Índice general

0. Preliminares **1**
Utilizando matemática . . . 1
Utilizando lógica . . . 3
Sobre este curso . . . 6

1. Sustitución, Igualdad y Asignación **7**
1.1. Nociones básicas, definiciones y notación . . . 7
1.2. Sustitución . . . 10
1.3. Sustitución y variables ocultas . . . 11
1.4. La sustitución como regla de inferencia . . . 11
1.5. Igualdad y sustitución. Regla de Leibniz . . . 12
1.6. La regla de Leibniz y la evaluación de funciones . . . 13
1.7. Razonando con la Regla de Leibniz . . . 14
1.8. La sentencia de asignación . . . 16
1.9. Ejercicios . . . 19

2. Expresiones booleanas **23**
2.1. Sintaxis y evaluación de expresiones booleanas . . . 23
2.2. Usando Tablas de verdad para evaluar expresiones booleanas . . . 26
2.3. Igualdad versus equivalencia . . . 27
2.4. Satisfabilidad, validez y dualidad . . . 28
2.5. Lenguaje y Lógica . . . 29
2.6. Análisis de razonamientos . . . 34
2.7. Resolución de acertijos lógicos . . . 36
2.8. Ejercicios . . . 36

3. Cálculo Proposicional **43**
3.1. El sistema MIU . . . 44
3.2. El sistema mg . . . 48
3.3. Sistemas formales . . . 51
3.3.1. La equivalencia . . . 54
3.3.2. La negación, discrepancia, y false . . . 56
3.3.3. La disyunción . . . 59

3.3.4. La conjunción 60
3.3.5. La implicación 64
3.4. Ejercicios 69

4. Aplicaciones del Cálculo Proposicional **75**
4.1. Forma abreviada en la prueba de implicaciones 76
4.2. Suponiendo el antecedente 78
4.3. Analizando razonamientos en lenguaje corriente 78
4.4. Construyendo contraejemplos 79
4.5. Resolución de acertijos lógicos 80
4.5.1. El dilema del pretendiente de Portia 80
4.5.2. Caballeros y pícaros 81
4.5.3. ¿Existe Superman? 83
4.5.4. El dilema de la lógica clásica 85
4.6. Ejercicios 86

5. Cálculo de Predicados **91**
5.1. Introducción 91
5.2. Predicados y Cálculo de Predicados 92
5.3. El cuantificador universal 93
5.4. El cuantificador existencial 97
5.5. Propiedades de las cuantificaciones universal y existencial 100
5.6. Aplicaciones del cálculo de predicados 102
5.7. Ejercicios 103
5.7.1. Ejercicios de traducción 103
5.7.2. Ejercicios sobre cuantificación universal 104
5.7.3. Ejercicios sobre cuantificación existencial 105
5.7.4. Ejercicios de razonamientos 106

6. Expresiones cuantificadas **107**
6.1. Introducción 107
6.2. Tipos 108
6.3. Sintaxis e interpretación de la cuantificación 111
6.4. Variables libres y ligadas 112
6.5. Revisión de la sustitución en expresiones cuantificadas 113
6.6. La regla de Leibniz para expresiones cuantificadas 114
6.7. Reglas generales para expresiones cuantificadas 115
6.8. Cuantificadores aritméticos 119
6.8.1. Máximos y mínimos 119
6.8.2. Operador de conteo 121
6.9. Ejercicios 121

7. El formalismo básico **125**
7.1. Definiciones y expresiones . . . 126
7.1.1. Reglas para el cálculo con definiciones . . . 127
7.2. Análisis por casos . . . 128
7.3. Pattern Matching . . . 130
7.4. Tipos . . . 131
7.4.1. Tipos básicos . . . 131
7.4.2. Tuplas . . . 132
7.4.3. Listas . . . 132
7.5. Currificación . . . 135
7.6. Ejercicios . . . 136

8. El Modelo computacional **139**
8.1. Valores . . . 139
8.2. Forma canónica . . . 141
8.3. Evaluación . . . 142
8.4. Un modelo computacional más eficiente . . . 144
8.5. Ejercicios . . . 146

9. El Proceso de construcción de programas **151**
9.1. Introducción . . . 151
9.2. Especificaciones . . . 152
9.3. Ejemplos . . . 153
9.4. Más ejemplos . . . 155
9.5. Ejercicios . . . 160

10. Inducción y recursión **163**
10.1. Introducción . . . 163
10.2. Inducción matemática . . . 164
10.3. Ayudas para pruebas por inducción . . . 168
10.4. Verificación de programas . . . 168
10.5. Inducción sobre listas . . . 170
10.6. Verificación de programas con listas . . . 172
10.7. Ejercicios . . . 174

11. Derivación de programas **177**
11.1. Introducción . . . 177
11.2. Derivación de funciones recursivas . . . 177
11.3. Generalización por abstracción . . . 181
11.4. Modularización . . . 184
11.5. Ejercicios . . . 186

12. La Programación Imperativa **189**
12.1. Introducción 189
12.2. Especificaciones de programas 189
12.2.1. Representación de variables iniciales y finales 191
12.3. Leyes sobre tripletas 192
12.4. El transformador de predicados wp 194
12.5. Ejercicios 194

13. Definición de un lenguaje de programación imperativo **197**
13.1. La sentencia skip 197
13.2. La sentencia abort 198
13.3. La sentencia de asignación 198
13.4. Concatenación o composición 200
13.5. La sentencia alternativa 202
13.6. Repetición 205
13.6.1. Terminación de ciclos 207
13.7. Ejemplos 208
13.8. Ejercicios 212

A. Axiomas y Teoremas **217**
A.1. Teoremas del Cálculo Proposicional 217
A.2. Teoremas del Cálculo de Predicados 220
A.3. Cuantificación Existencial 221
A.4. Propiedades de las cuantificaciones universal y existencial 221
A.5. Metatoremas 222
A.6. Leyes Generales de la Cuantificación 222
A.7. Cuantificador max 223

CAPÍTULO 0
Preliminares

Índice del Capítulo

Utilizando matemática **1**

Utilizando lógica **3**

Sobre este curso **6**

Utilizando matemática

La matemática se utiliza con frecuencia para representar o modelar distintos fenómenos reales. Esto se debe a que la matemática provee una forma de representación concisa, precisa y además apta para la manipulación de la estructura interna de los objetos que se modelan.

Por ejemplo, la ecuación $e = m \cdot c^2$ fue la forma en que Albert Einstein expresó su convencimiento sobre la relación entre la energía e y la masa m de una partícula (siendo c la velocidad de la luz).

Las leyes del movimiento planetario, o al menos los modelos de estas leyes, se utilizan en el lanzamiento y puesta en órbita de los satélites. Las ciencias sociales también emplean matemática a la hora de realizar estadísticas o analizar, comprender y predecir el comportamiento de la sociedad. Los modelos matemáticos también ayudan en la predicción del estado del tiempo o el stock de una empresa.

Como simple ejemplo de modelo matemático, presentamos el siguiente problema:

> *María tiene el doble de manzanas que Juan, María tira la mitad de las manzanas pues están en mal estado y Juan come una de sus manzanas. María todavía tiene el doble de manzanas que Juan. ¿Cuántas manzanas tienen María y Juan?*

Supongamos que mediante las letras m y j representamos las cantidades de manzanas que tienen María y Juan, entonces ponemos el problema anterior en fórmulas matemáticas del sigu-

iente modo:

$$m = 2j \qquad \text{y} \qquad \frac{\text{m}}{2} = 2 \cdot (\text{j} - 1)$$

Cualquier valor de m y j que haga ciertas a las dos igualdades expresadas arriba es una posibilidad en el número de manzanas que tenían inicialmente María y Juan. Observemos que la forma matemática de expresar el problema es más reducida que la forma de expresarlo en lenguaje español. En general podemos afirmar que todo modelo matemático es más preciso, conciso y riguroso que cualquier descripción informal de un problema escrito en lenguaje español.

Para ilustrar las ventajas del rigor, consideremos un proceso de cálculo mediante el cual se consigue una aproximación entera de la raíz cuadrada de un número entero n, a la que llamaremos b.

Para calcular $\sqrt{n}$, es necesario que $n \geq 0$, pues no existe la raíz cuadrada de un número entero negativo. Por lo tanto la expresión $n \geq 0$ especifica rigurosamente bajo que condiciones puede llevarse a cabo el proceso de cálculo, con lo cual esta expresión recibe el nombre de *precondición* del proceso. Mientras que la precondición indica qué debe suponerse cierto antes de la ejecución de un proceso, la *poscondición* determina qué es cierto luego de la ejecución de éste. En nuestro ejemplo, formalizar la poscondición requiere que nos pongamos de acuerdo acerca de b que, como ya dijimos, representa la aproximación de $\sqrt{n}$. A continuación damos dos posibles elecciones de aproximación adecuada de $\sqrt{n}$:

1. $b^2 \leq n < (b+1)^2$
2. $(b-1)^2 < n \leq b^2$

La primera elección de b consiste en calcular el mayor entero que es a lo sumo $\sqrt{n}$, y la segunda corresponde a calcular el menor entero que es al menos $\sqrt{n}$.

Observemos que en lenguaje español nos referimos simplemente a "una aproximación de $\sqrt{n}$", mientras que cuando usamos formulaciones matemáticas nos vemos forzados a ser precisos especificando con exactitud qué aproximación es considerada aceptable, el rigor nos conduce a un análisis más exhaustivo.

Otra importante ventaja en el uso de modelos matemáticos es la posibilidad que ofrecen de contestar preguntas sobre los objetos o fenómenos que se modelan.

El descubrimiento del planeta Neptuno ilustra esta ventaja. En el siglo XVII, Kepler, Newton y otros formularon modelos matemáticos que describían el movimiento planetario basados en la observación de los planetas y las estrellas. En el siglo XVIII los científicos descubrieron que el movimiento de Urano no acordaba con la descripción formulada por los modelos, estas discrepancias produjeron que la Real Sociedad de Ciencias de Göttingen, en Alemania, ofreciera un premio a quien estableciera una teoría aceptable sobre el movimiento de Urano. Los científicos de la época conjeturaron que la órbita de Urano era afectada por la presencia de un planeta desconocido. Luego de dos o tres años de cálculos (todos hechos a mano) obtuvieron la posición probable del planeta desconocido y rastreando el área con telescopios descubrieron el planeta Neptuno en 1846.

Por otra parte, es importante destacar que la matemática provee métodos de razonamiento: a través de la manipulación de expresiones se pueden demostrar propiedades de las expresiones y obtener nuevos resultados desde otros ya conocidos. Este razonamiento puede realizarse sin el conocimiento sobre el significado de los símbolos que son manipulados. Es decir, la matemática ofrece reglas que permiten aprender cuestiones sobre el objeto o el fenómeno que se describe mediante el modelo, sólo la formulación inicial y final necesitan ser interpretadas en términos del problema original. Esto es lo que se conoce como *manipulación sintáctica.*

Aquí presentamos un ejemplo sencillo de manipulación sintáctica. Supongamos que queremos una expresión equivalente a la ecuación $e = m \cdot c^2$ que permita calcular el valor de m suponiendo e dado. Sin necesidad de pensarlo demasiado podemos escribir $m = e/c^2$. En la escuela secundaria se enseñan reglas para la manipulación de expresiones aritmética y la práctica hace que se apliquen automáticamente utilizando una o más de estas reglas simultáneamente. A continuación daremos un detalle de las reglas que aplicadas a la expresión $e = m \cdot c^2$ permiten obtener $m = e/c^2$:

$$
\begin{aligned}
& e = m \cdot c^2 \\
= \quad & \langle \text{dividimos ambos miembros por } c^2 \neq 0 \rangle \\
& e/c^2 = (m \cdot c^2)\,/c^2 \\
= \quad & \langle \text{asociatividad de la multiplicación} \rangle \\
& e/c^2 = m \cdot (c^2/c^2) \\
= \quad & \langle c^2/c^2 = 1 \rangle \\
& e/c^2 = m \cdot 1 \\
= \quad & \langle m \cdot 1 = m \rangle \\
& m = e/c^2
\end{aligned}
$$

En los cálculos realizados arriba, existe un formato especial que adoptaremos en este curso: entre dos expresiones existe un signo igual seguido de una breve frase encerrada entre los signos $\langle\ \rangle$. El signo igual indica que las dos expresiones son iguales mientras que la frase es una explicación que sostiene esa igualdad. Por otra parte, como la relación de igualdad es transitiva (es decir si $a = b$ y $b = c$ podemos concluir que $a = c$), podemos concluir que $e = m \cdot c^2$ es equivalente a $m = e/c^2$.

Observemos que pudimos entender cada una de las manipulaciones anteriores sin entender el significado de e, m y c, es decir, sin conocer que las ecuaciones que fueron manipuladas corresponden a un modelo de la relación entre la masa y la energía; fuimos capaces de razonar sintácticamente.

Utilizando lógica

¿Qué es la lógica?. La lógica es el estudio de los métodos y principios usados para distinguir el buen (correcto) razonamiento del malo (incorrecto). No debe interpretarse esta definición en el sentido de que sólo el estudioso de la lógica puede razonar correctamente, pero la persona que ha estudiado lógica tiene mayor posibilidad de razonar correctamente que aquella que nunca ha pensado en los principios generales implicados en esta actividad.

La lógica ha sido definida a menudo como la ciencia de las leyes del pensamiento. Pero esta definición no es exacta. En primer lugar, el pensamiento es uno de los procesos estudiados por los psicólogos, y la lógica no es una rama de la psicología; es un campo de estudio separado y distinto.

En segundo lugar, si pensamiento es cualquier proceso mental que se produce en la mente de las personas, no todo pensamiento es un objeto de estudio para el lógico. Todo razonamiento es pensamiento, pero no todo pensamiento es razonamiento. Por ejemplo, es posible pensar en un número entre 1 y 10, sin elaborar ningún razonamiento acerca de él. Es posible recordar algo, imaginarlo o lamentarlo sin razonar sobre ello.

Otra definición común de la lógica es aquella que la considera como la ciencia del razonamiento. Esta definición es mejor, pero aún no es adecuada. El razonamiento es un tipo especial de pensamiento en el que se realizan inferencias, o sea en el que se realizan conclusiones que derivan de premisas. Pero los oscuros caminos por los cuales la mente llega a sus conclusiones durante los procesos reales de razonamiento, no son en absoluto de la incumbencia del lógico. Sólo le interesa la corrección del proceso, una vez terminado. Su problema es siempre el siguiente: la conclusión a la que se ha llegado ¿deriva de las premisas usadas o afirmadas?. Si las premisas brindan adecuados fundamentos para aceptar la conclusión, si afirmar que las premisas son verdaderas garantías de que la conclusión también será verdadera, entonces el razonamiento es correcto. De lo contrario es incorrecto.

La distinción entre el razonamiento correcto y el incorrecto es el problema central de la lógica. Los métodos y las técnicas de la lógica han sido desarrollados esencialmente con el propósito de aclarar esta distinción. La lógica se interesa por todos los razonamientos sin tomar en cuenta su contenido. Es decir, la lógica nos permitirá la manipulación sintáctica de expresiones más generales que las expresiones aritméticas que ya hemos mencionado.

Vamos a presentar ahora cierta terminología que se utilizará en lógica.

Una *proposición* es una frase o expresión que sólo puede ser verdadera o falsa. En esto las proposiciones se diferencian de otras frases o expresiones como las exclamaciones, las preguntas o las órdenes. También es necesario diferenciarlas de las oraciones aunque éstas bien pueden ser verdaderas o falsas. Dos oraciones pueden ser diferentes por estar compuestas de diferentes palabras o éstas pueden estar dispuestas de formas distintas, pero si expresan el mismo significado, constituyen la misma proposición, por ejemplo:

Juan ama a María.

Ama Juan a María.

Es claro, que se trata de dos oraciones diferentes, pero ambas tienen el mismo significado, por lo tanto lo que ellas afirman es considerado una proposición.

Otro ejemplo de oraciones diferentes que representan la misma proposición es el siguiente:

Llueve.

It rains.

Il pleut.

La primera está en castellano, la segunda en inglés y la tercera en francés, pero cualquiera de ellas expresa una misma idea.

Por otra parte, en contextos distintos, la misma oración puede ser usada para expresar enunciados diferentes:

El actual presidente de Argentina es un ex gobernador.

En 1998 expresaría un enunciado acerca de Carlos Menem, mientras que en 2004 corresponde a Néstor Kirchner. En estos contextos temporales diferentes, la oración anterior sería usada para afirmar proposiciones diferentes.

En el desarrollo de esta materia volveremos sobre el concepto de proposición a la que también llamaremos *expresión booleana.*

La *inferencia* es un proceso por el cual se llega a una proposición y se la afirma sobre la base de otra u otras proposiciones aceptadas como punto de partida del proceso. Al lógico no le interesa el proceso de inferencia, sino las proposiciones que constituyen los puntos inicial y final de este proceso, así como las relaciones existentes entre ellas.

Aunque el proceso de inferencia no concierne a los lógicos, para cada inferencia posible hay un razonamiento correspondiente y son esos razonamientos los que caen en el ámbito de la lógica. En este sentido un *razonamiento* es cualquier grupo de proposiciones tal que de una de ellas se afirma que deriva de las otras, las cuales son consideradas como elementos de juicio a favor de la verdad de la primera. La palabra ”razonamiento” se usa a menudo para indicar el proceso mismo, pero en lógica tiene el sentido técnico recién explicado.

Al describir esta estructura se emplean comúnmente los términos *premisa* y *conclusión*. La *conclusión* de un razonamiento es la proposición que se afirma sobre la base de las otras proposiciones del mismo, y a la vez estas proposiciones de las que se afirma que brindan los elementos de juicio o las razones para aceptar la conclusión son las *premisas* del razonamiento.

Debemos observar que premisa y conclusión son términos relativos; la misma proposición puede ser premisa en un razonamiento y conclusión en otro. Por ejemplo:

Todo lo que está predeterminado es necesario.

Todo suceso está predeterminado.

Por lo tanto, todo suceso es necesario.

Aquí, la proposición *todo suceso es necesario* es la conclusión, y las otras dos son premisas. Pero la proposición *Todo suceso está predeterminado* es la conclusión del siguiente razonamiento:

Todo suceso causado por otros sucesos está predeterminado.

Todo suceso está causado por otro suceso.

Por lo tanto, todo suceso está predeterminado.

Tomada aisladamente ninguna proposición es una premisa o una conclusión. Es una premisa cuando aparece como supuesto de un razonamiento. Es una conclusión sólo cuando aparece en un razonamiento en el que se afirma que se desprende de las proposiciones afirmadas en ese razonamiento.

En algunos razonamientos, como los dos anteriores, las premisas se enuncian primero y la conclusión al final. Pero no todos los razonamientos presentan ese orden, como por ejemplo:

> *En una democracia, los pobres tienen más poder que los ricos, porque son más, y la voluntad de la mayoría es suprema.*

Sobre este curso de Introducción a la Informática

La breve introducción sobre el uso de la matemática y la lógica que abordaremos con más detalles en los siguientes capítulos, pretende iniciar a los estudiantes en el objetivo de este curso.

La meta es enseñar lógica como una herramienta. No pretendemos que los alumnos sean profundos conocedores de la lógica sino que sepan como aplicarla. La lógica brinda todos los métodos de razonamiento posibles, mientras que la matemática aporta los modelos dotados del rigor necesario para las estructuras de estos razonamientos. Ambas disciplinas proveen un sostén fundamental para la *derivación de programas*, técnica que abordaremos en el próximo curso y que básicamente consiste en la construcción de *programas* correctos. A esta altura podemos decir que un programa es una secuencia lógica de proposiciones descriptas en un lenguaje formal (riguroso) que tiene por objeto la consecución de una tarea específica. Los alumnos de ciencias de la computación no serán lógicos ni matemáticos sino serios estudiosos de los métodos de programación para lo cual es necesario contar tanto con teoría de demostraciones como con modelos de esas teorías. Por esta razón es que haremos énfasis en la manipulación sintáctica de fórmulas como herramienta poderosa para poder descubrir y afirmar verdades.

CAPÍTULO 1
Sustitución, Igualdad y Asignación

Índice del Capítulo

1.1. Nociones básicas, definiciones y notación 7
1.2. Sustitución 10
1.3. Sustitución y variables ocultas 11
1.4. La sustitución como regla de inferencia 11
1.5. Igualdad y sustitución. Regla de Leibniz 12
1.6. La regla de Leibniz y la evaluación de funciones 13
1.7. Razonando con la Regla de Leibniz 14
1.8. La sentencia de asignación 16
1.9. Ejercicios 19

Vamos a introducir el concepto de *sustitución* e ilustrar su aplicación en el razonamiento con *igualdades* y *asignaciones* en los lenguajes de programación. También introducimos la definición de Leibniz de igualdad y la formalizaremos en términos de la sustitución, además damos un formato de *demostración* que nos permitirá probar la igualdad entre dos expresiones.

1.1. Nociones básicas, definiciones y notación

En la vida cotidiana es corriente considerar colecciones de objetos o conjunto de objetos, como por ejemplo el conjunto de alumnos que están inscriptos en el primer año de esta facultad, el conjunto de líneas de colectivo que cubren el tramo Rosario–San Lorenzo, etc. En Informática llamaremos *conjunto* a cualquier colección de objetos que compartan ciertas características, y un *elemento* de un conjunto es cualquiera de sus objetos. En particular consideraremos ciertos conjuntos de números que tienen una importancia especial.

- El conjunto $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$ al que llamaremos *conjunto de los números naturales*.

- El conjunto $\mathbb{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$ es el *conjunto de los números enteros*.
- El conjunto $\mathbb{Z}^+ = \{1, 2, 3, \ldots\}$ es el *conjunto de los números enteros positivos*.
- El conjunto $\mathbb{Z}^- = \{-1, -2, -3, \ldots\}$ es el *conjunto de los números enteros negativos*.

Una forma muy usual de definir conjuntos es establecer las propiedades que deben satisfacer sus elementos. Por ejemplo, mediante la frase: "Sea A el conjunto de todos los números enteros mayores que -1 y menores que 100"; estamos señalando a todos los elementos del conjunto A, una lista exhaustiva de ellos es también una definición del conjunto pero no es necesario explicar que sería extensa y tediosa, por ello es que en matemática es común recurrir a la siguiente definición:

$$A = \{x \in \mathbb{Z} : -1 < x < 100\}$$

aquí, mediante x estamos representando a cualquier número que cumpla con la propiedad de ser entero, mayor que -1 y menor que 100. Claramente cualquier número que satisfaga esta propiedad pertenecerá a A y con la letra x estamos simbolizando a cualquiera de las posibilidades. Cuando se utiliza de esta forma, x recibe el nombre de *variable*, puesto que su valor puede variar entre diversas alternativas posibles, 100 para ser precisos en el caso de nuestro ejemplo.

Una de las nociones fundamentales en Informática es el concepto de variable. Una variable es un mecanismo de notación que se toma prestado de la matemática que aporta concisión y precisión cuando es necesario representar ideas complejas, en resumen podemos decir que:

Una variable es un símbolo que se asocia a cualquier elemento de un conjunto predefinido.

Las expresiones aritméticas son muy utilizadas en la matemática básica que se enseña en la escuela secundaria, por lo tanto consideraremos cierta familiaridad con ellas. Con frecuencia aparecen variables en las expresiones aritméticas, como por ejemplo:

$$(-5) \cdot (3 + x) \text{ o } (4 + x)/2.$$

A continuación presentamos una definición axiomática de expresión aritmética:

(1.1) Definici¿$\frac{1}{2}$n. Expresiones aritméticas:

1 Una constante es una expresión aritmética.

2 Una variable es una expresión aritmética.

3 Si E es una expresión aritmética, (E) también es una expresión aritmética.

4 Si E es una expresión aritmética, $-E$ también es una expresión aritmética.

5 Si E y F son expresiones aritméticas, $(E + F)$, $(E - F)$, $(E \cdot F)$ y (E/F) también son expresiones aritméticas

A partir de esta definición sólo son expresiones aritméticas aquellas construídas utilizando las reglas anteriores. Es decir, hemos establecido la *sintaxis* de las expresiones aritméticas, con el término *sintaxis* nos estamos refiriendo a las leyes o reglas que aseguran que una determinada

sucesión de símbolos pertenece a una determinada clase o tiene cierta estructura. Por lo tanto para reconocer si estamos frente a una expresión aritmética debemos analizar si tal sucesión de símbolos aritméticos puede construirse siguiendo las reglas.

(1.2) Ejemplo. ¿Es $((3 + x) \cdot 2)$ una expresión aritmética?

Para contestar esta pregunta, comenzamos observando que por la regla 1, el número 3 es una expresión aritmética y por la regla 2, también lo es x. Ahora usando la regla 5, sigue que $(3 + x)$ es una expresión aritmética y otra vez la regla 5 y la regla 1 nos garantizan que $((3 + x) \cdot 2)$ es una expresión aritmética.

(1.3) Ejemplo. ¿Es $(\cdot 2)$ una expresión aritmética?

La regla **1** nos dice que 2 es una expresión aritmética, pero de acuerdo a la regla **5** el símbolo $\cdot$ correspondiente a la *multiplicación* es un *operador binario*, es decir necesita dos operandos, aquí sólo tenemos uno. Observemos además que la única expresión que puede construirse usando un operando es mediante el símbolo $-$ que corresponde al *operador unario* de la *negación*. Concluímos que la expresión $(\cdot 2)$ no es una expresión aritmética.

Precedencia: Los paréntesis en las expresiones aritméticas se utilizan en general para indicar qué operaciones preceden a otras. En matemática existen reglas que permiten quitar paréntesis en las expresiones sin dar lugar a interpretaciones ambiguas, esta reglas establecen una jerarquía entre las operaciones que tienen como objetivo abreviar la escritura y que adoptaremos para las expresiones aritméticas. Por ejemplo $7 \cdot 5 + 3$ es la misma expresión que $(7 \cdot 5) + 3$, es decir la operación producto $\cdot$ tiene mayor precedencia que la suma $+$.

Evaluar una expresión aritmética consiste en realizar las operaciones que se indican sobre los operandos que en ella aparecen, está claro en el caso en que sólo existan constantes como operandos, pero como ya mencionamos anteriormente una expresión puede contener variables, en este caso evaluar una expresión requiere conocer que valores deben considerarse para esas variables, para esto introducimos el concepto de *estado*.

(1.4) Definiciï¿½n. Sea E una expresión aritmética, y sean las variables $x_1, x_2, \ldots, x_n$ en E. Un estado de estas variables es una lista de ellas con sus valores asociados.

Por ejemplo, en el estado $(x, 5), (y, 6)$ la variable x está asociada al valor 5 y la variable y está asociada al valor 6.

Ahora entonces podemos definir qué entendemos por *evaluar una expresión E en un estado*.

(1.5) Definiciï¿½n. Dada una expresión E y un estado de las variables de E, evaluar a E en ese estado significa calcular el valor que se obtiene al realizar las operaciones de E sobre los valores asociados a las variables.

Si $E = x \cdot 5 + y$ y el estado es $(x, 8), (y, 2)$, la evaluación de la expresión en este estado da como resultado el número 42.

1.2. Sustitución

(1.6) **Definiciï¿½n.** Sean E y F dos expresiones aritméticas y x una variable en E. Sustituir x por la expresión F en E, significa reemplazar x por F en todas las ocurrencias de la variable x en la expresión E. Esta operación da como resultado una nueva expresión aritmética que denotaremos $E\,[x := F]$.

Veamos a continuación algunos ejemplos:

(1.7) **Ejemplo.** Si $E = (x + y)$ y sustituimos y por la expresión $F = (2 \cdot z)$ entonces $E\,[y := F] = (x + (2 \cdot z)) = x + 2 \cdot z$. Observemos que hemos quitado los paréntesis, utilizando las reglas de precedencia ya mencionadas.

(1.8) **Ejemplo.** Si $E = x$ y $F = (2 + z)$ entonces $E\,[x := F] = (2 + z) = 2 + z$. Esto último también puede expresarse así: $x\,[x := F] = 2 + z$.

(1.9) **Definiciï¿½n.** La sustitución simultánea de una lista x de variables $x_1, x_2, \ldots, x_n$ por una lista F de expresiones $F_1, F_2, \ldots F_n$, que notaremos $E\,[x := F]$ se realiza mediante el reemplazo de la variable x por las expresiones en F cada una de ellas encerradas entre paréntesis, siguiendo como regla de correspondencia el orden y el número de variables.

(1.10) **Ejemplo.** Si $E = x + z$, $F = z + 1$ y $G = 4$ entonces

$$E\,[x, z := F, G] = ((z + 1) + (4)) = z + 5.$$

o también

$$(x + z)\,[x, z := z + 1, 4] = ((z + 1) + (4)) = z + 5$$

(1.11) **Ejemplo.** Es posible sustituir sucesivamente, por ejemplo

$$(x + z)\,[x := z + 1]\,[z := 4] = ((z + 1) + z)\,[z := 4] = ((4) + 1) + (4) = 9$$

Observemos que el resultado no es el mismo que en el caso de sustitución simultánea.

(1.12) **Ejemplo.** Es posible sustituir una variable que no aparece en la expresión, en este caso dicha variable es ignorada, es decir

$$(x + z)\,[x, w := z + 1, 4] = ((z + 1) + z) = 2z + 1$$

Convención: La sustitución tiene mayor jerarquía que cualquier otra operación, es decir tiene precedencia sobre operaciones entre expresiones.

Por lo tanto en caso de que el orden deba alterarse, es necesario el uso de paréntesis, por ejemplo

$$x + z\,[x, z := z + 1, 4] = x + (z\,[x, z := z + 1, 4]) = x + (4) = x + 4$$

si quisiéramos realizar la sustitución indicada arriba en la expresión $E = x + z$, deberíamos utilizar paréntesis como se hizo en el ejemplo 1.11.

(1.13) **Observaciï¿½n.** Es importante no confundir una sustitución con una evaluación. La primera consiste en el reemplazo de variables por expresiones, obteniendo así nuevas expresiones, la segunda en cambio obtiene un valor de una expresión a partir del estado determinado de cada una de las variables de la expresión.

1.3. Sustitución y variables ocultas

Cuando es necesario abreviar escritura, es usual nombrar expresiones para luego utilizar este nombre en otras expresiones, por ejemplo podemos dar el nombre Q a una expresión como:

$$Q : \frac{-b + \sqrt{b^2 - 4 \cdot a \cdot c}}{2 \cdot a}$$

para luego escribir la expresión $x = Q$

Sin embargo, la expresión $x = Q$ tiene tres *variables ocultas*, a, b y c que deben tenerse en cuenta cuando se realice una sustitución. Por ejemplo, la sustitución

$$(x = Q)\,[b := 5]$$

conduce a la expresión $(x = Q')$ donde

$$Q' = \frac{-5 + \sqrt{5^2 - 4 \cdot a \cdot c}}{2 \cdot a}$$

1.4. La sustitución como regla de inferencia

El primer uso que le daremos a la sustitución es la sustitución como regla de inferencia, es decir la utilizaremos como un mecanismo sintáctico para derivar "verdades", o *teoremas*. Veremos más adelante la definición de teorema pero por ahora diremos que un teorema es una proposición verdadera.

Recordemos que una regla de inferencia consiste en un mecanismo por el cual bajo el supuesto de verdad de expresiones llamadas premisas se deriva la veracidad de otra expresión llamada conclusión. Es decir, una regla de inferencia asegura que si las premisas son teoremas, la conclusión también será un teorema. Aquí adoptaremos un formato especial para las reglas de inferencia en general: las premisas se describirán mediante una lista, una bajo la otra y luego de trazar una línea se escribirá la conclusión.

La regla de inferencia de la sustitución utiliza una expresión E, una lista de variables v, y la correspondiente lista de expresiones F, así tenemos

$$Sustitución : \frac{E}{E\,[v := F]\,,} \tag{1.1}$$

esta regla asegura que si E es un teorema, entonces también E es un teorema después de haber reemplazado todas las ocurrencias de v por las correspondientes expresiones de F.

(1.14) Ejemplo. Supongamos que la expresión E que corresponde a $x + y = y + x$, es un teorema, la regla (1.1) nos permite concluir que $b + 3 = 3 + b$ es también un teorema. Esta última expresión no es más que $E\,[x, y := b, 3]$.

(1.15) **Ejemplo.** Supongamos que la expresión $2 \cdot x/2 = x$ es un teorema, de acuerdo a (1.1) podemos concluir que $(2 \cdot x/2 = x)\,[x := j]$ es un teorema o bien $2 \cdot j/2 = j$ es un teorema

Observemos que una regla de inferencia como la sustitución (1.1), es un esquema que representa una infinidad de reglas, una para cada combinación de la expresión E, la lista de variables v, y la lista de expresiones F, podríamos considerar a E y x del ejemplo anterior y $F : j + 5$ y tendríamos:

$$\frac{2 \cdot x/2 = x}{(2 \cdot x/2 = x)\,[x := j + 5]} \qquad \text{o bien} \qquad \frac{2 \cdot x/2 = x}{2 \cdot (j + 5)\,/2 = j + 5}$$

1.5. Igualdad y sustitución. Regla de Leibniz

Hasta ahora hemos hablado del significado de evaluar una expresión aritmética E en un estado. Supongamos que consideramos una expresión un poco más compleja en donde participa la igualdad, como por ejemplo $E = F$, en este caso evaluar la expresión en un estado arroja como resultado *verdadero (true)* si las expresiones E y F tienen el mismo valor y *falso (false)* si tienen distintos valores

Vamos a ver algunos ejemplos, en donde utilizaremos la relación de igualdad entre expresiones aritméticas.

(1.16) **Ejemplo.** Si $E = 3 \cdot x - 1$ y $G = x + 1$, entonces la expresión $E = G$ devuelve true si el estado de x es 1 y devuelve false en cualquier otro estado de la variable x.

(1.17) **Ejemplo.** Si $E = x \cdot x - 1$ y $G = (x + 1) \cdot (x - 1)$ la expresión $E = G$ devuelve el valor true en cualquier estado de la variable x.

También es posible construir expresiones utilizando otros operadores relacionales entre expresiones aritméticas, como por ejemplo

(1.18) **Ejemplo.** Si $E = x - 2$ y $G = x - 1$ la expresión $E < G$ devuelve el valor true en cualquier estado de sus variables.

Una expresión es verdadera si su evaluación en cualquier estado posible de sus variables produce el valor true.

Las expresiones de los ejemplos (1.17) y (1.18) son verdaderas, mientras que la del ejemplo (1.16) es falsa.

Ahora, supongamos que queremos determinar si dos expresiones aritméticas E y F son iguales. De acuerdo con lo visto, ésto equivale a determinar si la expresión $E = F$ es verdadera, lo cual puede realizarse evaluando el valor de verdad de la expresión $E = F$ en todos sus estados posibles, tarea que puede resultar tediosa y hasta imposible en casos en que las variables puedan asumir infinitos estados. Una forma alternativa de determinar la igualdad entre dos expresiones es utilizar un conjunto de reglas para la relación de igualdad (estrategia ampliamente utilizada

en los cursos de Álgebra y Análisis). Por ejemplo, sabemos que $x = y$ es lo mismo que $y = x$, sin importar el valor de las variables x e y. Es más, una adecuada colección de esta reglas puede pensarse como una definición de igualdad, suponiendo válido que dos expresiones son iguales en cualquier estado si y sólo si una de ellas puede ser transformada en la otra mediante estas leyes.

Vamos a dar ahora cuatro leyes que caracterizan a la relación de igualdad. Las dos primeras son expresiones que postulamos como teoremas, es decir son verdaderas en cualquier estado.

$$Reflexividad : x = x \tag{1.2}$$

$$Simetría : (x = y) = (y = x) \tag{1.3}$$

La tercera regla, *transitividad*, está dada como regla de inferencia

$$Transitividad : \frac{X = Y, Y = Z}{X = Z} \tag{1.4}$$

Utilizaremos la regla (1.4) del siguiente modo: de $X = Y$ e $Y = Z$ concluimos que $X = Z$.

Por ejemplo de $x + y = w + 1$ y $w + 1 = 7$ concluímos por la regla (1.4) que $x + y = 7$. En el Capítulo 0, fue esta regla la que nos permitió afirmar que las expresiones $e = m \cdot c^2$ y $m = e/c^2$ son iguales.

Una cuarta regla para la relación de igualdad fue establecida por Gottfried Leibniz hace unos 350 años. En términos actuales la Regla de Leibniz puede expresarse así:

Dos expresiones son iguales en todos los estados si y sólo si al reemplazar una de ellas por la otra en cualquier expresión E, el valor de ésta no cambia.

Una consecuencia de esta ley puede formularse como regla de inferencia del siguiente modo:

$$Leibniz : \frac{X = Y}{E\left[z := X\right] = E\left[z := Y\right]} \tag{1.5}$$

Observemos que se utilizó la variable z en la formulación de la regla anterior, pues la sustitución está definida para el reemplazo de variables y no para el reemplazo de una expresión. En el lado izquierdo de la igualdad que aparece en la conclusión, z es reemplazada por X mientras que en lado derecho z es reemplazada por Y, así este uso de la variable z permite el reemplazo de una instancia de X en la expresión $E\left[z := X\right]$ por Y.

Veamos un ejemplo de la aplicación de la regla de Leibniz, supongamos que $b + 3 = c + 5$ es un teorema, podemos concluir que $d + b + 3 = d + c + 5$ es un teorema usando la regla (1.5), si llamamos $X : b + 3$, $Y : c + 5$, $E : d + z$ y $z : z$.

1.6. La regla de Leibniz y la evaluación de funciones

La definición de función se verá en detalle en los cursos de Análisis, por esta razón no nos detendremos demasiado en ella, pero podemos decir que una función provee una regla que a un

dado valor w le asocia otro valor v, w es llamado argumento de la función y v es el valor de la función, por ejemplo consideremos la función definida como:

$$g\,(z) = 3 \cdot z + 3 \tag{1.6}$$

entonces la función g asume el valor $3 \cdot w + 3$ para cualquier argumento w. El argumento está designado en la *aplicación de la función* que es una forma de expresión. La notación convencional para la aplicación de la función g al argumento 5 es $g\,(5)$, que resulta $3 \cdot 5 + 3$. A fin de reducir el uso de paréntesis en la notación usaremos la notación $g.5$ en vez de $g\,(5)$ cuando el argumento sea una constante o una variable. Por ejemplo la función (1.6) puede definirse mediante esta notación así:

$$g.z : 3 \cdot z + 3$$

A continuación daremos algunos ejemplos de evaluación de aplicación de funciones:

$$\begin{array}{ll} & g.5 \\ = & \langle \text{aplicación de función} \rangle \\ & 3 \cdot 5 + 3 \\ = & \langle \text{cálculo aritmético} \rangle \\ & 18 \end{array}$$

$$\begin{array}{ll} & g\,(y + 2) \\ = & \langle \text{aplicación de función} \rangle \\ & 3 \cdot (y + 2) + 3 \end{array}$$

La aplicación de función puede definirse a través de la sustitución, es decir si la función g se define como:

$$g.z : E$$

entonces la aplicación de la función $g.X$ para cualquier argumento X se define mediante $g.X = E\,[z := X]$. De este modo reescribiremos la regla de Leibniz en términos de aplicación de función y sustitución del siguiente modo:

$$Leibniz : \frac{X = Y}{g.X = g.Y}$$

Esta regla asegura que de la igualdad entre las expresiones X e Y, podemos deducir la igualdad de la aplicación de funciones $g.X$ y $g.Y$. Esta regla fundamental es válida para cualquier función g y expresiones X e Y.

1.7. Razonando con la Regla de Leibniz

La regla de Leibniz nos permite sustituir iguales por iguales en una expresión sin cambiar el valor de la expresión. Esto nos provee un método para demostrar que dos expresiones son iguales. En este método, el formato que utilizaremos será el siguiente:

$$\begin{array}{ll} & E\,[z := X] \\ = & \quad \langle X = Y \rangle \\ & E\,[z := Y] \end{array}$$

La primera y la última línea corresponden a la expresión de igualdad de la conclusión en la regla de Leibniz, la explicación entre los signos $\langle\,\rangle$ corresponde a la premisa $X = Y$.

Aquí ilustramos la regla de Leibniz para el problema enunciado en el Capítulo 0 sobre las manzanas de Juan y María. Recordemos el modelo asociado al problema:

$$m = 2j \qquad \text{y} \qquad m/2 = 2 \cdot (j-1)\,,$$

entonces

$$\begin{array}{ll} & m/2 = 2 \cdot (j-1) \\ = & \quad \langle m = 2 \cdot j \rangle \\ & 2 \cdot j/2 = 2 \cdot (j-1) \end{array}$$

En este caso E es la expresión $z/2 = 2 \cdot (j-1)$, X es m e Y es $2 \cdot j$.

Veamos otro ejemplo de aplicación de (1.5). Supongamos que sabemos que la siguiente expresión es un teorema:

$$2 \cdot x/2 = x \tag{1.7}$$

el siguiente cálculo utiliza las reglas (1.5) y (1.1):

$$\begin{array}{ll} & 2 \cdot j/2 = 2 \cdot (j-1) \\ = & \quad \langle (1.7) \text{ con } x := j \rangle \\ & j = 2 \cdot (j-1) \end{array}$$

Estamos usando Leibniz con la premisa $2 \cdot j/2 = j$. Podemos usar esta premisa sólo en el caso de que se trate de un teorema, pero suponemos que (1.7) lo es y entonces por (1.1), resulta que $(2 \cdot x/2 = x)\,[x := j]$ es también un teorema.

Si el uso de la sustitución es suficientemente obvio, como en este caso, podemos abreviar la explicación entre $\langle\,\rangle$, del siguiente modo:

$$\begin{array}{ll} & 2 \cdot j/2 = 2 \cdot (j-1) \\ = & \quad \langle 2 \cdot x/2 = x \rangle \\ & j = 2 \cdot (j-1) \end{array}$$

También podemos agregar un comentario aclaratorio después de un guión así:

$= \quad \langle 2 \cdot x/2 = x$ – el símbolo "/" significa "divide a"$\rangle$

Cualquier demostración que involucre una sucesión de aplicaciones de la regla de Leibniz, tendrá la siguiente forma:

$$
\begin{array}{ll}
& E_0 \\
= & \quad \langle \text{argumento a favor de } E_0 = E_1, \text{ usando Leibniz} \rangle \\
& E_1 \\
= & \quad \langle \text{argumento a favor de } E_1 = E_2, \text{ usando Leibniz} \rangle \\
& E_2 \\
= & \quad \langle \text{argumento a favor de } E_2 = E_3, \text{ usando Leibniz} \rangle \\
& E_3
\end{array}
$$

Los pasos individuales $E_0 = E_1$, $E_1 = E_2$ y $E_2 = E_3$ y la transitividad (regla (1.4)), demuestran que $E_0 = E_3$.

1.8. La sentencia de asignación

En la sección previa vimos como la sustitución interactúa con la igualdad, ahora veremos una correspondencia entre la sustitución y la sentencia de asignación que permitirá a los programadores razonar con asignaciones.

Supongamos E una expresión y un estado de sus variables, ejecutar la sentencia de asignación

$$x := E \tag{1.8}$$

significa evaluar la expresión E en ese estado y guardar el resultado en x. La asignación $x := E$ se lee "el valor de E se asigna a x."

La ejecución de la asignación (1.8) en un estado, guarda en x el valor de E en ese estado, alterando entonces el estado de x. Por ejemplo, supongamos que un estado consiste en la lista $(v, 5)$, $(w, 4)$, $(x, 8)$ y consideremos la asignación $v := v + w$, el valor de la expresión $v + w$ en ese estado es 9 y la ejecución de $v := v + w$ guarda 9 en v, con lo cual el estado cambia a $(v, 9)$, $(w, 4)$, $(x, 8)$.

La manera de ejecutar una asignación es tan importante como la manera de razonar sobre sus efectos. Por ejemplo, a partir de la precondición de una asignación, ¿cómo podemos determinar la correspondiente poscondición? O también, desde la poscondición, podríamos determinar una adecuada precondición? Recordemos que en el Capítulo 0, introducimos los conceptos de pre y poscondición para un proceso en general, ahora diremos que la precondición de una sentencia o instrucción es una afirmación sobre el estado de las variables de un programa para el cual la sentencia puede ser ejecutada, mientras que la poscondición es una afirmación sobre el estado de las variables una vez ejecutada la asignación. El modo usual de indicar la precondición y la poscondición de una sentencia S es

$$\{P\} \quad S \quad \{Q\}$$

donde P es la precondición y Q es la poscondición correspondientes a S. Esta notación es conocida como tripleta de Hoare, debido a que fue C.A.R. Hoare quien introdujo la notación que permitiría a los lenguajes de programación definir los términos bajo los cuales puede determinarse que un programa es correcto respecto de sus especificaciones en lugar de probar su efectividad mediante ejecuciones sucesivas.

Por ejemplo,

$$\{x = 0\} \quad x := x + 1 \quad \{x > 0\}$$

es una tripleta de Hoare que resultará *válida* si y sólo si la ejecución de la asignación $x := x + 1$ en cualquier estado en que x es 0 termina en un estado en el que $x > 0$.

Aquí mostramos otros dos ejemplos de tripletas de Hoare válidas para la asignación $x := x + 1$:

$$\{x > 5\} \quad x := x + 1 \quad \{x > 0\}$$
$$\{x + 1 > 0\} \quad x := x + 1 \quad \{x > 0\}$$

La tripleta de Hoare

$$\{x = 5\} \quad x := x + 1 \quad \{x = 7\}$$

no es válida, porque la ejecución de $x := x + 1$ en un estado en el cual $x = 5$ no termina en un estado en el cual $x = 7$.

La siguiente es una definición esquemática de tripleta de Hoare válida para una asignación $x := E$ en términos de sustitución:

(1.19) Definici¿½n. Dada una asignación $x := E$, para cualquier poscondición R, una adecuada precondición es $R\,[x := E]$, es decir

$$\{R\,[x := E]\} \quad x := E \quad \{R\}$$

Es decir, la precondición se calcula a partir de la asignación y la poscondición. Como ejemplo consideremos la asignación $x := x + 1$ y la poscondición $x > 4$. De acuerdo a la definición anterior la expresión E es $x + 1$ y R es $x > 4$. Concluimos entonces que una precondición para una tripleta de Hoare válida es $(x > 4)\,[x := x + 1]$ que resulta $x + 1 > 4$.

Aquí damos más ejemplos de la aplicación de la definición 1.19 para obtener tripletas de Hoare válidas:

$$\{x + 1 > 5\} \quad x := x + 1 \quad \{x > 5\}$$
$$\left\{x^2 > x^2 \cdot y\right\} \quad x := x^2 \quad \{x > x \cdot y\}$$

La propia definición de la pre y poscondición que ya vimos podría hacernos suponer que la poscondición debería calcularse desde la precondición y de ese modo considerar como una definición de tripleta de Hoare válida $\{R\} \quad x := E \quad \{R\,[x := E]\}$. Sin embargo, podríamos obtener usando esta regla incorrecta la tripleta $\{x = 0\} \quad x := 2 \quad \{(x = 0)\,[x := 2]\}$, que es inválida porque una vez finalizada la asignación, el estado no satisface la poscondición. Ahora vamos a ver porqué la definición (1.19) es consistente con la ejecución de la asignación $x := E$.

Supongamos un programa con un estado inicial s y un estado final s'. Vamos a demostrar que $R\,[x := E]$ tiene el mismo valor en el estado s que el que tiene R en el estado s'.

Observemos que $R\,[x := E]$ y R son exactamente lo mismo salvo que, cada ocurrencia de x en R fue reemplazada por E en $R\,[x := E]$. Ahora, la ejecución de $x := E$ guarda en x el valor de la expresión E evaluada en el estado s, luego de la asignación el valor de E en el estado s es

igual al valor de x en el estado s' y como el resto de las variables que no son x tienen el mismo estado en s como en s', sigue que la ejecución comenzó en un estado en donde $R\,[x := E]$ tiene el valor *true* y termina en un estado donde R tiene el valor *true*.

En algunos lenguajes de programación, la sentencia de asignación se extiende a la *múltiple asignación*

$$x_1, x_2, \ldots, x_n := E_1, E_2, \ldots, E_n$$

donde las variables x_i son todas distintas y E_i son expresiones. La múltiple asignación se ejecuta del siguiente modo; primero se evalúan todas las expresiones E_i, lo cual da como resultado valores que llamaremos v_i, luego se asignan v_1 a x_1, v_2 a $x_2, \ldots, v_n$ a x_n. Observemos que todas las asignaciones se realizan una vez que se han evaluado todas las expresiones. Por ejemplo:

$x, y := y, x$	intercambia las variables x e y
$x, i := 0, 0$	guarda 0 en x e i
$i, x := i + 1, x + i$	agrega 1 a i y agrega i a x
$x, i := x + i, i + 1$	agrega i a x y agrega 1 a i

Notemos que los dos últimos ejemplos en la tabla son idénticos.

La definición (1.19) puede reformularse para múltiple asignación, simplemente considerando x como una lista de variables y E como una lista de expresiones, con lo cual la múltiple asignación puede definirse en términos de sustitución simultánea. Veamos ahora ejemplos de tripletas de Hoare válidas para múltiple asignación, donde la precondición está dada por (1.19):

$$\{y > x\} \quad x, y := y, x \quad \{x > y\}$$

$$\{x + i = 1 + 2 + \ldots + (i + 1 - 1)\} \quad x, i := x + i, i + 1 \quad \{x = 1 + 2 + \ldots + (i - 1)\}$$

$$\{x + i = 1 + 2 + \ldots + (i + 1 - 1)\} \quad i, x := i + 1, x + i \quad \{x = 1 + 2 + \ldots + (i - 1)\}$$

las precondiciones de los dos últimos ejemplos son idénticas, aún cuando las variables y las expresiones en la asignación aparecen en diferente orden. Observemos además que en ambos ejemplos, la precondición y la poscondición coinciden (después de restar a ambos miembros i en la precondición), cuando esto ocurre se dice que la precondición *se mantiene* por ejecución de la asignación o equivalentemente, la asignación *preserva* la precondición.

Veamos ahora la diferencia entre múltiple asignación y asignación sucesiva, consideremos:

$$x, y := x + y, x + y \qquad \text{y} \qquad x := x + y;\ y := x + y$$

en el estado inicial $(x, 2)$ y $(y, 3)$ la ejecución de la múltiple asignación de arriba cambia el estado de x e y al valor 5, mientras que la asignación sucesiva cambia el estado de x a 5 y el estado de y a 8. Con lo cual no resultan ser iguales. La precondición que establece la definición (1.19) para el caso de múltiple asignación toma la forma:

$$\{R\,[x, y := E, F]\} \quad x, y := E, F \quad \{R\}\,,$$

mientras que en el caso de asignación sucesiva resulta

$$\{R\,[y := F]\,[x := E]\} \quad x := E; y := F \quad \{R\}\,,$$

en este caso la definición se usa para conseguir la precondición de la segunda asignación que resulta la poscondición de la primera, luego la definición se aplica una vez más para conseguir la precondición de la primera asignación.

Ya vimos que $R\,[y := F]\,[x := E]$ no es lo mismo en general que $R\,[x := E]\,[y := F]$, con lo cual no es sorprendente que estas dos asignaciones tengan efectos distintos.

1.9. Ejercicios

1.1 Realizar las siguientes sustituciones eliminando los paréntesis innecesarios:

a) $(x + 2)\,[x := 6]$

b) $(x + 2)\,[x := x + 6]$

c) $(x \cdot x)\,[x := x + 1]$

d) $(x + z)\,[y := z]$

e) $(x \cdot (z + 1))\,[x := z + 1]$

1.2 Realizar las siguientes sustituciones simultáneas eliminando los paréntesis innecesarios:

a) $(x + y)\,[x, y := 6, 3 \cdot z]$

b) $(x + 2)\,[x, y := y + 5, x + 6]$

c) $(x \cdot (y - z))\,[x, y := z + 1, z]$

d) $(x + y)\,[y, x := 6, 3 \cdot x]$

e) $(x \cdot (z + 1))\,[x, y, z := z, y, x]$

1.3 Realizar las siguientes sustituciones eliminando los paréntesis innecesarios:

a) $(x + 2)\,[x := 6]\,[y := x]$

b) $(x + 2)\,[x := y + 6]\,[y := x - 6]$

c) $(x + y)\,[x := y]\,[y := 3 \cdot x]$

d) $(x \cdot (z + 1))\,[x, y, z := z, y, x]\,[z := y]$

e) $(4 \cdot x \cdot x + 4 \cdot y \cdot x + y \cdot y)\,[x, y := y, x]\,[y := 3]$

1.4 Completar utilizando la regla de Leibniz. En cada caso decidir cuales son las expresiones E, X e Y. Dar todas las respuestas posibles.

a) $\dfrac{x = \quad x + 2}{4 \cdot x + y = \quad ?}$

b) $\dfrac{2 \cdot y + 1 = \quad 5}{x + (2 \cdot y + 1) \cdot w = \quad ?}$

c) $\dfrac{x+1 = \quad y}{3\cdot(x+1)+3\cdot x+1 = \quad ?}$

d) $\dfrac{x = \quad y}{x+x = \quad ?}$

e) $\dfrac{7 = \quad y+1}{7\cdot x+7\cdot y = \quad ?}$

1.5 Para cada una de las expresiones $E\,[z := X]$ y $X = Y$, obtener $E\,[z := Y]$ de acuerdo a la regla de Leibniz. Puede haber más de una respuesta correcta.

	$E\,[z := X]$	$X = Y$
(a)	$x + y + w$	$x = b + c$
(b)	$x + y + w$	$y + w = b \cdot c$
(c)	$x \cdot (x + y)$	$x + y = y + x$
(d)	$(x + y) \cdot w$	$w = x \cdot y$
(e)	$(x + y) \cdot q \cdot (x + y)$	$x + y = y + x$

1.6 Identificar las expresiones $X = Y$ y E que corresponden a la aplicación de la regla de Leibniz para cada uno de los siguientes pares de expresiones $E\,[z := X]$ y $E\,[z := Y]$.

	$E\,[z := X]$	$E\,[z := Y]$
(a)	$(x + y) \cdot (x + y)$	$(x + y) \cdot (y + x)$
(b)	$(x + y) \cdot (x + y)$	$(y + x) \cdot (y + x)$
(c)	$x + y + w + x$	$x + y \cdot w + x$
(d)	$x \cdot y \cdot x$	$(y + w) \cdot y \cdot x$
(e)	$x \cdot y \cdot x$	$y \cdot x \cdot x$

1.7 Utilizando la definición (1.19), determinar la precondición para las siguientes asignaciones y poscondiciones.

	asignación	poscondición
(a)	$x := x + 7$	$x + y > 20$
(b)	$x := x - 1$	$x^2 + 2 \cdot x = 3$
(c)	$x := x - 1$	$(x + 1) \cdot (x - 1) = 0$
(d)	$y := x + y$	$y = x$
(e)	$y := x + y$	$y = x + y$

1.8 Utilizando la definición de la sentencia de asignación múltiple determinar las precondiciones para las siguientes sentencias y poscondiciones.

	sentencia	poscondición
(a)	$y, x := x \cdot y, y \cdot y$	$x + 5 \cdot y > x$
(b)	$x, i := 3, 2$	$i + 1 < x + i + w$
(c)	$x, y := y, x$	$x + x \cdot y + x \cdot y \cdot z > x \cdot y$

1.9 Utilizando la definición de la secuencia de asignaciones determinar las precondiciones para las siguientes sentencias y postcondiciones.

	sentencia	poscondición
(a)	$y := x \cdot y; x := y \cdot y$	$x + 5 \cdot y > x$
(b)	$x := 3; i := 2$	$i + 1 < x + i + w$
(c)	$x := y; y := x$	$x + x \cdot y + x \cdot y \cdot z > x \cdot y$
(d)	$x := x + 1; y := y - 1$	$x + y > 5$

Comparar los resultados con las precondiciones del apartado anterior. En el item (d) ¿qué puede decir acerca de la asignación?

CAPÍTULO 2
Expresiones booleanas

Índice del Capítulo

2.1. Sintaxis y evaluación de expresiones booleanas 23
2.2. Usando Tablas de verdad para evaluar expresiones booleanas 26
2.3. Igualdad versus equivalencia 27
2.4. Satisfabilidad, validez y dualidad 28
2.5. Lenguaje y Lógica 29
Negación, conjunción y disyunción en el lenguaje 30
Implicación y equivalencia 32
2.6. Análisis de razonamientos 34
2.7. Resolución de acertijos lógicos 36
2.8. Ejercicios 36

En este capítulo trabajaremos sobre las expresiones booleanas, cuyo nombre se debe a George Boole (1815–1864). Las expresiones booleanas son utilizadas con frecuencia en distintos lenguajes de programación, por lo tanto el material de éste capítulo será familiar para aquellos que tengan alguna noción de Pascal, C, FORTRAN o algún otro lenguaje. Aquí también veremos cómo expresar frases en español a través de expresiones booleanas.

2.1. Sintaxis y evaluación de expresiones booleanas

Ya vimos en el capítulo anterior algunos ejemplos de expresiones booleanas construídos a partir de expresiones aritméticas. El proceso de construcción de expresiones booleanas es análogo al que vimos para obtener expresiones aritméticas, nada más que es necesario sustituir en la construcción las constantes aritméticas por las constantes *true* y *false* (constantes booleanas), las variables aritméticas por variables booleanas, (aquellas que asumen sólo los valores *true* o

false) y los operadores aritméticos $+$, $-$, $\cdot$ y $/$ por los operadores booleanos $\equiv$, $\not\equiv$, $\wedge$, $\vee$, $\Rightarrow$ y $\Leftarrow$.

Dado que los operadores lógicos actúan sobre constantes o variables booleanas que sólo asumen los valores *true* o *false*, pueden enumerarse los distintos valores que toman de acuerdo a cada posible combinación de los valores de sus argumentos.

Comenzaremos describiendo los operadores unarios, es decir aquellos que actúan sobre un operando solamente. Una forma de hacer esto es enumerando todas las funciones del conjunto $\{true, false\}$ en el conjunto $\{true, false\}$. Aquellas funciones que asumen valores en el conjunto $\{true, false\}$ se conocen como *funciones booleanas*.

Si consideramos las funciones booleanas de un argumento definidas sobre el conjunto $\{true, false\}$, entonces tenemos un total de cuatro funciones como aparecen en la siguiente tabla:

		id	$\neg$	
true	*true*	*true*	*false*	*false*
false	*true*	*false*	*true*	*false*

La tabla anterior es conocida como *tabla de verdad*, las entradas en la tabla de verdad tienen el significado siguiente: si p es una variable booleana, la primer columna se reserva para p y sus distintos estados, mientras que en cada una de las siguientes aparece el resultado de la aplicación de cada función sobre cada uno de los estados. Por ejemplo, en la segunda columna de la derecha vemos que $id.true = true$ y $id.false = false$, esta función recibe el nombre de *función idéntica* o también *identidad*. También observamos en la primera y la última columna dos funciones constantes que no tienen asignado un nombre específico. La función simbolizada con $\neg$ se llama *negación* y corresponde al operador unario *not*, por ejemplo notaremos $\neg false = true$.

Si ahora consideramos todas las funciones booleanas de dos argumentos, es decir definidas sobre el conjunto

$$\{(true, true), (true, false), (false, true), (false, false)\},$$

se obtienen dieciseis posibles funciones que se muestran en la siguiente tabla de verdad:

							$\equiv$		n	$\not\equiv$							
									a							n	
									n							o	
			$\vee$	$\Leftarrow$		$\Rightarrow$		$=$	$\wedge$	d	$\not=$					r	
t	*t*	*t*	*t*	*t*	*t*	*t*	*t*	*t*	*t*	*f*	*f*	*f*	*f*	*f*	*f*	*f*	*f*
t	*f*	*t*	*t*	*t*	*t*	*f*	*f*	*f*	*f*	*t*	*t*	*t*	*t*	*f*	*f*	*f*	*f*
f	*t*	*t*	*t*	*f*	*f*	*t*	*t*	*f*	*f*	*t*	*t*	*f*	*f*	*t*	*t*	*f*	*f*
f	*f*	*t*	*f*	*t*	*f*	*t*	*f*	*t*	*f*	*t*	*f*	*t*	*f*	*t*	*f*	*t*	*f*

Observemos que en la tabla hemos abreviado *true* con la letra *t* y *false* con la letra *f*.

Sólo ocho de las funciones que aparecen en la tabla anterior son suficientemente útiles como para ser conocidas por un nombre particular, y son llamadas *operadores booleanos* o *conectores*.

Por ejemplo la aplicación de las funciones correspondientes a la segunda y tercer columna se denota $b \vee c$ y $x \Leftarrow y$ respectivamente. A continuación daremos un detalle de cada uno de los operadores.

El operador $=$ es la igualdad habitual. La expresión $b = c$ se lee "b es igual a c". Al operador booleano igualdad también se le da el nombre de *equivalencia* y se lo nota con un segundo símbolo, $\equiv$. La expresión $b \equiv c$ se lee "b es *equivalente* a c", también se dice que los operandos b y c son *equivalentes*.

El operador $\neq$ es la desigualdad habitual. La expresión $b \neq c$ se lee "b es distinto de c". El operador $\neq$ satisface $(b \neq c) = \neg (b = c)$. Al operador booleano desigualdad también se le da el nombre de *discrepancia* y se lo nota con un segundo símbolo, $\not\equiv$. Se lo suele llamar también *disyunción exclusiva* u *operador xor*, dado que resulta verdadero cuando exactamente uno de los dos operandos lo es.

El operador $\vee$ es llamado *disyunción*. La expresión $b \vee c$ se lee "b o c" dado que el resultado es verdadero cuando al menos uno de los operandos lo es.

El operador $\wedge$ es llamado *conjunción*. La expresión $b \wedge c$ se lee "b y c", dado que el resultado es verdadero cuando ambos operandos lo son.

El operador $\Rightarrow$ es llamado *implicancia*. La expresión $b \Rightarrow c$ se lee "b implica c" o bien "si b entonces c", el primer operando b, es llamado *antecedente* y el segundo c, *consecuente*. $b \Rightarrow c$ será verdadera en todos los casos excepto cuando b sea verdadera y c falsa. En la tabla de verdad se observa un resultado poco intuitivo, y es que si el antecedente es falso, el resultado de la expresión $b \Rightarrow c$ será verdadero, sin importar el valor del consecuente, esto es consistente con la interpretación en español de frases como "Si la economía mejoró con esos ajustes, yo soy Gardel". En este sentido la frase representa una proposición verdadera simplemente porque la economía no mejoró con esos ajustes y de este modo, falso implica cualquier situación posible o imposible. Veremos con más detalle la implicación en secciones posteriores.

El operador $\Leftarrow$ es llamado *consecuencia*. La expresión $b \Leftarrow c$ se lee "b sigue de c". En la tabla de verdad se puede ver que son equivalentes $c \Rightarrow b$ y $b \Leftarrow c$. Se introduce en nuestras expresiones debido a su utilidad en el proceso de construcción de demostraciones.

Los nombres de los operadores "nand" y "nor" siguen de "not and"(no y) y "not or"(no o) en inglés respectivamente. La expresion $b \, nand \, c$ es igual a $\neg (b \wedge c)$, mientras que $b \, nor \, c$ es igual a $\neg (b \vee c)$.

Una vez presentados todos los operadores lógicos, vamos a definir una precedencia entre ellos, lo cual nos permitirá eliminar paréntesis en expresiones booleanas y abreviar escritura:

(2.1) **Definiciï¿½n.** Los operadores lógicos tendrán la precedencia que se indica a continuación, los números señalan la jerarquía entre ellos, el primero corresponde a la más alta, y los siguientes siguen en orden. Cuando dos operadores aparezcan con el mismo orden de jerarquía significa que tienen la misma precedencia respecto de los demás:

1. operador $=$
2. operador $\neg$
3. operadores $\wedge$ y $\vee$

4. operadores $\Rightarrow$y $\Leftarrow$
5. operadores $\equiv$ y $\not\equiv$

(2.2) Ejemplo. La expresión

$$((p \vee q) \Rightarrow r) \equiv ((p \Rightarrow r) \wedge (q \Rightarrow r))$$

puede simplificarse usando las reglas de precedencia anteriores así:

$$p \vee q \Rightarrow r \equiv (p \Rightarrow r) \wedge (q \Rightarrow r)$$

2.2. Usando Tablas de verdad para evaluar expresiones booleanas

Las tablas de verdad serán útiles entre otras cosas para evaluar expresiones booleanas en cualquier estado. Por ejemplo, la evaluación de la expresión

$$p \wedge \neg q \Rightarrow \neg p$$

en el estado $\{(p, true), (q, true)\}$ se calcula así: si q es verdadera, de acuerdo a la tabla del operador $\neg$, $\neg q$ es falsa, admás cuando uno de los argumentos es falso (en este caso $\neg q$), el resultado de la aplicación del operador $\wedge$, es falso. Por último, la columna de la tabla correspondiente al operador $\Rightarrow$, nos dice que si el primer argumento es falso, el resultado es verdadero independientemente del valor del segundo argumento. Observemos que las operaciones fueron realizadas de acuerdo a las reglas de precedencia citadas anteriormente.

