Análisis de Algoritmos

Natalia Baeza - Nadina Martínez Carod

6 de septiembre de 2024

Índice general

Índice de cuadros

Índice de figuras

Capítulo 1

Verificacion de Programas

1.1. Introducción

Uno de los enfoques para determinar si un programa es correcto es establecer una actividad de testing. Esta consiste en seleccionar un conjunto de datos de entrada para determinar si los resultados producidos por el programa con esos datos coinciden o no con los valores esperados.

Para asegurar que el programa es correcto se podría hacer lo que se denomina validación estática o dinámica.

> La correctitud de un programa se define como la coincidencia entre el comportamiento deseado del programa y su comportamiento real.

1.1.1. Validación Dinámica

Se analiza el programa con todos los valores posibles de los datos de entrada. Pero esto es imposible cuando este conjunto es infinito. Por eso, el *testing sólo puede mostrar la presencia de errores y no su ausencia.*

En general, para comprobar si un programa ha sido escrito adecuadamente se realizan *ejecuciones de prueba* que permitan conocer su comportamiento. Intentamos saber si calcula los resultados deseados, y es, por tanto, correcto. Así *se puede detectar la presencia de algún error*, y en muchos casos, difícil de reparar, sobre todo en casos de programas extensos y complejos. Pero puede darse el caso en que las pruebas no revelen errores, y éstos aparezcan màs tarde, con un daño mayor.

Un segundo inconveniente que ofrece esta validación dinámica es que, una vez detectada la existencia de errores, la información que obtenemos sobre los puntos del programa en que se encuentran es muy escasa.

1.1.2. Validación Estática

Consiste en obtener información a priori sobre el comportamiento del programa mediante el análisis de su propio texto. Este análisis debe ser capaz de demostrar que los resultados que el programa proporciona son los deseados, o bien de detectar la presencia de errores e identificarlos completamente; este proceso se llama verificación.

La prueba formal de programas, es una técnica que se basa en el cálculo de predicados. Primero se debe describir el comportamiento de cada instrucción del lenguaje formalmente. Es decir, se debe definir la semántica de un lenguaje de programación en términos de fórmulas lógicas. Para probar un programa, se debe expresar su semántica en términos de fórmulas lógicas y luego probar que el programa significa lo mismo que su especificación. Una prueba formal de un programa asegura que el programa es correcto con respecto a una especificación para todas las entradas.

Hay dos problemas importantes con la prueba formal de programas:

1. la manipulación lógica puede ser tediosa y propensa a errores;
2. la prueba solamente muestra que el programa implementa la especificación correctamente. No hay certeza de que la especificación describe lo que el usuario realmente desea.

Resumen

Validación Dinámica:

- Ejecuciones de prueba a partir de conjuntos de datos establecidos
- Errores que aparecen más tarde
- Difíciles de detectar el punto del programa donde se producen

Validación Estática:

- Obtención de información a priori en el texto del programa, proceso denominado **Verificación**.
- Se puede pensar en construir programas ya verificados desde su inicio. Este proceso se denomina **Derivación**

Para decirlo de una manera muy simplificada, el **desarrollo de software** consiste en encontrar un **programa** (o conjunto de programas) que resuelva un **problema** de forma **correcta y eficiente**.

Es sabido que el proceso de desarrollo de software posee una complejidad inherente. [Bus03]

En este capítulo se mostrarán diferentes conceptos y técnicas que facilitan el proceso de desarrollar software, a saber:

1. **Especificación formal:** Nos permitirá definir de forma no ambigua **qué** es lo que se quiere hacer y qué requisitos se tendrán que cumplir para que el programa obtenga los resultados deseados.

2. **Verificación de programas:** Nos ayuda a comprobar que un programa cumple con su especificación de manera que podamos detectar errores.

3. **Documentación de programas:** Nos permite que el código pueda pasar por un proceso de revisión sin perder tiempo en entender exhaustivamente el código.

4. **Derivación formal de programas:** Sirve para escribir de manera razonada y semi-automática un programa a partir de sus especificaciones formales.

1.2. Semántica Axiomática

Es la definición de un algoritmo mediante precondiciones y postcondiciones: también se utiliza una semántica algebraica para la definición de tipos de datos.

Para expresar el comportamiento esperado de un algoritmo (o de un tipo de datos) usaremos *especificaciones.* Estas constarán de una precondición (condiciones que deben cumplir los datos del programa) y una postcondición (relaciones que deben existir entre los datos recibidos y los resultados obtenidos). El significado de una especificación por precondición y postcondición es: si los datos cumplen la precondición, el programa ha de calcular unos resultados que cumplan con la postcondición. Para ello es preciso conocer con exactitud, qué significa cada instrucción, de la siguiente manera: describir simultáneamente con la sintaxis de la instrucción, el significado de ésta, es decir su semántica. Esta descripción puede ser informal, a partir de comentarios que completen la documentación del código, o formal a partir de mecanismos que ofrezca el lenguaje de programación utilizado.

Definición

Si un programa usa n variables $(x_1, x_2, ..., x_n)$ el estado s es una tupla de valores $(X_1, X_2, ..., X_n)$ donde X_1 es el valor de la variable x_i.

Ejemplo: Dado el estado $s = (x, y) = (7, 8)$, el resultado de ejecutar la sentencia de asignación $x := 2 \cdot y + 1$ es el estado $s' = (x, y) = (17, 8)$

Una variable se usa en un programa para describir una posición de memoria que puede contener valores diferentes en diferentes estados. Una manera de describir un conjunto de estados es utilizando fórmulas del cálculo de predicados.

Definición

Sea U el conjunto de todos los posibles estados del programa y sea $S \subseteq U$. Se define P_s, el predicado característico de $S = \{s' \in S | P_s(s')\}$. Es decir, es el predicado que solamente es satisfecho por los estados que pertenecen al conjunto S.

Ejemplos

- En vez de decir que la sentencia $x := 2 \cdot y + 1$ transforma el estado $s = (x, y)$ en el estado $s' = (2 \cdot y + 1, y)$, definiremos predicados $P(x, y)$ y $P'(x', y')$ tal que si $P(x, y)$ es verdadero en el estado s entonces $P'(x', y')$ será verdadero después de ejecutar la sentencia $x := 2 \cdot y + 1$
- Se quiere probar $x \leq 7$ después de ejecutar $x := 2 \cdot y + 1$. Esto es verdadero si $2 \cdot y + 1 \leq 7$ antes de ejecutar la sentencia, es decir $y \leq 3$.

$$\{y \leq 3\}\ x := 2 \cdot y + 1\{x \leq 7\}$$

$\{y \leq 3\}$ y $\{x \leq 7\}$ *se denominan* ***aserciones***.

1.2.1. Aserciones Lógicas

Un *aserto, predicado o aserción* es una expresión lógica que involucra las variables del programa que usamos para expresar el comportamiento de dicho programa en ciertos momentos.

Definición

Una **aserción** es una fórmula del cálculo de predicados ubicada en un programa, que es verdadera cada vez que el programa pasa por ese punto.

El análisis de un programa puede hacerse en términos de un conjunto de estados iniciales admisibles y un conjunto de estados finales apropiados. Para describir conjunto de estados usaremos aserciones.

Dado un conjunto de variables (que pueden o no tener un valor), un estado es una aplicación de este conjunto de variables en el conjunto de sus valores, de manera que se asocie a cada variable un valor coherente con su tipo. Si se observa el valor de las variables de un programa en un cierto momento obtenemos el estado del programa en ese momento.

Las aserciones no son las únicas expresiones lógicas que usamos. Hay aserciones no evaluables (no pueden aparecer en las instrucciones del programa), estas expresiones son las expresiones cuantificadas (expresiones introducidas por un cuantificador) y las operaciones ocultas (expresiones expresadas con un nombre).

1.2.2. Otras definiciones y notación

Estado de computo de un programa: es la descripción completa de los valores asociados a todas las variables del programa en un momento determinado del cómputo. Por ejemplo, si un programa tiene en un momento de su ejecución las variables x e y con los valores 3 y 15 respectivamente, el estado de cómputo en ese momento sería: $x = 3, y = 15$.

Cómputo: es la sucesión de los *estados de cómputo*. Refiere a la ejecución del programa.

Aserciones: son las **afirmaciones** sobre el estado del programa, las fórmulas y expresiones lógicas asociadas a un punto determinado de un programa. Como ya se dijo antes, son útiles para para expresar las variables del programa.

En la siguiente tabla se muestra un ejemplo de cada uno de estos conceptos.

Codigo	aserciones	Estado de cómputo
$x = 5$	$x = 5$	$\{x = 5\}$
$y = x + 2$	$x = 5 \land y = x + 2$	$\{x = 5, y = 7\}$
$z = x + y$	$x = 5 \land y = x + 2 \land z = x + y$	$\{x = 5, y = 7, z = 12\}$

Como se ve en el ejemplo anterior, el estado de cómputo refleja un único estado posible de las variables, una aserción es más genérica y puede corresponder a varios estados de cómputo diferentes. por ejemplo: la aserción $x = y \cdot 2$ puede ser cierta tanto para el estado de cómputo $\{x = 12, y = 6\}$ como para $\{x = 26, y = 13\}$. Particularmente en este caso esta aserción representa un número infinito de estados posibles, hablando de manera abstracta.

