Libros de Cátedra

Verificación de programas

Programas secuenciales y concurrentes

Ricardo Rosenfeld

exactas

FACULTAD DE
INFORMÁTICA

edulp
EDITORIAL DE LA UNLP

UNIVERSIDAD
NACIONAL
DE LA PLATA

VERIFICACIÓN DE PROGRAMAS

PROGRAMAS SECUENCIALES Y CONCURRENTES

Ricardo Rosenfeld

Facultad de Informática

Índice

Prólogo

El objetivo de este libro es presentar fundamentos de la *verificación de programas*, actividad formal para probar propiedades de programas. La exposición se hace en términos de un enfoque de uso muy extendido, la *verificación axiomática*, en su variante conocida como *lógica de Hoare*.

Este enfoque sale a la luz oficialmente en un artículo de 1969 publicado por C. Hoare, *An axiomatic basis for computer programming* (Una base axiomática para la programación de computadoras). Ya en la introducción del artículo queda perfectamente plasmada su fundamentación: todas las propiedades de un programa y todas las consecuencias de ejecutarlo en un determinado entorno pueden, en principio, ser descubiertas a partir del texto del propio programa, por medio del puro razonamiento deductivo. Así, el autor plantea un método lógico-matemático para probar propiedades de programas. A este hito fundacional le sucede una muy prolífica etapa de investigación en el área de verificación de programas, en la que surgen numerosas axiomáticas cubriendo distintos paradigmas de programación.

Este es mi cuarto libro sobre verificación de programas para la Colección de Libros de Cátedra de la Universidad Nacional de La Plata (UNLP), esta vez dedicado por completo a dicha disciplina (los anteriores incluyen elementos de teoría de la computación). ¿Por qué un libro más? Me pareció útil volcar en un nuevo documento los cambios que he venido efectuando durante los últimos años en mis clases y seminarios sobre verificación de programas, sobre todo en la Facultad de Informática de la UNLP, cambios en el estilo de presentación y en los contenidos, motivados fundamentalmente por modificaciones en los planes de estudio y por lecciones aprendidas de mi propia experiencia en la exposición de los temas.

Entre lo distintivo de este libro, comparándolo con los anteriores, destaco:

- La exposición prioriza lo conceptual por sobre lo formal.
- Se remarca el empleo de los métodos axiomáticos de prueba para guiar la construcción de programas. En sintonía con esta idea, se presentan ejemplos de desarrollo de programas en base a los métodos descriptos.
- Antes de la presentación de distintos métodos representativos de la lógica de Hoare, se incluye un capítulo preparatorio con definiciones y notaciones básicas relacionadas con la verificación de programas, de la matemática en general, los lenguajes y los métodos deductivos. La idea es que se utilice como material de consulta cuando fuera necesario a lo largo de la lectura de las distintas partes del libro.

- En las pruebas de terminación de programas se utilizan *funciones* en lugar de *predicados parametrizados*, por resultar más intuitivas.
- El estudio de las propiedades de los métodos axiomáticos, es decir de la *metateoría* de la verificación axiomática de programas, se restringe a los programas secuenciales y se muestra en un capítulo aparte. El foco se pone en la descripción de los métodos de prueba, reforzada con ejemplos de aplicación.
- Al final de los capítulos que describen los métodos, incluyendo el de la metateoría, se presentan notas adicionales para alentar al lector a profundizar en distintos aspectos relacionados con los temas centrales estudiados.
- En las referencias bibliográficas se privilegian citas y obras que considero inspiradoras (vuelvo a ellas cada tanto, sus ideas están más vigentes que nunca).

Más allá de las diferencias con mis libros anteriores, en este trabajo intento seguir reflejando mi firme y cada vez más fuerte convicción de la relevancia que tiene en la currícula de ciencias de la computación la enseñanza de principios fundamentales de la *correctitud de programas*, principios matemáticos y lógicos claramente identificados en los métodos axiomáticos de prueba. Parafraseando a E. Dijkstra, pionero como C. Hoare en la conformación de una auténtica ciencia de la programación, la tarea de un programador es derivar fórmulas de un sistema formal, y por lo tanto no hay manera de comprenderla sin sólidos conocimientos de matemática y lógica.

El libro está dirigido a los estudiantes de los últimos años de la carrera de ciencias de la computación y a graduados que deseen introducirse en la problemática de la correctitud de programas. Se puede utilizar para el dictado de una materia semestral sobre verificación de programas secuenciales y concurrentes. En la introducción del libro se sugieren distintas alternativas para su lectura.

Se asume un lector con cierta madurez en matemática en general y lógica de predicados en particular, conocimientos de algorítmica, programación y lenguajes de programación. Es deseable contar también, en alguna medida, con nociones básicas de teoría de la computación.

Agradezco a las autoridades de la Facultad de Informática de la UNLP por esta nueva posibilidad que me dan para publicar un texto de esta naturaleza. A mi familia, por haberme tolerado una vez más silencios y miradas perdidas en los momentos, que fueron muchos, en los que vaya a saber uno en qué párrafo estaba pensando para volcar en el libro de la mejor manera posible. A quienes leyeron borradores de este trabajo, porque a partir de sus observaciones surgieron importantes mejoras en la presentación de algunos temas. Y como siempre, agradezco especialmente a mis alumnos, por sus valiosos aportes en clase y fuera de clase, que a lo largo de los años vienen enriqueciendo sobremanera mi labor docente.

Ricardo Rosenfeld

La Plata, julio de 2024

Introducción

Nuestro propósito en este libro es describir sistemáticamente fundamentos de la *verificación axiomática de programas*, en su variante conocida como *lógica de Hoare* (para simplificar, de ahora en más empleamos de manera indistinta ambas denominaciones). En esta introducción planteamos aspectos generales de dicha variante y presentamos la estructura del libro y un resumen de sus contenidos.

Acerca de la verificación axiomática de programas

La verificación axiomática es un enfoque muy común de la verificación de programas. Consiste en un conjunto de métodos lógico-matemáticos para probar propiedades de programas. El lenguaje empleado para expresar las propiedades de un programa, reunidas en una *especificación*, es el de la *lógica de predicados (con igualdad)*. Y la familia de programas tratados es muy representativa, la de los programas *imperativos*, también conocidos como *procedurales*, los cuales se caracterizan por transformar *estados* (conjuntos de variables con sus contenidos) a través de sus instrucciones (de todos modos, varios conceptos manejados por la lógica de Hoare trascienden este paradigma).

Al ser una actividad formal, la verificación axiomática, en realidad cualquier variante de la verificación de programas, se diferencia del *testing*, comúnmente asociado con el término *validación* para acentuar su naturaleza no formal. E. Dijkstra explica claramente la diferencia en una de sus citas más conocidas: la verificación es la única vía para elevar significativamente el nivel de confianza de un programa, provee una prueba convincente de su correctitud, en tanto que el testing sólo se limita a revelar la presencia de errores, no su ausencia.

Una alternativa a la verificación axiomática para probar que un programa satisface una especificación se basa en un análisis *operacional*: se revisa el comportamiento del programa en términos de sus computaciones, es decir de sus secuencias de instrucciones (siempre que la semántica del lenguaje de programación sea lo suficientemente clara). El problema de esta modalidad, de naturaleza semántica, es que si bien puede resultar eficaz para ciertos tipos de programas, resulta prohibitiva cuando se aplica a programas muy complejos, sobre todo concurrentes.

Los métodos axiomáticos no tienen la dificultad del análisis operacional. Emulando el mecanismo lógico, para probar que un programa satisface (o es correcto con respecto a) una

especificación, se valen de axiomas y reglas que relacionan instrucciones con predicados, permitiendo derivar pruebas guiadas por la estructura del programa. Se reemplazan argumentos sobre el comportamiento de las computaciones por otros derivados sintácticamente de una sistemática aplicación de reglas formales. Este valor agregado es notorio principalmente en la verificación de los programas concurrentes, que tienen numerosas computaciones, lo que dificulta sobremanera la tarea de probar su correctitud. El método axiomático facilita las verificaciones, permite un análisis sistemático de todas las computaciones y de todos los detalles de cada una de ellas.

La aplicación de la metodología a estudiar requiere cálculo e intuición, considerando una especificación, un programa y una axiomática. No es posible automatizar la actividad, la teoría de la computación establece que en general la verificación de programas no es decidible (un escenario decidible lo constituyen programas que pueden modelizarse mediante sistemas de estados finitos, para los que existen algoritmos automáticos de *model checking*, tal como se los conoce). Esta limitación implica un problema de escalabilidad, pero atenuado notoriamente por abundante soporte computacional para asistir en las pruebas, sumado a la práctica industrial cada vez más habitual de construir módulos verificados acompañados por reglas de correctitud para combinarlos.

Otras limitaciones de la lógica de Hoare tienen que ver con aspectos de alcance y de índole metodológica:

- Sus reglas de prueba sólo permiten reflejar comportamientos de entrada/salida de los programas. No permiten tratar propiedades de las computaciones de los programas, ni tampoco el *fairness*.
- Su aplicación ya asume la existencia de una especificación y de un programa, es decir que plantea una actividad de verificación sólo *a posteriori*.

La primera limitación mencionada se puede resolver, por ejemplo, recurriendo a métodos que utilizan la *lógica temporal*, la cual permite expresar propiedades más generales, lo mismo que asunciones de *fairness*.

La segunda limitación se contrarresta empleando métodos de desarrollo sistemático de programas basados en los principios establecidos por las reglas de prueba. Las reglas constituyen una excelente guía para la construcción de programas correctos. En efecto, y volviendo a parafrasear a E. Dijkstra, una heurística muy eficaz para el desarrollo de un programa es, primero preguntarse cuál sería la estructura de una prueba convincente para el programa, después encontrarla, y finalmente construir el programa de modo tal que satisfaga los requisitos establecidos por la prueba (en síntesis, programar y probar simultáneamente, y así obtener programas correctos por construcción). Para ello, la axiomática de C. Hoare resulta sumamente útil, aportando nociones fundamentales de correctitud de programas. Así y todo, describir la verificación de programas como una actividad *a posteriori* resulta muy didáctica, lo que explica la profusión de esta modalidad de exposición en la literatura. Nuestra presentación sigue dicha

modalidad, pero siempre acentuando la idea de utilizar los axiomas y las reglas de prueba como guía para el desarrollo sistemático de programas.

Estructura del libro y resumen de sus contenidos

El cuerpo principal del libro está constituido por los capítulos 2, 3 y 5, en los que se describen, respectivamente, métodos axiomáticos para verificar programas secuenciales determinísticos, programas secuenciales no determinísticos y programas concurrentes. Las reglas de prueba se presentan a lo largo de los capítulos de manera incremental, considerando cada vez más propiedades, correspondientemente con la complejidad creciente de los programas tratados. Resumimos a continuación los contenidos del libro:

- En el capítulo 1 introducimos definiciones y notaciones básicas relacionadas con la verificación de programas, de la matemática en general, los lenguajes y los sistemas deductivos.
- En el capítulo 2 describimos un primer método axiomático, para probar programas secuenciales determinísticos, basado en el que presenta C. Hoare en su publicación fundacional de 1969 mencionada en el prólogo. Aprovechamos este capítulo para definir nociones fundamentales de la lógica de Hoare, como *fórmula de correctitud, precondición*, *postcondición*, *invariante* y *variante de una instrucción de repetición*, *composicionalidad, sensatez, completitud*. En particular, introducimos el lenguaje de la lógica de predicados, único lenguaje de especificación utilizado en todos los métodos. Como lenguaje de programación utilizamos un lenguaje muy simple con repeticiones *while*, fragmento del lenguaje Pascal. Tratamos la verificación de las dos propiedades básicas de los programas secuenciales determinísticos, la *correctitud parcial* y la *terminación* (o *no divergencia*), que establecen, respectivamente, que el programa especificado siempre produce el resultado esperado y nunca genera una computación infinita. El lenguaje de programación y los axiomas y reglas de este capítulo constituyen la base de los lenguajes y las axiomáticas que se utilizan en los capítulos siguientes.
- En el capítulo 3 describimos un segundo método axiomático, adaptación del método planteado en el capítulo anterior, para probar programas secuenciales no determinísticos, en los que una instrucción puede tener varias continuaciones, y así los programas varias computaciones. Como la concurrencia puede ser modelizada por el no determinismo, el capítulo sirve también como introducción a la verificación de programas concurrentes. La axiomática que presentamos se basa en aportes de varios autores desde inicios de los años 1970, como P. Lauer, J. de Bakker, K. Apt, E. Dijkstra, D. Gries, D. Harel y N. Francez. Trabajamos con una clase de programas no determinísticos muy conocida, la de los programas con *comandos guardados* (o *comandos con guardia*) del lenguaje GCL o *Guarded Command Language* (Lenguaje de Comandos Guardados) creado por E. Dijkstra. Además de la correctitud parcial y la terminación, tratamos la verificación de una tercera propiedad,

la *ausencia de falla*, que establece que ninguna computación de un programa manifiesta una falla (para simplificar, sólo consideramos como causal de falla una selección condicional erróneamente codificada). Al final presentamos dos variantes de la axiomática, que contemplan el agregado al lenguaje de programación de asignaciones aleatorias y de *fairness*.

- En el capítulo 4 hacemos un paréntesis en la descripción de métodos axiomáticos para introducir elementos de la *metateoría* de la verificación de programas, es decir del estudio de características de los métodos de verificación mismos, con la idea de reforzar el entendimiento de la problemática de la correctitud de programas. Específicamente, probamos que los métodos presentados son *sensatos* y (en algún sentido) *completos*, propiedades relacionadas con la correctitud y el alcance de aplicación de las reglas de prueba. Restringimos el análisis a los programas secuenciales porque nuestro objetivo principal es describir axiomáticas de la lógica de Hoare. Antes de presentar las pruebas profundizamos en la *inducción*, técnica con la que se desarrollan, y formalizamos la semántica de los programas en base a la cual se demuestran las propiedades, utilizando la *semántica operacional*, de amplia difusión.
- En el capítulo 5 presentamos las últimas axiomáticas del libro, extensiones de los métodos descriptos previamente, para verificar programas concurrentes tanto con procesos que comparten variables y se comunican a través de ellas como con procesos disjuntos que se comunican por mensajes, que en este libro identificamos como programas *paralelos* y programas *distribuidos*, respectivamente. El capítulo es más extenso que los anteriores, se divide en dos partes, una para cada modelo de concurrencia (las agrupamos en un solo capítulo para remarcar que la metodología de prueba es esencialmente la misma en ambos casos). Las axiomáticas se basan en los trabajos de S. Owicki y D. Gries de 1976, para los programas paralelos con *while*, y de K. Apt, N. Francez y W. de Roever de 1980, para los programas distribuidos con comandos guardados e instrucciones de comunicación correspondientes al lenguaje CSP o *Communicating Sequential Processes* (Comunicación de Procesos Secuenciales) creado por C. Hoare. Incorporamos más nociones fundamentales de la verificación axiomática, propias del paradigma concurrente, como *variable auxiliar*, *invariante global*, *libertad de interferencia*, *cooperación*. Consideramos una segunda posibilidad de falla, típica de la concurrencia, el *deadlock*, que se produce cuando algún proceso queda bloqueado indefinidamente a la espera de un evento que nunca sucede. Hacemos referencia también a otras dos propiedades típicas, la *exclusión mutua* (ausencia de interferencias no deseadas entre procesos) y la *ausencia de inanición* (aseguramiento de que todos los procesos pueden acceder a los recursos compartidos).
- Completamos los contenidos del libro con conclusiones y dos anexos, uno que repasa las axiomáticas descriptas, y otro que lista los ejemplos de aplicación de las mismas, presentados a lo largo de los capítulos.

Tratamos de utilizar siempre la terminología más difundida en la literatura sobre correctitud de programas, incluyendo palabras en inglés si su traducción al español no es de uso habitual.

Los capítulos 2, 3 y 5, que describen los métodos, siguen todos un mismo esquema. Primero se presenta la sintaxis y la semántica (informal) de los programas (el lenguaje de especificación, que siempre es el de la lógica de predicados, se introduce en el capítulo 2). Luego se plantean los axiomas y reglas y se muestran ejemplos de aplicación. Los ejemplos son sencillos, la idea no es presentar casos de estudio con soluciones sofisticadas sino explicar de la manera más simple posible cómo se pueden utilizar las reglas de prueba. Al final se introduce un ejemplo de desarrollo sistemático de programa basado en el método estudiado, para reforzar la idea del aporte de la metodología axiomática de pruebas a la construcción de programas.

Los mismos capítulos, más el capítulo 4 de metateoría, se complementan con observaciones finales, notas adicionales, referencias bibliográficas y ejercicios. Las observaciones finales incluyen aspectos salientes de lo tratado. En las notas adicionales desarrollamos sintéticamente distintos elementos de interés relacionados con la verificación de programas, que para no distraer al lector con respecto al tema central del capítulo los agrupamos aparte. Las referencias bibliográficas se concentran en los trabajos en los que más nos interesa que incursione el lector. Los ejercicios priorizan aspectos conceptuales; resolvemos algunos para repasar lo estudiado o para sugerir modelos de solución de ejercicios similares planteados.

Una de las formas recomendadas para leer el libro es naturalmente la secuencial, desde el capítulo 1 en adelante. Una alternativa es obviar dicho capítulo de definiciones y notaciones y recurrir a él sólo como guía de consulta cuando fuera necesario. También está la opción de dejar el capítulo de metateoría para el final, de modo tal de encarar antes, consecutivamente, todos los capítulos dedicados a los métodos axiomáticos de los distintos tipos de programas considerados.

CAPÍTULO 1
Definiciones preliminares

En este primer capítulo introducimos definiciones básicas y notaciones que se utilizan a lo largo del libro. Asumimos que en general las mismas son conocidas por el lector, por lo que una alternativa es saltear esta parte y acudir a ella como material de consulta cuando resulte necesario (tener en cuenta de todos modos para su lectura que algunas definiciones se basan en definiciones anteriores del mismo capítulo). Para facilitar el acceso a las definiciones, las agrupamos en tres secciones, relacionadas con la matemática en general, los lenguajes y los sistemas deductivos. Algunas definiciones se repiten o reformulan en los capítulos siguientes.

Definiciones matemáticas básicas

Utilizamos la nomenclatura estándar de la matemática para identificar distintos conjuntos de números, como Z para los números enteros y N para los números naturales o enteros positivos. Además, para formular las definiciones empleamos indistintamente palabras (o, y, no, entonces, sii como abreviación de sí y sólo si, para todo, existe, etc.) o sus respectivos símbolos ($\vee$, $\wedge$, $\neg$, $\longrightarrow$, $\leftrightarrow$, $\forall$, $\exists$, etc).

Conjuntos

Un *conjunto* es una colección de elementos. A = {a, b, c} denota un conjunto A con elementos *a*, b y c. En este caso, *a pertenece* a A ($a \in A$) y d no pertenece a A ($d \notin A$). La *cardinalidad* de un conjunto A, denotada con |A|, es la cantidad de elementos de A. Un conjunto es finito o infinito según tenga cardinalidad finita o infinita, respectivamente. El conjunto que no tiene elementos se conoce como conjunto *vacío* y se denota con $\varnothing$. El *conjunto de partes* de un conjunto A, denotado con P(A), es el conjunto de todos los subconjuntos de A.

Dos conjuntos A y B son iguales (A = B) si tienen los mismos elementos. A es un *subconjunto* de B o está *incluido* en B ($A \subseteq B$) si los elementos de A pertenecen a B. De esta manera, A = B

sii ($A \subseteq B \wedge B \subseteq A$). A es un subconjunto *propio* de B o está incluido *estrictamente* en B ($A \subset B$) si $A \subseteq B$ y $A \neq B$.

La *unión* de dos conjuntos A y B ($A \cup B$) es el conjunto de los elementos que están en A o en B. La *intersección* de A y B ($A \cap B$) es el conjunto de los elementos que están en A y en B. A y B son *disjuntos* si $A \cap B = \varnothing$. La *diferencia* de A y B (A – B) es el conjunto de los elementos que están en A y no están en B.

$A = \{a \mid a \in B$ y *a* satisface P$\}$ denota un conjunto A de elementos de B que satisfacen una propiedad P. Por ejemplo, $A = \{a \mid a \in N$ y *a* es par$\}$ denota el conjunto de los números naturales pares.

Relaciones

El producto cartesiano de n conjuntos $A_1, A_2, \ldots, A_n$ ($A_1 \times A_2 \times \ldots \times A_n$) es el conjunto de las tuplas $(a_1, a_2, \ldots, a_n)$ tales que $a_1 \in A_1, a_2 \in A_2, \ldots, a_n \in A_n$. Una tupla $(a_1, a_2, \ldots, a_n)$ y un conjunto $\{a_1, a_2, \ldots, a_n\}$ son distintos, sólo en una tupla importa el orden de los componentes y además los componentes pueden repetirse. Se cumple $(a_1, a_2, \ldots, a_n) = (b_1, b_2, \ldots, b_n)$ sii para todo i, $a_i = b_i$.

Cualquier subconjunto del producto cartesiano $A_1 \times A_2 \times \ldots \times A_n$ es una *relación* n-aria entre los conjuntos $A_1, A_2, \ldots, A_n$. $R(a_1, a_2, \ldots, a_n)$ denota que la tupla $(a_1, a_2, \ldots, a_n)$ es un elemento de la relación R entre $A_1, A_2, \ldots, A_n$. En particular, una relación binaria R entre A y A, o directamente en A, se denota con (A, R). R es *reflexiva* si todo $a \in A$ cumple R(a, a), *irreflexiva* si todo $a \in A$ cumple $\neg R(a, a)$, *simétrica* si todo par *a* y b de A cumple $R(a, b) \rightarrow R(b, a)$, *antisimétrica* si todo par *a* y b de A cumple $(R(a, b) \wedge R(b, a)) \rightarrow a = b$, y *transitiva* si toda terna *a*, b y c de A cumple $(R(a, b) \wedge R(b, c)) \rightarrow R(a, c)$.

Una relación en A es un *orden parcial* si es reflexiva, antisimétrica y transitiva. Se dice en este caso que A es un *conjunto parcialmente ordenado*, y se denota con $(A, \leq)$. Si en cambio la relación es irreflexiva, el orden parcial se conoce como *estricto*, y se dice que A es un *conjunto parcialmente ordenado estricto*, lo que se denota con (A, <). $a \leq b$ y $a < b$ también se denotan con $b \geq a$ y $b > a$, respectivamente. (A, <) es un *conjunto bien fundado* si no tiene ninguna cadena descendente infinita, es decir de la forma $a_1 > a_2 > a_3 > \ldots$. Todo elemento $a_i \in A$ para el que no existe ningún elemento $a_j \in A$ que cumple $a_j < a_i$, se denomina *minimal*. Por ejemplo, el conjunto N de los números naturales con la relación $<$ de la aritmética *estándar* (ver más adelante la definición de *modelo estándar*) es un conjunto bien fundado, denotado usualmente con (N_0, <), y que tiene sólo al número 0 como minimal. Los conjuntos bien fundados se utilizan en las pruebas de terminación de programas.

Funciones

Una *función* f de un conjunto A a un conjunto B (f : A $\rightarrow$ B) es una relación binaria entre A y B, que para todo elemento *a* de A contiene a lo sumo un elemento b de B. A y B se conocen como el *dominio* y el *codominio* de f, respectivamente. La notación f(a) = b es otra manera de expresar que (a, b) $\in$ f, siendo *a* el *argumento* y b el *valor*.

Una función f : A $\rightarrow$ B es *total* si está definida para todo elemento *a* de A, *parcial* en caso contrario, *inyectiva* si todo par de elementos *a* y *a*' de A cumple f(a) $\neq$ f(a'), *suryectiva* si está definida para todo elemento b de B, y *biyectiva* si es inyectiva y suryectiva (en este último caso se dice que existe una *biyección* entre A y B).

En una función de la forma particular f : A $\rightarrow$ A se define que a $\in$ A es un *punto fijo* de f si cumple f(a) = a. También puede haber funciones de la forma f : A_1 x A_2 x … x $A_n \rightarrow$ B, en cuyo caso se usa la notación f(a_1, a_2, …, a_n) = b.

Otra función particular es f : {1, …,n} $\rightarrow$ A, que define una *secuencia* de n elementos de un conjunto A. Las secuencias también pueden ser infinitas, en cuyo caso el dominio es N. Las secuencias se suelen denotar listando directamente los valores de las funciones que las definen, con o sin ningún tipo de puntuación, en el orden ascendente de los argumentos: a_1, a_2, …, a_n, o bien directamente $a_1a_2…a_n$.

Números ordinales y cardinales infinitos

Para la prueba de terminación de un programa en general alcanza con recurrir al conjunto de los números naturales. Sólo en algunos casos, relacionados con los programas no determinísticos, hay que utilizar un dominio mayor. En efecto, surge la necesidad de emplear números *ordinales* infinitos. En el capítulo 3 desarrollamos un ejemplo de prueba de terminación en el que agregamos al conjunto de los números naturales el primer ordinal infinito, conocido como ω, mayor que todos ellos.

Los números ordinales infinitos, lo mismo que los números *cardinales* infinitos, constituyen extensiones del dominio de los números naturales para representar, respectivamente, ubicaciones y tamaños en el marco de los conjuntos infinitos.

Los números ordinales se definen a partir de la aplicación sucesiva de dos principios de generación, el de la suma de una unidad a un número ya constituido, y el de la creación de un nuevo número, límite superior de los anteriores, inmediatamente mayor que todos ellos. Mediante una primera aplicación de los dos principios de generación se producen los números naturales 0, 1, 2, …, y el primer ordinal infinito, el número ω, mayor que todos los números naturales como ya se ha dicho. A partir de ω, aplicando de nuevo los dos principios, se obtiene una segunda secuencia, constituida por los números ordinales ω + 1, ω + 2, …, y el ordinal ω + ω = 2 **.** ω, este último mayor que todos los anteriores. Y así se puede seguir ampliando la secuencia indefinidamente.

En el contexto de la verificación de programas no determinísticos surge también la necesidad de recurrir al primer cardinal infinito, el número $\aleph_0$, que representa el tamaño del conjunto N. Esto sucede cuando en los programas se hace referencia a los conjuntos infinitos *contables* (o *numerables*). Un conjunto infinito es contable si se puede establecer una biyección entre él y N.

Demostraciones matemáticas

Además de las pruebas de correctitud de programas, en el libro también desarrollamos típicas demostraciones matemáticas, secuencias de pasos que incluyen operadores matemáticos y lógicos. En estos casos utilizamos fundamentalmente dos métodos de prueba, la *inducción* y la *reducción al absurdo*.

La inducción en su forma más simple, conocida como *inducción matemática*, plantea que para probar que una propiedad P se cumple para todos los números naturales, hay que probar que se cumple para el 0, y que si se cumple para un número natural $k \geq 0$, entonces también se cumple para el número k + 1. Un caso más general es la *inducción estructural*, que permite tratar con cualquier conjunto (con ciertas características), no solamente el de los números naturales. Mostramos sobre todo su uso en el capítulo 4, dedicado a las pruebas de sensatez y completitud de los métodos de prueba. La inducción sirve además para plantear definiciones; por ejemplo, para especificar la sintaxis de los lenguajes de programación.

Otro método de prueba que utilizamos es la reducción al absurdo, también llamado *método de prueba por contradicción*. Se basa en el *principio del tercero excluido*, según el cual si un enunciado afirma algo y otro lo contradice, necesariamente uno debe ser verdadero y el otro falso. El método consiste en suponer la falsedad del enunciado que se quiere demostrar, llegar a una contradicción, y así concluir que el enunciado es verdadero.

Definiciones relacionadas con los lenguajes

En la descripción de los métodos de verificación de programas distinguimos tres lenguajes, el de los programas, el de las especificaciones (que siempre es el de la *lógica de predicados*) y el de las *fórmulas de correctitud* o *ternas de Hoare* (con componentes de los otros dos lenguajes). Su sintaxis y semántica se formalizan en los próximos capítulos. En lo que sigue introducimos algunas definiciones básicas relacionadas con los lenguajes.

Alfabetos, cadenas y lenguajes

Un *alfabeto* es un conjunto de símbolos. Con un alfabeto se pueden generar *cadenas*, secuencias de símbolos; por ejemplo, la cadena x+1 a partir del alfabeto {0, 1, x, +}. Todo conjunto de cadenas constituye un *lenguaje*.

Sintaxis de un lenguaje

Una manera muy conocida de especificar las cadenas de un lenguaje es mediante una *gramática*, y en la notación denominada *Backus-Naur Form* (abreviada con BNF). Un ejemplo es la siguiente gramática, que define las cadenas que conforman un lenguaje muy simple, con el que se pretende representar expresiones aritméticas:

$e :: 0 \mid 1 \mid x \mid (e_1 + e_2) \mid (e_1 - e_2)$

La gramática, definida inductivamente, establece que toda cadena del lenguaje es una expresión aritmética *e* que tiene la forma 0, 1, x, $(e_1 + e_2)$ o $(e_1 - e_2)$, siendo las e_i a su vez expresiones aritméticas. Por ejemplo, algunas cadenas que se derivan de la gramática descripta son 0, x, (1 + x), (0 – (x + x)). 0 y 1 son las *constantes* del lenguaje, x es la única *variable*, y los símbolos + y – son los símbolos de *función*, los cuales también se consideran constantes.

Semántica de un lenguaje

Mientras la sintaxis de un lenguaje describe la forma de sus cadenas (dominio sintáctico del lenguaje), la semántica establece sus significados, los elementos semánticos asociados a ellas (dominio semántico del lenguaje).

La semántica de un lenguaje se especifica mediante una *función semántica* entre su dominio sintáctico y su dominio semántico. Por ejemplo, considerando el lenguaje de expresiones aritméticas de la subsección anterior, la función semántica podría asignar un número a toda cadena derivada de la gramática. Para ello, por un lado, a partir de una (función de) *interpretación*, podría asignar los números 0 y 1 a las constantes 0 y 1 y las operaciones de suma y resta a los símbolos + y –, y por otro lado, mediante un *estado* (función del conjunto de variables al dominio semántico), podría valorizar las variables.

Para cadenas como expresiones aritméticas, expresiones booleanas y predicados de la lógica, el mecanismo descripto, basado en *estructuras semánticas*, constituidas por un dominio semántico y una interpretación, es la manera habitual de especificar la semántica (se la conoce como semántica *denotacional*). En cambio, en el caso de las cadenas que conforman las instrucciones de los programas imperativos que utilizamos en este libro, la modalidad de especificación semántica más común es la semántica *operacional*, en la que la función semántica se define en términos de elementos tales como computaciones, configuraciones, continuaciones sintácticas, estados corrientes, etc.

En este libro trabajamos con constantes y variables de tipo entero y booleano, y así con los dominios semánticos de los números enteros y los valores de verdad *verdadero* y *falso*. Las variables en general son simples, rara vez recurrimos a variables estructuradas, como por ejemplo los arreglos.

Definiciones relacionadas con los sistemas deductivos

Los *sistemas deductivos* (o *métodos axiomáticos*, término que usamos más a menudo en los próximos capítulos) constituyen el tema central del libro. Son conjuntos de fórmulas, denominadas *axiomas*, y de *reglas de inferencia*, o directamente *reglas*, mediante las cuales pueden producir nuevas fórmulas. La forma de una regla es:

$$\frac{\varphi_1, \ldots, \varphi_k}{\varphi}$$

Las fórmulas $\varphi_1, \ldots, \varphi_k$ son las *premisas* de la regla. La fórmula φ es la *conclusión* de la regla. Se dice que de $\varphi_1, \ldots, \varphi_k$ se deduce (o se deriva, o se infiere) φ.

Una *prueba* (o *demostración*) de una fórmula φ en un sistema deductivo es una secuencia de fórmulas $\varphi_1, \ldots, \varphi_n$, tales que $\varphi = \varphi_n$ y toda fórmula φ_i es un axioma o puede ser obtenida por la aplicación de una o más reglas sobre las fórmulas $\varphi_1, \ldots, \varphi_{i-1}$. La fórmula φ_n se conoce como *teorema*. La longitud de la secuencia $\varphi_1, \ldots, \varphi_n$ determina la longitud o la cantidad de pasos de la prueba de φ_n (a los axiomas se los considera teoremas que tienen pruebas de un solo paso).

Ejemplo de prueba en un sistema deductivo

En lo que sigue, ejemplificamos una prueba en un sistema deductivo. El objetivo es anticipar la forma de las pruebas que mostramos más adelante, y distintas definiciones que utilizamos (las presentamos en la próxima subsección). Demostramos la fórmula 1 + 1 = 2, recurriendo a una axiomática habitual de los números naturales, la *aritmética de Peano de primer orden.* La elección no es casual, la idea es acercar al lector al escenario en el que desarrollamos después las verificaciones de programas. El sistema es el siguiente (los predicados se anotan con las letras p, q, …, las variables con x, *y*, …, y las expresiones con *e*):

Axiomas de la lógica de predicados

$K_1 : p \rightarrow (q \rightarrow p)$

$K_2 : (p \rightarrow (q \rightarrow r)) \rightarrow ((p \rightarrow q) \rightarrow (p \rightarrow r))$

$K_3 : (\neg p \rightarrow \neg q) \rightarrow (q \rightarrow p)$

$K_4 : \forall x: p(x) \rightarrow p[x|e]$, si las variables de la expresión *e* no están ligadas en el predicado p

$K_5 : \forall x: (p \rightarrow q) \rightarrow (p \rightarrow \forall x: q)$, si la variable x no está libre en p

Axiomas de la igualdad

$K_6 : x = x$

$K_7 : x = y \rightarrow y = x$

$K_8 : x = y \rightarrow (y = z \rightarrow x = z)$

$K_9 : x = y \rightarrow e(v_1, \ldots, v_{i-1}, x, v_{i+1}, \ldots, v_n) = e(v_1, \ldots, v_{i-1}, y, v_{i+1}, \ldots, v_n)$

$K_{10} : x = y \rightarrow p(v_1, \ldots, v_{i-1}, x, v_{i+1}, \ldots, v_n) = p(v_1, \ldots, v_{i-1}, y, v_{i+1}, \ldots, v_n)$, si p es atómico

Reglas de la lógica de predicados

Modus Ponens (MP): a partir de p y de $p \rightarrow q$ se infiere q

Generalización: de p se infiere $\forall x$: p

Axiomas de la aritmética

$N_1 : 0 \neq s(x)$	1er axioma del sucesor
$N_2 : s(x) = s(y) \rightarrow x = y$	2do axioma del sucesor
$N_3 : x + 0 = x$	1er axioma de la suma
$N_4 : x + s(y) = s(x + y)$	2do axioma de la suma
$N_5 : x \,.\, 0 = 0$	1er axioma de la multiplicación
$N_6 : x \,.\, s(y) = x \,.\, y + x$	2do axioma de la multiplicación
$N_7 : p(0) \rightarrow (\forall x: (p(x) \rightarrow p(s(x))) \rightarrow \forall x: p(x))$, si x no está ligada en p	inducción

El desarrollo de la prueba de la fórmula 1 + 1 = 2 en el sistema deductivo descripto es el siguiente (abreviamos algunos pasos y simplificamos algunas expresiones):

1) $x + 0 = x$	axioma N_3
2) $x + 0 = x \rightarrow 1 + 0 = 1$	axioma K_4
3) $1 + 0 = 1$	MP entre 1 y 2
4) $x + s(y) = s(x + y)$	axioma N_4
5) $x + s(y) = s(x + y) \rightarrow 1 + s(y) = s(1 + y)$	axioma K_4
6) $1 + s(y) = s(1 + y)$	MP entre 4 y 5
7) $1 + s(y) = s(1 + y) \rightarrow 1 + s(0) = s(1 + 0)$	axioma K_4
8) $1 + s(0) = s(1 + 0)$	MP entre 6 y 7
9) $x = y \rightarrow s(x) = s(y)$	axioma K_9
10) $1 + 0 = 1 \rightarrow s(1 + 0) = s(1)$	(se demuestra desde 9)
11) $s(1 + 0) = s(1)$	MP entre 3 y 10
12) $x = y \rightarrow (y = z \rightarrow x = z)$	axioma K_8
13) $1 + s(0) = s(1 + 0) \rightarrow (s(1 + 0) = s(1) \rightarrow 1 + s(0) = s(1))$	(se demuestra desde 12)
14) $s(1 + 0) = s(1) \rightarrow 1 + s(0) = s(1)$	MP entre 8 y 13
15) $1 + s(0) = s(1)$	MP entre 11 y 14
16) $1 + 1 = 2$	(por 15, abreviando s(0) y s(1))

Definiciones básicas

El ejemplo anterior sirve para introducir distintas definiciones relacionadas con los sistemas deductivos.

En la lógica de predicados distinguimos un lenguaje de fórmulas (*predicados*) de *primer orden* y un sistema deductivo conocido como *cálculo de predicados*. El lenguaje provee símbolos para las constantes, variables, funciones, relaciones, conectivos lógicos y cuantificadores (éstos aplicados sobre las variables, lo que hace que el lenguaje sea de primer orden, porque si también alcanzaran a las relaciones sería de *segundo orden*).

Un predicado es *atómico* si no incluye otros predicados (y así, tampoco contiene conectivos lógicos ni cuantificadores). Una variable está *libre* en un predicado cuando no está cuantificada, y está *ligada* en caso contrario. Por ejemplo, en el predicado p = ∀x: ∃y: (x = y + z), las variables x e *y* están ligadas y la variable z está libre. La notación p[x|e] denota la *sustitución* de todas las variables x que están libres en el predicado p por la expresión *e*. Por ejemplo, considerando el predicado anterior p, la sustitución p[z|w + 1] resulta ∀x: ∃y: (x = y + w + 1).

Los *axiomas de la igualdad* presentados antes se suelen incluir dentro de la lógica de predicados, por el uso habitual de ésta para los razonamientos matemáticos. Así lo hacemos nosotros en el libro, utilizamos la lógica de predicados *con igualdad*, tal como se la identifica en este caso.

El sistema deductivo empleado en la subsección anterior para probar la fórmula 1 + 1 = 2 muestra cómo se utiliza la lógica de predicados para analizar distintas áreas de estudio. Sobre el artefacto de los *axiomas lógicos* se montan axiomáticas especializadas, con axiomas extralógicos conocidos como *propios* (por ejemplo, los axiomas de los números naturales). Todo conjunto de axiomas lógicos y propios constituye una *teoría* (de números naturales, números enteros, números reales, números complejos, grupos, conjuntos, etc). Se dice que una teoría es una *extensión* de la lógica de predicados: con el empleo del mecanismo deductivo de la lógica de predicados permite conservar los teoremas originales y generar otros. También se pueden definir extensiones de las propias teorías.

Las pruebas en los sistemas deductivos son sintácticas, se desarrollan aplicando exclusivamente axiomas y reglas. Los axiomas son en general *esquemas de axiomas*, porque permiten infinitas instanciaciones (para simplificar, los seguimos denominando axiomas). Un sistema deductivo es *consistente* si no permite probar una fórmula y su negación. Es *independiente* si ningún axioma se deriva a partir de otros. Y es *decidible* si existe un algoritmo que establece para toda fórmula si es o no un teorema.

En el plano semántico, naturalmente un sistema deductivo, dada una interpretación, es útil sólo si sus teoremas son verdaderos, o en otras palabras, si es *sensato*. Una interpretación en la que todos los axiomas de un sistema deductivo son verdaderos es un *modelo* del sistema. En particular, un modelo es *normal* si el símbolo = se interpreta como la igualdad matemática, y es *estándar* si describe la semántica habitual de la axiomática considerada (en el libro nos basamos en los modelos estándar de los números enteros y los números naturales, que son los números

enteros y naturales como comúnmente los entendemos y utilizamos, conjuntos infinitos y ordenados {…, –2, –1, 0, 1, 2, …} y {0, 1, 2, …}, respectivamente, con las operaciones habituales de suma, multiplicación, etc., y la relación <). Otra propiedad que se espera de un sistema deductivo es la recíproca de la anterior, que sea *completo*, es decir, que permita probar, dada una interpretación, todas las fórmulas verdaderas. Por ejemplo, la teoría de los números fraccionarios con la suma y la teoría de los números complejos con la suma y la multiplicación son completas. La misma lógica de predicados es completa (en este caso sus teoremas son las fórmulas *lógicamente válidas*, que son las fórmulas verdaderas en todas las interpretaciones). En cambio, el sistema descripto antes basado en la aritmética de Peano, con el modelo estándar de los números naturales, es incompleto, existen enunciados verdaderos de la aritmética que no pueden probarse dentro del sistema

Los símbolos |– y |= sirven para distinguir lo sintáctico de lo semántico. La notación |– φ expresa que la fórmula φ se deriva del sistema deductivo de referencia, es decir sintácticamente, por uso exclusivo de sus axiomas y reglas. Y la notación |= φ expresa que la fórmula φ es verdadera, de acuerdo a la interpretación considerada. De esta manera, la *sensatez* de un sistema deductivo se puede expresar con la siguiente implicación:

$$\forall \varphi: (\vdash \varphi \longrightarrow \models \varphi)$$

y la *completitud* con la implicación recíproca:

$$\forall \varphi: (\models \varphi \longrightarrow \vdash \varphi)$$

En el libro tratamos con una *teoría de verificación de programas*, conjunto de métodos axiomáticos para probar propiedades en distintas clases de programas, interpretados en el dominio semántico de los números enteros, asumiendo la interpretación estándar. Las fórmulas utilizadas son fórmulas de correctitud, compuestas por programas y predicados lógicos. La semántica de las fórmulas se define en los capítulos siguientes. En este marco, los métodos son sensatos y completos. Por otra parte, no son decidibles, lo que impide que puedan automatizarse.

CAPÍTULO 2
Verificación de programas secuenciales determinísticos

Introducción

En este capítulo describimos un primer método axiomático de verificación de programas. Consideramos *programas secuenciales determinísticos* muy simples, como los que C. Hoare utiliza para exponer inicialmente su metodología.

Primero presentamos características generales del método: el lenguaje de programación; el lenguaje de especificación, único en la lógica de Hoare, que es el de la *lógica de predicados* (*con igualdad*); algunas definiciones y notaciones básicas; y las propiedades de programas a analizar, las dos propiedades básicas de los programas secuenciales determinísticos, la *correctitud parcial* y la *terminación*. Después describimos la axiomática correspondiente y desarrollamos ejemplos de aplicación. Sigue una sección con un ejemplo de programación sistemática basada en el método estudiado. Y completamos el capítulo con observaciones finales, notas adicionales, referencias bibliográficas (de este capítulo y de las partes precedentes) y ejercicios.

Características generales del método de verificación

Comenzamos con la descripción de los lenguajes utilizados. El lenguaje de programación es un fragmento muy simple del lenguaje Pascal (en éste y en el resto de los casos la idea es recurrir a lenguajes que proponen buenas prácticas de programación). La sintaxis del lenguaje, en la notación *Backus-Naur Form* o BNF que usamos a lo largo del libro, es la siguiente:

S :: skip | x := e | S_1 ; S_2 | if B then S_1 else S_2 fi | while B do S_1 od

La etiquetación de los programas es opcional. Los subíndices, en este caso y en los que siguen, no son parte del lenguaje, sólo se usan para facilitar las definiciones. La variable x (que representa distintas variables, como x, *y*, z, etc.) y la expresión *e* de la *asignación x := e* son de tipo entero. Esta restricción se aplica en todo el libro: en general nos limitamos al dominio

semántico de los números enteros, con la interpretación estándar, y a variables simples; excepcionalmente recurrimos a variables booleanas y a arreglos. La expresión B de las instrucciones *if then else* (*selección condicional*) y *while* (*repetición*) es de tipo booleano. La sintaxis de las expresiones es la siguiente (mostramos sólo algunas formas de expresiones):

$e :: n \mid x \mid (e_1 + e_2) \mid (e_1 - e_2) \mid (e_1 \cdot e_2) \mid \ldots \mid (\text{if } B \text{ then } e_1 \text{ else } e_2 \text{ fi})$

$B :: \text{true} \mid \text{false} \mid (e_1 = e_2) \mid (e_1 < e_2) \mid \ldots \mid \neg B_1 \mid (B_1 \vee B_2) \mid (B_1 \wedge B_2) \mid \ldots$

El símbolo n representa una constante entera (0, 1, – 2, etc.), el símbolo x una variable entera, y *true* (por verdadero) y *false* (por falso) son las constantes booleanas. Completamos la descripción del lenguaje presentando, informalmente, la semántica de sus instrucciones:

- La instrucción *skip* es atómica (se consume en un paso), y no tiene ningún efecto sobre las variables. Se puede usar, por ejemplo, para expresar una instrucción *condicional* (instrucción *if* sin *else*), no provista por el lenguaje.
- La *asignación x := e* también es atómica. Asigna el valor de *e* a la variable x.
- La *secuencia* S_1 *;* S_2 ejecuta S_1 y luego ejecuta S_2.
- La *selección condicional if B then* S_1 *else* S_2 *fi* evalúa B, si B es verdadera ejecuta S_1, y si B es falsa ejecuta S_2.
- Finalmente, la *repetición while B do S od* evalúa B, si B es verdadera ejecuta S y repite el ciclo, y si B es falsa termina.

Por ejemplo, el siguiente programa, históricamente el primero verificado con la lógica de Hoare, obtiene por medio del método de restas sucesivas el cociente c y el resto r de la división entera entre $x \geq 0$ e $y > 0$:

$S_{div} ::$ c := 0 ; r := x ; while $r \geq y$ do r := r – y ; c := c + 1 od

Describimos ahora el lenguaje de especificación, el lenguaje de la lógica de predicados. Algunas formas de predicados son las siguientes:

$p :: \text{true} \mid \text{false} \mid (e_1 = e_2) \mid (e_1 < e_2) \mid \ldots \mid \neg p_1 \mid (p_1 \vee p_2) \mid (p_1 \wedge p_2) \mid \ldots \mid \exists x: p_1 \mid \forall x: p_1$

Notar que un predicado, también conocido como *aserción* en la lógica de Hoare, puede ser directamente una expresión booleana. Para especificar un programa, el método plantea utilizar un par de predicados (p, q), la *precondición* y la *postcondición*, asociados a la entrada y la salida del programa, respectivamente. La idea es que la especificación establezca la relación que debe existir entre los contenidos de las variables al inicio y al final del programa. Por ejemplo, la especificación:

$(x = X, x = 2 \cdot X)$

es satisfecha por cualquier programa que duplica su entrada x. La variable X se conoce como *variable de especificación* (o *variable lógica*), no es una variable de programa, y está cuantificada universalmente de manera implícita, representando cualquier valor del dominio semántico. Sirve para fijar valores (lo que no es posible con las variables de programa, dado que éstas pueden modificarse a lo largo de una computación), y así su uso permite vincular a la precondición con la postcondición. En este otro ejemplo se pretende especificar un programa que debe terminar con el resultado x > y. Una posible especificación es:

$(x = X \wedge y = Y, x > y)$

y otra más simple, considerando la convención de que el predicado *true* representa cualquier condición y que en este caso no es necesario indicar los valores iniciales de x e *y*:

$(true, x > y)$

Otros elementos básicos a considerar en la descripción del método, de naturaleza semántica, son los *estados*. Constituyen las distintas configuraciones (variables con sus contenidos) por las que transita la computación de un programa. Formalmente, un estado es una función:

$\sigma : Var \rightarrow Z$

tal que Var contiene todas las variables de programa y Z es el conjunto de los números enteros (recordar que nos limitamos en general a usar variables de este dominio semántico). Cabe observar que un predicado denota un conjunto de estados. Por ejemplo, el predicado x > y mencionado antes denota el conjunto de estados en los que x es mayor que y. En particular, el predicado *true* denota el conjunto de todos los estados, y simétricamente el predicado *false* el conjunto vacío de estados (conjuntos Σ y ∅, respectivamente). Se suele utilizar la expresión:

$\sigma \models p$

para establecer que el estado σ pertenece al conjunto de estados denotado por el predicado p. El símbolo |= se asocia a lo semántico. Otra manera de decir lo anterior es que σ satisface p, y otra, que p evaluado en σ es verdadero. Por ejemplo, volviendo al predicado x > y, si en un estado σ se cumple que x = 7 e y = 5, entonces vale:

$\sigma \models x > y$

Cuantos más estados denote la precondición de un programa (en términos lógicos, cuanto más *débil* sea), mayor será el alcance de aplicación del programa. En el caso de la postcondición, reducir el conjunto de estados que denota (hacerla lógicamente más *fuerte*), en tanto no altere el resultado que se quiere expresar, produce una especificación más precisa. En particular, la *precondición más débil* y la *postcondición más fuerte* de un programa establecen, respectivamente, su máximo alcance de aplicación y su resultado más preciso.

Introducidos los componentes anteriores, llegamos a la definición de la correctitud de un programa S con respecto a una especificación (p, q), inicialmente en términos de sólo dos propiedades, las básicas de un programa secuencial determinístico. Se define que S es correcto con respecto a (p, q), o satisface (p, q), sii a partir de todo estado que satisface la precondición p, S termina y lo hace en un estado que satisface la postcondición q. Es decir, debe cumplirse:

1. A partir de todo estado que satisface p, si S termina lo hace en un estado que satisface q.
2. A partir de todo estado que satisface p, S termina (en algún estado).

La primera propiedad se denomina *correctitud parcial*. El nombre proviene de la noción de función parcial, dado que su cumplimiento se evalúa sólo si el programa termina (en caso contrario la propiedad se cumple trivialmente). La segunda propiedad es justamente la t*erminación* (también se la conoce como *no divergencia*). En este capítulo asumimos que un programa siempre termina bien, sin falla. Para ello nos basamos en ciertas asunciones (resultados predeterminados en casos como una división por cero, *overflow*, un intento de acceso con un índice fuera de rango si usáramos arreglos, etc). Lo apropiado es contemplar uno o más estados finales *de falla*, opción que reservamos para los capítulos siguientes.

En este contexto de sólo dos propiedades, cuando un programa las satisface su correctitud se dice que es *total*. La separación de las propiedades no es caprichosa, se debe a que se prueban con técnicas distintas. La prueba de correctitud parcial se basa en la técnica de inducción, mientras que la terminación no. Esto se aclara en las secciones siguientes.

Las propiedades definidas se expresan mediante fórmulas conocidas como *fórmulas de correctitud*. También se las denomina *ternas de Hoare*:

1. La fórmula {p} S {q} representa la correctitud parcial de un programa S con respecto a una especificación (p, q).
2. La fórmula ⟨p⟩ S ⟨true⟩ representa la terminación de un programa S con respecto a una precondición p.

Los delimitadores { } y ⟨ ⟩ distinguen las dos propiedades. Por lo tanto, {p} S {q} $\wedge$ ⟨p⟩ S ⟨true⟩ expresa la correctitud total de S con respecto a (p, q). La forma abreviada de la expresión es:

⟨p⟩ S ⟨q⟩

En rigor, la manera más adecuada de expresar las propiedades es anteponiendo el símbolo |= (el mismo que se utiliza para expresar la satisfacción de un predicado por parte de un estado), para indicar su naturaleza semántica. Por ejemplo, en el caso de la correctitud parcial se usa:

|= {p} S {q}

para expresar que la fórmula de correctitud {p} S {q} es verdadera (considerando el dominio semántico). Si el contexto lo amerita, hay que recurrir al símbolo |= para distinguir la satisfacción semántica de una fórmula de correctitud de su derivación en un método de prueba, en cuyo caso se usa el símbolo |–, asociado a lo sintáctico. Por ejemplo, la expresión:

|– {p} S {q}

indica que la fórmula de correctitud parcial {p} S {q} se deriva del método considerado.

Volviendo al aspecto semántico, dos ejemplos de fórmulas de correctitud verdaderas son:

{x = 0} x := x + 1 {x = 1} y ⟨x = 10⟩ while x ≠ 0 do x := x – 1 od ⟨x = 0⟩

También la fórmula siguiente es verdadera:

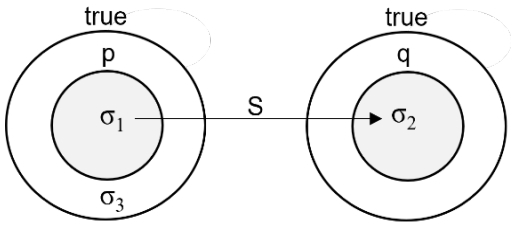
{true} S {true}

cualquiera sea el programa S, porque independientemente de su contenido, si S no termina a partir de un determinado estado no se viola la correctitud parcial. En cambio, la fórmula:

⟨true⟩ S ⟨true⟩

no lo es. Por ejemplo, a partir de cualquier estado que satisface la precondición x < 0, el programa *while x ≠ 0 do x := x – 1 od* no termina. Con que haya un estado que satisface la precondición de un programa, a partir del cual el programa no termina o termina en un estado que no satisface la postcondición, alcanza para que el programa no sea totalmente correcto con respecto a su especificación. Nada se puede afirmar, en cambio, cuando el estado inicial no satisface la precondición, porque el estado está fuera del alcance de aplicación del programa.

La siguiente figura ilustra lo que se requiere para el cumplimiento de la correctitud total de un programa S con respecto a una especificación (p, q):



El predicado *true* denota el conjunto de todos los estados. Desde un estado σ_1 del conjunto de estados denotado por la precondición p, S debe terminar en un estado σ_2 del conjunto de estados denotado por la postcondición q. No interesa el comportamiento de S a partir de un estado σ_3 que está fuera del alcance de p.

Dejamos para las próximas dos secciones la descripción del componente central del método, su axiomática, que permite probar sintácticamente las fórmulas de correctitud. Primero tratamos la parte dedicada a la correctitud parcial, y luego la parte correspondiente a la terminación.

Axiomática para las pruebas de correctitud parcial

Emulando el cálculo de la lógica, el método se basa en axiomas y reglas que proveen el significado, la semántica de las instrucciones, haciéndolas corresponder con pre y postcondiciones. Los axiomas se refieren a las instrucciones atómicas y las reglas a las instrucciones compuestas.

Siendo un programa S una secuencia de instrucciones especificada por un par de predicados (p, q), aplicando las reglas instrucción por instrucción, y valiéndonos de la propiedad de *composicionalidad* del método, llegamos a probar la fórmula {p} S {q} (lo mismo vale para la prueba de terminación que analizamos en la próxima sección). Más precisamente, cada instrucción se comporta como un *transformador de predicados* (y por lo tanto de estados), y así el programa completo actúa como tal.

Que el método sea composicional significa que la fórmula {p} S {q} se puede probar componiendo de determinada manera fórmulas $\{p_1\}\ S_1\ \{q_1\}$, $\{p_2\}\ S_2\ \{q_2\}$, …, $\{p_n\}\ S_n\ \{q_n\}$ probadas previamente, ignorando el contenido de los subprogramas S_i, que son los componentes sintácticos inmediatos del programa S. Es decir que los S_i se pueden manejar como cajas negras. Además, una vez verificado un componente, éste puede ser ignorado en lo que resta de la prueba. Y particularmente en lo que hace a la construcción y mantenimiento de programas, la composicionalidad permite reemplazar un componente por otro de igual especificación, y dejar *agujeros sintácticos* especificados a implementar oportunamente.

Axiomas y reglas

Los axiomas y reglas del método para probar la correctitud parcial son los siguientes:

1. *Axioma del skip* (SKIP)
   {p} skip {p}

El axioma establece que la instrucción *skip* no tiene efecto alguno sobre el estado inicial y que consume un solo paso. Así, si un predicado se cumple antes de su ejecución, sigue valiendo después.

2. *Axioma de la asignación* (ASI)
   {p[x|e]} x := e {p}

El axioma establece que si se cumple el predicado p en términos de x después de la ejecución de la asignación *x := e*, entonces antes de la asignación se cumplía p en términos de la expresión *e*. Con p[x|e] se denota el reemplazo de toda ocurrencia libre de la variable x en el predicado p por la expresión *e*. Por ejemplo, aplicando el axioma se puede establecer lo siguiente:

$\{x + 1 \geq 0\}\ x := x + 1\ \{x \geq 0\}$

Notar que el axioma se enuncia de derecha a izquierda, lo cual sugiere una manera de probar de la postcondición a la precondición. Está visión coincide con la del cálculo de programas de E. Dijkstra: partiendo del resultado que se espera de un programa, se lo construye por medio de transformaciones de predicados hasta llegar a la *precondición más débil*, que representa el máximo conjunto posible de estados iniciales. Resulta más natural plantear una forma de axioma que se enuncie de izquierda a derecha, como:

$\{true\}\ x := e\ \{x = e\}$

pero esta fórmula puede ser falsa si la expresión *e* incluye a la variable x. Es el caso, por ejemplo, de e = x + 1, porque queda:

$\{true\}\ x := x + 1\ \{x = x + 1\}$

El problema radica en que la x de la parte derecha de la postcondición se refiere a la variable antes de la asignación, mientras que la x de la parte izquierda se refiere a la variable luego de la asignación. Un axioma correcto enunciado de izquierda a derecha es:

$\{p\}\ x := e\ \{\exists z: (p[x|z] \wedge x = e[x|z])\}$

Nos quedamos con la primera versión, más simple, y además la más utilizada. En una de las notas adicionales nos referimos a la aplicación del axioma sobre variables de tipo arreglo.

3. *Regla de la secuencia* (SEC)

$$\frac{\{p\}\ S_1\ \{r\},\ \{r\}\ S_2\ \{q\}}{\{p\}\ S_1\ ;\ S_2\ \{q\}}$$

La regla plantea que del cumplimiento de $\{p\}\ S_1\ \{r\}$ y $\{r\}\ S_2\ \{q\}$ se deriva el cumplimiento de $\{p\}\ S_1\ ;\ S_2\ \{q\}$, sin el predicado intermedio r, el cual actúa como nexo de la secuencia. La regla se puede generalizar a cualquier cantidad de premisas:

$$\frac{\{p\}\ S_1\ \{r_1\},\ \{r_1\}\ S_2\ \{r_2\},\ \ldots,\ \{r_{n-1}\}\ S_n\ \{q\}}{\{p\}\ S_1\ ;\ S_2\ ;\ \ldots\ ;\ S_n\ \{q\}}$$

4. *Regla del condicional* (COND)

$$\frac{\{p \wedge B\}\ S_1\ \{q\},\ \{p \wedge \neg B\}\ S_2\ \{q\}}{\{p\}\ \text{if B then } S_1 \text{ else } S_2 \text{ fi } \{q\}}$$

La regla establece que si se cumplen las premisas $\{p \wedge B\}\ S_1\ \{q\}$ y $\{p \wedge \neg B\}\ S_2\ \{q\}$, entonces a partir de la precondición p, se cumpla o no la condición B, luego de la ejecución respectiva del subprograma S_1 o el subprograma S_2 vale la postcondición q. Se impone así un modo de verificar una selección condicional con un único punto de entrada y un único punto de salida.