Veamos ahora como evaluar una expresión booleana cualquiera utilizando tablas de verdad. Los posibles valores de las variables booleanas involucradas en la expresión se disponen en las primeras columnas de la tabla, mientras que en las siguientes columnas se colocan los valores parciales de los operadores lógicos intervinientes, en orden y de acuerdo a las reglas de precedencia, hasta colocar en la última columna de la tabla los valores de verdad asociados a la expresión. Cada fila describe un estado particular de las variables y la correspondiente evaluación de acuerdo a cada operador en la expresión.

Por ejemplo, queremos evaluar la expresión $p \wedge \neg q \Rightarrow r$ en todas las combinaciones posibles de valores de las variables p, q y r, entonces construimos:

p	q	r	$\neg q$	$p \wedge \neg q$	$p \wedge \neg q \Rightarrow r$
true	*true*	*true*	*false*	*false*	*true*
true	*true*	*false*	*false*	*false*	*true*
true	*false*	*true*	*true*	*true*	*true*
true	*false*	*false*	*true*	*true*	*false*
false	*true*	*true*	*false*	*false*	*true*
false	*true*	*false*	*false*	*false*	*true*
false	*false*	*true*	*true*	*false*	*true*
false	*false*	*false*	*true*	*false*	*true*

2.3. Igualdad versus equivalencia

La expresión booleana $b \equiv c$ es evaluada exactamente como $b = c$, excepto que la equivalencia puede usarse sólo entre expresiones booleanas.

Sin embargo ambos operadores tendrán propiedades distintas. Primero, la igualdad tiene la precedencia más alta sobre cualquier otro operador, mientras que la equivalencia tiene la precedencia más baja, lo cual nos permite quitar paréntesis en expresiones como

$$x \cdot y = 0 \quad \equiv \quad x = 0 \vee y = 0$$

Notemos que los espacios que rodean a $\equiv$ son útiles para recordarnos que este operador tiene la precedencia más baja.

Por otra parte, la igualdad múltiple se interpreta como "conjuntiva", es decir

$$a = b = c \quad \text{se interpreta como} \quad b = c \wedge c = d,$$

con lo cual $b = c = d$ y $(b = c) = d$ son diferentes, pues en el estado $(b, false)$, $(c, false)$ y $(d, true)$, $b = c = d$ es $false$, mientras que $(b = c) = d$ es $true$.

La equivalencia es asociativa, es decir $b \equiv c \equiv d$ se usará para referirse tanto a $(b \equiv c) \equiv d$ como a $b \equiv (c \equiv d)$, con lo cual no puede ser conjuntiva como ocurrió con la igualdad.

Por lo tanto, en fórmulas sin paréntesis, una secuencia de $=$ tendrá un significado distinto a una de $\equiv$. Veamos algunos ejemplos:

Queremos evaluar $false = false = true$ y $false \equiv false \equiv true$

	$false = false = true$		$false \equiv false \equiv true$	
$=$	$\langle =$ es conjuntiva$\rangle$	$=$	$\langle \equiv$ es asociativa$\rangle$	
	$(false = false) \wedge (false = true)$		$(false \equiv false) \equiv true$	
$=$	$\langle$evaluación de $=\rangle$	$=$	$\langle$evaluación del primer $\equiv\rangle$	
	$true \wedge false$		$true \equiv true$	
$=$	$\langle$evaluación de $\wedge\rangle$	$=$	$\langle$evaluación de $\equiv\rangle$	
	$false$		$true$	

Como segundo ejemplo, mostramos como cambiar ocurrencias de $=$ por ocurrencias de $\equiv$ y viceversa. Utilizaremos paréntesis para las operaciones $b = c$ y $b \equiv c$, antes de efectuar los reemplazos debido a que estos operadores tiene distinta precedencia y no queremos cambiar la estructura de las expresiones:

	$b \equiv c \equiv d$		$b = c = d$	
$=$	$\langle \equiv$ es asociativa$\rangle$	$=$	$\langle =$ es conjuntiva$\rangle$	
	$(b \equiv c) \equiv d$		$(b = c) \wedge (c = d)$	
$=$	$\langle$reemplazo de operadores$\rangle$	$=$	$\langle$reemplazo de operadores$\rangle$	
	$(b \equiv c) = d$		$(b \equiv c) \wedge (c \equiv d)$	
$=$	$\langle$reemplazo de operadores$\rangle$	$=$		
	$(b = c) = d$			

2.4. Satisfabilidad, validez y dualidad

(2.3) **Definiciï¿½n.** Una expresión booleana P se *satisface* en un estado de las variables que aparecen en ella, si su valor es $true$ en ese estado; P se dice *satisfactible* si existe algún estado en el cual P se satisface y P se dice *válida* si se satisface en cualquier estado. Una expresión booleana válida se llama *tautología*.

Es fácil reconocer una tautología, pues en su correspondiente tabla de verdad aparecerá siempre el valor *true* en la última columna. Por ejemplo, $p \Rightarrow p$ y $p \wedge p \equiv p$ son tautologías.

p	$p \Rightarrow p$
true	*true*
false	*true*

p	$p \wedge p$	$p \wedge p \equiv p$
true	*true*	*true*
false	*false*	*true*

La expresión booleana $p \vee q$ se satisface en cualquier estado $(p, true)$, por lo tanto es satisfactible, sin embargo no es válida pues no se satisface en el estado $(p, false)$, es decir, no es una tautología.

(2.4) **Definiciï¿½n.** Una *contradicción* es una expresión booleana cuya evaluación en cualquier estado es siempre $false$.

Observemos que la negación de una tautología es entonces una contradicción, son ejemplos de contradicción $p \not\equiv p$ o también $p \wedge \neg p$.

p	$p \not\equiv p$
true	*false*
false	*false*

p	$\neg p$	$p \wedge \neg p$
true	*false*	*false*
false	*true*	*false*

La familiaridad con expresiones booleanas incluye familiaridad con diversas expresiones válidas simples, la siguiente definición de dualidad ayuda a reducir el número de expresiones que deben recordarse.

(2.5) **Definiciï¿½n.** Dada una expresión booleana P su correspondiente expresión *dual* P_D se construye intercambiando la ocurrencia de los operadores lógicos de acuerdo a lo que se indica en la siguiente tabla

$true$	y	$false$
$\wedge$	y	$\vee$
$\equiv$	y	$\not\equiv$
$\Rightarrow$	y	$\not\Leftarrow$
$\Leftarrow$	y	$\not\Rightarrow$

En la siguiente tabla se ilustran algunos ejemplos de expresiones booleanas y sus respectivas expresiones duales

P	P_D
$p \vee q$	$p \wedge q$
$p \Rightarrow q$	$p \not\Leftarrow q$
$p \equiv \neg q$	$p \not\equiv \neg q$
$false \not\equiv true \vee p$	$true \equiv false \wedge p$
$\neg p \wedge \neg q \equiv r$	$\neg p \vee \neg q \not\equiv r$

Utilizaremos la noción de dualidad para enunciar el Metateorema de Dualidad. Definiremos más adelante la noción de metateorema y omitiremos por el momento la demostración puesto que requiere de técnicas que no hemos desarrollado hasta el momento.

(2.6) Metateorema. Metateorema de Dualidad

a) P es válida si y sólo si $\neg P_D$ es válida

b) $P \equiv Q$ es válida si y sólo si $P_D \equiv Q_D$ es válida.

Este resultado nos dice que si sabemos que una expresión es válida, la correspondiente expresión dual también lo será, con lo cual bastará con recordar una sola de las expresiones y la otra se obtiene por dualidad. En la siguiente tabla se ilustran expresiones válidas y sus correspondientes duales, en la última línea de la tabla aparecen dos expresiones válidas conocidas como "Leyes de Morgan".

P	$\neg P_D$
$true$	$\neg false$
$p \vee true$	$\neg(p \wedge false)$
$p \vee \neg p$	$\neg(p \wedge \neg p)$

$P \equiv Q$	$P_D \equiv Q_D$
$true \equiv true$	$false \equiv false$
$p \vee q \equiv q \vee p$	$p \wedge q \equiv q \wedge p$
$p \equiv q \equiv q \equiv p$	$p \not\equiv q \equiv q \not\equiv p$
$\neg\,(p \vee q) \equiv \neg p \wedge \neg q$	$\neg\,(p \wedge q) \equiv \neg p \vee \neg q$

2.5. Lenguaje y Lógica

La lógica simbólica surgió como una manera de analizar los razonamientos escritos en lenguaje natural. Los operadores presentados en la sección anterior fueron pensados como contrapartidas formales de operadores del lenguaje. Si se tiene cierto cuidado puede traducirse una proposición escrita en lenguaje natural en una expresión booleana. Esta traducción nos permitirá dos cosas: por un lado resolver ambigüedades del lenguaje natural, por otro manipular y analizar las expresiones booleanas así obtenidas usando reglas que serán introducidas en el

próximo capítulo. Como veremos luego, las reglas lógicas ofrecen una alternativa efectiva para razonamientos expresados en el lenguaje corriente.

La idea básica de traducción consiste en identificar las proposiciones elementales de un enunciado que serán representadas mediante variables booleanas y componerlas usando los operadores booleanos asociados a los conectivos del lenguaje que aparecen en el enunciado. Los conectivos del lenguaje serán traducidos a su interpretación "obvia", aunque veremos algunas sutilezas de esta traducción.

Por ejemplo, la oración

> *Hamlet defendió el honor de su padre pero no defendió la felicidad de su madre ni la suya propia*

puede analizarse como compuesta básicamente de tres proposiciones elementales

p : *Hamlet defendió el honor de su padre*
q : *Hamlet defendió la felicidad de su madre*
r : *Hamlet defendió su propia felicidad*

usando esta variables proposicionales la frase anterior puede traducirse como

$$p \wedge \neg (q \vee r)$$

(2.7) Observacïï¿½n. La palabra "pero" se traduce como una conjunción, puesto que afirma ambas componentes. Por otro lado la disyunción usada es inclusiva, ya que podría haber defendido la felicidad de su madre la propia o ambas.

Negación, conjunción y disyunción en el lenguaje

La operación de negación aparece usualmente en el lenguaje insertando un "no" en la posición correcta del enunciado que se quiere negar, alternativamente puede anteponerse la frase "es falso que" o la frase "no se da el caso que".

Por ejemplo el enunciado:

Todos los unicornios son azules (2.1)

puede negarse de las siguientes maneras:

No todos los unicornios son azules
No se da el caso de que todos los unicornios son azules
Es falso que todos los unicornios sean azules
Algunos unicornios no son azules

si simbolizamos con p a la proposición (2.1), cualquiera de las variantes de negación propuestas se simbolizan mediante el operador lógico de negación y se expresan $\neg p$.

La conjunción aparece en el lenguaje con la palabra "y", uniendo dos proposiciones. También se interpretarán como conjunción las palabres "pero" y "aunque". Por ejemplo, si con p y q representamos las proposiciones siguientes:

p : *Llueve*
q : *No hace frío*

las siguientes oraciones representan la proposición $p \wedge q$

Llueve y no hace frío
Llueve aunque no hace frío
Llueve pero no hace frío

Es importante notar que no siempre la palabra "y" representa una conjunción, como es el caso de la siguiente frase:

Joyce y Picasso fueron contemporáneos

aquí la palabra "y" se usa para expresar una relación entre Joyce y Picasso.

La palabra usual que corresponde a la disyunción es "o", también la variante "o bien". El caso de la disyunción es más complejo que los anteriores, dado que existen en el lenguaje dos tipos de disyunciones, la llamada *inclusiva* y la *exclusiva*. Básicamente ambos tipos difieren cuando las proposiciones intervinientes son ambas verdaderas. La disyunción inclusiva considera a este caso como verdadero. Por ejemplo:

Para ser emperador hay que tener el apoyo de la nobleza o del pueblo

obviamente teniendo el apoyo de ambos se está también en condiciones de ser emperador. Esta proposición la expresamos como $p \vee q$, donde p y q son las siguientes proposiciones elementales:

p : *Para ser emperador hay que tener el apoyo de la nobleza*
q : *Para ser emperador hay que tener el apoyo del pueblo*

Un ejemplo de disyunción exclusiva es:

El consejero del emperador pertenece a la nobleza o al pueblo (2.2)

donde se interpreta que el consejero no puede pertenecer simultáneamente a las dos clase sociales. Utilizaremos para este caso el operador discrepancia que simbolizamos como $\not\equiv$, por tanto si llamamos

p : *El consejero del emperador pertenece a la nobleza*
q : *El consejero del emperador pertenece al pueblo*

la proposición (2.2) se expresa así : $p \not\equiv q$.

En latín existen palabaras diferentes para la disyunción inclusiva y exclusiva. La palabra "vel" se usa para la primera y la palabra "aut" para la segunda. El símbolo utilizado en lógica para la disyunción proviene precisamente de la palabra latina.

Implicación y equivalencia

La implicación lógica suele representar lo que en el lenguaje natural se expresa mediante la construcción *si... entonces ...*. Este es uno de los conectivos sobre los que menos acuerdo existe y al que más alternativas se han propuesto. Casi todo el mundo acuerda que si el antecedente p es verdadero y el consecuente q es falso, entonces $p \Rightarrow q$ es falso. Por ejemplo, en la frase

Si se reforman las leyes laborales entonces bajará el desempleo (2.3)

aparecen dos proposiciones elementales

p : *Se reforman las leyes laborales*
q : *Baja el desempleo*

Supongamos que las leyes laborales se reforman y no baja el desempleo, entonces la proposición (2.3) que se expresa como $p \Rightarrow q$ resulta falsa.

Sin embargo, existen ciertas dudas acerca del valor de verdad (o del significado lógico) de frases como

Si dos más dos es cinco, entonces yo soy el Papa

Para la lógica clásica que presentamos aquí, la frase anterior es verdadera (mirando la tabla de verdad del operador $\Rightarrow$), pues ambos antecedente y consecuente son falsos, lo cual hace verdadera a la proposición.

Normalmente se usa una implicación con un consecuente falso como una manera elíptica de negar el antecedente. Por ejemplo, decir

Si la economía mejoró con esos ajustes, yo soy Gardel

es una manera elegante de decir que la economía no mejoró con los ajustes.

Veamos este ejemplo:

Si no tomás la sopa, estarás en problemas

usando las variables p y q del siguiente modo

p : *tomás la sopa*
q : *estarás en problemas*

la frase anterior puede traducirse como $\neg p \Rightarrow q$. Observemos que esta expresión es verdadera si tomás la sopa, pues en ese caso $\neg p$ es falsa y por ende resulta $\neg p \Rightarrow q$ verdadera. Este hecho puede parecer extraño en un principio pero las frases

Si no tomás la sopa, estarás en problemas
Tomás la sopa o estarás en problemas

tienen el mismo significado. Por lo tanto, como la segunda frase es verdadera si tomás la sopa, la primera también debe serlo. La equivalencia entre las expresiones $\neg p \Rightarrow q$ y $p \vee q$ se verá en el próximo capítulo, pero puede deducirse ahora utilizando tablas de verdad.

Otra forma de representar la implicación en el lenguaje natural es a través de las palabras suficiente y necesario.

Es suficiente que vacune a mi hijo para que no contraiga el sarampión (2.4)

Esta frase puede simbolizarse mediante $p \Rightarrow (\neg q)$ donde

p : *Vacuno a mi hijo*
q : *Mi hijo contrae el sarampión*

La proposición (2.4) asegura que vacunar es condición suficiente para no contraer el sarampión, nada dice acerca de aquellos niños que no son vacunados

Otro ejemplo donde aparece la idea de necesidad lógica es:

Para obtener una beca es necesario tener un buen promedio en la carrera (2.5)

que puede simbolizarse así $q \Rightarrow p$ o también $p \Leftarrow q$, donde

p : *Tener un buen promedio*
q : *Obtener una beca.*

La proposición (2.5) establece que tener un buen promedio es condición necesaria para conseguir una beca y no garantiza que todo alumno con buen promedio la obtendrá. Si formulamos esta proposición en términos de *si... entonces...*, resulta

Si obtiene una beca entonces tiene un buen promedio en la carrera

Es un error común interpretar la estructura *si ... entonces ...* en lenguaje natural como *si y sólo si* que corresponde a una equivalencia y no a una implicancia. Por ejemplo la frase

f es biyectiva si y sólo si f es inyectiva y sobreyectiva

puede expresarse como $p \equiv q \wedge r$, donde

p : *f es biyectiva*
q : *f es inyectiva*
r : *f es sobreyectiva*

De todos modos no existe ninguna construcción en el lenguaje corriente que represente fielmente a la equivalencia lógica. Es ésta la razón quizá por la cual la equivalencia suele tener un lugar secundario en los libros de lógica. Nosotros no seguiremos esa tradición, dado que el uso de la lógica en el desarrollo de programas, la equivalencia tiene un rol fundamental. Como último ejemplo de la inadecuación del lenguaje para parafrasear la equivalencia, presentamos la frase (verdadera en lenguaje corriente) acerca de las capacidades de Juan.

Juan ve bien si y sólo si Juan es tuerto si y sólo si Juan es ciego

Como ya vimos, la equivalencia es asociativa, por tanto la frase anterior puede representarte así: $p \equiv q \equiv r$, donde p, q y r son las proposiciones elementales obvias en la frase. Veamos la tabla de verdad correspondiente:

p	q	r	$p \equiv q$	$(p \equiv q) \equiv r$
true	*true*	*true*	*true*	*true*
true	*true*	*false*	*true*	*false*
true	*false*	*true*	*false*	*true*
true	*false*	*false*	*false*	*true*
false	*true*	*true*	*false*	*false*
false	*true*	*false*	*false*	*true*
false	*false*	*true*	*true*	*true*
false	*false*	*false*	*true*	*false*

Observemos que exactamente una de las tres proposiciones p, q y r es verdadera. En la tabla de verdad, en las filas en donde exactamente una de las proposiciones toma el valor *true*, el valor de la expresión booleana $p \equiv q \equiv r$ es también *true*.

2.6. Análisis de razonamientos

En el capítulo 0 vimos ejemplos de razonamientos, y ahora estamos en condiciones de analizar la validez de algunas formas de razonamiento. Hemos dicho anteriormente que en un razonamiento válido, si las premisas son verdaderas la conclusión también debe serlo. Una forma de analizar un razonamiento entonces, es traducir éste a expresiones booleanas y analizar el caso en que cada una estas expresiones es verdadera. Por ejemplo, supongamos un razonamiento en donde $p_0, p_1, \ldots p_n$ son las premisas y q es la conclusión, es decir:

$$\begin{array}{c} p_0 \\ \vdots \\ p_n \\ \hline q \end{array}$$

éste se considerará válido si siempre que p_0, p_1, . . ., p_n sean verdaderas entonces también lo es q.

Dadas las tablas de verdad de la conjunción y de la implicación, esto ocurre si y sólo si la siguiente expresión es una tautología

$$p_0 \wedge p_1 \wedge \ldots \wedge p_n \Rightarrow q$$

Por ejemplo en el razonamiento

Si tuviera pelo sería feliz
No soy feliz

No tengo pelo

si llamamos:

p : *Tengo pelo*
q : *Soy feliz*

podemos traducirlo como

$$\frac{p \Rightarrow q \quad \neg q}{\neg p}$$

Se puede ahora comprobar la validez del razonamiento construyendo la tabla de verdad asociada:

p	q	$p \Rightarrow q$	$\neg q$	$(p \Rightarrow q) \wedge \neg q$	$\neg p$	$(p \Rightarrow q) \wedge \neg q \Rightarrow \neg p$
true	*true*	*true*	*false*	*false*	*false*	*true*
true	*false*	*false*	*true*	*false*	*false*	*true*
false	*true*	*true*	*false*	*false*	*true*	*true*
false	*false*	*true*	*true*	*true*	*true*	*true*

Esta forma de razonamiento tiene un nombre en la literatura clásica sobre lógica: *modus tollens*

Veamos ahora otro razonamiento capilar:

$$\frac{\textit{Si tuviera pelo sería feliz} \quad \textit{No tengo pelo}}{\textit{No soy feliz}}$$

si lo traducimos a su forma simbólica como lo hicimos anteriormente

$$\frac{p \Rightarrow q \quad \neg p}{\neg q}$$

y construimos la tabla de verdad correspondiente, observamos que no es una tautología:

p	q	$p \Rightarrow q$	$\neg p$	$(p \Rightarrow q) \wedge \neg p$	$\neg q$	$(p \Rightarrow q) \wedge \neg p \Rightarrow \neg q$
true	*true*	*true*	*false*	*false*	*false*	*true*
true	*false*	*false*	*false*	*false*	*true*	*true*
false	*true*	*true*	*true*	*true*	*false*	*false*
false	*false*	*true*	*true*	*true*	*true*	*true*

Más aún, mirando la tabla de verdad es posible saber cuándo las premisas son verdaderas y la conclusión es falsa. En este caso, esto ocurre cuando p es falsa y q es verdadera, es decir cuando tenemos un pelado feliz.

Si bien este método para analizar razonamientos es efectivo para razonamientos con pocas variables booleanas, a medida que la cantidad de variables crece, el tamaño de las tablas de verdad crece de manera exponencial (hay 2^n filas en una tabla de verdad con n variables booleanas). En las secciones siguientes veremos métodos alternativos para determinar la validez de razonamientos y de expresiones booleanas en general, los cuales a veces resultan más cortos que las tablas de verdad.

2.7. Resolución de acertijos lógicos

Haremos ahora una visita a la isla de los *caballeros* y los *pícaros*. Esta isla está habitada solamente por estos dos tipos de gente. Los caballeros tienen la particularidad de que sólo dicen la verdad, mientras que los pícaros siempre mienten.

(2.8) **Problema.** Tenemos dos personas, A y B habitantes de la isla. A hace la siguiente afirmación: "*Al menos uno de nosotros es pícaro*". ¿Qué son A y B?

Solucï¿½n *Supongamos que A es un pícaro. Luego la afirmación es falsa y ninguno de ellos puede ser un pícaro. Pero esto contradice la suposición acerca de A, por lo tanto A es un caballero. Ahora, dado que A es un caballero, su afirmación es cierta. Pero entonces entre él y B hay al menos un pícaro. Ya habíamos visto que A sólo puede ser un caballero. Por lo tanto B es un pícaro.*

(2.9) **Problema.** Nuevamente tenemos dos personas, A y B habitantes de la isla. A dice: "*Soy un pícaro pero B no lo es*". ¿Qué son A y B?

Solucï¿½n *A no puede ser un caballero, dado que estaría diciendo que es un pícaro (y algo más). Luego A es un pícaro, con lo cual su frase es falsa. Luego dado que A es efectivamente un pícaro, es necesariamente falso que B no lo es. Por lo tanto B es también un pícaro.*

2.8. Ejercicios

2.1 Evaluar las siguientes expresiones en el estado $\{(p, true), (q, false), (r, false)\}$

a) $(p \vee q) \wedge r$

b) $(p \wedge q) \vee r$

c) $p \vee (q \wedge r)$

d) $p \equiv (q \equiv r)$

e) $(p \equiv q) \equiv r$

f) $(p \Rightarrow q) \Rightarrow r$

g) $(p \wedge q) \Rightarrow r$

2.2 Escribir las tablas de verdad para las siguientes expresiones, determinando los casos en los cuales son tautologías o contradicciones.

a) $(p \vee q) \vee \neg q$

b) $(p \wedge q) \Leftarrow \neg q$

c) $(p \not\equiv q) \wedge (p \vee q)$

d) $p \equiv (q \equiv p)$

e) $(p \not\equiv q) \not\equiv p$

f) $(p \Rightarrow q) \Rightarrow p$

g) $\neg q \wedge \neg p \equiv (q \Rightarrow p) \Rightarrow q$

h) $(p \equiv p \vee \neg p) \Rightarrow p$

2.3 Escribir las expresiones duales P_D de cada una de las expresiones booleanas P que se indican

a) $b \vee c \vee true$

b) $b \wedge c \wedge d$

c) $b \wedge (c \vee \neg d)$

d) $b \vee (c \wedge d)$

e) $\neg false \Rightarrow b \vee c$

f) $\neg b \Leftarrow b \vee c$

g) $(\neg b \equiv true) \vee b$

h) $(b \equiv c) \equiv (b \Rightarrow c) \wedge (c \Rightarrow b)$

2.4 Para cada una de las expresiones $P \equiv Q$ que se indican a continuación, escribir la correspondiente expresión $P_D \equiv Q_D$.

a) $p \equiv q$

b) $p \wedge p \equiv p$

c) $p \Rightarrow p \equiv true$

d) $p \Rightarrow q \equiv \neg p \vee q$

e) $true \Rightarrow p \equiv p$

f) $false \Rightarrow p \equiv true$

g) $p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$

h) $p \equiv q \equiv q \equiv p$

2.5 Traducir las siguientes frases en expresiones booleanas

a) Llueva o no, iré a nadar.

b) Llueve, no iré a nadar.

c) Llueven rayos y centellas.

d) Llueven rayos o centellas.

e) Llueven rayos y centellas pero iré a nadar.

2.6 Traducir las siguientes frases en expresiones booleanas

a) Ninguno entre p y q es verdadero.

b) Exactamente uno entre p y q es verdadero.

c) Cero, dos o cuatro entre p, q, r o s son verdaderos.

d) Uno o tres entre p, q, r o s son verdaderos.

2.7 Identificar las proposiciones elementales en las siguientes frases y traducirlas en expresiones booleanas.

a) $x < y$ o $x = y$.

b) $x < y$ o $x = y$ o $x > y$.

c) Si $x > y$ e $y > z$ entonces $v = w$.

d) Las siguientes expresiones son todas verdaderas: $x < y$, $y < z$ y $v = w$.

e) A lo sumo una de las siguientes expresiones es verdadera: $x < y$, $y < z$ y $v = w$.

f) Ninguna de las siguientes expresiones es verdadera: $x < y$, $y < z$ y $v = w$.

g) Las siguientes expresiones no son todas verdaderas al mismo tiempo: $x < y$, $y < z$ y $v = w$.

h) Cuando $x < y$ entonces $y < z$; cuando $x \geq y$ entonces $v = w$.

i) Cuando $x < y$ entonces $y < z$ significa que $v = w$, pero si $x \geq y$ entonces $y > z$ no ocurre; sin embargo si $v = w$ entonces $x < y$.

j) Si la ejecución del programa P comenzó con $x < y$, entonces la ejecución termina con $y = 2^x$.

k) La ejecución del programa P que comenzó con $x < 0$ no terminará.

2.8 En la isla de los pícaros y los caballeros, A dice *"O bien soy un pícaro o bien B es un caballero"*. ¿Qué son A y B?

2.9 Traducir los siguientes razonamientos presentados en lenguaje natural a expresiones booleanas y analizar su validez

a) Si el presidente entiende las protestas de la gente entonces si quiere ser reelegido cambiará su política. El presidente quiere ser reelegido. Luego, si el presidente entiende las protestas de la gente, entonces cambiará su política.

b) Si el presidente entiende las protestas de la gente entonces si quiere ser reelegido cambiará su política. El presidente cambiará su política. Luego, si el presidente entiende las protestas de la gente, entonces quiere ser reelegido.

c) Si el gobernador quiere mejorar su imagen, o bien mejora su política social o bien gasta más en publicidad. El gobernador no mejora su política social. Luego, si el gobernador quiere mejorar su imagen, entonces gastará más en publicidad.

d) Si el gobernador quiere mejorar su imagen, o bien mejora su política social o bien gasta más en publicidad. El gobernador mejoró su política social. Luego, si el gobernador quiere mejorar su imagen, entonces no gastará más en publicidad.

e) Si la ciudadanía romana hubiera sido una garantía de los derechos civiles, los romanos habrían gozado de libertad religiosa. Si los romanos hubieran gozado de libertad religiosa, entonces no se habría perseguido a los primeros cristianos. Pero los primeros cristianos fueron perseguidos. Por consiguiente, la ciudadanía romana no puede haber sido una garantía de los derechos civiles.

f) Si la descripción bíblica de la cosmogonía es estrictamente correcta, el Sol no fue creado hasta el cuarto día. Y si el Sol no fue creado hasta el cuarto día, no puede haber sido la causa de la sucesión del día y la noche durante los tres primeros días. Pero o bien la Biblia usa la palabra *día* en un sentido diferente al aceptado corrientemente en la actualidad, o bien el Sol debe haber sido la causa de la sucesión del día y la noche durante los tres primeros días. De esto se desprende que, o bien la descripción bíblica de la cosmogonía no es estrictamente correcta, o bien la palabra *día* es usada en la Biblia en un sentido diferente al aceptado corrientemente en la actualidad.

g) Se está a favor de la pena de muerte por miedo al delito o por deseo de venganza. Aquellos que saben que la pena de muerte no disminuye el delito, no están en favor de la pena de muerte por miedo al delito. Por lo tanto aquellos que están a favor de la pena de muerte, no saben que ésta no disminuye el delito o tienen deseo de venganza.

2.10 Resolver los siguientes acertijos lógicos, dando para cada uno de ellos el modelo del problema y la solución o soluciones que se obtienen a las preguntas planteadas.

a) Detrás de una de tres puertas hay un millón de dólares.

- La **primer puerta** tiene un cartel que dice: “Está Acá”.
- La **segunda puerta** tiene un cartel que dice: “No está Acá”.
- La **tercera puerta** tiene un cartel que dice: “No está en la primera puerta”.

Si se sabe que a lo sumo uno de los carteles es verdadero, ¿Detrás de qué puerta está el millón?

b) Utilizando el mismo planteo que el ítem anterior, pero ahora los carteles en las puertas dicen:

- En la **primer puerta**: “No está en la segunda puerta”.
- En la **segunda puerta**: “No está acá”.
- En la **tercer puerta**: “Está acá”.

Esta vez, se sabe que hay al menos un cartel falso, y al menos un cartel verdadero. Encontrar en que puerta esta el millón.

c) Una pareja deja a sus cuatro chicos, Alberto, Beatriz, Carlos y Diana, con una niñera. Le aclaran que uno de ellos siempre dice la verdad, pero que los otros tres siempre mienten. Le dicen quien dice siempre la verdad, pero la niñera no presta atención y se olvida. Mientras prepara la comida, escucha que se rompe un vaso.

- **Alberto** dice: "Beatriz lo hizo".
- **Beatriz** dice: "Diana lo hizo".
- **Carlos** dice: "Yo no lo hice".
- **Diana** dice: "Beatriz mintió cuando dijo que yo lo hice".

¿Quién es el culpable?

d) Alberto, Bernardo y Carlos son caballeros o pícaros. Los pícaros mienten siempre, y los caballeros siempre dicen la verdad.

- **Alberto** dice: "Bernardo y Carlos son pícaros".
- **Bernardo** dice: "No soy un pícaro".
- **Carlos** dice: "Bernardo es un pícaro".

¿Cuántos son pícaros?

e) Tres prisioneros, uno con **vista normal**, un **tuerto** y un **ciego**, están en una cárcel. El sádico guardia, para divertirse, dice que les dará un problema, y el que lo resuelva quedará libre. Les dice que tiene cinco sombreros, con un círculo marcado al frente de forma tal que el que tiene el sombrero puesto no puede ver el círculo, pero los demás pueden verlo sin problema. Hay tres sombreros con un círculo blanco y dos con un círculo rojo. El guarda apaga la luz, elije tres sombreros y se los pone a los prisioneros. Luego prende la luz, y les dice que si adivinan el color del círculo de su sombrero, saldrán libres, pero si adivinan mal, los matará de inmediato. Si no pueden adivinar, no pasa nada. Le da la primera oportunidad al prisionero que con **vista normal**, el cual mira los otros dos sombreros, y decide no arriesgarse. Luego, le da la oportunidad al **tuerto**, el cual prefiere tampoco arriesgarse. Con lo cual el guarda dice, "bueno, volvamos a las celdas". El **ciego** inmediatamente dice : "que, ¿y yo no puedo adivinar?". El guarda se ríe, y le dice que puede si quiere. El ciego adivina correctamente y sale libre.

¿De qué color era su sombrero?

f) Dos personas se dicen del mismo tipo si son ambas caballeros o ambas pícaros. Tenemos tres personas, A, B y C, y sabemos que:

- **A** dice: "B es un pícaro".
- **B** dice: "A y C son del mismo tipo".

¿Qué es C?

g) Alberto, Bernardo y Carlos son tres políticos que tienen que votar por una ley. Todos votan.

- Si Alberto vota que sí, entonces Bernardo y Carlos votan igual entre ellos.

- Si Bernardo vota que sí, entonces Alberto vota lo contrario de Carlos.
- Si Carlos vota que no, entonces Alberto y Bernardo votan igual entre ellos.

¿Se puede saber el voto de los tres? ¿Se puede saber el voto de alguno de los tres? ¿De quién o quiénes?

CAPÍTULO 3
Cálculo Proposicional

Índice del Capítulo

3.1. El sistema MIU 44
3.2. El sistema mg 48
3.3. Sistemas formales 51
3.3.1. La equivalencia 54
3.3.2. La negación, discrepancia, y false 56
Técnicas de demostración y principios 57
3.3.3. La disyunción 59
3.3.4. La conjunción 60
Usando la Regla dorada 62
Usando lemas 63
3.3.5. La implicación 64
Prueba de teoremas en donde participa la implicación 67
La Regla de leibniz como axioma 68
3.4. Ejercicios 69

En este capítulo se presenta una alternativa a las evaluaciones (tablas de verdad) para razonar con expresiones booleanas. En el capítulo anterior pudimos observar que las tablas de verdad se pueden construir de manera mecánica, sin embargo pueden ser extremadamente largas si el número de variables es grande, con lo cual determinar si una expresión booleana es válida, resulta frecuentemente, un problema tedioso y difícil de manipular. Con este fin introducimos un sistema para construir demostraciones que involucren expresiones de la lógica proposicional. El estilo utilizado se conoce como *cálculo proposicional* y está orientado a la programación, por lo cual provee herramientas para el manejo efectivo de fórmulas lógicas de tamaño considerable. El objetivo de un sistema formal consiste en explicitar un lenguaje a través del cual se realizarán

demostraciones y también definir las reglas para realizarlas. Esto permite tener una noción muy precisa de lo que se entiende por una demostración, como así también la posibilidad de precisar la sintaxis y la semántica.

3.1. El sistema MIU

Trataremos ahora de introducir la idea de sistema formal a través de un acertijo propuesto por Douglas R. Hofstadter en su libro "Gödel, Escher, Bach". El sistema formal que presentaremos ahora corresponde a una clase de sistemas inventada por el lógico americano Emil Post y conocida como "Sistema de producción de Post".

Supongamos que disponemos solamente de tres letras del alfabeto: M, I, U. La sucesión de estas letras o símbolos uno tras otro constituye una *cadena*. En toda cadena, las letras están situadas en un orden establecido, por ejemplo las cadenas MI e IM son dos cadenas diferentes. Algunas de las cadenas que pueden formarse con los símbolos M, I y U son:

- MU
- UIM
- MUUMUU
- UIIMIUUIMMIIUMIIUM

Proponemos a continuación un juego en donde el jugador tiene en su poder la cadena MI y mediante una serie de reglas precisas debe producir otras cadenas. Si bien las cadenas mencionadas arriba son todas cadenas legítimas (pues pueden construirse con los símbolos disponibles) aún no pertenecen a la colección privada del jugador, éste sólo dispone de MI, y sólo a través de las reglas que enunciaremos puede ampliar su colección. He aquí la primera regla:

Regla 1: Si se tiene una cadena cuya última letra es I, se le puede agregar una U al final.

Para enunciar la segunda regla recurrimos al concepto de variable, entendiendo que con la letra x no estamos ampliando la cantidad de símbolos disponibles en el juego sino que llamaremos x a una cadena determinada que sí fue construída con los símbolos permitidos.

Regla 2: Supongamos que se tiene la cadena Mx, entonces puede agregarse la cadena Mxx a la colección.

Veamos unos ejemplos de la aplicación de esta regla:

- Dado MIU se puede obtener MIUIU
- Dado MUM se puede obtener MUMUM

- Dado MU se puede obtener MUU

Observemos que esta regla no dice que MUU está en poder del jugador, sino que lo está si antes estuvo MU. Es decir, primero es necesario obtener MU para luego incorporar MUU.

Regla 3: Si en una cadena de la colección aparece la secuencia III, puede elaborarse una nueva cadena sustituyendo III por U.

Ejemplos:

- Dado UMIIIMU se puede construir UMUMU
- Dado MIIII se puede construir MIU o también MUI
- Dado IIMII no podemos aplicando esta regla construir nada, los símbolos III deben ser consecutivos
- Dado MIII se elabora MU

Regla 4: Si aparece UU en el interior de una cadena, está permitido eliminarlo y formar así otra cadena.

- Dado UUU, se obtiene U
- Dado MUUUIII se obtiene MUIII

Esto es todo lo que hay que saber para obtener cadenas en la colección de cada jugador, las cadenas generadas mediante el empleo de las reglas mencionadas se llaman *teoremas*. El sentido del término "teorema" es aquí distinto al que es común en matemática, que llama de esa manera a las afirmaciones formuladas en lenguaje corriente cuya veracidad ha sido probada por medio de una demostración rigurosa. En los sistemas formales, en cambio, no hay necesidad de considerar a los teoremas como afirmaciones, sino que éstos son simplemente una cadena de símbolos y por otra parte, no son demostrados sino producidos, como si los elaborara una máquina de acuerdo a determinadas reglas predefinidas. Cuando comenzamos este juego dijimos que MI es una cadena disponible para el jugador, en este sentido MI es entonces un teorema, pero en realidad no fue necesario aplicar ninguna de las reglas mencionadas para conseguirlo. Distinguimos entonces a los teoremas que están disponibles desde el comienzo del juego llamándolos *axiomas*. Un sistema formal puede tener cero, uno, varios o infinitos axiomas. Todo sistema formal cuenta con reglas de derivación de símbolos tales como las cuatro reglas del sistema MIU. Estas reglas son denominadas *reglas de inferencia*. Por último el concepto que ilustraremos ahora es el de derivación. Lo que sigue es una derivación del sistema MIU:

(1) MI	axioma
(2) MII	de (1) y Regla 2
(3) MIIII	de (2) y Regla 2
(4) MIIIIU	de (3) y Regla 1
(5) MUIU	de (4) y Regla 3
(6) MUIUUIU	de (5) y Regla 2
(6) MUIIU	de (6) y Regla 4

Una *derivación* de un teorema es una demostración explícita y paso a paso del modo en que se ha producido el mismo, de acuerdo a los axiomas y las reglas de inferencia del sistema formal. En el ejemplo anterior podemos decir que hemos probado MUIIU o bien que se ha *derivado* MUIIU dentro del sistema MIU. Por lo tanto MUIIU es un teorema del sistema MIU.

Vamos a plantear ahora un acertijo:

Problema 1: ¿Es MU un teorema del sistema MIU?

Antes de continuar, intente el lector derivar MU a partir de la cadena MI, usando las cuatro reglas de inferencia. Una manera de dar respuesta a este acertijo es a través de la derivación azarosa de gran cantidad de teoremas, sólo para ver qué sucede. Si se emplea este método, pronto se descubren ciertas propiedades en los teoremas que se han elaborado. Por ejemplo, no es obvio que todos los teoremas comienzan con M hasta que no se realizan unos pocos intentos. Y luego, una vez descubierto el modelo, se puede constatar que efectivamente así ocurrirá con todos los teoremas a través del análisis de las reglas, cuyo atributo es obligar a que cada nuevo teorema reciba su primera letra de un teorema precedente; con lo cual, en todos los casos las letras iniciales de los teoremas pueden ser obtenidas a partir de la primera letra del axioma MI: y esto constituye una demostración de que todos los teoremas del sistema MIU deben comenzar con M.

Lo anterior muestra una diferencia entre seres humanos y máquinas. Es perfectamente posible y muy fácil en realidad, programar una computadora para que genere teorema tras teorema del sistema MIU, y podríamos incluir en la programación la orden de que sólo se detenga al generar U. Nuestro análisis previo sobre los teoremas del sistema, nos permite concluir que una computadora así programada nunca se detendría. Es posible programar una máquina para que realice tareas rutinarias de modo tal que la máquina jamás advierta ni siquiera los hechos más obvios vinculados con lo que se está haciendo, pero en cambio percibir estos hechos es algo inherente a la conciencia humana. La inobservancia es un rasgo característico de las máquinas.

Uno de los atributos inherentes de la inteligencia es la capacidad de alejarse mediante un salto de lo que se está haciendo con el objeto de examinarlo. En todos los casos, esto es buscar, y a menudo con éxito, modelos. Cuando se estudian sistemas formales, es importante distinguir entre lo que se hace *dentro* del sistema, por un lado, y por otro los enunciados u observaciones que se formulan *acerca* del sistema. Cualquiera que aborde el acertijo MU, lo hace manteniéndose dentro del sistema, pero luego, poco a poco, se va impacientando cada vez más hasta el punto de salirse del sistema, sin pensarlo mucho, tratando de evaluar los resultados obtenidos y preguntándose por qué no ha podido derivar MU. Quizá descubrió una razón para explicar por qué no obtuvo MU: esto es pensar acerca del sistema. Quizá produjo MU en algún punto del desarrollo que elaboró: esto es actuar dentro del sistema.

Se puede observar que el sistema MIU, comprende reglas con características opuestas: *reglas ampliadoras* y *reglas reductoras*. Hay dos reglas (1 y 2) que permiten aumentar la extensión de las cadenas; y hay otras que permiten reducirlas un tanto. Parece haber una variedad interminable de instancias en las cuales pueden ser aplicados estos diferentes géneros de reglas, y como consecuencia se debe esperar que, de una manera u otra, se termine produciendo MU. Pero no existe garantía alguna acerca de lo que puede ocurrir. Ciertamente ya hemos observado

que U no puede ser producida de ninguna forma, no obstante el caso U y el caso MU parecen ser muy diferentes. Un rasgo muy obvio de U es que no la precede ninguna M, y es de mucha utilidad apelar a este método tan simple para detectar los no teoremas, pero ¿quién asegura que esta *verificación* permitirá detectar *todos* los no teoremas? Hay incontables cadenas que comienzan con M y no son producibles, quizá MU es una de ellas. Esto significa que la prueba de la primera letra es de utilidad limitada. Queda la posibilidad, sin embargo de que una verificación algo más elaborada distinga los teoremas de los no teoremas.

Imaginemos un genio, que tiene todo el tiempo del mundo y disfruta produciendo teoremas del sistema MIU, de un modo metódico. Lo que sigue es un posible método que el genio emplearía:

Paso 1: Emplear todas las reglas aplicables al axioma MI. Esto produciría dos nuevos teoremas: MIU y MII.

Paso 2: Emplear todas las reglas aplicables a los teoremas obtenidos en el paso 1. Esto produciría tres nuevos teoremas: MIIU, MIUIU, MIIII.

Paso 3: Emplear todas las reglas aplicables a los teoremas obtenidos en el paso 2. Esto produciría cinco nuevos teoremas: MIIIIU, MIIUIIU, MIUIUIUIU, MIIIIIIII, MUI.

⋮

Tarde o temprano, este método producirá todos los teoremas, porque las reglas son aplicadas en todos los órdenes concebibles, sin embargo no está claro cuánto habrá que esperar para que aparezca una cadena determinada. En síntesis, la verificación propuesta consiste entonces en esperar hasta que la cadena en cuestión se produzca, en cuyo caso es un teorema. Y si no se produce nunca es un no teorema. Esto es ridículo, pues supone que estaríamos dispuestos a esperar un lapso de tiempo infinito. Un *procedimiento de decisión* correspondiente al sistema formal que se trate, es una una verificación de la "teoremidad" que se completa dentro de un tiempo finito. Cuando se cuenta con un procedimiento de decisión, se tiene con él una caracterización muy concreta de todos los teoremas del sistema. A primera vista, puede parecer que las reglas y axiomas del sistema formal proveen una caracterización no menos completa que la aportada por el procedimiento de decisión, pero estas reglas sólo aportan una caracterización tácita de los teoremas. Cuando se dispone de un procedimiento de decisión en un sistema formal se dispone de una verificación de la "teoremidad" de una cadena que por más complicada que resulte, tiene garantía de finalizar.

Además, los sistemas formales cuentan con un método preciso que permite decidir si una cadena cualquiera es o no un axioma. Esta es la diferencia que separa el conjunto de los axiomas del conjunto de los teoremas: el primero siempre está dotado de un procedimiento de decisión que asegura al menos al comienzo, la posibilidad de ingreso al campo de juego, mientras que para el segundo, no tenemos garantía de la existencia de tal proceso.

En cuanto al sistema MIU, observamos que el único axioma es conocido y las reglas de inferencia son simples, sin embargo no está claro si MU es o no un teorema.

3.2. El sistema mg

Para ampliar la perspectiva sobre sistemas formales presentamos ahora otro sistema también presentado por Douglas R. Hofstadter en "Gödel, Escher, Bach." El sistema mg cuenta con tres símbolos:

$$\mathrm{m} \quad - \quad \mathrm{g}$$

Es decir las letras m y g y el guión.

El sistema mg tiene una cantidad infinita de axiomas, en consecuencia vamos a definir los axiomas a través de lo que se conoce como un *esquema de axioma*. Es decir, una especie de matriz que moldea todos los axiomas del sistema y de este modo pone a nuestra disposición un procedimiento que permite determinar si una cadena dada de los símbolos m, g y $-$ constituye un axioma. Para esto recurrimos nuevamente al concepto de variable, y esta vez con x estaremos indicando una secuencia de guiones.

Definición $x\mathrm{m}-\mathrm{g}x-$ es un axioma, siempre que con x estemos representando una cadena de guiones.

Por ejemplo la cadena $--\mathrm{m}-\mathrm{g}---$ es un axioma, mientras que la cadena $\mathrm{xm}-\mathrm{gx}$ no es un axioma puesto que x no pertenece a los símbolos disponibles en el sistema mg.

El sistema mg tiene solamente una regla de inferencia.

Regla 1: Si x, y y z representan cadenas de guiones y $x\mathrm{m}y\mathrm{g}z$ es un teorema entonces también es un teorema $x\mathrm{m}y-\mathrm{g}z-$.

Por ejemplo: si x vale "$--$", y vale "$---$" y z vale "$-$", la regla nos dice que:

Si $--\mathrm{m}---\mathrm{g}-$ es un teorema, entonces $--\mathrm{m}-----\mathrm{g}--$ es un teorema.

Tal como ocurre siempre con las reglas de inferencia, ésta establece una vinculación entre la "teoremidad" de ambas cadenas, pero no afirma la "teoremidad" de ninguna de ellas por sí misma. Pero supongamos que se nos presenta el problema:

Problema 1: Determinar si $--\mathrm{m}---\mathrm{g}-$ es un teorema.

Solucïï¿½n *Primero, observemos que no se trata de un axioma, pues el esquema establece que $xm-gx-$ es un axioma para x cadena de guiones, y de acuerdo a la forma de la cadena $--m---g-$ esto no puede ocurrir. Ahora analicemos si fue posible obtener $--m---g-$ usando la regla de inferencia 1, pero en ese caso x vale "$--$", y vale "$--$" y z no representa ningún guión, y es claro que ningún axioma es de la forma $xmyg$ en el sistema mg. Con lo cual la cadena $--m---g-$ no es un teorema.*

Problema 2: Determinar si $--\mathrm{m}---\mathrm{g}------$ es un teorema.

Solucïï¿½n *Aquí otra vez descartamos la posibilidad de axioma. Si se hubiera usado la regla de inferencia, para obtener $--m---g-----$, entonces x vale "$--$", y vale "$--$" y z vale "$----$", y la cadena más breve de partida debería ser $--m--g----$.que tampoco resulta un axioma, si aplicamos la regla de inferencia otra vez en sentido opuesto, esto nos da que x vale "$--$", y vale "$-$" y z "$---$", así llegamos a la cadena más corta $--m-g---$, que resulta un axioma. Por lo tanto $--m---g-----$ es un teorema.*

Supongamos ahora que tenemos delante nuestro una cadena cualquiera construída mediante los símbolos permitidos en el sistema mg y queremos saber si se trata de un teorema. En primer lugar, observemos que todos los teoremas del sistema mg tiene tres grupos separados de guiones, y que los elementos de separación son únicamente una m, y una g ubicadas en ese orden. La clase de razonamiento utilizado para sacar esta conclusión es similar al realizado para probar que todos los teoremas del sistema MIU tenían que comenzar con M. Lo anterior significa que, considerando solamente su forma podemos excluir cadenas como: $--$m$--$m$--$m$--$g$---$.

Ahora bien, llamaremos *cadena bien formada* a toda cadena que empiece con un grupo de guiones, tenga luego una m, seguida por un segundo grupo de guiones, luego una g, y por último otro grupo de guiones.

Volvemos entonces al problema de determinar la "teoremidad" de una cadena dada, suponemos entonces que se trata de una cadena bien formada, pues en caso contrario no vale la pena continuar el análisis. Podemos asegurar de acuerdo a los pasos analizados en los dos problemas anteriores que lo primero por hacer es verificar si se trata de un axioma, si es un axioma es por definición un teorema y no hace falta verificarlo, pero supongamos que no es un axioma, entonces para ser un teorema debe provenir de una cadena más breve a través de la aplicación de la regla de inferencia. Observemos que esta regla sólo produce cadenas más largas, con lo cual es posible establecer cuáles cadenas más cortas están en condiciones de ser antecesoras de la cadena en cuestión. De este modo el problema se reduce a determinar si alguna de estas nuevas cadenas más breves es un teorema y cada una de estas últimas puede ser sometida al mismo proceso, lo peor que puede ocurrir es la proliferación de cadenas más breves. Si se continúa con este proceso se logrará llegar muy cerca de la fuente de todos los teoremas: el esquema de axioma, entonces o bien se descubre que una de las cadenas cortas aparecidas es un axioma o bien se llega a la conclusión que ninguna de las cadenas más reducidas es un axioma y tampoco permiten ser reducidas de acuerdo a la regla. Es decir, el sistema mg cuenta con un *proceso de decisión* sobre la "teoremidad" de una cadena dada. Un sistema formal que nos indique cómo elaborar teoremas más prolongados a partir de otros más breves, pero nunca lo inverso, es seguro que incluye un proceso de decisión aplicable a sus teoremas.

Supongamos ahora que realizamos una biyección entre palabras y símbolos del sistema mg:

$$\begin{array}{lcl} \text{m} & \Leftrightarrow & \text{más} \\ \text{g} & \Leftrightarrow & \text{igual a} \\ - & \Leftrightarrow & \text{uno} \\ -- & \Leftrightarrow & \text{dos} \\ --- & \Leftrightarrow & \text{tres} \\ & & \text{etc.} \end{array} \tag{3.1}$$

esta correspondencia entre palabra y símbolo, tiene un nombre: *interpretación*.

La cadena $--m---g-----$ es un teorema de acuerdo a lo que vimos en el problema 2, y de acuerdo a la biyección dada en la tabla de arriba, podría enunciarse diciendo "dos más tres es igual a cinco", en tanto que la cadena analizada en el problema 1 no es un teorema y puede enunciarse "dos más tres no es igual a uno". Posiblemente, haya resultado bastante evidente que los teoremas del sistema mg son semejantes a la suma, observando que el esquema de axioma admite axiomas que satisfacen el criterio de adición y la regla de inferencia asegura que si la primera cadena satisface el criterio de adición la segunda también lo hará.

Se podría entonces pensar que el teorema $--m---g-----$ es un enunciado escrito mediante una notación particular, cuyo *significado* es que 2 más 3 es 5. ¿Es esta una manera razonable de analizar las cosas? ¿La cadena $--m---g-----$ en verdad significa "2 más 3 es igual a 5"?

En la tabla 3.1 vimos una interpretación de los símbolos del sistema mg, que constituye un "nivel inferior" de nuestra biyección, por otra parte, en un nivel más alto se sitúa la correspondencia entre proposiciones verdaderas y teoremas. Pero debe tenerse en cuenta que esta correspondencia de nivel superior puede no ser advertida si no se establece previamente una interpretación de los símbolos. Sería más preciso entonces hablar de correspondencia entre proposiciones verdaderas y teoremas *interpretados*.

Cuando uno se encuentra con un sistema formal del que no se conoce nada, con la esperanza de descubrir en él alguna significación recóndita, el problema se traduce en cómo asignar interpretaciones significativas a sus símbolos: en otros términos, cómo hacerlo de modo tal que surja una correspondencia de nivel superior entre proposiciones verdaderas y teoremas.

Es posible elegir otras interpretaciones distintas a las que aparecen en la tabla 3.1, una elección al azar corresponderá por ejemplo a :

$$\begin{array}{lcl} \text{m} & \Leftrightarrow & \text{caballo} \\ \text{g} & \Leftrightarrow & \text{manzana} \\ - & \Leftrightarrow & \text{feliz} \end{array} \tag{3.2}$$

Tenemos así una nueva interpretación para $-m-g--$:"manzana caballo manzana feliz manzana manzana". Lamentablemente, esta interpretación tiene muy escasa "significación", los teoremas no dan la impresión de ser más verdaderos o más aceptables que los no teoremas. En consecuencia, cabe distinguir entre dos tipos diferentes de interpretaciones. En primer lugar, podemos encontrarnos con una interpretación *no significativa*, bajo la cual no se advierte la menor asociación biyectiva entre los teoremas del sistema formal y la realidad. Un ejemplo de esto aparece en la tabla 3.2. Mientras que llamaremos *significativa* a la segunda clase de interpretación, bajo ésta, los teoremas y las verdades se corresponden: es decir existe una biyección entre los teoremas del sistema y una porción de la realidad. Por lo tanto la interpretación dada en la tabla 3.1 resulta significativa.

Es por ello que conviene distinguir entre *interpretaciones* y *significados*. Cualquier palabra podría haber sido adoptada como interpretación de "m", pero se adoptó "más" porque es la única elección significativa en la que se puede pensar. En síntesis, el significado de "m" parece ser "más", pese a que sea posible asignarle un millón de interpretaciones diferentes.

Volviendo al sistema mg, éste parece obligarnos a reconocer que los símbolos de un sistema

formal, aunque inicialmente carezcan de significado, no pueden evitar tomar alguna clase de "significado", en cuanto se descubre la biyección.

Ahora bien, hay una diferencia muy grande entre el significado relativo a un sistema formal, y el vinculado al lenguaje: cuando hemos aprendido el significado de una palabra dentro de un dado idioma, pasamos a elaborar nuevos enunciados basados en ese significado. Hasta cierto punto el significado se convierte en *activo*, ya que actúa como una regla de creación de frases. Esto quiere decir que nuestro dominio del idioma no se asemeja a un producto terminado: las reglas de elaboración de frases se multiplican en la medida en que aprendemos nuevos significados. En un sistema formal, en cambio, los teoremas son definidos a priori por las reglas de inferencia. Podemos elegir "significados" que se funden en una biyección (si es posible encontrarla) entre teoremas y proposiciones verdaderas, pero ello no nos autoriza a extender el campo, agregando nuevos teoremas a los ya establecidos.

En el sistema MIU, por ejemplo, no había motivo para sentir la tentación de ir más allá de las cuatro reglas, porque no se buscó ni se descubrió ninguna interpretación. Pero aquí, en el sistema mg, los "significados" recién hallados para cada símbolo pueden llevarnos a pensar que la cadena

$$--\text{m}--\text{m}--\text{m}--\text{g}---------$$

es un teorema. Cuando menos, uno puede *desear* que esta cadena sea un teorema, pero eso no cambia el hecho de que no lo es. Y también constituye un serio error pensar que *debe* ser un teorema, sólo porque 2 más 2 más 2 más 2 es igual a 8. Tampoco sería correcto atribuirle un significado, puesto que no es una cadena bien formada, y nuestra interpretación significativa procede exclusivamente de la observación de cadenas bien formadas.

En un sistema formal, el significado debe permanecer *pasivo*, podemos leer cada cadena siguiendo los significados de los símbolos que la integran, pero no estamos facultados para crear nuevos teoremas sobre la única base de los significados que hemos asignado a los símbolos. Los sistemas formales interpretados se encuentran en la frontera que separa los sistemas sin significado de los sistemas con significado: puede pensarse que sus cadenas expresan cosas, pero es imprescindible tener en cuenta que ello ocurre exclusivamente como consecuencia de las propiedades formales del sistema.

3.3. Sistemas formales

Los ejemplos anteriores pretendían ilustrar el concepto de sistema formal, podemos decir ahora que:

Un *sistema formal* es un conjunto de reglas definidas en términos de:

- Un conjunto de símbolos llamado *alfabeto*.
- Un conjunto de *expresiones bien formadas o fórmulas*, construídas a partir de los símbolos.
- Un conjunto de fórmulas distinguibles, llamadas *axiomas*.

- Un conjunto de *reglas de inferencia*.

El conjunto de fórmulas es llamado lenguaje del sistema formal. El lenguaje se define sintácticamente, no existe la noción de significado o semántica en un sistema formal en sí. Las reglas de inferencia permiten derivar fórmulas de otras fórmulas. Una fórmula es un teorema del sistema formal si es un axioma, o puede ser generado a partir de un axioma o de algún teorema ya demostrado usando las reglas de inferencia. Una demostración de que una fórmula es un teorema es un argumento que muestra cómo las reglas de inferencia son utilizadas para producir la fórmula.

El cálculo proposicional se presentará como un sistema formal, vamos ahora a dar un detalle de los elementos señalados arriba que serán los que conformen el sistema formal con el que trabajaremos:

Alfabeto: El alfabeto con el que construiremos las expresiones o palabras estará constituído por los siguientes símbolos:

constantes: las constantes *true* y *false* se usarán para los valores verdadero y falso respectivamente.

variables: las variables proposicionales o booleanas que representarán los valores *true* o *false*. Se usarán generalmente las letras p, q, y r para nombrar a estas variables.

operador unario: negación $\neg$.

operadores binarios:

equivalencia	$\equiv$
disyunción	$\vee$
conjunción	$\wedge$
discrepancia	$\not\equiv$
implicación	$\Rightarrow$
consecuencia	$\Leftarrow$

signos de puntuación: paréntesis "(" y ")".

Fórmulas: Las *expresiones bien formadas o fórmulas* del cálculo proposicional serán las que se obtienen a partir de las siguientes reglas:

i) Las variables proposicionales y las constantes son fórmulas.

ii) Si E es una fórmula, entonces $(\neg E)$ es también una fórmula.

iii) Si E y F son fórmulas y $\oplus$ es un operador binario, entonces $(E \oplus F)$ es una fórmula.

iv) Sólo son fórmulas las construídas con las reglas precedentes.

Ya vimos en los capítulos anteriores la definición de expresión booleana, y también mencionamos el rol fundamental que éstas cumplen en la definición de un lenguaje artificial libre de las ambigüedades y contradicciones usuales del lenguaje corriente. Observemos que el conjunto de fórmulas o expresiones bien formadas que tenemos en cuenta en este sistema formal, coincide con el concepto de expresión booleana que hemos estado manejando. Podemos afirmar entonces que las *expresiones booleanas o fórmulas* se construirán usando los símbolos y las reglas anteriores y formarán la *sintaxis* de nuestro cálculo proposicional.

Reglas de Inferencia: Las reglas de inferencia usadas serán las que ya presentamos para la relación de igualdad en el Capítulo 1. Además, la equivalencia entre expresiones booleanas satisfacerá las siguientes propiedades:

Transitividad Si P, Q, y R son expresiones booleanas, entonces

$$\frac{P = Q, Q = R}{P = R}$$

Regla de Leibniz Si P y Q son expresiones booleanas, y E es una expresión que involucra a la variable proposicional r, entonces:

$$\frac{P = Q}{E\,[r := P] = E\,[r := Q]}$$

Sustitución Si P y R son expresiones booleanas, y P involucra a la variable proposicional r, entonces:

$$\frac{P}{P\,[r := R]}$$

Axiomas: Los axiomas serán introducidos gradualmente. El cálculo proposicional tal como se presenta aquí no pretende la minimalidad en el número de reglas de inferencia y axiomas, dado que su principal objetivo es su utilización para realizar demostraciones que apuntan a la derivación y verificación de programas.

Utilizaremos el formato de demostración que presentamos en el Capítulo 1, pero antes recordaremos los distintos casos en los que una fórmula será considerada un *teorema* dentro de nuestro Cálculo Proposicional.

Un axioma es un teorema.

La conclusión de una regla de inferencia cuyas premisas son teoremas (ya demostrados) es un teorema.

Una expresión booleana que se demuestra equivalente a un teorema es un teorema.

A continuación presentaremos los axiomas y los teoremas que de éstos se deducen.

3.3.1. La equivalencia

La equivalencia es asociativa, esta propiedad se establece mediante el siguiente axioma:

(3.1) **Axioma.** Asociatividad de $\equiv$: $((p \equiv q) \equiv r) \equiv (p \equiv (q \equiv r))$

Este axioma nos permite borrar paréntesis, es decir podemos escribir

$$p \equiv q \equiv r \quad \text{en vez de} \quad p \equiv (q \equiv r) \text{ o } (p \equiv q) \equiv r$$

(3.2) **Axioma.** Conmutatividad (o simetría) de $\equiv$: $p \equiv q \equiv q \equiv p$

Podemos entender porqué este axioma se lo asocia a la simetría, pues si agregamos paréntesis:

$$(p \equiv q) \equiv (q \equiv p)$$

Ahora daremos la prueba de nuestro primer teorema:

$$p \equiv p \equiv q \equiv q$$

Recordemos que el Axioma 3.1 nos permite agregar paréntesis de diferentes maneras, con lo cual usaremos en la demostración este axioma como $(p \equiv q \equiv q) \equiv p$, de modo que usando Leibniz, podamos reemplazar la expresión $p \equiv q \equiv q$ por p.

$$\begin{array}{ll} & p \equiv p \equiv q \equiv q \\ = & \quad \langle \text{Axioma 3.2 – reemp. } p \equiv q \equiv q \text{ por } p \rangle \\ & p \equiv p \\ = & \quad \langle \text{Axioma 3.2 – reemp. la primer } p \text{ por } p \equiv q \equiv q \rangle \\ & p \equiv q \equiv q \equiv p \end{array}$$

Como la expresión final $p \equiv q \equiv q \equiv p$ es un axioma, y de acuerdo a la definición de teorema dada en el Caso 3.3, cualquier expresión booleana que se pruebe equivalente a un axioma es un teorema, la primera expresión resulta también un teorema.

El siguiente axioma sobre la equivalencia introduce el símbolo constante $true$ como abreviatura de $p \equiv p$, (usar una constante como abreviatura es razonable ya que el valor de $p \equiv p$ no depende de p).

(3.3) **Axioma.** Neutro de $\equiv$: $true \equiv p \equiv p$

En general, una constante C se dice que es el elemento *neutro* de una operación $\circ$, si para cualquier valor b se tiene que $b = b \circ C = C \circ b$. Llamamos al Axioma 3.3, *Neutro de la equivalencia*, debido a que de acuerdo los axiomas 3.1 y 3.2, puede reescribirse también como $(p \equiv true) \equiv p$ o $p \equiv (true \equiv p)$.

El siguiente teorema es de esperar, podría haberse agregado como un axioma.

(3.4) **Teorema.** true

$$
\begin{array}{ll}
 & true \\
= & \quad \langle \text{Axioma 3.3 – reemp. } p := true \rangle \\
 & true \equiv true \\
= & \quad \langle \text{Axioma 3.3 – reemp. el segundo } true \rangle \\
 & true \equiv p \equiv p
\end{array}
$$

Como la última expresión es un axioma, el teorema queda demostrado.