Cuando no se defina ninguna *precondición* (se explicará en siguientes secciones) o no se conozca ninguna propiedad de las variables, la aserción será $true$, ya que se cumplirá para cualquier estado de cómputo posible.

Cada aserto puede ser considerado como una descripción de un conjunto de estados: aquellos en los que los valores de las variables hacen que el aserto evalúe a $true$. Diremos que estos estados satisfacen el aserto.

1.2.2.1. Notación: Lógica de Primer Orden (LPO)

A la hora de escribir aserciones, necesitamos un lenguaje (Lógica de Primer Orden) del que valernos para expresar las propiedades que cumplen las variables, con su correspondiente alfabeto, sintaxis y semántica.

Para escribir aserciones utilizamos un lenguaje LPO del que nos valdremos para expresar las propiedades que cumplen las variables con su correspondiente alfabeto, sintaxis y semántica.

a) **Alfabeto:** variables (su valor cambia durante el cómputo), constantes (su valor es fijo durante el cómputo), funciones (operan sobre valores devolviendo otro valor) y predicados (funciones que devuelven un valor booleano).

b) **Sintaxis:** compuesta de **términos** y **funciones**.

Términos: Elementos atómicos que combinados forman una fórmula. Ejemplo: variable x, constante c, función $f(x_1, x_2, \cdots, x_n)$ de n términos.

Fórmulas: Conjunción de términos que se evalúa como $true$ o $false$. Usa **conectores** y **cuantificadores**.

c) **Semántica:** Asocia a cada fórmula un valor booleano dependiendo del estado de cómputo de las variables.

Diremos que una fórmula está definida o tiene valor si todas sus variables libres(variables del programa, no ligadas a un cuantificador) tienen un valor asociado en dicho estado.

1.2.3. Cuantificadores y Variables

Un cuantificador es una abreviatura de una secuencia de operaciones que requiere ir asociado a una variable ligada o ciega, que es simplemente un identificador, y a un dominio, que indica el conjunto de valores que permitimos tomar a la variable ligada y que frecuentemente es un intervalo de los naturales.

La expresión así obtenida denota la repetición de la operación sobre todos los valores del dominio. Usaremos con frecuencia los siguientes cuantificadores:

$$\sum \alpha(dominio \to E(\alpha))$$
$$\prod \alpha(dominio \to E(\alpha))$$

Denotan respectivamente la suma y el producto de todos los valores tomados por la expresión E, cuando recorre todos los valores indicados en el dominio.

$$\forall \alpha(dominio \to E(\alpha))$$
$$\exists \alpha(dominio \to E(\alpha))$$

Denotan la conjunción y la disyunción de todos los valores que toma la expresión E, cuando recorre todos los valores indicados en el dominio.

$$N\ \alpha(dominio \to E(\alpha))$$

Denota el número de valores en el dominio N para los cuales E es cierta.

$$max\ \alpha(dominio \to E(\alpha))$$
$$min\ \alpha(dominio \to E(\alpha))$$

Denotan el mayor y el menor de todos los valores que toma la expresión $E()$, cuando recorre el dominio. La variable ligada en estos casos será la letra griega; puede usarse cualquier otra variable en su lugar sin que cambie el significado de la expresión.

Es concebible aplicar algunos cuantificadores sobre un dominio vacío. El resultado del contador sobre un dominio vacío es cero. La iniciación de un bucle de la iteración y el caso directo de la recursividad se realizan mediante el neutro o el dominio vacío de los cuantificadores.

Por ejemplo: Supongamos que la variable a está declarada como un arreglo de enteros con índices entre 0 y $n-1$; entonces:

1. $\sum \alpha(0 \leq \alpha \leq n-1 \rightarrow a(\alpha))-$ entero resultante de sumar todos los elementos de a.
2. $\prod \alpha(0 \leq \alpha \leq n-1 \rightarrow a(\alpha))-$ entero resultante de multiplicar todos los elementos de a.
3. $\forall\alpha(0 \leq \alpha \leq n-1 \rightarrow a(\alpha) = 0)-$ vale $verdadero$ si a tiene todos los elementos igual a 0, y falso en caso contrario.
4. $\exists\alpha(0 \leq \alpha \leq n-1 \rightarrow a(\alpha) = 0)-$ vale $verdadero$ si algún elemento de a es igual a 0.
5. $N\alpha(0 \leq \alpha \leq n-1 \rightarrow a(\alpha) = 0)-$ contar los elementos de a que son iguales a 0.
6. $max\ \alpha(0 \leq \alpha \leq n-1 \rightarrow a(\alpha))-$ corresponde al mayor valor que aparece en a.
7. $min\ \alpha(0 \leq \alpha \leq n-1 \rightarrow a(\alpha))-$ corresponde al menor valor que aparece en a.

En todos estos ejemplos se dice que α es una variable ligada, es decir asociada al cuantificador, esto es puede usarse cualquier otra variable en su lugar sin que cambie el significado de la expresión.

Para a sin elementos resulta:

- $\sum \alpha(0 \leq \alpha \leq n-1 \rightarrow a(\alpha)) = 0$
- $\prod \alpha(0 \leq \alpha \leq n-1 \rightarrow a(\alpha)) = 1$
- $\forall\alpha(0 \leq \alpha \leq n-1 \rightarrow a(\alpha) = 0) = verdadero$
- $\exists\alpha(0 \leq \alpha \leq n-1 \rightarrow a(\alpha) = 0) = falso$
- $N\alpha(0 \leq \alpha \leq n-1 \rightarrow a(\alpha) = 0) = 0$

Cuando el dominio no es nulo, podemos separar uno de sus elementos y reducir en uno el dominio del cuantificador. El elemento separado se combina con el cuantificador reducido, mediante la correspondiente operación binaria. Por ejemplo, para a con más de un elemento:

$$\sum \alpha(0 \leq \alpha \leq n-1 \rightarrow a(\alpha)) = a(0) + \sum \alpha(1 \leq \alpha \leq n-1 \rightarrow a(\alpha))$$

y

$$\sum \alpha(0 \leq \alpha \leq n-1 \rightarrow a(\alpha)) = \sum \alpha(0 \leq \alpha \leq n-2 \rightarrow a(\alpha)) + a(n-1)$$

De la misma manera, podemos separar un elemento arbitrario del dominio de un cuantificador cualquiera, salvo que sea vacío, combinándolo con el resto mediante la operación binaria asociada al cuantificador.

1.2.4. Variables ligadas y libres

Las variables que aparezcan en los asertos podrán ser de los siguientes tipos:

- *Variable ligada:* está vinculada a un cuantificador, y puede ser sustituida por cualquier otra sin que se modifique el significado de la expresión.
- *Variable libre:* dentro de este grupo se encuentran:
  - *del programa:* que denotan, en cada punto del mismo, el último valor que se les haya asignado.
  - *iniciales:* se usan para denotar un valor que sólo se conoce en un punto diferente de la ejecución del programa.

Conviene evitar el uso del mismo nombre para variables del programa y ligadas.

1.2.5. Formulación de Predicados

Cuando se intenta representar enunciados complejos o extensos utilizando LPO, es deseable dividir el problema dividiéndolo en otros más pequeños.

Siguiendo la analogía de la programación modular y las funciones, podemos definir un predicado $P(t_1, \cdots, t_n)$, donde $P()$ es el predicado y t_i sus térnminos, con $1 \leq i \leq n$.

Ejemplos

- Expresar en LPO que todos los elementos de $A[1..n]$ es primo:

  Definimos $esPrimo(x)$ que nos diga si un número cualesquiera x es primo:

  $$esPrimo(x) = \forall i(1 < i < x \rightarrow x \; mod \; i \neq 0)$$

  Una vez definido el predicado se generaliza para todo el vector:

  $$\forall j(1 \leq j \leq n \rightarrow esPrimo(A[j]))$$

- Expresar en LPO que el vector $A[1..n]$ está ordenado de menor a mayor:

  Si está ordenado de menor a mayor, podemos definir el predicado $esMaximo(x, A, i, j)$ que define x como el máximo de del arreglo A desde la posición i a la posición j. Para generalizar al todo el vector.

  $$esMaximo(x, A, i, j) = \forall k(i \leq k \leq j \rightarrow A[k] \leq x)$$

  Generalizando a todo el vector:

  $$\forall i(1 < i \leq n \rightarrow esMaximo(A[i], A, 1, i - 1))$$

1.2.6. Fortaleza de las aserciones

Una aserción (o proposición) es considerada más fuerte que otra si implica a la otra, pero no es implicada por ella.

En otras palabras, una aserción A_1 es más fuerte que una aserción A_2 si la verdad de A_1 garantiza la verdad de A_2 , pero no viceversa.

