Cálculo de Programas

Javier Blanco Silvina Smith Damián Barsotti

Facultad de Matemática, Astronomía y Física
Universidad Nacional de Córdoba

e-mail: {blanco|smith}@mate.uncor.edu
damian@famaf.unc.edu.ar

20 de marzo de 2006

Índice general

1. Introducción **1**
- 1.1. ¿Qué se puede aprender de una torta? 2
- 1.2. Breve descripción de lo que sigue 5
- 1.3. Ejercicios 6

2. Preliminares **7**
- 2.1. Expresiones aritméticas 7
- 2.2. Sustitución 8
- 2.3. Igualdad y regla de Leibniz 9
- 2.4. Funciones 12
- 2.5. Ejercicios 13

3. Cálculo Proposicional **15**
- 3.1. Sistemas formales 16
- 3.2. La equivalencia 18
- 3.3. La negación 20
- 3.4. La discrepancia 20
- 3.5. La disyunción 21
- 3.6. La conjunción 22
- 3.7. La implicación 25
- 3.8. La consecuencia 26
- 3.9. Generalización del formato de demostración 27
- 3.10. Ejercicios 28

4. Aplicaciones del cálculo proposicional **31**
- 4.1. Análisis de argumentaciones 31
- 4.2. Resolución de acertijos lógicos 34
- 4.3. Ejercicios 35

5. Aplicaciones del cálculo proposicional 2: Piso y Techo **37**
- 5.1. La función piso 37
- 5.2. Igualdad indirecta 39
- 5.3. La función techo 40
- 5.4. Ejercicios 42

6. Expresiones cuantificadas **45**
6.1. Introducción . . . 46
6.2. Revisión de la sustitución y la regla de Leibniz . . . 48
6.3. Reglas generales para las expresiones cuantificadas . . . 49
6.4. Cuantificadores aritméticos . . . 52
6.5. Expresiones cuantificadas para conjuntos . . . 53
6.6. El cuantificador universal . . . 54
6.7. El cuantificador existencial . . . 55
6.8. Ejercicios . . . 56

7. Cálculo de predicados **59**
7.1. Predicados . . . 59
7.2. Propiedades de las cuantificaciones universal y existencial . . . 60
7.3. Aplicaciones del cálculo de predicados . . . 62
7.4. Algunas conclusiones . . . 62
7.5. Dios y la lógica* . . . 63
7.6. Ejercicios . . . 65

8. El formalismo básico **67**
8.1. Funciones . . . 67
8.2. Definiciones y expresiones . . . 69
8.3. Reglas para el cálculo con definiciones . . . 70
8.4. Definiciones locales . . . 71
8.5. Análisis por casos . . . 71
8.6. Pattern Matching . . . 73
8.7. Tipos . . . 74
8.8. Tipos básicos . . . 74
8.9. Tuplas . . . 75
8.10. Listas . . . 76
8.10.1. Constructores de listas . . . 76
8.10.2. Operaciones sobre listas . . . 77
8.10.3. Propiedades de las operaciones . . . 78
8.11. Ejercicios . . . 79

9. Modelo Computacional **81**
9.1. Introducción . . . 81
9.2. Valores . . . 82
9.3. Forma Canónica . . . 83
9.4. Evaluación . . . 85
9.5. Un modelos computacional más eficiente . . . 88
9.6. Nociones de eficiencia de programas funcionales . . . 90
9.7. Problema de la forma normal . . . 90
9.8. Ejercicios . . . 91

Capítulo 1

Introducción

> History, Stephen said, is a nightmare from which I am trying to awake.
>
> James Joyce: *Ulysses*

