definida dentro del montículo. Además, al estar disjuntas en la memoria se asegura que el modificar un nodo no afecta el contenido de otros.

Es importante aclarar que únicamente es posible acceder a direcciones de memoria que estén reservadas durante la ejecución del programa, de lo contrario se genera un error llamado violación de acceso a memoria. Este error puede producirse durante el acceso al contenido de una dirección de memoria (también denominada indirección) y cuando se intenta liberar memoria que no es propiedad del programa. En el ejemplo anterior, se asume que la memoria fue previamente reservada.

Las reglas de inferencia en la lógica de Hoare siguen aplicando en las lógicas de separación; sin embargo, algunos axiomas se derivan de los nuevos operadores, por ejemplo, algunas propiedades conmutativas, distributivas y asociativas de la conjunción de separación se muestran en la Figura 2.2. Estas pueden ser útiles si se desea probar una fórmula en lógicas de separación de manera manual.

2.3. Regla del marco

Previo a la creación de la regla del marco, en la lógica clásica fue creada la regla de la constancia, la cual garantiza que instrucciones de programa C no alterarán una porción de memoria, denominada por una fórmula R, después de la ejecución de dicho programa. A continuación se muestra de manera formal:

$$\frac{\{P\}\, C\, \{Q\}}{\{P \wedge R\}\, C\, \{Q \wedge R\}} \tag{2.1}$$

La fórmula R también se le conoce como marco o *frame* en inglés. Esta regla permite enfocarse solo en las partes del código que modifican el montículo. Sin embargo, surge un problema al

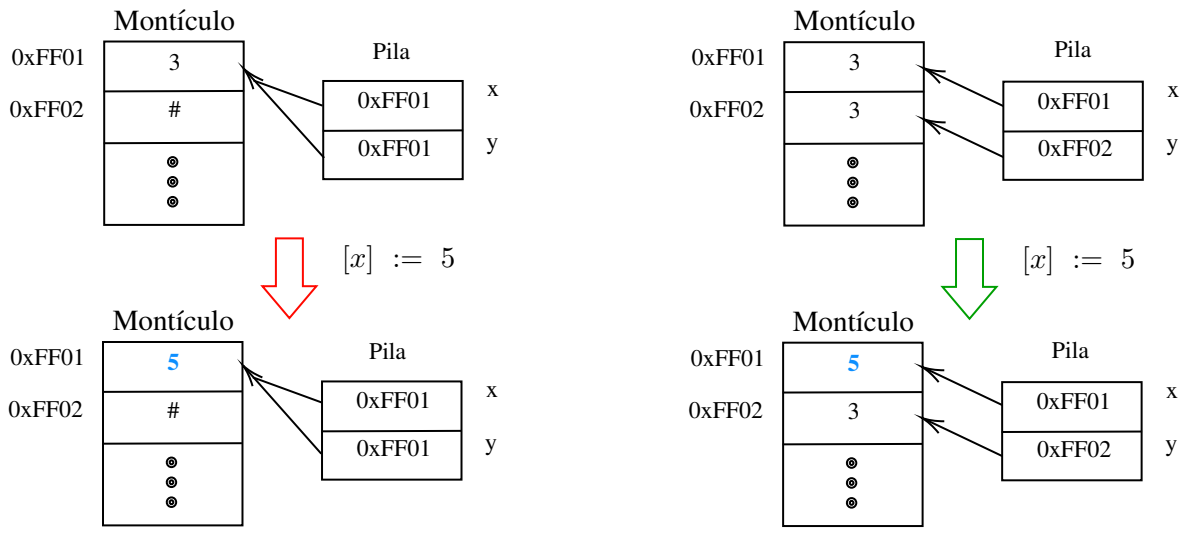


Figura 2.3: Problema con especificaciones de programa en lógica clásica.

momento de especificar variables que se encuentran apuntando a la misma dirección de memoria, por ejemplo, derivado de la regla de la constancia se tiene que:

$$\frac{\{\, x \mapsto 3 \,\}\ [x] := 5\ \{\, x \mapsto 5 \,\}}{\{\, x \mapsto 3 \wedge y \mapsto 3 \,\}\ [x] := 5\ \{\, x \mapsto 5 \wedge y \mapsto 3 \,\}} \tag{2.2}$$

Esta fórmula sería falsa si las variables tanto x como y son iguales, es decir, apuntan a la misma dirección de memoria y resulta imposible excluir ese caso de la especificación con los operadores tradicionales. En la Figura 2.3 se muestra de manera gráfica tanto el montículo como la pila de llamadas, antes y después de la ejecución de la instrucción $x := 5$.

Para corregir este problema, Peter O´Hearn extiende dicha regla con el fin de representar R como una porción de memoria que permanece sin cambios de manera separada, resolviendo el problema que se tenía en la lógica tradicional. Entonces, si se emplea la conjunción de separación al evaluar la siguiente especificación resulta ser verdadera, tanto la pre-condición como la post-condición, como se muestra a continuación:

$$\frac{\{\, x \mapsto 3 \,\}\ [x] := 5\ \{\, x \mapsto 5 \,\}}{\{\, x \mapsto 3 * y \mapsto 3 \,\}\ [x] := 5\ \{\, x \mapsto 5 * y \mapsto 3 \,\}} \tag{2.3}$$

Se asegura que tanto x como y son variables distintas, por lo que alterar el valor de x al ejecutar la instrucción $x := 5$ no altera el valor de la variable y. Gracias a los nuevos operadores O´Hearn establece la regla del marco, la cual resulto inspirada del conocido problema del marco en el área de inteligencia artificial, de allí su nombre. El elemento clave es remplazar la conjunción tradicional por la conjunción de separación.

$$\frac{\{P\}\, C\, \{Q\}}{\{P * R\}\, C\, \{Q * R\}} \tag{2.4}$$

Esta regla es de gran importancia al definir el comportamiento de un programa ya que se enfoca únicamente en la información escencial de la memoria para un estado determinado, también es conocida como huella. Así, es posible habilitar el razonamiento local del montículo, porque se asegura que lo que no está definido en la especificación permanezca sin cambios. Esto abre la posibilidad a verificar programas que hacen uso de memoria dinámica, que posean grandes líneas de código y sea posible escalar a nivel industrial.

2.4. Bi-abducción

Uno de los trabajos más importantes sobre bi-abducción se encuentra en [6]. Ofrece una solución que permite automatizar la generación de pre- y post-condiciones con ayuda de la abducción y las lógicas de separación. La abducción es una forma de inferencia lógica propuesta por Charles Peirce en su trabajo llamado "*Illustrations of the Logic of Science*", escrito durante el siglo XIX. Este método propone encontrar una hipótesis basada en los modos de inferencia deductivo e inductivo. Derivado de este razonamiento, se establece la bi-abducción, un proceso de doble inferencia lógica.

La bi-abducción consiste formalmente en determinar dos porciones desconocidas de las especificaciones de un programa. La primera consiste en dada una fórmula escrita en las lógicas de separación, como se muestra en la ecuación (2.5), encontrar una fórmula F denominada marco, que pueda ser inferida de una fórmula ϕ pero no de la fórmula ψ. El marco corresponde a una sección del montículo que permanece sin cambios durante una especificación descrita.

$$\phi \vdash \psi * F \tag{2.5}$$

Por otro lado, la segunda porción consiste en inferir la parte contraria, como se muestra en la ecuación (2.6). Dada una fórmula ϕ se debe encontrar la fórmula A, denominada anti-marco, que representa la información desconocida del montículo en un estado específico. El anti-marco se refiere a la porción de memoria que cambia durante una especificación descrita.

$$\phi * A \vdash \psi \tag{2.6}$$

Finalmente, la Ecuación (2.7) corresponde al proceso de descubrir el anti-marco y el marco de una fórmula. Cabe resaltar que ambos fragmentos se encuentran disjuntos por la conjunción de separación.

$$(\phi * ?Anti\text{-}Marco \vdash \psi * ?Marco) \tag{2.7}$$

Dentro de los beneficios de la bi-abducción, es posible generar un análisis composicional, que permite establecer inferencias sobre secciones de código aisladas del programa sin requerir conocimiento sobre el resto de éste. Así, los desarrolladores no deben preocuparse por invertir tiempo en generar las especificaciones que pueden resultar complejas en grandes cantidades de código.

Gracias a ello, es posible verificar grandes sistemas, como lo sería un gestor de bases de datos de código abierto, el núcleo de algún sistema operativo Linux y prácticamente cualquier programa del cual se tenga acceso completo al código fuente. A continuación se mostrará un

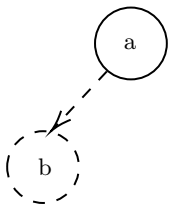
```
1  struct nodo {
2      struct nodo* izq;
3      struct nodo* der;
4      char dato;
5  };
6
7  struct nodo* crearNodo(char dato) {
8      struct nodo* nodo = (struct nodo*) malloc( sizeof(struct nodo) );
9      nodo->dato = dato;
10     nodo->izq = NULL;
11     nodo->der = NULL;
12     return(nodo);
13 }
14
15 void crearArbol() {
16     struct nodo *raiz = crearNodo('a');
17     raiz->der = crearNodo('b');
18 }
```

Figura 2.4: Fragmento de código escrito en C como ejemplo de bi-abducción.

ejemplo de bi-abducción aplicada sobre operación en un árbol binario.

Ejemplo 2.2. Mediante el uso de bi-abducción se busca generar la especificación considerando el código mostrado en la Figura 2.4, justo en el instante en que se ejecuta la siguiente instrucción: `raiz->der = crearNodo('b');`. Para ello se plantea la siguiente tripleta de Hoare:

$$\{emp\}\ nodo(raiz\_der) := crearNodo(\text{'}b\text{'});\ \{???\}$$



En donde `nodo(raiz_der)` corresponde a la representación del puntero al nodo hijo derecho de la raíz ($raiz->der$) para facilitar la lectura. Primeramente, la precondición comienza con el montículo vacío y se define la siguiente fórmula de bi-abducción en busca del marco y el anti-marco:

$$emp * ?Anti\text{-}marco \vdash raiz \mapsto [der : -] * ?Marco$$

En este caso, se escribe la fórmula $raiz \mapsto [der : -]$ en donde se infiere la información desconocida para ejecutar el comando anterior, debido a que se requiere de una referencia previamente definida en la pila de llamadas para alterar su valor en el montículo. La solución a la fórmula anterior corresponde a lo siguiente:

$$?Anti\text{-}marco = raiz \mapsto [der : -] \qquad ?Marco = emp.$$

El marco es el conjunto de memoria que permanece inalterado, en este caso se especifica que separadamente $raiz \mapsto [der : -] * emp$ es válida como precondición, que simplificando finalmente queda como $raiz \mapsto [der : -]$, la cual corresponde a la precondición de la especificación:

$$\{raiz \mapsto [der : -]\}\ nodo(raiz\_der) := crearNodo('b');\ \{???\}$$

Finalmente, se infiere la post-condición basada en la pre-condición encontrada, por lo que al ejecutar la instrucción de código se modifica el valor a la que la variable apunta en el montículo:

$$\{raiz \mapsto [der : -]\}\ nodo(raiz\_der) := crearNodo('b');\ \{raiz \mapsto [der : b]\}$$

Una vez completado la información desconocida y que se ha validado, se prosigue con la siguiente línea de código que se desea analizar repitiendo el mismo procedimiento. Claramente hay casos en los que existan múltiples soluciones, dando lugar a diversidad de hipótesis inferidas por el algoritmo. Para resolverlo, se define un cierto número de soluciones y de entre ellas se selecciona la más adecuada para completar la tripleta de Hoare.

La bi-abducción también se apoya en conjunto de otras técnicas como la ejecución simbólica y el análisis de formas. Esto es muy útil, debido a que el programa no es ejecutado y se requiere encontrar información adicional para que las especificaciones del programa sean más precisas.

El análisis de formas (*shape analysis* en inglés) es un método que permite dar soporte a la inferencia de las estructuras de datos. A pesar de que resulta costoso en tiempo de procesamiento, gracias a que es posible analizar secciones aisladas de código con las lógicas de separación, se logra mejorar el algoritmo para aproximarse lo más posible a la especificación de las estructuras de datos en un tiempo razonable. Este método ayuda principalmente a identificar indirecciones nulas y fugas de memoria mediante el análisis del contenido de la pila.

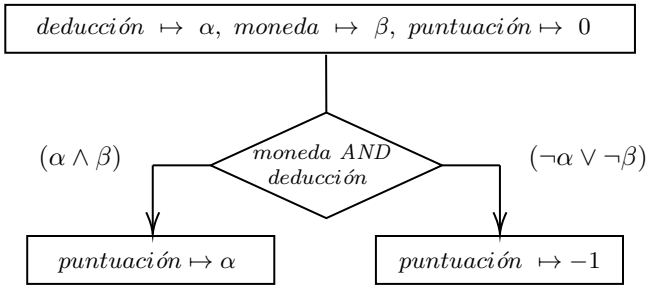
Por otro lado, el proceso de ejecución simbólica a grandes rasgos consiste en simular la ejecución de un programa remplazando los valores desconocidos por elementos simbólicos. Posteriormente, se van generando las posibles rutas de ejecución y se obtiene información sobre los posibles valores que pueden tomar las variables para lograr su correcta ejecución de cada ruta. A continuación se muestra un ejemplo.

