Trabajo Fin de Máster

Verificación formal de la lógica de Hoare en Isabelle/HOL

Presentado por:
Natividad González Blanco

Tutora:
Dra. María José Hidalgo Doblado,
Universidad de Sevilla

Septiembre de 2016

Abstract

Hoare logic is a formal system developed by C.A.R. Hoare. This logic was introduced to verify formally imperative programs. That is, with the Hoare logic we can ensure and prove that a particular program performs exactly the actions for which it has been designed. An advantage to do it through a proof assistant is that in this way errors can be detected that in the handmade proofs could go unnoticed.

In this dissertation, we will describe briefly the interactive theorem prover Isabelle/HOL. Then, we will present in detail the Hoare logic with examples. The major goal consists of an implementation of this logic in the proof assistant Isabelle/HOL.

According as the size of a system grows, the costs of formal verification increase disproportionately. So some people think that formal verification isn't very profitable but there are some situations in which this is vitally important such as braking systems of cars and aircraft piloting by electronic controls.

Índice general

Capítulo 1

Introducción

La lógica de Hoare es un sistema formal desarrollado por el científico C.A.R. Hoare en 1969. En esta publicación, Hoare mencionó la ayuda de algunos de los trabajos anteriores de Robert Floyd. La lógica de Hoare se introdujo para poder verificar formalmente programas imperativos, es decir, para razonar sobre la corrección de tales programas. De otra forma, sirve para comprobar y demostrar que realmente un programa realiza las acciones para las que ha sido diseñado. En algunas situaciones esto es de vital importancia, como es el caso de los sistemas de frenado de coches y el pilotaje de aviones por mandos electrónicos. Este propósito lógico especial se obtiene introduciendo un lenguaje que contiene comandos básicos con el que se construyen los programas, el lenguaje determinista IMP, y una formulación con la que poder expresar el comportamiento de los programas. Además, es necesario un conjunto de reglas de cálculo para simplificar expresiones o deducir otras. Nuestro lenguaje estará formado por los siguiente comandos: asignaciones a variables numéricas enteras, condicionales, bucles tipo WHILE y composición secuencial de comandos.

El sistema de la lógica de Hoare consiste en un lenguaje formado por ternas de la forma $\{P\}\ C\ \{Q\}$, donde C es un programa, P es una precondición y Q es una postcondición (ambas escritas en el lenguaje de la lógica matemática). Esta expresión corresponde a la correción parcial del programa C bajo la precondición P y la postcondición Q. Se dirá que una terna de este tipo es verdadera si se cumple que: si ejecutamos el programa C a partir de un estado inicial que cumple la precondición P, y, además, el programa C para dicho estado inicial termina, entonces, el estado final tras la ejecución de C satisface la postcondición Q. En estas condiciones si una terna $\{P\}\ C\ \{Q\}$ es verdadera lo que se puede asegurar acerca del programa es que bajo los estados iniciales para los que se haya ejecutado (que cumplan la precondición), en caso de que termine devuelve el cálculo deseado. Por lo tanto, para asegurarse de que el programa siempre que termine devuelve tal cálculo, habría que computar uno a uno todos los posibles estados que admite la precondición. Puede suceder que el conjunto de estados que cumplan la precondición no sea finito. Es por eso que surge la noción de demostrabilidad. Se construye un sistema deductivo que permite demostrar ternas de Hoare. Una terna de Hoare se dirá demostrable si existe una prueba en la lógica de Hoare de dicha terna. De esta forma queda

verificado el comportamiento del programa, en caso de que así sea; y que no existen situaciones bajo las cuales C presente un comportamiento distinto al esperado. Más tarde, surgen los teoremas de adecuación y completitud de la lógica de Hoare, en los que se demuestra la equivalencia de la veracidad y demostrabilidad de una terna.

En última instancia, lo que se desea es comprobar y demostrar si un determinado programa termina bajo cualquier estado inicial que admita y, que en ese caso, se tiene el cálculo esperado. Esto es, corrección total de la lógica de Hoare. Es otro tipo de corrección más fuerte que la parcial y se puede expresar mediante la ecuación

$$\text{Corrección total} = \text{Corrección parcial} + \text{Terminación}$$

Para ello se extiende el conjunto de reglas de la lógica de Hoare. Teniendo en cuenta nuestro pequeño lenguaje, la única situación bajo la que podría ser que un programa no termine es en caso de que contenga un comando WHILE. En las demás situaciones siempre se tiene la terminación. Por lo tanto, es fácil deducir que en dichas situaciones las correcciones parcial y total son equivalentes. Por lo tanto, la única regla que es ampliada es la referente al bucle WHILE.

Isabelle/HOL es un demostrador interactivo de teoremas. Concretamente, Isabelle es una herramienta genérica que sirve para implementar formalismos matemáticos y para ayudar en la demostración de estos. Este demostrador fue desarrollado por los científicos Tobías Nipkow y Lawrence C. Paulson. Por otro lado, HOL, del inglés Higher Order Logic, es otro sistema interactivo predecesor de Isabelle. Isabelle/HOL es la implementación de Isabelle para HOL. Los sistemas formales como Isabelle/HOL permiten la formalización y verificación de ternas de Hoare.

A lo largo del tiempo, Tobias Nipkow, junto con Gerwin Klein y Markus Wenzel, entre otros, han continuado con el desarrollo de trabajos en este demostrador. Cabe mencionar los tutoriales ([3]) y ([4]), así como el libro ([8]), fuentes importantes para la realización de este trabajo.

El propósito principal de este Trabajo de Fin de Máster es presentar la formalización de la lógica de Hoare en Isabelle/HOL, así como, las pruebas de los teoremas de adecuación y completitud nombrados anteriormente.

Como se ha explicado antes, la verificación formal de un programa puede realizarse utilizando la lógica de Hoare. El valor de hacerlo con ayuda de un demostrador como Isabelle/HOL es que, además de que permite verificar las ternas de corrección parcial y total, permite verificar las correcciones de programas concretos. Es decir, se pueden detectar errores que en las demostraciones a "mano" podrían pasar desapercibidos.

En cuanto a la organización del trabajo, se ha dividido en tres partes bien diferenciadas.

Capítulo 2: se describe de forma breve y con ejemplos el sistema de razonamiento automático Isabelle/HOL.

Capítulo 3: se explica detalladamente en qué consiste la lógica de Hoare. Se muestran ejemplos de programas que son correctos totalmente y otros que, en cambio, sólo son correctos parcialmente. Además, también se exponen ejemplos de ternas demostrables. Y, como consecuencia, se tratan la adecuación y completitud de esta lógica.

Capítulo 4: es el objetivo principal del trabajo. Consiste en una formalización del capítulo anterior en Isabelle/HOL. De esta manera se le da credibilidad a las demostraciones de los ejemplos expuestos en el Capítulo 3.

En lo referente a los trabajos relacionados, la lógica de Hoare y su extensión también se han formalizado en Coq, ([9]), y en HOL, ([10]), respectivamente.

En cuanto a la posibilidad de extensión del trabajo pueden ser las condiciones de verificación, cuyo objetivo es transformar la noción de demostrabilidad de la lógica de Hoare en demostrabilidad de asertos (predicados sobre estados).

Tras la realización de este trabajo he desarrollado conocimientos acerca de la verificación formal de un programa. También he aprendido sobre una lógica que desconocía, la lógica de Hoare, cúal es su utilidad y formalización, como se explicó al principio. Además, aunque el demostrador Isabelle/HOL sí que me era familiar, ahora he adquirido más experiencia en el lenguaje y proceso de formalización en él. Especialmente, sobre la formalización de la lógica de Hoare mediante este demostrador. Ahora tengo mayor conciencia sobre la importancia y ventajas de la formalización de la lógica y las teorías en un demostrador como el utilizado.

Capítulo 2

Introducción a Isabelle/HOL

En este capítulo se describe brevemente el sistema de razonamiento automático Isabelle/HOL. Para ello, se ha seguido el curso de *Lógica Matemática y Teoría de Modelos*, complementándose con ([3]) y ([4]).

Isabelle tiene un sistema de deducción natural, la capacidad de inferenciar tipos para poder verificar si los términos que se utilicen sean correctos o no y la de generar esquemas de inducción de forma estructural. También se caracteriza porque contiene conjuntos y tipos de datos recursivos. Además, se pueden realizar demostraciones interactivas.

Otra de sus características es que se organiza en módulos llamados *teorías*. La estructura de una *teoría* es:

`theory` T
`imports` B_1`,...,` B_n
`begin`
declaraciones, definiciones y pruebas
`end`

donde B_1`,...,` B_n son los nombres de las teorías existentes en las que se basa `T` y *declaraciones, definiciones y pruebas* son los nuevos conceptos y pruebas que se introducen.

2.1. Programación funcional en Isabelle/HOL

En esta sección se presenta el lenguaje funcional que está incluido en Isabelle.

2.1.1. Números naturales, enteros y booleanos

En Isabelle están definidos los números naturales con la sintaxis de Peano usando dos constructores: 0 (cero) y *Suc* (el sucesor). Los números naturales son abreviaturas de los correspondientes en la notación de Peano. Por ejemplo, el 2 es, en este caso, "*Suc* (*Suc* 0)".

El tipo de los números naturales es `nat`. La notación del par de dos puntos se usa para asignar un tipo a un término (por ejemplo, `(1::nat)` significa que se considera

que 1 es un número natural.

Las operaciones aritméticas $+, -, *$, `div`, `mod`, `min` y `max` están predefinidas, así como las relaciones $\leq$ y $<$.

También están definidos los números enteros. El tipo de los enteros se representa por `int`.

Por lo tanto, los numerales y las operaciones nombradas anteriormente están sobrecargados. Por ejemplo, el 1 puede ser un natural o un entero, dependiendo del contexto. Isabelle resuelve ambigüedades mediante inferencia de tipos. A veces, es necesario usar declaraciones de tipo para resolver dicha ambigüedad.

En Isabelle también están definidos los valores booleanos (`True` y `False`). El tipo de los booleanos se representa por `bool`. Sus funciones predefinidas asociadas son las conectivas ($\neg$, $\wedge$, $\vee$, $\longrightarrow$ y $\leftrightarrow$).

Ambos, el tipo de datos y las funciones siguen el esquema de la sintaxis de los lenguajes de programación funcional. Para más explicitud, se muestra la conjunción definida mediante reconocimiento de patrones:

```
fun conj :: "bool ⇒ bool ⇒ bool" where
 "conj True True = True"
|"conj _ _ = False"
```

Para poder enunciar las proposiciones que sugen de las definiciones Isabelle utiliza la palabra clave `lemma`, `theorem` o `corollary` indistintamente. Así, un lema, teorema o corolario introduce una proposición seguida de una demostración. Isabelle dispone de varios procedimientos automáticos para generar demostraciones, uno de los cuales es el de simplificación (llamado `simp`). El procedimiento `simp` aplica un conjunto de reglas de reescritura, que inicialmente contiene un gran número de reglas relativas a los objetos definidos. Veamos un ejemplo de `simp`:

```
lemma "⟦ xs @ zs = ys @ xs; [] @ xs = [] @ [] ⟧ ⟹ ys = zs"
by simp
```

Como ya se ha dicho, `simp` es un método de demostración. El formato general de este método es

`simp` *lista de modificadores*,

donde la lista de *modificadores* especifica las propiedades y definiciones que se van a utilizar. Este método sólo ataca el primer subobjetivo. Por lo tanto, se utiliza `simp_all` para tratar de simplificar todos los subobjetivos. Si nada cambia, la prueba no se puede realizar con este método.

2.1.2. Definiciones no recursivas

La definiciones no recursivas se pueden expresar con el comando `definition`. Tales definiciones no siguen ningún patrón. Una definición no recursiva se expresa

como $fx_1, \ldots, x_n = t$, donde f no ocurre en t. El símbolo $\equiv$ es una forma de igualdad que se utiliza en las definiciones constantes. Veamos un ejemplo, la disyunción exclusiva, `xor`, de dos fórmulas A y B se verifica si una es verdadera y la otra no lo es.

```
definition xor :: "bool ⇒ bool ⇒ bool" where
  "xor A B ≡ (A ∧ ¬B) ∨ (¬A ∧ B)"
```

Si escribimos `thm "xor_def"` Isabelle nos muestra la definición que hemos realizado.

A partir de la definición anterior se puede probar la siguiente proposición, que se demuestra por simplificación:

Proposición 2.1.1 *La disyunción exclusiva de dos fórmulas verdaderas es falsa.*

```
lemma "xor True True = False"
by (simp add: xor_def)
```

Con la expresión `declare xor_def [simp]` se añade la definición de la disyunción exclusiva al conjunto de reglas de simplificación automáticas. De esta manera no se tiene que indicar qué definiciones o lemas se deben usar.

```
declare xor_def [simp]

lemma "xor True False = True"
by simp
```

2.1.3. Definiciones locales

Las definiciones locales consisten en asignar valores concretos a las variables de una expresión. Estas asignaciones se realizan con los comandos `let` e `in`. Por ejemplo, si x es el número natural 3, entonces $x \cdot x = 9$.

```
value "let x = 3::int in x * x" -- "= 9"
```

2.1.4. Pares

Un par se representa escribiendo dos elementos entre paréntesis y separados por coma, (a, b). El tipo de los pares es el producto de los tipos. Es decir, el tipo del par (a, b) es $\tau_1 \times \tau_2$, tal que a es de tipo τ_1 y b de τ_2.

La funciones `fst` y `snd` devuelven el primer y segundo elemento del par. Por ejemplo, si p es el par de números naturales $(2, 3)$, entonces la suma del primer elemento de p y 1 es igual al segundo elemento de p.

```
value "let p = (2,3)::nat × nat in fst p + 1 = snd p"
```

2.1.5. Listas

Una lista se representa escribiendo los elementos entre corchetes y separados por comas. La lista vacía se representa por `[]` o `Nil`. Además, todos los elementos de una lista tienen que ser del mismo tipo. Una lista de elementos de tipo `'a` es la lista Vacía o se obtiene añadiendo, con `Cons`, un elemento de tipo `'a` a una lista de elementos de tipo `'a`. El tipo de las listas de elementos del tipo `'a` es `'a list`.

```
datatype 'a Lista = Vacia | Cons 'a "'a Lista"
```

El término `(x#xs)` representa la lista obtenida añadiendo el elemento `x` al principio de la lista `xs`. Por ejemplo, la lista obtenida añadiendo sucesivamente a la lista vacía los elementos `c`, `b` y `a` es `[a,b,c]`.

Dos funciones de descomposición de listas son:

- `(hd xs)` es el primer elemento de la lista `xs`.
- `(tl xs)` es el resto de la lista `xs`.

Por ejemplo, si `xs` es la lista `[a,b,c]`, entonces el primero de `xs` es `a` y el resto de `xs` es `[b,c]`.

```
value "let xs = [a,b,c] in hd xs = a ∧ tl xs = [b,c]" -- "= True"
```

Otra función de listas es `(length xs)`, que representa la longitud de la lista `xs`.

En "What's in Main" (página 8) se encuentran más definiciones y propiedades de las listas.

2.1.6. Funciones anónimas

En Isabelle pueden definirse funciones anónimas mediante expresiones lambda. Por ejemplo, el valor de la función que a un número le asigna su doble aplicada a 1 es 2.

```
value "(λx. x + x) 1::int" -- "= 2"
```

2.1.7. Condicionales

Isabelle/HOL también es compatible con algunos constructores básicos de la programación funcional, como las expresiones condicionales. Estas expresiones se pueden utilizar para definir funciones. Presentan una de las dos estructuras siguientes:

- `if` b `then` t_1 `else` t_2. Aquí b es de tipo `bool` y t_1 y t_2 son del mismo tipo. Veamos un ejemplo:

  El valor absoluto del entero x es x si $x \geq 0$ y es $-x$ en caso contrario.

```
definition absoluto :: "int ⇒ int" where
"absoluto x ≡ (if x ≥ 0 then x else -x)"
```

- (`case` e `of`
  $c_1 \Rightarrow e_1$
  $|c_2 \Rightarrow e_2$
  $\vdots$
  $|c_n \Rightarrow e_n$)

  Se evalúa e_i si e es de la forma c_i. Veamos un ejemplo:

  Un número natural n es un sucesor si es de la forma $(Suc\ m)$.

```
definition es_sucesor :: "nat ⇒ bool" where
  "es_sucesor n ≡ (case n of
    0     ⇒ False
  | Suc m ⇒ True)"
```

2.1.8. Definiciones recursivas

Generalmente, definir una función recursiva es tan simple como en los otros casos. La sintaxis es bastante autoexplicativa: se componen de un conjunto de ecuaciones recursivas. Por ejemplo, la *sucesión de Fibonacci*:

```
fun fib :: "nat ⇒  nat " where
 "fib 0 = 1"
|"fib (Suc 0) = 1"
|"fib (Suc(Suc n)) = fib n + fib (Suc n)"
```

2.2. Razonamiento sobre programas en Isabelle/HOL

En esta sección se explica cómo demostrar con Isabelle las propiedades de los programas funcionales mediante algunos ejemplos.

2.2.1. Razonamiento ecuacional

Se define la función intercambia tal que `(intercambia p)` es el par obtenido intercambiando las componentes del par `p`.

```
fun intercambia :: "'a × 'b ⇒ 'b × 'a" where
  "intercambia (x,y) = (y,x)"
```

Se puede probar el siguiente resultado.

Proposición 2.2.1 *Si se aplica dos veces la función* `intercambia` *al par* `p` *no se produce ningún cambio. Es decir,* `intercambia (intercambia (x,y)) = (x,y)`.

• La demostración detallada es:

```
lemma "intercambia (intercambia (x,y)) = (x,y)"
proof -
  have "intercambia (intercambia (x,y)) = intercambia (y,x)"
    by (simp only: intercambia.simps)
  also have "... = (x,y)"
    by (simp only: intercambia.simps)
  finally show "intercambia (intercambia (x,y)) = (x,y)" by simp
qed
```

En cuanto al lenguaje empleado en la demostración anterior se puede decir que se utiliza

- `proof` para iniciar la prueba,
- `-` (después de `proof`) para no usar el método por defecto,
- `have` para establecer un paso,
- `by (simp only: intercambia.simps)` para indicar que sólo se usa como regla de escritura la correspondiente a la definición de `intercambia`,
- `also` para encadenar pasos ecuacionales,
- `...` para representar la igualdad anterior en un razonamiento ecuacional,
- `finally` para indicar el último paso de un razonamiento ecuacional,
- `show` para establecer la conclusión.
- `by simp` para indicar el método de demostración por simplificación y
- `qed` para terminar la prueba.

La definición de la función `intercambia` genera una regla de simplificación. Si escribimos en Isabelle `thm intercambia.simps` se puede ver que dicha regla es `intercambia.simps: intercambia (x,y) = (y,x)`.

Cada lema, también se puede demostrar de forma estructurada, sin explicar cada paso.

• La demostración estructurada en este caso es:

```
lemma "intercambia (intercambia (x,y)) = (x,y)"
proof -
  have "intercambia (intercambia (x,y)) = intercambia (y,x)"  by simp
  also have "... = (x,y)" by simp
  finally show "intercambia (intercambia (x,y)) = (x,y)" by simp
qed
```

La diferencia entre las dos demostraciones es que en los dos primeros pasos de la segunda no se explicita la regla de simplificación.

Por último, las proposiciones también se pueden demostrar automáticamente, sin explicar ninguno de los pasos que se lleve a cabo en la demostración.

• La demostración automática de la proposición 2.2.1 es:

```
lemma "intercambia (intercambia (x,y)) = (x,y)"
by simp
```

2.2.2. Razonamiento por inducción sobre los naturales

Teorema 2.2.1 (Principio de inducción sobre los naturales) *Para demostrar que una determinada propiedad* `P` *es cierta para todos los números naturales basta probar que el* 0 *satisface la propiedad y que si es cierta para* **n**, *entonces también lo es para* **n**+1.

En Isabelle el principio de inducción sobre los naturales está formalizado en el teorema `nat.induct` y puede verse con `thm nat.induct`:

```
⟦P 0; ⋀n. P n ⟹ P (Suc n)⟧ ⟹ P m
```

Veamos un ejemplo de demostración por inducción sobre los naturales. Para ello, primero se define la función `suma_impares` tal que `(suma_impares n)` es la suma de los `n` primeros números impares.

```
fun suma_impares :: "nat ⇒ nat" where
  "suma_impares 0 = 0"
| "suma_impares (Suc n) = (2*(Suc n) - 1) + suma_impares n"
```

Proposición 2.2.2 *La suma de los n primeros números impares es* n^2. *Es decir,*

$$1 + 3 + ... + (2n - 1) = n^2$$

• Demostración de la proposición anterior por inducción y razonamiento ecuacional:

```
lemma "suma_impares n = n * n"
proof (induct n)
  show "suma_impares 0 = 0 * 0" by simp
next
  fix n assume HI: "suma_impares n = n * n"
  have "suma_impares (Suc n) = (2 * (Suc n) - 1) + suma_impares n" by simp
  also have "... = (2 * (Suc n) - 1) + n * n" using HI by simp
  also have "... = n * n + 2 * n + 1" by simp
  finally show "suma_impares (Suc n) = (Suc n) * (Suc n)" by simp
qed
```

• Demostración de la proposición anterior con patrones y razonamiento ecuacional:

```
lemma "suma_impares n = n * n" (is "?P n")
proof (induct n)
  show "?P 0" by simp
next
  fix n
  assume HI: "?P n"
  have "suma_impares (Suc n) = (2 * (Suc n) - 1) + suma_impares n" by simp
  also have "... = (2 * (Suc n) - 1) + n * n" using HI by simp
  also have "... = n * n + 2 * n + 1" by simp
  finally show "?P (Suc n)" by simp
qed
```

Comentarios sobre la demostración anterior:

- Con la expresión `"suma_impares n = n * n"(is "?P n")` se abrevia `"suma_impares n = n * n"` como `"?P n"`. Por tanto,
  `"?P 0"` es una abreviatura de `"suma_impares 0 = 0 * 0"` y,
  `"?P (Suc n)"` es una abreviatura de
  `"suma_impares (Suc n) = (Suc n) * (Suc n)"`
- En general, cualquier fórmula seguida de `(is patrón)` equipara el patrón con la fórmula.

• La demostración usando patrones es:

```
lemma "suma_impares n = n * n" (is "?P n")
proof (induct n)
  show "?P 0" by simp
next
  fix n
  assume "?P n"
  then show "?P (Suc n)" by simp
qed
```

• La demostración automática es:

```
lemma "suma_impares n = n * n"
by (induct n) auto
```

El método `auto` es un poco más fuerte que `simp`. Este método combina el razonamiento clásico con el de simplificación. La diferencia entre `auto` y los demás es que intenta llevar a cabo todos los subobjetivos de la demostración, por lo que, desafortunadamente, puede producir un gran número de nuevos subobjetivos.

Veamos ahora, un ejemplo de definición con cuantificadores o existenciales. Podemos decir, que un número natural n es par si existe un natural m tal que $n = m+m$. Por lo tanto, definir la función `par` tal que (par n) devuelve, en caso de existir, la mitad de n.

```
definition par :: "nat ⇒ bool" where
  "par n ≡ ∃m. n=m+m"
```

Esta definición puede verse con `thm par_def`.

La siguiente proposición se demuestra mediante inducción y existenciales:

Proposición 2.2.3 *Para todo número natural n, se verifica que $n(n+1)$ es par.*

- Demostración detallada por inducción:

```
lemma
  fixes n :: "nat"
  shows "par (n*(n+1))"
proof (induct n)
  show "par (0*(0+1))" by (simp add: par_def)
next
  fix n
  assume "par (n*(n+1))"
  then have "∃m. n*(n+1) = m+m" by (simp add:par_def)
  then obtain m where m: "n*(n+1) = m+m" ..
  then have "(Suc n)*((Suc n)+1) = (m+n+1)+(m+n+1)" by auto
  then have "∃m. (Suc n)*((Suc n)+1) = m+m" ..
  then show "par ((Suc n)*((Suc n)+1))" by (simp add:par_def)
qed
```

Comentarios sobre la demostración:

La declaración `(fixes n :: "nat")` es una abreviatura de "sea n un número natural".

En Isabelle puede demostrarse de manera más simple un lema equivalente usando en lugar de la función `par` la función `even` definida en la teoría `Parity` por

```
even x ⟷ x mod 2 = 0

lemma
  fixes n :: "nat"
  shows "even (n*(n+1))"
by auto
```

Comentarios sobre la demostración anterior:

Para poder usar la función `even` de la librería Parity es necesario importar dicha librería. Por ello, antes del inicio de la teoría aparece `imports Main Parity`.

Para completar la demostración basta demostrar la equivalencia de las funciones `par` y `even`.

```
lemma
  fixes n :: "nat"
  shows "par n = even n"
proof -
  have "par n = (∃m. n = m+m)" by (simp add:par_def)
  then show "par n = even n" by presburger
qed
```

Comentarios sobre la demostración anterior:

`by presburger` indica que se use como método de demostración el algoritmo de decisión de la aritmética de Presburger, implementado en Isabelle.

2.2.3. Razonamiento por inducción sobre listas

En Isabelle puede hacerse inducción estructural sobre cualquier tipo recursivo. La inducción matemática es la inducción sobre los naturales. Este apartado se centra en la inducción sobre listas.

Para demostrar una propiedad para todas las listas basta demostrar que la lista vacía tiene la propiedad y que al añadir un elemento a una lista que tiene la propiedad se obtiene otra lista que también tiene la propiedad.

En Isabelle el principio de inducción sobre listas está formalizado mediante el teorema `list.induct`

```
⟦P []; ⋀x xs. P xs ⟹ P (x#xs)⟧ ⟹ P xs
```

A continuación se muestra un ejemplo.

Se define la función `conc` tal que `(conc xs ys)` es la concatención de las listas `xs` e `ys`.

```
fun conc :: "'a list ⇒ 'a list ⇒ 'a list" where
  "conc []     ys = ys"
| "conc (x#xs) ys = x # (conc xs ys)"
```

A partir de la definición anterior se puede demostrar la proposición siguiente.

Proposición 2.2.4 *Dadas dos listas cualesquiera,* `xs` *e* `ys`*, siempre se verifica que*

`conc xs (conc ys zs) = (conc xs ys) zs.`

- La demostración estructurada es:

```
lemma "conc xs (conc ys zs) = conc (conc xs ys) zs"
proof (induct xs)
  show "conc [] (conc ys zs) = conc (conc [] ys) zs" by simp
next
  fix x xs
  assume HI: "conc xs (conc ys zs) = conc (conc xs ys) zs"
  have "conc (x # xs) (conc ys zs) = x # (conc xs (conc ys zs))" by simp
  also have "... = x # (conc (conc xs ys) zs)" using HI by simp
  also have "... = conc (conc (x # xs) ys) zs" by simp
  finally show "conc (x # xs) (conc ys zs) =
                conc (conc (x # xs) ys) zs" by simp
qed
```

Comentarios sobre la demostración anterior:

- `(induct xs)` genera dos subobjetivos:
  1. `conc [] (conc ys zs) = conc (conc [] ys) zs`
  2. $\bigwedge$`a xs. conc xs (conc ys zs) = conc (conc xs ys) zs` $\Longrightarrow$ `conc (a#xs) (conc ys zs) = conc (conc (a#xs) ys) zs`

- La demostración automática de 2.2.4 es:

```
lemma "conc xs (conc ys zs) = conc (conc xs ys) zs"
by (induct xs) auto
```

En Isabelle también se pueden refutar resultados, generándose un contraejemplo mediante `Quickcheck` o `Nitpick`. En este caso, en relación con la función `conc` se puede refutar que

`conc xs ys = conc ys xs.`

```
lemma "conc xs ys = conc ys xs"
quickcheck
oops
```

El contraejemplo que Isabelle encuentra es

```
xs = [a2]
  ys = [a1]
```

2.2.4. Inducción correspondiente a la definición recursiva

Se define la función `coge` tal que `(coge n xs)` es la lista de los `n` primeros elementos de `xs`.

```
fun coge :: "nat ⇒ 'a list ⇒ 'a list" where
  "coge n []           = []"
| "coge 0 xs           = []"
| "coge (Suc n) (x#xs) = x # (coge n xs)"
```

Por otro lado, se define la función `elimina` tal que `(elimina n xs)` es la lista obtenida eliminando los `n` primeros elementos de `xs`.

```
fun elimina :: "nat ⇒ 'a list ⇒ 'a list" where
  "elimina n []           = []"
| "elimina 0 xs           = xs"
| "elimina (Suc n) (x#xs) = elimina n xs"
```

La definición `coge` genera de forma automática el esquema de inducción `coge.induct`:

```
⟦⋀n. P n [];
     ⋀x xs. P 0 (x#xs);
     ⋀n x xs. P n xs ⟹ P (Suc n) (x#xs)⟧
    ⟹ P n x
```

Este esquema puede verse usando `thm coge.induct`.