La programación es una actividad que ofrece desafíos intelectuales interesantes, en la cual se combinan armónicamente la creatividad y el razonamiento riguroso. Lamentablemente no siempre es enseñada de esta manera. Muchos cursos de introducción a la programación se basan en el "método" de ensayo y error. Las construcciones de los lenguajes de programación son presentadas sólo operacionalmente, se estudia un conjunto de ejemplos y se espera resolver problemas nuevos por analogía, aún cuando estos problemas sean radicalmente diferentes de los presentados. Se asume a priori que los programas desarrollados con este método tendrán errores y se dedica una buena parte del tiempo de programación a encontrarlos y corregirlos. Es así que esta ingrata tarea se denomina eufemísticamente *debugging*, es decir eliminación de los "bichos" del progama, como si estos bichos hubieran entrado al programa involuntariamente infectando al programa sano. En este libro hablaremos simplemente de *errores* introducidos en el proceso de programación. En los libros de ingeniería del software se suele dedicar un considerable tiempo a explicar la actividad de corrección de errores, y se le asigna un peso muy relevante en el proceso de desarrollo de programas. Sin embargo, pese a todos los esfuerzos que se hagan, nunca se puede tener una razonable seguridad de que todos los errores hayan sido encontrados o de que alguna de las "reparaciones" de errores viejos no haya introducido otros nuevos (el llamado efecto 'Hydra' en [BEKV94]).

Este estado de cosas puede entenderse si se analiza la historia de la disciplina. En el inicio de la computación, la función de un programa era dar las instrucciones para que una máquina ejecutara una cierta tarea. Hoy resulta más provechoso pensar, inversamente, que es la función de las máquinas ejecutar nuestros programas, poniendo el énfasis en los programas como construcciones fundamentales. De la misma manera, los lenguajes de programación fueron cambiando su formulación y forma de diseño. Inicialmente, el diseño de los lenguajes de programación y de su semántica era una tarea esencialmente descriptiva, la cual intentaba modelar las operaciones que ocurrían en una máquina durante la ejecución de un programa. Esto es lo que daba lugar a definiciones operacionales de los lenguajes y a la imposibilidad de razonar con ellos, excepto a través de la ejecución 'a mano' de los programas, lo cual es una herramienta de probada ineficiencia.

Existe, afortunadamente, otra forma de aproximarse a la programación. Los programas pueden ser desarrollados de manera metódica a partir de *especificaciones*, de tal manera que la corrección del programa obtenido respecto de la especificación original pueda asegurarse por la forma en que el programa fue construido. Además de la utilidad práctica de dicha forma de programar, el ejercicio intelectual que ésta presenta es altamente instructivo acerca de las sutilezas de la resolución de problemas mediante algoritmos. Por último, vista de esta forma, la programación es una tarea divertida.

No estamos afirmando que el proceso de *debugging* pueda evitarse completamente, pero si se dispone de una notación adecuada para expresar los problemas a resolver y de herramientas simples y poderosas para asegurar la adecuación de un programa a su especificación, los errores serán en general más fáciles de corregir, dado que es mucho menos probable que dichos errores provengan de una mala comprensión del problema a resolver.

En este punto la lógica matemática es una herramienta indispensable como ayuda para la especificación y desarrollo de programas. La idea subyacente es *deducir* los programas a partir de una especificación formal del problema, entendiéndose ésto como un predicado sobre algún universo de valores. De esta manera, al terminar de escribir el programa, no se tendrá solamente 'un programa', sino también la demostración de que el mismo es correcto respecto de la especificación dada, es decir, resuelve el problema planteado. Veamos un ejemplo para aclarar estos conceptos. El mismo está tomado E.W.Dijkstra [Dij89] donde la solución se atribuye a A.Blokhuis. La idea de presentar los inconvenientes del razonamiento por ensayo y error usando este ejemplo está tomada de [Coh90].

1.1. ¿Qué se puede aprender de una torta?

Problema: En una torta circular se marcan N puntos sobre el contorno y luego se trazan todas las cuerdas posibles entre ellos. Se supone que nunca se cortan más de dos cuerdas en un mismo punto interno. ¿Cuántas porciones de torta se obtienen?

Antes de seguir leyendo, resuelva el problema de la torta.

Analizaremos los casos N=1, N=2, N=3, etc., tratando de inferir una formulación válida para N arbitrario:
Si N=1, obtenemos 1 pedazo. Si N=2, obtenemos 2 pedazos.