Una generalización usada habitualmente en lógica e indispensable para cualquier cálculo, es que los pasos de demostración puedan justificarse no sólo por las reglas ya mencionadas sino también por teoremas previamente demostrados. Podemos ahora probar el siguiente teorema usando el anterior:

(3.5) Teorema. Reflexividad de $\equiv$: $p \equiv p$

$$
\begin{array}{ll}
 & p \equiv p \\
= & \quad \langle \text{Axioma 3.3} \rangle \\
 & true
\end{array}
$$

Los Axiomas 3.2 y 3.3, implican que la ocurrencia de "$\equiv true$" (o "$true \equiv$") en una expresión es redundante. Por lo tanto $Q \equiv true$ puede ser reemplazada por Q en cualquier expresión sin cambiar el valor de ésta. De aquí que, en general eliminaremos tales ocurrencias, a menos que necesitemos específicamente hacer referencia de ellas.

Nuestro formato de demostración consistirá en general de una serie de pasos de equivalencia desde la expresión a demostrar hasta llegar a un axioma. Cada paso de demostración estará justificado por una aplicación de la Regla de Leibniz, y el teorema final seguirá por aplicación de la regla de transitividad.

Cuando se quiere demostrar que, dadas dos expresiones lógicas E y F, es válida la expresión $E \equiv F$, en lugar de partir de la equivalencia que pretendemos probar y terminar en el valor $true$ (o en algún otro teorema ya demostrado o axioma), podemos también comenzar con la expresión booleana E y a través de equivalencias llegar a F, es decir :

$$
\begin{array}{ll}
 & E_o \\
= & \quad \langle \text{justificación de } E_o \equiv E_1 \rangle \\
 & E_1 \\
\vdots & \\
 & E_{n-1} \\
= & \quad \langle \text{justificación de } E_{n-1} \equiv F \rangle \\
 & F
\end{array}
$$

El esquema de demostración presentado arriba es correcto debido a que puede transformarse inmediatamente en el siguiente:

$$
\begin{array}{ll}
 & true \\
= & \quad \langle \text{Axioma 3.3, } true \equiv p \equiv p \rangle \\
 & E_o \equiv E_o \\
= & \quad \langle \text{justificación de } E_o \equiv E_1 \rangle \\
 & E_o \equiv E_1 \\
\vdots & \\
 & E_o \equiv E_{n-1} \\
= & \quad \langle \text{justificación de } E_{n-1} \equiv F \rangle \\
 & E_o \equiv F
\end{array}
$$

(3.6) **Mï¿½todo de prueba.** Para probar que $P \equiv Q$ es un teorema, transformamos P en Q o Q en P usando Leibniz.

Estos resultados acerca del cálculo proposicional se llaman *metateoremas*, para distinguirlos de los teoremas que son fórmulas de cálculo. Un metateorema es en general, una afirmación que se hace sobre el sistema formal que se demuestra verdadera ($true$). El objetivo de los metateoremas es proveer herramientas para poder usar el cálculo de manera efectiva en la construcción de demostraciones.

La metademostración del siguiente metateorema se deja como ejercicio, nótese que hay que describir cómo se construye una demostración a partir de otra u otras.

(3.7) **Metateorema.** Dos teoremas cualesquiera son equivalentes.

3.3.2. La negación, discrepancia, y false

Introducimos tres axiomas. El primero define a $false$; el primero y el segundo de manera conjunta definen la negación, $\neg$, y el tercero define a la discrepancia, $\not\equiv$.

(3.8) **Axioma.** Definición de false: $false \equiv \neg\, true$

(3.9) **Axioma.** Negación y equivalencia: $\neg\,(p \equiv q) \equiv \neg p \equiv q$

Además se define el significado de la constante $false$ con el siguiente axioma:

(3.10) **Axioma.** Definición de $\not\equiv$: $p \not\equiv q \equiv \neg\,(p \equiv q)$

Los siguientes teoremas se dejan como ejercicio:

(3.11) **Teorema.** $\neg p \equiv q \equiv p \equiv \neg q$

(3.12) **Teorema.** Doble negación: $\neg\neg p \equiv p$

(3.13) **Teorema.** Negación de $false$: $\neg false \equiv true$

(3.14) **Teorema.** $(p \not\equiv q) \equiv \neg p \equiv q$

(3.15) Teorema. $\neg p \equiv p \equiv false$

(3.16) Teorema. Conmutatividad de $\not\equiv$: $(p \not\equiv q) \equiv (q \not\equiv p)$

(3.17) Teorema. Asociatividad de $\not\equiv$: $((p \not\equiv q) \not\equiv r) \equiv (p \not\equiv (q \not\equiv r))$

(3.18) Teorema. Asociatividad mutua: $((p \not\equiv q) \equiv r) \equiv (p \not\equiv (q \equiv r))$

(3.19) Teorema. Intercambiabilidad: $p \not\equiv q \equiv r \ \equiv \ p \equiv q \not\equiv r$

Ahora, podemos notar una interesante y útil propiedad sobre secuencias de equivalencias. La expresión booleana

$$P_0 \equiv P_1 \equiv \cdots \equiv P_n$$

es verdadera $(true)$ cuando exactamente un número par de P_i es falso $(false)$. En efecto, de acuerdo al axioma 3.3, cada subexpresión $false \equiv false$ en la secuencia puede ser reemplazada por $true$, hasta que se consiga o bien una expresión falsa o ninguna, en cuyo caso la secuencia es falsa o verdadera. Por ejemplo, podemos determinar sin ninguna manipulación formal que $false \equiv false \equiv false \equiv true$ es falsa, porque tres de sus expresiones son falsas.

Podemos utilizar este hecho sobre secuencias de equivalencias para formalizar frases en lenguaje corriente, como se indican a continuación:

- Ninguno o los dos, entre p y q son verdaderos: $p \equiv q$
- Exactamente uno entre p y q es verdadero: $\neg\,(p \equiv q)$, o $p \not\equiv q$
- Cero, dos o cuatro entre p, q, r y s son verdaderos: $p \equiv q \equiv r \equiv s$
- Uno o tres entre p, q, r y s son verdaderos: $\neg\,(p \equiv q \equiv r \equiv s)$

Técnicas de demostración y principios

Ilustraremos distintas técnicas para desarrollar demostraciones de fórmulas, a través de ejemplos. El primer método que presentamos, lo usaremos para probar el teorema 3.11

$$\neg p \equiv q \equiv p \equiv \neg q$$

y trataremos de demostrar que esta fórmula es equivalente a $true$. Solamente podemos usar los axiomas y los teoremas hasta ahora demostrados. Para ello, los analizamos uno a uno para encontrar un parecido con la estructura de (3.11). Observamos que el Axioma 3.9 parecería ser el más apropiado. Obviamente, a mayor cantidad de teoremas que se recuerden, y mayor práctica en encontrar similitudes con las expresiones y subexpresiones de la fórmula en cuestión, más fácil será desarrollar una prueba.

(3.20) Ti¿$\frac{1}{2}$cnica de prueba. Analizar qué teoremas pueden identificarse con la estructuras o subestructuras en las expresiones. Los operadores que aparecen en las expresiones booleanas junto con la forma de sus subexpresiones son la clave para elegir los teoremas que puedan aplicarse.

Comenzaremos con la demostración:

$$
\begin{array}{ll}
 & \neg p \equiv q \equiv p \equiv \neg q \\
= & \quad \langle (3.9),\ \neg\,(p \equiv q) \equiv \neg p \equiv q \rangle \\
 & \neg\,(p \equiv q) \equiv p \equiv \neg q \\
= & \quad \langle (3.9), \text{ con } p, q := q, p \text{ i.e. } \neg\,(q \equiv p) \equiv \neg q \equiv p \rangle \\
 & \neg\,(p \equiv q) \equiv \neg\,(p \equiv q)
\end{array}
$$

La última expresión obtenida es verdadera, pues es la Reflexividad de $\equiv$ (3.5), con la sustitución $p := \neg\,(p \equiv q)$. Además hemos hecho algunas simplificaciones en la demostración, los axiomas de asociatividad y conmutatividad de la equivalencia fueron utilizados sin hacer mención de ellos.

El teorema 3.11, puede demostrarse también transformando $\neg p \equiv q$ en $p \equiv \neg q$, o también transformando $\neg p \equiv p$ en $\neg q \equiv q$. Otra opción es comenzar con $\neg p$ y transformarlo en $q \equiv p \equiv \neg q$.

Ahora consideremos dos pruebas de $\neg p \equiv p \equiv false$ (teorema 3.15)

$$
\begin{array}{ll}
 & \neg p \equiv p \equiv false \\
= & \quad \langle (3.9),\ \neg\,(p \equiv q) \equiv \neg p \equiv q \text{ con } q := p \rangle \\
 & \neg\,(p \equiv p) \equiv false \\
= & \quad \langle (3.3) \text{ con } q := p \rangle \\
 & \neg\,(true) \equiv false
\end{array}
$$

La última expresión corresponde al Axioma 3.8. Ahora veamos otra demostración

$$
\begin{array}{ll}
 & \neg p \equiv p \\
= & \quad \langle (3.9),\ \neg\,(p \equiv q) \equiv \neg p \equiv q \text{ con } q := p \rangle \\
 & \neg\,(p \equiv p) \\
= & \quad \langle (3.3) \text{ con } q := p \rangle \\
 & \neg\,(true) \\
= & \quad \langle (3.8) \text{ Definición de } \textit{false} \rangle \\
 & false
\end{array}
$$

La diferencia entre las pruebas es que la primera contiene menos pasos, pero en todas las líneas se requiere la expresión $\equiv false$, y esto, cuando las expresiones que se repitan sean más complejas tiende a la propagación de errores de transcripción. Para evitar estos inconvenientes es preferible evitar la repetición de expresiones en cada línea de demostración.

(3.21) Principio. Evitar en los formatos de prueba la repetición de subexpresiones en cada línea de la demostración.

Por último demostraremos el teorema:

(3.22) Teorema. $p \not\equiv false \equiv p$

$$
\begin{array}{ll}
 & p \not\equiv false \\
= & \quad \langle \text{(3.10) y sustitución de } q := false \rangle \\
 & \neg\,(p \equiv false) \\
= & \quad \langle \text{(3.8) Definición de } false \rangle \\
 & \neg\,(p \equiv \neg true) \\
= & \quad \langle \text{(3.9) Negación y equivalencia, con } p, q := \neg true, p \rangle \\
 & \neg\,(\neg true) \equiv p \\
= & \quad \langle \text{(3.12) Doble negación y (3.3) } true \text{ como neutro de } \equiv \rangle \\
 & p
\end{array}
$$

(3.23) **Ti¿½cnica de prueba.** Para demostrar propiedades de un nuevo operador, un método frecuentemente "exitoso" es reemplazar éste operador por su definición en términos de otros y manipular las propiedades ya vistas de los otros operadores. Luego, introducir si es posible, el nuevo operador usando su definición.

3.3.3. La disyunción

El operador de disyunción $\vee$, se define de acuerdo a los siguientes axiomas:

(3.24) **Axioma.** Asociatividad de $\vee$: $(p \vee q) \vee r \equiv p \vee (q \vee r)$

(3.25) **Axioma.** Conmutatividad de $\vee$: $p \vee q \equiv q \vee p$

Antes del siguiente axioma, diremos que un operador binario $\circ$ es *idempotente* si $x \circ x = x$ para cualquier x. Por ejemplo, la multiplicación y la suma no son operadores idempotentes.

(3.26) **Axioma.** Idempotencia: $p \vee p \equiv p$

(3.27) **Axioma.** Distributividad de $\vee$ respecto de $\equiv$: $p \vee (q \equiv r) \equiv (p \vee q) \equiv (p \vee r)$

(3.28) **Axioma.** Tercero excluído: $p \vee \neg p \equiv true$

Este último axioma significa que en cualquier estado de p, o bien p es verdadero o $\neg p$ es verdadero, no existiendo situaciones intermedias.

Con estos axiomas podemos demostrar los siguientes teoremas.

(3.29) **Teorema.** Elemento neutro de $\vee$: $p \vee false \equiv p$

(3.30) **Teorema.** Distributividad de $\vee$ respecto de $\vee$: $p \vee (q \vee r) \equiv (p \vee q) \vee (p \vee r)$

(3.31) **Teorema.** $p \vee q \equiv p \vee \neg q \equiv p$

(3.32) **Teorema.** Elemento absorbente de $\vee$: $p \vee true \equiv true$

En general, diremos que Z es *elemento absorbente* del operador binario $\circ$, si para cualquier x vale $x \circ Z = Z \circ x = Z$. Por ejemplo 0 es el elemento absorbente de la multiplicación.

En la prueba del teorema 3.32, vamos a repasar un método de demostración ya visto (3.6), que consiste en probar que $P \equiv Q$, transformando o bien P en Q, o bien Q en P.

$$
\begin{array}{ll}
 & p \vee true \\
= & \quad \langle \text{Axioma 3.3 } (p \equiv p) \equiv true \rangle \\
 & p \vee (p \equiv p) \\
= & \quad \langle \text{Axioma 3.27 Distributividad de } \vee \text{ con respecto a } \equiv \rangle \\
 & (p \vee p) \equiv (p \vee p) \\
= & \quad \langle \text{Axioma 3.3 con } p := p \vee p \rangle \\
 & true
\end{array}
$$

Es claro que aplicando la técnica de demostración, hemos transformado $p \vee true$ en $true$, hubiera sido más complicado si hubiéramos optado por lo recíproco, pues $true$ no nos da suficientes pistas para comenzar una prueba. La expresión $p \vee true$ ofrece más "estructura", es decir nos permite la posibilidad de aplicar axiomas que conocemos sobre el operador disyunción.

(3.33) Ti¿½cnica de prueba. En general, conviene comenzar a demostrar una equivalencia $P \equiv Q$ a partir de la expresión más compleja, o de "mayor estructura" (sea ésta P o Q) y transformarla en la otra expresión.

Por otra parte, la demostración anterior puede reescribirse de modo de transformar $true$ en $p \vee true$, así:

$$
\begin{array}{ll}
 & true \\
= & \quad \langle \text{Axioma 3.3 con } p := p \vee p \rangle \\
 & (p \vee p) \equiv (p \vee p) \\
= & \quad \langle \text{(3.27), Distributividad de } \vee \text{ con respecto a } \equiv \rangle \\
 & p \vee (p \equiv p) \\
= & \quad \langle \text{Axioma 3.3 } (p \equiv p) \equiv true \rangle \\
 & p \vee true
\end{array}
$$

Esta prueba es tan válida como la anterior, pero en su presentación, no queda para nada claro cuál fue su motivación, y de ahí que no sea sencillo visualizar el desarrollo mismo de la prueba.

(3.34) Principio. En general, conviene que la estructura de demostración sea lo menos artificiosa posible, haciendo que cada paso de la prueba sea fácil de comprender, y que no pierda de vista el objetivo de la manipulación.

3.3.4. La conjunción

Definimos la conjunción $\wedge$ con un único axioma, conocido como *Regla dorada*.

(3.35) Axioma. Regla dorada: $p \wedge q \equiv p \equiv q \equiv p \vee q$

La regla dorada aprovecha fuertemente la asociatividad y la conmutatividad de la equivalencia. En principio la interpretamos como:

$p \wedge q \equiv (p \equiv q \equiv p \vee q)$,

pero nada nos impide hacer otras interpretaciones, por ejemplo:

$(p \wedge q \equiv p) \equiv (q \equiv p \vee q)$, o bien

$(p \wedge q \equiv p \equiv q) \equiv p \vee q$, o bien, usando la conmutatividad de la equivalencia,

$(p \equiv q) \equiv (p \wedge q \equiv p \vee q)$, etcétera.

La regla dorada será de gran utilidad para demostrar propiedades de la conjunción y la disyunción, dado que provee una relación entre ambas.

A continuación presentamos algunos teoremas relacionados con el operador $\wedge$.

(3.36) Teorema. Asociatividad de $\wedge$: $p \wedge (q \wedge r) \equiv (p \wedge q) \wedge r$

(3.37) Teorema. Conmutatividad de $\wedge$: $p \wedge q \equiv q \wedge p$

(3.38) Teorema. Idempotencia de $\wedge$: $p \wedge p \equiv p$

(3.39) Teorema. Neutro de $\wedge$: $p \wedge true \equiv p$

(3.40) Teorema. Elemento absorbente de $\wedge$: $p \wedge false \equiv false$

(3.41) Teorema. Distributividad de $\wedge$ sobre $\wedge$: $p \wedge (q \wedge r) \equiv (p \wedge q) \wedge (p \wedge r)$

(3.42) Teorema. Contradicción: $p \wedge \neg p \equiv false$

Los siguientes teoremas relacionan la conjunción con la disyunción:

(3.43) Teorema. Absorción:

a) $p \wedge (p \vee q) \equiv p$

b) $p \vee (p \wedge q) \equiv p$

(3.44) Teorema. Absorción:

a) $p \wedge (\neg p \vee q) \equiv p \wedge q$

b) $p \vee (\neg p \wedge q) \equiv p \vee q$

Los teoremas anteriores son llamados de absorción, pues la expresión q es absorbida por p en el primero y la expresión $\neg p$ es absorbida por q en el segundo.

(3.45) Teorema. Distributividad de $\wedge$ con respecto a $\vee$: $p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$

(3.46) Teorema. Distributividad de $\vee$ con respecto a $\wedge$: $p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$

El siguiente teorema es conocido como *Leyes de De Morgan*.

(3.47) Teorema. Leyes de De Morgan

a) $\neg(p \vee q) \equiv \neg p \wedge \neg q$

b) $\neg(p \wedge q) \equiv \neg p \vee \neg q$

El siguiente grupo de teoremas relaciona la conjunción con la equivalencia.

(3.48) Teorema. $p \wedge q \equiv p \wedge \neg q \equiv \neg p$

(3.49) Teorema. $p \wedge (q \equiv r) \equiv p \wedge q \equiv p \wedge r \equiv p$

El teorema anterior muestra como la conjunción distribuye respecto de la equivalencia, en la utilización de esta regla, es un error común olvidar la última equivalencia con p.

(3.50) Teorema. $p \wedge (q \equiv p) \equiv p \wedge q$

(3.51) Teorema. Reemplazo: $(p \equiv q) \wedge (r \equiv p) \equiv (p \equiv q) \wedge (r \equiv q)$

En otros sistemas de cálculo proposicional, es común usar los siguientes teoremas para definir a la equivalencia y a la discrepancia. El primer teorema indica que se da $p \equiv q$ exactamente cuando p y q son ambos $true$ o ambos $false$. El segundo teorema indica que $p \not\equiv q$ es cierta cuando exactamente una entre p y q es $true$ y la otra es $false$.

(3.52) Teorema. Definición alternativa de $\equiv$: $p \equiv q \equiv (p \wedge q) \vee (\neg p \wedge \neg q)$

(3.53) Teorema. "O" exclusivo: $p \not\equiv q \equiv (\neg p \wedge q) \vee (p \wedge \neg q)$

Usando la Regla dorada

Como la definición de $\wedge$ viene dada mediante la Regla Dorada, ésta es utilizada con frecuencia en la eliminación de la conjunción en el primer paso de la demostración de la mayoría de los teoremas listados en la sección correspondiente a este operador. Como ejemplo, probaremos el teorema 3.44(a), $p \wedge (\neg p \vee q) \equiv p \wedge q$

$$
\begin{array}{ll}
 & p \wedge (\neg p \vee q) \\
= & \quad \langle \text{(3.35), Regla dorada con } q := \neg p \vee q \rangle \\
 & p \equiv \neg p \vee q \equiv p \vee \neg p \vee q \\
= & \quad \langle \text{(3.28), Tercero excluido} \rangle \\
 & p \equiv \neg p \vee q \equiv true \vee q \\
= & \quad \langle \text{Axioma 3.32, } true \vee q \equiv true \rangle \\
 & p \equiv \neg p \vee q \equiv true \\
= & \quad \langle \text{Axioma 3.3, con } p := p \equiv \neg p \vee q \rangle \\
 & p \equiv \neg p \vee q \\
= & \quad \langle \text{(3.31), } p \vee q \equiv p \vee \neg q \equiv p, \text{ con } p, q := q, p \rangle \\
 & p \equiv p \vee q \equiv q \\
= & \quad \langle \text{(3.35) Regla dorada } \rangle \\
 & p \wedge q
\end{array}
$$

La Regla Dorada tiene cuatro operadores equivalencia. Por lo tanto, puede usarse para reemplazar una equivalencia por tres, dos equivalencias por otras dos, o tres equivalencias por una. Vamos a ver ahora como utilizar estas diferentes opciones. Primero, probemos $p \wedge true \equiv p$ (teorema 3.39) demostrando que éste equivale a otro teorema ya probado.

$$
\begin{array}{ll}
 & p \wedge true \equiv p \\
= & \quad \langle \text{(3.35) Regla dorada, reemp. dos } \equiv \rangle \\
 & p \vee true \equiv true
\end{array}
$$

Y este último es el teorema 3.32.

Ahora probemos (3.49), $p \wedge (q \equiv r) \equiv p \wedge q \equiv p \wedge r \equiv p$. Debido a que la equivalencia es asociativa y conmutativa vamos realizar la demostración transformando $p \wedge (q \equiv r) \equiv p$ en $p \wedge q \equiv p \wedge r$.

$$
\begin{array}{ll}
 & p \wedge (q \equiv r) \equiv p \\
= & \quad \langle \text{(3.35) Regla dorada, con } q := q \equiv r \text{, reemp. dos } \equiv \rangle \\
 & p \vee (q \equiv r) \equiv q \equiv r \\
= & \quad \langle \text{(3.27) Distributividad de } \vee \text{ con respecto a } \equiv \rangle \\
 & p \vee q \equiv p \vee r \equiv q \equiv r \\
= & \quad \langle \text{(3.2) Conmutatividad de } \equiv \rangle \\
 & p \vee q \equiv q \equiv p \vee r \equiv r \\
= & \quad \langle \text{(3.35) Regla dorada dos veces, reemp. } p \vee q \equiv q \text{ y } p \vee r \equiv r \rangle \\
 & p \wedge q \equiv p \equiv p \wedge r \equiv p \\
= & \quad \langle \text{(3.2) Conmutatividad de } \equiv \text{ con } q := p \wedge q \equiv p \wedge r \rangle \\
 & p \wedge q \equiv p \wedge r
\end{array}
$$

Usando lemas

Cuando una prueba se torna larga y complicada, es a veces útil implementar una estructura que la divida en lemas. Un lema es un teorema auxiliar que se utiliza en la prueba de un teorema. La diferencia entre el lema y el teorema está en el punto de vista del objetivo de prueba. El teorema apunta a aquello que se quiere demostrar, mientras que el lema es un teorema menor que se necesita en la prueba.

(3.54) Principio. Los lemas pueden proveer nuevas estructuras, esclarecer hechos relevantes, y acortar pruebas de otros teoremas.

Vamos a ilustrar la necesidad de lemas en la demostración del teorema de la asociatividad de la conjunción (3.36). Comenzaremos con la expresión $(p \wedge q) \wedge r$ que pretendemos transformar en $p \wedge (q \wedge r)$. Usaremos la Regla dorada para interpretar el operador conjunción.

$$
\begin{array}{ll}
 & (p \wedge q) \wedge r \\
= & \quad \langle \text{(3.35) Regla Dorada interpretada como } p \wedge q \equiv (p \equiv q \equiv p \vee q) \rangle \\
 & (p \equiv q \equiv p \vee q) \wedge r \\
= & \quad \langle \text{(3.35) Regla Dorada con } p, q := (p \equiv q \equiv p \vee q)\,, r \rangle \\
 & (p \equiv q \equiv p \vee q) \equiv r \equiv (p \equiv q \equiv p \vee q) \vee r \\
= & \quad \langle \text{(3.27)Distributividad de } \vee \text{ respecto de } \equiv \rangle \\
 & (p \equiv q \equiv p \vee q) \equiv r \equiv (p \vee r \equiv q \vee r \equiv (p \vee q) \vee r) \\
= & \quad \langle \text{asociatividad y conmutatividad de } \equiv \text{ y } \vee \rangle \\
 & p \equiv q \equiv r \equiv p \vee q \equiv q \vee r \equiv r \vee p \equiv p \vee q \vee r
\end{array}
$$

Hemos mostrado que $(p \wedge q) \wedge r$ es equivalente a la equivalencia entre todas las posibles disjunciones entre p, q y r.

$$(p \wedge q) \wedge r \equiv p \equiv q \equiv r \equiv p \vee q \equiv q \vee r \equiv r \vee p \equiv p \vee q \vee r \tag{3.3}$$

que es una equivalencia lo suficientemente interesante como para considerarla un lema. Más aún, como la equivalencia y la disyunción son asociativas y conmutativas podemos esperar que también resulte equivalente a la expresión $p \wedge (q \wedge r)$, que era nuestro objetivo inicial.

Por lo tanto ahora presentamos nuestra demostración del teorema 3.36, utilizando (3.3) como lema:

$$
\begin{array}{ll}
 & p \wedge (q \wedge r) \\
= & \quad \langle \text{(3.37) Conmutatividad de } \wedge \rangle \\
 & (q \wedge r) \wedge p \\
= & \quad \langle \text{lema 3.3, con } p, q, r := q, r, p \rangle \\
 & q \equiv r \equiv p \equiv q \vee r \equiv r \vee p \equiv p \vee q \equiv q \vee r \vee p \\
= & \quad \langle \text{Asociatividad y Conmutatividad de } \equiv \text{ y } \vee \rangle \\
 & p \equiv q \equiv r \equiv p \vee q \equiv q \vee r \equiv r \vee p \equiv p \vee q \vee r \\
= & \quad \langle \text{lema 3.3 } \rangle \\
 & (p \wedge q) \wedge r
\end{array}
$$

(3.55) **Tï¿½cnica de prueba.** Explotar la habilidad de descomponer teoremas como la Regla Dorada en varias formas distintas.

3.3.5. La implicación

Ahora definimos los dos últimos operadores, la implicación $\Rightarrow$ y la consecuencia $\Leftarrow$.

(3.56) **Axioma.** Definición de implicación: $p \Rightarrow q \equiv p \vee q \equiv q$

(3.57) **Axioma.** Definición de consecuencia: $p \Leftarrow q \equiv p \vee q \equiv p$

Debido a la similitud entre estos dos operadores, daremos solamente teoremas que involucren al operador implicación, los correspondientes a la consecuencia siguen inmediatamente del Axioma 3.57.

Lo primero que podemos observar sobre la implicación es que puede reescribirse de distintas formas. Los siguientes teoremas describen esta situación:

(3.58) Teorema. Definición alternativa de implicación: $p \Rightarrow q \equiv \neg p \vee q$

(3.59) Teorema. Definición alternativa de implicación: $p \Rightarrow q \equiv p \wedge q \equiv p$

(3.60) Teorema. Contrarrecíproco: $p \Rightarrow q \equiv \neg q \Rightarrow \neg p$

Tanto el teorema 3.58, como el teorema 3.59 se utilizan frecuentemente como definición de implicación. A continuación presentamos una lista de teoremas relativos a este operador.

(3.61) Teorema. $p \Rightarrow (q \equiv r) \equiv p \wedge q \equiv p \wedge r$

(3.62) Teorema. Distributividad de $\Rightarrow$ respecto de $\equiv$: $p \Rightarrow (q \equiv r) \equiv p \Rightarrow q \equiv p \Rightarrow r$

Los dos teoremas anteriores muestran cómo eliminar la equivalencia en el consecuente.

(3.63) Teorema. $p \Rightarrow (q \Rightarrow r) \equiv (p \Rightarrow q) \Rightarrow (p \Rightarrow r)$

(3.64) Teorema. Traslación: $p \wedge q \Rightarrow r \equiv p \Rightarrow (q \Rightarrow r)$

El teorema anterior indica cómo trasladar una conjunción del antecedente al consecuente.

(3.65) Teorema. $p \wedge (p \Rightarrow q) \equiv p \wedge q$

(3.66) Teorema. $p \wedge (q \Rightarrow p) \equiv p$

(3.67) Teorema. $p \vee (p \Rightarrow q) \equiv true$

(3.68) Teorema. $p \vee (q \Rightarrow p) \equiv q \Rightarrow p$

(3.69) Teorema. $p \vee q \Rightarrow p \wedge q \equiv p \equiv q$

Los siguientes teoremas relacionan la implicación con las constantes booleanas. El primero establece que la implicación es reflexiva.

(3.70) Teorema. Reflexividad de $\Rightarrow$: $p \Rightarrow p \equiv true$

(3.71) Teorema. Elemento absorbente a derecha de $\Rightarrow$: $p \Rightarrow true \equiv true$

(3.72) Teorema. Elemento neutro a izquierda de $\Rightarrow$: $true \Rightarrow p \equiv p$

(3.73) Teorema. $p \Rightarrow false \equiv \neg p$

(3.74) Teorema. $false \Rightarrow p \equiv true$

Los siguientes teoremas son llamados teoremas de debilitamiento o fortalecimiento, dependiendo de si se usan para transformar el antecedente en el consecuente (debilitándolo) o para transformar el consecuente en el antecedente (fortaleciéndolo).

(3.75) Teorema. de Debilitamiento o fortalecimiento

a) $p \Rightarrow p \vee q$

b) $p \wedge q \Rightarrow p$

c) $p \wedge q \Rightarrow p \vee q$

d) $p \vee (q \wedge r) \Rightarrow p \vee q$

e) $p \wedge q \Rightarrow p \wedge (q \vee r)$

(3.76) Teorema. Modus Ponens: $p \wedge (p \Rightarrow q) \Rightarrow q$

Modus Ponens es el término en latín correspondiente a Método del puente, en algunos sistemas de cálculo proposicional, éste teorema constituye la más importante de las reglas de inferencia, si bien en el nuestro no ocupa un papel tan importante, es de suma utilidad en demostraciones.

Los siguientes teoremas involucran *análisis de casos.* El primero indica que la prueba de $p \vee q \Rightarrow r$ puede hacerse mediante las pruebas de $p \Rightarrow r$ y $q \Rightarrow r$ por separado. El segundo asegura que la prueba de r puede partirse en dos casos.

(3.77) Teorema. $(p \Rightarrow r) \wedge (q \Rightarrow r) \equiv (p \vee q \Rightarrow r)$

(3.78) Teorema. $(p \Rightarrow r) \wedge (\neg p \Rightarrow r) \equiv r$

En la mayoría de los sistemas de cálculo proposicional, la equivalencia se define como último operador, y se hace a través de la *mutua implicación*, con lo cual el teorema que sigue, resulta un axioma en estos sistemas.

(3.79) Teorema. Implicación mutua: $(p \Rightarrow q) \wedge (q \Rightarrow p) \equiv p \equiv q$

Antes de establecer el siguiente teorema diremos que una operación binaria $\circ$, es antisimétrica si $x \circ y \wedge y \circ x \Rightarrow x = y$ es cierta para cualquier par x, y. Por ejemplo $\leq$ y $\geq$ son operadores antisimétricos.

(3.80) Teorema. Antisimetría: $(p \Rightarrow q) \wedge (q \Rightarrow p) \Rightarrow (p \equiv q)$

Los siguientes teoremas están relacionados con la transitividad y se usarán en el próximo capítulo para abreviar el formato de demostración:

(3.81) Teorema. Transitividad:

a) $(p \Rightarrow q) \wedge (q \Rightarrow r) \Rightarrow (p \Rightarrow r)$

b) $(p \equiv q) \wedge (q \Rightarrow r) \Rightarrow (p \Rightarrow r)$

c) $(p \Rightarrow q) \wedge (q \equiv r) \Rightarrow (p \Rightarrow r)$

Prueba de teoremas en donde participa la implicación

Haremos aquí una serie de consideraciones generales sobre los teoremas en donde aparece la implicación. En estos casos, la técnica de prueba 3.23 será de gran utilidad, para esto es necesario recordar no sólo la definición de implicación (Axioma 3.56) sino también los teoremas 3.58, 3.59 y 3.60 de acuerdo a la estructura de la expresión que debemos manipular. Para ilustrar esto último, probaremos (3.61), $p \Rightarrow (q \equiv r) \equiv p \wedge q \equiv p \wedge r$. Como esta expresión contiene conjunciones, la definición alternativa de implicación (3.59) parecería la más apropiada para utilizar, ya que muestra como reemplazar una implicación introduciendo una conjunción:

$$
\begin{array}{ll}
 & p \Rightarrow (q \equiv r) \\
= & \quad \langle \text{(3.59) Definición de implicación} \rangle \\
 & p \wedge (q \equiv r) \equiv p \\
= & \quad \langle \text{(3.49), } p \wedge (q \equiv r) \equiv p \wedge q \equiv p \wedge r \equiv p \rangle \\
 & p \wedge q \equiv p \wedge r
\end{array}
$$

Veamos ahora cómo probar la implicación mutua (3.79), $(p \Rightarrow q) \wedge (q \Rightarrow p) \equiv p \equiv q$:

$$
\begin{array}{ll}
 & (p \Rightarrow q) \wedge (q \Rightarrow p) \\
= & \quad \langle \text{(3.58) Definición alternativa en cada implicación} \rangle \\
 & (\neg p \vee q) \wedge (\neg q \vee p) \\
= & \quad \langle \text{3.45 Distributividad de } \wedge \text{ con respecto a } \rangle \\
 & (\neg p \wedge \neg q) \vee (\neg p \wedge p) \vee (q \wedge \neg q) \vee (q \wedge p) \\
= & \quad \langle \text{(3.42) Contradicción y (3.29) Elemento neutro de } \vee \rangle \\
 & (\neg p \wedge \neg q) \vee (q \wedge p) \\
= & \quad \langle \text{(3.52) Definición alternativa de } \equiv \rangle \\
 & p \equiv q
\end{array}
$$

Aquí presentamos una prueba corta de la Transitividad de la implicación (3.81a), en donde usaremos la Traslación (3.64), para trasladar p desde el consecuente al antecedente.

$$
\begin{array}{ll}
 & (p \Rightarrow q) \wedge (q \Rightarrow r) \Rightarrow (p \Rightarrow r) \\
= & \quad \langle \text{(3.64) Traslación, con } p, q := (p \Rightarrow q) \wedge (q \Rightarrow r), p \rangle \\
 & p \wedge (p \Rightarrow q) \wedge (q \Rightarrow r) \Rightarrow r \\
= & \quad \langle \text{(3.65), } p \wedge (p \Rightarrow q) \equiv p \wedge q \rangle \\
 & p \wedge q \wedge (q \Rightarrow r) \Rightarrow r \\
= & \quad \langle \text{(3.65), otra vez con } p, q := q, r \rangle \\
 & p \wedge q \wedge r \Rightarrow r
\end{array}
$$

Esta última expresión no es más que el teorema de Fortalecimiento (3.75b), con $p, q := p \wedge q, r$.

La Regla de leibniz como axioma

Ya hemos hablado y utilizado en varias oportunidades la Regla de Leibniz:

$$Leibniz : \frac{X = Y}{E\,[z := X] = E\,[z := Y]} \tag{3.4}$$

ahora que hemos introducido el operador $\Rightarrow$, podemos dar la siguiente versión de Leibniz como un *esquema de axioma:*

(3.82) **Axioma.** Si e y f son expresiones booleanas idénticas en un estado dado, es decir $e = f$ entonces al sustituir en una expresión E se obtiene $E\,[z := e] = E\,[z := f]$. Es decir

$$(e = f) \Rightarrow E\,[z := e] = E\,[z := f]$$

La forma de este axioma es diferente a los anteriores, ya que se trata de un esquema de axioma, es decir para cada expresión E se consigue una axioma diferente.

Si bien este axioma está inspirado en la regla de Leibniz su significado es distinto. La regla de Leibniz dice que: " si $X = Y$ es válida, es decir es $true$ para cualquier estado, entonces $E\,[z := X] = E\,[z := Y]$ también es válida". El axioma de Leibniz (3.82), en cambio, establece que "si $e = f$ es $true$ en un estado, entonces $E\,[z := e] = E\,[z := f]$ es $true$ en ese estado". Una diferencia importante con la regla de Leibniz, es que la recíproca del axioma no es cierta, mientras que la regla de Leibniz tiene una recíproca verdadera. Veamos esta diferencia reflejada en el siguiente ejemplo:

(3.83) **Ejemplo.** Sean E la expresión $true \vee z$, e el valor $true$ y f el valor $false$, luego es cierto que

$$E\,[z := true] = E\,[z := false]$$

dado que ambas expresiones son verdaderas, pero obviamente $true \neq false$

Las siguientes reglas de sustitución siguen del Axioma 3.82:

(3.84) **Teorema.** Reglas de Sustitución:

a) $(e = f) \wedge E\,[z := e] \equiv (e = f) \wedge E\,[z := f]$

b) $(e = f) \Rightarrow E\,[z := e] \equiv (e = f) \Rightarrow E\,[z := f]$

c) $q \wedge (e = f) \Rightarrow E\,[z := e] \equiv q \wedge (e = f) \Rightarrow E\,[z := f]$

Los teoremas que siguen establecen propiedades cuando se reemplazan variables por constantes $booleanas$.

(3.85) **Teorema.** Reemplazo por $true$:

a) $p \Rightarrow E\,[z := p] \equiv p \Rightarrow E\,[z := true]$

b) $q \wedge p \Rightarrow E\,[z := p] \equiv q \wedge p \Rightarrow E\,[z := true]$

(3.86) Teorema. Reemplazo por $false$

a) $E\,[z := p] \Rightarrow p \equiv E\,[z := false] \Rightarrow p$

b) $E\,[z := p] \Rightarrow q \vee p \equiv E\,[z := false] \Rightarrow q \vee p$

(3.87) Teorema. Reemplazo por $true$: $p \wedge E\,[z := p] \equiv p \wedge E\,[z := true]$

(3.88) Teorema. Reemplazo por $false$: $p \vee E\,[z := p] \equiv p \vee E\,[z := false]$

(3.89) Teorema. Shannon: $E\,[z := p] \equiv (p \wedge E\,[z := true]) \vee (\neg p \wedge E\,[z := false])$

Vamos a ilustrar el uso de estos teoremas demostrando $p \wedge q \Rightarrow (p \equiv q)$:

$$
\begin{array}{ll}
 & p \wedge q \Rightarrow (p \equiv q) \\
= & \quad \langle \text{(3.85b) Reemplazo por } true \rangle \\
 & p \wedge q \Rightarrow (true \equiv q) \\
= & \quad \langle \text{(3.85b) Reemplazo por } true \rangle \\
 & p \wedge q \Rightarrow (true \equiv true) \\
= & \quad \langle \text{(3.3) Neutro de } \equiv \rangle \\
 & p \wedge q \Rightarrow true \\
= & \quad \langle \text{teorema 3.71 con } p := p \wedge q \rangle \\
 & true
\end{array}
$$

3.4. Ejercicios

Ejercicios de Equivalencia

3.1 Utilizar las tablas de verdad para demostrar que los axiomas (3.1), (3.2) y (3.3) son válidos, es decir, $true$ en todo estado.

3.2 Probar la Reflexividad de $\equiv$ (3.5), $p \equiv p$.

Ejercicios de negación, discrepancia y false

3.3 Probar el teorema (3.11) de tres maneras diferentes: comenzar con $\neg p \equiv q$ y transformarlo en $p \equiv \neg q$, comenzar con $\neg p \equiv p$ y transformarlo en $q \equiv \neg q$, y finalmente comenzar con $\neg p$ y transformarlo en $q \equiv p \equiv \neg q$. Comparar las tres pruebas con la brindada en la página 58, ¿Cuál es más simple o más corta?.

3.4 Probar la Doble negación (3.12), $\neg\neg p \equiv p$.

3.5 Probar la Negación de $false$ (3.13), $\neg false \equiv true$.

3.6 Probar el teorema (3.14), $(p \not\equiv q) \equiv \neg p \equiv q$.

3.7 Probar el teorema (3.15) transformando $\neg p \equiv p \equiv false$ en $true$ utilizando (3.11). La prueba puede requerir solo dos usos de Leibniz.

3.8 Probar la Intercambiabilidad mutua (3.19), $p \not\equiv q \equiv r \equiv p \equiv q \not\equiv r$, utilizando la técnica de eliminación de la definición (3.23) —eliminando $\not\equiv$, utilizando una propiedad de $\equiv$ y reintroduciendo $\not\equiv$.

Ejercicios de disyunción

3.9 Probar que el elemento neutro de un operador binario $\oplus$ es único. (Un objeto es único si, cuando se supone que dos de ellos B y C existen, podemos probar que $B = C$)

3.10 Probar la Identidad de $\vee$ (3.29), $p \vee false \equiv p$, transformando el lado más complejo o estructurado en el lado más simple. El teorema (3.15) puede ser útil para introducir una equivalencia.

3.11 Probar la Distributividad de $\vee$ sobre $\vee$ (3.30), $p \vee (q \vee r) \equiv (p \vee q) \vee (p \vee r)$. La prueba requiere sólo la simetría, asociatividad y la idempotencia de $\vee$.

Ejercicios sobre conjunción

3.12 Mostrar la validez de la Regla Dorada (3.35), construyendo su tabla de verdad.

3.13 Probar la Conmutatividad del $\wedge$ (3.37), $p \wedge q \equiv q \wedge p$, usando la técnica de eliminación de la definición (3.23) —eliminar el $\wedge$ (usando la definición dada por la Regla Dorada), manipular la expresión y luego reintroducir el $\wedge$.

3.14 Probar la Idempotencia del $\wedge$ (3.38), $p \wedge p \equiv p$, usando la técnica de eliminación de la definición (3.23) —eliminar el $\wedge$ (usando la definición dada por la Regla Dorada), y manipular la expresión.

3.15 Probar el Elemento absorbente de $\wedge$ (3.40), $p \wedge false \equiv false$, usando la técnica de eliminación de la definición (3.23) —eliminar el $\wedge$ (usando la definición dada por la Regla Dorada), y manipular la expresión.

3.16 Probar la Contradicción del $\wedge$ (3.42), $p \wedge \neg p \equiv false$, usando la técnica de la definición de la eliminación (3.23) —eliminar el $\wedge$ (usando la definición dada por la Regla Dorada), y manipular la expresión.

3.17 Probar la Absorción (3.43a), $p \wedge (p \vee q) \equiv p$ usando la técnica de eliminación de la definición (3.23) —eliminar el $\wedge$ (usando la definición dada por la Regla Dorada), y manipular la expresión.

3.18 Probar la Absorción (3.43b), $p \vee (p \wedge q) \equiv p$. Usar la Regla Dorada.

3.19 Probar la Absorción (3.44b), $p \vee (\neg p \wedge q) \equiv p \vee q$. Usar la Regla Dorada y manipular la expresión.

3.20 Probar la Distribución del $\vee$ sobre el $\wedge$ (3.46), $p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$, usando la técnica de eliminación de la definición (3.23) —eliminar el $\wedge$ (usando la definición dada por la Regla Dorada), manipular la expresión y reintroducir $\wedge$ usando la Regla Dorada nuevamente.

3.21 Probar el teorema de la Distribución del $\wedge$ sobre el $\vee$ (3.45). Atención: no puede probarse de la misma forma que la Distribución del $\vee$ sobre el $\wedge$ (3.46), porque $\wedge$ no se distribuye sobre $\equiv$ tan elegantemente. En su lugar pruébelo usando (3.46) y Absorción.

3.22 Probar la Ley De Morgan (3.47a), $\neg (p \wedge q) \equiv \neg p \vee \neg q$. Comenzar usando la Regla Dorada; (3.31) debería resultar útil.

3.23 Probar (3.48), $p \wedge q \equiv p \wedge \neg q \equiv \neg p$. El teorema (3.31) debería resultar útil.

3.24 Probar el Reemplazo (3.51), $(p \equiv q) \wedge (r \equiv p) \equiv (p \equiv q) \wedge (r \equiv q)$, probando que el lado derecho y el lado izquierdo de la expresión son equivalentes a $p \equiv q \equiv r \equiv p \vee q \equiv q \vee r \equiv r \vee p$. La transformación del lado izquierdo (o el lado derecho) a ésta expresión puede realizarse aplicando (3.27) tres veces.

3.25 Probar el teorema del "O" Exclusivo (3.53), $p \not\equiv q \equiv (\neg p \wedge q) \vee (p \wedge \neg q)$. Ayuda: Tratar de aplicar la Definición de $\equiv$ (3.52).

Ejercicios sobre implicación

3.26 Probar Implicación (3.58), $p \Rightarrow q \equiv \neg p \vee q$.

3.27 Probar Implicación (3.59), $p \Rightarrow q \equiv p \wedge q \equiv p$.

3.28 Probar Contrarrecíproco (3.60), $p \Rightarrow q \equiv \neg q \Rightarrow \neg p$.

3.29 Probar $p \Rightarrow q \equiv \neg p \vee \neg q \equiv \neg p$.

3.30 Probar Distributividad de $\Rightarrow$ respecto de $\equiv$ (3.62),
$p \Rightarrow (q \equiv r) \equiv p \Rightarrow q \equiv p \Rightarrow r$.

3.31 Probar Traslación (3.64), $p \wedge q \Rightarrow r \equiv p \Rightarrow (q \Rightarrow r)$.

3.32 Probar el teorema (3.67), $p \vee (p \Rightarrow q) \equiv true$.

3.33 Probar el teorema (3.68), $p \vee (q \Rightarrow p) \equiv q \Rightarrow p$.

3.34 Probar el teorema (3.69), $p \vee q \Rightarrow p \wedge q \equiv p \equiv q$.

3.35 Probar Reflexividad de $\Rightarrow$ (3.70), $p \Rightarrow p \equiv true$.

3.36 Probar Elemento neutro a izquierda de $\Rightarrow$ (3.72), $true \Rightarrow p \equiv p$.

3.37 Probar el teorema (3.73), $p \Rightarrow false \equiv \neg p$.

3.38 Probar Debilitamiento/fortalecimiento (3.75a), $p \Rightarrow p \vee q$

3.39 Probar Debilitamiento/fortalecimiento (3.75b), $p \wedge q \Rightarrow p$.

3.40 Probar Debilitamiento/fortalecimiento (3.75c), $p \wedge q \Rightarrow p \vee q$.

3.41 Probar Debilitamiento/fortalecimiento (3.75d), $p \vee (q \wedge r) \Rightarrow p \vee q$.

3.42 Probar Debilitamiento/fortalecimiento (3.75e), $p \wedge q \Rightarrow p \wedge (q \vee r)$.

3.43 Probar Modus Ponens (3.76), $p \wedge (p \Rightarrow q) \Rightarrow q$.

3.44 Probar el teorema (3.77), $(p \Rightarrow r) \wedge (q \Rightarrow r) \equiv (p \vee q \Rightarrow r)$.

3.45 Demostrar el teorema (3.78), $(p \Rightarrow r) \wedge (\neg p \Rightarrow r) \equiv r$.

3.46 Demostrar Antisimetría (3.80) en dos pasos. Usar el teorema de Implicación mutua (3.79).

3.47 Demostrar Transitividad (3.81c). Hacer uso de los teoremas de Implicación mutua (3.79), Transitividad (3.81a) y Traslación (3.64).

Ejercicios sobre la Regla de Leibniz como axioma

3.48 **a)** Probar la sustitución (3.84a), $(e = f) \wedge E\,[z := e] \equiv (e = f) \wedge E\,[z := f]$. Comenzar con Leibniz (3.82), y reemplazar la implicancia.

b) Probar la sustitución (3.84b), $(e = f) \Rightarrow E\,[z := e] \equiv (e = f) \Rightarrow E\,[z := f]$.

c) Probar el reemplazo por $true$ (3.85a), $p \Rightarrow E\,[z := p] \equiv p \Rightarrow E\,[z := true]$. Para poder utilizar (3.84b), introducir el equivalente $true$ en el antecedente.

d) Probar el reemplazo por $true$ (3.85b), $q \wedge p \Rightarrow E\,[z := p] \equiv q \wedge p \Rightarrow E\,[z := true]$.

e) Probar el reemplazo por $false$ (3.86b), $E\,[z := p] \Rightarrow p \vee q \equiv E\,[z := false] \Rightarrow p \vee q$.

f) Probar Shannon (3.89) $E\,[z := p] \equiv (p \wedge E\,[z := true]) \vee (\neg p \wedge E\,[z := false])$.

g) Probar Debilitamiento/fortalecimiento (3.75a), $p \wedge q \Rightarrow p \wedge (q \vee r)$, usando el Remplazo por $true$.

3.49 Sea P una expresión de la forma $true, q \wedge r, q \equiv r$, o $q \Rightarrow r$, y su dual P_D. Demostrar que la proposición $P \equiv \neg P_D$ es válida para cualquier expresión P de la forma dada, asumiendo que es válida para cada una de sus subexpresiones. Ayuda: Por la definición de dualidad, para un operador como por ejemplo $\wedge$, se cumple que $(q \wedge r)_D \equiv q_D \vee r_D$.

Ejercicios sobre Formas Normales

3.50 Una expresión booleana está en su *forma normal conjuntiva* si es de la forma:

$E_0 \wedge E_1 \wedge ... \wedge E_n$

donde cada E_i es una disjunción de variables y variables negadas. Por ejemplo, la siguiente expresión está en su *forma normal conjuntiva*:

$(a \vee \neg b) \wedge (a \vee b \vee c) \wedge (\neg a)$

Una expresión está en *forma normal disyuntiva* si es de la forma:

$E_0 \vee E_1 \vee ... \vee E_n$

donde cada E_i es una conjunción de variables y variables negadas. Por ejemplo, la siguiente expresión está en su *forma normal disyuntiva*:

$(a \wedge \neg b) \vee (a \wedge b \wedge c) \vee (\neg a)$

En Ingeniería eléctrica, donde las formas normales son usadas cuando se trabaja con circuitos, una expresión de la forma $V_0 \vee V_1 \vee ... \vee V_n$, donde cada V_i es una variable, es llamada *maxtérmino*, por la siguiente razón. Si consideramos $false < true$, luego $x \vee y$ es el máximo entre x e y, por lo que el *maxtérmino* representa al máximo de sus operandos. De la misma manera, una expresión de la forma $V_0 \wedge V_1 \wedge ... \wedge V_n$ es llamada *mintérmino*.

a) La siguiente tabla de verdad define un conjunto de estados de las variables a, b, c y d. Dar una expresión booleanda en *forma normal disyuntiva* que sea verdadera exactamente en los estados definidos por la tabla de verdad. Basándose en este ejemplo describir un procedimiento para convertir una tabla de verdad en una expresión booleana equivalente en su *forma normal disyuntiva*.

a	b	c	d
true	*true*	*true*	*false*
true	*false*	*true*	*false*
false	*true*	*true*	*false*

Dado a que cada expresión booleana puede ser descripta por una tabla de verdad, podemos decir entonces que cada expresión booleana puede ser transformada a la *forma normal disyuntiva*.

b) Resolver el item $a)$ dando ahora la *forma normal conjuntiva*. Describir también un procedimiento para transformar una tabla a una expresión booleana en su *forma normal conjuntiva*.

CAPÍTULO 4
Aplicaciones del Cálculo Proposicional

Índice del Capítulo

4.1. Forma abreviada en la prueba de implicaciones 76

4.2. Suponiendo el antecedente 78

4.3. Analizando razonamientos en lenguaje corriente 78

4.4. Construyendo contraejemplos 79

4.5. Resolución de acertijos lógicos 80

4.5.1. El dilema del pretendiente de Portia 80

4.5.2. Caballeros y pícaros 81

4.5.3. ¿Existe Superman? 83

4.5.4. El dilema de la lógica clásica 85

4.6. Ejercicios 86

Forma abreviada en la prueba de implicaciones 86

Suponiendo el antecedente 86

Acertijos y razonamientos en lenguaje corriente 87

En el capítulo anterior, definimos el cálculo proposicional, y discutimos diferentes estrategias de demostraciones, además de probar numerosos teoremas. En este capítulo damos cierta flexibilidad al uso del cálculo proposicional. Primero, introducimos una extensión a nuestro formato de prueba de modo de abreviar demostraciones en donde se presenten implicaciones y mostramos cómo presentar pruebas en un formato menos riguroso. Luego, veremos una aplicación del cálculo proposicional: formalizaremos razonamientos realizados en lenguaje corriente mediante expresiones booleanas, para luego determinar si éstas resultan teoremas.

4.1. Forma abreviada en la prueba de implicaciones

Dejemos por un momento nuestro cálculo proposicional y consideremos relaciones aritméticas. Supongamos que sabemos cierta la siguiente igualdad $b = d - 1$. Como sabemos que $d - 1 < d$, podemos asegurar que $b < d$. Es decir, hemos probado que $b < d$ utilizando la ley de transitividad: $x = y \wedge y < z \Rightarrow x < z$.

Podemos extender el formato de prueba utilizado para igualdades entre expresiones, del siguiente modo:

$$\begin{array}{ll} & b \\ = & \quad \langle \text{razón por la cual } b = d - 1 \rangle \\ & d - 1 \\ < & \quad \langle \text{definición de } < \rangle \\ & d \end{array}$$

Aquí estamos haciendo uso implícito de la ley de transitividad: $x = y \wedge y < z \Rightarrow x < z$, con $x, y, z := b, d - 1, d$.

Un formato similar de prueba es también válido siempre que tengamos una relación que satisfaga leyes de transitividad. Es decir, si se tiene una relación $\circ$ que satisface reglas como: $x = y \wedge y \circ z \Rightarrow x \circ z$ y $x \circ y \wedge y \circ z \Rightarrow x \circ z$. En particular, podemos extender el formato de prueba en nuestro cálculo proposicional al formato indicado arriba, debido a que contamos con los teoremas 3.81. Entonces, supongamos cierto que $p \equiv q$ y que $q \Rightarrow r$, podemos demostrar que $p \Rightarrow r$ a través de la siguiente demostración:

$$\begin{array}{ll} & p \\ = & \quad \langle \text{razón por la cual } p \equiv q \rangle \\ & q \\ \Rightarrow & \quad \langle \text{razón por la cual } q \Rightarrow r \rangle \\ & r \end{array}$$

Para aceptar este nuevo formato de prueba, debemos probar que podemos transformar esta prueba en una que no utilice la extensión que permitimos arriba, aquí presentamos la prueba que parte del teorema 3.81b

$$\begin{array}{ll} & (p \equiv q) \wedge (q \Rightarrow r) \Rightarrow (p \Rightarrow r) \\ = & \quad \langle \text{razón por la cual } p \equiv q \equiv true \rangle \\ & true \wedge (q \Rightarrow r) \Rightarrow (p \Rightarrow r) \\ = & \quad \langle \text{razón por la cual } q \Rightarrow r \equiv true \rangle \\ & true \wedge true \Rightarrow (p \Rightarrow r) \\ = & \quad \langle \text{(3.38) y (3.72), } p \wedge p \equiv p \text{ y } true \Rightarrow p \equiv p \rangle \\ & (p \Rightarrow r) \end{array}$$

Los siguiente teoremas pueden probarse utilizando el nuevo formato de prueba.

(4.1) Teorema. $p \Rightarrow (q \Rightarrow p)$

Antes de enunciar los teoremas que siguen, diremos que una función booleana se dice *monótona* si $(x \Rightarrow y) \Rightarrow (f.x \Rightarrow f.y)$.

(4.2) Teorema. Monotonía de $\vee : (p \Rightarrow q) \Rightarrow (p \vee r \Rightarrow q \vee r)$

(4.3) Teorema. Monotonía de $\wedge : (p \Rightarrow q) \Rightarrow (p \wedge r \Rightarrow q \wedge r)$

Vamos a ilustrar la prueba del teorema 4.2, utilizando el formato abreviado de pruebas para implicaciones. Comenzaremos por el consecuente, pues tiene más estructura, y lo transformaremos en el antecedente $(p \Rightarrow q)$.

$$
\begin{array}{ll}
 & p \vee r \Rightarrow q \vee r \\
= & \quad \langle \text{(3.56)}, p \Rightarrow q \equiv p \vee q \equiv q, \text{ con } p, q := p \vee r, q \vee r \rangle \\
 & p \vee r \vee q \vee r \equiv q \vee r \\
= & \quad \langle \text{(3.26)}, r \vee r \equiv r \rangle \\
 & p \vee q \vee r \equiv q \vee r \\
= & \quad \langle \text{(3.27)}, p \vee (q \equiv r) \equiv p \vee q \equiv p \vee r, \text{ con } p, q, r := r, p \vee q, q \rangle \\
 & (p \vee q \equiv q) \vee r \\
\Leftarrow & \quad \langle \text{(3.75a)}, p \Rightarrow p \vee q \rangle \\
 & p \vee q \equiv q \\
= & \quad \langle \text{(3.56)}, p \Rightarrow q \equiv p \vee q \equiv q \rangle \\
 & p \Rightarrow q
\end{array}
$$

Observemos que si comenzamos con el consecuente, estamos forzados a utilizar la consecuencia $\Leftarrow$, mientras que si empezamos con el antecedente como en el ejemplo que sigue, usamos la implicación $\Rightarrow$:

$$
\begin{array}{ll}
 & p \Rightarrow q \\
= & \quad \langle \text{(3.56)}, p \Rightarrow q \equiv p \vee q \equiv q \rangle \\
 & p \vee q \equiv q \\
\Rightarrow & \quad \langle \text{(3.75a)}, p \Rightarrow p \vee q \rangle \\
 & (p \vee q \equiv q) \vee r \\
= & \quad \langle \text{(3.27)}, p \vee (q \equiv r) \equiv p \vee q \equiv p \vee r, \text{ con } p, q, r := r, p \vee q, q \rangle \\
 & p \vee q \vee r \equiv q \vee r \\
= & \quad \langle \text{(3.26)}, r \vee r \equiv r \rangle \\
 & p \vee r \vee q \vee r \equiv q \vee r \\
= & \quad \langle \text{(3.56)}, p \Rightarrow q \equiv p \vee q \equiv q, \text{ con } p, q := p \vee r, q \vee r \rangle \\
 & p \vee r \Rightarrow q \vee r
\end{array}
$$

En esta última prueba, aparecen pasos intermedios bastante artificiosos, por ejemplo, en el segundo paso aparece el operando r en la disyunción sin ningún motivo aparente. Este ejemplo muestra que la dirección que se toma en una prueba puede determinar que resulte simple o no, la obtención de la expresión buscada.

4.2. Suponiendo el antecedente

En matemática, una práctica común de prueba de una implicación $P \Rightarrow Q$, consiste en suponer cierto el antecedente P, y luego demostrar el consecuente Q. Cuando decimos que "suponemos cierto el antecedente", nos estamos refiriendo a él, momentáneamente, como un axioma y por lo tanto, equivalente a $true$. En la prueba del consecuente Q, cada variable del nuevo axioma P es considerada como constante, con lo cual la regla de Sustitución 1.6 no puede ser usada para reemplazar variables.

Para justificar este método presentamos el siguiente metateorema

(4.4) **Metateorema.** Teorema de deducción: Supongamos que si agregamos $P_1, \ldots, P_n$ como axiomas a nuestro cálculo proposicional, con las variables involucradas en cada P_i como constantes, podemos demostrar Q, entonces $P_1 \wedge \ldots \wedge P_n \Rightarrow Q$ es un teorema.

La prueba de este metateorema consiste en demostrar cómo una prueba de Q usando como axiomas adicionales $P_1, \ldots, P_n$ puede transformarse mecánicamente en la prueba de $P_1 \wedge \ldots \wedge P_n \Rightarrow Q$. No se probará este metateorema porque su demostración, si bien no es difícil, es bastante larga y engorrosa. Se ilustrará con un ejemplo cómo puede realizarse esta transformación.

(4.5) **Ejemplo.** Se quiere demostrar la siguiente proposición: $p \wedge q \Rightarrow (p \equiv q)$

El formato abreviado de demostración será el siguiente:

$$
\begin{array}{ll}
 & p \\
= & \quad \langle \text{suponiendo cierto } p \rangle \\
 & true \\
= & \quad \langle \text{suponiendo cierto } q \rangle \\
 & q
\end{array}
$$

Si una prueba es larga, puede resultar complicado recordar los supuestos, en este caso situamos las suposiciones al comienzo de la prueba como en el siguiente ejemplo.

Suponiendo p, q

$$
\begin{array}{ll}
 & p \\
= & \quad \langle \text{Suponiendo } p \rangle \\
 & true \\
= & \quad \langle \text{Suponiendo } q \rangle \\
 & q
\end{array}
$$

4.3. Analizando razonamientos en lenguaje corriente

Podemos determinar si un razonamiento en lenguaje corriente es correcto, formalizándolo primero mediante expresiones booleanas y probando luego que esta formalización resulta un teorema. Consideremos el siguiente razonamiento que comienza con dos frases en español,

cada una de las cuales se establece como verdadera. Éstas son seguidas por una conclusión que se supone, sigue de las dos frases anteriores:

> *Si Juan no termina el trabajo final correspondiente a la materia T-504, entonces no aprobará la materia T-504. Si Juan no aprueba la materia T-504, entonces no puede graduarse. Entonces, si Juan logra graduarse, deberá haber terminado su trabajo final.*

Llamemos P_0 y P_1 a las primeras dos frases de este razonamiento, y llamemos C a la conclusión. Entonces de la veracidad de P_0 y P_1 debe derivarse C. Es decir, debemos probar que $P_0 \wedge P_1 \Rightarrow C$.

Ahora traducimos las frases y la conclusión al cálculo proposicional, asociamos primero variables proposicionales a las proposiciones elementales:

s : *Juan termina el trabajo final correspondiente a la materia T-504.*
f : *Juan no aprueba la materia T-504.*
g : *Juan logra graduarse.*

Podemos ahora formalizar las frases así:

P_0 : $\neg s \Rightarrow f$
P_1 : $f \Rightarrow \neg g$
C : $g \Rightarrow s$

Para determinar si el razonamiento anterior es correcto, debemos probar que $P_0 \wedge P_1 \Rightarrow C$, es decir:

$$(\neg s \Rightarrow f) \wedge (f \Rightarrow \neg g) \Rightarrow (g \Rightarrow s)$$

Vamos a probar este teorema transformando el antecedente en el consecuente:

$$\begin{array}{ll} & (\neg s \Rightarrow f) \wedge (f \Rightarrow \neg g) \\ \Rightarrow & \quad \langle \text{(3.81a)},\ (p \Rightarrow q) \wedge (q \Rightarrow r) \Rightarrow (p \Rightarrow r) \rangle \\ & \neg s \Rightarrow \neg g \\ = & \quad \langle \text{(3.60)},\ p \Rightarrow q \equiv \neg q \Rightarrow \neg p \rangle \\ & g \Rightarrow s \end{array}$$

4.4. Construyendo contraejemplos

Cuando un razonamiento es incorrecto, una forma de probar su invalidez es a través de un *contraejemplo*, es decir, una asignación de valores a sus variables o proposiciones elementales que haga que el razonamiento completo sea falso. Por ejemplo:

> *Si X es mayor que cero, entonces si Y es cero, entonces Z es cero. El valor de Y es cero. Entonces, o bien X es mayor que cero o bien Z es cero.*

El razonamiento consiste de dos premisas y una conclusión. Consideraremos ahora variables para representar a sus proposiciones elementales.

x : *X es mayor que cero.*
y : *Y es cero.*
z : *Z es cero.*

Podemos formalizar el razonamiento anterior así:

$$(x \Rightarrow (y \Rightarrow z)) \wedge y \Rightarrow x \vee z \tag{4.1}$$

El antecedente tiene la mayor estructura, entonces planteamos:

$$\begin{array}{ll} & (x \Rightarrow (y \Rightarrow z)) \wedge y \\ = & \quad \langle \text{(3.64) Traslación, dos veces} \rangle \\ & (y \Rightarrow (x \Rightarrow z)) \wedge y \\ = & \quad \langle \text{(3.65) } p \wedge (p \Rightarrow q) \equiv p \wedge q \rangle \\ & (x \Rightarrow z) \wedge y \end{array}$$

Comparemos la última forma de escribir el antecedente con el consecuente $x \vee z$. La variable y no aparece en el consecuente y además $x \Rightarrow z$ que equivale a $\neg x \vee z$, no implica $x \vee z$. Podemos sospechar que (5.3) no es válida, y entonces el razonamiento propuesto no es correcto. Entonces, buscamos un contraejemplo, es decir, basándonos en la forma de la expresión (5.3), determinamos valores de sus variables de modo que la expresión resulte $false$. Observemos que (5.3), es una implicación, entonces debemos elegir el consecuente falso, es decir $x = z = false$ y por otra parte el antecedente verdadero, lo cual exige que $y = true$. Entonces, el contraejemplo que se obtuvo es $x = z = false$ e $y = true$.