5. *Regla de la repetición* (REP)

$$\frac{\{p \wedge B\}\ S\ \{p\}}{\{p\}\ \text{while B do S od } \{p \wedge \neg B\}}$$

La regla utiliza un predicado *invariante* p, que se cumple inicialmente y después de toda iteración del *while*. Se lee así: si la ejecución del cuerpo del *while* preserva p mientras vale la condición B, entonces al terminar el *while* se cumple $p \wedge \neg B$. Se establece una prueba por inducción (en este contexto se la conoce como inducción *computacional*): si se prueba que p vale al comienzo del *while*, y que toda iteración del *while* preserva p, entonces se prueba en definitiva que p vale a lo largo de toda la computación correspondiente. Notar que la regla no asegura la terminación del *while*.

La última regla del método es de otra naturaleza:

6. *Regla de consecuencia* (CONS)

$$\frac{p \rightarrow p_1,\ \{p_1\}\ S\ \{q_1\},\ q_1 \rightarrow q}{\{p\}\ S\ \{q\}}$$

Efectivamente, esta regla no se relaciona con ninguna instrucción sino con el dominio semántico. Su uso implica recurrir directamente a enunciados verdaderos, como se observa en

dos de las premisas, que no son otra cosa que axiomas de los números enteros, los cuales entonces también forman parte del método. En términos lógicos, la regla permite reforzar precondiciones y debilitar postcondiciones de las fórmulas, preservando su correctitud. Por ejemplo, si vale:

$r \rightarrow p$ y $\{p\}\ S\ \{q\}$

entonces aplicando la regla se puede derivar la fórmula:

$\{r\}\ S\ \{q\}$

que claramente es verdadera por la semántica de la implicación y la definición de correctitud parcial.

En realidad, el método incluye una séptima regla de prueba, de naturaleza semántica como la anterior. Suele omitirse en las presentaciones, aunque se asume su uso. Es la siguiente:

7. *Regla de instanciación* (INST):

$$\frac{f(X)}{f(c)}$$

Esta regla es necesaria para instanciar fórmulas de correctitud. X es una variable de especificación, representa cualquier valor del dominio semántico, y c varía en dicho dominio. La aplicación de la regla permite sustituir la fórmula f(X) por la fórmula f(c), como por ejemplo:

$\{x = X\}\ x := x + 1\ \{x = X + 1\}$

por:

$\{x = 5\}\ x := x + 1\ \{x = 5 + 1\}$

haciendo X = 5. La regla tiene una restricción: si X aparece en la pre y postcondición no puede instanciarse con una expresión que incluya una variable de programa. Por ejemplo, si X = x, la fórmula anterior queda $\{x = x\}\ x := x + 1\ \{x = x + 1\}$, que es falsa.

En lo que sigue ejemplificamos cómo utilizar los axiomas y reglas.

Ejemplo 1. Prueba de correctitud parcial de un programa que intercambia los valores de dos variables

El programa siguiente intercambia los contenidos de las variables x e *y*:

S_{swap} :: z := x ; x := y ; y := z

Vamos a probar $\{x = X \wedge y = Y\}\ S_{swap}\ \{y = X \wedge x = Y\}$.

Por la forma del programa, recurrimos al axioma ASI tres veces, una por cada asignación, y al final completamos la prueba utilizando la regla SEC:

1) $\{z = X \wedge x = Y\}\ y := z\ \{y = X \wedge x = Y\}$ (ASI)
2) $\{z = X \wedge y = Y\}\ x := y\ \{z = X \wedge x = Y\}$ (ASI)
3) $\{x = X \wedge y = Y\}\ z := x\ \{z = X \wedge y = Y\}$ (ASI)
4) $\{x = X \wedge y = Y\}\ z := x ; x := y ; y := z\ \{y = X \wedge x = Y\}$ (1, 2, 3, SEC)

Lo mismo que en las pruebas lógicas, enumeramos los pasos e indicamos en cada uno qué axioma o regla utilizamos y sobre qué pasos anteriores se aplica. Notar cómo el axioma ASI impone una forma de prueba de la postcondición a la precondición.

Supongamos ahora que en lugar de la fórmula obtenida en el paso (4) buscamos una fórmula equivalente, en la que la conjunción de la precondición aparezca permutada, es decir:

$\{y = Y \wedge x = X\}\ z := x ; x := y ; y := z\ \{y = X \wedge x = Y\}$

En este caso tenemos que recurrir necesariamente a la regla CONS, extendiendo la prueba anterior de la siguiente manera:

5) $(y = Y \wedge x = X) \rightarrow (x = X \wedge y = Y)$ (MAT)
6) $\{y = Y \wedge x = X\}\ z := x ; x := y ; y := z\ \{y = X \wedge x = Y\}$ (4, 5, CONS)

El paso (5) introduce directamente un enunciado verdadero del dominio semántico, por eso se justifica con el indicador MAT (matemática), y la regla CONS aplicada en el paso (6) es la que permite obtener la fórmula buscada, sustituyendo una precondición por otra.

Las variables de especificación X e Y representan cualquier valor entero. Si queremos instanciar la fórmula obtenida en el paso (6) con valores específicos, tenemos que aplicar la regla INST. Por ejemplo, si X = 1 e Y = 2, queda:

7) $\{y = 2 \wedge x = 1\}\ z := x ; x := y ; y := z\ \{y = 1 \wedge x = 2\}$ (6, INST)

Ejemplo 2. Prueba de correctitud parcial de un programa que calcula el valor absoluto

El programa siguiente obtiene en la variable *y* el valor absoluto de la variable x:

S_{va} :: if x > 0 then y := x else y := – x fi

Vamos a probar {true} S_{va} {y ≥ 0}.

Notar que (true, y ≥ 0) no es una especificación correcta de un programa que calcula el valor absoluto. Cualquier programa que termine con y ≥ 0, por ejemplo la simple asignación *y := 0*, satisface la especificación y no es el programa buscado. Elegimos una fórmula muy sencilla para simplificar el ejemplo de aplicación de la regla COND. Comenzamos la prueba considerando las dos asignaciones del programa y la postcondición y ≥ 0:

1) {x ≥ 0} y := x {y ≥ 0} (ASI)
2) {– x ≥ 0} y := – x {y ≥ 0} (ASI)

Las dos fórmulas obtenidas no alcanzan para poder aplicar la regla COND. Necesitamos dos premisas que tengan, una la forma p ∧ B y la otra la forma p ∧ ¬B, siendo B la expresión booleana de la selección condicional, en este caso x > 0. Tomando como p la precondición *true* del programa podemos completar la prueba. Para obtener {true ∧ x > 0} y := x {y ≥ 0} hacemos:

3) (true ∧ x > 0) → x ≥ 0 (MAT)
4) {true ∧ x > 0} y := x {y ≥ 0} (1, 3, CONS)

Y para obtener {true ∧ ¬(x > 0)} y := – x {y ≥ 0} hacemos :

5) (true ∧ ¬(x > 0)) → – x ≥ 0 (MAT)
6) {true ∧ ¬(x > 0)} y := – x {y ≥ 0} (2, 5, CONS)

Finalmente, con las formulas obtenidas en los pasos (4) y (6) aplicamos la regla COND y llegamos a la fórmula buscada:

7) {true} if x > 0 then y := x else y := – x fi {y ≥ 0} (4, 6, COND)

Notar nuevamente la utilidad de la regla CONS. Actúa como interfaz entre las fórmulas de correctitud y las fórmulas verdaderas del dominio semántico en que se interpretan los programas (enunciados sobre los números enteros).

Ejemplo 3. Prueba de correctitud parcial del programa que calcula la división entera

Completamos los ejemplos de esta sección con la verificación del programa de división entera presentado previamente, en la que se aplica la regla REP. Dado el programa:

S_{div} :: c := 0 ; r := x ; while r ≥ y do r := r – y ; c := c + 1 od

vamos a probar $\{x \geq 0 \wedge y > 0\}\ S_{div}\ \{x = y \,.\, c + r \wedge r \geq 0 \wedge r < y\}$.

Nuevamente simplificamos la especificación. En este caso la imprecisión es que las variables de entrada, el dividendo x y el divisor *y*, pueden modificarse a lo largo del programa. Por ejemplo, el programa que asigna 0 a las variables x, c y r satisface la especificación y no es el programa buscado.

Para probar el *while*, la regla REP requiere utilizar un predicado invariante que valga antes de la instrucción y también después de cada iteración. Proponemos el invariante:

$p = (x = y \,.\, c + r \wedge r \geq 0)$

obtenido debilitando la postcondición (se le quita la condición r < y que se produce al final del *while*). El invariante refleja la idea del algoritmo, la invariancia de la igualdad x = c . y + r a partir de las inicializaciones del cociente c en 0 y el resto r en x: después de cada iteración, c se incrementa en 1 y r se decrementa en *y*, que es la manera en que se implementa la división entera por restas sucesivas. La prueba es más compleja que las anteriores. Por eso, para facilitar su desarrollo, la estructuramos en tres partes. Además, para acortar su longitud, en la aplicación de la regla CONS damos por sobrentendido el enunciado matemático utilizado (práctica que mantenemos de ahora en más). El plan de prueba es el siguiente:

a) Probamos que las inicializaciones del programa conducen al invariante por primera vez:
$\{x \geq 0 \wedge y > 0\}$ c := 0 ; r := x $\{x = y \,.\, c + r \wedge r \geq 0\}$

b) Probamos que el predicado propuesto como invariante efectivamente lo es, y de esta manera probamos el *while*:
$\{x = y \,.\, c + r \wedge r \geq 0\}$ while r ≥ y do r := r – y ; c := c + 1 od $\{x = y \,.\, c + r \wedge r \geq 0 \wedge \neg(r \geq y)\}$

c) Aplicando las reglas SEC y CONS sobre lo anterior se completa la prueba:
$\{x \geq 0 \wedge y > 0\}\ S_{div}\ \{x = y \,.\, c + r \wedge r \geq 0 \wedge r < y\}$

Prueba de (a)

1) $\{x = y \,.\, c + x \wedge x \geq 0\}$ r := x $\{x = y \,.\, c + r \wedge r \geq 0\}$ (ASI)

2) $\{x = y \,.\, 0 + x \wedge x \geq 0\}\ c := 0\ \{x = y \,.\, c + x \wedge x \geq 0\}$ (ASI)

3) $\{x = y \,.\, 0 + x \wedge x \geq 0\}\ c := 0\ ;\ r := x\ \{x = y \,.\, c + r \wedge r \geq 0\}$ (1, 2, SEC)

4) $\{x \geq 0 \wedge y > 0\}\ c := 0\ ;\ r := x\ \{x = y \,.\, c + r \wedge r \geq 0\}$ (3, CONS)

Prueba de (b)

5) $\{x = y \,.\, (c + 1) + r \wedge r \geq 0\}\ c := c + 1\ \{x = y \,.\, c + r \wedge r \geq 0\}$ (ASI)

6) $\{x = y \,.\, (c + 1) + (r - y) \wedge (r - y) \geq 0\}\ r := r - y\ \{x = y \,.\, (c + 1) + r \wedge r \geq 0\}$ (ASI)

7) $\{x = y \,.\, (c + 1) + (r - y) \wedge (r - y) \geq 0\}\ r := r - y\ ;\ c := c + 1\ \{x = y \,.\, c + r \wedge r \geq 0\}$ (5, 6, SEC)

8) $\{x = y \,.\, c + r \wedge r \geq 0 \wedge r \geq y\}\ r := r - y\ ;\ c := c + 1\ \{x = y \,.\, c + r \wedge r \geq 0\}$ (7, CONS)

9) $\{x = y \,.\, c + r \wedge r \geq 0\}$ while $r \geq y$ do $r := r - y\ ;\ c := c + 1$ od $\{x = y \,.\, c + r \wedge r \geq 0 \wedge \neg(r \geq y)\}$ (8, REP)

Prueba de (c)

10) $\{x \geq 0 \wedge y > 0\}\ S_{div}\ \{x = y \,.\, c + r \wedge r \geq 0 \wedge \neg(r \geq y)\}$ (4, 9, SEC)

11) $\{x \geq 0 \wedge y > 0\}\ S_{div}\ \{x = y \,.\, c + r \wedge r \geq 0 \wedge r < y\}$ (10, CONS)

La fórmula obtenida en el paso (11) establece que si el programa termina lo hace con el resultado esperado. La prueba no asegura la terminación del programa, para ello se necesita una nueva regla, que presentamos en la sección siguiente.

Por lo observado, en el marco de las pruebas *a posteriori*, es decir asumiendo la existencia de programas ya construidos (lo que es insoslayable de acuerdo al método descripto), queda claro que el paso más difícil, el que requiere mayor creatividad, es el de la búsqueda de un invariante de un *while*. Desde el punto de vista de la programación, la noción de invariante como abstracción de una instrucción de repetición a ser implementada a partir del mismo, es un aporte fundamental del método axiomático para la obtención de programas correctos, asumiendo la construcción de los programas en simultáneo con sus pruebas de correctitud.

Definiciones y axiomas y reglas adicionales

En los ejemplos anteriores mostramos cómo derivar fórmulas de correctitud parcial empleando exclusivamente los axiomas y reglas descriptos, es decir sintácticamente (en verdad, también manipulando enunciados verdaderos de los números enteros). Por ejemplo, derivamos:

$\vdash \{true\}$ if $x > 0$ then $y := x$ else $y := -x$ fi $\{y \geq 0\}$

Pero lo que nos interesa es probar que las fórmulas son verdaderas, que se cumplen semánticamente, de acuerdo a la definición de correctitud parcial (y al dominio semántico considerado). Volviendo al ejemplo anterior, lo que buscamos entonces es:

$\models \{true\}$ if $x > 0$ then $y := x$ else $y := -x$ fi $\{y \geq 0\}$

Por lo tanto, el método estudiado será útil sólo si las fórmulas que permite derivar son verdaderas. Formalmente, si cumple:

$\vdash \{p\}\ S\ \{q\} \rightarrow \models \{p\}\ S\ \{q\}$, para todo programa S y toda especificación (p, q)

Un método que tiene esta propiedad es *sensato*. La *sensatez* es imprescindible, obviamente un método no sensato no tiene razón de ser. Por otro lado, un método es *completo* si goza de la propiedad inversa, si permite derivar todas las fórmulas verdaderas, es decir si cumple:

$\models \{p\}\ S\ \{q\} \rightarrow \vdash \{p\}\ S\ \{q\}$, para todo programa S y toda especificación (p, q)

La *completitud* de un método axiomático es deseable, para que su alcance de aplicación no esté restringido. No siempre se cumple, sobre todo cuando el lenguaje de programación tiene estructuras muy complicadas.

En nuestro caso, el método presentado es sensato, y con ciertas salvedades también completo (lo probamos en el capítulo 4). A pesar de la completitud, como se acostumbra en cualquier sistema deductivo el método tiene más reglas, para simplificar el desarrollo de las pruebas. Presentamos algunas, respetando la numeración anterior:

8. *Regla de la disyunción* (OR)

$$\frac{\{p\}\ S\ \{q\},\ \{r\}\ S\ \{q\}}{\{p \vee r\}\ S\ \{q\}}$$

La regla permite una prueba por casos, evitando propagar información para establecer oportunamente una disyunción en la precondición. Se puede generalizar a más premisas. Una forma particular de la regla, muy común, es:

$$\frac{\{p \wedge r\}\ S\ \{q\},\ \{p \wedge \neg r\}\ S\ \{q\}}{\{p\}\ S\ \{q\}}$$

9. *Regla de la conjunción* (AND)

$$\frac{\{p_1\}\ S\ \{q_1\},\ \{p_2\}\ S\ \{q_2\}}{}$$

$\{p_1 \wedge p_2\}\ S\ \{q_1 \wedge q_2\}$

La regla permite una prueba incremental, evitando propagar información para establecer oportunamente conjunciones. También en este caso la regla se puede generalizar a más premisas.

10. *Axioma de invariancia* (AINV)
    {p} S {p}, si ninguna variable libre de p puede ser modificada por S

La correctitud del axioma es obvia. Su empleo es más habitual en combinación con la regla AND, dando lugar a la siguiente regla:

11. *Regla de invariancia* (RINV):

$$\frac{\{p\}\ S\ \{q\}}{\{r \wedge p\}\ S\ \{r \wedge q\}}$$

si ninguna variable libre de r puede ser modificada por S

Otra propiedad comúnmente analizada en los métodos axiomáticos es la *consistencia*, que se cumple si en ningún caso se puede derivar una fórmula y su negación. No la consideramos porque la sintaxis de las fórmulas de correctitud no incluye su negación.

Una presentación alternativa de una prueba es la *proof outline* (para la verificación de los programas concurrentes esta alternativa es imprescindible). Consiste en intercalar los predicados utilizados en la prueba (algunos o todos) entre las instrucciones del programa considerado. Una *proof outline* de {p} S {q} se denota con {p} S* {q}. Se obtiene así una prueba más estructurada, y que documenta adecuadamente el programa. Por ejemplo, la siguiente es una *proof outline* de correctitud parcial del programa de división entera:

$\{x \geq 0 \wedge y > 0\}$
c := 0 ; r := x ;
$\{x = y \cdot c + r \wedge r \geq 0\}$
while $r \geq y$ do
$\{x = y \cdot c + r \wedge r \geq 0 \wedge r \geq y\}$
r := r – y ; c := c + 1
$\{x = y \cdot c + r \wedge r \geq 0\}$
od
$\{x = y \cdot c + r \wedge r \geq 0 \wedge r < y\}$

Una *proof outline* muestra qué predicados se cumplen en qué lugares de un programa (en los que se comportan entonces como invariantes, porque siempre se cumplen en esos lugares,

independientemente del estado inicial del programa). Es común insertar sólo los predicados relevantes, mínimamente la precondición, los invariantes de las repeticiones y la postcondición.

Axiomática para las pruebas de terminación

La regla REP no sirve para probar la terminación del *while*, única posible fuente de computación infinita. Para esta prueba el método tiene otra regla, definida a partir de la regla REP, con dos premisas más: la *regla de la terminación*. Mantiene la noción de un componente invariante p, y agrega la de un componente *variante* t, función entera definida, como el invariante, en términos de las variables de programa. La forma de la regla es la siguiente (continuamos con la numeración de la sección anterior, y utilizamos los delimitadores ⟨ ⟩ en lugar de { }):

12. *Regla de la terminación* (REP*)

$$\frac{\langle p \wedge B\rangle\ S\ \langle p\rangle,\ \langle p \wedge B \wedge t = Z\rangle\ S\ \langle t < Z\rangle,\ p \rightarrow t \geq 0}{\langle p\rangle \text{ while } B \text{ do } S \text{ od } \langle p \wedge \neg B\rangle}$$

Z es una variable de especificación, no aparece en p, B, S ni t. Su objetivo es resguardar el valor de t previo a la ejecución de S. La primera premisa es la misma que la de la regla REP. Las otras dos premisas son las que aseguran la terminación:

- Por la segunda premisa, la función t se decrementa después de cada iteración.
- Por la tercera premisa, t arranca y se mantiene positiva a lo largo de todo el *while*.

En tales condiciones el *while* debe terminar, porque los valores enteros de t se reducen de iteración en iteración y nunca son negativos, y lo hace obviamente con la postcondición $p \wedge \neg B$. La relación entre el invariante p y el variante t es la siguiente: mientras vale $p \wedge B$, la ejecución de S preserva p y decrementa t, y además p asegura que t nunca se hace negativo. El valor inicial de t representa la cantidad máxima de iteraciones del *while*, y su decremento progresivo indica el acercamiento al evento de finalización de la instrucción.

El agregado de la regla REP* no altera la sensatez y completitud de la axiomática.

Como se observa, la terminación no se prueba por inducción, no hay una noción de propiedad que se preserva a lo largo de una computación, como en el caso de la prueba de correctitud parcial. En este caso la prueba se fundamenta en la inexistencia de cadenas descendentes infinitas en el dominio N de los números naturales con la relación < y la interpretación estándar, comúnmente denotado con $(N_0, <)$, que es un ejemplo de *conjunto bien fundado*: conjunto con una relación de orden parcial estricto sin cadenas descendentes infinitas.

Notar también que la regla REP* permite verificar directamente $\langle p\rangle$ while B do S od $\langle p \wedge \neg B\rangle$, pero en realidad su aplicación siempre implica llevar a cabo dos pruebas, una prueba por

inducción determinada por la primera premisa y otra prueba que utiliza las dos premisas restantes basada en un conjunto bien fundado. La recomendación es partir la prueba en dos, por un lado probar {p} while B do S od {p $\wedge \neg$B} y por el otro probar ⟨p⟩ while B do S od ⟨true⟩, incluso utilizando invariantes distintos, porque el invariante para la prueba de terminación puede ser más simple al no tener que incluir toda la información requerida para la prueba de correctitud parcial.

A continuación, ejemplificamos el uso de la nueva regla.

Ejemplo 4. Prueba de terminación del programa que calcula la división entera

Completamos la prueba de correctitud total del programa de división entera. Ya probamos:

{x $\geq$ 0 $\wedge$ y > 0} c := 0 ; r := x ; while r $\geq$ y do r := r – y ; c := c + 1 od {x = y . c + r $\wedge$ r $\geq$ 0 $\wedge$ r < y}

Vamos a probar ⟨x $\geq$ 0 $\wedge$ y > 0⟩ c := 0 ; r := x ; while r $\geq$ y do r := r – y ; c := c + 1 od ⟨true⟩.

Como antes, estructuramos la prueba separando la parte de las inicializaciones de la parte del *while* y utilizando su invariante como nexo. Si bien los axiomas y reglas que se emplean son los de la sección anterior salvo la nueva regla REP*, los anotamos con los delimitadores ⟨ ⟩ en lugar de { } para explicitar que la prueba es de terminación. El plan de prueba es el siguiente:

a) ⟨x $\geq$ 0 $\wedge$ y > 0⟩ c := 0 ; r := x ⟨p⟩, siendo p el invariante a encontrar.
b) ⟨p⟩ while r $\geq$ y do r := r – y ; c := c + 1 od ⟨true⟩
c) Aplicando la regla SEC sobre lo anterior se completa la prueba.

Además del invariante, para probar el *while* la regla REP* requiere encontrar un variante (retomando lo comentado en la sección anterior, desde el punto de vista de una prueba en simultáneo con la programación, primero hay que definir el invariante, ahora también el variante, y luego plasmar ámbos en una instrucción de repetición basada en la información que brindan).

Como invariante proponemos el predicado p = (r $\geq$ 0 $\wedge$ y > 0); notar que es más simple que el predicado x = y . c + r $\wedge$ r $\geq$ 0 que utilizamos para la prueba de correctitud parcial. Y como variante proponemos la función t = r, teniendo en cuenta que el valor de r arranca con x $\geq$ 0 y se decrementa en cada iteración en y > 0 unidades mientras no sea menor que *y*, por lo que en algún momento alcanza la condición r < y, la cual provoca la terminación del *while*.

La prueba de la parte (a), del fragmento de inicializaciones, la omitimos, se logra fácilmente utilizando el axioma ASI y la regla SEC.

Para probar la parte (b) hay que recurrir a la regla REP*. Hay que probar que a lo largo del *while* se preserva el invariante, y que el variante se decrementa iteración tras iteración y nunca se hace negativo, es decir:

1) $\langle r \geq 0 \wedge y > 0 \wedge r \geq y \rangle$ r := r – y ; c := c + 1 $\langle r \geq 0 \wedge y > 0 \rangle$
2) $\langle r \geq 0 \wedge y > 0 \wedge r \geq y \wedge r = Z \rangle$ r := r – y ; c := c + 1 $\langle r < Z \rangle$
3) $(r \geq 0 \wedge y > 0) \rightarrow r \geq 0$

También omitimos la prueba de esta parte, las tres fórmulas se obtienen fácilmente empleando el axioma ASI, las reglas SEC y CONS, y manipulando algunos enunciados verdaderos de los números enteros. De esta manera probamos la terminación del *while*:

$\langle r \geq 0 \wedge y > 0 \rangle$ while r ≥ y do r := r – y ; c := c + 1 od $\langle r \geq 0 \wedge y > 0 \wedge \neg(r \geq y) \rangle$

Y por aplicación de las reglas SEC y CONS, la terminación del programa completo:

$\langle x \geq 0 \wedge y > 0 \rangle$ c := 0 ; r := x ; while r ≥ y do r := r – y ; c := c + 1 od $\langle true \rangle$

El programa es totalmente correcto considerando la precondición $x \geq 0 \wedge y > 0$. Se puede probar que con la precondición $x \geq 0 \wedge y \geq 0$ el programa es parcialmente correcto (no totalmente correcto porque partiendo de y = 0 no termina). El predicado $x \geq 0 \wedge y \geq 0$ se conoce como la precondición *liberal* más débil (asegura correctitud parcial pero no terminación).

Definiciones adicionales

Hay una clasificación que considera dos familias de propiedades de programas, propiedades *safety*, que se prueban por inducción y se refieren a situaciones no deseables, que no pueden suceder (entre ellas, que no se cumpla la correctitud parcial), y propiedades *liveness*, que se prueban en base a conjuntos bien fundados y se refieren a situaciones deseables, que deben suceder (como la terminación). De esta manera, entre la sección anterior y ésta cubrimos el tratamiento de una propiedad de cada familia.

Las *proof outlines* de terminación y de correctitud total se definen de la misma manera que las de correctitud parcial, salvo que utilizan delimitadores ⟨ ⟩ en lugar de { } e incluyen los variantes de las repeticiones. Una *proof outline* de ⟨p⟩ S ⟨q⟩ se denota con ⟨p⟩ S** ⟨q⟩.

Por ejemplo, la siguiente *proof outline* es una *proof outline* de terminación del programa de división entera:

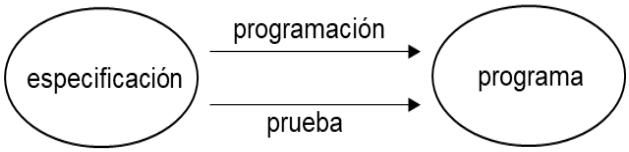
$\langle x \geq 0 \wedge y > 0 \rangle$
c := 0 ; r := x ;

⟨inv: $r \geq 0 \wedge y > 0$, var: r⟩
while $r \geq y$ do
⟨$r \geq 0 \wedge y > 0 \wedge r \geq y$⟩
r := r – y ; c := c + 1
⟨$r \geq 0 \wedge y > 0$⟩
od
⟨true⟩

Los predicados de la prueba de terminación no están incluidos, sólo figura el variante utilizado. Es lo habitual, la prueba se puede reconstruir fácilmente a partir de la información suministrada.

Ejemplo de desarrollo sistemático de programa

El método axiomático de verificación de programas presentado identifica principios fundamentales de correctitud de programas. Su descripción en términos de pruebas *a posteriori* resulta muy didáctica. Pero en la práctica de la programación, probar un programa ya construido es más difícil (y menos natural) que desarrollarlo e ir verificándolo al mismo tiempo. En efecto, el proceso más apropiado en este caso tiene esta segunda forma, a la que ya nos referimos en las secciones anteriores. La idea es encontrar primero la estructura de una prueba de correctitud del programa con respecto a su especificación, y después construirlo con la guía de los requerimientos establecidos por la prueba, programación y prueba avanzando simultáneamente:



En lo que resta de la sección ejemplificamos el desarrollo sistemático de un programa teniendo en cuenta esta idea. Antes de presentar el ejemplo con un programa concreto, planteamos un plan de desarrollo para un programa genérico con la guía del método axiomático. Consideramos un programa con una estructura muy sencilla, un fragmento de inicializaciones seguido de un *while*:

S :: S_1 ; while B do S_2 od

que debe ser totalmente correcto con respecto a una especificación determinada (r, q), o sea que debe satisfacer la fórmula:

⟨r⟩ S ⟨q⟩

Hay que construir el fragmento inicial S_1 y el *while*. De acuerdo al método, se tienen que encontrar un predicado p y una función t, invariante y variante del *while*, respectivamente, y se deben cumplir cinco requerimientos:

1) A partir de la precondición r, el fragmento S_1 termina estableciendo el predicado p: $\langle r \rangle\ S_1\ \langle p \rangle$.
2) El predicado p es un invariante del *while*: $\langle p \wedge B \rangle\ S_2\ \langle p \rangle$, y así: $\langle p \rangle$ while B do S_2 od $\langle p \wedge \neg B \rangle$.
3) La función t decrece después de cada iteración del *while*: $\langle p \wedge B \wedge t = Z \rangle\ S_2\ \langle t < Z \rangle$.
4) El predicado p asegura que la función t siempre es positiva: $p \rightarrow t \geq 0$.
5) El *while* termina estableciendo la postcondición q: $(p \wedge \neg B) \rightarrow q$.

La siguiente *proof outline* genérica de correctitud total reúne los requerimientos de una manera más conveniente para seguirla como guía de desarrollo:

$\langle r \rangle\ S_1$; $\langle$inv: p, var: t$\rangle$ while B do $\langle p \wedge B \rangle\ S_2\ \langle p \rangle$ od $\langle p \wedge \neg B \rangle\ \langle q \rangle$

En el ejemplo que presentamos a continuación desarrollamos sistemáticamente un programa concreto teniendo en cuenta dicha guía.

Ejemplo 5. Desarrollo sistemático de un programa que suma los elementos de un arreglo de números enteros

Permitiendo el uso de arreglos, vamos a construir un programa con la forma:

S_1 ; while B do S_2 od

que sume en una variable x los elementos de un arreglo a[0:N – 1] de números enteros, tal que $N \geq 0$. El arreglo es de solo lectura. Si N = 0, se define por convención que la suma es 0. Planteamos la siguiente especificación:

$(N \geq 0, x = \Sigma_{i=0,N-1}\ a[i])$

El primer paso es definir un invariante del *while*. La heurística habitual es debilitar la postcondición. En el ejemplo 3 de este capítulo la usamos quitando un componente de una conjunción. Ahora recurrimos a otra variante, que consiste en reemplazar una constante por una variable. Proponemos:

$p = (0 \leq k \leq N \wedge x = \Sigma_{i=0,k-1}\ a[i])$

La variable k reemplaza a la constante N en la sumatoria. El invariante refleja la idea del algoritmo, de ir sumando en la variable x los elementos del arreglo a lo largo de un *while*, recorrido por un índice k. El rango de k sugiere la condición de terminación del *while*.

El paso siguiente es la construcción del fragmento inicial del programa, cuya ejecución debe establecer el invariante por primera vez (requerimiento 1). Hacemos:

$k := 0 ; x := 0$

Las asignaciones determinan que al inicio k apunte al primer elemento del arreglo y que el valor inicial de la suma sea 0. Aplicando el axioma ASI y las reglas SEC y CONS el requerimiento 1 se cumple:

$\langle N \geq 0 \rangle\ k := 0 ; x := 0\ \langle 0 \leq k \leq N \wedge x = \Sigma_{i=0,k-1}\ a[i] \rangle$

Continuamos ahora con la construcción del *while*. Lo hacemos de modo tal que su cuerpo S_2 preserve el invariante (requerimiento 2), por lo que ya asumimos que el requerimiento se cumple. Teniendo en cuenta lo mencionado previamente, proponemos como variante del *while* la función:

$t = N - k$

incluyendo además dentro del *while* la asignación:

$k := k + 1$

e instanciando su condición B con la expresión:

$k \neq N$

La asignación *k := k + 1* determina que el arreglo se recorra en el orden a[0], a[1], etc. También produce el decremento del variante en cada iteración (requerimiento 3), pero para verificar su cumplimiento hay que esperar a que el *while* se complete. La condición del *while* establece su continuidad mientras no se haya procesado el último elemento del arreglo. El invariante asegura que el variante siempre sea positivo (requerimiento 4):

$(0 \leq k \leq N \wedge x = \Sigma_{i=0,k-1}\ a[i]) \rightarrow N - k \geq 0$

y también se cumple que luego del *while* vale la postcondición del programa (requerimiento 5):

$(0 \leq k \leq N \wedge x = \Sigma_{i=0,k-1}\ a[i] \wedge \neg(k \neq N)) \rightarrow x = \Sigma_{i=0,N-1}\ a[i]$

La siguiente *proof outline* parcial repasa lo construido hasta ahora:

$\langle N \geq 0 \rangle$
k := 0 ; x := 0 ;
$\langle$inv: $0 \leq k \leq N \wedge x = \Sigma_{i=0,k-1}\, a[i]$, var: $N - k\rangle$
while $k \neq N$ do
$\langle 0 \leq k \leq N \wedge x = \Sigma_{i=0,k-1}\, a[i] \wedge k \neq N\rangle$
S_2' ;
$\langle 0 \leq k + 1 \leq N \wedge x = \Sigma_{i=0,k}\, a[i]\rangle$
k := k + 1
$\langle 0 \leq k \leq N \wedge x = \Sigma_{i=0,k-1}\, a[i]\rangle$
od
$\langle x = \Sigma_{i=0,N-1}\, a[i]\rangle$

El único predicado para justificar en la *proof outline* es el que aparece antes de la asignación *k := k + 1*: se obtiene aplicando el axioma ASI. Como se observa, sólo falta definir S_2´, que debe satisfacer la fórmula:

$\langle 0 \leq k \leq N \wedge x = \Sigma_{i=0,k-1}\, a[i] \wedge k \neq N\rangle\ S_2'\ \langle 0 \leq k + 1 \leq N \wedge x = \Sigma_{i=0,k}\, a[i]\rangle$

La asignación:

x : = x + a[k]

la satisface (aplicación del axioma ASI y la regla CONS):

$\langle 0 \leq k \leq N \wedge x = \Sigma_{i=0,k-1}\, a[i] \wedge k \neq N\rangle$ x := x + a[k] $\langle 0 \leq k + 1 \leq N \wedge x = \Sigma_{i=0,k}\, a[i]\rangle$

al tiempo que es la instrucción apropiada de acuerdo a la idea del algoritmo. Completado el *while*, se comprueba que efectivamente el variante t = N – k decrece con cada iteración (requerimiento 3), porque la asignación *x := x + a[k]* no lo impacta. En definitiva, la *proof outline* completa de correctitud total queda de la siguiente manera:

$\langle N \geq 0 \rangle$
k := 0 ; x := 0 ;
$\langle$inv: $0 \leq k \leq N \wedge x = \Sigma_{i=0,k-1}\, a[i]$, var: $N - k\rangle$
while $k \neq N$ do
$\langle 0 \leq k \leq N \wedge x = \Sigma_{i=0,k-1}\, a[i] \wedge k \neq N\rangle$
x := x + a[k] ; k := k + 1
$\langle 0 \leq k \leq N \wedge x = \Sigma_{i=0,k-1}\, a[i]\rangle$

od

$\langle x = \Sigma_{i=0,N-1}\ a[i] \rangle$

Para construir el programa hizo falta cálculo e intuición, como en las pruebas de los programas anteriores.

Utilizamos un solo invariante para lograr la correctitud parcial y la terminación del programa, el predicado $0 \leq k \leq N \wedge x = \Sigma_{i=0,k-1}\ a[i]$.

Se puede comprobar que para verificar sólo la terminación del programa alcanza con un invariante más débil, el predicado $0 \leq k \leq N$.

Observaciones finales

Hemos presentado un primer método axiomático de verificación de programas secuenciales determinísticos muy simples, con el objetivo de introducir características generales de la lógica de Hoare, incluyendo definiciones y notaciones.

Nos hemos concentrado en la verificación de las dos propiedades básicas de los programas considerados, la correctitud parcial y la terminación, pertenecientes una a la familia de propiedades *safety* y la otra a la familia de propiedades *liveness*, que se diferencian por la técnica empleada para probarlas, razón por la cual sugerimos tratarlas por separado. En programas más complejos, con más propiedades para verificar, las pruebas individuales, una por propiedad, son aún más necesarias.

A partir de la axiomática estudiada no resulta difícil plantear reglas de prueba para otras instrucciones (por ejemplo, para la instrucción condicional, la selección condicional múltiple, la repetición con evaluación al final, etc).

Entre los aportes metodológicos del método para la construcción de programas correctos sobresalen la composicionalidad y la visión abstracta de la instrucción de repetición como una estructura que preserva un predicado y hace decrecer una función definida en un conjunto bien fundado.

El método presentado es sensato y (en cierto sentido) completo, conceptos de la metateoría de verificación de programas relacionados con la correctitud y el alcance, respectivamente, sobre los que profundizamos en el capítulo 4. Que un método sea composicional, sensato y completo es una situación deseable que lamentablemente no siempre es posible, entre otros motivos por la existencia de estructuras muy complicadas en los lenguajes de programación utilizados.

Notas adicionales

Verificación de programas con procedimientos

Dos años después de publicar su artículo fundacional, C. Hoare presenta una extensión a su axiomática para tratar programas con procedimientos (Hoare, 1971). En dicho trabajo destaca la importancia de seguir una determinada disciplina en el uso de los parámetros para que los programas, además de eficientes, resulten fácilmente verificables. Describimos sintéticamente a continuación las características de las reglas de prueba que plantea.

Para el caso más simple, sin recursión ni parámetros, la regla asociada a la invocación de un procedimiento es:

$$\frac{\{p\}\ S\ \{q\}}{\{p\}\ \text{call proc}\ \{q\}}$$

La instrucción *call proc* se entiende como una invocación a un procedimiento de nombre proc y cuerpo S. La regla refleja la macro expansión que involucra la invocación.

Naturalmente, la regla no sirve cuando el procedimiento es recursivo. Si S incluye una invocación *call proc*, la prueba de {p} S {q} requiere a su vez probar {p} call proc {q}, produciéndose una circularidad que no conduce a la solución. Lo que plantea el método en este caso es imitar el principio inductivo en que se basa la ejecución recursiva, partiendo de una hipótesis que luego se descarta: si asumiendo que se prueba {p} call proc {q}, siendo esta invocación interna de S, se llega a probar {p} S {q}, entonces se habrá probado {p} call proc {q}, siendo ahora esta invocación externa a S. La regla se formula así:

$$\frac{\{p\}\ \text{call proc}\ \{q\} \vdash \{p\}\ S\ \{q\}}{\{p\}\ \text{call proc}\ \{q\}}$$

Queda reflejada claramente la inducción empleada. De una prueba de {p} call proc {q} de cierta longitud, se llega a una prueba de {p} call proc {q} de longitud mayor. Notar que la regla no tiene como premisa una fórmula de correctitud sino un enunciado acerca de la existencia de una prueba, por lo que en este caso se la refiere como metarregla.

La recursión es otra posible fuente de no terminación de un programa. Para la prueba de terminación de un procedimiento recursivo (no considerada en el artículo de C. Hoare de 1971), una alternativa es utilizar, como mostramos en el caso del *while*, una función entera siempre positiva, expresando en todo momento la cantidad de invocaciones activas del procedimiento que se está ejecutando. En (Apt, 1981) se describe otra alternativa, en términos de un predicado parametrizado.

Finalmente, con respecto a la regla para una invocación parametrizada, su forma depende de los tipos de pasajes de parámetros utilizados. Restringimos en este espacio el escenario a parámetros pasados por valor o por resultado. El método plantea una combinación de dos reglas, una regla *canónica*, para derivar una propiedad general del procedimiento considerado, y una regla de *adaptación*, instanciada en cada invocación al procedimiento con los parámetros reales. La regla canónica tiene la siguiente forma (para simplificar la descripción, consideramos solamente procedimientos no recursivos, con un solo parámetro pasado por valor y un solo parámetro pasado por resultado, en este orden):

$$\frac{\{p'\}\ S\ \{q'\}}{\{p'\}\ \text{call proc}\ (y\ ;\ z)\ \{q'\}}$$

El cuerpo S está definido en términos de las variables *y* y z, lo mismo que la precondición p´, mientras que la postcondición q´, por la semántica del pasaje por valor, no incluye la variable *y*. Por su parte, la regla de adaptación se basa en los siguientes puntos:

- Supongamos que se quiere probar la fórmula {p} call proc (e ; x) {q}. Es decir que la expresión *e* y la variable x, los parámetros reales de la invocación, se corresponden con las variables *y* y z, los parámetros formales que utiliza el cuerpo S.
- Por la asignación de *e* a *y* al invocarse el procedimiento, debe cumplirse $p \rightarrow p'[y|e]$.
- La relación que debe cumplirse entre los predicados q y q´ sigue la misma lógica que la de los predicados p y p´, considerando la asignación de z a x a la vuelta de la invocación: si el resultado del procedimiento queda en una variable local *a*, así como q´ refleje la asignación de *a* a z, así q debe reflejar la asignación de *a* a x. De esta manera, se debe incluir como precondición de la invocación el predicado $\forall a: (q'[z|a] \rightarrow q[x|a])$.

En definitiva, la regla de adaptación queda así:

$$\frac{\{p'\}\ \text{call proc}\ (y\ ;\ z)\ \{q'\}}{\{p'[y|e] \wedge \forall a: (q'[z|a] \rightarrow q[x|a])\}\ \text{call proc}\ (e\ ;\ x)\ \{q\}}$$

La verificación de programas y los tipos de datos

Hemos descripto una axiomática considerando sólo variables simples de tipo entero. Aún en un contexto restringido así deben tomarse recaudos, porque la aritmética de las computadoras no es en general la de la matemática. Por ejemplo, en caso de *overflow* en la ejecución de un programa, de acuerdo a la implementación adoptada un resultado posible puede ser

directamente la cancelación del programa, otro el máximo valor entero permitido, otro el valor obtenido usando aritmética modular, etc. Por lo tanto, asumiendo la posibilidad de distintas implementaciones, la axiomática debe contemplar la opción correspondiente.

La axiomática definida se complica si además de variables simples considera arreglos, porque impactan sobre el mecanismo de sustitución sintáctica utilizado para capturar el efecto de una asignación. Por ejemplo, por el axioma ASI se obtiene la fórmula:

{true} a[a[2]] := 1 {a[a[2]] = 1}

la cual no es verdadera: si el estado inicial satisface a[1] = a[2] = 2, luego de la asignación se cumple a[a[2]] = 2. Aun evitando variables suscriptas en los índices continúan los problemas. Por ejemplo, aplicando el axioma ASI llegamos a:

{0 + 1 = a[y]} a[x] := 0 {a[x] + 1 = a[y]}

fórmula que tampoco es verdadera: si al comienzo vale a[1] = x = y = 1, después de la asignación se cumple a[x] = a[y]. En este caso la dificultad proviene de variables suscriptas que denotan un mismo elemento (son *alias*). En (Francez, 1992) se propone una solución para este segundo caso, asumiendo que los índices no contienen variables suscriptas: la sustitución se resuelve en tiempo de evaluación, esto es, la asignación a una variable de tipo arreglo se define mediante un mecanismo cuyo efecto es que la parte derecha se evalúe considerando el estado corriente. Teniendo en cuenta estas dificultades, para simplificar los ejemplos del libro con arreglos que no son de solo lectura no contemplamos el uso de variables suscriptas en los índices ni alias.

Otro tipo de datos problemático para la axiomática que presentamos es el *puntero*. Una extensión de la lógica de Hoare para tratarlo es la *lógica de separación* (O´Hearn, Reynolds y Yang, 2001), que se enfoca en el manejo de áreas de memoria disjuntas, entendiendo que el problema proviene de asumir una visión global del estado de un programa, mientras que los programadores razonan de una manera local, reorganizando pequeñas partes de las estructuras de datos que utilizan. La lógica de separación permite razonar exclusivamente sobre las celdas accedidas, asegurando que cualquier otra celda permanezca sin cambios. Considera un modelo de almacenamiento de tipo *heap*, es decir arbóreo y dinámico, con operaciones de búsqueda, actualización, asignación y desasignación, y una aritmética de direcciones determinada. Utiliza una conjunción denominada *espacial*, de la forma p $*$ q, que expresa que p y q se cumplen en partes disjuntas de una estructura de datos. Un cambio en una celda que afecta a p en p $*$ q no afecta a q. Esto proporciona una manera de componer aserciones que se refieren a diferentes áreas de memoria conservando la información de cada una, facilitando las verificaciones relacionadas con los alias. La lógica cuenta también con *pequeños axiomas*, tal como se los conoce, para definir las operaciones sobre el modelo de almacenamiento, y entre otras, una regla que permite extender una terna {p} S {q} a una terna {p $*$ r} S {q $*$ r}, para establecer la inalterabilidad de determinadas celdas por el efecto de una instrucción específica.

La verificación de programas y las estructuras de datos

C. Hoare amplía el alcance de su axiomática poco después de publicar su artículo fundacional a los programas no sólo con procedimientos sino también con estructuras de datos.

Primero, en un libro emblemático sobre la programación estructurada (Dahl, Dijkstra y Hoare, 1972), plantea la idea de aplicar al diseño de las estructuras de datos principios similares a los de la estructuración de programas, sugiriendo una primera etapa para manejar conceptos abstractos tales como conjuntos, secuencias y funciones, y una segunda etapa, en la instancia de construcción del código más detallado, para la representación concreta de los datos.

Luego, en (Hoare, 1972a), considerando esta perspectiva se enfoca particularmente en la problemática de la verificación de programas, estableciendo que el programador:

1. Primero debe diseñar y probar un algoritmo expresado como un programa abstracto que opera sobre datos abstractos.
2. Y después debe elegir una representación para los datos abstractos, probar su correctitud, y construir el programa concreto en términos de la representación adoptada.

De esta manera, cumplida la primera etapa, la correctitud del programa pasa a depender exclusivamente de la correctitud de la representación elegida. La prueba de correctitud de una representación con respecto a una abstracción se basa en una función definida entre ambas. La verificación completa, así factorizada, se facilita sobremanera. La cantidad de etapas puede variar de acuerdo a la complejidad del programa (recurriéndose a varios niveles de abstracción).

(Hoare, 1972b) contiene un ejemplo clásico de aplicación de esta metodología, en el que se construye y prueba mediante tipos de datos abstractos un programa que implementa la solución de la *criba de Eratóstenes* para obtener números primos.

La verificación de programas y los métodos de programación

Los principios de la verificación axiomática de programas constituyen una guía efectiva para la construcción de programas correctos. En este capítulo hemos ejemplificado la idea desarrollando y al mismo tiempo probando un programa muy sencillo en base al conjunto de axiomas y reglas descripto. Utilizamos el *método de los transformadores de predicados* (Dijkstra, 1976). Dadas una precondición y una postcondición, el método propone transformar una en la otra por medio de mecanismos especificados axiomáticamente (las instrucciones del lenguaje), los que a fin de cuentas conformarán el programa buscado. Partiendo de mecanismos complejos se van construyendo cada vez mecanismos más simples, de acuerdo a una estrategia de descomposición. Existe un axioma o una regla de transformación por cada mecanismo. Una particularidad del método es que transforma los predicados de atrás para adelante (consecuencia de la forma del axioma asociado a la instrucción de asignación), por lo que en realidad debe

entenderse como que arranca de la postcondición del programa y termina en su precondición, la más débil (por eso el método también se conoce como *método de la precondición más débil*). En (Gries, 1981) se profundiza en algunos aspectos del método; por ejemplo, se describen distintas heurísticas para la generación de invariantes y variantes de las repeticiones y para la implementación de las mismas a partir de ellos. Otros dos métodos de programación con acompañamiento de verificación, surgidos en la misma época, son el *método de los tipos de datos abstractos* (Liskov y Zilles, 1974) y el *método de las estructuras de datos* (Jackson, 1975), que a diferencia del método de E. Dijkstra basan los desarrollos en las estructuras de datos.

Una alternativa metodológica a la de la construcción y verificación axiomática en simultáneo propone anteponer la prueba de correctitud al desarrollo. Un método de programación representativo de este paradigma es el *método de transformaciones de programas* (Bauer y Wössner, 1982). Se parte de una especificación como primera versión del programa, correcta por definición. A partir de ella se generan sucesivamente nuevas versiones, aplicando reglas de transformación que preservan la correctitud. El proceso continúa hasta que se alcanza una última versión del programa, definida de acuerdo a determinados criterios (lenguaje objeto, performance, etc). El método utiliza un único lenguaje, de *amplio espectro* como se lo conoce, para expresar las versiones en todos los niveles de abstracción del ciclo de desarrollo: versiones declarativas de altísimo nivel (distintas versiones de la especificación inicial), versiones funcionales de alto nivel (programas recursivos), versiones procedurales de nivel intermedio (programas iterativos), hasta llegar a las versiones finales de bajo nivel, correspondientes al lenguaje objeto.

Acerca de las especificaciones y la actividad de especificar

En los ejemplos desarrollados en el capítulo observamos la utilidad del lenguaje de la lógica de predicados como lenguaje para expresar las especificaciones. Permite descripciones claras, concisas y fácilmente manipulables. De todos modos, tiene varias limitaciones. Por ejemplo, no permite expresar propiedades sobre computaciones. Tampoco características no funcionales, como eficiencia y performance. Estas limitaciones reducen su alcance de aplicación en la verificación de programas y en el proceso de desarrollo de software en general, a menos que se combine con otros lenguajes. En la lista siguiente nombramos algunos de éstos, de uso extendido en la industria y la academia (*googleando* se encuentran numerosas referencias bibliográficas sobre ellos):

- Z, VDM, Larch, para sistemas secuenciales. Utilizan conjuntos, funciones, relaciones, diagramas de transición de estados, pre y postcondiciones.
- CSP, CCS, SDL, lógica temporal, autómatas, redes de Petri, para sistemas concurrentes. Utilizan secuencias, árboles, órdenes parciales de eventos, álgebras de procesos.
- OCL, lenguaje de especificación de objetos.

- RAISE, LOTOS, lenguajes multi-paradigma.

Un problema mayor que el del lenguaje de especificación es el de la propia actividad de especificar, y no es de naturaleza matemática: ¿cómo comprobamos que describimos correctamente las intenciones formuladas? Las intenciones no son entidades formales, y por lo tanto no se puede aplicar un razonamiento formal para hacerlas corresponder con una especificación y verificar su correctitud, como hacemos en la verificación de programas (en (Haeberer, Veloso y Baum, 1988) se profundiza sobre esta problemática). En consecuencia, tenemos que conformarnos con contar con métodos que nos ayuden a describir lo mejor posible requerimientos a partir de intenciones. Un conocido enfoque en esta área es el de M. Jackson (Jackson, 1995), pionero de la ingeniería de software. Basado en las ideas del matemático G. Polya descriptas en su obra clásica (Polya, 1957), el autor establece una serie de principios centrados en el uso de *frames*, marcos o clases de problemas con una estructura en común. Entre las consignas principales plantea la importancia de distinguir entre la descripción de un problema y la descripción de la solución del problema, pertenecientes a ámbitos distintos: el dominio de aplicación y el dominio de ejecución, respectivamente. Y en lo que hace a los métodos de desarrollo para pasar de una a la otra, les atribuye un valor inversamente proporcional a su generalidad, acentuando la idea de que un método debe estar definido para los problemas de un solo *frame*, para explotar al máximo el conocimiento de sus propiedades.

Soporte computacional para las pruebas de programas

Existen numerosas herramientas computacionales de soporte a la verificación de programas en la industria y el ámbito académico, desde algoritmos totalmente automatizados de *model checking*, pasando por ambientes semiautomáticos que asisten al programador, hasta lenguajes de programación diseñados para facilitar las pruebas. Focalizándonos en las últimas dos categorías, sobre todo en ambientes y lenguajes basados en la lógica de Hoare, en lo que sigue mostramos algunos ejemplos.

En (Feinerer y Salzer, 2009) se describen y comparan tres ambientes para la enseñanza de la verificación de programas, *Frege program prover* (FPP), *The key project* (KeY) y *Perfect developer* (PD). FPP trabaja con programas Ada, permite especificar pre y postcondiciones e invariantes y variantes de repeticiones, calcula las precondiciones más débiles, y prueba la correctitud de ternas de Hoare. KeY se aplica sobre un sublenguaje de Java, genera especificaciones OCL, permite anotaciones de cláusulas JML, chequea la consistencia de especificaciones, calcula condiciones de prueba en lógica dinámica de primer orden, y asiste en las pruebas de programas si se explicitan las reglas a ser aplicadas. PD integra un lenguaje de programación orientado a objetos con un probador automático de teoremas, traductores a código Java y C++ y un lenguaje de especificación que permite anotaciones en un nivel bastante abstracto, y asiste en pruebas del estilo de la deducción natural.

(Kovács, Popov y Jebelean, 2006) presenta el sistema *Theorema*, que asiste en la verificación de programas imperativos y funcionales. El sistema genera invariantes por medio de técnicas algebraicas y combinatorias, y condiciones de verificación de la lógica de predicados. En el caso particular de los programas imperativos, Theorema asiste en las pruebas tomando como marco de referencia el método de los transformadores de predicados de E. Dijkstra: dados un programa y una especificación, aplica un procedimiento de transformación de predicados para obtener fórmulas lógicas que luego se verifican con alguno de los probadores de teoremas del sistema.

Syntax-guided synthesis o *SyGuS* (Alur, Singh, Fisman y Solar-Lezama, 2018) es un ambiente que no sólo asiste en las pruebas, sino que también soporta un método de programación, basado en búsquedas. La idea central de este conjunto de herramientas consiste en acompañar las especificaciones lógicas con algoritmos de búsqueda, con el propósito de restringir sintácticamente el espacio de implementaciones que se ajusten a los requerimientos y explorarlo eficientemente. La funcionalidad de SyGuS incluye síntesis de programas a partir de especificaciones lógicas, transformaciones de programas, programación por ejemplos e inferencia automática de invariantes.

Entre los lenguajes de programación diseñados para facilitar la verificación de programas se encuentra SPEC# (Barnett, Leino y Schulte, 2005), iniciativa de Microsoft. Es una extensión de C# que permite expresar pre y postcondiciones, invariantes y anotaciones relacionadas con el encapsulamiento de datos. Tiene un compilador que genera condiciones de verificación que procesa el probador de teoremas Z3. Otro ejemplo es Dafny (Leino, 2010), también iniciativa de Microsoft, implementado en el marco de la tecnología de SPEC#. Permite expresar especificaciones con estructuras abstractas como conjuntos y secuencias, funciones y tipos de datos recursivos, y cuenta con procedimientos que permiten automatizar partes de pruebas de programas. Un tercer ejemplo es Pest (de Caso, Garbervetsky y Gorín, 2011). Está orientado a programas muy básicos, procedurales, con instrucciones de tipo *while*, sin recursividad. La semántica está definida axiomáticamente. Provee un mecanismo de especificación de importante capacidad expresiva y admite distintas formas de inferencia de predicados, entre ellas la del cálculo de la precondición más débil.

Referencias bibliográficas

Los libros anteriores sobre verificación de programas de la Colección de Libros de Cátedra de la UNLP mencionados en el prólogo son (Rosenfeld e Irazábal, 2010), (Rosenfeld e Irazábal, 2013) y (Pons, Rosenfeld y Smith, 2017). En particular, el último contiene capítulos dedicados exclusivamente a la lógica.

Las citas de E. Dijkstra que se comentan en el prólogo (una) y en la introducción (dos) se encuentran en sus artículos (Dijkstra, 1988) y (Dijkstra, 1972), respectivamente.

Para profundizar en algunas definiciones y conceptos del capítulo 1 se pueden consultar, entre otras fuentes, (Mendelson, 2009), el capítulo 2 de (Apt y Olderog, 1997) y la parte 2 de (Papadimitriou, 1994).

El artículo fundacional de C. Hoare de verificación axiomática de programas es (Hoare, 1969), uno de los más citados en la literatura de las ciencias de la computación. Entre las publicaciones previas que lo influencian sobresale (Floyd, 1967), pero el disparador más importante del trabajo proviene de una experiencia personal que afecta profundamente al autor: el fracaso de un proyecto de 30 años hombre que lidera a mediados de los años 1960, descripto en detalle en (Hoare, 1981).

Las reglas del condicional y de la terminación del *while* que utilizamos, ausentes en el artículo de C. Hoare de 1969, están en (Lauer, 1971) y (Dijkstra, 1982), respectivamente. Otra regla que no aparece en la versión inicial y que C. Hoare describe recién en 1972 es la del salto incondicional o *goto* (Clint y Hoare, 1972), instrucción polémica en el marco de la programación estructurada como se refleja por ejemplo en (Dijkstra, 1968) y (Knuth, 1974). (Hoare y Jones, 1989), selección de varias publicaciones de C. Hoare, incluye otras extensiones a su axiomática, que tratan los procedimientos, las estructuras de datos y los programas paralelos (nos referimos a las dos primeras en las notas adicionales). En la misma selección se reproducen otros trabajos que trascienden el área exclusiva de la verificación de programas, como el artículo (Hoare, 1971), donde se presenta un ejemplo de desarrollo y prueba de correctitud de un programa de ordenamiento, clara influencia sobre las obras emblemáticas de programación sistemática (Wirth, 1973) y (Dijkstra, 1976), y el artículo (Hoare y Wirth, 1973), que utiliza el método axiomático para definir la semántica del lenguaje Pascal.

(Apt, 1981) analiza en profundidad distintas características de la lógica de Hoare, incluyendo la metateoría. Describe axiomáticas para verificar programas con *while* y con procedimientos, en este último caso considerando distintos mecanismos de pasajes de parámetros, recursión y declaración de variables locales. Particularmente, para las pruebas de terminación no emplea funciones enteras sino predicados parametrizados, propuestos en (Harel, 1979).

(Francez, 1992) y (Apt y Olderog, 1997) compendian sistemáticamente distintas axiomáticas para la verificación de programas secuenciales determinísticos y no determinísticos y programas concurrentes. Presentan numerosos ejemplos y referencias bibliográficas. Distintos pasajes de nuestro libro se basan en sus contenidos.

De muy reciente aparición, los libros (Jones y Misra, 2021) y (Apt y Hoare, 2022) incluyen artículos de varios autores sobre la influencia de los trabajos de C. Hoare y E. Dijkstra, respectivamente, en distintas áreas de la correctitud de programas.