Número de puntos	1	2
Número de porciones	1	2

Avancemos un poco más. Si N=3, obtenemos 4 pedazos. Para tratar de tener más información tomamos N=4 y obtenemos 8 pedazos.

Número de puntos	1	2	3	4
Número de porciones	1	2	4	8

Parecería que está apareciendo un patrón, dado que las cantidades de porciones son las sucesivas potencias de 2. Proponemos entonces como posible solución que la cantidad de pedazos es 2^{N-1}. Probemos con N=5. Efectivamente, se obtienen 16 pedazos (¡parece que 2^{N-1} es correcto!).

Número de puntos	1	2	3	4	5
Número de porciones	1	2	4	8	16

Para estar más seguros probemos con el caso N=6, para el cual deberíamos hallar N=32.

Número de puntos	1	2	3	4	5	6
Número de porciones	1	2	4	8	16	31

Si N=6, obtenemos 31 pedazos (¡2^{N-1} no sirve!).

Podemos probar con N = 7, a ver si reaparece algún patrón útil.

Número de puntos	1	2	3	4	5	6	7
Número de porciones	1	2	4	8	16	31	57

Probablemente ya estamos bastante cansados de contar.

Claramente, el procedimiento seguido no es el más indicado, pues no conduce a la solución del problema. Peor aún, puede conducir a un resultado erróneo (por ejemplo, si hubiésemos analizado sólo hasta N=5).

Podría resumirse el método usado como: adivine y asuma que adivinó bien hasta que se demuestre lo contrario.

Lamentablemente el método de ensayo y error, el cual fracasó para este ejemplo, es usado frecuentemente en programación. El problema principal de este método es que a través de adivinaciones erróneas se aprende muy poco acerca de la naturaleza del problema. En nuestro ejemplo, lo único que aprendimos es que el problema es más difícil de lo que podría esperarse, pero no descubrimos ninguna propiedad interesante que nos indique algún camino para encontrar una solución. Por otro lado, en general no es simple saber si se ha adivinado correctamente. En nuestro caso, la "solución" falló para N = 6, pero podría haber fallado para N = 30. Otro problema metodológico es la tendencia, demasiado frecuente en programación, a corregir adivinaciones erróneas a través de adaptaciones superficiales que hagan que la solución (el programa) "funcione", ensayando algunos casos para "comprobarlo". Este método de ensayo y error no es adecuado para la construcción de programas correctos, además, como ya lo enunciara Dijkstra en [Dij76] convirtiéndose luego en un slogan:

> *Los ensayos (tests) pueden revelar la presencia de errores, pero nunca demostrar su ausencia.*

Pensemos en una solución adecuada para el problema anterior. ¿En cuánto se incrementa la cantidad de porciones al agregar una cuerda? Como cada cuerda divide cada porción que atraviesa en dos partes, una cuerda agrega una porción por cada porción que atraviesa. A este razonamiento lo escribiremos de la siguiente manera:

número de porciones agregadas al agregar una cuerda
= { cuerdas dividen porciones en dos }
número de porciones cortadas por la cuerda

Entre llaves se coloca la explicación del vínculo entre las dos afirmaciones (en este caso el vínculo es el signo =). Completemos el razonamiento:

número de porciones agregadas al agregar una cuerda
= { cuerdas dividen porciones en dos }
número de porciones cortadas por la cuerda
= { una porción se divide por un segmento de cuerda }
número de segmentos de la cuerda agregada
= { los segmentos están determinados por puntos de intersección internos }
1 + número de puntos de intersección internos sobre la cuerda

Una vez que tenemos este resultado podemos calcular en cuánto se incrementa el número de porciones si se agregan c cuerdas.

número de porciones agregadas al agregar c cuerdas
= { resultado anterior y el hecho que cada punto de intersección interno es compartido por exactamente dos cuerdas }
c + total de puntos de intersección internos en las c cuerdas.