Ejemplo 2.3. En el código de la Figura 2.5 la variable `deducción` recibe un valor que al ser desconocido puede representarse como $int\ deduccion = \alpha$ de manera simbólica. Posteriormente, en la línea cinco la variable `moneda` corresponde a $int\ moneda = \beta$. En el caso de la línea siete se realiza una bifurcación en dos rutas, uno en el que `moneda` y `deducción` son verdaderas y otra para su caso contrario.

Cuando el valor de `moneda` y `deducción` son verdaderos, se afirma lo siguiente: $rv \mapsto 1 \wedge \alpha \wedge \beta$. En el caso contrario se puede asegurar que $rv \mapsto -1 \wedge (\neg\alpha \vee \neg\beta)$, de esta manera es que se van generando las especificaciones de acuerdo a las distintas rutas que puedan tomar durante la ejecución.

```
1  int caraOCruz(int deduccion)
2  {
3      /* La función valorAleatorio devuelve un
4      número entero ya sea uno o cero */
5      int moneda = valorAleatorio();
6      int puntuacion = 0;
7      if(moneda && deduccion)
8          puntuacion = moneda;
9      else
10         puntuacion = -1;
11     return puntuacion;
12 }
```

Figura 2.5: Fragmento de código escrito en C como ejemplo de ejecución simbólica



Una aplicación de la ejecución simbólica fue presentada en [9] para detectar fugas de memoria en programas escritos en Java, en conjunto con las ténicas de bi-abducción y lógicas de separación. Lo anterior, con el fin de encontrar porciones de memoria en las que permanece la referencia almacenada y no se utiliza en un estado futuro del programa, en consecuencia no son liberadas por el recolector de basura.

Capítulo 3

Verificación de programas

3.1. Antecedentes de las herramientas de verificación

A lo largo del tiempo se han desarrollado diversos métodos formales para la verificación de programas, por ejemplo, la interpretación abstracta, el análisis de flujo de datos, la verificación de modelos, la ejecución simbólica o en este caso, el análisis basado en aserciones propuesto por Hoare. En consecuencia, existen distintas herramientas que hacen uso de dichos métodos formales para lograr su objetivo. A continuación se realizará una comparación entre algunas de las herramientas existentes en contraste con los beneficios que proveen las lógicas de separación y bi-abducción.

Existe una herramienta llamada BLAST [1], la cual es capaz de verificar algunas propiedades de seguridad de la memoria en programas escritos en lenguaje C mediante la verificación de modelos. Sin embargo, se encontraron algunas limitaciones, entre ellas la incapacidad de razonar sobre los arreglos o estructuras de datos que se encuentran en las invariantes de ciclo. Además, únicamente es capaz de verificar programas de mediano calibre, debido a que para los casos en los que falle la prueba, alguna especificación genera un conjunto de rutas de ejecución que requiere de una gran cantidad de memoria de acuerdo al tamaño del programa.

En contraste, las lógicas de separación permiten definir de manera precisa lo que sucede en la memoria gracias a sus operadores especiales, por lo que cualquier estructura de datos es posible especificarla incluso si se encuentra dentro de ciclos. Además, de acuerdo con la propiedad de composicional, es capaz de analizar programas de un gran número de líneas de código. Debido a que se requiere únicamente de la información esencial del montículo y la pila que modifica una instrucción de código, permite así una verificación por partes evitando la saturación de la memoria.

Existe otra aplicación que utiliza interpretación abstracta [22] para realizar análisis de código en el lenguaje Java. Es capaz de inferir invariantes de ciclo, de clase y postcondiciones, con ello verificar la ausencia de errores en una clase específica, entre estos se encuentran la división por cero, acceso a un arreglo fuera del límite permitido e indirecciones nulas. Sin embargo, las lógicas de separación podrían ser más descriptivas y aproximarse de manera más precisa a la generación de especificaciones en estructuras de datos complejas, como arboles, pilas y listas ligadas, además de proveer el mismo nivel de automatización empleando el método de bi-abducción.

Por otro lado, en [10], una herramienta conocida como ESC/Java identifica errores para el lenguaje de programación Java, a través de anotaciones en JML, con el fin de generar las condiciones de verificación y evaluarlas con un probador automático de teoremas. Sin embargo, continúa haciendo uso de la lógica clásica y es débil en el tratamiento de punteros y memoria dinámica. Además, indica que en determinados casos se deben anotar manualmente las sentencias de código, y debido a ello, se incrementa el esfuerzo manual al momento de la verificación.

En cambio, la bi-abducción permite generar las especificaciones del programa sin necesidad de intervención y busca la mejor que se adecue a la situación, que en conjunto con las lógicas de separación son la mejor combinación para la detección de errores relacionados con la interacción de la memoria de manera dinámica.

3.2. Infer como herramienta de análisis estático

Infer es una herramienta de análisis estático que implementa las lógicas de separación y bi-abducción para identificar diversos errores de programación en los lenguajes de programación Java, C, C++ y Objective-C. Actualmente se encuentra en la versión 0.17.0 y continúa en un constante desarrollo por parte del equipo de Facebook. Además, diversas empresas tecnológicas lo emplean activamente en sus proyectos, como Amazon, Microsot, Spotify, Uber y productos relacionados a Facebook, por mencionar algunas.

Instalación de Infer en Linux

Debido a que es un software de código libre, está disponible para su descarga directa desde su repositorio en Github: `https://github.com/facebook/infer`. Dentro del mismo se encuentran los archivos binarios compilados en sus distintas versiones. Mediante el comando `curl` se descargan dichos archivos binarios usando el protocolo https como se muestra a continuación.

```
# Establece la versión en la variable de entorno VERSION empleada durante
# la instalación.
$ VERSION=X.YY.Z;

Comienza la descarga de los archivos binarios desde el repositorio en github.
$ curl -sSL \
"https://github.com/facebook/infer/releases/download/v$VERSION/
infer-linux64-v$VERSION.tar.xz"
```

Considerando que los archivos ya se encuentran en un directorio local se recomienda realizar la descompresión en la carpeta `opt`. Posteriormente, realizar la creación de un enlace simbólico blando para el archivo ejecutable `infer` dentro del directorio `/usr/local/bin/` con el fin de garantizar su ejecución directamente desde la terminal.

```
# Se descomprime en el directorio correspondiente y posteriormente se crea el
# enlace simbólico.
$ sudo tar -C /opt -xJ && \
$ ln -s "/opt/infer-linux64-v$VERSION/bin/infer" /usr/local/bin/infer
```

Al momento de redactar la presente tesis, la versión más reciente de Infer es la 0.17.0, sin embargo, probablemente en futuras versiones el procedimiento de instalación sea similar. Para comprobar que se ha instalado correctamente, se procede a ejecutar el siguiente comando:

```
# Se comprueba su correcta instalación y versión instalada.
$ infer --version

# Ejemplo de salida del comando anterior.
Infer version v0.17.0
Copyright 2009 - present Facebook. All Rights Reserved.
```

Ciclo de trabajo de infer

Fase de captura

Esta herramienta divide el trabajo en dos fases. La primera de ellas corresponde a la fase de captura, donde se hace uso de los comandos de compilación dependiendo de lenguaje en que esté escrito el programa, esto con el fin de recopilar información de ese proceso y después realizar una traducción al lenguaje intermedio propio de Infer. Es por ello que al momento de ejecutar Infer se debe incluir el comando de compilación. A continuación se muestran algunos ejemplos.

```
# Proceso de captura de un programa escrito en C, compilado con clang.
$ infer capture -- clang programa.c

# Proceso de captura de un programa escrito en C++, compilado con g++.
$ infer capture -- g++ programa.cpp

# Proceso de captura de un programa escrito en Java.
$ Infer capture -- javac Programa.java
```

El siguiente programa será usado como ejemplo en el resto del ciclo de trabajo de Infer.

```
#include <iostream>
using namespace std;

int main()
{
    cout << "Hola mundo";
    return 0;
}
```

Una vez que finaliza el proceso de captura, se crea un directorio con nombre `infer-out` en el mismo lugar donde fue ejecutado, esta carpeta contendrá los archivos intermedios necesarios para la siguiente fase del proceso. La ubicación de este directorio se puede cambiar usando el

parámetro `-o [ruta]`. Infer genera dos carpetas `events` y `specs`, esta última contiene información sobre la fecha de captura, la versión utilizada de Infer y las características de la base de datos. Por otro lado, se tiene un archivo de bitácoras `logs` y una base de datos de tipo SQLite 3 `results.db`. A continuación se muestran las rutas de manera gráfica.

```
infer-out
├── events
├── specs
│   └── infer_runstate.json.
├── logs
└── results.db
```

Fase de análisis

En esta parte, los archivos generados en la fase de captura son utilizados por Infer. Cada función o método, dependiendo del lenguaje de programación, es analizado de forma separada. Es decir, si se encuentra algún error se detiene el análisis para esa función o método, sin embargo, para las demás éste continua.

Siguiendo este planteamiento, una vez que se analiza el código, el programador arregla los fallos y ejecuta de nuevo la herramienta con el fin de encontrar errores adicionales o para verificar que el programa está corregido. El reporte es mostrado en la terminal, en caso de no encontrar alguno se indicará. Además, se guardará una copia de esos errores dentro de la carpeta generada por Infer, en un archivo llamado `bugs.txt`, y su representación en formato JSON en el archivo `report.json`.

Estructura de archivos después del análisis

La carpeta `specs` contiene el análisis realizado por cada función del programa, para el caso del hola mundo únicamente contiene a `main`, como se muestra en la Figura 3.1. Es posible obtener el reporte y mostrarlo en pantalla a través del siguiente comando.

```
$ infer report <archivo .specs>
```

Limitaciones

A pesar de que esta herramienta resulta ser muy poderosa gracias a las lógicas de separación, aún falta investigación en el campo para resolver los falsos positivos que pudieran generarse en el reporte, ya que resulta adverso invertir tiempo corroborando de manera manual cada uno de estos. Además, existen otros problemas relacionados con la competencia de recursos entre dos o más procesos o hilos en el área de concurrencia, e incluso el soporte de nuevos lenguajes.

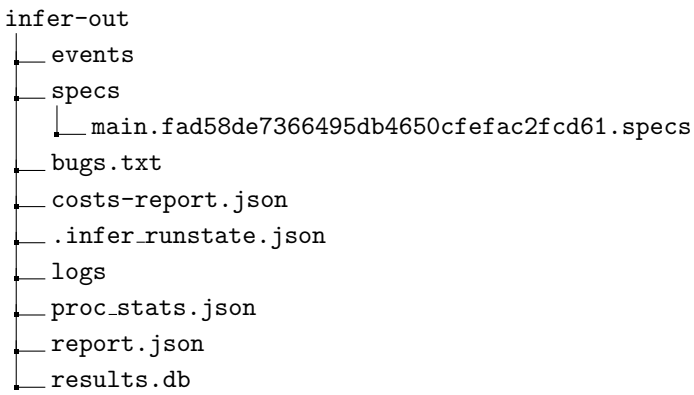


Figura 3.1: Árbol con los contenidos de la carpeta infer-out en la fase de análisis de un hola mundo en C++.

3.3. Experimentos con Infer

Almacenamiento muerto (*Dead store*)

La memoria es un recurso limitado, por esta razón un programa debe consumir la cantidad justa con el fin de evitar desperdicios, sin embargo, si existen grandes cantidades de código se puede pasar por alto la utilización del valor asignado a una variable.

El almacenamiento muerto ocurre cuando un valor guardado en memoria no es leído por alguna otra instrucción, es decir, se hace uso del almacenamiento, no obstante su contenido no es aprovechado. A continuación se muestra un fragmento de código en donde la herramienta Infer descubre que el contenido de la variable `temp` nunca es leído.

```
/* Existe un almacenamiento muerto en la variable 'k', debido a que no
se utiliza el valor almacenado. */
for(int j = 0, k = 10 ; j < numbersSize-1; j++)
{
        int temp = numbers[j];
        if(numbers[j] > numbers[j+1])
                swap(&numbers[j],&numbers[j+1]);
}
```

En dicho código, se asigna el valor que contiene el arreglo **numbers** en la posición **j** hacia la variable temp, sin embargo, su contenido no es empleado en ninguna otra parte del código, siempre y cuando el valor almacenado sea distinto de cero, por lo que no debe confundirse este tipo de error con una variable sin uso.

Por otro lado, para un caso del lenguaje C++ que permite la declaración de variables dentro del ciclo `for`, si se crea una variable y se asigna una constante diferente de 0 o algún valor proveniente de otra variable, Infer lo identificará y arrojará el mismo tipo de error identificado en la Figura 3.2a.