- Implicación Lógica:
  - Si $A_1 \Rightarrow A_2$ (es decir, "si A_1 es verdadera, entonces A_2 es verdadera") pero no necesariamente $A_2 \Rightarrow A_1$, entonces decimos que A_1 es más fuerte que A_2
  - Ejemplo:
    - Aserción A_1: "Todos los perros son animales"
    - Aserción A_2: .ªlgunos perros son animales."

    Aquí, A_1 es más fuerte que A_2 , porque si "todos los perros son animales.es cierto, entonces .ªlgunos perros son animales"también es cierto. Sin embargo, la verdad de A_2 no garantiza la verdad de A_1.

- Cantidad de Información:
  - Una aserción es más fuerte que otra si proporciona más información o establece más condiciones. Esto implica que la aserción más fuerte tiene un alcance más **restrictivo** o **específico**.
  - Ejemplo:
    - Aserción A_1: "Todas las manzanas son rojas."
    - Aserción A_2: .ªlgunas manzanas son rojas."

    Aquí, A_1 es más fuerte que A_2 porque A_1 establece una condición más específica (todas versus algunas).

- Restricciones y Especificidad:
  - Una aserción es más fuerte si introduce restricciones adicionales o es más específica sobre una propiedad o relación.
  - Ejemplo:
    - Aserción A_1: .ªna es una mujer alta."
    - Aserción A_2: .ªna es una mujer."

    La aserción A_1 es más fuerte que A_2 porque implica más detalles sobre Ana (no solo que es una mujer, sino que también es alta).

- Comparación con Cuantificadores:

  - En lógica de primer orden, las aserciones con cuantificadores universales ($\forall$) son generalmente más fuertes que las aserciones con cuantificadores existenciales ($\exists$), cuando ambos se refieren a la misma relación o propiedad.
  - Ejemplo:
    - Aserción $A_1 : \forall x(P(x))$ es verdadero.
    - Aserción $A_2 : \exists x(P(x))$ (existe al menos un x para el cual $P(x)$ es verdadero).

    A_1 es más fuerte que A_2 porque afirmar que $P(x)$ es verdadero para todos los X implica que es verdadero para al menos un x , pero no al revés.

- Contraposición o Negación:

  - Una aserción puede ser más fuerte si niega algo que es más amplio o general.
  - Ejemplo
    - Aserción A_1: "No todos los estudiantes aprobaron el examen."
    - Aserción A_2: .Algunos estudiantes no aprobaron el examen."
  - Ambas aserciones son lógicamente equivalentes, pero podemos considerar que "No todos.es más fuerte en términos de estilo o impacto que .Algunos no".

- Resumiendo:

  - A_1 es más fuerte que A_2 si la verdad de A_1 garantiza la verdad de A_2 , pero no al revés.
  - Aserciones más fuertes suelen ser más restrictivas, específicas o informativas.
  - Utilizan cuantificadores universales o establecen condiciones adicionales que no están presentes en las aserciones más débiles.

Definición

Dados dos asertos A_1 y A_2, diremos que: A_1 *es más fuerte que* A_2

- si todo estado que cumpla A_1 ha de cumplir también A_2;
- dicho de otro modo $A_1 \Rightarrow A_2$;
- dicho de otra manera, *si el conjunto de los estados que satisfacen* A_1 *es un subconjunto de los estados que satisfacen* A_2

Siguiendo la definición anterior sobre los asertos A_1 y A_2, en consecuencia emplearemos el término *más débil que* para decir que A_2 *es más débil que* A_1.

1.2.7. Operación de sustitución

Sea una fórmula A, la variable x y el término t, definirmenos A_x^t como la fórmula resultante de sustituir todas las apariciones libres de x por t en A. Como aparición libre entendemos todas aquellas que no sean variables ligadas del mismo nombre.

Si el término t tuviera alguna variable ligada en A, deberá renombrarse antes de realizar sustitución.

Ejemplos

- $(x^2 + 2 \cdot x + 17)_x^y \rightarrow y^2 + 2 \cdot y + 17$
- $(x < 5 \land 1 < y < 5)_x^{x-5} \rightarrow x - 5 < 5 \land 1 < y < 5 \rightarrow x < 10 \land 1 < y < 5$
- $(\sum_{k=i}^{j} A[k])_i^{i+1} \rightarrow \sum_{k=i+1}^{j} A[k]$

1.3. Especificación pre/post condición

Al expresar, mediante un aserto, las condiciones que se imponen a los datos, se está restringiendo el conjunto de estados en que se puede poner en marcha el programa con garantías de funcionamiento correcto. Este aserto que impone condiciones a los datos se denomina precondición. Cuanto más débil sea esta precondición, más útil será el programa, ya que podrá emplearse en más casos. De manera dual, el aserto que expresa las propiedades de los resultados del algoritmo se denomina postcondición.

Una especificación pre/post para un programa C se escribe $\{P\}C\{Q\}$

y denota que se requiere del programa C que, si comienza a funcionar en un estado que satisfaga P, termine en un tiempo finito y en un estado que satisfaga Q. La especificación describe el comportamiento esperado del programa C. Verificar el programa consiste entonces en *demostrar* que cumple su especificación.

Definición

$\{P\}\ C\ \{Q\}$, donde P y Q son aserciones llamadas pre-condición y post-condición respectivamente y C es un fragmento de programa, se interpreta como sigue: si la ejecución de C empieza en un estado caracterizado por P y C termina, entonces terminará en un estado caracterizado por Q

$\{P\}\ C\ \{Q\}$ se denomina **especificación formal** de C

Supongamos que se ha demostrado $\{P\}\ C\ \{Q\}$. Entonces:
- Si $P \Rightarrow P'$ entonces $\{P'\}\ C\ \{Q\}$ es verdadero y además se muestra que el aserto ***P es más***

fuerte
*- Si $Q \Rightarrow Q'$ entonces $\{P\}\ C\ \{Q'\}$ es verdadero y además se **debilita** el aserto Q.*

Para especificar un módulo necesitamos saber qué variables representan datos y cuáles resultados. La especificación consta de tres partes: cabecera, pre y post condición.

La cabecera indica el nombre del módulo y cómo se llaman y de qué tipo son los parámetros y las variables locales en las que se calcularán los resultados. La precondición involucrará los parámetros y la postcondición los resultados.

La sintaxis para presentar la cabecera del módulo es

```
MODULO nom (parámetros) RETORNA resultados
    {Pre}
    variables locales
    {Post}
    RETORNA resultados
FIN MODULO
```

Una vez calculados los resultados todos ellos aparecerán al final del cuerpo del módulo en una instrucción **RETORNA** que representa la devolución de resultados al punto en que se produjo la llamada. Sólo puede haber una instrucción **RETORNA** en cada módulo, y ha de estar situada como última instrucción a realizar.

Ejemplo

Especificar un módulo que calcule la división de dos números enteros positivos y retorne el resultado y el resto.
Primero analizamos el problema estableciendo los parámetros de entrada y los valores que pueden tomar -dominio- y luego establecemos el contenido de los resultados. Definimos que contamos con una asignación múltiple para la instrucción **RETORNA**, independiente del lenguaje de programación y solamente nos enfocamos en la especificación del algoritmo que dará solución al problema.
Entonces, como es una división, el divisor debe ser mayor que cero, y a partir de los parámetros de entrada, podemos armar la expresión que calcula los resultados requeridos.

```
MODULO dividir(a, b: entero) RETORNA q, r: entero
    {Pre: a ≥ 0, b > 0}
    {Post: a ← b * q + r ∧ r < b }
    RETORNA q, r
FIN MODULO
```

Luego de especificar pre y postcondición, se completa el algoritmo:

```
MODULO dividir(a, b: entero) RETORNA q, r: entero
    {Pre: a ≥ 0, b > 0}
```

```
3       var ... (declaración de variables locales)
4       (instrucciones)
5       {Post: a ← b * q + r ∧ r < b }
6       RETORNA q, r
7   FIN MODULO
```

De esta manera el algoritmo resultante ha sido verificado antes de construirlo.

Los siguientes son ejemplos de programas especificados en términos de *pre* y *post* condiciones.

1. Dados dos números naturales a y b, $b \neq 0$, encontrar el cociente entre a y b
   $P : \{a, b \in \mathbb{N} \land b > 0\}$
   $C : q = a\ DIV\ b; r = a\ MOD\ b;$
   $Q : \{a = b \cdot q + r \land 0 \leq r < b \land q \geq 0\}$

2. Dado un arreglo de n números enteros, ordenarlo en forma ascendente.

   $$P : \{\forall i(1 \leq i \leq n \rightarrow integer(a[i]) \land a[i] = A[i])\}$$

   A referencia el valor de los elementos del arreglo antes de iniciar la ejecución.

   $$Q : \{\forall i(1 \leq i \leq n \rightarrow a[i] \leq a[i+1] \land permutacion(a, A))\}$$

   El predicado $permutacion(a, A)$ se define como:
   $permutacion(a, A) =$
   $= \{\forall i(1 \leq i \leq n \rightarrow (N_j : 1 \leq j \leq n : a[i] = a[j]) = (N_k : 1 \leq k \leq n : a[i] = A[k]))\}$

3. Encontrar el valor máximo de un arreglo de n números enteros.

   $$P : \{\forall i(1 \leq i \leq n \rightarrow integer(a[i] \land a[i] = A[i])\}$$

   A referencia el valor de los elementos del arreglo antes de iniciar la ejecución.

   $$Q : \{\exists i(1 \leq i \leq n \rightarrow (max = a[i] \land \forall j(1 \leq j \leq n \rightarrow a[j] \leq max \land a[j] = A[j]))\}$$

Volviendo al ejemplo $\{y \leq 3\}x := 2 \cdot y + 1\{x \leq 7\}, \{y \leq 3\}$ no es la única precondición que hace verdadera la postcondición después de la ejecución de la sentencia de asignación.