Proposición 2.2.5 *Dada una lista cualquiera* `xs`*, siempre se verifica que*

`conc (coge n xs) (elimina n xs) = xs`.

- La demostración estructurada es

```
lemma "conc (coge n xs) (elimina n xs) = xs"
proof (induct rule: coge.induct)
  fix n
  show "conc (coge n []) (elimina n []) = []" by simp
next
  fix x xs
  show "conc (coge 0 (x#xs)) (elimina 0 (x#xs)) = x#xs" by simp
next
  fix n x xs
  assume HI: "conc (coge n xs) (elimina n xs) = xs"
  have "conc (coge (Suc n) (x#xs)) (elimina (Suc n) (x#xs)) =
        conc (x#(coge n xs)) (elimina n xs)" by simp
  also have "... = x#(conc (coge n xs) (elimina n xs))" by simp
  also have "... = x#xs" using HI by simp
  finally show
  "conc (coge (Suc n) (x#xs)) (elimina (Suc n) (x#xs)) = x#xs"
    by simp
qed
```

Comentarios sobre la demostración anterior:

- `(induct rule: coge.induct)` indica que el método de demostración es por el esquema de inducción correspondiente a la definición de la función `coge`.
- Se generan 3 subobjetivos:
  1. ⋀`n. conc (coge n []) (elimina n []) = []`
  2. ⋀`x xs. conc (coge 0 (x#xs)) (elimina 0 (x#xs)) = x#xs`
  3. ⋀`n x xs.conc (coge n xs) (elimina n xs) = xs` ⟹
     `conc (coge (Suc n) (x#xs)) (elimina (Suc n) (x#xs)) = x#xs`

• La demostración automática es

```
lemma "conc (coge n xs) (elimina n xs) = xs"
by (induct rule: coge.induct) auto
```

2.2.5. Razonamiento por distinción de casos

Distinción de casos booleanos

Ejemplo de demostración por distinción de casos booleanos:

Demostrar ¬ `A` ∨ `A`.

• La demostración estructurada es

```
lemma "¬A ∨ A"
proof cases
  assume "A"
  then show "¬A ∨ A" ..
next
  assume "¬A"
  then show "¬A ∨ A" ..
qed
```

Comentarios de la demostración anterior:

- `proof cases` indica que el método de demostración será por distinción de casos.
- Se generan 2 casos:
  1. `?P` ⟹ ¬ `A` ∨ `A`
  2. ¬`?P` ⟹ ¬ `A` ∨ `A`

  donde `?P` es una variable sobre las fórmulas.
- `(assume "A")` indica que se está usando `A` en lugar de la variable `?P`.

- `then` indica usando la fórmula anterior.
- `..` indica que se utiliza la regla lógica necesaria (las reglas lógicas se estudiarán en los siguientes temas).
- `next` indica el siguiente caso (se puede observar cómo ha sustituido `¬?P` por `¬A`.

• La demostración automática es

```
lemma "¬A ∨ A"
by auto
```

Ejemplo de demostración por distinción de casos booleanos con nombres:

Demostrar `¬ A ∨ A`.

• La demostración estructurada es

```
lemma "¬A ∨ A"
proof (cases "A")
  case True
  then show "¬A ∨ A" ..
next
  case False
  thus "¬A ∨ A" ..
qed
```

Comentarios sobre la demostración anterior:

- `(cases "A")` indica que la demostración se hará por casos según los distintos valores de `A`.
- Como `A` es una fórmula, sus posibles valores son verdadero o falso.
- `case True` indica que se está suponiendo que `A` es verdadera. Es equivalente a `assume A`.
- `case False` indica que se está suponiendo que `A` es falsa. Es equivalente a `assume ¬A`.
- En general,
  1. el método `(cases F)` es una abreviatura de la aplicación de la regla
     $$[\![F \Longrightarrow Q;\ \neg F \Longrightarrow Q]\!] \Longrightarrow Q$$
  2. La expresión `case True` es una abreviatura de `F`.
  3. La expresión `case False` es una abreviatura de `¬F`.
- Ventajas de `cases` con nombre:
  1. reduce la escritura de la fórmula y
  2. es independiente del orden de los casos

Distinción de casos sobre otros tipos de datos

Ejemplo de distinción de casos sobre listas:

Se define la función `esVacia` tal que `(esVacia xs)` se verifica si `xs` es la lista vacía.

```
fun esVacia :: "'a list ⇒ bool" where
  "esVacia []     = True"
| "esVacia (x#xs) = False"
```

Proposición 2.2.6 *Dada una lista cualquiera* `xs`*, siempre se verifica que*

`esVacia xs = esVacia (conc xs xs)`

• La demostración estructurada es

```
lemma "esVacia xs = esVacia (conc xs xs)"
proof (cases xs)
  assume "xs = []"
  then show "esVacia xs = esVacia (conc xs xs)" by simp
next
  fix y ys
  assume "xs = y#ys"
  then show "esVacia xs = esVacia (conc xs xs)" by simp
qed
```

Comentarios sobre la demostración anterior:

- `(cases xs)` es el método de demostración por casos según `xs`.
- Se generan dos subobjetivos correspondientes a los dos constructores de listas:
  1. `xs = [] ⟹ esVacia xs = esVacia (conc xs xs)`
  2. `⋀y ys. xs = y#ys ⟹ esVacia xs = esVacia (conc xs xs)`
- `then` indica "usando la propiedad anterior".

• La demostración estructurada simplificada es

```
lemma "esVacia xs = esVacia (conc xs xs)"
proof (cases xs)
  case Nil
  thus "esVacia xs = esVacia (conc xs xs)" by simp
next
  case Cons
  thus "esVacia xs = esVacia (conc xs xs)" by simp
qed
```

Comentarios sobre la demostración anterior:

- `case Nil` es una abreviatura de `assume xs = []`.
- `case Cons` es una abreviatura de `fix y ys assume xs = y#ys`.
- `thus` es una abreviatura de `then show`.

• La demostración automática es

```
lemma "esVacia xs = esVacia (conc xs xs)"
by (cases xs) auto
```

2.2.6. Heurística de generalización para la inducción

Heurística de generalización:
A veces es necesario cuantificar universalmente las variables libres o, equivalentemente, considerar las variables libres como variables arbitrarias. A continuación se muestra un ejemplo con la proposición 2.2.7.

Primero se define `inversa` de forma recursiva tal que `(inversa xs)` es la inversa de la lista `xs`.

```
fun inversa :: "'a list ⇒ 'a list" where
  "inversa [] = []"
| "inversa (x#xs) = (inversa xs) @ [x]"
```

Por otro lado, se construye la definición de *inversa* con acumuladores, es decir, se define la función `inversaAc` tal que `(inversaAc xs)` es la inversa de `xs` calculada usando acumuladores.

```
fun inversaAcAux :: "'a list ⇒ 'a list ⇒ 'a list" where
  "inversaAcAux [] ys     = ys"
| "inversaAcAux (x#xs) ys = inversaAcAux xs (x#ys)"

fun inversaAc :: "'a list ⇒ 'a list" where
  "inversaAc xs = inversaAcAux xs []"
```

Proposición 2.2.7 *Dadas dos listas cualesquiera,* `xs` *e* `ys`*, siempre verifican que*

`inversaAcAux xs ys = (inversa xs) @ ys`.

• La demostración estructurada es

```
lemma inversaAcAux_es_inversa:
  "inversaAcAux xs ys = (inversa xs) @ ys"
proof (induct xs arbitrary: ys)
  show "⋀ys. inversaAcAux [] ys = inversa [] @ ys" by simp
next
  fix a xs
```

```
  assume HI: "⋀ys. inversaAcAux xs ys = inversa xs@ys"
  show "⋀ys. inversaAcAux (a#xs) ys = inversa (a#xs)@ys"
  proof -
    fix ys
    have "inversaAcAux (a#xs) ys = inversaAcAux xs (a#ys)" by simp
    also have "... = inversa xs@(a#ys)" using HI by simp
    also have "... = inversa (a#xs)@ys" by simp
    finally show "inversaAcAux (a#xs) ys = inversa (a#xs)@ys" by simp
  qed
qed
```

Comentarios sobre la demostración anterior:

- `(induct xs arbitrary: ys)` es el método de demostración por inducción sobre `xs` usando `ys` como variable arbitraria. Se puede comprobar que si se fija `ys` desde el principio no es posible demostrar la proposición con inducción en `xs`.

- Se generan dos subobjetivos:

  1. ⋀`ys. inversaAcAux [] ys = inversa [] ys`
  2. ⋀`a xs ys. (`⋀`ys. inversaAcAux xs ys = inversa xs ys)` ⟹ `inversaAcAux (a # xs) ys = inversa (a # xs) @ ys`

- Dentro de una demostración se pueden incluir otras demostraciones.

- Para demostrar la propiedad universal ⋀`ys. P(ys)` se elige una lista arbitraria (con `fix ys`) y se demuestra `P(ys)`.

• La demostración automática es

```
lemma "inversaAcAux xs ys = (inversa xs)@ys"
by (induct xs arbitrary: ys) auto
```

2.2.7. Recursión mutua e inducción

A continuación se muestra un ejemplo de definición de tipos mediante recursión cruzada.

Se dice que un árbol de tipo `a` es una hoja o un nodo de tipo `a` junto con un bosque de tipo `a`. Por otro lado, un bosque de tipo `a` es el boque vacío o un bosque contruido añadiendo un árbol de tipo `a` a un bosque de tipo `a`.

```
datatype 'a arbol = Hoja | Nodo "'a" "'a bosque"
     and 'a bosque = Vacio | ConsB "'a arbol" "'a bosque"
```

Regla de inducción correspondiente a la recursión cruzada

La regla de inducción sobre árboles y bosques generada de forma automática por la definición es `arbol_bosque.induct`:

```
⟦P1 Hoja;
      ⋀x b. P2 b ⟹ P1 (Nodo x b);
      P2 Vacio;
      ⋀a b. ⟦P1 a; P2 b⟧ ⟹ P2 (ConsB a b)⟧
     ⟹ P1 a ∧ P2 b
```

Algunos ejemplos de definición por recursión cruzada son:

- `(aplana_arbol a)` es la lista obtenida aplanando el árbol `a`.
- `(aplana_bosque b)` es la lista obtenida aplanando el bosque `b`.
- `(map_arbol a h)` es el árbol obtenido aplicando la función `h` a todos los nodos del árbol `a`.
- `(map_bosque b h)` es el bosque obtenido aplicando la función `h` a todos los nodos del bosque `b`.

```
fun aplana_arbol :: "'a arbol ⇒ 'a list" and
    aplana_bosque :: "'a bosque ⇒ 'a list" where
  "aplana_arbol Hoja = []"
| "aplana_arbol (Nodo x b) = x#(aplana_bosque b)"
| "aplana_bosque Vacio = []"
| "aplana_bosque (ConsB a b) = (aplana_arbol a) @ (aplana_bosque b)"

fun map_arbol :: "('a ⇒ 'b) ⇒ 'a arbol ⇒ 'b arbol" and
    map_bosque :: "('a ⇒ 'b) ⇒ 'a bosque ⇒ 'b bosque" where
  "map_arbol  f Hoja        = Hoja"
| "map_arbol  f (Nodo x b)  = Nodo (f x) (map_bosque f b)"
| "map_bosque f Vacio       = Vacio"
| "map_bosque f (ConsB a b) = ConsB (map_arbol f a) (map_bosque f b)"
```

De las dos definiciones anteriores se puede demostrar por inducción cruzada que:

- `aplana_arbol (map_arbol f a) = map f (aplana_arbol a)`
- `aplana_bosque (map_bosque f b) = map f (aplana_bosque b)`

• La demostración detallada es

```
lemma "aplana_arbol  (map_arbol  f a) = map f (aplana_arbol a)
     ∧ aplana_bosque (map_bosque f b) = map f (aplana_bosque b)"
proof (induct_tac a and b)
  show "aplana_arbol (map_arbol f Hoja ) = map f (aplana_arbol Hoja)"
    by simp
```

```
next
  fix x b
  assume HI: "aplana_bosque (map_bosque f b) = map f (aplana_bosque b)"
  have "aplana_arbol (map_arbol f (Nodo x b)) =
        aplana_arbol (Nodo (f x) (map_bosque f b))" by simp
  also have "... = (f x)#(aplana_bosque (map_bosque f b))" by simp
  also have "... = (f x)#(map f (aplana_bosque b))" using HI by simp
  also have "... = map f (aplana_arbol (Nodo x b))" by simp
  finally show "aplana_arbol (map_arbol f (Nodo x b))
                = map f (aplana_arbol (Nodo x b))" .
next
  show "aplana_bosque (map_bosque f Vacio) =
   map f (aplana_bosque Vacio)" by simp
next
  fix a b
  assume HI1: "aplana_arbol (map_arbol f a) =
   map f (aplana_arbol a)"
     and HI2: "aplana_bosque (map_bosque f b) =
      map f (aplana_bosque b)"
  have "aplana_bosque (map_bosque f (ConsB a b)) =
        aplana_bosque (ConsB (map_arbol f a)
                             (map_bosque f b))" by simp
  also have "... =
  aplana_arbol(map_arbol f a)@aplana_bosque(map_bosque f b)"
    by simp
  also have "... = (map f (aplana_arbol a))@(map f (aplana_bosque b))"
    using HI1 HI2 by simp
  also have "... = map f (aplana_bosque (ConsB a b))" by simp
  finally show "aplana_bosque (map_bosque f (ConsB a b))
                = map f (aplana_bosque (ConsB a b))" by simp
qed
```

Comentarios sobre la demostración anterior:

- `(induct_tac a and b)` indica que el método de demostración es por inducción cruzada sobre `a` y `b`.
- Se generan 4 casos:
  1. `aplana_arbol (map_arbol arbol.Hoja h) =`
     `map h (aplana_arbol arbol.Hoja)`
  2. `⋀a bosque. aplana_bosque (map_bosque bosque h) =`
     `map h (aplana_bosque bosque)` $\Longrightarrow$
     `aplana_arbol (map_arbol (arbol.Nodo a bosque) h) =`
     `map h (aplana_arbol (arbol.Nodo a bosque))`
  3. `aplana_bosque (map_bosque Vacio h) =`
     `map h (aplana_bosque Vacio)`

4. ⟹arbol bosque.
   ⟦aplana_arbol (map_arbol arbol h) =
   map h (aplana_arbol arbol);
   aplana_bosque (map_bosque bosque h) =
   map h (aplana_bosque bosque)⟧
   ⟹ aplana_bosque (map_bosque (ConsB arbol bosque) h) =
   map h (aplana_bosque (ConsB arbol bosque))

• La demostración automática es

```
lemma "aplana_arbol  (map_arbol  f a) = map f (aplana_arbol a)
     ∧ aplana_bosque (map_bosque f b) = map f (aplana_bosque b)"
by (induct_tac a and b) auto
```

2.3. Definiciones inductivas en Isabelle/HOL

2.3.1. El conjunto de los números pares

El conjunto de los números pares se puede definir inductivamente como el menor conjunto que contiene al 0 y es cerrado por la operación +2.

El conjunto de los números pares también puede definirse como los naturales divisibles por 2.

Veremos cómo se escriben las dos definiciones en Isabelle/HOL y cómo se demuestra su equivalencia.

Definición inductiva del conjuntos de los pares

Mediante el comando `inductive_set` se declara la constante `par` como un conjunto de números naturales con unas determinadas propiedades deseadas.

Una definición inductiva está formada por reglas de introducción.

```
inductive_set par :: "nat set" where
  cero [intro!]: "0 ∈ par"
| paso [intro!]: "n ∈ par ⟹ (Suc (Suc n)) ∈ par"
```

Esta definición inductiva genera varios teoremas. Estos teoremas incluyen las reglas de introducción especificadas en la declaración, una regla de eliminación para el análisis por casos y una regla de inducción.

Reglas de introducción

- `par.cero: 0∈ par`
- `par.paso: n∈ par ⟹ Suc (Suc n) ∈ par`
- `par.simps: (a ∈ par) = (a = 0 ∨ (∃n. a = Suc (Suc n) ∧ n ∈ par))`

Regla de eliminación (reglas que pueden usarse como reglas de eliminación)

```
?a ∈ par ⟹ (?a = 0 ⟹ ?P) ⟹
(⋀n. ?a = Suc (Suc n) ⟹ n ∈ par ⟹ ?P) ⟹ ?P
```

Regla de inducción

```
par.induct: ⟦ x ∈ par;
                P 0;
                ⋀n. ⟦n ∈ par; P n⟧ ⟹ P (Suc (Suc n))⟧
              ⟹ P x
```

Uso de las reglas de introducción

Este primer lema afirma que los números de la forma $2k$, siendo $k \in \mathbb{N}$, son pares.

Proposición 2.3.1 *Los números de la forma $2k$ son pares.*

- La demostración estructurada es

```
lemma dobles_son_pares_2:
  "2*k ∈ par"
proof (induct k)
  show "2 * 0 ∈ par" by auto
next
  show "⋀k. 2 * k ∈ par ⟹ 2 * Suc k ∈ par" by auto
qed
```

- La demostración automática es

```
lemma dobles_son_pares [intro!]:
  "2*k ∈ par"
by (induct k) auto
```

Nuestro objetivo es demostrar la equivalencia de la definición anterior y la definición mediante divisibilidad. Uno de los sentidos de esta equivalencia es inmediato por el lema que se acaba de probar, ya que el comando `intro!` asegura que el lema se aplica automáticamente.

Proposición 2.3.2 *Si n es divisible por 2, entonces es par.*

```
thm dvd_def
```

```
lemma dvd_imp_par: "2 dvd n ⟹ n ∈ par"
by (auto simp add: dvd_def)
```

Regla de inducción

Entre las reglas generadas por la definión `par` está la regla de inducción siguiente:

```
par.induct: ⟦ x ∈ par;
                P 0;
                ⋀n. ⟦n ∈ par; P n⟧ ⟹ P (Suc (Suc n))⟧
              ⟹ P x
```

Una propiedad P se cumple para todos los números pares siempre que el 0 tenga la propiedad y sea cerrada para la operación $Suc\ (Suc\ \cdot)$. De esta forma, P es cerrada bajo las reglas de introducción de `par`, que es el menor conjunto cerrado bajo estas reglas. A este razonamiento inductivo se le llama *regla de inducción.*

La inducción es una forma habitual de demostrar que todos los elementos de un conjunto satisfacen una determinada propiedad.
A continuación, se prueba por inducción la otra implicación de la equivalencia anterior, es decir, que todos los números pares son múltiplos de 2.

Proposición 2.3.3 *Los números pares son divisibles por* 2*.*

- La demostración detallada es:

```
lemma par_imp_dvd:
  "n ∈ par ⟹ 2 dvd n"
proof (induction rule: par.induct)
  show "2 dvd (0::nat)" by simp (*(simp add: dvd_def)*)
next
  fix n::nat
  assume H1: "n ∈ par" and
         H2: "2 dvd n"
  have "∃k. n = 2*k" using H2 by (simp add: dvd_def)
  then obtain k where "n = 2*k" ..
  hence "Suc (Suc n) = 2*(k+1)" by arith
  hence "∃k. Suc (Suc n) = 2*k" ..
  thus "2 dvd Suc (Suc n)" by (simp add: dvd_def)
qed
```

El método `arith` trata de probar el primer subobjetivo siempre y cuando sea una fórmula aritmética lineal. Dichas fórmulas pueden incluir los conectores lógicos habituales ($\neg$,$\wedge$,$\vee$,$\longrightarrow$,$\exists$,$\forall$), las relaciones, $=$, $\leq$, y $<$, y las operaciones $+$, $-$, `min` y `max`.

- La demostración con `arith` es:

```
lemma par_imp_dvd_2:
  "n ∈ par ⟹ 2 dvd n"
proof (induction rule: par.induct)
  show "2 dvd (0::nat)" by simp (* (simp_all add: dvd_def)*)
next
```

```
  fix n::nat
  assume H1: "n ∈ par" and
         H2: "2 dvd n"
  thus "2 dvd Suc (Suc n)" by (auto simp add: dvd_def, arith)
qed
```

- La demostración automática es:

```
lemma par_imp_dvd_3:
  "n ∈ par ⟹ 2 dvd n"
by (induction rule:par.induct) (auto simp add: dvd_def, arith)
```

Si se combinan las dos proposiciones previas, se demuestra la equivalencia deseada:

Proposición 2.3.4 *Un número n es par $\Leftrightarrow$ es divisible por 2.*

```
theorem par_iff_dvd: "(n ∈ par) = (2 dvd n)"
by (auto simp add: dvd_imp_par par_imp_dvd)
```

o bien

```
by (blast intro: dvd_imp_par par_imp_dvd)
```

El método `blast` es la herramienta principal que tiene Isabelle para probar teoremas de forma automática, además de ser el más rápido. Este método es aún más fuerte que `auto`. También es efectivo para teoría de conjuntos.

Mientras que el método `blast` podría simplemente fallar, el método `clarify` nos muestra un subobjetivo que nos puede ayudar a entender porqué no se puede continuar con la prueba.

Uso de la regla de eliminación

Antes de aplicar inducción con frecuencia es conveniente generalizar la fórmula a probar.

Vamos a ilustrar el principio anterior en el caso de los conjuntos inductivamente definidos, con el siguiente ejemplo:

Proposición 2.3.5 *Si $n + 2$ es par, entonces n también lo es.*

El siguiente intento falla:

```
lemma "Suc (Suc n) ∈ par ⟹ n ∈ par"
apply (erule par.induct)
oops
```

En el intento anterior, los subobjetivos generados son:

1. `n ∈ par`,

2. ⋀na. ⟦na ∈ par; n ∈ par⟧ ⟹ n ∈ par,

que no se pueden demostrar. Se ha perdido la información sobre *Suc* (*Suc n*). En ese caso, se reformula el lema a demostrar. En el ejemplo que estamos tratando, la reformulación es:

Proposición 2.3.6 *Si n es par, entonces n − 2 también lo es.*

• La demostración estructurada es:

```
lemma par_imp_par_menos_2:
  "n ∈  par ⟹ n - 2 ∈ par"
proof (induction rule:par.induct)
  show "0 - 2 ∈ par" by auto
next
  show "⋀n. ⟦n ∈ par; n - 2 ∈ par⟧ ⟹ Suc (Suc n) - 2 ∈ par" by simp
qed
```

• La demostración automática es:

```
lemma par_imp_par_menos_2:
  "n ∈ par ⟹ n - 2 ∈ par"
by (induction rule:par.induct) auto
```

Utilizando este último lema se puede demostrar el lema original 2.3.5.

• La demostración estructurada es:

```
lemma
  assumes "Suc (Suc n) ∈ par"
  shows   "n ∈ par"
proof -
  have "Suc (Suc n) - 2 ∈ par" using assms by (rule par_imp_par_menos_2)
  thus "n ∈ par" by simp
qed
```

• La demostración aplicativa, usando el lema 2.3.6 como regla de destrucción para razonar hacia delante, es:

```
lemma
  "Suc (Suc n) ∈ par ⟹ n ∈ par"
apply (drule par_imp_par_menos_2) (* o bien con frule *)
apply (simp)
done
```

• La demostración automática es:

```
lemma Suc_Suc_par_imp_par:
  "Suc (Suc n) ∈ par ⟹ n ∈ par"
by (drule par_imp_par_menos_2, simp)
```

A partir de los dos lemas 2.3.5 y 2.3.6, se puede demostrar la equivalencia siguiente.

Proposición 2.3.7 *Un número natural n es par* $\Leftrightarrow$ $n+2$ *es par.*

```
lemma [iff]: "((Suc (Suc n)) ∈ par) = (n ∈ par)"
by (auto simp add: Suc_Suc_par_imp_par)
```

o bien

`by (auto dest: Suc_Suc_par_imp_par)` o con `blast`.

Comentario sobre la demostración anterior:

Se usa el atributo `iff` porque sirve como regla de simplificación.

Definiciones mutuamente inductivas

Existen conjuntos definidos mediante inducción mutua. Por ejemplo, la definición cruzada de los conjuntos inductivos de los pares y de los impares es:

```
inductive_set
  Pares    :: "nat set" and
  Impares  :: "nat set"
where
  ceroP:    "0 ∈ Pares"
| ParesI:   "n ∈ Impares ⟹ Suc n ∈ Pares"
| ImparesI: "n ∈ Pares   ⟹ Suc n ∈ Impares"
```

El esquema de inducción generado por la definición anterior es `Pares_Impares.induct`:

```
⟦P1 0;
     ⋀n. ⟦n ∈ Impares; P2 n⟧ ⟹ P1 (Suc n);
     ⋀n. ⟦n ∈ Pares;   P1 n⟧ ⟹ P2 (Suc n)⟧
    ⟹ (x1 ∈ Pares ⟶ P1 x1) ∧ (x2 ∈ Impares ⟶ P2 x2)
```

Ejemplo de demostración usando el esquema anterior.

```
lemma "(m ∈ Pares ⟶ 2 dvd m) ∧ (n ∈ Impares ⟶ 2 dvd (Suc n))"
proof (induction rule:Pares_Impares.induct)
  show "2 dvd (0::nat)" by simp
next
  fix n :: "nat"
  assume H1: "n ∈ Impares" and
         H2: "2 dvd Suc n"
  show "2 dvd Suc n" using H2 by simp
next
  fix n :: "nat"
```

```
  assume H1: "n ∈ Pares" and
         H2: "2 dvd n"
  have "∃k. n = 2*k" using H2 by (simp add: dvd_def)
  then obtain k where "n = 2*k" ..
  hence "Suc (Suc n) = 2*(k+1)" by simp
  hence "∃k. Suc (Suc n) = 2*k" ..
  thus "2 dvd Suc (Suc n)" by (simp add: dvd_def)
qed
```

Definición inductiva de predicados

En lugar de definir un conjunto de números con unas determinadas propiedades también se puede construir un predicado sobre los naturales.

Definición inductiva del predicado `es_par` tal que `(es_par n)` se verifica si `n` es par.

```
inductive es_par :: "nat ⇒ bool" where
  "es_par 0"
| "es_par n ⟹ es_par(Suc(Suc n))"
```

Heurística para elegir entre definir conjuntos o predicados:

- si se va a combinar con operaciones conjuntistas, definir conjunto;
- en caso contrario, definir predicado.

2.3.2. Clausura reflexiva transitiva

Isabelle admite la definición de funciones de orden superior, es decir, cuyos argumentos sean otras funciones o predicados.

Definición 2.3.1 *La clausura reflexiva y transitiva de una relación r es la menor relación reflexiva y transitiva que contiene a r. Se representa por r^*.*

Esta relación r^* se puede definir inductivamente, como conjunto:

- $(x, x) \in r^*$
- Si $(x, y) \in r$ e $(y, z) \in r^*$, entonces $(x, z) \in r^*$.

La definición inductiva, como conjunto, se puede expresar en Isabelle/HOL como sigue:

```
inductive_set
  crt :: "('a × 'a) set ⇒ ('a × 'a) set"   ("_*" [1000] 999)
  for r :: "('a × 'a) set"
where
  crt_refl [iff]: "(x,x) ∈ r*"
| crt_paso:       "⟦ (x,y) ∈ r; (y,z) ∈ r* ⟧ ⟹ (x,z) ∈ r*"
```

Comentarios sobre la definición anterior:

- La sintaxis concreta permite escribir `r*` en lugar de `crt r`.
- La definición consta de dos reglas.
- A la regla reflexiva se le añade el atributo `iff` para aumentar la automatización.
- A la regla del paso no se le añade ningún atributo, porque `r*` ocurre en la izquierda.
- En el resto de esta sección se demuestra que esta definición coincide con la menor relación reflexiva y transitiva que contiene a `r`.

