Preservación de Obligaciones de Prueba en Entornos Híbridos de Verificación

Tesina de grado presentada
por

Julián Samborski-Forlese
S-3025/2

al
Departamento de Ciencias de la Computación
en cumplimiento parcial de los requerimientos
para la obtención del grado de

Licenciado en Ciencias de la Computación

Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario
Av. Pellegrini 250, Rosario, República Argentina

Mayo 2009

Supervisores

Gilles Barthe César Kunz

Instituto Madrileño de Estudios Avanzados en Tecnologías de Software

Campus de Montegancedo s/n, Boadilla del Monte, España

A mi hermano Mauricio,
porque el mundo es de los osados.

Resumen

La producción de software confiable y eficiente requiere, al menos en parte, la automatización de su construcción. Para alcanzar este objetivo es indispensable estudiar a los programas y sus ejecuciones como objetos matemáticos.

Los entornos de verificación de programas se basan cada vez más en métodos híbridos que combinan análisis estático con generación de condiciones de verificación. Mientras que dichos entornos de verificación operan sobre programas fuente, a menudo es preferible obtener garantías sobre código ejecutable.

El objetivo de este trabajo es mostrar que, para métodos híbridos de verificación basados en análisis estático y generación de condiciones de verificación, la compilación de programas preserva obligaciones de prueba y, en consecuencia, es posible transferir evidencia de ciertas propiedades de programas fuente a programas compilados. Este resultado se sustenta en la preservación de soluciones de análisis por compilación. Esto se logra apoyándose en un análisis de bytecode que realiza una ejecución simbólica de las expresiones del stack con el fin de evitar la pérdida de precisión que conlleva realizar análisis estático en código compilado.

Se muestra, además, que los métodos híbridos de verificación son correctos, probando que todo programa demostrable por dichos métodos es también demostrable (a un costo mayor) por métodos estándares.

Finalmente, se presenta un caso de estudio en el que se analizan algunas de las principales ventajas que brindan los métodos híbridos en comparación con los métodos clásicos.

Agradecimientos

A mis viejos, papi y mami, Osvaldo y Analía, dos seres excepcionales que a diario me ayudan a transitar este hermoso camino que es la vida. A ellos, por enseñarme a vivir como una persona libre, responsable de sus elecciones. A ellos, porque se lo merecen más que nadie. A mi hermano, por ser un ejemplo de vida y por ser esa bella persona que tanto orgullo me inspira. La distancia que nos separa nunca ha sido un obstáculo para tanto amor.

A mis supervisores, Gilles y César. A Gilles por brindarme tan increíble oportunidad, abriéndome las puertas de un mundo totalmente nuevo para mí. A César por su amistad y por guiarme en este nuevo camino, haciendo que me sienta cómodo y seguro. A la gente de IMDEA, por todo lo que significó para mí vivir una experiencia tan enriquecedora en compañía de ellos. En especial a César Sanchez, por tan sabios consejos cuando más los necesité.

A mis profesores, que marcaron mi rumbo y me hicieron amar lo que hago. Un infinito gracias a Rafael Verdes y Hugo Navone, por mostrarme desde el comienzo que aprender cosas difíciles también puede ser divertido. A Gabriela, Guido, Dante y Fidel, simplemente gracias, pocos tenemos el inmenso placer de disfrutar de sus enseñanzas. A mis amigos de facultad y en especial a mis compañeros de ruta Juan Manuel Crespo, Damián Soriano y Germán Delbianco, por todo lo que aprendí de ellos en estos hermosos años de carrera, en lo académico y en lo personal. A Germán Vuletich y Agustín Ríos, porque con ellos viví momentos inolvidables y su amistad dejó una huella imborrable en mi corazón.

A mis Grancanarios queridos, Sara, Fachu, Jessy y Flor, por tantos momentos felices. A mi grupo de amigos en Madrid, por hacerme sentir en casa, cuando casa quedaba tan lejos. En especial a Javier Valdazo, por ser ese amigo que siempre está dispuesto a escucharte.

A mis grandes amigos, Dante, Mauro y Gustavo, por estar en las buenas y también en las malas. Por enseñarme que el verdadero valor de nuestra amistad no reside en nuestras similitudes, sino en lo enriquecedoras que resultan nuestras diferencias.

Y a todos aquellos que, de una manera u otra, son parte de mi vida, a todos, gracias.

Contenidos

Resumen iv

Contenidos vi

Lista de Figuras viii

1. Introducción **1**

1.1. Proof Carrying Code 3

1.2. Análisis Estático por Interpretación Abstracta 7

1.3. PCC y Métodos Híbridos 8

2. Preservación de Soluciones **11**

2.1. Definición de los Lenguajes 11

2.1.1. Lenguaje Fuente 12

2.1.2. Lenguaje Objeto 14

2.2. Análisis Estático por Interpretación Abstracta 18

2.2.1. Expresiones Simbólicas 20

2.2.2. Dominio Abstracto 22

2.2.3. Análisis de Código Fuente 22

2.2.4. Análisis de Bytecode 23

2.2.5. Preservación de Soluciones 25

3. Preservación de Obligaciones de Prueba **33**

3.1. VCgen para Programas Fuente 34

3.2. VCgen para Programas Bytecode 36

3.3. Preservación de Obligaciones de Prueba 37

3.4. De VCgen Híbrido a VCgen Estándar 40

4. QuickSort: Un Caso de Estudio **43**

4.1. Programa de Ejemplo 43

4.2. Análisis de Código 46

4.3. Generación de Condiciones de Verificación 49

5. Trabajos Relacionados y Conclusiones **54**

Bibliografía **57**

Lista de Figuras

1.1. Esquema típico de la arquitectura PCC 5

1.2. Verificación Híbrida con Preservación de Obligaciones de Preuba 9

2.1. Semántica denotacional de expresiones 14

2.2. Semántica operacional de alto nivel 15

2.3. Semántica operacional de alto nivel (reglas para excepciones) 16

2.4. Semántica operacional de bajo nivel 19

2.5. Semántica operacional de bajo nivel (reglas para excepciones) 20

2.6. Compilador 21

2.7. Requisitos sobre la función γ 23

2.8. Sistema de restricciones para el análisis de código fuente 24

2.9. Sistema de restricciones para el análisis de bytecode 26

3.1. Definición de la función WP 36

3.2. VCgen para Programas Bytecode 38

3.3. VCgen No-Híbrido para Bytecode 40

4.1. Análisis de Alto Nivel. Resultado para *main* 47

4.2. Análisis de Alto Nivel. Resultado para *getPivot* 47

4.3. Análisis de Alto Nivel. Resultado para *swap* 48

4.4. Análisis de Alto Nivel. Resultado para *split* 48

4.5. Análisis de Alto Nivel. Resultado para *quicksort* 49

1

Introducción

La interacción entre sistemas de software por medio de intefaces activas, o código móvil, es un poderoso método que permite instalar y ejecutar código dinámicamente. De este modo, un servidor puede proveer medios flexibles de acceso a sus recursos y servicios internos. En este escenario, la instalación y activación de código malicioso en un entorno receptor no sólo es factible, sino que resulta sencillo, constituyendo un serio riesgo para éste y pudiendo comprometer la integridad de los datos o, incluso, destruir el propio sistema.

Un enfoque muy usado actualmente para evitar este tipo de situaciones no deseadas es la utilización de firmas digitales. Sin embargo, las firmas digitales no proveen ninguna garantía sobre el comportamiento de componentes de software. Una firma digital puede aumentar la confianza de que un componente de software ha sido examinado y/o testeado por una entidad ajena al productor de código, el operador o el fabricante. Pero muchos programas testeados y firmados digitalmente contienen errores.

La producción de software confiable y eficiente requiere, al menos en parte, la automatización de su construcción. Para alcanzar este objetivo, es indispensable estudiar a los programas y sus ejecuciones como objetos matemáticos.

George Necula propuso una técnica para proporcionar confianza a los consumidores de código acerca de la correctitud del programa, librándolos de confiar en productores de código (que son potencialmente dañinos), redes (que pueden estar controladas por un atacante) y compiladores (que pueden contener errores). Esta técnica, llamada Proof Carrying Code (PCC) [25, 26], consiste en exigir que el código móvil sea enviado junto con evidencia verificable de su adhesión a una política apropiada que puede involucrar requisitos sobre protección,

seguridad y funcionalidad.

Las técnicas de verificación de programas son ampliamente utilizadas en el razonamiento sobre la correctitud de aplicaciones y desempeñan un rol importante en campos como código móvil y sistemas embebidos, en donde son requeridas fuertes garantías. Sin embargo, la verificación de programas, y en particular la verificación deductiva de programas, es tradicionalmente aplicada a código fuente, cuando en realidad, las garantías son a menudo requeridas sobre código ejecutable. Esta discrepancia se ve particularmente acentuada en el contexto del código móvil, en donde los consumidores de código no confían en los productores de código y pueden no tener acceso a las fuentes originales. Por tanto, es de especial interés desarrollar métodos para transferir evidencias de verificación de código fuente a los receptores. En [4], los autores se enfocan en la transferencia de evidencias de programas fuente a compilados en el contexto de métodos de verificación basados en generadores de condiciones de verificación, que son generalmente utilizados en Proof-Carrying Code y entornos de verificación de programas. En dicho trabajo, mostraron que la compilación no-optimizante preserva obligaciones de prueba, y por lo tanto que los certificados de programas fuente pueden ser reutilizados directamente para validar programas compilados. Sin embargo, las herramientas de verificación vanguardistas no utilizan solamente generación de condiciones de verificación; sino que se basan en métodos híbridos, que combinan análisis estáticos y generación de condiciones de verificación.

El objetivo de este trabajo es extender preservación de obligaciones de pruebas a métodos de verificación híbridos. Concretamente, consideramos un lenguaje imperativo pequeño, con funciones y arrays dinámicos y nos enfocamos en un método híbrido basado en un análisis numérico genérico [8, 22] —que puede ser instanciado a varios dominios numéricos, incluyendo polyhedra— y un análisis de nulidad simple. Primero definimos un método de verificación híbrido en el cual los programas son sometidos a análisis estático y luego a generación de condiciones de verificación. El VCgen se beneficia de la información proporcionada por el análisis de dos maneras muy útiles: por un lado, las condiciones de verificación originadas de aristas espurias en el grafo de control de flujo son descartadas —mas precisamente, el VCgen ignora los casos de acceso a variables nulas y accesos fuera-de-rango siempre que el análisis garantice que las variables están definidas y los accesos a arreglos dentro de los límites—, lo cuál conduce a menos condiciones de verificación, que a su vez son más pequeñas. Por otro lado, el VCgen incluye los resultados del análisis como supuestos para ayudar al usuario a probar las condiciones de verificación. Esto es particularmente útil para análisis relacionales, dado que éstos pueden proveer parte de los invariantes requeridos para probar correctitud de

programas.

Luego demostramos preservación de obligaciones de pruebas utilizando técnicas de Barthe et al. [4]. La demostración se sustenta en saber que las soluciones de los análisis son preservadas por compilación. A pesar de que analizar programas compilados resulta menos preciso que analizar programas fuente, como fue establecido por Logozzo y Fähndrich [21], en este trabajo conseguimos preservación de soluciones mediante la definición de un análisis a nivel de bytecode que realiza una ejecución simbólica de stacks [8, 32, 33].

Finalmente, relacionamos verificación híbrida con verificación estándar. Mostramos que los programas que son demostrablemente correctos usando nuestro método híbrido, son también demostrablemente correctos usando generación de condiciones de verificación estándar; con este fin, definimos un compilador que transforma una especificación híbrida (que combina afirmaciones lógicas y resultados de análisis) en una especificación lógica dando una interpretación lógica de los resultados del análisis.

La estructura del trabajo está organizada como se describe a continuación. En lo que resta de la introducción, se presentan los conceptos Proof-Carrying Code y Análisis Estático por Interpretación Abstracta en mayor profundidad. También se da una breve explicación de la relación entre PCC y los Métodos Híbridos. En el capítulo 2 se introducen los lenguajes utilizados junto con sus respectivas semánticas. Además se definen los análisis de código fuente y bytecode y se demuestra la preservación de soluciones. En el capítulo 3 se muestra la preservación de obligaciones de pruebas y se prueba la correctitud del método híbrido de verificación utilizado. En el capítulo 4 se presenta QuickSort como un caso de estudio particular. El trabajo finaliza con conclusiones y un repaso de los trabajos relacionados en el capítulo 5.

Esta tesina presenta un extensión al paper arbitrado **Preservation of proof obligations for hybrid verification methods** [6], publicado en la 6$^{\text{th}}$ IEEE International Conference on Software Engineering and Formal Methods (SEFM 2008), en co-autoría con Gilles Barthe, César Kunz y David Pichardie.

1.1. Proof Carrying Code

Proof-Carrying Code (PCC) es una innovadora técnica desarrollada por George Necula y Peter Lee. En un caso típico de PCC, un consumidor de código establece un conjunto de

reglas de seguridad que garantizan el comportamiento seguro de los programas, y el productor de código crea una prueba de seguridad formal que demuestra, para el código no confiable, adhesión a las reglas de seguridad. Luego, el receptor puede usar un validador de pruebas sencillo y eficiente, para chequear, con certeza, que la prueba es válida y por lo tanto el código no confiable puede ejecutarse con seguridad.

PCC tiene mucha utilidad en sistemas en los cuales su base computacional confiable (TCB) es dinámica, ya sea por código móvil o por corrección de errores o actualizaciones periódicas. Algunos ejemplos incluyen, pero no se limitan, a sistemas operativos extensibles, navegadores de Internet capaces de descargar código, nodos de red activos y controladores embebidos tolerantes a fallas. El PCC estándar se construye sobre varios componentes: una lógica, un generadores de condiciones de verificación, una representación formal de pruebas y un verificador de pruebas. La Figura 1.1 muestra el esquema típico de la arquitectura PCC. Luego de la compilación, el productor de código genera condiciones de verificación mediante la aplicación del VCgen al código móvil. Dichas condiciones son luego probadas por el demostrador automático, produciendo un certificado. Como en el caso del productor, el consumidor aplica el VCgen al código móvil. El certificado debe contener información suficiente como para permitir al verificador descartar las condiciones generadas. Sólo después de un chequeo exitoso el código es ejecutado. Describimos brevemente cada uno de los componentes de PCC:

Una lógica formal para especificación y verificación de políticas. El lenguaje de especificación es utilizado para expresar requerimientos sobre el componente entrante y la lógica es utilizada para verificar que el componente cumple con los requisitos impuestos. PCC estándar adopta la lógica de predicados de primer orden como formalismo tanto para especificar como para verificar la correctitud de componentes y se enfoca en las propiedades de seguridad. Por lo tanto, los requisitos son expresados como pre y post condiciones estableciendo, respectivamente, condiciones que deben ser satisfechas por los estados anterior y posterior a la invocación de un procedimiento o función dada.

Un Generador de Condiciones de Verificación (VCgen). El VCgen produce, para cada componente y cada política de protección, un conjunto de obligaciones de pruebas cuya demostrabilidad será suficiente para asegurar que el componente respeta la política. PCC adopta un VCgen basado en técnicas de verificación de programación, tales como Hoare-Floyd logics y *weakest precondition calculi*, y requiere que los componentes contengan anotaciones extra, como, por ejemplo, invariantes de bucles que permitan que la

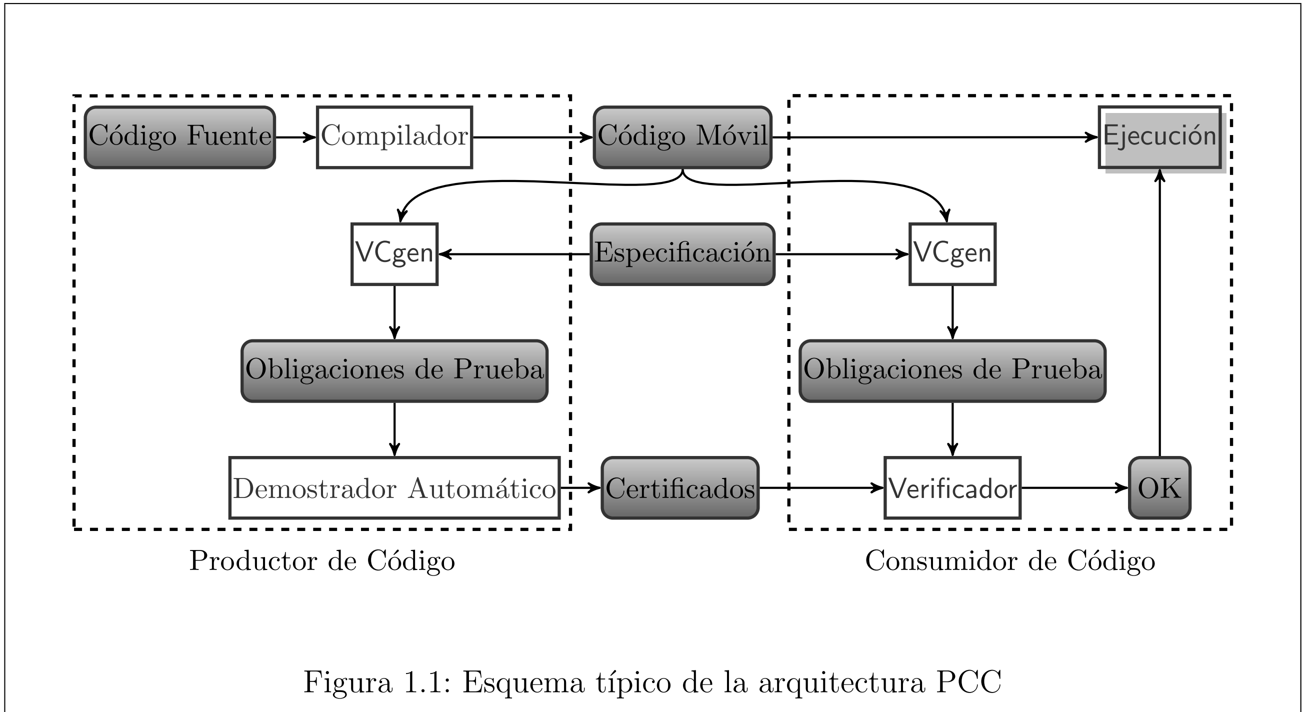

Figura 1.1: Esquema típico de la arquitectura PCC

generación de condiciones de verificación sea posible.

Una representación formal de pruebas (Certificados). Los certificados proveen una representación formal de pruebas y son utilizados para transmitir al consumidor de código evidencia sencilla de verificar que el código que recibe es seguro. Comúnmente, los certificados son términos del lambda cálculo, según lo sugerido por el isomorfismo de Curry-Howard, y habitualmente utilizados en los asistentes de prueba modernos como Coq y LF.

Un demostrador de pruebas que valide certificados contra especificaciones. El objetivo de un demostrador de pruebas es verificar que el certificado realmente demuestra las obligaciones de pruebas generadas por el VCgen. La verificación de pruebas puede ser reducida a verificación de tipos en virtud del isomorfismo de Curry-Howard, de modo que el verificador de pruebas sólo tiene que verificar que el certificado tiene el tipo correcto. Uno de los aspectos atractivos de este enfoque es que el verificador, que forma parte de la TCB, es particularmente simple.

Proof Carrying Code se beneficia de una serie de características distintivas que lo convierten en una base muy adecuada para arquitecturas de seguridad para sistemas globales. Primero, PCC está basado en verificacién en lugar de confianza. De hecho, PCC se enfoca

en el comportamiento de los componentes descargados en vez de enfocarse en el origen de dichos componentes. En particular, PCC no requiere la existencia de una infraestructura global de confianza (aunque puede ser utilizado en combinación con infraestructuras de confianza basadas en criptografía). Segundo, Proof Carrying Code es transparente a los usuarios finales. Mientras que PCC se basa en las ideas de la verificación de programas, la cual en general requiere pruebas interactivas, la arquitecura PCC no requiere que los consumidores construyan pruebas. Sólo exige que éstos las chequeen, lo cual es completamente automático. Tercero, el principio de Proof Carrying Code es general; la única restricción en las políticas de seguridad es que éstas deben ser expresables en la lógica formal, la cual a menudo es muy expresiva. Por otra parte, los principios básicos de PCC son aplicables a cualquier lógica [7]. Cuarto, la arquitectura PCC es flexible y configurable, ya que puede ser utilizada con diferentes políticas. En particular, el VCgen y el verificador de pruebas son independientes de la política, mientras que el compilador certificante puede, en principio, ser adaptado a diferentes propiedades de seguridad. Finalmente, la tecnología Proof Carrying Code no sacrifica performance en pos de la seguridad, ya que al basarse en verificación estática en tiempo de compilación, no incurre en los altos costos inherentes a las técnicas dinámicas de verificación basadas en monitores.

La Base Computacional Confiable (TCB) de un entorno PCC es el conjunto de componentes en los que se debe confiar con el fin de garantizar la correcitud de un programa certificado. Cualquier error en los componentes fuera de la TCB no afecta la correctitud de la ejecución del código. En la Figura 1.1, es necesario y suficiente depender de la correctitud del VCgen, del Verificador de Pruebas y del entorno de ejecución.

Una cuestión fundamental que debe abordar cualquier implementación de PCC basado en lógica es la generación de certificados. Si los certificados basados en lógica serán utilizados para verificar propiedades básicas de seguridad de código, y se espera que una gran clase de programas posean un certificado, entonces es indispensable que los certificados sean generados automáticamente. Los Compiladores Certificantes extienden a los compiladores tradicionales con un mecanismo para generar, de forma automática, certificados para propiedades de seguridad suficientemente simples, utilizando la información disponible sobre el programa durante su compilación, para producir un certificado que puede ser comprobado por el verificador de pruebas. Nótese que el compilador certificante no forma parte de la TCB; sin embargo, es un componente esencial de PCC, ya que reduce el trabajo de verificación por parte del productor.

Un ejemplo precoz de compilador certificante es el compilador Touchstone [26], el cual fue

concebido para explorar la viabilidad de la investigación pionera sobre PCC. Este compilador genera una prueba formal de protección de memoria y seguridad basada en tipos para el programa resultante en assembler DEC Alpha. El compilador Touchstone inserta automáticamente los invariantes de bucles en el programa resultante y las pruebas de correctitud generadas. La contraparte de este enfoque es que las propiedades garantizadas son deliberadamente restringidas a propiedades de seguridad de tipos.

1.2. Análisis Estático por Interpretación Abstracta

Análisis Estático es el proceso de extraer información semántica acerca de un programa en tiempo de compilación. Un ejemplo clásico es el problema de las variables vivas [17]; una variable x está viva en una sentencia s si y sólo si en alguna ejecución x es accedida luego de ejecutar s y sin haber sido redefinida. Otros problemas clásicos son reaching definitions, available expressions y very busy expressions. Existen dos frameworks principales para realizar Análisis Estático: Análisis de Flujo de Datos e Interpretación Abstracta.

La verificación formal de programas consiste en demostrar que la semántica de un programa satisface su especificación, entendiéndose por semántica, la caracterización matemática del posible comportamiento del mismo, y por especificación, la descripción matemática de éste.

El análisis estático de programas por Interpretación Abstracta [11] es el descubrimiento de propiedades de programas de manera estática y automática. Formalmente, la Interpretación Abstracta se basa en la idea de aproximación discreta, la cual consiste en reemplazar el razonamiento sobre una semántica concreta exacta por un cómputo en una semántica abstracta aproximada. La incertidumbre resultante de esta aproximación no concierne a la inexactitud de las propiedades inferidas automáticamente sino por la existencia de propiedades no computables para las cuales la respuesta "no lo sé" es posible. Un programa denota cómputos en algún universo de objetos. La Interpretación Abstracta de programas consiste en usar dicha denotación para describir cómputos en otro universo de objetos abstractos, de modo que el resultado de la ejecución abstracta provea cierta información sobre los cómputos reales. Un ejemplo intuitivo es la regla de los signos (obtenido de [30]). El texto $-1515 * 17$ puede ser intrepretado como cómputos en el universo abstracto $\{(+), (-), (\pm)\}$, en donde la semántica

de los operadores aritméticos es definida por la regla de los signos. La ejecución abstracta

$$-1515 * 17 \Rightarrow -(+) * (+) \Rightarrow (-) * (+) \Rightarrow (-)$$

prueba que $-1515 * 17$ es un número negativo. La Interpretación Abstracta se enfoca en una estructura particular subyacente del universo de cómputos usual (los signos, en nuestro ejemplo) y provee un resumen de algunas facetas de las ejecucióntes reales de un programa. En general, este extracto es sencillo de obtener, pero, como se dijo anteriormente, resulta impreciso. Por ejemplo

$$-1515 + 17 \Rightarrow -(+) + (+) \Rightarrow (-) + (+) \Rightarrow (\pm)$$

donde el resultado abstracto $(\pm)$ nos indica que no es posible computar una respuesta precisa sobre el signo de la operación si sólo se tiene en cuenta los signos de los operandos. A pesar de sus resultados fundamentalmente incompletos, la Interpretación Abstracta permite al programador —o al compilador— responder questiones que no requieren un conocimiento total de las ejecuciones del programa o que toleran una respuesta imprecisa (e.g. pruebas de correctitud parcial de programas ignorando los problemas de terminación, type checking, optimizaciones de programas que no son llevadas a cabo ante la falta de certeza acerca de su viabilidad, etc).

La Interpretación Abstracta ha permitido el desarrollo de sofisticados análisis de programas, que son, al mismo tiempo, demostrablemente correctos y prácticos. Las aproximaciones semánticas producidas por tales análisis han sido aplicadas tanto a optimización de programas como a verificación.

En especial, la aplicación de esta técnica a la verificación de programas y debugging ha recibido considerable atención. Un caso particularmente interesante ha sido propuesto e implementado en el sistema Astrée [10, 12] y aplicado con grán éxito en la industria aeroespacial.

1.3. PCC y Métodos Híbridos

Existen dos variantes principales de PCC, que se basan, respectivamente, en sistemas de tipos y verificación de programas. Describimos brevemente las dificultades de la generación y chequeo de certificados para cada caso.