Otra precondición podría ser $\{y = 1 \lor y = 3\}$. Pero esta última es menos interesante porque no caracteriza todos los estados desde los cuales se puede alcanzar un estado caracterizado por

la postcondición. Se desea determinar como precondiciones aquellas fórmulas que describan tantos estados como sea posible. Esto se hace eligiendo el predicado menos restrictivo posible.

Podemos deducir instrucciones algorítmicas a partir de su especificación, proceso que se denomina **derivación**.

La *derivación* es la deducción de las instrucciones algorítmicas a partir de su especificación. La construcción del algoritmo no es completamente mecánica pero el proceso proporciona abundantes sugerencias y datos sobre la corrección de las decisiones tomadas.

Este proceso es especialmente útil para concretar los detalles que pueden dificultar, por el grado de precisión que requieren, la tarea de programar; inicializaciones, conjunciones y disyunciones en las condiciones de los bucles, decisión entre comparaciones *menor que* o *menor o igual* y similares.

En un proceso de derivación, la postcondición resulta más relevante y proporciona mucha más información que la precondición, sin que ésta sea irrelevante.

El análisis de la postcondición es por tanto un buen método para desarrollar algoritmos.

Mencionemos algunas consideraciones:

- Si en la postcondición aparecen igualdades entre variables del programa y expresiones, puede satisfacerse mediante asignaciones.
- Si aparecen disyunciones se puede intentar diseñar una alternativa.
- Si aparecen conjunciones se puede intentar diseñar una alternativa o considerar cada una de ellas aisladamente, e intentar satisfacerlas por separado mediante asignaciones.
- Si aparecen cuantificadores se puede intentar el diseño de iteraciones (o recursiones)

1.4. Concepto de corrección

Un problema algorítmico puede ser dividido en dos partes:

1. una especificación para un conjunto de valores legales,y
2. la relación entre las entradas y las salidas deseadas.

Por ejemplo, supongamos que cada entrada legal de un algoritmo A consiste de una lista L de palabras en Español. La relación entre las entradas y las salidas deseadas podría especificarse como que la salida es una lista con las palabras contenidas en L ordenadas lexicográficamente de forma ascendente.

Para facilitar el tratamiento preciso del problema de la correctitud para los algoritmos, se distinguen dos tipos de correctitud, dependiendo de si se incluye o no la terminación del programa. En un caso, la terminación del programa se asume *a priori* y en el otro no. Así, más precisamente, se dice que un algoritmo *A* es **parcialmente correcto** (con respecto a su definición de entradas legales e interrelaciones con las salidas) si por cada entrada legal X, si A termina cuando se ejecuta con X entonces la relación especificada respecto de la salida esperada, se cumple. Así, un algoritmo de ordenamiento correcto podría no terminar en todas las listas legales, per cuando lo hace una lista correctamente ordenada es el resultado. Decimos que A **termina** si se detiene cuando se ejecuta cualquier entrada legal, Ambas nociones se cumplen juntas - correctitud parcial y terminación - logrando un algoritmo **totalmente correcto**, el cual resuelve correctamente el problema algorítmico para cada entrada legal: el proceso de ejecución de A en cualquier entrada X siempre termina y produce una salida que satisface la relación deseada.

Expresado entonces en términos de P, C y Q, si $\{P\}C\{Q\}$ es un código con la precondición $\{P\}$ y la poscondición $\{Q\}$, entonces $\{P\}C\{Q\}$ es correcto si cada estado inicial posible que satisfaga $\{P\}$ da como resultado un estado final que satisface $\{Q\}$.

- *Corrección parcial*: se dice que $\{P\}C\{Q\}$ es parcialmente correcto si el estado final de C, cuando termina el programa (aunque no se le exige esta premisa), satisface $\{Q\}$ siempre que el estado inicial satisface $\{P\}$.
- *Corrección total*: se da cuando un código además de ser correcto parcialmente termina. Los códigos sin bucles siempre terminan por lo que la corrección parcial implica la corrección total. Esta distinción es esencial sólo en el caso de códigos que incluyan bucles o recursiones.

1.5. Verificación formal de código

Para verificar la corrección total o parcial de un programa utilizaremos el cálculo de Hoare, que utilizará axiomas (por ejemplo, el Axioma de Asignación) y reglas de inferencia.

A la hora de verificar la corrección de un programa, el proceso a seguir es, a partir dela precondición, ir deduciendo, mediante axiomas y reglas de inferencia, las propiedades entrevariables que se mantendrán en los diferentes puntos del programa. El objetivo final siempreserá demostrar que al final del programa se cumplirá la postcondición.

Los **axiomas** serán propiedades básicas indemostrables de la forma $\{P\}C\{Q\}$ donde C podría ser, por ejemplo, una asignación. Si en algún paso anterior se hubiese demostrado que antes de C se cumplía P, quedará demostrado que después de la asignación se cumplirá Q.

La verificación la realizaremos con estas consideraciones:

- No se utilizarán declaraciones de tipo
- Ni sentencias de entrada/salida.

- La concatenación de dos códigos C_1 y C_2 supone la ejecución secuencial de dichos códigos y vendrá representada de la siguiente forma: $C_1; C_2$.
- Las aserciones o asertos son sentencias lógicas que hacen referencia a un estado del sistema. Para indicar que una sentencia es un aserto se encierra dicha sentencia entre llaves $\{A\}$
- Un estado vendrá dado por el conjunto de valores que toman en ese instante el conjunto de variables que conforman la estructura de datos del problema.
  - Las precondiciones indican las condiciones que deben satisfacer los datos de entrada para que el programa pueda cumplir su tarea.
  - Las poscondiciones indican las condiciones de salida que son aceptables como soluciones correctas del problema en cuestión.
  - Ejemplo: Programa que calcula la raíz cuadrada
    Precondición: Que el argumento de la función no sea negativo
    Poscondición: La raíz cuadrada de ese argumento.
- En la mayoría de los códigos el estado final depende del estado inicial. Por tanto Precondición y Poscondición no son independientes entre sí.
- Hay dos posibles nomenclaturas que permitan reflejar dicha dependencia:
  - Representación del subíndice: mediante subíndices se indica si las variables representan valores iniciales o finales. $\{a_\omega = b_\alpha\} \wedge \{b_\omega = a_\alpha\}$ donde ω representa el estado final y α el estado inicial.
  - Representación de las variables ocultas: Las variables ocultas aparecen o no en el código y se introducen para almacenar los valores iniciales de ciertas posiciones de memoria. Su definición debe aparecer en la precondición.

$$\{a = A, b = B\}h := a; a := b; b := h\{a = B, b = A\}$$

1.5.1. Fortalecimiento de las precondiciones

Si una parte de un código es correcta bajo la precondición $\{P\}$, entonces permanecerá correcto si se refuerza $\{P\}$.

Si $\{P\}C\{Q\}$ es correcto y además $P_1 \Rightarrow P$ entonces se puede afirmar que $\{P_1\}C\{Q\}$ es correcto. Esto conduce a la siguiente regla de inferencia:

Ejemplo

Supongamos que la siguiente terna de Hoare es correcta: $\{y \neq 0\}x := 1/y\{x = 1/y\}$, demostrar que también lo es $\{y = 4\}x := 1/y\{x = 1/y\}$.

$$\frac{\begin{array}{l} P_1 \Rightarrow P \\ \{P\}C\{Q\} \end{array}}{\{P_1\}C\{Q\}}$$

$$\frac{\begin{array}{l} y = 4 \Rightarrow y \neq 0 \\ \{y \neq 0\}x := 1/y\{x = 1/y\} \end{array}}{\{y = 4\}x := 1/y\{x = 1/y\}}$$

La aserción vacía puede ser reforzada para producir cualquier precondición $\{P\}$.

Ejemplo

$\{\ \}a := b\{a = b\}$ se puede utilizar para justificar $\{P\}a := b\{a = b\}$ donde $\{P\}$ puede ser cualquier condición.

Como regla general:

- Siempre será ventajoso intentar formular la precondición más débil posible que asegure que se cumple una cierta poscondición dada. De esta forma cualquier precondición que implique la anterior será satisfecha automáticamente.
- Además los programas deben ser escritos de modo que resulten lo más versátiles posibles (Generalidad).