Dado que si comenzamos con 0 puntos tenemos una porción (toda la torta) y agregar c cuerdas nos incrementa el número en c + (total de puntos de intersección internos), si definimos

$f =$ número de porciones
$c =$ número de cuerdas
$p =$ número de puntos de intersección internos

hemos deducido que

$$f = 1 + c + p$$

En este punto hemos podido expresar el problema original en téminos de la cantidad de cuerdas y de puntos de intersección internos. Falta expresar ahora estas cantidades en términos de la cantidad de puntos sobre la circunferencia. Ésto puede lograrse con relativa facilidad usando principios geométricos y combinatorios elementales. Para terminar, entonces, tenemos que expresar f en términos de N.

f
= { f=número de porciones, c=número de cuerdas agregadas, p=número de puntos de intersección internos }
$1 + c + p$
= { una cuerda cada dos puntos }
$1 + \binom{N}{2} + p$

$=$ { un punto de intersección interna cada 4 puntos en la circunferencia }

$$1 + \binom{N}{2} + \binom{N}{4}$$

$=$ { álgebra }

$$1 + \frac{N^4 - 6N^3 + 23N^2 - 18N}{24}$$

Observemos que además de haber obtenido un resultado, también tenemos una demostración de que el mismo es correcto.

Nótese que si alguien nos hubiera propuesto el resultado final como solución al problema no habríamos estado del todo convencidos de que es el resultado correcto. La convicción de la corrección del resultado está dada por el desarrollo que acompaña al mismo. Algo análogo ocurre con los programas.

¿Cuáles fueron las razones que hicieron al desarrollo convincente? Por un lado, los pasos del desarrollo fueron explícitos, no fue necesario descomponerlos en componentes más elementales. Éstos tuvieron además un tamaño adecuado. Otro factor importante fue la precisión; cada paso puede verse como una manipulación de fórmulas, en este caso aritméticas.

1.2. Breve descripción de lo que sigue

El estilo de demostración presentado en la sección precedente tiene la ventaja de ser bastante autocontenido y de ser explícito cuando se usan resultados de otras áreas (en este caso del análisis combinatorio y la geometría). Ésto hace a las demostraciones muy fáciles de leer. Puede parecer exagerado poner tanto énfasis en cuestiones de estilo, pero dada la cantidad de manipulaciones formales que es necesario realizar cuando se programa, estas cuestiones pueden significar el éxito o no de un cierto método de construcción de programas.

Las manipulaciones formales de símbolos son inevitables en la programación, dado que un programa es un objeto formal construido usando reglas muy precisas. Cuando se usa el método de ensayo y error no se evitan las manipulaciones formales, simplemente se realizan usando como única guía cierta intuición operacional. Ésto no sólo vuelve más difícil la construcción de dichos programas, sino que atenta contra la comprensión y comunicación de los mismos, aún en el caso en que éstos sean correctos. La misma noción de corrección es difícil de expresar si no se tiene alguna manera formal de expresar las especificaciones.

En lo que sigue del libro se presentará un formalismo que permite escribir especificaciones y programas y demostrar que un programa es correcro respecto de una especificación dada. Este formalismo estará basado en un estilo ecuacional de presentar la lógica matemática usual. Se dispondrá de un repertorio de reglas explícitas para manipular expresiones, las cuales pueden representar tanto especificaciones como programas. El hecho de limitarse a un conjunto predeterminado de reglas para razonar con los programas no será necesariamente una limitación para construirlos. Lo que se consigue con esto es acotar las opciones en las derivaciones, lo cual permite que sea la misma forma de las especificaciones la que nos guíe en la construcción de los programas. Por otro lado, al explicitar el lenguaje que se usará para construir y por lo tanto para expresar los programas y las demostraciones, se reduce el "ancho de banda" de la comunicación, haciendo que sea más fácil comprender lo realizado por otro programador y convencerse de la corrección de un programa dado.

Concluimos esta sección con una cita pertinente.