En el enfoque basado en tipos, los certificados son generados en forma automática por el analizador y adjuntados al código; del lado del consumidor, el certificado es verificado efi-

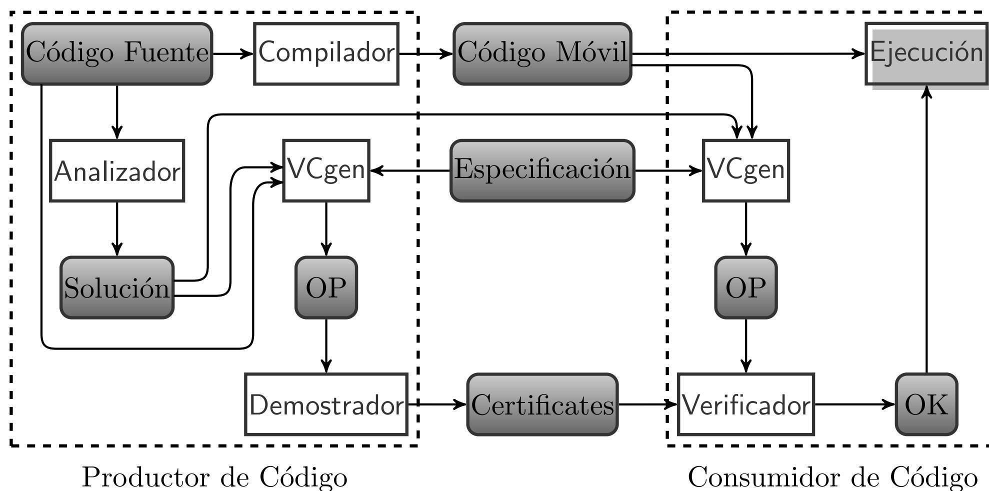

Figura 1.2: Verificación Híbrida con Preservación de Obligaciones de Preuba

cientemente por un chequeador que está estrechamente emparentado con el analizador (en el espíritu de verificación ligera de bytecode). En el enfoque basado en lógica, los certificados son generados automáticamente por compiladores certificantes, siempre que las políticas sean lo suficientemente simples. Sin embargo, los certificados deben ser construidos de manera interactiva una vez que las políticas se vuelven más complejas; la preservación de obligaciones de pruebas permite mantener la construcción de certificados a nivel de código fuente. El certificado (que puede incluir anotaciones) es empaquetado junto con el programa. En el lado del consumidor, el protocolo procede en tres pasos: primero, las anotaciones son contrastadas con la política; segundo, se utiliza generación de condiciones de verificación para generar las obligaciones de prueba; tercero, el chequeo del certificado es realizado utilizando un chequeador de pruebas, que verifica si el certificado demuestra las obligaciones de prueba.

Los métodos híbridos tienen por objetivo proporcionar una estrecha integración entre los sistemas de tipos y los métodos lógicos. Existen dos enfoques a la verificación híbrida de programas. En el enfoque explícito, el usuario provee anotaciones de seguridad que son utilizadas por el generador de condiciones de verificación y chequeadas por un chequeador de anotaciones. En contraste, el enfoque implícito aboga por que las anotaciones sean inferidas por el analizador estático y luego utilizadas en la generación de condiciones de verificación. Ambos enfoques son utilizados (a veces en conjunto) en la verificación deductiva de programas, así como en análisis

basados en tipos.

En la Figura 1.2 se muestra el esquema del método híbrido que desarrollamos en este trabajo. La verificación de programas procede del siguiente modo: primero, el programa anotado es sometido al análisis estático y una solución es computada (notar que el análisis no obtiene ventaja de las anotaciones). Luego, el generador de condiciones de verificación usa la solución para computar un conjunto reducido de obligaciones de pruebas que deberán ser descartadas interactivamente. Del lado del consumidor, el programa llega empaquetado con la solución del análisis (o como en la verificación ligera de bytecode, una solución parcial que contiene información suficiente para recomputar la solución eficientemente), las anotaciones y el certificado. El chequeo es llevado acabo en cuatro pasos; primero, se chequea la correctitud del análisis y los restantes tres pasos son los mismos que en PCC basado en lógica.

2

Preservación de Soluciones

"In the development of the understanding of complex phenomena, the most powerful tool available to the human intellect is abstraction. Abstraction arises from the recognition of similarities between certain objects, situations, or processes in the real world and the decision to concentrate on these similarities and to ignore, for the time being, their differences" [**C.A.R. Hoare**]

En este capítulo nos enfocamos en la preservación de soluciones de los análisis de alto y bajo nivel. Con este fin, definimos los respectivos lenguajes, el compilador para traducir programas fuente en pogramas de bytecode y los análisis para ambos lenguajes. Luego definimos lo que se entiende por solución de un análisis y finalmente mostramos cómo las soluciones pueden ser preservadas, es decir, dada una solución para el análisis de alto nivel, podemos computar, directamente de ésta, una solución para el análisis de bajo nivel sin perder precisión.

2.1. Definición de los Lenguajes

En esta sección introducimos el lenguaje fuente (un lenguaje imperativo con funciones y arreglos dinámicos de enteros), el lenguaje objeto (un lenguaje con saltos basado en stacks) y el compilador.

En lo que resta del trabajo, asumimos la existencia de dos conjuntos disjuntos, V_s y V_a, de variables escalares y de arreglo respectivamente, y denotamos por V a la suma disjunta $V_s + V_a$. Además, suponemos dados dos conjuntos V_s^{old} y V_a^{old} en correspondencia 1-1 con V_s y V_a, los cuales son utilizados para guardar los valores iniciales de las variables al comenzar la

ejecución de una función. También consideramos a res una variable especial, que es utilizada para representar el valor devuelto por una función. Finalmente, asumimos la existencia de un conjunto $\mathbf{Lab} \subset \mathbb{N}$ de etiquetas.

2.1.1. Lenguaje Fuente

El lenguaje de alto nivel que definimos aquí, es un lenguaje imperativo simple con arreglos dinámicos y funciones. Tiene una sintaxis similar a C que presenta ciertas restricciones, impuestas intencionalmente, en pos de facilitar la compresión de los conceptos y resultados que se pretenden mostrar en este trabajo.

Sintaxis

Los programas son definidos como listas de funciones, y las funciones están compuestas de un nombre, una lista de parámetros formales y una sentencia decorada con etiquetas a fin de expresar resultados de análisis. En lo que resta, para cualquier conjunto A, escribimos $A^\star$ para denotar el dominio de las listas con elementos en A. La sintaxis del lenguaje de alto nivel se presenta a continuación:

$$
\begin{array}{rcrcl}
P & \in & \mathit{Prog}^s & = & \mathit{Func}^{s\star} \\
 & & \mathit{Func}^s & = & \mathit{Sig} \times \mathit{Stmt} \\
 & & \mathit{Sig} & = & \mathit{FuncId} \times \mathit{FArgs} \\
 & & \mathit{FArgs} & = & V^\star \\
\mathtt{op} & \in & \mathit{Oper} & & \\
\bowtie & \in & \mathit{Comp} & & \\
e & \in & \mathit{Exp} & & \\
s & \in & \mathit{Stmt} & &
\end{array}
\qquad
\begin{array}{rcl}
\mathtt{op} & ::= & + \mid - \mid \times \mid \div \\
\bowtie & ::= & = \mid \neq \mid < \mid \leq \mid > \mid \geq \\
e & ::= & e \;\mathtt{op}\; e \mid n \mid x \mid a[e] \\
s & ::= & [\mathtt{x}{:=}e]^k \mid [\mathtt{a}[e]{:=}e]^k \mid [f(e^\star)]^k \\
 & \mid & [\mathtt{x}{:=}f(e^\star)]^k \mid [\mathtt{a}{:=}\mathtt{newarray}(e)]^k \\
 & \mid & \mathtt{if}\ [e \bowtie e]^k\ \mathtt{then}\ s\ \mathtt{else}\ s \\
 & \mid & \mathtt{while}\ [e \bowtie e]^k\ \mathtt{do}\ s \mid s; s \\
 & \mid & [\mathtt{Skip}]^k \mid [\mathtt{return}\ e]^k
\end{array}
$$

donde x, a, n y k oscilan, respectivamente, sobre V_s, V_a, $\mathbb{Z}$ y $\mathbf{Lab}$, $\mathtt{op}$ se mueve sobre operaciones aritméticas (binarias), $\bowtie$ sobre comparaciones aritméticas, y f sobre nombres de funciones. Suponemos que las etiquetas no se repiten. Consideramos programas válidos a los programas que tienen a $((main, []), s)$ como función, donde s es la sentencia que representa el cuerpo de dicha función, y todas sus funciones terminan con una sentencia $[\mathtt{return}\ e]^k$.

Semántica

La semántica de los programas fuente es formalizada por una relación de transición entre estados. Los estados pueden ser intermedios, en cuyo caso consisten de una sentencia y una memoria, o finales, en cuyo caso se componen de una memoria y, posiblemente, un identificador para indicar una terminación anormal. La memoria es modelada por tripletas de asignaciones, respectivamente de variables escalares a valores, de variables de arreglo a direcciones y de direcciones a arreglos de valores. Las variables escalares son locales a cada función y las variables de arreglo son siempre globales, es decir, son visibles desde cualquier función. La tercera componente de una tripleta representa el *Heap* global. Definimos los dominios de la semántica del lenguaje fuente como sigue:

$$
\begin{array}{rcl}
Location & & \\
Error & ::= & \mathbf{aob} \mid \mathbf{null} \\
Array & = & \mathbb{Z} \\
Heap & = & Location \to Array_{\perp} \\
VMem & = & V_s \to \mathbb{Z} \\
AMem & = & V_a \to Location \\
Mem & = & VMem \times AMem \times Heap \\
State_I^s & = & Stmt \times VMem \times AMem \times Heap \\
State_F^s & = & Mem + (VMem \times AMem \times Heap \times (\mathbb{Z} + Error) \\
State^s & = & State_I^s + State_F^s
\end{array}
$$

Dados un mapeo $\rho_{\mathsf{a}} \in AMem$, un heap $h \in Heap$ y una variable $a \in V_a$, definimos el tamaño $|a|$ de a como:

$$
|a| = \begin{cases} |h(\rho_{\mathsf{a}}\ a)| & \text{si } \ h(\rho_{\mathsf{a}}\ a) \neq \perp \\ \perp & \text{caso contrario} \end{cases}
$$

En la Figura 2.1 se presenta la semántica denotacional de expresiones. El significado de una expresión en un contexto particular es estándar. La evaluación de expresiones puede producir excepciones. La semántica operacional de los programas de alto nivel también es estándar y está definida por la relación $\leadsto \sqsubseteq State_I^s \times State^s$. Las reglas que describen esta relación se muestran en la Figura 2.2, donde utilizamos la notación $[f \mid s \mapsto r]$ para referirnos a la función que es idéntica a f en todo su dominio, excepto en r, que devuelve s, para cualesquiera conjuntos R y S y cualquier función $f : R \to S$, y usamos $b[n \mapsto m]$ para referirnos al arreglo que es idéntico a b en todas sus posiciones, excepto en la posición n, en la cual su valor es m,

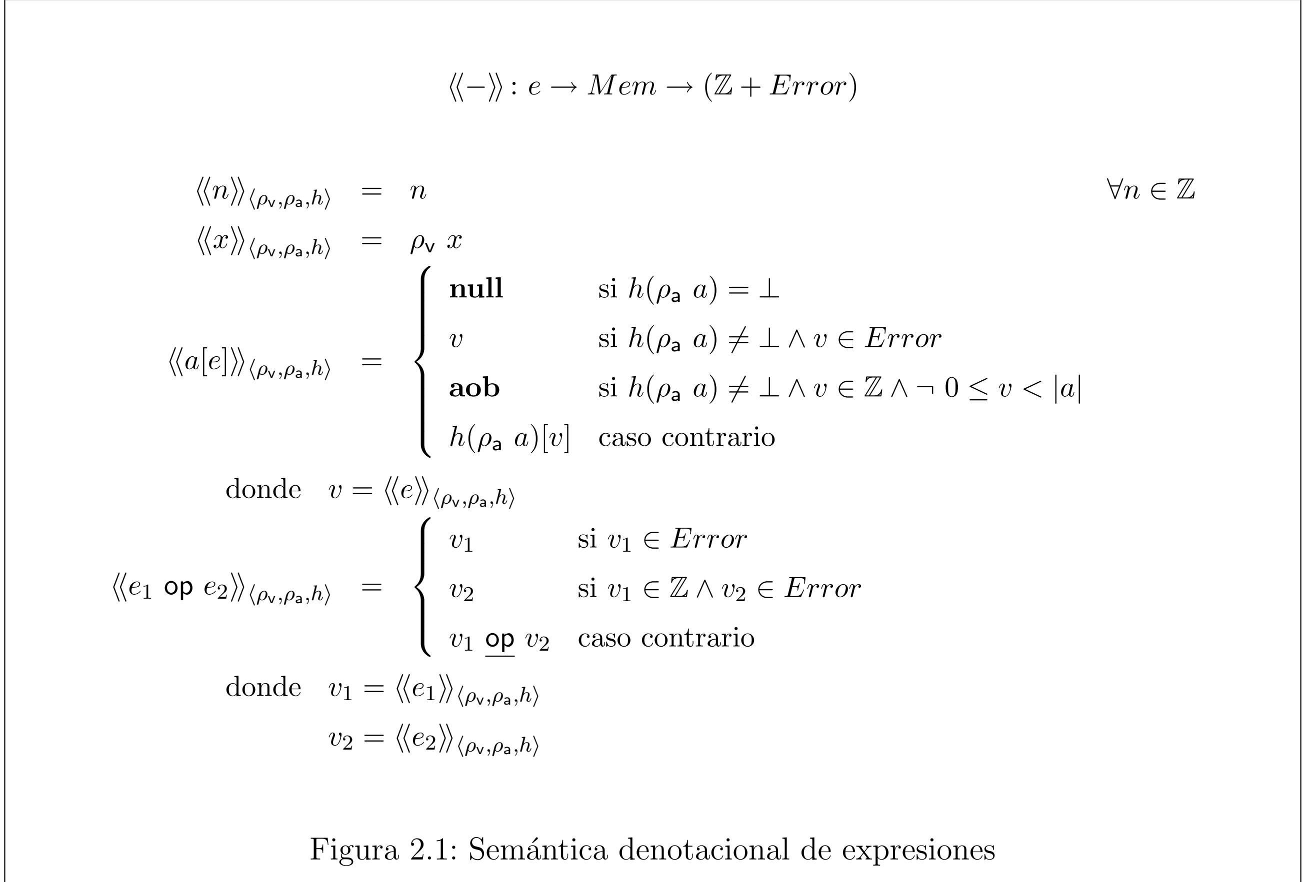

Figura 2.1: Semántica denotacional de expresiones

para cualquier arreglo $b \in Array$ y cualesquiera enteros $n, m \in \mathbb{Z}$. Matemáticamente hablando,

$$\forall x \in \mathsf{dom}(f) \;\bullet\; [f \mid r \mapsto s](i) = \begin{cases} s & \text{si } x = r \\ f(x) & \text{caso contrario} \end{cases}$$

and

$$\forall i \in \mathbb{Z} \mid 0 \leq i \leq |b| \;\bullet\; b[n \mapsto m][i] = \begin{cases} m & \text{si } i = n \\ b[i] & \text{caso contrario} \end{cases}$$

El comportamiento excepcional de los programas está especificado por la relación de transición $\rightsquigarrow \sqsubseteq State_I^s \times State_F^s$, cuyas reglas se muestran en la Figura 2.3. Sólo consideramos como excepciones los accesos nulos (**null**) y los accesos a arreglos que están fuera de rango (**aob**).

2.1.2. Lenguaje Objeto

El lenguaje objeto presentado aquí es similar al bytecode para la Java Virtual Machine (JVM). Es un lenguaje basado en stacks, con funciones y arreglos dinámicos. La terminación

$$\frac{((main, []), c) \in P}{\langle P \rangle \rightsquigarrow_P \langle c, \emptyset, \emptyset, \emptyset \rangle} \qquad \frac{}{\langle [\texttt{Skip}]^k, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \rangle \rightsquigarrow \langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \rangle}$$

$$\frac{\langle\!\langle e \rangle\!\rangle_{\langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \rangle} = n}{\langle [\mathtt{x}{:=}e]^k, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \rangle \rightsquigarrow \langle [\rho_{\mathsf{v}} \mid x \mapsto n], \rho_{\mathsf{a}}, h \rangle}$$

$$\frac{\langle\!\langle e_1 \rangle\!\rangle_{\langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \rangle} = n \qquad \langle\!\langle e_2 \rangle\!\rangle_{\langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \rangle} = m \qquad 0 \leq n < |a| \qquad h(\rho_{\mathsf{a}}\ a) = b}{\langle [\mathsf{a}[e_1]{:=}e_2]^k, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \rangle \rightsquigarrow \langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, [h \mid ref \mapsto b[n \mapsto m]] \rangle}$$

$$\frac{\begin{array}{c}((f, [x_1, \ldots, x_n]), s) \in P \\ \mathsf{dom}(\rho''_{\mathsf{v}}) = \{x_1, \ldots, x_n\} \qquad \forall x_i \bullet \rho''_{\mathsf{v}}\ x_i = \langle\!\langle e_i \rangle\!\rangle_{\langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \rangle} \qquad \langle s, \rho''_{\mathsf{v}}, \rho_{\mathsf{a}}, h \rangle \rightsquigarrow^\star \langle \rho'_{\mathsf{v}}, \rho'_{\mathsf{a}}, h', v \rangle\end{array}}{\langle [\mathtt{x}{:=}f(e_1, \ldots, e_n)]^k, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \rangle \rightsquigarrow \langle [\rho_{\mathsf{v}} \mid x \mapsto v], \rho_{\mathsf{a}}, h' \rangle}$$

$$\frac{\begin{array}{c}((f, [x_1, \ldots, x_n]), s) \in P \\ \mathsf{dom}(\rho''_{\mathsf{v}}) = \{x_1, \ldots, x_n\} \qquad \forall x_i \bullet \rho''_{\mathsf{v}}\ x_i = \langle\!\langle e_i \rangle\!\rangle_{\langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \rangle} \qquad \langle s, \rho''_{\mathsf{v}}, \rho_{\mathsf{a}}, h \rangle \rightsquigarrow^\star \langle \rho'_{\mathsf{v}}, \rho'_{\mathsf{a}}, h', v \rangle\end{array}}{\langle [f(e_1, \ldots, e_n)]^k, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \rangle \rightsquigarrow \langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h' \rangle}$$

$$\frac{\langle\!\langle e \rangle\!\rangle_{\langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \rangle} = n \qquad h(ref) = \bot \qquad n > 0}{\langle [\mathsf{a}{:=}\texttt{newarray}(e)]^k, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \rangle \rightsquigarrow \langle \rho_{\mathsf{v}}, [\rho_{\mathsf{a}} \mid a \mapsto ref], [h \mid ref \mapsto 0^n] \rangle}$$

$$\frac{\langle\!\langle e_1 \rangle\!\rangle_{\langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \rangle} \bowtie \langle\!\langle e_2 \rangle\!\rangle_{\langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \rangle}}{\langle \texttt{if } [e_1 \bowtie e_2]^k \texttt{ then } s_1 \texttt{ else } s_2, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \rangle \rightsquigarrow \langle s_1, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \rangle}$$

$$\frac{\neg\ \langle\!\langle e_1 \rangle\!\rangle_{\langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \rangle} \bowtie \langle\!\langle e_2 \rangle\!\rangle_{\langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \rangle}}{\langle \texttt{if } [e_1 \bowtie e_2]^k \texttt{ then } s_1 \texttt{ else } s_2, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \rangle \rightsquigarrow \langle s_2, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \rangle}$$

$$\frac{\langle\!\langle e_1 \rangle\!\rangle_{\langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \rangle} \bowtie \langle\!\langle e_2 \rangle\!\rangle_{\langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \rangle}}{\langle \texttt{while } [e_1 \bowtie e_2]^k \texttt{ do } s_1, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \rangle \rightsquigarrow \langle s_1; \texttt{while } [e_1 \bowtie e_2]^k \texttt{ do } s1, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \rangle}$$

$$\frac{\neg\ \langle\!\langle e_1 \rangle\!\rangle_{\langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \rangle} \bowtie \langle\!\langle e_2 \rangle\!\rangle_{\langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \rangle}}{\langle \texttt{while } [e_1 \bowtie e_2]^k \texttt{ do } s_1, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \rangle \rightsquigarrow \langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \rangle}$$

$$\frac{\langle s_1, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \rangle \rightsquigarrow^\star \langle \rho'_{\mathsf{v}}, \rho'_{\mathsf{a}}, h' \rangle}{\langle s_1; s_2, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \rangle \rightsquigarrow \langle s_2, \rho'_{\mathsf{v}}, \rho'_{\mathsf{a}}, h' \rangle} \qquad \frac{\langle\!\langle e \rangle\!\rangle_{\langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \rangle} = n}{\langle [\texttt{return } e]^k, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \rangle \rightsquigarrow \langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h, n \rangle}$$

Figura 2.2: Semántica operacional de alto nivel

$$\frac{\langle\!\langle e \rangle\!\rangle_{\langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h\rangle} = ex \qquad ex \in \mathit{Error}}{\langle [\mathtt{x}{:=}e]^k, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h\rangle \rightsquigarrow_{EX} \langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h, ex\rangle} \qquad \frac{a \notin \mathsf{dom}(\rho_{\mathsf{a}}) \vee h(\rho_{\mathsf{a}}\ a) = \bot}{\langle [\mathsf{a}[e_1]{:=}e_2]^k, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h\rangle \rightsquigarrow_{EX} \langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h, \mathbf{null}\rangle}$$

$$\frac{\langle\!\langle e_1 \rangle\!\rangle_{\langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h\rangle} = ex \qquad ex \in \mathit{Error}}{\langle [\mathsf{a}[e_1]{:=}e_2]^k, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h\rangle \rightsquigarrow_{EX} \langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h, ex\rangle} \qquad \frac{\langle\!\langle e_1 \rangle\!\rangle_{\langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h\rangle} = n \qquad \neg\ 0 \le n < |a|}{\langle [\mathsf{a}[e_1]{:=}e_2]^k, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h\rangle \rightsquigarrow_{EX} \langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h, \mathbf{aob}\rangle}$$

$$\frac{\langle\!\langle e_1 \rangle\!\rangle_{\langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h\rangle} \in \mathbb{Z} \qquad \langle\!\langle e_2 \rangle\!\rangle_{\langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h\rangle} = ex \qquad ex \in \mathit{Error}}{\langle [\mathsf{a}[e_1]{:=}e_2]^k, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h\rangle \rightsquigarrow_{EX} \langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h, ex\rangle}$$

$$\frac{((f, [x_1, \ldots, x_n]), s) \in P \qquad \exists e_i \bullet \langle\!\langle e_i \rangle\!\rangle_{\langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h\rangle} = ex \qquad ex \in \mathit{Error}}{\langle [f(e_1, \ldots, e_n)]^k, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h\rangle \rightsquigarrow_{EX} \langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h, ex\rangle}$$

$$\frac{\begin{array}{c}((f, [x_1, \ldots, x_n]), s) \in P \\ \mathsf{dom}(\rho''_{\mathsf{v}}) = \{x_1, \ldots, x_n\} \qquad \forall x_i \bullet \rho''_{\mathsf{v}}\ x_i = \langle\!\langle e_i \rangle\!\rangle_{\langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h\rangle} \qquad \langle s, \rho''_{\mathsf{v}}, \rho_{\mathsf{a}}, h\rangle \rightsquigarrow_{EX} \langle \rho'_{\mathsf{v}}, \rho'_{\mathsf{a}}, h', ex\rangle\end{array}}{\langle [f(e_1, \ldots, e_n)]^k, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h\rangle \rightsquigarrow_{EX} \langle \rho'_{\mathsf{v}}, \rho'_{\mathsf{a}}, h', ex\rangle}$$

$$\frac{\langle\!\langle e \rangle\!\rangle_{\langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h\rangle} = ex \qquad ex \in \mathit{Error}}{\langle [\mathsf{a}{:=}\mathtt{newarray}(e)]^k, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h\rangle \rightsquigarrow_{EX} \langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h, ex\rangle}$$

$$\frac{\langle\!\langle e_1 \rangle\!\rangle_{\langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h\rangle} = ex \qquad ex \in \mathit{Error}}{\langle \mathtt{if}\ [e_1 \bowtie e_2]^k\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h\rangle \rightsquigarrow_{EX} \langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h, ex\rangle}$$

$$\frac{\langle\!\langle e_1 \rangle\!\rangle_{\langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h\rangle} \in \mathbb{Z} \qquad \langle\!\langle e_2 \rangle\!\rangle_{\langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h\rangle} = ex \qquad ex \in \mathit{Error}}{\langle \mathtt{if}\ [e_1 \bowtie e_2]^k\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h\rangle \rightsquigarrow_{EX} \langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h, ex\rangle}$$

$$\frac{\langle\!\langle e_1 \rangle\!\rangle_{\langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h\rangle} = ex \qquad ex \in \mathit{Error}}{\langle \mathtt{while}\ [e_1 \bowtie e_2]^k\ \mathtt{do}\ s_1, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h\rangle \rightsquigarrow_{EX} \langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h, ex\rangle}$$

$$\frac{\langle\!\langle e_1 \rangle\!\rangle_{\langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h\rangle} \in \mathbb{Z} \qquad \langle\!\langle e_2 \rangle\!\rangle_{\langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h\rangle} = ex \qquad ex \in \mathit{Error}}{\langle \mathtt{while}\ [e_1 \bowtie e_2]^k\ \mathtt{do}\ s_1, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h\rangle \rightsquigarrow_{EX} \langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h, ex\rangle}$$

$$\frac{\langle s_1, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h\rangle \rightsquigarrow^{\star} \langle s'_1, \rho'_{\mathsf{v}}, \rho'_{\mathsf{a}}, h'\rangle \qquad \langle s'_1, \rho'_{\mathsf{v}}, \rho'_{\mathsf{a}}, h'\rangle \rightsquigarrow_{EX} \langle \rho'_{\mathsf{v}}, \rho'_{\mathsf{a}}, h', ex\rangle}{\langle s_1; s_2, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h\rangle \rightsquigarrow_{EX} \langle \rho'_{\mathsf{v}}, \rho'_{\mathsf{a}}, h', ex\rangle}$$

$$\frac{\langle\!\langle e \rangle\!\rangle_{\langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h\rangle} = ex \qquad ex \in \mathit{Error}}{\langle [\mathtt{return}\ e]^k, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h\rangle \rightsquigarrow_{EX} \langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h, ex\rangle}$$

Figura 2.3: Semántica operacional de alto nivel (reglas para excepciones)

abrupta de los programas puede producirse por un acceso nulo o por un acceso fuera de los límites de un arreglo. En primer lugar presentamos la sintaxis del lenguaje y luego definimos la semántica.