1.5.2. Debilitamiento de las poscondiciones

El principio de debilitamiento de postcondiciones permite concluir que $\{P\}C\{Q_1\}$ una vez que $\{P\}C\{Q\}$ y $Q \Rightarrow Q_1$ hayan sido establecidos formalmente. Esto conduce a la siguiente regla de inferencia:

$$\frac{\begin{array}{l} \{P\}C\{Q\} \\ Q \Rightarrow Q_1 \end{array}}{\{P\}C\{Q_1\}}$$

Ejemplo

Si $\{\ \}max := b\{max = b\}$ es correcto, demostrar que se cumple $\{\ \}max := b\{max \geq b\}$

$$\frac{\{\ \}max = b\{max = b\} \quad max = b \Rightarrow max \geq b}{\{\ \}max := b\{max \geq b\}}$$

1.5.3. Regla de la conjunción

La siguiente regla permite reforzar la precondición y postcondición de forma simultánea.

Definición

Si C es una parte de un código y se ha establecido que $\{P_1\}C\{Q_1\}$ y $\{P_2\}C\{Q_2\}$ se puede afirmar que $\{P_1 \wedge P_2\}C\{Q_1 \wedge Q_2\}$. Formalmente esto se expresa como:

$$\frac{\{P_1\}C\{Q_1\} \quad \{P_2\}C\{Q_2\}}{\{P_1 \wedge P_2\}C\{Q_1 \wedge Q_2\}}$$

Como caso particular:

$$\frac{\{\ \}C\{Q_1\} \quad \{P_2\}C\{Q_2\}}{\{P_2\}C\{Q_1 \wedge Q_2\}}$$

Ejemplo

Aplicar las ternas $\{\}i := i+1\{i_\omega = i_\alpha+1\}$, $\{i_\alpha > 0\}i := i+1\{i_\alpha > 0\}$, para demostrar que $\{i > 0\}i := i+1\{i > 1\}$, siendo i_α el valor inicial de la variable i, y i_ω el valor que obtiene luego ser modificado.

$$\frac{\{\ \}i := i+1\{i_\omega = i_\alpha + 1\} \quad \{i_\alpha > 0\}i := i+1\{i_\alpha > 0\}}{\{i_\alpha > 0\}i := i+1\{(i_\alpha > 0) \wedge (i_\omega = i_\alpha + 1)\}}$$

Sabiendo que $i_\alpha := i_\omega - 1$ y puesto que $i_\alpha > 0$ entonces $i_\omega > 1$ por tanto

$$\{i > 0\}\ i := i + 1\ \{i > 1\}$$

1.5.4. Regla de la disyunción

La siguiente regla permite debilitar la precondición y postcondición de forma simultánea.

Definición

Si C es una parte de un código y se ha establecido que $\{P_1\}C\{Q_1\}$ y $\{P_2\}C\{Q_2\}$ se puede afirmar que $\{P_1 \vee P_2\}C\{Q_1 \vee Q_2\}$. Formalmente esto se expresa como:

$$\frac{\{P_1\}C\{Q_1\} \quad \{P_2\}C\{Q_2\}}{\{P_1 \vee P_2\}C\{Q_1 \vee Q_2\}}$$

Como caso particular:

$$\frac{\{\,\}C\{Q_1\} \quad \{P_2\}C\{Q_2\}}{\{P_2\}C\{Q_1 \vee Q_2\}}$$

1.5.5. Verificación de códigos sin bucles

Para demostrar la corrección de las partes de un programa se parte de la poscondición final del código, es decir de las condiciones que deben satisfacer los resultados. A partir de ahí se deduce la precondición. A partir de la poscondición final se deben aplicar reglas propias de la lógica de predicados hasta llegara la precondición inicial. El código se verifica en sentido contrario a como se ejecuta (Figura 1.1).

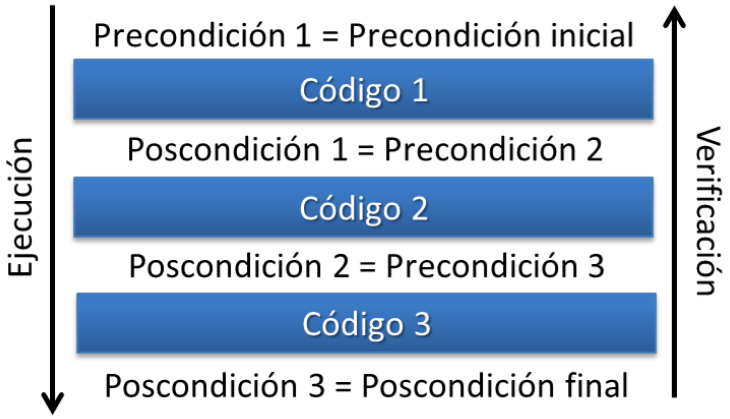

Figura 1.1: Verificación vs. Ejecución

1.5.5.1. Regla de la asignación

La regla de asignación requiere que las variables implicadas no compartan el mismo espacio de memoria. Esto excluye el uso de punteros y de variables con índices (arreglos).

Las sentencias de asignación son sentencias de la forma $V := E$, en donde V es una variable y E es una expresión.

$$\{\ \}V := E\{V = E_\alpha\}$$

Sin embargo esta expresión no siempre es correcta, hay casos en los que es necesario definir o encontrar una precondición que delimite el dominio, por ejemplo en $x := 1/y$, la precondición debe ser que $y \neq 0$. En dicho caso debemos generalizar:

Regla de la asignación

Si C es una sentencia de la forma $V := E$ con la postcondición $\{Q\}$, entonces la precondición de C puede hallarse sustituyendo todos los casos de V en Q por E. Si Q_E^V es la expresión que se obtiene mediante esto, entonces:

$$\{P\}V := E\{Q\} \tag{1.1}$$

$$\{P\} = \{Q_E^V\} \Rightarrow \{V = E_\alpha, Q\} \tag{1.2}$$

$$\{Q_E^V\}V := E\{Q\} \tag{1.3}$$

En otras palabras, en **(1.3)** se debe reemplazar en Q todos los casos de E por V para encontrar la precondición. En **(1.2)**, el aserto Q_E^V es, en definitiva, una conjunción de la asignación con la poscondición. Finalmente, en **(1.1)**, se define la terna de Hoare.

Ejemplos

1. Determinar la precondición para que la terna siguiente sea correcta:
   $\{P\}i := 2 \cdot i\{i < 6\}$

   $$\{Q_E^V\} = \{i_\omega = 2 \cdot i_\alpha, i_\omega < 6\} \Rightarrow \{2 \cdot i_\alpha < 6\} \Rightarrow \{i_\alpha < 3\}$$

2. Determinar la precondición para que la terna siguiente sea correcta:
   $\{P\}j := i + 1\{j > 0\}$

   $$\{Q_E^V\} = \{j_\omega = i_\alpha + 1, j_\omega > 0\} \Rightarrow \{i_\alpha + 1 > 0\}$$

3. Determinar la precondición para que la terna siguiente sea correcta:
   $\{P\}y := x^2\{y > 1\}$

   $$\{Q_E^V\} = \{x_\omega = x_\alpha \cdot x_\alpha, x_\omega > 1\} \Rightarrow \{x_\alpha^2 > 1\}$$

4. Determinar la poscondición para que la terna siguiente sea correcta:
   $\{x > 2\}x := x^2\{Q\}$

$$\{Q\} \Rightarrow \{x_\alpha > 2, x_\omega = x_\alpha^2\} \Rightarrow \{x_\omega > 4\}$$

5. Determinar la precondición para que la terna siguiente sea correcta:
   $\{P\}x := 1/x\{x \geq 0\}$

   $$\{Q_E^V\} \Rightarrow \{x_\omega = 1/x_\alpha, x_\omega \geq 0\} \Rightarrow \{x_\alpha > 0\}$$

   Se puede observar claramente en los 5 ejemplos anteriores que aquí la deducción se realiza desde la postcondición a la precondición.

Consideraciones:
Antes de realizar una asignación conoceremos una serie de propiedades sobre las variables que forman parte de la expresión E. Todas aquellas propiedades de E que se conozcan antes de la asignación, las cumplirá la variable V después de la asignación

$$\{z \; mod \; 2 = 0\}x := z; \{x \; mod \; 2 = 0 \wedge z \; mod \; 2 = 0\}$$

Seguiremos sabiendo lo mismo de z luego de la asignación porque ni z ni ninguna variable libre de la precondición ha sido modificada en la asignación.

En este otro ejemplo:

$$\{x - 2 > 0\}x := x - 2; \{x > 0 \wedge \cancel{x - 2 > 0}\}$$

Después de la asignación no es posible afirmar que $x - 2 > 0$ se siga cumpliendo, ya que la *variable libre x* que aparece en E ha cambiado por ser además la variable donde se guardará el resultado.

Otros ejemplos:

$$\{y \neq 0 \wedge z/y = 2 \wedge v = 12\}z := z/y; \{y \neq 0 \wedge z = 2 \wedge v = 12\}$$
$$\{y \neq 0 \wedge y/2 = 16\}z := y/2; \{y \neq 0 \wedge z = 16 \wedge y/2 = 16\}$$

Como podemos ver en estos últimos 4 ejemplos la manera de resolver es desde la precondición a la postcondición.