Teoremas que se han generado sólamente con la definición, cuando se ha construido:

```
thm crt.induct
thm crt_paso
thm crt_refl
thm crt.intros
```

`thm crt.induct` hay que estudiarlo en detalle porque es el que luego se utiliza para demostraciones.

Proposición 2.3.8 *La relación r^* es reflexiva.*

```
lemma "(x,x) ∈ r*"
by simp  (* o con "by (rule crt_refl)"*)
```

Proposición 2.3.9 *La relación r^* contiene a r.*

```
lemma [intro]: "(x,y) ∈ r ⟹ (x,y) ∈ r*"
by (blast intro: crt_paso)  (*o bien con "auto"*)
```

Comentarios sobre la demostración anterior:

- La ventaja del lema es que se puede declarar como regla de introducción, porque `r*` ocurre sólo en la derecha.
- Con la declaración, algunas demostraciones que usan `crt_paso` se hacen de manera automática.

El esquema de inducción de la clausura reflexiva transitiva es `crt.induct`:

```
⟦(x1, x2) ∈ r*;
    ⋀x. P x x;
    ⋀x y z. ⟦(x,y) ∈ r; (y,z) ∈ r*; P y z⟧ ⟹ P x z⟧
   ⟹ P x1 x2
```

Proposición 2.3.10 *La relación r^* es transitiva.*

• La demostración automática es:

```
lemma crt_trans_auto: "⟦ (x,y) ∈ r*; (y,z) ∈ r* ⟧ ⟹ (x,z) ∈ r*"
by (induction rule:crt.induct)
   (auto simp add:crt_paso)
```

• La demostración aplicativa es:

```
lemma crt_trans_apply:
"⟦ (x,y) ∈ r*; (y,z) ∈ r* ⟧ ⟹ (x,z) ∈ r*"
apply (induction rule:crt.induct)
apply (auto simp add:crt_paso)
done
```

La relación r^* está contenida en cualquier relación reflexiva y transitiva que contenga a r.

Mediante `crt2 r` se define la menor relación reflexiva y transitiva que contiene a `r`.

```
inductive_set
  crt2 :: "('a × 'a)set ⇒ ('a × 'a)set"
  for r :: "('a × 'a)set"
where
  "(x,y) ∈ r ⟹ (x,y) ∈ crt2 r" (* contiene a r *)
| "(x,x) ∈ crt2 r"                (* reflexiva *)
| "⟦ (x,y) ∈ crt2 r; (y,z) ∈ crt2 r ⟧ ⟹ (x,z) ∈ crt2 r"
(* transitiva *)
```

A continuación probamos que `r*` coincide con `crt2 r`.

Proposición 2.3.11 *La relación* `crt2 r` *está contenida en* `r*`.

```
lemma "(x,y) ∈ crt2 r ⟹ (x,y) ∈ r*"
proof (induction rule: crt2.induct)
  fix x y
  assume "(x,y) ∈ r"
  thus "(x,y) ∈ r*" by blast
next
  fix x
  show "(x,x) ∈ r*" by blast
next
  fix x y z
  assume H1: "(x,y) ∈ crt2 r" and
         H2: "(x,y) ∈ r*"     and
         H3: "(y,z) ∈ crt2 r" and
         H4: "(y,z) ∈ r*"
  show "(x,z) ∈ r*" using H2 H4 by (rule crt_trans)
qed
```

Proposición 2.3.12 *La relación* `r*` *está contenida en* `crt2 r`.

```
lemma "(x,y) ∈ r* ⟹ (x,y) ∈ crt2 r"
proof (induction rule:crt.induct)
  fix x
  show "(x,x) ∈ crt2 r" by (rule crt2.intros(2))
next
  fix x y z
  assume H1: "(x,y) ∈ r" and
         H2: "(y,z) ∈ r*" and
         H3: "(y,z) ∈ crt2 r"
  have "(x,y) ∈ crt2 r" using H1 by (rule crt2.intros(1))
  thus "(x,z) ∈ crt2 r" using H3 by (rule crt2.intros(3))
qed
```

Capítulo 3

Lógica de Hoare

El origen del concepto de verificación se remonta a Alan Turing, aunque la primera proposición de un método sistemático de verificación de programas se debe a Robert Floyd. En 1967, Floyd publicó un artículo basado en la idea de expresar las sentencias de los programas como asertos lógicos, una vez definida formalmente la semántica del lenguaje de programación. Dos años después, el científico Charles Antony Richard Hoare utilizó tales trabajos para ampliar la semántica e introducir, entre otros, los conceptos de precondición, postcondición e invariante. Así fue como surgió la Lógica de Hoare.

A lo largo del capítulo se estudia esta lógica donde, en primer lugar, se explican algunas especificaciones acerca del lenguaje de programación y de la notación que se utilizará, como se describe en [5] y en [6].

Se explica qué es una terna correcta parcialmente, o correcta totalmente, y se muestra con ejemplos. Además, se diferencia entre el concepto de terna verdadera y terna demostrable. Al final se prueba que ambos conceptos son equivalentes. Esto es, *Teorema de Adecuación* y *Teorema de Completitud de la Lógica de Hoare*.

3.1. Especificaciones

3.1.1. Un pequeño lenguaje de programación

El pequeño lenguaje de programación que estudiaremos aquí puede considerarse como el típico núcleo de la mayor parte de los lenguajes de programación imperativos. Módulo un conjunto de variaciones sintácticas simples, dicho lenguaje es, de hecho, un subconjunto de Pascal, C, C++ o Java. Nuestro lenguaje consistirá en una serie de "comandos" simples: asignaciones a variables numéricas enteras, condicionales, bucles tipo `WHILE` y composición secuencial de comandos. Las palabras "programa" y "comando" pueden considerarse sinónimas, aunque en esta sección la primera se utilizará para referirnos a comandos que representan algoritmos completos. En estas condiciones, diremos que un programa se construye a partir de una sucesión finita de comandos.

A continuación se describen la sintaxis y semántica empleadas en este lenguaje. Nuestro lenguaje consta de tres dominios sintácticos: expresiones numéricas enteras (expresadas formalmente mediante el conjunto de los términos de un lenguaje de primer orden) , expresiones booleanas (expresadas formalmente mediante el conjunto de las fórmulas de un lenguaje de primer orden con igualdad) y comandos.

Se utilizan las siguientes convenciones, como aparece en [5]:

1. Los símbolos $V, V_1, V_2, \ldots, V_n, \ldots$ se refieren a las *variables* utilizadas en el programa. Consideraremos siempre que dichas variables toman valores en el conjunto de los números enteros $\mathbb{Z}$.

2. Los símbolos $E, E_1, E_2, \ldots, E_n, \ldots$ designan *expresiones* (o *términos*) numéricas enteras. Dichas expresiones enteras se construyen de la manera habitual a partir de las variables, constantes numéricas 0,1,2, ..., -1,-2, ... y operaciones básicas como la suma (+) o el producto (*). Formalmente, el conjunto de los términos o expresiones se define recursivamente como sigue:

   a) Toda variable es un término.

   b) Toda constante numérica entera es un término.

   c) Si f es un símbolo de función n-ario que representa una operación aritmética, $0 < n$, y $t_1, \ldots, t_n$ son términos, entonces $f(t_1, \ldots, t_n)$ es un término.

   El conjunto de las expresiones coincide, por tanto, con el conjunto de los términos de un lenguaje de primer orden que contiene, al menos, las operaciones aritméticas básicas suma (+) y producto (*). En cuanto a la semántica, cada término tomará un valor numérico entero que se determinará tras asignar a las variables que intervengan un determinado valor y evaluar las operaciones aritméticas indicadas.
   Algunos términos denotan un valor fijo, como `4+5`; mientras que otros términos contienen variables y su valor puede variar. Por ejemplo, `X`, `X+1`, `R+(Y*Q)`.

3. Los símbolos $S, S_1, S_2, \ldots, S_n, \ldots$ representan *condiciones* booleanas. Se corresponden con las *fórmulas* de un lenguaje de primer orden con igualdad. Formalmente, el conjunto de las fórmulas se define recursivamente como sigue:

   a) Si p es un símbolo de predicado n-ario y $t_1, \ldots, t_n$ son términos, entonces $p(t_1, \ldots, t_n)$ es una fórmula, que se dirá *fórmula atómica.*

   b) Si P es una fórmula, entonces $\neg P$ es una fórmula.

   c) Si P, Q son fórmulas, entonces $P \vee Q$, $P \wedge Q$, $P \Rightarrow Q$, $P \Leftrightarrow Q$ son fórmulas.

   d) Si P es una fórmula y X es una variable, entonces $\exists X\ P$, $\forall X\ P$ son fórmulas.

   Es decir, las condiciones se construyen a partir de las fórmulas atómicas y las conectivas y los cuantificadores del lenguaje; y devuelven siempre un valor

booleano (verdadero o falso). Denotaremos por `T` a la condición atómica verdadera por construcción, y por `F` a la fórmula falsa. Otros ejemplos de fórmulas atómicas son `X=1`, `R<Y`. La interpretación de dichas fórmulas atómicas es la habitual en el dominio de los números enteros $\mathbb{Z}$. Por ejemplo, la primera fórmula es verdadera siempre que el valor asignado a la variable `X` sea `1`. Otras condiciones atómicas se construyen a partir de *predicados*. Por ejemplo, `ODD(X)`, `PRIME(3)`, `X=1`, $\texttt{(X+1)}^2 \geq \texttt{X}^2$, siendo los dos primeros predicados unarios cuya intarpretación es la usual, y los dos últimos *predicados infijos*.
Por otro lado, podemos considerar *condiciones compuestas*, formadas a partir de condiciones atómicas y de los operadores lógicos $\neg$ (negación), $\wedge$ (conjunción), $\vee$ (disyunción), $\Rightarrow$ (implicación) y $\Leftrightarrow$ (doble implicación). Estos operadores lógicos se comportan de la siguiente manera:

- $\neg P$: Es una condición verdadera si P es falsa, y es falsa si P es verdadera.
- $P \wedge Q$: Es verdadera si y sólo si P y Q son condiciones verdaderas.
- $P \vee Q$: Es verdadera si y sólo si alguna de las dos condiciones es verdadera.
- $P \Rightarrow Q$: Es verdadera siempre que Q sea verdadera, sin importar el valor de P, y en el caso en que ambas condiciones sean falsas.
- $P \Leftrightarrow Q$: Es verdadera si y sólo si ambas son verdaderas o ambas son falsas. De hecho, esta expresión es equivalente a $(P \Rightarrow Q) \wedge (Q \Rightarrow P)$.

Por ejemplo, la condición `ODD(X)` $\vee$ `EVEN(X)` es verdadera si el valor numérico entero asignado a la variable `X` es impar o par (por tanto, será siempre verdadera en $\mathbb{Z}$ independientemente del valor asignado a la variable `X`).

Por último, podemos construir condiciones más complejas usando cuantificadores de primer orden: $\exists X$, $\forall X$. Dichos cuantificadores se interpretarán como: "existe un número entero x tal que ..." o "para todo número entero x se tiene que ...", respectivamente.

Los paréntesis permiten establecer un orden de prioridad a la hora de evaluar una fórmula y así evitar confusiones. Con el fin de reducir su uso, se tienen los siguientes niveles de precedencia: $\neg$ se evalúa antes que $\wedge$ y $\vee$, los cuáles a su vez preceden a $\Rightarrow$ y $\Leftrightarrow$.

4. Los símbolos $C, C_1, C_2, \ldots, C_n, \ldots$ representan los comandos en el programa.

A continuación describiremos cuáles son los comandos que usaremos. A la par, describiremos también de manera informal e intuitiva la semántica asociada de manera natural a dichos comandos. Para ello, introducimos el concepto fundamental de *estado*. Nos referiremos con *estado* a una configuración única que contiene el conjunto de valores asignados a las variables de un programa en un momento dado. Formalmente, un estado para un cierto programa C es una aplicación del conjunto de las variables que aparecen en C en el conjunto de los números enteros. Así, el *estado inicial/final* está formado por el conjunto de todos los valores iniciales/finales

de las variables del programa. El *estado actual* es el último estado que se ha ejecutado. Representaremos los estados como un conjunto de igualdades, que expresan los valores asignados de algunas de las variables que intervienen en el programa, entre llaves. Por ejemplo, `{X=1,Y=3,Z=7,R=r}` es el estado en el que `X`, `Y`, `Z` toman los valores `1`, `3`, `7`, respectivamente. A la variable `R` se le ha asignado un valor fijo pero arbitrario r. El resto de variables tendrán un valor no especificado.

Vamos a describir los comandos del lenguaje de programación.

Asignación

Sintaxis: V`:=`E

Semántica: Se evalúa el valor de la expresión E y se le asigna a la variable V.

Ejemplo: `X:=X+1`. El valor de la variable `X` se incrementa en uno, dando lugar a un nuevo estado en la ejecución del programa.

Comando SKIP

Sintaxis: `SKIP`

Semántica: Transforma el estado actual en sí mismo, las variables del programa no cambian su valor después de la ejecución de esta instrucción.

Es un caso particular del anterior. De hecho, una instrucción `SKIP` puede simularse mediante una asignación del tipo V`:=`V.

Composición secuencial

Sintaxis: C_1`;`$\ldots$`;`C_n

Semántica: Se ejecutan en el orden dado los comandos descritos. Cada comando C_i se corresponde con alguno de los que se describen en la sección.

Ejemplo: `R:=X; X:=Y; Y:=R`. Computación a partir del estado inicial `{X=1,Y=2}`.

Paso 0. `{X=1,Y=2,R=r}` [Inicialización]

Paso 1. `{X=1,Y=2,R=1}` [Estado tras ejecutar el comando `R:=X`]

Paso 2. `{X=2,Y=2,R=1}` [Estado tras ejecutar el comando `X:=Y`]

Paso 3. `{X=2,Y=1,R=1}` [Estado tras ejecutar el comando `Y:=R`]

Este programa intercambia los valores de las variables `X` e `Y` utilizando la variable temporal `R`.

Condicional

Sintaxis: `IF` S `THEN` C_1 `ELSE` C_2

Semántica: Si la condición S es cierta en el estado actual, se ejecuta C_1; en caso contrario, se ejecuta el comando C_2.

Ejemplo: `IF X<Y THEN MAX:=Y ELSE MAX:=X`. Computación a partir del estado inicial `{X=2,Y=3}`.

Paso 0. `{X=2,Y=3,MAX=r}`

Paso 1. `{X=2,Y=3,MAX=3}` [Se ha ejecutado la asignación `MAX:=Y`]

El programa anterior calcula en la variable `MAX` el máximo entre `X` e `Y`.

Comando `WHILE`

Sintaxis: `WHILE` S `DO` C

Semántica: Si la condición S es cierta en el estado actual, se ejecuta C produciendo un nuevo estado actual y se vuelve a ejecutar el comando `WHILE` para dicho nuevo estado. Cuando S sea falsa, no se hace nada y se pasa, si existe, a la ejecución de la siguiente instrucción del programa. Es decir, C se ejecuta repetidamente mientras que la condición S se mantenga cierta. Si en algún momento la condición S se transforma en falsa tras i repeticiones del comando C, entonces el bucle termina y se pasa a la instrucción siguiente. Si la condición S siempre permanece verdadera, la ejecución del comando C se repite indefinidamente y el bucle no para entrando en una computación infinita.

Ejemplo: `WHILE X≠0 DO X:=X-2`. Computación a partir del estado inicial `{X=8}`.

Paso 0. `{X=8}`

Paso 1. `{X=6}`

Paso 2. `{X=4}`

Paso 3. `{X=2}`

Paso 4. `{X=0}`

Mientras que la variable `X` es distinta de cero, su valor va decreciendo dos unidades repetidamente. Este bucle termina si el valor inicial de `X` era un número par no negativo. En otro caso, la ejecución de esta instrucción entra en una computación infinita y, por tanto, no termina.

A continuación se muestran ejemplos de programas completos. Dichos ejemplos ponen de manifiesto la potencia del lenguaje de programación descrito a pesar de su aparente simplicidad. De hecho, puede demostrarse que el lenguaje WHILE aquí descrito constituye un modelo de computación universal, tan potente como el modelo de las máquinas de Turing o cualquier otro modelo de computación clásico.

Ejemplo 1:

$$C_1 = \begin{cases} \texttt{Y:=1;} \\ \texttt{R:=0;} \\ \texttt{WHILE R}\neq\texttt{X DO (R:=R+1; Y:=Y*R)} \end{cases}$$

Computación del programa C_1 a partir del estado inicial `{X=3}`.

Paso 0. `{X=3,Y=y,R=r}`

Paso 1. `{X=3,Y=1,R=r}`

Paso 2. `{X=3,Y=1,R=0}`

Paso 3. `{X=3,Y=1,R=1}`

Paso 4. `{X=3,Y=1,R=1}`

Paso 5. `{X=3,Y=1,R=2}`

Paso 6. `{X=3,Y=2,R=2}`

Paso 7. `{X=3,Y=2,R=3}`

Paso 8. `{X=3,Y=6,R=3}`

El programa anterior calcula el factorial de `X`, en caso de que sea un entero no negativo, en la variable de salida `Y`. Si el valor inicial de `X` es negativo, el programa anterior no para.

Ejemplo 2:

$$C_2 = \begin{cases} \texttt{R:=X;} \\ \texttt{Q:=0;} \\ \texttt{WHILE Y} \leq \texttt{R DO (R:=R-Y; Q:=Q+1)} \end{cases}$$

Computación del programa C_2 a partir del estado inicial `{X=7,Y=2}`.

Paso 0. `{X=7,Y=2,R=r,Q=q}`

Paso 1. `{X=7,Y=2,R=7,Q=q}`

Paso 2. {X=7,Y=2,R=7,Q=0}

Paso 3. {X=7,Y=2,R=5,Q=0}

Paso 4. {X=7,Y=2,R=5,Q=1}

Paso 5. {X=7,Y=2,R=3,Q=1}

Paso 6. {X=7,Y=2,R=3,Q=2}

Paso 7. {X=7,Y=2,R=1,Q=2}

Paso 8. {X=7,Y=2,R=1,Q=3}

Este programa calcula la división euclídea entre X e Y, números enteros positivos, siendo Q y R el cociente y el resto, respectivamente.

Ejemplo 3:

$$C_3 = \begin{cases} \texttt{Y:=X;} \\ \texttt{Z:=1;} \\ \texttt{R:=1;} \\ \texttt{WHILE 1<Y DO (R:=R+Z; Z:=R-Z; Y:=Y-1)} \end{cases}$$

Computación de C_3 a partir del estado inicial {X=3}.

Paso 0. {X=3,Y=y,Z=z,R=r}

Paso 1. {X=3,Y=3,Z=z,R=r}

Paso 2. {X=3,Y=3,Z=1,R=r}

Paso 3. {X=3,Y=3,Z=1,R=1}

Paso 4. {X=3,Y=3,Z=1,R=2}

Paso 5. {X=3,Y=3,Z=1,R=2}

Paso 6. {X=3,Y=2,Z=1,R=2}

Paso 7. {X=3,Y=2,Z=1,R=3}

Paso 8. {X=3,Y=2,Z=2,R=3}

Paso 9. {X=3,Y=1,Z=2,R=3}

Este programa calcula el valor del número que está en la posición `X`-ésima en la *sucesión de Fibonacci*, y lo devuelve en la variable `Z`.

3.1.2. Notación de Hoare

C.A.R. Hoare introdujo la siguiente notación para describir formalmente el comportamiento esperado de un programa

$$\{P\}\ C\ \{Q\}$$

donde:

- P es una condición que se denomina *precondición*.
- Q es una condición que se denomina *postcondición*.
- C es un *programa imperativo* cuyo lenguaje se ha descrito anteriormente.

Diremos que la *terna de Hoare* $\{P\}\ C\ \{Q\}$ es verdadera, y lo denotaremos por $\models \{P\}\ C\ \{Q\}$, cuando se cumple que:

- si ejecutamos el programa C a partir de un estado inicial que cumple la precondición P,
- y, además, el programa C para dicho estado inicial termina,
- entonces, el estado final tras la ejecución de C satisface la postcondición Q.

La expresión $\{P\}\ C\ \{Q\}$ corresponde a la *corrección parcial* del programa ya que si el programa para, ha de cumplirse la postcondición Q; si no para, no hay nada que comprobar ni refutar.

La *corrección total* es otro tipo de corrección más fuerte que la parcial. Se denota por

$$[P]\ C\ [Q]$$

Diremos que dicha terna es verdadera, y lo escribiremos como $\models [P]\ C\ [Q]$, si se cumple que:

- si ejecutamos el programa C a partir de un estado inicial que cumple la precondición P, entonces
- el programa C termina para dicho estado inicial y,
- además, el estado final tras la ejecución de C satisface la postcondición Q.

La relación que existe entre ambos tipos de corrección se puede expresar informalmente por la ecuación

$$\boxed{\text{Corrección total} = \text{Corrección parcial} + \text{Terminación}}$$

Para demostrar la corrección total de un programa se suele probar de forma separada, por una parte, la corrección parcial y, por otra, la terminación. La corrección total es lo que nos interesa en definitiva. La terminación es, normalmente, fácil de establecer, aunque existen algunos casos en los que no, como *la conjetura de Collatz*.

3.1.3. Algunos ejemplos

A continuación se muestran algunos ejemplos de ternas verdaderas y falsas, con respecto a la corrección parcial o total.

- `{`P`}` C `{T}`

Es verdadera para cualquier precondición P y cualquier programa C, ya que la postcondición `{T}` es, por definición, siempre verdadera.

- `{T}` C `{T}`

Esta terna es verdadera trivialmente, es un caso particular del ejemplo anterior tomando P=`T`.

- `[T]` C `[T]`

Esta terna es verdadera siempre que la computación de C termine a partir de cualquier estado inicial.

- `{X=1} X:=X+1 {X=2}`.

Esta terna es verdadera, ya que si se toma el estado inicial `{X=1}` que cumple la precondición y se ejecuta el programa, se obtiene el estado final `{X=2}`, como indica la postcondición.

- `{X=x,Y=y} X:=Y; Y:=X {X=y,Y=x}`.

Paso 0. `{X=x,Y=y}`

Paso 1. `{X=y,Y=y}`

Paso 2. `{X=y,Y=y}`

Esta terna es falsa (a menos que $x = y$), pues las variables `X` e `Y` no intercambian sus valores, como se muestra en la computación anterior.

- `{X=x,Y=y} R:=X; X:=Y; Y:=R {X=y,Y=x}`.

Paso 0. `{X=x,Y=y,R:=r}`

Paso 1. `{X=x,Y=y,R:=x}`

Paso 2. `{X=y,Y=y,R:=x}`

Paso 3. `{X=y,Y=x,R:=x}`

Esta terna de Hoare es verdadera, las variables `X` e `Y` intercambian sus valores.

- `{X=1} WHILE X≠0 DO X:=X {1=2}`

Esta terna es verdadera a pesar de que la postcondición es contradictoria, pues se trata de corrección *parcial* y el programa sobre `{X=1}` no para.

- `[X=1] WHILE X≠0 DO X:=X [1=2]`

Esta terna de Hoare falsa, porque el programa sobre `{X=1}` no para y ahora estamos considerando corrección *total* en lugar de parcial.

- `{T} WHILE X≠0 DO X:=X {F}`

Esta terna de Hoare es falsa, porque el programa termina para el estado inicial `{X=0}` que satisface trivialmente la precondición y la postcondición es, por definición, siempre falsa.

- Sea la siguiente terna de Hoare

`{T}`

$$C_4 = \left\{ \begin{array}{l} \texttt{R:=X;} \\ \texttt{Q:=0;} \\ \texttt{WHILE Y} \leq \texttt{R DO (R=R-Y; Q=Q+1)} \end{array} \right.$$

`{R<Y` $\wedge$ `X=R+(Y*Q)}`.

Dicha terna será verdadera si, para cualesquiera valores iniciales de `X` e `Y`, si la ejecución del programa C_4 termina, entonces `Q` guarda el cociente de dividir `X` entre `Y` y `R` guarda el resto.

• Sea la terna de Hoare `[0≤X ∧ 0<Y]` C_4 `[R<Y ∧ X=R+(Y*Q)]`, donde C_4 es el programa del ejemplo anterior.

Puesto que ahora estamos usando correción total, esta terna será verdadera si, para cualesquiera valores iniciales $X \geq 0$ e $Y > 0$, podemos asegurar que el programa C_4 para y, además, en el estado final se cumple que la variable `Q` guarda el cociente de dividir `X` entre `Y` y `R` guarda el resto.

• Sea la terna de Hoare

`{T}`

$$C_5 = \begin{cases} \texttt{Y:=1;} \\ \texttt{R:=0;} \\ \texttt{WHILE R}\neq\texttt{X DO (R:=R+1; Y:=Y*R)} \end{cases}$$

`{Y=fact(X)}`

La terna anterior es verdadera, ya que en caso de que C_5 termine se satisface la postcondición. Ahora bien, si en el estado inicial la variable `X` contiene un valor entero negativo, el programa anterior no para. Por tanto, la terna `[T]` C_5 `[Y=fact(X)]` es, en cambio, falsa.

3.2. Lógica de Hoare

En la sección anterior se introdujeron tres tipos de expresiones que podían ser verdaderas o falsas:

1. Corrección parcial de una terna de Hoare.
2. Corrección total de una terna de Hoare.
3. Declaraciones matemáticas, esto es, fórmulas de un lenguaje de primer orden.

Es bien conocido que las declaraciones matemáticas pueden demostrarse mediante *axiomas* y *reglas de inferencia* usando un cálculo de tipo *Hilbert* para la lógica de primer orden. Una prueba en dicho cálculo es una sucesión finita de fórmulas en la cual cada fórmula o bien es un axioma o bien puede deducirse de fórmulas anteriores en la sucesión aplicando una regla de inferencia. La última línea muestra la conclusión de la prueba, la declaración que se quería obtener. Esto es, una fórmula P se dirá *demostrable* si existe una prueba tal que la última fórmula de la prueba es, precisamente, P.

El objetivo de esta sección es presentar un cálculo similar que permita establecer la correción parcial de un programa imperativo. Para ello será necesario presentar una serie de axiomas y reglas de inferencia que traten tanto con declaraciones matemáticas (fórmulas de primer orden) como con ternas de Hoare. Dichos axiomas y reglas vendrán proporcionados por el sistema lógico conocido como la lógica de Hoare o lógica de Floyd-Hoare (la formulación del sistema deductivo de debe a Hoare, aunque algunas de las ideas subyacentes pertenecen a Floyd).

De manera análoga al caso de la lógica de primer orden, una demostración o una prueba en la lógica de Hoare será una sucesión finita, donde pueden aparecer tanto ternas de Hoare como fórmulas de primer orden, y tal que cada elemento de la sucesión o bien es un axioma de la lógica de Hoare o bien puede obtenerse a partir de elementos anteriores de la sucesión mediante la aplicación de reglas de inferencia. Una terna de Hoare $\{P\}\ C\ \{Q\}$ se dirá *demostrable*, y lo escribiremos

$$\vdash \{P\}\ C\ \{Q\},$$

si existe una prueba en la lógica de Hoare cuyo último elemento es, precisamente, dicha terna.

La lógica de Hoare proporciona un marco teórico para desarrollar la verificación formal de programas imperativos. Las pruebas de corrección sobre programas suelen ser complejas y normalmente se necesitan métodos formales para asegurar que son válidas. Por ello es importante que se muestren explícitamente los principios de razonamiento que se utilicen, con el fin de que se pueda analizar la robustez.

Nótese que en algunas situaciones la corrección de un sistema informático es muy importante o incluso crítica. Considérese el caso, por ejemplo, de los sistemas de los que depende la vida humana, como los controladores de reactores nucleares, los sistemas de frenado de coches, pilotaje de aviones por mandos electrónicos o equipos médicos controlados por software.

A continuación, se explica y se muestra con ejemplos el sistema deductivo de Hoare para el razonamiento sobre programas imperativos.

3.2.1. Axiomas y reglas de la lógica de Hoare

En este apartado, se describen los axiomas y reglas de inferencia de la lógica de Hoare. Expresaremos dichas reglas mediante el siguiente esquema

$$\frac{\vdash S_1, \ldots, \vdash S_n}{\vdash S}$$

Esto es, a partir de las *hipótesis* $\vdash S_1, \ldots, \vdash S_n$ se deduce la *conclusión* $\vdash S$. Estas hipótesis pueden ser ternas de Hoare o una mezcla entre ternas de Hoare y declaraciones matemáticas (esto es, fórmulas de primer orden).