Sintaxis

De un modo similar a como lo hicimos en la formalización del lenguaje de alto nivel, establecemos que un programa de bajo nivel queda definido por una lista de funciones. Una función se compone de un nombre, una lista de parámetros formales y una lista de instrucciones. Las instrucciones pueden manipular la memoria —que guarda los valores de las variables y el contenido de los arreglos— y el stack de operandos, así como también realizar saltos condicionales o incondicionales o llamar a una función.

$$
\begin{array}{rcrcl}
\dot{p} & \in & Prog^b & = & Func^{b*} \\
 & & Func^b & = & Sig \times Code \\
c & \in & Code & = & Instr^\star \\
ins & \in & Instr & & \\
ins & ::= & \multicolumn{3}{l}{\texttt{nop} \mid \texttt{pop} \mid \texttt{prim op}} \\
 & \mid & \multicolumn{3}{l}{\texttt{push}\ v \mid \texttt{load}\ x \mid \texttt{store}\ x} \\
 & \mid & \multicolumn{3}{l}{\texttt{aload}\ a \mid \texttt{astore}\ a \mid \texttt{newarray}\ a} \\
 & \mid & \multicolumn{3}{l}{\texttt{cjmp} \bowtie l \mid \texttt{jmp}\ l} \\
 & \mid & \multicolumn{3}{l}{\texttt{call}\ f \mid \texttt{return}}
\end{array}
$$

donde x oscila sobre V_s, a sobre V_a y l sobre **Lab**.

Sólo consideramos válidos aquellos programas que tienen a $((main, []), c)$ como función y todas las funciones terminan en `return`.

La semántica de los programas bytecode se formaliza usando una relación de transición entre estados. Los estados pueden ser tanto intermedios como finales; los estados intermedios están compuestos de un marco de activación (stack frame) y un heap. Cada frame consiste de una función, un contador de programa, un stack de operandos local al frame —que guarda el resultado de los cómputos intermedios— y una memoria. Los dominios de la semántica del lenguaje objeto se definen como sigue, donde implícitamente asumimos que el contador de

programa está dentro de los límites del programa.

$$\begin{array}{rcl} Stack & = & \mathbb{Z}^\star \\ Frame & = & Func^b \times \mathbb{N} \times VMem \times Stack \\ State^b_I & = & Frame \times AMem \times Heap \\ State^b_F & = & VMem \times AMem \times Heap \times (\mathbb{Z} + Error) \\ State^b & = & State^b_I + State^b_F \end{array}$$

Las reglas de la semántica operacional que describen el conjunto de ejecuciones normales de los programas se dan en la Figura 2.4, donde para cada función $f = ((id, args), c)$, escribimos $f[i]$ para simbolizar $c[i]$. El comportamiento de un programa es indefinido si la etiqueta de un salto, sea condicional o no, no pertenece al dominio de la función, es decir, $f[i]$ no está definido. Las instrucciones que manipulan arreglos, tales como `aload` a y `astore` a, pueden causar la finalización abrupta de un programa si el arreglo al que se intenta acceder no está en el contexto o si el acceso al arreglo está fuera de rango. Las reglas que definen el comportamiento excepcional de los programas se describen en la Figura 2.5.

Compilador

El compilador está definido en la Figura 2.6; usamos la función $init : \mathbf{Stm} \rightarrow \mathbf{Lab}$ para asociar a cada sentencia su etiqueta inicial. Suponemos *compatibilidad de etiquetas*, es decir que las etiquetas de una sentencia del código fuente es la misma que la del punto del programa correspondiente a su compilación.

2.2. Análisis Estático por Interpretación Abstracta

Es conocido que la compilación conlleva una potencial pérdida de precisión para análisis relacionales. El propósito de esta sección es mostrar que las soluciones de interpretaciones abstractas son preservadas por compilación, siempre que se usen expresiones simbólicas, como se hace en [8, 32, 33], con el fin de mitigar la presencia del stack de operandos y así recobrar la pérdida de precisión provocada por la compilación.

$$\frac{f = ((main, []), c) \in \dot{p}}{\langle \dot{p} \rangle \rightsquigarrow_{\dot{p}} \langle ((f, 0, \emptyset, \emptyset, []), \emptyset \rangle}
\qquad
\frac{n = n_1 \; \underline{\mathsf{op}} \; n_2}{n_1 :: n_2 :: s, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \;\rightsquigarrow_{\mathtt{prim\ op}}\; n :: s, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h}$$

$$\frac{}{n :: s, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \;\rightsquigarrow_{\mathtt{pop}}\; s, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h}
\qquad
\frac{}{s, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \;\rightsquigarrow_{\mathtt{push}\ n}\; n :: s, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h}$$

$$\frac{}{s, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \;\rightsquigarrow_{\mathtt{load}\ x}\; \rho_{\mathsf{v}}\ x :: s, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h}
\qquad
\frac{}{n :: s, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \;\rightsquigarrow_{\mathtt{store}\ x}\; s, [\rho_{\mathsf{v}} \mid x \mapsto n], \rho_{\mathsf{a}}, h}$$

$$\frac{}{s, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \;\rightsquigarrow_{\mathtt{nop}}\; s, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h}
\qquad
\frac{h(\rho_{\mathsf{a}}\ a) = b \qquad b \neq \bot \qquad 0 \leq n < |b|}{n :: s, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \;\rightsquigarrow_{\mathtt{aload}\ a}\; b[n] :: s, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h}$$

$$\frac{h(\rho_{\mathsf{a}}\ a) = b \qquad b \neq \bot \qquad 0 \leq n < |b|}{n :: v :: s, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \;\rightsquigarrow_{\mathtt{astore}\ a}\; s, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, [h \mid ref \mapsto b[n \mapsto v]]}$$

$$\frac{\rho_{\mathsf{a}}\ a = ref \qquad h(ref) = \bot \qquad n > 0}{n :: s, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \;\rightsquigarrow_{\mathtt{newarray}\ a}\; s, \rho_{\mathsf{v}}, [\rho_{\mathsf{a}} \mid a \mapsto ref], [h \mid ref \mapsto 0^n]}$$

$$\frac{f[i] = instr \qquad s, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \;\rightsquigarrow_{instr}\; s', \rho'_{\mathsf{v}}, \rho'_{\mathsf{a}}, h'}{\langle (f, i, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, s) :: sf, h \rangle \;\rightsquigarrow\; \langle (f, i+1, \rho'_{\mathsf{v}}, \rho'_{\mathsf{a}}, s') :: sf, h' \rangle}$$

$$\frac{f[i] = \mathtt{cjmp} \bowtie l \qquad n_1 \;\underline{\bowtie}\; n_2}{\langle (f, i, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, n_1 :: n_2 :: s) :: sf, h \rangle \;\rightsquigarrow\; \langle (f, l, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, s) :: sf, h \rangle}$$

$$\frac{f[i] = \mathtt{cjmp} \bowtie l \qquad \neg\, n_1 \;\underline{\bowtie}\; n_2}{\langle (f, i, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, n_1 :: n_2 :: s) :: sf, h \rangle \;\rightsquigarrow\; \langle (f, i+1, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, s) :: sf, h \rangle}$$

$$\frac{\begin{array}{c} f[i] = \mathtt{call}\ f' \\ f' = ((name, [x_1, \ldots, x_n]), c) \in \dot{p} \qquad (\forall x_i \in V_s \bullet (x_i \mapsto v_i) \in \rho'_{\mathsf{v}}) \qquad (\forall x_i \in V_a \bullet (x_i \mapsto v_i) \in \rho'_{\mathsf{a}}) \end{array}}{\langle (f, i, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, v_1 :: \ldots :: v_n :: s) :: sf, h \rangle \;\rightsquigarrow\; \langle (f', 0, \rho'_{\mathsf{v}}, \rho'_{\mathsf{a}}, []) :: (f, i, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, s) :: sf, h \rangle}$$

$$\frac{f[i] = \mathtt{return}}{\langle (f, i, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, n :: s) :: (f', i', \rho'_{\mathsf{v}}, \rho'_{\mathsf{a}}, s') :: sf, h \rangle \;\rightsquigarrow\; \langle (f', i'+1, \rho'_{\mathsf{v}}, \rho'_{\mathsf{a}}, n :: s') :: sf, h \rangle}$$

Figura 2.4: Semántica operacional de bajo nivel

$$\frac{h(\rho_{\mathsf{a}}\ a) = \bot}{n :: s, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \ \leadsto_{EX_{\mathtt{aload}\ a}} \ \mathbf{null}} \qquad \frac{h(\rho_{\mathsf{a}}\ a) \neq \bot \qquad \neg\ 0 \leq n < |a|}{n :: s, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \ \leadsto_{EX_{\mathtt{aload}\ a}} \ \mathbf{aob}}$$

$$\frac{h(\rho_{\mathsf{a}}\ a) = \bot}{n :: v :: s, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \ \leadsto_{EX_{\mathtt{astore}\ a}} \ \mathbf{null}} \qquad \frac{h(\rho_{\mathsf{a}}\ a) \neq \bot \qquad \neg\ 0 \leq n < |a|}{n :: v :: s, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \ \leadsto_{EX_{\mathtt{astore}\ a}} \ \mathbf{aob}}$$

$$\frac{f[i] = instr \qquad s, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h \ \leadsto_{EX_{instr}} \ ex \qquad ex \in Error}{\langle (f, i, \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, s) :: sf, h\rangle \ \leadsto_{EX} \ \langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h, ex\rangle}$$

Figura 2.5: Semántica operacional de bajo nivel (reglas para excepciones)

2.2.1. Expresiones Simbólicas

Las expresiones y guardas sirven como interfaz con el dominio numérico relacional en el análisis de bytecode.

$$\begin{aligned} Expr \ni e &::= n \mid x \mid x[e] \mid ? \mid ?[e] \mid e \ \mathsf{op}\ e \quad x \in V \\ Guard \ni t &::= e \bowtie e \end{aligned}$$

La expresión ? representa un valor desconocido; además, las expresiones son interpretadas como conjuntos de posibles valores. Formalmente, las semánticas $\llbracket e \rrbracket_{\rho_{\mathsf{v}}}$ y $\llbracket t \rrbracket_{\rho_{\mathsf{v}}}$ de expresiones con respecto a un entorno ρ_{v} quedan definidas por las cláusulas:

$$\begin{aligned} \llbracket n \rrbracket_{\rho_{\mathsf{v}}} &= \{n\} \\ \llbracket x \rrbracket_{\rho_{\mathsf{v}}} &= \rho_{\mathsf{v}}\ x \\ \llbracket ? \rrbracket_{\rho_{\mathsf{v}}} &= \mathbb{Z} \\ \llbracket ?[e] \rrbracket_{\rho_{\mathsf{v}}} &= \mathbb{Z} \\ \llbracket x[e] \rrbracket_{\rho_{\mathsf{v}}} &= \mathbb{Z} \\ \llbracket e_1\ \mathbf{op}\ e_2 \rrbracket_{\rho_{\mathsf{v}}} &= \{n_1\ \mathbf{op}\ n_2 \mid n1 \in \llbracket e_1 \rrbracket_{\rho_{\mathsf{v}}},\ n_2 \in \llbracket e_2 \rrbracket_{\rho_{\mathsf{v}}}\} \\ \llbracket e_1 \bowtie e_2 \rrbracket_{\rho_{\mathsf{v}}} &\iff \exists\ n_1 \in \llbracket e_1 \rrbracket_{\rho_{\mathsf{v}}},\ n_2 \in \llbracket e_2 \rrbracket_{\rho_{\mathsf{v}}} \bullet n_1 \bowtie n_2 \end{aligned}$$

Obsérvese que la expresión ? no se requiere para analizar programas de bajo nivel obtenidos por compilación de programas fuente, dado que el stack se encuentra vacío luego de la creación de un arreglo, la asignación a una variable o de una operación `astore` a. Sin embargo, ésta

$$\mathrm{J}n\mathrm{K}_\mathrm{e} = \texttt{push}\ n$$

$$\mathrm{J}x\mathrm{K}_\mathrm{e} = \texttt{load}\ x$$

$$\mathrm{J}a[e]\mathrm{K}_\mathrm{e} = \mathrm{J}e\mathrm{K}_\mathrm{e};\ \texttt{aload}\ a$$

$$\mathrm{J}e_1\ \texttt{op}\ e_2\mathrm{K}_\mathrm{e} = \mathrm{J}e_2\mathrm{K}_\mathrm{e};\ \mathrm{J}e_1\mathrm{K}_\mathrm{e};\ \texttt{prim op}$$

$$\mathrm{J}[\texttt{x}{:=}e]^k\mathrm{K} = k : \mathrm{J}e\mathrm{K}_\mathrm{e}; \texttt{store}\ x$$

$$\mathrm{J}[\texttt{a}[e_1]{:=}e_2]^k\mathrm{K} = k : \mathrm{J}e_2\mathrm{K}_\mathrm{e};\ \mathrm{J}e_1\mathrm{K}_\mathrm{e};\ \texttt{astore}\ a$$

$$\mathrm{J}[\texttt{a}{:=}\texttt{newarray}(e)]^k\mathrm{K} = k : \mathrm{J}e\mathrm{K}_\mathrm{e}; \texttt{newarray}\ a$$

$$\mathrm{J}[\texttt{x}{:=}f(e_1,\dots,e_n)]^k\mathrm{K} = k : \mathrm{J}e_n\mathrm{K}_\mathrm{e};\dots\mathrm{J}e_1\mathrm{K}_\mathrm{e};\ \texttt{call}\ f;\ \texttt{store}\ x$$

$$\mathrm{J}[f(e_1,\dots,e_n)]^k\mathrm{K} = k : \mathrm{J}e_n\mathrm{K}_\mathrm{e};\dots\mathrm{J}e_1\mathrm{K}_\mathrm{e};\ \texttt{call}\ f;\ \texttt{pop}$$

$$\mathrm{J}s1; s2\mathrm{K} = \mathrm{J}s_1\mathrm{K};\ \mathrm{J}s_2\mathrm{K}$$

$$\mathrm{J}[\texttt{return}\ e]^k\mathrm{K} = k : \mathrm{J}e\mathrm{K}_\mathrm{e};\ \texttt{return}$$

$$\mathrm{J}[\texttt{Skip}]^k\mathrm{K} = k : \texttt{nop}$$

$$\mathrm{J}\texttt{if}\ [e_1 \bowtie e_2]^k\ \texttt{then}\ s_1\ \texttt{else}\ s_2\mathrm{K} =$$

$$k : \mathrm{J}e_2\mathrm{K}_\mathrm{e}; \mathrm{J}e_1\mathrm{K}_\mathrm{e};\ \texttt{cjmp} \bowtie k_1; k_2 : \mathrm{J}s_2\mathrm{K}; \texttt{jmp}\ l; k_1 : \mathrm{J}s_1\mathrm{K}$$

$$\text{donde}\quad k_1 = init(s_1) = k_2 + |\mathrm{J}s_2\mathrm{K}| + 1$$

$$k_2 = init(s_2) = k + |\mathrm{J}e_2\mathrm{K}_\mathrm{e}| + |\mathrm{J}e_1\mathrm{K}_\mathrm{e}| + 1$$

$$l = k_1 + |\mathrm{J}s_1\mathrm{K}|$$

$$\mathrm{J}\texttt{while}\ [e_1 \bowtie e_2]^k\ \texttt{do}\ s\mathrm{K} =$$

$$k : \mathrm{J}e_2\mathrm{K}_\mathrm{e}; \mathrm{J}e_1\mathrm{K}_\mathrm{e}; \texttt{cjmp} \bowtie k_1; \texttt{jmp}\ l; k_1 : \mathrm{J}s\mathrm{K}; \texttt{jmp}\ k$$

$$\text{donde}\quad k_1 = k + |\mathrm{J}e_2\mathrm{K}_\mathrm{e}| + |\mathrm{J}e_1\mathrm{K}_\mathrm{e}| + 2$$

$$l = k_1 + |\mathrm{J}s\mathrm{K}| + 1$$

Figura 2.6: Compilador

provee una mayor precisión cuando se trata con programas que no son obtenidos por compilación.

2.2.2. Dominio Abstracto

Para nuestros propósitos, definimos dos tipos de análisis: uno numérico y uno de nulidad.

Siguiendo las ideas de Miné [22], suponemos la existencia de una interfaz de dominio numérico que puede ser concretizada con dominios relacionales abstractos estándar. La misma consiste de un domino $\mathbb{D}_{\mathsf{int}}$ equipado con un orden parcial $\sqsubseteq \subseteq \mathbb{D}_{\mathsf{int}} \times \mathbb{D}_{\mathsf{int}}$, operadores meet y join $\sqcap, \sqcup : \mathbb{D}_{\mathsf{int}} \times \mathbb{D}_{\mathsf{int}} \rightarrow \mathbb{D}_{\mathsf{int}}$, un elemento mínimo $\bot$ y un elemento máximo $\top$. Asumimos dada, además, una función de asignación abstracta $\llbracket x{:=}e \rrbracket^\sharp : \mathbb{D}_{\mathsf{int}} \rightarrow \mathbb{D}_{\mathsf{int}}$.

La interfaz de dominio abstracto para el análisis de nulidad está definida por el retículo completo $(\mathbb{D}_{\mathsf{null}}, \sqsubseteq, \bot, \top, \sqcap, \sqcup)$. Además de los operadores estándar, suponemos la existencia de una función de asignación abstracta $\llbracket x{:=}nnull \rrbracket^\sharp : \mathbb{D}_{\mathsf{null}} \rightarrow \mathbb{D}_{\mathsf{null}}$.

Habiendo descripto ambas interfaces, ahora definimos la interfaz de dominio abstracto común, como la combinación de éstas. Dicha interfaz está definida por el retículo completo $(\mathbb{D}, \sqsubseteq, \bot, \top, \sqcap, \sqcup)$, donde $\mathbb{D} = \mathbb{D}_{\mathsf{int}} \times \mathbb{D}_{\mathsf{null}}$. Esta nueva interfaz también proporciona las funciones de asignación abstracta $\llbracket x{:=}e \rrbracket^\sharp, \llbracket x{:=}nnull \rrbracket^\sharp, \llbracket x[e_1]{:=}e_2 \rrbracket^\sharp : \mathbb{D} \rightarrow \mathbb{D}$ y la función $assume^\sharp$ que mapea guardas a elementos abstractos.

Finalmente, asumimos la existencia de una función monótona de concreción $\gamma : \mathbb{D} \rightarrow \mathcal{P}(VMem \times AMem \times Heap)$ que mapea elementos abstractos a conjuntos de entornos en $VMem \times AMem \times Heap$ y satisface las propiedades detalladas en la Figura 2.7. La función γ utiliza los resultados devueltos por las funciones de concreción $\gamma_{\mathsf{int}} : \mathbb{D}_{\mathsf{int}} \rightarrow \mathcal{P}(VMem)$ y $\gamma_{\mathsf{null}} : \mathbb{D}_{\mathsf{null}} \rightarrow (AMem \times Heap)$, provistas por las interfaces abstractas numérica y de nulidad respectivamente. Definimos el test abstracto $\llbracket t \rrbracket^\sharp : \mathbb{D} \rightarrow \mathbb{D}$ de una guarda $t \in Guard$ como $\llbracket t \rrbracket^\sharp(l^\sharp) = assume^\sharp(t) \sqcap l^\sharp$.

2.2.3. Análisis de Código Fuente

El análisis de alto nivel está especificado por funciones de transferencia abstractas que mapean elementos del domino abstracto en elementos del dominio abstracto.

$$
\begin{array}{rcl}
\gamma_{\mathsf{int}}(d_1 \sqcap d_2) & \supseteq & \gamma_{\mathsf{int}}(d_1) \cap \gamma_{\mathsf{int}}(d_2) \\
\gamma_{\mathsf{int}}(d_1 \sqcup d_2) & \supseteq & \gamma_{\mathsf{int}}(d_1) \cup \gamma_{\mathsf{int}}(d_2) \\
\gamma_{\mathsf{int}}([\![x{:=}e]\!]^\sharp(d)) & \supseteq & \begin{cases} \{[\rho_{\mathsf{v}} \mid x \mapsto v] \mid \rho_{\mathsf{v}} \in \gamma_{\mathsf{int}}(d) \wedge v \in [\![e]\!]_{\rho_{\mathsf{v}}}\} & \text{if } x \in V_s \\ \gamma_{\mathsf{int}}(d) & \text{otherwise} \end{cases} \\
 & & \\
\gamma_{\mathsf{null}}(d_1 \sqcap d_2) & \supseteq & \gamma_{\mathsf{null}}(d_1) \cap \gamma_{\mathsf{null}}(d_2) \\
\gamma_{\mathsf{null}}(d_1 \sqcup d_2) & \supseteq & \gamma_{\mathsf{null}}(d_1) \cup \gamma_{\mathsf{null}}(d_2) \\
\gamma_{\mathsf{null}}([\![x{:=}nnull]\!]^\sharp(d)) & \supseteq & \{\langle[\rho_{\mathsf{a}} \mid x \mapsto ref], [h \mid ref \mapsto b]\rangle \mid \langle\rho_{\mathsf{a}}, h\rangle \in \gamma_{\mathsf{null}}(d) \wedge b \neq \bot\} \\
 & & \\
\gamma(d_1 \sqcap d_2) & \supseteq & \gamma(d_1) \cap \gamma(d_2) \\
\gamma(d_1 \sqcup d_2) & \supseteq & \gamma(d_1) \cup \gamma(d_2) \\
\gamma([\![x{:=}e]\!]^\sharp(d_i, d_n)) & \supseteq & \{\langle\rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h\rangle \mid \rho_{\mathsf{v}} \in \gamma_{\mathsf{int}}([\![x{:=}e]\!]^\sharp(d_i)) \wedge \langle\rho_{\mathsf{a}}, h\rangle \in \gamma_{\mathsf{null}}(d_n)\} \\
\gamma([\![x{:=}nnull]\!]^\sharp(d_i, d_n)) & \supseteq & \{\langle\rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h\rangle \mid \rho_{\mathsf{v}} \in \gamma_{\mathsf{int}}(d_i) \wedge \langle\rho_{\mathsf{a}}, h\rangle \in \gamma_{\mathsf{null}}([\![x{:=}nnull]\!]^\sharp(d_n))\} \\
\gamma([\![x[e_1]{:=}e_2]\!]^\sharp(d)) & \supseteq & \{\langle\rho_{\mathsf{v}}, \rho_{\mathsf{a}}, [h \mid \rho_{\mathsf{a}}\ a \mapsto h(\rho_{\mathsf{a}}\ a)[v_1 \mapsto v_2]]\rangle \mid \rho_{\mathsf{v}} \in \gamma_{\mathsf{int}}(d_i) \wedge \langle\rho_{\mathsf{a}}, h\rangle \in \gamma(d_n) \\
 & & \qquad \wedge\ v_1 \in [\![e_1]\!]_{\rho_{\mathsf{v}}} \wedge v_2 \in [\![e_2]\!]_{\rho_{\mathsf{v}}} \wedge h(\rho_{\mathsf{a}}\ a) \neq \bot\} \\
\{\langle\rho_{\mathsf{v}}, \rho_{\mathsf{a}}, h\rangle \mid [\![t]\!]_{\rho_{\mathsf{v}}}\} & \subseteq & \gamma(assume^\sharp(t))
\end{array}
$$

Figura 2.7: Requisitos sobre la función γ

Definición 2.1 (Dominio Abstracto para Alto Nivel)**.** *Un resultado del análisis para el programa fuente P está descripto por un elemento $(Pre, Post, Loc)$ en el retículo*

$$
\begin{array}{rcl}
State^\sharp & = & FuncId \to \mathbb{D} \\
 & \times & FuncId \to \mathbb{D} \\
 & \times & FuncId \times \mathbf{Lab} \to \mathbb{D}
\end{array}
$$

Definición 2.2 (Solución)**.** *Una tupla $(Pre, Post, Loc)$ para el programa fuente P es una solución del análisis si verifica el sistema de restricciones definido en la Figura 2.8, es decir, se cumple $Pre, Post, Loc \vdash P$.*

2.2.4. Análisis de Bytecode

Al igual que para el análisis de alto nivel, el análisis de bytecode está definido por funciones de transferencia abstractas que mapean estados abstractos en estados abstractos. En este caso,

$$\frac{stm \in \{\texttt{Skip}, \texttt{x}{:=}e, \texttt{x}[e_1]{:=}e_2, \texttt{x}{:=}\texttt{newarray}(n)\} \qquad F_{stm}(Loc(f,i)) \sqsubseteq l^\sharp}{Sol \vdash \{Loc(f,i)\}\ [stm]^i\ \{l^\sharp\}}$$

$$\frac{Sol \vdash \{\mathrm{J}t\mathrm{K}^\sharp(Loc(f,i))\}\ s_1\ \{l_1^\sharp\} \qquad Sol \vdash \{\mathrm{J}\neg t\mathrm{K}^\sharp(Loc(f,i))\}\ s_2\ \{l_2^\sharp\}}{Sol \vdash \{Loc(f,i)\}\ \texttt{if}\ [t]^i\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \{l_1^\sharp \sqcup l_2^\sharp\}}$$

$$\frac{Sol \vdash \{\mathrm{J}t\mathrm{K}^\sharp(Loc(f,i))\}\ s\ \{l^\sharp\} \qquad l^\sharp \sqsubseteq Loc(f,i)}{Sol \vdash \{Loc(f,i)\}\ \texttt{while}\ [t]^i\ \texttt{do}\ s\ \{\mathrm{J}\neg t\mathrm{K}^\sharp(Loc(f,i))\}} \qquad \frac{Sol \vdash \{l^\sharp\}\ s_1\ \{l_1^\sharp\} \qquad Sol \vdash \{l_1^\sharp\}\ s_2\ \{l_2^\sharp\}}{Sol \vdash \{l^\sharp\}\ s_1\ ;\ s_2\ \{l_2^\sharp\}}$$

$$\frac{\begin{array}{c}((f', [x_1, \ldots, x_n]), s) \in P \\ Loc(f,i) = (d_1, d_2) \qquad \big(\mathrm{d}_{i=1}^{n} assume^\sharp(e_i = x_i) \sqcap d_1, d_2\big) \sqsubseteq Pre(f') \qquad Post(f') = (l_1, l_2)\end{array}}{Sol \vdash \{Loc(f,i)\}\ [\texttt{x}{:=}f(e_1, \ldots, e_n)]^i\ \{\mathrm{J}x{:=}res\mathrm{K}^\sharp(d_1, d_2 \sqcap l_2)\}}$$

$$\frac{\begin{array}{c}((f', [x_1, \ldots, x_n]), s) \in P \\ Loc(f,i) = (d_1, d_2) \qquad \big(\mathrm{d}_{i=1}^{n} assume^\sharp(e_i = x_i) \sqcap d_1, d_2\big) \sqsubseteq Pre(f') \qquad Post(f') = (l_1, l_2)\end{array}}{Sol \vdash \{Loc(f,i)\}\ [f(e_1, \ldots, e_n)]^i\ \{(d_1, d_2 \sqcap l_2)\}}$$

$$\frac{\forall((f, args), s) \in P \bullet Sol \vdash \{Pre(f)\}\ s\ \{Post(f)\} \qquad ((main, []), s) \in P \qquad \top \sqsubseteq Pre(main)}{Sol \vdash P}$$

$$\frac{\mathrm{J}res{:=}e\mathrm{K}^\sharp(Loc(f,i)) \sqsubseteq Post(f)}{Sol \vdash \{Loc(f,i)\}\ [\texttt{return}\ e]^i\ \{Post(f)\}}$$

$$\text{donde} \quad F_{stm}(l^\sharp) = \begin{cases} l^\sharp & \text{if } stm = \texttt{Skip} \\ \mathrm{J}x{:=}e\mathrm{K}^\sharp(l^\sharp) & \text{if } stm = \texttt{x}{:=}e \\ \mathrm{J}x{:=}nnull\mathrm{K}^\sharp(l^\sharp) & \text{if } stm = \texttt{x}{:=}\texttt{newarray}(e) \\ \mathrm{J}a[e_1]{:=}e_2\mathrm{K}^\sharp(l^\sharp) & \text{if } stm = \texttt{a}[e_1]{:=}e_2 \end{cases}$$

y Sol refiere a $Pre, Post, Loc$.