1.5.5.2. Concatenación de código

La concatenación significa que las partes del programa se ejecutan secuencialmente de tal forma que el estado final de la primera parte de un código se convierte en el estado inicial de la segunda parte del programa.

Regla de la concatenación

Sean C_1 y C_2 dos partes de un código y sea $C_1; C_2$ su concatenación. Si $\{P\}C_1\{R\}$ y $\{R\}C_2\{Q\}$ son ternas de Hoare correctas entonces se puede afirmar que:

$$\frac{\{P\}C_1\{R\} \quad \{R\}C_2\{Q\}}{\{P\}C_1; C_2\{Q\}}$$

Ejemplos

1. Demostrar que el siguiente código es correcto:

$$\{\ \}\ c := a + b; c := c/2\ \{c = (a + b)/2\}$$

$$\{P\}\ c := c/2\ \{c = (a + b)/2\}$$
$$P = \{c_\omega = c/2, c_\omega = (a + b)/2\} \Rightarrow \{c/2 = (a + b)/2\}$$
$$\{c/2 = (a + b)/2\}\ c := c/2\ \{c = (a + b)/2\}$$

$$\{P\}\ c := a + b\ \{c/2 = (a + b)/2\}$$
$$P = \{c_\omega = a + b, c_\omega/2 = (a + b)/2\} \Rightarrow \{(a + b)/2 = (a + b)/2\} \Rightarrow \{\}$$
$$\{\ \}\ c := a + b\ \{c/2 = (a + b)/2\}$$

   Con lo que queda demostrado.

2. Demostrar que el siguiente código es correcto:

$$\{\ \}\ s := 1; s := s + r; s := s + r \cdot r\ \{s = 1 + r + r^2\}$$

$$\{P\}\ s := s + r \cdot r\ \{s = 1 + r + r2\}$$
$$P = \{s_\omega = s + r2, s_\omega = 1 + r + r2\} \Rightarrow \{s + r^2 = 1 + r + r^2\}$$
$$\{s + r^2 = 1 + r + r^2\}\ s := s + r \cdot r\ \{s = 1 + r + r2\}$$

$$\{P\}\ s := s + r\ \{s = 1 + r\}$$
$$P = \{s_\omega = s + r, s_\omega = 1 + r\} \Rightarrow \{s + r = 1 + r\}$$
$$\{s + r = 1 + r\}\ s := s + r\ \{s = 1 + r\}$$

$$\{P\}\ s := 1\ \{s = 1\}$$
$$P = \{s_\omega = 1, s_\omega = 1\} \Rightarrow \{1 = 1\} \Rightarrow \{\ \}$$
$$\{\ \}\ s := 1\ \{s = 1\}$$

   Con lo que queda demostrado.

3. Demostrar que el siguiente código es correcto:

$$\{a = A, b = B\}\ h := a; a := b; b := h\ \{a = B, b = A\}$$

$$\{P\}\ b := h\ \{a = B, b = A\}$$
$$P = \{b = h, a = B, b = A\} \Rightarrow \{h = A, a = B\}$$
$$\{h = A, a = B\}\ b := h\ \{a = B, b = A\}$$

$$\{P\}\ a := b\ \{h = A, a = B\}$$
$$P = \{a = b, h = A, a = B\} \Rightarrow \{h = A, b = B\}$$
$$\{h = A, b = B\}\ a := b\ \{h = A, a = B\}$$

$$\{P\}\ h := a\ \{h = A, b = B\}$$
$$P = \{h = a, h = A, b = B\} \Rightarrow \{a = A, b = B\}$$
$$\{a = A, b = B\}\ h := a\ \{h = A, b = B\}$$

Con lo que queda demostrado.

1.5.5.3. Invariante de un código

Cualquier aserción que es a la vez precondición y poscondición de una parte del código se denomina *invariante*.

Ejemplo

Demostrar que $r = 2^i$ es un invariante del siguiente código: $i = i + 1; r = r \cdot 2$.

$$\{P\}\ r := r \cdot 2\ \{r = 2^i\}$$
$$P = \{r = r \cdot 2, r = 2^i\} \Rightarrow \{r \cdot 2 = 2^i\} \Rightarrow \{r = 2^{i-1}\}$$
$$\{r = 2^{i-1}\}\ r := r \cdot 2\ \{r = 2^i\}$$

$$\{P\}\ i := i + 1\ \{r = 2^{i-1}\}$$
$$P = \{i = i + 1, r = 2^{i-1}\} \Rightarrow \{r = 2^{i+1-1}\} \Rightarrow \{r = 2^i\}$$
$$\{r = 2^i\}\ i := i + 1\ \{r = 2^{i-1}\}$$

Con lo que queda demostrado.

1.5.5.4. Regla para alternativas

Con cláusula *ELSE*

Si ahora se desea demostrar la corrección de una sentencia if con una precondición P y una postcondición Q tendremos dos posibilidades:

1. Si el estado inicial satisface B además de P entonces se ejecutará C_1 y por tanto la verificación equivaldrá a demostrar que $\{P \wedge B\}\ C_1\ \{Q\}$ es correcto.

2. Si el estado inicial no satisface B entonces esto equivaldría a demostrar que $\{P \wedge \neg B\} \Rightarrow \{Q\}$.

$$\frac{\{P \wedge B\}\ C_1\ \{Q\} \quad \{P \wedge \neg B\}\ C_2\ \{Q\}}{\{P\}\ if\ B\ then\ C_1\ else\ C_2\ \{Q\}}$$

Ejemplo

Demostrar que:

$$\{\ \}\ if\ a > b\ then\ m := a\ else\ m := b\ \{(m \geq a) \wedge (m \geq b)\}$$

Aplicando la regla de las alternativas, trabajamos con las dos posibilidades que provee el fragmento de programa, siendo $B = a > b$

- $\{\mathbf{a > b}\}\ \mathbf{m := a}\ \{(\mathbf{m \geq a}) \wedge (\mathbf{m \geq b})\}$

$$P = \{m = a, (m \geq a) \wedge (m \geq b)\} \Rightarrow \{a \geq a, a \geq b\} \xRightarrow[es\ true]{a \geq a} \{a \geq b\}$$
$$\{a > b\} \Rightarrow \{a \geq b\}\ \text{Fortalecimiento de la precondición}$$
$$\{a > b\}\ m := a\ \{(m \geq a) \wedge (m \geq b)\}$$

- $\{\neg(\mathbf{a > b})\}\ \mathbf{m := b}\ \{(\mathbf{m \geq a}) \wedge (\mathbf{m \geq b})\}$

$$P = \{m = b, (m \geq a) \wedge (m \geq b)\} \Rightarrow \{b \geq a, b \geq b\} \xRightarrow[es\ true]{b \geq b} \{b \geq a\}$$
$$\{\neg(a > b)\} \Rightarrow \{b \geq a\}$$
$$\{\neg(a > b)\}\ m := b\ \{(m \geq a) \wedge (m \geq b)\}$$

Con lo que queda demostrado

Sin cláusula *ELSE*

Si C_1 es una parte de un programa y B es una condición, entonces la sentencia $if\ B\ then\ C_1$ se interpreta de la siguiente forma: si B es verdadero se ejecuta C_1. Si ahora se desea demostrar la corrección de una sentencia if con una precondición P y una postcondición Q tendremos dos posibilidades:

1. Si el estado inicial satisface B además de P entonces se ejecutará C_1 y por tanto la verificación equivaldrá a demostrar que $\{P \wedge B\}\ C_1\ \{Q\}$ es correcto.

2. Si el estado inicial no satisface B entonces esto equivaldrá a demostrar que $\{P \wedge \neg B\} \Rightarrow \{Q\}$.

$$\frac{\{P \wedge B\}\ C_1\ \{Q\} \quad \{P \wedge \neg B\} \Rightarrow \{Q\}}{\{P\}\ if\ B\ then\ C_1\ \{Q\}}$$

Ejemplo

Demostrar que: $\{\ \}if\ max < a\ then\ max := a\ \{max \geq a\}$

- $\{\neg(\mathbf{max} < \mathbf{a})\} \Rightarrow \{\mathbf{max} \geq \mathbf{a}\}$

  Lo que es obvio

- $\{\mathbf{max} < \mathbf{a}\}\ \mathbf{max} := \mathbf{a}\ \{\mathbf{max} \geq \mathbf{a}\}$

  $$P = \{max = a, max \geq a\} \Rightarrow \{a \geq a\}$$
  $\{a \geq a\} \Rightarrow \{\ \}$ Y puesto que $\{max < a\} \Rightarrow \{\ \}$

Con lo que queda demostrado.