Axiomas del dominio

En primer lugar, es necesario disponer de un conjunto de axiomas que capturen las propiedades matemáticas del dominio subyacente sobre el cual interpretaremos nuestros programas imperativos. En nuestro caso, fijaremos un lenguaje de primer orden con igualdad adecuado para expresar propiedades de los números enteros y que contenga, al menos, la suma y el producto como operaciones básicas; y añadiremos como un axioma de la lógica de Hoare cada propiedad de primer orden que sea verdadera en la estructura estándar de los números enteros $(\mathbb{Z}, +, \times, \dots)$.

Axiomas del dominio

$$\vdash P$$

para cualquier fórmula P verdadera en la estructura $\mathbb{Z}$.

Cuando en una prueba en la lógica de Hoare empleemos un axioma del dominio, usualmente escribiremos como justificación de dicho paso de la prueba *por lógica* o *por aritmética*.

Axioma de asignación

El axioma de asignación representa el hecho de que el valor de una variable V después de ejecutar el comando de asignación $V := E$ es igual al valor de la expresión E en el estado antes de ejecutarlo. Por tanto, cualquier propiedad P que se verificara antes de la asignación para la expresión E también ha de verificarse después de la asignación para la variable V.
Formalmente, escribimos $P[E/V]$ para expresar el resultado de reemplazar todas las ocurrencias de la variable V en la fórmula P por la expresión E.

Axioma de asignación

$$\vdash \{P[E/V]\}\ V := E\ \{P\}$$

donde V es una variable, E es una expresión, P es una condición y la notación $P[E/V]$ denota el resultado de sustituir todas las ocurrencias de V por el término E en P.

Ejemplos:
Las siguientes ternas de Hoare son demostrables por ser una instancia de un axioma de asignación.

- $\vdash$ `{X+1=n+1} X:=X+1 {X=n+1}`.

- $\vdash$ `{X+1=2} X:=X+1 {X=2}`.
- $\vdash$ `{Y=2} X:=2 {Y=X}`.
- $\vdash$ `{E=E} X:=E {X=E}`.

Puede llamar la atención que la aplicación del axioma de asignación sea *hacia atrás*. Sin embargo, la intuición nos puede llevar a cometer alguno de los dos errores siguientes en cuanto a la expresión de este axioma si intentamos aplicar el axioma *hacia adelante*.

1. $\vdash \{P\}\ V := E\ \{P[V/E]\}$.
   Donde la notación $P[V/E]$ denota el resultado de sustituir E por V en P. Esta formulación del axioma nos llevaría a obtener incongruencias.
   Por ejemplo, tomemos $P =$ `(x=0)`, $V =$ `x`, y $E = 1$. Entonces, se obtendría $\vdash$ `{x=0} x:=1 {x=0}`, que claramente se trata de una terna de Hoare falsa. Nótese que `(x=0)[x/1]` es igual a `(x=0)`, ya que `1` no ocurre en `(x=0)`.
2. $\vdash \{P\}\ V := E\ \{P[E/V]\}$.
   Esta formulación de axioma también nos llevaría a conclusiones erróneas.
   Por ejemplo, tomemos de nuevo $P =$ `(x=0)`, $V =$ `x`, y $E = 1$. Entonces se obtendría $\vdash$ `{x=0} x:=1 {1=0}`, que claramente se trata de una terna de Hoare falsa.

Reforzamiento de la precondición

Reforzamiento de la precondición

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

Si a partir de la condición P se tiene P', y la terna de Hoare $\{P'\}\ C\ \{Q\}$ es demostrable, se tiene que esta otra $\{P\}\ C\ \{Q\}$ también es demostrable, cuya precondición P es más restrictiva que la de la terna anterior.

Ejemplo:

Probar $\vdash$ `{X=n} X:=X+1 {X=n+1}`.

Paso 0. $\vdash$ `{X+1=n+1} X:=X+1 {X=n+1}`. (Axioma de asignación)

Paso 1. $\vdash$ `X=n` $\Rightarrow$ `X+1=n+1`. (Aritmética)

Paso 2. $\vdash$ `{X=n} X:=X+1 {X=n+1}`. (Reforzamiento de la precondición 0,1)

□

Debilitamiento de la postcondición

Debilitamiento de la postcondición

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

Esto es, si la terna de Hoare $\{P\}\ C\ \{Q'\}$ es demostrable y a partir de su postcondición Q' se deduce Q; entonces la terna $\{P\}\ C\ \{Q\}$ también es demostrable (pues su postcondición Q es más general que la de la terna anterior).

Ejemplo:

Probar $\vdash$ `{R=X} Q:=0 {R=X+(Y*Q)}`.

Paso 0. $\vdash$ `{R=X` $\wedge$ `0=0} Q:=0 {R=X` $\wedge$ `Q=0}`. (Axioma de asignación)

Paso 1. $\vdash$ `R=X` $\Rightarrow$ `R=X` $\wedge$ `0=0`. (Lógica)

Paso 2. $\vdash$ `{R=X} Q:=0 {R=X` $\wedge$ `Q=0}`. (Reforzamiento de la precondición 0,1)

Paso 3. $\vdash$ `R=X` $\wedge$ `Q=0` $\Rightarrow$ `R=X+(Y*Q)`. (Aritmética)

Paso 4. $\vdash$ `{R=X} Q:=0 {R=X+(Y*Q)}`. (Debilitamiento de la postcondición 2,3)

□

Las dos reglas anteriores se pueden condensar en una sola, como se explica a continuación.

Regla de la consecuencia

Regla de la consecuencia

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

Si a partir de la condición P se tiene la condición P', la terna de Hoare $\{P'\}\ C\ \{Q'\}$ es demostrable y, además, de su postcondición Q' se deduce Q; entonces se puede concluir que la terna $\{P\}\ C\ \{Q\}$ también es demostrable.

Regla de secuenciación

Regla de secuenciación

$$\frac{\vdash \{P\}\ C_1\ \{Q\}, \qquad \vdash \{Q\}\ C_2\ \{R\}}{\vdash \{P\}\ C_1; C_2\ \{R\}}$$

Sean las ternas de Hoare $\{P\}\ C_1\ \{Q\}$ y $\{Q\}\ C_2\ \{R\}$ demostrables, donde la postcondición de la primera terna coincide con la precondición de la segunda. Se deduce entonces que la tercera terna $\{P\}\ C_1; C_2\ \{R\}$, cuyo programa consiste en una secuenciación formada por los programas de las dos anteriores, también es demostrable.

Ejemplo:

Probar $\vdash$ `{X=x` $\wedge$ `Y=y} R:=X; X:=Y; Y:=R {Y=x` $\wedge$ `X=y}`.

Paso 0. $\vdash$ `{R=x` $\wedge$ `X=y} Y:=R {Y=x` $\wedge$ `X=y}`. (Axioma de asignación)

Paso 1. $\vdash$ `{R=x` $\wedge$ `Y=y} X:=Y {R=x` $\wedge$ `X=y}`. (Axioma de asignación)

Paso 2. $\vdash$ `{X=x` $\wedge$ `Y=y} R:=X {R=x` $\wedge$ `Y=y}`. (Axioma de asignación)

Paso 3. $\vdash$ `{R=x` $\wedge$ `Y=y} X:=Y; Y:=R {Y=x` $\wedge$ `X=y}`. (Regla de secuenciación 0,1)

Paso 4. $\vdash$ `{X=x` $\wedge$ `Y=y} R:=X; X:=Y; Y:=R {Y=x` $\wedge$ `X=y}`.
(Regla de secuenciación 2,3)

□

Regla del condicional

Regla del condicional

$$\frac{\vdash \{P \wedge S\}\ C_1\ \{Q\}, \qquad \vdash \{P \wedge \neg S\}\ C_2\ \{Q\}}{\vdash \{P\}\ \texttt{IF}\ S\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2\ \{Q\}}$$

Si las ternas de Hoare $\{P \wedge S\}\ C_1\ \{Q\}$ y $\{P \wedge \neg S\}\ C_2\ \{Q\}$ son demostrables, la correspondiente terna $\{P\}$ `IF` S `THEN` C_1 `ELSE` $C_2\ \{Q\}$ también lo será.

Ejemplo:

Probar $\vdash$ `{T} IF Y<X THEN MAX:=X ELSE MAX:=Y {MAX=max(X,Y)}`. Para ello se tiene que probar:

1) $\vdash$ `{T` $\wedge$ `Y<X} MAX:=X {MAX=max(X,Y)}`

2) $\vdash$ `{T` $\wedge$ $\neg$`(Y<X)} MAX:=Y {MAX=max(X,Y)}`

Demostramos 1):

Paso 0. $\vdash$ `{X=max(X,Y)} MAX:=X {MAX=max(X,Y)}` . (Axioma de asignación)

Paso 1. $\vdash$ `T` $\wedge$ `(Y<X)` $\Rightarrow$ `X=max(X,Y)`. (Aritmética)

Paso 2. $\vdash$ `{T` $\wedge$ `(Y<X)} MAX:=X {MAX=max(X,Y)}`.
(Reforzamiento de la precondición 0,1)

Demostramos 2):

Paso 3. $\vdash$ `{Y=max(X,Y)} MAX:=Y {MAX=max(X,Y)}` . (Axioma de asignación)

Paso 4. $\vdash$ `T` $\wedge$ $\neg$`(Y<X)` $\Rightarrow$ `Y=max(X,Y)`. (Aritmética)

Paso 5. $\vdash$ `{T` $\wedge$ $\neg$`(Y<X)} MAX:=Y {MAX=max(X,Y)}`.
(Reforzamiento de la precondición 3,4)

Paso 6. $\vdash$ `{T} IF Y<X THEN MAX:=X ELSE MAX:=Y {MAX=max(X,Y)}`. (Regla del condicional 2,5)

□

Regla `While`

Es el axioma más interesante de esta lógica. La demostrabilidad de las ternas de Hoare cuyos programas contienen algún comando `WHILE` es más elaborada que los ya estudiados. Veamos cómo funciona.

El concepto clave es el de *invariante* de un bucle `WHILE`. Dada una instrucción del tipo `WHILE` S `DO` C, se dirá que una condición P es un invariante para dicho bucle `WHILE` si la terna $\{P \wedge S\}\ C\ \{P\}$ es demostrable. Esto es, P es un invariante del bucle si dicha condición se mantiene cierta tras cualquier número de ejecuciones del cuerpo del bucle.

Teniendo en cuenta la semántica de una instrucción tipo `WHILE` anteriormente descrita, la regla de la lógica de Hoare para dicho comando es como sigue:

Regla `While`

$$\frac{\vdash \{P \wedge S\}\ C\ \{P\}}{\vdash \{P\}\ \texttt{WHILE}\ S\ \texttt{DO}\ C\ \{P \wedge \neg S\}}$$

donde P se dirá un *invariante* del bucle.

Como veremos en los ejemplos que se detallan a continuación, gran parte de la dificultad en la demostración formal de la corrección parcial de un programa imperativo se encuentra en determinar un invariante adecuado para cada uno de los bucles `WHILE` que aparezcan en el programa.

3.2.2. Ejemplos de ternas demostrables: corrección parcial

En lo que sigue, se exponen varios ejemplos de ternas de Hoare demostrables que corresponden a la verificación formal de programas completos.

Ejemplo 1:

Probaremos que la siguiente terna de Hoare es demostrable, siendo C_6 un programa que calcula en la variable `Z` la potencia de `X` elevado a `Y`.

`{X=x ∧ Y=y ∧ 0≤Y}`

$$C_6 = \begin{cases} \texttt{Z:=1;} \\ \texttt{R:=0;} \\ \texttt{WHILE R}\neq\texttt{Y DO (Z:=Z*X; R:=R+1)} \end{cases}$$

`{Z=`X^Y `∧ X=x ∧ Y=y}`.

Para ello, en primer lugar, es necesario encontrar un invariante para el comando WHILE.

$$\text{Invariante } P \equiv \text{“}Z = X^R \wedge X = x \wedge Y = y\text{”}$$

Probemos que P es un invariante. Es decir, veamos que se cumple que:

$\vdash$ `{P ∧ R≠Y} Z:=Z*X; R:=R+1 {P}`

Paso 0. $\vdash$ `{Z=`X^{R+1} `∧ X=x ∧ Y=y} R:=R+1 {Z=`X^R `∧ X=x ∧ Y=y}`.
(Axioma de asignación)

Paso 1. $\vdash$ `{Z*X=`X^{R+1} `∧ X=x ∧ Y=y} Z:=Z*X {Z=`X^{R+1} `∧ X=x ∧ Y=y}`.
(Axioma de asignación)

Paso 2. $\vdash$ {Z*X=X^{R+1} $\wedge$ X=x $\wedge$ Y=y} Z:=Z*X; R:=R+1 {Z=X^R $\wedge$ X=x $\wedge$ Y=y}. (Regla de secuenciación 0,1)

Paso 3. $\vdash$ (Z=X^R $\wedge$ X=x $\wedge$ Y=y $\wedge$ R$\neq$Y) $\Rightarrow$ (Z*X=X^{R+1} $\wedge$ X=x $\wedge$ Y=y). (Aritmética)

Paso 4. $\vdash$ {Z=X^R $\wedge$ X=x $\wedge$ Y=y $\wedge$ R$\neq$Y} Z:=Z*X; R:=R+1 {Z=X^R $\wedge$ X=x $\wedge$ Y=y}. (Reforzamiento de la precondición 2,3)

✓ Es invariante.

Paso 5. $\vdash$ {Z=X^R $\wedge$ X=x $\wedge$ Y=y} WHILE R$\neq$Y DO Z:=Z*X; R:=R+1 {Z=X^R $\wedge$ X=x $\wedge$ Y=y $\wedge$ $\neg$(R$\neq$Y)}. (Regla While para el invariante anterior)

Paso 6. $\vdash$ (Z=X^R $\wedge$ X=x $\wedge$ Y=y $\wedge$ $\neg$(R$\neq$Y)) $\Rightarrow$ (Z=X^Y $\wedge$ X=x $\wedge$ Y=y). (Lógica)

Paso 7. $\vdash$ {Z=X^R $\wedge$ X=x $\wedge$ Y=y} WHILE R$\neq$Y DO Z:=Z*X; R:=R+1 {Z=X^Y $\wedge$ X=x $\wedge$ Y=y}. (Debilitamiento de la postcondición 5,6)

Paso 8. $\vdash$ {Z=X^0 $\wedge$ X=x $\wedge$ Y=y} R:=0 {Z=X^R $\wedge$ X=x $\wedge$ Y=y}. (Axioma de asignación)

Paso 9. $\vdash$ {1=X^0 $\wedge$ X=x $\wedge$ Y=y} Z:=1 {Z=X^0 $\wedge$ X=x $\wedge$ Y=y}. (Axioma de asignación)

Paso 10. $\vdash$ {1=X^0 $\wedge$ X=x $\wedge$ Y=y} C_6 {Z=X^Y $\wedge$ X=x $\wedge$ Y=y}. (Regla de secuenciación 7,8,9)

Paso 11. $\vdash$ (X=x $\wedge$ Y=y $\wedge$ 0$\leq$Y) $\Rightarrow$ (1=X^0 $\wedge$ X=x $\wedge$ Y=y). (Aritmética)

Paso 12. $\vdash$ {X=x $\wedge$ Y=y $\wedge$ 0$\leq$Y} C_6 {Z=X^Y $\wedge$ X=x $\wedge$ Y=y}. (Reforzamiento de la precondición 10,11)

□

Ejemplo 2:

Probemos que la siguiente terna de Hoare para C_7 es demostrable. El programa C_7 calcula en Y la suma de los números de 1 a X. Esto es, Y = sum(X)=$\sum_{i=1}^{X} i$.

{X=n $\wedge$ 0$\leq$n}

$$C_7 = \begin{cases} \texttt{Y:=0;} \\ \texttt{WHILE X}\neq\texttt{0 DO (Y=Y+X; X=X-1)} \end{cases}$$

{Y=sum(n)}.

Se propone como invariante la siguiente condición:

$$\text{Invariante } P \equiv \text{“}Y = sum(n) - sum(X) \ \wedge \ 0 \leq X\text{”}$$

Probamos que se cumple que:

$\vdash$ {P $\wedge$ X$\neq$0} Y:=Y+X; X:=X-1 {P}

Paso 0. $\vdash$ {Y=sum(n)-sum(X-1) $\wedge$ 0$\leq$X-1} X:=X-1 {Y=sum(n)-sum(X) $\wedge$ 0$\leq$X}. (Axioma de asignación)

Paso 1. $\vdash$ {Y+X=sum(n)-sum(X-1) $\wedge$ 0$\leq$X-1} Y:=Y+X {Y=sum(n)-sum(X-1) $\wedge$ 0$\leq$X-1}. (Axioma de asignación)

Paso 2. $\vdash$ {Y+X=sum(n)-sum(X-1) $\wedge$ 0$\leq$X-1} Y:=Y+X; X:=X-1 {Y=sum(n)-sum(X) $\wedge$ 0$\leq$X}. (Regla de secuenciación 0,1)

Paso 3. $\vdash$ (Y=sum(n)-sum(X) $\wedge$ 0$\leq$X $\wedge$ X$\neq$0) $\Rightarrow$ (Y+X=sum(n)-sum(X-1) $\wedge$ 0$\leq$X-1). (Aritmética)

Paso 4. $\vdash$ {Y=sum(n)-sum(X) $\wedge$ 0$\leq$X $\wedge$ X$\neq$0} Y:=Y+X; X:=X-1 {Y=sum(n)-sum(X) $\wedge$ 0$\leq$X}. (Reforzamiento de la precondición 2,3)

✓ Es invariante.

Paso 5. $\vdash$ {Y=sum(n)-sum(X) $\wedge$ 0$\leq$X} WHILE X$\neq$0 DO (Y:=Y+X; X:=X-1) {Y=sum(n)-sum(X) $\wedge$ 0$\leq$X $\wedge$ $\neg$(X$\neq$0)}. (Regla While para el invariante anterior)

Paso 6. $\vdash$ (Y=sum(n)-sum(X) $\wedge$ 0$\leq$X $\wedge$ $\neg$(X$\neq$0)) $\Rightarrow$ (Y=sum(n)). (Lógica)

Paso 7. $\vdash$ {Y=sum(n)-sum(X) $\wedge$ 0$\leq$X} WHILE X$\neq$0 DO (Y:=Y+X; X:=X-1) {Y=sum(n)}. (Debilitamiento de la postcondición 5,6)

Paso 8. $\vdash$ {0=sum(n)-sum(X) $\wedge$ 0$\leq$X} Y:=0 {Y=sum(n)-sum(X) $\wedge$ 0$\leq$X}. (Axioma de asignación)

Paso 9. $\vdash$ {0=sum(n)-sum(X) $\wedge$ 0$\leq$X} C_7 {Y=sum(n)}. (Regla de secuenciación 7,8)

Paso 10. $\vdash$ (X=n $\wedge$ 0$\leq$n) $\Rightarrow$ (0=sum(n)-sum(X) $\wedge$ 0$\leq$X). (Aritmética)

Paso 11. $\vdash$ {X=n $\wedge$ 0$\leq$n} C_7 {Y=sum(n)}. (Reforzamiento de la precondición 9,10)

□

Ejemplo 3:

Probaremos que la siguiente terna de Hoare para el programa C_8 es demostrable, donde con la ejecución de C_8 se obtiene en la variable `R` una aproximación de la raíz cuadrada de `X`.

`{0≤X}`

$$C_8 = \begin{cases} \texttt{R:=0;} \\ \texttt{T:=1;} \\ \texttt{WHILE T≤X DO (R:=R+1; T:=T+(2*R+1))} \end{cases}$$

$\texttt{\{R}^2 \texttt{≤X} \wedge \texttt{X<(R+1)}^2\texttt{\}}$.

Se ha encontrado el siguiente invariante para al bucle WHILE:

$$\text{Invariante } P \equiv \text{“}R^2 \leq X \ \wedge \ (R+1)^2 = T\text{”}$$

Veamos que, de hecho, P es un tal invariante. Es decir, hemos de comprobar que:

$\vdash$ `{P ∧ T≤X} R:=R+1; T:=T+(2*R+1) {P}`

Paso 0. $\vdash$ $\texttt{\{R}^2 \texttt{≤X} \wedge \texttt{(R+1)}^2\texttt{=T+(2*R+1)\}}$ `T:=T+(2*R+1)` $\texttt{\{R}^2 \texttt{≤X} \wedge \texttt{(R+1)}^2\texttt{=T\}}$. (Axioma de asignación)

Paso 1. $\vdash$ $\texttt{\{(R+1)}^2 \texttt{≤X} \wedge \texttt{(R+2)}^2\texttt{=T+(2*R+3)\}}$ `R:=R+1`
$\texttt{\{R}^2 \texttt{≤X} \wedge \texttt{(R+1)}^2\texttt{=T+(2*R+1)\}}$. (Axioma de asignación+Arimética+Reforzamiento de la precondición)

Paso 2. $\vdash$ $\texttt{\{(R+1)}^2 \texttt{≤X} \wedge \texttt{(R+2)}^2\texttt{=T+(2*R+3)\}}$ `R:=R+1; T:=T+(2*R+1)`
$\texttt{\{R}^2 \texttt{≤X} \wedge \texttt{(R+1)}^2\texttt{=T\}}$. (Regla de secuenciación 0,1)

Paso 3. $\vdash$ $\texttt{(R}^2 \texttt{≤X} \wedge \texttt{(R+1)}^2\texttt{=T} \wedge \texttt{T≤X)} \Rightarrow$
$\texttt{((R+1)}^2 \texttt{≤X} \wedge \texttt{(R+2)}^2\texttt{=T+(2*R+3))}$. (Aritmética)

Paso 4. $\vdash$ $\texttt{\{R}^2 \texttt{≤X} \wedge \texttt{(R+1)}^2\texttt{=T} \wedge \texttt{T≤X\}}$ `R:=R+1; T:=T+(2*R+1)`
$\texttt{\{R}^2 \texttt{≤X} \wedge \texttt{(R+1)}^2\texttt{=T\}}$. (Reforzamiento de la precondición 2,3)

✓ Es invariante.

Paso 5. $\vdash$ $\texttt{\{R}^2 \texttt{≤X} \wedge \texttt{(R+1)}^2\texttt{=T\}}$ `WHILE T≤X DO (R:=R+1; T:=T+(2*R+1))`
$\texttt{\{R}^2 \texttt{≤X} \wedge \texttt{(R+1)}^2\texttt{=T} \wedge \neg\texttt{(T≤X)\}}$. (Regla While para el invariante anterior)

Paso 6. $\vdash$ (R$^2 \leq$X $\wedge$ (R+1)^2=T $\wedge$ $\neg$(T$\leq$X)) $\Rightarrow$ (R$^2 \leq$X $\wedge$ X$<$(R+1)^2). (Aritmética)

Paso 7. $\vdash$ {R$^2 \leq$X $\wedge$ (R+1)^2=T} WHILE T$\leq$X DO (R:=R+1; T:=T+(2*R+1)) {R$^2 \leq$X $\wedge$ X$<$(R+1)^2}. (Debilitamiento de la postcondición 5,6)

Paso 8. $\vdash$ {R$^2 \leq$X $\wedge$ (R+1)^2=1} T:=1 {R$^2 \leq$X $\wedge$ (R+1)^2=T}. (Axioma de asignación)

Paso 9. $\vdash$ {0$\leq$X $\wedge$ 1=1} R:=0 {R$^2 \leq$X $\wedge$ (R+1)^2=1}. (Axioma de asignación + Aritmética + Reforzamiento de la precondición)

Paso 10. $\vdash$ {0$\leq$X $\wedge$ 1=1} C_8 {R$^2 \leq$X $\wedge$ (R+1)^2 $<$T}. (Regla de secuenciación 7,8,9)

Paso 11. $\vdash$ 0$\leq$X $\Rightarrow$ 0$\leq$X $\wedge$ 1=1. (Lógica)

Paso 12. $\vdash$ {0$\leq$X} C_8 {R$^2 \leq$X $\wedge$ (R+1)^2 $<$T}. (Reforzamiento de la precondición 10,11)

□

Ejemplo 4:

Probaremos que la siguiente terna de Hoare para C_9 es demostrable. El programa C_9 calcula en la variable T la diferencia entre Y y X.

{X=m $\wedge$ Y=n $\wedge$ 0$\leq$m}

$$C_9 = \begin{cases} \texttt{R:=m;} \\ \texttt{T:=n;} \\ \texttt{WHILE 0<R DO (R:=R-1; T:=T-1)} \end{cases}$$

{T=n-m}.

Se propone como invariante:

$$\text{Invariante } P \equiv \text{“}m + T = n + R \ \wedge \ 0 \leq R \ \wedge \ R \leq m\text{”}$$

Veamos que P es un invariante. Es decir, veamos que se cumple que:

$\vdash$ {P $\wedge$ 0$<$R} R:=R-1; T:=T-1 {P}

Paso 0. $\vdash$ {m+T-1=n+R $\wedge$ 0$\leq$R $\wedge$ R$\leq$m} T:=T-1 {m+T=n+R $\wedge$ 0$\leq$R $\wedge$ R$\leq$m}. (Axioma de asignación)

Paso 1. $\vdash$ {m+T-1=n+R-1 $\wedge$ 0$\leq$R-1 $\wedge$ R-1$\leq$m} R:=R-1
{m+T-1=n+R $\wedge$ 0$\leq$R $\wedge$ R$\leq$m}. (Axioma de asignación)

Paso 2. $\vdash$ {m+T-1=n+R-1 $\wedge$ 0$\leq$R-1 $\wedge$ R-1$\leq$m} R:=R-1; T:=T-1
{m+T=n+R $\wedge$ 0$\leq$R $\wedge$ R$\leq$m}. (Regla de secuenciación 0,1)

Paso 3. $\vdash$ (m+T=n+R $\wedge$ 0$\leq$R $\wedge$ R$\leq$m $\wedge$ 0<R) $\Rightarrow$ (m+T-1=n+R-1 $\wedge$ 0$\leq$R-1 $\wedge$ R-1$\leq$m).
(Aritmética)

Paso 4. $\vdash$ {m+T=n+R $\wedge$ 0$\leq$R $\wedge$ R$\leq$m $\wedge$ 0<R} R:=R-1; T:=T-1
{m+T=n+R $\wedge$ 0$\leq$R $\wedge$ R$\leq$m}. (Reforzamiento de la precondición 2,3)

✓ Es invariante.

Paso 5. $\vdash$ {m+T=n+R $\wedge$ 0$\leq$R $\wedge$ R$\leq$m} WHILE 0<R DO (R:=R-1; T:=T-1)
{m+T=n+R $\wedge$ 0$\leq$R $\wedge$ R$\leq$m $\wedge$ $\neg$(0<R)}. (Regla While para el invariante anterior)

Paso 6. $\vdash$ (m+T=n+R $\wedge$ 0$\leq$R $\wedge$ R$\leq$m $\wedge$ $\neg$(0<R)) $\Rightarrow$ T=n-m. (Aritmética)

Paso 7. $\vdash$ {m+T=n+R $\wedge$ 0$\leq$R $\wedge$ R$\leq$m} WHILE 0<R DO (R:=R-1; T:=T-1)
{T=n-m}. (Debilitamiento de la postcondición 5,6)

Paso 8. $\vdash$ {m+n=n+R $\wedge$ 0$\leq$R $\wedge$ R$\leq$m} T:=n {m+T=n+R $\wedge$ 0$\leq$R $\wedge$ R$\leq$m}.
(Axioma de asignación)

Paso 9. $\vdash$ {m+n=n+m $\wedge$ 0$\leq$m $\wedge$ m$\leq$m} R:=m {m+n=n+R $\wedge$ 0$\leq$R $\wedge$ R$\leq$m}.
(Axioma de asignación)

Paso 10. $\vdash$ {m+n=n+m $\wedge$ 0$\leq$m $\wedge$ m$\leq$m} C_9 {T=n-m}.
(Regla de secuenciación 7,8,9)

Paso 11. $\vdash$ (X=m $\wedge$ Y=n $\wedge$ 0$\leq$m) $\Rightarrow$ (m+n=n+m $\wedge$ 0$\leq$m $\wedge$ m$\leq$m). (Aritmética)

Paso 12. $\vdash$ {X=m $\wedge$ Y=n $\wedge$ 0$\leq$m} C_9 {T=n-m}.
(Reforzamiento de la precondición 10,11)

□

Ejemplo 5:

Probaremos que la siguiente terna de Hoare para C_{10} es demostrable, donde tras la ejecución del programa C_{10} se obtiene la potencia de X en base 2. Esta operación se ha expresado con la notación de función pot2(X)=2^X.