> *It is an error to believe that rigor in the proof is an enemy of simplicity. On the contrary, we find confirmed in numerous examples that the rigorous method is at the same time the simpler and the more easily comprehended. The very effort of rigor forces us to discover simpler methods of proof. It also frequently leads the way to methods which are more capable of development than the old methods of less rigor.*
>
> David Hilbert

1.3. Ejercicios

Ejercicio 1.1
Resolver a través de un cálculo análogo al usado para el problema de la torta el siguiente problema (mucho más simple) extraído de [Coh90]. El objetivo del ejercicio es construir una demostración al estilo de la usada para la torta, con pasos cortos y justificados.

> Un tren que avanza a 70 Km/h se cruza con otro tren que avanza en sentido contrario a 50 Km/h. Un pasajero que viaja en el segundo tren ve pasar al primero durante un lapso de 6 segundos. ¿Cuántos metros mide el primer tren?

Capítulo 2

Preliminares

> We must not, however, overlook the fact that human calculation is *also* an operation of nature, but just as trees do not represent or symbolize rocks our thoughts – even if intended to do so – do not necesarily represent trees and rocks (...) Any correspondence between them is abstract.
>
> Alan Watts: *Tao, the watercourse way*

2.1. Expresiones aritméticas

A fin de contar con las herramientas básicas para poder manipular expresiones, introduciremos algunos conceptos básicos, como los de estado, evaluación y sustitución. Estos conceptos tienen sentido para diferentes conjuntos de expresiones, por ejemplo expresiones aritméticas, booleanas, de conjuntos, etc. A modo de ejemplo consideremos las expresiones aritméticas. Una sintaxis posible para éstas puede ser la presentada en la siguiente definición:

- Una constante es una expresión aritmética (e.g. 42)
- Una variable es una expresión aritmética (e.g. x)
- Si E es una expresión aritmética, entonces (E) también lo es.
- Si E es una expresión aritmética, entonces $-E$ también lo es.
- Si E y F son expresiones aritméticas, también lo serán $(E+F)$, $(E-F)$, $(E*F)$ y (E/F).
- Sólo son expresiones las construidas usando las reglas precedentes.

La reglas enunciadas permiten la construcción de expresiones aritméticas. Por ejemplo la expresión $((3 + x) * 2)$ pudo haber sido construida usando la siguiente secuencia de pasos: se construyen las expresiones 3 por la primera regla y x por la segunda. Se aplica luego la regla del $+$ para construir $(3 + x)$. Por último se construye la expresión 2 y se juntan ambas con la regla que permite introducir el operador de producto, obteniendo finalmente la expresión $((3+x)*2)$.

Precedencia. Para abreviar las expresiones usando menos paréntesis se suelen usar *reglas de precedencia*. Estas reglas establecen cómo leer una expresión en los casos en los que pueda resultar ambigua. Por ejemplo, en la aritmética se entiende que la expresión $7*5+7$ debe leerse como $(7*5)+7$ y no como $7*(5+7)$. En este caso se dice que el operador $*$ tiene precedencia sobre el $+$, es decir que liga de manera más fuerte las expresiones que une. La precedencia que asumiremos para los operadores aritméticos es la usual. La expresión construida anteriormente puede abreviarse como $(3+x)*2$.

Definición 2.1
Una *asignación* de valores a las variables de una expresión es una función del conjunto de variables en el conjunto de números pertinente. Otra expresión usada frecuentemente para este concepto es *estado*, sobre todo en informática.

Definición 2.2
Dada una asignación de valores a las variables de una expresión, puede realizarse la *evaluación* de ésta, es decir, calcular el valor asociado a dicha expresión a través de la asignación de valores a las variables. Por ejemplo, la expresión $x*5+y$ evaluada en el estado $\{(x,8),(y,2)\}$ da como resultado el número 42, el cual se obtiene de multiplicar por cinco el valor de x y luego sumarle el valor de y.