4.5. Resolución de acertijos lógicos

En esta sección usaremos el cálculo proposicional para la resolución de acertijos lógicos. La ventaja que ofrece el cálculo proposicional es la de encontrar soluciones más elegantes a problemas como los planteados en el Capítulo 2.

4.5.1. El dilema del pretendiente de Portia

Consideremos el siguiente acertijo, que es una simplificación de una situación planteada en *El Mercader de Venecia* de *W. Shakespeare*.

Portia tiene dos cofres, uno de oro y otro de plata y ha colocado un retrato de ella en uno de estos cofres. En cada cofre, además ha escrito las siguiente leyendas:

Oro: *El retrato no está aquí.*
Plata: *Exactamente una de estas leyendas es verdadera.*

Portia le explica a su pretendiente que cada leyenda puede ser verdadera o falsa, pero que ella ha ubicado su retrato en uno de los cofres de modo que resulte consistente con la verdad o falsedad de las inscripciones. Si su pretendiente es capaz de elegir el cofre donde está guardado el retrato, Portia se casará con él. El problema para el pretendiente, es el de usar las leyendas (aún cuando puedan ser ciertas o no) de modo de determinar cuál cofre contiene el retrato.

Para comenzar a resolver el problema, lo formalizaremos introduciendo cuatro variables proposicionales:

co: *El retrato está en el cofre de oro.*
cp : *El retrato está en el cofre de plata.*
o : *El retrato no está en el cofre de oro.*
p : *Exactamente una entre o y p es $true$.*

Observemos que o es la leyenda en el cofre de oro y p es la del cofre de plata. Para modelizar el problema, procedemos así:

Primero, debemos reflejar el hecho de que el retrato está en exactamente un cofre, lo cual lo diremos del siguiente modo:

$$P_0 : co \equiv \neg cp$$

Ahora, notemos que la leyenda o en el cofre de oro es la negación de co:

$$P_1 : o \equiv \neg co$$

Del mismo modo que vimos para el caso de P_0, tenemos que la leyenda p en el cofre de plata es equivalente a $p \equiv \neg o$. No estamos asegurando con esto que $p \equiv \neg o$ es un hecho, dado que no sabemos que la leyenda sea verdadera, lo único que podemos asegurar es que la leyenda p equivale a $p \equiv \neg o$. Es decir,

$$P_2 : p \equiv (p \equiv \neg o)$$

Las expresiones P_0, P_1 y P_2 formalizan el problema. Ahora determinaremos si es posible derivar co o cp de ellas. Tomamos P_2 para comenzar, porque es la que ofrece la mayor estructura:

$$\begin{array}{ll} & p \equiv p \equiv \neg o \\ = & \quad \langle \text{(3.2) Conmutatividad de } \equiv, \neg o \equiv p \equiv p \equiv \neg o \rangle \\ & \neg o \\ = & \quad \langle P_1;\ \text{(3.12) Doble negación} \rangle \\ & co \end{array}$$

Es decir, de P_1 y P_2 (no precisamos P_0), concluímos co. El retrato está en el cofre de oro.

4.5.2. Caballeros y pícaros

Haremos nuevamente una visita a la isla de los *caballeros* y los *pícaros*. Recordemos que esta isla está habitada solamente por estos dos tipos de personas. Los caballeros tienen la particularidad de que sólo dicen la verdad, mientras que los pícaros siempre mienten.

Tenemos dos personas, A y B habitantes de la isla. A hace la siguiente afirmación: *Al menos uno de nosotros es pícaro*. ¿Qué son A y B?

Si tomamos las proposiciones elementales

a : *A es un caballero.*
b : *B es un caballero.*

podemos simbolizar la afirmación que hace A como $\neg a \vee \neg b$. Si sumamos a esto, que esa afirmación es verdadera si y sólo si A es un caballero, nos queda como dato del problema que

$$a \equiv \neg a \vee \neg b$$

Aplicando las técnicas correspondientes al Cálculo proposicional:

$$\begin{array}{ll} & a \equiv \neg a \vee \neg b \\ = & \quad \langle \text{(3.47a)}, \neg p \vee \neg q \equiv \neg\,(p \wedge q) \rangle \\ & a \equiv \neg\,(a \wedge b) \\ = & \quad \langle \text{(3.9)}, \neg\,(p \equiv q) \equiv \neg p \equiv q, \text{ con } p, q := a \wedge b, a \rangle \\ & \neg\,(a \equiv (a \wedge b)) \\ = & \quad \langle \text{(3.59)}, p \Rightarrow q \equiv p \wedge q \equiv p, \text{ con } p, q := a, b \rangle \\ & \neg\,(a \Rightarrow b) \\ = & \quad \langle \text{(3.58)}, p \Rightarrow q \equiv \neg p \vee q, \text{ con } p, q := a, b \rangle \\ & \neg\,(\neg a \vee b) \\ = & \quad \langle \text{(3.47b)}, \neg\,(p \vee q) \equiv \neg p \wedge \neg q, \text{ con } p, q := \neg a, b \rangle \\ & \neg\neg a \wedge \neg b \\ = & \quad \langle \text{(3.12)}, \neg\neg p \equiv p, \text{ con } p := a \rangle \\ & a \wedge \neg b \end{array}$$

Podemos concluir entonces que A es un caballero y B es un pícaro.
Veamos ahora otra situación también analizada anteriormente:

Nuevamente tenemos dos personas, A y B habitantes de la isla. A dice: *Soy un pícaro pero B no lo es*. ¿Qué son A y B?

Si tomamos las proposiciones elementales

a : *A es un caballero*
b : *B es un caballero*

podemos simbolizar la afirmación que hace A como $\neg a \wedge b$. Si sumamos a esto que esa afirmación es verdadera si y sólo si A es un caballero, nos queda como dato del problema que

$$a \equiv \neg a \wedge b$$

Comenzamos con esta expresión:

$$
\begin{array}{ll}
 & a \equiv \neg a \wedge b \\
= & \quad \langle \text{(3.35)}, p \wedge q \equiv p \equiv q \equiv p \vee q, \text{ con } p, q := \neg a, b \rangle \\
 & a \equiv \neg a \equiv b \equiv \neg a \vee b \\
= & \quad \langle \text{(3.15)}, p \equiv \neg p \equiv false, \text{ con } p := a \rangle \\
 & false \equiv b \equiv \neg a \vee b \\
= & \quad \langle \text{(3.15)}, p \equiv \neg p \equiv false, \text{ con } p := b \rangle \\
 & \neg b \equiv \neg a \vee b \\
= & \quad \langle \text{(3.9)}, \neg (p \equiv q) \equiv \neg p \equiv q, \text{ con } p, q := b, a \vee b \rangle \\
 & \neg (b \equiv \neg a \vee b) \\
= & \quad \langle \text{(3.56)}, p \Rightarrow q \equiv p \vee q \equiv q, \text{ con } p, q := \neg a, b \rangle \\
 & \neg (\neg a \Rightarrow b) \\
= & \quad \langle \text{(3.58)}, p \Rightarrow q \equiv \neg p \vee q, \text{ con } p, q := \neg a, b \rangle \\
 & \neg (a \vee b) \\
= & \quad \langle \text{(3.47b)}, \neg (p \vee q) \equiv \neg p \wedge \neg q, \text{ con } p, q := a, b \rangle \\
 & \neg a \wedge \neg b
\end{array}
$$

Se concluye entonces que ambos son pícaros.

4.5.3. ¿Existe Superman?

Consideremos el siguiente razonamiento acerca de la existencia de Superman.

> *Si Superman fuera capaz de evitar el mal y quisiera hacerlo, lo haría. Si Superman fuera incapaz de evitar el mal, no sería omnipotente; si no quisiera evitar el mal sería malévolo. Superman no evita el mal. Si Superman existe, es omnipotente y no es malévolo. Luego, Superman no existe.*

Para representar este razonamiento mediante el cálculo proposicional, identificamos primero las proposiciones elementales mediante variables proposicionales, luego encontramos las fórmulas que simbolizarán las proposiciones más complejas y demostraremos que la fórmula asociada a la conclusión, efectivamente se deduce de las asociadas a las premisas. Lo cual indica que, expresando la argumentación anterior en lenguaje proposicional, el razonamiento involucrado en él se traduce en la demostración de un teorema.

Sean entonces las siguientes variables proposicionales:

c : *Superman es capaz de evitar el mal.*
q : *Superman quiere evitar el mal.*
i : *Superman no es omnipotente.*
m : *Superman es malévolo.*
p : *Superman evita el mal.*
e : *Superman existe.*

Las premisas se expresan de la siguiente manera:

$P_0 : c \wedge q \Rightarrow p$ *(Si Superman fuera capaz de evitar el mal y quisiera hacerlo, lo haría).*

$P_1 : (\neg c \Rightarrow i) \wedge (\neg q \Rightarrow m)$ *(Si Superman fuera incapaz de evitar el mal, no sería omnipotente; si no quisiera evitar el mal sería malévolo).*

$P_2 : \neg p$ *(Superman no evita el mal).*

$P_3 : e \Rightarrow \neg i \wedge \neg m$ *(Si Superman existe, es omnipotente y no es malévolo).*

La conclusión es:

$\neg e$ *(Superman no existe).*

El razonamiento acerca de la existencia de Superman equivale a la expresión booleana

$$P_0 \wedge P_1 \wedge P_2 \wedge P_3 \Rightarrow \neg e$$

Una forma de probar esta expresión es suponer el antecedente y demostrar el consecuente. Entonces comenzaremos manipulando el consecuente $\neg e$. Comenzando con $\neg e$, la única suposición donde e aparece es P_3, si transformamos P_3 en su correspondiente contrarrecíproco (3.60), $\neg (\neg i \wedge \neg m) \Rightarrow \neg e$, aparece $\neg e$. En la prueba que sigue consideraremos con frecuencia cada término de la conjunción de P_1 como verdadero, ya que nuestro supuesto es que P_1 es un axioma. También usaremos el teorema 4.2, $(p \Rightarrow q) \Rightarrow (p \vee r \Rightarrow q \vee r)$, en el siguiente sentido:

$$\begin{array}{ll} & (\neg c \Rightarrow i) \Rightarrow (\neg c \vee r \Rightarrow i \vee r) \\ = & \quad \langle \text{ de } P_0 \text{ sigue que } \neg c \Rightarrow i \equiv true \rangle \\ & true \Rightarrow (\neg c \vee r \Rightarrow i \vee r) \\ = & \quad \langle \text{true es neutro a izquierda de } \Rightarrow \rangle \\ & (\neg c \vee r \Rightarrow i \vee r) \end{array}$$

Comenzamos la demostración:

$$\begin{array}{ll} & \neg e \\ \Leftarrow & \quad \langle \text{(3.60) Contrarrecíproco } \neg (\neg i \wedge \neg m) \Rightarrow \neg e \rangle \\ & \neg (\neg i \wedge \neg m) \\ = & \quad \langle \text{(3.47a) De Morgan y (3.12) Doble negación, dos veces} \rangle \\ & m \vee i \\ \Leftarrow & \quad \langle \text{primera conjunción de } P_1 \text{ y (4.2) Monotonía} \rangle \\ & \neg c \vee m \\ \Leftarrow & \quad \langle \text{segunda conjunción de } P_1 \text{ y (4.2) Monotonía} \rangle \\ & \neg c \vee \neg q \\ = & \quad \langle \text{(3.47a) De Morgan } \rangle \\ & \neg (c \wedge q) \\ \Leftarrow & \quad \langle \text{(3.60) Contrarrecíproco de } P_0 \ \neg p \Rightarrow \neg (c \wedge q) \rangle \\ & \neg p \\ = & \quad \langle P_2 \rangle \\ & true \end{array}$$

Esto último es P_2 que es verdadero, según nuestro supuesto.

4.5.4. El dilema de la lógica clásica

Consideremos ahora el siguiente razonamiento conocido como el dilema de la lógica clásica.

> *Si el general era leal, hubiera obedecido las órdenes, y si era inteligente las hubiera comprendido. O el general desobedeció las órdenes o no las comprendió. Luego, el general era desleal o no era inteligente.*

Identificamos las siguientes proposiciones elementales:

l : *El general es leal.*
o : *El general obedece las órdenes.*
i : *El general es inteligente.*
c : *El general comprende las órdenes.*

Las premisas se expresan de la siguiente manera:

$P_0 : l \Rightarrow o$ *(Si el general era leal, hubiera obedecido las órdenes).*
$P_1 : i \Rightarrow c$ *(Si el general era inteligente hubiera comprendido las órdenes).*
$P_2 : \neg o \vee \neg c$ *(O el general desobedeció las órdenes o no las comprendió).*

Y la conclusión es:

$\neg l \vee \neg i$ *(El general era desleal o no era inteligente).*

Observemos primero que usando el contrarrecíproco, P_0 puede expresarse también como $\neg o \Rightarrow \neg l$ y P_1 como $\neg c \Rightarrow \neg i$.

Probaremos la expresión booleana

$$P_0 \wedge P_1 \wedge P_2 \Rightarrow \neg l \vee \neg i,$$

suponiendo válido el antecedente, es decir cada una de las premisas serán consideradas axiomas. Comenzamos entonces por el consecuente:

$$\begin{array}{ll} & \neg i \vee \neg l \\ \Leftarrow & \quad \langle \text{contrarrecíproco de } P_1 \text{ y (4.2) Monotonía} \rangle \\ & \neg c \vee \neg l \\ \Leftarrow & \quad \langle \text{contrarrecíproco de } P_0 \text{ y (4.2) Monotonía} \rangle \\ & \neg o \vee \neg c \\ = & \quad \langle P_2 \rangle \\ & true \end{array}$$

Y esto último es P_2 que es verdadera de acuerdo a nuestro supuesto.

4.6. Ejercicios

Forma abreviada en la prueba de implicaciones

En los siguientes ejercicios, utilizar el formato abreviado de prueba de implicaciones.

4.1 Probar el teorema 4.1, $p \Rightarrow (q \Rightarrow p)$.

4.2 Probar la Monotonía de $\wedge$ (4.3), $(p \Rightarrow q) \Rightarrow (p \wedge r \Rightarrow q \wedge r)$.

Sugerencia: Comenzar con el consecuente, pues tiene la mayor estructura.

4.3 Probar Fortalecimiento y debilitamiento (3.75d), $p \vee (q \vee r) \Rightarrow p \vee q$.

Sugerencia: Comenzar con el antecedente, pues tiene mayor estructura y luego distribuir.

4.4 Probar $(p \Rightarrow q) \wedge (r \Rightarrow s) \Rightarrow (p \vee r \Rightarrow q \vee s)$.

Sugerencia: reemplazar la implicación en el antecedente por alguna definición alternativa, distribuir tanto como sea posible, luego usar el teorema 3.75d y algún teorema de absorción.

4.5 Probar $(p \Rightarrow q) \wedge (r \Rightarrow s) \Rightarrow (p \wedge r \Rightarrow q \wedge s)$.

Sugerencia: Antes de la prueba, usar el teorema 3.64 para trasladar $p \wedge r$ al antecedente.

Suponiendo el antecedente

En los siguientes ejercicios, utilizar el método de prueba que, suponiendo verdadero el antecedente, demuestra el consecuente:

4.6 Probar $p \Rightarrow (q \Rightarrow p)$.

4.7 Probar $(\neg p \Rightarrow q) \Rightarrow ((p \Rightarrow q) \Rightarrow q)$.

4.8 Probar $p \wedge q \Rightarrow (p \equiv q)$.

4.9 Probar $(p \Rightarrow p') \wedge (q \Rightarrow q') \Rightarrow (p \vee q \Rightarrow p' \vee q')$.

4.10 Probar Modus ponens (3.76), $p \wedge (p \Rightarrow q) \Rightarrow q$.

Acertijos y razonamientos en lenguaje corriente

4.11 Analizar los razonamientos presentados en el ejercicio 2.9 del Capítulo 2, usando las herramientas introducidas en esta sección.

4.12 Considérese la siguiente argumentación:

> *Si Dios quisiera evitar el mal pero fuera incapaz de hacerlo no sería omnipotente, si fuera capaz de evitar el mal pero no quisiera hacerlo sería malévolo. El mal sólo puede existir si Dios no puede o no quiere impedirlo. El mal existe. Si Dios existe, es omnipotente y no es malévolo. Luego, Dios no existe.*

Determinar si es correcta y demostrarla.

4.13 Formalizar los siguientes razonamientos y, o bien probar que son válidos o bien encontrar un contraejemplo:

a) O bien el programa no termina o bien n toma eventualmente el valor 0. Si n toma el valor 0, m tomará eventualmente el valor 0. El programa termina. Entonces, m eventualmente tomará el valor 0.

b) Si la inicialización es correcta y el bucle termina, entonces P es $true$ en su estado final. P es $true$ en su estado final. Entonces, si la inicialización es correcta, el bucle termina.

c) Si hay un hombre en la luna, la luna es de queso, y si la luna es de queso, yo soy un mono. O bien no hay ningún hombre en la luna o bien la luna no es de queso. Entonces o bien la luna no es de queso, o bien yo soy un mono.

d) Si Juan ama a María, entonces o bien mamá está loca o bien papá está triste. Papá está triste. Entonces, si mamá está loca entonces Juan no ama a María.

4.14 Probar que el siguiente razonamiento es válido si el conector "o" es considerado inclusivo, e inválido si es considerado exclusivo.

> *Si un algoritmo es confiable, entonces es correcto. Entonces, o bien el algoritmo es correcto o bien no es confiable.*

4.15 Supongamos que Portia puso una daga en uno de los siguientes tres cofres y además colocó las inscripciones:

cofre de oro	*La daga está en este cofre.*
cofre de plata	*La daga no está en este cofre.*
cofre de plomo	*A lo sumo uno de estos cofres tiene una inscripción verdadera.*

Portia le pide a su pretendiente que elija un cofre que no contenga la daga. ¿Cuál cofre debe elegir el pretendiente? Formalizar y obtener una respuesta.

4.16 Estamos en la isla de los caballeros y los pícaros. Los caballeros siempre dicen la verdad y los pícaros siempre mienten. Para formalizar los siguientes problemas, utilizar las variables booleanas que se indican:

b : *B es un caballero.*
c : *C es un caballero.*
d : *D es un caballero.*

Si B hace la afirmación "X", esto da origen a la expresión $b \equiv X$, ya que si b, entonces B es un caballero y dice la verdad, y si $\neg b$, B es un pícaro y miente.

a) Alguien pregunta a B: "¿Es usted un caballero?", B responde "Si soy un caballero, me comeré el sombrero". Probar que B deberá comerse el sombrero.

b) El habitante B dice del habitante C:"Si C es un caballero, yo soy un pícaro". ¿Qué son B y C?

c) Se rumorea que hay oro enterrado en la isla. Usted pregunta a B si hay oro en la isla. B responde: "Hay oro en la isla si y sólo si yo soy un caballero". ¿Puede determinarse si B es un caballero o un pícaro?. ¿Puede determinarse si hay oro en la isla?

d) Tres habitantes de la isla están parados en el jardín. Un visitante de la isla que pasaba por el lugar, pregunta a B, "¿Es usted un caballero o un pícaro?". B responde, pero el visitante no puede entender la respuesta, entonces pregunta a C, "¿Qué dijo B? C contesta, "B dice que es un caballero". En eso, el tercer habitante, D, dice "No le crea a C, está mintiendo!". ¿Qué son C y D?

Sugerencia: Sólo las afirmaciones de C y D son relevantes, además, la observación de D sobre la mentira que dijo C es equivalente a decir que C es un pícaro.

e) B, C y D están los tres sentados, C dice: "Hay un caballero entre nosotros". D contesta: "Estás mintiendo". ¿Qué se puede decir acerca de la caballerosidad o picardía de los tres?.

Sugerencia: Puede describirse el hecho de que uno o tres entre ellos es un caballero mediante la expresión $b \equiv c \equiv d$, ya que esta expresión es verdadera cuando el número de operandos falsos es par.

f) Un visitante de la isla se encuentra con B, C y D, y les hace una pregunta, B responde pero el visitante no comprende, entonces le pregunta a C, ¿Qué dijo B? C responde, "B dijo que hay un caballero entre nosotros." Entonces, D agrega, "No le crea a C, está mintiendo". ¿Qué son C y D?

Sugerencia: considerar la sugerencia del problema anterior.

g) En el grupo de tres habitantes, B dice que los tres son pícaros y C dice que exactamente uno de los tres es un caballero. ¿Qué son B, C y D?

Sugerencia: considerar la sugerencia del problema anterior.

h) Dos personas se dicen del mismo tipo si son ambas caballeros o ambas pícaros. Tenemos tres personas, B, C y D. B dice: "C es un pícaro" y C dice: "B y D son del mismo tipo". ¿Qué es D?

i) B realiza por separado las siguientes afirmaciones: "Amo a María" y "Si amo a María entonces amo a Yolanda". ¿Qué es B?

CAPÍTULO 5
Cálculo de Predicados

Índice del Capítulo

5.1. Introducción 91
5.2. Predicados y Cálculo de Predicados 92
5.3. El cuantificador universal 93
5.4. El cuantificador existencial 97
5.5. Propiedades de las cuantificaciones universal y existencial 100
5.6. Aplicaciones del cálculo de predicados 102
5.7. Ejercicios 103
5.7.1. Ejercicios de traducción 103
5.7.2. Ejercicios sobre cuantificación universal 104
5.7.3. Ejercicios sobre cuantificación existencial 105
5.7.4. Ejercicios de razonamientos 106

Introduciremos los conceptos referentes al Cálculo de Predicados, el cual es una extensión del Cálculo Proposicional que ya vimos en el Capítulo 3. Esta extensión nos permitirá trabajar con expresiones que usen variables de otro tipo además del tipo booleano y nos conducirá a un sistema formal que abarque una mayor cantidad de expresiones y con un mayor poder deductivo.

5.1. Introducción

Si bien el cálculo proposicional nos permitió analizar cierto tipo de razonamientos y resolver acertijos lógicos, su poder expresivo no es suficiente para comprobar la validez de algunos razonamientos simples. Por ejemplo, consideremos el siguiente razonamiento:

"*Todos los hombres son mortales. Sócrates es hombre. Por lo tanto, Sócrates es mortal.*"

Si lo formalizamos en la lógica proposicional obtendremos que tanto las premisas como la conclusión son proposiciones elementales:

p : Todos los hombres son mortales.
q : Sócrates es hombre.
r : Sócrates es mortal.

y el razonamiento sería $p \wedge q \Rightarrow r$, el cual no resulta verdadero.

No podremos demostrar la validez de este razonamiento usando lógica proposicional, porque ésta no depende de las relaciones entre las premisas y la conclusión, sino de relaciones entre partes de las proposiciones elementales que intervienen. Si analizamos la estructura internas de estas proposiciones, encontramos que el razonamiento es de la forma:

"*Todos los A son B. C es A. Por lo tanto, C es B.*"

En las secciones siguientes veremos herramientas que nos permiten demostrar este tipo de razonamientos. Trataremos primero el uso de símbolos para representar partes de enunciados simples (llamados símbolos de predicados); y luego veremos la naturaleza de expresiones tales como "Todos los As son B".

5.2. Predicados y Cálculo de Predicados

El Cálculo Proposicional nos permitió razonar con fórmulas construídas a partir de las constantes $true$ y $false$, variables y operadores booleanos, con lo cual nos fue posible expresar afirmaciones o frases que pueden modelizarse utilizando expresiones de tipo booleano.

El Cálculo de Predicados nos permitirá razonar sobre una clase más extensa y expresiva de expresiones booleanas.

Un *predicado* es una aplicación de una función booleana cuyos argumentos pueden ser expresiones no boolenas. Utilizaremos predicados para expresar propiedades o relaciones entre objetos. Los siguientes son ejemplos de predicados: $par.i$, $igual.(x, x-z+z)$, $menor.(x, y+z)$. Los nombres de las funciones ($igual$, $menor$) son llamados *símbolos de predicados*. También utilizaremos en los predicados la notación infija, como $x < y$ en lugar de $menor.(x, y)$.

Los argumentos de los predicados pueden ser expresiones de distintos tipos. Estas expresiones son llamadas *términos*, por ejemplo los términos de los predicados dados son i, x, $x-z+z$ y $y + z$.

Diremos que una fórmula del cálculo de predicados es una expresión booleana en la cual alguna de las variables booleanas ha sido reemplazada por:

- predicados.
- cuantificaciones existenciales o universales (las cuales serán introducidas a continuación).

Por ejemplo, la expresión $x < y \wedge x = z \Rightarrow q.(x, z + x)$ es una fórmula del cálculo de predicados. Ésta contiene tres predicados: $x < y$, $x = z$ y $q.(x, x + z)$, y los términos: x, y, z y $z + x$.

El cálculo puro de predicados incluye los axiomas y reglas de inferencia del cálculo proposicional y los axiomas y reglas correspondientes a las expresiones cuantificadas que veremos a continuación.

En este cálculo los símbolos de función no serán interpretados (a exepción de la igualdad $=$). Esto significa que se hará abstracción de su significado, y se utilizarán las reglas de la lógica para manipularlos. Es decir, que el cálculo nos permitirá razonar sobre sentencias sin tener en cuenta el significado de los símbolos de predicados.

5.3. El cuantificador universal

La frase "para todo x" se formaliza con el *cuantificador universal* como $\forall\, x$. Una afirmación que contenga esta frase se simboliza mediante una *cuantificación universal*. Por ejemplo,

"*Para todo entero n, $2 \times n$ es par.*"
se escribe usando nuestra notación para la cuantificación universal como:

$$(\forall\, n : n \in \mathbb{Z} : \text{par}.(2 \times n))$$

donde $n \in \mathbb{Z}$ es $true$ si y solo si n es entero, y par.n es $true$ si y solo si n es par.

En matemática, en general se dice "$(2 \times n)$ es par, con n entero" o "para cualquier entero n, $(2 \times n)$ es par". Estas afirmaciones se interpretan igual a la dada anteriormente, es decir que se formalizan en el cálculo de predicados con la misma cuantificación universal.

El formato general de una cuantificación universal es el siguiente:

$$(\forall\, x \;:\; R.x \;:\; T.x)$$

Esta expresión se lee "para todo x que satisfaga R se satisface T". R y T son predicados que toman como argumento la variable de cuantificación x. Llamaremos a estos predicados *rango* y *término de cuantificación*, respectivamente.

A continuación veremos algunos axiomas y teoremas que nos permitirán trabajar con cuantificaciones universales. Al final del capítulo utilizaremos éstos para demostrar razonamientos como el dado en la introducción.

(5.1) **Axioma.** Rango $true$

$$(\forall\, x : true : T.x) \equiv (\forall\, x :: T.x)$$

Este axioma nos permite omitir el rango de una cuantificación universal cuando éste es $true$.

(5.2) **Axioma.** Rango unitario

$$(\forall\, x : x = N : T.x) \equiv T.N$$

donde x no aparece en la expresión N. Este axioma dice que si hay un único x posible, al que llamamos N, decir que algo vale para todos los x equivale a decir que vale para N.

Ejemplo: "Para todo entero n tal que n es 3, $2 \times n$ es par" es equivalente a "2×3 es par".

(5.3) **Axioma.** Rango vacío

$$(\forall\, x : false : T.x) \equiv true$$

Este teorema afirma que es cierto que para cualquier x que satisfaga $false$ se satisface $T.x$, para cualquier predicado T. Esta cuantificación universal es cierta dado que ningún valor que pueda tomar x satisface $false$. Llamaremos rango vacío a la expresión booleana $false$ (dado que ningún valor lo satisface).

La cuantificación universal $(\forall\, x : R.x : T.x)$ nos dice que todos los x que satisfagan R satisfacen T. A esta expresión la podemos interpretar como la conjunción de todos los $T.x$ tal que $R.x$ es $true$. Por ejemplo, la expresión:

$$(\forall\, x : 0 \leq x < 5 : b[x] > 0)$$

es equivalente a

$$b[0] > 0 \wedge b[1] > 0 \wedge b[2] > 0 \wedge b[3] > 0 \wedge b[4] > 0$$

Es decir que podemos pensar a la cuantificación universal como la generalización de la conjunción. A continuación veremos que algunas propiedades de la conjunción pasan a la cuantificación universal.

(5.4) **Axioma.** Distributividad de $\vee$ respecto de $\forall$

$$(\forall\, x : R.x : P \vee T.x) \equiv P \vee (\forall\, x : R.x : T.x)$$

donde x no aparece en P.

La conmutatividad de $\wedge$ nos permite definir el siguiente axioma.

(5.5) **Axioma.** Regla del término

$$(\forall\, x : R.x : T.x \wedge G.x) \equiv (\forall\, x : R.x : T.x) \wedge (\forall\, x : R.x : G.x)$$

Cuando una cuantificación tiene como término otra cuantificación diremos que los cuantificadores están "anidados". La siguiente ley permite intercambiar cuantificadores anidados de la siguiente manera:

(5.6) Axioma. Intercambio de cuantificadores

$$(\forall\, x :: (\forall\, y :: T.x.y)) \equiv (\forall\, y :: (\forall\, x :: T.x.y))$$

Por ejemplo, usamos este axioma para afirmar que "para todos los m se da que para todo n, $2 \times (m - n)$ es par" es lo mismo que "para todos los n se da que para todo m, $2 \times (m - n)$ es par".

También podríamos expresar la frase anterior como "para todo m y n, $2 \times (m - n)$ es par". Utilizaremos el siguiente axioma para expresar los cuantificadores anidados mediante una sola cuantificación con varias variables:

(5.7) Axioma.

$$(\forall\, x :: (\forall\, y :: T.x.y)) \equiv (\forall\, x,\, y :: T.x.y)$$

El axioma que sigue nos permitirá trasladar el rango de especificación hacia el término de cuantificación:

(5.8) Axioma. Traslación

$$(\forall\, x : R.x : T.x) \equiv (\forall\, x :: R.x \Rightarrow T.x)$$

Ejemplo: "Para todos los enteros n, $2 \times n$ es par" es equivalente a "Para todos los n, si n es entero entonces $2 \times n$ es par"

A partir de este axioma se pueden demostrar los siguientes teoremas:

(5.9) Teorema. Traslación

a) $(\forall\, x : R.x : P.x) \equiv (\forall\, x :: \neg R.x \vee P.x)$

b) $(\forall\, x : R.x : P.x) \equiv (\forall\, x :: R.x \wedge P.x \equiv R.x)$

c) $(\forall\, x : R.x : P.x) \equiv (\forall\, x :: R.x \vee P.x \equiv P.x)$

(5.10) Teorema. Variantes de traslación

a) $(\forall\, x : Q.x \wedge R.x : P.x) \equiv (\forall\, x : Q.x : R.x \Rightarrow P.x)$

b) $(\forall\, x : Q.x \wedge R.x : P.x) \equiv (\forall\, x : Q.x : \neg R.x \vee P.x)$

c) $(\forall\, x : Q.x \wedge R : P.x) \equiv (\forall\, x : Q.x : R.x \wedge P.x \equiv R.x)$

d) $(\forall\, x : Q.x \wedge R.x : P.x) \equiv (\forall\, x : Q.x : R.x \vee P.x \equiv P.x)$

Probaremos (5.10a))

$$
\begin{array}{ll}
 & (\forall\, x : Q.x \wedge R.x : P.x) \\
= & \quad \langle \text{Traslación 5.8} \rangle \\
 & (\forall\, x :: Q.x \wedge R.x \Rightarrow P.x) \\
= & \quad \langle \text{Teorema 3.64} \rangle \\
 & (\forall\, x :: Q.x \Rightarrow (R.x \Rightarrow P.x)) \\
= & \quad \langle \text{Traslación 5.8} \rangle \\
 & (\forall\, x : Q.x : R.x \Rightarrow P.x)
\end{array}
$$

El axioma que sigue muestra que la variable de la cuantificación puede ser reemplazada por cualquier otra variable que no sea utilizada en la expresión.

(5.11) Axioma. Cambio de variable

Si y no ocurre en $R.x$ ni en $T.x$ entonces

$$(\forall\, x : R.x : T.x) \equiv (\forall\, y : R.y : T.y)$$

La restricción y no ocurre en R o en T asegura que y es una variable nueva.

Aplicando este teorema al primer ejemplo, podemos afirmar que es lo mismo decir "Para todos los enteros n, $2 \times n$ es par" que decir "Para todos los enteros x, $2 \times x$ es par". Las variables de cuantificación son llamadas también "variables bobas", dado que podemos reemplazarlas por cualquier variable, lo único que hacen es indicar sobre qué se cuantifica.

El siguiente teorema formaliza la idea de que cuando el rango de especificación es una disyunción de predicados, podemos dividir la cuantificación en dos cuantificaciones cuyos rangos son cada uno de estos predicados.

(5.12) Teorema. Partición de rango

$$(\forall\, i : R.i \vee S.i : T.i) \equiv (\forall\, i : R.i : T.i) \;\wedge\; (\forall\, i : S.i : T.i)$$

Ejemplo: "Para todo número n mayor que 3 o menor que -3, $|n-3|$ es mayor a 3" es equivalente a "Para todo número n mayor que 3 , $|n-3|$ es mayor a 3 y para todo número n menor que -3, $|n-3|$ es mayor a 3 ".

Si tenemos que un predicado $f.x$ vale para todo x, en particular debería ser cierto $f.t$ para cualquier t arbitrario que uno tome. Esta idea se formaliza en el siguiente teorema.

(5.13) Teorema. Instanciación

$$(\forall\, x :: P.x) \Rightarrow P.E$$

Veamos situaciones donde la regla 5.13 es de gran ayuda. Por ejemplo, supongamos que queremos probar

$$B \vee par.(i+j) \equiv B \vee par.(i+j)^2$$

y que vale:

$$(\forall\, x :: par.x \equiv par.x^2) \tag{5.1}$$

Usaremos Instanciación 5.13 para dar la siguiente prueba:

$$\begin{array}{ll} & B \vee par(i+j) \\ = & \quad \langle \text{ supuesto 5.1, Instanciación 5.13 con } E = i+j \rangle \\ & B \vee par((i+j)^2) \end{array}$$

5.4. El cuantificador existencial

Otra clase de cuantificador, necesario para expresar afirmaciones o frases del lenguage corriente, es el *cuantificador existencial*.

La frase "existe un x" se formaliza con este cuantificador como $\exists\, x$. Por ejemplo, la frase "existe al menos un número entre 10 y 20 que es primo", la cual puede expresarse también como "algún número entre 10 y 20 es primo", se formaliza con la siguiente *cuantificación existencial*:

$$(\exists\, x : 10 \leq x \leq 20 : primo.x) \tag{5.2}$$

donde $primo.x$ es $true$ si y solo si x es primo.

El formato general de una cuantificación existencial es el siguiente:

$$(\exists\, x : R.x : P.x) \tag{5.3}$$

el cual se lee "existe x en el rango R que satisface P". Al igual que en la cuantificación universal, R y T son predicados que toman como argumento la variable x, y se denominan rango y término de cuantificación respectivamente.

La cuantificación existencial se puede pensar como la generalización de la disyunción. Esto hace que algunas propiedades de la disyunción pasen a esta cuantificación.

El siguiente axioma caracteriza a la cuantificación existencial, relacionandola con la cuantificación universal.

(5.14) Axioma. De Morgan Generalizado

$$(\exists\, x : R.x : T.x) \equiv \neg(\forall\, x : R.x : \neg T.x)$$

Llamamos a este axioma De Morgan Generalizado, dado que es una generalización de la ley de De Morgan (3.47a), $\neg(p \wedge q) \equiv \neg p \vee \neg q$. El siguiente ejemplo ilustra la idea de esta generalización.

La expresión $(\exists\, x : 0 \leq x < 4 : par.x)$ también puede escribirse como $par.0 \vee par.1 \vee par.2 \vee par.3$. Si aplicamos los teoremas Doble negación 3.12 y De Morgan 3.47a varias veces obtenemos la expresión $\neg(\neg par.0 \wedge \neg par.1 \wedge \neg par.2 \wedge \neg par.3)$, la cual puede escribirse utilizando cuantificadores como: $\neg(\forall\, x : 0 \leq x < 4 : \neg par.x)$.

A partir de este axioma se pueden demostrar los siguientes teoremas:

(5.15) Teorema. Formas alternativas de De Morgan Generalizado

a) $\neg(\exists\, x : R.x : \neg P.x) \equiv (\forall\, x : R.x : P.x)$

b) $\neg(\exists\, x : R.x : P.x) \equiv (\forall\, x : R.x : \neg P.x)$

c) $(\exists\, x : R.x : \neg P.x) \equiv \neg(\forall\, x : R.x : P.x)$

El axioma De Morgan Generalizado nos permite expresar una cuantificación existencial como una cuantificación universal. Utilizando este axioma y los definidos para el cuantificador universal, pueden demostrarse los siguientes teoremas.

(5.16) Teorema. Rango $true$

$$(\exists\, x : true : T.x) \equiv (\exists\, x :: T.x)$$

(5.17) Teorema. Rango unitario

$$(\exists\, x : x = N : T.x) \equiv T.N$$

donde x no aparece en la expresión N.

(5.18) Teorema. Rango vacío

$$(\exists\, x : false : T.x) \equiv false$$

Este teorema refleja una diferencia importante entre los cuantificadores existencial y universal. En el primero es necesario que algún elemento del rango satisfaga la propiedad dada en el término, mientras que en el segundo no lo es. Por lo tanto, cuando no existe ningún elemento que satisface el rango la cuantificación existencial es falsa mientras que la universal es verdadera.

(5.19) Teorema. Distributividad de $\wedge$ respecto de $\exists$

$$(\exists\, x : R.x : P \wedge T.x) \equiv P \wedge (\exists\, x : R.x : T.x)$$

donde x no aparece en P.

(5.20) Teorema. Regla del término

$$(\exists\, x : R.x : T.x \vee G.x) \equiv (\exists\, x : R.x : T.x) \vee (\exists\, x : R.x : G.x)$$

Este teorema es consecuencia del teorema "Conmutatividad de $\vee$".

(5.21) Teorema. Intercambio de cuantificadores

$$(\exists\, x :: (\exists\, y :: T.x.y)) \equiv (\exists\, y :: (\exists\, x :: T.x.y))$$

(5.22) Teorema.

$$(\exists\, x :: (\exists\, y :: T.x.y)) \equiv (\exists\, x,\, y :: T.x.y)$$

El teorema que nos permite trasladar el rango al término en una cuantificación existencial es muy diferente al visto para la cuantificación universal.

(5.23) Teorema. Traslación

$$(\exists\, x : R.x : T.x) \equiv (\exists\, x :: R.x \wedge T.x)$$

Para entender este teorema recordaremos el significado de $\exists$. El lado izquierdo de 5.23 se lee "existe un x en R que satisface T". Dicho de otra forma "existe un x que satisface R y T".

Los siguientes teoremas se demuestran a partir de 5.23.

(5.24) Teorema. Traslación

$$(\exists\, x : R.x \wedge Q.x : T.x) \equiv (\exists\, x : R.x : Q.x \wedge T.x)$$

(5.25) Teorema. Intercambio entre rango y término

$$(\exists\, x : R.x : T.x) \equiv (\exists\, x : T.x : R.x)$$

Al igual que en la cuantificación universal, cuando el rango de especificación es una disyunción de predicados podemos dividir la cuantificación existencial en dos cuantificaciones.

(5.26) Teorema. Partición de rango

$$(\exists\, i : R.i \vee S.i : T.i) \equiv (\exists\, i : R.i : T.i) \;\vee\; (\exists\, i : S.i : T.i)$$

A partir del teorema Instanciación podemos probar el siguiente teorema, el cual introduce una cuantificación existencial.

(5.27) Teorema. Introducción de $\exists$

$$P.E \Rightarrow (\exists\, x :: P.x)$$

El teorema nos dice que si una propiedad $P.x$ es cierta para un elemento E, entonces es cierto que existe un elemento que satisface tal propiedad.

Podremos cambiar la variable de una cuantificación existencial por cualquier otra variable no utilizada en la expresión mediante el siguiente teorema.

(5.28) Teorema. Cambio de variable

Si y no ocurre en R ni en T entonces

$$(\exists\, x : R.x : T.x) \equiv (\exists\, y : R.y : T.y)$$

5.5. Propiedades de las cuantificaciones universal y existencial

A continuación enunciaremos algunos teoremas más sobre las cuantificaciones universal y existencial. Los primeros dos teoremas son consecuencia de los teoremas 3.75a y 3.75b del cálculo proposicional.

(5.29) Teorema. Fortalecimiento de rango

$$(\forall\, x : Q.x \vee R.x : P.x) \Rightarrow (\forall\, x : Q.x : P.x)$$

(5.30) Teorema. Fortalecimiento de término

$$(\forall\, x : R.x : P.x \wedge Q.x) \Rightarrow (\forall\, x : R.x : P.x)$$

El siguiente es una generalización del teorema Monotonía de $\wedge$:

(5.31) Teorema. Monotonía de $\forall$

$$(\forall\, x : R.x : Q.x \Rightarrow P.x) \Rightarrow ((\forall\, x : R.x : Q.x) \Rightarrow (\forall\, x : R.x : P.x))$$

Los que siguen son teoremas considerados muy útiles para la manipulación de expresiones cuantificadas.

(5.32) Teorema. Distributividad de $\wedge$ respecto de $\forall$

Si x no ocurre en P y el rango de especificación es no vacío, es decir $(\exists\, x \,::\, R.x)$, entonces

$$(\forall\, x : R.x : P \wedge Q.x) \equiv P \wedge (\forall\, x : R.x : Q.x))$$

Notar que el operador $\vee$ se distribuye sobre $\forall$ sin ninguna restricción sobre el rango de especificación, mientras que $\wedge$ se distribuye sobre $\forall$ siempre que el rango no sea vacío. Esta diferencia se debe a que $true$ es absorbente para $\vee$.

(5.33) Teorema.

$$(\forall\, x : R : true) \equiv true$$

(5.34) Teorema.

$$(\forall\, x : R.x : P.x \equiv Q.x) \Rightarrow ((\forall\, x : R.x : P.x) \equiv (\forall\, x : R.x : Q.x))$$

A partir de los teoremas vistos en esta sección para el cuantificador universal pueden demostrarse los siguiente teoremas.

(5.35) Teorema. Debilitamiento de rango

$$(\exists\, x : R.x : P.x) \Rightarrow (\exists\, x : R.x \vee Q.x : P.x)$$

(5.36) Teorema. Debilitamiento de término

$$(\exists\, x : R.x : P.x) \Rightarrow (\exists\, x : R.x : P.x \vee Q.x)$$

(5.37) Teorema. Monotonía de $\exists$

$$(\forall\, x : R.x : Q.x \Rightarrow P.x) \Rightarrow ((\exists\, x : R.x : Q.x) \Rightarrow (\exists\, x : R.x : P.x))$$

(5.38) Teorema. Distributividad de $\vee$ respecto de $\exists$

Si x no ocurre en P y el rango de especificación es no vacío, es decir $(\exists\, x \,::\, R.x)$, entonces

$$(\exists\, x : R.x : P \vee Q.x) \equiv P \vee (\exists\, x : R.x : Q.x))$$

(5.39) Teorema.

$$(\exists\, x : R.x : false) \equiv false$$

El siguiente teorema permite intercambiar $\forall$ con $\exists$. La recíproca del teorema no es cierta.

(5.40) Teorema. Intercambio de cuantificadores

$$(\exists\, x : R.x : (\forall\, y : Q.y : P.x.y)) \Rightarrow (\forall\, y : Q.y : (\exists\, x : R.x : P.x.y))$$

En el capítulo 3 se definieron algunos metateoremas para el cálculo proposicional, los cuales proveen una herramienta para las demostraciones de teoremas en éste cálculo. Veremos ahora un metateorema perteneciente al cálculo de predicados, llamado metateorema del testigo. Este metateorema nos permitirá usar un nombre de constante nuevo (llamado testigo) para referirse a un elemento cuya existencia es postulada por la cuantificación.

(5.41) Metateorema. Testigo

Si k no ocurre en P ni en Q, entonces

$(\exists\, x :: P.x) \Rightarrow Q$ es un teorema si y solo si $P.k \Rightarrow Q$ es un teorema

El metateorema del testigo es usado a menudo cuando una expresión cuantificada $(\exists\, x :: P.x)$ es una xioma o un teorema y queremos probar Q usando éste.

No debe confundirse este metateorema con un teorema, es decir, con una fórmula del cálculo de predicados. La expresión "si y solo si" no puede ser interpretada como una equivalencia, dado que la afirmación que provee un metateorema es sobre el sistema formal, no es parte del mismo.

En la siguiente sección veremos un ejemplo donde el metatoerema es aplicado para demostrar la validez de un razonamiento.

5.6. Aplicaciones del cálculo de predicados

En esta sección utilizaremos el cálculo de predicados para analizar razonamientos del lenguaje natural. En la introducción dimos un ejemplo de un razonamiento que no puede ser analizado mediante el cálculo proposicional, dado que es necesario analizar la estructura interna de las proposiciones. Retomaremos este ejemplo de razonamiento, el cual se formaliza de la siguiente manera:

$$\frac{(\forall\, x : h.x : m.x) \qquad h.s}{m.s}$$

donde s es una constante que representa al individuo Sócrates y los predicados h ym son definidos como:

$h.x$: "x es un hombre"
$m.x$: "x es mortal"
Este razonamiento se traduce en la siguiente fórmula del cálculo de predicados

$$(\forall\, x : h.x : m.x) \wedge h.s \Rightarrow m.s$$

Demostraremos que esta fórmula es un teorema partiendo del lado izquierdo de la implicación.

$$
\begin{array}{ll}
 & (\forall\, x : h.x : m.x) \wedge h.s \\
= & \quad \langle \text{Traslación 5.8} \rangle \\
 & (\forall\, x :: h.x \Rightarrow m.x) \wedge h.s \\
\Rightarrow & \quad \langle \text{Instanciación, monotonía de } \wedge \rangle \\
 & (h.s \Rightarrow m.s) \wedge h.s \\
\Rightarrow & \quad \langle \text{Modus Ponens} \rangle \\
 & m.s
\end{array}
$$

Un tipo de razonamiento que tuvo gran popularidad y fue considerado el paradigma de los razonamientos válidos es el de los silogismos. Un silogismo consta de dos premisas y una conclusión que puede inferirse a partir de éstas. Los silogismos fueron formulados por primera vez por Aristóteles, quien pensaba que era posible reducir cualquier razonamiento a partir de éstos. Si bien se sabe que no es así, los silogismos siguen siendo de gran importancia para la lógica. A continuación veremos dos formas de silogismos.

La primer forma tiene como premisas y conclusión cuantificaciones universales. Este tipo de razonamiento, llamado silogismo hipotético, tiene la siguiente forma:

$$
\frac{(\forall\, x : a.x : b.x) \quad (\forall\, x : b.x : c.x)}{(\forall\, x : a.x : c.x)}
$$

donde a, b y c son predicados.

Para demostrar este razonamiento partiremos de la conjunción de las premisas y llegaremos a la conclusión.

$$
\begin{array}{ll}
 & (\forall\, x : a.x : b.x) \wedge (\forall\, x : b.x : c.x) \\
= & \quad \langle \text{Traslación 5.8} \rangle \\
 & (\forall\, x :: a.x \Rightarrow b.x) \wedge (\forall\, x :: b.x \Rightarrow c.x) \\
= & \quad \langle \text{Regla del término } \wedge \rangle \\
 & (\forall\, x :: (a.x \Rightarrow b.x) \wedge (b.x \Rightarrow c.x)) \\
\Rightarrow & \quad \langle \text{Transitividad, 5.33, monotonía de } \forall \rangle \\
 & (\forall\, x :: a.x \Rightarrow c.x)
\end{array}
$$

La segunda forma de silogismo se caracteriza por el siguiente esquema.

$$
\frac{(\forall\, x : a.x : b.x) \quad (\exists\, x : c.x : \neg b.x)}{(\exists\, x : c.x : \neg a.x)}
$$

Este razonamiento puede demostrarse utilizando el metateorema del testigo (tomando como testigo el elemento que hace cierta la segunda premisa). Para ello es necesario partir de la expresión completa.

$$
\begin{array}{ll}
 & (\forall\, x : a.x : b.x) \wedge (\exists\, x : c.x : \neg b.x) \Rightarrow (\exists\, x : c.x : \neg a.x) \\
= & \quad \langle \text{Conmutatividad de } \wedge \text{, Traslación} \rangle \\
 & (\exists\, x : c.x : \neg b.x) \Rightarrow ((\forall\, x : a.x : b.x) \Rightarrow (\exists\, x : c.x : \neg a.x)) \\
= & \quad \langle \text{Traslación 5.8} \rangle \\
 & (\exists\, x :: c.x \wedge \neg b.x) \Rightarrow ((\forall\, x : a.x : b.x) \Rightarrow (\exists\, x : c.x : \neg a.x))
\end{array}
$$

Si demostramos ahora que $c.k \wedge \neg b.k \Rightarrow ((\forall\, x : a.x : b.x) \Rightarrow (\exists\, x : c.x : \neg a.x))$ es un teorema, por el metateorema del testigo podemos concluir que $(\exists\, x :: c.x \wedge \neg b.x) \Rightarrow ((\forall\, x : a.x : b.x) \Rightarrow (\exists\, x : c.x : \neg a.x))$ es también un teorema. Se deja como ejercicio la prueba de este teorema.

5.7. Ejercicios

5.7.1. Ejercicios de traducción

5.1 Traducir las siguientes frases al lenguaje del cálculo de predicados:

a) Algún entero es mayor que 23.

b) Un número entero positivo no es negativo.

c) La suma de dos números impares es par.

d) El número 1 es el único número natural que es menor que el entero positivo p y que divide a q.

e) Cada número entero positivo es menor que el valor absoluto de algún entero negativo.

f) Los cubos de enteros nunca son pares.

g) El número real i es la mayor solución real de la ecuación $f.x = x + 1$.

h) Para ningún entero i, $f.i$ es a la vez mayor y menor que i.

i) Ningún entero es mayor que todos los demás enteros.

5.2 Traducir las siguientes fórmulas del cálculo de predicados a lenguaje corriente.

a) $(\exists\, k : \mathbf{R} : (\forall\, i : \mathbf{Z} :: f.i = k))$

b) $(\exists\, z : \mathbf{R} : (\forall\, i : \mathbf{Z} :: f.j = f(j + i \cdot z)))$

c) $(\forall\, x : x \neq m : f.x > f.m)$

d) $(\exists\, x, y : \mathbf{R} :: f.x < 0 \wedge 0 < f.y \Rightarrow (\exists\, z : \mathbf{R} :: f.z = 0))$

5.3 Formalizar las siguientes frases:

a) Cada uno ama a alguien.

b) Alguien ama a alguien.

c) Cada uno ama a cada uno.

d) Nadie ama a todos.

e) Alguien ama a nadie.

5.4 Formalizar las siguientes frases:

a) Puedes engañar a alguien por algún tiempo.

b) Puedes engañar a todos por algún tiempo.

c) No puedes engañar a todos todo el tiempo.

d) No puedes engañar a alguien todo el tiempo.

5.7.2. Ejercicios sobre cuantificación universal

5.5 Suponiendo que x no ocurre en P, probar que la Distributividad de $\vee$ respecto de $\forall$ (5.4): $P \vee (\forall\, x : R.x : Q.x) \equiv (\forall\, x : R.x : P \vee Q.x)$ sigue directamente de la misma expresión con $R.x \equiv true$, es decir $P \vee (\forall\, x :: Q.x) \equiv (\forall\, x :: P \vee Q.x)$. Lo cual significa que podríamos haber definido un axioma más simple.

5.6 Probar que $(\forall\, x : R.x : P.x) \wedge (\forall\, x : R.x : Q.x) \equiv (\forall\, x : R.x : P.x \wedge Q.x)$ sigue de la misma expresión con $R.x \equiv true$, es decir de $(\forall\, x :: P.x) \wedge (\forall\, x :: Q.x) \equiv (\forall\, x :: P.x \wedge Q.x)$.

5.7 Suponiendo que x no ocurre en P, probar $(\forall\, x : R.x : P) \equiv P \vee (\forall\, x :: \neg R.x)$. *Sugerencia*: Comenzar con el lado izquierdo aplicando Traslación, ya que en el lado derecho $R.x \equiv true$.

5.8 Probar el teorema 5.33: $(\forall\, x : R.x : true) \equiv true$.

5.9 Probar el teorema 5.34: $(\forall\, x : R.x : P.x \equiv Q.x) \Rightarrow ((\forall\, x : R.x : P.x) \equiv (\forall\, x : R.x : Q.x))$. *Sugerencia*: Reemplazar la expresión completa usando el teorema (3.61) y a continuación la Regla del Término.

5.10 Probar el teorema: Fortalecimiento de rango (5.29): $(\forall\, x : Q.x \vee R.x : P.x) \Rightarrow (\forall\, x : Q.x : P.x)$. *Sugerencia*: Será de utilidad el axioma de Partición de Rango.

5.11 Probar el teorema: Fortalecimiento de término (5.30): $(\forall\, x : R.x : P.x \wedge Q.x) \Rightarrow (\forall\, x : R.x : P.x)$. *Sugerencia*: Usar la Regla del Término.

5.12 Probar el teorema: Monotonía de $\forall$ (): $(\forall\, x : R.x : Q.x \Rightarrow P.x) \Rightarrow ((\forall\, x : R.x : Q.x) \Rightarrow (\forall\, x : R.x : P.x))$. Usar (3.64) Traslación y luego la Regla del Término

5.13 Probar el teorema: Instanciación (5.13): $(\forall\, x :: P.x) \Rightarrow P[x := E]$. *Sugerencia*: La clave es renombrar la variable cuantificada usando el teorema Cambio de variable. Elegir para ello una variable que no aparezca en $P.x$ ni en E.

5.7.3. Ejercicios sobre cuantificación existencial

5.14 Probar el teorema 5.15: Formas alternativas de De Morgan Generalizado:

a) $\neg(\exists\, x : R.x : \neg P.x) \equiv (\forall\, x : R.x : P.x)$

b) $\neg(\exists\, x : R.x : P.x) \equiv (\forall\, x : R.x : \neg P.x)$

c) $(\exists\, x : R.x : \neg P.x) \equiv \neg(\forall\, x : R.x : P.x)$

5.15 Probar el teorema 5.23: Traslación para $\exists$: $(\exists\, x : R.x : P.x) \equiv (\exists\, x :: R.x \wedge P.x)$.

5.16 Probar el teorema 5.24: Traslación para $\exists$: $(\exists\, x : Q.x \wedge R.x : P.x) \equiv (\exists\, x : Q.x : R.x \wedge P.x)$.

5.17 Suponiendo que x no ocurre en P, probar el teorema 5.32: Distributividad de $\wedge$ respecto de $\exists$: $P \wedge (\exists\, x : R.x : Q.x) \equiv (\exists\, x : R.x : P \wedge Q.x)$.

5.18 Suponiendo que x no ocurre en P, probar el teorema: $(\exists\, x : R.x : P) \equiv P \wedge (\exists\, x :: R.x)$.

5.19 Suponiendo que x no ocurre en P, probar el teorema 5.38: Distributividad de $\vee$ respecto de $\exists$: $(\exists\, x :: R.x) \Rightarrow ((\exists\, x : R.x : P.x \vee Q.x) \equiv P \vee (\exists\, x : R.x : Q.x))$.

5.20 Probar el teorema 5.39: $(\exists\, x : R.x : false) \equiv false$.

5.21 Probar el teorema 5.35: Debilitamiento de rango: $(\exists\, x : R.x : P.x) \Rightarrow (\exists\, x : Q.x \vee R.x : P.x)$.

5.22 Probar el teorema 5.36: Debilitamiento de término: $(\exists\, x : R.x : P.x) \Rightarrow (\exists\, x : R.x : P.x \vee Q.x)$.

5.23 Probar el teorema 5.37: Monotonía de $\exists$: $(\forall\, x : R.x : Q.x \Rightarrow P.x) \Rightarrow ((\exists\, x : R.x : Q.x) \Rightarrow (\exists\, x : R.x : P.x))$.

5.24 Probar el teorema 5.27: Introducción de $\exists$: $P.E \Rightarrow (\exists\, x :: P.x)$.

5.7.4. Ejercicios de razonamientos

5.25 Demostrar que el siguiente razonamiento es correcto, formalizándolo mediante cálculo de predicados y demostrándolo como teorema.

> Todos los ballenas son mamíferos. Wally es una ballena. Por lo tanto, Wally es un mamífero.

5.26 Demostrar que los siguientes razonamientos son correctos formalizándolos mediante cálculo de predicados y demostrándolos como teoremas.

a) Los brujos son considerados individuos con poderes ocultos. Algún brujo es mago. Luego, algún mago es considerado como individuo con poderes ocultos.

b) Ningún fotógrafo pinta. Todos los que no son fotógrafos son escultores. Por lo tanto, todos los pintores son escultores.

c) Ningún feo despierta pasiones. Todos los atletas despiertan pasiones. Por lo tanto, ningún atleta es feo.

CAPÍTULO 6
Expresiones cuantificadas

Índice del Capítulo

6.1. Introducción 107
6.2. Tipos 108
6.3. Sintaxis e interpretación de la cuantificación 111
6.4. Variables libres y ligadas 112
6.5. Revisión de la sustitución en expresiones cuantificadas 113
6.6. La regla de Leibniz para expresiones cuantificadas 114
6.7. Reglas generales para expresiones cuantificadas 115
6.8. Cuantificadores aritméticos 119
6.8.1. Máximos y mínimos 119
6.8.2. Operador de conteo 121
6.9. Ejercicios 121

6.1. Introducción

Las expresiones cuantificadas o cuantificaciones, nos permiten expresar una secuencia de expresiones, las cuales dependen de una variable, unidas mediante una operación. Antes de presentar la definición formal de expresión cuantificada que utilizaremos en el capítulo, veremos algunos ejemplos de estas expresiones.

Una cuantificación, muy utilizada en las matemáticas, es la sumatoria. El siguiente es un ejemplo de una expresión que denota la suma de los primeros n números impares, mediante una sumatoria:

$$\sum_{i=0}^{n-1} 2 \times i + 1$$

En esta expresión pueden distiguirse varias componentes. Por un lado el operador sumatoria, el cual se corresponde con el operador binario $+$, por otro lado la variable i junto con un rango de variación (i puede tomar los valores entre 0 y $n-1$), y por último una expresión ($2 \times n + 1$) que indica cuales van a ser los términos de la sumatoria.

Utilizando la notación para cuantificaciones que veremos en la próxima sección, la expresión anterior se expresaría como:

$$(+i : 0 \leq i < n : 2 \times i + 1)$$

Esta notación tiene varias ventajas respecto a la notación anterior (sumatoria). Algunas de ellas son:

- Los paréntesis hacen explícito el *alcance* de la variable i, es decir el ámbito en donde i está referenciada. Este alcance comprende la expresión entre paréntesis.

- Las expresiones cuantificadas presentadas con esta notación admiten cualquier expresión booleana como rango de variación de i, mientras que la anterior solo admite intervalos de números naturales. Esto facilita la escritura de las cuantificaciones. Por ejemplo, podríamos escribir la cuantificación 6.1 como:

  $$(+i : 1 \leq i \leq n \wedge impar.i : i)$$

- La nueva notación permite con mayor facilidad la participación de más de una variable cuantificada. Por ejemplo,

  $$\big(+i, j : 1 \leq i \leq 2 \wedge 3 \leq j \leq 4 : i^j\big)$$

  denota la suma $1^3+1^4+2^3+2^4$ (para determinar qué combinaciones de i^j deben sumarse, elegimos todas la combinaciones de i y j que satisfagan $1 \leq i \leq 2 \wedge 3 \leq j \leq 4$).

Otros ejemplos de expresiones cuantificadas son las cuantificaciones universal y existencial, vistas en el capítulo anterior. Éstas son llamadas cuantificaciones lógicas. También existen cuantificaciones para definir conjuntos por comprensión y para otros operadores matemáticos, como ser el producto.

Antes de definir la sintaxis de expresiones cuantificadas generales se introducirá con mayor detalle el concepto de tipo de las expresiones, el cual es de gran de importancia para comprender las cuantificaciones.

6.2. Tipos

En los lenguajes de programación, se define *tipo* como el conjunto no vacío de valores que pueden asociarse a una variable. Recordemos que este concepto fue introducido en el Capítulo 1

cuando definimos variable. Luego, en el Capítulo 2, trabajamos con expresiones booleanas y observamos que éstas sólo asumen los valores *true* y *false*, de ahora en más notaremos con $\mathbb{B}$, al conjunto de éstos valores. Comenzaremos ahora a trabajar con otros tipos que describimos en la siguiente tabla:

Nombre	Símbolo	Tipo (conjunto de valores)
enteros	$\mathbb{Z}$	enteros:$\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots$
naturales	$\mathbb{N}$	números naturales: $0, 1, 2, \ldots$
positivos	$\mathbb{Z}^+$	enteros positivos: $1, 2, 3, \ldots$
negativos	$\mathbb{Z}^-$	enteros negativos: $-1, -2, -3, \ldots$
racionales	$\mathbb{Q}$	números racionales: i/j con i, j enteros y $j \neq 0$
reales	$\mathbb{R}$	números reales
reales positivos	$\mathbb{R}^+$	números reales positivos
booleanos	$\mathbb{B}$	*true* y *false*

El concepto de tipos nos obliga a observar en detalle nuestra noción de expresión. Una expresión, no sólo debe ser una secuencia de símbolos que satisfacen reglas sintácticas predefinidas sino que a partir de ahora, debe tener también el tipo correcto. Por lo tanto algunas expresiones que antes parecían correctas, a partir de ahora no lo serán si no satisfacen las reglas de *tipado*. En general, toda expresión E tendrá asociado un tipo t, el cual puede declararse escribiendo $E : t$. La constante 1 tiene tipo $\mathbb{Z}$ y la constante $true$ tiene tipo $\mathbb{B}$, podemos escribir entonces $1 : \mathbb{Z}$ y $true : \mathbb{B}$. Similarmente cada variable tiene un tipo. A veces el tipo de una variable se menciona en el texto que acompaña a la expresión que usa la variable, y otras veces está dado en alguna forma de declaración, como ocurre en la declaración de un lenguaje de programación **var** $x : integer$. Sin embargo cuando el tipo de una variable no es importante o queda claro del contexto puede omitirse. A veces es útil, declarar el tipo de una subexpresión dentro de una expresión dada, de modo de hacer lo más clara posible la expresión al lector. Por ejemplo, podemos escribir 1^n también así

$$(1 : \mathbb{Z})^{n:\mathbb{N}}$$

para indicar que 1 es un entero y además n es natural. Esta notación es necesaria cuando 1 puede simbolizar también una matriz identidad además de un número entero, y sea importante dejar en claro que n es no negativo. Cualquier subexpresión de una expresión debe escribirse con su correspondiente tipo. Por ejemplo aquí presentamos una expresión con su tipo completamente descripto:

$$((x : \mathbb{N} + y : \mathbb{N}) \cdot x : \mathbb{N}) : \mathbb{N}$$

Además de las constantes y variables, otras clases de expresiones que debemos tomar en cuenta en la definición de tipos es la aplicación de funciones. Cada función tiene asociado un tipo, el cual a su vez describe los tipos de sus parámetros y el tipo de su resultado. Si los parámetros $p_1, p_2, \ldots, p_n$ de una función f tienen los tipos $t_1, t_2, \ldots, t_n$ y el resultado de la función tiene tipo r, entonces f tiene tipo $t_1 \times \ldots \times t_n \rightarrow r$. Indicamos esto último así:

$$f : t_1 \times \ldots \times t_n \rightarrow r \tag{6.1}$$

En la siguiente tabla presentamos algunos ejemplos de funciones y sus tipos.

función	tipo	ejemplo de aplicación de función
mas	$\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$	$mas\,(1,3)$ o $1+3$
no	$\mathbb{B} \to \mathbb{B}$	$no.true$ o $\neg true$
$menor$	$\mathbb{Z} \times \mathbb{Z} \to \mathbb{B}$	$menor\,(5,3)$ o $5<3$

Para funciones f con tipos como los vistos en (6.1), definimos la aplicación de función $f\,(a_1,\ldots,a_n)$ como una expresión si y sólo si sus argumentos a_i son de tipo t_i. El tipo de resultado de una aplicación de función es entonces r. En este sentido, el tipo de una expresión se determina a través de los tipos de sus operandos. Los tipos correctos dependen solamente de la secuencia de símbolos en la expresión dada, y no de la evaluación de la expresión en un estado dado. Por ejemplo, $(1/\,(x:\mathbb{Z})):\mathbb{R}$ es una expresión, aún cuando su evaluación no está definida en el estado $x=0$.