Detallamos a continuación las referencias bibliográficas citadas más las que mencionamos en las notas adicionales:

Alur, R., Singh, R., Fisman, D. y Solar-Lezama, A. (2018). *Search-based program synthesis.* Comm. ACM, 61, 12, 84-93.

Apt, K. (1981). *Ten years of Hoare's logic: a survey - part 1*. ACM Trans. Prog. Lang. Syst., 3, 4, 431-483.

Apt, K. y Hoare, C. (2022). *Edsger Wybe Dijkstra. His life, work and legacy*. ACM Books.

Apt, K. y Olderog, E. (1997). *Verification of secuencial and concurrent programs, second edition*. Springer-Verlag.

Barnett, M., Leino, K. y Schulte, W. (2005). *The Spec# programming system: an overview*. Proc. of Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, Lecture Notes in Computer Science, 3362, 49-69. Springer-Verlag.

Bauer, L. y Wössner, H. (1982). *Algorithmic language and program development*. Springer-Verlag.

Clint, M. y Hoare, C. (1972). *Program proving: jumps and functions*. Acta Informatica, 1, 3, 214-224.

Dahl, O., Dijkstra, E. y Hoare, C. (1972). *Structured programming*. Academic Press.

de Caso, G., Garbervetsky, D. y Gorín, D. (2011). *Integrated program verification tools in education.* Software - Practice and Experience, 00:1-16.

Dijkstra, E. (1968). *Go to statement considered harmful*. Comm. ACM, 11, 3, 147-148.

Dijkstra, E. (1972). *The humble programmer.* ACM Annual Conference, Boston.

Dijkstra, E. (1976). *A discipline of programming*. Prentice Hall.

Dijkstra, E. (1982). *Selected writings of computing*. Springer-Verlag.

Dijkstra, E. (1988). *On the cruelty of really teaching computing science.* The University of Texas, Austin.

Feinerer, I. y Salzer, G. (2009). *A comparition of tools for teaching formal software verification.* Formal Aspects of Computing, 21, 3, 293-301.

Floyd, R. (1967). *Assigning meanings to programas*. Proc. of Symposium on Applied Mathematics, 19, 19-32.

Francez, N. (1992). *Program verification*. Addison-Wesley.

Gries, D. (1981). *The science of programming*. Springer-Verlag.

Haeberer, A., Veloso, P. y Baum, G. (1988). *Formalización del proceso de desarrollo de software.* Editorial Kapelusz.

Harel, D. (1979). *First-order dynamic logic*. Lecture Notes in Computer Science, 68. Springer-Verlag.

Hoare, C. (1969). *An axiomatic basis for computer programming*. Comm. ACM, 12, 10, 576-580.

Hoare, C. (1971). *Proof of a program: FIND.* Comm. ACM, 14, 1, 39-45.

Hoare, C. (1972a). *Proof of correctness of data representations*. Acta Informatica, 1, 4, 271-281.

Hoare, C. (1972b). *Proof of a structured program: the sieve of Eratosthenes*. BCS Computer Journal, 15, 4, 321-325.

Hoare, C. (1981). *The emperor´s old clothes*. Comm. ACM, 24, 2, 75-83.

Hoare, C. y Jones, C. (1989). *Essays in computing science*. Prentice Hall.

Hoare, C. y Wirth, N. (1973). *An axiomatic definition of the programming language Pascal*. Acta Informatica 2, 4, 335-355.

Jackson, M. (1975). *Principles of program design*. ACM Press.

Jackson, M. (1995). *Software requirements and specifications: a lexicon of practice, principles and prejudices*. ACM Press.

Jones, C. y Misra, J. (2021). *Theories of programming. The life and works of Tony Hoare.* ACM Books.

Knuth, D. (1974). *Structured programming with go to statements*. Computing Surveys, 6, 4, 261-301.

Kovács, L, Popov, N. y Jebelean, T. (2006). *Combining logical and algebraic techniques for program verification in Theorema.* Conf. Leveraging Applications of Formal Methods, Verification and Validation, IsoLA 2006.

Lauer, P. (1971). *Consistent formal theories of the semantic of programming languages*. Tech. Rep. 25, 121, IBM Laboratory, Viena.

Leino, K. (2010). *Dafny: an automatic program verifier for functional correctness*. Proc. of the Conf. on Logic for Programming, Artificial Intelligence, and Reasoning, 348-370.

Liskov, B. y Zilles, S. (1974). *Programming with abstract data types*. SIGPLAN Notices, 9, 4, 50-59.

Mendelson, E. (2009). *Introduction to mathematical logic, fifth edition*. CRC Press.

O´Hearn, P., Reynolds, J. y Yang, H. (2001). *Local reasoning about programs that alter data structures*. Lecture Notes in Computer Science, 2142, 1-19. Springer-Verlag.

Papadimitriou, C. (1994). *Computational complexity*. Addison-Wesley.

Polya, G. (1957). *How to solve it: a new aspect of mathematical method, 2nd edition*. Princeton University Press.

Pons, C., Rosenfeld, R. y Smith, C. (2017). *Lógica para informática*. EDULP.

Rosenfeld, R. e Irazábal, J. (2010). *Teoría de la computación y verificación de programas*. McGraw-Hill y EDULP.

Rosenfeld, R. e Irazábal, J. (2013). *Computabilidad, complejidad computacional y verificación de programas*. EDULP.

Wirth, N. (1973). *Systematic programming: an introduction*. Prentice Hall.

Ejercicios

1. Especificar un programa que termine con y = 1 si al comienzo se cumple una propiedad p sobre una variable de programa x, o con y = 0 en caso contrario.

Resolución

Primer intento:

$$\langle \text{true}, (y = 1 \rightarrow p(x)) \wedge (y = 0 \rightarrow \neg p(x)) \rangle$$

La especificación es incorrecta. Por ejemplo, el programa *y := 2* satisface la especificación y no es el programa buscado. La postcondición es demasiado débil.

Segundo intento:

$$\langle \text{true}, (0 \leq y \leq 1) \wedge (y = 1 \rightarrow p(x)) \wedge (y = 0 \rightarrow \neg p(x)) \rangle$$

Otra vez la especificación es incorrecta. Por ejemplo, si al comienzo x = 7, no se cumple p(7) y se cumple p(5), el programa *x := 5* satisface la especificación y no es el programa buscado. En este caso el error se produce porque se omite que x puede cambiar a lo largo del programa.

Tercer intento (exitoso):

$$\langle x = X, (0 \leq y \leq 1) \wedge (y = 1 \rightarrow p(X)) \wedge (y = 0 \rightarrow \neg p(X)) \rangle$$

2. Determinar si las especificaciones (x = X, x = 2 **.** X) y (true, $\exists$X: x = 2 **.** X) son equivalentes.

3. Especificar un programa que calcule la raíz cuadrada entera de un número natural x. El programa, además, debe terminar con el valor inicial de x. ¿Se puede especificar también que x no cambie a lo largo del programa?

4. Especificar correctamente el programa presentado en el capítulo (Ejemplo 2), que calcula el valor absoluto de un número entero.

5. Sea post(p, S) el conjunto de todos los estados finales que se pueden obtener ejecutando un programa S a partir de una precondición p. Probar: {p} S {q} sii post(p, S) $\subseteq$ {σ | $\sigma \models$ q}.

Resolución

Si {p} S {q}, entonces si S termina a partir de p lo hace en un estado que satisface q, y por lo tanto post(p, S) $\subseteq$ {σ | $\sigma \models$ q}.

Recíprocamente, si post(p, S) $\subseteq$ {σ | $\sigma \models$ q}, entonces todos los estados finales que se pueden obtener de la ejecución de S a partir de p satisfacen q, que es la definición de {p} S {q}.

6. Con la hipótesis {p} S {q}, determinar si se cumplen los siguientes enunciados:
   6.1. Si S termina en un estado que satisface q, el estado inicial satisface p.
   6.2. Si S termina en un estado que no satisface q, el estado inicial no satisface p.
   6.3. Si S no termina, el estado inicial no satisface p.

7. Responder los tres incisos del ejercicio anterior con la hipótesis $\langle p \rangle\ S\ \langle q \rangle$.

8. Determinar semánticamente, es decir recurriendo a las definiciones de correctitud parcial y total, si se cumplen las siguientes fórmulas de correctitud:
   8.1. $\{y = 20\}\ x := 4\ \{y = 20\}$
   8.2. $\{x = 0\}$ while $y = 0$ do $y := 0$ od $\{x = 0\}$
   8.3. $\langle x = 0 \rangle$ while $y = 0$ do $y := 0$ od $\langle x = 0 \rangle$
   8.4. $\{true\}\ x := 0\ \{false\}$
   8.5. $\{p\}\ x := e\ \{p \wedge x = e\}$, para todo predicado p.

9. Si $p \longrightarrow q$, ¿cuál es la relación de inclusión entre los conjuntos de estados denotados por ambos predicados?

10. Si se cumple $\{p_1 \wedge p_2\}\ S\ \{q_1 \wedge q_2\}$, ¿se cumple $\{p_1\}\ S\ \{q_1\}$ o $\{p_2\}\ S\ \{q_2\}$?

11. La expresión $\sigma[x|a]$, conocida como *variante* de σ, denota que el contenido de la variable x en el estado σ es *a*. De esta manera, el valor de $\sigma[x|a](x)$ es *a*, y el de $\sigma[x|a](y)$ para toda variable *y* distinta de x es $\sigma(y)$. Probar: $\sigma[x|a][x|b] = \sigma[x|b]$.

12. Si se cumple tanto $\{p\}\ S\ \{q\}$ como $\{p\}\ S\ \{\neg q\}$, ¿qué se puede afirmar sobre el comportamiento del programa S a partir de cualquier estado que satisface p?

13. Probar sintácticamente, es decir recurriendo al conjunto de axiomas y reglas descripto en el capítulo, las fórmulas verdaderas enunciadas en el ejercicio 8.

14. Probar sintácticamente $\{p\}$ while p do S od $\{q\}$, asumiendo que ya se probó $\{p \vee q\}\ S\ \{p \vee q\}$.

15. Probar sintácticamente $\{true\}$ while true do $x := x - 1$ od $\{false\}$.

16. En el capítulo se probó la correctitud parcial del programa que calcula la división entera entre dos números enteros a partir de la precondición $x \geq 0 \wedge y > 0$ (Ejemplo 3). Probar ahora su correctitud parcial con la misma postcondición pero con la precondición $x \geq 0 \wedge y \geq 0$.

17. El siguiente programa calcula en una variable z el producto de dos números enteros x e *y* por sumas sucesivas:

    S_{prod} :: $z := 0$; $k := y$; while $k > 0$ do $z := z + x$; $k := k - 1$ od

    Probar sintácticamente $\langle x \geq 0 \wedge y \geq 0 \rangle\ S_{prod}\ \langle z = x \cdot y \rangle$. Además, indicar si la precondición que se formula es la más débil, y si no lo es, formular la que corresponde.

Ayuda

Intentar, por ejemplo, con el invariante $p_1 = (z = x \,.\, (y - k) \wedge k \geq 0)$ para la prueba de correctitud parcial, y con el invariante $p_2 = (k \geq 0)$ y el variante $t = k$ para la prueba de terminación.

18. Proponer una regla de prueba de correctitud parcial para la instrucción *repeat S until B*, cuya semántica en términos del while es *S ; while ¬B do S od*.

Resolución

Una regla candidata, con una forma similar a la de la regla REP, es:

$$\frac{\{p \wedge \neg B\}\ S\ \{p\}}{\{p\}\ \text{repeat S until B}\ \{p \wedge B\}}$$

Pero esta regla no es sensata. La premisa establece que S preserva p sólo cuando vale ¬B. A partir de $p \wedge B$, luego de la primera iteración del *repeat* podría cumplirse $\neg p \wedge B$, provocando la terminación de la instrucción con dicha postcondición. Por ejemplo, esto sucede con p = (x = 0), B = (y = 0) y *S :: if y ≠ 0 then skip else x := 1 fi*.

Una solución, entonces, es tratar de manera especial la primera iteración del *repeat*, y utilizar la postcondición resultante como invariante. Así se obtiene la siguiente regla sensata:

$$\frac{\{p\}\ S\ \{q\},\ \{q \wedge \neg B\}\ S\ \{q\}}{\{p\}\ \text{repeat S until B}\ \{q \wedge B\}}$$

19. Proponer una regla de prueba alternativa a la regla REP tal que su conclusión sea la fórmula {p} while B do S od {q}.

20. Proponer reglas de prueba de correctitud parcial para las siguientes variantes de la instrucción de selección condicional *if then else* considerada en el capítulo:
    20.1. if B then S fi
    Semántica informal: si se cumple B se ejecuta S y se pasa a la siguiente instrucción, y si no se cumple B se pasa directamente a la siguiente instrucción.
    20.2. if B_1 then S_1 else if B_2 then S_2 else … else if B_{n-1} then S_{n-1} else S_n fi … fi fi
    Semántica informal: es como la instrucción if then else pero con más alternativas.

CAPÍTULO 3
Verificación de programas secuenciales no determinísticos

Introducción

En este capítulo describimos un segundo método axiomático de verificación de programas, adaptación del método presentado en el capítulo anterior, para probar *programas secuenciales no determinísticos*.

A diferencia de los programas determinísticos, los programas no determinísticos pueden producir a partir de un estado inicial varias computaciones, y por lo tanto varios estados finales. Aun así, en la lógica de Hoare se los puede seguir viendo como transformadores de predicados, con una noción de correctitud determinada.

El no determinismo en los programas no sólo sirve para implementar sistemas intrínsecamente no determinísticos. También facilita la actividad de programar, al permitir manejar adecuadamente distintos niveles de abstracción, evitando determinismo innecesario al menos hasta que eventualmente se lo requiera por razones de eficiencia. Particularmente en lo que hace a la verificación de programas no determinísticos, además de su importancia *per se* su estudio constituye una apropiada introducción a la verificación de programas concurrentes, ámbito donde más se justifica por su complejidad un tratamiento formal, porque la concurrencia puede ser modelizada por el no determinismo.

El capítulo se estructura de la siguiente manera. En la próxima sección describimos el lenguaje de programación que usamos, conocida variante del lenguaje determinístico visto antes, con selecciones condicionales y repeticiones no determinísticas, y definimos las propiedades de programas que contempla el método, la correctitud parcial y la terminación ya tratadas y la *ausencia de falla* como novedad. Siguen secciones para presentar la axiomática para las pruebas de dichas propiedades, con ejemplos de aplicación, y el desarrollo sistemático de un programa basado en el método, volviendo a remarcar la utilidad de los axiomas y reglas como guía para la construcción de programas correctos. En una nueva sección ampliamos el lenguaje de programación con la instrucción de asignación aleatoria, que modifica una característica del no determinismo observada previamente, y adaptamos el método para incluirla. En otra sección volvemos a adaptar el método, para tratar programas asumiendo *fairness*. Terminamos el capítulo con observaciones finales, notas adicionales, referencias bibliográficas y ejercicios.

Programas y propiedades a considerar

La sintaxis del lenguaje de programación se basa en la del lenguaje GCL o *Guarded Command Languaje* (Lenguaje de Comandos Guardados), creado por E. Dijkstra. Es la siguiente:

S :: skip | x := e | S_1 ; S_2 | if CG fi | do CG od
CG :: B $\rightarrow$ S | B $\rightarrow$ S or CG

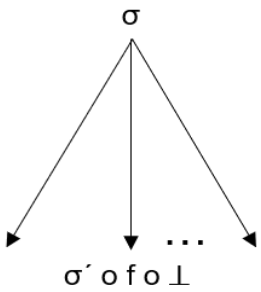
Con x se representan distintas variables (x, *y*, z, etc). Las únicas instrucciones novedosas con respecto al lenguaje utilizado antes son: *if CG fi,* la *selección condicional no determinística*, que también se puede expresar con *if* $B_1 \rightarrow S_1$ *or…or* $B_n \rightarrow S_n$ *fi* y abreviar con *IF*; y *do CG od*, la *repetición no determinística*, que también se puede expresar con *do* $B_1 \rightarrow S_1$ *or…or* $B_n \rightarrow S_n$ *od* y abreviar con *DO*. Los componentes $B \rightarrow S$ se denominan *comandos guardados* (o *comandos con guardia*), lo que explica el nombre del lenguaje. La sintaxis de las expresiones enteras *e* y de las expresiones booleanas B es la misma que antes. Para simplificar la presentación no consideramos instrucciones DO anidadas, y a veces en los ejemplos de programas utilizamos variables booleanas. La semántica informal de las dos instrucciones es la siguiente:

- Instrucción IF: Si algunas de las guardias B_i son verdaderas, se selecciona una no determinísticamente, se ejecuta el subprograma S_i asociado, y si termina, el programa continúa con la instrucción siguiente al IF. Si en cambio ninguna de las guardias es verdadera, el IF (y el programa) *falla,* termina incorrectamente. Para simplificar, consideramos sólo este caso de falla en los programas.
- Instrucción DO: Si algunas de las guardias B_i son verdaderas, se selecciona una no determinísticamente, se ejecuta el subprograma S_i asociado, se repite el mismo ciclo mientras existan guardias verdaderas, y si en algún momento todas las guardias son falsas el programa continúa con la instrucción siguiente al DO (es decir que no se produce una falla como en el IF sino que el DO simplemente termina). Si siempre existe alguna guardia verdadera, el DO (y el programa) *no termina*, *diverge*.

Cuando una guardia B_i de un IF o un DO es verdadera, se dice que la *dirección* i está *habilitada*. En la selección entre las direcciones habilitadas no se considera ninguna asunción de probabilidad.

Por la semántica de las instrucciones no determinísticas, a partir del estado inicial de un programa pueden producirse varias computaciones, y por lo tanto varios estados finales. Las computaciones de un programa se pueden representar mediante un *árbol de computaciones*, con el estado inicial del programa en el nodo raíz y los estados finales en las hojas. El caso más general es un árbol de computaciones en el que conviven computaciones que terminan sin falla, computaciones que terminan con falla y computaciones que no terminan (para facilitar la nomenclatura, de ahora en más usamos *terminación* por terminación sin falla, *falla* por

terminación con falla, y *divergencia* por no terminación). La figura siguiente muestra un árbol de computaciones con las tres clases de computaciones mencionadas:



En términos de estados, a partir de un estado inicial σ una computación puede terminar en un estado σ´, o fallar (en este caso el estado final es un *estado de falla* f), o divergir (el símbolo ⊥ representa una computación infinita).

Mostramos a continuación algunos ejemplos de programas. Comenzamos con dos programas que implementan la selección condicional determinística *if B then* S_1 *else* S_2 *fi* y la repetición determinística *while B do S od*. Para el primer caso corresponde:

S_{scd} :: if B $\rightarrow$ S_1 or ¬B $\rightarrow$ S_2 fi

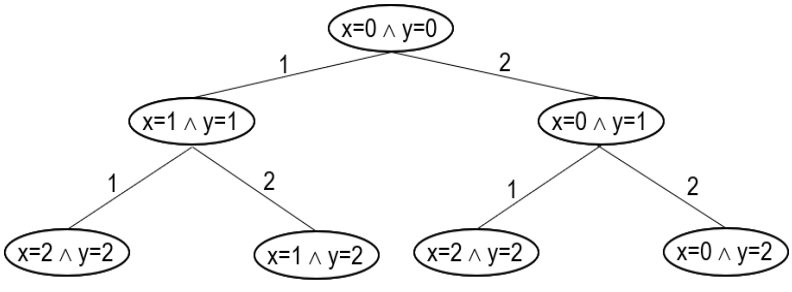
y para el segundo:

S_{rd} :: do B $\rightarrow$ S od

El siguiente programa devuelve en la variable x algún número entre 0 y N:

S_{sel} :: x := 0 ; y := 0 ; do y < N $\rightarrow$ y := y + 1 ; if true $\rightarrow$ x := y or true $\rightarrow$ skip fi od

El árbol de computaciones del programa cuando N = 2 es el siguiente:



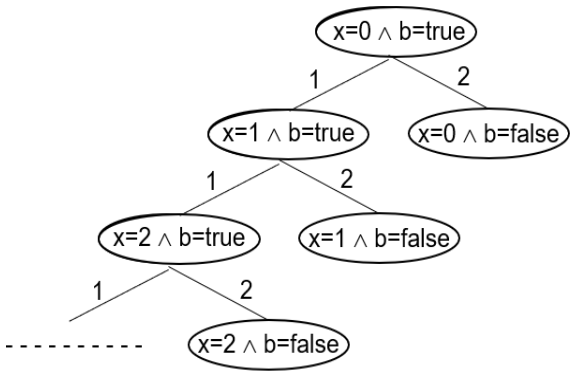
Hay cuatro computaciones posibles. El segundo y tercer nivel del árbol representan la primera y segunda iteración del DO, respectivamente. Los números 1 y 2 identifican las dos direcciones del IF. Las hojas del árbol contienen los estados finales. El contenido final de la variable x puede

ser 0, 1 o 2, y al no haber ninguna asunción de probabilidad, no se puede asegurar que al cabo de varias ejecuciones el programa devuelva los tres valores.

Este otro programa, variante del anterior, amplía el rango de valores finales de la variable x a todo el conjunto N de los números naturales:

S_{nat} :: x := 0 ; b := true ; do b $\rightarrow$ x := x + 1 or b $\rightarrow$ b := false od

En este caso, el árbol de computaciones del programa es el siguiente:



Los números 1 y 2 identifican las dos direcciones del DO. Hay una computación que termina con x = 0, otra con x = 1, otra con x = 2, y así sucesivamente, y además hay una computación que diverge, representada en el árbol por la rama infinita que siempre va a la izquierda. Dicha computación corresponde al caso en que siempre se toma la primera dirección del DO. De esta manera, si bien el programa cumple con su cometido, porque devuelve en la variable x cualquier número natural, puede divergir. Y esto sucede con cualquier programa con infinitos estados finales. Expresado de otro modo, un programa genera una cantidad finita de estados finales o una de sus computaciones diverge.

La justificación matemática de esta característica es el *lema de König* de la teoría de grafos, que establece que en un árbol infinito con grado finito, es decir con nodos con una cantidad finita de hijos, alguna rama es infinita (el grado de los árboles de computaciones de los programas definidos es finito porque las instrucciones IF y DO tienen una cantidad finita de direcciones). Este tipo de no determinismo que se manifiesta en los programas descriptos se conoce como no determinismo *acotado*.

Completamos los ejemplos con programas que revelan otras características del lenguaje.

El programa siguiente calcula el máximo común divisor de x e y, siendo ambos valores mayores que 0, mediante el *algoritmo de Euclides*:

S_{mcd} :: do x > y $\rightarrow$ x := x – y or x < y $\rightarrow$ y := y – x od

El programa termina cuando x = y (el máximo común divisor se obtiene en las dos variables). Notar cómo la simetría del programa facilita el entendimiento de lo que hace el algoritmo. Lo mismo se observa en este otro programa, que ordena cuatro valores de manera ascendente:

S_{ord} :: do $n_1 > n_2 \rightarrow aux := n_1 ; n_1 := n_2 ; n_2 := aux$
or $n_2 > n_3 \rightarrow aux := n_2 ; n_2 := n_3 ; n_3 := aux$
or $n_3 > n_4 \rightarrow aux := n_3 ; n_3 := n_4 ; n_4 := aux$
od

El programa termina cuando $n_1 \leq n_2 \leq n_3 \leq n_4$. Otra ventaja del lenguaje es que facilita la detección de situaciones no deseadas. Por ejemplo, para evitar una división por cero podemos escribir:

if $y \neq 0 \rightarrow x := x / y$ fi

para que si y = 0, el programa falle. De la misma manera, si permitimos arreglos, la instrucción:

if $0 \leq i \wedge i < n \rightarrow x := a[i]$ fi

produce una falla si se pretende acceder al arreglo *a* con un índice fuera del rango [0, n – 1]. El último ejemplo muestra cómo los programas no determinísticos permiten modelizar la concurrencia. A partir del programa concurrente (utilizamos el símbolo habitual ||):

S_{con} :: [x := 0 || x := 1 || x := 2]

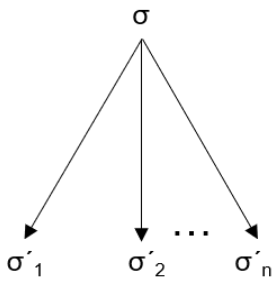
se puede derivar el siguiente programa secuencial no determinístico equivalente:

S_{sec} :: b_1 := true ; b_2 := true ; b_3 := true ;
do $b_1 \rightarrow x := 0 ; b_1 := false$
or $b_2 \rightarrow x := 1 ; b_2 := false$
or $b_3 \rightarrow x := 2 ; b_3 := false$
od

Las variables booleanas b_i, inicialmente verdaderas, dejan de valer una a una en un orden no determinístico, simulando el comportamiento (asumiendo *interleaving*) del programa original.

Pasamos ahora a definir las propiedades de programas que trata el método axiomático de verificación presentado en este capítulo.

De lo descripto antes, se desprende que un programa S a partir de una especificación (p, q) debe cumplir que a partir de todo estado inicial que satisface la precondición p, todas sus computaciones deben terminar en un estado final que satisface la postcondición q:



Es decir, a partir de todo estado inicial σ que satisface p, todas las computaciones de S tienen que terminar en estados finales $\sigma'_1, \sigma'_2, \ldots, \sigma'_n$ que satisfacen q. O expresado de otra manera, el programa S debe satisfacer tres propiedades. A partir de todo estado que satisface la precondición:

1. Toda computación de S que termina lo hace en un estado que satisface la postcondición.
2. Ninguna computación de S falla.
3. Ninguna computación de S diverge.

La primera y la tercera propiedad son, respectivamente, la *correctitud parcial* y la *no divergencia* que definimos en el capítulo anterior, una de tipo *safety*, probada por inducción, y la otra de tipo *liveness*, probada en base a una función definida en un conjunto bien fundado. La segunda propiedad es la *ausencia de falla*, también de tipo *safety*. Las tres propiedades constituyen conjuntamente en este marco la *correctitud total*. Algunos autores distinguen una categoría intermedia de *correctitud total débil*, que no incluye la ausencia de falla. La fórmula:

$\{p\}\ S\ \{q\}$

expresa la correctitud parcial de S con respecto a (p, q). Y la fórmula:

$\langle p \rangle\ S\ \langle true \rangle$

expresa la no divergencia y la ausencia de falla de S a partir de p, o en otras palabras, que a partir de la precondición p el programa S siempre termina correctamente (en algún estado final). La fórmula $\{p\}\ S\ \{q\} \wedge \langle p \rangle\ S\ \langle true \rangle$ expresa entonces la correctitud total de S con respecto a la especificación (p, q), que abreviamos con:

$\langle p \rangle\ S\ \langle q \rangle$

Por ejemplo, la siguiente fórmula de correctitud es verdadera:

$\langle x = 10 \rangle$ do $x > 1 \rightarrow x := x - 2$ or $x > 0 \rightarrow x := x - 1$ od $\langle x = 0 \rangle$

También lo es la fórmula:

{true} S {true}

cualquiera sea el programa S, porque si S falla o diverge a partir de un determinado estado inicial no se viola la correctitud parcial. Por otro lado, un ejemplo de fórmula no verdadera es:

$\langle true \rangle$ S $\langle true \rangle$

Un contraejemplo de este caso es el siguiente programa, que falla si al comienzo x = 0:

S_{vae} :: if $x > 0 \rightarrow y := x$ or $x < 0 \rightarrow y := -x$ fi

Un segundo contraejemplo, ahora considerando divergencia, es este otro programa cuando al comienzo vale x < 0:

S_{de} :: do $x > 1 \rightarrow x := x - 2$ or $x \neq 0 \rightarrow x := x - 1$ od

Axiomática para las pruebas de correctitud parcial

El método axiomático para las pruebas de los programas no determinísticos que acabamos de describir difiere del que utilizamos en el capítulo anterior sólo en las reglas correspondientes a las instrucciones de selección condicional y repetición, en consonancia con la adaptación no determinística efectuada sobre el lenguaje determinístico antes presentado. De todos modos, las nuevas reglas mantienen los mismos principios que las reglas de las instrucciones determinísticas.

El método conserva la composicionalidad del método anterior. También la sensatez y la completitud, que se prueban en el capítulo 4.

En esta sección nos concentramos en la correctitud parcial, después completamos la axiomática para las propiedades restantes. Las reglas referidas son las siguientes (continuamos la numeración de las reglas y de los ejemplos de aplicación del capítulo previo):

13. *Regla del condicional no determinístico* (NCOND)

$$\frac{\{p \wedge B_i\}\ S_i\ \{q\},\ i = 1, \ldots, n}{\{p\}\ \text{if } B_1 \rightarrow S_1 \text{ or} \ldots \text{or } B_n \rightarrow S_n \text{ fi } \{q\}}$$

La regla establece que si se cumplen las premisas $\{p \wedge B_1\}\ S_1\ \{q\}$, …, $\{p \wedge B_n\}\ S_n\ \{q\}$, entonces a partir de la precondición p, se cumpla la guardia B_1 o … o la guardia B_n, luego de la ejecución respectiva del subprograma S_1 o … o el subprograma S_n se cumple la postcondición q. La regla no asegura que el IF no falle, porque puede ocurrir que ninguna de las guardias B_i resulte verdadera.

14. *Regla de la repetición no determinística* (NREP)

$$\frac{\{p \wedge B_i\}\ S_i\ \{p\},\ i = 1, \ldots, n}{\{p\}\ \text{do } B_1 \rightarrow S_1 \text{ or} \ldots \text{or } B_n \rightarrow S_n \text{ od } \{p \wedge \bigwedge_{i=1,n} \neg B_i\}}$$

La regla utiliza un predicado *invariante* p, que se cumple antes del DO y después de todas sus iteraciones. Establece que si la ejecución del subprograma S_i preserva p mientras se cumple la guardia B_i, cualquiera sea la dirección i, entonces al terminar el DO se cumple $p \wedge \bigwedge_{i=1,n} \neg B_i$. La regla no asegura que el DO no diverja.

Desarrollamos en lo que sigue un ejemplo de aplicación de las reglas.

Ejemplo 6. Prueba de correctitud parcial de un programa que obtiene algún divisor

El programa siguiente obtiene en *y* algún divisor de $x \geq 1$ (la expresión z | x que se utiliza significa que *z* divide a x):

```
Sdivn :: z := 1 ; y := 1 ;
        do z < x → z := z + 1 ;
                   if z | x → if true → y := z or true → skip fi
                   or ¬(z | x) → skip
                   fi
        od
```

El IF externo del DO obtiene divisores de x, y el IF interno selecciona algunos no determinísticamente.

Vamos a probar $\{x \geq 1\}\ S_{divn}\ \{y \mid x\}$.

Estructuramos la prueba en dos partes, para las inicializaciones del programa y para el DO. Como invariante del DO proponemos la propia postcondición del programa, es decir p = (y | x),

que refleja la idea del algoritmo: la variable *y* se va modificando eventualmente recibiendo divisores de x. El plan de prueba queda así:

a) Luego de las inicializaciones se llega al invariante:
$\{x \geq 1\}\ z := 1\ ;\ y := 1\ \{y \mid x\}$

b) Prueba del DO:
$\{y \mid x\}\ DO\ \{y \mid x \wedge \neg(z < x)\}$

c) Por la aplicación de las reglas SEC y CONS sobre lo anterior se completa la prueba.

Omitimos la prueba de (a), la fórmula se obtiene fácilmente. Para la prueba de (b) nos basamos en la estructura del cuerpo del DO, que es una secuencia de una asignación y un IF. Primero tratamos la asignación, considerando la guardia z < x del DO:

1) $\{y \mid x \wedge z < x\}\ z := z + 1\ \{y \mid x\}$ (ASI, CONS)

Ahora tratamos el IF, que requiere a su vez tratar otro IF y una instrucción *skip*. Primero tratamos el IF interno, considerando sus dos guardias *true* y la guardia z < x del DO:

2) $\{y \mid x \wedge z \mid x \wedge true\}\ y := z\ \{y \mid x\}$ (ASI, CONS)
3) $\{y \mid x \wedge z \mid x \wedge true\}\ skip\ \{y \mid x\}$ (SKIP, CONS)
4) $\{y \mid x \wedge z \mid x\}$ if true $\rightarrow$ y := z or true $\rightarrow$ skip fi $\{y \mid x\}$ (2, 3, NCOND)

Queda por tratar el *skip*. Considerando la guardia $\neg(z \mid x)$ hacemos:

5) $\{y \mid x \wedge \neg(z \mid x)\}\ skip\ \{y \mid x\}$ (SKIP, CONS)

De esta manera obtenemos la siguiente fórmula para el IF externo:

6) $\{y \mid x\}$ if z | x $\rightarrow$ if true $\rightarrow$ y := z or true $\rightarrow$ skip fi or $\neg$z | x $\rightarrow$ skip fi $\{y \mid x\}$
(4, 5, NCOND)

Finalmente llegamos a probar el DO:

7) $\{y \mid x \wedge z < x\}\ z := z + 1\ ;$ if z | x $\rightarrow$ if true $\rightarrow$ y := z or true $\rightarrow$ skip fi or $\neg$z | x $\rightarrow$ skip fi $\{y \mid x\}$
(1, 6, SEC)
8) $\{y \mid x\}\ DO\ \{y \mid x \wedge \neg(z < x)\}$ (7, NREP, CONS)

La prueba no establece nada acerca de si el programa falla o diverge. Presentada como una *proof outline* de correctitud parcial, puede adquirir la siguiente forma:

$\{x \geq 1\}$
$z := 1 ; y := 1 ;$
$\{y \mid x\}$
do $z < x \rightarrow \{y \mid x\}$
$z := z + 1 ; \{y \mid x\}$
if $z \mid x \rightarrow \{y \mid x \wedge z \mid x\}$
if true $\rightarrow y := z$ or true $\rightarrow$ skip fi $\{y \mid x\}$
or $\neg z \mid x \rightarrow \{y \mid x \wedge \neg z \mid x\}$
skip $\{y \mid x\}$
fi
$\{y \mid x\}$
od
$\{y \mid x\}$

Axiomática para las pruebas de no divergencia y ausencia de falla

La instrucción de repetición es la única posible fuente de divergencia, y la de selección condicional, la única que puede provocar una falla en un programa. Las reglas NREP y NCOND no contemplan estos casos, por lo que el método axiomático cuenta con dos reglas más para cubrirlos, que son las siguientes:

15. *Regla de la no divergencia no determinística* (NREP*)

$$\frac{\langle p \wedge B_i \rangle\ S_i\ \langle p \rangle,\ i = 1, \ldots, n,\ \langle p \wedge B_i \wedge t = Z \rangle\ S_i\ \langle t < Z \rangle,\ i = 1, \ldots, n,\ p \rightarrow t \geq 0}{\langle p \rangle \text{ do } B_1 \rightarrow S_1 \text{ or} \ldots \text{or } B_n \rightarrow S_n \text{ od } \quad \langle p \wedge \bigwedge_{i=1,n} \neg B_i \rangle}$$

Al grupo de premisas de la regla NREP basado en un predicado p, se le agrega un segundo conjunto de premisas basado en una función entera t definida en términos de las variables de programa, y una premisa final que relaciona p con t. La variable de especificación Z no ocurre en p, B_i, S_i ni t, para todo i, su función es resguardar el valor de t previo a la ejecución de los S_i. Las premisas que se agregan son las que aseguran la no divergencia del DO:

- Por el segundo conjunto de premisas, t decrece después de cada iteración, cualquiera sea el subprograma S_i que se ejecute.
- Por la última premisa, t arranca y se mantiene positiva a lo largo de todo el DO.

De esta manera el DO no puede divergir, porque los valores enteros de t se reducen de iteración en iteración y nunca se hacen negativos. Obviamente la postcondición es $p \wedge \bigwedge_{i = 1,n} \neg B_i$. El valor inicial de t representa la cantidad máxima de iteraciones posibles.

Igual que la regla REP* para los *while*, la regla NREP* para los DO permite verificar al mismo tiempo su correctitud parcial y su no divergencia. La recomendación sigue siendo la que comentamos en el caso determinístico, verificar las dos propiedades por separado. Las técnicas para probarlas son distintas. En la prueba de correctitud parcial se plantea una inducción en torno de un predicado (invariante), se prueba que el predicado se cumple al principio del DO y que toda iteración lo preserva independientemente del subprograma que se ejecute. Así se concluye que el predicado se cumple a lo largo de toda la computación. En la prueba de no divergencia se utiliza una función (variante), que toda iteración del DO decrementa cualquiera sea el subprograma ejecutado. La función está definida en el conjunto bien fundado $(N_0, <)$ de los números naturales estándar, lo que asegura la inexistencia de cadenas descendentes infinitas.

16. *Regla del condicional no determinístico sin falla* (NCOND*)

$$\frac{p \rightarrow \bigvee_{i = 1,n} B_i,\ \langle p \wedge B_i \rangle\ S_i\ \langle q \rangle,\ i = 1, \ldots, n}{\langle p \rangle\ \text{if } B_1 \rightarrow S_1 \text{ or} \ldots \text{or } B_n \rightarrow S_n \text{ fi } \langle q \rangle}$$

Se agrega a la regla NCOND una premisa para asegurar que exista alguna guardia B_i verdadera, y por lo tanto, que el IF no falle. Algunos autores proponen aplicar directamente la regla NCOND* para tratar conjuntamente la correctitud parcial y la ausencia de falla. En el ejemplo siguiente mostramos su uso. Luego completamos la sección con un par de ejemplos de aplicación de la regla NREP*.

Ejemplo 7. Prueba de ausencia de falla en el programa que obtiene algún divisor

En el ejemplo anterior probamos la correctitud parcial del programa, que repetimos abajo, con respecto a la especificación $(x \geq 1, y \mid x)$:

```
Sdivn :: z := 1 ; y := 1 ;
        do z < x → z := z + 1 ;
                   if z | x → if true → y := z or true → skip fi
                   or ¬(z | x) → skip
                   fi
        od
```

Con la precondición establecida (con cualquiera en realidad) el programa no falla, porque en los dos IF del programa la disyunción de las guardias es *true*, como lo requiere la regla NCOND*.

Ejemplo 8. Prueba de no divergencia del programa que calcula el máximo común divisor

La forma del programa, también presentado previamente, es:

S_{mcd} :: do $x > y \rightarrow x := x - y$ or $x < y \rightarrow y := y - x$ od

Vamos a probar $\langle x = X \wedge X > 0 \wedge y = Y \wedge Y > 0 \rangle\ S_{mcd}\ \langle true \rangle$.

Tenemos que recurrir a la regla NREP*. El programa debe alcanzar la condición x = y para terminar. En cada iteración x o *y* se decrementan, y se mantienen mayores que 0 de acuerdo a sus valores iniciales y cómo decrecen cada vez. En base a esto proponemos como invariante el predicado $p = (x > 0 \wedge y > 0)$ y como variante la función t = x + y. El plan de prueba es:

a) La precondición del programa implica el invariante:
$(x = X \wedge X > 0 \wedge y = Y \wedge Y > 0) \rightarrow (x > 0 \wedge y > 0)$

b) Primeras premisas de NREP*. El predicado p es efectivamente un invariante del DO:
$\langle x > 0 \wedge y > 0 \wedge x > y \rangle\ x := x - y\ \langle x > 0 \wedge y > 0 \rangle$
$\langle x > 0 \wedge y > 0 \wedge x < y \rangle\ y := y - x\ \langle x > 0 \wedge y > 0 \rangle$

c) Segundas premisas de NREP*. El variante se decrementa después de cada iteración:
$\langle x > 0 \wedge y > 0 \wedge x > y \wedge x + y = Z \rangle\ x := x - y\ \langle x + y < Z \rangle$
$\langle x > 0 \wedge y > 0 \wedge x < y \wedge x + y = Z \rangle\ y := y - x\ \langle x + y < Z \rangle$

d) Ultima premisa de NREP*. El invariante implica que el variante siempre es positivo:
$(x > 0 \wedge y > 0) \rightarrow (x + y \geq 0)$

Todas las pruebas se logran fácilmente (las omitimos). La siguiente es una *proof outline* de no divergencia del programa:

$\langle x = X \wedge X > 0 \wedge y = Y \wedge Y > 0 \rangle$
$\langle$inv: $x > 0 \wedge y > 0$, var: $x + y\rangle$
do $x > y \rightarrow x := x - y$ or $x < y \rightarrow y := y - x$ od
$\langle true \rangle$

Ejemplo 9. Prueba de no divergencia del programa que devuelve un número entre 0 y N

Desarrollamos un ejemplo más de prueba de no divergencia por la forma del variante que utilizamos, típica en los programas no determinísticos. Escrito de otra manera, con una variable booleana, volvemos al programa que devuelve un número entre 0 y N:

S_{sel2} :: do $b \wedge x < N \rightarrow x := x + 1$ or $b \rightarrow b :=$ false od

Vamos a probar $\langle b \wedge x = 0 \wedge N \geq 0 \rangle$ S_{sel2} $\langle$true$\rangle$.

Los valores de x crecen de a una unidad, y el programa finaliza con x = N si siempre toma la primera dirección hasta llegar a dicha igualdad, o finaliza con x < N en caso contrario. En base a lo comentado, como invariante proponemos p = ($x \leq N$), y como variante:

t = if b then N – x + 1 else 0 fi

El valor inicial de t es N + 1, que representa la cantidad máxima de iteraciones posibles. Mientras se toma la primera dirección, t decrece en 1. En el momento que se toma la segunda dirección t vale al menos 1, y luego de la asignación booleana pasa a valer 0. Así, t decrece en escalones asociados a los números N + 1, N, N – 1, …, llegando a 0 con decrementos de una unidad o saltando de golpe desde un escalón intermedio. Las pruebas requeridas por la regla NREP* son las siguientes:

a) La precondición del programa implica el invariante:
   $(b \wedge x = 0 \wedge N \geq 0) \rightarrow x \leq N$

b) El predicado p es efectivamente un invariante del DO:
   $\langle x \leq N \wedge b \wedge x < N \rangle$ x := x + 1 $\langle x \leq N \rangle$
   $\langle x \leq N \wedge b \rangle$ b := false $\langle x \leq N \rangle$

c) El variante se decrementa después de cada iteración:
   $\langle x \leq N \wedge b \wedge x < N \wedge t = Z \rangle$ x := x + 1 $\langle t < Z \rangle$
   $\langle x \leq N \wedge b \wedge t = Z \rangle$ b := false $\langle t < Z \rangle$

d) El invariante implica que el variante siempre es positivo.
   $x \leq N \rightarrow t \geq 0$

Omitimos las pruebas de (a) y (b), todas las fórmulas se obtienen fácilmente.

La primera fórmula de la parte (c) se cumple porque de $x \leq N \wedge b \wedge x < N$, $t = N - x + 1$, y luego de la asignación *x := x + 1* el valor de t se decrementa en 1.

La segunda fórmula de la parte (c) se cumple porque de $x \leq N \wedge b$, $t = N - x + 1 > 0$, y luego de la asignación *b := false* el valor de t es 0.

Finalmente, también la parte (d) se cumple, t siempre es positivo, considerando el invariante $x \leq N$: si vale b, $t = N - x + 1 > 0$, y si no vale b, $t = 0$.

Ejemplo de desarrollo sistemático de programa

En esta sección ejemplificamos el desarrollo sistemático de un programa no determinístico guiado por el método de verificación descripto. El esquema de programación y prueba en simultáneo que presentamos es una adaptación del que utilizamos para los programas determinísticos. Nuevamente consideramos un programa con una estructura muy sencilla, un fragmento de inicializaciones seguido de una instrucción DO:

$S :: S_0$; do $B_1 \rightarrow S_1$ or…or $B_n \rightarrow S_n$ od

De acuerdo a la axiomática, para que el programa satisfaga la fórmula de correctitud total:

$\langle r \rangle\ S\ \langle q \rangle$

tienen que cumplirse cinco requerimientos, una vez definidos un predicado p y una función t para el DO:

1. A partir de la precondición r, el fragmento S_0 termina estableciendo el invariante p: $\langle r \rangle\ S_0\ \langle p \rangle$.
2. El predicado p es un invariante del DO: $\langle p \wedge B_i \rangle\ S_i\ \langle p \rangle$, i = 1, …, n.
3. La función t decrece después de cada iteración del DO: $\langle p \wedge B_i \wedge t = Z \rangle\ S_i\ \langle t < Z \rangle$, i = 1, …, n.
4. El predicado p asegura que la función t siempre es positiva: $p \rightarrow t \geq 0$.
5. El DO termina estableciendo la postcondición q: $(p \wedge \bigwedge_{i = 1,n} \neg B_i) \rightarrow q$.

Los requerimientos se pueden resumir con la siguiente *proof outline* genérica:

$\langle r \rangle\ S_0$; $\langle$inv: p, var: t$\rangle$ do $B_1\ \langle p \wedge B_1 \rangle \rightarrow S_1\ \langle p \rangle$ or…or $B_n\ \langle p \wedge B_n \rangle \rightarrow S_n\ \langle p \rangle$ od $\langle p \wedge \bigwedge_{i = 1,n} \neg B_i \rangle\ \langle q \rangle$

A continuación, desarrollamos un programa concreto siguiendo los lineamientos de este esquema.

Ejemplo 10. Desarrollo sistemático de un programa que busca el menor elemento común de tres arreglos de números enteros

Vamos a construir un programa con la forma:

S_0 ; do $B_1 \rightarrow S_1$ or…or $B_n \rightarrow S_n$ od

asumiendo que el lenguaje incluye arreglos. Los datos de entrada son tres arreglos de números enteros *a*, b y c, de solo lectura, ordenados de manera ascendente, sin elementos repetidos y con uno o más elementos comunes. El programa debe devolver las posiciones en los tres arreglos del menor elemento común. Omitimos la especificación de la precondición, en lo que sigue la referenciamos con r. En cuanto a la postcondición, la especificamos del siguiente modo:

$q = (i = im \wedge j = jm \wedge k = km)$

con la idea de que i, j y k sean variables del programa utilizadas como índices de los arreglos *a*, b y c, respectivamente, siendo im, jm y km variables de especificación que representan las correspondientes posiciones del menor elemento común. Considerando un algoritmo de búsqueda que recorra mediante una instrucción DO los tres arreglos hasta encontrar dicho elemento, proponemos el invariante:

$p = (r \wedge 0 \leq i \leq im \wedge 0 \leq j \leq jm \wedge 0 \leq k \leq km)$

volviendo a recurrir a la heurística de debilitamiento de la postcondición: sustituimos i = im, j = jm y k = km, por $i \leq im$, $j \leq jm$ y $k \leq km$.

Como fragmento inicial del programa definimos:

i := 0 ; j := 0 ; k := 0

para que al comienzo i, j y k apunten al inicio de los arreglos: a[0], b[0] y c[0]. La ejecución del fragmento debe establecer el invariante por primera vez (requerimiento 1), lo que se cumple aplicando el axioma ASI y las reglas SEC y CONS:

$\langle r \rangle$ i := 0 ; j := 0 ; k := 0 $\langle r \wedge 0 \leq i \leq im \wedge 0 \leq j \leq jm \wedge 0 \leq k \leq km \rangle$

Pasamos ahora a construir el DO, de modo tal que su cuerpo preserve el invariante, así que ya consideramos cumplido el requerimiento 2. Como variante del DO proponemos:

$t = (im - i) + (jm - j) + (km - k)$

y definimos como instrucciones del DO las asignaciones *i := i + 1*, *j := j + 1* y *k := k + 1*. Las mismas implementan el recorrido de los arreglos y decrementan el valor de t (requerimiento 3). El invariante implica que el variante siempre es positivo (requerimiento 4):

$(r \wedge 0 \leq i \leq im \wedge 0 \leq j \leq jm \wedge 0 \leq k \leq km) \rightarrow (im - i) + (jm - j) + (km - k) \geq 0$

La siguiente *proof outline* parcial muestra lo construido hasta el momento:

$\langle r \rangle$
$i := 0 ; j := 0 ; k := 0 ;$
$\langle inv: r \wedge 0 \leq i \leq im \wedge 0 \leq j \leq jm \wedge 0 \leq k \leq km, var: (im - i) + (jm - j) + (km - k) \rangle$
do $B_1 \rightarrow \langle r \wedge 0 \leq i \leq im \wedge 0 \leq j \leq jm \wedge 0 \leq k \leq km \wedge B_1 \rangle$ $i := i + 1$
or $B_2 \rightarrow \langle r \wedge 0 \leq i \leq im \wedge 0 \leq j \leq jm \wedge 0 \leq k \leq km \wedge B_2 \rangle$ $j := j + 1$
or $B_3 \rightarrow \langle r \wedge 0 \leq i \leq im \wedge 0 \leq j \leq jm \wedge 0 \leq k \leq km \wedge B_3 \rangle$ $k := k + 1$
od
$\langle r \wedge 0 \leq i \leq im \wedge 0 \leq j \leq jm \wedge 0 \leq k \leq km \wedge \neg B_1 \wedge \neg B_2 \wedge \neg B_3 \rangle$

Resta encontrar B_1, B_2 y B_3 y probar el requerimiento 5. Para las guardias, lo lógico sería utilizar $i \neq im$, $j \neq jm$ y $k \neq km$, pero las variables de especificación no pueden aparecer en el programa. Veamos cómo llegamos a otras guardias que expresen lo mismo. Se cumple que:

- $p \wedge i \neq im$ es equivalente a $p \wedge i < im$, que establece que a[i] no es el menor elemento común.
- $p \wedge a[i] < b[j]$ implica $a[i] < b[jm] = a[im]$, que a su vez implica $i < im$.

De esta manera, se puede elegir como B_1 la expresión $a[i] < b[j]$. De manera similar se llega a $b[j] < c[k]$ para B_2 y $c[k] < a[i]$ para B_3. Así obtenemos la siguiente *proof outline* completa:

$\langle r \rangle$
$i := 0 ; j := 0 ; k := 0 ;$
$\langle inv: r \wedge 0 \leq i \leq im \wedge 0 \leq j \leq jm \wedge 0 \leq k \leq km, var: (im - i) + (jm - j) + (km - k) \rangle$
do $a[i] < b[j] \rightarrow \langle r \wedge 0 \leq i \leq im \wedge 0 \leq j \leq jm \wedge 0 \leq k \leq km \wedge a[i] < b[j] \rangle$ $i := i + 1$
or $b[j] < c[k] \rightarrow \langle r \wedge 0 \leq i \leq im \wedge 0 \leq j \leq jm \wedge 0 \leq k \leq km \wedge b[j] < c[k] \rangle$ $j := j + 1$
or $c[k] < a[i] \rightarrow \langle r \wedge 0 \leq i \leq im \wedge 0 \leq j \leq jm \wedge 0 \leq k \leq km \wedge c[k] < a[i] \rangle$ $k := k + 1$
od
$\langle r \wedge 0 \leq i \leq im \wedge 0 \leq j \leq jm \wedge 0 \leq k \leq km \wedge \neg(a[i] < b[j]) \wedge \neg(b[j] < c[k]) \wedge \neg(c[k] < a[i]) \rangle$
$\langle i = im \wedge j = jm \wedge k = km \rangle$

En efecto, la prueba se completa porque se verifica el requerimiento 5: se cumple porque a la salida del DO vale a[i] = b[j] = c[k], que en conjunción con el invariante implica la postcondición del programa:

$(r \wedge 0 \leq i \leq im \wedge 0 \leq j \leq jm \wedge 0 \leq k \leq km \wedge a[i] = b[j] = c[k]) \rightarrow i = im \wedge j = jm \wedge k = km$

En las próximas dos secciones completamos la presentación del método, planteando algunas variantes de la axiomática descripta. Las mismas contemplan cambios en el lenguaje de programación, con el objeto de analizar otros aspectos relevantes de la verificación de programas no determinísticos.

Asignaciones aleatorias

El primero de los cambios referidos en el lenguaje de programación consiste en la inclusión de la instrucción de *asignación aleatoria*. A diferencia de la selección condicional y la repetición no determinísticas, la asignación aleatoria introduce no determinismo mediante una selección de datos. Su sintaxis es:

x := ?

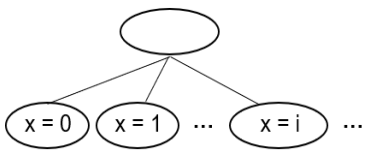
y su semántica informal, la asignación de algún número natural a la variable x. De esta manera, la instrucción fuerza el uso de un solo tipo de datos.

Un mejor nombre para la instrucción sería *asignación no determinística*, porque a la aleatoriedad se la asocia en general con alguna asunción de probabilidad. Por ejemplo, el programa:

do $x \neq n \rightarrow$ x := ? od

puede divergir, puede suceder que la asignación x := ? nunca seleccione el número natural n.

Otra particularidad de la asignación aleatoria es que el no determinismo que introduce es *no acotado*: ahora se puede producir un conjunto infinito (contable) de estados finales sin computaciones que divergen, como lo muestra la siguiente figura:



Notar que el árbol de computaciones correspondiente a *x := ?* a partir de un determinado estado tiene infinitas hojas y todas sus ramas son finitas.

En la literatura se describe también una variante, la asignación $x := ? \leq y$, que selecciona un valor entre 0 e y. El no determinismo que introduce se denomina *finito*, es acotado, pero en particular, como se observa, la cantidad de estados finales generados no depende sólo de la sintaxis. De todos modos, el análisis del no determinismo finito se puede cubrir con el del no determinismo acotado que consideramos previamente, porque la asignación $x := ? \leq y$ se puede simular con una instrucción DO de la siguiente manera:

b := true ; x := 0 ;
do b $\wedge$ x < y $\rightarrow$ x := x + 1
 or b $\wedge$ x < y $\rightarrow$ b := false
od

Adaptación de la axiomática

Para las pruebas de programas con asignaciones aleatorias, extendemos el método con el siguiente axioma:

17. Axioma de la asignación aleatoria (NASI)
 $\{\forall x: (x \geq 0 \rightarrow p)\}\ x := ?\ \{p\}$

El axioma establece que si se cumple el predicado p en términos de x después de la asignación, significa que antes de la asignación se cumplía p para cualquier valor entero $x \geq 0$.

Además del nuevo axioma, la inclusión de la asignación aleatoria requiere una adaptación de la axiomática relacionada con la prueba de no divergencia: la nueva instrucción provoca que el dominio N de los números naturales no alcance para definir todos los casos posibles de variantes de los DO, como podemos apreciar en el siguiente ejemplo. Dado el programa:

S_{asin} :: do b $\wedge$ x > 0 $\rightarrow$ x := x – 1
 or b $\wedge$ x < 0 $\rightarrow$ x := x + 1
 or $\neg$b $\rightarrow$ x := ? ; b := true
 od

si al comienzo la variable b es verdadera, el programa termina al cabo de |x| iteraciones, y si es falsa, el programa también termina, pero no se puede establecer en cuántas iteraciones porque la cantidad no depende del estado inicial, se conoce recién después de la ejecución de la asignación aleatoria. Por lo tanto, el valor inicial de un variante t para el DO debe cumplir:

$t \geq x$, para todo $x \geq 0$

lo que es imposible si t es una función entera. En este caso hay que recurrir a un conjunto bien fundado más grande que el que venimos utilizando. Por ejemplo, podemos emplear N ∪ {ω}, con la relación < habitual, siendo ω el primer ordinal infinito, que cumple que es mayor que todos los números naturales. La regla de la no divergencia no determinística adaptada tiene la siguiente forma:

18. *Regla de la no divergencia no determinística generalizada* (NREPG*)

$$\frac{\langle p \wedge B_i \rangle\ S_i\ \langle p \rangle,\ i = 1, \ldots, n,\ \langle p \wedge B_i \wedge t = w \rangle\ S_i\ \langle t < w \rangle,\ i = 1, \ldots, n,\ p \rightarrow t \in W}{\langle p \rangle\ \text{do } B_1 \rightarrow S_1 \text{ or} \ldots \text{or } B_n \rightarrow S_n \text{ od}\quad \langle p \wedge \bigwedge_{i=1,n} \neg B_i \rangle}$$

La regla generaliza la que presentamos previamente. Requiere utilizar algún conjunto bien fundado apropiado W. Todo S_i del DO debe decrementar el variante t, y el invariante p debe asegurar que t siempre varíe dentro del conjunto W (hasta alcanzar a lo sumo un valor w_0 sin predecesores, es decir un *minimal*). La variable w no ocurre en p, B_i, S_i ni t, para todo i. Presentamos a continuación un ejemplo de aplicación de la regla.