2.2. Sustitución

Sean E y F dos expresiones y sea x una variable. Usaremos la notación

$$E(x := F)$$

para denotar la expresión que es igual a E donde todas las ocurrencias de x han sido reemplazadas por (F). Cuando los paréntesis agregados a F sean innecesarios nos tomaremos la libertad de borrarlos sin avisar. Por ejemplo,

$$(x+y)(y := 2*z) = (x+(2*z)) = x+2*z$$

Es posible sustituir simultáneamente una lista de variables por una lista de expresiones de igual longitud. Esto no siempre es equivalente a realizar ambas sustituciones en secuencia. Por ejemplo:

$$(x+z)(x,z := z+1,4) = z+1+4$$

mientras que

$$(x+z)(x := z+1)(z := 4) = (z+1+z)(z := 4) = 4+1+4$$

Tomaremos como convención que la sustitución tiene precedencia sobre cualquier otra operación, por ejemplo

$$x+z(x,z := z+1,4) = x+(z(x,z := z+1,4)) = x+4$$

Nótese que sustituir una variable que no aparece en la expresión es posible y no tiene ningún efecto.

Cuando se den las reglas de inferencia, frecuentemente se usará el símbolo x para denotar una secuencia no vacía de variables distintas. Si entonces F denota una secuencia de expresiones de la misma longitud que x, la expresión $E(x := F)$ denotará la expresión E donde se han sustituido simultáneamente las ocurrencias de cada elemento de x por su correpondiente expresión en F.

Sustitución y evaluación. Es importante no confundir una sustitución, la cual consiste en el reemplazo de variables por expresiones (obteniendo así nuevas expresiones) con una evaluación, la cual consiste en dar un valor a una expresión a partir de un estado dado (es decir, de asumir un valor dado para cada una de las variables de la expresión).

2.3. Igualdad y regla de Leibniz

Definiremos ahora el operador de igualdad, el cual nos permitirá construir expresiones booleanas a partir de dos expresiones de cualquier otro tipo. Las expresiones booleanas son aquellas cuya evaluación en un estado dado devolverá sólo los valores de verdad *True* (verdadero) o *False* (falso).

La expresión $E = F$ evaluada en un estado será igual al valor *True* si la evaluación de ambas expresiones E y F en ese estado producen el mismo valor. La expresión $E = F$ se considerará verdadera si su evaluación en cualquier estado posible produce el valor *True*. En caso contrario la expresión será igual a *False*. Una forma alternativa de demostrar que dos expresiones son iguales sin tener que recurrir a su evaluación en cualquier estado posible es aplicar leyes conocidas para ese tipo de expresiones y las leyes de la igualdad. Esta manera es más útil cuando se quiere razonar con expresiones. Si las leyes caracterizan la igualdad (es decir que dos expresiones son iguales si y sólo si pueden demostrarse iguales usando las leyes), puede pensarse a éstas como la definición de la igualdad.

La igualdad es una relación de equivalencia, o sea que satisface las leyes de *reflexividad, simetría* y *transitividad.* La reflexividad nos da un axioma del cual partir (o al cual llegar) en una demostración de igualdad, la simetría permite razonar "hacia adelante" o "hacia atrás" indistintamente, mientras que la transitividad permite descomponer una demostración en una secuencia de igualdades más simples.

Otra propiedad característica de la igualdad es la de reemplazo de iguales por iguales. Una posible formulación de dicha regla es la siguiente:

$$\textbf{Leibniz: } \frac{X = Y}{E(x := X) = E(x := Y)}$$

En la siguiente tabla se resumen las leyes que satisface la igualdad.

reflexividad:	$X = X$
simetría:	$(X = Y) = (Y = X)$
transitividad:	$\frac{X=Y, Y=Z}{X=Z}$
Leibniz:	$\frac{X=Y}{E(x:=X)=E(x:=Y)}$

Fue Leibniz quien introdujo la propiedad de que es posible sustituir en una expresión (lógica en su definición) elementos por otros que satisfacen el predicado de igualdad con ellos sin que ésto altere el significado de la expresión. En realidad, Leibniz fue más allá y asumió también la conversa, es decir que si dos elementos pueden sustituirse en cualquier expresión sin cambiar

el significado, entonces estos elementos son iguales. Esta última propiedad aparecerá recurrentemente en varios conceptos (por ejemplo en el de extensionalidad), pero no la asumiremos a priori.