Para cualquier tipo (o conjunto) t y cualquier expresión E, definimos la expresión $E \in t$ como el resultado de "E está en t". Por ejemplo, podemos escribir:

$$i \in \mathbb{N} \Rightarrow -i \leq 0.$$

Por lo tanto, $E \in t$ es una expresión, tanto como $x < y$, que es evaluada en la medida que la expresión E es evaluada, mientras que $E : t$ es simplemente la expresión E descripta junto con su tipo. Para observar la relación entre la descripción sintáctica $E : t$ y la expresión $E \in t$ veamos la siguiente conexión entre ellas:

> Si E tiene tipo t, es decir $E : t$, entonces $E \in t$ es evaluada en el valor *true* en todo los estados en los cuales E está bien definida.

Un lenguaje con reglas sintácticas que asigna un tipo a cada expresión se llama *fuertemente tipado*. Pascal, Ada y ML son lenguajes de programación fuertemente tipados. Esta característica en los lenguajes provee una medida de control sintáctico en dos sentidos. Primero, nos libera de ubicar expresiones $E \in t$ en diversos lugares dentro de expresiones. Si la sintaxis indica $E : t$, entonces necesariamente debe darse $E \in t$ (siempre que E esté definida). Por otra parte, cuando el lenguaje se implementa, el tipado fuerte permite la detección de errores en forma temprana por parte del editor de texto, el compilador o alguna otra herramienta de software.

Un lenguaje sin reglas sintácticas de tipo se llama *no tipado*. Lisp, Scheme y Prolog son lenguajes de esta clase. En un lenguaje no tipado, $-true$ es una expresión y el error en ésta es considerado un error semántico, que se detecta (si es el caso) sólo cuando la expresión es evaluada.

Con la noción de tipo, son necesarias ciertas restricciones para asegurar la corrección de tipos en la manipulación de expresiones:

a) En una sustitución $E[x := F]$, x y F deben ser del mismo tipo.

b) La igualdad $b = c$ se define sólo si b y c tienen el mismo tipo. Esto es, la igualdad $=$ tiene tipo $t \times t \to \mathbb{B}$ para cualquier tipo t.

La restricción (a) asegura que la sustitución no produce "no-expresiones". La restricción (b) asegura que la aplicación de la Regla de Leibniz o la Sustitución como Regla de inferencia no viole la restricción (a).

6.3. Sintaxis e interpretación de la cuantificación

Una *expresión cuantificada* o *cuantificación* tendrá la siguiente forma:

$$(\oplus\, x :\ R\ :\ T)$$

donde

- $\oplus$ designa un operador asociativo, conmutativo y con elemento neutro.
- x denota una secuencia variables, llamadas variables cuantificadas. Cuando exista más de una variable cuantificada, se separarán las variables entre comas. Denotaremos con $V.x$ al conjunto de variables cuantificadas que hay en x.
- R es un predicado que se denomina *rango de especificación*.
- T es una expresión llamada *término de cuantificación*.

La variable cuantificada x solo tiene sentido dentro de los símbolos (y). Cuando necesitemos hacer referencia explícita a la variable cuantificada escribiremos $R.x$ y $T.x$ para el rango y el término respectivamente.

Hasta aquí hemos presentado la sintaxis de una expresión cuantificada. Veremos ahora su semántica o interpretación:

"*La expresión* $(\oplus\, x :\ R\ :\ T)$ *denota la aplicación del operador* $\oplus$ *a los valores de* T *evaluada en los estados en que la variable* x *satisface el rango* R*, es decir los estados para los cuales* R *es* $true$*.*"

Aquí vemos algunos ejemplos de cuantificaciones junto con las expresiones que éstas denotan.

$$\begin{array}{lcl}
(+\, i : 0 \leq i < 4 : i{,}8) & = & 0{,}8 + 1{,}8 + 2{,}8 + 3{,}8 \\
(\times\, i : 0 \leq i < 3 : i + (i+1)) & = & (0+1) \times (1+2) \times (2+3) \\
(\wedge\, i : 0 \leq i < 2 : b[i] \neq 6) & = & b[0] \neq 6 \wedge b[1] \neq 6 \\
(\vee\, i : 0 \leq i < 21 : b[i] = 0) & = & b[0] = 0 \vee \ldots \vee b[20] = 0
\end{array}$$

La cuantificación que tiene como operador $\wedge$ es la cuantificación universal, mientras que la existencial también se denota con la cuantificación que tiene como operador $\vee$. También se utilizan símbolos en lugar de un operador en algunas cuantificaciones matemáticas, como la

sumatoria y la productoria. En lo que sigue utilizaremos indistintamente en las cuantificaciones los símbolos $\forall$, $\exists$, $\sum$ y $\prod$ y los correspondientes operadores $\wedge$, $\vee$, $+$ y $\times$.

En resumen, aceptaremos:

$(+\, i : R : P)$ como $(\sum\, i : R : P)$
$(\times\, i : R : P)$ como $(\prod\, i : R : P)$
$(\wedge\, i : R : P)$ como $(\forall\, i : R : P)$
$(\vee\, i : R : P)$ como $(\exists\, i : R : P)$

6.4. Variables libres y ligadas

Uno de los conceptos básicos para comprender y manejar las expresiones cuantificadas es el de variable *libre*. Asociados a este concepto están también los conceptos de variable *ligada* y *alcance*. Vamos a considerar primero algunos ejemplos antes de dar una definición formal de estos conceptos.

Consideremos otra vez el ejemplo de la sumatoria $(\sum\, i : 0 \leq i < n : 2 \times i + 1)$. Es claro que el valor de esta expresión depende de n, para diferentes estados de n va a tomar diferentes valores. Sin embargo su valor no depende del valor de i en un estado dado, y podemos cambiar esta variable, por cualquier otra que no aparezca en la expresión, manteniendo el valor de la misma, por ejemplo, podemos escribir la misma expresión como $(\sum\, j : 0 \leq j < n : 2 \times j + 1)$. Diremos que las ocurrencias de la variable i están *ligadas* (a la variable que aparece luego del operador $+$), mientras que la ocurrencia de n está *libre*.

En general, la ocurrencia de una variable es ligada en una expresión E si se encuentra bajo el alcance de un cuantificador, o si es la variable de un cuantificador. Si la ocurrencia de una variable no es ligada en E diremos que es libre.

Una variable x es ligada en una expresión E si tiene alguna ocurrencia ligada en E y es libre en E si tiene alguna ocurrencia libre.

A continuación veremos la definición formal de variable libre y ligada.

(6.1) Definiciï¿½n. Variable Libre

- Una variable i está libre en la expresión i.
- Si la variable i está libre en E entonces estará libre en (E).
- Si la variable i está libre en E y f es una función que toma a E como parámetro, entonces i está libre en $f\,(\ldots, E, \ldots)$.
- Si la variable i está libre en R, donde R es un rango de especificación, e i no aparece en la secuencia de variables de x ($i \notin V.x$), entonces i también está libre en $(\oplus\, x : R : T)$ para cualquier término T.
- Si la variable i está libre en T, donde T es un término de cuantificación, e i no aparece en la secuencia de variables de x ($i \notin V.x$), entonces i también está libre en $(\oplus\, x : R : T)$ para cualquier rango de especificación R.

(6.2) **Notaciï¿½n.** Dada una expresión E, el conjunto de todas las variables libres de E se denotará $FV.E$.

(6.3) **Definiciï¿½n.** Variable Ligada

Sea i una variable libre en la expresión E y i una variable de la secuencia de variables x, luego la variable i está ligada en las expresiones $(\oplus\, x : E : F)$ y en $(\oplus\, x : F : E)$.

Además, si i está ligada en E, también lo está (a la misma variable de cuantificación) en (E), $(\oplus\, x : R : T)$, $(\oplus\, y : E : T)$ y $(\oplus\, y : R : E)$.

(6.4) **Notaciï¿½n.** Dada una expresión E, el conjunto de todas las variables ligadas de E se denotará $BV.E$.

(6.5) **Ejemplo.** Consideremos la expresión

$$E = k + (\sum\, i : 0 \leq i < n : 2 \times i + 1)$$

luego, las variables libres y ligadas en E son dadas en los conjuntos $FV.E = \{k, n\}$ y $BV.E = \{i\}$.

(6.6) **Ejemplo.** Dada la expresión

$$E = (i > n) \vee (\forall\, i, j : j < i < n : i^2 + j^2 < n)$$

se definen los siguientes conjuntos de variables $FV.E = \{i, n\}$ y $BV.E = \{i, j\}$. Notamos que la variable i aparece libre y ligada en E, dado que la primer ocurrencia es libre y las demás ocurrencias son ligadas. Una variable puede ser libre y ligada en una misma fórmula pero una ocurrencia de variable en una fórmula es o bien libre o bien ligada, no ambas.

6.5. Revisión de la sustitución en expresiones cuantificadas

Las variables ligadas tienen su alcance delimitado de manera explícita y están ligadas a una variable de cuantificación. Si se cambian ambas por un nuevo nombre (que no aparece dentro del alcance), el significado de la expresión cuantificada no cambiará, como ya lo habíamos mencionado:

$$\Big(\sum i : 0 \leq i < n : 2 \cdot i + 1\Big) = \Big(\sum j : 0 \leq j < n : 2 \cdot j + 1\Big)$$

Debe tenerse cuidado sin embargo con las duplicaciones de nombres, dado que si por ejemplo se reemplaza i por n se obtiene una expresión diferente a las anteriores:

$$\Big(\sum n : 0 \leq n < n : 2 \cdot n + 1\Big)$$

Veremos más adelante que esta expresión es igual a 0, independientemente del estado de la variable n.

A continuación extenderemos la noción de sustitución en expresiones cuantificadas a través del siguiente axioma:

(6.7) **Axioma.** Sustitución para expresiones cuantificadas

$$V.y \cap (V.x \cup FV.E) = \emptyset \Rightarrow (\oplus\, y : R : T)\,[x := E] = (\oplus\, y : R\,[x := E] : T\,[x := E])$$

(6.8) **Observacii¿$\frac{1}{2}$n.** La condición $V.y \cap (V.x \cup FV.E) = \emptyset$ es necesaria para evitar duplicaciones de nombres, si la condición no se cumple es necesario renombrar la variable de cuantificación. En sección 6.7 veremos como hacer ésto mediante un axioma.

6.6. La regla de Leibniz para expresiones cuantificadas

La regla de Leibniz nos permite reemplazar una expresión por otra expresión igual a ésta. En expresiones cuantificadas, ésta regla, que fue introducida en el capítulo 3,

$$\frac{X = Y}{E\,[x := X] = E\,[x := Y]}$$

no siempre nos permite realizar tales reemplados.

Por ejemplo, dado que $2 * i + 1 = 2 * (i + 1) - 1$ es de esperar que aplicando la regla de Leibniz pueda deducirse que

$$\Big(\sum i : 0 \le i < n : 2 \cdot i + 1\Big) = \Big(\sum i : 0 \le i < n : 2 \cdot (i + 1) - 1\Big)$$

La forma en que podría usarse esta regla para obtener la expresión anterior es la siguiente:

$$\frac{2 \cdot i + 1 = 2 \cdot (i + 1) - 1}{(\sum i : 0 \le i < n : y)\,[y := 2 \cdot i + 1] = (\sum i : 0 \le i < n : y)\,[y := 2 \cdot (i + 1) - 1]}$$

Pero dado que i es la variable de cuantificación, para aplicar el Axioma 6.7, es necesario renombrar i antes de aplicar la sustitución, con lo cual el resultado de

$$\Big(\sum i : 0 \le i < n : y\Big)\,[y := 2 \cdot i + 1]$$

será

$$\Big(\sum j : 0 \le j < n : 2 \cdot i + 1\Big)$$

y no el esperado.

La regla de Leibniz puede generalizarse para ser utilizada en expresiones cuantificadas agregando las siguientes reglas para el rango y el término:

Regla de Leibniz para expresiones cuantificadas:

$$\frac{X = Y}{(\oplus\, i : R\,[z := X] : T) = (\oplus\, i : R\,[z := Y] : T)}$$

$$\frac{X = Y}{(\oplus\, i : R : T\,[z := X]) = (\oplus\, i : R : T\,[z := Y])}$$

6.7. Reglas generales para expresiones cuantificadas

A continuación enunciaremos axiomas y teoremas que nos permitirán manipular expresiones cuantificadas generales (sobre un operador binario $\oplus$). En los casos en que el axioma o teorema sea una generalización de alguno del capítulo 5, se conservará el nombre de éste, ya que quedará claro a partir del contexto qué versión del axioma o teorema está siendo utilizada.

En lo que sigue asumiremos que $\oplus$ es un operador conmutativo, asociativo y con elemento neutro u.

(6.9) **Axioma.** Rango vacío. Cuando el rango de especificación es vacío (es $false$), la expresión cuantificada es igual al elemento neutro u del operador $\oplus$.

$$(\oplus\, i : false : T) = u$$

(6.10) **Axioma.** Rango unitario. Si el rango de especificación consiste en un solo elemento, e i no es una variable libre en la expresión N, la expresión cuantificada es igual al término evaluado en ese elemento.

$$(\oplus\, i : i = N : T) = T\,[i := N]$$

otra forma de expresar esta regla es hacer explícita la dependencia de T a la variable i (lo cual no significa que i deba aparecer en T)

$$(\oplus\, i : i = N : T.i) = T.N$$

(6.11) **Observaciï¿½n.** Observemos que para poder aplicar el axioma, la variable i no debe estar libre en la expresión N. Esta restricción es necesaria por la siguiente razón: la expresión del lado izquierdo del axioma no depende de i, debido a que todas las variables i son ligadas en la expresión; para que la igualdad se cumpla, la expresión del lado derecho tampoco debe depender de i, ésto requiere que i no esté libre en N.

(6.12) **Axioma.** Distributividad. Si el operador $\otimes$ es distributivo a izquierda con respecto a $\oplus$, $i \notin FV.E$ y se cumple al menos una de las siguientes condiciones:

- el rango de especificación es no vacío,

- el elemento neutro del operador $\oplus$ existe y es absorbente para $\otimes$,

entonces:

$$(\oplus i : R : E \otimes T) = E \otimes (\oplus i : R : T)$$

análogamente si $\otimes$ es distributivo a derecha con respecto a $\oplus$ y se cumple al menos una de las condiciones anteriores, entonces:

$$(\oplus i : R : T \otimes E) = (\oplus i : R : T) \otimes E$$

Veamos por qué en caso que la primera condición no se cumpla es necesario que se cumpla la segunda para que el axioma sea válido. Supongamos que $R = false$, aplicando el axioma Rango vacío en ambos lados de la igualdad obtenemos:

$$u = x \otimes u$$

Deducimos entonces, que para que se cumpla esta igualdad es necesario que el nuetro operador del operador $\oplus$ (u) sea absorbente para $\otimes$.

Veamos un ejemplo,

$$\begin{array}{ll} & (+\, i : 0 \leq i < 10 : (x+4) \times 2) \\ = & \quad \langle \times \text{ es asociativo respecto a } + \text{ , } 0 \text{ es absorvente para } \times \text{ , Distributividad (6.12)} \rangle \\ & (x+4) \times (+\, i : 0 \leq i < 10 : 2) \end{array}$$

Antes de enunciar el siguiente axioma, recordemos que un operador $\oplus$ es idempotente si cumple la siguiente condición:

$$P \oplus P = P$$

donde P es una variable del tipo sobre el cual está definido el operador. Ejemplos de operadores idempotentes son: $\vee$ y $\wedge$.

(6.13) **Axioma.** Partición de rango. Cuando el rango de especificación es de la forma $R \vee S$ y además se cumple al menos una de las siguientes condiciones:

- el operador $\oplus$ es idempotente
- $R \wedge S = false$, es decir no existe un elemento que satisfaga R y S.

la expresión cuantificada puede reescribirse como sigue

$$(\oplus i : R \vee S : T) = (\oplus i : R : T) \oplus (\oplus i : S : T)$$

(6.14) **Observacïï¿½n.** La restricción $R \wedge S = false$ en el axioma anterior asegura que los términos no son acumulados dos veces en el lado derecho cuando el operador no es idempotente, el axioma que sigue contempla la situación de la doble acumulación agregando en el lugar indicado la expresión $(\oplus i : R \wedge S : P)$, que involucra los términos de P para los valores de las variables cuantificadas que satisfacen tanto R como S.

(6.15) **Axioma.** Partición de Rango.

$$(\oplus i : R \vee S : P) \oplus (\oplus i : R \wedge S : P) = (\oplus i : R : P) \oplus (\oplus i : S : P)\,.$$

(6.16) **Axioma.** Partición de Rango generalizada.

Si el operador operador $\oplus$ es idempotente y el rango de especificación es una cuantificación existencial, entonces

$$(\oplus i : (\exists j : S.i.j : R.i.j) : T.i) = (\oplus i, j : S.i.j \wedge R.i.j : T.i)$$

(6.17) **Axioma.** Regla del término. Cuando el operador $\oplus$ aparece en el término de la cuantificación, la expresión cuantificada puede reescribirse de la siguiente manera:

$$(\oplus i : R : T \oplus G) = (\oplus i : R : T) \oplus (\oplus i : R : G)$$

(6.18) **Axioma.** Regla del término constante. Si el término de la cuantificación es igual a una constante C (es decir, la variable cuantificada i no aparece en C), el operador $\oplus$ es idempotente y el rango de especificación es no vacío, entonces

$$(\oplus i : R : C) = C$$

A partir de este teorema se pueden demostrar las siguientes propiedades para la cuantificación universal y existencial.

$$(\vee i :: true) = true$$
$$(\wedge i :: false) = false$$

(6.19) **Axioma.** Regla de anidado. Cuando hay más de una variable cuantificada y el rango de especificación es una conjunción de predicados, uno de los cuales es independiente de alguna de las variables de cuantificación, la expresión cuantificada puede reescribirse de la siguiente manera:

$$(\oplus i, j : R.i \wedge S.i.j : T.i.j) = (\oplus i : R.i : (\oplus j : S.i.j : T.i.j))$$

(6.20) **Axioma.** Regla de intercambio de variables. Si $V.j \cap FV.R = \emptyset$ y $V.i \cap FV.Q = \emptyset$, entonces

$$(\oplus i : R : (\oplus j : Q : T)) = (\oplus j : Q : (\oplus i : R : T))$$

(6.21) **Axioma.** Regla de cambio de variable. Si $V.j \cap (FV.R \cup FV.T) = \emptyset$ pueden renombrarse las variables

$$(\oplus i : R : T) = (\oplus j : R\,[i := j] : T\,[i := j])$$

puede también escribirse usando una referencia explícita a i

$$(\oplus i : R.i : T.i) = (\oplus j : R.j : T.j)$$

(6.22) Observacïï¿½n. La restricción del axioma nos asegura que j es una variable nueva, o dicho de otra forma que j no aparece libre en R y T.

A continuación veremos una generalización de este último axioma. Para comprender la utilidad del mismo introduciremos un ejemplo.

Consideremos la siguiente expresión

$$\left(+i : 2 \leq i \leq 10 : i^2\right)$$

Si reescribimos esta expresión de modo que el rango de variación de i comience en 0 en vez de 2 conseguimos la expresión

$$\left(+k : 0 \leq k \leq 8 : (k+2)^2\right)$$

Observemos que la relación entre i y k es $i = k + 2$ o también $k = i - 2$. La igualdad entre las expresiones anteriores es una instancia particular del siguiente teorema. Recordemos que una función biyectiva f admite inversa f^{-1} y la función inversa se define de acuerdo a:

$$x = f.y \equiv y = f^{-1}.x$$

(6.23) Teorema. Regla del cambio de variable.

Sea $(\oplus i : R : T)$ una expresión cuantificada, y sea f una función biyectiva definida sobre R y j una variable que no aparece libre en R ni en T (es decir $V.j \cap (FV.R \cup FV.T) = \emptyset$) entonces,

$$(\oplus i : R.i : T.i) = (\oplus j : R.(f.j) : T.(f.j))$$

Probaremos este teorema comenzando por la expresión más compleja.

$$\begin{array}{ll} & (\oplus j : R.(f.j) : T.(f.j)) \\ = & \quad \langle \text{Rango Unitario (Introducción de la cuantificación sobre } i) \rangle \\ & (\oplus j : R.(f.j) : (\oplus i : i = f.j : T.i)) \\ = & \quad \langle \text{Anidado 6.19} \rangle \\ & (\oplus i, j : R.(f.j) \wedge i = f.j : T.i) \\ = & \quad \langle \text{Sustitución 3.84a)} \rangle \\ & (\oplus i, j : R.i \wedge i = f.j : T.i) \\ = & \quad \langle \text{Anidado 6.19} \rangle \\ & (\oplus i : R.i : (\oplus j : i = f.j : T.i)) \\ = & \quad \langle \text{Definición de inversa} \rangle \\ & (\oplus i : R.i : (\oplus j : j = f^{-1}.i : T.i)) \\ = & \quad \langle \text{Rango Unitario 6.10, } j \notin FV.T \rangle \\ & (\oplus i : R.i : T.i) \end{array}$$

Veremos ahora dos teorema que nos permitirán sacar el primer y último término de una cuantificación fuera de la misma.

(6.24) Teorema. Separación de un término.

Sea $n < m$,

a) $(\oplus\, i : n \leq i < m : T.i) = T.n \oplus (\oplus\, i : n < i < m : T.i)$

b) $(\oplus\, i : n < i \leq m : T.i) = (\oplus\, i : n < i < m : T.i) \oplus T.m$

Demostraremos 6.24a comenzando por el lado izquierdo de la igualdad:

$$
\begin{array}{ll}
 & (\oplus\, i : n \leq i < m : T.i) \\
= & \quad \langle n \leq i < m \equiv i = n \vee n < i < m \rangle \\
 & (\oplus\, i : i = n \vee n < i < m : T.i) \\
= & \quad \langle \text{Partición de rango 6.13, } i = n \wedge n < i < m = false \rangle \\
 & (\oplus\, i : i = n : T.i) \oplus (\oplus\, i : n < i < m : T.i) \\
= & \quad \langle \text{Rango unitario 6.10} \rangle \\
 & T.n \oplus (\oplus\, i : n \leq i < m : T.i)
\end{array}
$$

6.8. Cuantificadores aritméticos

6.8.1. Máximos y mínimos

Los operadores mateméticos max y min, que calculan el máximo y el mínimo valor entre dos números, también pueden cuantificarse.

Una definición formal de estos operadores es la siguiente:

(6.25) Axioma. Definición de max y min:

$$
\begin{aligned}
max.(a, b) = c &\;\equiv\; (c = a \vee c = b) \wedge a \leq c \wedge b \leq c \\
min.(a, b) = c &\;\equiv\; (c = a \vee c = b) \wedge a \geq c \wedge b \geq c
\end{aligned}
$$

La definición anterior no es constructiva en el sentido que no muestra como calcular tanto el máximo como el mínimo entre dos elementos, pero sí provee una herramienta que permite manipular expresiones donde aparecen los operadores max y min.

Los operadores max y min son simétricos y asociativos sobre el conjunto de los números enteros, pero no poseen elemento neutro. Extenderemos el conjunto de los números enteros agregando dos símbolos: ∞ y $-\infty$, los cuales por definición serán los neutros de min y max respectivamente. De esta manera podremos definir las cuantificaciones asociadas a estos operadores:

$$(min\, i : R : E) \quad \text{y} \quad (max\, i : R : E) \tag{6.2}$$

Cuyos significados son el mínimo y el máximo de los valores obtenidos al evaluar la expresión E sobre el conjunto de valores de i que satisfacen el rango R. Por ejemplo:

$$(min\, i : 0 \leq i \leq 10 : b\,[i])$$

es el mínimo de los valores de la lista $b\,[0]$,..., $b\,[10]$.

Las fórmulas (6.2) satisfacen las leyes generales dadas para los cuantificadores en el capítulo 6. A continuación instanciaremos algunas de estas leyes para el operador max.

(6.26) Teorema. Rango vacío

$$(max\ i : false : T.i) = -\infty$$

Dado que el operador min es distributivo respecto max, y que $-\infty$ es absorvente para min, el teorema de Distributividad 6.12 para los operadores max y min puede expresarse como:

(6.27) Teorema. Distributividad de min sobre max.

Suponiendo que i no es variable libre en E:

$$min.(E, (max\ i : R : T)) = (max\ i : R : min.(E, T))$$

La suma también se distribuyen sobre max. Al instanciar el teorema de Distributividad con el operador $+$ se obtiene el siguiente teorema.

(6.28) Teorema. Distributividad de $+$ sobre max.

Suponiendo que i no es variable libre en E y que $R \neq false$:

$$E + (max\ i : R : T) = (max\ i : R : E + T)$$

La restricción $R \neq false$ es necesaria dado que $-\infty$ no es absorvente para la suma (además de que la suma no está definida para las constantes ∞ y $-\infty$).

El hecho de que max sea idempotente hace que el teorema Partición de rango (6.15) para este operador pueda expresarse como:

(6.29) Teorema. Partición de rango.

$$(max\ i : R \vee S : F) = max.((max\ i : R : F), (max\ i : S : F))$$

Se deja como ejercicio para el lector instanciar las restantes leyes para max y las correspondientes al operador min.

A continuación veremos dos propiedades que relacionan las cuantificaciones para los operadores max y min con la cuantificación universal.

(6.30) Teorema.

$$T.x = (max\ i : R.i : T.i) \equiv R.x \wedge (\forall\ i : R.i : T.i \leq T.x)$$
$$T.x = (min\ i : R.i : T.i) \equiv R.x \wedge (\forall\ i : R.i : T.i \geq T.x)$$

6.8.2. Operador de conteo

Hasta aquí hemos definido cuantificadores que provienen de operadores binarios, conmutativos, asociativos y con elementos neutros. También es posible definir cuantificadores a partir de otros. Éste es el caso del cuantificador N, definido como:

$$(N\, i : R : T) = (\sum\, i : R \wedge T : 1) \tag{6.3}$$

Esta cuantificador cuenta la cantidad de elementos en el rango de especificación R que satisfacen el término de cuantificación T.

Los siguientes son algunos ejemplos:

$$(N\, i : 0 \leq i \leq n : par.i)$$
$$(N\, i : 0 \leq i \leq n : (\exists\, j :: i = 2^j))$$

El primero cuenta la cantidad de elementos pares que hay en el interbalo $[0, \ldots, n]$ y el segundo la cantidad de potencias de 2 que hay en el mismo interbalo.

6.9. Ejercicios

6.1 Dadas las siguientes funciones con sus correspondientes tipos:

$$\begin{array}{lcl} a & : & A \to B \\ b & : & B \to C \\ c & : & C \to A \\ d & : & A \times C \to D \\ e & : & B \times B \to E \end{array}$$

Decidir cuáles de las siguientes expresiones es correcta, justificando la respuesta. Suponer que $u : A$, $w : B$, $x : C$, $y : D$ y $z : E$.

(a) $e\,(a.u, w)$

(b) $b.x$

(c) $e\,(a\,(c.x)\,, a.u)$

(d) $a\,(c\,(b\,(a.y)))$

(e) $d\,(c.x, c.x)$

6.2 Verificar si es posible dar una expresión cuantificada sobre los siguientes operadores: $\equiv$, $\Rightarrow$, $\neg$ y $/$. En cada caso, ya sea afirmativo o negativo, justificar su respuesta.

6.3 Dar la expresión cuantificada asociada a las siguientes expresiones:

a) $0, 3^1 + 0, 3^2 + \ldots + 0, 3^{100}$

b) $1 \times 2 \times 3 \times \ldots \times 25$

c) $b[0] \times b[1] \times \ldots \times b[n]$

d) $b[0] > 0 \wedge b[1] > 0 \wedge \ldots \wedge b[n] > 0$

e) $b[0] > 1 \wedge b[1] > 3 \wedge \ldots \wedge b[n] > 2 \times n + 1$

f) $((p + 0) = 0) \vee ((p + 1) = 1) \vee ((p + 2) = 2) \vee \ldots \vee ((p + k) = k)$

g) $(3 \times b[0])^1 + (4 \times b[0])^{1/2} + (5 \times b[0])^{1/3} + \ldots + (27 \times b[0])^{1/25}$

6.4 ¿Cuál es el valor de $(\Sigma\, k \ : k^2 = 4 : k^2)$ en el caso que el tipo de la variable k sea:

a) un número natural,

b) un número entero?

6.5 Determinar todas las ocurrencias de variables libres y ligadas en las siguientes expresiones haciendo explícito los conjuntos FV y BV respectivamente en cada una de ellas.

a) $E_1 : 4 \times i$

b) $E_2 : (\Sigma\, j \ : 1 \leq j \leq 3 : 4 \times i)$

c) $E_3 : (\Sigma\, j \ : 1 \leq j \leq 3 : 4 \times j)$

d) $E_4 : (\Sigma\, j \ : 1 \leq j \leq 3 : m \times j) + (\Sigma\, j \ : 1 \leq j \leq 3 : n \times j)$

e) $E_5 : (\Sigma\, j \ : 1 \leq j \leq 3 : m \times j) + (\Sigma\, k \ : 1 \leq k \leq 3 : n \times j)$

f) $E_6 : x + n + (+x : 0 \leq x < n : x + i)$

g) $E_7 : (\vee i : 0 \leq i \leq j : (\wedge j : 1 \leq j < m : j \cdot i \leq k))$

h) $E_8 : true \wedge \left(\vee k : 0 \leq k \leq a : k^{\,2} = z\right)$

6.6 Realizar las siguientes sustituciones, en caso de ser necesario aplicar la Regla 6.21 correspondiente a cambio de variable dummy.

(a) $(\oplus x : 0 \leq x + r < n : x + v)\,[v := 3]$

(b) $(\oplus x : 0 \leq x + r < n : x + v)\,[x := 3]$

(c) $(\oplus x : 0 \leq x + r < n : x + v)\,[n := n + x]$

(d) $(\oplus x : 0 \leq x : (\oplus y : 0 \leq y : x + y + n))\,[n := x + y]$

(e) $(\oplus x : 0 \leq x < r : (\oplus y : 0 \leq y : x + y + n))\,[r := y]$

6.7 Probar los siguientes teoremas, suponiendo que $n > 0$,

(a) $(+i : 0 \leq i < n + 1 : b[i]) = b[0] + (+i : 1 \leq i < n + 1 : b[i])$

(b) $(+i : 0 \leq i \leq n : b[i]) = (+i : 0 \leq i < n : b[i]) + b[n]$

(c) $(+i : 0 \leq i \leq n : b[i]) = b[0] + (+i : 0 < i < n : b[i]) + b[n]$

6.8 Probar los siguientes teoremas, suponiendo que $n \geq 0$,

(a) $(\vee i : 0 \leq i < n+1 : b[i] = 0) \equiv (\vee i : 0 \leq i < n : b[i] = 0) \vee b[n] = 0$

(b) $(\wedge i : 0 \leq i < n+1 : b[i] = 0) \equiv (\wedge i : 0 \leq i < n : b[i] = 0) \wedge b[n] = 0$

(c) $(\vee i : 0 \leq i < n+1 : b[i] = 0) \equiv b[0] = 0 \vee (\vee i : 0 < i < n+1 : b[i] = 0)$

(d) $(\wedge i : 0 \leq i < n+1 : b[i] = 0) \equiv b[0] = 0 \wedge (\wedge i : 0 \leq i < n+1 : b[i] = 0)$

6.9 Probar los siguientes teoremas:

a) $(+i : 0 \leq i \leq n : i) = (+i : 0 \leq i \leq n \wedge par.i : i) + (+i : 0 \leq i \leq n \wedge impar.i : i)$

b) $(+i : 0 \leq i \leq 10 : 0) = 0$

c) $(\wedge m : m = false : m \vee \neg m) \wedge (\vee n : n = true : n \equiv n) \Rightarrow (\vee k : k = 5 : 2 \cdot k < k^2)$

d) $(\wedge i : 0 < i \leq n : t[i] > 4) \equiv t[n] > 4 \ \wedge \ (\wedge j : 0 < j \leq n-1 : t[j] \geq 5)$, donde $n \geq 1$

e) $(\cdot i : i > 2 \wedge i < 1 : i \cdot 2 + 3) > (+j : j > j + 1 : j^3)$

f) $(+i : 0 \leq i \leq n : i^2 \cdot 2) =$
$(+j : par.j \wedge 0 \leq j \leq n : 2 \cdot j^2) + 2 \cdot \ (+k : impar.k \wedge 0 \leq k \leq n : k^2)$

g) $(\cdot i, j : i \geq 0 \wedge i + j < 4 : (i + j) \cdot (i - j)) = (\cdot i : i \geq 0 : (\cdot j : j < 4 - i : i + j)) \cdot$
$(\cdot i : i \geq 0 : (\cdot j : i \leq 3 - j : i - j))$

6.10 Demostrar que para cualquier elemento z que satisfaga $R.i$, $T.z$ es menor al máximo de los valores obtenidos al evaluar $T.i$ sobre el conjunto de valores que satisfacen $R.i$.

Nota: Esta propiedad puede probarse demostrando:

$$F.x = (max\ i : R.i : F.i) \Rightarrow (R.z \Rightarrow F.z \leq F.x)$$

6.11 Suponiendo que:

i) $x = (max\ i, j : R.i.j \wedge j < N + 1 : f.i.j)$

ii) $y = (max\ i : R.i.(N+1) : f.i.(N+1))$

calcular una expresión libre de cuantificadores que sea equivalente a

$$(max\ i, j : r.i.j \wedge j \leq N + 1 : f.i.j)$$

6.12 Enunciar y demostrar la regla partición de rango para el cuantificador N.

6.13 En lo que sigue $b\,[\,0\,,\,\ldots\,,\,n-1\,]$ es una lista de $n \geq 0$ números enteros. Traducir las siguientes frases en expresiones cuantificadas.

a) El mínimo valor de la lista b es positivo.

b) El máximo valor de la lista b no es cero.

c) Si $b\,[0] \neq 0$ entonces $b\,[0]$ es el máximo valor de la lista.

d) La lista b contiene tantos números positivos como negativos.

e) La lista b contiene exactamente dos ceros.

f) El número de valores pares en la lista b es mayor que el número de valores impares.

g) El promedio de los valores de b no supera al número de elementos no nulos en la lista.

6.14 En lo que sigue $b\,[\,0\,,\,\ldots\,,\,n-1]$ es una lista de $n \geq 0$ números enteros. Traducir las siguientes expresiones cuantificadas en expresiones dadas en lenguaje corriente.

a) $(max\; i : 1 \leq i \leq n : i+1) = n+1$

b) $(N\, i : 0 \leq i \leq 3 : i^2 > i) = 2$

c) $(min\; i : 0 \leq i \leq 3 : i^2) = 2$

d) $(max\; i : 0 \leq i < n-1 : b\,[i]) = b\,[n-1]$

e) $(N\, i : 0 \leq i \leq n-1 : par.\, b\,[i]) = (N\, i : 0 \leq i \leq n-1 : impar.\, b\,[i])$

CAPÍTULO 7
El formalismo básico

Índice del Capítulo

7.1. Definiciones y expresiones 126
7.1.1. Reglas para el cálculo con definiciones 127
7.2. Análisis por casos 128
7.3. Pattern Matching 130
7.4. Tipos 131
7.4.1. Tipos básicos 131
7.4.2. Tuplas 132
7.4.3. Listas 132
Constructores de listas 133
Operaciones entre listas 133
Propiedades de los operadores 134
Definiciones de operadores 135
7.5. Currificación 135
7.6. Ejercicios 136

En este capítulo definiremos una notación simple y abstracta para escribir y manipular programas. Esta notacińn, que llamaremos *formalismo básico*, se basa en la programación funcional, siendo más simple que la mayoría de los lenguaje de programación existentes y lo suficientemente rica para expresar programas funcionales.

$$f(x) = x^3$$

Para economizar paréntesis (y por otras razones que no detallamos) usaremos un punto para denotar la aplicación de una función a un argumento; así la aplicación de la función f al

argumento x se escribirá como $f.x$. Por otro lado, para evitar confundir la igualdad ($=$) con la definición de funciones, usaremos un símbolo ligeramente diferente: $\doteq$. Con estos cambios notacionales la función que eleva al cubo se escribirá como:

$$f.x \doteq x^3$$

En lo que sigue, tendremos cuidado de no confundir a la función f con su aplicación a un argumento $f.x$, pues ésto último representa un valor, mientras que una función es un concepto totalmente diferente.

La regla básica para aplicación de funciones, es la regla de "sustitución de iguales por iguales", que ya hemos definido en el capítulo 1 como *regla de Leibniz*. Sean $f : A \to B$ una función y x e y valores de tipo A:

$$Leibniz : \frac{x = y}{f.x = f.y}$$

Recordemos que, mediante esta regla, la igualdad entre las expresiones x e y, asegura la igualdad de la aplicación de función entre $f.x$ y $f.y$.

De acuerdo a lo visto en el Capítulo 6, en expresiones cuantificadas con el operador universal, podemos reescribir la regla de Leibniz así:

$$(\forall\, f, x, y :: x = y \Rightarrow f.x = f.y)$$

7.1. Definiciones y expresiones

Uno de los elementos que forman el formalismo básico presentado aquí, es el conjunto de *expresiones*. Al igual que en las matemáticas, las expresiones son usadas sólo para denotar valores. Estos valores pertenecerán a conjuntos que definiremos más adelante como Tipos.

Es importante distinguir entre las expresiones y el valor que éstas denotan. Por ejemplo, veamos cuál es el valor que denota la expresión $cuadrado.(3 + 4)$, donde la función $cuadrado$ se define como: $cuadrado.x \doteq x \times x$. Para ello evaluaremos la expresión aplicando definiciones de funciones.

$$\begin{array}{ll} & cuadrado.(3 + 4) \\ = & \quad \langle \text{definición de } + \rangle \\ = & \quad \langle \text{definición de cuadrado} \rangle \\ & 7 \times 7 \\ = & \quad \langle \text{definición de } \times \rangle \\ & 49 \end{array}$$

Aquí vemos claramente que la expresión $cuadrado.(3 + 4)$ denota el número abstracto cuarenta y nueve, al igual que la expresión 49. La igualdad a la que hemos llegado $cuadrado.(3+4) = 49$, significa que ambas expresiones denotan el mismo valor, pero no hay que confundir el número denotado por la expresión decimal 49 con la expresión misma.

La expresión 49 no puede reducirse más, hemos llegado a la forma canónica de una expresión. Llamaremos *reducción* o *simplificación* al proceso de evaluación de una expresión y diremos que una expresión está en *forma canónica* o en *forma normal* si no puede reducirse más.

En nuestro formalismo básico la única forma de referirse a valores será a través de expresiones. El proceso de cómputo consistirá, en general, en reducir una expresión dada a su forma canónica cuando ésta exista.

Otro elemento que formará parte del formalismo básico será el conjunto de *definiciones*, con las cuales podremos introducir nuevos valores al formalismo y definir operaciones para manejarlos.

Una definición es una asociación de una expresión a un nombre. La forma de una definición será la siguiente:

$$f.x \doteq E$$

donde las variables en x pueden ocurrir en E. Si la secuencia de variables x es vacía, diremos que f es una *constante*. Si f ocurre en E, se dice que f es recursiva.

Dado un conjunto de definiciones, es posible utilizar éstas, para formar expresiones. Por ejemplo:

Son definiciones: $pi \doteq 3{,}1416$
$cuadrado.x \doteq x \times x$

Son expresiones: pi
$pi \times cuadrado.(3 \times 5)$

Un *programa funcional* es un conjunto de definiciones. La ejecución de un programa funcional consistirá en la evaluación de una expresión y reducción a su forma canónica.

7.1.1. Reglas para el cálculo con definiciones

En un contexto de definiciones, las reglas básicas para manejar expresiones son las reglas de *plegado* y *desplegado* (más conocidas por sus nombres en inglés: folding y unfolding). Cada una de éstas es la inversa de la otra.

Si se tiene la definición $f.x \doteq E$, entonces para una expresión A tenemos que

$$f.A = E[x := A]$$

Es decir, la expresión $f.A$ corresponde a la expresión E en la que toda aparición de x fue sustituída por A. Si x es una lista de variables, A deberá serlo también y ambas deberán tener la misma longitud. En tal caso, la sustitución se realizará en forma simultánea. Por ejemplo:

$$(x + y)[x, y := cuadrado.y, 3] = cuadrado.y + 3$$

La regla de desplegado consiste en reemplazar $f.A$ por $E[x := A]$, mientras que la de plegado consiste en reemplazar $E[x := A]$ por $f.A$. En los cálculos que realizaremos, frecuentemente diremos "definición de f" entendiéndose si es plegado o desplegado a partir del contexto.

La regla que nos permite demostrar que dos definiciones de funciones son iguales es la siguiente:

$$(\forall\, f, g :: (\forall\, x :: f.x = g.x) \Rightarrow f = g)$$

Es decir, que las definiciones son iguales, si devuelven resultados iguales para argumentos iguales. Esta regla se conoce con el nombre *Principio de extensionalidad*. Podemos ver que lo importante acerca de una función es la correspondencia entre argumentos y resultados y no la descripción de tal correspondencia, por ejemplo, la función que duplica su argumento puede expresarse de dos formas distintas: $f.x = 2 \times x$ y $g.x = x + x$. Ambas definiciones definen la misma función pero utilizan formas diferentes para hacerlo. La extensionalidad, nos permite demostrar que $f = g$ demostrando simplemente que $f.x = g.x$ para todo x.

Por último, introduciremos mediante un ejemplo el concepto de *definiciones locales*, las cuales pueden ser inluídas en cualquier definición. Supongamos la definición de la función $f.(x, y) = (a+1)(a+2)$ donde $a = (x+y)/2$. Ésta puede ser expresada de la siguiente forma:

$$\begin{array}{lcl} f.(x,y) & \doteq & (a+1) \times (a+2) \\ & & [\![\, a \doteq (x+y)/2 \,]\!] \end{array}$$

Esta definición es más simple y clara que la que podríamos dar sin utilizar una variable local. Que la variable a esté localmente definida en la definición de f, significa que a sólo tiene sentido dentro de la definición de f. En particular, en este caso, la definición de a hace referencia a nombres de variables como x e y que sólo tienen sentido dentro de la definición de f.

Si necesitamos definir más de una variable local escribiremos una debajo de otra. Por ejemplo:

$$\begin{array}{lcl} f.(x,y) & \doteq & (a+1) \times (b+2) \\ & & [\![\, a \doteq (x+y)/2 \\ & & \;\; b \doteq (x+y)/3 \,]\!] \end{array}$$

7.2. Análisis por casos

Es frecuente la definición de funciones por *análisis por casos*. Por ejemplo :

$$f.x = \begin{cases} x+2 & \text{si } x \leq 10 \\ x-2 & \text{si } x > 10 \end{cases}$$

La definición de arriba consiste en dos expresiones, cada una de ellas contiene una expresión booleana, que llamamos *guarda*, $x \leq 10$ y $x > 10$.

Consideremos otro ejemplo, el de la función min,

$$min(x,y) = \begin{cases} x & \text{si } x \leq y \\ y & \text{si } x > y \end{cases}$$

La primer alternativa de esta definición, dice que el valor de $min(x, y)$ es x siempre que la evaluación de la expresión $x \leq y$ sea $True$. La segunda alternativa, establece que el valor de la función $min(x, y)$ será y cuando la evaluación $x > y$ dé como resultado $True$. Los dos casos, $x \leq y$ y $x > y$ abarcan todas las posibilidades, por tanto, la función min está definida para todos los posibles valores de x e y.

Adoptaremos el siguiente formato, para definir funciones mediante análisis por casos:

$$\begin{array}{lcl} f.x & \doteq & (\quad B_0 \quad \rightarrow \quad E_0 \\ & & \vdots \\ & & \Box \quad B_n \quad \rightarrow \quad E_n \\ & &) \end{array}$$

donde B_i son expresiones booleanas y E_i son expresiones del mismo tipo que $f.x$. Se entiende que el valor de $f.x$ será el valor de E_i cuando B_i sea $True$. Las expresiones B_i son las guardas.

La función min anterior la escribimos entonces como:

$$\begin{array}{lcl} min.a.b & \doteq & (\quad a \leq b \quad \rightarrow \quad a \\ & & \Box \quad a \geq b \quad \rightarrow \quad b \\ & &) \end{array}$$

De acuerdo a esta definición las guardas no son disjuntas, esto no significa que la función pueda comportarse de manera diferente en dos evaluaciones, sino que no es relevante cual de las dos guardas se elige en el caso en que ambas sean verdaderas. Esto permite cierto grado de libertad a la hora de implementar un programa.

La función factorial puede definirse usando análisis por casos así:

$$\begin{array}{lcl} fac.n & \doteq & (\quad n = 0 \quad \rightarrow \quad 1 \\ & & \Box \quad n \neq 0 \quad \rightarrow \quad n \times fac.(n-1) \\ & &) \end{array}$$

Observemos que esta es una definición recursiva.

Veremos ahora el uso de definiciones locales en una definición que utiliza análisis por casos, por ejemplo:

$$\begin{array}{lcl} f.x.y & \doteq & (\quad x > 10 \quad \rightarrow \quad x + a \\ & & \Box \quad x \leq 10 \quad \rightarrow \quad x - a \\ & & \qquad\qquad\quad [\![a \doteq (x + y)/2]\!] \\ & &) \end{array}$$

Por último, también es posible definir expresiones mediante análisis por casos, sea por ejemplo, una expresión definida de acuerdo a dos alternativas:

$$\begin{array}{lcl} E & \doteq & (\quad B_0 \quad \rightarrow \quad E_0 \\ & & \Box \quad B_1 \quad \rightarrow \quad E_1 \\ & &) \end{array}$$

Existen reglas que permiten manipular tales expresiones. Supongamos que queremos demostrar que la expresión E definida arriba cumple una cierta propiedad P, es decir queremos probar que se satisface $P.E$. Para esto es suficiente demostrar que se cumple la conjunción de:

- $B_0 \vee B_1$
- $B_0 \Rightarrow P.E_0$
- $B_1 \Rightarrow P.E_1$

El primer punto requiere que al menos una de las guardas sea verdadera. El segundo y el tercero, piden que suponiendo la guarda verdadera, se pueda probar el caso correspondiente. Esto nos da un método de demostración y por tanto de derivación de programas que utilizaremos más adelante.

7.3. Pattern Matching

Vamos a introducir una abreviatura cómoda para escribir ciertos análisis por casos que ocurren con frecuencia. Muchas veces las guardas se usan para discernir la forma del argumento y hacer referencia a sus componentes. Por ejemplo, en el caso del factorial, las guardas se usan para determinar si el argumento es igual a 0 o no. Una forma alternativa de describir esta función es usar en los argumentos de la definición de función un patrón o *pattern*, el cual permite distinguir si el número es 0 o no. Para ello, se usa el hecho de que un número natural es o bien 0 o bien de la forma $(n+1)$, con n cualquier natural (incluído el 0). Luego la función factorial podría definirse como:

$$\begin{array}{rcl} fac.0 & \doteq & 1 \\ fac.(n+1) & \doteq & (n+1) \times fac.n \end{array}$$

Es importante notar que el pattern sirve no sólo para distinguir los casos sino también para tener un nombre en el cuerpo de la definición (en este caso n) el cual refiera a la componente del pattern.

Otros patterns posibles para los naturales podrían ser 0, 1 y $n+2$, con los cuales una definición de función sería de la forma:

$$\begin{array}{rcl} f.0 & \doteq & E_0 \\ f.1 & \doteq & E_1 \\ f.(n+2) & \doteq & E_2 \end{array}$$

Aquí los patterns están asociados a tres guardas, dos consideran los casos en que el argumento es 0 y 1 y el último para el caso en que el argumento es 2 o más. Esta definición se traduce a análisis por casos como:

$$\begin{array}{lcll} f.n & \doteq & (& n = 0 \ \rightarrow \ E_0 \\ & & \square & n = 1 \ \rightarrow \ E_1 \\ & & \square & n \geq 2 \ \rightarrow \ E_2[n := n - 2] \\ & &) & \end{array}$$

Otra posibilidad para el caso de los naturales es separar entre pares e impares, por ejemplo la función:

$$\begin{array}{rcl} f.(2 \times n) & \doteq & E_0 \\ f.(2 \times n + 1) & \doteq & E_1 \end{array}$$

se traduce en una definición que utiliza análisis por casos, definiendo el predicado par,

$$\begin{array}{lcll} f.n & \doteq & (& par.n \ \rightarrow \ E_0[n := \frac{n}{2}] \\ & & \square & \neg par.n \ \rightarrow \ E_1[n := \frac{n-1}{2}] \\ & &) & \end{array}$$

7.4. Tipos

Toda expresión tiene un tipo asociado, lo cual significa que la expresión se evalúa como un miembro del conjunto descripto por su tipo.

Si a una expresión no se le puede asignar un tipo, la misma será considerada incorrecta.

En nuestra notación de programas utilizaremos 5 tipos básicos. Tres de ellos son tipos numéricos: $\mathbf{Nat}$, que incluye los números naturales; $\mathbf{Int}$, los enteros y $\mathbf{Real}$, los números reales. Los 2 restantes son $\mathbf{Bool}$, para los valores $True$ y $False$; y $\mathbf{Char}$, para los caracteres.

Ejemplos:	$1/2$	es de tipo $\mathbf{Real}$
	'c'	es de tipo $\mathbf{Char}$
	par	es de tipo $\mathbf{Int} \rightarrow \mathbf{Bool}$

Además de los tipos básicos existen los tipos compuestos, que se forman a partir de otros tipos. En esta sección definiremos 2 de ellos: tuplas y listas.

En ocasiones no se desea especificar un tipo en particular. En estos casos, el tipo se indicará con una letra mayúscula. Por ejemplo, la función $id.x = x$ tiene sentido como $id : \mathbf{Int} \rightarrow \mathbf{Int}$, pero también como $id : \mathbf{Char} \rightarrow \mathbf{Char}$ o $id : \mathbf{Bool} \rightarrow \mathbf{Bool}$. Si no nos interesa especificar un tipo en particular, escribiremos $id : A \rightarrow A$. La variable A se denomina *variable de tipo*. Cuando el tipo de una expresión incluya variables, diremos que es un tipo *polimórfico*.

7.4.1. Tipos básicos

Los tipos básicos mencionados anteriormente serán descriptos por las expresiones canónicas y las operaciones posibles entre elementos de ese tipo.

Tipos $\mathbf{Nat}$, $\mathbf{Int}$ y $\mathbf{Real}$: Las expresiones canónicas son las constantes (números naturales, enteros y reales). Las operaciones usadas para procesar elementos de estos tipos son: $+$,$-$,$\times$,

$/$, con las propiedades usuales. Además para los tipos $\mathbf{Nat}$ e $\mathbf{Int}$ tendremos las funciones div y mod, que devuelven, respectivamente, el cociente y el resto de la división entre dos números. Por ejemplo: $div.\,16.\,5 = 3$ y $mod.\,16.\,5 = 1$.

Tipo $\mathbf{Bool}$: Hay dos expresiones canónicas, $True$ y $False$. Una función que retorna un valor de verdad se llama predicado. Los booleanos son el resultado de los operadores de comparación $=$, $\neq$, $>$, $<$, $\geq$ y $\leq$. Estos operadores se aplican no sólo a números sino a otros tipos (por ejemplo: Char), siempre y cuando los dos argumentos a comparar tengan el mismo tipo. Es decir, cada operador de comparación es una función polimórfica de tipo $A \to A \to Bool$. Los booleanos pueden combinarse usando los operadores lógicos $\vee, \wedge, \neg,$, etc. con las propiedades usuales.

Tipo $\mathbf{Char}$: Constituído por todos los caracteres, que se denotan encerrados entre comillas simples, por ejemplo: 'a', 'B', '7'. También incluye los caracteres de control no visibles, como el espacio en blanco, el *return*, etc. Una secuencia de caracteres se denomina $\mathbf{String}$, y se denota encerrada entre comillas dobles, por ejemplo: "hola".

7.4.2. Tuplas

Una *tupla* es una colección ordenada de expresiones, no necesariamente del mismo tipo. Si b y c son expresiones, la 2-tupla (b, c) también es llamada par ordenado. También podemos formar ternas, cuaternas, y n-tuplas en general. Las tuplas tienen una utilidad concreta, supongamos que necesitamos considerar datos, correspondientes al nombre, dirección y teléfono de una persona, la estructura más adecuada para esta clase de información es una tupla. Son ejemplos de tuplas ('B', $3.\,8, x + 2$), ('María', 45) y ('Peña','Córdoba 350', 4400999). El tipo de cada tupla se indica entre paréntesis de acuerdo al tipo de cada una de sus componentes, por ejemplo el tipo del primer ejemplo es $(\mathbf{Char}, \mathbf{Real}, \mathbf{Real})$.

Dada una n-tupla, supondremos definido el operador de indexación de tuplas

$$. : (A_0, \ldots, A_n) \to \mathbf{Nat} \to A_i$$

el cual satisface la propiedad

$$(a_0, \ldots, a_n).i = a_i$$

7.4.3. Listas

Una lista es una colección de valores ordenados, todos del mismo tipo. El tipo de una lista es $[A]$, donde A es el tipo de sus elementos. La lista vacía podría considerarse de cualquier tipo, por eso se le asigna el tipo polimórfico $[A]$, a menos que quede claro del contexto que tiene un tipo en particular.

Las listas pueden ser finitas o infinitas, pero en este curso sólo utilizaremos listas finitas. Cuando las listas son finitas se denotan entre corchetes, con sus elementos separados por comas.

A continuación presentamos ejemplos de listas:

i) $[0, 1, 2, 3]$ lista de tipo $[\mathbf{Nat}]$

ii) [('B',$True$),('C',$False$)] lista de tipo $[(\mathbf{Char}, \mathbf{Bool})]$

iii) ['a'] lista de tipo $[\mathbf{Char}]$

iv) $[[2,3,5,6],[1,4,5]]$ lista de tipo $[[\mathbf{Real}]]$

Convencionalmente utilizaremos el símbolo x para indicar un elemento de una lista, xs para una lista, xss para una lista de listas. etc.

Constructores de listas

Una lista se puede generar con los siguientes constructores:

1. $[\,]$ lista vacía.
2. $\triangleright$ agrega un elemento a una lista por la izquierda.

   Si x es de tipo A y xs es de tipo $[A]$, la operación $x \triangleright xs$ agrega x a la izquierda de la lista xs.

   Ejemplo: $[0,1,2,3] = 0 \triangleright [1,2,3] = 0 \triangleright (1 \triangleright (2 \triangleright (3 \triangleright [\,])))$

Los tipos de estos constructores son:

$$[\,] : [A]$$

$$\triangleright : A \to [A] \to [A]$$

Operaciones entre listas

Existen cinco operaciones fundamentales sobre listas, que enunciaremos a continuación:

Concatenar $+\!\!+ : [A] \to [A] \to [A]$

Dadas dos listas b y c del mismo tipo, la concatenación de las listas b y c que denotaremos $b +\!\!+ c$ da como resultado otra lista consistente en los elementos de c ubicados a continuación de los de b:

Por ejemplo, $[0,1] +\!\!+ [2,3] = [0,1,2,3]$, $[0,1] +\!\!+ [] +\!\!+ [2,3] = [0,1,2,3]$, y $[0,1] +\!\!+ [1,3] = [0,1,1,3]$.

Calcular longitud $\# : [A] \to \mathbf{Nat}$

Dada una lista b, el operador $\#$ aplicado a b que denotaremos $\#b$ devuelve el número de elementos que contiene b.

Por ejemplo, $\#([0,1,2,3]) = 4$, $\#[\,] = 0$.

Tomar $n \uparrow [A] \to \mathbf{Nat} \to [A]$

Dada una lista b y un número n la operación que denotaremos $b \uparrow n$ devuelve otra lista formada por los primeros n elementos de la lista b. Cuando n es mayor que la longitud de b, $b \uparrow n = b$.

Por ejemplo, $[0,1,2,3] \uparrow 3 = [0,1,2]$, $[0,1,2,3] \uparrow 5 = [0,1,2,3]$ y $[\,] \uparrow n = [\,]$.

Tirar $n \downarrow: [A] \to \mathbf{Nat} \to [A]$

Dada una lista b y un número n la operación que denotaremos $b \downarrow n$ devuelve otra lista sin los primeros n elementos de la lista b. Cuando n es mayor que la longitud de b, $b \downarrow n = [\,]$.

Por ejemplo, $[0,1,2,3] \downarrow 3 = [3]$, $[0,1,2,3] \downarrow 5 = [\,]$ y $[\,] \downarrow n = [\,]$.

Indexar $\cdot : [A] \to \mathbf{Nat} \to A$

Dada una lista b y un número n la operación que denotaremos $b.n$ devuelve el elemento de la lista b que se encuentra en la posición indicada por n. Esta operación no está definida cuando $n > \#b - 1$.

Por ejemplo, $[0,1,2,3].\,3 = 3$, $[1,2,3].\,0 = [1]$ y $[1,2].\,3$ no está definido.

Propiedades de los operadores

A continuación enunciamos algunas propiedades importantes que poseen los operadores definidos en la sección anterior:

Concatenar:

- $(xs +\!\!+ ys) +\!\!+ zs = xs +\!\!+ (ys +\!\!+ zs)$
- $xs +\!\!+ [\,] = xs = [\,] +\!\!+ xs$
- $(x \triangleright xs) +\!\!+ ys = x \triangleright (xs +\!\!+ ys)$
- $$(xs +\!\!+ ys).i \;=\; \begin{array}{lcl} (& i < \#xs & \to \; xs.i \\ \square & i \geq \#xs & \to \; ys.(i - \#xs) \\ ) & & \end{array}$$
- $(xs +\!\!+ ys) = [\,] \Leftrightarrow xs = [\,] \wedge ys = [\,]$

Calcular longitud:

- $\#[\,] = 0$
- $\#(xs +\!\!+ ys) = \#xs + \#ys$
- $$\#(xs \uparrow n) \;=\; \begin{array}{lcl} (& n \leq \#xs & \to \; n \\ \square & n > \#xs & \to \; \#xs \\ ) & & \end{array}$$
- $$\#(xs \downarrow n) \;=\; \begin{array}{lcl} (& n \leq \#xs & \to \; \#xs - n \\ \square & n > \#xs & \to \; 0 \\ ) & & \end{array}$$

Definiciones de operadores

Tomar n:

- $[\,] \uparrow n \doteq [\,]$
- $(x \triangleright xs) \uparrow 0 \doteq [\,]$
- $(x \triangleright xs) \uparrow (n+1) \doteq x \triangleright (xs \uparrow n)$

Tirar n:

- $[\,] \downarrow n \doteq [\,]$
- $(x \triangleright xs) \downarrow 0 \doteq (x \triangleright xs)$
- $(x \triangleright xs) \downarrow (n+1) \doteq xs \downarrow n$

Indexar:

- $(x \triangleright xs).0 \doteq x$
- $(x \triangleright xs).(i+1) \doteq xs.i$

Longitud:

- $\#[\,] \doteq 0$
- $\#(x \triangleright xs) \doteq 1 + \#xs$

7.5. Currificación

Consideremos una vez más la función min,

$$\begin{array}{lcl} min.x.y & \doteq & (\quad x \leq y \ \rightarrow \ x \\ & & \Box \ \ x > y \ \rightarrow \ y \\ & &) \end{array}$$

Observemos que en esta definición, los argumentos están escritos sin paréntesis y sin comas, si agregáramos paréntesis y comas conseguiríamos otra función:

$$\begin{array}{lcl} \hat{min}.(x,y) & \doteq & (\quad x \leq y \ \rightarrow \ x \\ & & \Box \ \ x > y \ \rightarrow \ y \\ & &) \end{array}$$

Las funciones min y $\hat{min}$ están íntimamente relacionadas, pero existe una diferencia sutil entre ellas, tienen distinto tipo. La función $\hat{min}$ toma un argumento estructurado, es decir toma un argumento constituído por dos números, su tipo es:

$$\hat{min} : (\mathbf{Int}, \mathbf{Int}) \to \mathbf{Int}$$

La función min en cambio, toma dos argumentos, pero de a uno a la vez, su tipo está dado por:

$$min : \mathbf{Int} \to (\mathbf{Int} \to \mathbf{Int})$$

Dicho de otro modo, min es una función que toma un valor numérico y devuelve una función (de tipo $\mathbf{Int} \to \mathbf{Int}$). Es decir, para cada valor de x la expresión $(min.x)$ describe a una función que toma un valor y y devuelve el mínimo entre x e y.

Otro ejemplo, la función add:

$$add.x.y \doteq x + y$$

también tiene tipo $\mathbf{Int} \to (\mathbf{Int} \to \mathbf{Int})$. Para cada x la función $add.x$ agrega x a "algo". En particular, la función $add.\,1$ es la función *sucesor*, que incrementa en una unidad su argumento y la función $add.\,0$ es la identidad sobre números.

El mecanismo simple de reemplazar argumentos estructurados por una sucesión de argumentos simples es conocido como *currificación*, en honor al lógico estadounidense Haskell B. Curry. Una de las ventajas de la currificación es que permite reducir el número de paréntesis en una expresión. Para trabajar correctamente en la currificación, es necesario que la operación de aplicación de función sea asociativa a izquierda. Esto es, $min.x.y$ significa $(min.x).y$ y no $min(x, y)$

7.6. Ejercicios

7.1 Definir la función $sgn : Int \to Int$ que dado un número devuelve $1, 0$ o -1, en caso que el número sea positivo, cero o negativo respectivamente.

7.2 Definir la función $abs : Int \to Int$ que calcula el valor absoluto de un número.

7.3 Definir el predicado $bisiesto : Nat \to Bool$ que determina si un año es bisiesto. Recordar que los años bisiestos son aquellos que son divisibles por 4 pero no por 100 a menos que también lo sean por 400. Por ejemplo, 1900 no es bisiesto pero 2000 sí lo es.

7.4 Definir los predicados $equi$ e $isoc$ que toman tres números, que representan las longitudes de los lados de un triángulo y determinan respectivamente si el triángulo dado es equilátero o isósceles.

7.5 Definir la función $edad : (Nat, Nat, Nat) \to (Nat, Nat, Nat) \to Int$, que dadas dos fechas indica el tiempo en años transcurrido entre ellas. Por ejemplo:

$$edad.(20, 10, 1968).(30, 4, 1987) = 18$$

7.6 En un prisma rectangular, llamaremos h a la altura, b al ancho y d a la profundidad. Completar la siguiente definición del área del prisma:

$$\begin{array}{lcl} area.h.b.d & \doteq & 2 \times frente + 2 \times lado + 2 \times arriba \\ & & [\![\qquad\qquad]\!] \end{array}$$

donde *frente, lado* y *arriba* son las caras frontal, lateral y superior del prisma.

7.7 Definir la función $raices$, que devuelve las raíces de un polinomio de segundo grado.

7.8 Suponer el siguiente juego: m jugadores en ronda comienzan a decir números naturales consecutivamente. Cuando toca un múltiplo de 7 o un número con algún dígito igual a 7, el jugador debe decir "pip" en lugar del número. Se pide, encontrar un predicado que sea $True$ cuando el jugador debe decir "pip" y $False$ en caso contrario. Resolver el problema para $0 \leq n \leq 9999$. ***Sugerencia***: Definir una función $unidad$, que devuelva la cifra de las unidades de un número. Ídem con las decenas, etc: para ello usar las funciones div y mod.