Figura 2.8: Sistema de restricciones para el análisis de código fuente

los estados abstractos son pares de la forma $(s^\sharp, l^\sharp)$, donde $l^\sharp$ es un elemento del dominio abstracto y la lista de expresiones simbólicas $s^\sharp$ abstrae el stack de operandos. El dominio simbólico abstracto para stacks es $Expr^\star$. El conjunto de variables consideradas por el análisis de bajo nivel es el mismo que en el análisis de código fuente.

Definición 2.3 (Dominio Abstracto para Bytecode)**.** *El valor abstracto para un programa* $\dot{p}$ *es descripto por un elemento* $(\dot{pre}, \dot{post}, \dot{loc})$ *del retículo*

$$\begin{aligned} \dot{state}^\sharp &= FuncId \to \mathbb{D} \\ &\times\ FuncId \to \mathbb{D} \\ &\times\ FuncId \times \mathbf{Lab} \to (Expr^\star \times \mathbb{D}) \end{aligned}$$

Un resultado de análisis es una solución del análisis si satisface el sistema de restricciones asociado a cada programa. El sistema de restricciones está definido en la Figura 2.9. Para las instrucciones intraprocedurales que no provocan salto, las restricciones están definidas mediante funciones de transferencia parciales en $Expr^{\star} \times \mathbb{D} \rightharpoonup (Expr^{\star} \times \mathbb{D})$, la mayoría de ellas definidas como ejecuciones simbólicas que afectan la representación abstracta del stack de operandos.

Definición 2.4 (Solution)**.** *Una tupla $\dot{pre}, \dot{post}, \dot{loc}$ para el programa de bajo nivel $\dot{p}$ es una solución del análisis si satisface el sistema de restricciones de la Figura 2.9, es decir, el juicio $\dot{pre}, \dot{post}, \dot{loc} \vdash \dot{p}$ es válido.*

2.2.5. Preservación de Soluciones

En esta sección mostramos la preservación de soluciones de análisis. Para ello, primero definimos la compilación de una solución de análisis de alto nivel y luego demostramos que es una solución para el análisis de bytecode. Por claridad y conveniencia de escritura, denotamos por $\dot{f}_{s_1;\cdots;s_n}(s^\sharp, l^\sharp)$ la composición $\dot{f}_{s_n}(\cdots(\dot{f}_{s_1}(s^\sharp, l^\sharp))\cdots)$, donde $s_1;\cdots;s_n$ es una secuencia de instrucciones de bajo nivel. Sea f una función de $\dot{p}$. Usamos $\mathsf{succ}(f, l)$ para referirnos al conjunto de sucesores de la etiqueta l. Por ejemplo, $\mathsf{succ}(f, l) = \emptyset$ y $\mathsf{succ}(f, l) = \{l + 1, l'\}$ para $f[l] = \texttt{return}$ y $f[l] = \texttt{cjmp} \bowtie l'$ respectivamente. El conjunto $\mathsf{pred}(f, l)$ es definido como $\{l' \mid l \in \mathsf{succ}(f, l')\}$. Si la etiqueta l no pertenece al dominio de f, tanto $pred(f, l)$ como $succ(f, l)$ denotan el conjunto vacío.

Observación 2.5. *Para cada programa de bajo nivel $\dot{p}$, podemos extraer, del sistema de restricciones previo, un conjunto de funciones $(\dot{g}_{i,j})_{(i,j)\in\mathbf{Lab}^2}$ tales que $\dot{pre}, \dot{post}, \dot{loc} \vdash \dot{p}$ si y sólo si $\bigsqcup_{k'\in\mathsf{pred}(f,k)} \dot{g}_{k',k}(\dot{loc}(f, k')) \sqsubseteq \dot{loc}(f, k)$ para toda $f \in \dot{p}$ y para todo $k \in \mathsf{dom}(f)$.*

Podemos extender una función parcial $\dot{loc}_{partial} \in \dot{state}^\sharp$ a una función total $\dot{loc}$ en $\mathsf{dom}(\dot{p})$ si hacemos

$$\dot{loc}(f, k) = \begin{cases} \dot{loc}_{partial}(f, k) & si\ (f, k) \in \mathsf{dom}(\dot{loc}_{partial}) \\ \bigsqcup_{k'\in\mathsf{pred}(f,k)} \dot{g}_{k',k}(\dot{loc}(f, k')) & caso\ contrario \end{cases}$$

Esta definición sólo tiene sentido si, al considerar el grafo de control de flujo de cada f en $\dot{p}$ —cuyas aristas son $\{(i, j) \mid i \in \mathsf{dom}(f) \wedge j \in \mathsf{succ}(f, i)\}$—, todo bucle contiene una

$instr$	$\dot{f}_{instr}$	$instr$	$\dot{f}_{instr}$
`prim` op	$(e_1 :: e_2 :: s^\sharp, l^\sharp) \to (\llcorner e_1\ \underline{op}\ e_2 \lrcorner :: s^\sharp, l^\sharp)$	`push` n	$(s^\sharp, l^\sharp) \to (n :: s^\sharp, l^\sharp)$
`store` r	$(e :: s^\sharp, l^\sharp) \to (s^\sharp[?/r], \llbracket r{:=}e \rrbracket^\sharp(l^\sharp))$	`load` r	$(s^\sharp, l^\sharp) \to (\llcorner r \lrcorner :: s^\sharp, l^\sharp)$
`astore` a	$(e_1 :: e_2 :: s^\sharp, l^\sharp) \to (s^\sharp[?/a], \llbracket a[e_1]{:=}e_2 \rrbracket^\sharp(l^\sharp))$	`aload` a	$(e :: s^\sharp, l^\sharp) \to (\llcorner a[e] \lrcorner :: s^\sharp, l^\sharp)$
`nop`	$(s^\sharp, l^\sharp) \to (s^\sharp, l^\sharp)$	`pop`	$(e :: s^\sharp, l^\sharp) \to (s^\sharp, l^\sharp)$
`newarray` a	$(e :: s^\sharp, l^\sharp) \to (s^\sharp[?/a], \llbracket a{:=}nnull \rrbracket^\sharp(l^\sharp))$		

$$\frac{\mathsf{Instr} \notin \{\texttt{jmp}\ i', \texttt{cjmp} \bowtie i', \texttt{call}\ f', \texttt{return}\} \qquad \dot{f}_{instr}(\dot{loc}(f,i)) \sqsubseteq \dot{loc}(i+1)}{\dot{pre}, \dot{post}, \dot{loc} \vdash f[i] : \mathsf{Instr}}$$

$$\frac{\dot{loc}(f,i) = (e_1 :: e_2 :: s^\sharp, l^\sharp) \qquad (s^\sharp, \llbracket \neg(e_1 \bowtie e_2) \rrbracket^\sharp(l^\sharp)) \sqsubseteq \dot{loc}(f,i+1) \qquad (s^\sharp, \llbracket e_1 \bowtie e_2 \rrbracket^\sharp(l^\sharp)) \sqsubseteq \dot{loc}(f,j)}{\dot{pre}, \dot{post}, \dot{loc} \vdash f[i] : \texttt{cjmp} \bowtie j}$$

$$\frac{\begin{array}{c}((f',[x_1,\ldots,x_n]),s) \in \dot{p} \qquad \dot{loc}(f,i) = (e_1 :: \ldots :: e_n :: s^\sharp, (d_1,d_2)) \\ \left(\bigsqcap_{i=1}^{n} assume^\sharp(e_i = x_i) \sqcap d_1, d_2\right) \sqsubseteq \dot{pre}(f') \qquad \dot{post}(f') = (l_1, l_2) \qquad (d_1, d_2 \sqcap l_2) \sqsubseteq \dot{loc}(f,i+1)\end{array}}{\dot{pre}, \dot{post}, \dot{loc} \vdash f[i] : \texttt{call}\ f'}$$

$$\frac{\dot{loc}(f,i) \sqsubseteq \dot{loc}(f,j)}{\dot{pre}, \dot{post}, \dot{loc} \vdash f[i] : \texttt{jmp}\ j} \qquad \frac{\dot{loc}(f,i) = (e :: s^\sharp, l^\sharp) \qquad \llbracket res{:=}e \rrbracket^\sharp(l^\sharp) \sqsubseteq \dot{post}(f)}{\dot{pre}, \dot{post}, \dot{loc} \vdash f[i] : \texttt{return}}$$

$$\frac{((f,args),s) \in \dot{p} \qquad \forall i \in dom(s) \bullet \dot{pre}, \dot{post}, \dot{loc} \vdash f[i] : s[i]}{\dot{pre}, \dot{post}, \dot{loc} \vdash f}$$

$$\frac{((main,[]),s) \in \dot{p} \qquad \top \sqsubseteq \dot{pre}(main) \qquad \forall ((f,args),s) \in \dot{p} \bullet \dot{pre}, \dot{post}, \dot{loc} \vdash f}{\dot{pre}, \dot{post}, \dot{loc} \vdash \dot{p}}$$

Figura 2.9: Sistema de restricciones para el análisis de bytecode

etiqueta en $\mathsf{dom}(\dot{loc}_{partial})$*. Nos referimos a la función* $\dot{loc}$ *como la* conclusión de *la función parcial* $\dot{loc}_{partial}$.

Definición 2.6 (Resultados de Análisis Compilados)**.** *Dado un resultado* $(Pre, Post, Loc)$ *para el programa* P*, un resultado de análisis compilado de* $(Pre, Post, Loc)$ *es igual a* $(\dot{pre}, \dot{post}, \dot{loc})$*, donde* $\dot{pre} = Pre$*,* $\dot{post} = Post$ *y* $\dot{loc}$ *es la conclusión de la función* $\dot{loc}_{partial}$ *definida en cada* $k \in \mathsf{dom}(Loc)$ *por* $\dot{loc}_{partial}(f,k) = ([\,], Loc(f,k))$*.*

Se puede mostrar que esta definición es buena por el hecho de que $(Pre, Post, Loc)$ anota cada bucle en P y cada bucle en el grafo de control de flujo de $\dot{p}$ contiene una etiqueta de un

bucle en P.

Lema 2.7. *Sea* $((f, args), body) \in \dot{p}$ *y sean* $s_1, s_2 \in Instr^\star$ *y* $e \in Exp$ *tales que se verifica* $body = s_1 :: l : \text{J}e\text{K}_\text{e} :: l' : s_2$. *Entonces,* $l\dot{o}c(f, l') = f_{i_1;\ldots;i_k}(s^\sharp, l^\sharp) = (e :: s^\sharp, l^\sharp)$ *donde* $(s^\sharp, l^\sharp) = l\dot{o}c(f, l)$ *y* $[i_1; \ldots; i_k] = \text{J}e\text{K}_\text{e}$.

Demostración. Procedemos con la demostración haciendo inducción estructural sobre la expresión e. Observemos que, por ser $\dot{p}$ un programa obtenido por compilación, $l' \notin \mathsf{dom}(Loc)$ y $pred(l') = \{l' - 1\}$.

Caso $e = n$**:** Sabemos que $\text{J}e\text{K}_\text{e} = \texttt{push}\ n$ y $pred(l') = \{l\}$. Por tanto,

$$l\dot{o}c(f, l) = \dot{f}_{\texttt{push}\ n}(l\dot{o}c(f, l)) = \dot{f}_{\texttt{push}\ n}(s^\sharp, l^\sharp) = (n :: s^\sharp, l^\sharp).$$

Caso $e = x$**:** Tenemos que $\text{J}e\text{K}_\text{e} = \texttt{load}\ x$ y $pred(l') = \{l\}$. Así,

$$l\dot{o}c(f, l) = \dot{f}_{\texttt{load}\ x}(l\dot{o}c(f, l)) = \dot{f}_{\texttt{load}\ x}(s^\sharp, l^\sharp) = (x :: s^\sharp, l^\sharp).$$

Caso $e = a[e]$**:** Aquí resulta $\text{J}e\text{K}_\text{e} = \text{J}e'\text{K}_\text{e}; \texttt{aload}\ a$. Sea $l'' = l + |\text{J}e'\text{K}_\text{e}|$. Por hipótesis de inducción, $l\dot{o}c(f, l'') = (e' :: s^\sharp, lsh)$, y por tanto, dado que $pred(f, l') = \{l''\}$, se verifica

$$l\dot{o}c(f, l') = \dot{f}_{\texttt{aload}\ a}(l\dot{o}c(f, l'')) = \dot{f}_{\texttt{aload}\ a}(e' :: s^\sharp, l^\sharp) = (\llcorner a[e] \lrcorner :: s^\sharp, l^\sharp).$$

Caso $e = e' \mathsf{op}\ e''$**:** Se tiene $\text{J}e\text{K}_\text{e} = \text{J}e''\text{K}_\text{e}; \text{J}e'\text{K}_\text{e}; \texttt{prim op}$. Sea $l'' = l + |\text{J}e''\text{K}_\text{e}| + |\text{J}e'\text{K}_\text{e}|$. Aplicando nuestra hipótesis de inducción, resulta $l\dot{o}c(f, l'') = (e' :: e'' :: s^\sharp, l^\sharp)$. Luego, puesto que $pred(f, l') = \{l''\}$, se cumple

$$l\dot{o}c(f, l') = \dot{f}_{\texttt{prim op}}(l\dot{o}c(f, l'')) = \dot{f}_{\texttt{aload}\ a}(e' :: e'' :: s^\sharp, l^\sharp) = (\llcorner e' \mathsf{op}\ e''_\lrcorner :: s^\sharp, l^\sharp).$$

De este modo, el lema queda probado. □

Lema 2.8. *Sea* $((f, args), body) \in \dot{p}$ *y sean* $s_1, s_2 \in Instr^\star$ *y* $e \in Exp$*, tales que se verifica* $body = s_1 :: k_1 : \text{J}e\text{K}_\text{e} :: k_2 : s_2$. *Entonces,*

$$\forall k \in [k_1, k_2),\ \dot{pre}, \dot{post}, l\dot{o}c \vdash f[k] : body[k].$$

Demostración. Probamos el lema por inducción estructural sobre la expresión e. Observemos que $k_2 \notin \mathsf{dom}(Loc)$ dado que $\dot{p}$ es un programa compilado.

Caso $e = n$: En este caso $\llbracket e \rrbracket_e = \texttt{push}\ n$, $[k_1, k_2) = \{k_1\}$ y puesto que $\mathsf{pred}(k_1+1) = \{k_1\}$, $\dot{loc}(f, k_1+1) = \dot{f}_{\texttt{push}\ n}(\dot{loc}(f, k_1))$, lo cual implica que $\dot{pre}, \dot{post}, \dot{loc} \vdash f[k_1] : \texttt{push}\ n$.

Caso $e = x$: Tenemos $\llbracket e \rrbracket_e = \texttt{load}\ x$, $[k_1, k_2) = \{k_1\}$ y por ser $\mathsf{pred}(f, k_1+1) = \{k_1\}$, resulta $\dot{loc}(f, k_1+1) = \dot{f}_{\texttt{load}\ x}(\dot{loc}(f, k_1))$. Así, $\dot{pre}, \dot{post}, \dot{loc} \vdash f[k_1] : \texttt{load}\ x$.

Caso $e = a[e']$: Aquí, $\llbracket e \rrbracket_e = \llbracket e' \rrbracket_e;\ \texttt{aload}\ a$. Sea $k' = k_1 + |\llbracket e' \rrbracket_e|$. Por hipótesis de inducción, $\forall k \in [k_1, k')$, $\dot{pre}, \dot{post}, \dot{loc} \vdash f[k] : body[k]$, y como $\mathsf{pred}(f, k'+1) = \{k'\}$ entonces $\dot{f}_{\texttt{aload}\ a}(\dot{loc}(f, k'))$, lo que nos permite concluir $\dot{pre}, \dot{post}, \dot{loc} \vdash f[k'] : \texttt{aload}\ a$.

Caso $e = e_1 \texttt{op}\ e_2$: Por compilación, obtenemos $\llbracket e \rrbracket_e = \llbracket e_2 \rrbracket_e;\ \llbracket e_1 \rrbracket_e;\ \texttt{prim op}$. Sean k' y k'' tales que $k'' = k_1 + |\llbracket e'' \rrbracket_e|$ y $k' = k'' + |\llbracket e' \rrbracket_e|$. Por hipótesis de inducción, sabemos que $\forall k \in [k_1, k'') \cup [k'', k')$, $\dot{pre}, \dot{post}, \dot{loc} \vdash f[k] : body[k]$ y dado que el único predecesor de $k'+1$ is k', $\dot{loc}(f, k'+1) = \dot{f}_{\texttt{prim op}}(\dot{loc}(f, k'))$, lo cual significa la validez del juicio $\dot{pre}, \dot{post}, \dot{loc} \vdash f[k'] : \texttt{prim op}$.

Por tanto, hemos probado que $\forall k \in [k_1, k_2)$, $\dot{pre}, \dot{post}, \dot{loc} \vdash f[k] : body[k]$. $\square$

El siguiente lema establece el resultado principal de este capítulo: la compilación preserva soluciones de análisis.

Lema 2.9. *Si $(Pre, Post, Loc)$ es tal que $Pre, Post, Loc \vdash P$, entonces el resultado de análisis $(\dot{pre}, \dot{post}, \dot{loc})$ compilado de $(Pre, Post, Loc)$ es tal que $\dot{pre}, \dot{post}, \dot{loc} \vdash \dot{p}$, es decir, es una solución del análisis de bytecode.*

Demostración. Sea $((f, [x_1, \ldots, x_n]), body) \in \dot{p}$ y supongamos que $body = s_1 :: k_1 : \llbracket s \rrbracket :: k_2 : s_2$ y que existe $l^\sharp$ tal que $Pre, Post, Loc \vdash \{Loc(f, k_1)\}\ s\ \{l^\sharp\}$ y $([], l^\sharp) \sqsubseteq \dot{loc}(f, k_2)$. Debemos probar que $\forall k \in [k_1, k_2)$, $\dot{pre}, \dot{post}, \dot{loc} \vdash f[k] : body[k]$. Para ello, procedemos por inducción sobre la sentencia s. En esta prueba omitimos el cálculo de las etiquetas intermedias.

Caso $s = [\texttt{Skip}]^{k_1}$: Por compilación sabemos que $\llbracket s \rrbracket = \texttt{nop}$. Dado que $\texttt{nop}$ es una instrucción que no provoca salto, para probar $\dot{pre}, \dot{post}, \dot{loc} \vdash f[k_1] : \texttt{nop}$, bastará con mostrar que $\dot{f}_{\texttt{nop}}(\dot{loc}(k_1)) \sqsubseteq \dot{loc}(k_1+1)$. Ahora bien,

$$\begin{aligned} \dot{f}_{\texttt{nop}}(\dot{loc}(f, k_1)) &= \dot{loc}(f, k_1) = ([], Loc(f, k_1)) \\ &= ([], F_{\texttt{Skip}}(\dot{loc}(f, k_1))) \\ &\sqsubseteq ([], l^\sharp) \sqsubseteq \dot{loc}(f, k_2) = \dot{loc}(f, k_1+1). \end{aligned}$$

Por tanto, podemos concluir que $\dot{pre}, \dot{post}, \dot{loc} \vdash f[k_1] : \texttt{nop}$.

Caso $s = [\texttt{x}{:=}e]^{k_1}$**:** Al compilar una asignación, obtenemos que $\llbracket s \rrbracket = \llbracket e \rrbracket_{\mathrm{e}}; k_1' : \texttt{store}\ x$. El Lema 2.8 nos garantiza que $\forall k \in [k_1, k_1'),\ \dot{pre}, \dot{post}, \dot{loc} \vdash f[k] : \mathit{body}[k]$. Por tanto nos resta probar la validez del juicio $\dot{pre}, \dot{post}, \dot{loc} \vdash f[k_1'] : \texttt{store}\ x$. Observemos que $\texttt{store}\ x$ es una instrucción que no provoca salto y, además,

$$
\begin{aligned}
\dot{f}_{\texttt{store}\ x}(\dot{loc}(f, k_1')) &= \dot{f}_{\texttt{store}\ x}(([e], Loc(f, k_1'))) \\
&= ([], \llbracket x{:=}e \rrbracket^{\sharp}(Loc(f, k_1))) \\
&= ([], F_{\texttt{x}:=e}(\dot{loc}(f, k_1))) \\
&\sqsubseteq ([], l^{\sharp}) \sqsubseteq \dot{loc}(f, k_2) = \dot{loc}(f, k_1' + 1).
\end{aligned}
$$

Esto es, $\dot{pre}, \dot{post}, \dot{loc} \vdash f[k_1'] : \texttt{store}\ x$ es un juicio válido.

Caso $s = [\texttt{a}[e_1]{:=}e_2]^{k_1}$**:** La prueba de este caso es similar al anterior. Por compilación resulta $\llbracket s \rrbracket = \llbracket e_2 \rrbracket_{\mathrm{e}}; \llbracket e_1 \rrbracket_{\mathrm{e}}; k_1' : \texttt{astore}\ a$, y por el Lema 2.8, $\forall k \in [k_1, k_1'),\ \dot{pre}, \dot{post}, \dot{loc} \vdash f[k] : \mathit{body}[k]$. Como $\texttt{astore}\ a$ es una instrucción que no provoca salto y

$$
\begin{aligned}
\dot{f}_{\texttt{astore}\ a}(\dot{loc}(f, k_1')) &= \dot{f}_{\texttt{astore}\ a}(([e_1, e_2], Loc(f, k_1'))) \\
&= ([], \llbracket a[e_1]{:=}e_2 \rrbracket^{\sharp}(Loc(f, k_1))) \\
&= ([], F_{\texttt{a[e}_1\texttt{]}:=e_2}(\dot{loc}(f, k_1))) \\
&\sqsubseteq ([], l^{\sharp}) \sqsubseteq \dot{loc}(f, k_2) = \dot{loc}(f, k_1' + 1),
\end{aligned}
$$

entonces $\dot{pre}, \dot{post}, \dot{loc} \vdash f[k_1'] : \texttt{astore}\ a$, único juicio que restaba probar.

Caso $s = \texttt{a}{:=}\texttt{newarray}(e)$**:** La prueba en el caso de la creación de arreglos procede del siguiente modo. Tenemos que $\llbracket s \rrbracket = \llbracket e \rrbracket_{\mathrm{e}}; k_1' : \texttt{newarray}\ a$ y, además, el Lema 2.8 establece que $\forall k \in [k_1, k_1'),\ \dot{pre}, \dot{post}, \dot{loc} \vdash f[k] : \mathit{body}[k]$. Razonando como en los anteriores casos, al ser $\texttt{newarray}\ a$ una instrucción que no provoca salto, nos resta probar que $\dot{f}_{\texttt{newarray}\ a}(\dot{loc}(f, k_1')) \sqsubseteq \dot{loc}(f, k_1'{+}1)$ para garantizar que $\dot{pre}, \dot{post}, \dot{loc} \vdash f[k_1'] : \texttt{newarray}\ a$ sea un juicio válido. Pero esto queda probado puesto que

$$
\begin{aligned}
\dot{f}_{\texttt{newarray}\ a}(\dot{loc}(f, k_1')) &= \dot{f}_{\texttt{newarray}\ a}(([e], Loc(f, k_1'))) \\
&= ([], \llbracket a{:=}nnull \rrbracket^{\sharp}(Loc(f, k_1))) \\
&= ([], F_{\texttt{a}:=\texttt{newarray}(e)}(\dot{loc}(f, k_1))) \\
&\sqsubseteq ([], l^{\sharp}) \sqsubseteq \dot{loc}(f, k_2) = \dot{loc}(f, k_1' + 1).
\end{aligned}
$$

Caso $s = \mathtt{x}{:=}f'(e_1, \ldots, e_n)$**:** La compilación de un llamado a función da como resultado la secuencia de instrucciones $\llbracket e_n \rrbracket_e; \ldots \llbracket e_1 \rrbracket_e; k_1' : \mathtt{call}\ f'; k_2' : \mathtt{store}\ x$. Por el Lema 2.8, $\forall k \in [k_1, k_1'),\ \dot{pre}, \dot{post}, \dot{loc} \vdash f[k] : body[k]$. Nos resta probar la validez de los juicios $\dot{pre}, \dot{post}, \dot{loc} \vdash f[k_1'] : \mathtt{call}\ f'$ y $\dot{pre}, \dot{post}, \dot{loc} \vdash f[k_2'] : \mathtt{store}\ x$.