{X=n $\wedge$ 0$\leq$n}

$$C_{10} = \begin{cases} \texttt{R:=0;} \\ \texttt{T:=1;} \\ \texttt{WHILE R}\neq\texttt{n DO (R:=R+1; T:=2*T)} \end{cases}$$

{T=pot2(n)}.

Primeramente, se demuestra que se tiene un invariante:

$$\text{Invariante } P \equiv \text{“}T = pot2(R) \ \wedge \ 0 \leq R \ \wedge \ R \leq n\text{”}$$

Veamos que se cumple que:

$\vdash$ {P $\wedge$ R$\neq$n} R:=R+1; T:=2*T {P}

Paso 0. $\vdash$ {2*T=pot2(R) $\wedge$ 0$\leq$R $\wedge$ R$\leq$n} T:=2*T
{T=pot2(R) $\wedge$ 0$\leq$R $\wedge$ R$\leq$n}. (Axioma de asignación)

Paso 1. $\vdash$ {2*T=pot2(R+1) $\wedge$ 0$\leq$R+1 $\wedge$ R+1$\leq$n} R:=R+1
{2*T=pot2(R) $\wedge$ 0$\leq$R $\wedge$ R$\leq$n}. (Axioma de asignación)

Paso 2. $\vdash$ {2*T=pot2(R+1) $\wedge$ 0$\leq$R+1 $\wedge$ R+1$\leq$n} R:=R+1; T:=2T
{T=pot2(R) $\wedge$ 0$\leq$R $\wedge$ R$\leq$n}. (Regla de secuenciación 0,1)

Paso 3. $\vdash$ (T=pot2(R) $\wedge$ 0$\leq$R $\wedge$ R$\leq$n $\wedge$ R$\neq$n) $\Rightarrow$
(2*T=pot2(R+1) $\wedge$ 0$\leq$R+1 $\wedge$ R+1$\leq$n). (Aritmética)

Paso 4. $\vdash$ {T=pot2(R) $\wedge$ 0$\leq$R $\wedge$ R$\leq$n $\wedge$ R$\neq$n} R:=R+1; T:=2*T
{T=pot2(R) $\wedge$ 0$\leq$R $\wedge$ R$\leq$n}. (Reforzamiento de la precondición 2,3)

✓ Es invariante.

Paso 5. $\vdash$ {T=pot2(R) $\wedge$ 0$\leq$R $\wedge$ R$\leq$n} WHILE R$\neq$n DO (R:=R+1; T=2*T)
{T=pot2(R) $\wedge$ 0$\leq$R $\wedge$ R$\leq$n $\wedge$ $\neg$(R$\neq$n)}. (Regla While para el invariante anterior)

Paso 6. $\vdash$ (T=pot2(R) $\wedge$ 0$\leq$R $\wedge$ R$\leq$n $\wedge$ $\neg$(R$\neq$n)) $\Rightarrow$ T=pot2(n). (Lógica)

Paso 7. $\vdash$ {T=pot2(R) $\wedge$ 0$\leq$R $\wedge$ R$\leq$n} WHILE R$\neq$n DO (R:=R+1; T=2*T)
{T=pot2(n)}. (Debilitamiento de la postcondición 5,6)

Paso 8. $\vdash$ {1=pot2(R) $\wedge$ 0$\leq$R $\wedge$ R$\leq$n} T:=1 {T=pot2(R) $\wedge$ 0$\leq$R $\wedge$ R$\leq$n}.
(Axioma de asignación)

Paso 9. $\vdash$ {1=pot2(0) $\wedge$ 0$\leq$0 $\wedge$ 0$\leq$n} R:=0 {1=pot2(R) $\wedge$ 0$\leq$R $\wedge$ R$\leq$n}.
(Axioma de asignación)

Paso 10. $\vdash$ {1=pot2(0) $\wedge$ 0≤0 $\wedge$ 0≤n} C_{10} {T=pot2(n)}.
(Regla de secuenciación 7,8,9)

Paso 11. $\vdash$ (X=n $\wedge$ 0≤n) $\Rightarrow$ (1=pot2(0) $\wedge$ 0≤0 $\wedge$ 0≤n). (Aritmética)

Paso 12. $\vdash$ {X=n $\wedge$ 0≤n} C_{10} {T=pot2(n)}.
(Reforzamiento de la precondición 10,11)

□

Ejemplo 6:

Probaremos que la siguiente terna de Hoare para C_{11} es demostrable. El programa C_{11} calcula el cociente y el resto de la división euclídea de X entre Y.

{X=a $\wedge$ Y=b $\wedge$ 0≤a $\wedge$ 0<b}

$$C_{11} = \begin{cases} \texttt{C:=0;} \\ \texttt{R:=a;} \\ \texttt{WHILE b≤R DO (C:=C+1; R:=R-b)} \end{cases}$$

{a=b*C+R $\wedge$ R<b}.

Primeramente, consideramos el siguiente invariante para el bucle:

$$\text{Invariante } P \equiv \text{“}a = b * C + R \ \wedge \ 0 \leq R\text{”}$$

Veamos que, de hecho, es invariante. Es decir, veamos que se cumple que:

$\vdash$ {P $\wedge$ b≤R} C:=C+1; R:=R-b {P}

Paso 0. $\vdash$ {a=b*C+(R-b) $\wedge$ 0≤R-b} R:=R-b {a=b*C+R $\wedge$ 0≤R}.
(Axioma de asignación)

Paso 1. $\vdash$ {a=b*(C+1)+(R-b) $\wedge$ 0≤R-b} C:=C+1 {a=b*C+(R-b) $\wedge$ 0≤R-b}.
(Axioma de asignación)

Paso 2. $\vdash$ {a=b*(C+1)+(R-b) $\wedge$ 0≤R-b} C:=C+1; R:=R-b {a=b*C+R $\wedge$ 0≤R}.
(Regla de secuenciación 0,1)

Paso 3. $\vdash$ (a=b*C+R $\wedge$ 0≤R $\wedge$ b≤R) $\Rightarrow$ (a=b*(C+1)+(R-b) $\wedge$ 0≤R-b).
(Aritmética)

Paso 4. $\vdash$ {a=b*C+R $\wedge$ 0≤R $\wedge$ b≤R} C:=C+1; R:=R-b {a=b*C+R $\wedge$ 0≤R}.
(Reforzamiento de la precondición 2,3)

✓ Es invariante.

Paso 5. $\vdash$ {a=b*C+R $\wedge$ 0$\leq$R} WHILE b$\leq$R DO (C:=C+1; R:=R-b) {a=b*C+R $\wedge$ 0$\leq$R $\wedge$ $\neg$(b$\leq$R)}. (Regla While para el invariante anterior)

Paso 6. $\vdash$ (a=b*C+R $\wedge$ 0$\leq$R $\wedge$ $\neg$(b$\leq$R)) $\Rightarrow$ (a=b*C+R $\wedge$ R<b). (Aritmética)

Paso 7. $\vdash$ {a=b*C+R $\wedge$ 0$\leq$R} WHILE b$\leq$R DO (C:=C+1; R:=R-b) {a=b*C+R $\wedge$ R<b}. (Debilitamiento de la postcondición 5,6)

Paso 8. $\vdash$ {a=b*C+a $\wedge$ 0$\leq$a} R:=a {a=b*C+R $\wedge$ 0$\leq$R}. (Axioma de asignación)

Paso 9. $\vdash$ {a=a $\wedge$ 0$\leq$a} C:=0 {a=b*C+a $\wedge$ 0$\leq$a}. (Axioma de asignación + Aritmética + Reforzamiento de la precondición)

Paso 10. $\vdash$ {a=a $\wedge$ 0$\leq$a} C_{11} {a=b*C+R $\wedge$ R<b}.
(Regla de secuenciación 7,8,9)

Paso 11. $\vdash$ (X=a $\wedge$ Y=b $\wedge$ 0$\leq$a $\wedge$ 0<b) $\Rightarrow$ (a=a $\wedge$ 0$\leq$a). (Aritmética)

Paso 12. $\vdash$ {X=a $\wedge$ Y=b $\wedge$ 0$\leq$a $\wedge$ 0<b} C_{11} {a=b*C+R $\wedge$ R<b}.
(Reforzamiento de la precondición 10,11)

□

3.2.3. Adecuación y completitud para la corrección parcial

Un sistema lógico se dirá *adecuado* si toda expresión demostrable en él es verdadera. Un sistema lógico se dirá *completo* si toda expresión verdadera escrita en su lenguaje es demostrable en el sistema. La lógica de Hoare para la corrección parcial que hemos estudiado en las secciones anteriores es tanto adecuada como completa.

Teorema 3.2.1 (Adecuación) $\vdash \{P\}\ C\ \{Q\} \implies \models \{P\}\ C\ \{Q\}$.

Prueba: (Idea) Basta comprobar que cada axioma y cada regla de la lógica de Hoare son adecuados y razonar por inducción sobre la longitud de una prueba en la lógica de Hoare.

Teorema 3.2.2 (Completitud) $\models \{P\}\ C\ \{Q\} \implies \vdash \{P\}\ C\ \{Q\}$.

La prueba de la completitud de la lógica de Hoare es más elaborada y descansa en el concepto de *precondición más débil* que describimos a continuación.

Para motivar la definición, consideremos, por ejemplo, la asignación X=:2*Y+1. Una terna de Hoare verdadera para dicho comando es:

{Y$\leq$ 3} X:=2*Y+1 {(X$\leq$ 7) $\wedge$ (Y$\leq$ 3)}.

Pero `Y`≤ 3 no es la única precondición que hace la postcondición cierta. Otra tal precondición podría ser:

$$\texttt{\{Y=1} \lor \texttt{Y=3\}\ X:=2*Y+1\ \{(X}\leq 7) \land \texttt{(Y}\leq 3)\texttt{\}}.$$

Ahora bien, la segunda precondición `Y=1` $\lor$ `Y=3` es menos interesante que la primera `Y`≤ 3, pues la segunda no caracteriza *todos* los estados iniciales desde los cuales la computación del programa alcanzará un estado satisfaciendo la postcondición.

Queremos pues elegir la *precondición menos restrictiva* que haga cierta una terna de Hoare. Ello lo conseguiremos mediante el concepto de precondición más débil.

Definición 3.2.1 *Diremos que una condición P es más débil que una condición Q si la fórmula de primer orden $Q \Rightarrow P$ es verdadera en $\mathbb{Z}$.*

Definición 3.2.1 *Dados un programa C y una condición Q, la precondición más débil para C y Q, que denotaremos por $wp(C, Q)$, es la condición P más débil tal que la terna de Hoare $\{P\}\ C\ \{Q\}$ es verdadera.*

Por ejemplo, $wp(\texttt{X:=2*Y+1}, \texttt{X} \leq 7) = \texttt{Y} \leq 3$.

De la propia definición se sigue que:

Lema 3.2.1 *La terna de Hoare $\{P\}\ C\ \{Q\}$ es verdadera si, y sólo si, la fórmula $P \Rightarrow wp(C, Q)$ es verdadera.*

Una propiedad esencial para la prueba del teorema de completitud es que la condición $wp(C, Q)$ es, de hecho, expresable en el lenguaje de primer orden subyacente a la lógica de Hoare. Es por ello que hemos supuesto que nuestro lenguaje de primer orden contiene, al menos, a la suma y al producto como operaciones básicas, pues para expresar en la lógica de primer orden la precondición más débil correspondiente a un comando tipo `WHILE` es necesario usar la función β de Gödel (u otra de similar naturaleza) para codificar convenientemente sucesiones finitas.

Usando que la condición $wp(C, Q)$ es expresable y razonando por inducción estructural, se prueba que:

Lema 3.2.2 *Para todo C y Q, se tiene que $\vdash \{wp(C, Q)\}\ C\ \{Q\}$.*

Podemos ahora dar un esquema de la prueba del teorema de completitud para la lógica de Hoare:

$$\begin{array}{lcll} \models \{P\}\ C\ \{Q\} & \Longrightarrow & \models P \Rightarrow wp(C, Q) & \text{(Lema 3.2.1)} \\ & \Longrightarrow & \vdash P \Rightarrow wp(C, Q) & \text{(Axioma del dominio)} \\ & \Longrightarrow & \vdash \{wp(C, Q)\}\ C\ \{Q\} & \text{(Lema 3.2.2)} \\ & \Longrightarrow & \vdash \{P\}\ C\ \{Q\} & \text{(Reforzamiento de la precondición)} \end{array}$$

3.2.4. Corrección total

Los axiomas de la lógica de Hoare, ya descritos en la sección (3.2.1), sirven para probar que un programa es correcto parcialmente. No obstante, el sistema de la lógica de Hoare se puede ampliar para probar que tales programas son totalmente correctos, en el caso que corresponda. Es decir, para que se pueda demostrar que su ejecución termina. [1]

Como se dijo anteriormente, para demostrar la corrección total se suele probar de forma separada la corrección parcial y la terminación.

$$\boxed{\text{Corrección total} = \text{Corrección parcial} + \text{Terminación}}$$

De esta forma, si una terna de Hoare es correcta totalmente entonces también lo es parcialmente. Esto es, se tiene que:

$$\frac{\vdash [P]\ C\ [Q]}{\vdash \{P\}\ C\ \{Q\}}$$

Dicha relación ya no es cierta, en general, en sentido inverso. Ahora bien, si nos restringimos a programas en los que no interviene el comando `WHILE`, entonces dichos programas siempre terminan. Todos los comandos estudiados, a excepción del comando `WHILE`, terminan para cualquier estado inicial a partir del cual se ejecuten. En consecuencia, la formulación de sus axiomas será la misma que en el caso de la correción parcial, reemplazando "{ }"por "[ ]". Esto es, para todos los comandos excepto el comando `WHILE` se cumple que:

$$\frac{\vdash \{P\}\ C\ \{Q\}}{\vdash [P]\ C\ [Q]}$$

Por tanto, todo el esfuerzo para obtener la versión de la lógica de Hoare para la corrección total se centrará en proponer una nueva formulación para la regla del comando `WHILE`.

A continuación se muestran las reglas y los axiomas de la lógica de Hoare para la corrección total. La única diferencia importante respecto al sistema deductivo descrito en la sección (3.2.1) radica en la nueva regla para el comando `WHILE` descrita al final de la presente sección.

[1] En cuanto a la terminación de los programas imperativos introducidos, como en [7], supondremos que los errores del tipo $1/0$ o $fact(-1)$ no causan problemas en el lenguaje. Por tanto, la única causa que puede provocar que un programa no pare será la ejecución infinita de un bucle `WHILE`.

Axiomas del dominio para la corrección total

Axiomas del dominio

$$\vdash P$$

para cualquier fórmula P verdadera en la estructura $\mathbb{Z}$.

Axioma de asignación para la corrección total

Axioma de asignación

$$\vdash [P[E/V]]\ V := E\ [P]$$

donde V es una variable, E es una expresión, P es una condición y la notación $P[E/V]$ denota el resultado de sustituir todas las ocurrencias de V por el término E en P.

Reforzamiento de la precondición para la corrección total

Reforzamiento de la precondición

$$\frac{\vdash P \Rightarrow P', \qquad \vdash [P']\ C\ [Q]}{\vdash [P]\ C\ [Q]}$$

Debilitamiento de la postcondición para la corrección total

Debilitamiento de la postcondición

$$\frac{\vdash [P]\ C\ [Q'], \qquad \vdash Q' \Rightarrow Q}{\vdash [P]\ C\ [Q]}$$

Las dos reglas anteriores se pueden condensar en una sola, como se muestra a continuación.

Regla de la consecuencia para la corrección total

Regla de la consecuencia

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

Regla de secuenciación para la corrección total

Regla de secuenciación

$$\frac{\vdash [P]\ C_1\ [Q], \qquad \vdash [Q]\ C_2\ [R]}{\vdash [P]\ C_1; C_2\ [R]}$$

Regla del condicional para la corrección total

Regla del condicional

$$\frac{\vdash [P \wedge S]\ C_1\ [Q], \qquad \vdash [P \wedge \neg S]\ C_2\ [Q]}{\vdash [P]\ \texttt{IF}\ S\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2\ [Q]}$$

Regla `While` para la corrección total

En el caso de la corrección parcial, este axioma se presentó como "el más interesante de la lógica de Hoare", por razones ya explicadas. Este comentario es, si cabe, aún más apropiado en el caso de la corrección total. Este comando es el único de nuestro pequeño lenguaje imperativo que no presenta una terminación inmediata, que podría no terminar para ciertos estados iniciales. Considérese por ejemplo el comando

```
WHILE X≠ 0 DO X:=X-1
```

que sólo termina para estados iniciales en los que `X` tenga un valor no negativo; o bien el comando

```
WHILE X=X DO X:=X-1
```

que no termina para ningún estado inicial.

¿Cómo podemos pues demostrar formalmente que un bucle tipo `WHILE` termina en caso de que así sea? La idea consiste en encontrar alguna "cantidad" numérica

entera, ya sea alguna de las variables que intervenga en el programa o ya sea alguna expresión aritmética entera construida a partir de ellas, que tome solamente valores enteros no negativos y que decrezca con cada iteración del bucle `WHILE`. Diremos entonces que dicha cantidad o variable es una *variante* del comando `WHILE` correspondiente, y la denotaremos por E. Puesto que el conjunto de los enteros no negativos no posee cadenas infinitas en orden descendiente ($\mathbb{N}$ es un conjunto bien ordenado), la existencia de una tal variante para el bucle `WHILE` garantiza su terminación.

En la formulación de la regla `While`, debemos hacer explícito que la variante E toma simpre valores no negativos y usaremos una variable auxiliar n para expresar que la variante E decrece. Por otra parte, incorporamos también el concepto de *invariante* del bucle para expresar la semántica de la instrucción, tal y como se hizo en la regla `While` para la corrección parcial. Por tanto, la formulación de la regla `While` queda ahora como sigue:

Regla `WHILE`

$$\frac{\vdash [P \wedge S \wedge (E = n)]\ C\ [P \wedge (E < n)], \quad \vdash P \wedge S \Rightarrow 0 \leq E}{\vdash [P]\ \texttt{WHILE}\ S\ \texttt{DO}\ C\ [P \wedge \neg S]}$$

donde la condición P se dirá un *invariante* del bucle, la expresión E se dirá una *variante* del bucle, y n es una variable auxiliar.

3.2.5. Ejemplos de ternas demostrables: corrección total

A modo de ejemplo, daremos una prueba en la lógica de Hoare de la corrección total del programa C_5 (descrito al final de la sección (3.1.3)) para calcular el factorial de un entero no negativo `X`, Es decir, probaremos que: 1) la ejecución del programa C_5 termina para cualquier estado inicial que satisfaga `X`$\geq$`0`, y 2) para dichos estados iniciales C_5 calcula en su variable de salida `Y` el factorial de `X`.

Esto es, daremos una prueba de la siguiente terna de Hoare:

`[0` $\leq$ `X]`

$$C_5 = \left\{ \begin{array}{l} \texttt{Y:=1;} \\ \texttt{R:=0;} \\ \texttt{WHILE R}\neq\texttt{X DO (R:=R+1; Y:=Y*R)} \end{array} \right.$$

`[Y=fact(X)]`

Proponemos como invariante del bucle la condición

$$P \equiv \text{“}Y = fact(R) \wedge R \leq X\text{”}$$

y como variante del bucle la expresión

$$E = \text{“}X - R\text{”}$$

Para comprobar que, de hecho, P es un invariante y E es una variante del bucle `WHILE`, hemos de demostar que:

1) $\vdash$ [(Y=fact(R) $\wedge$ R$\leq$X) $\wedge$ R$\neq$X $\wedge$ (X-R=n)] R:=R+1; Y:=Y*R
[(Y=fact(R) $\wedge$ R$\leq$X) $\wedge$ (X-R<n)], y

2) $\vdash$ (Y=fact(R)$\wedge$R$\leq$X $\wedge$ R$\neq$X) $\Rightarrow$ 0$\leq$X-R.

La parte 2) se sigue de manera inmediata usando una axioma del dominio. Daremos ahora una prueba de la parte 1).

Paso 0. $\vdash$ [Y*R=fact(R) $\wedge$ R$\leq$X $\wedge$ X-R<n] Y:=Y*R
[Y=fact(R) $\wedge$ R$\leq$X $\wedge$ X-R<n]. (Axioma de asignación)

Paso 1. $\vdash$ [Y*(R+1)=fact(R+1) $\wedge$ R+1$\leq$X $\wedge$ X-(R+1)<n] R:=R+1
[Y*R=fact(R) $\wedge$ R$\leq$X $\wedge$ X-R<n]. (Axioma de asignación)

Paso 2. $\vdash$ [Y*(R+1)=fact(R+1) $\wedge$ R+1$\leq$X $\wedge$ X-(R+1)<n] R:=R+1; Y:=Y*R
[Y=fact(R) $\wedge$ R$\leq$X $\wedge$ X-R<n]. (Regla de secuenciación 0,1)

Paso 3. $\vdash$ (Y=fact(R) $\wedge$ R$\leq$X $\wedge$ R$\neq$X $\wedge$ X-R=n) $\Rightarrow$
(Y*(R+1)=fact(R+1) $\wedge$ R+1$\leq$X $\wedge$ X-(R+1)<n). (Aritmética)

Paso 4. $\vdash$ [Y=fact(R) $\wedge$ R$\leq$X $\wedge$ R$\neq$X $\wedge$ X-R=n] R:=R+1; Y:=Y*R
[Y=fact(R) $\wedge$ R$\leq$X $\wedge$ X-R<n]. (Reforzamiento de la precondición 2,3)

✓ Es invariante.

Paso 5. $\vdash$ [Y=fact(R) $\wedge$ R$\leq$X] WHILE R$\neq$X DO (R:=R+1; Y:=Y*R)
[Y=fact(R) $\wedge$ R$\leq$X $\wedge$ $\neg$(R$\neq$X)].
(Regla While para el invariante anterior)

Paso 6. $\vdash$ (Y=fact(R) $\wedge$ R$\leq$X $\wedge$ $\neg$(R$\neq$X)) $\Rightarrow$ Y=fact(X).
(Aritmética)

Paso 7. $\vdash$ [Y=fact(R) $\wedge$ R$\leq$X] WHILE R$\neq$X DO (R:=R+1; Y:=Y*R)
[Y=fact(X)]. (Debilitamiento de la postcondición 5,6)

Paso 8. $\vdash$ [Y=1 $\wedge$ 0$\leq$X] R:=0 [Y=fact(R) $\wedge$ R$\leq$X].
(Axioma de asignación + Aritmética + Reforzamiento de la precondición)

Paso 9. $\vdash$ `[1=1` $\wedge$ `0≤X] Y:=1 [Y=1` $\wedge$ `0≤X]`.
(Axioma de asignación)

Paso 10. $\vdash$ `[1=1` $\wedge$ `0≤X]` C_5 `[Y=fact(X)]`. (Regla de secuenciación 7,8,9)

Paso 11. $\vdash$ `0≤X` $\Rightarrow$ `(1=1` $\wedge$ `0≤X)`. (Aritmética)

Paso 12. $\vdash$ `[0≤X]` C_5 `[Y=fact(X)]`. (Reforzamiento de la precondición)

□

Por lo tanto, se concluye que la terna de Hoare `[0` $\leq$ `X]` C_5 `[Y=fact(X)]` es demostrable.

Capítulo 4

Lógica de Hoare en Isabelle/HOL

Este último capítulo es el objetivo principal del trabajo. Todo este se basa en el curso de *Lógica Matemática y Teoría de Modelos*, ([1]), y en el libro *Concrete Semántics*, ([8]). Se trata la lógica de Hoare, la cual se ha descrito en el capítulo anterior, formalizada en Isabelle/HOL. Lo que se desea en sí es realizar las pruebas de las ternas de Hoare en el demostrador, con el fin poder verificar los programas deseados. Para ello, son necesarias cada una de las acciones que se van a llevar a cabo a continuación. Se procederá en el siguiente orden dado:

- Formalización de la expresiones del lenguaje imperativo IMP:
  - Definición de las expresiones aritméticas.
  - Definición de las expresiones booleanas.
- Sintaxis del lenguaje IMP (comandos del capítulo anterior): Descripción de las instrucciones básicas.
- Descripción de una semántica que dé significado a dichas instrucciones: *semántica operacional de paso largo*.
- Construcción del sistema deductivo, las reglas de la lógica de Hoare.
- Adecuación y completitud de la corrección parcial.
- Ampliación de las reglas de la lógica: corrección total.
- Adecuación y completitud de la corrección total.

Se quiere expresar las sentencias de los programas como asertos lógicos. En Isabelle es necesaria la formalización de estas expresiones mediante la creación de algunos tipos de datos.

4.1. Expresiones aritméticas

En primer lugar, definimos un tipo para representar los nombres de variables como cadenas

```
type_synonym vname = string
```

Definición 4.1.1 *Una **expresión aritmética** es:*

- *un número entero,*
- *una variable, o*
- *la suma de dos expresiones aritméticas.*

Una expresión aritmética se representa mediante el tipo `aexp`*, definido por*

```
datatype aexp = N int | V vname | Plus aexp aexp
```

El comando `thm aexp.split` *nos muestra por casos la estructura de un dato* `aexp`*.*

La semántica, o significado, de una expresión es su valor. El valor de una expresión con variables depende de los valores de esas variables. En Isabelle, se tratan los valores como números enteros.

```
type_synonym val = int
```

Estos valores se almacenan en el estado del programa. Ya se introdujo en el capítulo anterior (sección 3.1.1) una definición de dicho concepto. En Isabelle también es necesario introducir esta noción, con más precisión que desde el punto de vista teórico. Por eso se define un tipo de dato que represente los estados. Es decir, un estado es una función cuyo dominio es el conjunto de variables que intervienen en el programa y su imagen es el conjunto de los números enteros. En otras palabras, un estado es una aplicación que asigna un valor a cada nombre de variable.

```
type_synonym state = "vname ⇒ val"
```

El valor de una expresión aritmética `e` en un estado `s`, se define por recursión en la estructura de la expresión como sigue:

- el valor de un número es él mismo,
- el valor de una variable es su valor en el estado,
- el valor de una suma se calcula de forma recursiva.

```
fun aval :: "aexp ⇒ state ⇒ val" where
"aval (N n) s = n" |
"aval (V x) s = s x" |
"aval (Plus a1 a2) s = aval a1 s + aval a2 s"
```

Veamos un par de ejemplos,

```
value "aval (Plus (V ''x'') (N 5)) (λx. if x = ''x'' then 7 else 0)"
value "aval (Plus (V ''x'') (N 5)) (λx. 0)"
```

Observación:
Para representar un estado de forma más concisa, se puede utilizar la notación `f(a:=b)` que denota la función que toma los mismos valores que `f`, excepto para `a`, cuyo valor es `b`. Es decir,

```
f(a:=b) = (λx. if x = a then b else  f x)
```

Por ejemplo,

```
((λx. 0) (''x'':= 7))(''y'':=3)
```

transforma ''x'' en 7, ''y'' en 3, y todos los demás nombres de variables en 0.

Observación 4.1.1 *Se puede usar una notación más compacta. Por lo que se establece la siguiente notación:*

```
definition null_state ("<>") where
  "null_state ≡ λx. 0"
syntax
  "_State" :: "updbinds => 'a" ("<_>")
translations
  "_State ms" == "_Update <> ms"
  "_State (_updbinds b bs)" <= "_Update (_State b) bs"
```

El ejemplo anterior expresado con la notación más compacta es

```
<''x'':= 7, ''y'' := 3>
```

Lema 4.1.1 *Las dos notaciones anteriores son equivalentes:*

```
lemma "<a := 1, b := 2> = (<> (a := 1)) (b := (2::int))"
  by (rule refl)
```

Con el fin de optimizar los programas sobre estas expresiones se establecen las siguientes simplificaciones.