7.9 Dar el tipo de las siguientes funciones:

$$\begin{array}{lll} \text{a)} & f.x & \doteq & 3 + cuadrado.\, x \\ \text{b)} & g.(a, b) & \doteq & a + cuadrado.\, b \\ \text{c)} & h.\, x & \doteq & (x + 1, x > 1{,}5) \\ \text{d)} & k.(x, y) & \doteq & x \\ \text{e)} & f.x & \doteq & x + 1 \end{array}$$

$$\begin{array}{lll} \text{f)} & f.[\,] & \doteq & [\,] \\ & f.\,(x \triangleright xs) & \doteq & [\,cuadrado.x\,] +\!\!+ f.xs \end{array}$$

donde la función cuadrado tiene asociada la siguiente signatura: $cuadrado : Int \to Int$.

7.10 Reescribir las siguientes funciones utilizando una definición por pattern matching si la misma se encuentra definida por análisis por casos y viceversa. Además para cada función dar su tipo.

$$\begin{array}{lll} \text{a)} & f.3 & \doteq & 0 \\ & f.4 & \doteq & 0 \\ & f.(n + 5) & \doteq & n \times 3 \end{array}$$

b) $$\begin{array}{lcll} f.n & \doteq & (& impar.n \;\to\; n/4 \\ & & \Box & \neg\, impar.n \;\to\; n \times 2 \\ & &) & \end{array}$$

c) $$\begin{array}{lcl} f.0 & \doteq & 35 \\ f.(n+1) & \doteq & 2 \times f.n + 1 \end{array}$$

d) $$\begin{array}{lcl} f.0 & \doteq & 1 \\ f.(n+1) & \doteq & f.n + 2 \times n + 1 \end{array}$$

e) $$\begin{array}{lcl} g.0 & \doteq & 1 \\ g.(2 \times n + 1) & \doteq & n \times g.n \\ g.(2 \times n + 2) & \doteq & g.(n \times n) \end{array}$$

f) $$\begin{array}{lcl} h.[\,] & \doteq & [\,1\,] \\ h.(x \triangleright xs) & \doteq & [\,x+1\,] +\!\!+ h.xs \end{array}$$

g) $$\begin{array}{lcll} f.xs & \doteq & (& \#xs \geq 3 \;\to\; xs \downarrow 2 \\ & & \Box & \#xs < 3 \;\to\; [\,0,1\,] +\!\!+ xs \\ & & \Box & \#xs = 0 \;\to\; [\,] \\ & &) & \end{array}$$

7.11 Definir las siguientes funciones utilizando las funciones para listas definidas en el capítulo:

a) $hd : [A] \to A$ devuelve el primer elemento de una lista. (hd es la abreviatura de *head*).

b) $tl : [A] \to [A]$ devuelve toda la lista menos el primer elemento. (tl es la abreviatura de *tail*).

c) $last : [A] \to A$ devuelve el último elemento de una lista.

d) $init : [A] \to [A]$ devuelve toda la lista menos el último elemento.

7.12 Definir las funciones dadas en el ejercicio anterior usando la técnica de pattern matching.

7.13 Dar una definición no currificada de las función area definida en los ejercicios 8.6.

7.14 Dar una definición currificada de las función sum_rac, que suma dos números racionales:

$$\begin{array}{l} sum\_rac : ((Int, Int), (Int, Int)) \to (Int, Int) \\ sum\_rac.((x, y), (z, w)) \doteq (w \times x + y \times z, y \times w) \end{array}$$

CAPÍTULO 8
El Modelo computacional

Índice del Capítulo

8.1. Valores **139**
8.2. Forma canónica **141**
8.3. Evaluación **142**
8.4. Un modelo computacional más eficiente **144**
8.5. Ejercicios **146**

8.1. Valores

En el formalismo básico, como en la programación funcional, una expresión es usada para describir (o denotar) un valor. Entre las clases de valores que una expresión puede denotar, se encuentran los números, valores booleanos, listas, tuplas y funciones.

Es importante distinguir entre un valor y su representación por medio de expresiones. Por ejemplo, la función $cuadrado : Int \to Int$ definida como

$$cuadrado.x \doteq x \cdot x$$

permite representar al número "doscientos veinticinco" como $cuadrado.(3 \times 5)$ o también $cuadrado.\,15$. Sin embargo, ninguna de estas representaciones (ni siquiera 225) es este valor. Todas ellas son representaciones concretas del valor abstracto "doscientos veinticinco".

Lo mismo sucede con otros tipos de valores.

booleanos: Las expresiones

- $False \wedge True$

- $False \vee False$
- $\neg True$
- $False$

denotan todas el valor “falso”.

pares: Las expresiones

- $(cuadrado.(3 \cdot 5), 4)$
- $((225, 2 + 2), True).\, 0$
- $(225, 4)$

todas denotan el “par cuya primera coordenada es el número doscientos veinticinco y su segunda coordenada es el número cuatro”.

listas: Las expresiones

- $[cuadrado.\, 1, 4 - 2, -44 + 47]$
- $[[\,], [1, 2, 3]].\, 1$
- $[1, 2, 3]$

denotan la “lista con los elementos uno, dos y tres en este orden”.

números: Las expresiones

- $(2, 3).\, 1$
- $\#[0, 1, 2]$
- 3

denotan el número “tres”.

funciones: Dadas las funciones

$$\begin{array}{lcl} f.x.y & \doteq & (\quad x \geq y \quad \rightarrow \quad x \\ & & \square \quad x \leq y \quad \rightarrow \quad y \\ & &) \end{array}$$

$$\begin{array}{lcl} g.x.y & \doteq & (\quad x > y \quad \rightarrow \quad x \\ & & \square \quad x < y \quad \rightarrow \quad y \\ & & \square \quad x = y \quad \rightarrow \quad x \\ & &) \end{array}$$

las expresiones

- f
- g
- $id \circ f$
- $id \circ g$

denotan todas la función "máximo entre dos números".

Las computadoras no manipulan valores, sólo pueden manejar representaciones concretas de éstos. Así, por ejemplo, las computadoras utilizan la representación binaria de números enteros para denotar este tipo de valores. Con otros tipos de valores (caracteres, listas, pares, funciones, etc.) sucede lo mismo, las computadoras sólo manejan sus representaciones.

8.2. Forma canónica

Volvamos nuevamente a las diferentes formas de representar el número doscientos veinticinco:

$$cuadrado.(3 \cdot 5), \quad cuadrado.\,15 \quad y \quad 225.$$

Cuando se evalúa alguna de estas expresiones, en realidad se consigue el valor que denota la expresión. Pero, como ya dijimos, las computadoras no manejan valores. La única forma que tenemos para reconocer el valor abstracto que se obtiene como resultado de la ejecución, es por medio de alguna representación del mismo que nos devuelva la computadora.

Es así que la computadora deberá elegir una representación para mostrar un resultado que nos haga reconocer el valor de la expresión. Pero, ¿cuál de todas?. La computadora podría elegir como resultado a la misma expresión. Por ejemplo, podría elegir la expresión $cuadrado.(3 \cdot 5)$ haciendo de la evaluación de la expresión una operación trivial. Pero esto no nos ayudaría a reconocer el valor de la expresión de una forma inmediata.

Por otra parte, es necesario que la representación del valor resultante de la evaluación sea única. De esta forma, seleccionaremos del conjunto de expresiones que denotan el mismo valor una expresión que llamaremos *expresión canónica* de ese valor. Una forma de resolver esta cuestión es definiendo el *conjunto de expresiones canónicas* como un subconjunto propio de todas las expresiones posibles. Al construirlo, se deberán tener en cuenta algunas cuestiones de índole práctico. Sería conveniente, que las expresiones canónicas sean las más simples posible, de modo de identificarlas con rapidez, por ejemplo para identificar el número "doscientos veinticinco" elegimos 225 en lugar de $15 \cdot 15$.

Definimos entonces, el conjunto de expresiones canónicas para los distintos tipos:

booleanas: $True$ y $False$.

números: $-1;\ \ 0;\ \ 1;\ \ 2;\ \ 3,15$, etc, o sea la representación decimal del número.

tuplas: $(E_0, E_1, \ldots, E_n)$ donde $n > 0$ y $E_0 \ldots E_n$ son expresiones canónicas.

listas: $[E_0, \ldots, E_{n-1}]$ donde $n > 0$ y $E_0, \ldots, E_{n-1}$ son expresiones canónicas.

Observemos que no hemos incluído ejemplos de formas canónicas de funciones. No lo hicimos porque los valores funcionales no tienen una representación canónica. Por ejemplo, la función $cuadrado$ se puede definir de diferentes maneras, pero no podemos considerar alguna de estas definiciones como canónica y usar el mismo criterio para elegir la forma canónica de cualquier función.

Existen otros valores que tampoco tienen una representación canónica. Por ejemplo el número π. En el formalismo básico todas las expresiones son finitas y no es posible obtener una expresión decimal finita del número π.

Hasta ahora hemos visto valores que no tienen una forma canónica (funciones, número π). También puede suceder que una expresión no tenga forma canónica. Por ejemplo, dada la siguiente definición:

$$inf \doteq inf + 1$$

La expresión inf (de tipo **Int**) no tiene forma canónica, dado que la expresión no denota un valor bien definido.

Otro ejemplo es la expresión $1/0$, la cual tampoco corresponde a ningún valor numérico.

8.3. Evaluación

De acuerdo a lo visto, diremos que la ejecución de un programa es el proceso de encontrar su forma canónica. Vamos a ver en forma más detallada, el proceso de búsqueda de expresiones canónicas que hacen las computadoras. Una computadora evalúa una expresión (o ejecuta un programa) buscando su forma canónica y mostrando ese resultado. Con los lenguajes funcionales las computadoras alcanzan este objetivo a través de múltiples pasos de *reducción* de las expresiones para obtener otra equivalente más simple. El sistema de evaluación dentro de una computadora está hecho de forma tal que cuando ya no es posible reducir la expresión, se ha llegado a la forma canónica (ver Figura 8.1).

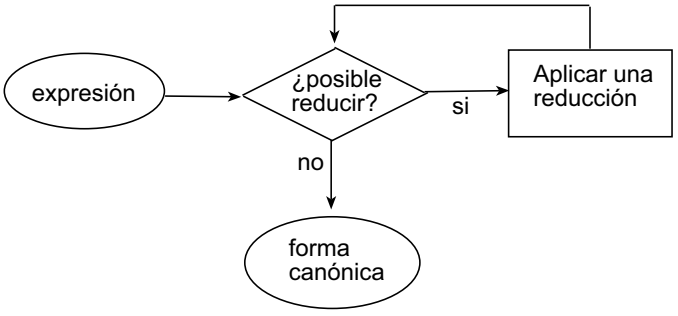

Figura 8.1: Sistema de evaluación

Podemos ver el proceso que ejecuta este sistema como una forma particular del cálculo de nuestro formalismo básico en la cual siempre se usa la regla de desplegado en las definiciones. Consideremos la expresión $cuadrado.(3 \times 5)$,

$$
\begin{array}{ll}
 & cuadrado.(3 \times 5) \\
= & \quad \langle \text{definición de } \times \rangle \\
 & cuadrado.\, 15 \\
= & \quad \langle \text{definición de cuadrado} \rangle \\
 & 15 \times 15 \\
= & \quad \langle \text{definición de } \times \rangle \\
 & 225
\end{array}
$$

La última expresión no puede seguir siendo reducida y es la expresión canónica del valor buscado. Notemos que no hay una única forma de reducir una expresión; la expresión anterior podría haberse evaluado de la siguiente manera:

$$
\begin{array}{ll}
 & cuadrado.(3 \times 5) \\
= & \quad \langle \text{definición de cuadrado} \rangle \\
 & (3 \times 5) \times (3 \times 5) \\
= & \quad \langle \times \text{ aplicado al primer factor} \rangle \\
 & 15 \times (3 \times 5) \\
= & \quad \langle \times \text{ aplicado al segundo factor} \rangle \\
 & 15 \times 15 \\
= & \quad \langle \text{definición de } \times \rangle \\
 & 225
\end{array}
$$

En el primer paso de reducción hemos elegido primero la regla para $cuadrado$, decisión que no fue tomada en el proceso de reducción anterior. Una característica típica de la programación funcional es que si dos secuencias de reducción diferentes terminan, conducen al mismo resultado. En otras palabras, el significado de una expresión es su valor y la tarea del computador consiste simplemente en obtenerlo.

Puede suceder que el proceso de evaluación nunca termine. Por ejemplo tratemos de evaluar la expresión inf definida en la sección anterior:

$$
\begin{array}{ll}
 & inf \\
= & \langle \text{definición de } inf \rangle \\
 & inf + 1 \\
= & \langle \text{definición de } inf \rangle \\
 & (inf + 1) + 1 \\
= & \langle \text{definición de } inf \rangle \\
 & ((inf + 1) + 1) + 1 \\
= & \langle \text{definición de } inf \rangle \\
 & (((inf + 1) + 1) + 1) + 1 \\
 & \ldots
\end{array}
$$

Esta secuencia de reducción no termina. Ahora consideremos la función $K.x.y \doteq x$. ¿Qué sucede con la expresión $K.\,3.inf$? Claramente, denota el valor "tres", luego se esperaría que reduzca a la forma canónica 3. Veamos qué sucede cuando aplicamos distintas estrategias de reducción:

$$
\begin{array}{ll}
 & K.\,3.inf \\
= & \langle\text{definición de inf}\rangle \\
 & K.\,3.(inf + 1) \\
= & \langle\text{definición de inf}\rangle \\
 & K.\,3.((inf + 1) + 1) \\
= & \langle\text{definición de inf}\rangle \\
 & K.\,3.(((inf + 1) + 1) + 1)) \\
 & \ldots
\end{array}
$$

Esta secuencia de reducción no termina, en cambio, si aplicamos primero la definición de K, entonces obtenemos la secuencia:

$$
\begin{array}{ll}
 & K.\,3.inf \\
= & \langle\text{definición de } K\rangle \\
 & 3
\end{array}
$$

Esta secuencia termina en un paso, por lo tanto, ciertos modos de simplificar una expresión pueden terminar y otros no.

Los ejemplos anteriores ilustran dos estrategias de reducción. La primera estrategia. llamada reducción normal, consiste en reducir primero la expresión reducible más externa y, en caso en que halla más de una expresión reducible, elige la que está más a la izquierda. La segunda estrategia reduce primero la expresión reducible más interna. Si el proceso de reducción termina utilizando ambas estrategias, el resultado de la reducción será el mismo. Sin embargo la reducción normal tiene la siguiente propiedad:

(8.1) **Teorema.** Si una expresión tiene forma canónica, entonces la reducción que utiliza la estrategia normal la encontrará.

En otras palabras, si nuestro modelo computacional usa la estrategia de reducción normal, va a conseguir la forma canónica, siempre que ella exista.

8.4. Un modelo computacional más eficiente

Según el teorema 8.1 nuestro modelo computacional deberá usar la estrategia de reducción normal si desea encontrar la forma canónica, en el caso de que ésta exista. Pero esta estrategia es, en algunos casos, menos eficiente. Veamos cómo se evalúa la expresión $cuadrado.(cuadrado.\,3)$ cuando la usamos:

$$
\begin{array}{ll}
 & cuadrado.(cuadrado.\,3) \\
= & \quad \langle \text{definición de cuadrado} \rangle \\
 & (cuadrado.\,3) \times (cuadrado.\,3) \\
= & \quad \langle \text{definición de cuadrado} \rangle \\
 & (3 \times 3) \times (cuadrado.\,3) \\
= & \quad \langle \times \text{ aplicado al primer factor} \rangle \\
 & 9 \times (cuadrado.\,3) \\
= & \quad \langle \text{definición de cuadrado } \rangle \\
 & 9 \times (3 \times 3) \\
= & \quad \langle \text{aplicación de } \times \rangle \\
 & 9 \times 9 \\
= & \quad \langle \text{aplicación de } \times \rangle \\
 & 81
\end{array}
$$

En seis pasos de reducción logramos encontrar la forma canónica. Pero ahora utilicemos la siguiente estrategia:

$$
\begin{array}{ll}
 & cuadrado.(cuadrado.\,3) \\
= & \quad \langle \text{definición de cuadrado} \rangle \\
 & cuadrado.(3 \times 3) \\
= & \quad \langle \text{aplicación de } \times \rangle \\
 & cuadrado.\,9 \\
= & \quad \langle \text{definición de cuadrado} \rangle \\
 & 9 \times 9 \\
= & \quad \langle \text{aplicación de } \times \rangle \\
 & 81
\end{array}
$$

y sólo se realizaron cuatro pasos de reducción.

Analicemos detenidamente las dos reducciones. En la expresión $cuadrado.(cuadrado.\,3)$ la función $cuadrado$ aparece dos veces. En los ejemplos se puede ver que en el primer caso se "copió" la subexpresión $cuadrado.\,3$ al ejecutar el primer paso de reducción, produciendo tres desplegados de la función $cuadrado$. En el segundo caso se produjeron dos desplegados de la misma función. Por otra parte, la subexpresión 3×3 fue reducida una vez más que en el segundo caso.

A simple vista esta ineficiencia del modelo computacional parecería no ser importante, pero si se trata de la evaluación de funciones complejas, puede conducir a un grave desperdicio de recursos computacionales.

El fenómeno de "copiado" de una subexpresión se debe a la forma en que fue definida la función cuadrado:

$$cuadrado.x \doteq x \times x$$

donde el parámetro x se repite en la definición.

Una forma de solucionar este problema es cambiando nuestro modelo computacional para que cuando el sistema deba desplegar funciones con esta estructura, lo haga a través de defini-

ciones locales. Así por ejemplo, manteniendo la estrategia de reducción normal, la expresión $cuadrado.(cuadrado.\,3)$ se puede evaluar de esta forma:

$$
\begin{array}{ll}
 & cuadrado.(cuadrado.\,3) \\
= & \quad \langle \text{definición de cuadrado} \rangle \\
 & x \times x \\
 & \quad [\![x = cuadrado.\,3]\!] \\
= & \quad \langle \text{definición de cuadrado} \rangle \\
 & x \times x \\
 & \quad [\![x = 3 \times 3]\!] \\
= & \quad \langle \text{aplicación de } \times \rangle \\
 & x \times x \\
 & \quad [\![x = 9]\!] \\
= & \quad \langle \text{definición local} \rangle \\
 & 9 \times 9 \\
= & \quad \langle \text{aplicación de } \times \rangle \\
 & 81
\end{array}
$$

y así, sólo desplegamos dos veces la función $cuadrado$.

Este modelo computacional, que se basa en la estrategia de reducción normal agregando los mecanismos necesarios para que se puedan compartir subexpresiones, se denomina *modelo computacional lazy* (perezoso) y se utiliza en la mayoría de la implementaciones de los lenguajes funcionales puros. Adoptaremos la evaluación lazy como nuestro modelo computacional porque tiene dos propiedades deseables: (i) termina siempre que cualquier otro método de reducción termine y (ii) no requiere más, sino posiblemente menos pasos que la evaluación más interna y más a la izquierda.

8.5. Ejercicios

8.1 Obtener la forma canónica de la siguiente expresión

$$2 \times cuadrado.(hd.[2, 4, 5, 6, 7, 8]),$$

aplicando las siguientes estrategias:

a) Reducción más interna y más a la izquierda.

b) Reducción normal.

c) Modelo computacional Lazy.

En cada tipo de reducción contar la cantidad de desplegados necesarios de la funciones hd y $cuadrado$. Comparar la eficiencia teniendo en cuenta este último valor.

8.2 Dada la definición

$$linf \doteq 1 \triangleright linf$$

mostrar los pasos de reducción, hasta llegar a la forma canónica de la expresión:

$$hd.linf$$

en los siguientes casos:

a) Reduciendo primero las subexpresiones más internas.

b) Utilizando la estrategia de reducción normal.

Comparar ambos resultados.

8.3 Dada la definición:

$$\begin{array}{rcl} f.x.0 & \doteq & x \\ f.x.(n+1) & \doteq & cuadrado.(f.x.n) \end{array}$$

mostrar los pasos de reducción hasta llegar a la forma canónica de la expresión

$$f.2.3$$

en los siguientes casos:

a) Utilizando la estrategia de reducción normal.

b) Utilizando el modelo computacional lazy.

Comparar ambos resultados.

8.4 Utilizando las definiciones por pattern matching dadas en el capítulo 8 para las funciones: $hd, tl, last, init, \downarrow (tirar), \uparrow (tomar), . (indexar), . (proyeccin)$ y $\#(cardinalidad)$ evaluar las siguientes expresiones utilizando la estrategia de reducción normal:

a) $hd.[\,3,5,8\,]$

b) $tl.[\,[\,1\,],[\,2,3,4\,]\,]$

c) $last.[\,3,4,5\,]$

d) $init.[\,3,4,5\,]$

e) $([\,(3,4),(5,6),(7,8)\,].0).0$

f) $[\,3,4,5,6,7,8\,] \downarrow 3$

g) $[\,3,4,5,6,7,8\,] \uparrow 5$

h) $\#[\,(3,5),(7,9),(11,13)\,] + \#[\,1,2,3\,]$

Para cada uno de estos ejercicios, tener en cuenta la siguiente definición de reescritura de listas:

$$[e_0, e_1, e_2, \ldots, e_{n-2}, e_{n-1}] \doteq e_0 \triangleright [e_1, e_2, \ldots, e_{n-1}] \qquad (R_1)$$

8.5 Utilizando las siguientes funciones

$$\begin{array}{lcl} inf & \doteq & inf + 1 \\ \\ K.x.y & \doteq & x \\ \\ fact.0 & \doteq & 1 \\ fact.(n+1) & \doteq & (n+1) \times fact.n \end{array}$$

a) Reducir hasta llegar a su forma canónica la siguiente expresión, utilizando el modelo computacional lazy:

$$K.(fact.4 \times fact.4).inf$$

b) Dar una estrategia de reducción en la cual el proceso de evaluación de la expresión del apartado anterior nunca termine.

8.6 Imaginemos un lenguaje de expresiones para representar enteros definidos por las siguientes reglas sintácticas:

i. *cero* es una expresión.

ii. Si *E* es una expresión, entonces también lo son *pred(E)* y *suc(E)*.

Un evaluador reduce expresiones de este lenguaje aplicando las siguientes reglas repetidamente hasta que no puedan aplicarse más:

(R1) *suc(pred(E)) = E.*

(R2) *pred(suc(E)) = E.*

a) Simplificar la expresión *suc(pred(suc(pred(pred(cero)))))*.

b) ¿De cuántas maneras posibles se pueden aplicar las reglas de reducción a esta expresión?. ¿Conducen todas ellas al mismo resultado final?

8.7 Continuando con el ejercicio anterior, supongamos que se añade una regla sintáctica adicional al lenguaje:

iii. Si E_1 y E_2 son expresiones, entonces también lo es *suma(E_1, E_2)*.

Las reglas de reducción correspondiente son:

(R3) *suma(cero, E_2) = E_2*.

(R4) *suma(suc(E_1),E_2) = suc(suma(E_1, E_2))*.

(R5) *suma(pred(E_1),E_2) = pred(suma(E_1, E_2))*.

a) Simplificar la expresión *suma(suc(pred(cero)), cero)*.

b) Contar de cuantos modos diferentes se pueden aplicar las reglas de reducción a la expresión anterior. ¿Todos ellos conducen al mismo resultado final?

CAPÍTULO 9
El Proceso de construcción de programas

Índice del Capítulo

9.1. Introducción 151
9.2. Especificaciones 152
9.3. Ejemplos 153
9.4. Más ejemplos 155
9.5. Ejercicios 160

9.1. Introducción

El desarrollo de software es un proceso por el cual, dado un problema, se encuentra un programa (o un conjunto de programas) que lo resuelva eficientemente.

Una de las dificultades esenciales de este proceso consiste en que la descripción del problema a resolver, suele ser poco precisa o incompleta. Sin embargo, esta dificultad no cobró importancia hasta el final de la década del 60', dado que hasta ese momento las computadoras sólo eran usadas para resolver problemas científicos, los cuales estaban expresados en un lenguaje suficientemente preciso.

Con el abaratamiento de los costos las computadoras comenzaron a ser utilizadas para problemas originados en otros ámbitos (en general problemas administrativos). El principal método utilizado entonces, era simplemente partir del problema, el cual estaba expresado de manera informal y poco detallada, y obtener un programa que, por definición, es muy detallado y está escrito en una notación formal. Este salto, fue una de las principales razones de la llamada "crisis del software", la cual produjo un decaimiento en la confiabilidad del software y por lo tanto un decaimiento de los métodos usados para su desarrollo. Otra dificultad de esta metodología, se presentaba a la hora de la corrección del programa. Para ello, se realizaban ensayos con conjuntos de datos para los cuales se conocía el resultado, y si éstos daban los resultados esperados se daba por terminada la tarea. En el caso frecuente en que los resultados no fueran correctos,

se procedía a modificar el programa para intentar corregirlo. Esta tarea era sumamente difícil, debido a que no se sabía a priori si el resultado inesperado se debía a meros errores de programación o a una concepción inadecuada del problema.

En la actualidad, es ampliamente aceptado que el proceso de construcción de programas, debe dividirse en al menos dos etapas: la etapa de *especificación* del problema y la etapa de *programación* o desarrollo del programa.

El resultado de la primera etapa es una *especificación formal* del problema, la cual seguirá siendo abstracta (poco detallada) pero estará escrita con precisión en algún lenguaje cuya semántica esté definida rigurosamente. La segunda etapa dará como resultado un programa y una demostración de que el programa es *correcto* respecto de la especificación dada.

La separación en dos etapas, permite discernir ahora si un programa cuyos resultados no son los esperados, es incorrecto o si, en cambio, es la especificación la que no describe al programa de la manera adecuada.

Este paradigma de desarrollo de software, (primero escribir una especificación clara y luego desarrollar una implementación aceptablemente eficiente), ha sido un foco de investigación activa durante los últimos veinte años, y no debería tomarse como un método panacea aplicable a todas las circunstancias. Sin embargo, se puede mejorar mucho la fiabilidad de los programas tratando de aplicar estos principios siempre que sea posible.

En lo que sigue de este capítulo, ilustraremos a través de ejemplos como pueden construirse especificaciones para problemas relativamente simples (pero para nada triviales).

9.2. Especificaciones

Podemos decir, en un sentido general que una especificación es una descripción formal de la tarea que un programa tiene que realizar. Es común, también definir a la especificación como la respuesta a la pregunta ¿qué hace el programa?, mientras que el programa mismo, conocido como *implementación*, responde a la pregunta ¿cómo se realiza la tarea?

En la literatura, se presenta también a una especificación como un *contrato* entre el programador y el potencial usuario. Este contrato establece de manera precisa cual es el comportamiento del producto que el programador debe proveer al usuario. Usualmente las especificaciones dicen que si el programa se ejecuta para un valor de un conjunto dado, el resultado va a satisfacer una cierta propiedad. Una de las consecuencias del contrato, es que el usuario se compromete a usar el programa sólo para valores de ese conjunto predeterminado y el programador a asegurar que el programa producirá un resultado satisfactorio. En principio, si el usuario ingresa al programa datos que no están en el conjunto de datos aceptables, el comportamiento del programa puede ser cualquiera sin que ello viole el contrato establecido por la especificación.

En el contexto de nuestro formalismo básico, un modo de especificar una función consiste en establecer explícitamente la regla de correspondencia entre argumentos y resultados. La notación que hemos utilizado puede ser muy expresiva y, a veces, una especificación de una función es de hecho un programa. En este caso, se puede ejecutar la especificación directamente. Sin embargo, puede ser tan altamente ineficiente que la posibilidad de ser ejecutada será simplemente de interés teórico. A pesar de haber escrito una especificación ejecutable, el programador

no queda necesariamente eximido de la tarea de producir una versión alternativa equivalente, pero aceptablemente eficiente.

Tomemos por ejemplo, el siguiente problema: dado el conjunto de alumnos de un curso, se pide obtener el alumno con el segundo promedio más alto. Este problema tiene una formulación bastante precisa, pero sin embargo existen ciertas cuestiones sin resolver. Por ejemplo, puede ser que tal alumno no exista (es el caso en que todos los alumnos del curso tengan el mismo promedio), o que haya más de un alumno que cumpla con la misma propiedad. Si nos quedamos con un enunciado informal, la solución para estos casos queda al criterio del programador.

En lo que queda del capítulo, usaremos herramientas introducidas anteriormente para escribir especificaciones precisas. Para ello, interpretaremos las fórmulas del cálculo de predicados y las expresiones cuantificadas como es usual y construiremos con ellas y con el formalismo básico las especificaciones formales. Esto nos permitirá resolver las ambigüedades en la formulación de problemas.

9.3. Ejemplos

(9.1) **Ejemplo.** En el siguiente ejemplo, el usuario propone al programador el problema de calcular la raíz cuadrada de un número real dado.

Primer intento El programador que cuenta con conocimientos de matemática, propone resolver el problema sólo para números reales que no sean negativos. El usuario acepta la propuesta, con lo cual el contrato entre ambos (la especificación) queda establecido como sigue:

El programador se compromete a construir una función $sqrt : Real \to Real$, tal que:

$$(\forall\, x : 0 \leq x : (sqrt.x)^2 = x)$$

Puede pensarse a la especificación como una ecuación a resolver donde la incógnita es $sqrt$, es decir, el problema es encontrar una función $sqrt$ de modo que satisfaga la especificación. Nótese que esta función no necesariamente es única, dado que hay dos soluciones posibles para la raíz cuadrada de un número estrictamente positivo. Esto da lugar a que haya infinitas soluciones, dado que en principio, pueden elegirse determinados valores para los cuales la raíz debe ser positiva y otros para los cuales debe ser negativa.

Una forma estilizada de escribir la misma especificación es separar la **precondición** de la **postcondición**. La primera condición establece las restricciones que determinan los datos aceptables, mientras que la segunda establece las propiedades que serán satisfechas por el resultado. En nuestro ejemplo, podemos escribir:

$$\begin{array}{ll} pre: & 0 \leq x \\ post: & (sqrt.x)^2 = x \end{array}$$

Nótese que la cuantificación de la variable queda implícita. Esto será una práctica usual. En general se supondrá que las variables que aparecen como argumentos de la función especificada están cuantificadas universalmente, siendo esta cuantificación acotada al conjunto de valores definidos por el tipo de la función. La especificación original puede entonces escribirse simplemente así:

$$0 \leq x \Rightarrow (sqrt.x)^2 = x$$

Segundo intento Un fenómeno usual ocurre cuando el programador por alguna razón no puede satisfacer la especificación acordada. Por ejemplo, no será posible para el programador encontrar una solución exacta de $sqrt.\,2$, debido a que cualquier sistema de cómputo en el cual implemente su programa, sólo manejará aproximaciones finitas a los números irracionales. El problema aquí, es que la especificación requiere que la solución sea exacta, por lo tanto el programador tiene que cambiar el contrato con el usuario. Una posible especificación es:

$$\begin{array}{ll} pre: & 0 \leq x \\ post: & |(sqrt.x)^2 - x| < \epsilon \end{array}$$

donde ϵ es una constante a ser negociada.

Escrita de esta manera es obvio que la especificación es más favorable para el programador, pues la postcondición es más débil y por lo tanto más fácil de satisfacer. Si escribimos esta nueva especificación:

$$0 \leq x \Rightarrow \left|(sqrt.x)^2 - x\right| < \epsilon$$

es claro que la especificación del primer intento implica ésta.

Cambios en la especificación:

Como se vio en el ejemplo anterior, es frecuente la necesidad de modificar las especificaciones. En el ejemplo, la especificación fue *debilitada*, es decir se optó por una especificación que requería menos del programa. La forma de debilitarla fue a través de la postcondición. Otra forma de debilitar la especificación es fortalecer la precondición, es decir exigiendo que el programa pueda funcionar sólo con un conjunto más restringido de valores. Por ejemplo, podríamos exigir como requisito que los argumentos para $sqrt$ sean cuadrados perfectos (con lo cual siempre será posible encontrar valores exactos para las raíces). La especificación entonces, resulta:

$$\begin{array}{ll} pre: & (\exists\, y : y \in \mathbb{N} : y^2 = x) \\ post: & (sqrt.x)^2 = x \end{array}$$

Otra situación común es la necesidad de fortalecer la especificación. Esto ocurre cuando el usuario observa que necesita un programa con mejores propiedades que el que tiene, o también

cuando es necesario utilizar el programa para un conjunto de datos más amplio del que consideraba inicialmente. En el ejemplo anterior, puede requerirse que el resultado sea siempre no negativo, con lo cual se está fortaleciendo la postcondición y la especificación completa:

$$0 \leq x \Rightarrow \left|(sqrt.x)^2 - x\right| < \epsilon \wedge 0 \leq sqrt.x$$

(9.2) Ejemplo.

Consideremos ahora la especificación de una función que calcula la segunda mejor nota de un curso. Supondremos que las notas de los alumnos están almacenadas en una lista, con lo cual el tipo de la función sería $[Nat] \rightarrow Nat$. Una manera simple de construir esta especificación es usando una especificación auxiliar que exprese la mayor nota del curso:

$$\begin{array}{ll} pre: & true \\ post: & maxNota.xs = (max\, i : 0 \leq i < \#xs : xs.i) \end{array}$$

La especificación de la función requerida es entonces:

$$\begin{array}{ll} pre: & (\exists\, i, j : 0 \leq i < j < \#xs : \neg(xs.i = xs.j)) \\ post: & segNota.xs = (max\, i : 0 \leq i < \#xs \wedge \neg(xs.i = maxNota.xs) : xs.i) \end{array}$$

La función $maxNota$ es una función auxiliar usada para especificar la función $segNota$, y no es en principio necesario construir un programa para ella. Durante el proceso de construcción del programa para $segNota$ se verá si es también necesario construir un programa para la función auxiliar.

No deben confundirse las igualdades que aparecen en una especificación, denotadas con el signo "$=$", con las definiciones del formalismo básico, para las cuales definimos y utilizamos el signo "$\doteq$".

Notemos que la precondición de esta función, determina que la función no está definida para listas que no contienen al menos dos elementos distintos. Es necesaria esta restricción para estar seguros de que la segunda mejor nota de un curso existe y podamos calcularla.

9.4. Más ejemplos

Dado que no existen reglas generales para determinar las especificaciones de problemas, incluímos a continuación algunos ejemplos. La clave está en saber expresar los problemas planteados en lenguaje corriente de manera concisa, usando el lenguaje formal.

Escribiremos el problema entre comillas. Cabe aclarar que con frecuencia, los problemas pueden especificarse de diferentes formas, con lo cual no deben tomarse las especificaciones que aparezcan a continuación como las únicas posibles.

(9.3) **Ejemplo.** "Dados $x \geq 0$ e $y > 0$ enteros, calcular el cociente y el resto de la división entera de x por y".

El algoritmo de la división entera nos asegura la existencia de un único entero q cociente de la división entera de x por y y un único entero r resto de esta división. Estos números están caracterizados por las propiedades $x = q \times y + r$ y $0 \leq r < y$. Por lo tanto, una función $divMod : Int \to Int \to (Int, Int)$, que calcule la división entera y el resto de la división entera, puede especificarse como:

$$\begin{array}{ll} pre: & x \geq 0 \ \wedge \ y > 0 \\ post: & divMod.x.y = (q, r) \equiv x = q \times y + r \ \wedge \ 0 \leq r < y \end{array}$$

(9.4) **Ejemplo.** "Dado un número entero $x \geq 0$, $sqrtInt.x$ es la raíz cuadrada de x".

La raíz cuadrada entera de un número está definida como el mayor entero positivo cuyo cuadrado es menor o igual que el número en cuestión. La función $sqrtInt : Int \to Int$ puede expresarse en lenguaje formal de la siguiente manera:

$$\begin{array}{ll} pre: & 0 \leq x \\ post: & sqrtInt.x = (max\, z : 0 \leq z \wedge z^2 \leq x) \end{array}$$

alternativamente, puede evitarse el cuantificador máximo de la siguiente manera:

$$\begin{array}{ll} pre: & 0 \leq x \\ post: & 0 \leq sqrtInt.x \ \wedge \ (sqrInt.x)^2 \leq x \ \wedge \ x < (sqrtInt.x + 1)^2 \end{array}$$

(9.5) **Ejemplo.**

Sea xs una lista no vacía de enteros, "$m.xs$ es el mínimo de xs".

Usando el cuantificador min, la especificación de $m : [Int] \to Int$ es inmediata:

$$\begin{array}{ll} pre: & \#xs > 0 \\ post: & m.xs = (min\, i : 0 \leq i < \#xs : xs.i) \end{array}$$

Otra especificación posible se obtiene describiendo con más detalles el mínimo:

$$\begin{array}{ll} pre: & \#xs > 0 \\ post: & (\forall\, i : 0 \leq i < \#xs : xs.i \geq m.xs) \wedge (\exists\, i : 0 \leq i < \#xs : xs.i = m.xs) \end{array}$$

(9.6) **Ejemplo.**

Sea xs una lista no vacía de enteros, "$iguales.xs$ determina si todos los elementos son iguales".

Una manera de especificar la función $iguales : [Int] \to Bool$ se obtiene al considerar que todos los elementos de la lista son iguales si cada elemento es igual al de su derecha.

$$\begin{array}{ll} pre: & \#xs > 0 \\ post: & iguales.xs = (\forall\, i : 0 \leq i < \#xs - 1 : xs.i = xs.(i+1)) \end{array}$$

Pero también podríamos expresar que todos son iguales diciendo que cada uno de ellos es igual a uno fijo, por ejemplo, el primero (sabemos que existe, pues suponemos que la lista es no vacía).

$$\begin{array}{ll} pre: & \#xs > 0 \\ post: & iguales.xs = (\forall\, i : 0 < i < \#xs : xs.i = xs.\,0) \end{array}$$

(9.7) Ejemplo.

"La función $creciente : [Int] \to Bool$ toma una lista no vacía de enteros y determina si la lista está ordenada en forma creciente.".

Si una lista está ordenada en forma creciente, cada elemento de ésta debe ser mayor que el anterior.

$$\begin{array}{ll} pre: & \#xs > 0 \\ post: & creciente.xs = (\forall\, i : 0 < i < \#xs : xs.i > xs.(i-1)) \end{array}$$

Observemos que fue necesario excluir al cero del rango de especificación, para que el término esté siempre definido.

También puede especificarse esta función expresando la relación de desigualdad correspondiente entre cualquier par de índices:

$$\begin{array}{ll} pre: & \#xs > 0 \\ post: & creciente.xs = (\forall\, i : 0 \leq i \leq \#xs : (\forall\, j : i < j < \#xs : xs.i < xs.j)) \end{array}$$

(9.8) Ejemplo. "La función $f : Int \to [Int] \to Bool$ determina si el k-ésimo elemento de la lista xs aloja el mínimo valor de xs".

Notemos que la función f debe aplicarse a dos argumentos, un entero que indica una posición de la lista y una lista. Para que la evaluación tenga sentido, el entero debe estar comprendido en el rango que corresponde al tamaño de la lista. Entonces, podemos escribir:

$$\begin{array}{ll} pre: & 0 \leq k < \#xs \\ post: & f.k.xs = (xs.k = (min\, i : 0 \leq i < \#xs : xs.i)) \end{array}$$

o bien

$$\begin{array}{ll} pre: & 0 \leq k < \#xs \\ post: & f.k.xs = (\forall\, i : 0 \leq i < \#xs : xs.k \leq xs.i) \end{array}$$

(9.9) **Ejemplo.** "La función $minout : [Nat] \to Nat$ selecciona el menor natural que no está en xs".

La función $minout$ extrae el mínimo de un conjunto, éste deberá especificarse dentro del rango, por ejemplo:

$$\begin{array}{ll} pre: & true \\ post: & minout.xs = (min\, i : 0 \leq i \wedge (\forall\, j : 0 \leq j < \#xs : xs.j \neq i) : i) \end{array}$$

(9.10) **Ejemplo.** "La función $lmax$ toma una lista xs de enteros y calcula la longitud del máximo intervalo de la forma $[xs.0, \ldots, xs.k]$, que verifica $xs.i = 0$ para todo $0 \leq i < k$".

Analicemos primero la función $lmax$. Si el primer elemento de la lista xs no es igual a 0, entonces no existe un intervalo de la forma $[xs.0, \ldots, xs.k]$ con elementos nulos, con lo cual la función debe devolver 0. En caso contrario, la lista xs contiene un intervalo nulo y debemos calcular su longitud. Una especificación posible para la función $lmax$ es la siguiente:

$$\begin{array}{ll} pre: & true \\ post: & lmax.xs = max(0, (max\, k : 0 \leq k < \#xs \wedge (\forall\, i : 0 \leq i < k : xs.i = 0) : k)) \end{array}$$

Esto también puede escribirse usando listas en lugar del entero k:

$$\begin{array}{ll} pre: & true \\ post: & lmax.xs = max(0, (max\, as, bs : xs = as +\!\!+ bs \wedge \\ & (\forall\, i : 0 \leq i < \#as : xs.i = 0) : \#as)) \end{array}$$

(9.11) **Ejemplo.** "Dada una lista de enteros, la función P determina si algún elemento de la lista es igual a la suma de todos los anteriores a él".

La función $P : [Int] \to Bool$ se puede especificar como:

$$\begin{array}{ll} pre: & true \\ post: & P.xs = (\exists\, i : 0 < i < \#xs : xs.i = (\sum\, j : 0 \leq j < i : xs.j)) \end{array}$$

(9.12) **Ejemplo.** "Dado un polinomio $p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$, la función ev realiza la evaluación del polinomio para un valor determinado de la variable x. Para especificar esta función, debemos elegir una forma adecuada de representar el polinomio. Como un polinomio queda unívocamente determinado por sus coeficientes, una posibilidad es formar con éstos una lista: $xs = [a_0, a_1, \ldots, a_{n-1}, a_n]$. Más aun, cada lista de reales puede pensarse como lista de coeficientes de un polinomio. El polinomio nulo se representará con la lista $[0]$ o con una lista cuyos elementos sean todos nulos, con esta convención la lista vacía no representa a ningún polinomio. Podemos especificar $ev : [Real] \to Real \to Real$ de la siguiente manera:

$$\begin{array}{ll} pre: & \#xs > 0 \\ post: & ev.xs.y = (\sum\, i : 0 \leq i < \#xs : xs.i \times y^i) \end{array}$$

(9.13) **Ejemplo.** "Dada una lista de valores booleanos, la función $bal : [Bool] \to Bool$ indica si en la lista hay igual cantidad de elementos $True$ que $False$".

El cuantificador N nos permite contar la cantidad de $True$ y $False$ que hay en la lista. Lo que nos interesa es el resultado de la comparación de estas dos cantidades:

$$\begin{array}{ll} pre: & true \\ post: & bal.xs = ((N\, i : 0 \leq i < \#xs : xs.i = True) = \\ & (N\, i : 0 \leq i < \#xs : xs.i = False)) \end{array}$$

o equivalentemente:

$$\begin{array}{ll} pre: & true \\ post: & bal.xs = ((N\, i : 0 \leq i < \#xs : xs.i) = (N\, i : 0 \leq i < \#xs : \neg xs.i)) \end{array}$$

(9.14) **Ejemplo.** "Dada una lista xs de booleanos, la función $balmax.xs$ indica la longitud del máximo segmento de xs que satisface bal".

Dada una lista xs, un segmento de ella es una lista cuyos elementos están en xs, en el mismo orden y consecutivamente. Por ejemplo, si $xs = [2, 4, 6, 8]$ entonces $[2, 4]$, $[4, 6, 8]$, $[\,]$, son segmentos, mientras que $[4, 2]$ y $[2, 6, 8]$ no lo son.

En problemas como éste, suele ser conveniente expresar la lista como una concatenación. Si escribimos $xs = as +\!\!+ bs +\!\!+ cs$, entonces as, bs y cs son segmentos de xs, por lo que podemos usarlos para expresar las condiciones que necesitamos que se satisfagan. Observemos que de los tres segmentos mencionados, el más general es bs, puesto que as comienza en la posición cero de xs, mientras que cs termina necesariamente en la posición última de xs. El segmento bs puede estar en cualquier parte, puesto que tanto as como cs pueden ser vacíos. Como queremos obtener la máxima longitud utilizamos el cuantificador max.

Una especificación de la función $balmax$ es la siguiente:

$$\begin{array}{ll} pre: & true \\ post: & balmax.xs = max(0, (max\, bs : (\exists\, as, cs :: xs = as +\!\!+ bs +\!\!+ cs) \wedge\ bal.bs : \\ & \#bs)) \end{array}$$

o también

$$\begin{array}{ll} pre: & true \\ post: & balmax.xs = max(0, (max\, as, bs, cs : xs = as +\!\!+ bs +\!\!+ cs\ \wedge\ bal.bs : \#bs)) \end{array}$$

Si la lista xs no contiene un segmento balanceado, $balmax$ debe devolver 0, por esta razón $balmax.xs$ es igual al máximo entre 0 y, el entero que corresponde a la longitud del máximo segmento balanceado o $-\infty$ si xs no contiene un segmento balanceado. Esta especificación podría simplificarse si hubiésemos definido el operador max sobre naturales en lugar de enteros, dado que sobre los números naturales el elemento neutro de max es 0.

9.5. Ejercicios

9.1 Expresar en lenguaje formal, los siguientes problemas:

i) Dado un entero $n > 0$: "$pot2.n$ es el mayor entero que vale a lo sumo n y es una potencia de 2".

ii) Dado una lista xs de naturales: "Si xs es no vacía, $mayor.xs$ es el mayor elemento de xs. Si $xs = [\,]$, $mayor.xs$ es igual a 0".

9.2 Sea xs una lista no vacía, con elementos enteros. Expresar en lenguaje formal:

i) Suponiendo que $\#xs > 1$ y que existen al menos dos valores distintos en xs: "$segundoMax.xs$ es el segundo valor más grande de xs". (Ejemplo: si $xs = [3, 6, 1, 7, 6, 4, 7]$, debe resultar $segundoMax.xs = 6$)

ii) "$sum.xs$ es la suma de los elementos de xs".

iii) Dado un entero n: "$ocurrencias.n.xs$ es la cantidad de veces que n aparece en xs".

iv) "La función $distintos : [Int] \to Bool$ determina si todos los valores de la lista que recibe como argumento son distintos".

v) "$prod.xs$ es el producto de todos los valores positivos de xs".

vi) Dado un entero x: "$elem.x.xs$ coincide con el valor de verdad de la afirmación: x está en xs".

9.3 Sea xs una lista no vacía, con elementos booleanos, tal que al menos un elemento de xs es $True$. Expresar en lenguaje formal:

i) "$posmenor.xs$ es igual a la posición del primer elemento $True$ de la lista xs".

ii) "$posmayor.xs$ indica la posición del último elemento de la lista que es equivalente a $True$".

9.4 Sea xs una lista no vacía. Expresar las siguientes especificaciones en lenguaje corriente:

i)
$$\begin{array}{ll} pre: & n \leq \#xs \\ post: & f.n.xs = (\forall\, i : 0 \leq i < n : xs.i \geq 0) \end{array}$$

ii) $\begin{array}{ll} pre: & n \leq \#xs \\ post: & g.n.xs = (\exists\, i : 0 < i < n : xs.(i-1) < xs.i) \end{array}$

iii) $\begin{array}{ll} pre: & n \leq \#xs \\ post: & h.n.xs = (\exists\, i : 0 \leq i < n : xs.i = 0) \end{array}$

iv) $\begin{array}{ll} pre: & n \leq \#xs \\ post: & i.n.xs = (\forall\, p, q : 0 \leq p \wedge 0 \leq q \wedge p + q = n - 1 \wedge n \leq \#xs : xs.p = xs.q) \end{array}$

9.5 Sea xs una lista no vacía de enteros.

i) Especificar la función $decreciente.xs$, que determina si la lista está ordenada en forma decreciente.

ii) Especificar la función $promedio.xs$, que calcula el promedio de todos los elementos de la lista.

9.6 Especificar las siguientes funciones:

i) $f : [Int] \to Bool$
$f.xs$ determina si xs contiene igual cantidad de elementos pares que impares.

ii) $cp : [Int] \to Nat$
$cp.xs$ determina la cantidad de números pares que contiene xs.

iii) $g : Int \to [Int] \to Bool$
$g.k.xs$ determina si el k-ésimo elemento de xs aloja al máximo valor de xs.

iv) $h : Nat \to [Int] \to Bool$
$h.k.xs$ determina si el k-ésimo elemento de xs aloja la primera ocurrencia del máximo valor de xs.

v) $noNulo : [Int] \to Bool$
$noNulo.xs$ es $True$ si y solo si xs no contiene elementos nulos.

vi) $prod : Int \to [Int] \to Bool$
$prod.x.xs$ es $True$ si y solo si x es igual al producto de los elementos de xs.

vii) $meseta : [Int] \to Nat \to Nat \to Bool$
$meseta.xs.i.j$ determina si todos los valores de la lista xs que están entre los índices i y j (ambos incluídos) son iguales.

viii) $ordenada : [Int] \to Nat \to Nat \to Bool$
$ordenada.xs.i.j$ determina si la lista xs está ordenada entre los índices i y j (ambos incluídos). Notar que "ordenado" puede ser creciente o decreciente.

CAPÍTULO 10
Inducción y recursión

Índice del Capítulo

10.1. Introducción 163
10.2. Inducción matemática 164
10.3. Ayudas para pruebas por inducción 168
10.4. Verificación de programas 168
10.5. Inducción sobre listas 170
10.6. Verificación de programas con listas 172
10.7. Ejercicios 174

10.1. Introducción

Una técnica muy utilizada para demostrar propiedades sobre el conjunto de números naturales, es el *principio de inducción*. Este principio provee una herramienta poderosa y fácil de usar. La idea de las demostraciones por inducción se basa en lo siguiente: puesto que sumando 1 al natural k se obtiene $k+1$ que es mayor que k, no existe ningún natural que sea el *mayor* de todos. Sin embargo, partiendo del número 1 se pueden alcanzar todos los enteros positivos, después de un número finito de pasos, pasando sucesivamente de k a $k+1$. Supongamos entonces una cierta propiedad $P(n)$ que sabemos cierta para $n = 1$, luego si la propiedad es válida para un entero en particular y de esto se concluye que también será válida para el entero siguiente, entonces se podrá afirmar que resultará cierta para todos los enteros positivos.

La idea de inducción se puede ilustrar de muchas maneras no matemáticas. Por ejemplo, consideremos una fila de fichas de dominó ubicadas en posición vertical y numeradas, de modo tal que, si se cae hacia atrás la ficha situada en la posición k hará caer a la que está detrás, en la posición $k + 1$. En seguida, se puede intuir lo que occurrirá si la ficha situada en la posición 1 se cae hacia atrás. También, es claro que si fuera empujada hacia atrás la ficha en la

posición n_1 todas las fichas ubicadas por detrás de ésta caerían. Este ejemplo ilustra una ligera generalización del principio de inducción.

10.2. Inducción matemática

Consideremos el siguiente predicado:

$$P.n : (\sum i : 1 \leq i \leq n : 2 \times i - 1) = n^2 \tag{10.1}$$

Por ejemplo, para n igual a 2 y 3 respectivamente, 10.1 establece que $1+3 = 2^2$ y $1+3+5 = 3^2$, lo cual indica que $P.2$ y $P.3$ resultan $true$. ¿Será $P.n$ cierto siempre?, o bien ¿es verdera la cuantificación $(\forall\, n : \mathbb{N} : 0 \leq n : P.n)$?

Una forma de probar que P es cierta para todo número natural es demostrar que $P.0$ es cierta y que suponiendo que P es cierta para todos los naturales menores o iguales a un natural n dado, entonces también es cierta $P.(n+1)$. Esta última afirmación puede escribirse como:

$$(\forall\, n : \mathbb{N} : 0 < n : P.0 \wedge P.1 \wedge \ldots \wedge P(n-1) \Rightarrow P.n) \tag{10.2}$$

Ahora, si ambas cosas son ciertas, podemos ver cómo construir una demostración de P para un número N dado:

- La demostración de $P.0$ ya está hecha.
- De la veracidad de $P.0$ y $P.0 \Rightarrow P.1$ (que no es más que (10.2) en la instancia $n := 1$), y usando Modus Ponens (3.76) concluímos que $P.1$ es verdadera.
- De la veracidad de $P.0 \wedge P.1$ y $P.0 \wedge P.1 \Rightarrow P.2$ (que no es más que (10.2) en la instancia $n := 2$), y usando Modus Ponens (3.76) concluímos que $P.2$ es verdadera.
- ...
- De la veracidad de $P.0 \wedge \ldots \wedge P.(N-1)$ y $P.0 \wedge \ldots \wedge P.(N-1) \Rightarrow P.N$ (que no es más que (10.2) en la instancia $n := N$), y usando Modus Ponens (3.76) concluímos que $P.N$ es verdadera.

Por supuesto, no necesitamos hacer todo esto para demostrar que $P.N$ es verdadera, es suficiente saber que en principio, podemos hacer esto para probarlo.

El principio de *inducción matemática*, permite inferir a partir de las pruebas de $P.0$ y (10.2) que $P.n$ es verdadera para cualquier natural n.

Consideremos de nuevo el ejemplo. Demostraremos primero $P.0$ (llamado *caso base*).

Caso base

$$\begin{array}{ll} & (\sum i : 1 \leq i \leq 0 : 2 \times i - 1) \\ = & \quad \langle \text{neutro de } + \text{ (debido a que el rango es vacío) 6.9} \rangle \\ & 0 \\ = & \quad \langle \text{aritmética} \rangle \\ & 0^2 \end{array}$$

Demostraremos ahora el *paso inductivo*, suponiendo que $P.0$, $P.1$, .. y $P.n$ son ciertas demostraremos que $P.(n+1)$ es cierta.

Paso inductivo

$$
\begin{array}{ll}
 & (\sum\, i : 1 \leq i \leq n+1 : 2 \times i - 1) \\
= & \quad \langle \text{Separación de término (6.24)} \rangle \\
 & (\sum\, i : 1 \leq i \leq n : 2 \times i - 1) + 2 \times (n+1) - 1 \\
= & \quad \langle \text{Hipótesis inductiva } P.n \rangle \\
 & n^2 + 2 \times (n+1) - 1 \\
= & \quad \langle \text{Aritmética } \rangle \\
 & (n+1)^2
\end{array}
$$

La prueba anterior, emplea una técnica que se utiliza a menudo: el lado izquierdo de $P(n+1)$ es manipulado de forma tal de "exponer" $P.n$, es decir, hacer posible el uso de la hipótesis de inducción $P.n$. Aquí, con la separación de término, se puso en evidencia a $P.n$.

El principio de inducción se formaliza en un axioma de nuestro cálculo de predicados como sigue, donde $P : \mathbb{N} \to \mathbb{B}$:

(10.1) Axioma. Inducción Matemática sobre $\mathbb{N}$:

$$(\forall\, n : \mathbb{N} :: (\forall\, i : 0 \leq i < n : P.i) \Rightarrow P.n) \Rightarrow (\forall\, n : \mathbb{N} :: P.n).$$

El consecuente en el axioma 10.1 trivialmente implica el antecedente, con lo cual podemos reescribir el axioma anterior utilizando Implicación mutua (3.79) de la siguiente manera:

(10.2) Teorema. Inducción Matemática sobre $\mathbb{N}$:

$$(\forall\, n : \mathbb{N} :: (\forall\, i : 0 \leq i < n : P.i) \Rightarrow P.n) \equiv (\forall\, n : \mathbb{N} :: P.n).$$

En general utilizaremos el axioma 10.2 en lugar del 10.1 para probar propiedades por inducción, dado que este axioma es más fácil de usar por ser el operador $\equiv$ simétrico, mientras que el operador $\Rightarrow$ no lo es.

El caso $P.0$ está incluido en el axioma 10.1. Veremos ahora que al manipular el antecedente el caso $P.0$ aparece.

$$
\begin{array}{ll}
 & (\forall\, n : 0 \leq n : (\forall\, i : 0 \leq i < n : P.i) \Rightarrow P.n) \\
= & \quad \langle \text{Separación de término 6.24} \rangle \\
 & ((\forall\, i : 0 \leq i < 0 : P.i) \Rightarrow P.0) \wedge (\forall\, n : 1 \leq n : (\forall\, i : 0 \leq i < n : P.i) \Rightarrow P.n) \\
= & \quad \langle \text{Rango vacío 6.9} \rangle \\
 & (true \Rightarrow P.0) \wedge (\forall\, n : 1 \leq n : (\forall\, i : 0 \leq i < n : P.i) \Rightarrow P.n) \\
= & \quad \langle \text{Neutro a izquierda de } \Rightarrow \text{ 3.72; Cambio de dummy 6.23; artimética} \rangle \\
 & P.0 \wedge (\forall\, n : 0 \leq n : (\forall\, i : 0 \leq i < n+1 : P.i) \Rightarrow P.(n+1)
\end{array}
$$

Entonces, podemos reescribir el axioma 10.1 de la siguiente forma, que es la forma que generalmente se usa cuando es necesario probar propiedades mediante inducción:

(10.3) Teorema. Inducción Matemática sobre $\mathbb{N}$:

$$P.0 \wedge (\forall\, n : \mathbb{N} : (\forall\, i : 0 \leq i \leq n : P.i) \Rightarrow P(n+1)) \equiv (\forall\, n : \mathbb{N} :: P.n).$$

La primera conjunción en el axioma 10.3 es el *caso base* de la inducción matemática, la segunda conjunción del antecedente:

$$(\forall\, n : \mathbb{N} : (\forall\, i : 0 \leq i \leq n : P.i) \Rightarrow P(n+1)) \tag{10.3}$$

es el *paso inductivo*, y $(\forall\, i : 0 \leq i \leq n : P.i)$ se llama *hipótesis de inducción*.

Para probar $(\forall\, n : \mathbb{N} :: P.n)$ por inducción, probamos el caso base y el paso inductivo por separado, y luego aseguramos en lenguaje corriente, que $P.n$ se satisface para todo número natural n. La prueba del paso inductivo, se hace probando $(\forall\, i : 0 \leq i \leq n : P.i) \Rightarrow P(n+1)$ para un $n \geq 0$ arbitrario. Más aún, $(\forall\, i : 0 \leq i \leq n : P.i) \Rightarrow P(n+1)$ se prueba usualmente suponiendo $(\forall\, i : 0 \leq i \leq n : P.i)$ y luego probando $P(n+1)$.

El axioma (10.1) y los teoremas (10.2) y (10.3), describen la técnica de inducción sobre todos los números naturales, sin embargo, la inducción puede realizarse sobre cualquier subconjunto $n_0, n_0+1, n_0+2, \ldots$ de $\mathbb{N}$. La única diferencia es el punto de partida, el caso base será $P.n_0$. Reformulamos entonces el principio de inducción matemática para estos casos del siguiente modo:

(10.4) Teorema. Inducción matemática sobre $\{n_0, n_0+1, \ldots\}$:

$$P.n_0 \wedge (\forall\, n : n_0 \leq n : (\forall\, i : n_0 \leq i \leq n : P.i) \Rightarrow P(n+1)) \equiv (\forall\, n : n_0 \leq n : P.n).$$

Veamos un ejemplo,

(10.5) Ejemplo. En este caso, queremos probar $(\forall\, n : 3 \leq n : P.n)$ donde

$$P.n : 2 \times n + 1 < 2^n.$$

Caso base

$P.3$ es $2 \times 3 + 1 < 2^3$, lo cual es cierto.

Paso inductivo

Para un valor arbitrario $n \geq 3$, probaremos $P.(n+1)$ usando la hipótesis de inducción $P.n$.

$$\begin{array}{ll} & 2^{n+1} \\ = & \quad \langle \text{aritmética} \rangle \\ & 2 \times 2^n \\ > & \quad \langle \text{hipótesis de inducción } P.n \rangle \\ & 2 \times (2 \times n + 1) \\ = & \quad \langle \text{aritmética} \rangle \\ & 2 \times (n+1) + 1 + 2 \times n - 1 \\ > & \quad \langle \text{aritmética } 2 \times n - 1 > 0, \text{ ya que } 3 \leq n \rangle \\ & 2 \times (n+1) + 1 \end{array}$$

Aquí, para probar el paso inductivo, transformamos el lado derecho de $P.(n+1)$ en el lado izquierdo. En vez de esto, podríamos haber transformado el lado izquierdo en el derecho o también $P.(n+1)$ en $true$.

Nuestro siguiente ejemplo, concierne a los números de *Fibonacci*. Estos números son interesantes cuando trabajamos con inducción matemática, pero son muy interesantes también por sí solos. Fueron introducidos por Leonardo Fibonacci (1202) a quien se le debe el nombre.

Los primeros 8 números de Fibonacci son $0, 1, 1, 2, 3, 5, 8, 13$, y a excepción de los primeros dos, los siguientes números se consiguen sumando los dos anteriores. Los números de Fibonacci, se definen recursivamente del siguiente modo:

$$\begin{array}{rcl} fib.\,0 & \doteq & 0 \\ fib.\,1 & \doteq & 1 \\ fib.(n+2) & \doteq & fib.(n+1) + fib.n \end{array}$$

Como ya dijimos, esta sucesión de números tiene propiedades muy interesantes, algunas de las cuales se presentan en los ejercicios. Vamos a demostrar ahora, una muy simple:

$$(\forall\, n : 0 \leq n : fib.n < 2^n)$$

Dada la definición de fib, parece conveniente considerar dos casos base, cuando $n = 0$ y cuando $n = 1$. El paso inductivo podrá considerarase entonces para $n \geq 2$.

Caso base

$$\begin{array}{ll} & fib.\,0 \\ = & \quad \langle \text{definición de } fib \rangle \\ & 0 \\ < & \quad \langle 0 < 1 \rangle \\ & 1 \\ = & \quad \langle 2^0 = 1 \rangle \\ & 2^0 \end{array}$$

El caso base cuando $n = 1$ es similar al anterior.

Paso inductivo

Sea $P.n : fib.n < 2^n$, vamos a partir de $fib.(n+2)$ y probaremos que es menor que $2^{(n+2)}$

$$
\begin{array}{ll}
& fib.(n+2) \\
= & \quad \langle \text{definición de } fib \rangle \\
& fib.n + fib.(n+1) \\
< & \quad \langle \text{hipótesis inductiva } P.n \rangle \\
& 2^n + fib.(n+1) \\
< & \quad \langle \text{hipótesis inductiva } P.(n+1) \rangle \\
& 2^n + 2^{(n+1)} \\
< & \quad \langle \text{la función } 2^x \text{ es creciente} \rangle \\
& 2^{(n+1)} + 2^{(n+1)} \\
= & \quad \langle \text{propiedad de la función exponencial} \rangle \\
& 2^{(n+2)}
\end{array}
$$

10.3. Ayudas para pruebas por inducción

El primer paso para probar una cuantificación universal por inducción es escribir la fórmula de la forma

$$(\forall\, n : n_0 \leq n : P.n)$$

Esto significa identificar n_0 y también la propiedad a probar: $P.n$, la cual tiene que ser de tipo booleano.

Por lo general, el paso inductivo

$$(\forall\, n : n_0 \leq n : (\forall\, i : n_0 \leq i \leq n : P.i) \Rightarrow P.(n+1))$$

se realiza probando $(\forall\, i : n_0 \leq i \leq n : P.i) \Rightarrow P.(n+1)$ para un arbitrario $n \geq n_0$. Y esto último, se hace suponiendo $P.n_0, \ldots, P.n$ y probando $P.(n+1)$. Esta clase de prueba, a menudo exige la manipulación de $P.(n+1)$ de alguna manera.

El objetivo de manipular $P.(n+1)$ es hacer posible el uso de las conjunciones $P.\,0, \ldots, P.n$ en la hipótesis inductiva, es decir poner en evidencia la hipótesis inductiva. Por ejemplo, en (10.5) reescribimos $2^{(n+1)}$ como 2×2^n, de modo de poder usar $P.n : 2 \times n + 1 < 2^n$. Mientras que en (10.1), obtuvimos $P.n$ mediante separación de término. En estos ejemplos pudimos probar $P.(n+1)$ usando solo la hipótesis $P.n$, pero en otros casos necesitaremos más de una hipótesis inductiva para probar $P.(n+1)$.

10.4. Verificación de programas

Utilizaremos el principio de inducción para comprobar que un programa (que en la programación funcional sería una definición de función) satisface su especificación. A ésto se lo llama *verificación de programas*.

También utilizaremos inducción para construir una definición de función a partir de su especificación. A esta técnica se la llama *derivación de programas*, y la veremos en el capítulo siguiente.