Sea $(d_1, d_2) = Loc(f, k_1)$ y $Post(f') = (l_1, l_2)$. Por el Lema 2.7,

$$\dot{loc}(f, k_1') = ([e_1, \ldots, e_n], Loc(f, k_1)) \tag{2.1}$$

y por hipótesis de inducción

$$\left(\textstyle\bigsqcap_{i=1}^{n} assume^\sharp(e_i = x_i) \sqcap d_1, d_2\right) \sqsubseteq Pre(f') = \dot{pre}(f') \tag{2.2}$$

Ahora bien, como $k_2' \notin Loc$, entonces

$$\begin{aligned} \dot{loc}(f, k_2') &= \textstyle\bigsqcup_{k \in \mathsf{pred}(f, k_2')} \dot{g}_{k, k_2'}(\dot{loc}(f, k)) \\ &= \dot{g}_{k_1', k_2'}(\dot{loc}(f, k_1')) \\ &= ([\llcorner res \lrcorner], (d_1, d_2 \sqcap l_2)) \end{aligned} \tag{2.3}$$

De este modo, de (2.1) (2.2) y (2.3) podemos concluir que se verifica la validez del juicio $\dot{pre}, \dot{post}, \dot{loc} \vdash f[k_1'] : \mathtt{call}\ f'$. Para mostrar que $\dot{pre}, \dot{post}, \dot{loc} \vdash f[k_2'] : \mathtt{store}\ x$, bastará mostrar que $\dot{f}_{\mathtt{store}\ x}([\llcorner res \lrcorner], (d_1, d_2 \sqcap l_2)) \sqsubseteq \dot{loc}(f, k_2)$. Observemos que, por hipótesis, $([], \llbracket x{:=}res \rrbracket^\sharp(d_1, d_2 \sqcap l_2)) \sqsubseteq \dot{loc}(f, k_2)$. Por lo cual

$$\dot{f}_{\mathtt{store}\ x}([\llcorner res \lrcorner], (d_1, d_2 \sqcap l_2)) = ([], \llbracket x{:=}res \rrbracket^\sharp(d_1, d_2 \sqcap l_2)) \sqsubseteq \dot{loc}(f, k_2).$$

Caso $s = f'(e_1, \ldots, e_n)$**:** El llamado a procedimiento se compila casi idéntico al llamado a función, con la única diferencia que en el llamado a procedimiento el resultado es descartado. Esto es, la instrucción k_2': `store` x es reemplazada por la instrucción k_2': `pop`. Por tanto, omitimos la demostración para el resto de las etiquetas y sólo mostramos que $\dot{pre}, \dot{post}, \dot{loc} \vdash f[k_2'] : \mathtt{pop}$. Observemos que `pop` es una instrucción que no provoca salto y que

$$\begin{aligned} \dot{f}_{\mathtt{astore}\ a}(\dot{loc}(f, k_2')) &= \dot{f}_{\mathtt{pop}}([\llcorner res \lrcorner], (d_1, d_2 \sqcap l_2)) \\ &= ([], (d_1, d_2 \sqcap l_2)) = ([], l^\sharp) \\ &\sqsubseteq \dot{loc}(f, k_2) = \dot{loc}(f, k_2' + 1). \end{aligned}$$

Concluimos, así, la validez del juicio.

Caso $s = s_1; s_2$: La prueba para el caso de la secuenciación no presenta complicaciones. Por compilación, $\llbracket s \rrbracket = \llbracket s_1 \rrbracket; k'_1 \colon \llbracket s_2 \rrbracket$. No tenemos más que aplicar la hipótesis de inducción, la cual establece que $\forall k \in [k_1, k'_1) \cup [k'_1, k_2),\ \dot{pre}, \dot{post}, \dot{loc} \vdash f[k] : \mathit{body}[k]$.

Caso $s = [\texttt{return}\ e]^{k_1}$: Según las reglas del compilador, $\llbracket s \rrbracket = \llbracket e \rrbracket_e; k'_1 : \texttt{return}$. Por el Lema 2.8, podemos asegurar que $\forall k \in [k_1, k'_1),\ \dot{pre}, \dot{post}, \dot{loc} \vdash f[k] : \mathit{body}[k]$. Debemos probar ahora que $\dot{pre}, \dot{post}, \dot{loc} \vdash f[k'_1] : \texttt{return}$. Para ello, observemos que por el Lema 2.7, $\dot{loc}(f, k'_1) = ([e], \mathit{Loc}(f, k_1))$, y además, por ser $(\mathit{Pre}, \mathit{Post}, \mathit{Loc})$ una solución del análisis de P,

$$\llbracket res{:=}e \rrbracket^\sharp(\mathit{Loc}(f, k_1)) \sqsubseteq \mathit{Post}(f) = \dot{post}(f).$$

Así, $\dot{pre}, \dot{post}, \dot{loc} \vdash f[k'_1] : \texttt{return}$ es un juicio válido.

Caso $s = \texttt{if}\ [e_1 \bowtie e_2]^{k_1}\ \texttt{then}\ s'\ \texttt{else}\ s''$: Podemos ver que, si compilamos un condicional, obtenemos $\llbracket s \rrbracket = \llbracket e_2 \rrbracket_e; \llbracket e_1 \rrbracket_e; k'_1 : \texttt{cjmp} \bowtie k'_4; k'_2 : \llbracket s'' \rrbracket; k'_3 : \texttt{jmp}\ k_2; k'_4 : \llbracket s' \rrbracket_e$. Aquí son varios los juicios que debemos probar válidos. Por hipótesis de inducción y el Lema 2.8 sabemos que $\forall k \in [k_1, k'_1) \cup [k'_2, k'_3) \cup [k'_4, k_2),\ \dot{pre}, \dot{post}, \dot{loc} \vdash k : \mathit{body}[k]$

Asumiendo nuestra hipótesis, tenemos

$$\begin{array}{rl} & \mathit{Pre}, \mathit{Post}, \mathit{Loc} \vdash \{\llbracket e_1 \bowtie e2 \rrbracket^\sharp(\mathit{Loc}(f, k_1))\}\ s'\ \{l_1^\sharp\} \\ \text{y} & \mathit{Pre}, \mathit{Post}, \mathit{Loc} \vdash \{\llbracket \neg(e_1 \bowtie e2) \rrbracket^\sharp(\mathit{Loc}(f, k_1))\}\ s''\ \{l_2^\sharp\} \end{array}$$

donde $l_1^\sharp \sqcup l_2^\sharp = l^\sharp$.

Se puede probar que para todo juicio de la forma $\mathit{Pre}, \mathit{Post}, \mathit{Loc} \vdash \{d_1^\#\}\ s\ \{d_2^\#\}$, se tiene que $d_1^\# = \mathit{Loc}(f, \mathit{init}(s))$. Por lo tanto,

$$\begin{array}{rl} & \dot{loc}(f, k'_4) = ([], \mathit{Loc}(f, k'_4)) = ([], \llbracket e_1 \bowtie e_2 \rrbracket^\sharp(\mathit{Loc}(f, k_1))) \\ y & \dot{loc}(f, k'_2) = ([], \mathit{Loc}(f, k'_2)) = ([], \llbracket \neg(e_1 \bowtie e_2) \rrbracket^\sharp(\mathit{Loc}(f, k_1))). \end{array}$$

Además, $\mathit{Loc}(f, k'_1) = ([e_1, e_2], \mathit{Loc}(f, k_1))$ por el Lema 2.7. De este modo, se concluye que $\dot{pre}, \dot{post}, \dot{loc} \vdash f[k'_1] : \texttt{cjmp} \bowtie k'_4$.

Si bien no hacemos la demostración aquí, se verifica que para toda sentencia s tal que $f = s_1 :: k : \llbracket s \rrbracket :: k' : s_2$, $k' \notin \mathsf{dom}(\mathit{Loc})$ y que $\mathit{Loc} \vdash \{\mathit{Loc}(f, \mathit{init}(s))\}\ s\ \{l^\sharp\}$, entonces $\dot{loc}(f, k') \sqsubseteq ([], l^\sharp)$. Ahora, dado que $k'_3 \notin \mathsf{dom}(\mathit{Loc})$, resulta $\dot{loc}(f, k'_3) \sqsubseteq ([], l_2^\sharp)$. Adicionalmente, $([], l_2^\sharp) \sqsubseteq ([], l^\sharp) \sqsubseteq \dot{loc}(f, k_2)$. Por lo tanto, $\dot{loc}(f, k'_3) \sqsubseteq \dot{loc}(f, k_2)$ implica que el juicio $\dot{pre}, \dot{post}, \dot{loc} \vdash f[k'_3] : \texttt{jmp}\ k_2$ es válido, lo cual completa la prueba de este caso.

Caso $s = \texttt{while}\ [e_1 \bowtie e_2]^{k_1}\ \texttt{do}\ s'$: En el caso de un bucle, las reglas de compilación establecen que $[\![s]\!] = [\![e_2]\!]_e; [\![e_1]\!]_e; k'_1 : \texttt{cjmp} \bowtie k'_3; k'_2 : \texttt{jmp}\ k_2; k'_3 : [\![s']\!]; k'_4 : \texttt{jmp}\ k_1$. La hipótesis de inducción y el Lema 2.8 establecen que $\forall k \in [k_1, k'_1) \cup [k'_3, k'_4),\ \dot{pre}, \dot{post}, \dot{loc} \vdash k : body[k]$. Por tanto, debemos probar la validez de los juicios para las etiquetas k'_1, k'_2 y k'_4. Usando el Lema 2.7 sabemos que

$$\dot{loc}(f, k'_1) = ([e_1, e_2], Loc(f, k_1)) \tag{2.4}$$

Además, dado que $k'_2 \notin Loc$ y $\mathsf{pred}(k'_2) = \{k'_1\}$ se verifica la siguiente igualdad,

$$\begin{aligned} \dot{loc}(f, k'_2) &= \dot{g}_{k'_1, k'_2}(\dot{loc}(f, k'_1)) \\ &= ([], [\![\neg(e_1 \bowtie e_2)]\!]^\sharp(Loc(f, k_1))) \end{aligned} \tag{2.5}$$

Asumiendo nuestra hipótesis, el juicio $Loc \vdash \{[\![e_1 \bowtie e_2]\!]^\sharp(Loc(f, k_1))\}\ s'\ \{l^\sharp_{s'}\}$ es válido. De esto y del hecho de que $k'_3 = init(s')$, resulta

$$\dot{loc}(f, k'_3) = ([], [\![e_1 \bowtie e_2]\!]^\sharp(Loc(f, k_1))) \tag{2.6}$$

Tal como afirmamos en el caso anterior, $\dot{loc}(f, k'_4) \sqsubseteq l^\sharp_{s'}$ como consecuencia de que $k'_4 \notin Loc$. Esto nos permite llegar a la siguiente conclusión,

$$\dot{loc}(f, k'_4) \sqsubseteq \dot{loc}(f, k_2) \tag{2.7}$$

Finalmente, por hipótesis, resulta

$$[\![\neg(e_1 \bowtie e_2)]\!]^\sharp(Loc(f, k_1)) \sqsubseteq \dot{loc}(f, k_2) \tag{2.8}$$

Así, (2.4), (2.5) y (2.6) implican la validez de $\dot{pre}, \dot{post}, \dot{loc} \vdash k'_1 : \texttt{cjmp} \bowtie k'_3$. Además, (2.5) y (2.8) implican $\dot{pre}, \dot{post}, \dot{loc} \vdash k'_2 : \texttt{jmp}\ k_2$ y (2.7) permite verificar que el juicio $\dot{pre}, \dot{post}, \dot{loc} \vdash k'_4 : \texttt{jmp}\ k_1$ es válido. De este modo, completamos la prueba para este caso.

La inducción estructural realizada sobre el cuerpo de la función f nos permite concluir que $\dot{pre}, \dot{post}, \dot{loc} \vdash f$ es un juicio válido. Esto, por supuesto, se verifica cualquiera sea f, ya que no hemos hecho ningún supuesto especial sobre la función. Observemos, también, que por ser $(Pre, Post, Loc)$ una solución del análisis de alto nivel, se verifica que $((main, []), s)$ es una función de P, y por tanto de $\dot{p}$, y además $\top \sqsubseteq Pre(main)$. Dado que $\dot{pre} = Pre$, de todo lo anterior concluimos que $\dot{loc} \vdash \dot{p}$, lo cual concluye esta demostración. $\square$

3

Preservación de Obligaciones de Prueba

"Errors are not in the art but in the artificers." [**Issac Newton**]

En este capítulo definimos dos frameworks de verificación, respectivamente, para programas de alto y bajo nivel del capítulo anterior. Consideramos como lenguaje de especificación las fórmulas de primer orden, es decir, el dominio de aserciones $\mathcal{A}$. La validez de una afirmación en un estado particular de ejecución $\eta \in State^s$ es estándar. En particular, una aserción que contiene la expresión $a[e]$ es inválida en aquellas ejecuciones en las cuales a es nula o e está fuera de los límites de a.

Consideramos como una especificación de un programa una tupla $(\mathsf{pre}, \mathsf{annot}, \mathsf{post}, \chi)$, donde $\mathsf{pre}, \mathsf{post}, \chi : FuncId \rightharpoonup \mathcal{A}$ son funciones parciales que especifican, respectivamente, las pre y post condiciones de las funciones del programa. post especifica las post-condiciones de terminación normal y χ las post-condiciones de terminación anormal. La función parcial annot : $FuncId \times \mathbf{Lab} \rightharpoonup \mathcal{A}$ mapea etiquetas a especificaciones de puntos internos para cada función. La variable especial res hace referencia al valor devuelto por una función y cada pre sólo hace referencia a variables de V. Cuando se especifica un programa bytecode, las aserciones pueden referirse a la variable especial s que representa el stack de operandos.

Decimos que un programa satisface la especificación $(\mathsf{pre}, \mathsf{annot}, \mathsf{post}, \chi)$, si, para cada función f del programa, toda ejecución que comienza en un estado que satisface $\mathsf{pre}(f)$ sólo alcanza estados finales normales que satisfacen $\mathsf{post}(f)$ o estados anormales que satisfacen $\chi(f)$ y sólo alcanza puntos intermedios l-etiquetados que satisfacen $\mathsf{annot}(f, l)$. Dada una especificación de programa $(\mathsf{pre}, \mathsf{annot}, \mathsf{post}, \chi)$, un framework generador de condiciones de verificación (VC-gen) proporciona suficientes obligaciones de prueba que garantizan que el programa satisface

la especificación.

Los VCgens definidos en este capítulo son híbridos en el sentido de que aprovechan el análisis previamente computado para reducir el tamaño de las obligaciones de prueba Suponemos que el resultado de un análisis relacional ($(Pre, Post, Loc)$ y $(\dot{pre}, \dot{post}, \dot{loc})$ para programas fuente y bytecode, respectivamente) se da como entrada al VCgen. Para el dominio abstracto $\mathbb{D}$, consideramos una relación $\models \subseteq \mathbb{D} \times \mathcal{A}$ tal que para cualquier guarda b y cualquier $d \in \mathbb{D}$, $d \models b$ indica que la interpretación del elemento abstracto d garantiza la validez de la condición b. Por ejemplo, al intentar acceder a un arreglo en la expresión $a[x]$, chequearemos que la variable de arreglo a sea no nula y que la variable x esté dentro de los límites de a. Si instanciamos $\mathbb{D}$ con una combinación del dominio de los poliedros convexos (convex polyhedra) y un dominio para nullness, cada elemento representa un conjunto de restricciones lineales y una función que mapea variables de arreglo a $\{null, nnull\}$ de donde podemos descubrir si la condición $a \neq null \wedge 0 \leq x < |a|$ se satisface.

Otra mejora respecto a los VCgen estándar consiste en reusar el resultado del análisis para reforzar los invariantes de los bucles. Esta técnica ayuda a reducir el tamaño de las anotaciones y la carga de la especificación interactiva. Con este fin, suponemos la existencia de una función de concreción $\gamma_{\mathsf{a}} : \mathbb{D} \to \mathcal{A}$ para interpretar elementos abstractos $d \in \mathbb{D}$ como aserciones.

3.1. VCgen para Programas Fuente

Consideremos una especificación $(\mathsf{pre}, \mathsf{annot}, \mathsf{post}, \chi)$ para el programa fuente P. En esta sección, asumimos que annot es una anotación suficiente de P, esto es, para todo subprograma $\texttt{while}\ [t]^l\ \texttt{do}\ c$ de P, tenemos que $l \in \mathsf{dom}(\mathsf{annot})$.

Un VCgen para programas fuente está definido por los conjuntos de obligaciones de prueba $\mathsf{PO}(f)$, donde para cada función f en $\dot{p}$, $\mathsf{PO}(f)$ se define como $\{\mathsf{pre}(f) \wedge \gamma_{\mathsf{a}}(Pre(f)) \Rightarrow \phi[^{\vec{V}}/_{\vec{V}^{old}}]\} \cup \theta$, donde $\langle \phi, \theta \rangle = \mathsf{WP}(s, \mathsf{post}(f))$, $\phi[^{\vec{V}}/_{\vec{V}^{old}}]$ representa el resultado de sustituir en ϕ cualquier variable de arreglo o escalar x^{old} en $V_s^{old} + V_a^{old}$ por x, s es el cuerpo de f y la función WP se define en la Figura 3.1. En la figura, R refiere a las variables de arreglo modificadas por la función llamada y la aserción $\mathsf{okA}(e)$ refiere a la condición que debe satisfacer un estado de ejecución para garantizar que todo acceso a arreglo en e es válido, es decir, que el arreglo está definido y el índice está dentro de los límites de éste. Por ejemplo, si e no contiene expresiones de

arreglo, $\mathsf{okA}(e)$ se define como $\mathtt{true}$ y $\mathsf{okA}(a[e])$ como $a \neq null \wedge 0 \leq e < |a|$. Seguimos la hipótesis simplificadora de que las expresiones no contienen más de un acceso a arreglo. Para cualquier variable a y expresiones e_1 y e_2, interpretamos $\mathsf{upd}(a, e_1, e_2)$ como el arreglo a tal que $a'[e]$ se evalúa a e_2 si $e_1 = e$ y a $a[e]$ en caso contrario. Para simplificar la presentación de los ejemplos, las obligaciones de prueba para las sentencias `while` están divididas en dos aserciones correspondientes a las ramas true y false.

La función WP considera el resultado del análisis $(Pre, Post, Loc)$ para reducir el tamaño de las obligaciones de prueba. Esto es, si el valor abstracto $Loc(f, l)$ asociado al punto de programa bajo consideración indica que cualquier acceso a arreglo en la sentencia es válido, el predicado devuelto es simplificado al omitir la post-condición excepcional.

Consideremos el Programa 3.1. Si el análisis es capaz de computar en la etiqueta k_1 un valor abstracto d tal que $d \models A \neq null \wedge 0 \leq i < |A|$, la función WP retornará la aserción $\mathsf{upd}(A, i, A[0])[i + 1 - 1] = A[0]$, la cual junto con el invariante de bucle en la etiqueta k da lugar a la obligación de prueba:

$$
\begin{aligned}
A[i-1] = 0 \wedge \boxed{A \neq null \wedge 0 \leq i \leq |A|} \Rightarrow \\
i < |A| \Rightarrow \mathsf{upd}(A, i, A[0])[i + 1 - 1] = A[0]
\end{aligned}
$$

donde la aserción enmarcada $\boxed{A \neq null \wedge 0 \leq i \leq |A|}$ representa el resultado del análisis en el punto de entrada al bucle.

En contraste, si no aprovechamos el resultado del análisis, debemos proporcionar una fórmula mayor:

$$
\begin{aligned}
& A[i-1] = 0 \wedge \boxed{A \neq null \wedge 0 \leq i \leq |A|} \Rightarrow \\
& \quad i < |A| \Rightarrow \\
& \quad\quad (A \neq null \wedge 0 \leq i < |A| \Rightarrow \mathsf{upd}(A, i, A[0])[i + 1 - 1] = A[0] \\
& \quad\quad \wedge \neg(A \neq null \wedge 0 \leq i < |A|) \Rightarrow \mathsf{false})
\end{aligned}
$$

Como puede observarse en la definición de WP, las obligaciones de prueba son de la forma $\phi_1 \wedge \boxed{\gamma_{\mathsf{a}}(d)} \Rightarrow \phi_2$, mientras que un estándar VCgen produce la obligación de prueba más fuerte $\phi_1 \Rightarrow \phi_2$. En consecuencia, uno puede proporcionar el código con un invariante más débil ϕ_1 siempre que el analizador sea capaz de, eventualmente, inferir la información faltante $\gamma_{\mathsf{a}}(d)$. Por ejemplo, para el sencillo Programa 3.1, un VCgen estándar retornará la obligación de prueba inválida

$$A[i-1] = A[0] \Rightarrow \neg(i < |A|) \Rightarrow A[|A| - 1] = A[0]$$

$$\frac{}{\mathsf{WP}([\texttt{Skip}]^l, \phi) = \langle \phi, \emptyset \rangle} \qquad \frac{}{\mathsf{WP}([\texttt{return}\ e]^l, \phi) = \langle \mathsf{ckA}(e, \mathsf{post}(f)[^{e}/_{\mathsf{res}}]), \emptyset \rangle}$$

$$\frac{}{\mathsf{WP}([\mathtt{x}{:=}e]^l, \phi) = \langle \mathsf{ckA}(e, \phi[^{e}/_{x}]), \emptyset \rangle} \qquad \frac{}{\mathsf{WP}([\mathsf{a}[e_1]{:=}e_2]^l, \phi) = \langle \mathsf{ckA}(e_2, \mathsf{ckA}(a[e_1], \phi[^{\mathsf{upd}(a,e_1,e_2)}/_{a}])), \emptyset \rangle}$$

$$\frac{}{\mathsf{WP}([\mathsf{a}{:=}\texttt{newarray}(e)]^l, \phi) = \langle \mathsf{ckA}(e, \phi[^{[0\mapsto 0,\ldots,e-1\mapsto 0]}/_{a}]), \emptyset \rangle} \qquad \frac{\mathsf{WP}(c_1, \phi_2) = \langle \phi_1, \theta_1 \rangle \qquad \mathsf{WP}(c_2, \phi) = \langle \phi_2, \theta_2 \rangle}{\mathsf{WP}(c_1; c_2, \phi) = \langle \phi_1, \theta_1 \cup \theta_2 \rangle}$$

$$\frac{\Phi = \mathsf{pre}(f')[^{e_1,\ldots,e_n}/_{x_1,\ldots,x_n}] \wedge \forall \mathsf{res}, R' \bullet \mathsf{post}(f')[^{R',R}/_{R,R^{old}}][^{e_1,\ldots,e_n}/_{x_1^{old},\ldots,x_n^{old}}] \Rightarrow \phi[^{R'}/_{R}][^{\mathsf{res}}/_{x}]}{\mathsf{WP}([\mathtt{x}{:=}f'(x_1,\ldots,x_n)]^l, \phi) = \langle \mathsf{ckA}(e_1, \ldots \mathsf{ckA}(e_n, \Phi)\ldots), \emptyset \rangle}$$

$$\frac{\Phi = \mathsf{pre}(f')[^{e_1,\ldots,e_n}/_{x_1,\ldots,x_n}] \wedge \forall \mathsf{res}, R' \bullet \mathsf{post}(f')[^{R',R}/_{R,R^{old}}][^{e_1,\ldots,e_n}/_{x_1^{old},\ldots,x_n^{old}}] \Rightarrow \phi[^{R'}/_{R}]}{\mathsf{WP}([f'(x_1,\ldots,x_n)]^l, \phi) = \langle \mathsf{ckA}(e_1, \ldots \mathsf{ckA}(e_n, \Phi)\ldots), \emptyset \rangle}$$

$$\frac{\mathsf{WP}(c_1, \phi) = \phi_1, \theta_1 \qquad \mathsf{WP}(c_2, \phi) = \phi_2, \theta_2}{\mathsf{WP}(\texttt{if}\ [t]^l\ \texttt{then}\ c_1\ \texttt{else}\ c_2, \phi) = \langle \mathsf{ckA}(t, t \Rightarrow \phi_1 \wedge \neg t \Rightarrow \phi_2), \theta_1 \cup \theta_2 \rangle}$$

$$\frac{\mathsf{WP}(c, \phi) = \langle \phi_1, \theta_1 \rangle \qquad \Phi = (t \Rightarrow \phi_1) \wedge (\neg t \Rightarrow \phi)}{\mathsf{WP}(\texttt{while}\ [t]^l\ \texttt{do}\ c, \phi) = \langle \mathsf{annot}(f,l), \{\mathsf{annot}(f,l) \wedge \gamma_{\mathsf{a}}(Loc(f,l)) \Rightarrow \mathsf{ckA}(t, \Phi)\} \cup \theta_1 \rangle}$$

donde $\mathsf{ckA}(e, \varphi)$ refiere a φ si $Loc(f,l) \models \mathsf{okA}(e)$ y a la fórmula $\mathsf{okA}(e) \Rightarrow \varphi \wedge \neg\mathsf{okA}(e) \Rightarrow \chi$ en caso contrario.