Ejemplo 11. Prueba de no divergencia de un programa con una asignación aleatoria

Vamos a probar la no divergencia del programa que mostramos antes:

S_{asin} :: do $b \wedge x > 0 \rightarrow x := x - 1$
or $b \wedge x < 0 \rightarrow x := x + 1$
or $\neg b \rightarrow x := ?\ ;\ b := true$
od

Recurrimos a la regla NREPG*. De acuerdo a lo que observamos previamente, proponemos utilizar el conjunto bien fundado (W = N ∪ {ω}, <) y como variante del DO la función:

t = if b then |x| else ω fi

que expresa la cantidad de iteraciones del programa cuando b es verdadera, y un valor mayor que todos los números naturales cuando b es falsa. Como invariante del DO elegimos simplemente el predicado *true*. El primer conjunto de premisas de la regla se verifica trivialmente:

$\langle true \wedge b \wedge x > 0 \rangle\ x := x - 1\ \langle true \rangle$
$\langle true \wedge b \wedge x < 0 \rangle\ x := x + 1\ \langle true \rangle$
$\langle true \wedge \neg b \rangle\ x := ?\ ;\ b := true\ \langle true \rangle$

Lo mismo sucede con la última premisa de la regla:

$true \rightarrow (\text{if } b \text{ then } |x| \text{ else } \omega \text{ fi}) \in N \cup \{\omega\}$

Nos queda por probar el segundo grupo de premisas:

$\langle true \wedge b \wedge x > 0 \wedge (\text{if } b \text{ then } |x| \text{ else } \omega \text{ fi}) = w \rangle\ x := x - 1\ \langle (\text{if } b \text{ then } |x| \text{ else } \omega \text{ fi}) < w \rangle$
$\langle true \wedge b \wedge x < 0 \wedge (\text{if } b \text{ then } |x| \text{ else } \omega \text{ fi}) = w \rangle\ x := x + 1\ \langle (\text{if } b \text{ then } |x| \text{ else } \omega \text{ fi}) < w \rangle$
$\langle true \wedge \neg b \wedge (\text{if } b \text{ then } |x| \text{ else } \omega \text{ fi}) = w \rangle\ x := ?\ ;\ b := true\ \langle (\text{if } b \text{ then } |x| \text{ else } \omega \text{ fi}) < w \rangle$

Las primeras dos fórmulas se obtienen fácilmente, por lo que omitimos las pruebas. Finalmente, la prueba de la última fórmula es la siguiente:

1. $\langle |x| < w \rangle\ b := true\ \langle (\text{if } b \text{ then } |x| \text{ else } \omega \text{ fi}) < w \rangle$ (ASI, CONS)
2. $\langle \forall x: (x \geq 0 \rightarrow |x| < w) \rangle\ x := ?\ \langle |x| < w \rangle$ (NASI)
3. $\langle w = \omega \rangle\ x := ?\ ;\ b := true\ \langle (\text{if } b \text{ then } |x| \text{ else } \omega \text{ fi}) < w \rangle$ (1, 2, SEC, CONS)
4. $\langle true \wedge \neg b \wedge (\text{if } b \text{ then } |x| \text{ else } \omega \text{ fi}) = w \rangle\ x := ?\ ;\ b := true\ \langle (\text{if } b \text{ then } |x| \text{ else } \omega \text{ fi}) < w \rangle$ (3, CONS)

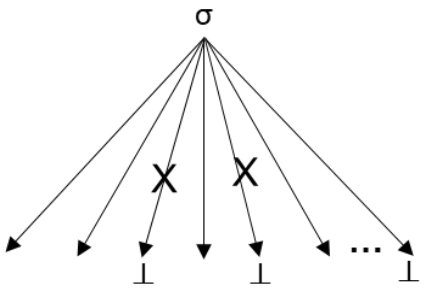
Las asignaciones aleatorias se utilizan también para modelizar el *fairness*, como mostramos en la próxima sección.

Fairness

Un último cambio que tratamos a continuación, de una manera muy introductoria, en el lenguaje de programación (y correspondientemente en el método axiomático), es la incorporación de *fairness* en su semántica.

El *fairness* es una característica semántica presente en muchos lenguajes de programación, que impone restricciones en la forma en que pueden progresar las computaciones infinitas de los programas no determinísticos. Su definición varía según el contexto en el que se utiliza. Considerando específicamente los programas con los que trabajamos, establece que en una computación infinita no puede existir una dirección de una instrucción DO habilitada infinitas veces, permanente o intermitentemente, que nunca sea tomada, es decir, no puede quedar postergada indefinidamente. Si el requerimiento es que la dirección esté habilitada de manera permanente el *fairness* se denomina *fairness débil*, y si alcanza con que esté habilitada de manera intermitente el *fairness* se identifica como *fairness fuerte*. Ambas denominaciones se justifican porque la existencia de *fairness* fuerte implica la existencia de *fairness* débil (en algunos textos se conocen como *fairness y justice*, respectivamente).

De esta manera, un escenario con *fairness* puede reducir el grado de no determinismo de un programa: se eliminan todas sus computaciones infinitas inválidas, *unfair*, sólo quedan sus computaciones infinitas válidas, *fair*, y sus computaciones finitas. En términos del árbol de computaciones asociado al programa, el efecto del *fairness* se puede pensar como una poda de algunas ramas infinitas, como mostramos en la siguiente figura:



Las computaciones infinitas del árbol se identifican con el símbolo ⊥. El programa representado debe entenderse como que produce desde un estado inicial σ tres computaciones infinitas en un escenario sin *fairness*, y sólo una en un escenario con *fairness*. Las ramas tachadas representan las computaciones *unfair* del programa.

Presentamos algunos ejemplos de programas explicando su comportamiento sin y con *fairness*, para repasar las definiciones anteriores. Comenzamos, volviendo al programa que obtiene cualquier número natural:

S_{nat} :: x := 0 ; b := true ; do b $\rightarrow$ x := x + 1 or b $\rightarrow$ b := false od

Sin *fairness*, el programa puede divergir tomando siempre la primera dirección del DO. Pero con *fairness* débil, que asegura que en una computación infinita una dirección de un DO habilitada permanentemente no puede quedar postergada indefinidamente, el programa termina siempre: b siempre es verdadera, así que el programa alguna vez tomará inexorablemente la segunda dirección, que es la que conduce a la terminación.

El siguiente programa, derivado del anterior, devuelve sólo números naturales pares (el predicado par(x) es verdadero sii x es par):

S_{par} :: x := 0 ; b := true ; do b $\rightarrow$ x := x + 1 or b $\wedge$ par(x) $\rightarrow$ b := false od

También en este caso, sin *fairness*, el programa puede divergir tomando siempre la primera dirección del DO. Y ahora el *fairness* débil no lo evita, porque la segunda dirección, que es la que lleva a la terminación, no está habilitada permanentemente (la guardia b $\wedge$ par(x) resulta alternativamente verdadera y falsa). Para que el programa la tome inexorablemente alguna alguna vez el *fairness* debe ser fuerte, el cual asegura que en una computación infinita una dirección de un DO habilitada infinitas veces no puede quedar postergada indefinidamente.

Como último ejemplo mostramos un programa que puede divergir aún con *fairness* fuerte:

S_{syb} :: x := 1 ; do $x > 0 \rightarrow x := x + 1$ or $x > 0 \rightarrow x := x - 1$ od

En efecto, computaciones infinitas del programa como la que alterna sucesivamente entre las dos direcciones del DO comenzando por la primera, no violan la hipótesis de *fairness* fuerte.

Notar que asumiendo *fairness*, los programas anteriores que generan números naturales no divergen, pero aun así producen conjuntos infinitos de estados finales. Esto significa que con una semántica con *fairness*, en los programas no determinísticos se manifiesta el no determinismo no acotado, igual que cuando se utilizan asignaciones aleatorias. La relación entre el *fairness* y las asignaciones aleatorias la tratamos más adelante.

Adaptación de la axiomática

Pasamos a describir la adaptación del método axiomático para la verificación de programas asumiendo *fairness*. Por el carácter introductorio de la sección sólo presentamos la idea general. Como el f*airness* sólo impacta en las computaciones infinitas, la única regla que se debe considerar es la de la prueba de no divergencia de una instrucción DO. Cabe remarcar en este punto algo ya anticipado en la introducción del libro, la imposibilidad de la lógica de Hoare para incluir el *fairness* directamente en las reglas de prueba, y así en las fórmulas de correctitud.

Primero consideramos el *fairness* débil:

19. *Regla de la no divergencia no determinística asumiendo fairness débil* (NREPD*)

Básicamente, la forma de la regla es la misma que planteamos antes: se elige un conjunto bien fundado apropiado W (no necesariamente los números naturales porque el no determinismo es no acotado), se definen un invariante p y un variante t del DO, y se requiere que cada iteración preserve p y decremente t, con la función t siempre definida en W. Lo que ahora cambia es que no se exige que el variante se decremente cualquiera sea la dirección que se tome, sino que alcanza con que lo haga sólo al tomarse determinadas direcciones *útiles* (por eso esta variante del método se denomina *método de las direcciones útiles*), y que no se incremente al tomarse cualquiera de las otras. Más precisamente:

- Por cada valor w no minimal que pueda adoptar el variante a lo largo del DO, se define una partición del conjunto de direcciones: un conjunto D_w, no vacío, y un conjunto D'_w.
- Los conjuntos D_w tienen direcciones útiles. Se las denomina así porque al tomarlas se acorta la distancia a la finalización del DO, es decir, se decrementa el variante:

  $\langle p \wedge B_i \wedge t = w \rangle \; S_i \; \langle t < w \rangle, \; i \in D_w$

- Por su parte, las direcciones de los conjuntos D'_w cumplen que al tomarlas no se alarga la distancia a la finalización del DO, es decir, no se incrementa el variante:

$$\langle p \wedge B_i \wedge t = w \rangle \; S_i \; \langle t \leq w \rangle, \; i \in D'_w$$

Con estas premisas el DO no diverge, porque el *fairness* débil asegura que en todas las instancias de t con un w no minimal, se va a tomar alguna vez una dirección que provoque su decremento (en un conjunto bien fundado). La siguiente prueba ejemplifica el uso de la regla.

Ejemplo 12. Prueba de no divergencia de un programa asumiendo *fairness* débil

Tomando $b_1 \wedge b_2 \wedge x = 0 \wedge y = 0$ como precondición, el siguiente programa devuelve algún par de números naturales x e *y*:

S_{xy} :: do $b_1 \wedge b_2 \rightarrow x := x + 1$
or $b_1 \wedge b_2 \rightarrow b_1 :=$ false
or $\neg b_1 \wedge b_2 \rightarrow y := y + 1$
or $\neg b_1 \wedge b_2 \rightarrow b_2 :=$ false
od

Claramente, con *fairness débil* el programa no diverge: en algún momento se debe tomar la segunda dirección, y después la cuarta dirección, que lleva a la terminación del programa. Vamos a probarlo formalmente con la regla descripta.

Como conjunto bien fundado elegimos (W = {0, 1, 2}, <). Estos números representan la distancia a la terminación del programa: el 2 corresponde a las instancias en que b_1 y b_2 son verdaderas, el 1 a las instancias con b_1 falsa y b_2 verdadera, y finalmente el 0 a la instancia final con las dos variables booleanas falsas.

Definimos el variante en términos de dichos números, alternativa a la definición en términos de las variables de programa que venimos utilizando:

t = if $b_1 \wedge b_2$ then 2 else if $\neg b_1 \wedge b_2$ then 1 else 0 fi fi

y planteamos las siguientes particiones del conjunto de direcciones del DO:

$D_2 = \{2\}$ y $D'_2 = \{1, 3, 4\}$ para t = 2
$D_1 = \{4\}$ y $D'_1 = \{1, 2, 3\}$ para t = 1

porque cuando t = 2, la única dirección útil es la segunda, y cuando t = 1, la única dirección útil es la cuarta.

Como invariante del DO sirve el predicado $p = (b_1 \rightarrow b_2)$.

Las fórmulas para probar, en términos de p y t, son las siguientes:

a) Premisas $\langle p \wedge B_i \rangle\ S_i\ \langle p \rangle$, i = 1, …, n:

$\langle p \wedge b_1 \wedge b_2 \rangle\ x := x + 1\ \langle p \rangle$

$\langle p \wedge b_1 \wedge b_2 \rangle\ b_1 := \text{false}\ \langle p \rangle$

$\langle p \wedge \neg b_1 \wedge b_2 \rangle\ y := y + 1\ \langle p \rangle$

$\langle p \wedge \neg b_1 \wedge b_2 \rangle\ b_2 := \text{false}\ \langle p \rangle$

b) Premisas $\langle p \wedge B_i \wedge t = 2 \rangle\ S_i\ \langle t < 1 \rangle$, i = 1, …, n:

$\langle p \wedge b_1 \wedge b_2 \wedge t = 2 \rangle\ x := x + 1\ \langle t \leq 2 \rangle$

$\langle p \wedge b_1 \wedge b_2 \wedge t = 2 \rangle\ b_1 := \text{false}\ \langle t < 2 \rangle$

$\langle p \wedge \neg b_1 \wedge b_2 \wedge t = 2 \rangle\ y := y + 1\ \langle t \leq 2 \rangle$

$\langle p \wedge \neg b_1 \wedge b_2 \wedge t = 2 \rangle\ b_2 := \text{false}\ \langle t \leq 2 \rangle$

c) Premisas $\langle p \wedge B_i \wedge t = 1 \rangle\ S_i\ \langle t < 1 \rangle$, i = 1, …, n:

$\langle p \wedge b_1 \wedge b_2 \wedge t = 1 \rangle\ x := x + 1\ \langle t \leq 1 \rangle$

$\langle p \wedge b_1 \wedge b_2 \wedge t = 1 \rangle\ b_1 := \text{false}\ \langle t \leq 1 \rangle$

$\langle p \wedge \neg b_1 \wedge b_2 \wedge t = 1 \rangle\ y := y + 1\ \langle t \leq 1 \rangle$

$\langle p \wedge \neg b_1 \wedge b_2 \wedge t = 1 \rangle\ b_2 := \text{false}\ \langle t < 1 \rangle$

d) Premisa $p \rightarrow t \in W$:

$p \longrightarrow t \in \{0, 1, 2\}$

Omitimos las pruebas, todas se logran fácilmente. En particular, hay fórmulas en las partes (b) y (c) que se prueban trivialmente porque tienen precondiciones falsas, derivadas de situaciones que no pueden ocurrir en el programa.

Ahora consideramos el *fairness* fuerte. El método de las direcciones útiles también se puede aplicar cuando se asume *fairness* fuerte. De todos modos, para este caso aprovechamos a presentar los aspectos salientes de otra variante del método axiomático, variante transformacional denominada *método del planificador explícito*, porque incluye un planificador *fair* en el programa a verificar, implementado por medio de asignaciones aleatorias. Se aprovecha la relación que existe entre el *fairness* y las asignaciones aleatorias, en el sentido de que en ambos casos el no determinismo producido es no acotado. La regla correspondiente es la siguiente:

20. *Regla de la no divergencia no determinística asumiendo fairness fuerte* (NREPF*)

Consiste en dos pasos:

- Primero se transforma el programa, agregándole asignaciones aleatorias con el objeto de eliminarle todas sus computaciones *unfair*.
- Luego se prueba que el programa transformado no diverge, recurriendo directamente a las reglas de verificación con asignaciones aleatorias.

De esta manera, el método también da una idea de cómo implementar un escenario con *fairness*. A continuación, ejemplificamos su uso. En lugar de considerar un programa en particular utilizamos un esquema de programa.

Ejemplo 13. Prueba de no divergencia de un programa asumiendo *fairness* fuerte

Consideramos un esquema de programa muy simple, con dos direcciones. Etiquetamos las direcciones para facilitar la descripción:

S_{pla} :: do 1: $B_1 \rightarrow S_1$ or 2: $B_2 \rightarrow S_2$ od

De acuerdo al método, primero introducimos en el programa asignaciones aleatorias para implementar un planificador de *fairness* fuerte. El programa transformado queda de la siguiente manera:

S'_{pla} :: z_1 := ? ; z_2 := ? ;
do 1: $B_1 \wedge z_1 \leq z_2 \rightarrow S_1$; z_1 := ? ; if $B_2 \rightarrow z_2 := z_2 - 1$ or $\neg B_2 \rightarrow$ skip fi
or 2: $B_2 \wedge z_2 < z_1 \rightarrow S_2$; z_2 := ? ; if $B_1 \rightarrow z_1 := z_1 - 1$ or $\neg B_1 \rightarrow$ skip fi
od

La transformación del programa permite efectivamente suprimir sus computaciones *unfair*:

- Se introducen dos variables, z_1 y z_2, que representan las prioridades asignadas a las direcciones 1 y 2, respectivamente. El decremento de una variable implica el aumento de la prioridad de la dirección asociada.
- Al elegirse una dirección, su prioridad se reinicializa, y la prioridad de la otra dirección aumenta, siempre que esté habilitada. El decremento gradual de cada z_i asegura que la dirección i asociada no queda postergada indefinidamente.

Finalmente, la prueba debe completarse, ya sin asunción de *fairness*, empleando el axioma NASI y la regla NREPG*.

Observaciones finales

Hemos analizado la verificación axiomática de una clase muy conocida de programas secuenciales no determinísticos, con selecciones condicionales y repeticiones con comandos guardados, utilizando una generalización del método axiomático presentado en el capítulo anterior. Estructuramos el análisis en base a tres versiones del lenguaje de programación, la primera sólo con comandos guardados, la segunda incorporando asignaciones aleatorias, y la tercera asumiendo una semántica con *fairness*. La axiomática descripta sigue siendo composicional, consistente y completa, y una guía efectiva para el desarrollo sistemático de programas.

La manera de especificar programas no cambia con respecto a lo definido previamente. A pesar de que la ejecución de un programa no determinístico determina ya no una función de un estado inicial a un estado final sino una relación de un estado inicial a un conjunto de estados finales, se pueden seguir utilizando las fórmulas de correctitud con pre y postcondiciones, interpretando apropiadamente el concepto de correctitud.

Las reglas de prueba se basan en los mismos conceptos fundamentales que destacamos en el método para programas determinísticos, como el invariante y el variante que caracterizan una instrucción de repetición. La novedad en el escenario no determinístico es que las reglas deben asegurar que los resultados sean correctos en múltiples computaciones. En particular, el invariante debe preservarse y el variante debe decrementarse cualquiera sea la dirección que se tome toda vez en una repetición.

Además de la correctitud parcial y la no divergencia, hemos considerado una tercera propiedad para verificar, la ausencia de falla, de la familia de propiedades *safety*. Para simplificar la presentación, hemos tenido en cuenta sólo un tipo de falla, producto de una selección condicional sin guardias verdaderas. En el capítulo 5, dedicado a los programas concurrentes, consideramos un segundo tipo de falla.

El no determinismo producido por programas sólo con comandos guardados es acotado, la cantidad de computaciones es finita. De esta manera, un programa genera un conjunto finito de estados finales o tiene una computación que diverge. Incorporando asignaciones aleatorias el no determinismo producido es no acotado, aún sin divergencia el grado del árbol de computaciones de un programa es infinito (contable). Esto provoca que deba recurrirse a los ordinales infinitos para probar la no divergencia. El *fairness* también produce no determinismo no acotado, y como su efecto es la eliminación de determinadas computaciones infinitas, *unfair*, la única propiedad impactada vuelve a ser la no divergencia. Para este caso hemos descripto dos adaptaciones a la regla de prueba, que por limitación de la lógica de Hoare no puede incluir el *fairness* directamente en las fórmulas de correctitud. Una adaptación relaja el requerimiento del

decrecimiento del variante, no lo exige para todas las direcciones de una repetición sino para determinadas direcciones útiles. La otra adaptación de la regla es de índole transformacional, plantea primero introducir asignaciones aleatorias al programa a verificar para eliminar sus computaciones *unfair*, y luego probar el programa transformado sin asunción de *fairness*.

Notas adicionales

Método reduccionista para verificar programas concurrentes

Los programas concurrentes pueden transformarse sistemáticamente en programas secuenciales no determinísticos, asumiendo la semántica de *interleaving*. Hemos presentado un ejemplo muy simple en este capítulo. Como verificar programas concurrentes es más difícil que verificar programas secuenciales, resulta razonable recurrir a dichas transformaciones para verificar directamente los programas secuenciales, utilizando una axiomática como la que hemos descripto. Este método se conoce como *reduccionista* (Ashcroft y Manna, 1971). Así planteado no sirve como guía para el desarrollo sistemático de programas concurrentes. Sin embargo, existen métodos que basados en la transformación que establece sistematizan la construcción de programas concurrentes a partir de programas secuenciales no determinísticos, por ejemplo, el que se propone en (Chandy y Misra, 1988).

Presentamos a continuación un segundo ejemplo del método, con un programa más complejo, con variables compartidas. El programa es el siguiente (ya adelantamos el primer paso de la transformación, que consiste en anotar la posición de las acciones atómicas con números consecutivos - sobre las acciones atómicas damos precisiones en el capítulo 5 -):

```
        i := 1 ; j := 2 ; k1 := N + 1 ; k2 := N + 1 ;
[S1 :: 0. while i < min(k1, k2) do  ||  S2 :: 0. while j < min(k1, k2) do
        1. if a[i] > 0                          1. if a[j] > 0
        2. then k1 := i                         2. then k2 := j
        3. else i := i + 2 fi                   3. else j := j + 2 fi
        4. od                                   4. od]
        k := min(k1, k2)
```

El programa devuelve el índice del primer elemento positivo de un arreglo de números enteros *a* de tamaño $N \geq 1$. El proceso S_1 recorre los elementos con índice i impar, y el proceso S_2 los que tienen índice j par. Las variables k_1 y k_2 se utilizan para determinar las finalizaciones de los dos recorridos, respectivamente. La función min obtiene el mínimo. Si se encuentra un elemento positivo, el programa devuelve el índice resultante k y termina. Si en cambio todos los elementos del arreglo son negativos, cuando los dos procesos llegan hasta las posiciones tope el programa

termina y las variables k_1 y k_2 quedan con su valor inicial N + 1. Mostramos en lo que sigue cómo resulta el programa secuencial no determinístico (sólo lo que corresponde a la composición concurrente). Se contemplan todas las posibles intercalaciones de las acciones atómicas, y se incorpora una variable auxiliar por cada proceso para representar sus posiciones (variables pc_1 y pc_2):

$pc_1 := 0 ; pc_2 := 0$
do $pc_1 = 0 \wedge i < \min(k_1, k_2) \longrightarrow pc_1 := 1$
or $pc_1 = 0 \wedge \neg(i < \min(k_1, k_2)) \longrightarrow pc_1 := 4$
or $pc_1 = 1 \wedge a[i] > 0 \longrightarrow pc_1 := 2$
or $pc_1 = 1 \wedge \neg(a[i] > 0) \longrightarrow pc_1 := 3$
or $pc_1 = 2 \longrightarrow k_1 := i ; pc_1 := 0$
or $pc_1 = 3 \longrightarrow i := i + 2 ; pc_1 := 0$
or $pc_2 = 0 \wedge j < \min(k_1, k_2) \longrightarrow pc_2 := 1$
or $pc_2 = 0 \wedge \neg(j < \min(k_1, k_2)) \longrightarrow pc_2 := 4$
or $pc_2 = 1 \wedge a[j] > 0 \longrightarrow pc_2 := 2$
or $pc_2 = 1 \wedge \neg(a[j] > 0) \longrightarrow pc_2 := 3$
or $pc_2 = 2 \longrightarrow k_2 := j ; pc_2 := 0$
or $pc_2 = 3 \longrightarrow j := j + 2 ; pc_2 := 0$
od

Notar cómo se pierde la estructura del programa original. En (Apt, 86) se describe una transformación para un tipo de programa concurrente que conserva la estructura. El método reduccionista se complica cuando se aplica en programas concurrentes con instrucciones de sincronización, porque las transformaciones deben considerar también los casos de *deadlock*.

Aspectos semánticos y sintácticos del no determinismo y el *fairness*

La interpretación del no determinismo en los programas es *universal*, las propiedades deben cumplirse en todas las computaciones. Esta interpretación contrasta con la de otras áreas. Por ejemplo, en la teoría de la computación es *existencial*, para que una cadena de entrada sea aceptada por un autómata no determinístico alcanza con que una de sus computaciones a partir de dicha cadena termine en un estado final de aceptación. La interpretación existencial es útil para la definición y categorización de problemas computacionales, mientras que la universal apunta a la abstracción, componente muy relevante en el marco de la programación. Los siguientes son algunos hitos en la evolución de las dos interpretaciones mencionadas:

- La primera referencia al no determinismo en las ciencias de la computación aparece en (Rabin y Scott, 1959), donde se describen los *autómatas finitos no determinísticos*. No se establece ningún detalle de implementación.
- (Floyd, 1967) introduce los *algoritmos no determinísticos*, con el objeto de facilitar las descripciones algorítmicas. Al igual que en los autómatas no determinísticos, se plantea la posibilidad de seleccionar no determinísticamente entre varias alternativas, y la interpretación del no determinismo es *existencial*, distintas computaciones pueden producir distintos resultados. Pero ahora se define un modelo computacional concreto, el *backtracking*, para recorrer exhaustivamente todas las computaciones necesarias hasta llegar eventualmente a un resultado correcto.
- El no determinismo llega a la programación por E. Dijkstra, con el lenguaje secuencial no determinístico de comandos guardados GCL (Dijkstra, 1975). El no determinismo se plantea como la opción natural, siendo el determinismo un caso especial. Un poco después, C. Hoare incorpora esta misma idea en un lenguaje concurrente con comandos guardados e instrucciones de entrada/salida, el lenguaje CSP (Hoare, 1978). En ambos casos, para la implementación de los programas se deja un rango de opciones, pero todas las computaciones tienen que terminar correctamente, la interpretación del no determinismo es *universal*. Se prioriza la eficiencia, no se fuerza a que deban ejecutarse todas las computaciones.
- En los siguientes 20 años se multiplican los trabajos sobre la participación del no determinismo en el proceso de desarrollo de software. Una de las ideas prevalentes es la combinación del refinamiento de programas con el cálculo de la precondición más débil. Se utilizan lenguajes de programación con instrucciones no determinísticas no necesariamente ejecutables, para dar soporte a la idea de partir de un programa abstracto y refinarlo gradualmente hasta llegar a un programa concreto final.

Vimos que una forma particular de interpretar el no determinismo en los programas es con asunción de *fairness*. El *fairness* es importante sobre todo en el estudio de los programas concurrentes, porque determina un escenario en el que todos los componentes deben progresar aún con procesadores con velocidades arbitrariamente distintas. En este contexto se distinguen más variantes que el *fairness* débil y el *fairness* fuerte que consideramos en el capítulo. Entre otras: la variante *incondicional*, que no requiere condiciones para la ocurrencia de los eventos; la que impone en un conjunto de eventos habilitados igualdad de ocurrencias; la que prioriza las ocurrencias de los eventos de acuerdo al orden en que son habilitados; y la que limita el intervalo entre las ocurrencias de dos eventos consecutivos. Cualquiera sea el caso, el *fairness* es un concepto controversial en distintas disciplinas: en la semántica denotacional, por cómo impacta en el comportamiento de las funciones semánticas; en la verificación de programas, por requerir el uso de ordinales infinitos para probar no divergencia; y en los lenguajes de programación, por el no determinismo no acotado que genera.

En el plano sintáctico, ya hemos comentado la limitación de la lógica de Hoare para incluir asunciones de *fairness* en las fórmulas de correctitud. Una extensión de la lógica de predicados que permite expresar el *fairness* es el *lenguaje* L_μ, basado en un cálculo conocido como *μ-cálculo*. Con esta extensión se pueden derivar precondiciones más débiles en un escenario con *fairness*, utilizando constantes que denotan ordinales y variables que representan puntos fijos. Otra alternativa para tratar sintácticamente el *fairness* es la *lógica temporal*, que se interpreta directamente sobre computaciones con forma de secuencias o árboles. Todos estos aspectos del *fairness*, incluyendo su tratamiento en la verificación de programas secuenciales y concurrentes, se desarrollan ampliamente en (Francez, 1986).

Referencias bibliográficas

(Lauer, 1971) contiene la primera referencia a la verificación de programas no determinísticos. Presenta una regla de prueba para una instrucción de la forma S_1 *or* S_2, cuya semántica es la ejecución de S_1 o S_2. Esta misma idea se describe en (de Bakker, 1980).

En (Apt, 1984) se pasa revista a distintos resultados sobre la verificación de programas no determinísticos en la lógica de Hoare, incluyendo el uso de asignaciones aleatorias y *fairness*, y consideraciones en el marco de la metateoría.

El desarrollo sistemático de programas no determinísticos se inicia con (Dijkstra, 1976). La metodología se explica con mayor detalle en (Gries, 1981).

Para profundizar en la verificación de programas no determinísticos con asignaciones aleatorias y con *fairness* se pueden consultar, por ejemplo, en el primer caso (Apt y Plotkin, 1986), y en el segundo caso (Grumberg, Francez, Makowsky y de Roever, 1985) y (Apt y Olderog, 1983). Estas últimas constituyen las primeras publicaciones del método de las direcciones útiles y del método del planificador explícito, respectivamente.

En (Armoni y Ben-Ari, 2009) se proponen distintas alternativas para la enseñanza del no determinismo. La propuesta surge de un análisis de material pedagógico (textos y software) y currículas de ciencias de la computación a partir del cual los autores concluyen que el tratamiento del no determinismo es en general fragmentario y asistemático. El trabajo se complementa con una taxonomía de criterios para definir el no determinismo, y una revisión de la evolución de su uso.

El detalle de las referencias bibliográficas mencionadas, incluyendo las de las notas adicionales, es el siguiente:

Apt, K. (1984). *Ten years of Hoare's logic, a survey, part 2: nondeterminism.* Theoretical Computer Science, 28, 83-109.

Apt, K. (1986). *Correctness proofs of distributed termination algorithms*. ACM Trans. Prog. Lang. Syst., 8, 3, 388-405.

Apt, K. y Olderog, E. (1983). *Proof rules and transformations dealing with fairness*. Sci. Comput. Programming, 3, 65-100.

Apt, K. y Plotkin, G. (1986). *Countable nondeterminism and random assignment*. JACM, 33, 4, 724-767.

Armoni, M. y Ben-Hari, M. (2009). *The concept of nondeterminism: its development and implications for teaching.* ACM SIGCSE Bulletin, 41, 2, 141-160.

Ashcroft, E. y Manna, Z. (1971). *Formalization of properties of parallel programs*. Machine Intelligence, 6, 17-41.

Chandy, K. y Misra, J. (1988). *Parallel program design: a foundation*. Addison-Wesley.

de Bakker, J. (1980). *Mathematical theory of programm correctness*. Prentice Hall International.

Dijkstra, E. (1975). *Guarded command, nondeterminacy and formal derivation of programs*. Comm. ACM, 18, 8, 453-457.

Dijkstra, E. (1976). *A discipline of programming*. Prentice-Hall.

Floyd, R. (1967). *Nondeterministic algorithms.* JACM, 14, 4, 636-644.

Francez, N. (1986). *Fairness*. Springer-Verlag.

Gries, D. (1981). *The science of programming*. Springer-Verlag.

Grumberg, O., Francez, N., Makowsky, J. y de Roever, W. (1985). *A proof rule for fair termination of guarded commands*. Information and Control, 66, 83-102.

Hoare, C. (1978). *Communicating sequential processes*. Comm. ACM, 21, 8, 666-677.

Lauer, P. (1971). *Consistent formal theories of the semantics of programming languages*. Tech. Rep. 25, 121, IBM Laboratory, Viena.

Rabin, M. y Scott, D. (1959). *Finite automata and their decision problems*. IBM Journal of Research and Development, 3, 2, 114-125.

Ejercicios

1. Sean los programas:

   S_1 :: skip, S_2 :: if $b_1 \rightarrow$ skip or $b_2 \rightarrow$ skip fi, S_3 :: do $b_1 \rightarrow$ skip or $b_2 \rightarrow$ skip od

   Determinar si algún programa es equivalente semánticamente a otro.

Resolución

Ningún programa es equivalente a otro. En el caso del par S_1 y S_2, a partir de un estado con las variables b_1 y b_2 falsas S_1 termina y S_2 falla. En el caso del par S_1 y S_3, por ejemplo, a partir de un estado con b_1 verdadera S_1 termina y S_3 diverge. Y en el caso del par S_2 y S_3, por ejemplo, a partir de un estado con b_2 verdadera S_2 termina y S_3 diverge.

2. Determinar en cada caso si los programas son equivalentes semánticamente:
   2.1. S_1 :: if $1 \leq i \leq n$ then x := a[i] else skip fi, S_2 :: if $1 \leq i \leq n \rightarrow$ x := a[i] fi
   2.2. S_1 :: [x := 0 || x := 1 || x :=2], S_2 :: if true $\rightarrow$ x := 0 or true $\rightarrow$ x := 1 or true $\rightarrow$ x := 2 fi

3. ¿Cómo tienen que ser las instrucciones IF y DO para comportarse determinísticamente?

4. Dar un ejemplo de programa con comandos guardados que produzca al menos una computación que termina, una computación que falla y una computación que diverge.

5. ¿Cómo es el conjunto de estados finales de un programa con comandos guardados totalmente correcto, en cuanto a su tamaño y a los tipos de estados que contiene?

6. Determinar semánticamente si las fórmulas de correctitud parcial siguientes son verdaderas. En los casos que lo sean probarlas sintácticamente, es decir utilizando la axiomática descripta en el capítulo:
   6.1. $\{x \geq 0\}$ if $x \geq 0 \rightarrow$ x := 1 or $x \leq 0 \rightarrow$ skip $\{x = 1\}$
   6.2. $\{x \geq 0\}$ if $x \geq 0 \rightarrow$ x := 1 or $x \leq 0 \rightarrow$ x := 1 fi $\{x = 1\}$

7. Probar sintácticamente {p} do B $\rightarrow$ S od {q} sii {p} do B $\rightarrow$ S or B $\rightarrow$ S od {q}, siendo S algún programa con comandos guardados.

Resolución

Si se prueba la fórmula de correctitud {p} do B $\rightarrow$ S od {q}, se hace aplicando la regla NREP, utilizando un invariante r que cumple p $\rightarrow$ r, {r $\wedge$ B} S {r} y (r $\wedge$ ¬B) $\rightarrow$ q. Con estas premisas, aplicando la regla NREP también se prueba la fórmula {p} do B $\rightarrow$ S or B $\rightarrow$ S od {q}. De la misma manera se demuestra la implicación recíproca.

8. Probar sintácticamente la no divergencia, a partir de la precondición $N \geq 0$, del siguiente programa, presentado en el capítulo, que devuelve un número entre 0 y N:

   S_{sel} :: x := 0 ; y := 0 ; do y < N $\rightarrow$ y := y + 1 ; if true $\rightarrow$ x := y or true $\rightarrow$ skip fi od

Ayuda

Hay que recurrir a la regla NREP*. Se puede intentar con el invariante p = ($N \geq y$) y con el variante t = N – y.

9. Formular una regla para probar la no divergencia de una instrucción DO en al menos una de sus computaciones.

10. Explicar cómo tiene que ser el predicado p para que se cumpla la fórmula {p} x := ? {p}.

11. Formular un axioma para la asignación aleatoria en su variante $x := ? \preceq y$.

12. Probar sintácticamente la no divergencia, a partir de la precondición $y \geq 0$, del siguiente programa con una asignación aleatoria:

    S_{aa} :: do $x = 0 \wedge y > 0 \rightarrow y := ? ; x := 1$ or $x \neq 0 \wedge y > 0 \rightarrow y := y - 1$ od

Ayuda

Hay que recurrir a la regla NREPG*. El predicado $y \geq 0$ es un invariante del DO. Por otro lado, notar que si $x \neq 0$, el DO termina luego de *y* iteraciones, mientras que si $x = 0$, recién se puede saber en cuántas iteraciones termina después de la asignación aleatoria *y := ?*.

13. Probar, en el marco del lenguaje de programación con comandos guardados, que si su semántica incluye *fairness* fuerte entonces también incluye *fairness* débil.

Resolución

El *fairness* fuerte establece que las direcciones de una instrucción DO que están habilitadas infinitas veces en una computación infinita deben tomarse alguna vez. En particular, las direcciones pueden estar habilitadas permanentemente, que es lo que requiere el *fairness* débil.

14. Asumiendo *fairness* fuerte, ¿por qué en una computación infinita de un DO, una dirección habilitada infinitas veces se toma infinitas veces?

15. Asumiendo *fairness*, ¿cómo es el conjunto de estados finales de un programa con comandos guardados totalmente correcto, en cuanto a su tamaño y a los tipos de estados que contiene?

16. Determinar semánticamente en cada caso, si a partir de la precondición *true* el programa no diverge, considerando ausencia de *fairness*, *fairness* débil y *fairness* fuerte:
    16.1. S_1 :: do $x = 0 \rightarrow y := y + 1$ or $x = 0 \rightarrow x := 1$ or $x > 0 \wedge y > 0 \rightarrow y := y - 1$ od
    16.2. S_2 :: do $b \rightarrow x := x + 1$ or $b \rightarrow x := 0$ or $b \wedge x = 1 \rightarrow b :=$ false od
    16.3. S_3 :: do $b \rightarrow x := x + 1$ or $b \rightarrow x := 0$ or $b \wedge x = 2 \rightarrow b :=$ false od

Resolución del inciso 16.1

Si al comienzo $x \neq 0$, aun sin *fairness* el programa no diverge porque siempre toma la tercera dirección, que es la que lleva a la terminación. Si en cambio al comienzo $x = 0$, sin *fairness*

el programa puede divergir porque puede tomar siempre la primera dirección, pero con *fairness* débil (y por lo tanto también con *fairness* fuerte) no diverge, porque debe tomar alguna vez la segunda dirección, que lo lleva enseguida a la tercera dirección.

17. Probar sintácticamente la no divergencia, asumiendo *fairness* débil, del programa analizado en el ejercicio anterior (inciso 16.1), a partir de la precondición $x = 0 \wedge y = 0$.

Ayuda

Hay que recurrir al método de las direcciones útiles. El predicado $x \geq 0$ es un invariante del DO. Por otro lado, notar que cuando x = 0, la dirección que acerca a la terminación es la segunda, y cuando x = 1, la única dirección habilitada es la tercera, que se toma hasta el final. Tener en cuenta para la definición del variante del DO que el valor de *y* no debería afectarlo cuando se toma la primera dirección pero sí cuando se toma la tercera.

18. En el capítulo se ejemplifica la transformación propuesta por el método del planificador explícito para probar no divergencia asumiendo *fairness* fuerte (Ejemplo 13). Se utiliza el esquema de programa S_{pla} *:: do 1:* $B_1 \rightarrow S_1$ *or 2:* $B_2 \rightarrow S_2$ *fi*, y para eliminar sus computaciones infinitas *unfair* se le agregan asignaciones aleatorias de la siguiente manera:

S'_{pla} :: z_1 := ? ; z_2 := ? ;
do 1: $B_1 \wedge z_1 \leq z_2 \rightarrow S_1$; z_1 := ? ; if $B_2 \rightarrow z_2 := z_2 - 1$ or $\neg B_2 \rightarrow$ skip fi
or 2: $B_2 \wedge z_2 < z_1 \rightarrow S_2$; z_2 := ? ; if $B_1 \rightarrow z_1 := z_1 - 1$ or $\neg B_1 \rightarrow$ skip fi
od

Responder:

18.1. ¿Por qué el decremento gradual de la variable z_i (i = 1 ó 2) asegura que la dirección i se va a tomar alguna vez?

18.2. Explicar por qué el valor $z_i + 1$ representa el número máximo de iteraciones que pueden ocurrir antes de que se tome la dirección i, contando sólo las veces que está habilitada.

18.3. Explicar por qué lo que se le agrega al programa no afecta el no determinismo de las computaciones *fair*.

CAPÍTULO 4
Sensatez y completitud de los métodos de verificación de programas

Introducción

Antes de embarcarnos en el último capítulo del libro en la verificación de programas concurrentes, en este capítulo tratamos de manera muy introductoria la *sensatez* y *completitud* de los métodos de verificación.

Demostramos que los métodos presentados hasta el momento, referidos a los programas secuenciales, cumplen las dos propiedades, es decir que las fórmulas de correctitud que prueban son verdaderas (los métodos son *sensatos*), y recíprocamente que pueden probar todas las fórmulas de correctitud verdaderas (los métodos son *completos*).

También describimos formalmente la semántica de los lenguajes de programación utilizados, necesidad insoslayable para demostrar las propiedades de sensatez y completitud, dado que las pruebas consisten en relacionar formalmente el aspecto sintáctico de la aplicación de los axiomas y reglas de los métodos con el aspecto semántico del comportamiento de las computaciones de los programas.

El estudio de elementos de la metateoría de la verificación de programas, es decir de características de los propios métodos de verificación, y de la semántica de lenguajes, aun resumidamente y circunscripto al paradigma de programación secuencial y a la interpretación estándar de los números enteros como lo hacemos en este capítulo, contribuye sobremanera al entendimiento general de la problemática de la correctitud de programas. Por eso le dedicamos un espacio en este libro, y nos parece adecuado hacerlo en esta instancia de su desarrollo.

Primero introducimos los principios de *inducción matemática* e *inducción estructural* en los que se basan las pruebas de sensatez y completitud. Después describimos la semántica formal de los lenguajes de programación. Utilizamos la *semántica operacional*, muy intuitiva y difundida. En la siguiente sección desarrollamos las pruebas, en mayor o menor grado de detalle según la axiomática considerada. Finalmente, como en los capítulos anteriores, cerramos con secciones de observaciones finales, notas adicionales (que incluyen una breve descripción de otra semántica de uso extendido, la *semántica denotacional*), referencias bibliográficas y ejercicios.

Inducción matemática y estructural

En los capítulos anteriores nos referimos en varias ocasiones a la inducción. Por ejemplo, al definir el invariante de una instrucción de repetición. Otro ejemplo corresponde a la especificación de la sintaxis de un lenguaje de programación. Los dos casos mencionados sirven para identificar las dos variantes inductivas con las que se prueban, respectivamente, las propiedades de sensatez y completitud de un método de verificación de programas: la inducción matemática y la inducción estructural, que pasamos a describir.

La inducción matemática se basa en la manera en que se construye el conjunto N de los números naturales, comenzando por el número 0 y sumando sucesivamente 1 (modelo estándar). Establece que para demostrar el cumplimiento de una propiedad P en todos los números naturales, hay que probar:

1. P(0), es decir que la propiedad se cumple para el número 0. Esta es la *base inductiva*.
2. P(k) implica P(k + 1) para todo $k \geq 0$, es decir que asumiendo que la propiedad se cumple para un número natural determinado, que es la *hipótesis inductiva*, la propiedad también se cumple para su sucesor. Este es el *paso inductivo*.

Claramente, probando (1) y (2) se demuestra que la propiedad P se cumple para todos los números naturales. Por ejemplo, por inducción matemática se puede probar que la suma de los primeros números naturales hasta n es igual a n **.** (n + 1) / 2, o expresado más formalmente:

$\sum_{i = 0,n} i = n \,.\, (n + 1) / 2$

La prueba es la siguiente:

1. Base inductiva: $\sum_{i = 0,0} i = 0 \,.\, (0 + 1) / 2$.
2. Paso inductivo: a partir de la hipótesis inductiva $\sum_{i = 0,k} i = k \,.\, (k + 1) / 2$, hay que llegar a:

   $\sum_{i = 0,k+1} i = (k + 1) \,.\, ((k + 1) + 1) / 2$, o lo que es lo mismo, $\sum_{i = 0,k+1} i = (k + 1) \,.\, (k + 2) / 2$.

   Como $\sum_{i = 0,k+1} i = \sum_{i = 0,k} i + (k + 1)$, usando la hipótesis inductiva se prueba lo requerido:

   $\sum_{i = 0,k+1} i = k \,.\, (k + 1) / 2 + (k + 1) = (k + 1) \,.\, (k / 2 + 1) = (k + 1) \,.\, (k + 2) / 2$.

A veces se necesita recurrir a una forma de inducción matemática con un paso inductivo más exigente (se la conoce como inducción matemática *fuerte*): en lugar de P(k), la hipótesis inductiva es $P(i) \wedge P(i + 1) \wedge P(i + 2) \wedge \ldots \wedge P(k – 1) \wedge P(k)$, para algún $i \geq 0$.

La segunda variante inductiva, la inducción estructural, es una generalización de la inducción matemática. En efecto, se puede aplicar a más estructuras que la estructura $(N_0, <)$ de los números naturales estándar (siempre estructuras bien fundadas (A, <), como las que utilizamos para definir los variantes de las instrucciones de repetición, con una relación *predecesor* < en el

conjunto A que no produce cadenas descendentes infinitas), y además permite más de un caso tanto en la base inductiva como en el paso inductivo. Establece que para demostrar el cumplimiento de una propiedad en todos los componentes de una estructura (por ejemplo, un programa, una expresión, una estructura de datos de tipo árbol, o el propio conjunto de los números naturales estándar), hay que probar:

1. Base inductiva: la propiedad se cumple en todos los componentes *atómicos* o *minimales* de la estructura (siguiendo con los ejemplos anteriores, las instrucciones atómicas del programa, las constantes y las variables de la expresión, los nodos que constituyen las hojas de la estructura de datos de tipo árbol, o el número natural 0).
2. Paso inductivo: si la propiedad se cumple en todos los componentes de una subestructura, entonces también se cumple en la subestructura (que en particular puede ser la estructura completa).

Dicho de otra manera, la propiedad debe valer en todos los componentes atómicos de la estructura, y debe ser preservada por todos los constructores de componentes más complejos. La inducción estructural también es muy útil para definir estructuras. Por ejemplo, por inducción estructural se puede definir de la siguiente manera el conjunto de expresiones aritméticas formadas por las constantes 0 y 1, la variable x, los operadores + y **.**, y los paréntesis:

1. Base inductiva: 0, 1 y x son expresiones aritméticas.
2. Paso inductivo: si e_1 y e_2 son expresiones aritméticas, también lo son $(e_1 + e_2)$ y $(e_1 \,.\, e_2)$.

Notar que hay tres casos en la base inductiva, referidos a los átomos de las expresiones, y dos casos en el paso inductivo, asociados a los operadores de suma y multiplicación. Continuando con este ejemplo, completamos la sección mostrando cómo probar por inducción estructural la siguiente propiedad de las expresiones aritméticas definidas: el número de constantes y variables de una expresión aritmética *e* supera en 1 al número de operadores de *e*. Identificando los dos valores con c(e) y o(e), respectivamente, entonces hay que demostrar:

$c(e) = 1 + o(e)$

La prueba es la siguiente:

1. Base inductiva: si e = 0, 1 o x, entonces c(e) = 1 y o(e) = 0, y por lo tanto c(e) = 1 + o(e).
2. Paso inductivo: dada la expresión aritmética $e = (e_1 + e_2)$, a partir de la hipótesis inductiva $c(e_i) = 1 + o(e_i)$ para i = 1, 2, hay que probar c(e) = 1 + o(e), y lo mismo para el caso de la expresión aritmética $e = (e_1 \,.\, e_2)$:
   Si $e = (e_1 + e_2)$, entonces $c(e) = c((e_1 + e_2)) = c(e_1) + c(e_2)$ por la definición de c
   $= (1 + o(e_1)) + (1 + o(e_2))$ por la hipótesis inductiva

= 1 + o((e1 + e2)) por la definición de *o*

= 1 + o(e) por la definición de *e.*

Si e = (e1 **.** e2) se llega de manera similar también a c(e) = 1 + o(e).

Semántica formal de los lenguajes de programación utilizados

Comenzamos por los programas determinísticos. En su condición de lenguaje de programación imperativo, de entrada/salida y secuencial determinístico, el lenguaje de los *while* que utilizamos en el capítulo 2 (de ahora en más el lenguaje L_w) se puede caracterizar semánticamente por medio de una *función semántica* que tiene la siguiente forma:

$M_w : L_w \longrightarrow (\Sigma \longrightarrow \Sigma)$

La función semántica M_w, aplicada sobre un programa S de L_w (objeto sintáctico) y un estado inicial σ del conjunto de estados Σ (objeto semántico), devuelve un estado final (otro objeto semántico), que puede ser un estado *propio* σ' o un estado *indefinido* ⊥ de Σ según S termine o diverja a partir de σ, respectivamente (en estos programas no consideramos fallas). Es decir:

$M_w(S)(\sigma) = \sigma'$, si S termina a partir de σ

$M_w(S)(\sigma) = \perp$, en caso contrario

El uso de ⊥ permite que la función semántica sea total.

En particular, para especificar la semántica de L_w *operacionalmente*, que es el enfoque que seguimos en esta sección, debemos describir M_w en términos de una máquina abstracta que ejecuta computaciones y transforma estados, o más precisamente, *configuraciones*. Una configuración $C_i = (S_i, \sigma_i)$ es un par formado por una *continuación sintáctica* S_i que representa lo que queda por ejecutarse del programa S considerado, y un *estado corriente* σ_i que incluye los contenidos actuales de las variables de S (en verdad existe un tercer componente, con variables de control, que en el nivel de abstracción asumido se ignora). En este contexto, una computación $\pi(S, \sigma_0)$ de un programa S a partir de una *configuración inicial* $C_0 = (S, \sigma_0)$ es una secuencia, finita o infinita, de configuraciones $C_0 \longrightarrow C_1 \longrightarrow \ldots$, vinculadas por una relación conocida como *relación de transición* ⟶. Una computación finita se denota con $C_0 \longrightarrow^* C_k$ (el símbolo $\longrightarrow^*$ significa cero o más pasos), y tiene como *configuración final* a $C_k = (E, \sigma_k)$, tal que E es la *continuación sintáctica vacía*, que cumple E ; S = S ; E = S para todo programa S. En este caso se usa la expresión $val(\pi(S, \sigma_0)) = \sigma_k$, y en el caso de una computación divergente, $val(\pi(S, \sigma_0)) = \perp$. En definitiva, la especificación de la semántica operacional del lenguaje L_w consiste en describir la relación de transición ⟶, la cual caracteriza a las computaciones de sus programas. La formulamos a continuación, considerando todas las formas posibles de programas, determinadas por las instrucciones *skip*, asignación, secuencia, selección condicional y repetición, es decir:

S :: skip | x := e | S_1 ; S_2 | if B then S_1 else S_2 fi | while B do S_1 od

con la variable x representando cualquier variable entera, y empleando inducción estructural:

1. (skip, σ) ⟶ (E, σ)
2. (x := e, σ) ⟶ (E, σ[x|e])
3. Si (S, σ) ⟶ (S', σ'), entonces para todo T, (S ; T, σ) ⟶ (S' ; T, σ')
4. Si σ(B) = verdadero, entonces (if B then S_1 else S_2 fi, σ) ⟶ (S_1, σ)
   Si σ(B) = falso, entonces (if B then S_1 else S_2 fi, σ) ⟶ (S_2, σ)
5. Si σ(B) = verdadero, entonces (while B do S od, σ) ⟶ (S ; while B do S od, σ)
   Si σ(B) = falso, entonces (while B do S od, σ) ⟶ (E, σ)

La expresión σ[x|e] abrevia σ[x|σ(e)], estado que coincide con σ salvo eventualmente en la variable x, que tiene el resultado de la evaluación de la expresión *e* en σ. A propósito, notar que σ(e), y también σ(B), son abusos de notación, porque σ no se aplica a expresiones sino a variables. Lo hacemos para simplificar las definiciones (en la nota adicional de este capítulo sobre la semántica denotacional especificamos formalmente la semántica de las expresiones *e* y B). También notar que la evaluación de B no modifica el estado corriente. Las variables son enteras, y se asume la interpretación estándar de los números enteros.

Queda así planteado una suerte de sistema de axiomas y reglas, correspondientes a las instrucciones atómicas y compuestas del lenguaje. Las únicas transiciones posibles son las que se deducen del sistema. Se aprecia claramente el determinismo del lenguaje, porque toda configuración tiene a lo sumo un sucesor, y también que no hay posibilidad de bloqueo en una computación, porque toda configuración, salvo una configuración final, tiene sucesor.

Por ejemplo, aplicando la definición de ⟶ a partir del estado inicial σ[x|0] sobre el programa:

S_{comp} :: a[0] := 1 ; a[1] := 0 ; while a[x] ≠ 0 do x := x + 1 od

se obtiene la siguiente computación:

(a[0] := 1 ; a[1] := 0 ; while a[x] ≠ 0 do x := x + 1 od, σ[x|0]) ⟶
(a[1] := 0 ; while a[x] ≠ 0 do x := x + 1 od, σ[x|0][a[0]|1]) ⟶
(while a[x] ≠ 0 do x := x + 1 od, σ[x|0][a[0]|1][a[1]|0]) ⟶
(x := x + 1 ; while a[x] ≠ 0 do x := x + 1 od, σ[x|0][a[0]|1][a[1]|0]) ⟶
(while a[x] ≠ 0 do x := x + 1 od, σ[a[0]|1][a[1]|0][x|1]) ⟶
(E, σ[a[0]|1][a[1]|0][x|1])

A partir de la definición inductiva de ⟶ se pueden obtener todas las formas posibles de las computaciones de los programas de L_w (necesarias para las pruebas de sensatez y completitud).

Dada una configuración inicial $C_0 = (S, \sigma_0)$, se cumple (la numeración se corresponde con la definición inductiva anterior):

1. Si S :: skip, entonces $\pi(S, \sigma_0)$ es $(skip, \sigma_0) \longrightarrow (E, \sigma_0)$.
2. Si S :: x := e, entonces $\pi(S, \sigma_0)$ es $(x := e, \sigma_0) \longrightarrow (E, \sigma_0[x|e])$.
3. Si S :: S_1 ; S_2, entonces $\pi(S, \sigma_0)$ puede ser:
   3.1. $(S_1 ; S_2, \sigma_0) \longrightarrow (T_1 ; S_2, \sigma_1) \longrightarrow (T_2 ; S_2, \sigma_2) \longrightarrow \ldots$, si S_1 no termina desde σ_0.
   3.2. $(S_1 ; S_2, \sigma_0) \longrightarrow^* (S_2, \sigma_1) \longrightarrow (T_1, \sigma_2) \longrightarrow (T_2, \sigma_3) \longrightarrow \ldots$, si S_1 termina desde σ_0 y S_2 no termina desde σ_1.
   3.3. $(S_1 ; S_2, \sigma_0) \longrightarrow^* (S_2, \sigma_1) \longrightarrow^* (E, \sigma_2)$, si S_1 y S_2 terminan desde σ_0 y σ_1, respectivamente.
4. Si S :: if B then S_1 else S_2 fi, entonces $\pi(S, \sigma_0)$ puede ser:
   4.1. (if B then S_1 else S_2 fi, σ_0) $\longrightarrow (S_1, \sigma_0) \longrightarrow (T_1, \sigma_1) \longrightarrow (T_2, \sigma_2) \longrightarrow \ldots$, si al comienzo vale B y S_1 no termina desde σ_0.
   4.2. (if B then S_1 else S_2 fi, σ_0) $\longrightarrow (S_1, \sigma_0) \longrightarrow^* (E, \sigma_1)$, si al comienzo vale B y S_1 termina desde σ_0.
   Siguen otras dos posibilidades, (4.3) y (4.4), que se definen como las anteriores pero considerando que al comienzo no vale B.
5. Si S :: while B do S od, entonces $\pi(S, \sigma_0)$ puede ser:
   5.1. (while B do S od, σ_0) $\longrightarrow^* (S ;$ while B do S od, $\sigma_1) \longrightarrow (T_1 ;$ while B do S od, $\sigma_2) \longrightarrow (T_2 ;$ while B do S od, $\sigma_3) \longrightarrow \ldots$, si al comienzo vale B y en alguna iteración S no termina desde σ_1.
   5.2. (while B do S od, σ_0) $\longrightarrow^*$ (while B do S od, σ_1) $\longrightarrow^*$ (while B do S od, σ_2) $\longrightarrow \ldots$, si toda vez que se evalúa, B es verdadera, y en todas las iteraciones S termina desde su estado inicial σ_i.
   5.3. (while B do S od, σ_0) $\longrightarrow$ (S ; while B do S od, σ_0) $\longrightarrow^* (E, \sigma_k)$, si al comienzo vale B, en todas las iteraciones S termina desde su estado inicial σ_i, y alguna vez que se evalúa, B es falsa.
   5.4. (while B do S od, σ_0) $\longrightarrow (E, \sigma_0)$, si al comienzo no vale B.

Seguimos ahora por los programas no determinísticos que utilizamos en el capítulo 3, con selecciones condicionales y repeticiones con comandos guardados y asignaciones aleatorias (identificamos al lenguaje de programación con L_{cg}). En este caso, la función semántica asociada al lenguaje tiene la siguiente forma:

$$M_{cg} : L_{cg} \longrightarrow (\Sigma \longrightarrow P(\Sigma))$$

El conjunto de partes $P(\Sigma)$ se explica porque a partir de un estado inicial, un programa del lenguaje L_{cg} produce un conjunto de estados finales, que en el caso más general incluye estados propios σ', el estado indefinido $\perp$ y el estado de falla f. En otras palabras, a partir de un estado inicial σ_0, un programa S de L_{cg} puede tener varias computaciones, $\pi_1(S, \sigma_0)$, $\pi_2(S, \sigma_0)$, …,

algunas de las cuales pueden terminar, otras divergir y otras fallar. $\Pi(S, \sigma_0)$ denota el árbol de computaciones que parten de la configuración inicial (S, σ_0). Considerando todas las formas posibles de programas de L_{cg}, determinadas por las instrucciones *skip*, asignación, secuencia, selección condicional no determinística (o IF), repetición no determinística (o DO) y asignación aleatoria, es decir:

$S :: skip \mid x := e \mid S_1 ; S_2 \mid \text{if } B_1 \longrightarrow S_1 \text{ or } \ldots \text{ or } B_n \longrightarrow S_n \text{ fi} \mid \text{do } B_1 \longrightarrow S_1 \text{ or } \ldots \text{ or } B_n \longrightarrow S_n \text{ od} \mid x := ?$

sin DO anidados y tal que x representa cualquier variable entera, la relación de transición $\longrightarrow$ que especifica la semántica operacional de L_{cg} se define inductivamente de la siguiente manera:

1. $(skip, \sigma) \longrightarrow (E, \sigma)$
2. $(x := e, \sigma) \longrightarrow (E, \sigma[x|e])$
3. Si $(S, \sigma) \longrightarrow (S', \sigma')$, entonces para todo T, $(S ; T, \sigma) \longrightarrow (S' ; T, \sigma')$
4. Si i pertenece al conjunto de direcciones {1, …, n} y $\sigma(B_i)$ = verdadero, entonces
   $(\text{if } B_1 \longrightarrow S_1 \text{ or } \ldots \text{ or } B_n \longrightarrow S_n \text{ fi}, \sigma) \longrightarrow (S_i, \sigma)$
   Si i pertenece al conjunto de direcciones {1, …, n} y para todo i, $\sigma(B_i)$ = falso, entonces
   $(\text{if } B_1 \longrightarrow S_1 \text{ or } \ldots \text{ or } B_n \longrightarrow S_n \text{ fi}, \sigma) \longrightarrow (E, f)$
5. Si i pertenece al conjunto de direcciones {1, …, n} y $\sigma(B_i)$ = verdadero, entonces
   $(\text{do } B_1 \longrightarrow S_1 \text{ or } \ldots \text{ or } B_n \longrightarrow S_n \text{ od}, \sigma) \longrightarrow (S_i ; \text{do } B_1 \longrightarrow S_1 \text{ or } \ldots \text{ or } B_n \longrightarrow S_n \text{ od}, \sigma)$
   Si i pertenece al conjunto de direcciones {1, …, n} y para todo i, $\sigma(B_i)$ = falso, entonces
   $(\text{do } B_1 \longrightarrow S_1 \text{ or } \ldots \text{ or } B_n \longrightarrow S_n \text{ od}, \sigma) \longrightarrow (E, \sigma)$
6. $(x := ?, \sigma) \longrightarrow (E, \sigma[x|n]>$, con $n \in N$

Las variables son enteras y se asume la interpretación estándar de los números enteros. El no determinismo del lenguaje se aprecia en las últimas tres instrucciones: en las instrucciones IF y DO, la elección de una dirección habilitada del conjunto {1, …, n} es totalmente arbitraria, y lo mismo sucede en la elección de un número natural en una asignación aleatoria. No hay posibilidad de bloqueo en una computación, pero sí de falla si una instrucción IF no tiene guardias verdaderas.