Fuga de recursos (*Resource leak*)

Desde la creación de un programa hasta su ejecución, se requiere del uso de los recursos limitados que provee una computadora. Para el caso de un programador, es importante ahorrar la mayor cantidad de estos ya sea en tiempo de uso o en tamaño. Por ejemplo, si un archivo es abierto por un programa y otro programa desea acceder a éste, donde idealmente se tiene exclusión mutua, el segundo programa deberá esperar a que el recurso sea liberado. Para reducir ese tiempo de espera lo mínimo posible, el desarrollador debe liberar los recursos en el instante en que ya no sea utilizado y así evitar lo que se conoce como fuga de recursos [7].

Otros ejemplos son las conexiones de red como los *sockets* o la reservación dinámica de memoria, de la cual se hablará más adelante ya que este último caso se conoce como fuga de memoria. En la Figura 3.2b se muestra cómo Infer encuentra una fuga de recursos en un programa que abre un archivo para almacenar datos, sin embargo al llegar al final de la función principal no encuentra la liberación del archivo.

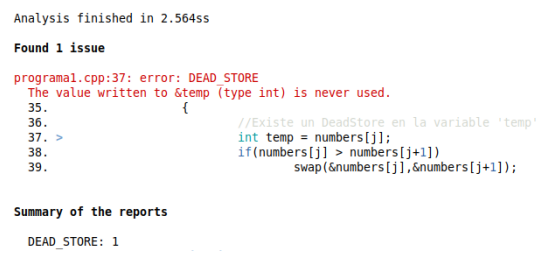
Fuga de memoria (*Memory leak*)

Este problema ocurre cuando se reserva una cantidad de memoria y a pesar de que es posible liberarla no se declara la instrucción para realizarlo. Generalmente, se da en lenguajes de programación que no cuentan con recolectores de basura, por ejemplo C y Objective-C, en donde la persona que realiza el programa es responsable del manejo de memoria. A continuación se muestra un fragmento de código perteneciente a una función en donde se genera este problema.

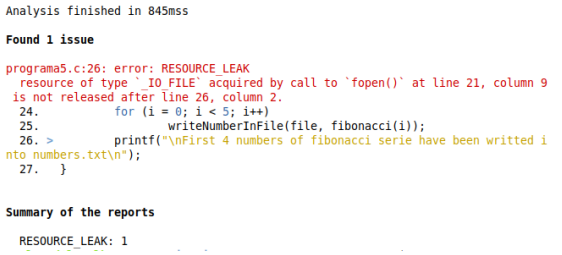
```
int add(int number1, int number2) {
        int *result = malloc(sizeof(int));
        if(result == NULL) {
                printf("Error: memory is full!");
                exit(EXIT_FAILURE);
        }
        *result = number1 + number2;
        return *result;
}
```

Dentro de la función `add` se asigna dinámicamente una variable de tipo entero y se almacena la referencia en el puntero `result`. Posteriormente se valida que el valor no resulte ser nulo para evitar indirecciones nulas en la memoria. Finalmente, se realiza una operación de suma sobre `result` y ese valor es retornado por la función, sin embargo, la localidad reservada para dicha variable nunca fue liberada y es por ello que se presenta la fuga de memoria como se muestra en el reporte de Infer en la Figura 3.2c. Para corregir el error, como el valor es requerido hasta el final de la función, se debe plantear usar una variable estática con el fin de que al momento de retornar el resultado, ésta sea liberada de manera automática.

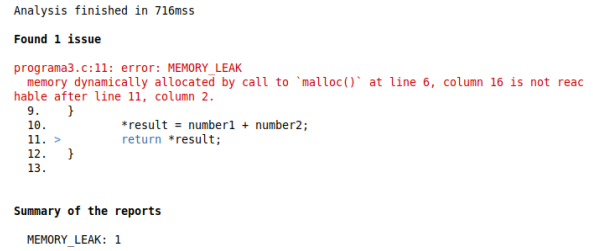
Una limitación por parte de la herramienta Infer, es que sólo puede encontrar este tipo de error si la referencia a la dirección de memoria no se devuelve como valor de una función. Por ejemplo, si la función `add` se modifica tal que retorne la dirección de la memoria reservada, a



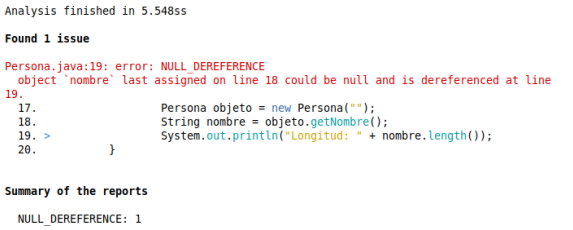
(a) Almacenamiento muerto dentro de un programa en C++.



(b) Fuga de recursos dentro de un programa en C.



(c) Fuga de memoria dentro de un programa en C.



(d) Indirección nula dentro de un programa en Java.

Figura 3.2: Fragmentos de las salidas de los reportes obtenidos con Infer.

pesar que no se libere posteriormente o se pierda la referencia a ésta, no es detectada la fuga de memoria, como se muestra a continuación.

```
int* add(int number1, int number2) {
        int *result = malloc(sizeof(int));
        ...
        *result = number1 + number2;
        return result;
}
```

Indirección nula (*Null Dereference*)

Este tipo de error se presenta cuando se indirecciona una variable que hace referencia a una dirección de memoria que almacena un valor nulo. Por ejemplo, en el lenguaje C o C++ cuando se emplea el operador de indirección `*` sobre una variable nula, el programa termina su ejecución mostrando un mensaje de error, indicando que se ha generado una violación de segmento [19] de la siguiente manera: `Violación de segmento (‘core’ generado)`.

En el lenguaje de programación Java este error se traduce como una excepción en tiempo de ejecución, llamada `NullPointerException`, la cual especifica que se trató de acceder a los métodos o propiedades de un objeto que contiene una referencia nula, como se muestra a continuación:

```
Exception in thread "main" java.lang.NullPointerException
        at Persona.main(Persona.java:19)
```

Para el caso de Infer, este error lo puede detectar en lenguajes de programación como C, Objective-C y Java. En el siguiente caso se analiza el siguiente programa escrito en Java:

```
public class Persona {
        ...
        public String getNombre() {
                if(nombre.length() > 0)
                        return nombre;
                else
                        return null;
        }

        public static void main(String[] args) {
                Persona objeto = new Persona("");
                String nombre = objeto.getNombre();
                System.out.println("Longitud: " + nombre.length());
        }
}
```

En el código se define una clase que lleva como nombre `Persona`, la cual cuenta con el atributo `nombre` únicamente. Al momento de instanciar dicha clase es asignado un *string* de longitud cero. Por otro lado, posee un método llamado `getNombre` que retorna el atributo `nombre` si y solo si su longitud es mayor a cero, en caso contrario retorna un valor nulo. Para simular el almacenamiento muerto, se instanciará una nueva persona pasando como parámetro explícitamente una cadena de caracteres vacía. Posteriormente se almacenará en una variable `nombre` y se intentará llamar al método `length`. Sin embargo, como la longitud es igual a cero, se genera una excepción y el programa termina inesperadamente como se muestra en la Figura 3.2d.

Bloqueo mutuo (*Deadlock*)

Un bloqueo mutuo sucede cuando dos o más hilos o procesos se encuentran a la espera de la liberación de un recurso compartido los unos con los otros, y por tanto no pueden proceder su ejecución hasta que alguno de estos libere el recurso [23].

Este tipo de error es detectado por Infer en el lenguaje de programación Java. Puede pasar desapercibido en el código, y se puede sospechar que ocurre cuando la aplicación o el programa deja de responder, como si se tratase de un bucle infinito. A continuación se muestra un fragmento de código donde Infer detectó este error:

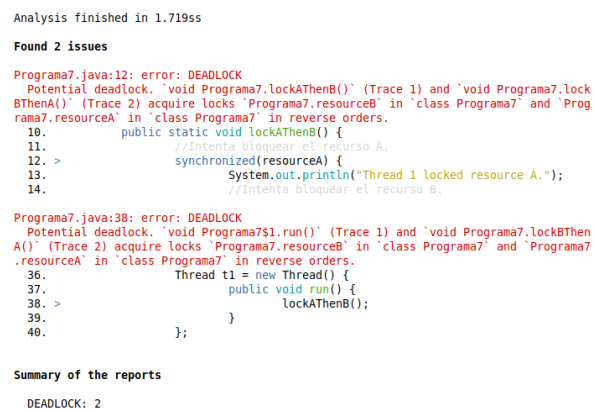
```
public class Persona {
                ...
                Thread t1 = new Thread() {
                        public void run() {
                                lockAThenB();
                        }
                };
                Thread t2 = new Thread() {
                        public void run() {
                                lockBThenA();
                        }
                };
                ...
                t1.start();
                t2.start();
                ...
}
```

De acuerdo con el código, existen dos recursos, uno `A` y otro `B`, que se consideran como un recurso crítico por las funciones `lockAThenB()` y `lockBThenA()`. Cada una de estas bloqueará ambos recursos hasta finalizar su ejecución en el orden que su nombre lo indica, debido a que los dos hilos tratarán de bloquear el acceso a los recursos pero en órdenes inversos, por ejemplo, dado un caso en el que `t1` bloqueará `A` y luego `t2` bloqueará `B`. Sin embargo el primero hilo `t1` no desbloqueará `A` hasta que finalice la función, por lo que necesita de `B`, pero dicho recurso está bloqueado por el segundo hilo y éste no puede proseguir porque `A` está siendo ocupado por el primer hilo. Dicho error es identificado como se muestra en la Figura 3.3a.

Atributo sin etiqueta Nullable (*Field not nullable*)

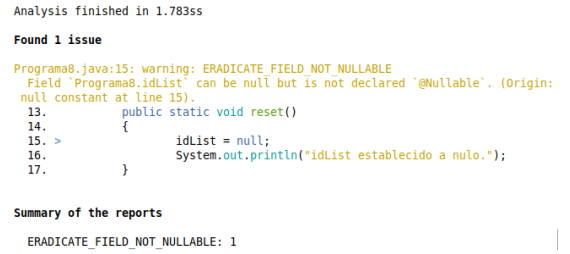
La herramienta Infer a través de su módulo `Eradicate` representa este tipo de error con una advertencia, y puede acontecer únicamente en el lenguaje de programación Java. Sucede cuando un atributo de una clase puede llegar a tomar un valor nulo en algún método, así como se muestra en la Figura 3.3b. El objetivo es erradicar todas las posibles excepciones de puntero nulo. Para usar este módulo al momento de ejecutar Infer, se debe agregar el parámetro `--eradicate-only` si sólo se desea examinar este tipo de advertencias, en caso contrario anexar el parámetro `--eradicate` para agregar estas advertencias al reporte de errores.

Para eliminar la advertencia se debe colocar la etiqueta `Nullable`, lo que permitirá que Infer pueda analizar el código en busca de instrucciones donde se quiera acceder a la referencia de ese atributo sin antes realizar una verificación, como se muestra en la Figura 3.3c.



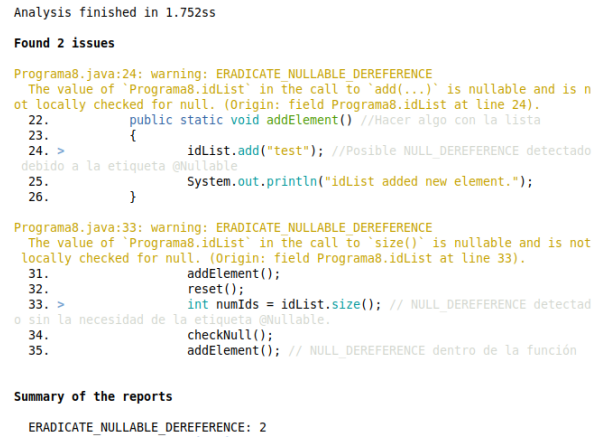
```
Analysis finished in 1.719ss

Found 2 issues

Programa7.java:12: error: DEADLOCK
  Potential deadlock. `void Programa7.lockAThenB()` (Trace 1) and `void Programa7.lock
BThenA()` (Trace 2) acquire locks `Programa7.resourceB` in `class Programa7` and `Prog
rama7.resourceA` in `class Programa7` in reverse orders.
  10.           public static void lockAThenB() {
  11.                   //Intenta bloquear el recurso A.
  12. >                 synchronized(resourceA) {
  13.                           System.out.println("Thread 1 locked resource A.");
  14.                           //Intenta bloquear el recurso B.

Programa7.java:38: error: DEADLOCK
  Potential deadlock. `void Programa7$1.run()` (Trace 1) and `void Programa7.lockBThen
A()` (Trace 2) acquire locks `Programa7.resourceB` in `class Programa7` and `Programa7
.resourceA` in `class Programa7` in reverse orders.
  36.                   Thread t1 = new Thread() {
  37.                           public void run() {
  38. >                                 lockAThenB();
  39.                           }
  40.                   };

Summary of the reports

  DEADLOCK: 2
```

(a) Bloqueo mutuo dentro de un programa en Java.