Figura 3.1: Definición de la función WP

para el camino que no entra al bucle. Es suficiente con proporcionar un invariante más fuerte, es decir en conjunción con la condición $i \leq |A|$, para demostrar la correctitud del programa. Sin embargo, como una alternativa a incrementar el tamaño de las anotaciones, asumiendo que la condición $i \leq |A|$ es inferida por el análisis, el VCgen híbrido genera la siguiente obligación de prueba más débil (y válida):

$$\begin{gathered} A[i-1] = A[0] \wedge \boxed{A \neq null \wedge 0 \leq i \leq |A|} \Rightarrow \\ \neg(i < |A|) \Rightarrow A[|A|-1] = A[0] \end{gathered}$$

3.2. VCgen para Programas Bytecode

Sea $(\mathsf{pre}, \mathsf{annot}, \mathsf{post}, \chi)$ una especificación para el programa bytecode $\dot{p}$. Así como en el VCgen para programas fuente antes definido, la pre-condición $\mathsf{pre}(f)$ y las anotaciones internas $\mathsf{annot}(f,l)$ son fortalecidas con el resultado del análisis, para cada función f en $\dot{p}$. Con tal fin, interpretamos el resultado del análisis con la ayuda de las funciones de concreción $\gamma_{\mathsf{a}} : \mathbb{D} \to \mathcal{A}$ y $\bar{\gamma_{\mathsf{a}}} : (Expr^{\star} \times \mathbb{D}) \to \mathcal{A}$. Un VCgen para bytecode se define mediante la extracción de las

Programa 3.1 Ejemplo

```
//   pre(f) : true,  χ(f) : false
f() {
  [i := 1]^{k_0};
    //   A[i − 1] = A[0]
  while [i < |A|]^k do {
    [A[i] := A[0]]^{k_1};  [i := i + 1]^{k_2}
  }
    //   A[|A| − 1] = A[0]
  ...
}
```

obligaciones de prueba:

$$\mathsf{po}(f) = \{\mathsf{pre}(f) \wedge \gamma_{\mathsf{a}}(\dot{pre}(f)) \Rightarrow \mathsf{wpi}(f, init(s))[{}^{V}/_{\vec{V_{old}}}]\} \cup$$
$$\{\mathsf{annot}(f,l) \wedge \bar{\gamma_{\mathsf{a}}}(\dot{loc}(f,l)) \Rightarrow \mathsf{wpi}(f,l) \mid (f,l) \in \mathsf{dom}(\mathsf{annot})\}$$

para cada f en $\dot{p}$, donde s es el cuerpo de la función f y el transformador de predicados wp se muestra en la Figura 3.2. Si un punto de programa está anotado, la función wp retorna $\mathsf{annot}(f,l)$. En caso contrario, aplica la función wpi, definido en términos de la instrucción en un punto de programa l, tomando como parámetros las anotaciones computadas para el punto de programa sucesor. La definición de wp y wpi es hecha por inducción a lo largo de los caminos de control de flujo del programa. Un programa $\dot{p}$ es suficientemente anotado si el grafo de control de flujo de $\dot{p}$ no contiene bucles no anotados. El principio de inducción derivado de la definición de programas suficientemente anotados es suficiente para garantizar que wp y wpi' están bien definidas. Para una lista s, $s[0]$ y $s[1]$ representan el primer y segundo elemento de s, y $\uparrow s$ denota el resultado de remover de s su primer elemento.

3.3. Preservación de Obligaciones de Prueba

Consideremos la especificación $(\mathsf{pre}, \mathsf{annot}, \mathsf{post}, \chi)$ para el programa fuente P, y supongamos que annot es una anotación suficiente para P, es decir, todo bucle está anotado. Hagamos que $(\mathsf{pre}, \mathsf{annot}, \mathsf{post}, \chi)$ también defina la especificación para el programa bytecode $\dot{p}$. De resultados previos, sabemos que si annot es una anotación suficiente para P, entonces también es una anotación suficiente para el resultado de la compilación $\dot{p}$. Sea $(Pre, Post, Loc)$ una solución

$$
\begin{array}{ll}
\mathsf{wpi}(f,l) = \mathsf{wp}(f,l+1)[^{\mathsf{s}[0]\ \mathsf{op}\ \mathsf{s}[1]::\uparrow^2\mathsf{s}}/_{\mathsf{s}}] & \dot{p}[l] = \texttt{prim op} \\
\mathsf{wpi}(f,l) = \mathsf{wp}(f,l+1)[^{v::\mathsf{s}}/_{\mathsf{s}}] & \dot{p}[l] = \texttt{push}\ v \\
\mathsf{wpi}(f,l) = \mathsf{wp}(f,l+1)[^{\mathsf{s}[0],\uparrow\mathsf{s}}/_{x,\mathsf{s}}] & \dot{p}[l] = \texttt{store}\ x \\
\mathsf{wpi}(f,l) = \mathsf{wp}(f,l+1)[^{x::\mathsf{s}}/_{\mathsf{s}}] & \dot{p}[l] = \texttt{load}\ x \\
\mathsf{wpi}(f,l) = \mathsf{ckA}(\mathsf{wp}(f,l+1)[^{\mathsf{upd}(a,\mathsf{s}[0],\mathsf{s}[1]),\uparrow^2\mathsf{s}}/_{a,\mathsf{s}}]) & \dot{p}[l] = \texttt{astore}\ a \\
\mathsf{wpi}(f,l) = \mathsf{ckA}(\mathsf{wp}(f,l+1)[^{a[\mathsf{s}[0]]::\uparrow\mathsf{s}}/_{\mathsf{s}}]) & \dot{p}[l] = \texttt{aload}\ a \\
\mathsf{wpi}(f,l) = \mathsf{wp}(f,l+1)[^{[0\mapsto 0,\ldots,\mathsf{s}[0]-1\mapsto 0],\uparrow\mathsf{s}}/_{a,\mathsf{s}}] & \dot{p}[l] = \texttt{newarray}\ a \\
\mathsf{wpi}(f,l) = \mathsf{s}[0] \bowtie \mathsf{s}[1] \Rightarrow \mathsf{wp}(f,l')[^{\uparrow^2\mathsf{s}}/_{\mathsf{s}}] & \dot{p}[l] = \texttt{cjmp} \bowtie l' \\
\qquad \wedge\neg(\mathsf{s}[0] \bowtie \mathsf{s}[1]) \Rightarrow \mathsf{wp}(f,l+1)[^{\uparrow^2\mathsf{s}}/_{\mathsf{s}}] & \\
\mathsf{wpi}(f,l) = \mathsf{wp}(f,l') & \dot{p}[l] = \texttt{jmp}\ l' \\
\mathsf{wpi}(f,l) = \mathsf{wp}(f,l+1) & \dot{p}[l] = \texttt{nop} \\
\mathsf{wpi}(f,l) = \mathsf{post}[^{\mathsf{s}[0]}/_{\mathsf{res}}] & \dot{p}[l] = \texttt{return} \\
\mathsf{wpi}(f,l) = \mathsf{pre}(f')[^{\mathsf{s}[0],\ldots,\mathsf{s}[n-1]}/_{x_1,\ldots,x_n}] & \dot{p}[l] = \texttt{call}\ f' \\
\qquad \wedge\forall \mathsf{res}, R' \bullet \mathsf{post}(f')[^{R,R^{old}}/_{R',R}][^{e_1,\ldots,e_n}/_{x_1^{old},\ldots,x_n^{old}}] & \\
\qquad\qquad \Rightarrow \mathsf{wp}(f,l+1)[^{R'}/_{R}] &
\end{array}
$$

donde $\mathsf{ckA}(\psi)$ refiere a ψ si $\dot{loc}(f,l) \models \mathsf{okA}(x[\mathsf{s}[0]])$ y a $\mathsf{okA}(x[\mathsf{s}[0]]) \Rightarrow \psi \wedge \neg\mathsf{okA}(x[\mathsf{s}[0]]) \Rightarrow \chi$ en caso contrario.

$$
\mathsf{wp}(f,l) = \begin{cases} \mathsf{annot}(f,l) & \text{si } (f,l) \in \mathsf{dom}(\mathsf{annot}) \\ \mathsf{wpi}(f,l) & \text{caso contrario} \end{cases}
$$

Figura 3.2: VCgen para Programas Bytecode

del análisis del programa P y sea $(\dot{pre}, \dot{post}, \dot{loc})$ una solución del análisis del programa bytecode $\dot{p}$, obtenida por compilación de $(Pre, Post, Loc)$ tal como se ha descripto en la Sección 2.2.5 capítulo anterior.

Supongamos que las funciones de concreción satisfacen la propiedad $\bar{\gamma}_{\mathsf{a}}([], d) = \gamma_{\mathsf{a}}(d)$, de modo que la intrepretación de los resultados del análisis abstracto, tanto de alto como bajo nivel, coincidan (recordemos que por definición $\dot{loc}(f,l) = ([\ ], Loc(f,l))$ para todo (f,l) en $\mathsf{dom}(Loc)$). Además, para cualquier expresión e y cualquier $d \in \mathbb{D}$, si e no contiene expresiones de arreglos, es decir $\mathsf{okA}(e) = \mathsf{true}$, entonces $d \models \mathsf{okA}(e)$.

El siguiente resultado sobre compilación de expresiones es útil para provar preservación de obligaciones de pruebas:

Lema 3.1. *Sea* $((f, args), s) \in \dot{p}$ *tal que* s *es igual a* $s_1 :: l_1 : \mathrm{JeK_e} :: l_2 : s_2$. *Entonces,* $\mathsf{wpi}(f, l_1)$ *es igual a* $\mathsf{wpi}(f,l_2)[^{e::\mathsf{s}}/_{\mathsf{s}}]$ *si* $\dot{loc}(f,l_1) \models \mathsf{okA}(e)$ *e igual a* $\mathsf{okA}(e) \Rightarrow \mathsf{wpi}(f,l_2)[^{e::\mathsf{s}}/_{\mathsf{s}}] \wedge \neg\mathsf{okA}(e) \Rightarrow \chi$ *en caso contrario.*

Programa 3.2 Ejemplo (compilación del Programa 3.1)

```
f :
  k0:push 1              load i
     store i             prim +
   k:jmp k'              store i
  k1:push 0           k':push |A|
     aload A             load i
     load i              cjmp < k1
     astore A        k'':...
  k2:push 1
```

Demostración. El resultado se verifica bajo la hipótesis antes realizada de que la expresión e contiene como máximo un acceso a arreglo. De lo contrario, la igualdad sintáctica no se cumple, pero no es complicado mostrar una equivalencia lógica.

La demostración procede por inducción sobre la expresión e. □

La siguiente proposición establece la coincidencia de los conjuntos de obligaciones de pruebas PO y po, siempre que el programa bytecode $\dot{p}$ sea el resultado de compilar el programa fuente P.

Proposición 3.2. *Sea f una función de P. Para todo subprograma c de f_P, las obligaciones de prueba correspondientes a c son iguales a las obligaciones de prueba en $f_{\dot{p}}$ que corresponden a la subsecuencia* $\mathrm{J}c\mathrm{K}$.

Consideremos el Programa 3.2, el cual es un programa bytecode obtenido por compilación del Programa 3.1. Podemos ver que la obligación de prueba en la etiqueta k es

$$\begin{array}{l} A[i-1] = A[0] \wedge \boxed{A \neq null \wedge 0 \leq i \leq |A|} \Rightarrow \\ \quad (i < |A| \Rightarrow (A[i-1] = A[0])[^{\mathsf{upd}(A,i,A[0]),i+1}/_{A,i}]) \wedge \\ \quad (\neg(i < |A|) \Rightarrow A[|A|-1] = A[0]) \end{array}$$

que es igual a la obligación de prueba en la etiqueta k del Programa Fuente 3.1.

$$
\begin{array}{ll}
\widehat{\mathsf{wpi}}(f,l) = \mathsf{ckA}(\hat{\mathsf{wp}}(f,l+1)[^{\mathsf{upd}(x,\mathbf{s}[0],\mathbf{s}[1]),\uparrow^2\mathbf{s}}/_{x,\mathbf{s}}]) & \dot{p}[l] = \texttt{astore}\ x \\
\widehat{\mathsf{wpi}}(f,l) = \mathsf{ckA}(\hat{\mathsf{wp}}(f,l+1)[^{x[\mathbf{s}[0]]::\uparrow\mathbf{s}}/_{\mathbf{s}}]) & \dot{p}[l] = \texttt{aload}\ x \\
\widehat{\mathsf{wpi}}(f,l) = \mathsf{wpi}(f,l) & \text{caso contrario}
\end{array}
$$

donde $\mathsf{ckA}(\psi)$ referencia a $\mathsf{okA}(x[\mathbf{s}[0]]) \Rightarrow \psi \wedge \neg\mathsf{okA}(x[\mathbf{s}[0]]) \Rightarrow \chi$ sin importar si $\dot{loc}(f,l) \models \mathsf{okA}(x[\mathbf{s}[0]])$ se satisface.

$$
\hat{\mathsf{wp}}(f,l) = \begin{cases} \mathsf{annot}(f,l) & \text{si } (f,l) \in \mathsf{dom}(\mathsf{annot}) \\ \widehat{\mathsf{wpi}}(f,l) & \text{caso contrario} \end{cases}
$$

$$
\hat{\mathsf{po}}(f) = \{\widehat{\mathsf{annot}}(f,l) \Rightarrow \widehat{\mathsf{wpi}}(f,l) \mid (f,l) \in \mathsf{dom}(\widehat{\mathsf{annot}})\}
$$

Figura 3.3: VCgen No-Híbrido para Bytecode

3.4. De VCgen Híbrido a VCgen Estándar

En esta sección mostramos la correspondencia entre el VCgen híbrido para bytecode, definido en la sección anterior, y un VCgen estándar que no aprovecha el resultado del análisis. Más precisamente, interpretando el resultado abstracto como fórmulas lógicas, mostramos la equivalencia entre las obligaciones de pruebas de ambos VCgen. Suponiendo que la relación $\models$ satisface una condición de corrección, la correctitud del VCgen híbrido es una consecuencia de la correctitud del VCgen estándar. Además, la correctitud del VCgen para programas está asegurada si el compilador preserva semántica.

Dada una especificación $(\hat{\mathsf{pre}}, \widehat{\mathsf{annot}}, \mathsf{post}, \chi)$ para el programa bytecode $\dot{p}$, un VCgen no híbrido extrae el conjunto de obligaciones de pruebas $\hat{\mathsf{po}}(f) \cup \{\hat{\mathsf{pre}}(f) \Rightarrow \widehat{\mathsf{wpi}}(f, init(s))[^{V}/_{V^{old}}]\}$, para cada f en $\dot{p}$, donde s es el cuerpo de f y $\widehat{\mathsf{wpi}}$ y $\hat{\mathsf{po}}$ están definidas en la Figura 3.3. Para evitar ambigüedad, en lo que resta hacemos explícitos algunos parámetros necesarios en la definición de wpi, wp, $\widehat{\mathsf{wpi}}$ y $\hat{\mathsf{wp}}$. Escribimos, por ejemplo, $\widehat{\mathsf{wpi}}(f, l, \widehat{\mathsf{annot}}, \mathsf{post}, \chi)$ en vez de $\widehat{\mathsf{wpi}}(f,l)$.

Sea $(\dot{pre}, \dot{post}, \dot{loc})$ un resultado del análisis de programa de bajo nivel $\dot{p}$. Consideremos las especificaciones $(\mathsf{pre}, \mathsf{annot}, \mathsf{post}, \chi)$ y $(\hat{\mathsf{pre}}, \widehat{\mathsf{annot}}, \mathsf{post}, \chi)$ para el programa $\dot{p}$, tales que toda f en $\dot{p}$, $\hat{\mathsf{pre}}$ está definida como $\mathsf{pre}(f) \wedge \gamma_{\mathsf{a}}(\dot{pre}(f))$ y para todo (f,l) en $\mathsf{dom}(\mathsf{annot})$, $\widehat{\mathsf{annot}}(f,l)$ está definido como $\mathsf{annot}(f,l) \wedge \bar{\gamma}_{\mathsf{a}}(\dot{loc}(f,l))$. Decimos que la relación $\models \subseteq \mathbb{D} \times Guard$ es válida

si para cada elemento abstracto $d \in \mathbb{D}$ y $b \in Guard$ tenemos que $d \models b$ implica la validez universal de $\gamma_{\mathsf{a}}(d) \Rightarrow b$. El resultado del análisis $(\dot{pre}, \dot{post}, \dot{loc})$ se dice *verificable* si el conjunto de obligaciones de prueba $\mathsf{po}(f, \mathsf{true}, \bar{\gamma_{\mathsf{a}}} \circ \dot{loc}, \mathsf{true}, \mathsf{true})$ es demostrable, para cada f en $\dot{p}$.

Lema 3.3. *Sea f en $\dot{p}$. Para cada etiqueta l en f:*

$$\mathsf{wpi}(f, l, \mathsf{annot}, \mathsf{post}, \chi) \wedge \bar{\gamma_{\mathsf{a}}}(\dot{loc}(f, l)) \Rightarrow \hat{\mathsf{wpi}}(f, l, \hat{\mathsf{annot}}, \mathsf{post}, \chi)$$

siempre que la relación $\models \subseteq \mathbb{D} \times Guard$ sea válida y el análisis $(Pre, Post, Loc)$ sea verificable.

La correctitud del VCgen po se sigue del siguiente resultado y de la correctitud del VCgen estándar $\hat{\mathsf{po}}$:

Proposición 3.4. *Sea f en $\dot{p}$. La demostrabilidad del conjunto de obligaciones de prueba $\hat{\mathsf{po}}(f, \mathsf{pre}, \hat{\mathsf{annot}}, \mathsf{post}, \chi)$ se sigue de la demostrabilidad de $\mathsf{po}(f, \mathsf{pre}, \mathsf{annot}, \mathsf{post}, \chi)$.*

Consideremos, por ejemplo, la secuencia de bajo nivel del Programa 3.2. Recordemos que annot está definida como $A[i-1] = A[0]$ y $A[|A|-1] = A[0]$ en k y k'' respectivamente. Sea definido $\hat{\mathsf{annot}}$ mediante el fortalecimiento de annot con el resultado del análisis, es decir, $\hat{\mathsf{annot}}(f, k) = \mathsf{annot}(f, k) \wedge A \neq null \wedge 0 \leq i \leq |A|$ (podemos dejar $\hat{\mathsf{annot}}(k'') = \mathsf{annot}(k'')$). Sea Ψ la precondición más débil computada por el VCgen no híbrido en la etiqueta k_1:

$$\begin{array}{l} A \neq null \wedge 0 \leq i < |A| \Rightarrow (\mathsf{upd}(A, i, A[0])[i+1-1] = A[0] \\ \qquad\qquad\qquad\qquad\qquad \wedge\ 0 \leq i+1 \leq |A|) \\ \wedge \neg(A \neq null \wedge 0 \leq i < |A|) \Rightarrow \mathsf{false} \end{array}$$

la cual, por el Lema 3.3 es implicada por el VCgen híbrido wp y el resultado del análisis, es decir, por

$$\mathsf{upd}(A, i, A[0])[i+1-1] = A[0] \wedge \boxed{A \neq null \wedge 0 \leq i < |A|}$$

Tal como se establece en la Proposición 3.4, si las obligaciones de prueba devueltas por el VCgen híbrido son válidas, y asumiendo que el análisis es verificable, tenemos que

$$\begin{array}{c} A[i-1] = A[0] \wedge \boxed{0 \leq i \leq |A|} \Rightarrow i < |A| \Rightarrow \\ \mathsf{upd}(A, i, A[0])[i+1-1] = A[0] \end{array}$$

y $0 \leq i \leq |A| \Rightarrow i < |A| \Rightarrow 0 \leq i < |A|$ son demostrables. Luego, se cumple que la condición de verificación

$$A[i-1] = A[0] \wedge A \neq null \wedge 0 \leq i \leq |A| \Rightarrow i < |A| \Rightarrow \Psi$$

devuelta por el VCgen estándar es demostrable.

Los resultados de esta sección establecen que los métodos híbridos de verificación pueden ser mapeados a métodos estándares de verificación. En el contexto de PCC, desearíamos establecer el resultado más fuerte de que los certificados híbridos pueden ser compilados a certificados estándares. De hecho, es posible probar dicho resultado utilizando el framework de [5]. Sin embargo, la compilación de certificados híbridos a certificados estándares requiere el uso de analizadores certificantes, que generan automáticamente pruebas lógicas de la correctitud de los resultados del análisis. Mientras que es posible evitar los métodos híbridos, por ejemplo para sustentarse en arquitecturas PCC estándares, éstos son beneficiosos tanto para el productor de código, porque reduce el número de obligaciones de prueba requeridas para certificar código, como para los consumidores de código, porque los certificados son más compactos y más eficientes de verificar. La traducción de certificados híbridos a estándares es interesante para escenarios de PCC en los cuales las obligaciones de prueba originales son generadas por un VCgen híbrido, pero en los cuales la Base Computacional Confiable destino no tiene soporte para certificados híbridos.

4

QuickSort: Un Caso de Estudio

En este capítulo presentamos un caso de estudio completo en el cual aplicamos los resultados obtenidos y mostramos las ventajas de métodos híbridos de verificación respecto a los métodos estándares. Comenzamos describiendo el programa elegido y los motivos de su elección y hacemos las aclaraciones pertinentes que facilitan la comprensión de lo expresado en este capítulo. A continuación presentamos un resultado para el análisis de alto nivel y explicamos cómo se obtiene. En la sección de verificación, damos la especificación del programa y mostramos los conjuntos completos de obligaciones de prueba, tanto los generados por el VCgen híbrido como los generados por el VCgen estándar. Por razones de espacio, omitimos el cálculo de dichos conjuntos. Finalizamos el capítulo estudiando las principales diferencias entre la utilización de ambos métodos.

4.1. Programa de Ejemplo

Tal como el título del capítulo lo explicita, el programa elegido para nuestro caso de estudio es QuickSort [18], un algoritmo de ordenación eficiente desarrollado por Charles Hoare en 1962. La elección de Quicksort se debe al uso intensivo que hace del arreglo que ordena y a la forma en que los índices son manejados. Si bien este algoritmo es muy conocido y la mayoría de los lenguajes lo proveen como parte integral de sus librerías, cuando se lo intenta programar suelen cometerse errores muy sutiles que dan lugar a comportamientos anómalos de sus ejecuciones. El cálculo incorrecto de los índices es, en general, el error más común. El Programa 4.1 muestra el código fuente QuickSort. En nuestro lenguaje de alto nivel pueden producirse dos tipos de

excepciones: accesos nulos o fuera de rango. Si bien el algoritmo maneja sólo un arreglo, dado que los arreglos son globales, la mayoría de las funciones asumen que el arreglo ha sido creado antes de ser llamadas. Por ello, tanto el análisis de nulidad como el numérico son interesantes, ya que pueden brindar información importante sobre la validez de los accesos a arreglos que se efectuan durante la ejecución del algoritmo y de este modo reducir considerablemente el tamaño de las obligaciones de prueba.

En lo que resta, asumimos la existencia de las funciones $random()$ y $rand(\mathtt{n}, \mathtt{m})$. Siendo sus respectivas pre y pos condiciones $Pre(random) : \top$, $Post(random) : \top$, $Pre(rand) : \mathtt{n} \leq \mathtt{m}$ y $Post(rand) : \mathtt{n}^{old} \leq res \leq \mathtt{m}^{old}$, es decir, $random()$ computa un entero aleatorio cualquiera y $rand(\mathtt{n}, \mathtt{m})$ computa un entero aleatorio entre m y n.

Programa 4.1 Código Fuente de QuickSort

```
main () {
    [ n←-1 ] k1;
    while ( [ n<0 ] k2) {
         [ n ← random() ] k3
    };

    [ vec ← newarray(n) ] k4;
    [ i ← 0 ] k5;
    while ( [ i < n ] k6) {
        [ x ← random() ] k7;
        [ vec[i] ← x ] k8;
        [ i ← i+1 ] k9
    };

    [ quicksort(0, n) ] k10;
    [ return 0 ] k11
}

getPivot(low, hi) {
    [ x ← rand(low, hi-1) ] k12;
    [ return x ] k13
}
```

Programa 4.1 Código Fuente de QuickSort

```
swap(i, j) {
    [ t ← vec[i] ] k14;
    [ vec[i] ← vec[j] ] k15;
    [ vec[j] ← t ] k16;
    [ return 0 ] k17
}

split(low, hi) {
     [ piv ← getPivot(low, hi) ] k18;
     [ x ← vec[piv] ] k19;
     [ sIdx ← low ] k20;
     [ i ← low ] k21;
     [ swap(piv, hi - 1 ) ] k22;

      while ( [ i < hi - 1 ] k23 ) {
          if ( [ vec[i] ≤ x ] k24 ) {
               [ swap(sIdx, i) ] k25;
               [ sIdx ← sIdx + 1 ] k26;
           } else {
              [ skip ] k27
           };
          [ i ← i + 1 ] k28;
      };
     [ swap(sIdx, hi-1) ] k29;
     [ return sIdx ] k30;
}

quicksort (low, hi) {
    if ( [ hi - low ≥ 1 ] k31) {
        [ sp ← split(low, hi) ] k32;
        [ quicksort(low, sp) ] k33;
        [ quicksort(sp+1, hi) ] k34;
    }
     [ return 0 ] k35
}
```

4.2. Análisis de Código

El análisis que realizamos sobre el código fuente es un análisis interprocedural que combina dos tipos de análisis: uno numérico relacional y otro de nulidad. Es interprocedural porque el proceso no se limita a analizar cada función de manera aislada, sino que trata al programa como un conjunto, permitiendo obtener mayor información acerca de las variables y, por lo tanto, aumentar la precisión del análisis.