Al igual que en la especificación de la semántica del lenguaje determinístico, de la definición inductiva de la relación de transición $\longrightarrow$ correspondiente al lenguaje L_{cg} se pueden derivar las distintas formas de las computaciones de los programas. Tres diferencias con respecto a la definición anterior son: a partir de una configuración inicial, ahora se debe considerar un árbol de computaciones; una computación puede terminar en un estado de falla; y si la semántica incluye alguna hipótesis de *fairness*, determinadas computaciones infinitas no pueden existir (computaciones *unfair*). Con respecto a este último punto, utilizando la semántica operacional sólo podemos quedarnos con la descripción informal de *fairness* que planteamos en el capítulo 3 (restricción global sobre el entorno de ejecución, que se traduce en la poda de determinadas

computaciones de los árboles de computaciones de los programas), debido a su limitación para expresar formalmente todas y sólo las computaciones *fair*.

Pruebas de sensatez y completitud

En esta sección desarrollamos las pruebas de sensatez y completitud de los métodos, teniendo en cuenta las semánticas recién especificadas. Inicialmente consideramos las axiomáticas para la correctitud parcial de los programas, y luego completamos tratando las reglas de prueba de no divergencia y ausencia de falla.

Sensatez de las axiomáticas para las pruebas de correctitud parcial

Para demostrar la sensatez de una axiomática para verificar la correctitud parcial de una clase de programas, hay que probar que para todo programa S de la clase y toda especificación (p, q) se cumple:

$$\vdash \{p\}\ S\ \{q\} \rightarrow \models \{p\}\ S\ \{q\}$$

es decir que toda fórmula de correctitud parcial {p} S {q} derivada de la axiomática (sintácticamente) es verdadera (semánticamente), lo que significa, de acuerdo a la definición de correctitud parcial:

$$\forall \sigma: ((\sigma \models p \wedge M(S)(\sigma) \neq \perp) \longrightarrow M(S)(\sigma) \models q)$$

La demostración se basa en el siguiente razonamiento: si los axiomas son verdaderos y las reglas son sensatas (a partir de premisas verdaderas obtienen conclusiones verdaderas, o en otras palabras, preservan la verdad), entonces todas las fórmulas derivadas de la axiomática son verdaderas. De esta manera, para la demostración se puede aplicar inducción matemática (fuerte) sobre la longitud de la prueba de una fórmula, teniendo en cuenta que las pruebas de los axiomas miden 1 (son de un solo paso).

Empezamos por demostrar la sensatez de la axiomática que describimos para los programas determinísticos del lenguaje L_w:

1. Base inductiva (prueba de que los axiomas son verdaderos).

   Axioma del skip (SKIP).
   Hay que probar $\models \{p\}$ skip $\{p\}$.

Sea σ un estado inicial tal que σ |= p. Como (skip, σ) ⟶ (E, σ), entonces el estado final σ también cumple σ |= p.

Axioma de la asignación (ASI).
Hay que probar |= {p[x|e]} x := e {p}.
La prueba sigue el mismo esquema que la anterior.

2. Paso inductivo (prueba de que las reglas son sensatas).

Regla de la secuencia (SEC).
Asumiendo |– {p} S_1 ; S_2 {q}, hay que probar |= {p} S_1 ; S_2 {q}.
Por SEC, {p} S_1 ; S_2 {q} se prueba de {p} S_1 {r} y {r} S_2 {q}, para algún r.
Hipótesis inductiva: |= {p} S_1 {r} y |= {r} S_2 {q} (porque las pruebas de {p} S_1 {r} y {r} S_2 {q} son más cortas que la prueba de {p} S_1 ; S_2 {q}).
Sea σ_0 un estado tal que σ_0 |= p y val(π(S_1 ; S_2, σ_0)) = $\sigma_2 \neq \perp$ (en éste y el resto de los casos ignoramos las computaciones infinitas, porque en ellas la correctitud parcial se cumple trivialmente). Por lo tanto, por la semántica del lenguaje y la hipótesis inductiva se cumple: (S_1 ; S_2, σ_0) ⟶* (S_2, σ_1) ⟶* (E, σ_2), con σ_2 |= q, lo que completa la prueba.

Regla del condicional (COND).
Asumiendo |– {p} if B then S_1 else S_2 fi {q}, hay que probar |= {p} if B then S_1 else S_2 fi {q}.
Por COND, {p} if B then S_1 else S_2 fi {q} se prueba de {p ∧ B} S_1 {q} y {p ∧ ¬B} S_2 {q}.
La prueba se completa siguiendo el mismo esquema que la anterior.

Regla de la repetición (REP).
Si |– {p} while B do S od {p ∧ ¬B}, hay que probar |= {p} while B do S od {p ∧ ¬B}.
Por REP, {p} while B do S od {p ∧ ¬B} se prueba de {p ∧ B} S {p}.
También esta prueba se completa siguiendo el mismo esquema que las anteriores, pero igual la desarrollamos por la forma un tanto particular de la computación que se plantea.
Hipótesis inductiva: |= {p ∧ B} S {p}.
Sea σ_0 un estado tal que σ_0 |= p y val(π(while B do S od, σ_0)) = $\sigma_n \neq \perp$. Por lo tanto, por la semántica del lenguaje y la hipótesis inductiva, la computación del *while* tiene la siguiente forma: C_0 ⟶* … ⟶* C_{n-1} ⟶* C_n, con C_i = (while B do S od, σ_i) y σ_i |= p para i = 0, .., n – 1, y con C_i = (E, σ_i) y σ_i |= (p ∧ ¬B) para i = n, lo que completa la prueba.

Finalmente, la sensatez de la *regla de consecuencia* (CONS) y la *regla de instanciación* (INST) se verifica directamente a partir de conceptos semánticos, dada su naturaleza justamente semántica. En el caso de la regla CONS, según la cual {p} S {q} se prueba de {p_1} S {q_1} y dos enunciados verdaderos p → p_1 y q_1 → q de los números enteros, de la hipótesis inductiva |= {p_1} S {q_1} se llega a |= {p} S {q} por la definición de estado. Y en el

caso de INST, que establece que una fórmula de correctitud instanciada f(c) se prueba de una fórmula de correctitud genérica f(X), siendo c un elemento del dominio semántico en que está definida la variable de especificación X (los números enteros), el cumplimiento de |= f(c) se debe a la definición de variable de especificación.

En lo que hace a la demostración de la sensatez de la axiomática que describimos para los programas no determinísticos del lenguaje L_{cg}, con lo realizado previamente sólo falta probar la sensatez de las reglas de la *selección condicional no determinística* (NCOND), la *repetición no determinística* (NREP) y la *asignación aleatoria* (NASI), es decir las reglas correspondientes a las instrucciones no determinísticas. En los tres casos las pruebas son generalizaciones de las que desarrollamos para las reglas de las instrucciones determinísticas, en las que las hipótesis inductivas se aplican sobre árboles de computaciones.

Completitud de las axiomáticas para las pruebas de correctitud parcial

En esta segunda parte de la sección demostramos la completitud de las axiomáticas referidas anteriormente. Una axiomática para verificar la correctitud parcial de una clase de programas es completa si para todo programa S de la clase y toda especificación (p, q) cumple:

$$\models \{p\}\ S\ \{q\} \rightarrow \vdash \{p\}\ S\ \{q\}$$

es decir, si tiene la propiedad de probar todas las fórmulas de correctitud parcial {p} S {q} verdaderas (es la propiedad recíproca de la sensatez). Antes de desarrollar la demostración, cabe señalar que la completitud de las axiomáticas es *relativa*, por dos motivos:

- Un motivo es que las axiomáticas deben incluir necesariamente todos los axiomas de los números enteros, sin los cuales poco podrían probar (por ejemplo, para probar la simple fórmula {true} skip {p} hay que probar la implicación true ⟶ p, es decir directamente que p es un enunciado verdadero de los enteros). Y no sirve agregar a las axiomáticas axiomas y reglas para probar todos los enunciados verdaderos de los enteros, porque por el *teorema de incompletitud de Gödel* esto es imposible (existen enunciados verdaderos que no pueden probarse).
- El segundo motivo es para señalar que la propiedad de completitud se cumple cuando el lenguaje de especificación es *expresivo* con respecto al lenguaje de programación y la interpretación que se utilizan. Precisando para nuestro caso, las axiomáticas son completas porque el lenguaje de la lógica de predicados, en el marco de los lenguajes L_w y L_{cg} y los números enteros estándar, permite expresar con un predicado, para toda precondición p y todo programa S, el conjunto de todos los estados obtenidos por S a partir de p (que se suele identificar con post(p, S)). En una nota adicional profundizamos sobre este aspecto.

Para demostrar la completitud de las axiomáticas aplicamos inducción estructural, teniendo en cuenta todas las formas posibles de los programas y en base al siguiente razonamiento: asumiendo que la conclusión de una regla de prueba es verdadera, demostramos que las premisas son verdaderas de acuerdo a la semántica definida, y luego, por la hipótesis inductiva, que las mismas se pueden probar (decimos en este caso que la regla es completa). Comenzamos con la axiomática que describimos para los programas determinísticos:

1. Base inductiva (considera los programas minimales, es decir las instrucciones atómicas).

   S :: skip
   Asumiendo |= {p} skip {q}, hay que probar |– {p} skip {q}.
   Como (skip, σ) $\rightarrow$ (E, σ), si σ |= p entonces debe cumplirse σ |= q, y por lo tanto p $\rightarrow$ q.
   La prueba de {p} skip {q} en la axiomática es la siguiente:
   (1) {q} skip {q} (SKIP), (2) p $\rightarrow$ q (MAT), (3) {p} skip {q} (CONS, 1, 2).

   S :: x := e
   Asumiendo |= {p} x := e {q}, hay que probar |– {p} x:= e {q}.
   La prueba sigue el mismo esquema que la anterior.

2. Paso inductivo (considera el resto de los programas).

   S :: S_1 ; S_2
   Asumiendo |= {p} S_1 ; S_2 {q}, hay que probar |– {p} S_1 ; S_2 {q}.
   Por la semántica de la secuencia, se cumple |= {p} S_1 {r} y |= {r} S_2 {q}, siendo r algún predicado que expresa el conjunto post(p, S_1).
   Hipótesis inductiva: |– {p} S_1 {r} y |– {r} S_2 {q} (porque los programas S_1 y S_2 son más simples que S_1 ; S_2).
   Aplicando la regla SEC se obtiene {p} S_1 ; S_2 {q}.

   S :: if B then S_1 else S_2 fi
   Asumiendo |= {p} if B then S_1 else S_2 fi {q}, hay que probar |– {p} if B then S_1 else S_2 fi {q}.
   Por la semántica de la selección condicional, se cumple |= {p $\wedge$ B} S_1 {q} y |= {p $\wedge$ ¬B} S_2 {q}.
   La prueba se completa siguiendo el mismo esquema que la anterior.

   S :: while B do S od
   Asumiendo |= {r} while B do S od {q}, hay que probar |– {r} while B do S od {q}.
   Por la semántica del *while*, se cumple |= {p $\wedge$ B} S {p}, r $\rightarrow$ p y (p $\wedge$ ¬B) $\rightarrow$ q, siendo p un invariante del mismo (la prueba de su expresividad la vemos enseguida).
   Hipótesis inductiva: |– {p $\wedge$ B} S {p}, r $\rightarrow$ p y (p $\wedge$ ¬B) $\rightarrow$ q.
   Aplicando las reglas REP y CONS se obtiene {r} while B do S od {q}.

Veamos cómo se puede expresar el invariante p. Desde el punto de vista semántico, debe representar a todos los estados alcanzables por el *while* desde un estado que satisfaga r, es decir $\{\sigma \mid \exists k, \sigma_0, \ldots, \sigma_k: k \in N \wedge \sigma = \sigma_k \wedge \sigma_0 \models r \wedge (\forall i < k: M(S)(\sigma_i) = \sigma_{i+1} \wedge \sigma_i(B) = \text{verdadero})\}$. Por lo tanto, sintácticamente su forma podría plantearse como la disyunción, posiblemente infinita, $p_0 \vee p_1 \vee \ldots$, siendo $p_0 = r$ y p_{i+1} un predicado que denota el conjunto post($p_i \wedge B$, S), con $i \geq 0$. Pero si bien los p_i son expresables, una disyunción infinita no es un predicado válido. En lo que sigue planteamos una expresión finita para el invariante p, muy intuitiva:
Sean: el conjunto $\{y_1, \ldots, y_n\}$ formado por las variables de S más las variables libres de r y q; un conjunto de nuevas variables $\{z_1, \ldots, z_n\}$ de igual tamaño que el conjunto de las y_i; el *while* S^* *:: while* $B \wedge (y_1 \neq z_1 \vee \ldots \vee y_n \neq z_n)$ *do S od*, derivado del original *while B do S od*; y un predicado p^* que denota el conjunto post(r, S^*). Notar que el nuevo *while* S^* se comporta como el original, salvo que eligiendo de determinada manera las variables z_i puede terminar sin que deje de valer la condición original B. De esta manera, se cumple que $p = \exists z_1 \ldots z_n$: p^* es un invariante del *while* original, considerando $\models \{r\}$ while B do S od $\{q\}$:
(a) $r \rightarrow p$, eligiendo $z_i = y_i$ para todo i (S^* termina inmediatamente).
(b) $(p \wedge \neg B) \rightarrow q$, porque si vale $p^* \wedge \neg B$, significa que las variables y_i tienen los valores finales del *while* original, y como se cumple $\models \{r\}$ while B do S od $\{q\}$, entonces vale q.
(c) $\models \{p \wedge B\}\ S\ \{p\}$, porque si $\sigma \models (p^* \wedge B)$, los valores $\sigma(y_i)$ se obtuvieron al ejecutar S una cantidad finita de pasos, quedando $\sigma(B)$ = verdadero, y así, eligiendo $z_i = M(S)(\sigma(y_i))$ para todo i, vale $M(S)(\sigma) \models p$.

Con las pruebas previas, para demostrar la completitud de la axiomática que describimos para los programas no determinísticos sólo faltan analizar las reglas NCOND, NREP y NASI. Igual que lo observado con la sensatez, las pruebas que consideran estas tres reglas son generalizaciones de las correspondientes a las instrucciones determinísticas. La única diferencia para destacar aparece en el tratamiento de la regla NREP. Ahora el invariante de una repetición debe representar al conjunto de todos los estados alcanzables por la ejecución iterativa de distintos subprogramas S_i, conjunto sintácticamente definible por medio de la disyunción posiblemente infinita $p_0 \vee p_1 \vee \ldots$, tal que p_0 es la precondición y p_{i+1} es un predicado que denota la unión de los conjuntos post($p_i \wedge B_j$, S_j), con j variando en el conjunto de direcciones de la repetición. Dicha disyunción, no permitida por el lenguaje de especificación, se puede expresar con un predicado válido, recurriendo a la misma idea desarrollada para los programas determinísticos.

Sensatez de las reglas de prueba de no divergencia

Continuamos la sección con las pruebas de sensatez de las reglas de prueba de no divergencia en esta parte, y las pruebas de completitud de las mismas en la siguiente.

La *regla de la repetición* (REP*) para los *while* de los programas determinísticos establece que |– ⟨p⟩ while B do S od ⟨p ∧ ¬B⟩ se deriva de |– ⟨p ∧ B⟩ S ⟨p⟩, |– ⟨p ∧ B ∧ t = Z⟩ S ⟨t < Z⟩ y la implicación p ⟶ t ≥ 0, siendo p y t el invariante y el variante definidos para el *while*, respectivamente, con t variando en el conjunto bien fundado (N_0, <). Partiendo de la hipótesis inductiva |= ⟨p ∧ B⟩ S ⟨p⟩, |= ⟨p ∧ B ∧ t = Z⟩ S ⟨t < Z⟩ y p ⟶ t ≥ 0, supongamos que a partir de la precondición p el *while* diverge. Veamos que llegamos a una contradicción. Sea σ_0 |= p y la computación π(while B do S od, σ_0) una computación infinita de la forma:

(while B do S od, σ_0) ⟶* … ⟶* (while B do S od, σ_i) ⟶* …

con σ_i |= p ∧ B ∧ t ≥ 0, para i ≥ 0, de acuerdo a la hipótesis. En consecuencia, la cadena:

$\sigma_0(t)$, …, $\sigma_i(t)$, …

también es infinita. Y como para todo i se cumple que $\sigma_i(t) > \sigma_{i+1}(t)$, esto quiere decir que existe una cadena descendente infinita en (N_0, <), lo que es imposible. Así, el *while* debe terminar, y en un estado σ_k tal que σ_k |= p ∧ ¬B, Por lo tanto, se cumple |= ⟨p⟩ while B do S od ⟨p ∧ ¬B⟩.

En el marco de los programas no determinísticos hay que considerar distintas reglas de prueba de no divergencia.

La prueba de la sensatez de la *regla de la no divergencia no determinística* (NREP*) sigue el mismo esquema que la anterior. Se llega al absurdo de encontrar una cadena descendente infinita en (N_0, <), ahora teniendo en cuenta que el variante se puede decrementar a lo largo de la ejecución sucesiva de distintos subprogramas S_i.

Lo mismo sucede con la prueba de la sensatez de la *regla de la no divergencia no determinística generalizada* (NREPG*), utilizada cuando hay asignaciones aleatorias, en la que se llega a una cadena descendente infinita en un conjunto bien fundado W que puede incluir ordinales infinitos.

Y también se puede replicar el mismo esquema de prueba para la *regla de la no divergencia no determinística asumiendo fairness débil* (NREPD*), salvo que en este caso el variante t de la instrucción DO puede mantener un mismo valor a lo largo de una cantidad finita de iteraciones de distintos subprogramas S_i. La regla se basa en una partición de las direcciones del DO por cada valor w no minimal de t, integrada por un conjunto D_w no vacío de direcciones útiles que decrementan t y un conjunto D'_w con direcciones que no lo incrementan. Es decir, dados t y un invariante p del DO, la regla requiere, para todo valor w no minimal de t, que para toda dirección i del conjunto D_w se cumpla ⟨p ∧ B_i ∧ t = w⟩ S_i ⟨t < w⟩, y que para toda dirección i del conjunto D'_w se cumpla ⟨p ∧ B_i ∧ t = w⟩ S_i ⟨t ≤ w⟩. Con esta hipótesis inductiva, asumiendo *fairness* débil no puede haber un DO que diverge. Suponiendo lo contrario, una computación infinita del DO que tuviera la forma (DO, σ_0) ⟶* … ⟶* (DO, σ_i) ⟶* … produciría una cadena infinita $\sigma_0(t)$, …, $\sigma_i(t)$,

…, lo cual es imposible: como t varía en un conjunto bien fundado, la única posibilidad es que la cadena contenga infinitos valores iguales, contradiciendo la hipótesis de *fairness*.

Finalmente, con respecto a la *regla de la no divergencia no determinística asumiendo fairness fuerte* (NREPF*), como se basa en una transformación de programa que elimina las computaciones *unfair* de los DO mediante asignaciones aleatorias, su prueba de sensatez es directamente la de la regla NREPG*, que ya comentamos.

Completitud de las reglas de prueba de no divergencia

La prueba de la completitud de la regla REP* para los *while* de los programas determinísticos sigue el mismo razonamiento inductivo que la de la regla REP, incluyendo ahora las premisas que se refieren al variante. Asumiendo $\models \langle p \rangle$ while B do S od $\langle p \wedge \neg B \rangle$, siendo p un invariante del *while*, por la semántica del lenguaje se cumple $\models \langle p \wedge B \rangle$ S $\langle p \rangle$, $\models \langle p \wedge B \wedge t = Z \rangle$ S $\langle t < Z \rangle$ y la implicación $p \longrightarrow t \geq 0$, tal que t es un variante del *while*, función entera definida en $(N_0, <)$ en términos de las variables de programa, y Z es una variable de especificación. De esta manera, a partir de la hipótesis inductiva de que $\langle p \wedge B \rangle$ S $\langle p \rangle$, $\langle p \wedge B \wedge t = Z \rangle$ S $\langle t < Z \rangle$ y $p \longrightarrow t \geq 0$ se pueden probar por la axiomática, llegamos a $\vdash \langle p \rangle$ while B do S od $\langle p \wedge \neg B \rangle$. La expresividad del lenguaje de especificación con respecto a los invariantes ya la tratamos en la prueba de completitud de la regla REP, partiendo de su capacidad para expresar los conjuntos de estados post(p, S) para todo p y todo S. Y con respecto a los variantes, el lenguaje de especificación debe tener adicionalmente la capacidad para expresar con una función entera la cantidad de iteraciones de todo *while* que termina, y así debe poder especificar todas las funciones computables, lo que justificamos a continuación. Sea el programa *S :: while B do* S_1 *od*, una variable entera x que no ocurre en S, y el programa siguiente, ampliado a partir de S:

S_x :: x := 0 ; while B do x := x + 1 ; S_1 od

Si S termina a partir de un estado σ en un estado σ´, identificando a σ´(x) con iter(S, σ) se puede entender a iter(S, σ) como una función parcial en σ, computable como lo muestra el programa S_x, definida cuando S termina desde σ y que calcula el número de iteraciones del *while*. En estos términos, la expresividad de los variantes significa entonces que para todo *while* S que termina desde un estado σ, existe una función entera t que cumple σ(t) = iter(S, σ).

La prueba de completitud de la regla NREP* para las instrucciones DO de los programas no determinísticos es una generalización de la de la regla REP para los *while*. Por su parte, el tratamiento de la completitud de las reglas de prueba de no divergencia relacionadas con las asignaciones aleatorias y el *fairness* requiere otro formalismo.

Sensatez y completitud de la regla de prueba de ausencia de falla

Finalizamos la sección con las pruebas relacionadas con la última propiedad de programa a considerar, la ausencia de falla. La única regla para analizar es la *regla del condicional no determinístico sin falla* (NCOND*), asociada a la única instrucción que puede provocar una falla en un programa, la selección condicional no determinística *if* $B_1 \longrightarrow S_1$ *or … or* $B_n \longrightarrow S_n$ *fi*.

Asumiendo $\vdash \langle p \rangle$ if $B_1 \rightarrow S_1$ or…or $B_n \rightarrow S_n$ fi $\langle q \rangle$, de acuerdo a la regla NCOND* se cumple la premisa $p \rightarrow \bigvee_{i=1,n} B_i$, y por lo tanto se llega a $\models \langle p \rangle$ if $B_1 \rightarrow S_1$ or…or $B_n \rightarrow S_n$ fi $\langle q \rangle$. En efecto, el IF no produce falla porque la hipótesis inductiva $p \rightarrow \bigvee_{i=1,n} B_i$ asegura que al menos una de sus guardias B_i es verdadera. De esta manera, la regla NCOND* es sensata.

Recíprocamente, si $\models \langle p \rangle$ if $B_1 \rightarrow S_1$ or…or $B_n \rightarrow S_n$ fi $\langle q \rangle$, es decir si el IF no produce falla, es porque de acuerdo a la semántica del lenguaje alguna de sus guardias B_i es verdadera cuando se evalúa en un estado que satisface p, lo que se expresa con la implicación $p \rightarrow \bigvee_{i=1,n} B_i$. Con esta hipótesis inductiva, aplicando NCOND* se llega a $\vdash \langle p \rangle$ if $B_1 \rightarrow S_1$ or…or $B_n \rightarrow S_n$ fi $\langle q \rangle$, y así la regla NCOND* es completa.

Observaciones finales

Contar con métodos de verificación de programas sensatos y completos es un escenario muy deseable. La sensatez es una propiedad imprescindible, un método no sensato no tiene razón de ser. La completitud depende de la expresividad del lenguaje de especificación con respecto a la clase de programas utilizados y la interpretación semántica considerada.

El estudio de la sensatez y la completitud de los métodos revela la importancia de las características de los lenguajes de programación y de especificación en el área de correctitud de programas, sobre todo en lo relacionado a la construcción de programas correctos, actividad en la que ponemos especial atención en este libro.

Por el carácter introductorio del capítulo nos hemos circunscripto a la interpretación estándar de los números enteros. El lector interesado puede encontrar en las notas adicionales referencias a la sensatez y completitud de los métodos en un marco más amplio.

Notas adicionales

Sensatez y completitud desde una perspectiva más amplia

Analizando las pruebas de sensatez de las axiomáticas para la correctitud parcial que desarrollamos en el capítulo, se observa que son correctas con cualquier interpretación. Notar

que, si en lugar de los números enteros se asume otro dominio semántico y se utilizan los axiomas del nuevo dominio, las fórmulas de correctitud obtenidas siguen siendo verdaderas. Por ejemplo, la prueba de correctitud parcial presentada en el capítulo 2 del programa de división entera:

S_{div} :: c := 0 ; r := x ; while r ≥ y do r := r – y ; c := c + 1 od

con respecto a la especificación ($x \geq 0 \wedge y > 0$, $x = y \cdot c + r \wedge r \geq 0 \wedge r < y$), también sirve asumiendo que las variables del programa son de tipo real. Se define así que las axiomáticas estudiadas tienen sensatez *total*.

No sucede lo mismo con las reglas de prueba de no divergencia REP* y NREP*. Un contraejemplo muy simple en el ámbito de los programas determinísticos es el siguiente. El programa *S :: while x > 0 do x := x – 1 od* termina si se considera el modelo estándar N_0 de los números naturales, pero no termina si el modelo no es estándar y el valor inicial de la variable x es un número no estándar (en un modelo no estándar de los números naturales, a la sucesión de los números estándar, que son los que utilizamos comúnmente, le siguen sucesiones de números no estándar, cada una sin mínimo ni máximo). Sin embargo, sintácticamente, por aplicación de la regla REP*, cualquiera sea el modelo considerado se prueba ⟨true⟩ S ⟨true⟩, dado que de acuerdo a la regla, el variante definido para el *while* varía en el conjunto bien fundado (N_0, <). Por lo tanto, en este caso la sensatez no es total. Se la identifica como *aritmética*, porque requiere que el lenguaje de especificación se extienda con el lenguaje de la aritmética estándar (conocido como *lenguaje de la aritmética de Peano*), incluyendo un predicado unario N que caracterice a los números naturales estándar, y correspondientemente que la interpretación se extienda con la interpretación estándar N_0 de los números naturales. Notar que de esta manera el problema en el ejemplo anterior desaparece, dado que la fórmula a probar se puede expresar con ⟨N(x)⟩ S ⟨true⟩, que se cumple debido a que los números no estándar, que son los que provocan divergencia, no satisfacen la precondición. En el caso de las reglas NREPG* y NREPD* las consideraciones son similares, pero en un entorno más amplio de conjuntos bien fundados.

Por su parte, a la completitud de las axiomáticas para la correctitud parcial se la identifica como *relativa*. Por un lado, porque hay que agregarles todos los axiomas de la interpretación cuando no pueden generarse sistemáticamente. Es lo que efectivamente sucede, de acuerdo al *teorema de incompletitud de Gödel*, con los programas con variables enteras con los que trabajamos (la demostración del teorema se describe, por ejemplo, en (Mosterín, 1988)). Y, por otro lado, porque la expresividad del lenguaje de especificación, es decir la capacidad para expresar la postcondición de cualquier programa a partir de cualquier precondición, depende del lenguaje de programación y de la interpretación. Cabe destacar que, con el lenguaje de la lógica de predicados, el lenguaje de la aritmética de Peano y la interpretación estándar de los números naturales, la expresividad se cumple en el marco de numerosos lenguajes de programación. Las computaciones de los programas se pueden codificar con números naturales, recurriendo a las

operaciones de suma y multiplicación (en (de Bakker, 1980) se explica la codificación, conocida como *gödelización*).

En lo que hace a las reglas para la prueba de no divergencia referidas, su completitud es *aritmética* por la misma razón que lo es su sensatez. Particularmente, a diferencia de lo que se requiere en las pruebas de correctitud parcial, con las operaciones de suma y multiplicación del lenguaje aritmético no alcanza para expresar todos los variantes (es el caso, por ejemplo, de los variantes de los *while* con una cantidad de iteraciones del orden exponencial): las expresiones enteras deben permitir describir todas las funciones computables.

(Clarke, 1985) incluye varios resultados sobre la completitud en la lógica de Hoare.

Introducción a la semántica denotacional

La semántica operacional, descripta en el capítulo, permite en general especificar la semántica de un lenguaje de programación de una manera simple e intuitiva. Incluso puede actuar como guía de implementación, si se basa en cada caso en una máquina abstracta adecuada. El nivel de abstracción elegido es muy relevante, dado que si se abunda en detalles innecesarios en la descripción de un programa se condiciona su ulterior desarrollo, y peor aún, se pone en riesgo su comprensión.

La semántica denotacional, presentada inicialmente en (Scott y Strachey, 1971), no tiene este riesgo, ya que no recurre a máquinas, configuraciones ni computaciones, sino a objetos matemáticos más abstractos, como conjuntos, funciones, funcionales, etc. Se define mediante funciones que relacionan estructuras sintácticas con objetos matemáticos. Toda estructura denota un valor (de acá surge el nombre de la semántica), determinado a partir del valor de los componentes de la estructura. El problema con esta semántica es que su uso se dificulta cuando se debe tratar con lenguajes de programación complejos, sobre todo los lenguajes concurrentes. En definitiva, las dos semánticas son complementarias más que alternativas.

A continuación, presentamos una breve introducción a la semántica denotacional, mostrando directamente cómo especificar denotacionalmente las expresiones, los predicados y las instrucciones del lenguaje de programación determinístico que utilizamos en el libro.

Primero consideramos las expresiones y los predicados:

$e :: n \mid x \mid (e_1 + e_2) \mid (e_1 - e_2) \mid (e_1 \,.\, e_2) \mid \ldots \mid (\text{if } B \text{ then } e_1 \text{ else } e_2 \text{ fi})$

$B :: \text{true} \mid \text{false} \mid (e_1 = e_2) \mid (e_1 < e_2) \mid \ldots \mid \neg B_1 \mid (B_1 \vee B_2) \mid (B_1 \wedge B_2) \mid \ldots$

$p :: \text{true} \mid \text{false} \mid (e_1 = e_2) \mid (e_1 < e_2) \mid \ldots \mid \neg p_1 \mid (p_1 \vee p_2) \mid (p_1 \wedge p_2) \mid \ldots \mid \exists x: p_1 \mid \forall x: p_1$

Con x se representa una variable entera y con n una constante entera. Para especificar la semántica denotacionalmente, usamos los conjuntos Z de los números enteros (con elementos típicos α), X de los valores de verdad *verdadero* y *falso* (con elementos típicos β) y Σ de los

estados (con elementos típicos σ); las funciones que implementan las operaciones asociadas a dichos conjuntos (suma, igualdad, negación, etc); y las siguientes funciones semánticas:

- V : Aexp ⟶ (Σ ⟶ Z) describe la semántica del conjunto Aexp de las expresiones aritméticas. Dados una expresión aritmética y un estado, V devuelve un número entero. Su definición inductiva es:

  1. $V(n)(\sigma) = \alpha_n$, tal que n es la representación del número entero α_n (por ejemplo, el 1 sintáctico representa al 1 semántico).
  2. $V(x)(\sigma) = \sigma(x)$.
  3. $V((e_1 + e_2))(\sigma) = V(e_1)(\sigma) + V(e_2)(\sigma)$, y lo mismo se define para los otros operadores.
  4. $V((\text{if } B \text{ then } e_1 \text{ else } e_2 \text{ fi}))(\sigma) = \text{if } W(B)(\sigma) \text{ then } V(e_1)(\sigma) \text{ else } V(e_2)(\sigma) \text{ fi}$, tal que la expresión *if β then α_1 else α_2 fi* es igual a α_1 o α_2 según β sea verdadero o falso, respectivamente. W se especifica en lo que sigue.

- W : Bexp ⟶ (Σ ⟶ X) describe la semántica del conjunto Bexp de las expresiones booleanas. Dados una expresión booleana y un estado, W devuelve un valor de verdad. Su definición inductiva es:

  1. $W(\text{true})(\sigma)$ = verdadero.
     $W(\text{false})(\sigma)$ = falso.
  2. $W((e_1 = e_2))(\sigma) = (V(e_1)(\sigma) = V(e_2)(\sigma))$, y lo mismo se define para los otros operadores.
  3. $W(\neg B_1)(\sigma) = \neg W(B_1)(\sigma)$, $W((B_1 \vee B_2))(\sigma) = W(B_1)(\sigma) \vee W(B_2)(\sigma)$, etc.

- T : Aser ⟶ (Σ ⟶ X) describe la semántica del conjunto Aser de los predicados (o aserciones). Dados un predicado y un estado, T devuelve un valor de verdad. Su definición inductiva es:

  1. $T(\text{true})(\sigma)$ = verdadero.
     $T(\text{false})(\sigma)$ = falso.
  2. $T((e_1 = e_2))(\sigma) = (V(e_1)(\sigma) = V(e_2)(\sigma))$, y lo mismo se define para los otros operadores.
  3. $T(\neg p_1)(\sigma) = \neg T(p_1)(\sigma)$, $T((p_1 \vee p_2))(\sigma) = T(p_1)(\sigma) \vee T(p_2)(\sigma)$, etc.
  4. $T(\exists x: p_1)(\sigma)$ = verdadero sii existe algún α tal que $T(p_1)(\sigma[x|\alpha])$ = verdadero.
     $T(\forall x: p_1)(\sigma)$ = verdadero sii para todo α vale $T(p_1)(\sigma[x|\alpha])$ = verdadero.

Las propiedades de *generación finita* y *única descomposición* de las expresiones y los predicados garantizan la *existencia* y *unicidad* de las soluciones de las funciones semánticas definidas. Notar que la semántica de σ(e) y σ(B), abusos de notación que empleamos en el capítulo para simplificar las descripciones, ahora queda clara con las descripciones de V y W. Notar también que la expresión T(p)(σ) = verdadero es equivalente a σ |= p, es decir que es otra manera de indicar que un determinado estado satisface un determinado predicado.

Ahora consideramos el lenguaje de los programas determinísticos:

S :: skip | x := e | S_1 ; S_2 | if B then S_1 else S_2 fi | while B do S_1 od (x es alguna variable entera)

La función semántica correspondiente es:

- $M : L_w \longrightarrow (\Sigma \longrightarrow \Sigma)$. Dados un programa y un estado inicial, M devuelve un estado final. Su definición inductiva es:

  1. $M(skip)(\sigma) = \sigma$
  2. $M(x := e)(\sigma) = \sigma[x|V(e)(\sigma)]$
  3. $M(S_1 ; S_2)(\sigma) = M(S_2)(M(S_1)(\sigma))$
  4. $M(\text{if B then } S_1 \text{ else } S_2 \text{ fi})(\sigma) = M(S_1)(\sigma)$, si $W(B)(\sigma)$ = verdadero
     $M(S_2)(\sigma)$, si $W(B)(\sigma)$ = falso
  5. $M(\text{while B do S od})(\sigma) = M(\text{while B do S od})(M(S)(\sigma))$, si $W(B)(\sigma)$ = verdadero
     σ, si $W(B)(\sigma)$ = falso

  tal que $M(S)(\sigma) = \perp$ si S diverge a partir de σ (M es una función *total*). En particular, se define $M(S)(\perp) = \perp$ para todo S (se dice que M es una función *estricta*). Pero la especificación así planteada está incompleta, falta explicitar la solución de la función semántica cuando se aplica a la instrucción *while*: notar que, a diferencia del resto de las instrucciones, en este caso la semántica de la instrucción no está expresada en términos de componentes más simples. En lo que resta de esta nota describimos una de las maneras existentes para obtener la solución (función φ), basada en la noción matemática de *límite* (otra alternativa, de alcance más amplio, se basa en la noción de *punto fijo*).

La idea es calcular φ como el límite de una secuencia de funciones φ_i, con $i \geq 0$, conocidas como *aproximaciones*, que representan las sucesivas iteraciones del *while*. Para asegurar su existencia y unicidad se recurre a los *órdenes parciales completos* (abreviados con *opc*), órdenes parciales (C, $\leq$) tales que:

- Tienen primer elemento: existe un x de C tal que para todo *y* de C se cumple $x \leq y$.
- Toda cadena ascendente tiene cota superior mínima: si $\{x_0, x_1, \ldots\} \subseteq C$, con $x_0 \leq x_1 \leq \ldots$, entonces existe un *y* de C que cumple $x_i \leq y$ para todo x_i , y si también existe un z de C que cumple $x_i \leq z$ para todo x_i , entonces $y \leq z$.

Para facilitar la notación, abreviamos $x_0 \leq x_1 \leq \ldots$ con $\{x_i\}_{i \geq 0}$, y a un opc (C, $\leq$) lo llamamos directamente C cuando queda claro el contexto.

El primer paso es definir los siguientes dos opc:

- $(\Sigma, \leq_1)$. Para todo par de estados, $\sigma \leq_1 \sigma'$ sii $(\sigma = \perp \vee \sigma = \sigma')$, es decir que todo estado se relaciona sólo consigo mismo, salvo $\perp$ que se relaciona con todos y por lo tanto es el primer elemento. Se prueba fácilmente que $(\Sigma, \leq_1)$, o directamente Σ, es un opc.
- $(F_e, \leq_2)$. Se define sobre el conjunto de las funciones estrictas de Σ a Σ de la siguiente manera: para todo par de funciones, $\varphi \leq_2 \varphi'$ sii $(\varphi(\sigma) = \perp \vee \varphi(\sigma) = \varphi'(\sigma))$, cualquiera sea el estado σ. Intuitivamente, φ' es mejor aproximación que φ. Se prueba fácilmente que $(F_e, \leq_2)$, o directamente F_e, es un opc.

El segundo paso es plantear la siguiente secuencia de funciones φ_i:

1. $\varphi_0(\sigma) = \perp$
2. $\varphi_{i+1}(\sigma) = \varphi_i(M(S)(\sigma))$, si $W(B)(\sigma)$ = verdadero
   σ, si $W(B)(\sigma)$ = falso

Se demuestra sin dificultad que todas las funciones φ_i son estrictas y que $\{\varphi_i\}_{i \geq 0}$ es una cadena ascendente de F_e. Y así se llega al paso final: la función solución φ buscada es la cota superior mínima de $\{\varphi_i\}_{i \geq 0}$. Por ejemplo, aplicando φ al programa *while x > 0 do x := x – 1 od* y el estado inicial $\sigma[x|2]$ resulta:

$\varphi_0(\sigma[x|2]) = \perp$
$\varphi_1(\sigma[x|2]) = \varphi_0(\sigma[x|1]) = \perp$
$\varphi_2(\sigma[x|2]) = \varphi_1(\sigma[x|1]) = \varphi_0(\sigma[x|0]) = \perp$
$\varphi_3(\sigma[x|2]) = \varphi_2(\sigma[x|1]) = \varphi_1(\sigma[x|0]) = \sigma[x|0]$
y en general, $\varphi_i(\sigma[x|2]) = \sigma[x|0]$, con i = 4 en adelante

es decir que el estado final es $\sigma[x|0]$.

Referencias bibliográficas

Las pruebas de sensatez y completitud desarrolladas en el capítulo se basan en las que se presentan en (Francez, 1992) y (Apt y Olderog, 1997). En (Apt, 1981) y (Apt, 1984) se profundiza en las mismas, incluyendo la caracterización de distintos tipos de sensatez y completitud. Las nociones de completitud relativa y aritmética se originan en (Cook, 1978) y (Harel, 1979), respectivamente. La prueba de completitud de la regla de no divergencia con asignaciones aleatorias se puede encontrar en (Apt y Plotkin, 1986), y con *fairness* en (Stomp, de Roever y Gerth, 1989).

Los textos fundacionales de la semántica operacional de los lenguajes de programación son (Hennessy y Plotkin, 1979) y (Plotkin, 1981). Además de la semántica operacional y denotacional,

también se considera, entre otras, a la propia lógica de Hoare como un tipo de especificación semántica, la semántica axiomática, sobre todo a partir de la publicación (Hoare y Wirth, 1973), en donde se especifica de esta manera el lenguaje Pascal. En (Winskel, 1993) se comparan las tres semánticas, en términos de un mismo lenguaje de programación. Dicha obra, además, dedica capítulos a los principios inductivos, la completitud en el marco de la lógica de Hoare, y conceptos generales sobre incompletitud e indecibilidad.

Presentamos a continuación el detalle de las referencias bibliográficas, incluyendo las que figuran en las notas adicionales:

Apt, K. (1981). *Ten years of Hoare's logic: a survey - part 1*. ACM Trans. Prog. Lang. Syst., 3, 4, 431-483.

Apt, K. (1984). *Ten years of Hoare's logic, a survey, part 2: nondeterminism.* Theoretical Computer Science, 28, 83-109.

Apt, K. y Olderog, E. (1997). *Verification of secuencial and concurrent programs, second edition*. Springer-Verlag.

Apt, K. y Plotkin, G. (1986). *Contable nondeterminism and random assignment.* JACM, 33, 4, 724-767.

Clarke, E. (1985). *The characterization problem in Hoare logics*. Mathematical Logic and Programming Languages, 89-106. Englewood Cliffs NJ.

Cook, S. (1978). *Soundness and completeness of an axiom system for program verification*. SIAM J. Comput., 7, 1, 70-90.

de Bakker, J. (1980). *Mathematical theory of programm correctness*. Prentice Hall International.

Francez, N. (1992). *Program verification*. Addison-Wesley.

Harel, D. (1979). *First order dynamic logic*. Lecture Notes in Computer Science, 68. Springer-Verlag.

Hennessy, M. y Plotkin, G. (1979). *Full abstraction for a simple programming language*. Lecture Notes in Computer Science, 74, 108-120.

Hoare, C. y Wirth, N. (1973). *An axiomatic definition of the programming language PASCAL*. Acta Informatica, 2, 335-355.

Mosterín, J. (1988). *Kurt Gödel. Obras completas, 2da edición*. Alianza Editorial.

Plotking, G. (1981). *A structural approach to operational semantics*. Tech. Rep., Dep. of Computer Science, Aarhus Univ.

Scott, D. y Strachey, C. (1971). *Towards a mathematical semantics for computer languages*. Tech. Report., Univ. of Oxford.

Stomp, F., de Roever, W. y Gerth, R. (1989). *The μ-calculus as an assertion language for fairness arguments*. Information and Computation, 82, 3, 278-322.

Winskel, G. (1993). *The formal semantics of programming languages. An introduction.* The MIT Press.

Ejercicios

1. Definir inductivamente el conjunto var(e) de las variables de una expresión aritmética *e*, la cual tiene una de las siguientes formas: e :: 0 | 1 | x | (e_1 + e_2) | (e_1 **.** e_2).

Resolución

var(0) = ∅.
var(1) = ∅.
var(x) = {x}.
var((e_1 + e_2)) = var(e_1) ∪ var(e_2).
var((e_1 **.** e_2)) = var(e_1) ∪ var(e_2).

2. Definir inductivamente el conjunto var(B) de las variables de una expresión booleana B, la cual tiene una de las siguientes formas: B :: true | false | (e_1 = e_2) | ¬B_1 | (B_1 ∨ B_2), siendo las e_i las expresiones aritméticas definidas en el ejercicio anterior.

3. En el capítulo se probó, para las alternativas 0, 1 y (e_1 + e_2) de las expresiones aritméticas *e* definidas en el ejercicio 1, que se cumple c(e) = 1 + o(e), donde c(e) y o(e) son las cantidades de constantes y operadores de *e*, respectivamente. Completar la prueba considerando la alternativa (e_1 **.** e_2).

4. Sea la siguiente variante para especificar la semántica del lenguaje determinístico L_w de los programas con *while*: a cada programa S y cada estado σ se le asocia una secuencia de estados σ, $σ_1$, $σ_2$, …, σ´, identificada con comp(S, σ) y definida inductivamente de la siguiente manera para el *skip*, la asignación y la secuencia:

   1) comp(skip)(σ) = σ
   2) comp(x := e)(σ) = σ[x|e]
   3) comp(S_1 ; S_2)(σ) = comp(S_1)(σ) **.** comp(S_2)(σ**'**), tal que **.** es el operador de concatenación de secuencias y σ' es el último estado de comp(S_1)(σ).

   Completar la definición para la selección condicional y la repetición.

5. Probar: val(π(S_1, σ)) = val(π(S_2, σ)) sii (|= {p} S_1 {q} sii |= {p} S_2 {q}), cualesquiera sean los programas determinísticos S_i, el estado σ y los predicados p y q.

6. La función semántica del lenguaje no determinístico L_{cg} de los programas con comandos guardados se define con M_{cg} : L_{cg} ⟶ (Σ ⟶ P(Σ)). Responder:

   6.1 ¿Por qué se utiliza P(Σ) en lugar de Σ?

6.2 ¿Qué conjuntos de estados de P(Σ) no puede generar un programa de L_{cg} cuando no hay hipótesis de *fairness*? ¿Y cuando la hay?

6.3 Se define: $M_{cg}(skip)(\sigma) = \{\sigma\}$ y $M_{cg}(x := e)(\sigma) = \{\sigma[x|e)\}$. Completar la definición para la secuencia, el IF y el DO.

7. Sea la instrucción *if B then S_1 fi*, que se incorpora al lenguaje determinístico. Definir su semántica operacional.

Resolución

Si $\sigma(B)$ = verdadero, entonces (if B then S_1 fi, σ) $\longrightarrow$ (S_1, σ).

Si $\sigma(B)$ = falso, entonces (if B then S_1 fi, σ) $\longrightarrow$ (E, σ).

8. Definir la semántica operacional de las siguientes instrucciones, suponiendo que se incorporan al lenguaje determinístico:

8.1 *repeat S until B*, que primero ejecuta una vez incondicionalmente el cuerpo S y luego lo itera en tanto no se cumpla la condición B.

8.2 *S_1 or S_2*, que ejecuta S_1 o S_2.

8.3 *S_1 || S_2*, que ejecuta S_1 y luego S_2, o S_2 y luego S_1.

9. Describir, de acuerdo a la semántica operacional de los lenguajes L_w y L_{cg}, las formas posibles de las computaciones correspondientes a la selección condicional determinística (las que no desarrollamos en el capítulo) y todas las instrucciones no determinísticas.

10. Probar la sensatez de la siguiente regla (regla AND, que describimos en el capítulo 2):

$$\frac{\{p_1\}\ S\ \{q_1\},\ \{p_2\}\ S\ \{q_2\}}{\{p_1 \wedge p_2\}\ S\ \{q_1 \wedge q_2\}}$$

Resolución

Asumiendo |– $\{p_1 \wedge p_2\}$ S $\{q_1 \wedge q_2\}$ luego de la aplicación de la regla AND, hay que probar que se cumple |= $\{p_1 \wedge p_2\}$ S $\{q_1 \wedge q_2\}$.

De acuerdo a la regla AND, $\{p_1 \wedge p_2\}$ S $\{q_1 \wedge q_2\}$ se obtiene de $\{p_1\}$ S $\{q_1\}$ y $\{p_2\}$ S $\{q_2\}$. Así podemos partir de la hipótesis inductiva |= $\{p_1\}$ S $\{q_1\}$ y |= $\{p_2\}$ S $\{q_2\}$ para llegar a la fórmula de correctitud buscada. Sea σ_0 un estado tal que $\sigma_0 \models p_1 \wedge p_2$ y $val(\pi(S, \sigma_0)) = \sigma_1 \neq \perp$. Entonces, por la semántica del lenguaje y la hipótesis inductiva, (S, σ_0) $\longrightarrow^*$ (E, σ_1), tal que $\sigma_1 \models q_1 \wedge q_2$, lo que completa la prueba.

11. Probar la sensatez de los axiomas ASI y NASI y de las reglas COND, NCOND y NREP.

12. Probar la sensatez de las siguientes reglas para los programas determinísticos:

    12.1 *Regla OR´* (variante de la regla OR, descripta en el capítulo 2)

$$\frac{\{p_1\}\ S\ \{q_1\},\ \{p_2\}\ S\ \{q_2\}}{\{p_1 \vee p_2\}\ S\ \{q_1 \vee q_2\}}$$

    12.2 *Regla SHIFT*

$$\frac{\{p \wedge q\}\ S\ \{r\}}{\{p\}\ S\ \{q \rightarrow r\}}$$

tal que q y S no comparten variables

    12.3. *Regla REPEAT*

$$\frac{\{p\}\ S\ \{q\},\ (q \wedge \neg B) \longrightarrow p}{\{p\}\ \text{repeat } S \text{ until } B\ \{q \wedge B\}}$$

13. Probar la completitud de los axiomas ASI y NASI y de las reglas COND, NCOND y NREP.

14. Probar que el método de verificación de correctitud parcial de los programas determinísticos permite derivar, para todo programa S, la fórmula {true} S {true}.

Resolución

Aplicamos inducción estructural, considerando todas las formas posibles de S:

1) S :: skip. Se obtiene {true} skip {true} por el axioma SKIP.
2) S :: x := e. Se obtiene {true} x := e {true} por el axioma ASI.
3) S :: S_1 ; S_2. La hipótesis inductiva establece que se prueban las fórmulas {true} S_1 {true} y {true} S_2 {true}. A partir de ellas, {true} S_1 ; S_2 {true} se obtiene aplicando SEC.
4) S :: if B then S_1 else S_2 fi. La hipótesis inductiva establece que se prueban {true} S_1 {true} y {true} S_2 {true}. A partir de ellas se obtienen {true $\wedge$ B} S_1 {true} y {true $\wedge \neg$B} S_2 {true} aplicando CONS. Finalmente, aplicando COND sobre estas dos últimas fórmulas se llega a {true} if B then S_1 else S_2 fi {true}.
5) S :: while B do S_1 od. La hipótesis inductiva establece que se prueba {true} S_1 {true}. Por CONS se obtiene {true $\wedge$ B} S_1 {true}, luego por REP, {true} while B do S_1 od {true $\wedge \neg$B}, y finalmente por CONS de nuevo se llega a {true} while B do S_1 od {true}.

15. Probar que el método de verificación de correctitud parcial de los programas determinísticos permite derivar, para todo programa S y todo predicado p, la fórmula {false} S {p}.

16. Probar que la siguiente variante de la regla AND es redundante, es decir que el método de verificación de correctitud parcial de los programas determinísticos permite derivar todas las fórmulas de correctitud parcial sin recurrir a ella:

$$\frac{\{p\}\ S\ \{q\},\ \{p\}\ S\ \{r\}}{\{p\}\ S\ \{q \wedge r\}}$$

Resolución

Aplicamos inducción estructural, considerando todas las formas posibles de S:

1) S :: skip. Supongamos que se prueban {p} skip {q} y {p} skip {r}. Por SKIP, debe cumplirse $p \rightarrow q$ y $p \rightarrow r$, y por lo tanto $p \rightarrow (q \wedge r)$. También por SKIP, {p} skip {p}. Finalmente, por CONS, llegamos a {p} skip {$q \wedge r$}.
2) S :: x := e. Supongamos que se prueban {p} x := e {q} y {p} x := e {r}. Por ASI, debe cumplirse $p \rightarrow q[x|e]$ y $p \rightarrow r[x|e]$, y por lo tanto $p \rightarrow (q \wedge r)[x|e]$. También por ASI se obtiene {$(q \wedge r)[x|e]$} x := e {$q \wedge r$}. Finalmente, por CONS, llegamos a {p} x := e {$q \wedge r$}.
3) S :: S_1 ; S_2. Supongamos que se prueban {p} S_1 ; S_2 {q} y {p} S_1 ; S_2 {r}. Así, de acuerdo a SEC, se prueban {p} S_1 {t_1}, {t_1} S_2 {q}, {p} S_1 {t_2} y {t_2} S_2 {r}, para algún t_1 y algún t_2. Como $(t_1 \wedge t_2) \rightarrow t_1$ y $(t_1 \wedge t_2) \rightarrow t_2$, por CONS, {$t_1 \wedge t_2$} S_2 {q} y {$t_1 \wedge t_2$} S_2 {r}.
   Entonces, dados {p} S_1 {t_1}, {p} S_1 {t_2}, {$t_1 \wedge t_2$} S_2 {q} y {$t_1 \wedge t_2$} S_2 {r}, planteamos como hipótesis inductiva: {p} S_1 {$t_1 \wedge t_2$} y {$t_1 \wedge t_2$} S_2 {$q \wedge r$}, y así por SEC obtenemos la fórmula {p} S_1 ; S_2 {$q \wedge r$}.
4) S :: if B then S_1 else S_2 fi. Como en (3), partimos de suponer que se prueban las fórmulas, en este caso, {p} if B then S_1 else S_2 fi {q} y {p} if B then S_1 else S_2 fi {r}. Así, por COND también se prueban {$p \wedge B$} S_1 {q}, {$p \wedge \neg B$} S_2 {q}, {$p \wedge B$} S_1 {r} y {$p \wedge \neg B$} S_2 {r}.
   Entonces, planteando como hipótesis inductiva: {$p \wedge B$} S_1 {$q \wedge r$} y {$p \wedge \neg B$} S_2 {$q \wedge r$}, llegamos por COND a {p} if B then S_1 else S_2 fi {$q \wedge r$}.
5) S :: while B do S_1 od. Siguiendo el esquema anterior, supongamos que se prueban las fórmulas {p} while B do S_1 od {q} y {p} while B do S_1 od {r}. De esta manera, por REP, se cumple {$inv \wedge B$} S_1 {inv}, $p \rightarrow inv$, $(inv \wedge \neg B) \rightarrow q$ y $(inv \wedge \neg B) \rightarrow r$, para algún predicado inv que actúa como invariante del *while*. Por lo tanto, se cumple $(inv \wedge \neg B) \rightarrow (q \wedge r)$.
   Entonces, dada la hipótesis inductiva {$inv \wedge B$} S_1 {inv}, $p \rightarrow inv$, $(inv \wedge \neg B) \rightarrow (q \wedge r)$, aplicando REP se obtiene la fórmula {p} while B do S_1 od {$q \wedge r$}.

17. Probar la redundancia de la siguiente regla en el método de verificación de programas determinísticos (la regla OR del capítulo 2):

$$\frac{\{p\}\ S\ \{q\},\ \{r\}\ S\ \{q\}}{\{p \vee r\}\ S\ \{q\}}$$

CAPÍTULO 5
Verificación de programas concurrentes

Introducción

Con este capítulo completamos la descripción de los métodos axiomáticos. Presentamos un tercer método, con distintas variantes axiomáticas, para verificar *programas concurrentes*. Así como planteamos en el capítulo 3 la verificación de programas secuenciales no determinísticos por medio de una adaptación del método de prueba estudiado en el capítulo 2 para programas secuenciales determinísticos, así lo planteamos también en este capítulo para la verificación de programas concurrentes con axiomáticas que son extensiones de aquéllas, para seguir introduciendo de manera gradual los principios fundamentales de la verificación de programas.

Un programa concurrente contiene subprogramas secuenciales, conocidos como *procesos*, que se ejecutan concurrentemente por medio de uno o más procesadores, en este último caso en general con velocidades que pueden diferir ampliamente. Así, durante su ejecución existen simultáneamente varios puntos o posiciones de control, uno por proceso. Los procesos se pueden *comunicar* y *sincronizar*. Estas y otras características de los programas concurrentes hacen que su verificación resulte mucho más compleja que la de los programas secuenciales.

Consideramos una semántica muy común de la concurrencia, el *interleaving*, que consiste en la intercalación no determinística de las acciones atómicas (ininterrumpibles) de los procesos, por lo que, como en los programas no determinísticos, en este marco los programas concurrentes pueden producir a partir de un estado inicial varias computaciones y varios estados finales.

El capítulo es más extenso que los anteriores. Además de trabajar con distintas clases de programas y de tratar más propiedades, la propia problemática de la programación concurrente amerita analizar un mayor número de aspectos (relacionados con la composicionalidad, la completitud, el *fairness*, etc). Así y todo, por razones de espacio no incluimos algunos temas, y otros los desarrollamos resumidamente. Por ejemplo, ejemplificamos el desarrollo sistemático de una única clase de programas, y sólo hacemos breves referencias al *fairness*.

Para facilitar la lectura del capítulo lo dividimos en dos partes, según el modelo de comunicación entre procesos que utilizan los programas considerados. La primera parte está dedicada a los programas con procesos que se comunican mediante *variables compartidas*, y la segunda parte a los programas con procesos disjuntos, o sea que no comparten variables, que se comunican a través de *pasajes de mensajes*. En la terminología de este libro, identificamos a

las dos clases de programas como programas *paralelos* y programas *distribuidos*, respectivamente. La primera parte sirve además para introducir conceptos generales de la verificación de programas del paradigma concurrente. La exposición de los temas de cada parte sigue la estructura habitual: primero introducimos los programas con los que trabajamos y las propiedades a verificar, después describimos las variantes axiomáticas correspondientes y mostramos ejemplos de aplicación, y finalmente, en este caso sólo en la primera parte, presentamos un ejemplo de desarrollo sistemático de programa basado en la axiomática descripta. Concluimos el capítulo, como siempre, con secciones de observaciones finales, notas adicionales, referencias bibliográficas y ejercicios. En particular, aprovechando los elementos de semántica formal introducidos en el capítulo anterior, en una nota adicional describimos algunas características de la semántica operacional de los lenguajes concurrentes tratados.

Parte 1. Verificación de programas paralelos

En esta primera parte trabajamos con dos lenguajes de programación paralelos, ambos basados en el lenguaje secuencial determinístico de repeticiones *while* introducido en el capítulo 2. Los lenguajes difieren en la instrucción de sincronización que utilizan, una muy elemental y la otra más estructurada, diferencia que se traduce en variantes axiomáticas de distinta complejidad.

Lenguaje de variables compartidas

Los programas del primer lenguaje paralelo que consideramos tienen la siguiente forma:

$S :: S_0 ; [S_1 \,||\, \ldots \,||\, S_n]$

S_0 es un subprograma secuencial con instrucciones del lenguaje de los *while* (puede no existir, su función es inicializar variables).