1.5.6. Precondición a partir de la poscondición en estructuras *IF*

Cuando se trabaja con la estructura if es bastante fácil obtener la poscondición a partir de la precondición. Sin embargo normalmente lo que se necesita es lo contrario. Para facilitar la labor anterior es mejor utilizar la siguiente regla de inferencia:

$$\frac{\{B\}\ C_1\ \{B\} \quad \{\neg B\}\ C_2\ \{\neg B\} \quad \{P\}\ C_1\ \{Q_1\} \quad \{P\}\ C_2\ \{Q_2\}}{\{P\}\ if\ B\ then\ C_1\ else\ C_2\{(B \wedge Q_1) \vee (\neg B \wedge Q_2)\}}$$

1.5.7. Bucles

Mientras que todos los programas sin bucles terminan, no se puede decir lo mismo de aquellos que si lo poseen o son recursivos. Por ahora sólo nos preocupará la corrección parcial. El *invariante del bucle* es una aserción que captura las características que no varían al ejecutarse un bucle. El *variante de un bucle* es una expresión que mide el progreso realizado por el bucle hasta satisfacer la condición de salida. Las estructuras iterativas elegidas para establecer su corrección son:

- *while B do C*, donde B es la condición de entrada y C es el código contenido dentro del bucle.
- *repeat C until B*, donde B es la condición de salida y C es el código contenido dentro del bucle.

Como podremos comprobar en cada iteración el bucle debe progresar hacia la postcondición final, que debe derivarse del invariante cuando la condición de entrada no se cumple (condición de salida).

- El invariante del bucle es el precursor de la poscondición, lo que indica que este debe ser parecido a la poscondición.
- En cada iteración debe progresar hacia la poscondición, por lo que deberá contener variables que son modificadas dentro del bucle.
- En última instancia el invariante es solamente una formalización de los objetivos del programador.

Ejemplo

Encontrar el invariante asociado al código interno del bucle:

```
1       sum←0;
2       j←0;
3       while (j !← n) {
4            sum←sum+a;
5            j←j+1;
6       }
```

Si el invariante es $I = \{sum = j \cdot a\}$

$$\{P\}\ j := j + 1\ \{sum = j \cdot a\}$$
$$P = \{j = j + 1, sum = j \cdot a\} \Rightarrow \{sum = (j + 1) \cdot a\}$$
$$\{sum = (j + 1) \cdot a\}\ j = j + 1\ \{sum = j \cdot a\}$$

$$\{P\}\ sum := sum + a\ \{sum = (j + 1) \cdot a\}$$
$$P = \{sum = sum + a, sum = (j + 1) \cdot a\} \Rightarrow \{sum + a = (j + 1) \cdot a\}$$
$$\{sum = j \cdot a\}\ sum = sum + a\ \{sum = (j + 1) \cdot a\}$$

Con lo que queda demostrado que el invariante propuesto es correcto, entonces:

$$\{I\}\ C\ \{I\} = \{sum = j \cdot a\}\ j := j + 1; sum := sum + a\ \{sum = j \cdot a\}$$

$$\{P\}\ j := 0\ \{sum = j \cdot a\}$$
$$P = \{j = 0, sum = j \cdot a\} \Rightarrow \{sum = 0\}$$
$$\{sum = 0\}\ j := 0\ \{sum = j \cdot a\}$$

$$\{P\}\ sum := 0\ \{sum = 0\}$$
$$P = \{sum = 0, sum = 0\} \Rightarrow \{\ \}$$
$$\{\ \}\ sum := 0\ \{sum = 0\}$$

$$\{(\neg B \wedge I)\} \Rightarrow \{(sum = j \cdot a) \wedge (j = n)\} \Rightarrow \{sum = n \cdot a\}$$

1.5.7.1. Regla de inferencia para el bucle *WHILE*

Si C es un código tal que se cumple $\{B \wedge I\}\ C\ \{I\}$ entonces se puede inferir lo siguiente:

$$\frac{\{B \wedge I\}\ C\ \{I\}}{\{I\}\ while\ B\ do\ C\ \{\neg B \wedge I\}}$$

Ejemplo

Determinar la corrección del siguiente código que contiene un bucle $while$:

```
1       f ← 1;
2       t ← 0;
3       i ← 0;
4       while(i ≪← n) {
5           t ← t + f;
6           f ← f * r;
7           i ← i + 1
8       }
```

Se supone que el código calcula $t = \sum_{i=0}^{n} r^i$ cuando $i > n$

Si la poscondicion es $(t = \sum_{i=0}^{n} r^i) \wedge (i > n)$ y el invariante $I = (t = \sum_{j=0}^{i-1} r^j) \wedge (f = r^j) \wedge (i \leq n + 1)$
Regla de la asignación en $i = i + 1$

$$\{P\} i := i + 1 \{(t = \sum_{j=0}^{i-1} r^j) \wedge (f = r^i) \wedge (i \leq n + 1)\}$$
$$P = \{i = i + 1, t = \sum_{j=1}^{i-1} r^j, f = r^i, i \leq n + 1\} \Rightarrow \{t = \sum_{j=1}^{i} r^j, f = r^{i+1}, i + 1 \leq n + 1\}$$
$$\{i + 1 \leq n + 1\} \Rightarrow \{i \leq n\} \Rightarrow \{i < n + 1\}$$
$$\{(t = \sum_{j=1}^{i} r^j) \wedge (f = r^{i+1}) \wedge (i < n+1)\} i := i+1 \{(t = \sum_{j=1}^{i-1} r^j) \wedge (f = r^i) \wedge (i \leq n+1)\}$$

Regla de la asignación en $f = f * r$

$$\{P\} f := f \cdot r; \{(t = \sum_{j=1}^{i} r^j) \wedge (f = r^{i+1}) \wedge (i < n + 1)\}$$
$$P = \{f = f \cdot r, t = \sum_{j=1}^{i} r^j, f = r^{i+1}, i < n + 1\} \Rightarrow$$
$$\Rightarrow \{t = \sum_{j=1}^{i} r^j, f \cdot r = r^{i+1}, i < n + 1\} \Rightarrow \{t = \sum_{j=1}^{i} r^j, f = r^i, i < n + 1\}$$
$$\{(t = \sum_{j=1}^{i} r^j) \wedge (f = r^i) \wedge (i < n+1)\} f := f \cdot r; \{(t = \sum_{j=1}^{i} r^j) \wedge (f = r^{i+1}) \wedge (i < n+1)\}$$

Regla de la asignación en $t = t + f$

$$\{P\}t := t + f; \{(t = \sum_{j=1}^{i} r^j) \wedge (f = r^i) \wedge (i < n + 1)\}$$
$$P = \{t = t + f, t = \sum_{j=1}^{i} r^j, f = r^i, i < n + 1\} \Rightarrow$$
$$\Rightarrow \{t + f = \sum_{j=1}^{i} r^j, f = r^i, i < n + 1\} \Rightarrow \{t + r^i = r^i + \sum_{j=1}^{i-1} r^j, f = r^i, i < n + 1\} \Rightarrow$$
$$\{t = \sum_{j=1}^{i-1} r^j, f = r^i, i < n + 1\}$$
$$\{(t = \sum_{j=1}^{i-1} r^j) \wedge (f = r^i) \wedge (i < n+1)\}t := t+f; \{(t = \sum_{j=1}^{i} r^j) \wedge (f = r^i) \wedge (i < n+1)\}$$

Usando la regla de la conjunción

$$\{(t = \sum_{j=1}^{i-1} r^j) \wedge (f = r^i) \wedge (i < n+1)\}t := t+f; f := f \cdot r; i := i+1\{(t = \sum_{j=1}^{i-1} r^j) \wedge (f = r^i) \wedge (i \leq n+1)\}$$

Con lo que queda demostrado que $\{B \wedge I\}C\{I\}$

Regla de la asignación en $i = 0$

$$\{P\}i := 0; \{(t = \sum_{j=1}^{i-1} r^j) \wedge (f = r^i) \wedge (i \leq n + 1)\}$$
$$P = \{i = 0, t = \sum_{j=1}^{i-1} r^j, f = r^i, i \leq n + 1\} \Rightarrow$$
$$\Rightarrow \{t = \sum_{j=1}^{-1} r^j, f = r^0, 0 \leq n + 1\} \Rightarrow \{t = 0, f = 1, 0 \leq n + 1\}$$
$$\{(t = 0) \wedge (f = 1) \wedge (0 \leq n + 1)\}i := 0; \{(t = \sum_{j=1}^{i-1} r^j) \wedge (f = r^i) \wedge (i \leq n + 1)\}$$

Regla de la asignación en $t = 0$

$$\{P\}t := 0; \{(t = 0) \wedge (f = 1) \wedge (-1 \leq n)\}$$
$$P = \{t = 0, t = 0, f = 1, -1 \leq n\} \Rightarrow \{0 = 0, f = 1, -1 \leq n\}$$
$$\{(f = 1) \wedge (-1 \leq n)\}t := 0; \{(t = 0) \wedge (f = 1) \wedge (-1 \leq n)\}$$

Regla de la asignación en $f = 1$

$$\{P\}f := 1; \{(f = 1) \wedge (-1 \leq n)\}$$
$$P = \{f = 1, f = 1, -1 \leq n\} \Rightarrow \{1 = 1, -1 \leq n\}$$
$$\{-1 \leq n\}f := 1; \{(f = 1) \wedge (-1 \leq n)\}$$
$$\{(i > n)\} \Rightarrow \{i = n + 1\}$$