Para el análisis numérico, instanciamos la interfaz de dominio abstracto usando relaciones lineales en la forma de poliedros convexos. El análisis de programas mediante poliedros tiene una teoría muy bien establecida [13] con varias implementaciones [1, 23]. Utilizando el analizador Interproc [20], una herramienta basada en la librería de dominios numéricos abstractos APRON [23], realizamos un análisis hacia adelante (forward analysis) con iteraciones guiadas. Dado que Interproc no soporta arreglos, el programa debe ser modificado pertinentemente a fin de obtener los resultados deseados. Las modificaciones incluyen el reemplazo de cada instrucción que contiene expresiones con arreglos por una instrucción equivalente —en términos del análisis— que no contenga variables de arreglo y el pasaje de argumentos extras a las funciones, que representan longitudes de arreglos que éstas utilizan. Por ejemplo, la sentencia `x:=vec[0]` debe ser reemplazada por la sentencia equivalente `x:=random()`.

El análisis de nulidad está definido por el retículo completo $(\mathcal{P}(V_a), \supseteq, V_s, \emptyset, \cup, \cap)$. De este modo, un elemento abstracto está representado por un conjunto de variables de arreglo no nulas. El análisis es seguro en el sentido de que no puede inferirse información errónea de sus resultados. Aunque se sabe que el proceso de analizar con precisión la nulidad en tiempo de compilación es indecidible, existen analizadores que, mediante la evaluación de expresiones, brindan resultados aproximados. No es el caso de este análisis y por tanto existen casos en los que la precisión queda relegada en pos de garantizar la veracidad de los resultados. Por ejemplo, si consideramos el siguiente fragmento de código,

$\ldots$

`if` $[1 > 0]^k$ `then` $[$`a:=newarray(10)`$]^{k_1}$ `else` $[$`b:=newarray(10)`$]^{k_2}$;

$[\mathtt{a}[0]{:=}x]^{k'}$

$\ldots$

en la etiqueta k', el análisis sólo brinda garantías de la no nulidad de a si se sabe que antes de la entrada al condicional —es decir, en la etiqueta k— la variable a es no nula. De lo contrario,

$$
\begin{aligned}
Pre(main) &= \top \\
Post(main) &= (\mathsf{res} = 0 \bullet \{\texttt{vec}\}) \\
Loc(main, k_1) &= \top \\
Loc(main, k_2) &= \top \\
Loc(main, k_3) &= (-\texttt{n} \geq 1 \bullet \emptyset) \\
Loc(main, k_4) &= (\texttt{n} \geq 0 \bullet \emptyset) \\
Loc(main, k_5) &= (\texttt{n} \geq 0 \wedge \texttt{n} = |\texttt{vec}| \bullet \{\texttt{vec}\}) \\
Loc(main, k_6) &= (-\texttt{i} + \texttt{n} \geq 0 \wedge \texttt{i} \geq 0 \wedge \texttt{n} = |\texttt{vec}| \bullet \{\texttt{vec}\}) \\
Loc(main, k_7) &= (-\texttt{i} + \texttt{n} \geq 1 \wedge \texttt{i} \geq 0 \wedge \texttt{n} = |\texttt{vec}| \bullet \{\texttt{vec}\}) \\
Loc(main, k_8) &= Loc(main, k_7) \\
Loc(main, k_9) &= Loc(main, k_7) \\
Loc(main, k_{10}) &= (-\texttt{i} + \texttt{n} = 0 \wedge \texttt{i} \geq 0 \wedge \texttt{n} = |\texttt{vec}| \bullet \{\texttt{vec}\}) \\
Loc(main, k_{11}) &= Loc(main, k_{10})
\end{aligned}
$$

Figura 4.1: Análisis de Alto Nivel. Resultado para *main*

al no realizar evaluación de expresiones, el resultado del análisis no garantiza la no nulidad de a. Claramente, un analizador más inteligente podría inferir dicha condición.

El resultado del análisis es el producto directo de los elementos abstractos, obtenidos de los análisis, correspondientes a cada punto del programa. El mismo se muestra en las Figuras 4.1, 4.2, 4.3, 4.4 y 4.5, donde se detallan las precondiciones, las poscondiciones y el mapping *Loc*. Cada par abstracto se compone del valor numérico abstracto (poliedro, o sistema de restricciones) y del valor abstracto de nulidad (conjunto de variables de arreglo no nulas).

$$
\begin{aligned}
Pre(getPivot) &= (-\texttt{hi} \geq -|\texttt{vec}| \wedge \texttt{low} \geq 0 \wedge \texttt{hi} - \texttt{low} \geq 2 \bullet \{\texttt{vec}\}) \\
Post(getPivot) &= (-\texttt{hi} \geq -|\texttt{vec}| \wedge -\texttt{low} + \mathsf{res} \geq 0 \wedge \texttt{low} \geq 0 \wedge \texttt{hi} - \texttt{low} \geq 2 \\
&\qquad \wedge \texttt{hi} - \mathsf{res} \geq 1 \bullet \{\texttt{vec}\}) \\
Loc(getPivot, k_{12}) &= (-\texttt{hi} \geq -|\texttt{vec}| \wedge \texttt{low} \geq 0 \wedge \texttt{hi} - \texttt{low} \geq 2 \bullet \{\texttt{vec}\}) \\
Loc(getPivot, k_{13}) &= (-\texttt{hi} \geq -|\texttt{vec}| \wedge -\texttt{low} + \texttt{x} \geq 0 \wedge \texttt{low} \geq 0 \wedge \texttt{hi} - \texttt{low} \geq 2 \\
&\qquad \wedge \texttt{hi} - \texttt{x} \geq 1 \bullet \{\texttt{vec}\})
\end{aligned}
$$

Figura 4.2: Análisis de Alto Nivel. Resultado para *getPivot*

$$
\begin{aligned}
\mathit{Pre}(\mathit{swap}) &= (-\mathtt{i} + \mathtt{j} \geq 0 \wedge -\mathtt{j} \geq -|\mathtt{vec}| + 1 \wedge \mathtt{i} \geq 0 \bullet \{\mathtt{vec}\}) \\
\mathit{Post}(\mathit{swap}) &= (-\mathtt{i} + \mathtt{j} \geq 0 \wedge -\mathtt{j} \geq -|\mathtt{vec}| + 1 \wedge \mathtt{i} \geq 0 \wedge \mathsf{res} = 0 \bullet \{\mathtt{vec}\}) \\
\mathit{Loc}(\mathit{swap}, k_{14}) &= (-\mathtt{i} + \mathtt{j} \geq 0 \wedge -\mathtt{j} \geq -|\mathtt{vec}| + 1 \wedge \mathtt{i} \geq 0 \bullet \{\mathtt{vec}\}) \\
\mathit{Loc}(\mathit{swap}, k_{15}) &= \mathit{Loc}(\mathit{swap}, k_{14}) \\
\mathit{Loc}(\mathit{swap}, k_{16}) &= \mathit{Loc}(\mathit{swap}, k_{14}) \\
\mathit{Loc}(\mathit{swap}, k_{17}) &= \mathit{Loc}(\mathit{swap}, k_{14})
\end{aligned}
$$

Figura 4.3: Análisis de Alto Nivel. Resultado para *swap*

$$
\begin{aligned}
\mathit{Pre}(\mathit{split}) &= (-\mathtt{hi} \geq -|\mathtt{vec}| \wedge \mathtt{low} \geq 0 \wedge \mathtt{hi} - \mathtt{low} \geq 2 \bullet \{\mathtt{vec}\}) \\
\mathit{Post}(\mathit{split}) &= (-\mathtt{hi} \geq -|\mathtt{vec}| \wedge -\mathtt{low} + \mathsf{res} \geq 0 \wedge \mathtt{low} \geq 0 \wedge \mathtt{hi} - \mathtt{low} \geq 2 \wedge \mathtt{hi} - \mathsf{res} \geq 1 \bullet \{\mathtt{vec}\}) \\
\mathit{Loc}(\mathit{split}, k_{18}) &= (-\mathtt{hi} \geq -|\mathtt{vec}| \wedge \mathtt{low} \geq 0 \wedge \mathtt{hi} - \mathtt{low} \geq 2 \bullet \{\mathtt{vec}\}) \\
\mathit{Loc}(\mathit{split}, k_{19}) &= (-\mathtt{hi} \geq -|\mathtt{vec}| \wedge \mathtt{piv} - \mathtt{low} \geq 0 \wedge \mathtt{low} \geq 0 \wedge \mathtt{hi} - \mathtt{low} \geq 2 \wedge \mathtt{hi} - \mathtt{piv} \geq 1 \bullet \{\mathtt{vec}\}) \\
\mathit{Loc}(\mathit{split}, k_{20}) &= (-\mathtt{hi} \geq -|\mathtt{vec}| \wedge \mathtt{piv} - \mathtt{low} \geq 0 \wedge \mathtt{low} \geq 0 \wedge \mathtt{hi} - \mathtt{low} \geq 2 \bullet \{\mathtt{vec}\}) \\
\mathit{Loc}(\mathit{split}, k_{21}) &= (-\mathtt{low} + \mathtt{sIdx} = 0 \wedge -\mathtt{hi} \geq -|\mathtt{vec}| \wedge \mathtt{piv} - \mathtt{low} \geq 0 \wedge \mathtt{low} \geq 0 \wedge \mathtt{hi} - \mathtt{low} \geq 2 \\
&\qquad \wedge \mathtt{hi} - \mathtt{piv} \geq 1 \bullet \{\mathtt{vec}\}) \\
\mathit{Loc}(\mathit{split}, k_{22}) &= (-\mathtt{i} + \mathtt{sIdx} \geq 0 \wedge -\mathtt{i} + \mathtt{low} = 0 \wedge -\mathtt{hi} \geq -|\mathtt{vec}| \wedge \mathtt{piv} - \mathtt{low} \geq 0 \wedge \mathtt{low} \geq 0 \\
&\qquad \wedge \mathtt{hi} - \mathtt{low} \geq 2 \wedge \mathtt{hi} - \mathtt{piv} \geq 1 \bullet \{\mathtt{vec}\}) \\
\mathit{Loc}(\mathit{split}, k_{23}) &= (-\mathtt{hi} \geq -|\mathtt{vec}| \wedge -\mathtt{low} + \mathtt{sIdx} \geq 0 \wedge \mathtt{piv} - \mathtt{low} \geq 0 \wedge \mathtt{low} \geq 0 \wedge \mathtt{i} - \mathtt{sIdx} \geq 0 \\
&\qquad \wedge \mathtt{hi} - \mathtt{i} \geq 1 \wedge \mathtt{hi} - \mathtt{low} \geq 2 \wedge \mathtt{hi} - \mathtt{piv} \geq 1 \bullet \{\mathtt{vec}\}) \\
\mathit{Loc}(\mathit{split}, k_{24}) &= (-\mathtt{hi} \geq -|\mathtt{vec}| \wedge -\mathtt{low} + \mathtt{sIdx} \geq 0 \wedge -\mathtt{hi} + \mathtt{piv} \geq 0 \wedge \mathtt{low} \geq 0 \wedge \mathtt{i} - \mathtt{sIdx} \geq 0 \\
&\qquad \wedge \mathtt{hi} - \mathtt{i} \geq 2 \wedge \mathtt{hi} - \mathtt{piv} \geq 1 \bullet \{\mathtt{vec}\}) \\
\mathit{Loc}(\mathit{split}, k_{25}) &= \mathit{Loc}(\mathit{split}, k_{24}) \\
\mathit{Loc}(\mathit{split}, k_{26}) &= \mathit{Loc}(\mathit{split}, k_{24}) \\
\mathit{Loc}(\mathit{split}, k_{27}) &= \mathit{Loc}(\mathit{split}, k_{24}) \\
\mathit{Loc}(\mathit{split}, k_{28}) &= (-\mathtt{hi} \geq -|\mathtt{vec}| \wedge -\mathtt{low} + \mathtt{sIdx} \geq 0 \wedge \mathtt{piv} - \mathtt{low} \geq 0 \wedge \mathtt{low} \geq 0 \wedge \mathtt{i} - \mathtt{low} \geq 0 \\
&\qquad \wedge \mathtt{i} - \mathtt{sIdx} \geq -1 \wedge \mathtt{hi} - \mathtt{i} \geq -2 \wedge \mathtt{hi} - \mathtt{piv} \geq 1 \bullet \{\mathtt{vec}\}) \\
\mathit{Loc}(\mathit{split}, k_{29}) &= (-\mathtt{hi} + \mathtt{i} = -1 \wedge -\mathtt{hi} \geq -|\mathtt{vec}| \wedge -\mathtt{low} + \mathtt{sIdx} \geq 0 \wedge -\mathtt{hi} + \mathtt{piv} \geq 0 \wedge \mathtt{low} \geq 0 \\
&\qquad \wedge \mathtt{hi} - \mathtt{low} \geq 2 \wedge \mathtt{hi} - \mathtt{piv} \geq 1 \wedge \mathtt{hi} - \mathtt{sIdx} \geq 1 \bullet \{\mathtt{vec}\}) \\
\mathit{Loc}(\mathit{split}, k_{30}) &= \mathit{Loc}(\mathit{split}, k_{29})
\end{aligned}
$$

Figura 4.4: Análisis de Alto Nivel. Resultado para *split*

$$
\begin{aligned}
Pre(quicksort) &= (-\texttt{hi} \geq -|\texttt{vec}| \wedge \texttt{low} \geq 0 \wedge \texttt{hi} - \texttt{low} \geq 0 \bullet \{\texttt{vec}\}) \\
Post(quicksort) &= (-\texttt{hi} \geq -|\texttt{vec}| \wedge \texttt{low} \geq 0 \wedge \texttt{hi} - \texttt{low} \geq 2 \wedge \mathsf{res} = 0 \bullet \{\texttt{vec}\}) \\
Loc(quicksort, k_{31}) &= (-\texttt{hi} \geq -|\texttt{vec}| \wedge \texttt{low} \geq 0 \wedge \texttt{hi} - \texttt{low} \geq 0 \bullet \{\texttt{vec}\}) \\
Loc(quicksort, k_{32}) &= Loc(quicksort, k_{31}) \\
Loc(quicksort, k_{33}) &= (-\texttt{hi} \geq -|\texttt{vec}| \wedge \texttt{sp} - \texttt{low} \geq 0 \wedge \texttt{low} \geq 0 \wedge \texttt{hi} - \texttt{low} \geq 2 \\
&\qquad \wedge \texttt{hi} - \texttt{sp} \geq 1 \bullet \{\texttt{vec}\}) \\
Loc(quicksort, k_{34}) &= Loc(quicksort, k_{33}) \\
Loc(quicksort, k_{35}) &= Loc(quicksort, k_{33})
\end{aligned}
$$

Figura 4.5: Análisis de Alto Nivel. Resultado para *quicksort*

4.3. Generación de Condiciones de Verificación

En esta sección nos proponemos mostrar las principales ventajas de utilizar los métodos híbirdos para la generación de obligaciones de pruebas. Para ello, consideremos las siguientes especificaciones de QuickSort. La primera la utilizamos en el VCgen híbrido y la segunda en el VCgen estándar.

$$
\begin{aligned}
\mathsf{pre}(main) &= \mathsf{true} \\
\mathsf{post}(main) &= \mathsf{true} \\
\mathsf{pre}(getPivot) &= \mathsf{true} \\
\mathsf{post}(getPivot) &= \mathsf{true} \\
\mathsf{pre}(swap) &= \mathsf{true} \\
\mathsf{post}(swap) &= \mathsf{swapped}(\texttt{i}^{old}, \texttt{j}^{old}) \\
\mathsf{pre}(split) &= \mathsf{true} \\
\mathsf{post}(split) &= \mathsf{partition}(\mathsf{res}, \texttt{low}^{old}, \texttt{hi}^{old}) \wedge \mathsf{perm}(\texttt{low}^{old}, \texttt{hi}^{old}) \\
\mathsf{pre}(quicksort) &= \mathsf{true} \\
\mathsf{post}(quicksort) &= \mathsf{sorted}(\texttt{low}^{old}, \texttt{hi}^{old}) \wedge \mathsf{perm}(\texttt{low}^{old}, \texttt{hi}^{old})
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{annot}(main, k_2) &= \mathsf{true} \\
\mathsf{annot}(main, k_6) &= \mathsf{true} \\
\mathsf{annot}(split, k_{23}) &= \mathsf{smaller}(\texttt{x}, \texttt{low}, \texttt{xIdx}) \\
\chi(f) &= \mathsf{false}\ \forall f \in QuickSort
\end{aligned}
$$

$$
\begin{aligned}
\hat{\mathsf{pre}}(main) &= \mathsf{true} \\
\hat{\mathsf{post}}(main) &= \mathsf{true} \\
\hat{\mathsf{pre}}(getPivot) &= \mathtt{low} < \mathtt{hi} \\
\hat{\mathsf{post}}(getPivot) &= \mathtt{low}^{old} \leq \mathsf{res} \leq \mathtt{hi}^{old} \\
\hat{\mathsf{pre}}(swap) &= \mathtt{vec} \neq \mathit{null} \wedge 0 \leq \mathtt{i}, \mathtt{j} < |\mathtt{vec}| \\
\hat{\mathsf{post}}(swap) &= \mathsf{swapped}(\mathtt{i}^{old}, \mathtt{j}^{old}) \\
\hat{\mathsf{pre}}(split) &= \mathtt{vec} \neq \mathit{null} \wedge 0 \leq \mathtt{low} \leq \mathtt{hi} \leq |\mathtt{vec}| \\
\hat{\mathsf{post}}(split) &= \mathsf{partition}(\mathsf{res}, \mathtt{low}^{old}, \mathtt{hi}^{old}) \wedge \mathsf{perm}(\mathtt{low}^{old}, \mathtt{hi}^{old}) \\
\hat{\mathsf{pre}}(quicksort) &= \mathtt{vec} \neq \mathit{null} \wedge 0 \leq \mathtt{low} \leq \mathtt{hi} \leq |\mathtt{vec}| \\
\hat{\mathsf{post}}(quicksort) &= \mathsf{sorted}(\mathtt{low}^{old}, \mathtt{hi}^{old}) \wedge \mathsf{perm}(\mathtt{low}^{old}, \mathtt{hi}^{old}) \\
\\
\hat{\mathsf{annot}}(main, k_2) &= \mathsf{true} \\
\hat{\mathsf{annot}}(main, k_6) &= \mathtt{vec} \neq \mathit{null} \wedge 0 \leq \mathtt{i} \leq |\mathtt{vec}| \\
\hat{\mathsf{annot}}(split, k_{23}) &= \mathtt{vec} \neq \mathit{null} \wedge 0 \leq \mathtt{low} < \mathtt{hi} - 1 \leq |\mathtt{vec}| \\
&\quad \wedge \mathtt{low} \leq \mathtt{sIdx} \leq \mathtt{i} < \mathtt{hi} \wedge \mathtt{low} \leq \mathtt{piv} < \mathtt{hi} \\
&\quad \wedge \mathsf{smaller}(\mathtt{x}, \mathtt{low}, \mathtt{xIdx}) \\
\hat{\chi}(f) &= \mathsf{false}\ \forall f \in \mathit{QuickSort}
\end{aligned}
$$

donde

$$
\begin{aligned}
\mathsf{smaller}(x, i, j) &\doteq \forall k \in \mathbb{N} \bullet\ (i \leq k < j \Rightarrow \mathtt{vec}[k] \leq x) \\
\mathsf{partition}(x, i, j) &\doteq \forall k \in \mathbb{N} \bullet\ (i \leq k \leq x \Rightarrow \mathtt{vec}[k] \leq \mathtt{vec}[x]) \wedge \\
&\quad (x < k < j \Rightarrow \mathtt{vec}[x] < \mathtt{vec}[k]) \\
\mathsf{perm}(i, j) &\doteq \forall n \in \mathbb{Z} \bullet\ \sharp\{i \leq k < j \mid \mathtt{vec}[k] = n\} = \sharp\{i \leq k < j \mid \mathtt{vec}^{old}[k] = n\} \\
\mathsf{swapped}(i, j) &\doteq \mathtt{vec}[i] = \mathtt{vec}^{old}[j] \wedge \mathtt{vec}[j] = \mathtt{vec}^{old}[i] \wedge \\
&\quad \forall k \in \mathbb{N} \bullet\ (i \neq k \neq j \Rightarrow \mathtt{vec}[k] = \mathtt{vec}^{old}[k]) \\
\mathsf{sorted}(i, j) &\doteq \forall k, k' \bullet\ (i \leq k \leq k' < j \Rightarrow \mathtt{vec}[k] \leq \mathtt{vec}[k'])
\end{aligned}
$$

Como puede observarse, ambas especificaciones difieren en las precondiciones y los invariantes. Esto se debe a que el VCgen híbrido requiere precondiciones e invariantes más débiles que el VCgen estándar dado que utiliza el resultado del análisis para inferir el resto de las condiciones. Por supuesto que si el analizador no es capaz de obtener todas las condiciones faltantes, el descarte de ciertas obligaciones de prueba no puede ser llevado a cabo. Pero si se toman los recaudos necesarios, el tamaño de la especificación se ve reducido considerablemente.

A continuación presentamos las obligaciones de prueba generadas por el VCgen híbrido. Las aserciones recuadradas son extraidas del análisis e insertadas por el VCgen para facilitar

el descarte interactivo de las condiciones de verificación.

$$\mathsf{PO}(main) =$$

$$\{\boxed{\mathtt{vec} \neq null \wedge 0 \leq \mathtt{i} \leq \mathtt{n} \wedge n = |\mathsf{vec}|} \Rightarrow$$

$$(\mathtt{i} < \mathtt{n} \Rightarrow \forall \mathsf{res}, \mathtt{vec} \bullet 0 \leq \mathtt{i} \leq \mathtt{n} \leq |\mathsf{vec}|') \wedge (\neg \mathtt{i} < \mathtt{n} \Rightarrow 0 \leq \mathtt{i} \leq \mathtt{n} \leq |\mathsf{vec}|)\}$$

$$\mathsf{PO}(getPivot) =$$

$$\{\boxed{\mathtt{vec} \neq null \wedge \mathtt{low} < \mathtt{hi} - 1} \Rightarrow \mathtt{low} \leq \mathtt{hi}\}$$

$$\mathsf{PO}(swap) =$$

$$\{\boxed{\mathtt{vec} \neq null \wedge 0 \leq \mathtt{i} \leq \mathtt{j} \leq |\mathsf{vec}|} \Rightarrow$$

$$\mathsf{swapped}(\mathtt{i}^{old}, \mathtt{j}^{old})[^{\mathtt{upd(vec,j,t)}}/_{\mathtt{vec}}][^{\mathtt{upd(vec,i,vec[j])}}/_{\mathtt{vec}}][^{\mathtt{vec[i]}}/_{\mathtt{t}}][^{\mathtt{vec}}/_{\mathtt{vec}^{old}}]\}$$

$$\mathsf{PO}(split) =$$

$$\{\boxed{\mathtt{vec} \neq null \wedge 0 \leq \mathtt{low} < \mathtt{hi} - 1 < |\mathsf{vec}|} \Rightarrow$$

$$\forall \mathsf{res} \bullet (\forall \mathsf{res}, \mathtt{vec} \bullet \mathsf{swapped}(\mathtt{piv}, \mathtt{hi} - 1)[^{\mathtt{vec}', \mathtt{vec}}/_{\mathtt{vec}, \mathtt{vec}^{old}}] \Rightarrow$$

$$\mathsf{smaller}(\mathtt{vec}[\mathtt{piv}], low, low)[^{\mathtt{vec}'}/_{\mathtt{vec}}])[^{\mathsf{res}}/_{piv}][^{i,j,vec}/_{\mathtt{i}^{old}, \mathtt{j}^{old}, \mathtt{vec}^{old}}]\}$$

$$\cup$$

$$\{\mathsf{smaller}(\mathtt{x}, \mathtt{low}, \mathtt{sIdx}) \wedge$$

$$\boxed{\mathtt{vec} \neq null \wedge 0 \leq \mathtt{low} < \mathtt{hi} - 1 < |\mathsf{vec}| \wedge \mathtt{low} \leq \mathtt{sIdx} \leq \mathtt{i} < \mathtt{hi} \wedge \mathtt{low} \leq \mathtt{piv} < \mathtt{hi}} \Rightarrow$$

$$(\mathtt{i} < \mathtt{hi} - 1$$

$$(\mathtt{vec}[\mathtt{i}] \leq \mathtt{x} \Rightarrow$$

$$\forall \mathsf{res}, \mathtt{vec} \bullet \mathsf{swapped}(\mathtt{sIdx}, \mathtt{i})[^{\mathtt{vec}', \mathtt{vec}}/_{\mathtt{vec}, \mathtt{vec}^{old}}] \Rightarrow$$

$$(\forall \mathsf{res}, \mathtt{vec} \bullet \mathsf{swapped}(\mathtt{sIdx}, \mathtt{hi} - 1)[^{\mathtt{vec}', \mathtt{vec}}/_{\mathtt{vec}, \mathtt{vec}^{old}}] \Rightarrow$$

$$(\mathsf{partition}(\mathtt{sIdx}, \mathtt{low}, \mathtt{hi}) \wedge \mathsf{perm}(\mathtt{low}, \mathtt{hi}))[^{\mathtt{vec}'}/_{\mathtt{vec}}])[^{\mathtt{sIdx}}/_{\mathtt{sIdx}+1}][^{\mathtt{vec}'}/_{\mathtt{vec}}])$$

$$(\neg \mathtt{vec}[\mathtt{i}] \leq \mathtt{x} \Rightarrow$$

$$\forall \mathsf{res}, \mathtt{vec} \bullet \mathsf{swapped}(\mathtt{sIdx}, \mathtt{i})[^{\mathtt{vec}', \mathtt{vec}}/_{\mathtt{vec}, \mathtt{vec}^{old}}] \Rightarrow$$

$$(\mathsf{partition}(\mathtt{sIdx}, \mathtt{low}, \mathtt{hi}) \wedge \mathsf{perm}(\mathtt{low}, \mathtt{hi}))[^{\mathtt{vec}'}/_{\mathtt{vec}}]))$$

$$\wedge (\neg \mathtt{i} < \mathtt{hi} - 1 \Rightarrow \forall \mathsf{res}, \mathtt{vec} \bullet \mathsf{swapped}(\mathtt{sIdx}, \mathtt{hi} - 1)[^{\mathtt{vec}', \mathtt{vec}}/_{\mathtt{vec}, \mathtt{vec}^{old}}] \Rightarrow$$

$$(\mathsf{partition}(\mathtt{sIdx}, \mathtt{low}, \mathtt{hi}) \wedge \mathsf{perm}(\mathtt{low}, \mathtt{hi}))[^{\mathtt{vec}'}/_{\mathtt{vec}}])\}$$

$$\mathsf{PO}(quicksort) =$$

$$\{\boxed{\mathtt{vec} \neq null \wedge 0 \leq \mathtt{low} \leq \mathtt{hi} \leq |\mathsf{vec}|} \Rightarrow$$

$$((\mathtt{low} < \mathtt{hi} \Rightarrow$$

$$\forall \mathsf{res}, \mathtt{vec} \bullet (\mathsf{partition}(\mathsf{res}, \mathtt{low}, \mathtt{hi}) \wedge \mathsf{perm}(\mathtt{low}, \mathtt{hi}))[^{\mathtt{vec}', \mathtt{vec}}/_{\mathtt{vec}, \mathtt{vec}^{old}}] \Rightarrow$$

$$(\forall \mathsf{res}, \mathtt{vec} \bullet (\mathsf{sorted}(\mathtt{low}, \mathtt{sp}) \wedge \mathsf{perm}(\mathtt{low}, \mathtt{sp}))[^{\mathtt{vec}', \mathtt{vec}}/_{\mathtt{vec}, \mathtt{vec}^{old}}] \Rightarrow$$

$$(\forall \mathsf{res}, \mathtt{vec} \bullet (\mathsf{sorted}(\mathtt{sp} + 1, \mathtt{hi}) \wedge \mathsf{perm}(\mathtt{sp} + 1, \mathtt{hi}))[^{\mathtt{vec}', \mathtt{vec}}/_{\mathtt{vec}, \mathtt{vec}^{old}}] \Rightarrow$$

$$(\mathsf{sorted}(\mathtt{low}^{old}, \mathtt{hi}^{old}) \wedge \mathsf{perm}(\mathtt{low}^{old}, \mathtt{hi}^{old}))[^{\mathtt{vec}'}/_{\mathtt{vec}}])[^{\mathtt{vec}'}/_{\mathtt{vec}}])[^{\mathtt{vec}'}/_{\mathtt{vec}}][^{\mathsf{res}}/_{\mathtt{sp}}])$$

$$\wedge (\neg \mathtt{low} < \mathtt{hi} \Rightarrow \mathsf{sorted}(\mathtt{low}^{old}, \mathtt{hi}^{old}) \wedge \mathsf{perm}(\mathtt{low}^{old}, \mathtt{hi}^{old})))[^{\mathtt{low}, \mathtt{hi}, \mathtt{vec}}/_{\mathtt{low}^{old}, \mathtt{hi}^{old}, \mathtt{vec}^{old}}]\}$$