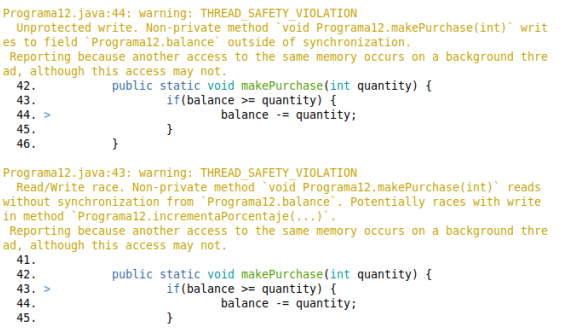
```
Analysis finished in 1.783ss

Found 1 issue

Programa8.java:15: warning: ERADICATE_FIELD_NOT_NULLABLE
  Field `Programa8.idList` can be null but is not declared `@Nullable`. (Origin:
 null constant at line 15).
  13.           public static void reset()
  14.           {
  15. >                 idList = null;
  16.                   System.out.println("idList establecido a nulo.");
  17.           }

Summary of the reports

  ERADICATE_FIELD_NOT_NULLABLE: 1
```

(b) Atributo sin etiqueta “Nullable” dentro de un programa en Java.



```
Analysis finished in 1.752ss

Found 2 issues

Programa8.java:24: warning: ERADICATE_NULLABLE_DEREFERENCE
  The value of `Programa8.idList` in the call to `add(...)` is nullable and is n
ot locally checked for null. (Origin: field Programa8.idList at line 24).
  22.           public static void addElement() //Hacer algo con la lista
  23.           {
  24. >                 idList.add("test"); //Posible NULL_DEREFERENCE detectado
 debido a la etiqueta @Nullable
  25.                   System.out.println("idList added new element.");
  26.           }

Programa8.java:33: warning: ERADICATE_NULLABLE_DEREFERENCE
  The value of `Programa8.idList` in the call to `size()` is nullable and is not
 locally checked for null. (Origin: field Programa8.idList at line 33).
  31.                   addElement();
  32.                   reset();
  33. >                 int numIds = idList.size(); // NULL_DEREFERENCE detectad
o sin la necesidad de la etiqueta @Nullable.
  34.                   checkNull();
  35.                   addElement(); // NULL_DEREFERENCE dentro de la función

Summary of the reports

  ERADICATE_NULLABLE_DEREFERENCE: 2
```

(c) Posible indirección nula dentro de un programa en Java.



```
Programa12.java:44: warning: THREAD_SAFETY_VIOLATION
  Unprotected write. Non-private method `void Programa12.makePurchase(int)` writ
es to field `Programa12.balance` outside of synchronization.
 Reporting because another access to the same memory occurs on a background thre
ad, although this access may not.
  42.           public static void makePurchase(int quantity) {
  43.                   if(balance >= quantity) {
  44. >                         balance -= quantity;
  45.                   }
  46.           }

Programa12.java:43: warning: THREAD_SAFETY_VIOLATION
  Read/Write race. Non-private method `void Programa12.makePurchase(int)` reads
without synchronization from `Programa12.balance`. Potentially races with write
in method `Programa12.incrementaPorcentaje(...)`.
 Reporting because another access to the same memory occurs on a background thre
ad, although this access may not.
  41.
  42.           public static void makePurchase(int quantity) {
  43. >                 if(balance >= quantity) {
  44.                           balance -= quantity;
  45.                   }
```

(d) Violación de la seguridad en hilos de un programa en Java.

Figura 3.3: Fragmentos de las salidas de los reportes obtenidos con Infer.

Violación de la seguridad en hilos (*Thread-safety violation*)

Esta advertencia es arrojada por Infer en un programa escrito en el lenguaje de programación Java. Se dice que “una clase cuenta con seguridad en hilos si se comporta correctamente cuando es accedida por múltiples hilos, independientemente de los intervalos de ejecución de estos.” [13]. Por lo tanto, existe una violación en la seguridad en hilos cuando se accede a atributos públicos de una clase, ya sea directamente o a través de un método público de manera concurrente, sin cumplir ciertas reglas que aseguren la exclusión mutua a la sección crítica de la memoria. Esto es válido tanto para lecturas como para escrituras de dichos atributos de la clase.

Por otro lado, dentro de un método público que puede ser accedido concurrentemente, si sólo cuenta con variables locales a dicho método, éste se considera que cuenta con seguridad en hilos, ya que cada uno de ellos contará con su propio espacio de memoria reservado para dicha variable [21].

```
Programa13.java:17: error: UNSAFE_GUARDED_BY_ACCESS
  The field `Programa13.balance` is annotated with `@GuardedBy("Programa13.this"
)`, but the lock `Programa13.this` is not held during the access to the field at
 line 17. Since the current method is non-private, it can be called from outside
 the current class without synchronization. Consider wrapping the access in a `s
ynchronized(Programa13.this)` block or making the method private.
  15.
  16.           public void makeDeposit(int amount) {
  17. >                 balance += amount;
  18.           }
  19.
```

Figura 3.4: Salida parcial del análisis de Infer de un programa en Java con un acceso inseguro.

Un ejemplo de violación de la seguridad en hilos puede darse de la siguiente manera. Se tienen dos hilos ejecutando al mismo tiempo, donde el hilo A accede a un método que lee el balance de una cuenta de banco, con el fin de realizar una compra para después descontar el saldo consumido de ésta, y el hilo B en ese momento realiza un depósito a esa misma cuenta, si no se posee seguridad en hilos se podrían llegar a tener una escritura sucia. Es decir, se puede dar el caso en que el hilo A accede a la dirección de memoria para leer el valor del balance, posteriormente valida si cuenta con saldo suficiente, pero en ese momento el hilo B accede al balance para incrementarlo debido al depósito, así termina su ejecución. Entonces el hilo A en ese momento escribe el balance después de haber realizado la compra, en ese momento el depósito realizado por el hilo B será como si nunca se hubiera ejecutado.

En la Figura 3.3d se muestra parte de la salida de Infer. La primera advertencia se obtiene sobre un método público llamado **makePurchase**, que trata de escribir sobre un atributo **balance**, sin embargo, como se muestra en la línea 44, esa no es una operación atómica, sino que internamente se divide en operaciones más pequeñas, por lo que mientras se realiza alguna de estas podría darse el caso de que otro hilo acceda a la misma dirección de memoria.

La segunda advertencia indica una carrera al momento de realizar la lectura de **balance**, ya que otro hilo podría actualizar el contenido, lo que se conoce como carrera por escritura, generando así lecturas sucias.

Acceso inseguro en la anotación "GuardedBy" (*Unsafe GuardedBy Access*)

La anotación **@GuardedBy** es usada en el lenguaje Java para indicar que sobre un atributo o método debe existir un bloqueo de la sección crítica. La herramienta Infer detecta este error cuando a algún atributo de una clase le es colocada la anotación y no existe ninguna instrucción de bloqueo durante su manipulación en algún otro método, ya sea lectura o escritura. Por ejemplo, cuando un método público que realiza operaciones sobre el atributo sea llamado en alguna otra función y no es realizado un bloqueo de la sección crítica correspondiente. Para evitar que suceda, se debe colocar el método de acceso privado o englobar las instrucciones que realizan operaciones sobre el atributo en un bloque **synchronized**.

En la Figura 3.4 se muestra una función que realiza un depósito sobre una variable llamada **balance**. Este atributo cuenta con la anotación **@GuardedBy**, por lo que al estar realizando una operación de lectura y escritura sobre ésta, Infer identifica el error, ya que cumple con ser una función pública que puede ser accedida externamente a la clase y que además no se bloquea el acceso al recurso mientras se manipula.

3.4. Verificación de un manejador de base de datos

Un sistema de gestión de bases de datos DBMS por sus siglas en inglés, es un software que permite realizar operaciones de lectura y escritura sobre una estructura de datos. Ofrece servicios de seguridad como control de acceso, para permitir o denegar el acceso a determinados recursos; disponibilidad de los datos, para asegurar que cada una de las peticiones de los usuarios son atendidas; integridad de los datos, al cumplir un conjunto de reglas especificadas por el administrador de bases de datos; entre otras funciones. Es por ello que debe ser un medio fiable y eficaz con el fin de evitar perdida o corrupción de dichos datos.

Es importante verificar formalmente sistemas de esta índole, debido a que son ampliamente utilizados en la industria y comercio, por ejemplo en sistemas bancarios donde se almacenan los saldos de los clientes, transacciones, prestamos, y todos aquellos servicios financieros que ofrecen. Por otro lado, en áreas orientadas a la salud, para almacenar los expedientes clínicos de los pacientes, generar pases de medicamentos en farmacia, por mencionar algunos. Pueden existir situaciones en las que un error puede costar desde una pérdida económica a una empresa u organización e inclusive la vida de una o más personas.

En este caso, la verificación fue realizada sobre el DBMS, conocido como MariaDB, debido a que es un software de código abierto y así es posible acceder a su código fuente para ejecutar el análisis. Además, tiene una gran presencia en el mercado actual. Empresas tanto financieras como orientadas a la producción de equipos médicos migraron sus bases de datos a este gestor, por ejemplo, Development Bank of Singapore, Ansell o Ingenico por mencionar algunas. Por otro lado, la comunidad es muy activa en el proyecto ya que la fundación está patrocinada por empresas como Microsoft, Alibaba Cloud, Tencent Cloud, entre otras más.

Dentro del código fuente de MariaDB, en su versión 10.5.2, se encuentran diversas carpetas separadas por funcionalidad. Para la verificación de este sistema y con el fin de aprovechar al máximo los beneficios que las lógicas de separación ofrecen, fueron elegidas dos carpetas en particular. La primera de ellas llamada **client** contiene programas individuales para interactuar con la base de datos, como lo es la línea de comandos, verificación y mantenimiento de la base de datos, importar tablas desde archivos de texto, y probar el funcionamiento de la base de datos, esto debido a que son una de las principales vías para interactuar con la base de datos del lado del cliente. Por otro lado, fue elegida la carpeta llamada **mysys** que contiene bibliotecas del sistema como lo son funciones de muy bajo nivel, que gestionan los recursos como la memoria, procesos y archivos.

Para llevar a cabo la verificación, fue empleada la versión de Infer 0.17.0, en el sistema operativo Ubuntu 18.04 de 64 bits con 4 vCPUs Intel Xeon E5-2697A @ 2.6GHz, sin embargo, puede es posible utilizar cualquier otro sistema operativo Linux o Mac, al momento de escribir este trabajo. El tiempo empleado en la fase de captura por parte de Infer fue de 26 minutos, sin contabilizar lo que demora el proceso de compilación. El tiempo total que incluye la compilación del DBMS y la fase de captura fue de 57 minutos.

Dependiendo de los cambios realizados entre versiones de código, en un desarrollo incremental, el proceso de análisis del programa puede diferir en el tiempo de ejecución, considerando que se emplean las mismas condiciones de *hardware* y *software*. En este caso, el código es una

versión estática y el análisis se limita a un conjunto de archivos. En la carpeta `client` el tiempo del de análisis tuvo una duración de 4 minutos con 30 segundos, en cambio, para la carpeta `mysys` fue de 2 minutos con 13 segundos. A continuación se describirán gran parte de los errores detectados por la herramienta en ambas carpetas y una posible solución para los que resulten válidos.

Carpeta client

En esta carpeta se encontró un total de 37 errores, de los cuales 18 son detecciones correctas. Los errores serán clasificados de acuerdo al tipo y a la carpeta en la que fueron encontrados, en este caso en la carpeta `client` se incluirán errores que no han sido reportados actualmente, así como algunas detecciones que resultaron en falsos positivos. En la Tabla 3.1 se muestra el detalle del análisis, entre paréntesis se muestra el número de líneas que posee cada archivo.

Almacenamientos muertos

El primer error fue localizado en el archivo `mysql.cc`. En la Figura 3.5a se muestra la salida presentada por Infer, en la cual se indica que el valor escrito en la variable `out` no es utilizado después de haber sido asignado. En la siguiente función se muestra resaltado de color azul las secciones de código relevantes para su análisis.

```
2277  static bool add_line(String &buffer, char *line, ...)
 ...    ...
2281    char buff[80], *pos, *out;
 ...    ...
2295    for (pos= out = line; pos < end_of_line; pos++)
2296    {
 ...      ...
2534      if (...)
2535      {
 ...        ...
2546        *out++='\n';
2547        length++;
2548      }
 ...      ...
```

La variable `out` se declara en la línea 2,281, corresponde al tipo puntero de carácter. Lineas después en el ciclo `for` le es asignado el valor que posee la variable `line`, por lo que ahora apunta a una dirección de memoria que fue recibida por los parámetros de la función. El almacenamiento muerto identificado es válido porque la operación `*out++='\n'` realizada dentro del ciclo actualiza el contenido del puntero `out` con un salto de línea y posteriormente realiza un incremento a la dirección de memoria a la que apunta. Debido a ello cuando el ciclo `for` finaliza, el incremento se convierte en una operación innecesaria, ya que dicho valor almacenado no es empleado nuevamente. Para corregir el almacenamiento muerto, se puede separar las operaciones de asignación e incremento, de la siguiente manera: `*out='\n';`. Posteriormente realizar una validación para que en la última iteración no sea ejecutado el incremento: `out++;` y así, el código no representa un problema. Probablemente la operación de comparación se realizaría más de una vez al encontrarse dentro del ciclo, por lo que se sugiere al equipo desarrollador replantear el código de manera que se evite el uso de la operación de incremento y asginación `++=` dentro de un ciclo `for`.