$[S_1 \,||\, \ldots \,||\, S_n]$, que se puede abreviar con $[\,||_{i=1,n} S_i\,]$, es la composición paralela de un conjunto de procesos S_1, …, S_n (se pueden etiquetar), los cuales contienen instrucciones del lenguaje de los *while* y eventualmente instrucciones de sincronización *await*, que describimos enseguida.

Los procesos pueden compartir variables. Para no complicar las descripciones, no se permite concurrencia anidada ni procesos dinámicos.

La ejecución de un programa $S_0 ; [\,||_{i=1,n} S_i\,]$ consiste en la ejecución de S_0 seguida de la ejecución concurrente de S_1, …, S_n.

El programa puede tener varias computaciones, producto de la semántica de *interleaving* asumida, y, por lo tanto, a partir de un estado inicial puede obtener varios estados finales. Mientras haya una instrucción posible para ejecutar, el programa la ejecuta. Esta es la única

hipótesis de progreso que manejamos para todas las clases de programas del capítulo, conocida como *propiedad de progreso fundamental* (salvo mención explícita, no consideramos ninguna asunción de *fairness*).

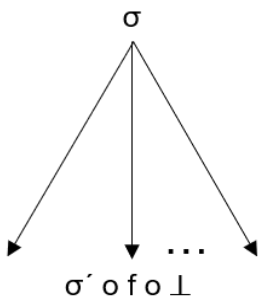
Los procesos se pueden comunicar a través de sus variables compartidas. De esta manera, el contenido de una variable compartida en un momento dado depende de las asignaciones que se le hayan aplicado desde distintos procesos.

Las ejecuciones de los *skip* y las asignaciones y las evaluaciones de las expresiones booleanas son atómicas (que no es lo que ofrecen los lenguajes concurrentes reales, que sólo garantizan la atomicidad de una *referencia crítica simple* - lectura o escritura exclusiva de una variable compartida -, pero optamos por este nivel de abstracción para simplificar la presentación, en lugar de complicar los programas con variables auxiliares o instrucciones de sincronización para lograr la atomicidad requerida). Y para ejecutar fragmentos de programa con variables compartidas más amplios que requieran ser ejecutados atómicamente, conocidos como *secciones críticas*, el lenguaje provee la instrucción de sincronización *await* que ya anticipamos, cuya sintaxis es:

await B $\rightarrow$ S end

B es una expresión booleana, y S es un subprograma que no contiene otras instrucciones *await* ni instrucciones *while*, lo que hace que la ejecución de un *await* siempre se complete. La ejecución consiste en evaluar B, si B resulta verdadera se ejecuta el *await* completo atómicamente, y si en cambio B resulta falsa, el proceso queda *bloqueado* hasta que en una instancia posterior progresa si es que el escenario se modifica.

Al igual que lo que observamos en los programas secuenciales no determinísticos, el *árbol de computaciones* de un programa del lenguaje de variables compartidas tiene en el caso más general computaciones que *terminan* (terminan sin falla), computaciones que *fallan* (terminan con falla) y computaciones que *divergen* (no terminan):



σ´ es un estado final sin falla, f es un estado de falla, y el símbolo ⊥ representa una computación infinita. La novedad es que las fallas son casos de *deadlock*, situaciones que se producen cuando uno o más procesos quedan bloqueados indefinidamente (para simplificar, no consideramos otras fallas).

Por ejemplo, el siguiente programa devuelve en la variable n el factorial de un número N $\geq$ 1:

```
Sfac :: c1 := true ; c2 := true ; i := 1 ;  j := N ; n := N ;
    [S1 :: while c1 do await true →                    S2 :: while c2 do await true →
        if i + 1 < j then i := i + 1 ; n := n . i   ||     if j – 1 > i then j := j – 1 ; n := n . j
        else c1 := false fi end                            else c2 := false fi end
        od                                                 od]
```

El programa tiene dos procesos que colaboran en el cálculo. Como n se inicializa con N, un proceso multiplica iterativamente n por 2, 3, …, y el otro multiplica iterativamente n por (N – 1), (N – 2), …. Los procesos terminan cuando sus respectivos índices i y j cumplen i + 1 = j. Los valores finales de i y j dependen del *interleaving* ejecutado (notar que puede darse el caso de que un solo proceso lleve a cabo todo el cálculo). Los *await* se utilizan sólo para lograr la *exclusión mutua* entre los procesos: cada cálculo parcial realizado por un proceso utiliza las variables compartidas n, i y j, por lo que debe efectuarse sin *interferencias* del otro proceso.

Este otro programa, en cambio, utiliza los *await* para sincronizar los procesos, y aprovecha mejor la concurrencia. El programa resuelve el clásico problema del productor-consumidor:

```
Spc :: in := 0 ; out := 0 ; i := 1 ; j := 1 ;
    [prod :: while i ≤ n do                          cons :: while j ≤ n do
            x := a[i] ;                                      await in – out > 0 → skip end
            await in – out < b → skip end ;   ||             y := buffer[out mod b] ;
            buffer[in mod b] := x ;                          out := out + 1 ;
            in := in + 1 ;                                   b[j] := y ;
            i := i + 1                                       j := j + 1
            od                                               od]
```

El programa copia el arreglo *a*, recorrido por el índice i, en el arreglo b, recorrido por el índice j. Los arreglos, de números enteros, tienen n elementos. Para la copia se utiliza un tercer arreglo, *buffer*, de tamaño b, recorrido por los índices in y out, cuyos valores representan la cantidad de elementos que se agregan y se extraen de *buffer*, respectivamente. De esta manera, *buffer* tiene en todo momento in – out elementos, e in y out deben evaluarse módulo b. El proceso *prod* escribe en *buffer* siempre que haya espacio disponible, es decir si se cumple in – out < b, y el proceso *cons* lee de *buffer* siempre que éste tenga elementos, es decir si se cumple in – out > 0. Cuando alguna de estas condiciones no se cumple, el proceso respectivo queda bloqueado hasta que la evaluación de la condición resulta verdadera.

En lo que sigue, presentamos una axiomática para verificar programas del lenguaje de variables compartidas. Describimos axiomas y reglas para probar las propiedades que constituyen su *correctitud total*: la *correctitud parcial*, la *no divergencia* y la *ausencia de deadlock*. Postergamos hasta las secciones dedicadas a la segunda clase de programas paralelos que tratamos, referencias a otras dos propiedades comúnmente consideradas en el paradigma concurrente: la *exclusión mutua* (o *ausencia de interferencia*) y la *ausencia de inanición*, que

establecen, respectivamente, la ejecución sin interferencias de las secciones críticas definidas y el acceso a las mismas por parte de todos los procesos que las requieran.

Axiomática para las pruebas de correctitud parcial

La axiomática para las pruebas de correctitud parcial de programas paralelos del lenguaje de variables compartidas incluye los axiomas y reglas que presentamos para la prueba de correctitud parcial de programas secuenciales determinísticos, e incorpora tres reglas más, dos correspondientes a las instrucciones novedosas del lenguaje, el *await* y la composición paralela, y una tercera, cuya necesidad explicamos más adelante. Describimos a continuación las dos primeras (seguimos, a lo largo del capítulo, con la notación y con la numeración anterior de las reglas y los ejemplos de aplicación):

21. *Regla del await* (AWAIT)

$$\frac{\{p \wedge B\}\ S\ \{q\}}{\{p\}\ \text{await } B \rightarrow S \text{ end } \{q\}}$$

Por la forma del *await*, la regla es muy similar a la regla del condicional (COND). De hecho, no hace referencia a la posibilidad de bloqueo, porque la ausencia de *deadlock* se trata en otra prueba.

En lo que respecta a la regla correspondiente a la composición paralela, siguiendo el principio composicional sostenido en el marco secuencial su forma natural sería:

$$\frac{\{p_i\}\ S_i\ \{q_i\},\ i = 1, \ldots, n}{\{\bigwedge_{i=1,n} p_i\}\ [\ ||_{i=1,n} S_i\ ]\ \{\bigwedge_{i=1,n} q_i\}}$$

tratando a los procesos S_i como cajas negras, en este caso considerando las conjunciones de sus pre y postcondiciones en lugar de secuencializarlas. Pero desafortunadamente esta regla no es sensata, la conclusión no tiene por qué ser verdadera, como se demuestra con el siguiente contraejemplo. Dadas las fórmulas:

$$\{x = 0\}\ x := x + 2\ \{x = 2\}\ y\ \{x = 0\}\ y := x\ \{y = 0\}$$

no se cumple:

$$\{x = 0 \wedge x = 0\}\ [x := x + 2\ ||\ y := x]\ \{x = 2 \wedge y = 0\}$$

sino:

$\{x = 0 \wedge x = 0\}\ [x := x + 2 \ ||\ y := x]\ \{x = 2 \wedge (y = 0 \vee y = 2)\}$

porque la variable *y* también puede terminar con el valor 2 si la asignación *y := x* se ejecuta después de la asignación *x := x + 2*. La pérdida de la composicionalidad también se comprueba por el hecho de que ahora puede no ser inocuo reemplazar en un programa un proceso por otro semánticamente equivalente. Por ejemplo, reemplazando el proceso anterior *x := x + 2* por el proceso *x := x + 1 ; x := x + 1,* de las fórmulas:

$\{x = 0\}\ x := x + 1\ ;\ x := x + 1\ \{x = 2\}\ y\ \{x = 0\}\ y := x\ \{y = 0\}$

se obtiene la fórmula:

$\{x = 0 \wedge x = 0\}\ [x := x + 1\ ;\ x := x + 1\ ||\ y := x]\ \{x = 2 \wedge (y = 0 \vee y = 1 \vee y = 2)\}$

que como se observa, tiene una postcondición distinta a la de la fórmula obtenida con el proceso *x := x + 2* (la variable *y* ahora también puede terminar con el valor 1 si la asignación *y := x* se realiza entre las dos asignaciones *x := x + 1*).

En definitiva, el enfoque composicional que presentamos para los programas secuenciales no sirve para los programas paralelos, consecuencia de las interferencias entre los procesos. Una regla de prueba para la composición paralela debe contemplarlas.

La regla que describimos a continuación pertenece a S. Owicki y D. Gries. Si bien no es composicional, es una buena aproximación, proponiendo un esquema sistemático y estructurado para las pruebas como las reglas anteriores. Existen variantes composicionales; por ejemplo, la de C. Jones, más orientada al desarrollo sistemático de programas, y que en un sentido reformula las ideas de S. Owicki y D. Gries, modificando la manera de especificar los programas (la comentamos brevemente en una nota adicional). Optamos por la regla no composicional para mantener el estilo de las descripciones anteriores, y también porque es más accesible.

La regla se define en términos de *proof outlines* (de correctitud parcial) de los procesos, imprescindibles en este contexto para tratar globalmente el comportamiento de los programas. Plantea, para probar $\{p\}\ [\ ||_{i = 1,n}\ S_i\ ]\ \{q\}$, encontrar para todos los procesos de la composición paralela, *proof outlines* $\{p_i\}\ S_i^*\ \{q_i\}$ que satisfagan las implicaciones $p \longrightarrow \bigwedge_{i = 1,n} p_i$ y $\bigwedge_{i = 1,n} q_i \longrightarrow q$ y cumplan la siguiente condición:

- para todo par de *proof outlines* $\{p_i\}\ S_i^*\ \{q_i\}$ y $\{p_j\}\ S_j^*\ \{q_j\}$,
- para todo predicado r de $\{p_i\}\ S_i^*\ \{q_i\}$ no interno de un *await* (conocido como *predicado normal*),
- y para toda instrucción T de $\{p_j\}\ S_j^*\ \{q_j\}$ que sea una asignación no interna de un *await* (conocida como *asignación normal*) o un *await*:

$\{r \wedge pre(T)\}\ T\ \{r\}$

siendo pre(T) la precondición de T en $\{p_j\}\ S_j^*\ \{q_j\}$. Las *proof outlines* que cumplen dicha condición se conocen como *libres de interferencia*. Efectivamente, notar que en ellas todo predicado normal es invariante en la posición que ocupa, cualquiera sea la asignación normal o el *await* que se ejecute (los predicados no normales no cuentan porque son invisibles globalmente dada la atomicidad de los *await*, y sólo se consideran las instrucciones que pueden modificar variables). En estas condiciones, claramente a partir de un estado que satisface la conjunción de las precondiciones de los procesos, si la composición paralela termina lo hace en un estado que satisface la conjunción de las postcondiciones de los mismos.

Con estas definiciones formulamos de la siguiente manera la regla que adoptamos para verificar la correctitud parcial de programas del lenguaje de variables compartidas:

22. *Regla de la composición paralela* (PAR)

$$\frac{\{p_i\}\ S_i^*\ \{q_i\},\ i = 1, \ldots, n, \text{ son } \textit{proof outlines} \text{ libres de interferencia}, \quad p \longrightarrow \bigwedge_{i=1,n} p_i,\ \bigwedge_{i=1,n} q_i \longrightarrow q}{\{p\}\ [\ ||_{\ i=1,n}\ S_i\ ]\ \{q\}}$$

PAR es una regla especial, una metarregla, porque además de fórmulas de correctitud tiene *proof outlines* como premisas. Su uso implica un esquema de pruebas en dos etapas, una etapa de *pruebas locales*, para verificar mediante *proof outlines* la correctitud parcial de los procesos considerados aisladamente, y una etapa de *prueba global*, para validar la libertad de interferencia de las *proof outlines*.

La prueba global suele ser muy trabajosa, consecuencia de la manera no estructurada con la que se pueden hacer referencias y actualizaciones a las variables compartidas: la cantidad de validaciones con n procesos de tamaño k es del orden de k^n (de todos modos, en la práctica muchas validaciones son triviales).

Mostramos en lo que sigue algunos ejemplos de pruebas con el esquema planteado.

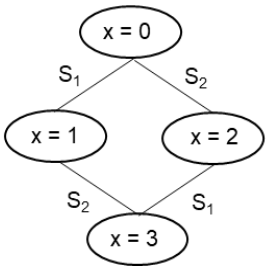
Ejemplo 14. Prueba de correctitud parcial de un programa de suma

Vamos a probar $\{x = 0\}\ [S_1 :: x := x + 1\ ||\ S_2 :: x := x + 2]\ \{x = 3\}$.

Las *proof outlines* naturales de los procesos S_1 y S_2 considerados aisladamente, respectivamente $\{x = 0\}\ S_1 :: x := x + 1\ \{x = 1\}$ y $\{x = 0\}\ S_2 :: x := x + 2\ \{x = 2\}$, no son libres de interferencia (por ejemplo, al final de S_1 también puede cumplirse x = 3).

Proponemos las siguientes *proof outlines*, justificadas por el diagrama que las acompaña, que representa los estados iniciales, intermedios y final del programa y cómo se obtienen:

$\{x = 0 \vee x = 2\}$ $\{x = 0 \vee x = 1\}$
$[S_1 :: x := x + 1 \;||\; S_2 :: x := x + 2]$
$\{x = 1 \vee x = 3\}$ $\{x = 2 \vee x = 3\}$



Claramente, las *proof outlines* son correctas. También libres de interferencia, las siguientes fórmulas, definidas de acuerdo a la regla PAR, son verdaderas:

$\{(x = 0 \vee x = 2) \wedge (x = 0 \vee x = 1)\}\ x := x + 2\ \{x = 0 \vee x = 2\}$
$\{(x = 1 \vee x = 3) \wedge (x = 0 \vee x = 1)\}\ x := x + 2\ \{x = 1 \vee x = 3\}$
$\{(x = 0 \vee x = 1) \wedge (x = 0 \vee x = 2)\}\ x := x + 1\ \{x = 0 \vee x = 1\}$
$\{(x = 2 \vee x = 3) \wedge (x = 0 \vee x = 2)\}\ x := x + 1\ \{x = 2 \vee x = 3\}$

Además, se cumple que la precondición del programa implica la conjunción de las precondiciones de los procesos, y que la conjunción de las postcondiciones de los procesos implica la postcondición del programa:

$x = 0 \longrightarrow ((x = 0 \vee x = 2) \wedge (x = 0 \vee x = 1))$
$((x = 1 \vee x = 3) \wedge (x = 2 \vee x = 3)) \longrightarrow x = 3$

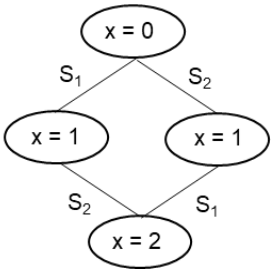
lo que completa la prueba. Se observa que para obtener *proof outlines* libres de interferencia hay que debilitar los predicados utilizados en las pruebas locales. Esta técnica no siempre deriva en una prueba exitosa, como mostramos en el siguiente ejemplo.

Ejemplo 15. Prueba de correctitud parcial de otro programa de suma

Vamos a probar $\{x = 0\}\ [S_1 :: x := x + 1 \;||\; S_2 :: x := x + 1]\ \{x = 2\}$.

Las *proof outlines* naturales $\{x = 0\}\ S_1 :: x := x + 1\ \{x = 1\}$ y $\{x = 0\}\ S_2 :: x := x + 1\ \{x = 1\}$ no son libres de interferencia (al final de un proceso también se puede cumplir x = 2). Siguiendo el mecanismo del ejercicio anterior, proponemos las siguientes *proof outlines*:

$\{x = 0 \vee x = 1\}$ $\{x = 0 \vee x = 1\}$
$[S_1 :: x := x + 1 \;||\; S_2 :: x := x + 1]$
$\{x = 1 \vee x = 2\}$ $\{x = 1 \vee x = 2\}$



pero tampoco son libres de interferencia. Notar por ejemplo que la fórmula:

$$\{(x = 0 \vee x = 1) \wedge (x = 0 \vee x = 1)\}\ x := x + 1\ \{x = 0 \vee x = 1\}$$

no es verdadera. Se puede demostrar formalmente que no existen *proof outlines* libres de interferencia en este caso. Intuitivamente, lo que sucede es que expresando el estado intermedio solamente con x = 1 no alcanza para registrar la *historia* del *interleaving* llevado a cabo, es decir, si el proceso que ejecutó la primera asignación *x := x + 1* fue S_1 o S_2.

Para resolver estos casos inevitables de incompletitud, el esquema de pruebas contempla, antes de la validación de libertad de interferencia, la ampliación del programa considerado con *variables auxiliares*. La idea es agregarlas a los procesos, sin alterar el cómputo básico del programa, con el objetivo de reforzar los predicados de sus *proof outlines*.

Primero completamos la prueba de este ejemplo con esta idea, y luego formalizamos la extensión necesaria en la axiomática. Proponemos la siguiente ampliación del programa (resaltamos las partes que se refieren a las variables auxiliares para distinguirlas del programa original):

y := 0 ; **z := 0** ;

$\{(x = 0 \wedge \mathbf{y = 0} \wedge \mathbf{z = 0}) \vee$		$\{(x = 0 \wedge \mathbf{y = 0} \wedge \mathbf{z = 0}) \vee$
$(x = 1 \wedge \mathbf{y = 0} \wedge \mathbf{z = 1})\}$		$(x = 1 \wedge \mathbf{y = 1} \wedge \mathbf{z = 0})\}$
[S_1 :: await true $\rightarrow$ x := x + 1 ; **y := 1** end	||	S_2 :: await true $\rightarrow$ x := x + 1 ; **z := 1** end]
$\{(x = 1 \wedge \mathbf{y = 1} \wedge \mathbf{z = 0}) \vee$		$\{(x = 1 \wedge \mathbf{y = 0} \wedge \mathbf{z = 1}) \vee$
$(x = 2 \wedge \mathbf{y = 1} \wedge \mathbf{z = 1})\}$		$(x = 2 \wedge \mathbf{y = 1} \wedge \mathbf{z = 1})\}$

La idea es la siguiente. Se agregan la variable *y* al proceso S_1 y la variable z al proceso S_2, inicializadas en 0, y se las cambia a 1 después de que el proceso respectivo ejecuta su asignación *x := x + 1* (los *await* son necesarios porque las dos asignaciones de cada proceso deben ejecutarse atómicamente). Esto refuerza los predicados y permite obtener *proof outlines* libres de interferencia (se comprueba fácilmente, lo mismo que la postcondición x = 2 a partir de la precondición x = 0). Y dado que el agregado de las variables auxiliares y sus asignaciones no alteran el cómputo básico, las *proof outlines* también sirven para probar el programa original.

La regla de prueba que formaliza el uso de variables auxiliares se formula de la siguiente manera:

23. *Regla de las variables auxiliares* (AUX)

$$\frac{\{p\}\ S\ \{q\}}{\{p\}\ S_{|A}\ \{q\}}$$

tal que:

- S es un programa ampliado con variables auxiliares de un conjunto A, y $S_{|A}$ es el programa original.
- Las variables auxiliares sólo aparecen en asignaciones de S, pero nunca en la parte derecha de una asignación a una variable original (si no, se alteraría el cómputo básico).
- La postcondición q no incluye variables auxiliares.

Con estas definiciones, queda claro que de {p} S {q} se deriva {p} $S_{|A}$ {q}.

Completamos la serie de ejemplos de esta sección mostrando la prueba de correctitud parcial del programa que calcula el factorial presentado en la sección anterior.

Ejemplo 16. Prueba de correctitud parcial del programa que calcula el factorial

Vamos a probar:

$\{c_1 \wedge c_2 \wedge i = 1 \wedge j = N \wedge n = N \wedge N \geq 1\}$

[S_1 :: while c_1 do		S_2 :: while c_2 do
await true $\rightarrow$ if $i + 1 < j$		await true $\rightarrow$ if $j - 1 > i$
then $i := i + 1$; $n := n \,.\, i$	||	then $j := j - 1$; $n := n \,.\, j$
else c_1 := false fi end		else c_2 := false fi end
od		od]

$\{n = N!\}$

Proponemos como invariantes de los *while* los siguientes predicados:

$p_1 = (i \leq j \wedge (\neg c_1 \rightarrow i + 1 = j) \wedge n \,.\, (i + 1) \,.\, \ldots \,.\, (j - 1) = N!)$, para el *while* del proceso S_1

$p_2 = (i \leq j \wedge (\neg c_2 \rightarrow j - 1 = i) \wedge n \,.\, (i + 1) \,.\, \ldots \,.\, (j - 1) = N!)$, para el *while* del proceso S_2

y como *proof outlines* de los procesos:

$\{i \leq j \wedge (\neg c_1 \rightarrow i + 1 = j) \wedge$		$\{i \leq j \wedge (\neg c_2 \rightarrow j - 1 = i) \wedge$
$n \,.\, (i + 1) \,.\, \ldots \,.\, (j - 1) = N!\}$		$n \,.\, (i + 1) \,.\, \ldots \,.\, (j - 1) = N!\}$
[S_1 :: while c_1 do		S_2 :: while c_2 do
$\{c_1 \wedge i \leq j \wedge n \,.\, (i + 1) \,.\, \ldots \,.\, (j - 1) = N!\}$	||	$\{c_2 \wedge i \leq j \wedge n \,.\, (i + 1) \,.\, \ldots \,.\, (j - 1) = N!\}$
await true $\rightarrow$ if $i + 1 < j$		await true $\rightarrow$ if $j - 1 > i$
then $i := i + 1$; $n := n \,.\, \mathbf{i}$		then $j := j - 1$; $n := n \,.\, j$
else c_1 := false fi end		else c_2 := false fi end
$\{i \leq j \wedge (\neg c_1 \rightarrow i + 1 = j) \wedge$		$\{i \leq j \wedge (\neg c_2 \rightarrow j - 1 = i) \wedge$
$n \,.\, (i + 1) \,.\, \ldots \,.\, (j - 1) = N!\}$		$n \,.\, (i + 1) \,.\, \ldots \,.\, (j - 1) = N!\}$
od		od]
$\{i \leq j \wedge i + 1 = j \wedge n \,.\, (i + 1) \,.\, \ldots \,.\, (j - 1) = N!\}$		$\{i \leq j \wedge j - 1 = i \wedge n \,.\, (i + 1) \,.\, \ldots \,.\, (j - 1) = N!\}$

Se prueba fácilmente que las *proof outlines* son correctas (no incluimos los predicados no normales) y libres de interferencia (como en los dos ejemplos previos, la validación se simplifica porque sólo hay que considerar una instrucción por proceso, el *await*). También se prueba fácilmente que se cumplen las implicaciones requeridas entre la precondición y la postcondición del programa y las conjunciones de las precondiciones y las postcondiciones de los procesos.

Axiomática para las pruebas de no divergencia

La descripción del esquema de verificación en dos etapas introducido en la sección anterior sirve como base para el resto de las secciones del capítulo. En efecto, el enfoque axiomático que adoptamos para el paradigma concurrente utiliza dicho esquema para las pruebas de todas las propiedades, tanto de programas paralelos como de programas distribuidos. En una primera etapa de pruebas locales se prueban los procesos aisladamente por medio de *proof outlines*, y en una segunda etapa, la prueba global, se validan las *proof outlines* obtenidas en la primera etapa, con un criterio que depende de la propiedad considerada y el lenguaje utilizado.

Para el caso de la verificación de no divergencia de programas del lenguaje de variables compartidas, la axiomática cuenta para las *proof outlines* (de no divergencia) de las pruebas locales con la regla REP* introducida en el capítulo 2, asociada a los *while* (los *while* constituyen la única fuente posible de computaciones infinitas en este contexto). En cuanto a la prueba global, utiliza una regla de composición paralela que establece una validación de las *proof outlines* con más requisitos que la de la prueba de correctitud parcial, consecuencia del uso no sólo de invariantes sino también de variantes para los *while*. Concretamente, chequea también que el valor del variante de un *while* en un proceso no sea impactado por otro proceso, de modo tal que provoque la divergencia del programa. Este segundo requisito se explica porque, tal como las definimos, las *proof outlines* no registran las pruebas de no divergencia sino sólo las definiciones de los variantes.

Precisando, la nueva regla plantea, para probar $\langle p \rangle$ [$||_{i = 1,n} S_i$] $\langle q \rangle$, encontrar para todos los procesos *proof outlines* $\langle p_i \rangle$ S_i^{**} $\langle q_i \rangle$ que cumplan las implicaciones $p \longrightarrow \bigwedge_{i = 1,n} p_i$ y $\bigwedge_{i = 1,n} q_i \longrightarrow q$, que sean libres de interferencia, y que satisfagan las siguientes dos condiciones adicionales:

- para todo par de *proof outlines* $\langle p_i \rangle$ S_i^{**} $\langle q_i \rangle$ y $\langle p_j \rangle$ S_j^{**} $\langle q_j \rangle$,
- para todo variante t de $\langle p_i \rangle$ S_i^{**} $\langle q_i \rangle$,
- y para toda asignación normal o await T de $\langle p_j \rangle$ S_j^{**} $\langle q_j \rangle$:

1. $\langle t = Z \wedge pre(T) \rangle$ T $\langle t \leq Z \rangle$, siendo pre(T) la precondición de T en $\langle p_j \rangle$ S_j^{**} $\langle q_j \rangle$.
2. Toda *secuencia sintácticamente posible* de asignaciones normales y *await* dentro de un *while* decrementa el valor de su variante asociado.

Las secuencias sintácticamente posibles de instrucciones de un fragmento de programa U se obtienen teniendo en cuenta solamente la sintaxis del fragmento. Por ejemplo, las de $U_1 ; U_2$ se

obtienen secuencializando las de U_1 con las de U_2, y las de *if B then* U_1 *else* U_2 *fi* contienen las de U_1 y las de U_2. Notar que la condición (1) permite que un proceso acelere la terminación de otro. La necesidad de la condición (2) se aclara más adelante por medio de un ejemplo. Las *proof outlines* libres de interferencia que cumplen las dos condiciones adicionales se conocen como *fuertemente libres de interferencia*. Con estas definiciones enunciamos la regla que adoptamos para verificar la no divergencia de programas del lenguaje de variables compartidas:

24. *Regla de la no divergencia paralela* (PAR*)

$\langle p_i \rangle\ S_i^{**}\ \langle q_i \rangle$, i = 1, …, n, son *proof outlines* fuertemente libres de interferencia,
$p \longrightarrow \bigwedge_{i=1,n} p_i$, $\bigwedge_{i=1,n} q_i \longrightarrow q$

$$\langle p \rangle\ [\ ||_{i=1,n}\ S_i\ ]\ \langle q \rangle$$

La regla PAR* tiene la misma característica que la regla PAR definida para la prueba de correctitud parcial: es una metarregla, porque incluye *proof outlines* como premisas. Al igual que las reglas que describimos para los programas secuenciales, permite probar conjuntamente la correctitud parcial y la no divergencia (la *correctitud total débil* tal como la definen algunos autores). No cubre la prueba de ausencia de *deadlock*, en su aplicación se asume que dicha propiedad se verifica aparte. Y como en el caso de los programas secuenciales no determinísticos, se puede relajar si se asume alguna hipótesis de *fairness*. Por ejemplo, en el caso del programa:

S_{loop} :: [while x = 1 do skip od || x := 0]

no sería necesario probar la no divergencia del primer proceso localmente, es decir considerándolo aisladamente, contando con una hipótesis de *fairness* que asegure que la asignación *x := 0* se va a ejecutar alguna vez.

Finalizamos la sección mostrando el uso de la regla con un par de ejemplos.

Ejemplo 17. Prueba de no divergencia de un programa de resta

Vamos a probar:

$\langle x > 0 \wedge par(x) \rangle\ [S_1 :: \text{while } x > 2 \text{ do } x := x - 2 \text{ od } ||\ S_2 :: x := x - 1]\ \langle x = 1 \rangle$

siendo el predicado par(x) verdadero sii x es par. Por lo simple que es el programa, verificaremos directamente su correctitud total (no hay posibilidad de *deadlock* por la inexistencia de instrucciones *await*).

Proponemos para la prueba las siguientes *proof outlines* (el predicado impar(x) es verdadero sii x es impar):

⟨inv: x > 0, var: x⟩ ⟨par(x)⟩
[S_1 :: while x > 2 do || S_2 :: x := x – 1]
⟨x > 2⟩ ⟨impar(x)⟩
x := x – 2
⟨x > 0⟩
od
⟨x = 1 ∨ x = 2⟩

Se comprueba fácilmente que las *proof outlines* son correctas y libres de interferencia, y que se cumplen las implicaciones requeridas entre la especificación del programa y las especificaciones de los procesos. Las *proof outlines* también cumplen las dos condiciones adicionales requeridas para ser fuertemente libres de interferencia:

1. La asignación *x := x – 1* del proceso S_2 no aumenta el valor del variante t = x.
2. La única secuencia sintácticamente posible de asignaciones normales e instrucciones *await* del *while* del proceso S_1 es la asignación *x := x – 2*, la cual decrementa el valor del variante.

Ejemplo 18. Prueba de no divergencia del programa del productor-consumidor

Vamos a considerar solamente la composición concurrente del programa referido, con la precondición in = 0 ∧ out = 0 ∧ i = 1 ∧ j = 1 ∧ b > 0 ∧ n > 0, obtenida de las asignaciones iniciales *in := 0*, *out := 0*, *i := 1* y *j := 1* (b es el tamaño del arreglo *buffer* y n es el tamaño de los arreglos *a* y b). Proponemos las siguientes *proof outlines*:

⟨inv: 1 ≤ i ≤ n + 1, var: n + 1 – i⟩
[prod :: while i ≤ n do ⟨1 ≤ i ≤ n⟩
x := a[i] ; ⟨1 ≤ i ≤ n⟩
await in – out < b →
skip end; ⟨1 ≤ i ≤ n⟩
buffer[in mod b] := x ; ⟨1 ≤ i ≤ n⟩
in := in + 1 ; ⟨1 ≤ i ≤ n⟩
i := i + 1 ⟨1 ≤ i ≤ n + 1⟩
od
⟨true⟩

||

⟨inv: 1 ≤ j ≤ n + 1, var: n + 1 – j⟩
cons :: while j ≤ n do ⟨1 ≤ j ≤ n⟩
await in – out > 0 →
skip end ; ⟨1 ≤ j ≤ n⟩
y := buffer[out mod b] ; ⟨1 ≤ j ≤ n⟩
out := out + 1 ; ⟨1 ≤ j ≤ n⟩
b[j] := y ; ⟨1 ≤ j ≤ n⟩
j := j + 1 ⟨1 ≤ j ≤ n + 1⟩
od]
⟨true⟩

También en este caso es fácil comprobar la correctitud de la prueba desarrollada. La precondición implica la conjunción de los invariantes, las *proof outlines* son correctas y libres de

interferencia, ninguna instrucción de un proceso aumenta el variante del *while* del otro proceso, y cada *while* tiene una única secuencia sintácticamente posible de instrucciones, la cual decrementa el variante respectivo.

En los dos ejemplos anteriores no puede apreciarse bien el requerimiento de la regla de prueba PAR* de que todas las secuencias sintácticamente posibles de asignaciones normales y *await* de los *while* decrementen sus variantes. Para aclarar este punto presentamos el siguiente programa:

```
[S1 :: while x > 0 do              S2 :: while x > 0 do
       y := 0;                            y := 1;
       if y = 0 then x := 0    ||         if y = 1 then x := 0
       else y := 0 fi                     else y := 1 fi
       od                                 od]
```

Considerados individualmente, los procesos S_1 y S_2 terminan, lo que se puede probar fácilmente, por ejemplo, con *proof outlines* de no divergencia con todos predicados *true*, incluyendo los invariantes de los *while*, y con los dos variantes definidos con el valor max(x, 0), es decir el máximo entre x y 0. De esta forma, las *proof outlines* resultan libres de interferencia, y además cumplen con la primera condición adicional de la regla PAR*: las asignaciones de un proceso no aumentan el valor del variante del *while* del otro proceso. Sin embargo, el programa puede divergir, si ejecuta repetidamente la secuencia formada por la primera asignación *y := 0* de S_1, la primera asignación *y := 1* de S_2, la segunda asignación y := 0 de S_1 y la segunda asignación y := 1 de S_2. Sucede en este caso que la segunda condición adicional de la regla PAR* no se cumple: la secuencia *y := 0 ; y := 0* de S_1 y la secuencia *y := 1 ; y := 1* de S_2 son sintácticamente posibles y no decrementan el variante definido.

Axiomática para las pruebas de ausencia de *deadlock*

La regla para verificar la ausencia de *deadlock*, única falla que contemplamos en los programas paralelos descriptos, es otro ejemplo de metarregla. En verdad, esta característica se repite en todas las reglas siguientes que se aplican sobre la composición concurrente de un programa. Otra característica común de esta clase de reglas es que plantean arrancar de un conjunto de *proof outlines* de correctitud parcial libres de interferencia, el más adecuado según la propiedad que se pretenda verificar.

La particularidad en el caso de la propiedad de ausencia de *deadlock* es que no puede analizarse localmente, sino que su tratamiento requiere considerar directamente el programa completo. Adicionalmente, por limitaciones de la lógica de predicados, para su prueba se debe recurrir a configuraciones expresadas en términos de los puntos de control de los procesos. Específicamente, la prueba consiste en definir todas las configuraciones que representan

potenciales casos de *deadlock*, y comprobar que ninguno es posible. Considerando un programa $[\|_{i = 1,n} S_i]$ y una precondición p, la regla de prueba de ausencia de *deadlock* se formula de la siguiente manera:

25. *Regla de la ausencia de deadlock en programas paralelos* (*DEADLOCK*)

1. Obtener *proof outlines* $\{p_i\}\ S_i^*\ \{q_i\}$ de los procesos S_i del programa, libres de interferencia, y que cumplan la implicación $p \longrightarrow \bigwedge_{i = 1,n} p_i$.
2. Definir sintácticamente en las *proof outlines* todas las configuraciones $C = (l_1, \ldots, l_n)$ que representan casos posibles de *deadlock*: cada l_i de una configuración es una etiqueta asociada a una instrucción *await* del proceso S_i o al final del mismo, con la salvedad de que en toda configuración debe haber al menos una etiqueta asociada a un *await*, porque si no no representaría una situación de *deadlock*.
3. Caracterizar semánticamente las configuraciones definidas en el paso (2), mediante predicados, denominados *imágenes semánticas*. La imagen semántica M de una configuración C es una conjunción de predicados $r_1 \wedge \ldots \wedge r_n$ que tiene la siguiente forma: Si la etiqueta l_i de C está asociada a una instrucción *T :: await B → T´ end* del proceso S_i, entonces r_i es $pre(T) \wedge \neg B$, siendo pre(T) la precondición de T en la *proof outline* de S_i. Y si la etiqueta l_i de C está asociada al final de S_i, entonces r_i es q_i.
4. Probar que todas las imágenes semánticas son falsas, lo que significa que no existe ningún caso de *deadlock* en el programa.

El siguiente ejemplo sobre un esquema de programa con tres procesos aclara los pasos anteriores. Supongamos que luego del paso (2) llegamos a:

$\{p_1\}\ [S_1 :: \ldots \{pre(T)\}\ l_{11} \Rightarrow T :: \text{await } B_T \rightarrow T' \text{ end} \ldots l_{12} \Rightarrow \{q_1\}$
$\quad \|$
$\{p_2\}\ S_2 :: \ldots l_2 \Rightarrow \{q_2\}$
$\quad \|$
$\{p_3\}\ S_3 :: \ldots \{pre(U)\}\ l_{31} \Rightarrow U :: \text{await } B_U \rightarrow U' \text{ end} \ldots]\ l_{32} \Rightarrow \{q_3\}$

es decir que las configuraciones que representan los casos posibles de *deadlock* son:

$C_1 = (l_{11}, l_2, l_{31})$, $C_2 = (l_{12}, l_2, l_{31})$ y $C_3 = (l_{11}, l_2, l_{32})$

En el paso (3) definimos las imágenes semánticas de las configuraciones:

$M_1 = (pre(T) \wedge \neg B_T) \wedge q_2 \wedge (pre(U) \wedge \neg B_U)$
$M_2 = q_1 \wedge q_2 \wedge (pre(U) \wedge \neg B_U)$
$M_3 = (pre(T) \wedge \neg B_T) \wedge q_2 \wedge q_3$

Finalmente, en el paso (4) verificamos que todas las imágenes semánticas sean falsas. En lo que sigue ejemplificamos el uso de la regla DEADLOCK con un programa concreto. Volvemos una vez más al programa del productor-consumidor.

Ejemplo 19. Prueba de ausencia de *deadlock* en el programa del productor-consumidor

Partimos como antes de la precondición $in = 0 \wedge out = 0 \wedge i = 1 \wedge j = 1 \wedge b > 0 \wedge n > 0$. Proponemos la siguiente *proof outline* para el proceso del productor, en el que ya insertamos las etiquetas apropiadas:

{inv: $0 \leq in - out \leq b \wedge 1 \leq i \leq n + 1 \wedge 1 \leq j \leq n + 1 \wedge b > 0 \wedge i = in + 1$}
while $i \leq n$ do
{$0 \leq in - out \leq b \wedge 1 \leq i \leq n \wedge 1 \leq j \leq n + 1 \wedge b > 0 \wedge i = in + 1$}
x := A[i] ;
{$0 \leq in - out \leq b \wedge 1 \leq i \leq n \wedge 1 \leq j \leq n + 1 \wedge b > 0 \wedge i = in + 1$}
$I_{p1} \Rightarrow$ await $in - out < b \rightarrow$ skip end ;
{$0 \leq in - out < b \wedge 1 \leq i \leq n \wedge 1 \leq j \leq n + 1 \wedge b > 0 \wedge i = in + 1$}
buffer[in mod b] := x ;
{$0 \leq in - out < b \wedge 1 \leq i \leq n \wedge 1 \leq j \leq n + 1 \wedge b > 0 \wedge i = in + 1$}
in := in + 1 ;
{$0 \leq in - out \leq b \wedge 1 \leq i \leq n \wedge 1 \leq j \leq n + 1 \wedge b > 0 \wedge i = in$}
i := i + 1
{$0 \leq in - out \leq b \wedge 1 \leq i \leq n + 1 \wedge 1 \leq j \leq n + 1 \wedge b > 0 \wedge i = in + 1$}
od
$I_{p2} \Rightarrow$ {$0 \leq in - out \leq b \wedge i = n + 1 \wedge 1 \leq j \leq n + 1 \wedge b > 0 \wedge i = in + 1$}

De una manera similar se puede plantear la *proof outline* etiquetada para el proceso del consumidor, libre de interferencia con respecto a la del proceso del productor. No la desarrollamos, sólo formulamos los predicados que interesan para la prueba, y asociamos la etiqueta I_{c1} a la instrucción *await* y la etiqueta I_{c2} al final del proceso. El invariante del *while* es:

$0 \leq in - out \leq b \wedge 1 \leq i \leq n + 1 \wedge 1 \leq j \leq n + 1 \wedge b > 0 \wedge j = out + 1$,

la precondición de la instrucción *await in – out > 0 → skip end* es:

$0 \leq in - out \leq b \wedge 1 \leq i \leq n + 1 \wedge 1 \leq j \leq n \wedge b > 0 \wedge j = out + 1$,

y la postcondición del proceso es:

$$0 \leq in - out \leq b \wedge 1 \leq i \leq n + 1 \wedge j = n + 1 \wedge b > 0 \wedge j = out + 1.$$

Claramente, la precondición del programa implica la conjunción de los invariantes de los *while*. Hay tres casos posibles de *deadlock*, representados por $C_1 = (I_{p1}, I_{c1})$, $C_2 = (I_{p2}, I_{c1})$ y $C_3 = (I_{p1}, I_{c2})$. Veamos a qué situaciones se refieren y si sus imágenes semánticas son falsas:

- C_1 representa el caso en el que el productor y el consumidor están bloqueados en sus *await*. De la conjunción de las precondiciones de los *await* y las negaciones de sus expresiones booleanas se obtiene $b > 0 \wedge in - out \geq b \wedge in - out \leq 0$, que implica $b > 0 \wedge b \leq 0$ (falso).
- C_2 representa el caso en el que el productor termina y el consumidor está bloqueado en su *await*. De la postcondición del productor se obtiene $in = n$. De la precondición del *await* del consumidor se obtiene $0 \leq in - out \leq b \wedge 1 \leq j \leq n \wedge j = out + 1$. Y de la negación de la expresión booleana de dicho *await* se obtiene $in - out \leq 0$. A partir de dichos predicados llegamos primero a $0 \leq in - out \leq 0$, luego a $in = out = n$, y finalmente a $j = out + 1 = n + 1 \leq n$ (falso).
- El último caso posible de *deadlock*, representado por la configuración C_3, corresponde a cuando el productor está bloqueado en su *await* y el consumidor termina. También la imagen semántica resulta falsa, lo que se prueba razonando como en el caso anterior.

Por lo tanto, se cumple la ausencia de *deadlock*. Notar que las *proof outlines* utilizadas son distintas de las que consideramos para la prueba de no divergencia del programa.

Lenguaje de recursos de variables

Por medio del lenguaje de variables compartidas, utilizado previamente, con una instrucción de sincronización muy elemental, el *await*, con la que S. Owicki y D. Gries publican originalmente su axiomática, aprovechamos a presentar aspectos salientes de la verificación de programas concurrentes. Entre otros: *proof outlines* de correctitud parcial para probar todas las propiedades; pruebas en dos etapas como acercamiento a un esquema composicional, con chequeo de consistencia de las *proof outlines*; variables auxiliares; y configuraciones expresadas en términos de puntos de control de los procesos.

Con esta base introducimos en esta sección una segunda clase de programas paralelos, con una instrucción de sincronización más estructurada, sin la permisividad irrestricta sobre referencias y actualizaciones a las variables compartidas observada antes, lo que facilita la construcción y verificación de programas. La nueva instrucción fuerza la exclusión mutua entre las secciones críticas especificadas. Además, las variables compartidas se disponen en conjuntos denominados *recursos*, sobre los que los procesos obtienen acceso exclusivo. Un programa de este segundo lenguaje (*lenguaje de recursos de variables*) tiene la siguiente forma:

S :: resource r_1 (lista-var$_1$) ; … ; resource r_m (lista-var$_m$); S_0 ; [S_1 || … || S_n]

Los componentes r_1, …, r_m son recursos de variables, conjuntos disjuntos de variables definidos mediante las listas *lista-var$_1$*, …, *lista-var$_m$*. El subprograma S_0 tiene las mismas características que en el lenguaje anterior. Los procesos S_1, …, S_n también, salvo que cuentan con una instrucción de sincronización distinta, el *with*, que provee acceso exclusivo a los recursos. La sintaxis del *with*, también conocido como *sección crítica condicional*, es la siguiente:

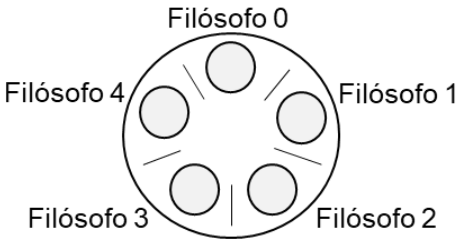
with r_j when B do S endwith

r_j es un recurso, B una expresión booleana, y S un subprograma sin *with* y que siempre termina (tiene las mismas restricciones del *await* para simplificar la presentación). Toda variable de un recurso referida en un proceso debe estar dentro de un *with* del proceso, y toda variable compartida modificable debe estar definida en un recurso.

Cuando un proceso S_i ejecuta una instrucción *with r_j when B do S endwith*, si r_j está libre (es decir, si no está ocupado por otro proceso), y B es verdadera, entonces S_i puede ocupar el recurso, utilizar sus variables y progresar en su ejecución. Ningún proceso tiene acceso a un recurso ocupado por otro. La liberación de un recurso se produce cuando el proceso que lo ocupa termina de ejecutar el *with* correspondiente. Si dos o más procesos compiten por un recurso, su ocupación se decide no determinísticamente, no hay ningún condicionamiento al respecto.

Como en la clase de programas anterior, el uso indebido de la instrucción de sincronización puede causar *deadlock*, y el *while* sigue siendo la única fuente posible de divergencia. Notar que ahora las secciones críticas se pueden identificar sintácticamente, lo que hace que las pruebas de correctitud resulten más estructuradas.

Como ejemplo de programa, presentamos a continuación un esquema muy simple que modeliza el clásico problema de la cena de los filósofos, propuesto por E. Dijkstra para plantear una típica situación de sincronización en los sistemas operativos. El problema considera cinco filósofos sentados alrededor de una mesa, en la que hay cinco platos con espagueti y cinco tenedores, dispuestos como lo muestra la siguiente figura:



Los filósofos alternativamente comen o piensan. Para comer, un filósofo debe utilizar dos tenedores, el de su derecha y el de su izquierda, por lo que dos filósofos vecinos no pueden comer al mismo tiempo. El esquema de implementación propuesto es el siguiente:

S_{fcp} :: resource forks(f[0:4]) ; f[0]:= 2 ; f[1] := 2 ; f[2] := 2 ; f[3] := 2 ; f[4] := 2 ; [|| $_{i = 0,4}$ F_i]

con:

F_i :: while true do
with forks when f[i] = 2 do f[i – 1] := f[i – 1] – 1 ; f[i + 1] := f[i + 1] – 1 endwith ;
… comer …
with forks when true do f[i – 1] := f[i – 1] + 1 ; f[i + 1] := f[i + 1] + 1 endwith ;
… pensar …
od

El proceso F_i representa al filósofo i, y el elemento f[i] del arreglo compartido f (por eso está en un recurso, el recurso *forks*) representa la cantidad de tenedores a los que el filósofo tiene acceso en todo momento (la aritmética sobre los índices de f es módulo 5). Las secciones correspondientes a cuando el filósofo come o piensa no modifican f. El primer *with* representa la espera del filósofo para conseguir los dos tenedores que le permitan comer, y si lo logra, la inhabilitación para que los tomen sus vecinos. Y el segundo *with* implementa la situación en la que el filósofo, habiendo terminado de comer y disponiéndose a pensar, libera los tenedores para permitir que los utilicen sus vecinos.

Axiomática para las pruebas de correctitud parcial

Las únicas dos reglas de prueba de correctitud parcial que cambian, con respecto a la axiomática para los programas del lenguaje de variables compartidas, son las relacionadas con la instrucción de sincronización y la composición paralela. Las nuevas reglas, en conjunto, determinan pruebas más simples y estructuradas que las anteriores, explotando la característica del lenguaje de programación, que fuerza a agrupar las variables compartidas modificables en recursos, invocados por *with*.

La idea central de esta variante axiomática es definir por cada recurso r un invariante I_r, expresado en términos de las variables del recurso, que debe valer toda vez que el recurso está libre, es decir, en el momento en que un proceso lo ocupa y en el momento en que es liberado. Entre ambas instancias el invariante no tiene por qué valer (los estados intermedios correspondientes a la ejecución del proceso que utiliza el recurso son invisibles para el resto de los procesos).

La regla correspondiente a la instrucción de sincronización *with* (WITH) se formula de la siguiente manera (de ahora en más asumimos un solo recurso, para simplificar la presentación):

26. *Regla del with* (WITH)

$$\frac{\{I_r \wedge p \wedge B\}\ S\ \{I_r \wedge q\}}{\{p\}\ \text{with r when B do S endwith}\ \{q\}}$$

tal que p y q no tienen variables compartidas modificables e I_r se define con las variables de r. Como en el caso del *await*, la regla WITH tiene, por la forma de la instrucción, la misma estructura que la regla del condicional (COND). En este caso, la regla establece que pasar de la precondición p a la postcondición q mediante la ejecución del *with* es posible siempre que el subprograma S, a partir de $p \wedge B$, preserve el invariante I_r. Por lo dicho antes y que se refuerza más abajo, I_r no se traslada a la conclusión de la regla. La utilidad de WITH se aclara formulando la segunda regla novedosa de la axiomática, correspondiente a la composición paralela:

27. *Regla de la composición paralela con recursos* (RPAR)

$\{p_i\}\ S_i^*\ \{q_i\}$, i = 1, …, n, son *proof outlines* que utilizan I_r

$p \longrightarrow (I_r \wedge \bigwedge_{i=1,n} p_i),\ (I_r \wedge \bigwedge_{i=1,n} q_i) \longrightarrow q$

$$\frac{}{\{p\}\ [\ ||_{i=1,n}\ S_i\ ]\ \{q\}}$$

tal que los predicados de las *proof outlines* no incluyen variables compartidas modificables, las cuales figuran exclusivamente en el invariante I_r. Las *proof outlines* obtenidas en la primera etapa de pruebas locales utilizan el invariante del recurso a través de la aplicación de la regla WITH. Por cómo se conforman las *proof outlines*, la regla RPAR no requiere que se efectúe el chequeo de libertad de interferencia de las mismas en la etapa de prueba global. La conclusión de la regla se justifica por lo siguiente:

- La información de las variables del recurso r (variables compartidas modificables) se propaga desde la precondición hasta la postcondición de la composición paralela a través del invariante I_r. Al comienzo y al final I_r vale porque en ambas instancias el recurso está libre.
- La información del resto de las variables (variables no compartidas o compartidas de solo lectura) se propaga desde la precondición hasta la postcondición de la composición paralela a través de los predicados de las *proof outlines*.

Al igual que en los programas del lenguaje de variables compartidas, se sigue necesitando la regla AUX.

Como ejemplo de aplicación, probamos en lo que sigue la correctitud parcial de un esquema de programa de recursos de variables que implementa un *semáforo binario*, otro mecanismo de sincronización. El semáforo es un tipo de datos, creado por E. Dijkstra, con variables enteras y tres operaciones, que definidas en términos de la instrucción *await* tienen la siguiente forma:

Instrucción I(s): s := k, con $k \geq 0$ (inicialización del semáforo).

Instrucción P(s): await $s > 0 \rightarrow s := s - 1$ end.

Instrucción V(s): s := s + 1.

Su efecto es similar al del *await* (el *await* es más flexible y estructurado, pero como contrapartida menos eficiente, porque cuando un proceso lo ejecuta los otros procesos quedan suspendidos). Un semáforo es binario si sus variables pueden valer sólo 0 o 1.

Ejemplo 20. Prueba de correctitud parcial de un esquema de programa que implementa un semáforo binario

Vamos a probar:

```
resource sem(s) ; s = 1 ;
{s = 1}
[with sem when s = 1 do s := 0 endwith ;          with sem when s = 1 do s := 0 endwith ;
            … sección 1 …                  ||              … sección 2 …
with sem when true do s := 1 endwith              with sem when true do s := 1 endwith]
{s = 1}
```

La sección i entre los dos *with* del proceso i no modifica el valor de la variable s. Para la prueba tenemos que incluir variables auxiliares, porque de lo contrario estamos forzados a utilizar solamente predicados *true* en las *proof outlines* y el predicado $I_{sem} = 0 \leq s \leq 1$ como invariante del semáforo, que no conducen a la postcondición s = 1. Agregamos una variable *a* inicializada en 0 en el primer proceso, para indicar con el valor 1 o 0, respectivamente, si el proceso está ejecutando o no su sección entre los *with*, y lo mismo hacemos en el segundo proceso con una variable b. Proponemos las siguientes *proof outlines*:

```
{a = 0}                                          {b = 0}
[with semáforo when s = 1 do                     with semáforo when s = 1 do
     s := 0 ; a := 1 endwith ;   {a = 1}              s := 0 ; b := 1 endwith ;   {b = 1}
        … sección 1 …            {a = 1}     ||          … sección 2 …            {b = 1}
with semáforo when true do                       with semáforo when true do
     s := 1 ; a := 0 endwith                          s := 1 ; b := 0 endwith]
{a = 0}                                          {b = 0}
```

utilizando como invariante del semáforo:

$$I_{sem} = 0 \leq s \leq 1 \wedge ((a = 0 \wedge b = 0) \vee (a = 1 \wedge b = 0) \vee (a = 0 \wedge b = 1)) \wedge$$
$$((a = 0 \wedge b = 0) \rightarrow s = 1) \wedge (((a = 1 \wedge b = 0) \vee (a = 0 \wedge b = 1)) \rightarrow s = 0)$$

Al comienzo vale el invariante I_{sem}, implicado por la precondición $s = 1 \wedge a = 0 \wedge b = 0$.

La *proof outline* del primer proceso se obtiene de la siguiente manera. Aplicando la regla WITH, como se cumple:

$\{I_{sem} \wedge a = 0 \wedge s = 1\}$ s := 0 ; a := 1 $\{I_{sem} \wedge a = 1\}$

se deriva:

{a = 0} with semaforo when s = 1 do s := 0 ; a := 1 endwith {a = 1}

en lo que hace al primer *with*, y como se cumple:

$\{I_{sem} \wedge a = 1\}$ s := 1 ; a := 0 $\{I_{sem} \wedge a = 0\}$

se deriva:

{a = 1} with semaforo when true do s := 1 ; a := 0 endwith {a = 0}

en lo que hace al segundo *with*. La *proof outline* del segundo proceso se obtiene de manera similar, utilizando la variable b.

Finalmente, a la terminación del programa vale $I_{sem} \wedge a = 0 \wedge b = 0$, que implica la postcondición s = 1, válida también para el programa original, sin las variables *a* y b, por la aplicación de la regla AUX.

Axiomática para las pruebas de otras propiedades

Las reglas para probar no divergencia y ausencia de *deadlock* en los programas paralelos del lenguaje de recursos de variables tienen las mismas características que las descriptas para los programas paralelos del lenguaje de variables compartidas:

28. *Regla de la no divergencia paralela con recursos* (RPAR*)

La única diferencia de esta regla con la regla PAR* utilizada para los programas anteriores es que no requiere chequear la libertad de interferencia de las *proof outlines* de correctitud parcial, elaboradas como base para la prueba de no divergencia.

29. *Regla de la ausencia de deadlock con recursos* (RDEADLOCK)

En este caso la regla tiene dos diferencias con respecto a la regla DEADLOCK utilizada para los programas del lenguaje de variables compartidas. La primera diferencia es la misma que mencionamos antes, no requiere chequear la libertad de interferencia de las *proof outlines* de correctitud parcial elaboradas en las pruebas locales. La segunda diferencia consiste en que ahora las imágenes semánticas de las configuraciones correspondientes a los posibles casos de *deadlock*, definidas en términos de puntos de control asociados a instrucciones *with*, incluyen el invariante definido para el recurso utilizado, porque en una situación de *deadlock* el recurso no está ocupado.

Otras dos propiedades comúnmente consideradas en el paradigma concurrente son la *exclusión mutua* y la *ausencia de inanición*, pertenecientes, respectivamente, a las familias de propiedades *safety* y *liveness*. No las hemos tratado en los programas paralelos anteriores, así que aprovechamos esta sección para analizarlas al menos resumidamente. Sus pruebas tienen sentido sobre todo en programas que no terminan, usualmente denominados *reactivos*, cuya correctitud se evalúa en términos de su comportamiento continuo.

El lenguaje de recursos de variables garantiza la exclusión mutua entre las secciones críticas condicionales, por lo que la verificación de la propiedad tiene sentido únicamente para secciones de programa de mayor nivel de abstracción, sobre las que las interferencias indeseables sólo pueden ser evitadas algorítmicamente. La manera de probar la exclusión mutua es similar a cómo se prueba la ausencia de *deadlock*: a partir de *proof outlines* de correctitud parcial, con postcondiciones cualesquiera, hay que identificar sintácticamente todas las situaciones posibles de interferencia, y comprobar que las imágenes semánticas asociadas son falsas. La prueba es más laboriosa que la de ausencia de *deadlock*, porque ahora no sólo se deben considerar los *with* sino también todos los puntos de control internos de las secciones definidas.

Ejemplificamos a continuación la aplicación de la regla sobre el programa que implementa el problema de la cena de los filósofos.