4.1.1. Simplificación de constantes

Primera simplificación:

La función `asimp_const` reemplaza cada subexpresión cuyo valor sea una constante por dicha constante. Por ejemplo, la expresión

```
Plus (V ''x'') (Plus (N 3) (N 1))
```

por la expresión

```
Plus (V ''x'') (N 4)
```

Dicha función se define como sigue

```
fun asimp_const :: "aexp ⇒ aexp" where
"asimp_const (N n) = N n" |
"asimp_const (V x) = V x" |
"asimp_const (Plus a1 a2) =
  (case (asimp_const a1, asimp_const a2) of
    (N n1, N n2) ⇒ N(n1+n2) |
    (b1,b2) ⇒ Plus b1 b2)"
```

Lema 4.1.2 *La simplificación es correcta. Es decir, no cambia el valor de la expresión aritmética.*

• La demostración aplicativa es:

```
theorem aval_asimp_const:
  "aval (asimp_const a) s = aval a s"
apply(induction a)
apply (auto split: aexp.split)
done
```

Comentarios sobre la demostración anterior:

La prueba se realiza por inducción en la expresión `a`. Si `a` es un número o una variable, es trivial. Y, si `a` es de la forma `Plus a1 a2`, se distiguen dos casos, según que `a1` y `a2` sean o no expresiones numéricas.

• En la prueba automática: `split` proporciona la ayuda para que se realice la distinción de casos cuando una expresión `case` se aplica sobre un dato `aexp`.

Segunda simplificación:

Eliminación de los ceros en las sumas. Para ello, definimos la función `plus`

```
fun plus :: "aexp ⇒ aexp ⇒ aexp" where
"plus (N i1) (N i2) = N(i1+i2)" |
"plus (N i) a = (if i=0 then a else Plus (N i) a)" |
"plus a (N i) = (if i=0 then a else Plus a (N i))" |
"plus a1 a2 = Plus a1 a2"
```

Lema 4.1.3 *La función* `plus` *se comporta como el constructor de tipo* `Plus` *mediante la evaluación.*

• La demostración aplicativa es:

```
lemma aval_plus[simp]:
  "aval (plus a1 a2) s = aval a1 s + aval a2 s"
apply(induction a1 a2 rule: plus.induct)
apply simp_all
done
```

• La demostración automática es:

```
lemma "aval (plus a1 a2) s = aval a1 s + aval a2 s"
by (induct a1 a2 rule: plus.induct) simp_all
```

Simplificación total:

La función `asimp` simplifica una expresión aritmética, eliminando los ceros de las sumas y efectuando las sumas de las subexpresiones numéricas.

```
fun asimp :: "aexp ⇒ aexp" where
"asimp (N n) = N n" |
"asimp (V x) = V x" |
"asimp (Plus a1 a2) = plus (asimp a1) (asimp a2)"
```

Por ejemplo,

```
value "asimp (Plus (V ''x'')
             (Plus (Plus (N 3) (N 1))
                   (Plus (V ''y'') (N 0))))"
```

queda de la forma

```
"Plus (V ''x'') (Plus (N 4) (V ''y''))"
```

Lema 4.1.4 *La función* `asimp` *es correcta: el valor de la expresión aritmética no varía.*

• La demostración aplicativa es:

```
theorem aval_asimp[simp]:
  "aval (asimp a) s = aval a s"
apply(induction a)
apply simp_all
done
```

• La demostración automática es:

```
theorem "aval (asimp a) s = aval a s"
by (induct a) simp_all
```

Comentarios sobre la demostración anterior: La demostración es por inducción estructurada en la expresión `a`.

4.2. Expresiones booleanas

Definición 4.2.1 *Una **expresión booleana** es:*

- *una constante booleana,*
- *la negación de una expresión booleana,*
- *la conjunción de dos expresiones booleanas, o*
- *la comparación de dos expresiones aritméticas.*

Una expresión booleana se representa mediante el tipo `bexp`*, definido por*

```
datatype bexp = Bc bool | Not bexp | And bexp bexp | Less aexp aexp
```

En cuanto a la semántica, el valor de una expresión booleana en un estado se define:

```
fun bval :: "bexp ⇒ state ⇒ bool" where
"bval (Bc v) s = v" |
"bval (Not b) s = (¬ bval b s)" |
"bval (And b1 b2) s = (bval b1 s ∧ bval b2 s)" |
"bval (Less a1 a2) s = (aval a1 s < aval a2 s)"
```

Veamos un par de ejemplos:

```
value "bval (Less (V ''x'') (Plus (N 3) (V ''y'')))
            <''x'' := 3, ''y'' := 1>"

value "bval (Less (V ''x'') (Plus (N 3) (V ''y'')))
            <''x'' := 5, ''y'' := 1>"
```

Las reglas de simplificación introducidas por la definición se observan con `thm bval.simps`.

Se quiere eliminar la tercera regla y definirla de otra forma más eficiente computacionalmente. Establecemos el siguiente lema como regla de simplificación de las expresiones condicionales, con objeto de mejorar los procesos automáticos de demostración:

```
lemma bval_And_if[simp]:
  "bval (And b1 b2) s = (if bval b1 s then bval b2 s else False)"
by(simp)
```

Para eliminar la tercera regla de simplificación del proceso automático hay que ordenar `declare bval.simps(3)[simp del]`.

Esto no significa que ya no se pueda utilizar, basta con escribir `bval.simps(3)` para volver a hacer uso de ella.

Como en el caso de las expresiones aritméticas, aquí también es conveniente realizar algunas optimizaciones.

4.2.1. Simplificación de constantes

Primera simplificación:

La función `less` simplificará las expresiones de la forma

```
"Less (N n1) (N n2)" a "Bc (n1 < n2)".
```

```
fun less :: "aexp ⇒ aexp ⇒ bexp" where
"less (N n1) (N n2) = Bc(n1 < n2)" |
"less a1 a2 = Less a1 a2"
```

Lema 4.2.1 *La función* `less` *es correcta. El valor de la expresión* `less a1 a2` *en un estado* `s` *coincide con el valor de* `aval a1 s < aval a2 s`.

- La demostración aplicativa es:

```
lemma [simp]: "bval (less a1 a2) s = (aval a1 s < aval a2 s)"
apply(induction a1 a2 rule: less.induct)
apply simp_all
done
```

Segunda simplificación:

Definimos una función `and` que simplificará las expresiones booleanas conjuntivas, una de cuyas componentes sea una constante booleana.

```
fun "and" :: "bexp ⇒ bexp ⇒ bexp" where
"and (Bc True) b = b" |
"and b (Bc True) = b" |
"and (Bc False) b = Bc False" |
"and b (Bc False) = Bc False" |
"and b1 b2 = And b1 b2"
```

Lema 4.2.2 *La función* `and` *es correcta.*

- La demostración aplicativa es:

```
lemma bval_and[simp]: "bval (and b1 b2) s = (bval b1 s ∧ bval b2 s)"
apply(induction b1 b2 rule: and.induct)
apply simp_all
done
```

Tercera simplificación:

Definimos una función `not` que simplificará las expresiones booleanas negativas de constantes booleanas.

```
fun not :: "bexp ⇒ bexp" where
"not (Bc True) = Bc False" |
"not (Bc False) = Bc True" |
"not b = Not b"
```

Lema 4.2.3 *La función* `not` *es correcta.*

- La demostración aplicativa es:

```
lemma bval_not[simp]: "bval (not b) s = (¬ bval b s)"
apply(induction b rule: not.induct)
apply simp_all
done
```

Simplificación total:

La función `bsimp` simplifica las expresiones booleanas, mediante las funciones previas.

```
fun bsimp :: "bexp ⇒ bexp" where
"bsimp (Bc v) = Bc v" |
"bsimp (Not b) = not (bsimp b)" |
"bsimp (And b1 b2) = and (bsimp b1) (bsimp b2)" |
"bsimp (Less a1 a2) = less (asimp a1) (asimp a2)"
```

Lema 4.2.4 *La función* `bsimp` *es correcta: el valor de la expresión booleana simplificada coincide con el valor de la expresión inicial, en cualquier estado.*

- La demostración aplicativa es:

```
theorem "bval (bsimp b) s = bval b s"
apply(induction b)
apply simp_all
done
```

A continuación se muestra un par de ejemplos

```
value "bsimp (And (Less (N 0) (N 1)) b)"
```

se reemplaza por `"bsimp b"`,

```
value "bsimp (And (Less (N 1) (N 0)) (Bc True))"
```

se reemplaza por `"Bc False"`.

4.3. Sintaxis del lenguaje imperativo simple IMP

En la sección (3.1.1) se describen cuáles son los comandos que forman el lenguaje. Se muestra qué notación se utiliza (sintaxis) y qué significado se le da a esta notación (semántica), es decir, explicaciones teóricas que se pueden deducir o intuir. En Isabelle/HOL es necesario precisar dicha sintaxis y semántica. Para la sintaxis se define un nuevo tipo de dato, que se explica a continuación, y para la semántica se construye un predicado ternario, como se muestra más adelante.

En cuanto a la sintaxis, para especificar la notación elegida definimos un lenguaje imperativo con las instrucciones básicas del lenguaje: asignación, composición secuencial, condicional, `WHILE` y `SKIP` (para representar una instrucción sin ninguna acción a realizar).

Instrucciones del lenguaje:

- `Assign x e`: asignar la expresión aritmética `e` a la variable `x`.
- `Seq c1 c2`: realizar la instrucción `c1` y, a continuación, `c2`.
- `If b c1 c2`: si la expresión booleana `b` es verdadera, realizar `c1`; en caso contrario, realizar `c2`.
- `While b c`: mientras la expresión booleana `b` sea verdadera, realizar `c`.
- `SKIP`: no hacer nada.

Representamos el lenguaje mediante el siguiente tipo de dato:

```
datatype
  com = SKIP
      | Assign vname aexp     ("_ ::= _" [1000, 61] 61)
      | Seq    com  com       ("_;;/ _"  [60, 61] 60)
      | If     bexp com com   ("(IF _/ THEN _/ ELSE _)"  [0, 0, 61] 61)
      | While  bexp com       ("(WHILE _/ DO _)"  [0, 61] 61)
```

Comentarios acerca de las instrucciones:

- Escribiremos `Assign x a` como `x ::= e`.
- Escribiremos `Seq c1 c2` como `c1 ;; c2`.
- Escribiremos `If b c1 c2` como `IF b THEN c1 ELSE c2`.
- Escribiremos `While b c` como `WHILE b DO c`.
- La instrucción `;;` asocia por la izquierda. Es decir,
  `c1 ;; c2 ;; c3` es `(c1 ;; c2) ;; c3`
- Las instrucciones `IF` y `WHILE` tienen prelación sobre `;;`. Es decir,
  `WHILE b DO c1 ;; c2` es `(WHILE b DO c1) ;; c2`.

4.4. Semántica operacional del lenguaje imperativo simple IMP

En esta sección se define una *semántica operacional* para dar una interpretación a cada instrucción del lenguaje IMP. Esto es, el significado exacto que tendrá cada comando de la sección (3.1.1). La idea es que la semántica describa el proceso de ejecución de un programa, en lugar de centrarse en los resultados. Se ha elegido una *semántica operacional natural* o *de paso largo*, así se simplifica la notación y se ocultan los detalles, dándose un paso más grande en la computación.

4.4.1. Semántica de paso largo

La *semántica de paso largo* establece una relación entre el programa, el estado inicial y el estado final. Los estados intermedios no se tienen en cuenta en dicha relación. Aunque a través de las reglas inductivas que van a definir la semántica se puede ver cómo es la ejecución del programa, la relación propiamente dicha sólo muestra como si el programa se ejecutara en un único paso, *paso largo*.

Formalizamos la semántica mediante un predicado ternario `big_step`, tal que `big_step c s t` significa que la ejecución de la instrucción `c`, empezando en el estado `s`, termina en el estado `t`.

Nota:
Usaremos la notación `(c,s) ⇒ t`, en vez de `big_step c s t`.

Semántica del lenguaje:

```
inductive
  big_step :: "com × state ⇒ state ⇒ bool" (infix "⇒" 55)
where
  Skip: "(SKIP,s) ⇒ s"
| Assign: "(x ::= a,s) ⇒ s(x := aval a s)"
| Seq: "⟦ (c1,s1) ⇒ s2;  (c2,s2) ⇒ s3 ⟧ ⟹ (c1;;c2, s1) ⇒ s3"
| IfTrue: "⟦ bval b s;  (c1,s) ⇒ t ⟧ ⟹
          (IF b THEN c1 ELSE c2, s) ⇒ t"
| IfFalse: "⟦ ¬bval b s;  (c2,s) ⇒ t ⟧ ⟹
          (IF b THEN c1 ELSE c2, s) ⇒ t"
| WhileFalse: "¬bval b s ⟹ (WHILE b DO c,s) ⇒ s"
| WhileTrue: "⟦ bval b s1;  (c,s1) ⇒ s2;  (WHILE b DO c, s2) ⇒ s3 ⟧
              ⟹ (WHILE b DO c, s1) ⇒ s3"
```

Descripción de la semántica de cada instrucción:

- Si la instrucción es `SKIP`, el estado inicial y el final han de ser el mismo.
- Si la instrucción es `x::= a` y el estado inicial es `s`, entonces el estado final es el estado `s` en el que el valor de la variable `x` es el valor de `a` en `s`.

- Si la instrucción es `c1;;c2` y el estado inicial es `s1`, entonces el estado final es `s3` si existe un estado intermedio `s2` tal que `c1` termina en `s2`, empezando en `s1`, y `c2` termina en `s3`, empezando en `s2`.
- La instrucción condicional `If b c1 c2` se corresponde con la descrita en (3.2.1). Para darle significado en Isabelle se ha dividido en dos reglas, dependiendo del valor de `b` en el estado inicial `s`:
  - Si es `True`, entonces el estado final es el estado final de ejecutar `c1` empezando en `s`.
  - Si es `False`, entonces el estado final es el estado final de ejecutar `c2` empezando en `s`.
- La instrucción `While b c` se corresponde con la descrita en (3.2.1). Igual que en el caso previo, esta instrucción también se ha dividido en dos reglas, dependiendo del valor de `b` en el estado inicial `s`:
  - Si es `False`, entonces el estado final es `s`.
  - Si es `True` en un estado inicial `s1`, entonces el estado final será un estado `s3` si existe un estado intermedio `s2` tal que:
    - la instrucción `c` termina en `s2` empezando en `s1`, y
    - el bucle `While b c` termina en `s3`, empezando en `s2`.

Ejemplo.

Sea el programa `P`:

```
P = ''x''::= N 5 ;; ''y::= V ''x''.
```

Si ejecutamos `P` en un estado inicial `s`, el estado final será el mismo estado `s` en el que `x` e `y` valen 5.

Veamos cómo usar Isabelle para seguir la ejecución del programa.

```
schematic_lemma ex: "(''x'' ::= N 5;; ''y'' ::= V ''x'', s) ⇒ ?t"
apply(rule Seq)
apply(rule Assign)
apply simp
apply(rule Assign)
done
```

Una vez finalizada la prueba, Isabelle instancia el lema con el comando `thm ex[simplified]` y, después de simplificarlo, se obtiene

```
(''x'' ::= N 5;; ''y'' ::= V ''x'', ?s) ⇒ ?s(''x'' := 5, ''y'' := 5)
```

Otra forma mejor de ejecutar simbólicamente los programas IMP es usando el generador de código de Isabelle `code_pred`, y usando el comando `values`, análogo a `value` pero que se puede aplicar a definiciones inductivas.

```
code_pred big_step .

values "{t. (SKIP, λ_. 0) ⇒ t}"
```

El código generado se puede ver mediante el comando `thm big_step.equation`.

En los siguientes ejemplos, se calcula el valor de una lista de variables en el estado final de la ejecución de un programa.

```
values "{map t [''x''] |t. (SKIP, <''x'' := 42>) ⇒ t}"

values "{map t [''x''] |t. (''x'' ::= N 2, <''x'' := 42>) ⇒ t}"

values "{map t [''x'',''y''] |t.
  (WHILE Less (V ''x'') (V ''y'') DO (''x'' ::= Plus (V ''x'') (N 5)),
   <''x'' := 0, ''y'' := 13>) ⇒ t}"
```

Automatización de las demostraciones:
Con el comando `declare big_step.intros [intro]` declaramos las reglas `big_step.intros` como reglas de introducción, con objeto de automatizar la ejecución de programas pequeños.

Nota:
Como la definición de `big_step` es inductiva, las demostraciones suelen hacerse por inducción en la estructura de la definición. Si escribimos `thm big_step.induct` podemos observar tal esquema de inducción.

```
?x1.0 ⇒ ?x2.0 ⟹

(⋀s. ?P (SKIP, s) s) ⟹

(⋀x a s. ?P (x ::= a, s) (s(x := aval a s))) ⟹

(⋀c1 s1 s2 c2 s3.
    (c1, s1) ⇒ s2 ⟹
    ?P (c1, s1) s2 ⟹
    (c2, s2) ⇒ s3 ⟹ ?P (c2, s2) s3 ⟹ ?P (c1;; c2, s1) s3) ⟹

(⋀b s c1 t c2.
    bval b s ⟹ (c1, s) ⇒ t ⟹ ?P (c1, s) t ⟹
    ?P (IF b THEN c1 ELSE c2, s) t) ⟹

(⋀b s c2 t c1.
    ¬ bval b s ⟹ (c2, s) ⇒ t ⟹ ?P (c2, s) t ⟹
    ?P (IF b THEN c1 ELSE c2, s) t) ⟹

(⋀b s c. ¬ bval b s ⟹ ?P (WHILE b DO c, s) s) ⟹
```

```
(⋀b s1 c s2 s3.
    bval b s1 ⟹
    (c, s1) ⇒ s2 ⟹
    ?P (c, s1) s2 ⟹
    (WHILE b DO c, s2) ⇒ s3 ⟹
    ?P (WHILE b DO c, s2) s3 ⟹ ?P (WHILE b DO c, s1) s3)⟹

?P ?x1.0 ?x2.0
```

Comentarios:
El esquema de inducción tiene dos variables, `?x1.0` para representar la instrucción y `?x2.0` para el estado inicial, en vez de las tres que aparecerán en las pruebas: `s`, `c` y `t`.

Con objeto de modificar la forma del esquema de inducción, introducimos el siguiente lema:

```
lemmas big_step_induct = big_step.induct[split_format(complete)]
```

Y observamos la nueva forma del esquema de inducción, ahora con las tres variables, que representan la instrucción, el estado inicial y el estado final. Si escribimos `thm big_step_induct` podremos observar tal esquema:

```
(?x1a, ?x1b) ⇒ ?x2a ⟹

(⋀s. ?P SKIP s s) ⟹

(⋀x a s. ?P (x ::= a) s (s(x := aval a s))) ⟹

(⋀c1 s1 s2 c2 s3.
    (c1, s1) ⇒ s2 ⟹
    ?P c1 s1 s2 ⟹ (c2, s2) ⇒ s3 ⟹ ?P c2 s2 s3 ⟹
    ?P (c1;; c2) s1 s3) ⟹

(⋀b s c1 t c2.
    bval b s ⟹ (c1, s) ⇒ t ⟹ ?P c1 s t ⟹
    ?P (IF b THEN c1 ELSE c2) s t) ⟹

(⋀b s c2 t c1.
    ¬ bval b s ⟹ (c2, s) ⇒ t ⟹ ?P c2 s t ⟹
    ?P (IF b THEN c1 ELSE c2) s t) ⟹

(⋀b s c. ¬ bval b s ⟹ ?P (WHILE b DO c) s s) ⟹

(⋀b s1 c s2 s3.
    bval b s1 ⟹
    (c, s1) ⇒ s2 ⟹
```

```
    ?P c s1 s2 ⟹
    (WHILE b DO c, s2) ⇒ s3 ⟹ ?P (WHILE b DO c) s2 s3 ⟹
  ?P (WHILE b DO c) s1 s3)⟹

?P ?x1a ?x1b ?x2a
```

4.4.2. Regla de inversión

Las reglas de inversión asociadas a la definición inductiva del predicado `big_step` determinan qué es lo que se puede inferir a partir de `(c,s) ⇒ t`. Por ejemplo, si tenemos `(SKIP,s) ⇒ t`, está claro que podemos inferir que `t = s`.

Para cada una de las reglas, tenemos lo siguiente:

```
(SKIP, s) ⇒ t  ⟹  t  = s

(x::=a, s) ⇒ t  ⟹ t = s(x:= aval a s)

(c1;;c2, s) ⇒ t  ⟹  ∃ s'. (c1,s) ⇒ s' ∧  (c2,s')⇒ t

(IF b THEN c1 ELSE c2, s) ⇒ t  ⟹
  (bval b s ∧ (c1,s) ⇒ t) ∨ (¬ bval b s ∧ (c2,s) ⇒ t)

(WHILE b DO c, s) ⇒ t  ⟹
 (¬bval b s ∧ t = s) ∨
 (∃s'. (c,s) ⇒ s' ∧ (WHILE b DO c, s') ⇒ t)
```

Podemos decirle a Isabelle que genere de forma automática reglas de inversión para cada conclusión, usando el comando `inductive_cases`.

Además, todas ellas se pueden usar en el demostrador como reglas de eliminación, de forma segura. Por ello, las etiquetamos con el atributo `elim!`, con objeto de que se puedan usar de forma automática mediante `auto` y `blast`.

```
inductive_cases SkipE[elim!]: "(SKIP,s) ⇒ t"
thm SkipE

inductive_cases AssignE[elim!]: "(x ::= a,s) ⇒ t"
thm AssignE

inductive_cases SeqE[elim!]: "(c1;;c2,s1) ⇒ s3"
thm SeqE

inductive_cases IfE[elim!]: "(IF b THEN c1 ELSE c2,s) ⇒ t"
thm IfE

inductive_cases WhileE[elim]: "(WHILE b DO c,s) ⇒ t"
thm WhileE
```

Comentarios:
En el último caso sólo lo etiquetamos con el atributo `elim`, pues `elim!` podría no terminar.

Veamos un ejemplo de prueba automática, usando las reglas de eliminación.

```
lemma "(IF b THEN SKIP ELSE SKIP, s) ⇒ t  ⟹  t = s"
by blast
```

La misma prueba detallada, mediante la distinción de casos, es:

```
lemma assumes "(IF b THEN SKIP ELSE SKIP, s) ⇒ t"
shows "t = s"
proof-
  from assms show ?thesis
  proof cases
    case IfTrue
    thus ?thesis by (blast intro: IfTrue)
  next
    case IfFalse thus ?thesis by blast
  qed
qed
```

Las reglas de inversión se pueden combinar con las reglas originales para obtener equivalencias mejor que implicaciones. Veamos algunos ejemplos:

```
lemma assign_simp:
  "(x ::= a,s) ⇒ s' ⟷ (s' = s(x := aval a s))"
  by auto

lemma "(IF b THEN c1 ELSE c2, s) ⇒ t ⟷
  (bval b s ∧ (c1,s) ⇒ t) ∨ (¬ bval b s ∧ (c2,s) ⇒ t)"
by auto
```

En la prueba siguiente se combina el uso de la regla de inversión y la de derivación.

```
lemma Seq_assoc:
  "(c1;; c2;; c3, s) ⇒ s' ⟷ (c1;; (c2;; c3), s) ⇒ s'"
proof
  assume "(c1;; c2;; c3, s) ⇒ s'"
  then obtain s1 s2 where
    c1: "(c1, s) ⇒ s1" and
    c2: "(c2, s1) ⇒ s2" and
    c3: "(c3, s2) ⇒ s'" by auto
  from c2 c3
  have "(c2;; c3, s1) ⇒ s'" by (rule Seq)
  with c1
  show "(c1;; (c2;; c3), s) ⇒ s'" by (rule Seq)
```

```
next
  -- "Análogamente, la otra implicación"
  assume "(c1;; (c2;; c3), s) ⇒ s'"
  thus "(c1;; c2;; c3, s) ⇒ s'" by auto
qed
```

4.4.3. Equivalencia de instrucciones

Decimos que dos **instrucciones** `C` y `C'` son **equivalentes** con respecto a la semántica de paso largo si para cualesquiera estados `s` y `t` se verifica lo siguiente: `C` empezando en `s` termina en `t` syss `C'` empezando en `s` termina en `t`.

La definición formal en Isabelle es:

```
abbreviation
  equiv_c :: "com ⇒ com ⇒ bool" (infix "∼" 50) where
  "c ∼ c' ≡ (∀s t. (c,s) ⇒ t  = (c',s) ⇒ t)"
```

Observación:
En la construcción de la definición anterior se podría haber utilizado el comando `definition` en lugar de `abbreviation`. La diferencia está en que las definiciones construidas con el primer comando se pueden utilizar como reglas de simplificación, pero hay que especificárselo al sistema, no se incorporan por defecto. Las definiciones construidas con `abbreviation` no son más que *azúcar sintáctico*, es decir, reescrituras de expresiones cuya finalidad es facilitar tales construcciones, o expresarlas de otra manera.
El comando `definition` se suele utilizar para elaborar conceptos más abstractos, frente a `abbreviation`.

Ejemplo

Sea el siguiente ejemplo de desplegado de la instrucción `WHILE`:

Demostramos que la instrucción `WHILE` b `DO` c es equivalente a la instrucción `IF` b `THEN` c`;; WHILE` b `DO` c `ELSE SKIP`.

• La prueba detallada es

```
lemma unfold_while: "(WHILE b DO c) ∼
            (IF b THEN c;; WHILE b DO c ELSE SKIP)" (is "?w ∼ ?iw")
proof -
  -- "para probar la equivalencia, desarrollamos el árbol de "
  -- "computación de un lado y, a partir de él, construimos"
  -- "el árbol de computación del otro"
  { fix s t assume "(?w, s) ⇒ t"
   -- "en primer lugar, hay que tener en cuenta que si b es falso"
   -- "en el estado s, ambas instrucciones no hacen nada"
```

```
  { assume "¬bval b s"
    hence "t = s" using ‘(?w,s) ⇒ t‘ by blast
    hence "(?iw, s) ⇒ t" using ‘¬bval b s‘ by blast
  }
  moreover
  -- "por otra parte, si b es cierto en el estado s, entonces"
  -- "sólo se puede usar la regla WhileTrue para computar"
  -- "(?w, s) ⇒ t"
  { assume "bval b s"
    with ‘(?w, s) ⇒ t‘ obtain s' where
      "(c, s) ⇒ s'" and "(?w, s') ⇒ t" by auto
    -- "ahora podemos construir el árbol de computación para"
    -- "la rama verdadera de IF"
    hence "(c;; ?w, s) ⇒ t" by (rule Seq)
    -- "por tanto"
    with ‘bval b s‘ have "(?iw, s) ⇒ t" by (rule IfTrue)
  }
  ultimately
  -- "finalmente, con ambos casos"
  have "(?iw, s) ⇒ t" by blast
}
moreover
-- "ahora, en la otra dirección"
{ fix s t assume "(?iw, s) ⇒ t"
  -- "de nuevo, si b es falso en el estado s, se ejecuta"
  -- "la rama falsa de IF; ambas instrucciones no hacen nada:"
  { assume "¬bval b s"
    hence "s = t" using ‘(?iw, s) ⇒ t‘ by blast
    hence "(?w, s) ⇒ t" using ‘¬bval b s‘ by blast
  }
  moreover
  -- "por otra parte, si b es cierta en el estado s, entonces"
  -- "sólo se puede aplicar IfTrue"
  { assume "bval b s"
    with ‘(?iw, s) ⇒ t‘ have "(c;; ?w, s) ⇒ t" by auto
    -- "y por tanto, sólo se aplica la regla Seq"
    then obtain s' where
      "(c, s) ⇒ s'" and "(?w, s') ⇒ t" by auto
    -- "con esta información, podemos construir un árbol"
    -- "de computación para WHILE"
    with ‘bval b s‘
    have "(?w, s) ⇒ t" by (rule WhileTrue)
  }
  ultimately
  -- "por último, de ambos casos:"
  have "(?w, s) ⇒ t" by blast
```

```
  }
  ultimately
  show ?thesis by blast
qed
```

- La prueba automática es

```
lemma while_unfold:
"(WHILE b DO c) ∼ (IF b THEN c;; WHILE b DO c ELSE SKIP)"
by blast
```

Otras propiedades:

Contracción de la instrucción condicional:

```
lemma triv_if: "(IF b THEN c ELSE c) ∼ c"
by blast
```

Conmutatividad de la instrucción condicional:

```
lemma commute_if:
  "(IF b1 THEN (IF b2 THEN c11 ELSE c12) ELSE c2)∼
   (IF b2 THEN (IF b1 THEN c11 ELSE c2) ELSE (IF b1 THEN c12 ELSE c2))"
by blast
```

No todas las propiedades de equivalencia son triviales. Por ejemplo, para demostrar esta propiedad

```
lemma sim_while_cong: "c ∼ c' ⟹ WHILE b DO c ∼ WHILE b DO c'"
```

ha sido necesario construir el siguiente lema auxiliar

```
lemma sim_while_cong_aux:
  "(WHILE b DO c,s) ⇒ t  ⟹ c ∼ c' ⟹  (WHILE b DO c',s) ⇒ t"
apply(induction "WHILE b DO c" s t arbitrary: b c rule: big_step_induct)
apply blast
apply blast
done
```

Por lo tanto, en la demostración de la propiedad, mediante el uso de `Sledgehammer`, se ha recurrido al lema auxiliar anterior.

```
lemma sim_while_cong: "c ∼ c' ⟹ WHILE b DO c ∼ WHILE b DO c'"
by (metis sim_while_cong_aux) (* uso de Sledgehammer *)
```

Es decir, para poder probar que si dos instrucciones `c` y `c'` son equivalentes entonces también se cumple que `WHILE b DO c` es equivalente a `WHILE b DO c'` ha sido necesario el lema auxiliar anterior.