Si bien la técnica de verificación de programas es muy útil, ya que nos permite probar que un programa es correcto según su especificación, por lo general no es una tarea trivial, por lo que es conveniente realizarla, al menos en parte, a medida que se construye el programa. Esta construcción conjunta de programa y demostración tiene la ventaja adicional de que la misma nos va guiando en la construcción del programa.

Veamos un ejemplo en el que verificamos que la función $f : Nat \to Nat$ definida recursivamente como:

$$\begin{array}{rcl} f.0 & \doteq & 0 \\ f.(i+1) & \doteq & f.i + 2 \times i + 1 \end{array}$$

satisface la siguiente especificación:

$$\begin{array}{ll} pre: & true \\ post: & f.i = i^2 \end{array}$$

Primero definimos la propiedad a probar ($P.n$) a partir de las expresiones booleanas pre y $post$. Dado que la precondición es $true$, $P.n$ resulta:

$$P.n : f.n = n^2$$

Caso base

$$\begin{array}{ll} & f.0 \\ = & \quad \langle \text{definición de } f \rangle \\ & 0 \\ = & \quad \langle \text{aritmética} \rangle \\ & 0^2 \end{array}$$

Paso inductivo

Probaremos $P.(n+1)$ suponiendo la hipótesis de inducción $P.n$:

$$\begin{array}{ll} & f.(n+1) \\ = & \quad \langle \text{definición de } f \rangle \\ & f.n + 2 \times n + 1 \\ = & \quad \langle \text{hipótesis inductiva } \rangle \\ & n^2 + 2 \times n + 1 \\ = & \quad \langle \text{aritmética} \rangle \\ & (n+1)^2 \end{array}$$

De esta demostración concluímos que f satisface $(\forall\, n : \mathbb{N} :: f.n = n^2)$.

Veamos otro ejemplo de verificación de programas: sea fac la función que calcula el factorial de un número, que definimos recursivamente como:

$$\begin{array}{rcl} fac.0 & \doteq & 1 \\ fac.(n+1) & \doteq & (n+1) \times fact.n \end{array}$$

Vamos a probar que $fact$ satisface la siguiente especificación:

$$\begin{array}{ll} pre: & true \\ post: & fact.n = (\prod\ i : 1 \leq i \leq n : i) \end{array}$$

A partir de esta especificación, la propiedad a probar $P.n$ resulta:

$$P.n : fact.n = (\prod\ i : 1 \leq i \leq n : i)$$

Caso base

Podemos reducirlo asi:

$$fact.0 = (\prod\ i : 1 \leq i \leq 0 : i) = (\prod\ i : false : i) = 1$$

Paso inductivo

Probaremos $P.(n+1)$ suponiendo la hipótesis de inducción $P.n$:

$$\begin{array}{ll} & fact.(n+1) \\ = & \quad \langle \text{definición de factorial} \rangle \\ & (n+1) \times fact.n \\ = & \quad \langle \text{hipótesis inductiva} \rangle \\ & (n+1) \times (\prod\ i : 1 \leq i \leq n : i) \\ = & \quad \langle \text{Separación de término (6.24)} \rangle \\ & (\prod\ i : 1 \leq i \leq n+1 : i) \end{array}$$

Por lo tanto hemos probado que la definición dada de $fact$ satisface la especificación dada.

10.5. Inducción sobre listas

El principio de inducción puede también aplicarse a listas. Cuando aplicamos inducción sobre los números naturales, discriminamos dos casos: cuando el número es 0 y cuando es de la forma $n+1$ para algún natural n. De modo similar la inducción sobre listas se basa en dos casos: o bien la lista es vacía o bien es de la forma $x \triangleright xs$, para algún valor x y lista xs.

Por lo tanto, para demostrar por inducción que una propiedad $P.xs$ se cumple para cualquier lista xs, es suficiente demostrar que:

Caso base: $P.[\,]$ se satisface, y

Paso inductivo: Si $P.xs$ se satisface, entonces $P.(x \triangleright xs)$ vale para cualquier x.

Esto es válido por el mismo argumento que ya hemos utilizado para inducción sobre números naturales. Sabemos por el caso base que $P.[]$ se satisface, y entonces aplicando el paso inductivo $P.[x]$ también vale para cualquier x, (ya que $[x] = x \triangleright [\,]$), nuevamente, aplicando el paso inductivo $P.[y,x]$ vale para cualquier x e y, (ya que $[y,x] = y \triangleright [x]$), y así sucesivamente. Entonces

$P.xs$ vale para cualquier lista finita xs. Es fácil formalizar esta prueba por inducción sobre el tamaño de la lista xs, con lo cual el principio de inducción sobre listas, es una consecuencia directa del principio de inducción sobre los números naturales.

Al igual que en la inducción sobre naturales, en la inducción sobre listas el caso base puede ser una lista con n_0 elementos. En tal caso la hipótesis inductiva es $P.xs$, donde $\#xs \geq n_0$. Por ejemplo, el principio de inducción puede ser reformulado para listas con al menos un elemento como:

Caso base: $P.[x]$ se satisface para cualquier x, y

Paso inductivo: Si $P.(x \triangleright xs)$ se satisface, entonces $P.(x' \triangleright x \triangleright xs)$ vale para cualquier x'.

Aquí cuncluimos que $P.xs$ vale para cualquier lista finita con al menos un elemento.

En el Capítulo 8, enunciamos diversas propiedades sobre los operadores definidos sobre listas sin dar una demostración explícita. Vamos a probar ahora algunas de las propiedades citadas.

Por ejemplo, vamos a demostrar que la concatenación es asociativa:

$$xs +\!\!+ (ys +\!\!+ zs) = (xs +\!\!+ ys) +\!\!+ zs$$

para cada lista finita xs, ys y zs.

Si bien aparecen tres listas en esta propiedad, la demostración se hace por inducción sobre una de ellas: xs. Llamaremos $P.xs$ a la propiedad a probar:

$$P.xs : xs +\!\!+ (ys +\!\!+ zs) = (xs +\!\!+ ys) +\!\!+ zs$$

Caso base:

$$\begin{array}{ll} & [\,] +\!\!+ (ys +\!\!+ zs) \\ = & \quad \langle \text{ definición recursiva de } +\!\!+ \rangle \\ & ys +\!\!+ zs \\ = & \quad \langle \text{ definición recursiva de } +\!\!+ \rangle \\ & ([\,] +\!\!+ ys) +\!\!+ zs \end{array}$$

Paso inductivo

Suponemos que $P.xs$ es cierta para cualquier lista xs con $\#xs = n$, entonces probamos $P.(x \triangleright xs)$:

$$\begin{array}{ll} & (x \triangleright xs) +\!\!+ (ys +\!\!+ zs) \\ = & \quad \langle \text{ definición recursiva de } +\!\!+ \rangle \\ & x \triangleright (xs +\!\!+ (ys +\!\!+ zs)) \\ = & \quad \langle \text{ hipótesis inductiva, pues } \#xs = n \rangle \\ & x \triangleright ((xs +\!\!+ ys) +\!\!+ zs) \\ = & \quad \langle \text{ definición recursiva de } +\!\!+ \rangle \\ & x \triangleright (xs +\!\!+ ys) +\!\!+ zs \\ = & \quad \langle \text{ definición recursiva de } +\!\!+ \rangle \\ & ((x \triangleright xs) +\!\!+ ys) +\!\!+ zs \end{array}$$

Otra de las propiedades que vimos relaciona la longitud con la concatenación:

$$\#(xs +\!\!+ ys) = \#xs + \#ys$$

para cualquier par de listas finitas xs y ys.
Haremos la prueba por inducción sobre xs, con lo cual:

$$P.xs : \#(xs +\!\!+ ys) = \#xs + \#ys$$

Caso base:

$$\begin{array}{ll} & \#([\,] +\!\!+ ys) \\ = & \quad \langle \text{ definición recursiva de } +\!\!+ \rangle \\ & \#ys \\ = & \quad \langle \text{ elemento neutro de la suma} \rangle \\ & 0 + (\#ys) \\ = & \quad \langle \text{ definición recursiva de } \# \rangle \\ & (\#[\,]) + (\#ys) \end{array}$$

Paso inductivo

Suponemos que $P.xs$ es cierta para cualquier lista xs con $\#(xs) = n$, entonces probamos $P.(x \triangleright xs)$:

$$\begin{array}{ll} & \#((x \triangleright xs) +\!\!+ ys) \\ = & \quad \langle \text{ definición recursiva de } +\!\!+ \rangle \\ & \#(x \triangleright (xs +\!\!+ ys)) \\ = & \quad \langle \text{ definición recursiva de } \# \rangle \\ & 1 + (\#(xs +\!\!+ ys)) \\ = & \quad \langle \text{ hipótesis de inducción, pues } \#xs = n \rangle \\ & 1 + (\#xs) + (\#ys) \\ = & \quad \langle \text{ asociatividad de la suma y definición recursiva de } \# \rangle \\ & (\#(x \triangleright xs)) + (\#ys) \end{array}$$

10.6. Verificación de programas con listas

Utilizaremos inducción sobre listas también para la verificación de programas, es decir, probaremos que una función satisface cierta especificación por inducción.

Veamos un ejemplo, la función $map : (A \to B) \to [A] \to [B]$, la cual aplica una función a cada elemento de una lista (por ejemplo, $map.cuadrado.[3,5] = [9,25]$), puede definirse recursivamente de la siguiente manera:

$$\begin{array}{lcl} map.f.[] & \doteq & [] \\ map.f.(x \triangleright xs) & \doteq & f.x \triangleright map.f.xs \end{array}$$

Probaremos que esta definición satisface la siguiente especificación:

$$\begin{array}{ll} pre: & 0 \leq i < \#xs \\ post: & (map.f.xs).i = f.(xs.i) \end{array}$$

Primero derivamos la propiedad a probar, a partir de la especificación dada,

$$P.xs : 0 \leq i < \#xs \Rightarrow (map.f.xs).i = f.(xs.i)$$

Ahora, probamos que $P.xs$ es cierta para cualquier xs por inducción sobre listas.

Caso base:

$$\begin{array}{ll} & 0 \leq i < \#[] \Rightarrow (map.f.[]).i = f.([].i) \\ = & \quad \langle \text{ definición recursiva de } \# \rangle \\ & 0 \leq i < 0 \Rightarrow (map.f.[]).i = f.([].i) \\ = & \quad \langle \text{ aritmética } \rangle \\ & false \Rightarrow (map.f.[]).i = f.([].i) \\ = & \quad \langle \text{ 3.73 } \rangle \\ & true \end{array}$$

Paso inductivo

Suponemos que $P.xs$ es cierta para cualquier lista xs con $\#(xs) = n$, y probamos $P.(x \triangleright xs)$:

$$\begin{array}{ll} & 0 \leq i < \#(x \triangleright xs) \Rightarrow (map.f.(x \triangleright xs)).i = f.((x \triangleright xs).i) \\ = & \quad \langle \text{definición de } map \rangle \\ & 0 \leq i < \#(x \triangleright xs) \Rightarrow (f.x \triangleright map.f.xs).i = f.((x \triangleright xs).i) \\ = & \quad \langle \text{ aritmética y definición de } \# \rangle \\ & i = 0 \vee 1 \leq i < \#xs + 1 \Rightarrow (f.x \triangleright map.f.xs).i = f.((x \triangleright xs).i) \\ = & \quad \langle \text{ 3.76 } \rangle \\ & (i = 0 \Rightarrow (f.x \triangleright map.f.xs).i = f.((x \triangleright xs).i)) \wedge \\ & (1 \leq i < \#xs + 1 \Rightarrow (map.f.(x \triangleright xs)).i = f.((x \triangleright xs).i)) \end{array}$$

Hasta acá lo que hicimos fue dividir la prueba en dos, de manera que podamos probar cada parte de forma independiente. Tuvimos que hacer ésto para poder usar la definición recursiva de indexar (.), la cual devuelve resultados diferentes cuando se aplica sobre 0 y sobre cualquier natural mayor que 0.

Comenzaremos probando que el primer elemento de la conjunción es igual a $true$. Para ello supondremos que el antecedente $i = 0$ es cierto y probaremos el consecuente.

$$
\begin{array}{ll}
 & (f.x \triangleright map.f.xs).i \\
= & \quad \langle \text{ antecedente } \rangle \\
 & (f.x \triangleright map.f.xs).\,0 \\
= & \quad \langle \text{ definición de indexar } \rangle \\
 & f.x \\
= & \quad \langle \text{ definición de indexar} \rangle \\
 & f.((x \triangleright xs).\,0) \\
= & \quad \langle \text{ antecedente } \rangle \\
 & f.((x \triangleright xs).i)
\end{array}
$$

Ahora probaremos que $1 \leq i < \#xs+1 \Rightarrow (f.x \triangleright map.f.xs).i = f.((x \triangleright xs).i)$. Suponemos el antedecente $1 \leq i < \#xs + 1$, o equivalentemente $1 \leq i$ e $i < \#xs + 1$.

$$
\begin{array}{ll}
 & (f.x \triangleright map.f.xs).i = f.((x \triangleright xs).i) \\
= & \quad \langle \text{ definición de indexar, antecedente } 1 \leq i \rangle \\
 & (map.f.xs).(i-1) = f.(xs.(i-1)) \\
\Leftarrow & \quad \langle \text{ hipótesis inductiva, pues } \#xs = n \rangle \\
 & 0 \leq i - 1 < \#xs \\
= & \quad \langle \text{ aritmética } \rangle \\
 & 1 \leq i < \#xs + 1 \\
= & \quad \langle \text{ antecedente} \rangle \\
 & true
\end{array}
$$

Hemos probado que $true \Rightarrow (f.x \triangleright map.f.xs).i = f.((x \triangleright xs).i)$, aplicamos el teorema *elemento neutro a izquierda de* $\Rightarrow$ (3.71) para concluir $(f.x \triangleright map.f.xs).i = f.((x \triangleright xs).i)$.

10.7. Ejercicios

10.1 Probar las siguientes expresiones aritméticas por inducción sobre n

a) Para $n \geq 0$, $(\sum\, i : 1 \leq i \leq n : i) = n \times (n+1)/2$.

b) Para $n \geq 0$, $(\sum\, i : 0 \leq i < n : 2 \times i + 1) = n^2$.

c) Para $n \geq 0$, $(\sum\, i : 0 \leq i < n : 3^i) = (3^n - 1)/2$.

d) Para $n \geq 0$, $(\sum\, i : 0 \leq i \leq n : i^2) = n \times (n+1) \times (2 \times n + 1)/6$.

10.2 Probar por inducción sobre n que $4^n - 1$ es divisible por 3 cuando $n \geq 0$.

10.3 Probar por inducción sobre n que $n^2 \leq 2^n$ cuando $n \geq 4$.

10.4 Definimos la función f recursivamente así:

$$
\begin{array}{ll}
f.\,0 & \doteq \; 0 \\
f.(n+1) & \doteq \; 2 \times f.n + 1
\end{array}
$$

Probar por inducción que $f.n = 2^n - 1$ cuando $n \geq 0$.

10.5 Demostrar cual es el error en el siguiente razonamiento, que demuestra que un grupo cualquiera de gatos está compuesto sólo por gatos negros.

> La demostración es por inducción en el número de gatos de un grupo dado. Para el caso de 0 gatos el resultado es inmediato, dado que todo gato del grupo es negro. Consideremos ahora un grupo de $n+1$ gatos. Tomemos un gato cualquiera del grupo. El grupo de gatos restantes tiene n gatos, y por hipótesis inductiva son todos negros. Luego, tenemos n gatos negros y uno que no sabemos aún de que color es. Intercambiamos ahora el gato que sacamos con uno del grupo y un grupo de n gatos. Por hipótesis inductiva los n gatos son negros, pero como el que tenemos aparte también es negro, entonces tenemos que los $n+1$ gatos son negros.

10.6 Probar las siguientes propiedades de las operaciones sobre listas

a) $(xs \uparrow n) +\!\!+ (xs \downarrow n) = xs$

b) $(xs \downarrow n) \uparrow m = (xs \uparrow (m+n)) \downarrow n$

c) $(xs \downarrow n) \downarrow m = xs \downarrow (m+n)$

10.7 Probar las siguientes propiedades:

$$\begin{array}{lcll} (xs +\!\!+ ys) \cdot i & = & xs \cdot i & \text{si} \quad 0 \leq i < \#xs \\ (xs +\!\!+ ys) \cdot i & = & ys \cdot (i - \#xs) & \text{si} \quad i \geq \#xs \end{array}$$

10.8 Consideremos las definiciones de las funciones vistas en el Capítulo 8:

i) $hd.(x \triangleright xs) \doteq x$

ii) $tl.(x \triangleright xs) \doteq xs$

iii)
$$\begin{array}{lcl} init.[x] & \doteq & [\,] \\ init.(x \triangleright x' \triangleright xs) & \doteq & x \triangleright (init.(x' \triangleright xs)) \end{array}$$

iv)
$$\begin{array}{lcl} last.[x] & \doteq & x \\ last.(x \triangleright x' \triangleright xs) & \doteq & last.(x' \triangleright xs) \end{array}$$

Demostrar las siguientes propiedades:

a) $init.xs = xs \uparrow (\#xs - 1)$

b) $init.(xs +\!\!+ [x]) = xs$

c) $last.(xs +\!\!+ [x]) = x$

10.9 Probar que la siguiente definición de la función $split : Nat \to [A] \to ([A], [A])$:

$$\begin{array}{lcl} split.\,0.xs & \doteq & ([\,], xs) \\ split.(n+1).[\,] & \doteq & ([\,], [\,]) \\ split.(n+1).(x \triangleright xs) & \doteq & (x \triangleright ys, zs) \\ & & [\![\, (ys, zs) \doteq split.n.xs \,]\!] \end{array}$$

satisface la especificación:

$$\begin{array}{ll} pre: & true \\ post: & split.n.xs = (xs \uparrow n, xs \downarrow n) \end{array}$$

10.10 Probar que la siguiente definición recursiva de la función $sum : [Int] \to Int$:

$$\begin{array}{lcl} sum.[\,] & \doteq & 0 \\ sum.(x \triangleright xs) & \doteq & x + sum.xs \end{array}$$

satisface la especificación encontrada en el capítulo 10.

CAPÍTULO 11
Derivación de programas

Índice del Capítulo

11.1. Introducción 177
11.2. Derivación de funciones recursivas 177
11.3. Generalización por abstracción 181
11.4. Modularización 184
11.5. Ejercicios 186

11.1. Introducción

Si bien un paso importante en la solución de un problema es escribir una especificación correcta que describa el problema, una vez hallada ésta, es necesario manipularla adecuadamente de modo de transformarla en la implementación de un programa. Veremos a continuación técnicas que nos permitirán llevar a cabo este objetivo.

11.2. Derivación de funciones recursivas

Veremos cómo obtener una definición recursiva de una función a partir de especificación aplicando inducción.

Este tipo de derivación de programas consiste en definir una función pudiendo usar para ello los valores de la función aplicada a elementos con menor estructura. Por ejemplo, en el caso de los números, la función utilizará valores de la función sobre números más pequeños, y en el caso de listas utilizará valores de la función sobre listas de longitud menor.

(11.1) **Ejemplo.** Veamos un ejemplo sencillo de derivación. Dado un número natural se desea obtener una función que calcule su factorial. Primero daremos la especificación de esta función ($fac : Nat \to Nat$):

$$\begin{array}{ll} pre: & true \\ post: & fac.n = (\prod\ i : 0 < i \leq n : i) \end{array}$$

Para hallar una definición recursiva que satisfaga esta especificación podemos pensar que la especificación de fac es una ecuación a resolver con incógnita fac . Luego el proceso de derivación consistirá en "despejar" fac. El proceso de despejar nos asegura que si ahora reemplazamos el valor obtenido en la ecuación original (la especificación) obtenemos un enunciado verdadero. Veamos cómo sería esto.

Caso base

$$\begin{array}{ll} & fac.0 \\ = & \quad \langle \text{especificación de } f \rangle \\ & (\prod\ i : 0 < i \leq 0 : i) \\ = & \quad \langle \text{rango vacío} \rangle \\ & 1 \end{array}$$

Paso inductivo

$$\begin{array}{ll} & fac.(n+1) \\ = & \quad \langle \text{especificación de } f \rangle \\ & (\prod\ i : 0 < i \leq n+1 : i) \\ = & \quad \langle \text{separación de un término} \rangle \\ & (\prod\ i : 0 < i \leq n : i) \times (n+1) \\ = & \quad \langle \text{hipótesis inductiva} \rangle \\ & fac.n \times (n+1) \end{array}$$

Por lo tanto definiendo recursivamente fac como:

$$\begin{array}{rcl} fac.0 & \doteq & 1 \\ fac.(n+1) & \doteq & (n+1) \times fac.n \end{array}$$

se satisface la especificación dada.

Además de haber construído un programa que satisface una especificación hemos obtenido una demostración de que lo hace. Esta demostración no es más que la anterior invirtiendo los pasos de atrás para adelante, a partir del segundo paso de la derivación y cambiando la justificación "especificación de fac" por "definición de fac". Veamos otro ejemplo.

(11.2) **Ejemplo.** Dada una lista de enteros se desea obtener una función que calcule la suma de sus elementos. Primero especificamos formalmente esta función y definimos su tipo: $[Int] \rightarrow Int$.

$$\begin{array}{ll} pre: & true \\ post: & sum.xs = (\sum\, i : 0 \leq i \leq \#xs : xs.i) \end{array}$$

A partir de esta especificación calcularemos una definición recursiva haciendo inducción sobre la lista.

Caso base $xs = [\,]$

$$\begin{array}{ll} & sum.[\,] \\ = & \quad \langle \text{especificación } sum \rangle \\ & (\sum\, i : 0 \leq i \leq \#[\,] : [\,].i) \\ = & \quad \langle \text{definición de} \#\text{; rango vacío} \rangle \\ & 0 \end{array}$$

En esta prueba podemos ver la diferencia entre una derivación y una verificación. El objetivo de la verificación es llegar a la fórmula (para el caso que se está probando), mientras que el objetivo de la derivación es simplificar esta fórmula hasta obtener una expresión del formalismo básico que pueda tomarse como definición, en este caso la constante 0.

Consideremos ahora el paso inductivo,

Paso inductivo

$$\begin{array}{ll} & sum.(x \triangleright xs)) \\ = & \quad \langle \text{especificación de } sum \rangle \\ & (\sum\, i : 0 \leq i \leq \#(x \triangleright xs) : (x \triangleright xs).i) \\ = & \quad \langle \text{definición de } \# \rangle \\ & (\sum\, i : 0 \leq i \leq 1 + \#xs : (x \triangleright xs).i) \\ = & \quad \langle \text{separación de un término} \rangle \\ & (x \triangleright xs).0 \;+\; (\sum\, i : 1 \leq i \leq 1 + \#xs : (x \triangleright xs).i) \\ = & \quad \langle \text{cambio de variable dummy (f.i=i+1); definición de .} \rangle \\ & x \;+\; (\sum\, j : 1 \leq j + 1 \leq 1 + \#xs : (x \triangleright xs).(j+1)) \\ = & \quad \langle \text{definición de indexar, aritmética} \rangle \\ & x \;+\; (\sum\, j : 0 \leq j \leq \#xs : xs.j) \\ = & \quad \langle \text{hipótesis inductiva} \rangle \\ & x \;+\; sum.xs \end{array}$$

Por lo tanto, sum puede definirse recursivamente como:

$$\begin{array}{lcl} sum.[\,] & \doteq & 0 \\ sum.(x \triangleright xs) & \doteq & x + sum.xs \end{array}$$

(11.3) **Ejemplo.** En el capítulo 10 dimos una especificación de la función ev que evalúa un polinomio en un valor dado. Para ello, decidimos representar a los polinomios como listas de números $[a_0, \ldots, a_n]$, donde cada a_i representa un coeficiente del polinomio $a^n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$.

Derivaremos una definición de esta función a partir de la especificación dada:

$$\begin{array}{ll} pre: & \#xs > 0 \\ post: & ev.xs.y = (\sum\ i : 0 \leq i < \#xs : xs.i \times y^i) \end{array}$$

Caso base: $xs = [x]$

$$\begin{array}{ll} & ev.\,[x].\,y \\ = & \quad \langle \text{especificación de } ev \rangle \\ & (\sum\ i : 0 \leq i < \#[x] : [x].\,i \times y^i) \\ = & \quad \langle \text{definición de \#;rango unitario} \rangle \\ & [x].\,0 \times y^0 \\ = & \quad \langle \text{aritmética; definición de .} \rangle \\ & x \end{array}$$

Paso inductivo: $(z \triangleright x \triangleright xs)$

$$\begin{array}{ll} & ev.\,(z \triangleright x \triangleright xs).\,y \\ = & \quad \langle \text{especificación de } ev \rangle \\ & (\sum\ i : 0 \leq i < \#(z \triangleright x \triangleright xs) : (z \triangleright x \triangleright xs).\,i \times y^i) \\ = & \quad \langle \text{definición de \#; separación de un término} \rangle \\ & (z \triangleright x \triangleright xs).\,0 \times y^0 + (\sum\ i : 1 \leq i < 1 + \#(x \triangleright xs) : (z \triangleright x \triangleright xs).\,i \times y^i) \\ = & \quad \langle \text{definición de indexar; cambio de variable dummy (f.i=i+1); aritmética} \rangle \\ & z \times y^0 + (\sum\ j : 0 \leq j < \#(x \triangleright xs) : (z \triangleright x \triangleright xs).\,(j+1) \times y^{j+1}) \\ = & \quad \langle \text{definición de indexar; aritmética} \rangle \\ & z + (\sum\ j : 0 \leq j < \#(x \triangleright xs) : (x \triangleright xs).\,j \times y \times y^j) \\ = & \quad \langle \text{distributividad de } \times \text{ respecto de } + \rangle \\ & z + y \times (\sum\ j : 0 \leq j < \#(x \triangleright xs) : (x \triangleright xs).i \times y^i) \\ = & \quad \langle \text{hipótesis inductiva } \rangle \\ & z + y \times ev.\,(x \triangleright xs).\,y \end{array}$$

Por lo tanto una definición recursiva para ev es:

$$\begin{array}{lll} ev & :: & [Real] \to Real \to Real \\ ev.\,[x].\,y & \doteq & x \\ ev.\,(z \triangleright x \triangleright xs).\,y & \doteq & z + y \times ev.(x \triangleright xs).y \end{array}$$

11.3. Generalización por abstracción

En la sección anterior hemos analizado ejemplos simples de derivación de programas. Si bien no hemos tenido inconvenientes en el proceso de derivación de estos programas, no siempre podremos construir un programa a partir de la especificación utilizando la técnica dada, encontraremos ejemplos en los cuales la hipótesis inductiva no puede aplicarse de manera directa. Una técnica utilizada para resolver este tipo de derivaciones es la *generalización por abstracción*. La idea de ésta consiste en buscar una especificación más general que la dada y que pueda derivarse en forma directa. La función obtenida tendrá como caso particular la función que se desea encontrar. Para encontrar la generalización adecuada se introducirán parámetros nuevos a la función.

Veamos un ejemplo.

(11.4) Ejemplo. Supogamos que queremos hallar la definición recursiva de una función que determina si todas las sumas parciales de una lista son mayores o iguales a 0. La especificación de esta función está dada por:

$$\begin{array}{ll} pre: & true \\ post: & p.xs = (\forall\, i : 1 \leq i \leq \#xs : sum.(xs \uparrow i) \geq 0) \end{array}$$

donde $p : [Int] \to Bool$. Derivemos ahora una definición recursiva para p, haciendo inducción sobre la lista.

Caso base: $xs = [\,]$

$$\begin{array}{ll} & p.[\,] \\ = & \quad \langle \text{especificación de } p \rangle \\ & (\forall\, i : 1 \leq i \leq \#[\,] : sum.([\,] \uparrow i) \geq 0) \\ = & \quad \langle \text{definición de } \#\text{; rango vacío} \rangle \\ & true \end{array}$$

Paso inductivo: $x \triangleright xs$

$$\begin{array}{ll} & p.(x \triangleright xs) \\ = & \quad \langle \text{especificación de } p \rangle \\ & (\forall\, i : 1 \leq i \leq \#(x \triangleright xs) : sum.((x \triangleright xs) \uparrow i) \geq 0) \\ = & \quad \langle \text{definición de } \#\text{; separación de un término} \rangle \\ & sum.((x \triangleright xs) \uparrow 1) \geq 0 \ \wedge\ (\forall\, i : 2 \leq i \leq 1 + \#xs : sum.((x \triangleright xs) \uparrow i) \geq 0) \\ = & \quad \langle \text{definición de } \uparrow \text{; cambio de variable dummy (f.i=i+1)} \rangle \\ & sum.[x] \geq 0 \ \wedge\ (\forall\, j : 2 \leq j + 1 \leq 1 + \#xs : sum.((x \triangleright xs) \uparrow (j+1)) \geq 0) \\ = & \quad \langle \text{definición de } sum\text{; aritmética} \rangle \\ & x \geq 0 \ \wedge\ (\forall\, j : 1 \leq j \leq \#xs : sum.((x \triangleright xs) \uparrow (j+1)) \geq 0) \\ = & \quad \langle \text{definición de } \uparrow \rangle \\ & x \geq 0 \ \wedge\ (\forall\, j : 1 \leq j \leq \#xs : sum.(x \triangleright (xs \uparrow j)) \geq 0) \\ = & \quad \langle \text{definición de } sum \rangle \\ & x \geq 0 \ \wedge\ (\forall\, j : 1 \leq j \leq \#xs : x + sum.(xs \uparrow j) \geq 0) \end{array}$$

En este punto notamos que podría aplicarse la hipótesis inductiva de no ser por la x que esta sumando. No parece haber ninguna forma de eliminar esta x por lo cual se propone derivar la definición de una función más general que p. Llamaremos a esta función generalizada gp : $Int \to [Int] \to Bool$ y la especificaremos de la siguiente forma.

$$\begin{array}{ll} pre: & true \\ post: & gp.n.xs = (\forall\, i : 1 \leq i \leq \#xs : n + sum.(xs \uparrow i) \geq 0) \end{array}$$

Esta nueva especificación está inspirada en la última expresión de la derivación de p . Que esta nueva derivación pueda llevarse adelante dependerá de las propiedades del dominio en cuestión y puede no llegar a buen puerto o tener que volver a generalizarse a su vez. La programación es una actividad creativa y esto se manifiesta en este caso en la elección de las posibles generalizaciones.

Antes de comenzar la derivación de gp, debe determinarse si efectivamente gp generaliza a p, o lo que es lo mismo, si podemos definir p en términos de gp. En el ejemplo, ésto se cumple dado que $p.xs = gp.\,0.xs$, por lo tanto derivaremos gp directamente.

El caso base es similar al de p, el resultado es $true$ por la aplicación de rango vacío sobre el cuantificador.

Paso inductivo: $x \triangleright xs$

$$\begin{array}{ll} & gp.\,n.(x \triangleright xs) \\ = & \quad \langle \text{especificación de } gp \rangle \\ & (\forall\, i : 1 \leq i \leq \#(x \triangleright xs) : n + sum.\,((x \triangleright xs) \uparrow i) \geq 0) \\ = & \quad \langle \text{definición de } \#\text{; separación de un término} \rangle \\ & n + sum.\,((x \triangleright xs) \uparrow 1) \geq 0 \ \wedge\ (\forall\, i : 2 \leq i \leq 1 + \#xs : n + sum.\,((x \triangleright xs) \uparrow i) \geq 0) \\ = & \quad \langle \text{definición de } \uparrow \text{; cambio de variable dummy (f.i=i+1)} \rangle \\ & n + sum.\,[x] \geq 0 \ \wedge\ (\forall\, j : 2 \leq j + 1 \leq 1 + \#xs : n + sum.\,((x \triangleright xs) \uparrow (j+1)) \geq 0) \\ = & \quad \langle \text{definición de } sum\text{; aritmética} \rangle \\ & n + x \geq 0 \ \wedge\ (\forall\, j : 1 \leq j \leq \#xs : n + sum.\,((x \triangleright xs) \uparrow (j+1)) \geq 0) \\ = & \quad \langle \text{definición de } \uparrow \rangle \\ & n + x \geq 0 \ \wedge\ (\forall\, j : 1 \leq j \leq \#xs : n + sum.\,(x \triangleright (xs \uparrow j)) \geq 0) \\ = & \quad \langle \text{definición de } sum \rangle \\ & n + x \geq 0 \ \wedge\ (\forall\, j : 1 \leq j \leq \#xs : n + (x + sum.\,(xs \uparrow j)) \geq 0) \\ = & \quad \langle \text{asociatividad de } + \text{; hipótesis inductiva} \rangle \\ & n + x \geq 0 \ \wedge\ gp.(x+n).xs \end{array}$$

El resultado completo de la derivación es el siguiente programa:

$$\begin{array}{lcl} p.xs & \doteq & gp.\,0.xs \\ gp.n.[\,] & \doteq & True \\ gp.n.(x \triangleright xs) & \doteq & n + x \geq 0 \ \wedge\ gp.(n+x).xs \end{array}$$

Veamos otro ejemplo:

(11.5) **Ejemplo.** En el capítulo 10 especificamos la función $bal : [Bool] \to Bool$, que determina si la lista que recibe como argumento contiene igual cantidad de elementos $True$ que $False$.

$$\begin{array}{ll} pre: & true \\ post: & bal.xs = ((N\, i : 0 \leq i < \#xs : xs.i) = (N\, i : 0 \leq i < \#xs : \neg xs.i)) \end{array}$$

Derivemos una función recursiva para bal haciendo inducción sobre xs.

Caso base: $xs = [\,]$

$$\begin{array}{ll} & bal.[\,] \\ = & \quad \langle \text{especificación de } bal \rangle \\ & ((N\, i : 0 \leq i < \#[\,] : [\,].i) = (N\, i : 0 \leq i < \#[\,] : \neg[\,].i)) \\ = & \quad \langle \text{definición de } \#\text{; rango vacío} \rangle \\ & 0 = 0 \\ = & \quad \langle \text{igualdad de enteros} \rangle \\ & true \end{array}$$

Paso inductivo: $x \triangleright xs$

$$\begin{array}{ll} & bal.(x \triangleright xs) \\ = & \quad \langle \text{especificación de } bal \rangle \\ & ((N\, i : 0 \leq i < \#(x \triangleright xs) : (x \triangleright xs).i) = \\ & (N\, i : 0 \leq i < \#(x \triangleright xs) : \neg(x \triangleright xs).i)) \\ = & \quad \langle \text{definición de } \#\text{; aritmética} \rangle \\ & ((N\, i : i = 0 \vee 1 \leq i < \#xs + 1 : (x \triangleright xs).i) = \\ & ((N\, i : i = 0 \vee 1 \leq i < \#xs + 1 : \neg(x \triangleright xs).i) = \\ = & \quad \langle \text{partición de rango} \rangle \\ & (N\, i : i = 0 : (x \triangleright xs).i) + (N\, i : 1 \leq i < \#xs + 1 : (x \triangleright xs).i) = \\ & (N\, i : i = 0 : \neg(x \triangleright xs).i) + (N\, i : 1 \leq i < \#xs + 1 : \neg(x \triangleright xs).i) = \\ = & \quad \langle \text{definición de } N\text{; cambio de variable dummy (f.i=i+1); aritmética} \rangle \\ & (+\, i : i = 0 \wedge (x \triangleright xs).i : 1) + (N\, j : 0 \leq j < \#xs : (x \triangleright xs).(j+1)) = \\ & (+\, i : i = 0 \wedge \neg(x \triangleright xs).i : 1) + (N\, j : 0 \leq j < \#xs : \neg(x \triangleright xs).(j+1)) = \\ = & \quad \langle \text{leibniz; definición de indexar; aritmética} \rangle \\ & (+\, i : i = 0 \wedge x : 1) + (N\, j : 0 \leq j < \#xs : xs.j) = \\ & (+\, i : i = 0 \wedge \neg x : 1) + (N\, j : 0 \leq j < \#xs : \neg xs.j) = \end{array}$$

Para continuar con la derivación necesitamos saber el valor de x. Por lo tanto, dividiremos la prueba en dos casos: trataremos primero el caso en que el primer elemento de la lista es $true$ y luego el caso en que es $false$. La función derivada quedará definida por análisis por casos.

Caso $x = true$

$$
\begin{array}{ll}
 & bal.\,(true \triangleright xs) \\
= & \quad \langle \text{prueba anterior} \rangle \\
 & (+\,i : i = 0 \wedge true : 1) + (N\,j : 0 \leq j < \#xs : xs.j) = \\
 & (+\,i : i = 0 \wedge \neg true : 1) + (N\,j : 0 \leq j < \#xs : \neg xs.j) = \\
= & \quad \langle \text{neutro de } \wedge \text{ ; rango unitario} \rangle \\
 & 1 + (N\,j : 0 \leq j < \#xs : xs.j) = \\
 & (+\,i : i = 0 \wedge \neg true : 1) + (N\,j : 0 \leq j < \#xs : \neg xs.j) = \\
= & \quad \langle \text{definición de } false\text{; rango vacío; aritmética} \rangle \\
 & 1 + (N\,j : 0 \leq j < \#xs : xs.j) = \\
 & (N\,j : 0 \leq j < \#xs : \neg xs.j) =
\end{array}
$$

Aquí notamos que no es posible aplicar la hipótesis a raíz del sumando 1. Para el caso en que $x = false$, tenemos el mismo problema, aparece un sumando en el lado derecho de la igualdad. Decidimos entonces definir una función más general $gbal : Int \to [Bool] \to Bool$, cuya especificación es:

$$
\begin{array}{ll}
pre: & true \\
post: & gbal.y.xs = (y + (N\,i : 0 \leq i < \#xs : xs.i) = (N\,i : 0 \leq i < \#xs : \neg xs.i))
\end{array}
$$

Notemos que es suficiente agregar una variable de tipo entera, ya que en el caso $x = false$ pasaremos restando el valor 1 a la izquierda de la igualdad para poder aplicar la hipótesis.

La derivación de $gbal$ es similar a la de bal y la dejamos como ejercicio. El resultado obtenido de la derivación será el siguiente:

$$
\begin{array}{lcll}
bal.xs & \doteq & gbal.\,0.xs & \\
gbal.y.[\,] & \doteq & y = 0 & \\
gbal.y.(x \triangleright xs) & \doteq & (\ x = True & \to \quad gbal.\,1.xs \\
 & & \Box\ x = False & \to \quad gbal.(-1).xs \\
 & &) &
\end{array}
$$

11.4. Modularización

Una técnica muy utilizada en la construcción de programas que ahora utilizaremos en la derivación de los mismos es la *modularización*. Esta técnica se utiliza cuando la solución de un problema requiere la solución de un "subproblema", y consiste en no atacar ambos problemas simultáneamente sino por módulos, donde cada módulo es independiente del otro.

Veamos un ejemplo.

(11.6) **Ejemplo.** Queremos derivar la definición de una función $g :: [Int] \to [Int] \to Bool$ que toma 2 listas no vacías y determina si todos los elementos de la primer lista son mayores al mínimo valor de la segunda lista.

Especificamos esta función de la siguiente manera:

$$\begin{array}{ll} pre: & \#xs > 0 \ \wedge \ \#ys > 0 \\ post: & g.xs.ys = (\forall\, i : 0 \leq i < \#xs : xs.i > (min\, j : 0 \leq j < \#ys : ys.j)) \end{array}$$

A partir de esta especificación derivamos g, haciendo inducción sobre xs:

Caso base: $xs = [x]$

$$\begin{array}{ll} & g.\,[x].\,ys \\ = & \quad \langle \text{especificación de } g \rangle \\ & (\forall\, i : 0 \leq i < \#[x] : [x].i > (min\, j : 0 \leq j < \#ys : ys.j)) \\ = & \quad \langle \text{definición de } \#\text{; rango unitario} \rangle \\ & [x].\,0 > (min\, j : 0 \leq j < \#ys : ys.j) \\ = & \quad \langle \text{definición de .} \rangle \\ & x > (min\, j : 0 \leq j < \#ys : ys.j) \end{array}$$

Paso inductivo: $(y \triangleright x \triangleright xs)$

$$\begin{array}{ll} & g.\,(y \triangleright x \triangleright xs).ys \\ = & \quad \langle \text{especificación de } g \rangle \\ & (\forall\, i : 0 \leq i < \#(y \triangleright x \triangleright xs) : (y \triangleright x \triangleright xs).i > (min\, j : 0 \leq j < \#ys : ys.j)) \\ = & \quad \langle \text{definición de } \#\text{; separación de un término} \rangle \\ & (y \triangleright x \triangleright xs).\,0 > (min\, j : 0 \leq j < \#ys : ys.j) \wedge \\ & (\forall\, i : 1 \leq i < \#(x \triangleright xs) + 1 : (y \triangleright x \triangleright xs).i > (min\, j : 0 \leq j < \#ys : ys.j)) \\ = & \quad \langle \text{cambio de variable dummy (f.i=i+1); aritmética; definición de .} \rangle \\ & y > (min\, j : 0 \leq j < \#ys : ys.j) \wedge \\ & (\forall\, k : 0 \leq k < \#(x \triangleright xs) : (y \triangleright x \triangleright xs).(k+1) > (min\, j : 0 \leq j < \#ys : ys.i)) \\ = & \quad \langle \text{definición de .} \rangle \\ & y > (min\, j : 0 \leq j < \#ys : ys.j) \wedge \\ & (\forall\, k : 0 \leq k < \#(x \triangleright xs) : (x \triangleright xs).k > (min\, j : 0 \leq j < \#ys : ys.i)) \\ = & \quad \langle \text{hipótesis inductiva} \rangle \\ & y > (min\, j : 0 \leq j < \#ys : ys.j) \ \wedge \ g.(x \triangleright xs).ys \end{array}$$

Para obtener una definición de g agregaremos una definición de función que calcule el mínimo valor de una lista. Llamamos a esta nueva función m y la especificamos de la siguiente manera:

$$\begin{array}{ll} pre: & \#xs > 0 \\ post: & m.xs = (min\, i : 0 \leq i < \#xs : xs.i) \end{array}$$

La independencia de los módulos significa que una vez definida m, debe quedar definida g. Derivamos ahora la función m:

Caso base: $xs = [x]$

$$
\begin{array}{ll}
 & m.[x] \\
= & \quad \langle \text{especificación de } m \rangle \\
 & (min\, i : 0 \leq i < \#[x] : [x].i) \\
= & \quad \langle \text{definición de } \#\text{; rango unitario} \rangle \\
 & [x].\, 0 \\
= & \quad \langle \text{definición de .} \rangle \\
 & x
\end{array}
$$

Paso inductivo: $(y \triangleright x \triangleright xs)$

$$
\begin{array}{ll}
 & m.\,(y \triangleright x \triangleright xs) \\
= & \quad \langle \text{especificación de } m \rangle \\
 & (min\, i : 0 \leq i < \#(y \triangleright x \triangleright xs) : (y \triangleright x \triangleright xs).i) \\
= & \quad \langle \text{definición de } \#\text{; separación de un término} \rangle \\
 & min((y \triangleright x \triangleright xs).\, 0, (min\, i : 1 \leq i < \#(x \triangleright xs) + 1 : (y \triangleright x \triangleright xs).i)) \\
= & \quad \langle \text{definición de indexar; cambio de variable dummy (f.i=i+1); aritmética} \rangle \\
 & min(y, (min\, j : 1 \leq j + 1 < \#(x \triangleright xs) + 1 : (y \triangleright x \triangleright xs).(j + 1)) \\
= & \quad \langle \text{definición de indexar; aritmética } \rangle \\
 & min(y, (min\, j : 0 \leq j < \#(x \triangleright xs) : (x \triangleright xs).j)) \\
= & \quad \langle \text{hipótesis inductiva} \rangle \\
 & min(y, m.(x \triangleright xs))
\end{array}
$$

La definición recursiva de g es :

$$
\begin{array}{lcl}
g.[x].(y \triangleright ys) & \doteq & x > m.(y \triangleright ys) \\
g.(z \triangleright x \triangleright ys).(y \triangleright ys) & \doteq & z > m.(y \triangleright ys) \ \wedge \ g.(x \triangleright ys).(y \triangleright ys)
\end{array}
$$

$$
\begin{array}{lcl}
m.[x] & \doteq & x \\
m.(x \triangleright xs) & \doteq & (\quad x \geq m.xs \ \rightarrow \ x \\
 & & \Box \ \ x < m.xs \ \rightarrow \ m.xs \\
 & &)
\end{array}
$$

11.5. Ejercicios

11.1 Derivar una definición recursiva para la función $f : Nat \rightarrow Nat$, que satisface la siguiente especificación:

$$
\begin{array}{ll}
pre: & true \\
post: & f.n = (\sum\, i : 0 \leq i \leq n : 2^i + 1)
\end{array}
$$

11.2 Derivar una definición recursiva para cada una de las especificaciones dadas de la función $iguales : [A] \longrightarrow Bool$, la cual determina si los elementos de una lista dada son todos iguales entre sí. Comparar los resultados obtenidos en cada caso.

a) $pre: \quad \#xs > 0$
$post: \quad iguales.xs = (\forall\, i : 0 \leq i < \#xs - 1 : xs.i = xs.(i+1))$

b) $pre: \quad \#xs > 0$
$post: \quad iguales.xs = (\forall\, i : 0 \leq i < \#xs : xs.i = xs.\,0)$

c) $pre: \quad \#xs > 0$
$post: \quad iguales.xs = (\forall\, i : 0 \leq i < \#xs : xs.i = last.xs)$

11.3 Derivar una definición recursiva de la función $creciente : [Int] \longrightarrow Bool$, que dada una lista no vacía de enteros, determina si los elementos de la lista están ordenados en forma creciente.

11.4 Derivar una definición recursiva de la función $p : [Nat] \longrightarrow Bool$, que dada una lista no vacía de naturales, determina si algún elemento de la lista es igual a la suma de los demás elementos de la misma.

11.5 Derivar una definición recursiva de la función $f : Int \longrightarrow [Int] \longrightarrow Bool$, que determina si el k-ésimo elemento de una lista dada de enteros aloja el mínimo valor de la misma.

11.6 Derivar una definición recursiva para la función $consec\_dif : [Int] \rightarrow Bool$, la cual recibe una lista con al menos dos elementos y determina si para todo par de elementos consecutivos de la lista se cumple que la diferencia entre ambos valores es igual al índice de la primera posición considerada. Por ejemplo, $consec\_dif.[1,1,2,4,7] = True$, mientras que $consec\_dif.[1,1000,1001] = False$.

11.7 Derivar una función recursiva para la función $g : [Int] \rightarrow Int$, que toma una lista de enteros y devuelve el producto de la diferencia de cada uno de sus elementos con la longitud de la lista, satisfaciendo la siguiente especificación:

$$\begin{array}{ll} pre: & \#xs > 0 \\ post: & g.xs = (\prod i : 0 \leq i \leq \#xs : xs.i - \#xs) \end{array}$$

11.8 En la sección 11.3 se derivó una definición recursiva para la función $gbal : [Bool] \rightarrow Bool$ usando la técnica generalización por abstracción. Otra manera de derivar una definición para esta función es utilizando la técnica modularización, donde se distinguen dos módulos independientes:

$$\begin{array}{ll} pre: & true \\ post: & cantTrue.xs = (N\, i : 0 \leq i < \#xs : xs.i) \end{array}$$

$$\begin{array}{ll} pre: & true \\ post: & cantFalse.xs = (N\, i : 0 \leq i < \#xs : \neg xs.i) \end{array}$$

Una vez que se obtienen las definiciones de estas 2 funciones se puede definir $gbal$ como:

$$gbal.xs \doteq (cantTrue.xs = cantFalse.xs)$$

Completar esta definición con las definiciones de $cantTrue$ y $cantFalse$ obtenidas por derivación a partir de sus especificaciones.

CAPÍTULO 12
La Programación Imperativa

Índice del Capítulo

12.1. Introducción 189
12.2. Especificaciones de programas 189
12.2.1. Representación de variables iniciales y finales 191
12.3. Leyes sobre tripletas 192
12.4. El transformador de predicados wp 194
12.5. Ejercicios 194

12.1. Introducción

A partir de ahora cambiaremos el modelo de computación subyacente, y por lo tanto el formalismo para expresar los programas. Esto no significa que lo hecho hasta ahora no nos será útil en lo que sigue, ya que la experiencia adquirida en el cálculo de programas, y la consecuente habilidad para el manejo de fórmulas, seguirá siendo escencial en este caso, mostrando así que pese a sus diferencias, la programación funcional y la imperativa tienen varias cosas en común.

12.2. Especificaciones de programas

Los problemas que se presentan a un programador están, en general, expresados de manera informal, lo cual resulta demasiado impreciso a la hora de programar. Por eso es necesario precisar los problemas por medio de especificaciones formales. En la primera parte del curso se especificaba, usando lógica y un cálculo de funciones, el valor que debía calcularse. Este tipo de especificaciones era adecaudo para el desarrollo de programas funcionales. En el estilo de programación imperativa, la computación no se expresa como cálculo de valores sino como modificación de estados. Para desarrollar programas imperativos es por lo tanto deseable expresar las especificaciones como relaciones entre estados iniciales y finales.

Así es que un programa en el modelo de la programación imperativa estará especificado mediante:

- Una precondición: expresión booleana que describe los estados iniciales de las variables del programa, para los cuales se define el mismo.
- Una postcondición: expresión booleana que describe los estados finales del programa luego de su ejecución.

La notación que utilizaremos:

$$\{P\}\ S\ \{Q\}$$

donde P y Q son predicados y S es un programa (o secuencia de comandos), tiene la siguiente interpretación:

> Siempre que se ejecuta S comenzando en un estado que satisface P, el programa termina y el estado final satisface Q.

P es llamada precondición de S, y Q su postcondición. La terna $\{P\}\ S\ \{Q\}$ se denomina tripleta de Hoare.

Derivar un programa imperativo consiste en encontrar S que satisfaga las especificación dada, es decir, consiste en encontrar S tal que, aplicado a un estado que satisfaga P, termine en un estado que satisfaga Q. Mientras que una verificación de que S es correcto (respecto a su especificación) consiste en probar que la tripleta $\{P\}\ S\ \{Q\}$ es verdadera.

Cuando querramos dar solo la especificación de un programa, no utilizaremos la tripleta $\{P\}\ S\ \{Q\}$, dado que esta notación no indica qué variables cambiaron en S. Por ejemplo, $\{true\}\ S\ \{x = y\}$, dice que S termina en un estado que satisface $x = y$, pero no cuáles variables entre x e y cambiaron luego de la ejecución.

Denotaremos formalmente una especificación como:

$$\{P\}\ x :=?\ \{Q\} \tag{12.1}$$

donde P es la precondición del programa, Q la postcondición, y x es la lista de variables que pueden cambiar de valor.

En los lenguajes imperativos, también es necesario explicitar el tipo de las variables y constantes del programa, lo cual se hace a través de una *declaración de variables y constantes*. Esto completa la especificación del programa a la vez que indica implícitamente las operaciones que se pueden realizar con las variables y constantes declaradas.

Como ejemplo consideremos la siguiente especificación de un programa que calcula el mayor múltiplo de 5 menor que n.

$|[$ **var** z : **Integer**;
const n : **Integer**;
$\{true\}$
$z :=?$
$\{z = (max\, i : 0 \leq i < n \wedge mod.i{,}5 = 0 : i)\}$
$]|$

Usamos $|[\ ]|$ para encerrar la declaración de variables y constantes junto con la terna de Hoare. La sintaxis que utilizaremos para esta declaración es la del lenguage Pascal.

En el ejemplo n es declarada como una constante porque es un valor que no cambia nunca en la ejecución del programa.

12.2.1. Representación de variables iniciales y finales

El siguiente programa:

$$x, y := y, x$$

intercambia los valores de las variables x e y. Para poder especificar este programa, necesitamos una forma de poder describir los valores iniciales de las variables x e y. Usaremos para ello dos *constantes de especificación* que llamamos X e Y. La especificación del programa, junto con el programa mismo sería:

$$\{x = X \wedge y = Y\}\ \ x, y := y, x\ \ \{x = Y \wedge y = X\}$$

Es importante notar que estas constantes no son constantes del programa , por lo tanto no pueden aparecer en sentencias del mismo. Sólo pueden ocurrir en los predicados que forman la pre y post condición. En estos predicados pueden aparecer tres tipos de nombres: las variables de especificación, que denotaremos con letras mayúsculas; las variables del programa y variables de cuantificación.

A continuación se verán algunos ejemplos de especificación de programas, usando la notación 12.1. En el capítulo siguiente obtendremos los programas que satisfacen las especificaciones dadas, por ahora sólo nos abocaremos a definir las pre y post condiciones.

(12.1) **Ejemplo.** Especificaremos un programa que dadas dos variables enteras no negativas a y b, calcula el producto de ambas, mediante los siguientes predicados:

$$|[\ \textbf{var}\ a, b, z : \textbf{Integer};\ \ \{a \geq 0 \wedge b \geq 0\}\ \ z :=?\ \ \{z = a * b\}\ \ ]|$$

El producto de a y b tiene que ser almacenado en alguna variable, elegimos z.

(12.2) **Ejemplo.** Especificaremos ahora un programa que dada una variable entera n, calcula la suma de los n primeros números naturales y almacena el resultado en n.

$$
\begin{array}{ll}
|[& \textbf{var } n : \textbf{Integer}; \\
& \{n \geq 0 \wedge n = N\} \\
& n :=? \\
& \{n = (\sum\, i : 0 < i < N : i)\} \\
]|
\end{array}
$$

(12.3) **Ejemplo.** Los arreglos (arrays en inglés) son estructuras de datos que proveen la mayoría de los lenguajes de programación imperativos. Un arreglo puede pensarse como una lista de tamaño fijo. Utilizaremos la siguiente notación para describir un arreglo de $n+1$ elementos: $b[0, \ldots, n]$. El acceso a los elementos es similar al de listas, se realiza mediante índices. Por ejemplo, $b[0]$ corresponde al primer elemento del arreglo b.

Especificaremos un programa que dado un entero no negativo n y un arreglo $b[0, \ldots, n]$, calcula el producto de los primeros $n+1$ elementos del arreglo.

$$
\begin{array}{ll}
|[& \textbf{const } n : \textbf{Integer}; \\
& \textbf{var } b : \textbf{array } [0 \ldots n] \textbf{ of Integer}; \\
& \{n \geq 0\} \\
& z :=? \\
& \{z = (\prod\, i : 0 \leq i \leq n : b[i])\} \\
]|
\end{array}
$$

(12.4) **Ejemplo.** Por último daremos la especificación de un programa que dados una variable entera n y un arreglo $b[0, \ldots, n]$ de enteros, ordena el arreglo de forma creciente.

$$
\begin{array}{ll}
|[& \textbf{const } n : \textbf{Integer}; \\
& \textbf{var } b : \textbf{array } [0 \ldots n] \textbf{ of Integer}; \\
& \{n \geq 0 \wedge b = B\} \\
& b :=? \\
& \{perm(b, B) \ \wedge \ (\forall\, i : 0 \leq i < n : b[i] \leq b[i+1])\} \\
]|
\end{array}
$$

donde $perm$ es un predicado que toma dos arreglos y determina si uno es una permutación del otro. Dejamos como ejercicio la definición de este predicado.

Antes de continuar con el desarrollo de programas, veremos algunas reglas para la correcta interpretación y utilización de tripletas de Hoare.

12.3. Leyes sobre tripletas

El primer teorema, conocido como *ley de exclusión de milagros*, expresa:

(12.5) Teorema.

$$\{P\}\ S\ \{false\} \equiv P \equiv false$$

Esta ley establece que para cualquier programa S, si se requiere que termine y que los estados finales satisfagan $false$, entonces ese programa no puede ejecutarse exitosamente para ningún estado inicial. O bien, leído de otra manera, no existe ningún estado tal que si se ejecuta un programa S comenzando en él se puede terminar en un estado que satisfaga $false$.

La expresión

$$\{P\}\ S\ \{true\}$$

indica que, cada vez que comienza en un estado que satisface P, la ejecución de S concluye en un estado que satisface el predicado $true$. Esto se interpreta diciendo que la ejecución de S *termina* cada vez que se comienza en un estado que satisface P.

Las leyes que siguen a continuación expresan el hecho de que una precondición puede ser fortalecida y una postcondición puede ser debilitada, lo cual formulamos de la siguiente manera:

(12.6) Teorema. Fortalecimiento de precondición.

$$\{P\}\ S\ \{Q\} \wedge (P_o \Rightarrow P) \Rightarrow \{P_o\}\ S\ \{Q\}$$

(12.7) Teorema. Debilitamiento de postcondición.

$$\{P\}\ S\ \{Q\} \wedge (Q \Rightarrow Q_o) \Rightarrow \{P\}\ S\ \{Q_o\}$$

Se dice que una expresión P_0 es más fuerte que P si y solo si $P_0 \Rightarrow P$. Recíprocamente, P_0 es más débil que P si y solo si $P \Rightarrow P_0$. Con lo cual, la primera regla dice que si P_0 es más fuerte que P y vale $\{P\}\ S\ \{Q\}$, entonces $\{P_0\}\ S\ \{Q\}$ también es válida.

La segunda regla dice que si Q_0 es más débil que Q y vale $\{P\}\ S\ \{Q\}$, entonces también vale $\{P\}\ S\ \{Q_0\}$.

Utilizaremos los dos teoremas anteriores para demostrar que ciertas tripletas son verdaderas.

(12.8) Ejemplo. Probaremos que la tripleta $\{x = 1\}\ x := x + 1\ \{x > 0\}$ es cierta. Para ello supondremos válida la tripleta $\{x \geq 0\}\ x := x + 1\ \{x > 0\}$ (en el capítulo siguiente veremos cómo demostrar su validez).

Dado que la primer tripleta tiene una precondición más fuerte que la segunda y siendo esta la única diferencia, podemos probar que la tripleta es cierta de la siguiente manera:

$$\begin{array}{ll} & \{x = 1\}\ x := x + 1\ \{x > 0\} \\ \Leftarrow & \quad \langle \text{Fortalecimiento de precondición} \rangle \\ & \{x \geq 0\}\ x := x + 1\ \{x > 0\} \wedge (x = 1 \Rightarrow x \geq 0) \\ = & \quad \langle \text{supuesto; aritmética; neutro de } \wedge \rangle \\ & true \end{array}$$

(12.9) Ejemplo. Ahora veremos un ejemplo donde probamos que una tripleta es cierta debilitando su postcondición. Supondremos válida la tripleta del ejemplo anterior y probaremos $\{x \geq 0\}\ x := x + 1\ \{x \geq 0\}$.

$$
\begin{array}{ll}
 & \{x \geq 0\}\ x := x + 1\ \{x \geq 0\} \\
\Leftarrow & \quad \langle \text{Debilitamiento de postcondición} \rangle \\
 & \{x \geq 0\}\ x := x + 1\ \{x > 0\} \wedge (x > 0 \Rightarrow x \geq 0) \\
= & \quad \langle \text{supuesto; aritmética; neutro de } \wedge \rangle \\
 & true
\end{array}
$$

Supongamos que las tripletas $\{P\}\ S\ \{Q\}$ y $\{P\}\ S\ \{R\}$ son válidas. Entonces la ejecución de S en un estado que satisface P termina en uno que satisface Q y R, por lo tanto termina en un estado que satisface $Q \wedge R$. Esta ley se expresa de la siguiente manera:

(12.10) Teorema.

$$\{P\}\ S\ \{Q\} \wedge \{P\}\ S\ \{R\} \equiv \{P\}\ S\ \{Q \wedge R\}$$

Por último, la ley que sigue

(12.11) Teorema.

$$\{P\}\ S\ \{Q\} \wedge \{R\}\ S\ \{Q\} \equiv \{P \vee R\}\ S\ \{Q\}$$

12.4. El transformador de predicados wp

Para cada comando S se puede definir una función $wp.S : Predicados \rightarrow Predicados$ tal que si Q es un predicado, $wp.S.Q$ representa el predicado P más débil para el cual vale $\{P\}\ S\ \{Q\}$. Para cada Q, $wp.S.Q$ se denomina precondición más débil de S con respecto a Q. En otras palabra, para todo predicado Q, $wp.S.Q = P$ si y solo si $\{P\}\ S\ \{Q\}$ y para cualquier predicado P_0 tal que $\{P_0\}\ S\ \{Q\}$ vale $P_0 \Rightarrow P$ (P es más débil que P_0).

La definición de $wp.S$ permite relacionar las expresiones $\{P\}\ S\ \{Q\}$ y $wp.S.Q$:

(12.12) Teorema. Relación entre Terna de Hoare y wp.

$$\{P\}\ S\,\{Q\}\ \equiv\ P \Rightarrow wp.S.Q$$

12.5. Ejercicios

12.1 Escribir especificaciones para los programas que realizan las siguientes actividades:

a) Dadas dos variables enteras, copia el valor de la primera en la segunda.

b) Almacena en la variable x el máximo valor del arreglo de enteros $b[0, \ldots, n]$.

c) Dada una variable entera, almacena en ella su valor absoluto.

d) Dadas tres variables enteras, devuelve el máximo de las tres.

e) Dada una variable entera x, cuyo valor es mayor a 1, determina si x almacena un número primo.

f) Dada una variable entera $n \geq 0$ y un arreglo $b[0, \ldots, n]$, almancena en cada posición del arreglo la suma de todos los elementos del mismo.

12.2 Dar el significado operativo de las tripletas: $\{true\}\ S\ \{true\}$ y $\{false\}\ S\ \{true\}$

12.3 Deducir a partir de las leyes dadas en la sección 12.3 que

$$\{P_0\}\ S\ \{Q_0\} \quad \text{y} \quad \{P_1\}\ S\ \{Q_1\}$$

implican

$$\{P_0 \wedge P_1\}\ S\ \{Q_0 \wedge Q_1\} \quad \text{y} \quad \{P_0 \vee P_1\}\ S\ \{Q_0 \vee Q_1\}$$

12.4 Demostrar que $\{false\}\ S\ \{P\}$ es válida para cualquier S y cualquier P.

12.5 Demostrar que las leyes vistas en la sección 12.3, siguen de las siguientes leyes para $wp.S$:

a) $wp.S.false \equiv false$

b) $wp.S.Q \wedge wp.S.R \equiv wp.S.(Q \wedge R)$

c) $wp.S.Q \vee wp.S.R \Rightarrow wp.S.(Q \vee R)$

CAPÍTULO 13

Definición de un lenguaje de programación imperativo

Índice del Capítulo

13.1. La sentencia skip 197

13.2. La sentencia abort 198

13.3. La sentencia de asignación 198

13.4. Concatenación o composición 200

13.5. La sentencia alternativa 202

13.6. Repetición 205

13.6.1. Terminación de ciclos 207

13.7. Ejemplos 208

13.8. Ejercicios 212

En este capítulo veremos una serie de sentencias o comandos que definen un lenguaje de programación imperativo básico. Las sentencias serán definidas utilizando tripletas de Hoare. Para cada sentencia S y predicado Q, describiremos primero el resultado deseado de la ejecución de S, mediante el predicado $wp.S.Q$ (el cual representa el conjunto de todos los estados, tal que, si S se ejecuta en alguno de ellos su ejecución termina en un estado que satisface Q).

13.1. La sentencia skip

La primera sentencia que veremos es la sentencia **skip**. La ejecución de **skip** no produce cambios sobre los estados de las variables. La utilidad de este comando se entenderá más adelante, pero podemos adelantar que será como una sentencia neutra para los programas.

Queremos caracterizar la sentencia **skip** usando tripletas de Hoare. Comencemos estudiando $wp.\textbf{skip}.Q$. Claramente el predicado más débil P tal que $\{P\}$ **skip** $\{Q\}$ es válida es Q, pues lo mínimo necesario para asegurar el estado final Q cuando no se hace nada es tener Q como estado inicial. Por lo tanto,

$$wp.\textbf{skip}.Q \equiv Q$$

Esto podría inducirnos a caracterizar **skip** usando la tripleta $\{Q\}$ **skip** $\{Q\}$ como axioma, pero como ya hemos visto que es posible fortalecer la precondición, daremos una caracterización más general de este comando:

(13.1) Axioma.

$$\{P\} \textbf{ skip } \{Q\} \equiv P \Rightarrow Q$$

Por ejemplo, la tripleta $\{x \geq 1\}$ **skip** $\{x \geq 0\}$ es válida pues $x \geq 1 \Rightarrow x \geq 0$.