Para finalizar trabajamos con la poscondición del $while$, $\{(\neg B \wedge I)\}$

$$\begin{aligned} \{(\neg B \wedge I)\} &\Rightarrow \{(i = n + 1) \wedge ((t = \sum_{j=0}^{i-1} r^j) \wedge (f = r^i) \wedge (i \leq n + 1))\} \\ &\Rightarrow \{((t = \sum_{j=0}^{n} r^j) \wedge (f = r^{n+1}) \wedge (i = n + 1))\} \end{aligned} \tag{1.4}$$

Verificación del programa, analizando cada terna

$$
\begin{array}{rlr}
\{-1 \leq n\} & f := 1; & \{(f = 1) \wedge (-1 \leq n)\} \\
\{(f = 1) \wedge (-1 \leq n)\} & t := 0; & \{(t = 0) \wedge (f = 1) \wedge (-1 \leq n)\} \\
\{(t = 0) \wedge (f = 1) \wedge (0 \leq n + 1)\} & i := 0; & \{(t = \sum_{j=1}^{i-1} r^j) \wedge (f = r^i) \wedge (i \leq n + 1)\} \\
while & i \leq n & do \\
\{(t = \sum_{j=1}^{i-1} r^j) \wedge (f = r^i) \wedge (i < n + 1)\} & t := t + f; & \{(t = \sum_{j=1}^{i} r^j) \wedge (f = r^i) \wedge (i < n + 1)\} \\
\{(t = \sum_{j=1}^{i} r^j) \wedge (f = r^i) \wedge (i < n + 1)\} & f := f \cdot r; & \{(t = \sum_{j=1}^{i} r^j) \wedge (f = r^{i+1}) \wedge (i < n + 1)\} \\
\{(t = \sum_{j=1}^{i} r^j) \wedge (f = r^{i+1}) \wedge (i < n + 1)\} & i := i + 1; & \{(t = \sum_{j=1}^{i-1} r^j) \wedge (f = r^i) \wedge (i \leq n + 1)\}
\end{array}
$$

y finnalmente, por 1.4, se demuestra la correctitud del programa

1.5.7.2. Regla de inferencia para el bucle *REPEAT*

Si C es un código tal que se cumple $\{I\}\ C\ \{Q\}$ donde $I = \{\neg B \wedge Q\}$ entonces se puede inferir lo siguiente:

$$\frac{\{I\}\ C\ \{Q\}}{\{I\}\ repeat\ C\ until\ B\ \{B \wedge Q\}}$$

```
1  //DOCUMENTACION EN PSEUDOCODIGO
2  //**************************************
3  //* Calcula el producto de dos enteros *
4  //* Argumentos: x>0 *
5  //* Resultado: z *
6  //**************************************
7  MODULO ejemplo (ENTERO x, ENTERO u)
8      {Prec : x > 0; Dec : u − 1}
9      z:←0; u:←x;
10     REPETIR{z + u ∗ y = x ∗ y ∧ u > 0}
11         z:←z+y;
12         u:← u−1
13     HASTA (u←0);
14 FIN MODULO {Pos : z = x ∗ y ∧ u = 0}
```

1.6. Lenguaje de Programación

Existen lenguajes de programación para la especificación formal y la verificación. Dafny como muchas herramientas dependen del uso de probado automático de teoremas para para probar, se compila en el lenguaje intermediario Boogie, que usa el probador automático de teoremas.

Dafny es un lenguaje compilado imperativo que permite la especificación formal a través de precondiciones, postcondiciones, invariantes de bucles y variantes de bucles. El lenguaje combina ideas principalmente de los paradigmas funcional e imperativo, e incluye soporte limitado para Programación Orientada a Objetos. Las características incluyen genéricos, asignación

dinámica, tipos de datos inductivos y una variación de lógica de separación conocida como marcos implícitos dinámicamente1 para inferencia de efectos colaterales. Dafny fue creado por por Microsoft, es empleado en la enseñanza y aparece regularmente en competiciones de verificación de software

Dafny es un lenguaje diseñado para facilitar la escritura de código correcto. correcto en el sentido de no tener ningún error de tiempo de ejecución, Y hacer lo que el programador pretendía que hiciera. El efecto de un fragmento de código puede darse de forma abstracta, utilizando una expresión natural de alto nivel del comportamiento deseado, que es más fácil y menos propenso a errores de escritura. Dafny luego genera una prueba de que el código coincide con las anotaciones. Escribir anotaciones sin errores suele ser más fácil que escribir código, ya que las anotaciones son más cortas y directas.

Dafny tiene métodos, variables, tipos, bucles, sentencias alternativas, matrices, enteros y más. Un método es una trozo imperativo, ejecutable de código. En otros lenguajes, podrían llamarse procedimientos o funciones, pero en Dafny Un método es declarado de la siguiente manera:

```
method UnaSalida(x: int) returns (r: int)
    {
      . . .
    }
method MuchasSalidas(x: int, y: int) returns (more: int, less: int)
   {
     . . .
   }
```

Las asignaciones se utilizan mediante := y la igualdad mediante == Las precondiciones se utilizan con el comando $requires$, las poscondiciones usan el comando $ensures$, y puede haber más de una. Puede ser escrita en forma completa o abreviada.

A diferencia de las pre y poscondiciones las aserciones se coloca en algún lugar en el medio de un método.

Ejemplo:

```
method MultipleReturns(x: int, y: int) returns (more: int, less: int)
    requires 0 < y;
    ensures less < x < more;
{
    more :← x + y;

    less :← x - y;
}
```

La poscondición podría también ser escrita así:

```
ensures less < x && x < more;
```

1.6.0.1. Programa totalmente correcto

El programa es parcialmente correcto, se puede asegurar que si termina lo hace satisfaciendo la postcondición final. Si el programa finaliza, entonces se puede afirmar que es totalmente correcto.

- Sólo es necesario definir la finalización cuando se trabajan con bucles.
- Para analizar la finalización hay que estudiar la evolución a través de las iteraciones de las variables que están involucradas en la condición de salida.
- El vector variantes es el de todas las variables que aparecen en la condición de salida y que pueden ver modificado su valor durante la ejecución del bucle.
- Si cada secuencia de vectores variantes alcanza la condición de salida tras un número finito de iteraciones entonces el bucle termina.

Para determinar la corrección de un loop, Dafny determina si las siguientes afirmaciones se cumplen:

1. La invariante se cumple antes del comienzo del loop.
2. La invariante se cumple luego de ejecutar el cuerpo del loop.
3. La invariante y la negación de la condición del loop implican la propiedad deseada al final del loop.
4. La variante no es negativa cuando la invariante del loop se mantiene.
5. La variante decrementa su valor en cada paso del loop siempre y cuando la invariante y la condición del loop se mantengan.

Estas reglas son el fundamento teórico para poder afirmar que Dafny está basado en la lógica de Hoare y en el método de la variante.

Por ejemplo en un código con bucle demostrar que para $n \geq 0$ el programa siguiente termina:

Ejemplo programa totalmente completo

```
1       i ← 0;
2       f ← 1;
3       while (i < n-1) {
4           i ← i + 1;
5           f ← f * r;
6       }
```

La única variable de la condición de salida presente en el bucle es i. Por tanto el vector

variante es i. El valor inicial de i es 0.

- Si $n = 0$ entonces no se cumple la condición de entrada y no se ejecuta el bucle. **El programa finaliza correctamente**.
- Si $n > 0$ entonces i alcanzará la condición de salida $i = n$ en número finito de iteraciones igual a n ya que en cada iteración se ve incrementado en una unidad. **El programa finaliza correctamente**.

Ejemplos de código en Dafny. a) Invariantes y aserciones:

```
var i :← 0;
while (i < n)
    invariant 0 ≤← i;
    {
        i :← i + 1;
    }
assert i = n;
```

b) uso de arreglos

```
method Find(a: array<int>, int key) returns (index: int)
    requires . . .
    ensures 0 ≤← index => index < a.Length && a[index] = key;
    {
    . . .
    }
```

c) cuantificadores

$$forallk : int :: 0 <= k < a.Length ==> 0 < a[k]$$

Esto significa que para todos los enteros k que índice en la matriz (por eso va de 0 hasta la dimensión del arreglo). Al escribir estas anotaciones, uno está seguro de que el código es correcto. Además, el solo hecho de escribir las anotaciones puede ayudar a comprender qué es el código.

Decir que la clave no está en el arreglo con un cuantificador:

$$forallk :: 0 <= k < a.Length ==> a[k]! = key$$

```
method BinarySearch(a: array<int>, key: int) returns (index: int)
    requires a !← null && sorted(a);
    ensures 0 ≤← index => index < a.Length && a[index] = key;
    ensures index < 0 => forall k :: 0 ≤← k < a.Length => a[k] !←
        key;
```

```
5       {
6           . . .
7       }
```