Tabla 3.1: Resumen del reporte de bugs en la carpeta client.

Archivo	Tipo de bug	Bugs detectados	Detecciones correctas
mysqltest.cc (11,349)	Indirección nula	3	3
	Almacenamiento muerto	3	2
	Valor sin inicializar	8	0
	Fuga de recursos	1	0
	Total	**15**	**5**
mysldump.c (6,321)	Almacenamiento muerto	2	2
	Indirección nula	1	1
	Total	**3**	**3**
mysql.cc (5,398)	Almacenamiento muerto	2	2
	Indirección nula	1	0
	Total	**3**	**2**
mysqladmin.cc (1,752)	Indirección nula	4	4
mysqlshow.c (932)	Almacenamiento muerto	2	2
mysql_plugin.c (1,215)	Indirección nula	1	1
mysqlslap.c (2,301)	Almacenamiento muerto	1	0
mysqlcheck.c (1,257)	Almacenamiento muerto	7	0
readline.cc (268)	Valor sin inicializar	1	0
	Total	**16**	**7**
Total		**37**	**17**

Posteriormente, en el mismo archivo se encuentra un segundo almacenamiento muerto mostrado en la Figura 3.5b. Éste indica que el valor almacenado en la variable `tmp` no es utilizado después de haber sido asignado en la línea 4,325. Por otro lado, en la línea 4,326 si la directiva del preprocesador se cumple, ejecuta una operación sobre la variable, en caso contrario el valor almacenado en `tmp` no es empleado, como se muestra a continuación.

```
...        ...
4325       tmp = strmake(buff, line, sizeof(buff) - 2);
4326   #ifdef EXTRA_DEBUG
4327       tmp[1] = 0;
4328   #endif
4329       tmp= get_arg(buff, GET);
...    ...
```

Lo más probable es que los programadores trataron de reutilizar código de una función existente y no se percataron que existe un caso en que la asignación a `tmp` resulta innecesaria

```
client/mysql.cc:2546: error: DEAD_STORE
  The value written to &out (type char*) is never used.
  2544.           SELECT '\ndelimiter\n';\n)
  2545.         */
  2546. >       *out++='\n';
  2547.         length++;
  2548.       }
```

(a) Fragmento del reporte de errores en el archivo mysql.cc.

```
client/mysql.cc:4325: error: DEAD_STORE
  The value written to &tmp (type char*) is never used.
  4323.         get_arg to find end of string the second time
it's called.
  4324.       */
  4325. >     tmp= strmake(buff, line, sizeof(buff)-2);
  4326.   #ifdef EXTRA_DEBUG
  4327.       tmp[1]= 0;
```

(b) Fragmento del reporte de errores en el archivo mysql.cc.

```
client/mysqldump.c:1628: error: DEAD_STORE
  The value written to &query_ptr (type char*) is never used.
  1626.     query_ptr= strnmov(query_ptr, C_STRING_WITH_LEN
("*/ /*!"));
  1627.     query_ptr= strnmov(query_ptr, stmt_version_str,
stmt_version_length);
  1628. >   query_ptr= strxmov(query_ptr, definer_end, NullS);
  1629.
  1630.     return query_str;
```

(c) Fragmento del reporte de errores en el archivo mysqldump.c.

```
client/mysqltest.cc:10023: error: DEAD_STORE
  The value written to &to (type char*) is never used.
  10021.     while (*from)
  10022.     {
  10023. >     char *to= buff;
  10024.       to= get_string(&buff, &from, command);
  10025.       if (!*from)
```

(d) Fragmento del reporte de errores en el archivo mysqltest.cc.

Figura 3.5: Salida de Infer acerca de algunos almacenamientos muertos.

en la línea 4,325, resultando en un almacenamiento muerto genuino, debido a que la función `strmake` recibe por referencia los parámetros necesarios. Para corregirlo, basta con agregar un caso `else` a la directiva de pre-procesamiento omitiendo la asignación a `tmp`, y mover la línea 4,325 arriba de la asignación `tmp[1]= 0;`, evitando así una operación de asignación innecesaria, como se muestra en el siguiente código:

```
...   ...
4325  #ifdef EXTRA_DEBUG
4326        tmp = strmake(buff, line, sizeof(buff) - 2);
4327        tmp[1]= 0;
4328  #else
4329        strmake(buff, line, sizeof(buff) - 2);
4330  #endif
4331        tmp= get_arg(buff, GET);
...   ...
```

Para continuar, en el archivo llamado `mysqldump.c` se encuentra otro almacenamiento muerto, en este caso la variable `query_prt` es utilizada como auxiliar para mover una o más cadenas de caracteres a `query_srt`. La salida de Infer se muestra en la Figura 3.5c. En el código, la última asignación destacada por Infer en la línea 1,628 no es necesaria, puesto que la función retorna un valor y termina como se muestra en el siguiente fragmento de código:

```
...    ...
1604   char *query_str= NULL;
1605   char *query_ptr;
...    ...
1620   query_str= alloc_query_str(stmt_length + 23);
1621
1622   query_ptr= strnmov(query_str, stmt_str, definer_begin - stmt_str);
...    ...
1628   query_ptr = strxmov(query_ptr, definer_end, NullS);
1629
1630   return query_str;
1631 }
... ...
```

Si se desea corregir el error, simplemente debe omitirse la asignación en la línea señalada, así se ahorra tiempo de procesamiento, puesto que se evita un acceso a memoria para la escritura del nuevo valor. El código modificado se muestra a continuación:

```
...    ...
1622   query_ptr= strnmov(query_str, stmt_str, definer_begin - stmt_str);
...    ...
1628   strxmov(query_ptr, definer_end, NullS);
1629
1630   return query_str;
1631 }
... ...
```

Como siguiente error se tiene el código ubicado en la carpeta `mysqltest.cc`. Se trata de otro almacenamiento muerto en la línea 10,023. Este caso es muy fácil de distinguir, puesto que el valor al puntero `to` se escribe en la línea 10,023 como se muestra en la Figura 3.5d. Justamente una línea posterior se vuelve a sobre-escribir el valor que retorna la función `get_string()`, por lo que para corregir este tipo de error basta con eliminar la primer asignación que fue indicada en el reporte.

En esta carpeta se muestra un último almacenamiento muerto, como se observa en la Figura 3.6a, Infer indica que el valor escrito en la variable `s` no es utilizado después de ser asignado en la línea 11,332. Si se observa el siguiente fragmento de código:

```
...    ...
11321    char *s = buf;
...      ...
11326    if (!fgets(buf, buf_len-1, stdin))
11327      buf[0]= 0;
11328    else if (buf[0] && (s = strend(buf))[-1] == '\n')
11329      s[-1]= 0;
11330
11331    for (s = buf; *s; s++)
11332      fputc(type == 2 ? '*' : *s, stdout);
11333
11334    fputc('\n', stdout);
11335
11336    return buf;
11337  }
...    ...
```

Esta función declara un puntero `s` y lo inicializa con la dirección de memoria a la que apunta `buf`. Sin embargo, en ninguna situación se realiza una lectura del valor, tanto en la línea 11,329 como en la 11,332 se sobre-escribe la dirección de memoria a la que apunta `s`, resultando así en una detección correcta por parte de la herramienta. Para corregir este error es suficiente con eliminar la línea indicada por Infer donde se inicializa la variable.

Indirecciones nulas

A continuación serán descritos las indirecciones nulas encontradas en la carpeta `client`. La primera de estas se muestra en la Figura 3.6b, donde Infer indica que el puntero llamado `destination` que recibió una dirección de memoria en la línea 194, podría llegar a ser nula y se realiza la operación de indirección en la línea siguiente. Observe la función llamada `my_strndup` en el siguiente código:

```
...  ...
237    char *my_strndup(PSI_memory_key key, const char *from, size_t length, myf
       my_flags)
238  {
239    char *ptr;
240    DBUG_ENTER("my_strndup");
241
242    if ((ptr= (char*) my_malloc(key,length+1,my_flags)))
243    {
244      memcpy(ptr, from, length);
245      ptr[length]= 0;
246    }
247    DBUG_RETURN(ptr);
248  }
...  ...
```

El puntero `ptr` efectivamente puede ser nulo, puesto que en la función `my_malloc` existe un caso en el que no haya suficiente espacio de memoria para reservar y dicha función devuelva un puntero nulo, lo que se traduce en una detección correcta por parte de Infer.

Para prevenir este tipo de errores, debe establecerse una condición de validación antes de la indirección de cualquier puntero que pudiera tomar un valor nulo. En este caso, sería colocada para proteger el acceso a la memoria en la línea 195, como se muestra a continuación:

```
...      ...
193      s = line + item_len + 1;
194      destination= my_strndup(PSI_NOT_INSTRUMENTED, s, line_len - start, MYF(
       MY_FAE));
195      if(destination)
196        destination[line_len - item_len - 2]= 0;
197    }
198    return destination;
199  }
...  ...
```

```
client/mysqltest.cc:11332: error: DEAD_STORE
  The value written to &s (type char*) is never used.
  11330.                                          char *buf,
int buf_len)
  11331.   {
  11332. >   char *s=buf;
  11333.
  11334.     fputs(prompt, stdout);
```

(a) Fragmento del reporte de errores en el archivo mysqltest.cc.

```
client/mysql_plugin.c:195: error: NULL_DEREFERENCE
  pointer `destination` last assigned on line 194 could be
null and is dereferenced at line 195, column 5.
  193.       s = line + item_len + 1;
  194.       destination= my_strndup(PSI_NOT_INSTRUMENTED, s,
line_len - start, MYF(MY_FAE));
  195. >     destination[line_len - item_len - 2]= 0;
  196.     }
  197.     return destination;
```

(b) Fragmento del reporte de errores en el archivo mysql_plugin.cc.

```
client/mysql.cc:2596: error: NULL_DEREFERENCE
  pointer `ptr` last assigned on line 2591 could be null and
is dereferenced at line 2596, column 10.
  2594.
  2595.     /* find out how many lines we have and remove
newlines */
  2596. >   while (*ptr != '\0')
  2597.     {
  2598.       switch (*ptr) {
```

(c) Fragmento del reporte de errores en el archivo mysql.cc.

```
client/mysqldump.c:2164: error: NULL_DEREFERENCE
  pointer `lengths` last assigned on line 2136 could be null and
is dereferenced at line 2164, column 49.
  2162.             print_quoted_xml(xml_file, "Aria", sizeof
("Aria") - 1, 0);
  2163.           else
  2164. >           print_quoted_xml(xml_file, (*row)[i],
lengths[i], 0);
  2165.           fputc('"', xml_file);
  2166.           check_io(xml_file);
```

(d) Fragmento del reporte de errores en el archivo mysqladmin.cc.

Figura 3.6: Salida de Infer acerca de la posible detección de un almacenamiento muerto y tres indirecciones nulas.

Si la variable `destination` contiene una dirección de memoria nula, la condición es falsa y no realiza la indirección. En caso contrario, sigue funcionando normalmente. Cabe resaltar que al no ejecutarse dicha operación, también debe ser incluido un caso `else` que soporte dicha excepción, para evitar el mal funcionamiento del sistema.

La siguiente indirección nula se muestra en la Figura 3.6c, indica que el puntero asignado en `char *ptr = final_command->c_ptr();` línea 2,596 realiza la indirección y podría tomar un valor nulo, lo que ocasionaría un cierre inesperado del programa. En ese caso al ser un archivo de C++, `final_command` corresponde a una instancia de una clase llamada `String`, definida en otro archivo. Esta clase posee un método llamado `c_ptr()`, que retorna un puntero de tipo carácter. Para modificar el contenido de dicha propiedad existe un método llamado `set`, como se muestra a continuación:

```
...     ...
249     void set(char *str, size_t len)
250     {
251       Ptr = str;
252       str_length= (uint32) len;
253     }
...     ...
```

La indirección se realiza sin dar importancia al contenido del puntero, sin embargo, a pesar de que `set` pueda contener un valor nulo, se realizan las validaciones pertinentes para evitar cualquier problema con la indirección. La creación de la variable `ptr` se muestra en la línea 2,591 de la siguiente manera:

```
...   ...
2588  static void fix_history(String *final_command)
2589  {
2590    int total_lines = 1;
2591    char *ptr = final_command->c_ptr();
2592    ...
2596    while(*ptr != '\0')
2597    {
...     ...
```

Como la variable recibida por parámetros `final_command` pertenece a una clase llamada String de un código escrito en C++, si se analiza el archivo ubicado en la ruta `sql/sql_string.h`, se encuentra definida la estructura de datos de la siguiente manera: `class String: public Charset, public Binary_string`. Se observa que mediante herencia deriva de las clases `Charset` y `Binary_string`, el método que se busca con el nombre `c_ptr` se encuentra en la clase `Charset` y contiene lo siguiente:

```
...   ...
601   inline char *c_ptr()
602   {
603     DBUG_ASSERT(!alloced||!Ptr||!Alloced_length||
604     (Alloced_length >= (str_length+1)));
605
606     if (!Ptr || Ptr[str_length])      // Should be safe
607       (void) realloc(str_length);
608     return Ptr;
609   }
...   ...
```

En la línea 603 se muestra que las validaciones son realizadas de manera adecuada, con esto se concluye que el código está libre de fallos y se considera como un falso positivo, en este caso, probablemente porque la validación se está realizando a través de la función `DBUG_ASSERT` y la herramienta no es capaz de identificarlo.