Las obligaciones de prueba generadas por el VCgen estándar son las siguientes:

$$
\begin{array}{l}
\mathsf{PO}(main) = \\
\quad \{\hat{\mathsf{annot}}(main, k_6) \Rightarrow \\
\quad\quad (\mathtt{i} < \mathtt{n} \Rightarrow \forall \mathsf{res}, \mathtt{vec} \bullet (0 \le \mathtt{in} \Rightarrow 0 \le \mathtt{i} \le \mathtt{n} \le |\mathtt{vec}'|) \\
\quad\quad\quad \wedge(\neg \mathtt{i} < \mathtt{n} \Rightarrow \mathsf{false})) \\
\quad\quad \wedge(\neg \mathtt{i} < \mathtt{n} \Rightarrow 0 \le \mathtt{i} \le \mathtt{n} \le |\mathtt{vec}|)\} \\
\mathsf{PO}(getPivot) = \\
\quad \{\hat{\mathsf{pre}}(getPivot) \Rightarrow \forall \mathsf{res} \bullet \mathtt{low} \le \mathsf{res} \le \mathtt{hi} \Rightarrow \mathtt{low} \le \mathsf{res} \le \mathtt{hi}\}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{PO}(swap) = \\
\quad \{\hat{\mathsf{pre}}(swap) \Rightarrow \\
\quad\quad (0 \le \mathtt{i} < |\mathtt{vec}| \wedge 0 \le \mathtt{i} < |\mathtt{vec}| \Rightarrow \\
\quad\quad\quad \mathsf{swapped}(\mathtt{i}^{old}, \mathtt{j}^{old})[^{\mathtt{upd(vec,j,t)}}/_{\mathtt{vec}}][^{\mathtt{upd(vec,i,vec[j])}}/_{\mathtt{vec}}][^{\mathtt{vec[i]}}/_{\mathtt{t}}][^{\mathtt{vec}}/_{\mathtt{vec}^{old}}]) \\
\quad\quad (\neg 0 \le \mathtt{i} < |\mathtt{vec}| \wedge 0 \le \mathtt{i} < |\mathtt{vec}| \Rightarrow \mathsf{false})\} \\
\mathsf{PO}(split) = \\
\quad \{\hat{\mathsf{pre}}(split) \Rightarrow \mathtt{low} < \mathtt{hi} \wedge \forall \mathsf{res} \bullet \mathtt{low} \le \mathsf{res} \le \mathtt{hi} \Rightarrow \\
\quad\quad ((\mathtt{vec} \neq null \wedge 0 \le \mathtt{piv} < |\mathtt{vec}| \Rightarrow \\
\quad\quad\quad (\mathtt{vec} \neq null \wedge 0 \le \mathtt{i}, \mathtt{j} < |\mathtt{vec}| \\
\quad\quad\quad \wedge \forall \mathsf{res}, \mathtt{vec} \bullet \mathsf{swapped}(\mathtt{piv}, \mathtt{hi} - 1)[^{\mathtt{vec}', \mathtt{vec}}/_{\mathtt{vec}, \mathtt{vec}^{old}}] \Rightarrow \\
\quad\quad\quad\quad \hat{\mathsf{annot}}(split, k_{23})[^{\mathtt{vec}'}/_{\mathtt{vec}}])[^{\mathtt{low}}/_{\mathtt{i}}][^{\mathtt{low}}/_{sIdx}][^{\mathtt{vec[piv]}}/_{x}]) \\
\quad\quad \wedge(\neg \mathtt{vec} \neq null \wedge 0 \le \mathtt{piv} < |\mathtt{vec}| \Rightarrow \mathsf{false})[^{\mathsf{res}}/_{\mathtt{piv}}])[^{\mathtt{vec}}/_{\mathtt{vec}^{old}}]\} \\
\quad \cup \\
\quad \{\hat{\mathsf{annot}}(split, k_{23}) \Rightarrow \\
\quad\quad (\mathtt{i} < \mathtt{hi} - 1 \\
\quad\quad\quad (\mathtt{vec} \neq null \wedge 0 \le \mathtt{i} < |\mathtt{vec}| \Rightarrow \\
\quad\quad\quad\quad (\mathtt{vec[i]} \le \mathtt{x} \Rightarrow 0 \le \mathtt{sIdx}, \mathtt{i} < |\mathtt{vec}| \\
\quad\quad\quad\quad\quad \wedge \forall \mathsf{res}, \mathtt{vec} \bullet \mathsf{swapped}(\mathtt{sIdx}, \mathtt{i})[^{\mathtt{vec}', \mathtt{vec}}/_{\mathtt{vec}, \mathtt{vec}^{old}}] \Rightarrow \\
\quad\quad\quad\quad\quad\quad (0 \le \mathtt{sIdx}, \mathtt{hi} - 1 < |\mathtt{vec}| \\
\quad\quad\quad\quad\quad\quad \wedge \forall \mathsf{res}, \mathtt{vec} \bullet \mathsf{swapped}(\mathtt{sIdx}, \mathtt{hi} - 1)[^{\mathtt{vec}', \mathtt{vec}}/_{\mathtt{vec}, \mathtt{vec}^{old}}] \Rightarrow \\
\quad\quad\quad\quad\quad\quad\quad (\mathsf{partition}(\mathtt{sIdx}, \mathtt{low}, \mathtt{hi}) \wedge \mathsf{perm}(\mathtt{low}, \mathtt{hi}))[^{\mathtt{vec}'}/_{\mathtt{vec}}])[^{\mathtt{sIdx}}/_{\mathtt{sIdx}+1}][^{\mathtt{vec}'}/_{\mathtt{vec}}]) \\
\quad\quad\quad\quad (\neg \mathtt{vec[i]} \le \mathtt{x} \Rightarrow 0 \le \mathtt{sIdx}, \mathtt{hi} - 1 < |\mathtt{vec}| \\
\quad\quad\quad\quad\quad \wedge \forall \mathsf{res}, \mathtt{vec} \bullet \mathsf{swapped}(\mathtt{sIdx}, \mathtt{i})[^{\mathtt{vec}', \mathtt{vec}}/_{\mathtt{vec}, \mathtt{vec}^{old}}] \Rightarrow \\
\quad\quad\quad\quad\quad\quad (\mathsf{partition}(\mathtt{sIdx}, \mathtt{low}, \mathtt{hi}) \wedge \mathsf{perm}(\mathtt{low}, \mathtt{hi}))[^{\mathtt{vec}'}/_{\mathtt{vec}}])) \\
\quad\quad\quad (\neg(\mathtt{vec} \neq null \wedge 0 \le i < |\mathtt{vec}|) \Rightarrow \mathsf{false})) \\
\quad\quad \wedge(\neg \mathtt{i} < \mathtt{hi} - 1 \Rightarrow \forall \mathsf{res}, \mathtt{vec} \bullet \mathsf{swapped}(\mathtt{sIdx}, \mathtt{hi} - 1)[^{\mathtt{vec}', \mathtt{vec}}/_{\mathtt{vec}, \mathtt{vec}^{old}}] \Rightarrow \\
\quad\quad\quad (\mathsf{partition}(\mathtt{sIdx}, \mathtt{low}, \mathtt{hi}) \wedge \mathsf{perm}(\mathtt{low}, \mathtt{hi}))[^{\mathtt{vec}'}/_{\mathtt{vec}}])\}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{PO}(quicksort) = \\
\quad \{\hat{\mathsf{pre}}(quicksort) \Rightarrow \\
\quad\quad ((\texttt{low} < \texttt{hi} \Rightarrow \\
\quad\quad\quad \texttt{vec} \neq null \wedge 0 \leq \texttt{low} \leq \texttt{hi} \leq |\texttt{vec}| \\
\quad\quad\quad \wedge \forall \mathsf{res}, \texttt{vec} \bullet (\mathsf{partition}(\mathsf{res}, \texttt{low}, \texttt{hi}) \wedge \mathsf{perm}(\texttt{low}, \texttt{hi}))[^{\texttt{vec}', \texttt{vec}}/_{\texttt{vec}, \texttt{vec}^{old}}] \Rightarrow \\
\quad\quad\quad\quad \texttt{vec} \neq null \wedge 0 \leq \texttt{low} \leq \texttt{sp} \leq |\texttt{vec}| \\
\quad\quad\quad\quad \wedge (\forall \mathsf{res}, \texttt{vec} \bullet (\mathsf{sorted}(\texttt{low}, \texttt{sp}) \wedge \mathsf{perm}(\texttt{low}, \texttt{sp}))[^{\texttt{vec}', \texttt{vec}}/_{\texttt{vec}, \texttt{vec}^{old}}] \Rightarrow \\
\quad\quad\quad\quad\quad \texttt{vec} \neq null \wedge 0 \leq \texttt{low} \leq \texttt{hi} \leq |\texttt{vec}| \\
\quad\quad\quad\quad\quad \wedge (\forall \mathsf{res}, \texttt{vec} \bullet (\mathsf{sorted}(\texttt{sp}+1, \texttt{hi}) \wedge \mathsf{perm}(\texttt{sp}+1, \texttt{hi}))[^{\texttt{vec}', \texttt{vec}}/_{\texttt{vec}, \texttt{vec}^{old}}] \Rightarrow \\
\quad\quad\quad\quad\quad\quad (\mathsf{sorted}(\texttt{low}^{old}, \texttt{hi}^{old}) \wedge \mathsf{perm}(\texttt{low}^{old}, \texttt{hi}^{old}))[^{\texttt{vec}'}/_{\texttt{vec}}])[^{\texttt{vec}'}/_{\texttt{vec}}])[^{\texttt{vec}'}/_{\texttt{vec}}][^{\mathsf{res}}/_{\texttt{sp}}]) \\
\quad\quad \wedge (\neg \texttt{low} < \texttt{hi} \Rightarrow \mathsf{sorted}(\texttt{low}^{old}, \texttt{hi}^{old}) \wedge \mathsf{perm}(\texttt{low}^{old}, \texttt{hi}^{old})))[^{\texttt{low}, \texttt{hi}, \texttt{vec}}/_{\texttt{low}^{old}, \texttt{hi}^{old}, \texttt{vec}^{old}}]\}
\end{array}
$$

Como ya hemos dicho, el generador de condiciones de verificación híbrido no sólo utiliza el resultado del análisis para fortalecer las precondiciones y los invariantes provistos por la especificación, sino que también lo utiliza para reducir el tamaño de las obligaciones de prueba, eliminando aquellas generadas por aristas espurias en el grafo de control de flujo del programa. Además, incluye como hipótesis el resultado del análisis permitiendo que las demostraciones se lleven a cabo con mayor facilidad.

Los resultados obtenidos en este caso de estudio son altamente satisfactorios, ya que todas estas ventajas se pueden apreciar claramente. A pesar de ser un programa pequeño, la diferencia en el tamaño de las condiciones de verificación generadas por ambos VCgen es notable. Incluso, al permitir que sea el análisis quien obtenga gran parte de los invariantes numéricos y de nulidad, todo el esfuerzo involucrado en la especificación puede ser focalizado en la búsqueda de invariantes más complejos que permitan describir el comportamiento del programa, tales como sorted y perm.

Si bien en la práctica la implementación de métodos híbridos de verificación conlleva un esfuerzo mayor que la implementación de métodos estándares, los beneficios que brinda su utilización hacen que dicho esfuerzo valga la pena. Los métodos híbridos son un ingrediente esencial en la verificación de software y, aunque ésta no pueda automatizarse por completo, reducen gratamente la necesidad de intervención humana.

5

Trabajos Relacionados y Conclusiones

Existen dos enfoques a la verificación híbrida de programas. El enfoque explícito, el usuario provee anotaciones de seguridad que son utilizadas por el generador de condiciones de verificación y chequeadas por un chequeador de anotaciones. En contraste, el enfoque implícito aboga por que las anotaciones sean inferidas por el analizador estático, y luego utilizadas en la generación de condiciones de verificación. Ambos enfoques son utilizados (a veces en conjunto) en la verificación deductiva de programas así como en análisis basados en tipos.

Además, algunos autores han formalizado y demostrado la correctitud de métodos de verificación híbridos. Por ejemplo, Wildmoser, Chaieb y Nipkow [31] han utilizado Isabelle/HOL para demostrar la correctitud de métodos híbridos para bytecode Java; se basan en análisis de intervalos para detectar accesos a arreglos fuera de rango e implementan una versión del análisis que genera pruebas de que el resultado del análisis es correcto. Más recientemente, Grégoire y Sacchini [14] formalizaron en Coq un método híbrido de verificación para programas para la JVM. Se enfocan en un análisis de punteros nulos; a pesar de que el analizador no está formalizado en Coq, Hubert et al. [19] provee un punto de partida para llevar a cabo dicha implementación. El método de [14] soporta interacción bidireccional entre el análisis y la generación de condiciones de verificación, dado que el análisis estático puede extraer información de utilidad de las anotaciones del programa, al mismo tiempo que el resultado del análisis es aprovechado por el generador de condiciones de verificación para reducir el número de obligaciones de prueba (a pesar de no haber hecho esto en el presente trabajo, es posible y relativamente simple integrar dicha interacción bidireccional a al mismo). Sin embargo, no consideran la posibilidad de fortalecer los invariantes con los resultados del análisis y tampoco estudian la relación entre la verificación de programas fuente y programas compilados.

Es conocido que los métodos de verificación deductivos pueden ser vistos como Interpretación Abstracta [9, 11]. La Interpretación Abstracta Lógica [15] explora la interacción entre análisis y verificación desde la perspectiva de usar demostración de teoremas para mejorar la precisión de las interpretaciones abstractas y combinaciones de las mismas.

Finalmente, nuestro trabajo está estrechamente relacionado con trabajos previos sobre transformación de pruebas por compilación y análisis de programas generadores de pruebas. Saabas y Uustalu [28] proporcionan un algoritmo para transformar pruebas en lógica de Hoare en pruebas en sistemas de prueba composicional para programas assembler. Moviéndonos a lenguajes más realísitcos, Müler et al. definen transformación de pruebas por compilación para Java y Eiffel [2, 24, 27].

Los trabajos aquí mencionados, así como el presente trabajo, se enfocan en compilación no-optimizante. Compilar demostraciones considerando optimizaciones de programa exige utilizar análisis generadores de pruebas que producen pruebas formales de su correctitud. Dichos análisis son también necesarios para extender los resultados de la sección 2.1.2 a certificados. Los análisis generadores de pruebas han sido estudiados por varios autores, incluidos Wildmoser, Chaieb and Nipkow [31] en el contexto de generación de condiciones de verificación para un lenguaje de bytecode, y Seo, Yang y Yi [29] en el contexto de una lógica de Heare para un lenguaje imperativo simple.

Para más detalles sobre trabajos relacionados en este área, pueden consultarse [3, 4].

Los entornos de verificación de programas se basan cada vez más en métodos híbridos para probar correctitud de software. En este trabajo hemos mostrado que los métodos híbridos de verificación a nivel fuente y bytecode están estrechamente relacionados y que los métodos híbridos de verificación pueden ser "compilados" a métodos estándares de verificación, los cuales garantizan la corectitud de los métodos híbridos.

Para lograr esto, hemos definido dos frameworks de verificación. Uno para código fuente y otro para bytecode. Hemos definido los lenguajes sobre los cuales se definieron dos tipos de análisis —uno numérico y uno de nulidad. Intentando ser lo más generales posible, en vez de optar por implementaciones particulares de dichos análisis, hemos descripto las interfaces genéricas de los mismos y hemos dado las respectivas definiciones de solución de análisis utilizando sistemas de restricciones sobre los elementos de los dominios abstractos. Para mitigar la presencia del stack de operandos y recuperar la pérdida de precisión generada por la compi-

lación, en el análisis de bytecode se utilizaron expresiones simbólicas que permiten conservar la información de las relaciones entre variables.

La preservación de obligaciones de prueba se sustentó en el hecho de que las soluciones de la interpretación abstracta también se preservan. Esto ha sido expuesto y demostrado en la sección 2.2.5 del capítulo 2.

El resultado del análisis no sólo fue utilizado para descartar obligaciones de pruebas generadas de aristas espurias del grafo de control de flujo del programa y reducir el conjunto de condiciones de verificación generadas por el VCgen, sino que también se utilizó para fortalecer las anotaciones y así facilitar la demostración de las obligaciones de prueba. Esto brinda un gran beneficio a los productores de código, ya que reduce considerablemente el tiempo empleado en la demostración interactiva de las condiciones de verificación que genera el certificado.

Finalmente, en el capítulo 4, hemos presentado QuickSort como caso de estudio. Presentamos una solución del análisis de source code, obtenida combinando la solución generada por Interproc Analyzer [20] —una herramienta de análisis estático basada en la librería de dominios numéricos abstractos APRON [23]— para el análisis numérico con una solución del análisis de nulidad. Se mostraron las obligaciones de prueba generadas tanto por el VCgen híbrido como por el estándar y se analizaron las ventajas de utilizar el primero.

En esta línea, un próximo objetivo es extender nuestros resultados a lenguajes y análisis más realistas. Los lenguajes modernos incluyen características tales como excepciones, que pueden conllevar grafos de control de flujo muy grandes haciendo que los certificados híbridos sean particularmente necesarios.

Además, sería beneficioso permitir que los métodos híbridos se basen en análisis estáticos más avanzados que provean información valiosa para demostrar propiedades de programas. Desarrollar un método híbrido de verificación basado en el análisis presentado en [16] podría resultar más que interesante.

Bibliografía

[1] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Quaderno 457, Dipartimento di Matematica, Università di Parma, Italy, 2006.

[2] F. Y. Bannwart and P. Müller. A program logic for bytecode. In F. Spoto, editor, *Bytecode Semantics, Verification, Analysis and Transformation*, volume 141 of *Electronic Notes in Theoretical Computer Science*, pages 255–273. Elsevier, 2005.

[3] G. Barthe, B. Grégoire, C. Kunz, and T. Rezk. Certificate translation for optimizing compilers. In K. Yi, editor, *Static Analysis Symposium*, number 4134 in Lecture Notes in Computer Science, pages 301–317, Seoul, Korea, August 2006. Springer-Verlag.

[4] G. Barthe, B. Grégoire, and M. Pavlova. Preservation of proof obligations for Java. In *International Joint Conference on Automated Reasoning*, Lecture Notes in Computer Science. Springer-Verlag, 2008. To appear.

[5] G. Barthe and C. Kunz. Certificate translation in abstract interpretation. In S. Drossopoulou, editor, *European Symposium on Programming*, volume 4960 of *Lecture Notes in Computer Science*, pages 368–382, Budapest, Hungary, April 2008. Springer-Verlag.

[6] G. Barthe, C. Kunz, D. Pichardie, and J. Samborski-Forlese. Preservation of proof obligations for hybrid verification methods. In *Software Engineering and Formal Methods*. IEEE Press, 2008.

[7] A. Bernard and P. Lee. Temporal logic for proof-carrying code. In A. Voronkov, editor, *Conference on Automated Deduction*, volume 2392 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[8] F. Besson, T. Jensen, D. Pichardie, and T. Turpin. Result certification for relational program analysis. Research Report 6333, IRISA, September 2007.

[9] P. Cousot. Semantic foundations of program analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

[10] P. Cousot. Automatic verification by abstract interpretation, invited tutorial. In L.D. Zuck, P.C. Attie, A. Cortesi, and S. Mukhopadhyay, editors, *Proceedings of the Fourth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2003)*, pages 20–24, Courant Institute, NYU, New York, N.Y., USA, January 9–11 2003. LNCS 2575, Springer, Berlin.

[11] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages*, pages 238–252, 1977.

[12] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE Analyser. In M. Sagiv, editor, *Proceedings of the European Symposium on Programming (ESOP'05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30, Edinburgh, Scotland, April 2–10 2005. © Springer.

[13] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth ACM Symposium on Principles of Programming Languages*, pages 84–97. ACM Press, 1978.

[14] B. Grégoire and J. Sacchini. Combining a verification condition generator for a bytecode language with static analyses. In *Trustworthy Global Computing*, volume 4912 of *Lecture Notes in Computer Science*, pages 23–40. Springer-Verlag, 2007.

[15] S. Gulwani and A. Tiwari. Combining abstract interpreters. In M. Schwartzbach and T. Ball, editors, *PLDI*, pages 376–386. ACM, 2006.

[16] Nicolas Halbwachs and Mathias Péron. Discovering properties about arrays in simple programs. In R. Gupta and S. P. Amarasinghe, editors, *PLDI*, pages 339–348. ACM, 2008.

[17] Matthew S. Hecht. *Flow Analysis of Computer Programs.* Elsevier Science Inc., New York, NY, USA, 1977.

[18] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

[19] L. Hubert, T. Jensen, and D. Pichardie. Semantic foundations and inference of non-null annotations. In *International Conference on Formal Methods for Open Object-based Distributed Systems*, Lecture Notes in Computer Science. Springer-Verlag, 2008. To appear.

[20] Gaël Lalire, Mathias Argoud, and Bertrand Jeannet. Interproc static analyzer. Available at `http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/interproc/`.

[21] Francesco Logozzo and Manuel Fähndrich. On the relative completeness of bytecode analysis versus source code analysis. In Laurie Hendren, editor, *CC*, volume 4959 of *Lecture Notes in Computer Science*, pages 197–212. Springer, 2008.

[22] A. Miné. *Weakly Relational Numerical Abstract Domains*. PhD thesis, École Polytechnique, Palaiseau, France, December 2004. `http://www.di.ens.fr/~mine/these/these-color.pdf`.

[23] A. Miné and B. Jannet. Apron: A library of numerical abstract domains for static analysis, 2009. To appear.

[24] P. Müller and M.Ñordio. Proof-transforming compilation of programs with abrupt termination. In *SAVCBS '07: Proceedings of the 2007 conference on Specification and verification of component-based systems*, pages 39–46, New York, NY, USA, 2007. ACM.

[25] G. C. Necula. Proof-carrying code. In *Principles of Programming Languages*, pages 106–119, New York, NY, USA, 1997. ACM Press.

[26] G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *Programming Languages Design and Implementation*, volume 33, pages 333–344, New York, NY, USA, 1998. ACM Press.

[27] M.Ñordio, P. Müller, and B. Meyer. Proof-transforming compilation of eiffel programs. In R. Paige, editor, *TOOLS-EUROPE*, Lecture Notes in Business and Information Processing. Springer-Verlag, 2008.

[28] A. Saabas and T. Uustalu. A compositional natural semantics and Hoare logic for low-level languages. *Theoretical Computer Science*, 373(3):273–302, 2007.

[29] S. Seo, H. Yang, and K. Yi. Automatic Construction of Hoare Proofs from Abstract Interpretation Results. In A. Ohori, editor, *Asian Programming Languages and Systems*

Symposium, volume 2895 of *Lecture Notes in Computer Science*, pages 230–245. Springer-Verlag, 2003.

[30] Michel Sintzoff. Calculating properties of programs by valuations on specific models. *SIGACT News*, (14):203–207, 1972.

[31] M. Wildmoser, A. Chaieb, and T.Ñipkow. Bytecode analysis for proof carrying code. In F. Spoto, editor, *Bytecode Semantics, Verification, Analysis and Transformation*, volume 141 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2005.

[32] M. Wildmoser, T.Ñipkow, G. Klein, and S.Ñanz. Prototyping proof carrying code. In J-J. Levy, E. W. Mayr, and J. C. Mitchell, editors, *Theoretical Computer Science*, pages 333–347. Kluwer Academic Publishing, August 2004.

[33] H. Xi and S. Xia. Towards array bound check elimination in java tm virtual machine language. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 14. IBM Press, 1999.