Nota:

`Sledgehammer` es una parte de Isabelle/HOL que permite llamar a un número de demostradores de teoremas automáticos externos para asistir en la búsqueda y construcción de una prueba interactiva de cualquier lema dado. La ejecución de estos demostradores dura 30 segundos.

Definición 4.4.1 (Relación de equivalencia) *Una relación $\mathcal{R}$ se dice que es una* ***relación de equivalencia*** *si y sólo si es*

reflexiva: $\forall\, x.\ \mathcal{R}\ x\ x$

simétrica: $\forall\, x\ y.\ \mathcal{R}\ x\ y \longrightarrow \mathcal{R}\ y\ x$

transitiva: $\forall\, x\ y\ z.\ \mathcal{R}\ x\ y \longrightarrow \mathcal{R}\ y\ z \longrightarrow \mathcal{R}\ x\ z$

Lema 4.4.1 *La relación $\sim$ es una relación de equivalencia.*

Isabelle realiza las pruebas de este lema de forma automática.

```
lemma sim_refl:  "c ∼ c" by simp
lemma sim_sym:   "(c ∼ c') = (c' ∼ c)" by auto
lemma sim_trans: "c ∼ c' ⟹ c' ∼ c'' ⟹ c ∼ c''" by auto
```

A esta relación también se le llama *congruencia*.

4.4.4. IMP es determinista

Definición 4.4.2 *Un* ***lenguaje*** *es* ***determinista*** *si dos ejecuciones de las misma instrucción sobre el mismo estado inicial siempre produce el mismo estado final. De lo contrario se dice que es un* ***lenguaje no determinista****.*

Una vez definida la semántica del lenguaje como una relación, no es inmediatamente obvio que la ejecución del lenguaje sea determinista o no. El siguiente lema establece esto en Isabelle.

Lema 4.4.2 *IMP es determinista.*

• Prueba aplicativa, por inducción en la estructura de `big_step`, con `t'` arbitrario.

```
theorem big_step_determ: "⟦ (c,s) ⇒ t; (c,s) ⇒ t' ⟧ ⟹ t' = t"
apply (induction arbitrary: t' rule: big_step.induct)
apply blast+
done
```

• Demostración automática, por inducción en la estructura de `big_step`, con `t'` arbitrario.

```
theorem big_step_determ: "⟦ (c,s) ⇒ t; (c,s) ⇒ t' ⟧ ⟹ t' = t"
by (induction arbitrary: t' rule: big_step.induct) blast+
```

• En la siguiente demostración, sólo detallamos los casos significativos, dejando los demás de forma automática.

```
theorem
  "(c,s) ⇒ t  ⟹  (c,s) ⇒ t'  ⟹  t' = t"
proof (induction arbitrary: t' rule: big_step.induct)
  -- "el único caso interesante es WhileTrue:"
  fix b c s s1 t t'
  -- "las hipótesis de la regla:"
  assume "bval b s" and "(c,s) ⇒ s1" and "(WHILE b DO c,s1) ⇒ t"
  -- {* H.I.: nótese ⋀ por el uso de arbitrary: *}
  assume IHc: "⋀t'. (c,s) ⇒ t' ⟹ t' = s1"
  assume IHw: "⋀t'. (WHILE b DO c,s1) ⇒ t' ⟹ t' = t"
  -- "premisa de la implicación:"
  assume "(WHILE b DO c,s) ⇒ t'"
  with ‘bval b s‘ obtain s1' where
      c: "(c,s) ⇒ s1'" and
      w: "(WHILE b DO c,s1') ⇒ t'"
    by auto
  from c IHc have "s1' = s1" by blast
  with w IHw show "t' = t" by blast
qed blast+ -- "el resto se prueba de forma automática"
```

4.4.5. Ejemplos de funciones sobre programas

En este apartado se muestran ejemplos de funciones sobre programas, así como algunas propiedades de las mismas.

Ejemplo 1

Se define la función `assigned`, que obtiene el conjunto de las variables asignadas en un programa `c`.

```
fun assigned :: "com ⇒ vname set" where
  "assigned (SKIP) = {}"|
  "assigned (Assign v a) = {v}"|
  "assigned (Seq c1 c2) = (assigned c1)∪(assigned c2)"|
  "assigned (If a c1 c2) = (assigned c1)∪(assigned c2)"|
  "assigned (While b c) = (assigned c)"
```

Lema 4.4.3 *Si una variable no está asignada en un programa* `c`*, dicha variable nunca se modifica mediante* `c`*.*

• Demostración aplicativa:

```
lemma "⟦ (c, s) ⇒ t; x ∉ assigned c ⟧ ⟹ s x = t x"
apply(induction rule: big_step_induct)
apply auto
done
```

Ejemplo 2

La función `skip` comprueba si el programa `c` realiza lo mismo que la instrucción `SKIP`.

```
fun skip :: "com ⇒ bool" where
  "skip (SKIP) = True"|
  "skip (Assign x a) = (∀s. (x ::= a,s) ⇒ s)"|
  "skip (Seq c1 c2) = conj(skip c1)(skip c2)"|
  "skip (If a c1 c2) = conj(skip c1)(skip c2)"|
  "skip (While b c) = (∀s. (¬ bval b s) )"
```

Lema 4.4.4 *La función* `skip` *es correcta.*

- Demostración estructurada para la cuál se han necesitado dos lemas auxiliares:

```
lemma skip_1: "skip c ⟹ (∀s.(c,s) ⇒ s)"
by (induct) auto

lemma skip_2 : "(∀s t. (SKIP,s) ⇒ t ⟷ s = t)"
by auto

lemma "skip c ⟹ c ∼ SKIP"
proof (induction c)
  case SKIP thus ?case by simp
next
  case Assign thus ?case using skip_2 by auto
next
  case (Seq a b)
  hence "a ∼ SKIP" and "b ∼ SKIP" by auto
  moreover hence "(a;; b) ∼ (SKIP;; SKIP)" by blast
  moreover hence "(SKIP;; SKIP) ∼ SKIP" by auto
  ultimately show ?case by auto
next
  case (If a b c)
  hence "b ∼ SKIP" and "c ∼ SKIP" by auto
  thus ?case by blast
next
  case While thus ?case by auto
qed
```

Ejemplo 3

La función `deskip` elimina de un programa `c` todas las instruciones `SKIP` posibles.

```
fun deskip :: "com ⇒ com" where
"deskip SKIP = SKIP"|
"deskip (Assign x a) = Assign x a"|
"deskip (Seq c1 c2) = (if(deskip c1)=SKIP then deskip c2 else if
                      (deskip c2)=SKIP then deskip c1 else
                      Seq (deskip c1) (deskip c2))" |
"deskip (If b c1 c2) = ( If b (deskip c1)(deskip c2))"|
"deskip (While b c) = (While b (deskip c))"
```

Lema 4.4.5 *La función* `deskip` *es correcta. Es decir,* `c ∼ (deskip c)`.

- Demostración detallada:

```
lemma "deskip c ∼ c"
proof (induction c)
  case SKIP show ?case by simp
next
  case Assign show ?case by simp
next
  case (Seq a b)
  hence "a;; b ∼ deskip a;; deskip b" by auto
  moreover have "deskip (a;; b) ∼ deskip a;; deskip b" by auto
  ultimately show ?case by auto
next
  case If thus ?case by auto
next
  case While
  thus ?case by (simp add: sim_while_cong)
qed
```

Ejemplo 4

Se define una nueva instrucción `Or` como sigue

```
definition Or :: "bexp ⇒ bexp ⇒ bexp" where
"Or b1 b2 = Not (And (Not b1) (Not b2))"
```

Probar los siguientes lemas:

- Lema 1:

```
lemma While_end: "(WHILE b DO c, s) ⇒ t ⟹ ¬bval b t"
proof(induction "WHILE b DO c" s t rule: big_step_induct)
  case WhileFalse thus ?case .
next
  case WhileTrue show ?case by fact
qed
```

◦ Lema 2:

```
lemma "WHILE Or b1 b2 DO c ∼
          WHILE Or b1 b2 DO c;; WHILE b1 DO c"
proof -
  {
    fix s
    assume "¬bval (Or b1 b2) s"
    hence "¬bval b1 s" by (auto simp add: Or_def)
  }
  then show ?thesis by (blast intro!: While_end)
qed
```

Ejemplo 5

La siguiente función `Do` define la nueva instrucción `DO c WHILE b`, tal que la instrucción `c` se ejecuta una vez antes de comprobar si la expresión `b` es cierta en un estado.

```
fun Do :: "com ⇒ bexp ⇒ com" ("DO _ WHILE _"  [0, 61] 61) where
  "Do c b = Seq c (While b c)"
```

Ejemplo 6

La función `dewhile` realiza una traducción de las instrucciones, reemplazando las instrucciones de la forma `WHILE b DO c` por instrucciones de la forma `DO c WHILE b`, manteniendo la semántica.

```
fun dewhile :: "com ⇒ com" where
"dewhile SKIP = SKIP" |
"dewhile (a ::= b) = a ::= b" |
"dewhile (a;; b) = dewhile a;; dewhile b" |
"dewhile (IF a THEN b ELSE c) = IF a THEN dewhile b ELSE dewhile c" |
"dewhile (WHILE a DO b) = If a (Do b a) SKIP"
```

Lema 4.4.6 *La traducción conserva la semántica.*

```
lemma "dewhile c ∼ c"
proof (induction c)
  case SKIP thus ?case by simp
  next case Assign thus ?case by simp
  next case Seq thus ?case by auto
  next case If thus ?case by auto
  next
    case (While b c)
    hence "WHILE b DO c ∼ WHILE b DO dewhile c" using sim_while_cong
     by simp
    thus ?case using Do_def while_unfold by auto
qed
```

4.5. Lógica de Hoare en Isabelle/HOL

Como se explicó en el capítulo (3), la lógica de Hoare es una lógica para probar propiedades de programas imperativos. Las fórmulas de esta lógica son ternas de la forma

$$\{P\}\ C\ \{Q\}.$$

El significado es el siguiente: si la fórmula P es cierta antes de ejecutar la instrucción C, entonces la fórmula Q es cierta después de ejecutar C.

4.5.1. Corrección parcial

Recordatorio: Sección (3.1.2).

Las pre y postcondiciones de una terna de Hoare, también denominadas asertos, se pueden representar bien como objetos sintácticos concretos (como las expresiones booleanas), o bien como predicados sobre estados. Elegimos esta segunda representación porque es más adecuada para el razonamiento automático.

Formalizamos los asertos (`assn`) como predicados sobre estados.

```
type_synonym assn = "state ⇒ bool"
```

Veracidad de una terna:

Como se explicó en el capítulo anterior, si la terna $\{P\}\ C\ \{Q\}$ es verdadera escribiremos $\models \{P\}\ C\ \{Q\}$.

Definimos la veracidad de una terna formalmente.

```
definition
hoare_valid :: "assn ⇒ com ⇒ assn ⇒ bool"
                          ("⊨ {(1_)}/ (_)/ {(1_)}" 50) where
"⊨ {P}c{Q} = (∀s t. P s ∧ (c,s) ⇒ t  ⟶  Q t)"
```

Notación:

El estado `s` en el que el valor de la variable `x` es el valor de la expresión `a` en `s`, `s(x := aval a s)`. En el capítulo anterior denotamos esto como `s[a/x]`. Formalizamos en Isabelle/HOL esta simplificación.

```
abbreviation state_subst :: "state ⇒ aexp ⇒ vname ⇒ state"
  ("_[_'/_]" [1000,0,0] 999)
where "s[a/x] == s(x := aval a s)"
```

Sistema deductivo:

Hemos definido cuando una terna de Hoare es verdadera. Ahora presentamos el conjunto de reglas de inferencia o sistema deductivo que se utiliza para obtener o demostrar ternas de Hoare, que se corresponde con los axiomas (3.2.1), (3.2.1), (3.2.1), (3.2.1) y (3.2.1).

```
inductive
  hoare :: "assn ⇒ com ⇒ assn ⇒ bool" ("⊢ ({(1_)}/ (_)/ {(1_)})" 50)
where
Skip: "⊢ {P} SKIP {P}"  |

Assign:  "⊢ {λs. P(s[a/x])} x::=a {P}"  |

Seq: "⟦ ⊢ {P} c1 {Q};  ⊢ {Q} c2 {R} ⟧
      ⟹ ⊢ {P} c1;;c2 {R}"  |

If: "⟦ ⊢ {λs. P s ∧ bval b s} c1 {Q};
                            ⊢ {λs. P s ∧ ¬ bval b s} c2 {Q} ⟧
     ⟹ ⊢ {P} IF b THEN c1 ELSE c2 {Q}"  |

While: "⊢ {λs. P s ∧ bval b s} c {P} ⟹
        ⊢ {P} WHILE b DO c {λs. P s ∧ ¬ bval b s}"  |

conseq: "⟦ ∀s. P' s ⟶ P s;  ⊢ {P} c {Q};  ∀s. Q s ⟶ Q' s ⟧
        ⟹ ⊢ {P'} c {Q'}"
```

Para añadir reglas de simplificación asociadas a la definición se declaran los siguientes lemas.

```
lemmas [simp] = hoare.Skip hoare.Assign hoare.Seq hoare.If

lemmas [intro!] = hoare.Skip hoare.Assign hoare.Seq hoare.If
```

Recordatorio:

Es importante no confundir las nociones de terna verdadera y terna demostrable.

- Validez ($\models \{P\}\ C\ \{Q\}$), basada en la semántica operacional.
- Demostrabilidad ($\vdash \{P\}\ C\ \{Q\}$), basada en el conjunto de reglas.

Posteriormente, (4.5.3), probaremos que ambos conceptos son equivalentes:

$$\models \{P\}\ C\ \{Q\} \text{ si y sólo si } \vdash \{P\}\ C\ \{Q\}$$

Algunas de las reglas que forman parte del sistema deductivo, `Skip`, `Assign` o `While`, no son fáciles de manejar, pues sólo se pueden aplicar hacia atrás si la precondición o la postcondición tienen una forma exacta. Por ello, establecemos las reglas

siguientes, que se prueban usando la regla de consecuencia.

Reforzamiento de la precondición:

```
lemma strengthen_pre:
  "⟦ ∀s. P' s ⟶ P s;  ⊢ {P} c {Q} ⟧ ⟹ ⊢ {P'} c {Q}"
by (blast intro: conseq)
```

Debilitamiento de la postcondición:

```
lemma weaken_post:
  "⟦ ⊢ {P} c {Q};  ∀s. Q s ⟶ Q' s ⟧ ⟹  ⊢ {P} c {Q'}"
by (blast intro: conseq)


lemma Assign': "∀s. P s ⟶ Q(s[a/x]) ⟹ ⊢ {P} x ::= a {Q}"
by (simp add: strengthen_pre[OF _ Assign])


lemma While':
assumes "⊢ {λs. P s ∧ bval b s} c {P}" and
          "∀s. P s ∧ ¬ bval b s ⟶ Q s"
shows "⊢ {P} WHILE b DO c {Q}"
by(rule weaken_post[OF While[OF assms(1)] assms(2)])
```

Observación:
En las deducciones en la lógica de Hoare es útil realizar razonamiento hacia atrás con la regla `Seq`, modificando el orden de los subobjetivos generados. Para hacerlo en un sólo paso, establecemos el lema `Seq_bwd`.

```
lemmas Seq_bwd = Seq[rotated]
```

Ejemplo a)

Consideremos el programa que suma en `y` los números de 1 a `x`:

```
y := 0;
WHILE 0 < x DO (y := y + x; x := x-1)
```

Formalmente, el programa en el lenguaje IMP es la sucesión de la asignación `y:=0` y el siguiente bucle

```
abbreviation "wsum ==
  WHILE Less (N 0) (V ''x'')
  DO (''y'' ::= Plus (V ''y'') (V ''x'');;
      ''x'' ::= Plus (V ''x'') (N -1))"
```

Para probar que, efectivamente, el programa realiza la suma de los números de 1 a `x`, definimos la función `sum`, tal que `sum i` calcula la suma de los números de 1 a `i`.

```
fun sum :: "int ⇒ int" where
"sum i = (if i ≤ 0 then 0 else sum (i - 1) + i)"
```

Con el comando `declare sum.simps[simp del]` eliminamos las reglas de simplificación que generadas por la definición, y se establecen las siguientes:

```
lemma sum_simps[simp]:
  "0 < i ⟹ sum i = sum (i - 1) + i"
  "i ≤ 0 ⟹ sum i = 0"
by(simp_all)
```

El comportamiento del bucle `WHILE` con respecto a la función `sum` es: Si `t` es el estado que resulta de aplicar `wsum` al estado inicial `s`, entonces el valor de `y` en `t` es el valor inicial de `y` en `s` más la suma de los números desde 1 hasta el valor de `x` en `s`.

Se trata de probar, usando las reglas de inferencia, que el programa realmente guarda en `y` dicha suma, es decir, que la terna de Hoare

```
{x=n}
y := 0;
WHILE 0 <x DO (y := y + x; x := x-1)
{y=sum(n)}.
```

es demostrable. La prueba es la siguiente.

```
lemma "⊢ {λs. s ''x'' = n} ''y'' ::= N 0;; wsum {λs. s ''y'' = sum n}"
apply(rule Seq)
   -- "Se generan dos subobjetivos: "
   -- "1. ⊢ {λs. s ''x'' = n} ''y'' ::= N 0 {?Q}"
   -- "2. ⊢ {?Q} wsum {λs. s ''y'' = sum n}"
prefer 2
   -- "Intercambiamos el orden en el que decidimos probarlos."
   -- "Como es un bucle, aplicamos la regla While', instanciando"
   -- "P con el invariante del bucle"
apply(rule While' [where P = "λs. (s ''y'' = sum n - sum(s ''x''))"])
   -- "En el primer subobjetivo hay que probar que P es un"
   -- "invariante del bucle. Como la instrucción es una composición,"
   -- "aplicamos la regla Seq:"
apply(rule Seq)
prefer 2
apply(rule Assign)
apply(rule Assign')
apply simp
apply simp
apply(rule Assign')
apply simp
done
```

Queda probado que, efectivamente, el programa guarda en `y` dicha suma.

Ejemplo b)

Sea la terna de Hoare

```
{λ_. True} MAX {λs. s ''c'' = max (s ''a'') (s ''b'')}
```

El programa `MAX` es análogo al del ejemplo expuesto para la *regla del condicional* en el capítulo anterior, (3.2.1). En este caso, `MAX` hace referencia al programa en sí, que calcula el máximo de dos números y lo almacena en `c`.

```
definition MAX :: com
    where "MAX ==
      IF (Less (V ''a'') (V ''b''))
      THEN
        ''c'' ::= V ''b''
      ELSE
        ''c'' ::= V ''a''"
```

A continuación probamos que la terna anterior es demostrable.

```
lemma "⊢{λ_. True} MAX {λs. s ''c'' = max (s ''a'') (s ''b'')}"
    unfolding MAX_def
    apply (rule If)
    apply (rule Assign')
    apply simp
    apply arith
    apply (rule Assign')
    apply simp
    apply arith
    done
```

4.5.2. Ejemplos

En este apartado se resuelven de forma aplicativa, y detalladamente, ejemplos similares a los dos anteriores. Se trata de algunas de las ternas que se demostraron en la sección (3.2.2) del capítulo (3). La mayoría de estos ejemplos pertenecen al capítulo 12 del libro ([8]).

Ejemplo 1

Probamos que la terna de Hoare del ejemplo 3 de la sección (3.2.2) es demostrable. Es decir, demostramos el siguiente lema.

```
lemma
  "⊢ { λs. 0 ≤ s ''x'' }
     ''r'' ::= N 0;; ''t'' ::= N 1;;
     WHILE (Not (Less (V ''x'') (V ''t'')))
```

```
    DO (''r'' ::= Plus (V ''r'') (N 1);;
        ''t'' ::= Plus (V ''t'')
                       (Plus (Plus (V ''r'') (V ''r''))
                             (N 1)))
    {λs. (s ''r'')^2 ≤ s ''x'' ∧ s ''x'' < (s ''r'' + 1)^2}"
```

El programa calcula una aproximación entera de la raíz cuadrada de `x`.

Nota: Usaremos las reglas de simplificación `algebra_simps` y `power2_eq_square`.

```
lemma
  "⊢ { λs. 0 ≤ s ''x'' }
     ''r'' ::= N 0;; ''t'' ::= N 1;;
     WHILE (Not (Less (V ''x'') (V ''t'')))
     DO (''r'' ::= Plus (V ''r'') (N 1);;
         ''t'' ::= Plus (V ''t'')
                        (Plus (Plus (V ''r'') (V ''r''))
                              (N 1)))
     {λs. (s ''r'')^2 ≤ s ''x'' ∧ s ''x'' < (s ''r'' + 1)^2}"
apply (rule Seq)
prefer 2
apply (rule While'[where P = "λs. (s ''r'')^2 ≤ (s ''x'') ∧
                                  (s ''t'') = (s ''r'' + 1)^2"])
-- "Observamos que se pueden simplificar los subobjetivos"
-- "mediante reglas algebraicas"
apply (simp add:algebra_simps)
apply (rule Seq)
prefer 2
apply (rule Assign)
apply (rule Assign')
apply (simp add:algebra_simps)
-- "Se simplifican los subobjetivos mediante
-- "la regla de simplificación power2_eq_square"
apply (auto simp add:power2_eq_square)
prefer 2
apply (rule Assign')
apply (simp add:power2_eq_square)
apply (simp add:algebra_simps)
done
```

Ejemplo 2

Probemos la demostrabilidad de la terna del ejemplo 4 de la sección (3.2.2). Es decir,

```
lemma
  "⊢ {λs. s ''x'' = m ∧ s ''y'' = n ∧ 0 ≤ m}
```

```
    diferencia n m
     {λs. s ''t'' = n - m}
```

El programa calcula la diferencia `y - x`.

```
abbreviation "diferencia n m ==
    ''r'' ::= N m;;
    ''t'' ::= N n;;
     WHILE (Less (N 0) (V ''r''))
     DO (''r'' ::= Plus (V ''r'') (N (-1));;
         ''t'' ::= Plus (V ''t'') (N (-1)))"

lemma
  "⊢ {λs. s ''x'' = m ∧ s ''y'' = n ∧ 0 ≤ m}
    diferencia n m
     {λs. s ''t'' = n - m}"
apply (rule Seq_bwd)
apply (rule While'[where P =  "λs. (0 ≤ s ''r'') ∧ (s ''r'' ≤ m)
                                ∧ (m + s ''t'' = n + s ''r'')"])
apply (rule Seq_bwd)
apply (rule Assign)
apply (rule Assign')
apply simp
apply (simp add: algebra_simps)
apply (rule Seq_bwd)
apply (rule Assign)
apply (rule Assign')
apply simp
done
```

Ejemplo 3

Probemos la demostrabilidad de la terna de Hoare del ejemplo 5 de la sección (3.2.2), donde el programa calcula la potencia i-ésima de 2. Es decir, se quiere probar:

```
lemma
  "⊢ {λs. s ''x'' = i ∧ 0 ≤ i}
     ''r'' ::= N 0;;
     ''t'' ::= N 1;;
     WHILE Not(Eq (V ''r'') (N i))
     DO (''r'' ::= Plus (V ''r'') (N 1);;
         ''t'' ::= Plus (V ''t'') (V ''t''))
     {λs. s ''t'' = pot2 i}"
```

Para ello, primeramente, se ha definido la función `pot2`, de manera que `pot2 i` calcula el valor de 2 elevado a `i`.

```
fun pot2 :: "int ⇒ int" where
"pot2 i = (if i ≤ 0 then 1 else pot2 (i - 1) * 2)"
```

Establecemos las siguientes reglas de simplificación, que sustituyen a las reglas de simplificación generadas por la definición.

```
lemma pot2_simps[simp]:
  "0 < i ⟹ pot2 i = pot2 (i - 1) * 2"
  "i ≤ 0 ⟹ pot2 i = 1"
by(simp_all)
```

Se eliminan entonces las que se generaron de manera automática con el comando `declare pot2.simps[simp del]`.

```
lemma
  "⊢ {λs. s ''x'' = i ∧ 0 ≤ i}
     ''r'' ::= N 0;;
     ''t'' ::= N 1;;
     WHILE Not(Eq (V ''r'') (N i))
     DO (''r'' ::= Plus (V ''r'') (N 1);;
         ''t'' ::= Plus (V ''t'') (V ''t''))
     {λs. s ''t'' = pot2 i}"
apply (rule Seq_bwd)
apply (rule While'[where P =  "λs. (0 ≤ s ''r'') ∧ (s ''r'' ≤ i)
                               ∧ (s ''t'' = pot2 (s ''r''))"])
apply (rule Seq_bwd)
apply (rule Assign)
apply (rule Assign')
apply simp
prefer 2
apply (rule Seq_bwd)
apply (rule Assign)
apply simp
apply (rule Assign')
apply simp
-- "se aplica el lema notEq,"
-- "correspondiente al ejemplo que se muestra a continuación"
apply (metis notEq)
done
```

∗ Ejemplo adicional

Dada la siguiente operación de igualdad para expresiones aritméticas

```
fun Eq :: "aexp ⇒ aexp ⇒ bexp" where
  "Eq a1 a2 = And (Not (Less a1 a2)) (Not (Less a2 a1))"
```

probamos algunas de sus propiedades:

```
lemma bval_Eq[simp]: "bval (Eq a1 a2) s = (aval a1 s = aval a2 s)"
by auto

lemma bval_NotEq: "bval (Not (Eq a1 a2)) s = (aval a1 s ≠ aval a2 s)"
by auto

lemma notEq: "¬ bval (bexp.Not (Eq (V ''r'') (N i))) s ⟶ s ''r'' = i"
by (metis aval.simps(1) aval.simps(2) bval_NotEq)
```

Ejemplo 4

Demostramos que la terna del ejemplo 6 de la sección (3.2.2) es demostrable. El programa calcula el cociente y el resto de la división euclídea, y los almacena en `c` y `r`, respectivamente. Esto es, se quiere probar el siguiente lema

```
lemma
  "⊢ {λs. s ''x'' = a ∧ s ''y'' = b ∧ 0 ≤ a ∧ 0 < b}
    ''c'' ::= N 0;;
    ''r'' ::= N a;;
     WHILE (Not (Less (V ''r'') (N b)))
     DO (''c'' ::= Plus (V ''c'') (N 1);;
         ''r'' ::= Plus (V ''r'') (N (-b)))
     {λs. a = b*(s ''c'') + (s ''r'') ∧ (s ''r'' < b) }"
apply (rule Seq_bwd)
apply (rule While'[where P =  "λs. (0 ≤ s ''r'') ∧
                                    a = b*(s ''c'') + (s ''r'')"])
apply (rule Seq_bwd)
apply (rule Assign)
apply (rule Assign')
apply (simp add: algebra_simps)
prefer 2
apply (rule Seq_bwd)
apply (rule Assign)
apply (rule Assign')
apply (simp add: algebra_simps)
apply simp
done
```

Ejemplo 5

Probemos que la terna del ejemplo 2 de la sección (3.2.2), cuyo programa es una variante del ejemplo (4.5.1) para sumar los números de 1 a `n`, es demostrable.

```
lemma
  "⊢ {λs. s ''x'' = i ∧ 0 ≤ i}
     ''y'' ::= N 0;;
     WHILE Not(Eq (V ''x'') (N 0))
```

```
     DO (''y'' ::= Plus (V ''y'') (V ''x'');;
         ''x'' ::= Plus (V ''x'') (N (-1)))
     {λs. s ''y'' = sum i}"
```

Conforme se ha ido construyendo la demostración se han ido definiendo varios lemas auxiliares necesarios (`l1`, `l2`, `l3`, `l4`), que se corresponden con los subobjetivos que han ido surgiendo. Para demostrarlos se ha recurrido a `Sledgehammer`.