Para continuar, Infer encontró otra indirección nula en el archivo `mysqldump.c`, en este caso dentro de los parámetros que recibe la función señalada en la Figura 3.6d se encuentra `lengths[i]`, sobre la que se realiza una operación de indirección para ser empleada en la escritura a un archivo.

A continuación se observa la declaración de la variable `lenghts`, es de tipo `long` sin signo y recibe valor a través de la función `mysql_fetch_lengths(...)`.

```
...     ...
2134    ulong create_stmt_len= 0;
2135    MYSQL_FIELD *field;
2136    ulong *lengths= mysql_fetch_lengths(tableRes);
...     ...
```

Analizando dicha función, se encuentra un caso en el que es posible que retorne una dirección nula, como se indica de color en la línea 3,787. Es por ello que resulta ser una detección correcta.

```
...   ...
3782  mysql_fetch_lengths(MYSQL_RES *res)
3783  {
3784    MYSQL_ROW column;
3785
3786    if (!(column=res->current_row))
3787      return 0;          /* Something is wrong */
3788    if (res->data)
3789      (*res->methods->fetch_lengths)(res->lengths, column, res->field_count);
3790    return res->lengths;
3791  }
...   ...
```

Para solucionarlo, se realiza algo similar a los casos anteriores. Basta con verificar si la variable `lenghts` toma un valor nulo, de ser así, dependiendo de la gravedad del asunto se realiza una acción que evite la indirección, y se indica que ha ocurrido algún error, en caso contrario procesar la excepción sin afectar el flujo del programa.

Valores sin inicializar

De acuerdo con la Figura 3.7a, este error indica que la variable `ds_prepare_warnings.length` nunca es inicializada, sin embargo, al analizar el código manualmente para determinar si es válido o no, existe una variable global llamada `disable_warnings`, la cual posee un valor de cero a lo largo de la ejecución de todo el programa. A continuación se muestra el fragmento del código donde es empleada y se relaciona con el error:

```
...     ...
8416    if(!disable_warnings)
8417    {
8418      init_dynamic_string(&ds_prepare_warnings, NULL, 0, 256);
8419      init_dynamic_string(&ds_execute_warnings, NULL, 0, 256);
8420    }
...     ...
8454      if(!disable_warnings)
8455        ...
8459        if (append_warnings(&ds_execute_warnings, mysql) ||
8460            ds_execute_warnings.length ||
8461            ds_prepare_warnings.length ||
8462            ds_warnings->length)
8463        {
...         ...
```

La lectura del valor se realiza en la línea 8,461, y la inicialización del valor se encuentra ubicada en la línea 8,418. Sin embargo, la variable únicamente será inicializada en el momento en que la condición en la línea 8,416 sea verdadera. De acuerdo a como está programado el código, la variable `disable_warnings` nunca podrá tomar un valor distinto de cero, por lo que siempre se cumplirá la condición y se inicializará el valor, es por ello que es clasificada como un falso positivo.

Probablemente la herramienta lo reporta como válido, porque se declara como una variable `static my_bool disable_warnings= 0` en lugar de ser definida como una constante. En la Figura 3.7b se muestra que Infer detecta este mismo error en líneas posteriores, que deriva del mismo problema.

```
client/mysqltest.cc:8461: error: UNINITIALIZED_VALUE
  The value read from ds_prepare_warnings.length was never
initialized.
  8459.         if (append_warnings(&ds_execute_warnings, mysql)
||
  8460.             ds_execute_warnings.length ||
  8461. >           ds_prepare_warnings.length ||
  8462.             ds_warnings->length)
  8463.         {
```

(a) Fragmento del reporte de errores en el archivo mysqltest.cc.

```
client/mysqltest.cc:8468: error: UNINITIALIZED_VALUE
  The value read from ds_prepare_warnings.length was never
initialized.
  8466.             dynstr_append_mem(ds, ds_warnings->str,
  8467.                               ds_warnings->length);
  8468. >         if (ds_prepare_warnings.length)
  8469.             dynstr_append_mem(ds,
ds_prepare_warnings.str,
  8470.
ds_prepare_warnings.length);
```

(b) Fragmento del reporte de errores en el archivo mysqltest.cc.

```
client/mysqltest.cc:10311: error: UNINITIALIZED_VALUE
  The value read from re.icase was never initialized.
  10309.       get_dynamic(&r->regex_arr, &re, i);
  10310.
  10311. >     if (!reg_replace(&out_buf, buf_len_p, re.pattern,
re.replace,
  10312.                      in_buf, re.icase))
  10313.       {
```

(c) Fragmento del reporte de errores en el archivo mysqltest.cc.

```
client/readline.cc:203: error: UNINITIALIZED_VALUE
  The value read from read_count was never initialized.
  201.
  202.     /* Read in new stuff. */
  203. >   if ((read_count= my_read(buffer->file, (uchar*)
buffer->end, read_count,
  204.                             MYF(MY_WME))) ==
MY_FILE_ERROR)
  205.     {
```

(d) Fragmento del reporte de errores en el archivo readline.cc.

Figura 3.7: Salida de Infer acerca de la detección de valores sin inicializar.

El siguiente valor sin inicializar fue detectado en el mismo archivo, como se muestra en la Figura 3.7c. En este caso, la variable `re.icase` es leída en la línea 10,311, pero nunca es inicializada según el reporte de error. Al analizar el código, la última asignación se realiza en la línea 10,309, que realiza una llamada a la función `get_dynamic`, en la cual se envía como segundo parámetro la dirección de memoria de la variable `re`. A continuación se muestra dicha función a detalle.

```
...  ...
277  void get_dynamic(DYNAMIC_ARRAY *array, void *element, uint idx)
278  {
279    if (idx >= array->elements)
...      ...
283      bzero(element, array->size_of_element);
284      return;
285    }
286    memcpy(element,array->buffer+idx*array->size_of_element,
287           (size_t) array->size_of_element);
288  }
...  ...
```

Dentro de la función `get_dynamic` se inicializa el valor con ceros, únicamente si la condición en la línea 279 se cumple. Posteriormente en la línea 286 realiza una copia del primer parámetro llamado `array`, por lo que la variable es inicializada con instrucciones de bajo nivel. Es por ello que se considera un falso positivo, debido a que Infer no es capaz de reconocerlo.

Finalmente, en la Figura 3.7d se muestra otro caso de una variable sin inicializar, es llamada `read_count` y es de tipo `size_t`. La lectura del valor se realiza en la línea 203, como se muestra en el reporte, debido a que la variable es recibida por parámetros dentro de la función `my_read`.

Sin embargo, en el siguiente código se observa claramente que el valor de la variable es inicializada en la línea 172, es por lo que se considera un falso positivo, ya que a pesar de encontrarse dentro de un ciclo `for`, éste siempre se ejecuta y probablemente esté relacionado con el falso positivo anterior, en el que Infer no es capaz de reconocer este tipo de inicialización.

```
...     ...
161     size_t read_count;
162     uint bufbytes= (uint) (buffer->end - buffer->start_of_line);
...     ...
169     for (;;)
170     {
171       uint start_offset=(uint) (buffer->start_of_line - buffer->buffer);
172       read_count = (buffer- > bufread - bufbytes)/IO_SIZE;
173       if ((read_count*=IO_SIZE))
174         break;
...           ...
```

Carpeta mysys

En la Tabla 3.2 se muestra el detalle del análisis realizado por Infer, se representa entre paréntesis el número de líneas contenido en cada archivo, así como los errores identificados y aquellos que resultaron correctos.

Almacenamientos muertos

Un almacenamiento muerto detectado en esta carpeta se encuentra en el archivo `ma_dyncol.c`, como se muestra en la Figura 3.8a. Dentro de el ciclo `for` se encuentra una asignación a la variable `prev_name` en la línea 3,760. A continuación se muestra el fragmento del código que precede al mostrado en el reporte.

```
...     ...
3679    if (header.format == dyncol_fmt_num)
3680    {
3681      key= &num;
3682      prev_key= &prev_num;
3683    }
3684    else
3685    {
3686      key= &name;
3687      prev_key = &prev_name;
3688    }
3689    for (i= 0, header.entry= header.header;
3690         i < header.column_count;
3691         i++, header.entry+= header.entry_size)
3692    {
...       ...
3752      if ((*fmt->column_sort)(&prev_key, &key) >= 0)
3753      {
...         ...
```

Tabla 3.2: Resumen del reporte de *bugs* en la carpeta mysys.

Archivo	**Tipo de bug**	**Bugs detectados**	**Detecciones correctas**
lf_hash.c (576)	Valor sin inicializar	3	0
my_bitmap.c (745)		1	0
lf_alloc-pin.c (535)		1	0
my_atomic_writes.c (334)		1	0
my_pread.c (201)		1	0
my_write.c (121)		1	0
my_read.c (113)		1	0
	Total	**9**	**0**
ma_dyncol.c (4,427)	Almacenamiento muerto	3	1
mf_iocache.c (2,131)		2	1
tree.c (805)		1	1
string.c (230)		1	1
my_copy.c (152)		1	1
my_seek.c (104)		1	1
my_default.c (1,066)		1	0
	Total	**10**	**6**
mf_keycache.c (6,578)	Valor sin inicializar	1	0
	Indirección nula	2	0
	Total	**3**	**0**
my_getopt.c (1,704)	Valor sin inicializar	1	0
	Almacenamiento muerto	2	1
	Total	**3**	**1**
thr_timer.c (600)	Indirección nula	1	1
my_create.c (61)	Fuga de recursos	1	0
my_getwd.c (169)	Fuga de memoria	1	0
	Total	**3**	**1**
Total		**28**	**8**

Se observa que existe otra variable que apunta a la misma dirección de memoria que `prev_name` en la línea 3,687, y posteriormente se envía como parámetro de la función en la línea 3,752. La variable `fmt` corresponde a un puntero declarada dentro del mismo archivo como: `struct st_service_funcs *fmt;`. En esta estructura se encuentra el miembro `column_sort` que corresponde a un puntero de función en C, como se muestra en el código inferior.

```
...     ...
578  struct st_service_funcs
579  {
580    ...
581    int (*column_sort)
582          (const void *a, const void *b);
...     ...
```

Finalmente, al seguir el rastro de la función a la cual el puntero hace referencia, los valores almacenados en las variables son realmente utilizados como se muestra a continuación.

```
...     ...
228  int mariadb_dyncol_column_cmp_named(const LEX_STRING *s1, const LEX_STRING *s2)
229{
230    ...
234  int rc= (s1->length > s2->length ? 1 :
235           (s1->length < s2->length ? -1 : 0));
236  if (rc == 0)
237    rc= memcmp((void *)s1->str, (void *)s2->str,
238               (size_t) s1->length);
239  return rc;
240}
...     ...
```

Por lo tanto, la detección que realiza esta herramienta es un falso positivo, probablemente se deba a que no cuenta con las adecuaciones necesarias para seguir el rastro del tratamiento de punteros a funciones, dado que las lógicas de separación permiten la identificación de los punteros de manera precisa.

El siguiente error se encuentra en el archivo `mf_iocache.c`, como se muestra en la Figura 3.8b. En el reporte, se describe que el valor almacenado en la variable `rc` de tipo entero nunca es utilizado, sin embargo, en la línea 1,371 se observa que esa variable se emplea como parámetro de un macro. Una macro es una sección de código definida en tiempo de compilación a través de la directiva de pre-procesamiento `#define`. De acuerdo a ello, probablemente Infer no sea capaz de reconocerlo como una forma válida de lectura a la variable, sin embargo, el valor realmente está siendo utilizado, por lo que se considera como un falso positivo.