Ejemplo 21. Prueba de exclusión mutua en el programa que implementa el problema de la cena de los filósofos

Vamos a probar que en la implementación elegida nunca se produce la situación en la que dos filósofos vecinos comen al mismo tiempo. Para ello, agregamos al programa un arreglo auxiliar *e*, de igual tamaño que el arreglo compartido f incluido en el recurso *forks*. El elemento e[i] se utiliza para indicar la residencia o no en la sección *comer* del proceso F_i que representa al filósofo i-ésimo (valor 1 o 0, respectivamente). El arreglo *e* se inicializa con todos sus elementos en 0 (recordar que todos los f[i] se inicializan con 2). De esta manera, la precondición de las *proof outlines* es:

$p = (I_{forks} \wedge \bigwedge_{i=0,4} e[i] = 0)$

Como invariante del recurso *forks* proponemos el siguiente predicado:

$I_{forks} = (\bigwedge_{i=0,4} ((0 \leq e[i] \leq 1) \wedge (e[i] = 1 \longrightarrow f[i] = 2) \wedge (f[i] = 2 - e[i - 1] - e[i + 1])))$

y como *proof outline* de cada proceso F_i:

{e[i] = 0}

```
while true do
{e[i] = 0}
with forks when f[i] = 2 do
{e[i] = 0}
      f[i – 1] := f[i – 1] – 1 ; f[i + 1] := f[i + 1] – 1 ; e[i] := 1 endwith ;
{e[i] = 1}
… comer …
{e[i] = 1}
with forks when true do
{e[i] = 1}
      f[i – 1] := f[i – 1] + 1 ; f[i + 1] := f[i + 1] + 1 ; e[i] := 0 endwith ;
{e[i] = 0}
… pensar …
{e[i] = 0}
od
{e[i] = 0}
```

El cumplimiento del invariante I_{forks} al comienzo de la composición paralela se prueba fácilmente. Tampoco reviste dificultad probar la correctitud de las *proof outlines*, con la aplicación de la regla WITH sobre los dos *with* de cada proceso. Veamos cómo con las *proof outlines* planteadas se verifica la exclusión mutua de dos procesos F_i y F_{i+1} con respecto a sus secciones *comer*. Supongamos por el contrario que no se cumple la propiedad, para llegar a una contradicción:

- De acuerdo a las *proof outlines*, vale: $e[i] = 1 \wedge e[i + 1] = 1$.
- También vale el invariante I_{forks}, lo que se justifica de la siguiente manera. Por un lado, cuando el programa reside en las secciones *comer* de F_i y F_{i+1}, ninguno de los dos procesos está ocupando el recurso *forks*. Pero, por otro lado, el recurso sí podría estar ocupado por un tercer proceso (representante de un filósofo no vecino a los filósofos i e i + 1). De todos modos, se puede probar que si existe una computación con estas características, entonces también existe otra, semánticamente equivalente, sin que el tercer proceso resida en una sección crítica condicional (intuitivamente, se obtiene modificando la secuencia de intercalación de las acciones atómicas).
- A partir de lo anterior, se obtiene la imagen semántica $e[i] = 1 \wedge e[i + 1] = 1 \wedge I_{forks}$, que implica el predicado falso $f[i] = 2 \wedge f[i] < 2$, lo que completa la prueba.

La prueba de ausencia de inanición de un proceso con respecto a un evento determinado (que en los programas paralelos descriptos sería el acceso a una sección crítica condicional), como la prueba de no divergencia, y en general de toda propiedad *liveness*, se basa en la utilización de una función definida en un conjunto bien fundado. Y propio de las propiedades

liveness, en general la ausencia de inanición no puede garantizarse si no se incorpora alguna asunción de *fairness* en la semántica del lenguaje de programación. Por ejemplo, sin *fairness* en el programa que implementa el problema de la cena de los filósofos puede producirse la situación en la que un filósofo nunca accede a sus dos tenedores, por su apropiación permanente por parte de los filósofos vecinos. Tal caso es imposible con una hipótesis de *fairness* que asegure que no pueden haber computaciones con infinitas ocurrencias de estados con f[i] = 2 sin que el proceso F_i acceda a su sección *comer*.

Ejemplo de desarrollo sistemático de programa

Para el único ejemplo de desarrollo sistemático de programa de este capítulo, consideramos a continuación un programa del modelo de variables compartidas, muy sencillo, con dos procesos y sin instrucciones de sincronización. A pesar de su simplicidad, el ejemplo sirve para destacar cómo se mantienen las ideas fundamentales descriptas anteriormente. En efecto, el esquema de desarrollo de los procesos es básicamente el que planteamos para los programas secuenciales determinísticos en el capítulo 2, con la salvedad, por la no composicionalidad del método axiomático, de que en la construcción de un proceso se deben tener en cuenta las interferencias del otro, y así la prueba de correctitud del programa obtenido debe incluir, además de las *proof outlines* de los procesos, el chequeo de libertad de interferencia de las mismas (la alternativa composicional que mencionamos previamente pero no describimos, es derivar los programas a partir de especificaciones que incluyan la información de las interferencias entre sus procesos).

Ejemplo 22. Desarrollo sistemático de un programa que busca el primer elemento positivo de un arreglo de números enteros

El programa a construir, ya referido en el capítulo 3, debe buscar el primer elemento positivo de un arreglo a[1:N] de números enteros, de solo lectura, con $N \geq 1$. Su especificación es:

$$\langle true \rangle\ S_{pep}\ \langle 1 \leq k \leq N + 1 \wedge \forall m: (1 \leq m < k \longrightarrow a[m] \leq 0) \wedge k \leq N \longrightarrow a[k] > 0 \rangle$$

que establece que el programa debe devolver el menor índice k del arreglo *a* tal que a[k] > 0, si existe, o el valor k = N + 1 en caso contrario. Como esquema general de solución, más eficiente que la anterior, proponemos un programa con dos procesos, S_1 y S_2, uno con un *while* que recorra los elementos del arreglo con índice impar y el otro con un *while* que recorra los elementos con índice par, hasta que alguno de los procesos encuentre un elemento positivo o hasta que ámbos alcancen el final del arreglo. La forma del programa a construir sería entonces:

S_{pep} :: i := 1 ; j := 2 ; k_1 := N + 1 ; k_2 := N + 1 ;

$[S_1 \,||\, S_2]$;
$k := \min(k_1, k_2)$

Los índices i y j son los manejados por los procesos S_1 y S_2 para recorrer el arreglo, respectivamente. La variable k_i pertenece a S_i y se utiliza para guardar el índice requerido si se encuentra (k_1 y k_2 deben ser variables compartidas para que un proceso termine ni bien el otro tenga eventualmente éxito en la búsqueda). El resultado del programa es el contenido menor de las dos variables, que denotamos con la función $\min(k_1, k_2)$. De acuerdo a esta idea de solución, especificamos el proceso S_1 del siguiente modo:

$$\langle N \geq 1 \wedge i = 1 \wedge k_1 = N + 1 \rangle \; S_1 \; \langle q_1 \rangle$$

con:

$$q_1 = (1 \leq k_1 \leq N + 1 \wedge \forall m: ((impar(m) \wedge 1 \leq m < k_1) \longrightarrow a[m] \leq 0) \wedge k_1 \leq N \longrightarrow a[k_1] > 0)$$

donde impar(m) es verdadero sii m es impar. Simétricamente, la especificación del proceso S_2 es:

$$\langle N \geq 1 \wedge j = 2 \wedge k_2 = N + 1 \rangle \; S_2 \; \langle q_2 \rangle$$

con:

$$q_2 = (2 \leq k_2 \leq N + 1 \wedge \forall m: ((par(m) \wedge 1 \leq m < k_2) \longrightarrow a[m] \leq 0) \wedge k_2 \leq N \longrightarrow a[k_2] > 0)$$

siendo par(m) verdadero sii m es par. Luego de las inicializaciones del programa se cumple la conjunción de las precondiciones de S_1 y S_2, y a partir de la conjunción de las postcondiciones de los mismos, con la asignación $k := \min(k_1, k_2)$ se obtiene la postcondición del programa.

El paso siguiente es definir los invariantes y variantes de los *while* de los procesos. Para el primer *while*, generalizando la postcondición de S_1 con el índice i llegamos al invariante:

$$p_1 = (1 \leq k_1 \leq N + 1 \wedge impar(i) \wedge 1 \leq i \leq k_1 + 1 \wedge \forall m: ((impar(m) \wedge 1 \leq m < i) \longrightarrow a[m] \leq 0) \wedge k_1 \leq N \longrightarrow a[k_1] > 0)$$

La cota superior $k_1 + 1$ del índice i determina la forma del *while*: primero se evalúa el elemento corriente a[i] y luego se avanza al siguiente elemento con índice impar. Como variante del *while* proponemos:

$$t_1 = k_1 + 1 - i$$

que a partir de N + 1 se decrementa después de cada iteración, y cumple $p_1 \longrightarrow t_1 \geq 0$. Análogamente, para el segundo *while* definimos el invariante:

$$p_2 = (2 \leq k_2 \leq N + 1 \wedge par(j) \wedge 2 \leq j \leq k_2 + 1 \wedge \forall m: ((par(m) \wedge 1 \leq m < j) \longrightarrow a[m] \leq 0) \wedge k_2 \leq N \longrightarrow a[k_2] > 0)$$

y el variante:

$$t_2 = k_2 + 1 - j$$

Con las definiciones anteriores derivamos la siguiente composición paralela del programa. Presentamos directamente las *proof outlines* de los procesos:

$\langle inv: p_1, var: t_1 \rangle$
$[S_1 ::$ while $i < \min(k_1, k_2)$ do $\langle p_1 \wedge i < k_1 \rangle$
if $a[i] > 0$ then $\langle p_1 \wedge i < k_1 \wedge a[i] > 0 \rangle$
$k_1 := i$
else $\langle p_1 \wedge i < k_1 \wedge a[i] \leq 0 \rangle$
$i := i + 2$ fi od
$\langle p_1 \wedge i \geq \min(k_1, k_2) \rangle$

||

$\langle inv: p_2, var: t_2 \rangle$
$S_2 ::$ while $j < \min(k_1, k_2)$ do $\langle p_2 \wedge j < k_2 \rangle$
if $a[j] > 0$ then $\langle p_2 \wedge j < k_2 \wedge a[j] > 0 \rangle$
$k_2 := j$
else $\langle p_2 \wedge j < k_2 \wedge a[j] \leq 0 \rangle$
$j := j + 2$ fi od]
$\langle p_2 \wedge j \geq \min(k_1, k_2) \rangle$

Se verifica fácilmente la correctitud de las *proof outlines*. También que la conjunción de las precondiciones de los procesos S_1 y S_2 implica $p_1 \wedge p_2$, y que la conjunción de las postcondiciones de los mismos es implicada por $p_1 \wedge i \geq \min(k_1, k_2) \wedge p_2 \wedge j \geq \min(k_1, k_2)$.

Para concluir la prueba de correctitud parcial del programa resta chequear que las *proof outlines* son libres de interferencia. Todas las validaciones correspondientes son triviales, salvo las que consideran el predicado $p_1 \wedge i \geq \min(k_1, k_2)$ con respecto a la asignación $k_2 := j$, y el predicado $p_2 \wedge j \geq \min(k_1, k_2)$ con respecto a la asignación $k_1 := i$. La primera validación requiere el cumplimiento de la fórmula:

$$\{p_1 \wedge i \geq \min(k_1, k_2) \wedge p_2 \wedge j < k_2 \wedge a[j] > 0\}\ k_2 := j\ \{p_1 \wedge i \geq \min(k_1, k_2)\}$$

que se verifica porque por el axioma ASI y la regla CONS:

$$\{p_1 \wedge i \geq \min(k_1, j)\}\ k_2 := j\ \{p_1 \wedge i \geq \min(k_1, k_2)\},$$

y el predicado $p_1 \wedge i \geq \min(k_1, k_2) \wedge p_2 \wedge j < k_2 \wedge a[i] > 0$) implica el predicado $p_1 \wedge i \geq \min(k_1, j)$:

$$(p_1 \wedge i \geq \min(k_1, k_2) \wedge p_2 \wedge j < k_2 \wedge a[i] > 0) \longrightarrow (p_1 \wedge i \geq \min(k_1, k_2) \wedge j < k_2) \longrightarrow (p_1 \wedge i \geq \min(k_1, j).$$

Del mismo modo verificamos el cumplimiento de la fórmula:

$\{p_2 \wedge j \geq \min(k_1, k_2) \wedge p_1 \wedge i < k_1 \wedge a[i] > 0\}\ k_1 := i\ \{p_2 \wedge j \geq \min(k_1, k_2)\}$

Por último, la no divergencia del programa se justifica porque las asignaciones de un proceso no alteran el valor del variante del *while* del otro proceso, al tiempo que cada variante decrece a lo largo de las secuencias sintácticamente posibles asociadas. No hay posibilidad de *deadlock* porque el programa no tiene instrucciones de sincronización.

Parte 2. Verificación de programas distribuidos

En esta segunda parte del capítulo completamos la descripción del método axiomático de verificación de programas concurrentes, con su variante axiomática correspondiente al modelo distribuido. Destacamos lo distintivo de la variante. Para la presentación nos valemos de una extensión concurrente del lenguaje de programación secuencial no determinístico utilizado en el capítulo 3, instanciada en el modelo de comunicaciones *sincrónicas* (en una nota adicional hacemos una breve referencia a la verificación de programas con comunicaciones *asincrónicas*).

Lenguaje de pasajes de mensajes

El lenguaje de programación que utilizamos se basa en el lenguaje CSP o *Communicating Sequential Processes* (Comunicación de Procesos Secuenciales), creado por C. Hoare. Los procesos no comparten variables, y se comunican y sincronizan por medio de *pasajes de mensajes*, generados por instrucciones de *entrada/salida*, también conocidas como instrucciones de *comunicación*.

Consideramos comunicaciones *sincrónicas*, esquema según el cual el envío de un mensaje por parte de un proceso y la recepción del mensaje por parte de otro se ejecutan atómicamente (un proceso puede quedar bloqueado a la espera de la comunicación con el otro, lo que puede producir una situación de *deadlock*). Los programas tienen la siguiente forma:

$S :: [P_1 :: S_1 \ ||\ \ldots\ ||\ P_n :: S_n]$

que se puede abreviar con $[\ ||_{i=1,n} P_i\ ]$. Es decir, son directamente composiciones distribuidas de procesos. Los procesos siempre se etiquetan, para identificarlos en los pasajes de mensajes (en este caso usamos etiquetas P_i). No hay anidamiento de concurrencia ni procesos dinámicos. Además del *skip*, la asignación, la secuencia y la composición distribuida, el repertorio de instrucciones del lenguaje incluye otras cuatro instrucciones, que pasamos a describir. Comenzamos con las instrucciones de entrada/salida:

- Instrucción de *entrada*. Su sintaxis es:

  P_i ? x

  En un proceso P_j, con $j \neq i$, que tiene una variable local x, la instrucción efectúa un pedido al proceso P_i para que asigne un valor a x.

- Instrucción de *salida*. Su sintaxis es:

  P_j ! e

  En un proceso P_i, con $i \neq j$, la instrucción efectúa un pedido al proceso P_j para que reciba el valor de la expresión *e*, definida con variables locales de P_i.

P_i ? x (en P_j) y P_j ! e (en P_i) progresan simultáneamente. Un proceso debe esperar al otro para que se produzca la comunicación entre ellos, cuyo efecto es la asignación del valor de *e* a x. También se pueden comunicar tuplas de valores. Se define que P_i ? x (en P_j) y P_j ! e (en P_i) *coinciden sintácticamente* si además de la coincidencia de los índices coinciden los tipos de datos de x y *e*. Para denotar una instrucción de entrada/salida se utiliza el símbolo α, y para denotar instrucciones de entrada/salida que coinciden sintácticamente, el par (α, α´), cuya comunicación asociada está *habilitada* cuando α y α´ son las siguientes instrucciones a ser llevadas a cabo por los procesos respectivos, y los mismos están *preparados* o *listos* para realizarlas.

Las instrucciones de entrada/salida pueden aparecer aisladamente en un proceso o bien ser parte de *selecciones condicionales y repeticiones con comunicaciones*, las dos instrucciones del lenguaje que nos quedan por describir:

- *Selección condicional con comunicaciones*. Su sintaxis más general es:

  if B_1 ; $\alpha_1 \rightarrow S_1$ or … or B_n ; $\alpha_n \rightarrow S_n$ fi

  siendo B_1, …, B_n expresiones booleanas y α_1, …, α_n instrucciones de entrada/salida, estas últimas opcionales. Es decir, la selección tiene la misma forma que la del lenguaje secuencial de comandos guardados, pero ahora una guardia puede incluir una instrucción de entrada/salida, en cuyo caso un comando guardado B_i ; $\alpha_i \rightarrow S_i$ puede ejecutarse (o lo que es lo mismo, la *dirección* i está *habilitada*) si B_i es verdadera y α_i es parte de una comunicación habilitada. La evaluación de las guardias es atómica. Por lo tanto, una selección condicional con comunicaciones se comporta de la siguiente manera: si tiene direcciones habilitadas, puede progresar no determinísticamente por alguna de ellas y pasar a la siguiente instrucción si el componente S_i respectivo termina; y si no tiene direcciones

habilitadas, si es porque todas las B_i son falsas la selección falla, y si no, queda bloqueada, temporaria o definitivamente, de acuerdo a lo que suceda en el programa más adelante.

- *Repetición con comunicaciones*. Su sintaxis más general es:

  do B_1 ; $\alpha_1 \rightarrow S_1$ or … or B_n ; $\alpha_n \rightarrow S_n$ od

  Valen las mismas definiciones del item anterior, y mantenemos la restricción del lenguaje secuencial no determinístico de comandos guardados, para simplificar la presentación, de no permitir repeticiones anidadas. Por lo tanto, una repetición con comunicaciones se comporta de la siguiente manera: si tiene direcciones habilitadas, puede progresar no determinísticamente por alguna de ellas y repetir el mismo ciclo mientras existan; si en algún momento ninguna dirección está habilitada, si es porque todas las B_i son falsas la repetición termina, y si no, queda bloqueada, temporaria o definitivamente, según lo que ocurra después en el programa; y si siempre existe una dirección habilitada, la repetición diverge.

Al igual que los programas paralelos, los programas distribuidos terminan, fallan (por causa de una selección condicional sin ninguna B_i verdadera o por *deadlock*) o divergen. La ejecución de una composición distribuida consiste en el *interleaving* de las instrucciones atómicas de los distintos procesos que la constituyen (instrucciones *skip*, asignaciones y comunicaciones), en base a la propiedad de progreso fundamental (luego nos referimos al *fairness*). La novedad semántica en los programas distribuidos es la ejecución conjunta de una instrucción de entrada y una instrucción de salida, y así el avance en dos procesos simultáneamente. Otro aspecto distintivo relacionado con las instrucciones de comunicación es la manifestación de dos tipos de no determinismo, uno *local*, proveniente del lenguaje secuencial no determinístico de comandos guardados, y el otro *global*, propio de la semántica de *interleaving*, que ejemplificamos a continuación. Los procesos:

P_1 :: if P_3 ? x $\rightarrow$ skip or P_3 ! 0 $\rightarrow$ skip fi
P_2 :: if true $\rightarrow$ P_3 ? x or true $\rightarrow$ P_3 ! 0 fi

implementan de dos maneras distintas las alternativas de recibir un valor del proceso P_3 o enviarle el valor 0. Que P_1 ejecute P_3 ? x o P_3 ! 0 depende de la situación global del programa. En cambio, en P_2 la elección entre una u otra instrucción de entrada/salida es local. Esta diferencia es irrelevante en términos de la correctitud parcial, pero crucial con respecto a la ausencia de *deadlock*. Por ejemplo, si el proceso P_3 envía el valor 1, mientras que el programa:

[P_1 :: if P_3 ? x $\rightarrow$ skip or P_3 ! 0 $\rightarrow$ skip fi || P_3 :: P_1 ! 1]

termina con x = 1, en el programa:

$[P_2 :: \text{if true} \rightarrow P_3 \text{ ? x or true} \rightarrow P_3 \text{ ! 0 fi} \;||\; P_3 :: P_2 \text{ ! 1}]$

se puede producir una situación de *deadlock* si el proceso P_2 elige la segunda dirección.

Completamos la descripción del lenguaje de pasajes de mensajes con otros dos ejemplos de programas. El primero consiste en la propagación de un valor desde un proceso P_1 hasta un proceso P_n:

$S_{prop} :: [P_1 \;||\; \dots \;||\; P_i \;||\; \dots \;||\; P_n]$, con:

$P_1 :: P_2 \text{ ! } y_1$

…..

$P_i :: P_{i-1} \text{ ? } y_i \text{ ; } P_{i+1} \text{ ! } y_i$

…..

$P_n :: P_{n-1} \text{ ? } y_n$

El último ejemplo es un programa que obtiene el mínimo de un conjunto de valores:

$S_{min} :: [P_1 \;||\; \dots \;||\; P_i \;||\; \dots \;||\; P_n \;||\; Q]$, con:

$P_i ::$ (mi_min_i , $mi\_tamaño_i$) := (a_i , 1) ;

do $0 < mi\_tamaño_i < n$; P_1 ! (mi_min_i , $mi\_tamaño_i$) $\rightarrow$ $mi\_tamaño_i$:= 0

……

or $0 < mi\_tamaño_i < n$; P_{i-1} ! (mi_min_i , $mi\_tamaño_i$) $\rightarrow$ $mi\_tamaño_i$:= 0

or $0 < mi\_tamaño_i < n$; P_{i+1} ! (mi_min_i , $mi\_tamaño_i$) $\rightarrow$ $mi\_tamaño_i$:= 0

……

or $0 < mi\_tamaño_i < n$; P_n ! (mi_min_i , $mi\_tamaño_i$) $\rightarrow$ $mi\_tamaño_i$:= 0

or $0 < mi\_tamaño_i < n$; P_1 ? (su_min_i , $su\_tamaño_i$) $\rightarrow$

(mi_min_i , $mi\_tamaño_i$) := ($min(mi\_min_i , su\_min_i)$, $mi\_tamaño_i + su\_tamaño_i$)

……

or $0 < mi\_tamaño_i < n$; P_{i-1} ? (su_min_i , $su\_tamaño_i$) $\rightarrow$

(mi_min_i , $mi\_tamaño_i$) := ($min(mi\_min_i , su\_min_i)$, $mi\_tamaño_i + su\_tamaño_i$)

or $0 < mi\_tamaño_i < n$; P_{i+1} ? (su_min_i , $su\_tamaño_i$) $\rightarrow$

(mi_min_i , $mi\_tamaño_i$) := ($min(mi\_min_i , su\_min_i)$, $mi\_tamaño_i + su\_tamaño_i$)

……

or $0 < mi\_tamaño_i < n$; P_n ? (su_min_i , $su\_tamaño_i$) $\rightarrow$

(mi_min_i , $mi\_tamaño_i$) := ($min(mi\_min_i , su\_min_i)$, $mi\_tamaño_i + su\_tamaño_i$)

od ;

if $mi\_tamaño_i = 0 \rightarrow$ skip or $mi\_tamaño_i = n \rightarrow Q$! mi_min_i fi

Q :: if P_1 ? min $\rightarrow$ skip or … or P_n ? min $\rightarrow$ skip fi

Para facilitar la escritura utilizamos asignaciones simultáneas de la forma (x_1, x_2) := (e_1, e_2). Cada proceso P_i del programa S_{min} se inicializa con un valor, su mínimo inicial, y un tamaño, su tamaño inicial, de valor 1. Luego ejecuta una repetición, en la que en cada iteración hace lo siguiente. O bien le envía a algún proceso P_j, con $i \neq j$, su mínimo actual y su tamaño actual y termina su participación en el programa asignándose un 0 a su tamaño. O bien recibe de algún

proceso P_j, con $i \neq j$, su mínimo actual y su tamaño actual, por medio de la función min se queda con el mínimo entre el mínimo actual propio y el que recibe, incrementa su tamaño actual con el tamaño que recibe, y continúa participando en el programa a menos que su tamaño actual sea n, lo que implica que obtuvo el mínimo buscado, en cuyo caso se lo envía al proceso Q.

En las secciones que siguen describimos la variante axiomática para verificar programas distribuidos. Por las características del lenguaje de pasajes de mensajes, la variante se basa en el método de verificación que presentamos para los programas secuenciales no determinísticos.

Axiomática para las pruebas de correctitud parcial

Comenzamos con la descripción de la axiomática para la prueba de correctitud parcial. Como con los programas paralelos, la meta es contar con una regla basada en la siguiente estructura:

$$\frac{\{p_i\}\ P_i^*\ \{q_i\},\ i = 1, \ldots, n}{\{\Lambda_{i=1,n}\ p_i\}\ [\ ||_{i=1,n}\ P_i\ ]\ \{\Lambda_{i=1,n}\ q_i\}}$$

siendo las $\{p_i\}\ P_i^*\ \{q_i\}$ *proof outlines* de correctitud parcial de los procesos que integran el programa distribuido $[\ ||_{i=1,n}\ P_i\ ]$, obtenidas en una primera etapa de pruebas locales y chequeadas en una segunda etapa de prueba global. En este caso no hace falta chequear que las *proof outlines* sean libres de interferencia, porque los procesos no comparten variables. Lo que sí se necesita es ratificar los predicados que necesariamente deben asumirse en las pruebas locales, producto de los pasajes de mensajes definidos entre los procesos (lo vemos enseguida).

Para las pruebas locales, la axiomática incluye, además de los axiomas SKIP y ASI y la regla SEC, reglas para las instrucciones de entrada/salida y adaptaciones de las reglas de la selección condicional y la repetición no determinísticas que utilizamos en la verificación de programas secuenciales no determinísticos con comandos guardados (NCOND y NREP, respectivamente):

30. *Axioma de la instrucción de entrada* (IN)

    $\{p\}\ P_i\ ?\ x\ \{q\}$

p y q son dos predicados arbitrarios con variables locales de un proceso P_j que incluye la instrucción P_i ? x. La arbitrariedad de la relación entre p y q se explica por las características de P_i ? x: en la prueba local de P_j, el valor asignado a la variable x proveniente del proceso P_i sólo puede asumirse, lo mismo que si P_i está preparado para comunicarse con P_j. Las asunciones deben validarse en la segunda etapa de la prueba, en la que se tiene en cuenta el programa completo.

31. *Axioma de la instrucción de salida* (OUT)

    $\{p\}\ P_j\ !\ e\ \{p\}$

p es un predicado con variables locales de un proceso P_i que incluye la instrucción P_j ! e, la cual no tiene efecto sobre las variables locales de P_i. También puede utilizarse el axioma más general {p} P_j ! e {q}, para incluir, como en el axioma IN, una asunción acerca del estado de P_j con respecto a su preparación para comunicarse con P_i. Cualquiera sea el caso, lo que se asume debe chequearse en la segunda etapa de la prueba.

32. *Regla del condicional con comunicaciones* (DCOND)

$$\frac{\{p \wedge B_i\}\ \alpha_i\ ;\ S_i\ \{q\},\ i = 1, \ldots, n}{\{p\}\ \text{if}\ B_1\ ;\ \alpha_1 \rightarrow S_1\ \text{or}\ \ldots\ \text{or}\ B_n\ ;\ \alpha_n \rightarrow S_n\ \text{fi}\ \{q\}}$$

La regla es la misma que utilizamos para los programas secuenciales no determinísticos con comandos guardados, pero adaptada considerando la posibilidad de que existan guardias con instrucciones de entrada/salida. La forma planteada es la más general (en los casos en los que los comandos guardados son $B_i \rightarrow S_i$, las premisas correspondientes son {p $\wedge$ B_i} S_i {q}).

33. *Regla de la repetición con comunicaciones* (DREP)

$$\frac{\{p \wedge B_i\}\ \alpha_i\ ;\ S_i\ \{p\},\ i = 1, \ldots, n}{\{p\}\ \text{do}\ B_1\ ;\ \alpha_1 \rightarrow S_1\ \text{or}\ \ldots\ \text{or}\ B_n\ ;\ \alpha_n \rightarrow S_n\ \text{do}\ \{p \wedge \bigwedge_i \neg B_i\}}$$

También esta regla, presentada en su forma más general, adapta contemplando guardias con instrucciones de entrada/salida una regla de la axiomática para programas secuenciales con comandos guardados, en este caso la que corresponde a la prueba de no divergencia.

Complementariamente, para la prueba global de la segunda etapa, en la que deben validarse las asunciones de las *proof outlines*, la axiomática incluye los siguientes componentes:

34. *Axioma de comunicación* (COM)

{p[x|e]} P_i ? x || P_j ! e {p}

El axioma captura la semántica de una comunicación entre dos instrucciones de entrada/salida que coinciden sintácticamente, que es la asignación del valor de la expresión *e* a la variable x. Su utilidad se aprecia en el chequeo de consistencia de las *proof outline*s, que en el marco de los programas distribuidos se conoce como *test de cooperación*, el cual pasamos a describir.

El test de cooperación requiere validar, por cada par de *proof outlines* {p_i} P_i* {q_i} y {p_j} P_j* {q_j}, que para todo par de instrucciones de entrada/salida que coinciden sintácticamente, P_i ? x en el proceso P_j y P_j ! e en el proceso P_i, la fórmula {p_1} P_i ? x {q_1} correspondiente a P_i ? x en la *proof outline* de P_j y la fórmula {p_2} P_j ! e {q_2} correspondiente a P_j ! e en la *proof outline* de P_i satisfagan:

{$p_1 \wedge p_2$} P_i ? x || P_j ! e {$q_1 \wedge q_2$}

De esta manera, el axioma COM permite validar todas las asunciones de las pruebas locales. Si se cumplen, se define que las *proof outlines cooperan*. En base a lo definido, la regla de prueba de correctitud parcial de una composición distribuida debería ser:

$$\frac{\{p_i\}\ P_i^*\ \{q_i\},\ i = 1, \ldots, n, \text{ son } \textit{proof outlines} \text{ que cooperan}}{\{\bigwedge_{i=1,n} p_i\}\ [\ ||_{i=1,n} P_i\ ]\ \{\bigwedge_{i=1,n} q_i\}}$$

pero así planteada, la regla está incompleta. Su formulación definitiva la mostramos después de los dos ejemplos de aplicación que presentamos a continuación (en el segundo se explicita cómo completar la regla).

Ejemplo 23. Prueba de correctitud parcial del programa que propaga un valor

Vamos a verificar la correctitud parcial del programa presentado en la sección previa, que propaga un valor desde un proceso P_1 hasta un proceso P_n. Para simplificar la prueba, acotamos el programa a tres procesos:

S_{prop} :: $[P_1 \,||\, P_2 \,||\, P_n]$, con:
P_1 :: P_2 ! y_1
P_2 :: P_1 ? y_2 ; P_3 ! y_2
P_3 :: P_2 ? y_3

Probaremos la fórmula $\{y_1 = Y\}\ S_{prop}\ \{y_3 = Y\}$.

Proponemos las siguientes *proof outlines*:

P_1 :: $\{y_1 = Y\}\ P_2$! y_1 $\{y_1 = Y\}$
P_2 :: $\{true\}\ P_1$? y_2 $\{y_2 = Y\}$; P_3 ! y_2 $\{y_2 = Y\}$
P_3 :: $\{true\}\ P_2$? y_3 $\{y_3 = Y\}$

La correctitud de las *proof outlines* se verifica fácilmente, aplicando los axiomas IN y OUT y la regla SEC. Resta probar que las *proof outlines* cooperan. Hay dos pares de instrucciones de entrada/salida que coinciden sintácticamente: P_2 ! y_1 (en P_1) y P_1 ? y_2 (en P_2), y P_3 ! y_2 (en P_2) y P_2 ? y_3 (en P_3). Por lo tanto, de acuerdo al test de cooperación, hay que verificar:

$\{y_1 = Y \wedge true\}\ P_2$! y_1 || P_1 ? y_2 $\{y_1 = Y \wedge y_2 = Y\}$
$\{y_2 = Y \wedge true\}\ P_3$! y_2 || P_2 ? y_3 $\{y_2 = Y \wedge y_3 = Y\}$

Las fórmulas se prueban aplicando el axioma COM y la regla CONS. Así obtenemos la fórmula $\{y_1 = Y \wedge true \wedge true\}\ S_{prop}\ \{y_1 = Y \wedge y_2 = Y \wedge y_3 = Y\}$, y por la regla CONS, $\{y_1 = Y\}\ S_{prop}\ \{y_3 = Y\}$.

Ejemplo 24. Prueba de correctitud parcial de un programa que devuelve el número uno

Dado el programa:

S_{0a1} :: $[P_1 \,||\, P_2]$, con:
P_1 :: x := 0 ; P_2 ! x ; x := x + 1 ; P_2 ! x
P_2 :: P_1 ? y ; P_1 ? y

vamos a verificar $\{true\}\ S_{0a1}\ \{y = 1\}$. Las *proof outlines* naturales de P_1 y P_2 son:

P_1 :: {true} x := 0 {x = 0} ; P_2 ! x {x = 0} ; x := x + 1 {x = 1} ; P_2 ! x {x = 1}
P_2 :: {true} P_1 ? y {y = 0} ; P_1 ? y {y = 1}

que de acuerdo al test de cooperación deben satisfacer:

a) $\{x = 0 \wedge true\}\ P_2$! x || P_1 ? y $\{x = 0 \wedge y = 0\}$
b) $\{x = 1 \wedge true\}\ P_2$! x || P_1 ? y $\{x = 1 \wedge y = 0\}$
c) $\{x = 0 \wedge y = 0\}\ P_2$! x || P_1 ? y $\{x = 0 \wedge y = 1\}$
d) $\{x = 1 \wedge y = 0\}\ P_2$! x || P_1 ? y $\{x = 1 \wedge y = 1\}$

Las fórmulas de (a) y (d) se prueban fácilmente por medio del axioma COM y la regla CONS. Sin embargo, las fórmulas de (b) y (c) no se cumplen. Por el axioma COM debe valer:

$(x = 1 \wedge true) \longrightarrow (x = 1 \wedge x = 0)$
$(x = 0 \wedge y = 0) \longrightarrow (x = 0 \wedge x = 1)$

y ambas implicaciones son falsas. Esto va a ocurrir indefectiblemente con cualquier par de *proof outlines* que se propongan, porque las fórmulas de (b) y (c) corresponden a comunicaciones imposibles de ejecutarse en el programa, situación que el test de cooperación no contempla tal como está definido. Así, el test debe ajustarse, para que no requiera tratar pares de instrucciones de entrada/salida que coinciden sintácticamente pero no semánticamente.

Lo que plantea la axiomática en este caso es, otra vez, recurrir a variables auxiliares (y la regla AUX), para reforzar predicados que precisen la historia del *interleaving* ejecutado. Pero además introduce una novedad, los *invariantes globales*, que en alguna medida recuerdan los

invariantes de recursos usados en la primera parte del capítulo. Antes de reformular el test de cooperación, completamos la prueba del ejemplo con la idea comentada:

- Primero agregamos una variable local auxiliar c_1 en el proceso P_1 y una variable local auxiliar c_2 en el proceso P_2, ámbas inicializadas en 0 y que se incrementan en 1 toda vez que el proceso respectivo ejecuta una instrucción de entrada/salida. El incremento y la instrucción de entrada/salida asociada se ejecutan atómicamente. De esta manera, los valores de las variables c_1 y c_2 siempre son iguales antes y después de cada comunicación del programa, lo que resulta útil para la prueba, como mostramos enseguida. Las *proof outlines* del programa ampliado quedan así (las secciones atómicas se delimitan con los símbolos < >):

$\{c_1 = 0\}$		$\{c_2 = 0\}$
$[P'_1 :: x := 0 ; \{x = 0 \wedge c_1 = 0\}$		$P'_2 :: < P_1 ? y ; c_2 := c_2 + 1 > ; \{y = 0 \wedge c_2 = 1\}$
$< P_2 ! x ; c_1 := c_1 + 1 > ; \{x = 0 \wedge c_1 = 1\}$	$\|\|$	$< P_1 ? y ; c_2 := c_2 + 1 >]$
$x := x + 1 ; \{x = 1 \wedge c_1 = 1\}$		$\{y = 1 \wedge c_2 = 2\}$
$< P_2 ! x ; c_1 := c_1 + 1 >$		
$\{x = 1 \wedge c_1 = 2\}$		

- Complementariamente, definimos un invariante global IG = $(c_1 = c_2)$, que, en sintonía con lo anterior, al agregarlo en los chequeos del test de cooperación permite descartar los pares de instrucciones de entrada/salida que coinciden sintácticamente pero no semánticamente, es decir las comunicaciones imposibles de ejecutarse en el programa. Las nuevas fórmulas a verificar con este agregado son las siguientes:

a) $\{x = 0 \wedge c_1 = 0 \wedge c_2 = 0 \wedge (c_1 = c_2)\}$
$< P_2 ! x ; c_1 := c_1 + 1 > \|\| < P_1 ? y ; c_2 := c_2 + 1 >$
$\{x = 0 \wedge c_1 = 1 \wedge y = 0 \wedge c_2 = 1 \wedge (c_1 = c_2)\}$

b) $\{x = 1 \wedge c_1 = 1 \wedge c_2 = 0 \wedge (c_1 = c_2)\}$
$< P_2 ! x ; c_1 := c_1 + 1 > \|\| < P_1 ? y ; c_2 := c_2 + 1 >$
$\{x = 1 \wedge c_1 = 2 \wedge y = 0 \wedge c_2 = 1 \wedge (c_1 = c_2)\}$

c) $\{x = 0 \wedge c_1 = 0 \wedge y = 0 \wedge c_2 = 1 \wedge (c_1 = c_2)\}$
$< P_2 ! x ; c_1 := c_1 + 1 > \|\| < P_1 ? y ; c_2 := c_2 + 1 >$
$\{x = 0 \wedge c_1 = 1 \wedge y = 1 \wedge c_2 = 2 \wedge (c_1 = c_2)\}$

d) $\{x = 1 \wedge c_1 = 1 \wedge y = 0 \wedge c_2 = 1 \wedge (c_1 = c_2)\}$
$< P_2 ! x ; c_1 := c_1 + 1 > \|\| < P_1 ? y ; c_2 := c_2 + 1 >$
$\{x = 1 \wedge c_1 = 2 \wedge y = 1 \wedge c_2 = 2 \wedge (c_1 = c_2)\}$

Se comprueba fácilmente que ahora se cumplen todas las fórmulas (asumiendo que las reglas utilizan secciones atómicas, lo que comentamos más adelante). Las fórmulas de (b) y (c) se

cumplen trivialmente porque las precondiciones son falsas, producto de la información global aportada por el invariante IG, que determina que las comunicaciones correspondientes no pueden ocurrir. Finalmente, llegamos a la fórmula {true $\wedge$ true} S_{0a1} {x = 1 $\wedge$ y = 1}, por aplicación de la regla AUX, y por la regla CONS, a {true} S_{0a1} {y = 1}.

Completado el ejemplo, formalizamos en lo que sigue las modificaciones necesarias para obtener la forma definitiva de la regla de la composición distribuida:

- Según cómo aparezca en el programa, toda instrucción de entrada/salida α de toda *proof outline* se encapsula en una sección atómica < S_1 ; α ; S_2 > o < α ⟶ S_1 >, en que ni S_1 ni S_2 contienen instrucciones de entrada/salida (se define que dos secciones atómicas coinciden sintácticamente si lo hacen las instrucciones de entrada/salida que encapsulan).
- Se especifica un invariante global IG en términos de las variables de los procesos y de variables auxiliares locales que se agregan al programa, que sólo pueden modificarse dentro de las secciones atómicas. IG debe valer al inicio y ser preservado por toda comunicación. Su rol es proveer información para evitar en las pruebas el tratamiento de pares de instrucciones de entrada/salida que coinciden sintácticamente pero no semánticamente, y por eso se agrega en los chequeos del test de cooperación.
- Se reformula el test de cooperación de la siguiente manera:
  Por cada par de *proof outlines* {p_i} P_i* {q_i} y {p_j} P_j* {q_i}, para todo par de secciones atómicas < S > (en P_j) y < S´ > (en P_i) que coinciden sintácticamente, la fórmula {p_1} < S > {q_1} asociada a < S > en la *proof outline* de P_j y la fórmula {p_2} < S´ > {q_2} asociada a < S´ > en la *proof outline* de P_i deben satisfacer:

  $$\{p_1 \wedge p_2 \wedge IG\} < S > \,||\, < S' > \{q_1 \wedge q_2 \wedge IG\}$$

  Se define en este caso que las *proof outlines* cooperan con respecto al invariante global IG.

- Por el uso de secciones atómicas, deben agregarse a la axiomática algunas reglas auxiliares relacionadas con las dos formas definidas, < S_1 ; α ; S_2 > y < α ⟶ S_1 >. Por ejemplo:

$$\frac{\{p\}\ S_1 ; S_3\ \{r_1\},\ \{r_1\}\ \alpha \,||\, \alpha'\ \{r_2\},\ \{r_2\}\ S_2 ; S_4\ \{q\}}{\{p\} < S_1 ; \alpha ; S_2 > \,||\, < S_3 ; \alpha' ; S_4 > \{q\}}$$

$$\frac{\{p\}\ \alpha \,||\, \alpha'\ \{r\},\ \{r\}\ S_1 ; S_2\ \{q\}}{\{p\} < \alpha \longrightarrow S_1 > \,||\, < \alpha' \longrightarrow S_2 > \{q\}}$$

Con la nueva definición de cooperación, necesaria en realidad en la prueba de toda propiedad de un programa distribuido para descartar los pares de instrucciones de entrada/salida que coinciden sintácticamente pero no semánticamente, llegamos a la versión definitiva de la regla de la composición distribuida (cuya aplicación luego es seguida por la de la regla AUX):

35. *Regla de la composición distribuida* (DIST)

$\{p_i\}\ S_i^*\ \{q_i\}$, i = 1, …, n, son *proof outlines* que cooperan con respecto a IG
$p \longrightarrow (\bigwedge_{i=1,n} p_i \wedge IG),\ (\bigwedge_{i=1,n} q_i \wedge IG) \longrightarrow q$

$$\{p\}\ [\ ||_{i=1,n} P_i\ ]\ \{q\}$$

tal que ninguna variable de IG puede ser modificada fuera de las secciones atómicas.

Axiomática para las pruebas de no divergencia

En los programas distribuidos, a diferencia de lo que observamos en los programas paralelos, al no haber variables compartidas las pruebas no tienen que reparar en el impacto de un proceso por esta vía sobre el valor del variante definido para una repetición de otro proceso.

Pero aparece otra dificultad, que es que la no divergencia de un proceso no siempre puede probarse localmente, dadas sus comunicaciones con otros procesos.

Por ejemplo, que el proceso:

P_1 :: do $x \neq 0$; P_2 ? $x \rightarrow$ skip od

no diverja depende de si recibe alguna vez del proceso P_2 el valor 0.

Así que en este caso también hay que recurrir a asunciones en las pruebas locales y verificarlas en el test de cooperación de la prueba global (incluyendo variables auxiliares locales y un invariante global). En particular, puede ser necesario asumir el valor inicial de algunos variantes, si la cantidad de iteraciones de las repeticiones correspondientes depende de información global.

En las pruebas locales, para verificar la no divergencia de una repetición se utiliza la misma regla que definimos para los programas secuenciales con comandos guardados, pero ahora contemplando instrucciones de entrada/salida en las guardias. Su forma más general es:

36. *Regla de la no divergencia con comunicaciones* (DREP*)

$\langle p \wedge B_i \rangle\ \alpha_i\ ;\ S_i\ \langle p \rangle$, i = 1, …, n, $\langle p \wedge B_i \wedge t = Z \rangle\ \alpha_i\ ;\ S_i\ \langle t < Z \rangle$, i = 1, …, n, $p \rightarrow t \geq 0$

$$\langle p \rangle\ \text{do}\ B_1 \rightarrow S_1\ \text{or}\ldots\text{or}\ B_n \rightarrow S_n\ \text{od}\quad \langle p \wedge \bigwedge_{i=1,n} \neg B_i \rangle$$

tal que p y t son el invariante y el variante de la repetición, respectivamente. Para la prueba global se recurre al test de cooperación de la misma manera que en la prueba de correctitud parcial:

37. *Regla de la no divergencia distribuida* (DIST*)

$\langle p_i \rangle\ S_i^{**}\ \langle q_i \rangle$, i = 1, …, n, son *proof outlines* que cooperan con respecto a IG
$p \longrightarrow (\bigwedge_{i=1,n} p_i \wedge IG),\ (\bigwedge_{i=1,n} q_i \wedge IG) \longrightarrow q$

$$\langle p \rangle\ [\ ||_{i=1,n} P_i\ ]\ \langle q \rangle$$

tal que ninguna variable de IG puede ser modificada fuera de las secciones atómicas. Mostramos a continuación un ejemplo de aplicación de estas reglas sobre un programa muy sencillo.

Ejemplo 25. Prueba de no divergencia de un programa con una repetición de diez iteraciones

Vamos a probar:

$\langle 0 \leq x \leq 10 \wedge y > 0 \rangle\ S_{diez} :: [P_1 \parallel P_2]\ \langle true \rangle$, con:

$P_1 :: do\ x \geq 0\ ;\ P_2\ !\ x \rightarrow x := x - 1\ od$

$P_2 :: do\ y \neq 0\ ;\ P_1\ ?\ y \rightarrow skip\ od$

Primero definimos las secciones atómicas del programa, y agregamos en el proceso P_2 una variable auxiliar w para diferenciar las instancias anterior y posterior de la primera iteración de su instrucción de repetición, cuya utilidad explicamos enseguida:

$P'_1 :: do\ x \geq 0\ ;\ <P_2\ !\ x \rightarrow x := x - 1>\ od$

$P'_2 :: w := 0\ ;\ do\ y \neq 0\ ;\ <P_1\ ?\ y \rightarrow skip\ ;\ w := 1>\ od$

Para la prueba de no divergencia del proceso P_1 definimos como invariante de su repetición el predicado $p_1 = (x \geq -1)$, y como variante la función $t_1 = x + 1$. De acuerdo a la regla DREP*, deben cumplirse las siguientes fórmulas:

1. $\langle x \geq -1 \wedge x \geq 0 \rangle\ P_2\ !\ x\ ;\ x := x - 1\ \langle x \geq -1 \rangle$
2. $\langle x \geq -1 \wedge x \geq 0 \wedge x + 1 = Z \rangle\ P_2\ !\ x\ ;\ x := x - 1\ \langle x + 1 < Z \rangle$
3. $x \geq -1 \rightarrow x + 1 \geq 0$

que se verifican fácilmente por medio de los axiomas ASI y OUT y las reglas SEC y CONS.

La prueba de no divergencia de P_2 es más complicada, porque el valor de la variable *y* depende de lo que recibe de P_1. Definimos para la repetición de P_2 el invariante $p_2 = (y \geq 0)$ y el variante t_2 = (if w = 0 then 11 else y fi). La variable auxiliar w sirve para asignar un valor inicial al variante. En este caso, las fórmulas que deben cumplirse por DREP* son:

1. $\langle y \geq 0 \wedge y \neq 0 \rangle\ P_1\ ?\ y\ ;\ skip\ ;\ w := 1\ \langle y \geq 0 \rangle$
2. $\langle y \geq 0 \wedge y \neq 0 \wedge$ (if w = 0 then 11 else y fi) $= Z \rangle$
   $P_1\ ?\ y\ ;\ skip\ ;\ w := 1$
   $\langle$(if w = 0 then 11 else y fi) $< Z \rangle$
3. $y \geq 0 \rightarrow$ (if w = 0 then 11 else y fi) ≥ 0

que también se prueban fácilmente, aplicando los axiomas SKIP, ASI e IN y las reglas SEC y CONS. Las *proof outlines* de los procesos ampliados quedan así:

⟨inv: $x \geq -1$, var: $x + 1$⟩
P'_1 :: do $x \geq 0$; ⟨$x \geq -1 \wedge x \geq 0$⟩ < P_2 ! $x \rightarrow x := x - 1$ > od
⟨$x \geq -1$⟩

⟨$y > 0$⟩
P'_2 :: $w := 0$;
⟨inv: $y \geq 0$, var: if $w = 0$ then 11 else y fi⟩
    do $y \neq 0$; ⟨$y \geq 0 \wedge y \neq 0$⟩ < P_1 ? $y \rightarrow$ skip ; $w := 1$ > od
⟨$y \geq 0$⟩

Se cumple que la precondición del programa implica la conjunción de las precondiciones de los procesos:

$(0 \leq x \leq 10 \wedge y > 0) \rightarrow (x \geq -1 \wedge y > 0)$.

Para la segunda etapa de la prueba planteamos el siguiente invariante global:

$IG = (0 \leq w \leq 1 \wedge (w = 1 \rightarrow y = x + 1))$

alineado con el comportamiento del programa modificado, por lo que el chequeo requerido por el test de cooperación utilizado por la regla DIST* queda planteado de esta forma:

⟨$x \geq -1 \wedge x \geq 0 \wedge y \geq 0 \wedge y \neq 0 \wedge (0 \leq w \leq 1 \wedge (w = 1 \rightarrow y = x + 1)$⟩
< P_2 ! $x \longrightarrow x := x - 1$ > || < P_1 ? $y \longrightarrow$ skip ; $w := 1$ >
⟨$x \geq -1 \wedge y \geq 0 \wedge (0 \leq w \leq 1 \wedge (w = 1 \rightarrow y = x + 1)$⟩

La fórmula se verifica por medio de la segunda regla auxiliar presentada en la sección anterior, los axiomas SKIP, ASI y COM y las reglas SEC y CONS. La prueba se completa aplicando la regla AUX.

Con hipótesis de *fairness* en los programas distribuidos, en las pruebas de no divergencia (y en general en las pruebas de cualquier propiedad de tipo *liveness*) se plantean más escenarios que el de nivel *proceso* que mencionamos en el marco de los programas paralelos. Por ejemplo, con hipótesis de *fairness* fuerte se pueden definir los siguientes dos niveles adicionales:

1. Nivel *canal*: todo par de procesos infinitas veces preparados para comunicarse se comunican infinitas veces.
2. Nivel *comunicación*: toda comunicación infinitas veces habilitada se ejecuta infinitas veces.

El siguiente programa sirve para aclarar las definiciones anteriores. En este caso, la no divergencia sólo se logra con *fairness* fuerte de nivel comunicación:

$[P_1$ :: b := true ; $\quad\quad\quad\quad\quad\quad$ P_2 :: c := true ;

$\quad\quad$ do b ; P_2 ? b $\rightarrow$ skip od $\quad\|\quad$ do c ; P_1 ! true $\rightarrow$ skip

$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad$ or c ; P_1 ! false $\rightarrow$ c := false

$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad$ od]

El programa termina si P_2 toma alguna vez la segunda dirección de su instrucción de repetición, lo que no se puede asegurar con *fairness* fuerte de niveles proceso o canal.

La no divergencia con *fairness* se puede probar en base a *direcciones útiles*, como hicimos con los programas secuenciales no determinísticos, pero ahora considerando, en lugar de direcciones habilitadas, pares de direcciones habilitadas.

Axiomática para las pruebas de ausencia de falla

Completamos la presentación de la axiomática con una breve referencia a la verificación de la propiedad de ausencia de falla. Un programa distribuido falla si durante su ejecución, en una selección condicional todas las expresiones booleanas de sus guardias son falsas, o si se produce una situación de *deadlock*. Para verificar la ausencia de falla del primer caso, la axiomática incluye una regla similar a la regla DCOND utilizada para la prueba de correctitud parcial, con una premisa adicional. Su forma más general es:

38. *Regla del condicional con comunicaciones sin falla* (DCOND*)

$$\frac{p \rightarrow V_{i=1,n}\, B_i,\ \{p \wedge B_i\}\ \alpha_i\ ;\ S_i\ \{q\},\ i = 1, \ldots, n}{\{p\}\ \text{if } B_1\ ;\ \alpha_1 \rightarrow S_1 \text{ or } \ldots \text{ or } B_n\ ;\ \alpha_n \rightarrow S_n \text{ fi } \{q\}}$$

La premisa $p \rightarrow V_{i=1,n}\, B_i$ asegura que al menos una guardia de la selección condicional tiene una expresión booleana verdadera. Con respecto a la prueba de ausencia de *deadlock*, la regla correspondiente se basa en el mismo esquema que planteamos para los programas paralelos:

39. *Regla de la ausencia de deadlock en programas distribuidos* (DDEADLOCK)

Dadas *proof outlines* (con secciones atómicas y que cooperan con respecto a un invariante global IG), se identifican sintácticamente todas las posibles situaciones de *deadlock*, se obtienen las imágenes semánticas asociadas, y se prueba que todas resultan falsas. Por las características de los programas distribuidos, los predicados que integran las imágenes semánticas pueden ser:

- post(P_i), la postcondición de la *proof outline* del proceso P_i.
- pre(< S >), la precondición de una sección atómica S de la forma S_1 ; α ; S_2.

- pre(S) $\wedge \bigwedge_{i \in H} B_i \wedge \bigwedge_{j \notin H} \neg B_j$, la precondición de una selección condicional o una repetición S, en conjunción con un predicado que establece que S tiene un subconjunto H no vacío de direcciones con expresiones booleanas verdaderas.

No puede haber sólo predicados del primer tipo, porque si no la imagen semántica no representaría una situación de *deadlock*. Como ejemplo de aplicación de esta regla, consideramos nuevamente el programa de propagación de un valor.

Ejemplo 26. Prueba de ausencia de *deadlock* en el programa que propaga un valor

Para verificar la correctitud parcial del programa P_{prop} :: [P_1 :: P_2 ! y_1 || P_1 ? y_2 ; P_3 ! y_2 || P_2 ? y_3] hemos utilizado en una sección anterior las *proof outlines*:

P_1 :: {$y_1 = Y$} P_2 ! y_1 {$y_1 = Y$}
P_2 :: {true} P_1 ? y_2 {$y_2 = Y$}; P_3 ! y_2 {$y_2 = Y$}
P_3 :: {true} P_2 ? y_3 {$y_3 = Y$}

que no sirven en este caso para probar la ausencia de *deadlock*. Por ejemplo, la imagen semántica correspondiente a la situación de *deadlock* en que P_1 terminó y P_2 y P_3 no empezaron:

$y_1 = Y \wedge true \wedge true$

no es falsa. Se necesita recurrir a variables auxiliares y un invariante global. Agregamos una variable w_i en cada proceso P_i, tal que $w_i = 0$ significa que P_i está preparado para recibir un valor, $w_i = 1$ que P_i está preparado para enviar un valor, y $w_i = 2$ que P_i terminó. La variable w_1 se inicializa en 1, y las variables w_2 y w_3 en 0. Correspondientemente con esta idea, definimos el siguiente invariante global:

$IG = (w_1 = 1 \wedge w_2 = 0 \wedge w_3 = 0) \vee (w_1 = 2 \wedge w_2 = 1 \wedge w_3 = 0) \vee (w_1 = 2 \wedge w_2 = 2 \wedge w_3 = 2)$

Las *proof outlines* de los procesos ampliados quedan de la siguiente manera:

{$w_1 = 1$}		{$w_2 = 0$}		{$w_3 = 0$}
[< P_2 ! y_1 ; w_1 := 2>	||	<P_1 ? y_2 ; w_2 := 1>	||	<P_2 ? y_3 ; w_3 := 2>]
{$w_1 = 2$}		{$w_2 = 1$}		{$w_3 = 2$}
		<P_3 ! y_2 ; w_2 := 2>		
		{$w_2 = 2$}		

La cooperación entre las *proof outlines* con respecto a IG se prueba fácilmente, empleando la primera regla auxiliar presentada previamente. Notar que ahora la imagen semántica que representa la situación en la que P_1 terminó y P_{2} y P_3 no empezaron, incluyendo el invariante IG, resulta falsa:

$(w_1 = 2 \wedge w_2 = 0 \wedge w_3 = 0) \wedge$
$[(w_1 = 1 \wedge w_2 = 0 \wedge w_3 = 0) \vee (w_1 = 2 \wedge w_2 = 1 \wedge w_3 = 0) \vee (w_1 = 2 \wedge w_2 = 2 \wedge w_3 = 2)]$

Se comprueba fácilmente lo mismo con el resto de las situaciones posibles de *deadlock*.

Observaciones finales

El método descripto en este último capítulo del libro utiliza los mismos conceptos generales estudiados previamente. Sin embargo, tiene una diferencia relevante con respecto a lo observado en las variantes axiomáticas para los programas secuenciales, que es que no es composicional, producto del tipo de especificación que maneja. De todos modos, es una buena aproximación, determinando pruebas guiadas por la sintaxis de los programas, estructuradas en dos etapas, una primera etapa de elaboración de *proof outlines* de procesos, obtenidas composicionalmente, y una segunda etapa de validación de las mismas, con un criterio y un costo que dependen de la propiedad considerada y lenguaje de programación utilizado.

Otra característica diferencial del método de verificación de programas concurrentes que presentamos es la utilización de variables auxiliares, para registrar las historias de las computaciones cuando las variables de programa no resultan suficientes, y de invariantes globales, particularmente en el caso de los programas distribuidos, ante la ausencia de estados que reúnan información común a todos los procesos.

En la verificación de un programa concurrente se tratan más propiedades que en un programa secuencial, como mínimo la ausencia de *deadlock*. La prueba de esta propiedad, al igual que otras propiedades *safety*, se basa en asignar a determinadas configuraciones, definidas en términos de puntos de control de procesos, predicados que la contradigan. Por su parte, las pruebas de propiedades *liveness*, como la no divergencia, dependen marcadamente de la hipótesis de progreso de las computaciones que se asuma en la semántica de los programas.

Desde el punto de vista del desarrollo sistemático de programas, en general se acepta que el modelo distribuido tiene ventajas en relación al modelo paralelo, lo que se percibe en la práctica por la existencia de metodologías de construcción de programas del primer modelo más disciplinadas, característica que se refleja también en las pruebas asociadas.

Notas adicionales

Verificación de programas distribuidos con comunicaciones asincrónicas

En los programas distribuidos con comunicaciones asincrónicas, los procesos ejecutan las instrucciones de salida sin tener en cuenta la disponibilidad de sus contrapartes para recibir los mensajes. Sólo las instrucciones de entrada pueden tener que esperar para ejecutarse, y así provocar eventualmente bloqueos en los procesos que las contienen. La semántica informal de ambas instrucciones en esta clase de programas es la siguiente (consideramos un lenguaje distribuido similar al presentado en el capítulo, reemplazando los símbolos ? y ! por ?? y !!, respectivamente, y utilizando identificadores de canales en lugar de identificadores de procesos y guardias sin instrucciones de salida):

- Instrucción de *entrada* c ?? x (en un proceso receptor P_r): Si en el canal c existe un valor disponible, éste se asigna a la variable local x de P_r. Si no, P_r queda bloqueado hasta que haya un valor en c.
- Instrucción de *salida* c !! e (en un proceso emisor P_e): P_e envía al canal c el valor de la expresión *e*, definida en términos de sus variables locales, y continúa con la ejecución de la siguiente instrucción.