- Lema `l1`:

```
lemma l1: "∀s. (s ''y'' = sum i - sum (s ''x'') ∧ 0 ≤ s ''x'') ∧
        ¬ bval (bexp.Not (Eq (V ''x'') (N 0))) s ⟶
        s ''y'' = sum i"
by force
(*by (metis aval.simps(1) aval.simps(2) bval_NotEq
    semiring_numeral_div_class.diff_zero sum_simps(2))*)
```

- Lema `l2`:

```
lemma l2: "∀s. (0 ≤ s ''x'') ∧
        bval (bexp.Not (Eq (V ''x'') (N 0))) s ⟶
        bval (Less (N 0) (V ''x'')) s"
by simp
```

- Lema `l3`:

```
lemma l3: "∀s. (s ''y'' = sum i - sum (s ''x'') ∧
                  bval (Less (N 0) (V ''x'')) s) ⟶
        ((s[Plus (V ''y'') (V ''x'')/''y''])
               [Plus (V ''x'') (N (- 1))/''x'']) ''y'' =
        sum i - sum (((s[Plus (V ''y'') (V ''x'')/''y''])
               [Plus (V ''x'') (N (- 1))/''x'']) ''x'')"
by simp
```

- Lema `l4`:

```
lemma l4:
"∀s. (0 ≤ s ''x'') ∧ ( bval (Less (N 0) (V ''x'')) s) ⟶
      0 ≤ ((s[Plus (V ''y'') (V ''x'')/''y''])
      [Plus (V ''x'') (N (- 1))/''x'']) ''x''"
by auto

lemma
  "⊢ {λs. s ''x'' = i ∧ 0 ≤ i}
     ''y'' ::= N 0;;
     WHILE Not(Eq (V ''x'') (N 0))
     DO (''y'' ::= Plus (V ''y'') (V ''x'');;
         ''x'' ::= Plus (V ''x'') (N (-1)))
     {λs. s ''y'' = sum i}"
```

```
-- "Se trata de una secuencia."
-- "Como preferimos demostrar primero el segundo subobjetivo,"
-- "aplicamos la regla Seq hacia atrás y no ahorramos un paso"
-- "en la demostración."
apply (rule Seq_bwd)
-- "Se generan dos subobjetivos: "
-- "1. ⊢ {?Q} WHILE bexp.Not (Eq (V ''x'') (N 0))
            DO (''y'' ::= Plus (V ''y'') (V ''x'');;
            ''x'' ::= Plus (V ''x'') (N (- 1)))
       {λs. s ''y'' = sum i}"
-- "2. ⊢ {λs. s ''x'' = i ∧ 0 ≤ i} ''y'' ::= N 0 {?Q}"
-- "Como es un bucle, aplicamos la regla While', instanciando"
-- "P con el invariante del bucle"
apply(rule While' [where P =
      "λs. (s ''y'' = sum i - sum(s ''x'')) ∧ (0 ≤ s ''x'')"])
apply (rule Seq_bwd)
apply (rule Assign)
apply (rule Assign')
prefer 3
apply (rule Assign')
apply simp
prefer 2
-- "Se trata del lema l1 que se ha definido anteriormente."
using l1 apply blast
-- "Utilizamos los lemas l2, l3 y l4 anteriores"
using l2 l3 l4 apply blast
done
```

Ejemplo 6

Probamos la siguiente propiedad: La terna {P} C {True} es demostrable para cualesquiera P y C.

```
lemma postCondicion_dem: "⊢ {P} c {λs. True}"
```

Para ello, ha sido necesario definir el siguiente lema auxiliar.

```
lemma postCondicion_valid: "⊨ {P} c {λs. True}"
 by (metis hoare_valid_def)

lemma postCondicion_dem: "⊢ {P} c {λs. True}"
by (metis hoare_complete hoare_valid_def)
```

4.5.3. Adecuación y completitud para la corrección parcial

Como ya se expuso en la sección (3.2.3) del capítulo (3) el teorema de adecuación es como sigue.

Teorema 4.5.1 (Teorema de adecuación de la lógica de Hoare con respecto a la semántica operacional)
Si una terna {P} C {Q} es demostrable, entonces es verdadera:

$$\vdash \{P\}\ C\ \{Q\} \implies \models \{P\}\ C\ \{Q\}.$$

Hemos explicado que la demostración es por inducción en $\vdash$ {P} C {Q}. Hay que probar que cada regla de la lógica conserva la validez. En Isabelle/HOL la prueba es automática para todas las reglas, excepto para la regla `While`. La prueba es la siguiente.

```
theorem hoare_sound: "⊢ {P}c{Q} ⟹ ⊨ {P}c{Q}"
proof(induction rule: hoare.induct)
  case (While P b c)
  { fix s t
    have "(WHILE b DO c,s) ⇒ t ⟹ P s ⟹ P t ∧ ¬ bval b t"
    proof(induction "WHILE b DO c" s t rule: big_step_induct)
      case WhileFalse thus ?case by blast
    next
    case WhileTrue thus ?case
         using While.IH unfolding hoare_valid_def by blast
    qed
  }
  thus ?case unfolding hoare_valid_def by blast
qed (auto simp: hoare_valid_def)
```

Comentarios sobre la demostración anterior:

Para el caso de la regla `While` el subobjetivo que se genera es

```
5. ⋀P b c.
       ⊢ {λs. P s ∧ bval b s} c {P} ⟹
       ⊨ {λa. P a ∧ bval b a} c {P} ⟹
   ⊨ {P} WHILE b DO c {λa. P a ∧ ¬ bval b a}
```

Para este subobjetivo basta probar que para todo `s` y `t`, se tiene

```
"(WHILE b DO c,s) ⇒ t ⟹ P s ⟹ P t ∧ ¬ bval b t"
```

La prueba es por inducción en `"WHILE b DO c"`, usando el esquema de inducción generado por `big_step_induct`. Se tienen dos casos:

Caso 1:

```
¬ bval b s y P s, entonces P s ⟹ P t ∧ ¬ bval b t
```

Esta prueba es inmediata.

Caso 2: dadas las hipótesis

```
bval b s1,
(c,s1) ⇒s2
(c = WHILE b DO c ⟹ P s1 ⟹ P s2 ∧ ¬ bval b s2)
(WHILE b DO c, s2_) ⇒ s3
P s2 ⟹ P s3 ∧ ¬ bval b s3
P s1
```

hay que probar

```
P s3 ∧ ¬ bval b s3.
```

Teniendo `P s1`, `bval b s1`, `(c,s1) ⇒s2` , aplicamos la hipótesis de inducción externa y se obtiene `P s2`. Luego, aplicando la hipótesis de inducción externa, tenemos `P s3 ∧ ¬ bval b s3`.

Veamos ahora el teorema de completitud, también tratado en el capítulo anterior.

Teorema 4.5.2 (Teorema de completitud de la lógica de Hoare con respecto a la semántica operacional)
Si una terna {P} C {Q} es verdadera, entonces es demostrable. Esto es,

$$\models \{P\}\ C\ \{Q\} \implies \vdash \{P\}\ C\ \{Q\}.$$

Ya sabemos que esta demostracion es más elaborada y que se basa en el concepto de *precondición más débil*. La precondición más débil de una instrucción `C` y una postcondición `Q`, es el aserto más débil que, siendo cierto antes de la ejecución de `C`, garantiza que `Q` es cierto después.

Formalizamos ahora en Isabelle/HOL la definición (3.2.1).

```
definition wp :: "com ⇒ assn ⇒ assn" where
"wp c Q = (λs. ∀t. (c,s) ⇒ t  ⟶  Q t)"
```

Observación:
La precondición más debil de `C` y `Q` es un aserto, es decir un predicado sobre estados, que está definido mediante su λ expresión: aplicado a un estado `s` cumple que, para cualquier estado `t` en el que finalice la ejecución de `C`, que ha empezado en `s`, se verifique `Q t`.

El concepto de precondición más débil es central en la lógica de Hoare, pues hace posible la construcción de las pruebas de la lógica, obteniendo las precondiciones hacia atrás, de forma sucesiva.

A partir de esta última definición y de los comandos que aquí se tratan se tienen las siguientes propiedades.

```
lemma wp_SKIP[simp]: "wp SKIP Q = Q"
apply (rule ext) -- "aplicamos extensionalidad: dos funciones son"
                 -- "iguales si lo son en todos sus argumentos"
apply (auto simp: wp_def)
done


lemma wp_Ass[simp]: "wp (x::=a) Q = (λs. Q(s[a/x]))"
by (rule ext) (auto simp: wp_def)


lemma wp_Seq[simp]: "wp (c1;;c2) Q = wp c1 (wp c2 Q)"
by (rule ext) (auto simp: wp_def)


lemma wp_If[simp]:
 "wp (IF b THEN c1 ELSE c2) Q =
 (λs. if bval b s then wp c1 Q s else wp c2 Q s)"
by (rule ext) (auto simp: wp_def)


lemma wp_While_If: -- "no se almacena como regla de simplificación"
 "wp (WHILE b DO c) Q s =
  wp (IF b THEN c;;WHILE b DO c ELSE SKIP) Q s"
by (metis while_unfold wp_def) -- "usando Sledgehammer"


lemma wp_While_True[simp]: "bval b s ⟹
  wp (WHILE b DO c) Q s = wp (c;; WHILE b DO c) Q s"
apply (simp add: wp_While_If)
done


lemma wp_While_False[simp]: "¬ bval b s ⟹ wp (WHILE b DO c) Q s = Q s"
by(simp add: wp_While_If)
```

En la prueba de completitud es clave la propiedad esencial de la precondición más débil de `C` y `Q`: también es una precondición con respecto a la noción de demostrabilidad. Esto es, el lema (3.2.2), que se prueba a continuación.

```
lemma wp_is_pre: "⊢ {wp c Q} c {Q}"
proof(induction c arbitrary: Q)
  case If thus ?case by(auto intro: conseq)
next
  case (While b c)
  let ?w = "WHILE b DO c"
  show "⊢ {wp ?w Q} ?w {Q}"
```

```
  proof(rule While')
    show "⊢ {λs. wp ?w Q s ∧ bval b s} c {wp ?w Q}"
    proof(rule strengthen_pre[OF _ While.IH])
      show "∀s. wp ?w Q s ∧ bval b s ⟶ wp c (wp ?w Q) s" by auto
    qed
    show "∀s. wp ?w Q s ∧ ¬ bval b s ⟶ Q s" by auto
  qed
qed auto
```

Por lo tanto, se muestra a continuación la prueba del teorema de completitud en Isabelle/HOL.

```
theorem hoare_complete: assumes "⊨ {P}c{Q}" shows "⊢ {P}c{Q}"
proof(rule strengthen_pre)
  show "∀s. P s ⟶ wp c Q s" using assms
    by (auto simp: hoare_valid_def wp_def)
  show "⊢ {wp c Q} c {Q}" by(rule wp_is_pre)
qed
```

De estos dos teoremas se sigue el siguiente corolario.

```
corollary hoare_sound_complete: "⊢ {P}c{Q} ⟷ ⊨ {P}c{Q}"
by (metis hoare_complete hoare_sound)
```

4.5.4. Corrección total

Veracidad de una terna:

La noción informal de corrección total de una terna `{`P`}` C `{`Q`}` es la siguiente, tal y como se explica en (3.2.4): si P es cierta antes de la ejecución de C, entonces C termina y Q es cierta después. Es ese caso escribimos $\models$ `[`P`]` C `[`Q`]`.

Definimos la veracidad de una terna formalmente.

```
definition hoare_tvalid :: "assn ⇒ com ⇒ assn ⇒ bool"
  ("⊨t {(1_)}/ (_)/ {(1_)}" 50) where
"⊨t {P}c{Q}  ⟷  (∀s. P s ⟶ (∃t. (c,s) ⇒ t ∧ Q t))"
```

Nota:
Observar que esta definición necesita que la ejecución sea determinista, que es nuestro caso.

Sistema deductivo para la corrección total:

Presentamos un conjunto de reglas de inferencia que son análogas a las presentadas para la corrección parcial, excepto para la instrucción `While`. En este caso, añadimos un predicado

```
T:: state ⇒ nat ⇒ bool
```

para garantizar la terminación del bucle.

Esto es, la formalización de los axiomas y las reglas (3.2.4), (3.2.4), (3.2.4)y (3.2.4). También se formaliza la demostrabilidad de una terna correcta totalmente en cuanto al comando SKIP, aunque teóricamente se pueda considar como un caso particular del comando de asignación.

```
inductive
  hoaret :: "assn ⇒ com ⇒ assn ⇒ bool"
                ("⊢t ({(1_)}/ (_)/ {(1_)})" 50)
where

Skip:  "⊢t {P} SKIP {P}" |

Assign:  "⊢t {λs. P(s[a/x])} x::=a {P}" |

Seq: "⟦ ⊢t {P1} c1 {P2}; ⊢t {P2} c2 {P3} ⟧ ⟹ ⊢t {P1} c1;;c2 {P3}" |

If: "⟦ ⊢t {λs. P s ∧ bval b s} c1 {Q};
      ⊢t {λs. P s ∧ ¬ bval b s} c2 {Q} ⟧
  ⟹ ⊢t {P} IF b THEN c1 ELSE c2 {Q}" |

While:
  "(⋀n::nat.
    ⊢t {λs. P s ∧ bval b s ∧ T s n} c {λs. P s ∧ (∃n'<n. T s n')})
   ⟹ ⊢t {λs. P s ∧ (∃n. T s n)} WHILE b DO c {λs. P s ∧ ¬bval b s}" |

conseq: "⟦ ∀s. P' s ⟶ P s; ⊢t {P}c{Q}; ∀s. Q s ⟶ Q' s  ⟧ ⟹
         ⊢t {P'}c{Q'}"
```

Observación:
En la regla While, la relación T es una medida que necesariamente tiene que decrecer en cada paso del bucle.

Modificamos la regla y obtenemos una versión funcional más intuitiva y útil para el razonamiento.

```
thm While [where T="λs n. n = f s"]
thm While [where T="λs n. n = f s", simplified]

lemma While_fun:
  "⟦ ⋀n::nat. ⊢t {λs. P s ∧ bval b s ∧ n = f s} c {λs. P s ∧ f s < n}⟧
   ⟹ ⊢t {P} WHILE b DO c {λs. P s ∧ ¬bval b s}"
  by (rule While [where T="λs n. n = f s", simplified])
```

Igual que en el caso de la corrección parcial, las reglas `Skip`, `Assign` y `While` no son fáciles de manejar tal y como están definidas. Por ello, se construyen estas otras reglas, que se prueban usando la regla de consecuencia.

```
lemma strengthen_pre:
  "⟦ ∀s. P' s ⟶ P s;  ⊢t {P} c {Q} ⟧ ⟹ ⊢t {P'} c {Q}"
by (blast intro: conseq)


lemma weaken_post:
  "⟦ ⊢t {P} c {Q};  ∀s. Q s ⟶ Q' s ⟧ ⟹  ⊢t {P} c {Q'}"
by (metis hoaret.conseq) -- "usando Sledgehammer"


lemma Assign': "∀s. P s ⟶ Q(s[a/x]) ⟹ ⊢t {P} x ::= a {Q}"
by (simp add: strengthen_pre[OF _ Assign])


lemma While_fun':
assumes "⋀n::nat. ⊢t {λs. P s ∧ bval b s ∧ n = f s} c {λs. P s ∧ f s < n}"
    and "∀s. P s ∧ ¬ bval b s ⟶ Q s"
shows "⊢t {P} WHILE b DO c {Q}"
by(blast intro: assms(1) weaken_post[OF While_fun assms(2)])
```

4.5.5. Ejemplos

En este apartado se resuelve de forma aplicativa, y detalladamente, un ejemplo de terna demostrable con respecto a la correccón total. Se trata del **Ejemplo a)**, (4.5.1), que se expuso al final de la sección (3.2.4). Tras la ejecución del programa se obtiene en `y` la suma de los números de 1 a `x`.

```
lemma "⊢t {λs. s ''x'' = i} ''y'' ::= N 0;; wsum {λs. s ''y'' = sum i}"
  -- "aplicamos la regla Seq"
apply(rule Seq)
prefer 2
  -- "cambiamos el orden de los subobjetivos"
  -- "Se trata de un bucle."
  -- "Aplicamos la regla While_fun', instanciando el invariante"
  -- "y la función de medida"
apply(rule While_fun'
  [where P = "λs. (s ''y'' = sum i - sum(s ''x''))"
    and f = "λs. nat(s ''x'')"])
  -- "En el primer subobjetivo hay que probar
  -- "que la variante elegida decrece."
  -- "Como la instrucción en una composición,"
  -- "aplicamos la regla Seq:"
apply(rule Seq)
```

```
prefer 2
apply(rule Assign)
apply(rule Assign')
apply simp
apply(simp)
apply(rule Assign')
apply simp
done
```

4.5.6. Adecuación y completitud para la corrección total

Teorema 4.5.3 (Teorema de adecuación de la lógica de Hoare para la corrección total)
Si una terna [P] C [Q] es demostrable, entonces es verdadera.

$$\vdash t\ [P]\ C\ [Q] \Longrightarrow \models t\ [P]\ C\ [Q].$$

En este caso, la demostración también es por inducción según la regla `hoaret.induct`. Hay que probar que cada regla de esta lógica conserva la validez. Todos los casos se demuestran de forma automática, excepto para la regla `While`. La prueba es la siguiente:

```
theorem hoaret_sound: "⊢t {P}c{Q}  ⟹  ⊨t {P}c{Q}"
proof(unfold hoare_tvalid_def, induction rule: hoaret.induct)
  -- "todos los casos son automáticos, excepto While"
  case (While P b T c)
  {
    fix s n
    have "⟦ P s; T s n ⟧ ⟹ ∃t. (WHILE b DO c, s) ⇒
     t ∧ P t ∧ ¬ bval b t"
    proof(induction "n" arbitrary: s rule: less_induct)
      case (less n)
      thus ?case by (metis While.IH WhileFalse WhileTrue)
    qed
  }
  thus ?case by auto
next
  case If thus ?case by auto blast
qed fastforce+
```

Comentarios sobre la demostración anterior:

Para el caso de la regla `While` el subobjetivo que se genera es:

```
⋀P b T c.
(⋀n. ⊢t {λs. P s ∧ bval b s ∧ T s n} c {λs. P s ∧ (∃n'<n. T s n')})
⟹ (⋀n. ∀s. P s ∧ bval b s ∧ T s n ⟶ (∃t. (c, s) ⇒
    t ∧ P t ∧ (∃n'<n. T t n'))) ⟹
∀s. P s ∧ Ex (T s) ⟶ (∃t. (WHILE b DO c, s) ⇒ t ∧ P t ∧ ¬ bval b t)
```

Para este subobjetivo basta probar que para todo `s` y `t` se tiene que

```
∃t. (WHILE b DO c, s) ⇒ t ∧ P t ∧ ¬ bval b t"
```

La prueba es por inducción fuerte en `n`, con `s` arbitrario (ver `thm less_induct`). Con esto y, usando las hipótesis de inducción, se tiene el resultado.

Veamos ahora el teorema de completitud.

Teorema 4.5.4 (Teorema de completitud de la lógica de Hoare para la corrección total)
Si una terna [P] C [Q] es demostrable, entonces es verdadera.

$$\vdash_t [P]\ C\ [Q] \Longrightarrow \models_t [P]\ C\ [Q].$$

La prueba de completitud es análoga a la prueba realizada para la completitud de la corrección parcial.

En primer lugar, definimos una noción más fuerte de la precondición más débil, que tiene en cuenta la terminación: es un predicado tal que, aplicado a un estado `s` cumple que, existe un estado `t` tal que la ejecución de `C`, empezando en `s`, termina en `t` y se verifica `Q`.

Formalizamos en Isabelle/HOL la idea de precondición más debil para el caso de la corrección total.

```
definition wpt :: "com ⇒ assn ⇒ assn" ("wpt") where
"wpt c Q  =  (λs. ∃t. (c,s) ⇒ t ∧ Q t)"
```

De esta última definición se siguen las siguientes propiedades.

```
lemma [simp]: "wpt SKIP Q = Q"
apply (rule ext)
apply (auto simp: wpt_def)
done


lemma [simp]: "wpt (x ::= e) Q = (λs. Q(s(x := aval e s)))"
by(auto intro: ext simp: wpt_def)


lemma [simp]: "wpt (c1;;c2) Q = wpt c1 (wpt c2 Q)"
unfolding wpt_def
apply(rule ext)
apply auto
done
```

```
lemma [simp]: "wpt (IF b THEN c1 ELSE c2) Q =
               (λs. wpt (if bval b s then c1 else c2) Q s)"
apply(unfold wpt_def)
apply(rule ext)
apply auto
done
```

Definimos el número de iteraciones necesarias para que el bucle `WHILE b DO c` termine si empieza en el estado `s`. Como es, en realidad, una función parcial, lo definimos mediante `inductive`, con dos reglas:

- Si `b` no es cierta en `s` ⇒ el número de iteraciones es 0.
- Si `b` es cierta en `s`, `(c,s) ⇒ s'` y el número de iteraciones para que el bucle termine si empieza en `s'` es `n`, entonces el número de iteraciones para que el bucle termine si empieza en `s` es `n+1`.

```
inductive Its :: "bexp ⇒ com ⇒ state ⇒ nat ⇒ bool" where
Its_0: "¬ bval b s ⟹ Its b c s 0" |
Its_Suc: "⟦ bval b s; (c,s) ⇒ s'; Its b c s' n ⟧ ⟹ Its b c s (Suc n)"
```

El comando `thm Its.cases` muestra un conjunto de reglas que pueden usarse como reglas de eliminación.

Lema 4.5.1 *La relación* `Its` *es funcional.*

```
lemma Its_fun: "Its b c s n ⟹ Its b c s n' ⟹ n=n'"
proof(induction arbitrary: n' rule:Its.induct)
  case Its_0 thus ?case by (metis Its.cases)
-- "Sledgehammer"
next
  case Its_Suc thus ?case by(metis Its.simps big_step_determ)
-- "Sledgehammer"
qed
```

Lema 4.5.2 *Si el bucle* `WHILE b DO c` *termina, entonces el número de iteraciones* `n` *verifica* `Its b c s n`.

```
lemma WHILE_Its: "(WHILE b DO c,s) ⇒ t ⟹ ∃n. Its b c s n"
proof(induction "WHILE b DO c" s t rule: big_step_induct)
  case WhileFalse thus ?case by (metis Its_0) -- "Sledgehammer"
next
  case WhileTrue thus ?case by (metis Its_Suc) -- "Sledgehammer"
qed
```

Lema 4.5.3 *La precondición total más débil también es una precondición con respecto a la noción de demostrabilidad total.*

```
lemma wpt_is_pre: "⊢t {wpt c Q} c {Q}"
proof (induction c arbitrary: Q)
  case SKIP show ?case by (auto simp:hoaret.Skip)
next
  case Assign show ?case by (auto intro:hoaret.Assign)
next
  case Seq thus ?case by (auto intro:hoaret.Seq)
next
  case If thus ?case by (auto intro:hoaret.If hoaret.conseq)
next
  case (While b c)
  let ?w = "WHILE b DO c"
  let ?T = "Its b c"
  have "∀s. wpt ?w Q s ⟶ wpt ?w Q s ∧ (∃n. Its b c s n)"
       by (metis WHILE_Its wpt_def) -- "Sledgehammer"
    (* unfolding wpt_def by (metis WHILE_Its) *)
  moreover
  { fix n
    let ?R = "λs'. wpt ?w Q s' ∧ (∃n'<n. ?T s' n')"
    { fix s t assume "bval b s" and "?T s n" and "(?w, s) ⇒ t" and "Q t"
      from ‘bval b s‘ and ‘(?w, s) ⇒ t‘ obtain s' where
        "(c,s) ⇒ s'" "(?w,s') ⇒ t" by auto
      from ‘(?w, s') ⇒ t‘ obtain n' where "?T s' n'"
        by (blast dest: WHILE_Its)
      with ‘bval b s‘ and ‘(c, s) ⇒ s'‘ have "?T s (Suc n')"
          by (rule Its_Suc)
      with ‘?T s n‘ have "n = Suc n'" by (rule Its_fun)
      with ‘(c,s) ⇒ s'‘ and ‘(?w,s') ⇒ t‘ and ‘Q t‘ and ‘?T s' n'‘
      have "wpt c ?R s" by (auto simp: wpt_def)
    }
    hence "∀s. wpt ?w Q s ∧ bval b s ∧ ?T s n ⟶ wpt c ?R s"
          by (auto simp:wpt_def)
    note strengthen_pre[OF this While.IH]
  } note hoaret.While[OF this]
  moreover have "∀s. wpt ?w Q s ∧ ¬ bval b s ⟶ Q s"
    by (auto simp add:wpt_def)
  ultimately show ?case by (rule conseq)
qed
```

Comentarios sobre la demostración anterior:

Observar que en el caso `While`, se usa `Its` como argumento para la terminación.

Por lo tanto, se muestra en Isabelle/HOL el teorema de completitud para la corrección total.

```
theorem hoaret_complete: "⊨t {P}c{Q} ⟹ ⊢t {P}c{Q}"
```

```
apply(rule strengthen_pre[OF _ wpt_is_pre])
apply(auto simp: hoare_tvalid_def wpt_def)
done
```

De estos dos teoremas clave se sigue el siguiente resultado.

```
corollary hoaret_sound_complete: "⊢t {P}c{Q} ⟷ ⊨t {P}c{Q}"
by (metis hoaret_sound hoaret_complete)
```

Bibliografía

[1] Universidad de Sevilla, *Lógica Matemática y Teoría de Modelos*
http://www.glc.us.es/~jalonso/DAT2015/index.php5/Temas_LCyTM_2015

[2] Tobias Nipkow, «WHAT'S IN MAIN», 17 February 2016.

[3] Tobias Nipkow, Lawrence C. Paulson y Markus Wenzel,
«A PROOF ASSISTANT FOR HIGHER-ORDER LOGIC», Springer-Verlag,
17 February 2016.

[4] Tobias Nipkow, «PROGRAMMING AND PROVING IN ISABELLE/HOL»,
17 February 2016.

[5] Michael J. C. Gordon,
«PROGRAMMIG LANGUAGE THEORY AND ITS IMPLEMENTATION»,
Prentice Hall, International Series in Computer Science, págs. 3–55, 1988.

[6] Michael J. C. Gordon, «HOARE LOGIC»,
http://www.cl.cam.ac.uk/~mjcg/HoareLogic/Lectures/Oct10.pdf.

[7] Michael J. C. Gordon, «BACKGROUND READING ON Hoare Logic»,
http://www.cl.cam.ac.uk/~mjcg/Teaching/2011/Hoare/Notes/Notes.pdf,
págs. 7–27, 79–84.

[8] Tobias Nipkow, Gerwin Klein, «CONCRETE SEMANTICS»,
with Isabelle/HOL, 29 February 2016.

[9] Reynald Affeldt, «PROVING PROPERTIES ON PROGRAMS»,
From de Coq tutorial at ITP 2015, August 29, 2015.

[10] Michael J.C. Gordon,
«MECHANIZING PROGRAMMING LOGICS IN HIGHER ORDER LOGIC»