En la Figura 3.8c, se muestra otro posible almacenamiento muerto detectado por la herramienta, este caso resultó ser una detección correcta puesto que al analizar el código la variable únicamente se declara y se inicializa con el valor cero, como se muestra a continuación:

```
...   ...
1401  static int _my_b_seq_read(IO_CACHE *info, uchar *Buffer, size_t Count)
1402  {
1403    size_t length, diff_length, left_length= 0, save_count, max_length;
1404    my_off_t pos_in_file;
1405    save_count=Count;
...     ...
```

Posteriormente, dentro de la función donde se encuentra el error, se realiza una operación de suma en la línea 1,454, como se puede observar en el reporte del fallo. Es por ello que el valor almacenado no es utilizado y corresponde a una detección correcta por parte de la herramienta.

```
mysys/ma_dyncol.c:3760: error: DEAD_STORE
  The value written to &prev_name (type st_mysql_lex_string) is
never used.
  3758.       }
  3759.       prev_num= num;
  3760. >     prev_name= name;
  3761.       prev_data_offset= data_offset;
  3762.       prev_name_offset= name_offset;
```

(a) Fragmento del reporte de errores en el archivo ma_dyncol.c.

```
mysys/mf_iocache.c:1369: error: DEAD_STORE
  The value written to &rc (type int) is never used.
  1367.       int  __attribute__((unused)) rc;
  1368.
  1369. >     rc= lock_io_cache(write_cache, pos_in_file);
  1370.       /* The writing thread does always have the lock
when it awakes. */
  1371.       DBUG_ASSERT(rc);
```

(b) Fragmento del reporte de errores en el archivo mf_iocache.c.

```
mysys/mf_iocache.c:1454: error: DEAD_STORE
  The value written to &left_length (type unsigned long) is
never used.
  1452.         goto read_append_buffer;
  1453.       }
  1454. >     left_length+=length;
  1455.       diff_length=0;
  1456.     }
```

(c) Fragmento del reporte de errores en el archivo mf_iocache.c.

```
mysys/mf_qsort.c:133: error: DEAD_STORE
  The value written to &b (type char**) is never used.
  131.          for (ptr = low_ptr; ptr > low && CMP(ptr - size,
ptr) > 0;
  132.               ptr -= size)
  133. >          SWAP(ptr, ptr - size, size, ptr_cmp);
  134.         }
  135.         POP(low, high);
```

(d) Fragmento del reporte de errores en el archivo mf_qsort.c.

Figura 3.8: Salida de Infer acerca de la detección de almacenamientos muertos.

En este caso, un detalle importante a resaltar es que la variable no se encuentra dentro de algún tipo de ciclo, debido a que Infer es capaz de reconocer aquellas variables que deban ser inicializadas en cero con el fin de emplearse como acumuladores. Al no ser el caso, debido a que la instrucción `left_length+=length;` sólo es ejecutada una única vez, no tiene sentido inicializar la variable en cero y bastaría con asignar el valor de la siguiente manera: `left_length = length;`.

Para continuar con los almacenamientos muertos, en la Figura 3.8d se muestra que el valor almacenado en la variable `a` no es empleada, sin embargo, en el reporte no se muestra el código que la contiene debido a que se encuentra definida la función dentro de una macro mediante la directiva de preprocesador, a continuación se muestra el código correspondiente a dicha macro.

```
...  ...
36  #define SWAP(A, B, size,swap_ptrs)        \
37  do {                                      \
38      if (swap_ptrs)                        \
39      {                                     \
40        reg1 char **a = (char**) (A), **b = (char**) (B);   \
41        char *tmp = *a; *a++ = *b; *b++ = tmp;      \
42      }
...     ...
```

En la macro mostrada, a partir de la línea marcada de color azul, es donde se origina el supuesto almacenamiento muerto. Dicha operación se encuentra dentro de un ciclo `do-while`. En la sentencia `*a++ = *b` se realiza una operación de asignación a la variable `a` y finalmente se incrementa en una unidad la dirección a la que hace referencia.

Se considera un almacenamiento muerto, porque en la última iteración del ciclo es en donde la operación de incremento sobre `a` se realiza, y su valor no es leído en instrucciones posteriores. Sin embargo, al igual que el primer almacenamiento muerto mostrado en la carpeta `client`, la forma de corregirlo es dividiendo las operaciones y agregando alguna validación previa que evite dicho incremento sobre `a`.

Por otro lado, la mejor alternativa es reestructurar el código de manera que esto no suceda, ya que en determinadas iteraciones del ciclo se estará realizando la validación de la condición. Se invita a los desarrolladores a evitar el uso de dicha instrucción dentro de un ciclo, sin embargo, no resulta una afectación de impacto al programa.

Posteriormente, la herramienta encontró otro almacenamiento muerto en el archivo `my_copy.c`, éste resultó ser una detección correcta. En la Figura 3.9a se muestra que el valor almacenado en la variable `from_file` en la línea 65 nunca es leído o utilizado por alguna otra instrucción. Dentro de la función se encuentra la declaración de la variable, posteriormente se inicializa como se indica anteriormente y luego en la línea 70 se sobrescribe el valor que contiene la variable, como se muestra a continuación:

```
...  ...
57     int create_flag;
58     File from_file, to_file;
59     uchar buff[IO_SIZE];
60     MY_STAT stat_buff, new_stat_buff;
61     my_bool file_created= 0;
...    ...
65     from_file = to_file = -1;
66     DBUG_ASSERT(!(MyFlags & (MY_FNABP | MY_NABP))); /* for my_read/my_write */
67     if (MyFlags & MY_HOLD_ORIGINAL_MODES)    /* Copy stat if possible */
68       new_file_stat= MY_TEST(my_stat((char*) to, &new_stat_buff, MYF(0)));
69
70     if ((from_file = my_open(from, O_RDONLY | O_SHARE, MyFlags)) >= 0)
71     {
...      ...
```

Por tanto, es cierto que allí ocurre un almacenamiento muerto y para corregirlo basta con eliminar la asignación de la línea marcada por la herramienta, resultando de la siguiente manera: `to_file= -1;`.

El siguiente almacenamiento muerto fue detectado en el archivo `my_getopt.c`. La salida de Infer se muestra en la Figura 3.9b, la cual indica que el contenido escrito en la variable `is_cmdline_arg` en la línea 200 no es utilizado posteriormente a su asignación. Al analizar el código de forma manual se encontró que es una detección correcta, puesto que no se realiza ninguna lectura sobre ese valor. Posteriormente en la línea 210, como se muestra a continuación, un nuevo contenido es asignado a dicha variable.

```
mysys/my_copy.c:65: error: DEAD_STORE
  The value written to &from_file (type int) is never used.
  63.     DBUG_PRINT("my",("from %s to %s MyFlags %lu", from,
to, MyFlags));
  64.
  65. >   from_file=to_file= -1;
  66.     DBUG_ASSERT(!(MyFlags & (MY_FNABP | MY_NABP))); /* for
my_read/my_write */
  67.     if (MyFlags & MY_HOLD_ORIGINAL_MODES)         /* Copy
stat if possible */
```

(a) Fragmento del reporte de errores en el archivo my_copy.c.

```
mysys/my_getopt.c:200: error: DEAD_STORE
  The value written to &is_cmdline_arg (type char) is never
used.
  198.     void *value;
  199.     int error, i;
  200. >   my_bool is_cmdline_arg= 1;
  201.     DBUG_ENTER("handle_options");
  202.
```

(b) Fragmento del reporte de errores en el archivo my_getopt.c.

```
mysys/my_seek.c:49: error: DEAD_STORE
  The value written to &newpos (type long) is never used.
  47.   my_off_t my_seek(File fd, my_off_t pos, int whence, myf
MyFlags)
  48.   {
  49. >   os_off_t newpos= -1;
  50.     DBUG_ENTER("my_seek");
  51.     DBUG_PRINT("my",("fd: %d Pos: %llu  Whence: %d
MyFlags: %lu",
```

(c) Fragmento del reporte de errores en el archivo my_seek.c.

```
mysys/string.c:163: error: DEAD_STORE
  The value written to &next_pos (type char const *) is never
used.
  161.     {
  162.       const char  *cur_pos= append;
  163. >     const char *next_pos= cur_pos;
  164.
  165.       /* Search for quote in each string and replace with
escaped quote */
```

(d) Fragmento del reporte de errores en el archivo string.c.

Figura 3.9: Salida de Infer acerca de la detección de almacenamientos muertos.

```
...     ...
199     int error, i;
200     my_bool is_cmdline_arg = 1;
...     ...
204     DBUG_ASSERT(*argc >= 1);
205     DBUG_ASSERT(*argv);
206     (*argc)--; /* Skip the program name */
207     (*argv)++; /*      --- || ---       */
208     init_variables(longopts, init_one_value);
209
210     is_cmdline_arg = !is_file_marker(**argv);
211
212     for (pos= *argv, pos_end=pos+ *argc; pos != pos_end ; pos++)
213     {
...     ...
```

En el fragmento de código mostrado se realizó una asignación en la línea 210, por lo que no es necesario inicializar la variable si posteriormente el valor se sobrescribirá. Para corregir el error bastaría con eliminar la inicialización de la variable para que únicamente se declare como `my_bool is_cmdline_arg;`.

Continuando con el mismo tipo de error, en la Figura 3.9c se muestra que la variable `newpos` es inicializada en la línea número 49, y ese valor no es utilizado en una instrucción posterior. A continuación se muestra el fragmento de código perteneciente a la función `my_seek`.

```
...   ...
47    my_off_t my_seek(File fd, my_off_t pos, int whence, myf MyFlags)
48    {
49      os_off_t newpos= -1;
...     ...
58      DBUG_ASSERT(fd >= 0);
59    #ifdef _WIN32
60      newpos= my_win_lseek(fd, pos, whence);
61    #else
62      newpos= lseek(fd, pos, whence);
63    #endif
64      if (newpos == (os_off_t) -1)
...     ...
```

En dicha función se sobrescribe el valor de `newpos`, probablemente con el fin de definir un valor por defecto a la variable, sin embargo, resulta en un almacenamiento muerto acertado, puesto que el hecho de inicializar la variable no modifica el comportamiento del programa. Para solucionarlo, basta con eliminar la inicialización en la línea 49, con el fin de aprovechar mejor el tiempo de procesamiento, evitando la ejecución de una instrucción innecesaria.

Otra situación similar a la anterior fue detectada en el archivo `string.c`. En la Figura 3.9d se muestra el reporte del error indicando un almacenamiento muerto en la variable `next_pos`. A continuación se muestra un fragmento del código.

```
...     ...
160     while (append != NullS)
161     {
162       const char  *cur_pos= append;
163        const char *next_pos= cur_pos;
164
165      /* Search for quote in each string and replace with escaped quote */
166       while(*(next_pos= strcend(cur_pos, quote.str[0])) != '\0')
...       ...
```

Esta variable recibe un valor en la línea 163. Después, en la línea 166 el contenido es sobrescrito por otra instrucción, por lo que se considera una detección correcta por parte de Infer. Claramente, es un almacenamiento muerto, porque se está inicializando la variable con el valor contenido en `cur_post` y no es utilizado posteriormente, ya que se vuelve a asignar un nuevo valor a la variable `next_post`. Para solucionarlo se debe eliminar la inicialización a esta última variable, resultando de la siguiente manera: `const char *next_post;`.

Indirección nula

Para finalizar, se mostrará un falso positivo de una indirección nula encontrada en la carpeta `mysys`. La Figura 3.10 muestra parte de la salida expuesta por Infer en el archivo `mf_keycache.c`, indica que la última asignación de la variable `thread->next` de tipo puntero es en la línea 1,738, y podría resultar nula, dado que se realiza una indirección en la llamada de la función `unlink_from_queue()` en esa misma línea. Dicha función recibe como primer parámetro una estructura a una cola doblemente ligada que almacena referencias a hilos, en el segundo parámetro recibe la estructura del hilo de tipo puntero llamada `thread`, que se eliminará de dicha cola. Sin embargo, aún no hay rastro todavía del error descrito por la herramienta, para ello se analiza el código interno de la función.

```
mysys/mf_keycache.c:1738: error: NULL_DEREFERENCE
  pointer `thread->next` last assigned on line 1738 could be
null and is dereferenced by call to `unlink_from_queue()` at
line 1738, column 9.
  1736.                                ("thread %ld", (ulong)
thread->id));
  1737.           keycache_pthread_cond_signal(&thread-
>suspend);
  1738. >         unlink_from_queue(&keycache-
>waiting_for_hash_link, thread);
  1739.         }
  1740.       }
```

Figura 3.10: Salida de Infer de la detección de una indirección nula.

```
...  ...
1062 static void unlink_from_queue(...,
1063                              struct st_my_thread_var *thread)
1064 {
1065   KEYCACHE_DBUG_PRINT("unlink_from_queue", ("thread %ld", (ulong) thread->id));
1066   DBUG_ASSERT(thread->next && thread->prev);
1067
1068   if (thread->next == thread)
...    ...
1073   else
1074   {
1075     /* Remove current element from list */
1076     thread->next->prev= thread->prev;
1077     *thread->prev= thread->next;
...      ...
```

Se observa en la línea 1,066 del código anterior una validación con el fin de asegurar que el contenido de `thread->next` por alguna circunstancia tome un valor nulo mediante el macro `DBUG_ASSERT`. En caso de resultar falsa la aserción, se registra en un archivo de bitácora y finaliza la ejecución del programa. La primer indirección se realiza en la línea 1,076, sin embargo, como se ha descrito anteriormente la validación previa evita la indirección nula. Como conclusión, resulta en un falso positivo probablemente debido a que el código de la macro evita una correcta interpretación por parte de la herramienta de análisis o debido a que la función `assert(...)`, empleada dentro de la macro, no se tenga contemplada aún en la herramienta como una manera de evitar la indirección nula.