13.2. La sentencia abort

La sentencia **abort** está especificada por:

(13.2) Axioma.

$$\{P\} \textbf{ abort } \{Q\} \quad \equiv \quad P \equiv \mathit{false}$$

Después de esta regla, ¿cómo se ejecuta **abort**?. La expresión de arriba establece que este comando *nunca* debe ejecutarse, ya que solo puede hacerse en un estado que satisface false, y no existe tal estado. Si la ejecución de un programa alcanza alguna vez un punto en el que **abort** deba ejecutarse, entonces el programa completo es un error, y termina prematuramente.

En términos de la precondición más débil, tenemos que

$$wp.\textbf{abort}.Q \equiv \mathit{false}$$

13.3. La sentencia de asignación

Cualquier cambio de estado que ocurra durante la ejecución de un programa se debe a la sentencia de asignación. Una sentencia de asignación tiene la forma $x := E$, donde x denota una variable o varias variables del programa separadas con coma y E denota una de expresión o varias expresiones, también separadas con coma (donde la cantidad de variables es igual a la de expresiones).

Dado un predicado Q, lo mínimo que se requiere para que Q sea verdadera luego de ejecutar $x := E$ es el predicado $Q[x := E]$. Es decir,

$$wp.(x := E).Q \equiv Q[x := E]$$

Entonces, para que la terna $\{P\}$ $x := E$ $\{Q\}$ sea correcta, el predicado P debe ser más fuerte que $Q[x := E]$. De esta manera, la asignación está especificada por:

(13.3) Axioma.

$$\{P\}\ x := E\ \{Q\}\ \equiv\ P \Rightarrow Q[x := E]$$

Asumiremos que en E solo se admitirán expresiones bien definidas, sino habría que agregar un predicado $def.E$ (que sea verdadero solo si E está bien definida) en el consecuente de la implicación: $\{P\}\ x := E\ \{Q\}\ \equiv\ P \Rightarrow def.E \wedge Q[x := E]$.

Notemos que el símobolo $:=$ utilizado en el programa $x := E$ no es el mismo que el usado en el predicado $Q[x := E]$, la primera expresión es una sentencia válida del lenguaje de programación que estamos definiendo, mientras que en la segunda expresión, $:=$ es una operación sintáctica que se aplica a expresiones lógicas, que llamamos sustitución.

(13.4) Ejemplo. Probaremos que la tripleta $\{x \geq 3\}\ x := x + 1\ \{x \geq 0\}$ es válida.

utilizando el axioma 13.3.

$$\begin{array}{ll} & \{x \geq 3\}\ x := x + 1\ \{x \geq 0\} \\ = & \quad \langle \text{axioma 13.3} \rangle \\ & x \geq 3 \Rightarrow (x \geq 0)[x := x + 1] \\ = & \quad \langle \text{aplicación de sustitución} \rangle \\ & x \geq 3 \Rightarrow x + 1 \geq 0 \\ = & \quad \langle \text{aritmética} \rangle \\ & x \geq 3 \Rightarrow x \geq -1 \\ = & \quad \langle \text{aritmética} \rangle \\ & x \geq 3 \Rightarrow -1 \leq x < 3 \vee x \geq 3 \\ = & \quad \langle \text{deb/fort a)} \rangle \\ & true \end{array}$$

Por lo tanto, hemos probado que el programa $x := x + 1$ satisface la especificación dada.

(13.5) Ejemplo. También utilizaremos el axioma 13.3 para obtener programas correctos según la especificación. Por ejemplo, encontraremos e tal que se satisfaga $\{true\}\ x, y := x + 1, e\ \{y = x + 1\}$

Según el axioma 13.3, para que el programa sea correcto E debe cumplir:

$$\begin{array}{ll} & true \Rightarrow (y = x + 1)\,[x, y := x + 1, e] \\ = & \quad \langle \text{Elemento neutro a izquierda de } \Rightarrow \rangle \\ & (y = x + 1)\,[x, y := x + 1, e] \\ = & \quad \langle \text{aplicación de sustitución} \rangle \\ & e = x + 1 + 1 \\ = & \quad \langle \text{aritmética} \rangle \\ & e = x + 2 \end{array}$$

Es decir, cualquier expresión e que sea equivalente a $x + 2$ es válida; en particular, podemos tomar $e = x + 2$.

(13.6) **Ejemplo.** Por último, hallaremos un programa que satisfaga $\{x = A \wedge y = B\}\ x, y := e, f\ \{x = 2 \times B \wedge y = 2 \times A\}$.

Según 13.3, las expresiones e y f deben cumplir:

$x = A \wedge y = B \Rightarrow (x = 2 \times B \wedge y = 2 \times A)\,[x, y := e, f]$

Suponemos cierto el antedecente $x = A \wedge y = B$ y partimos del consecuente,

$$
\begin{array}{ll}
 & (x = 2 \times B \wedge y = 2 \times A)\,[x, y := e, f] \\
= & \quad \langle \text{aplicación de sustitución} \rangle \\
 & e = 2 \times B \wedge f = 2 \times A \\
= & \quad \langle \text{antecedente} \rangle \\
 & e = 2 \times y \wedge f = 2 \times x
\end{array}
$$

13.4. Concatenación o composición

La concatenación permite escribir secuencias de sentencias. La ejecución de dos sentencias S y T, una a continuación de la otra se indicará separando las mismas con punto y coma ($S; T$), o una debajo de la otra, siendo la de la izquierda (o la de arriba) la primera que se ejecuta. Para demostrar $\{P\}\ S; T\ \{Q\}$, hay que encontrar un predicado intermedio R que sirva como poscondición de S y precondición de T, es decir:

(13.7) **Axioma.**

$$\{P\}\ S; T\ \{Q\} \equiv (\exists R :: \{P\}\ S\ \{R\} \wedge \{R\}\ T\ \{Q\})$$

La precondición más débil de la concatenación $S; T$ con respecto al predicado Q, se obtiene tomando primero la precondición más débil de T con respecto a Q y luego la precondición más débil de S con respecto a este último predicado. Es decir,

$$wp.(S; T).Q \equiv wp.S.(wp.T.Q)$$

(13.8) **Ejemplo.** Como ejemplo probaremos que el programa dado a continuación es correcto, probando la siguiente tripleta:

$$
\begin{array}{ll}
|[& \textbf{var}\ a, b : \textbf{Boolean}; \\
 & \{(a \equiv A) \wedge (b \equiv B)\} \\
 & a := a \equiv b \\
 & b := a \equiv b \\
 & a := a \equiv b \\
 & \{(a \equiv B) \wedge (b \equiv A)\} \\
]| &
\end{array}
$$

Para ello anotaremos el programa con los siguientes predicados intermedios:

$$
\begin{array}{ll}
|[& \textbf{var } a, b : \textbf{Boolean}; \\
& \{P : (a \equiv A) \wedge (b \equiv B)\} \\
& a := a \equiv b \\
& \{R_2 : wp.(b := a \equiv b).R_1\} \\
& b := a \equiv b \\
& \{R_1 : wp.(a := a \equiv b).Q\} \\
& a := a \equiv b \\
& \{Q : (a \equiv B) \wedge (b \equiv A)\} \\
]|
\end{array}
$$

y probaremos que las tripletas:

$$
\begin{array}{l}
\{P\}\, a := a \equiv b\, \{R_2\} \\
\{R_2\}\, b := a \equiv b\, \{R_1\} \\
\{R_1\}\, a := a \equiv b\, \{Q\}
\end{array}
$$

son válidas. Las letras P, Q, R_1 y R_2 fueron agregadas para hacer más legible la notación.

Por el axioma de asignación 13.3 podemos concluir que las dos últimas tripletas son ciertas. Con lo cual, solo debemos probar la primera tripleta. Hallamos primero R_1 y R_2:

$$
\begin{array}{ll}
 & ((a \equiv B) \wedge (b \equiv A))[a := a \equiv b] \\
= & \quad \langle \text{aplicación de sustitución} \rangle \\
 & (a \equiv b \equiv B) \wedge (b \equiv A)
\end{array}
$$

$$
\begin{array}{ll}
 & ((a \equiv b \equiv B) \wedge (b \equiv A))[b := a \equiv b] \\
= & \quad \langle \text{aplicación de sustitución} \rangle \\
 & (a \equiv a \equiv b \equiv B) \wedge (a \equiv b \equiv A) \\
= & \quad \langle \text{elemento neutro de } \equiv \rangle \\
 & (b \equiv B) \wedge (a \equiv b \equiv A)
\end{array}
$$

La primera tripleta es cierta si:

$$
\begin{array}{ll}
 & (a \equiv A) \wedge (b \equiv B) \Rightarrow ((b \equiv B) \wedge (a \equiv b \equiv A))[a := a \equiv b] \\
= & \quad \langle \text{aplicación de sustitución} \rangle \\
 & (a \equiv A) \wedge (b \equiv B) \Rightarrow (b \equiv B) \wedge (a \equiv b \equiv b \equiv A) \\
= & \quad \langle \text{elemento neutro de } \equiv \rangle \\
 & (a \equiv A) \wedge (b \equiv B) \Rightarrow (b \equiv B) \wedge (a \equiv A) \\
= & \quad \langle \text{reflexividad de } \equiv \rangle \\
 & true
\end{array}
$$

13.5. La sentencia alternativa

Los lenguajes de programación imperativos usualmente proveen una notación para el comando llamado condicional o sentencia **if**, el cual permite la ejecución de sentencias dependiendo del estado de las variables del programa. Un ejemplo del comando condicional es el siguiente, escrito es Pascal:

$$\textbf{if } x \geq 0 \textbf{ then } z := x \textbf{ else } z := -x \tag{13.1}$$

La ejecución de este comando almacena el valor absoluto de x en z. Si $x \geq 0$ entonces se ejecuta la primer sentencia $z := x$, sino se ejecuta la segunda sentencia $z := -x$.

En la notación que utilizaremos en nuestro lenguaje de programación, este comando se escribe como:

$$\begin{array}{lrcl} \textbf{if} & x \geq 0 & \rightarrow & z := x \\ [] & \neg(x \geq 0) & \rightarrow & z := -x \\ \textbf{fi} & & & \end{array}$$

Este comando contiene dos expresiones de la forma $B \rightarrow S$ (separadas por el símbolo []), donde B es una expresión booleana, llamada guarda y S es una sentencia. Para ejecutar el comando se debe encontrar una guarda verdadera y ejecutar su sentencia correspondiente.

La forma general de esta sentencia es la siguiente:

$$\begin{array}{lllll} IF: & \textbf{if} & B_0 & \rightarrow & S_0 \\ & [] & B_1 & \rightarrow & S_1 \\ & \vdots & & & \\ & [] & B_n & \rightarrow & S_n \\ & \textbf{fi} & & & \end{array}$$

donde para todo $0 \leq i \leq n$, B_i es una expresión booleana y S_i es una sentencia.

La sentencia IF funciona de la siguiente manera: todas las guardas son evaluadas; si ninguna es $True$ se produce un **abort**; en caso contrario, se elige (de alguna manera) una guarda B_i que sea $True$ y se ejecuta la correspondiente sentencia S_i.

Ya hemos visto que **abort** no es una sentencia ejecutable; por lo tanto, un **if** con ninguna guarda será considerado un error

Si más de una guarda es $true$, la sentencia alternativa se dice *no determinística*. El "no-determinismo", es útil para escribir con claridad determinados programas.

Teniendo en cuenta lo dicho anteriormente podemos especificar la sentencia IF de la siguiente forma:

(13.9) Axioma.

$$
\begin{array}{llllll}
\{P\} & \textbf{if} & B_0 & \rightarrow & S_0 & \{Q\} \quad \equiv \quad (P \Rightarrow B_0 \vee B_1 \vee \ldots \vee B_n) \\
 & [] & B_1 & \rightarrow & S_1 & \wedge \{P \wedge B_0\}\, S_0\, \{Q\} \\
 & \vdots & & & & \wedge \{P \wedge B_1\}\, S_1\, \{Q\} \\
 & [] & B_n & \rightarrow & S_n & \vdots \\
 & \textbf{fi} & & & & \wedge \{P \wedge B_n\}\, S_n\, \{Q\}
\end{array}
$$

Por lo tanto, para probar $\{P\}\, IF\, \{Q\}$, es suficiente con probar:

i) $P \Rightarrow B_0 \vee B_1 \vee \ldots \vee B_n$

ii) $\{P \wedge B_i\}\, S_i\, \{Q\}$ para $0 \leq i \leq n$.

La implicación $P \Rightarrow B_0 \vee B_1 \vee \ldots \vee B_n$ es necesaria para asegurarse que si la precondición es cierta al menos una de las guardas es verdadera y por lo tanto el programa no es erróneo (no se produce un **abort**). Las expresiones $\{P \wedge B_i\}\, S_i\, \{Q\}$ aseguran que ejecutar S_i, cuando la precondición y la guarda S_i son verdaderas, conduce a un estado que satisface Q, es decir, cualquiera sea la guarda verdadera, ejecutar el respectivo comando conduce al estado final deseado.

De la misma forma que hicimos con la asignación, asumiremos que las expresiones B_i están bien definidas, de otra forma tendríamos que agregar una condición más en el término derecho de la $\equiv$.

La definición de $wp.IF.Q$ no es muy simple. Lo mínimo requerido para que Q sea cierta luego de ejecutar IF es que al menos una de las guardas sea cierta y que la ejecución de cada comando S_i con guarda B_i verdadera, termine en un estado que satisfaga Q. Esto último lo expresamos como: $B_i \Rightarrow wp.S_i.Q$.

$$wp.IF.Q = (B_0 \vee B_1 \vee \ldots \vee B_n) \wedge (B_0 \Rightarrow wp.S_0.Q) \wedge \ldots \wedge (B_n \Rightarrow wp.S_n.Q)$$

(13.10) Ejemplo.

Para probar que la siguiente tripleta es cierta,

$$
\begin{array}{l}
\{true\} \\
\quad \textbf{if} \;\; x \leq y \;\; \rightarrow \;\; \textbf{skip} \\
\quad [] \;\; x > y \;\; \rightarrow \;\; x, y := y, x \\
\quad \textbf{fi} \\
\{x \leq y\}
\end{array}
$$

debemos probar:

i) $true \Rightarrow x \leq y \vee x > y$

ii) $\{true \wedge x \leq y\}$ **skip** $\{x \leq y\} \wedge \{true \wedge x > y\}\ x, y := y, x\ \{x \leq y\}$

La demostración de i) es inmediata, dado que $(x \leq y \vee x > y) \equiv true$. Para demostrar ii) probaremos cada una de las tripletas:

$$
\begin{array}{ll}
 & \{true \wedge x \leq y\}\ \textbf{skip}\ \{x \leq y\} \\
= & \quad \langle \text{axioma 13.1} \rangle \\
 & true \wedge x \leq y \Rightarrow x \leq y \\
= & \quad \langle \text{elemento neutro de } \wedge \text{ ; reflexividad de } \Rightarrow \rangle \\
 & true
\end{array}
$$

$$
\begin{array}{ll}
 & \{true \wedge x > y\}\ x, y := y, x\ \{x \leq y\} \\
= & \quad \langle \text{axioma 13.3} \rangle \\
 & true \wedge x > y \Rightarrow (x \leq y)[x, y := y, x] \\
= & \quad \langle \text{elemento neutro de } \wedge \text{ ; aplicación de sustitución } \rangle \\
 & x > y \Rightarrow y \leq x \\
= & \quad \langle \text{aritmética} \rangle \\
 & x > y \Rightarrow x > y \vee x = y \\
= & \quad \langle \text{deb/fort a)} \rangle \\
 & true
\end{array}
$$

(13.11) Ejemplo. Veamos otra situación, donde queremos determinar un programa que satisfaga la siguiente especificación:

$$\{true\}\ z :=?\ \{z = max(x, y)\}$$

Vamos a utilizar la siguiente definición de máximo entre dos números:

$$z = max(x, y) \equiv (z = x \vee z = y) \wedge z \geq x \wedge z \geq y$$

Esto significa que un posible valor para el máximo es el valor de x, por tanto una sentencia posible a ejecutar es $z := x$. Veamos entonces cuál es la precondición más débil para la sentencia de asignación $z := x$ respecto del predicado $z = max(x, y)$:

$$
\begin{array}{ll}
 & ((z = x \vee z = y) \wedge z \geq x \wedge z \geq y)[z := x] \\
= & \quad \langle \text{Sustitución} \rangle \\
 & (x = x \vee x = y) \wedge x \geq x \wedge x \geq y) \\
= & \quad \langle \text{def. de igualdad; } (x \geq x) \equiv true \rangle \\
 & (true \vee x = y) \wedge true \wedge x \geq y) \\
= & \quad \langle \text{elemento neutro de } \wedge \text{ ; absorbente de } \vee \rangle \\
 & x \geq y
\end{array}
$$

Lo cual demuestra que si elegimos el predicado $x \geq y$, la tripleta $\{x \geq y\}\ z := x\ \{z = max(x, y)\}$ es válida.

Con el mismo razonamiento también será válida $\{y \geq x\}\ z := y\ \{z = max(x, y)\}$.

Además, observando que

$$
\begin{array}{ll}
 & x \geq y \vee y \geq x \\
= & \quad \langle \text{aritmética} \rangle \\
 & x = y \vee x > y \vee \neg(x > y) \\
= & \quad \langle \text{ tercero excluido; elemento absorvente de } \vee \rangle \\
 & true
\end{array}
$$

es fácil ver que $true \Rightarrow x \geq y \vee y \geq x$

Hasta aquí hemos probado:

i) $true \Rightarrow x \geq y \vee y \geq x$

ii) $\{x \geq y\}\ z := x\ \{z = max(x, y)\}\ \wedge\ \{y \geq x\}\ z := y\ \{z = max(x, y)\}$

Por lo tanto hemos encontrado un programa que es correcto respecto a la especificación dada:

if $x \geq y \ \rightarrow\ z := x$
[] $y \geq x \ \rightarrow\ z := y$
fi

13.6. Repetición

La lista de comandos se completa con una sentencia que permitirá la ejecución repetida de una acción (ciclo o bucle). La sintaxis es :

$DO:$ **do** $B_0 \rightarrow S_0$
[] $B_1 \rightarrow S_1$
$\vdots$
[] $B_n \rightarrow S_n$
od

donde para todo $0 \leq i \leq n$, B_i es una expresión booleana (también llamada guarda) y S_i una sentencia.

El ciclo funciona de la siguinete manera: mientras exista alguna guarda equivalente a $true$, se elige alguna (de alguna manera) y se ejecuta la sentencia correspondiente; luego vuelven a evaluarse las guardas. Esto se repite hasta que ninguna guarda sea $true$. Si desde el comienzo del ciclo, ninguna guarda es $true$, el ciclo completo equivale a un $skip$.

Luego de ejecutarse el ciclo todas las guardas son falsas. Llamaremos *iteración de un ciclo* al proceso que consiste en elegir una guarda y ejecutar su sentencia correspondiente.

Esta sentencia también puede ser no-determinista; si más de una guarda es verdadera solo se elige una por iteración.

La especificación de esta sentencia es la siguiente:

(13.12) Axioma.

$$
\begin{array}{llll}
\{P\} & \textbf{do} \;\; B_0 \to S_0 & \{Q\} & \equiv \;\; (P \wedge \neg B_0 \wedge \neg B_1 \wedge \ldots \wedge \neg B_n \Rightarrow Q) \\
 & [] \;\; B_1 \to S_1 & & \wedge \{P \wedge B_0\} \;\; S_0 \;\; \{P\} \\
 & \vdots & & \wedge \{P \wedge B_1\} \;\; S_1 \;\; \{P\} \\
 & [] \;\; B_n \to S_n & & \vdots \\
 & \textbf{od} & & \wedge \{P \wedge B_n\} \;\; S_n \;\; \{P\} \\
 & & & \wedge \text{ el ciclo termina}
\end{array}
$$

Es decir que para probar que $\{P\} \;\; DO \;\; \{Q\}$ es correcta, es suficiente con probar:

i) $P \wedge \neg B_0 \wedge \neg B_1 \wedge \ldots \wedge \neg B_n \Rightarrow Q$

ii) $\{P \wedge B_i\} \;\; S_i \;\; \{P\}$ para $0 \leq i \leq n$

iii) el ciclo termina

La implicación $P \wedge \neg B_0 \wedge \neg B_1 \wedge \ldots \wedge \neg B_n \Rightarrow Q$ asegura el cumplimiento de la post-condición cuando el ciclo termina, es decir cuando ninguna guarda es verdadera; mientras que $\{P \wedge B_i\} \;\; S_i \;\; \{P\}$ asegura que ejecutar S_i cuando se satisface $P \wedge B_i$ conduce a un estado que satisface P, lo cual es necesario para que tenga sentido volver a evaluar las guardas. La condición "el ciclo termina" es necesaria porque de otra manera el programa podría ejecutarse indefinidamente. Más adelante formalizaremos esta condición.

Asumiremos que las expresiones B_i están todas bien definidas.

Un predicado P que satisface $\{P \wedge B_i\} \;\; S_i \;\; \{P\}$ se llama *invariante*. Encontrar invariantes es la dificultad mayor a la hora de escribir bucles. Existen varias técnicas para determinar invariantes, por ahora solo veremos el funcionamiento de los ciclos suponiendo que ya se ha determinado el invariante.

(13.13) Ejemplo. Utilizaremos el axioma 13.12 para probar la siguiente tripleta de Hoare.

$$
\begin{array}{l}
\{P : 0 \leq i \leq n \wedge z = i \times x\} \\
\textbf{do } i \neq n \to i, z := i + 1, z + x \textbf{ od} \\
\{P \wedge i = n\}
\end{array}
$$

Este programa calcula el producto de valores de las variables x y n y lo guarda en z. El invariante propuesto para el bucle es el predicado P.

Probaremos primero el punto i),

$$
\begin{array}{ll}
 & P \ \wedge \ \neg(i \neq n) \Rightarrow P \ \wedge \ i = n \\
= & \quad \langle \text{definición de } \neq \text{; doble negación} \rangle \\
 & P \ \wedge \ i = n \Rightarrow P \ \wedge \ i = n \\
= & \quad \langle \text{reflexividad de } \Rightarrow \rangle \\
 & true
\end{array}
$$

Ahora probamos el punto ii),

$$
\begin{array}{ll}
 & \{P \ \wedge \ i \neq n\}\, i, z := i + 1, z + x \,\{P\} \\
= & \quad \langle \text{axioma 13.3} \rangle \\
 & P \ \wedge \ i \neq n \Rightarrow P[i, z := i + 1, z + x] \\
= & \quad \langle \text{definición de } P \text{; aplicación de sistitución} \rangle \\
 & 0 \leq i \leq n \ \wedge \ z = i \times x \ \wedge \ i \neq n \Rightarrow 0 \leq i + 1 \leq n \ \wedge \ z + x = (i + 1) \times x \\
= & \quad \langle \text{aritmética} \rangle \\
 & 0 \leq i \leq n - 1 \ \wedge \ z = i \times x \Rightarrow -1 \leq i \leq n - 1 \ \wedge \ z = i \times x \\
= & \quad \langle \text{aritmética} \rangle \\
 & 0 \leq i \leq n - 1 \ \wedge \ z = i \times x \Rightarrow (i = -1 \vee 0 \leq i \leq n - 1) \ \wedge \ z = i \times x \\
= & \quad \langle \text{deb/fort e)} \rangle \\
 & true
\end{array}
$$

La demostración formal de que el ciclo termina la veremos a continuación, antes podemos ver fácilmente que esta condición se cumple observando los valores que va tomando la variable i, ésta toma un valor no negativo antes de la ejecución del ciclo y aumenta uno en cada iteración, hasta alcanzar el valor n, con lo cual la guarda se evalúa falsa.

13.6.1. Terminación de ciclos

Para probar que un ciclo termina se debe encontrar una función $t : Estados \rightarrow Int$ que dado un estado de las variables del programa provea una cota superior del número de iteraciones que falten realizarse. Llameremos a t *función cota*. Podemos asegurarnos que t es una función cota si t decrece en una unidad en cada iteración y si a medida que se realiza una iteración la condición $t > 0$ se mantiene. Escribimos ésto formalmente de la siguiente manera:

Un ciclo termina si existe una función cota t tal que:

i) $P \wedge (B_0 \vee B_1 \vee \ldots \vee B_n) \Rightarrow t \geq 0$

ii) $\{P \wedge B_i \wedge t = T\} \ S_i \ \{t < T\}$ para $0 \leq i \leq n$.

La condición i) asegura que mientras alguna guarda es verdadera, la función cota es no negativa; o bien, leído al revés, si la función cota se hace negativa, entonces ninguna guarda es verdadera (puesto que P es el invariante y es siempre verdadero).

La condición ii) asegura que la función cota decrece luego de la ejecución de cualquier S_i. De esta manera, en algún momento la cota será negativa y por lo explicado más arriba el ciclo habrá terminado.

En el ejemplo anterior una posible elección para la función cota es $t = n - i$. Para hallar esta función tuvimos en cuenta que el programa termina cuando i alcanza el valor n y que inicialmente $i \leq n$ y aumenta uno en cada iteración.

La prueba formal de que el ciclo termina es la siguiente:

Prueba de i):

$$
\begin{array}{ll}
 & 0 \leq i \leq n \ \wedge \ z = i \times x \ \wedge \ i \neq n \\
= & \quad \langle \text{aritmética} \rangle \\
 & 0 \leq i \wedge i \leq n \ \wedge \ z = i \times x \ \wedge \ i \neq n \\
\Rightarrow & \quad \langle \text{deb/fort b)} \ \rangle \\
 & i \leq n \\
= & \quad \langle \text{aritmética} \rangle \\
 & n - i \geq 0
\end{array}
$$

Prueba de ii):

$$
\begin{array}{ll}
 & \{P \ \wedge \ i \neq n \ \wedge \ t = T\} \ i, z := i + 1, z + x \ \{t < T\} \\
= & \quad \langle \text{axioma 13.3} \rangle \\
 & P \ \wedge \ i \neq n \ \wedge \ t = T \Rightarrow (t < T)[i, z := i + 1, z + x] \\
= & \quad \langle \text{definición de } t\text{; sustitución} \rangle \\
 & P \ \wedge \ i \neq n \ \wedge \ n - i = T \Rightarrow n - (i + 1) < T
\end{array}
$$

Para continuar la prueba suponemos cierto el antecedente y demostramos el consecuente:

$$
\begin{array}{ll}
 & n - (i + 1) < T \\
= & \quad \langle \text{aritmética; antecedente } n - i = T \rangle \\
 & n - i - 1 < n - i \\
= & \quad \langle \text{aritmética} \rangle \\
 & true
\end{array}
$$

13.7. Ejemplos

En esta sección veremos algunos ejemplos de corrección de ciclos en contextos más generales, por ejemplo donde se inicializan las variables antes de comenzar el ciclo.

Las derivaciones de programas que contienen ciclos se basan en el cálculo invariantes. Por simplicidad solo haremos verificaciones de programas que contienen ciclos, en lugar de derivaciones.

(13.14) Ejemplo. Probaremos que el siguiente programa es correcto respecto a la especificación dada.

```
|[  const n : Integer;
    var s : Integer;
        b : array[0 ... n] of Integer;
    {n ≥ 0}
    i, s := 1, b[0]
    do i ≤ n → i, s := i + 1, s + b[i] od
    {s = (∑ i : 0 ≤ k ≤ n : b[k])}
]|
```

Comenzaremos proponiendo un invariante P, es decir, un predicado que sea cierto luego de cada iteración del ciclo. Utilizaremos este predicado para anotar el programa con un predicado intermedio que sirva como postcondición de la sentencia $i, s := 1, b[0]$ y precondición del ciclo.

```
|[  const n : Integer;
    var s : Integer;
        b : array[0 ... n] of Integer;
    {n ≥ 0}
    i, s := 1, b[0]
    {P}
    do i ≤ n → i, s := i + 1, s + b[i] od
    {s = (+ i : 0 ≤ k < n + 1 : b[k])}
]|
```

Las variables que cambian de estado en cada iteración son i y s, la primera toma valores dentro del rango $1, \ldots, n+1$ y la variables s acumula siempre la suma de los primeros i elementos del arreglo. Por lo tanto, proponemos el invariante: $1 \leq i \leq n+1 \wedge s = (+\, k : 0 \leq k < i : b[k])$.

Para probar que la tripleta es cierta utilizaremos el axioma 13.7 y probaremos que:

1) $\{n \geq 0\}\ \ i, s := 1, b[0]\ \ \{P\}$
2) $\{P\}$ **do** $i \leq n \rightarrow i, s := i + 1, s + b[i]$ **od** $\{s = (+\, i : 0 \leq k < n + 1 : b[k])\}$

El punto 1) es muy sencillo y lo dejamos como ejercicio. Para probar el punto 2) debemos demostrar los siguientes items:

i) $P \wedge \neg(i \leq n) \Rightarrow s = (+\, i : 0 \leq k \leq n : b[k])$
ii) $\{P \wedge i \leq n\}\ \ i, s := i + 1, s + b[i]\ \ \{P\}$
iii) Existe una función t tal que:

a) $P \wedge i \leq n \Rightarrow t \geq 0$
b) $\{P \wedge i \leq n \wedge t = T\}\ i, s := i+1, s+b[i]\ \{t < T\}$

Prueba de i):

$$
\begin{array}{ll}
 & 1 \leq i \leq n+1 \wedge s = (+k : 0 \leq k < i : b[k]) \wedge \neg(i \leq n) \\
= & \quad \langle \text{aritmética} \rangle \\
 & 1 \leq i \leq n+1 \wedge s = (+k : 0 \leq k < i : b[k]) \wedge i > n \\
= & \quad \langle \text{aritmética} \rangle \\
 & i = n+1 \wedge s = (+k : 0 \leq k < i : b[k]) \\
\Rightarrow & \quad \langle \text{regla de sustitución; deb/fort b)} \rangle \\
 & s = (+k : 0 \leq k < n+1 : b[k])
\end{array}
$$

Prueba de ii):

$$
\begin{array}{ll}
 & \{P \wedge i \leq n\}\ i, s := i+1, s+b[i]\ \{P\} \\
= & \quad \langle \text{axioma 13.3} \rangle \\
 & P \wedge i \leq n \Rightarrow P[i, s := i+1, s+b[i]] \\
= & \quad \langle \text{definición de } P\text{; sustitución} \rangle \\
 & 1 \leq i \leq n+1 \wedge s = (+k : 0 \leq k < i : b[k]) \wedge i \leq n \Rightarrow \\
 & 1 \leq i+1 \leq n+1 \wedge s + b[i] = (+k : 0 \leq k < i+1 : b[k]) \\
= & \quad \langle \text{aritmética; separación de un término} \rangle \\
 & 1 \leq i \leq n \wedge s = (+k : 0 \leq k < i : b[k]) \Rightarrow \\
 & 0 \leq i \leq n \wedge s + b[i] = b[i] + (+k : 0 \leq k < i : b[k]) \\
= & \quad \langle \text{aritmética; separación de un término} \rangle \\
 & 1 \leq i \leq n \wedge s = (+k : 0 \leq k < i : b[k]) \Rightarrow \\
 & (i = 0 \vee 1 \leq i \leq n) \wedge s = (+k : 0 \leq k < i : b[k]) \\
= & \quad \langle \text{deb/fort e)} \rangle \\
 & true
\end{array}
$$

Para la prueba de iii) proponemos la función cota $t = n - i$, la demostración es similar a la del ejemplo anterior y la dejamos como ejercicio.

(13.15) Ejemplo. Por último veremos la corrección del siguiente ciclo que calcula el máximo común divisor entre dos números enteros positivos:

$|[$ **var** x, y : **Integer**
$\{P : x > 0 \wedge y > 0 \wedge mcd.x.y = mcd.\mathrm{X}.\mathrm{Y}\}$
do $x > y \rightarrow x := x - y$
$[]\ y > x \rightarrow y := y - x$
od
$\{x = mcd.\mathrm{X}.\mathrm{Y}\}$
$]|$

Recordemos las siguientes propiedades de mcd:

1) $mcd.x.x = x$

2) $x > y \Rightarrow mcd.x.y = mcd.(x - y).y$

3) $y > x \Rightarrow mcd.x.y = mcd.x.(y - x)$

Primero probaremos que la postcondición se satisface al terminar el ciclo y que P es un invariante, es decir:

i) $P \wedge \neg(x > y) \wedge \neg(y > x) \Rightarrow x = mcd.X.Y$

ii) $\{P \wedge x > y\}\ x := x - y\ \{P\}$

iii) $\{P \wedge y > x\}\ y := y - x\ \{P\}$

Prueba de i):

$$
\begin{array}{ll}
 & P \wedge \neg(x > y) \wedge \neg(y > x) \\
= & \quad \langle \text{aritmética} \rangle \\
 & P \wedge x = y \\
= & \quad \langle \text{definición de } P\text{; regla de sustitución} \rangle \\
 & x > 0 \wedge x > 0 \wedge mcd.x.x = mcd.\mathtt{X}.\mathtt{Y} \\
\Rightarrow & \quad \langle \text{deb/fort b)} \rangle \\
 & mcd.x.x = mcd.\mathtt{X}.\mathtt{Y} \\
= & \quad \langle \text{Propiedad 1)} \rangle \\
 & x = mcd.\mathtt{X}.\mathtt{Y}
\end{array}
$$

Prueba de ii):

$$
\begin{array}{ll}
 & \{P \wedge x > y\}\ x := x - y\ \{P\} \\
= & \quad \langle \text{axioma 13.3} \rangle \\
 & P \wedge x > y \Rightarrow P[x := x - y] \\
= & \quad \langle \text{definición de } P\text{; sustitución} \rangle \\
 & x > 0 \wedge y > 0 \wedge mcd.x.y = mcd.\mathtt{X}.\mathtt{Y} \wedge x > y \Rightarrow \\
 & x - y > 0 \wedge y > 0 \wedge mcd.(x - y).y = mcd.\mathtt{X}.\mathtt{Y} \\
= & \quad \langle \text{aritmética; propiedad 2)} \rangle \\
 & x > 0 \wedge x > 0 \wedge mcd.(x - y).y = mcd.\mathtt{X}.\mathtt{Y} \wedge x > y \Rightarrow \\
 & x > y \wedge y > 0 \wedge mcd.(x - y).y = mcd.\mathtt{X}.\mathtt{Y} \\
= & \quad \langle \text{deb/fort b)} \rangle \\
 & true
\end{array}
$$

La prueba de iii) es similar a ii).

Tomaremos como función cota $t = x + y$, entonces debemos probar que t decrece en cada iteración y que mientras se ejecuta el ciclo $t > 0$, es decir:

iv) $\{P \wedge x > y \wedge t = T\}\ x := x - y\ \{t < T\}$

v) $\{P \wedge y > x \wedge t = T\}\ y := y - x\ \{t < T\}$

vi) $P \wedge x > y \Rightarrow t > 0$

vii) $P \wedge y > x \Rightarrow t > 0$

La prueba de que el ciclo termina se deja como ejercicio.

13.8. Ejercicios

13.1 Demostrar que las siguientes tripletas son válidas:

a)
$|[$**var** x, y : **Integer**
$\{x > 0 \wedge y > 0\}$
skip
$\{x > 0\}$
$]|$

b)
$|[$**var** x, y : **Integer**
$\{x > 0 \wedge y > 0\}$
skip
$\{y \geq 0\}$
$]|$

c)
$|[$**var** x, y : **Boolean**;
$\{x \equiv y\}$
skip
$\{x \Rightarrow y\}$
$]|$

13.2 Demostrar que las siguientes tripletas no son válidas:

a)
$|[$**var** x, y : **Integer**;
$\{x > 0 \wedge y > 0\}$
skip
$\{x = 1\}$
$]|$

b)
$|[$**var** x, y : **Integer**;
$\{x > 0 \wedge y > 0\}$
skip
$\{y \geq x\}$
$]|$

$|[$**var** x, y : **Boolean**;
$\{x \equiv y\}$
c) **skip**
$\{x \wedge y\}$
$]|$

13.3 Determinar la precondición más débil P en cada caso:

a) $\{P\}\ x := x + 1\ \{x > 0\}$

b) $\{P\}\ x := x * x\ \{x > 0\}$

c) $\{P\}\ x := x + 1\ \{x = x + 1\}$

d) $\{P\}\ x := E\ \{x = E\}$

e) $\{P\}\ x, y := x + 1, y - 1\ \{x + y > 0\}$

f) $\{P\}\ x, y := y + 1, x - 1\ \{x > 0\}$

g) $\{P\}\ a := a \Rightarrow b\ \{a \vee b\}$

h) $\{P\}\ x := x + 1; x := x + 1\ \{x > 0\}$.

i) $\{P\}\ x := x \not\equiv y; y := x \not\equiv y; x := x \not\equiv y\ \{(x \equiv \mathtt{X}) \wedge (y \equiv \mathtt{Y})\}$.

j) $\{P\}\ x := x + 1; skip\ \{x^2 - 1 = 0\}$.

13.4 Determinar el predicado Q más fuerte para que el programa sea correcto en cada caso:

a) $\{x = 10\}\ x := x + 1\ \{Q\}$

b) $\{x \geq 10\}\ x := x - 10\ \{Q\}$

c) $\{x^2 > 45\}\ x := x + 1\ \{Q\}$

d) $\{0 \leq x < 10\}\ x := x^2\ \{Q\}$

13.5 Demostrar que la siguiente tripleta es válida: $\{x = A \wedge y = B\}\ x := x - y; y := x + y; x := x - y\ \{x = -B \wedge y = A\}$

13.6 Calcular la expresión e que haga válida la tripleta en cada caso:

a) $\{A = q \times b + r\}\ q := e; r := r - \mathtt{b}\ \{A = q \times b + r\}$.

b) $\{true\}\ y := e; x := x\, div\, 2\ \{2 \times x = y\}$.

c) $\{x \times y + p \times q = \mathtt{N}\}\ x := x - p; q := e\ \{x \times y + p \times q = \mathtt{N}\}$.

13.7 Recordemos que los números de Fibonacci $F.i$ están dados por $F.0 = 0$, $F.1 = 1$ y $F.n = F.(n-1) + F.(n-2)$ para $n \geq 2$. Sea $P : n > 0 \wedge a = F.n \wedge b = F.(n-1)$, calcular las expresiones e y f que hagan válida la tripleta:

$$\{P\}\ n, a, b := n + 1, e, f\ \{P\}$$

13.8 Demostrar que los siguientes programas son correctos, donde x, y : Integer y a, b : Boolean.

a)
$\{true\}$
if $x \geq 1 \rightarrow x := x + 1$
[] $x \leq 1 \rightarrow x := x - 1$
fi
$\{x \neq 1\}$

b)
$\{true\}$
if $x \geq y \rightarrow$ **skip**
[] $x \leq y \rightarrow x, y := y, x$
fi
$\{x \geq y\}$

c)
$\{true\}$
if $\neg a \vee b \rightarrow a := \neg a$
[] $a \vee \neg b \rightarrow b := \neg b$
fi
$\{a \vee b\}$

13.9 Encontrar el predicado P más débil que haga correcto el siguiente programa:

$\{P\}$
$x := x + 1$
if $x > 0 \rightarrow x := x - 1$
[] $x < 0 \rightarrow x := x + 2$
[] $x = 1 \rightarrow$ **skip**
fi
$\{x \geq 1\}$

13.10 Probar que las siguientes tripletas son equivalentes:

$\{P\}$
if $B_0 \rightarrow S_0; S$
[] $B_1 \rightarrow S_1; S$
fi
$\{Q\}$

$\{P\}$
if $B_0 \rightarrow S_0$
[] $B_1 \rightarrow S_1$
fi
S
$\{Q\}$

13.11 Suponiendo que el programa de la izquierda es correcto, demostrar que el de la derecha también lo es. Esto significa que se puede lograr que el if sea determinístico (solo una guarda verdadera).

$\{P\}$	$\{P\}$
if $B_0 \rightarrow S_0$	**if** $B_0 \wedge \neg B_1 \rightarrow S_0$
$[]$ $B_1 \rightarrow S_1$	$[]$ $B_1 \rightarrow S_1$
fi	**fi**
$\{Q\}$	$\{Q\}$

13.12 Demostrar que todo ciclo con más de una guarda puede reescribirse como un ciclo con una sola guarda:

Probar que el bucle

do $B_0 \rightarrow S_0$
$[]$ $B_1 \rightarrow S_1$
od

puede reescribirse como

do $B_0 \vee B_1 \rightarrow$ **if** $B_0 \rightarrow S_0$
$[]\, B_1 \rightarrow S_1$
fi
od

Este resultado es útil, pues no todos los lenguajes de programación admiten ciclos con varias guardas.

13.13 Tomando como invariante $P : x \leq n \wedge y = 2^x$ y como función cota $t : n - x$ probar que el siguiente programa es correcto:

$|[$ **var** x, y, n : **Integer**
$\{n \geq 0\}$
$x, y := 0, 1$
do $x \neq n \rightarrow x, y := x + 1, y + y$ **od**
$\{y = 2^n\}$
$]|$

13.14 Probar que el siguiente programa es correcto tomando como invariante $P : x \geq 0 \wedge y \leq n$ y como función cota $t : x + 2 \times (n - y)$.

$|[$ **var** x, y, n : **Integer**
$\{n \geq 0\}$
$x, y := 0, 0$
do $x \neq 0 \rightarrow x := x - 1$
$[]\ \ y \neq n \rightarrow x, y := x + 1, y + 1$
od
$\{x = 0 \wedge y = n\}$
$]|$

13.15 Probar la corrección del siguiente bucle, tomando $P : 1 \leq k \leq n \wedge b = fib.(k-1) \wedge c = fib.k$ y $t : n - k$.

$|[$ **var** k, b, c : **Integer**;
$\{n > 0\}$
$k, b, c := 1, 0, 1$
do $k \neq n \rightarrow k, b, c := k + 1, c, b + c$ **od**
$\{c = fib.n\}$
$]|$

donde fib es la función de Fibonacci. Explicar el propósito del programa.

13.16 Probar la corrección del siguiente bucle, tomando $P : 0 \leq i \leq n + 1 \wedge \neg(\exists\, j : 0 \leq j < i : x = b[j])$ y función cota $t : n - i$.

$|[$ **const** n : **Integer**;
var i, x : **Integer**;
b : **array**$[0 \ldots n]$ **of Integer**;
$\{n \geq 0\}$
$i := 0$
do $i \leq n \wedge x \neq b[i] \rightarrow i := i + 1$ **od**
$\{(0 \leq i \leq n \wedge x = b[i]) \vee (i = n + 1 \wedge \neg(\exists\, j : 0 \leq j \leq n : x = b[j]))\}$
$]|$

Explicar el propósito del programa.

APÉNDICE A
Axiomas y Teoremas

Índice del Capítulo

A.1. Teoremas del Cálculo Proposicional 217
A.2. Teoremas del Cálculo de Predicados 220
A.3. Cuantificación Existencial 221
A.4. Propiedades de las cuantificaciones universal y existencial 221
A.5. Metatoremas 222
A.6. Leyes Generales de la Cuantificación 222
A.7. Cuantificador max 223

A.1. Teoremas del Cálculo Proposicional

La equivalencia

(3.1) Asociatividad de $\equiv$: $((p \equiv q) \equiv r) \equiv (p \equiv (q \equiv r))$

(3.2) Conmutatividad (o simetría) de $\equiv$: $p \equiv q \equiv q \equiv p$

(3.3) Neutro de $\equiv$: $true \equiv p \equiv p$

(3.4) true

(3.5) Reflexividad de $\equiv$: $p \equiv p$

La negación, discrepancia, y false

(3.8) Definición de false: $false \equiv \neg true$

(3.9) Negación y equivalencia: $\neg (p \equiv q) \equiv \neg p \equiv q$

(3.10) Definición de $\not\equiv$: $p \not\equiv q \equiv \neg (p \equiv q)$

(3.11) $\neg p \equiv q \equiv p \equiv \neg q$

(3.12) Doble negación: $\neg\neg p \equiv p$

(3.13) Negación de $false$: $\neg false \equiv true$

(3.14) $(p \not\equiv q) \equiv \neg p \equiv q$

(3.15) $\neg p \equiv p \equiv false$

(3.16) Conmutatividad de $\not\equiv$: $(p \not\equiv q) \equiv (q \not\equiv p)$

(3.17) Asociatividad de $\not\equiv$: $((p \not\equiv q) \not\equiv r) \equiv (p \not\equiv (q \not\equiv r))$

(3.18) Asociatividad mutua: $((p \not\equiv q) \equiv r) \equiv (p \not\equiv (q \equiv r))$

(3.19) Intercambiabilidad: $p \not\equiv q \equiv r \ \equiv \ p \equiv q \not\equiv r$

La disyunción

(3.24) Asociatividad de $\vee$: $(p \vee q) \vee r \equiv p \vee (q \vee r)$

(3.25) Conmutatividad de $\vee$: $p \vee q \equiv q \vee p$

(3.26) Idempotencia: $p \vee p \equiv p$

(3.27) Distributividad de $\vee$ respecto de $\equiv$: $p \vee (q \equiv r) \equiv (p \vee q) \equiv (p \vee r)$

(3.28) Tercero excluído: $p \vee \neg p \equiv true$

(3.29) Elemento neutro de $\vee$: $p \vee false \equiv p$

(3.30) Distributividad de $\vee$ respecto de $\vee$: $p \vee (q \vee r) \equiv (p \vee q) \vee (p \vee r)$

(3.31) $p \vee q \equiv p \vee \neg q \equiv p$

(3.32) Elemento absorbente de $\vee$: $p \vee true \equiv true$

La conjunción

(3.35) Regla dorada: $p \wedge q \equiv p \equiv q \equiv p \vee q$

(3.36) Asociatividad de $\wedge$: $p \wedge (q \wedge r) \equiv (p \wedge q) \wedge r$

(3.37) Conmutatividad de $\wedge$: $p \wedge q \equiv q \wedge p$

(3.38) Idempotencia de $\wedge$: $p \wedge p \equiv p$

(3.39) Neutro de $\wedge$: $p \wedge true \equiv p$

(3.40) Elemento absorbente de $\wedge$: $p \wedge false \equiv false$

(3.41) Distributividad de $\wedge$ sobre $\wedge$: $p \wedge (q \wedge r) \equiv (p \wedge q) \wedge (p \wedge r)$

(3.42) Contradicción: $p \wedge \neg p \equiv false$

(3.43) Absorción: **a)** $p \wedge (p \vee q) \equiv p$
b) $p \vee (p \wedge q) \equiv p$

(3.44) Absorción: **a)** $p \wedge (\neg p \vee q) \equiv p \wedge q$
b) $p \vee (\neg p \wedge q) \equiv p \vee q$

(3.45) Distributividad de $\wedge$ con respecto a $\vee$: $p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$

(3.46) Distributividad de $\vee$ con respecto a $\wedge$: $p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$

(3.47) Leyes de Morgan **a)** $\neg (p \vee q) \equiv \neg p \wedge \neg q$
b) $\neg (p \wedge q) \equiv \neg p \vee \neg q$

(3.48) $p \wedge q \equiv p \wedge \neg q \equiv \neg p$

(3.49) $p \wedge (q \equiv r) \equiv p \wedge q \equiv p \wedge r \equiv p$
(3.50) $p \wedge (q \equiv p) \equiv p \wedge q$
(3.51) Reemplazo: $(p \equiv q) \wedge (r \equiv p) \equiv (p \equiv q) \wedge (r \equiv q)$
(3.52) Definición alternativa de $\equiv$: $p \equiv q \equiv (p \wedge q) \vee (\neg p \wedge \neg q)$
(3.53) "O" exclusivo: $p \not\equiv q \equiv (\neg p \wedge q) \vee (p \wedge \neg q)$

La implicación

(3.56) Definición de implicación: $p \Rightarrow q \equiv p \vee q \equiv q$
(3.57) Definición de consecuencia: $p \Leftarrow q \equiv p \vee q \equiv p$
(3.58) Definición alternativa de implicación: $p \Rightarrow q \equiv \neg p \vee q$
(3.59) Definición alternativa de implicación: $p \Rightarrow q \equiv p \wedge q \equiv p$
(3.60) Contrarrecíproco: $p \Rightarrow q \equiv \neg q \Rightarrow \neg p$
(3.61) $p \Rightarrow (q \equiv r) \equiv p \wedge q \equiv p \wedge r$
(3.62) Distributividad de $\Rightarrow$ respecto de $\equiv$: $p \Rightarrow (q \equiv r) \equiv p \Rightarrow q \equiv p \Rightarrow r$
(3.63) $p \Rightarrow (q \Rightarrow r) \equiv (p \Rightarrow q) \Rightarrow (p \Rightarrow r)$
(3.64) Traslación: $p \wedge q \Rightarrow r \equiv p \Rightarrow (q \Rightarrow r)$
(3.65) $p \wedge (p \Rightarrow q) \equiv p \wedge q$
(3.66) $p \wedge (q \Rightarrow p) \equiv p$
(3.67) $p \vee (p \Rightarrow q) \equiv true$
(3.68) $p \vee (q \Rightarrow p) \equiv q \Rightarrow p$
(3.69) $p \vee q \Rightarrow p \wedge q \equiv p \equiv q$
(3.70) Reflexividad de $\Rightarrow$: $p \Rightarrow p \equiv true$
(3.71) Elemento absorbente a derecha de $\Rightarrow$: $p \Rightarrow true \equiv true$
(3.72) Elemento neutro a izquierda de $\Rightarrow$: $true \Rightarrow p \equiv p$
(3.73) $p \Rightarrow false \equiv \neg p$
(3.74) $false \Rightarrow p \equiv true$
(3.75) Debilitamiento o fortalecimiento
a) $p \Rightarrow p \vee q$
b) $p \wedge q \Rightarrow p$
c) $p \wedge q \Rightarrow p \vee q$
d) $p \vee (q \wedge r) \Rightarrow p \vee q$
e) $p \wedge q \Rightarrow p \wedge (q \vee r)$
(3.76) Modus Ponens: $p \wedge (p \Rightarrow q) \Rightarrow q$
(3.77) $(p \Rightarrow r) \wedge (q \Rightarrow r) \equiv (p \vee q \Rightarrow r)$
(3.78) $(p \Rightarrow r) \wedge (\neg p \Rightarrow r) \equiv r$
(3.79) Implicación mutua: $(p \Rightarrow q) \wedge (q \Rightarrow p) \equiv p \equiv q$
(3.80) Antisimetría: $(p \Rightarrow q) \wedge (q \Rightarrow p) \Rightarrow (p \equiv q)$

(3.81) Transitividad: **a)** $(p \Rightarrow q) \wedge (q \Rightarrow r) \Rightarrow (p \Rightarrow r)$
b) $(p \equiv q) \wedge (q \Rightarrow r) \Rightarrow (p \Rightarrow r)$
c) $(p \Rightarrow q) \wedge (q \equiv r) \Rightarrow (p \Rightarrow r)$

La Regla de leibniz como axioma

(3.82) $(e = f) \Rightarrow E\,[z := e] = E\,[z := f]$

(3.84) Reglas de Sustitución: **a)** $(e = f) \wedge E\,[z := e] \equiv (e = f) \wedge E\,[z := f]$
b) $(e = f) \Rightarrow E\,[z := e] \equiv (e = f) \Rightarrow E\,[z := f]$
c) $q \wedge (e = f) \Rightarrow E\,[z := e] \equiv q \wedge (e = f) \Rightarrow E\,[z := f]$

(3.85) Reemplazo por $true$: **a)** $p \Rightarrow E\,[z := p] \equiv p \Rightarrow E\,[z := true]$
b) $q \wedge p \Rightarrow E\,[z := p] \equiv q \wedge p \Rightarrow E\,[z := true]$

(3.86) Reemplazo por $false$ **a)** $E\,[z := p] \Rightarrow p \equiv E\,[z := false] \Rightarrow p$
b) $E\,[z := p] \Rightarrow q \vee p \equiv E\,[z := false] \Rightarrow q \vee p$

(3.87) Reemplazo por $true$: $p \wedge E\,[z := p] \equiv p \wedge E\,[z := true]$

(3.88) Reemplazo por $false$: $p \vee E\,[z := p] \equiv p \vee E\,[z := false]$

(3.89) Shannon: $E\,[z := p] \equiv (p \wedge E\,[z := true]) \vee (\neg p \wedge E\,[z := false])$

(4.1) $p \Rightarrow (q \Rightarrow p)$

(4.2) Monotonía del $\vee$: $(p \Rightarrow q) \Rightarrow (p \vee r \Rightarrow q \vee r)$

(4.3) Monotonía del $\wedge$: $(p \Rightarrow q) \Rightarrow (p \wedge r \Rightarrow q \wedge r)$

A.2. Teoremas del Cálculo de Predicados

Cuantificación Universal

(5.1) Rango $true$: $(\forall\, x : true : T.x) \equiv (\forall\, x :: T.x)$

(5.2) Rango unitario: $(\forall\, x : x = N : T.x) \equiv T.N$

(5.3) Rango vacío: $(\forall\, x : false : T.x) \equiv true$

(5.4) Distributividad de $\vee$ respecto de $\forall$: $(\forall\, x : R.x : P \vee T.x) \equiv P \vee (\forall\, x : R.x : T.x)$, donde x no aparece en P.

(5.5) Regla del término: $(\forall\, x : R.x : T.x \ \wedge G.x) \equiv (\forall\, x : R.x : T.x) \wedge (\forall\, x : R.x : G.x)$

(5.6) Intercambio de cuantificadores: $(\forall\, x :: (\forall\, y :: T.x.y)) \equiv (\forall\, y :: (\forall\, x :: T.x.y))$

(5.7) $(\forall\, x :: (\forall\, y :: T.x.y)) \equiv (\forall\, x,\, y :: T.x.y)$

(5.8) Traslación: $(\forall\, x : R.x : T.x) \equiv (\forall\, x : : R.x \Rightarrow T.x)$

(5.9) Traslación:
a) $(\forall\, x : R.x : P.x) \equiv (\forall\, x :: \neg R.x \vee P.x)$
b) $(\forall\, x : R.x : P.x) \equiv (\forall\, x :: R.x \wedge P.x \equiv R.x)$
c) $(\forall\, x : R.x : P.x) \equiv (\forall\, x :: R.x \vee P.x \equiv P.x)$

(5.10) Variantes de traslación:

a) $(\forall\, x : Q.x \wedge R.x : P.x) \equiv (\forall\, x : Q.x : R.x \Rightarrow P.x)$
b) $(\forall\, x : Q.x \wedge R.x : P.x) \equiv (\forall\, x : Q.x : \neg R.x \vee P.x)$
c) $(\forall\, x : Q.x \wedge R : P.x) \equiv (\forall\, x : Q.x : R.x \wedge P.x \equiv R.x)$
d) $(\forall\, x : Q.x \wedge R.x : P.x) \equiv (\forall\, x : Q.x : R.x \vee P.x \equiv P.x)$

(5.11) Cambio de variable: Si y no ocurre en $R.x$ ni en $T.x$ entonces $(\forall\, x : R.x : T.x) \equiv (\forall\, y : R.y : T.y)$

(5.12) Partición de rango: $(\forall\, i : R.i \vee S.i : T.i) \equiv (\forall\, i : R.i : T.i) \;\wedge\; (\forall\, i : S.i : T.i)$

(5.13) Instanciación: $(\forall\, x :: P.x) \Rightarrow P.E$

A.3. Cuantificación Existencial

(5.14) De Morgan Generalizado: $(\exists\, x : R.x : T.x) \equiv \neg(\forall\, x : R.x : \neg T.x)$

(5.15) Formas alternativas de De Morgan Generalizado:
a) $\neg(\exists\, x : R : \neg P) \equiv (\forall\, x : R : P)$
b) $\neg(\exists\, x : R : P) \equiv (\forall\, x : R : \neg P)$
c) $(\exists\, x : R : \neg P) \equiv \neg(\forall\, x : R : P)$

(5.16) Rango $true$: $(\exists\, x : true : T.x) \equiv (\exists\, x :: T.x)$

(5.17) Rango unitario: $(\exists\, x : x = N : T.x) \equiv T.N$, donde x no aparece en la expresión N.

(5.18) Rango vacío: $(\exists\, x : false : T.x) \equiv false$

(5.19) Distributividad de $\wedge$ respecto de $\exists$: $(\exists\, x : R.x : X \wedge T.x) \equiv X \wedge (\exists\, x : R.x : T.x)$, donde x no aparece en X.

(5.20) Regla del término: $(\exists\, x : R.x : T.x \vee G.x) \equiv (\exists\, x : R.x : T.x) \vee (\exists\, x : R.x : G.x)$

(5.21) Intercambio de cuantificadores: $(\exists\, x :: (\exists\, y :: T.x.y)) \equiv (\exists\, y :: (\exists\, x :: T.x.y))$

(5.22) $(\exists\, x :: (\exists\, y :: T.x.y)) \equiv (\exists\, x,\, y :: T.x.y)$

(5.23) Traslación: $(\exists\, x : R.x : T.x) \equiv (\exists\, x :: R.x \wedge T.x)$

(5.24) Traslación: $(\exists\, x : R.x \wedge Q.x : T.x) \equiv (\exists\, x : R.x : Q.x \wedge T.x)$

(5.25) Intercambio entre rango y término: $(\exists\, x : R.x : T.x) \equiv (\exists\, x : T.x : R.x)$

(5.26) Partición de rango: $(\exists\, i : R.i \vee S.i : T.i) \equiv (\exists\, i : R.i : T.i) \;\vee\; (\exists\, i : S.i : T.i)$

(5.27) Introducción de $\exists$: $P.E \Rightarrow (\exists\, x :: P.x)$

(5.28) Cambio de variable: Si y no ocurre en R o en T entonces,
$(\exists\, x : R.x : T.x) \equiv (\exists\, y : R.y : T.y)$

A.4. Propiedades de las cuantificaciones universal y existencial

(5.29) Fortalecimiento de rango: $(\forall\, x : Q.x \vee R.x : P.x) \Rightarrow (\forall\, x : Q.x : P.x)$

(5.30) Fortalecimiento de término: $(\forall\, x : R.x : P.x \wedge Q.x) \Rightarrow (\forall\, x : R.x : P.x)$

(5.7.2) Monotonía de $\forall$: $(\forall\, x : R.x : Q.x \Rightarrow P.x) \Rightarrow ((\forall\, x : R.x : Q.x) \Rightarrow (\forall\, x : R.x : P.x))$

(5.32) Distributividad de $\wedge$ respecto de $\forall$: Si x no ocurre en P y el rango de especificación es no vacío, es decir $(\exists\, x :: R.x)$, entonces $(\forall\, x : R.x : P \wedge Q.x) \equiv P \wedge (\forall\, x : R.x : Q.x))$

(5.33) $(\forall\, x : R.x : true) \equiv true$

(5.34) $(\forall\, x : R.x : P.x \equiv Q.x) \Rightarrow ((\forall\, x : R.x : P.x) \equiv (\forall\, x : R.x : Q.x))$

(5.35) Debilitamiento de rango: $(\exists\, x : R.x : P.x) \Rightarrow (\exists\, x : R.x \vee Q.x : P.x)$

(5.36) Debilitamiento de término: $(\exists\, x : R.x : P.x) \Rightarrow (\exists\, x : R.x : P.x \vee Q.x)$

(5.37) Monotonía de $\exists$: $(\forall\, x : R.x : Q.x \Rightarrow P.x) \Rightarrow ((\exists\, x : R.x : Q.x) \Rightarrow (\exists\, x : R.x : P.x))$

(5.38) Distributividad de $\vee$ respecto de $\exists$: Si x no ocurre en P y el rango de especificación es no vacío, es decir $(\exists\, x :: R.x)$, entonces $(\exists\, x : R.x : P \vee Q.x) \equiv P \vee (\exists\, x : R.x : Q.x))$

(5.39) $(\exists\, x : R.x : false) \equiv false$

(5.40) Intercambio de cuantificadores: $(\exists\, x : R.x : (\forall\, y : Q.y : P.x.y)) \Rightarrow (\forall\, y : Q.y : (\exists\, x : R.x : P.x.y))$

A.5. Metatoremas

(5.41) Testigo: Si k no ocurre en P ni en Q, entonces, $(\exists\, x :: P.x) \Rightarrow Q$ es un teorema si y solo si $P.k \Rightarrow Q$ es un teorema.

A.6. Leyes Generales de la Cuantificación

Para un operador $\oplus$ simétrico y asociativo, con elemento neutro u.

(6.7) Sustitución para expresiones cuantificadas. $V.y \cap (V.x \cup FV.E) = \emptyset \Rightarrow (\oplus\, y : R : T)\,[x := E] = (\oplus\, y : R\,[x := E] : T\,[x := E])$

(6.9) Rango vacío: $(\oplus\, i : false : T) = u$

(6.10) Rango unitario:

Si i no es una variable libre en la expresión N, entonces $(\oplus\, i : i = N : T.i) = T.N$

(6.12) Distributividad:

Si el operador $\otimes$ es distributivo a izquierda con respecto a $\oplus$, $i \notin FV.E$ y se cumple al menos una de las siguientes condiciones:

a) el rango de especificación es no vacío,

b) el elemento neutro del operador $\oplus$ existe y es absorbente para $\otimes$

entonces $(\oplus\, i : R : E \otimes T) = E \otimes (\oplus\, i : R : T)$

(6.13) Partición de rango:

Si alguna de las siguientes condiciones es cierta:

a) el operador $\oplus$ es idempotente

b) $R \wedge S = false$

se cumple que $(\oplus\, i : R \vee S : T) = (\oplus\, i : R : T) \oplus (\oplus\, i : S : T)$.

(6.15) Partición de Rango: $(\oplus\, i : R \vee S : P) \oplus (\oplus\, i : R \wedge S : P) = (\oplus\, i : R : P) \oplus (\oplus\, i : S : P)$.

(6.16) Partición de Rango generalizada: Si el operador operador $\oplus$ es idempotente, entonces, $(\oplus\, i : (\exists\, j : S.i.j : R.i.j) : T.i) = (\oplus\, i, j : S.i.j \wedge R.i.j : T.i)$

(6.18) Regla del término constante: Si el término de la cuantificación es igual a una constante C, el operador $\oplus$ es idempotente y el rango de especificación es no vacío, entonces $(\oplus\, i : R : C) = C$

(6.19) Regla de anidado: $(\oplus\, i, j : R.i \wedge S.i.j : T.i.j) = (\oplus\, i : R.i : (\oplus j : S.i.j : T.i.j))$

(6.20) Regla de intercambio de variables: Si $V.j \cap FV.R = \emptyset$ y $V.i \cap FV.Q = \emptyset$, entonces $(\oplus\, i : R : (\oplus j : Q : T)) = (\oplus j : Q : (\oplus\, i : R : T))$

(:) Regla de cambio de variable[6.21 Si $V.j \cap (FV.R \cup FV.T) = \emptyset$ entonces, $(\oplus\, i : R : T) = (\oplus j : R[i := j\ T[i := j])$

(6.23) Regla del cambio de variable:

Sea f una función biyectiva definida sobre R y j una variable que no aparece libre en R ni en T, entonces, $(\oplus\, i : R.i : T.i) = (\oplus j : R.(f.j) : T.(f.j))$

(6.24) Separación de un término: Sea $n < m$,

a) $(\oplus\, i : n \leq i < m : T.i) = T.n \oplus (\oplus\, i : n < i < m : T.i)$

b) $(\oplus\, i : n < i \leq m : T.i) = (\oplus\, i : n < i < m : T.i) \oplus T.m$

A.7. Cuantificador max

(6.26) Rango vacío: $(max\ i : false : T.i) = -\infty$

(6.27) Distributividad de min sobre max:

Suponiendo que i no es variable libre en E,
$min.(E, (max\ i : R : T)) = (max\ i : R : min.(E, T))$

(6.28) Distributividad de $+$ sobre max:

Suponiendo que i no es variable libre en E y que $R \neq false$,
$E + (max\ i : R : T) = (max\ i : R : E + T)$

(6.29) Partición de rango: $(max\ i : R \vee S : F) = max.((max\ i : R : F), (max\ i : S : F))$

(6.30)
$$T.x = (max\ i : R.i : T.i) \equiv R.x \wedge (\forall\ i : R.i : T.i \leq T.x)$$
$$T.x = (min\ i : R.i : T.i) \equiv R.x \wedge (\forall\ i : R.i : T.i \geq T.x)$$