Otras características del lenguaje son: todo canal conecta a un único proceso emisor con un único proceso receptor, la única hipótesis sobre el tiempo entre el envío y la recepción de un mensaje es que es finito, y los mensajes llegan en el orden en el que se envían. Por ejemplo, el siguiente programa calcula el máximo de un conjunto de n valores x_i, todos distintos, en base a una estructura de anillo de n procesos distribuidos:

S_{max} :: [|| $_{i = 1,n}$ P_i], con:

P_i :: z_i := 0 ; y_i := 0 ; c_i !! x_i ;
do $y_i \neq N \rightarrow c_{i-1}$?? y_i ;
if $x_i = y_i \rightarrow z_i := 1$; $y_i := N$; c_i !! N
or $x_i > y_i \rightarrow$ skip
or $x_i < y_i \rightarrow c_i$!! y_i
fi
od

El canal c_i comunica al proceso P_i con su vecino de la derecha, P_{i+1} (o P_1 si i = n). P_i contiene el valor x_i, que envía al inicio. Luego, iterativamente, P_i envía todo valor y_i mayor que su valor x_i, que recibe de su vecino de la izquierda, P_{i-1}, en el canal c_{i-1} (o P_n en c_n si i = 1). De esta manera, sólo el valor máximo recorre el anillo completo de procesos, situación que detecta el proceso P_i

que recibe su valor x_i, en cuyo caso se identifica ejecutando la asignación $z_i := 1$ y hace finalizar el programa enviando un valor N mayor que todos los x_i al resto de los procesos.

En (Schlichting y Schneider, 1984) se describe una variante axiomática para verificar programas del lenguaje descripto, basada en el esquema de pruebas en dos etapas de S. Owicki y D. Gries que analizamos antes. A cada canal c se le asocian dos variables de tipo *secuencia* para utilizar en las pruebas, una variable IN_c, local del proceso receptor, y una variable OUT_c, local del proceso emisor, que representan la secuencia de valores recibidos por uno y la secuencia de valores enviados por el otro, respectivamente.

Los axiomas para las pruebas locales relacionados con las instrucciones de entrada/salida son:

Axioma de la instrucción de entrada asincrónica (AIN)

{p} c ?? x {q}

Al igual que el axioma IN para las comunicaciones sincrónicas, plantea en el caso más general asunciones sobre el valor recibido y sobre la posibilidad de recibir un mensaje, a ser validadas en el test de cooperación de la prueba global.

Axioma de la instrucción de salida asincrónica (AOUT)

{p[OUT_c|OUT_C.e]} c !! e {p}

El axioma captura el efecto de la instrucción de salida: el envío del valor de la expresión *e* al final de la secuencia de valores enviados al canal c (OUT_C.e expresa la concatenación de la secuencia representada por OUT_c con *e*). En este caso, la precondición y la postcondición coinciden siempre, porque en una comunicación asincrónica no se necesita asumir nada sobre la disponibilidad del proceso receptor para recibir un mensaje.

Axioma del orden de llegada (FIFO)

Para todo k, si $|IN_c| = k$ entonces se cumple $|OUT_c| \geq k$ y también $IN_c[k] = OUT_c[k]$.

$|IN_c|$ expresa el tamaño de la secuencia IN_c, e $IN_c[k]$ el valor k-ésimo de la misma. Lo mismo se define para OUT_c. El axioma establece que los mensajes se reciben en el orden en el que se envían (FIFO abrevia *first in first out*, es decir, el primero que entra es el primero que sale).

En lo que hace a los componentes de la axiomática para la prueba global, cabe destacar que por el uso de las variables IN_c y OUT_c, en el test de cooperación no hace falta recurrir a un invariante global. El test consiste en confrontar, para cada par de *proof outlines*, todos los pares de instrucciones de entrada/salida que coinciden sintácticamente, que en este caso están constituidos por una instrucción de entrada y una instrucción de salida que actúan sobre un mismo canal. Específicamente, dadas las fórmulas $\{p_1\}$ c ?? x $\{q_1\}$ y $\{p_2\}$ c !! e $\{q_2\}$, debe cumplirse:

$\{p_1 \wedge p_2 \wedge IN_c = OUT_c\}$ c ?? x || c !! e $\{q_1\}$

No se considera la postcondición de la instrucción de salida, acorde con la semántica de las comunicaciones asincrónicas. La igualdad $IN_c = OUT_c$ permite descartar las instrucciones de entrada/salida que no coinciden semánticamente. Para llevar a cabo el test se emplea el siguiente el axioma:

Axioma de comunicación asincrónica (ACOM)

$\{p[x|e][IN_c|IN_C.e]\}$ P_i ? x || P_j ! e $\{p\}$

que captura la semántica de la comunicación asincrónica. Con lo último definido, se puede formular la regla de la composición distribuida para la prueba de correctitud parcial:

Regla de la composición distribuida con comunicaciones asincrónicas (ADIST)

$\{p_i\}$ S_i* $\{q_i\}$, i = 1, …, n, son *proof outlines* que cooperan

$p \longrightarrow (\bigwedge_{i = 1,n} p_i \wedge \bigwedge_c IN_c = OUT_c = \lambda)$, $(\bigwedge_{i = 1,n} q_i) \longrightarrow q$

———————————————————————————

$\{p\}$ [|| $_{i = 1,n}$ P_i] $\{q\}$

Se agrega $\bigwedge_c IN_c = OUT_c = \lambda$ en la precondición para indicar que al comienzo todos los canales están vacíos (tienen la secuencia vacía λ). El resto de las propiedades se prueban a partir de *proof outlines* de correctitud parcial que cooperan.

Composicionalidad y desarrollo sistemático de programas concurrentes

A la variante axiomática no composicional de S. Owicki y D. Gries le suceden varias propuestas composicionales. La primera, para la correctitud parcial de programas paralelos, es (Jones, 1981). En (Xu, de Roever y He, 1997) se la describe y amplía para la prueba de correctitud total, incluyendo además consideraciones sobre la sensatez y la completitud de la axiomática. La idea básica es incluir en la especificación la información de las interferencias entre los procesos y el entorno de ejecución. A cada proceso, además de su pre y postcondición, se le asocia una *condición de fiabilidad*, que establece lo que el proceso espera del entorno, y una *condición de garantía*, que establece cómo el proceso influencia en el entorno. Así, la verificación debe contemplar las relaciones entre dichas condiciones (por ejemplo, debe asegurar que lo que garantiza un proceso implica lo que esperan los otros procesos de él). Más en detalle, utilizando para simplificar sólo dos procesos S_1 y S_2, se plantea lo siguiente:

- En cuanto a la especificación: dada una precondición p y una postcondición q de una composición concurrente [S_1 || S_2], definir un conjunto de variables auxiliares y dos tuplas de predicados (p_1, q_1, $rely_1$, $guar_1$) y (p_2, q_2, $rely_2$, $guar_2$) para S_1 y S_2, respectivamente, siendo p_1 y p_2 las precondiciones, q_1 y q_2 las postcondiciones, y $rely_1$, $guar_1$, $rely_2$ y $guar_2$ relaciones

que establecen transformaciones entre las variables compartidas y auxiliares. El significado de la tupla (p_i, q_i, $rely_i$, $guar_i$) es que cuando S_i se ejecuta a partir de un estado que satisface p_i, en un entorno en el que se pueden modificar las variables compartidas y auxiliares de acuerdo a la condición $rely_i$, el proceso lleva a cabo transformaciones que respetan la condición $guar_i$, y si termina, lo hace en un estado que satisface q_i.

- En cuanto a la verificación: elaborar *proof outlines* de S_1 y S_2 que satisfagan las condiciones establecidas (por ejemplo, que los invariantes se preserven teniendo en cuenta las condiciones $rely_i$), para llegar a la fórmula $\{p_1 \wedge p_2\}\ [S_1 \parallel S_2]\ \{q_1 \wedge q_2\}$, y finalmente, mediante las reglas de las variables auxiliares y de consecuencia, a $\{p\}\ [S_1 \parallel S_2]\ \{q\}$.

Para el tratamiento de la ausencia de *deadlock* se sigue una línea similar, agregando en la especificación información acerca de los posibles bloqueos de los procesos. Con respecto a la no divergencia, la idea es directamente adaptar la regla de no divergencia de la variante no composicional, considerando las nuevas condiciones. Las mismas nociones generales pueden aplicarse a los programas distribuidos.

El enfoque descripto convive con el de S. Owicki y D. Gries, lo que se explica porque este último resulta más accesible, sobre todo para especificar los invariantes.

Un método axiomático composicional es más conveniente fundamentalmente para el desarrollo sistemático de programas, permite que los procesos se deriven y prueben directamente desde las especificaciones, y que en general las propiedades de los programas se infieran a partir de las propiedades de los procesos, sin necesidad de chequear la consistencia de las *proof outlines*. Entre las metodologías de desarrollo con los principios mencionados se destaca UNITY o *Unbounded Nondeterministic Iterative Transformations* (Transformaciones Iterativas no Determinísticas no Acotadas), presentada por primera vez en (Chandy y Misra, 1988). Esta obra se considera fundacional para el desarrollo sistemático de programas concurrentes (la nombramos en el capítulo 3 por su relación con el no determinismo). La metodología incluye las etapas de especificación y verificación. Aunque se basa en la lógica temporal, comentamos en lo que sigue algunas de sus características para profundizar en lo que aporta un método composicional.

Un programa UNITY es esencialmente una repetición infinita de asignaciones condicionales que se ejecutan no determinísticamente en tanto estén habilitadas infinitas veces. Existe una noción de terminación, que se manifiesta si al cabo de una cantidad finita de iteraciones se alcanza un estado que no se modifica más (se lo conoce como *punto fijo*). A partir de una primera versión de programa, la idea es refinarla gradualmente, hasta alcanzar el nivel de detalle esperado, instanciado a una de varias arquitecturas posibles. Existen dos estrategias de construcción de programas, la *unión* y la *superposición*, que representan la composición y el refinamiento, respectivamente. La unión de dos procesos S_1 y S_2, denotada con $[S_1 \parallel S_2]$, consiste en su concatenación, mientras que la superposición establece una programación por capas, de modo tal que a una nueva capa se le agregan variables y asignaciones que no alteran

el cómputo de las capas inferiores. Las especificaciones se definen en términos de cuatro operadores básicos, *FP*, *unless*, *ensures* y *leads-to*:

- FP**.**S establece el conjunto de puntos fijos del programa S.
- p *unless* q establece que, si se cumple p, q no se cumple nunca y p se cumple siempre, o q se cumple a futuro y p se cumple mientras q no se cumpla. De este operador se derivan *invariant* p (p se cumple siempre) y *stable* p, equivalente a p *unless false*.
- p *ensures* q establece que si se cumple p, q se cumple a futuro y p se cumple mientras q no se cumpla.
- p *leads-to* q establece que si se cumple p, q se cumple a futuro.

La metodología propone hacer gran parte de la tarea de desarrollo de un programa en la etapa de especificación, para facilitar su cálculo *top-down* y la verificación de sus propiedades a partir de las propiedades de los procesos. En este último sentido, se demuestra por ejemplo que:

- Se cumple p *unless* q en $[S_1 \parallel S_2]$ si se cumple p *unless* q en los dos procesos.
- Se cumple p *ensures* q en $[S_1 \parallel S_2]$ si se cumple p *ensures* q en un proceso y p *unless* q en el otro.

En general, no se puede inferir que *p leads-to* q se cumple en $[S_1 \parallel S_2]$ aún cumpliéndose en los dos procesos. Con respecto a la estrategia de superposición, mientras facilita la verificación, como contrapartida exige conocer en detalle las capas previamente desarrolladas, al tiempo que su tratamiento algebraico es limitado.

Acerca de la semántica operacional de los lenguajes concurrentes

La especificación de la semántica operacional ((Hennessy y Plotkin, 1979), (Plotkin, 1981)) de un lenguaje concurrente asumiendo *interleaving*, usa dos tipos de configuraciones: *configuraciones secuenciales*, propias de las computaciones de los procesos considerados aisladamente, y como novedad en relación a lo descripto precedentemente, *configuraciones concurrentes*, correspondientes a las distintas computaciones generadas por la intercalación no determinística de las instrucciones atómicas de los procesos. Las configuraciones concurrentes de un programa tienen una continuación sintáctica S_i por cada proceso P_i y un estado corriente σ, global o formado por la unión de los estados locales de cada proceso según el programa sea paralelo o distribuido, respectivamente. A continuación, mostramos algunas particularidades de las configuraciones concurrentes (y de la relación de transición ⟶ que las vincula) de las distintas clases de programas concurrentes que tratamos en este capítulo.

Una transición de un programa paralelo del lenguaje de variables compartidas, producida por ejemplo por la ejecución de un *await* en un proceso P_i, se representa de la siguiente manera:

- $([S_1 \,||\, \ldots \,||\, \text{await B} \longrightarrow \text{S end} ; T_i \,||\, \ldots \,||\, S_n], \sigma) \longrightarrow_{(i,\sigma)} ([S_1 \,||\, \ldots \,||\, T_i \,||\, \ldots \,||\, S_n], \sigma')$.

El *await* se consume en un paso y transforma el estado corriente σ en σ´. El indicador (i, σ) asociado a la transición explicita que la computación avanza por el proceso P_i. Las transiciones con otras instrucciones del lenguaje se representan de la misma forma. En todos los casos, sólo una continuación sintáctica se modifica.

En el caso de un programa paralelo con recursos de variables, las configuraciones concurrentes tienen un componente adicional, necesario para especificar el progreso a través de la ejecución de un *with* y su impacto en el estado del recurso correspondiente. Se utiliza un arreglo ρ[1:m] para indicar el estado de los m recursos del programa (ρ[j] = 0 significa que el recurso r_j está libre y ρ[j] = 1 que está ocupado). Las formas de las transiciones son:

- $([S_1 \,||\, \ldots \,||\, S_i \,||\, \ldots \,||\, S_n], \rho, \sigma) \longrightarrow_{(i,\sigma)} ([S_1 \,||\, \ldots \,||\, S'_i \,||\, \ldots \,||\, S_n), \rho, \sigma')$,
  si la continuación sintáctica S_i no comienza con un *with* y $(S_i , \sigma) \longrightarrow (S'_i , \sigma')$.
- $([S_1 \,||\, \ldots \,||\, \text{with } r_j \text{ when B do S endwith}_j ; T_i \,||\, \ldots \,||\, S_n], \rho, \sigma) \longrightarrow_{(i,\sigma)}$
  $([S_1 \,||\, \ldots \,||\, \text{S endwith}_j ; T_i \,||\, \ldots \,||\, S_n], \rho', \sigma)$,
  con $\rho'[j] = 1$ y $\rho'[h] = \rho[h]$ para todo $h \neq j$, si $\rho[j] = 0$ y B es verdadero en el estado σ.
- $([S_1 \,||\, \ldots \,||\, \text{endwith}_j ; T_i \,||\, \ldots \,||\, S_n], \rho, \sigma) \longrightarrow_{(i,\sigma)} ([S_1 \,||\, \ldots \,||\, T_i \,||\, \ldots \,||\, S_n], \rho', \sigma)$,
  con $\rho'[j] = 0$ y $\rho'[h] = \rho[h]$ para todo $h \neq j$.

En palabras, cuando se avanza por una instrucción que no es un *with*, la situación de los recursos no cambia. Al ejecutarse un *with*, el recurso correspondiente pasa a estar ocupado, y se libera cuando el *with* finaliza. La ejecución de un *with* no es atómica.

Con respecto a los programas distribuidos con comunicaciones sincrónicas, lo distintivo de una transición es que si es producto de la ejecución de una comunicación entre dos procesos, los dos procesos progresan simultáneamente. Dicha transición se representa genéricamente así:

- $([S_1 \,||\, \ldots \,||\, S_i \,||\, \ldots \,||\, S_j \,||\, \ldots \,||\, S_n], \sigma) \longrightarrow_{(i,j,\sigma)} ([S_1 \,||\, \ldots \,||\, S'_i \,||\, \ldots \,||\, S'_j \,||\, \ldots \,||\, S_n], \sigma')$.

Los procesos P_i y P_j avanzan a través de una comunicación en alguna de sus variantes (instrucciones de entrada y salida aisladas o en comandos guardados de una selección condicional o una repetición). σ' representa la unión de los estados de los procesos luego de la comunicación, y por lo tanto difiere de σ sólo en el estado del proceso receptor: si σ_e y σ_r son los estados del proceso emisor y el proceso receptor antes de la comunicación, respectivamente, entonces $\sigma_r' = \sigma_r[x|\sigma_e(e)]$, siendo $\sigma_e(e)$ el valor enviado y x la variable que lo recibe.

Finalmente, en cuanto al lenguaje distribuido con comunicaciones asincrónicas, las configuraciones concurrentes de un programa también incluyen un componente adicional, ahora para representar los mensajes enviados aún no recibidos. Se utiliza un arreglo γ[1:k], tal que cada γ[j] es una secuencia de mensajes no recibidos residentes en el canal j. Las transiciones correspondientes a las instrucciones de entrada y salida se representan del siguiente modo:

- $([S_1 \parallel \dots \parallel S_i \parallel \dots \parallel S_n], \gamma, \sigma) \longrightarrow_{(i, j, \sigma)} ([S_1 \parallel \dots \parallel S'_i \parallel \dots \parallel S_n], \gamma', \sigma)$,
  si P_i envía una expresión *e* a un determinado canal j. La secuencia $\gamma'[j]$ resulta de la concatenación de $\gamma[j]$ con el valor $\sigma(e)$.
- $([S_1 \parallel \dots \parallel S_i \parallel \dots \parallel S_n], \gamma, \sigma) \longrightarrow_{(i, j, \sigma)} ([S_1 \parallel \dots \parallel S'_i \parallel \dots \parallel S_n], \gamma', \sigma')$,
  si P_i recibe en una variable x el primer elemento de un determinado canal j, identificado con first($\gamma[j]$), La secuencia $\gamma'[j]$ resulta de la extracción de first($\gamma[j]$) de $\gamma[j]$, y $\sigma_i' = \sigma_i[x|\text{first}(\gamma[j])]$.

Las instrucciones de entrada/salida modifican los canales, quitando o agregando mensajes. Sólo la recepción de un mensaje por parte de un proceso altera su estado local.

Referencias bibliográficas

Ya en versiones previas de su artículo fundacional de 1969, C. Hoare menciona su intención de extender el método axiomático para los programas concurrentes. En (Hoare, 1972) presenta un primer conjunto de axiomas para el marco concurrente, que amplía en (Hoare, 1975), contemplando una regla para procesos disjuntos y otras con asunciones más generales, categorizadas en reglas para procesos que compiten, procesos que cooperan y procesos que se comunican.

Las axiomáticas descriptas en este capítulo aparecen en (Owicki y Gries, 1976a) y (Owicki y Gries, 1976b), en el caso de los programas paralelos con instrucciones de sincronización *await* y *with*, y (Apt, Francez y de Roever, 1980), en el caso de los programas distribuidos basados en el lenguaje CSP, el cual se introduce en (Hoare, 1978). Un método similar al de S. Owicki y D. Gries es (Lamport, 1977). Uno similar al de K. Apt, N. Francez y W. de Roever es (Levin y Gries, 1981), que no utiliza invariantes globales en el test de cooperación sino variables auxiliares globales.

(Clint, 1973) muestra por primera vez el uso de variables auxiliares en las axiomáticas para programas concurrentes. En (Brinch-Hansen, 1973) y (Hoare, 1974) se describe una instrucción de sincronización estructurada distinta del *with*, el *monitor*, que permite liberaciones temporarias de recursos; entre las axiomáticas que lo utilizan figura (Nielsen y Black, 1985). (Francez, 1989) describe una extensión al método de verificación de programas distribuidos con comunicaciones sincrónicas entre un número arbitrario de procesos. Una generalización de los métodos de prueba de correctitud parcial para programas tanto paralelos como distribuidos es (Lamport, 1990), centrada en la invariancia, que se define como una relación entre los estados de un programa antes y después de la ejecución de toda operación atómica.

(Lamport, 2015) presenta algunos hitos en la evolución del tratamiento de la correctitud de los programas concurrentes; en particular, destaca como iniciales a los trabajos de E. Dijkstra (Dijkstra, 1965) y (Dijkstra, 1968) sobre la exclusión mutua y la sincronización entre procesos, en ocasión de la construcción del sistema operativo THE. (Schneider, 2018) también trata la historia de la verificación de los programas concurrentes, pero centrada en la evolución de las

definiciones de las propiedades *safety* y *liveness*. Otra publicación sobre el camino recorrido en el marco de la concurrencia es (Jones, 1992), la cual destaca los hitos más relevantes del desarrollo sistemático de programas.

El lector interesado en profundizar en distintas técnicas de verificación de programas paralelos y distribuidos puede consultar (Barringer, 1985) y (Schneider y Andrews, 1986). También el libro de reciente aparición (Malkhi, 2019), que reúne artículos de distintos autores sobre los numerosos e importantes aportes de L. Lamport en distintos campos de la programación concurrente (verificación, especificación, algorítmica, etc).

Se detallan a continuación las referencias bibliográficas citadas, incluidas las de las notas adicionales:

Apt, K., Francez, N. y de Roever, W. (1980). *A proof system for communicating sequential processes*. ACM Trans. Prog. Lang. Syst., 2, 3, 359-385.

Barringer, H. (1985). *A survey of verification techniques for parallel programs.* Lecture Notes in Computer Science, 191. Springer-Verlag.

Brinch-Hansen, P. (1973). *Operating systems principles.* Englewood Cliffs.

Chandy, K. y Misra J. (1988). *Parallel program design: a foundation*. Addison-Wesley.

Clint, M. (1973). *Program proving: coroutines.* Acta Informática, 2, 1, 50-63.

Dijkstra, E. (1965). *Solution of a problem in concurrent programming control*. Comm. ACM, 8, 9, 569.

Dijkstra, E. (1968). *Cooperating sequential processes*. Programming Languages, 43-112. Academic Press.

Francez, N. (1989). *Cooperating proofs for distributed programs with multiparty interactions*. Information Processing Lett., 32, 235-242.

Hennessy, M. y Plotkin, G. (1979). *Full abstraction for a simple programming language*. Lecture Notes in Computer Science, 74, 108-120.

Hoare, C. (1972). *Towards a theory of parallel programming.* Operating Systems Techniques, 61-71. Academic Press.

Hoare. C. (1974). *Monitors: an operating system structuring concept.* Comm. ACM, 17, 10, 549-557.

Hoare. C. (1975). *Parallel programming: an axiomatic approach.* Computer Languages, 1, 2, 151-160.

Hoare, C. (1978). *Communicating secuential processes*. Comm. ACM, 21, 8, 666-677.

Jones, C. (1981). *Development methods for computer programs including a notion of interference.* Technical Monograph PRG-25, Oxford University.

Jones, C. (1992). *On the search for tractable ways of reasoning about programs.* Technical Report UMCS-92-4-4, University of Manchester.

Lamport, L. (1977). *Proving the correctness of multiprocess programs.* IEEE Trans. Software Engineering, 3, 2, 125-143.

Lamport, L. (1990). *win and sin: predicate transformers for concurrency.* ACM Trans. Prog. Lang. Syst., 12, 3, 396-428.

Lamport, L. (2015). *The computer science of concurrency: the early years.* Comm. ACM, 58, 6, 71-76.

Levin, G. y Gries, D. (1981). *Proof techniques for communicating sequential processes*. Acta Informática, 15, 281-302.

Malkhi, D. (2019). *Concurrency. The works of Leslie Lamport*. ACM Books.

Nielsen, L. y Black, A. (1985). *Proving monitor proof rules*. Technical Report 85-08-01. Department of Computer Science, University of Washington, Seattle WA.

Owicki, S. y Gries, D. (1976a). *An axiomatic proof technique for parallel programs*. Acta Informatica, 6, 319-340.

Owicki, S. y Gries, D. (1976b). *Verifying properties of parallel programs: an axiomatic approach*. Comm. ACM, 19, 5, 279-285.

Plotking, G. (1981). *A structural approach to operational semantics*. Tech. Rep., Dep. of Computer Science, Aarhus Univ.

Schlichting, R. y Schneider, F. (1984). *Message passing for distributed programming*. ACM Trans. Prog. Lang. Syst., 6, 3, 402-431.

Schneider, F. (2018). *History and context for defining liveness.* ACM SIGACT News, 49, 4, 60-63.

Schneider, F y Andrews, G. (1986). *Concepts for concurrent programming.* Current Trends in Concurrency, Lecture Notes in Computer Science, 224. Springer-Verlag.

Xu, Q., de Roever, W y He, J. (1997). *The rely-guarantee method for verifying shared variable concurrent pograms*. Formal Aspects of Computing, 9, 2, 149-174.

Ejercicios

1. Dada una función entera f con un cero (es decir un x tal que f(x) = 0), el siguiente programa paralelo, con variables compartidas *ok* y *turno*, lo obtiene en la variable *a* si x > 0 (mediante el proceso S_1) o en la variable b si $x \leq 0$ (mediante el proceso S_2):

```
ok := false ; turno := 1 ;
[S1 :: a := 0 ;                                    S2 :: b := 1 ;
      while ¬ok do                                       while ¬ok do
      await turno = 1 ⟶ turno := 2 end   ||              await turno = 2 ⟶ turno := 1 end
      a := a + 1 ;                                       b := b – 1 ;
      if f(a) = 0 then ok := true fi                     if f(b) = 0 then ok := true fi
                  od                                                 od]
```

   Explicar por qué el programa no es totalmente correcto, y corregirlo.

Resolución

Puede haber *deadlock* en el programa. Supongamos que el cero de la función es positivo. Puede suceder que S_1 encuentre el cero, y antes de asignar *true* a *ok* y terminar, S_2 asigne

1 a *turno*, lo que lo bloquea indefinidamente. Para evitar esta situación, hay que agregar después de cada *while* una asignación que modifique el valor de *turno*:

```
ok := false ; turno := 1 ;
[S1 :: a := 0 ;                                          S2 :: b := 1 ;
      while ¬ok do                                             while ¬ok do
      await turno = 1 ⟶ turno := 2 end   ||                  await turno = 2 ⟶ turno := 1 end
      a := a + 1 ;                                             b := b – 1 ;
      if f(a) = 0 then ok := true fi                           if f(b) = 0 then ok := true fi
                  od                                                       od]
      turno := 2 ;                                             turno := 1]
```

2. ¿Por qué es errónea la siguiente modificación al programa paralelo, presentado en el capítulo, que calcula el factorial?

```
Sfac2 :: i := 1 ;  j := N ; n := N ;
      [S1 :: while i + 1 < j do                              S2 :: while j – 1 > i do
         await true → i := i + 1 ; n := n . i end   ||       await true → j := j – 1 ; n := n . j end
                        od                                                od]
```

3. Justificar por qué el no determinismo que se manifiesta en la ejecución de los programas paralelos descriptos, asumiendo la propiedad de progreso fundamental, es acotado.

4. Detallar la prueba de correctitud parcial del programa paralelo que calcula el factorial, planteada en el Ejemplo 16. Es decir: probar la correctitud y la libertad de interferencia de las *proof outlines*, y las implicaciones requeridas entre la precondición y la postcondición del programa y las conjunciones de las precondiciones y las postcondiciones de los procesos.

5. La postcondición q de la fórmula {p} $S_{|A}$ {q}, obtenida de la aplicación de la regla AUX, no incluye variables auxiliares. En cambio, esto sí se permite para el caso de la precondición p. Explicar por qué.

6. Probar la fórmula de correctitud parcial {x = y} [x := x + 1 || y := y + 1] {x = y}. ¿Se puede lograr sin recurrir al uso de variables auxiliares? Justificar la respuesta.

Resolución

Si bien los procesos son disjuntos, la precondición y la postcondición del programa contienen variables de ambos, lo que obliga a utilizar variables auxiliares para relacionarlas. Proponemos las siguientes *proof outlines*, con una variable auxiliar z inicializada con la asignación ***z := x***:

{x = z} {y = z}
[x := x + 1 || y := y + 1]
{x = z + 1} {y = z + 1}

Las *proof outlines* son libres de interferencia y además se cumple:

{x = y} z := x {x = z ∧ y = z}
(x = z + 1 ∧ y = z + 1) ⟶ x = y

La prueba se completa aplicando la regla AUX.

7. Detallar la prueba de correctitud parcial con variables auxiliares planteada en el Ejemplo 15 (correspondiente a la fórmula de correctitud {x = 0} [S_1 :: x := x + 1 || S_2 :: x := x + 1] {x = 2}).

8. Probar las siguientes fórmulas de correctitud parcial:
   a. {true} [x := 0 || x := x + 1] {x = 0 ∨ x = 1}
   b. {true} [x := x + 2 ; x := x + 2 || x := 0] {x = 0 ∨ x = 2 x = 0 ∨ x = 4}

9. Demostrar formalmente que sin variables auxiliares no se puede probar la fórmula de correctitud parcial {true} [x := x + 2 || x := 0] {x = 0 ∨ x =2}.

Resolución

Intuitivamente, no se puede expresar como postcondición del proceso *x := 0*, sólo en términos de la variable x, que aún falta ejecutarse el proceso *x := x + 2*. Supongamos que la fórmula se puede probar sin variables auxiliares, para llegar a una contradicción. Asumimos entonces que existen las siguientes *proof outlines* libres de interferencia:

{p_1} x := x + 2 {q_1} y {p_2} x := 0 {q_2}

con:

true ⟶ (p_1 ∧ p_2) y (q_1 ∧ q_2) ⟶ (x = 0 ∨ x = 2).

Queda:

1) p_1 = p_2 = true, por true ⟶ (p_1 ∧ p_2).
2) {true} x := x + 2 {q_1}, por (1) y {p_1} x := x + 2 {q_1}.
3) true ⟶ q_1[x|x + 2], por (2) y ASI.
4) q_1[x|x + 2], por (3).
5) q_1 = true, por (4), dado que x varía en los enteros.

6) {true} x := 0 {q_2}, por (1) y {p_2} x := 0 {q_2}.
7) true $\longrightarrow$ $q_2[x|0]$, por (6) y ASI.
8) $q_2[x|0]$, por (7).
9) {true $\wedge$ q_2} x := x + 2 {q_2}, por la libertad de interferencia de las *proof outlines*.
10) (true $\wedge$ q_2) $\longrightarrow$ $q_2[x|x + 2]$, por (9) y ASI.
11) $q_2 \longrightarrow q_2[x|x + 2]$, por (10).
12) $\forall x$: (($x \geq 0 \wedge$ par(x)) $\longrightarrow$ q_2), por (8) y (11), con par(x) verdadero sii x es par.
13) Finalmente, por (5), (12) y ($q_1 \wedge q_2$) $\longrightarrow$ ($x = 0 \vee x = 2$), se obtiene el siguiente predicado falso: $\forall x$: (($x \geq 0 \wedge$ par(x)) $\longrightarrow$ ($x = 0 \vee x = 2$)).

10. Demostrar formalmente que sin variables auxiiares no se puede probar la fórmula de correctitud parcial considerada en el ejercicio 7.

11. Detallar la prueba de no divergencia del programa paralelo del productor-consumidor planteada en el Ejemplo 18.

12. Probar, dada la especificación (x = 0, x = 1), la correctitud parcial y la ausencia de *deadlock* del programa [await x = 1 $\longrightarrow$ skip end || x := 1].

Resolución

Para la prueba de correctitud parcial proponemos las siguientes *proof outlines*:

{x = 0 $\vee$ x = 1}		{x = 0}
[await x = 1 $\longrightarrow$ skip end	||	x := 1]
{x = 1}		{x = 1}

claramente correctas y libres de interferencia. Además vale x = 0 $\longrightarrow$ ((x = 0 $\vee$ x = 1) $\wedge$ x = 0) y (x = 1 $\wedge$ x = 1) $\longrightarrow$ x = 1. Las mismas *proof outlines* sirven para probar la ausencia de *deadlock* en el programa. La única imagen semántica a considerar resulta falsa:

((x = 0 $\vee$ x = 1) $\wedge$ x $\neq$ 1) $\wedge$ x = 1

13. Probar que no es posible que ocurra el tercer caso de *deadlock* indicado en la prueba de ausencia de *deadlock* del programa paralelo del productor-consumidor, planteada en el Ejemplo 19.

Ayuda

La imagen semántica asociada a la situación mencionada, en la que el productor está bloqueado en su *await* y el consumidor termina, es la siguiente:

$$(0 \leq in - out \leq b \wedge 1 \leq i \leq n \wedge 1 \leq j \leq n + 1 \wedge b > 0 \wedge i = in + 1 \wedge in - out \geq b) \wedge$$
$$(0 \leq in - out \leq b \wedge 1 \leq i \leq n + 1 \wedge j = n + 1 \wedge b > 0 \wedge j = out + 1)$$

Hay que probar que dicho predicado es falso.

14. Probar que los dos procesos del siguiente programa paralelo nunca están simultáneamente dentro de sus respectivas secciones A y B:

```
b1 := true ; b2 := false ;
[P1 :: while true do                          P2 :: while true do
        await b1 ⟶ b1 := false end ;                  await b2 ⟶ b2 := false end ;
        A1 ;                                          A2 ;
        await ¬b2 ⟶ b2 := true end ;    ||            await ¬b1 ⟶ b1 := true end ;
        await b1 ⟶ b1 := false end ;                  await b2 ⟶ b2 := false end ;
        B1 :                                          B2 ;
        await ¬b2 ⟶ b2 := true end                    await ¬b1 ⟶ b1 := true end
                od                                            od]
```

¿Puede suceder que algún proceso nunca ingrese a sus secciones A y B? Justificar la respuesta.

15. Justificar por qué en la regla WITH, el invariante del recurso utilizado en la premisa de la regla no se traslada a la conclusión.

16. Detallar la prueba de exclusión mutua, planteada en el Ejemplo 21, para el programa paralelo con recursos que implementa el problema de la cena de los filósofos.

17. Una alternativa a la utilización de un invariante global para la verificación de programas distribuidos es la utilización de variables auxiliares globales en las *proof outlines*. Probar de esta manera el programa [P_1 :: x := 0 ; P_2 ! x ; x := x + 1 ; P_2 ! x || P_2 :: P_1 ? y ; P_1 ? y] (ya probado en el Ejemplo 24 con un invariante global).

Resolución

Utilizamos una variable auxiliar global z, inicializada en 0, para no considerar en el test de cooperación los dos pares de instrucciones de entrada/salida del programa que coinciden sintácticamente pero no semánticamente:

```
{z = 0}                {z = 0}
```

[x := 0 ; || P_1 ? (y, z) ;
{x = 0 ∧ z = 0} || {y = 0 ∧ z = 1}
P_2 ! (x, z + 1) ; || P_1 ? (y, z)]
{x = 0 ∧ z = 1} || {y = 1 ∧ z = 2}
x := x + 1 ;
{x = 1 ∧ z = 1}
P_2 ! (x, z + 1) ;
{x = 1 ∧ z = 2}

Claramente, las *proof outlines* son correctas.

Con respecto al test de cooperación, en el caso de los pares de instrucciones de entrada/salida que coinciden semánticamente, por aplicación del axioma COM se cumplen las fórmulas:

{x = 0 ∧ z = 0 ∧ z = 0} P_2 ! (x, z + 1) || P_1 ? (y, z) {x = 0 ∧ z = 1 ∧ y = 0 ∧ z = 1}
{x = 1 ∧ z = 1 ∧ y = 0 ∧ z = 1} P_2 ! (x, z + 1) || P_1 ? (y, z) {x = 1 ∧ z = 2 ∧ y = 1 ∧ z = 2}

En el caso de los pares de instrucciones de entrada/salida que no coinciden semánticamente, notar que las precondiciones correspondientes:

x = 0 ∧ z = 0 ∧ y = 0 ∧ z = 1
x = 1 ∧ z = 1 ∧ z = 0

son falsas, lo que permite probar las fórmulas asociadas trivialmente.

En general, en esta alternativa de prueba es necesario el chequeo de libertad de interferencia de las *proof outlines*, dado que se introducen variables compartidas. En este caso no es necesario, porque la variable z sólo se modifica en las comunicaciones.

18. Indicar con qué nivel de *fairness* fuerte (proceso, canal, comunicación) el siguiente programa distribuido [P_1 || P_2 || P_3] no diverge, siendo:

P_1 :: a := true ; do a ; P_2 ? a → skip od
P_2 :: b := true ; go := true ; do b ; P_1 ! go → if go → skip or ¬go → b := false fi
or b ; P_3 ? go → skip od
P_3 :: P_2 ! false

Resolución

Los procesos P_1 y P_2 se pueden comunicar varias veces por medio del mensaje *true*.

Pero una vez que el proceso P_3 le envía a P_2 el mensaje *false*, P_3 termina, P_2 en su próxima iteración le envía a P_1 el mensaje *false*, y finalmente en sus respectivas próximas iteraciones tanto P_1 como P_2 terminan.

En este caso, entonces, alcanza con *fairness* fuerte de nivel proceso para que el programa no diverja, porque permite que P_3 pueda comunicarse con P_2 alguna vez.

19. Indicar con qué nivel de *fairness* fuerte el siguiente programa distribuido [P_1 || P_2 || P_3] no diverge, siendo:

    P_1 :: a_2 := true ; a_3 := true ;
        do a_2 ; P_2 ? a_2 → skip or a_3 ; P_3 ? a_3 → skip od
    P_2 :: b := true ; go2 := true ; x := 0 ;
        do b ; P_1 ! go2 → if ¬go2 → b := false or go2 → skip fi
        or b ; P_3 ! x → go2 := false od
    P_3 :: c := true ; go3 := true ; y := 0 ;
        do c ; P_1 ! go3 → if ¬go3 → c := false or go3 → skip fi
        or c ; P_2 ? y → go3 := false od

20. Completar la prueba de ausencia de *deadlock* en el programa que propaga un valor, planteada en el Ejemplo 26.

Conclusiones

Reflexionando sobre la enseñanza de la programación, E. Dijkstra remarcaba dos aspectos cruciales de las computadoras. Por un lado, su poder brutal, demandante de jerarquías conceptuales muy profundas, no utilizadas antes de su aparición. Y por otro lado su incómoda característica, propia de los artefactos discretos, de que la modificación de un solo bit puede tener consecuencias drásticas, imposibilitando de este modo el manejo de métricas razonables que relacionen causas con consecuencias. Dada esta situación, insistía en la necesidad imperiosa de formar al programador en habilidades apropiadas para la construcción sistemática de programas, postulado que la comunidad informática fue incorporando progresivamente.

Este libro también sigue tal postulado, y con una motivación reforzada al menos por dos circunstancias de estos tiempos. Una, consecuencia del vertiginoso avance de la tecnología para hacer inferencias a gran escala, es el surgimiento de nuevas disciplinas que exigen sólidos conocimientos matemáticos (disciplinas en las que ya están trabajando muchos profesionales). La otra circunstancia, tan o más relevante que la primera, la constituyen los cambios en la currícula de ciencias de la computación durante los últimos años, reduciendo los contenidos formales.

Una teoría de programas incluye un método de especificación, para establecer y combinar requerimientos; un método de razonamiento sobre especificaciones, para evaluar diseños alternativos de programas; un método de construcción de programas, asociado con un método de verificación para asegurar la satisfacción de la especificación; y un método de transformación de programas, para mejorar su eficiencia en términos de la máquina objeto disponible. Frente a este amplio escenario, en este trabajo nos hemos concentrado en la verificación de programas, pero de un modo que permita adquirir adicionalmente conocimientos, y sobre todo tomar conciencia, de los principios y las herramientas con que contamos para construir programas correctos. Evitando sobreabundancia de material, que muchas veces resulta contraproducente, nos hemos enfocado en las nociones fundamentales de la verificación axiomática, y hemos querido presentar los temas de la manera más simple posible, con ejemplos sencillos, analizando las propiedades de programas más representativas y algunos componentes básicos de la metateoría, con particular énfasis en la composicionalidad por su relación con la escalabilidad y así la factibilidad del enfoque analizado.

Aun no siguiendo al pie de la letra la metodología presentada, la utilización de las técnicas por ella sugeridas, con sentido común combinado con el valor agregado de la experiencia, contribuye sobremanera a la construcción de software confiable. Si recorriendo este trabajo el lector se ha convencido de esta idea, entonces el propósito principal de quien lo ha escrito se habrá cumplido.

ANEXO 1
Axiomas y reglas de los métodos presentados

Método para los programas secuenciales determinísticos

1. *Axioma del skip* (SKIP)

   $\{p\}\ skip\ \{p\}$

2. *Axioma de la asignación* (ASI)

   $\{p[x|e]\}\ x := e\ \{p\}$

3. *Regla de la secuencia* (SEC)

$$\frac{\{p\}\ S_1\ \{r\},\ \{r\}\ S_2\ \{q\}}{\{p\}\ S_1\ ;\ S_2\ \{q\}}$$

4. *Regla del condicional* (COND)

$$\frac{\{p \wedge B\}\ S_1\ \{q\},\ \{p \wedge \neg B\}\ S_2\ \{q\}}{\{p\}\ \text{if B then } S_1 \text{ else } S_2 \text{ fi } \{q\}}$$

5. *Regla de la repetición* (REP)

$$\frac{\{p \wedge B\}\ S\ \{p\}}{\{p\}\ \text{while B do S od } \{p \wedge \neg B\}}$$

6. *Regla de consecuencia* (CONS)

$$\frac{p \rightarrow p_1,\ \{p_1\}\ S\ \{q_1\},\ q_1 \rightarrow q}{\{p\}\ S\ \{q\}}$$

7. *Regla de instanciación* (INST):

$$\frac{f(X)}{f(c)}$$

   c pertenece al dominio de X.

8. *Regla de la disyunción* (OR)

$$\frac{\{p\}\ S\ \{q\},\ \{r\}\ S\ \{q\}}{\{p \vee r\}\ S\ \{q\}}$$

9. *Regla de la conjunción* (AND)

$$\frac{\{p_1\}\ S\ \{q_1\},\ \{p_2\}\ S\ \{q_2\}}{\{p_1 \wedge p_2\}\ S\ \{q_1 \wedge q_2\}}$$

10. *Axioma de invariancia* (AINV)

$\{p\}\ S\ \{p\}$

Ninguna variable libre de p puede ser modificada por S.

11. *Regla de invariancia* (RINV):

$$\frac{\{p\}\ S\ \{q\}}{\{r \wedge p\}\ S\ \{r \wedge q\}}$$

Ninguna variable libre de r puede ser modificada por S.

12. *Regla de la terminación* (REP*)

$$\frac{\langle p \wedge B\rangle\ S\ \langle p\rangle,\ \langle p \wedge B \wedge t = Z\rangle\ S\ \langle t < Z\rangle,\ p \rightarrow t \geq 0}{\langle p\rangle \text{ while } B \text{ do } S \text{ od } \langle p \wedge \neg B\rangle}$$

t varia en $(N_0, <)$.

Método para los programas secuenciales no determinísticos

13. *Regla del condicional no determinístico* (NCOND)

$$\frac{\{p \wedge B_i\}\ S_i\ \{q\},\ i = 1, \ldots, n}{\{p\} \text{ if } B_1 \rightarrow S_1 \text{ or} \ldots \text{or } B_n \rightarrow S_n \text{ fi } \{q\}}$$

14. *Regla de la repetición no determinística* (NREP)

$$\frac{\{p \wedge B_i\}\ S_i\ \{p\},\ i = 1, \ldots, n}{\{p\} \text{ do } B_1 \rightarrow S_1 \text{ or} \ldots \text{or } B_n \rightarrow S_n \text{ od } \{p \wedge \bigwedge_{i = 1,n} \neg B_i\}}$$

15. *Regla de la no divergencia no determinística* (NREP*)

$$\frac{\langle p \wedge B_i\rangle\ S_i\ \langle p\rangle,\ i = 1, \ldots, n,\ \langle p \wedge B_i \wedge t = Z\rangle\ S_i\ \langle t < Z\rangle,\ i = 1, \ldots, n,\ p \rightarrow t \geq 0}{\langle p\rangle \text{ do } B_1 \rightarrow S_1 \text{ or} \ldots \text{or } B_n \rightarrow S_n \text{ od } \langle p \wedge \bigwedge_{i = 1,n} \neg B_i\rangle}$$

t varia en $(N_0, <)$.

16. *Regla del condicional no determinístico sin falla* (NCOND*)

$$\frac{p \rightarrow \bigvee_{i = 1,n} B_i,\ \langle p \wedge B_i\rangle\ S_i\ \langle q\rangle,\ i = 1, \ldots, n}{\langle p\rangle \text{ if } B_1 \rightarrow S_1 \text{ or} \ldots \text{or } B_n \rightarrow S_n \text{ fi } \langle q\rangle}$$

17. Axioma de la asignación aleatoria (NASI)

$\{\forall x: (x \geq 0 \rightarrow p)\}\ x := ?\ \{p\}$

18. *Regla de la no divergencia no determinística generalizada* (NREPG*)

$$\frac{\langle p \wedge B_i\rangle\ S_i\ \langle p\rangle,\ i = 1, \ldots, n,\ \langle p \wedge B_i \wedge t = w\rangle\ S_i\ \langle t < w\rangle,\ i = 1, \ldots, n,\ p \rightarrow t \in W}{\langle p\rangle \text{ do } B_1 \rightarrow S_1 \text{ or} \ldots \text{or } B_n \rightarrow S_n \text{ od } \langle p \wedge \bigwedge_{i=1,n} \neg B_i\rangle}$$

t varia en algún (W, <).

19. *Regla de la no divergencia no determinística asumiendo fairness débil* (NREPD*)

Elegir un conjunto bien fundado W, establecer un invariante p y un variante t definido en W del DO, y por cada valor w no minimal de W adoptado por t definir una partición del conjunto de direcciones del DO (conjuntos D_w no vacío y D'_w), tal que:

$\langle p \wedge B_i\rangle\ S_i\ \langle p\rangle,\ i = 1, \ldots, n$

$\langle p \wedge B_i \wedge t = w\rangle\ S_i\ \langle t < w\rangle,\ i \in D_w$

$\langle p \wedge B_i \wedge t = w\rangle\ S_i\ \langle t \leq w\rangle,\ i \in D'_w$

$p \rightarrow t \in W$

20. *Regla de la no divergencia no determinística asumiendo fairness fuerte* (NREPF*)

Transformar el programa agregándole asignaciones aleatorias para eliminarle todas sus computaciones *unfair* (planificador *fair*) y probar no divergencia en el programa transformado con las reglas con asignaciones aleatorias.

Método para los programas paralelos

21. *Regla del await* (AWAIT)

$$\frac{\{p \wedge B\}\ S\ \{q\}}{\{p\} \text{ await } B \rightarrow S \text{ end } \{q\}}$$

22. *Regla de la composición paralela* (PAR)

$\{p_i\}\ S_i^*\ \{q_i\}$, i = 1, …, n, son *proof outlines* libres de interferencia,

$$\frac{p \longrightarrow \bigwedge_{i=1,n} p_i,\ \bigwedge_{i=1,n} q_i \longrightarrow q}{\{p\}\ [\ ||_{\ i=1,n}\ S_i\ ]\ \{q\}}$$

23. *Regla de las variables auxiliares* (AUX)

$$\frac{\{p\}\ S\ \{q\}}{\{p\}\ S_{|A}\ \{q\}}$$

$S_{|A}$ es el programa original y S es el programa ampliado con variables auxiliares de A.

24. *Regla de la no divergencia paralela* (PAR*)

$\langle p_i\rangle\ S_i^{**}\ \langle q_i\rangle$, i = 1, …, n, son *proof outlines* fuertemente libres de interferencia,

$$\frac{p \longrightarrow \bigwedge_{i=1,n} p_i,\ \bigwedge_{i=1,n} q_i \longrightarrow q}{\langle p\rangle\ [\ ||_{\ i=1,n}\ S_i\ ]\ \langle q\rangle}$$

25. *Regla de la ausencia de deadlock en programas paralelos* (DEADLOCK)

    Dado $[\|\|_{i = 1,n} S_i]$ y una precondición p, obtener *proof outlines* $\{p_i\}\ S_i^*\ \{q_i\}$ libres de interferencia tales que $p \longrightarrow \bigwedge_{i = 1,n} p_i$, detectar en ellas todas las configuraciones que representan casos de *deadlock*, caracterizarlas mediante imágenes semánticas y probar que son falsas.

26. *Regla del with* (WITH)

$$\frac{\{I_r \wedge p \wedge B\}\ S\ \{I_r \wedge q\}}{\{p\}\ \text{with r when B do S endwith}\ \{q\}}$$

27. *Regla de la composición paralela con recursos* (RPAR)

$$\frac{\{p_i\}\ S_i^*\ \{q_i\},\ i = 1, \ldots, n,\ \text{son } \textit{proof outlines}\ \text{que utilizan } I_r \qquad p \longrightarrow (I_r \wedge \bigwedge_{i = 1,n} p_i),\ (I_r \wedge \bigwedge_{i = 1,n} q_i) \longrightarrow q}{\{p\}\ [\|\|_{i = 1,n}\ S_i]\ \{q\}}$$

28. *Regla de la no divergencia paralela con recursos* (RPAR*)

    Como la regla PAR* pero sin el chequeo de libertad de interferencia de las *proof outlines*.

29. *Regla de la ausencia de deadlock con recursos* (RDEADLOCK)

    Como la regla DEADLOCK pero sin el chequeo de libertad de interferencia de las *proof outlines* y con la inclusión del invariante del recurso en las imágenes semánticas.

Método para los programas distribuidos

30. *Axioma de la instrucción de entrada* (IN)

    $\{p\}\ P_i\ ?\ x\ \{q\}$

31. *Axioma de la instrucción de salida* (OUT)

    $\{p\}\ P_j\ !\ e\ \{p\}$

32. *Regla del condicional con comunicaciones* (DCOND)

$$\frac{\{p \wedge B_i\}\ \alpha_i\ ;\ S_i\ \{q\},\ i = 1, \ldots, n}{\{p\}\ \text{if } B_1\ ;\ \alpha_1 \rightarrow S_1\ \text{or} \ldots \text{or}\ B_n\ ;\ \alpha_n \rightarrow S_n\ \text{fi}\ \{q\}}$$

33. *Regla de la repetición con comunicaciones* (DREP)

$$\frac{\{p \wedge B_i\}\ \alpha_i\ ;\ S_i\ \{p\},\ i = 1, \ldots, n}{\{p\}\ \text{do } B_1\ ;\ \alpha_1 \rightarrow S_1\ \text{or} \ldots \text{or}\ B_n\ ;\ \alpha_n \rightarrow S_n\ \text{do}\ \{p \wedge \bigwedge_i \neg B_i\}}$$

34. *Axioma de comunicación* (COM)

    $\{p[x|e]\}\ P_i\ ?\ x\ \|\|\ P_j\ !\ e\ \{p\}$

35. *Regla de la composición distribuida* (DIST)

$\{p_i\}\ S_i^*\ \{q_i\}$, i = 1, …, n, son *proof outlines* que cooperan con respecto a IG
$p \longrightarrow (\Lambda_{i = 1,n}\ p_i \wedge IG),\ (\Lambda_{i = 1,n}\ q_i \wedge IG) \longrightarrow q$

$$\{p\}\ [\ ||_{\ i = 1,n}\ P_i\ ]\ \{q\}$$

Ninguna variable de IG puede ser modificada fuera de las secciones atómicas.

36. *Regla de la no divergencia con comunicaciones* (DREP*)

$\langle p \wedge B_i \rangle\ \alpha_i\ ;\ S_i\ \langle p \rangle$, i = 1, …, n, $\langle p \wedge B_i \wedge t = Z \rangle\ \alpha_i\ ;\ S_i\ \langle t < Z \rangle$, i = 1, …, n, $p \rightarrow t \geq 0$

$$\langle p \rangle\ \text{do}\ B_1 \rightarrow S_1\ \text{or} \ldots \text{or}\ B_n \rightarrow S_n\ \text{od}\quad \langle p \wedge \Lambda_{i = 1,n}\ \neg B_i \rangle$$

t varia en $(N_0, <)$.

37. *Regla de la no divergencia distribuida* (DIST*)

$\langle p_i \rangle\ S_i^{**}\ \langle q_i \rangle$, i = 1, …, n, son *proof outlines* que cooperan con respecto a IG
$p \longrightarrow (\Lambda_{i = 1,n}\ p_i \wedge IG),\ (\Lambda_{i = 1,n}\ q_i \wedge IG) \longrightarrow q$

$$\langle p \rangle\ [\ ||_{\ i = 1,n}\ P_i\ ]\ \langle q \rangle$$

Ninguna variable de IG puede ser modificada fuera de las secciones atómicas.

38. *Regla del condicional con comunicaciones sin falla* (DCOND*)

$p \rightarrow V_{i = 1,n}\ B_i,\ \{p \wedge B_i\}\ \alpha_i\ ;\ S_i\ \{q\}$, i = 1, …, n

$$\{p\}\ \text{if}\ B_1\ ;\ \alpha_1 \rightarrow S_1\ \text{or} \ldots \text{or}\ B_n\ ;\ \alpha_n \rightarrow S_n\ \text{fi}\ \{q\}$$

39. *Regla de la ausencia de deadlock en programas distribuidos* (DDEADLOCK)
Obtener *proof outlines* con secciones atómicas y que cooperan con respecto a un invariante global IG, detectar en ellas todas las configuraciones que representan casos de *deadlock*, caracterizarlas mediante imágenes semánticas y probar que son falsas.

ANEXO 2
Ejemplos de aplicación desarrollados

Método para los programas secuenciales determinísticos

1. Prueba de correctitud parcial de un programa que intercambia los valores de dos variables.
2. Prueba de correctitud parcial de un programa que calcula el valor absoluto.
3. Prueba de correctitud parcial de un programa que calcula la división entera.
4. Prueba de terminación de un programa que calcula la división entera.
5. Desarrollo sistemático de un programa que suma los elementos de un arreglo de números enteros.

Método para los programas secuenciales no determinísticos

6. Prueba de correctitud parcial de un programa que obtiene algún divisor.
7. Prueba de ausencia de falla en un programa que obtiene algún divisor.
8. Prueba de no divergencia de un programa que calcula el máximo común divisor.
9. Prueba de no divergencia de un programa que devuelve un número entre 0 y N.
10. Desarrollo sistemático de un programa que busca el menor elemento común de tres arreglos de números enteros.
11. Prueba de no divergencia de un programa con una asignación aleatoria.
12. Prueba de no divergencia de un programa asumiendo *fairness* débil.
13. Prueba de no divergencia de un programa asumiendo *fairness* fuerte

Método para los programas paralelos

14. Prueba de correctitud parcial de un programa de suma.
15. Prueba de correctitud parcial de otro programa de suma.
16. Prueba de correctitud parcial de un programa que calcula el factorial.
17. Prueba de no divergencia de un programa de resta.
18. Prueba de no divergencia del programa del productor-consumidor.

19. Prueba de ausencia de *deadlock* en el programa del productor-consumidor.
20. Prueba de correctitud parcial de un esquema de programa que implementa un semáforo binario.
21. Prueba de exclusión mutua en el programa que implementa el problema de la cena de los filósofos.
22. Desarrollo sistemático de un programa que busca el primer elemento positivo de un arreglo de números enteros.

Método para los programas distribuidos

23. Prueba de correctitud parcial de un programa que propaga un valor.
24. Prueba de correctitud parcial de un programa que devuelve el número uno.
25. Prueba de no divergencia de un programa con una repetición de diez iteraciones.
26. Prueba de ausencia de *deadlock* en un programa que propaga un valor.

El autor

Ricardo Rosenfeld obtuvo el título de Calculista Científico de la Facultad de Ciencias Exactas de la Universidad Nacional de La Plata (UNLP), Argentina, en 1983, y complementó sus estudios en el Instituto de Tecnología Technión, Israel, entre 1988 y 1991 (Maestría en Ciencias de la Computación).

Desde 1991 se desempeña como Profesor Adjunto en la UNLP (primero en la Facultad de Ciencias Exactas y después en la Facultad de Informática), en las áreas de Teoría de la Computación y Verificación de Programas, y desde 2021 como Profesor e Investigador en la Universidad Abierta Interamericana (UAI), en el área de Métodos Formales en la Ingeniería de Software.

Previamente, entre 1984 y 1990, fue docente en la Facultad de Ciencias Exactas de la UNLP (lenguajes y metodologías de programación), en la Universidad de Buenos Aires (derivación y verificación de programas), en la Escuela Superior Latinoamericana de Informática (algorítmica y teoría de compiladores), y en el Instituto de Tecnología Technión de Israel (programación).

En el ámbito de la industria informática, fue Líder de Proyectos y Responsable del Area de Arquitectura Informática en Telefónica Argentina, entre los años 1991 y 1997, y desde 1997 hasta 2023 formó parte de Grupo Practia, empresa de Tecnología de la Información con oficinas en América y Europa, donde se desempeñó como Consultor, Líder de Proyectos, y desde 1999 como Director y Socio del Grupo.

Entre sus publicaciones se destacan los libros *Teoría de la Computación y Verificación de Programas* (Rosenfeld e Irazábal, 2010, EDULP y McGraw-Hill), *Computabilidad, Complejidad Computacional y Verificación de Programas* (Rosenfeld e Irazábal, 2013, EDULP), y *Lógica para Informática* (Pons, Rosenfeld y Smith, 2017, EDULP).

Libros de Cátedra

Rosenfeld, Ricardo
Verificación de programas : programas secuenciales y concurrentes / Ricardo Rosenfeld. - 1a ed. - La Plata : Universidad Nacional de La Plata ; La Plata : EDULP, 2024.
Libro digital, PDF - (Libros de cátedra)

Archivo Digital: descarga
ISBN 978-950-34-2450-6

1. Aplicaciones Informáticas. I. Título.
CDD 005

Diseño de tapa: Dirección de Comunicación Visual de la UNLP

Universidad Nacional de La Plata – Editorial de la Universidad de La Plata
48 N.º 551-599 / La Plata B1900AMX / Buenos Aires, Argentina
+54 221 644 7150
edulp.editorial@gmail.com
www.editorial.unlp.edu.ar

Edulp integra la Red de Editoriales Universitarias Nacionales (REUN)

Primera edición, 2024
ISBN 978-950-34-2450-6

exactas

Edulp
EDITORIAL DE LA UNLP

UNIVERSIDAD NACIONAL DE LA PLATA