Tabla 3.3: Resumen general de detecciones correctas y falsos positivos.

Carpeta	Tipo de bug	Detecciones correctas	Falsos positivos
client	Indirecciones nulas	9	1
	Almacenamientos muertos	8	9
	Valor sin inicializar	0	9
	Fuga de recursos	0	1
mysys	Almacenamientos muertos	7	5
	Indirecciones nulas	1	2
	Valor sin inicializar	0	11
	Fuga de recursos	0	1
	Fuga de memoria	0	1
Total	Almacenamientos muertos	13	14
	Indirecciones nulas	10	3
	Valor sin inicializar	0	20
	Fuga de recursos	0	2
	Fuga de memoria	0	1
		25	**40**

Reporte general

En la Tabla 3.3 se muestra el resumen de los fallos identificados, separados por las detecciones correctas y los falsos positivos. Cabe mencionar que en el código seleccionado para su análisis incluye tanto código de C y C++, en donde se descubrieron algunos tipos de errores específicos. Sin embargo, los desarrolladores de Infer continúan trabajando en mejorar la herramienta, por ejemplo, en agregar un error muy solicitado en el caso del lenguaje C, que sucede cuando se intenta acceder a un arreglo pero el índice se encuentra fuera del rango definido. Además, en mejorar el soporte para la detección de errores en programas concurrentes. Éste último se encuentra todavía en fase temprana de desarrollo y se continua produciendo trabajos de investigación al respecto. En el siguiente capítulo, se discutirán los resultados y se darán las conclusiones finales.

Capítulo 4

Conclusiones y trabajo futuro

En primer lugar, en el presente trabajo de tesis se desarrollan los antecedentes para la verificación formal mediante la especificación de programas. La lógica de primer orden resulta de relevancia para la creación de fórmulas más descriptivas, y con base en ello se establece la lógica de Hoare, capaz de especificar las propiedades que aseguran el correcto funcionamiento de los programas. Sin embargo, se encuentran limitaciones al describir estructuras de datos que hacen uso de punteros.

En vista de ello, las lógicas de separación añaden dos nuevos operadores especiales a la lógica de Hoare, que resultan capaces de precisar los estados de la memoria y al mismo tiempo discernir entre dos o más porciones de manera independiente. Además, dichos operadores permiten describir fragmentos aislados de código gracias a la regla del marco, la cual se centra únicamente en aquellas porciones de la memoria que mutan. Como beneficio, es posible analizar grandes cantidades de código cargando únicamente lo necesario y dando la posibilidad de realizar el proceso de manera concurrente. Por otro lado, el proceso de bi-abducción resulta importante para automatizar la generación de las especificaciones del código.

El principal resultado presentado en esta tesis es la verificación formal de un gestor de bases de datos empleando las lógicas de separación. En particular, se verificaron librerías de MariaDB asociadas con el uso dinámico de memoria y aplicaciones del cliente. Este resultado se publicó en [25] y [24]. El gestor de bases de datos fue seleccionado por ser ampliamente utilizado en la industria y estar relacionado con actividades que pueden resultar críticas, dado que un error grave puede traer consecuencias en cuestión de recursos o impactar la seguridad del sistema y de las personas que hacen uso de este.

De acuerdo con el reporte de Infer correspondiente a las dos carpetas analizadas del gestor de base de datos, se identificaron un total de 65 errores que, al ser inspeccionados, 25 de estos fueron detecciones correctas, representando una cantidad considerable de falsos positivos. La debilidad de la herramienta se establece en los valores sin inicializar, puesto que 20 de ellos fueron reportados equivocadamente. Este tipo de error tiene una característica en común dentro del análisis, ya que explícitamente a través de una instrucción de código se precisa que de manera intencionada no deben inicializarse algunas variables, y a pesar de ello se clasifican como errores reales. Sin embargo, los falsos positivos identificados proporcionaran una vía de mejora para el programa de verificación.

En contraste, la fortaleza de la herramienta se concentra en determinar indirecciones nulas, debido a que se interactúa directamente con la memoria y las lógicas de separación permiten describirlas de manera precisa. En consecuencia el número de falsos positivos fue mínimo. Dentro de estas detecciones correctas sólo una se encuentra del lado del servidor, y es poco probable que ocurra, aún así, resulta importante atenderse para evitar interrupciones en el servicio. Por otro lado, las que se concentran en el cliente pueden afectar al usuario que trata de interactuar con la base de datos, Sin embargo, al no relacionarse directamente con la disponibilidad del servicio puede considerarse de prioridad baja para su resolución.

Aunado a esto, tanto en la carpeta que interactúa con los procesos, archivos y estructuras de datos dinámicas, como en la carpeta del lado del cliente, se detectaron almacenamientos muertos que si bien no afectan el funcionamiento directamente, los recursos del sistema se desaprovechan. La mitad de los almacenamientos muertos fueron detectados como falsos positivos y su contraparte como detecciones correctas, por lo que es necesario determinar que está sucediendo a nivel de implementación con el fin de corregirlo.

Con las detecciones correctas, se demostró que actualmente hay sistemas que poseen errores a pesar de contar con una constante actualización, por lo que se busca incentivar que los desarrolladores adopten el uso de herramientas basadas en las lógicas de separación y puedan automatizar la fase de pruebas en los ciclos de desarrollo del software en determinados aspectos. De esta forma, proporcionar una mayor seguridad para los usuarios finales, obtener un ahorro de tiempo durante la fase de pruebas e incrementar la confianza en el uso de nuevos sistemas. Las detecciones acertadas fueron enviadas para su análisis al equipo de desarrollo de MariaDB, en consecuencia actualmente planean integrar Infer a su proceso de desarrollo para la detección temparana de errores.

Hay un gran trabajo de investigación futuro por delante en esta área, por lo que se perseguirá mejorar el algoritmo interno a la implementación, con el fin de evitar en la medida de lo posible la producción de falsos positivos que entorpezcan la fase de análisis posterior al reporte, al cual, se tiene que dedicar tiempo adicional para comprobar cada uno de ellos. Por otro lado, también se buscará incluir el soporte para distintos lenguajes de programación que actualmente no han sido desarrollados, por ejemplo, Python o JavaScript, ya sea colaborando con el proyecto Infer o en busca de innovar en una nueva herramienta.

Referencias

[1] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, y Rupak Majumdar. The software model checker Blast. *International Journal on Software Tools for Technology Transfer*, 9(5):505–525, 2007.

[2] Aaron R. Bradley y Zohar Manna. *The calculus of computation: Decision procedures with applications to verification*. Springer Berlin Heidelberg, 2007.

[3] Rodney M. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine intelligence*, 7(23-50):3, 1972.

[4] Cristiano Calcagno y Dino Distefano. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, y Rajeev Joshi, editores, *NASA Formal Methods - Third International Symposium*, volumen 6617, páginas 459–465, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[5] Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, y Dulma Rodriguez. Moving Fast with Software Verification. In Klaus Havelund, Gerard J. Holzmann, y Rajeev Joshi, editores, *NASA Formal Methods 7th International Symposium*, volumen 9058, páginas 3–11. Springer International Publishing, 2015.

[6] Cristiano Calcagno, Dino Distefano, Peter O'Hearn, y Hongseok Yang. Compositional Shape Analysis by Means of Bi-Abduction. *Journal of the Association for Computing Machinery*, 58(6):1–66, 2011.

[7] Harvey M. Deitel y Paul J. Deitel. *C# for Programmers*. Pearson Education, 2005.

[8] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Association for Computing Machinery*, 18(8):453–457, August 1975.

[9] Dino Distefano y Ivana Filipović. Memory Leaks Detection in Java by Bi-abductive Inference. In David S. Rosenblum y Gabriele Taentzer, editores, *Fundamental Approaches to Software Engineering*, volumen 6013, páginas 278–292, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[10] Cormac Flanagan, Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, y Raymie Stata. Programming language design and implementation 2002: Extended static checking for java. *Association for Computing Machinery*, 48(4S):22–33, 2013.

[11] Robert W. Floyd. Assigning meanings to programs. In *Proceedings of Symposia in Applied Mathematics*, volumen 19, páginas 19–32, 1967.

[12] Didier Galmiche y Daniel Méry. Tableaux and resource graphs for separation logic. *Journal of Logic and Computation*, 20(1):189–231, 2010.

[13] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, y Doug Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006.

[14] Charles Antony R. Hoare. An Axiomatic Basis for Computer Programming. *Association for Computing Machinery*, 12(10):576–580, 1969.

[15] Zhé Hóu, Ranald Clouston, Rajeev Goré, and Alwen Tiu. Modular labelled sequent calculi for abstract separation logics. *Association for Computing Machinery Transactions on Computational Logic*, 19(2), 2018.

[16] Samin Ishtiaq y Peter O'Hearn. BI as an Assertion Language for Mutable Data Structures. In Chris Hankin and Dave Schmidt, editores, *Proceedings of the 28th Association for Computing Machinery Symposium on Principles of Programming Languages*, volumen 36, páginas 14–26. Association for Computing Machinery, 2001.

[17] Omar A. Jasim y Sandor M. Veres. Formal Verification of Quadcopter Flight Envelop Using Theorem Prover. In *2018 Institute of Electrical and Electronics Engineers Conference on Control Technology and Applications*, páginas 1502–1507. Institute of Electrical and Electronics Engineers, 2018.

[18] Maryam Kamali, Louise Abigail Dennis, Owen McAree, Michael Fisher, y Sandor M. Veres. Formal verification of autonomous vehicle platooning. *Science of Computer Programming*, 148:88–106, 2017.

[19] Brian W. Kernighan and Dennis M. Ritchie. *El lenguaje de programación C*. Pearson Educación, 1991.

[20] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, y Simon Winwood. SeL4: Formal Verification of an OS Kernel. In *Proceedings of the Association for Computing Machinery SIGOPS 22nd Symposium on Operating Systems Principles*, páginas 207–220. Association for Computing Machinery, 2009.

[21] Bil Lewis y Daniel J. Berg. *Multithreaded Programming with Java Technology*. Sun Microsystems Press Java series. Prentice Hall, 2000.

[22] Francesco Logozzo. Cibai: An Abstract Interpretation-Based Static Analyzer for Modular Analysis and Verification of Java Classes. In Byron Cook and Andreas Podelski, editores, *Verification, Model Checking, and Abstract Interpretation*, volumen 4349, páginas 283–298. Springer Berlin Heidelberg, 2007.

[23] Michael McCool, Arch D. Robison, y James Reinders. *Structured Parallel Programming*. Elsevier, 2012.

[24] Diego Medina-Martínez, Everardo Bárcenas, Guillermo Molero-Castillo, Alejandro Velázquez-Mena, y Rocío Aldeco-Pérez. Database Management System Verification with Separation Logics. *Programming and Computer Software (En prensa)*, 2021.

[25] Diego Medina-Martínez, Everardo Bárcenas, Guillermo Molero-Castillo, Alejandro Velázquez-Mena, y Rocío Aldeco-Pérez. Formal verification of a database management system. In *2020 8th International Conference in Software Engineering Research and Innovation*, páginas 102–109, 2020.

[26] Peter W. O'Hearn y David J. Pym. The Logic of Bunched Implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999.

[27] Peter W. O'Hearn, John C. Reynolds, y Hongseok Yang. Local reasoning about programs that alter data structures. In Laurent Fribourg, editor, *Computer Science Logic, 15th International Workshop*, volumen 2142 of *Lecture Notes in Computer Science*, páginas 1–19. Springer Verlag, 2001.

[28] Grant O. Passmore y Denis Ignatovich. Formal Verification of Financial Algorithms. In Leonardo de Moura, editor, *26th International Conference on Automated Deduction*, volumen 10395 of *Lecture Notes in Computer Science*, páginas 26–41. Springer International Publishing, 2017.

[29] André Platzer y Jan-David Quesel. European Train Control System: A Case Study in Formal Verification. In Karin K. Breitman and Ana Cavalcanti, editores, *Formal Methods and Software Engineering*, volumen 5885, páginas 246–265. Springer Berlin Heidelberg, 2009.

[30] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th Annual Institute of Electrical and Electronics Engineers Symposium on Logic in Computer Science*, páginas 55–74, 2002.

[31] John Charles Reynolds. Intuitionistic reasoning about shared mutable data structure. *Millennial perspectives in computer science*, 2(1):303–321, 2000.

[32] Jhon M. Rushby y Friedrich W. von Henke. Formal verification of algorithms for critical systems. *Institute of Electrical and Electronics Engineers Transactions on Software Engineering*, 19(1):13–23, 1993.

[33] Manuel Sousa, José Creissac Campos, Miriam Alves, y Michael D. Harrison. Formal verification of safety-critical user interfaces: a space system case study. In *Formal Verification and Modeling in Human Machine Systems: Papers from the Association for the Advancement of Artificial Intelligence Spring Symposium*, 2014.

[34] Junbeom Yoo, Eunkyoung Jee, y Sungdeok Cha. Formal Modeling and Verification of Safety-Critical Software. *Institute of Electrical and Electronics Engineers Software*, 26(3):42–49, 2009.