Ejemplo 2.1 (Máximo entre dos números [Coh90])
Consideremos el operador binario $\max$, el cual aplicado a dos números devuelve el mayor. El operador $\max$ tendrá más precedencia que el $+$, o sea que $P + Q \max R = P + (Q \max R)$. Satisfará además los siguientes axiomas:

Axioma 2.1 (Conmutatividad)

$$P \max Q = Q \max P$$

Axioma 2.2 (Asociatividad)

$$P \max (Q \max R) = (P \max Q) \max R$$

Axioma 2.3 (Idempotencia)

$$P \max P = P$$

Axioma 2.4 (Distributividad de + con respecto a $\max$**)**

$$P + (Q \max R) = (P + Q) \max (P + R)$$

Axioma 2.5 (Conexión entre $\max$ **y** $\geq$**)**

$$P \max Q \geq P$$

La forma axiomática de trabajar es asumir que los axiomas son ciertos y demostrar las propiedades requeridas a partir de ellos y de las *reglas de inferencia*, las cuales nos permiten inferir proposiciones válidas a partir de otras. Para el caso del máximo, asumiremos las reglas de inferencia de la igualdad, las de orden del $\geq$, y la siguiente regla de sustitución, la cual dice que si tenemos una proposición que sabemos verdadera (e.g. un axioma o un teorema previamente demostrado), podemos sustituir sus variables por cualquier expresión y seguiremos teniendo una expresión verdadera.

$$\textbf{Sustitución: } \frac{P}{P(x := X)}$$

Dado que el operador $\max$ es asociativo y conmutativo, evitaremos el uso de paréntesis cuando sea posible, por ejemplo en la propiedad a demostrar.

Usando estas reglas se demostrará la siguiente propiedad:

Teorema 2.6

$$W \max X + Y \max Z = (W + Y) \max (W + Z) \max (X + Y) \max (X + Z)$$

Demostración:

La demostración se realiza paso a paso aplicando Leibniz combinado a veces con la regla de sustitución. Un paso de demostración de una igualdad tendrá en general la forma

$$\begin{aligned} & E(x := X) \\ = & \ \{ \ X = Y \ \} \\ & E(x := Y) \end{aligned}$$

Trataremos de transformar la expresión más compleja en la más simple.

Puede justificarse el formato de demostración mostrando que en realidad es simplemente una forma estilizada de aplicar las reglas de inferencia definidas antes. Por ejemplo, la primera igualdad del teorema a demostrar puede justificarse como sigue:

$$\dfrac{\dfrac{P+(Q \max\ R)=(P+Q) \max\ (P+R)}{(W+Y \max\ Z)=(W+Y) \max\ (W+Z)}}{\dfrac{(W+Y \max\ Z) \max\ (X+Y) \max\ (X+Z)=(W+Y) \max\ (W+Z) \max\ (X+Y) \max\ (X+Z)}{(W+Y) \max\ (W+Z) \max\ (X+Y) \max\ (X+Z)=(W+Y \max\ Z) \max\ (X+Y) \max\ (X+Z)}}$$

Donde el primer paso es una aplicación de la regla de sustitución, el segundo de Leibniz y el último de simetría de la igualdad.

$$\begin{aligned} & (W+Y) \max\ (W+Z) \max\ (X+Y) \max\ (X+Z) \\ = & \ \{ \text{ Distributividad del + respecto del max , con } P := W, Q := Y, R := Z \ \} \\ & (W+Y \max\ Z) \max\ (X+Y) \max\ (X+Z) \\ = & \ \{ \text{ Distributividad del + respecto del max , con } P := X, Q := Y, R := Z \ \} \\ & (W+Y \max\ Z) \max\ (X+Y \max\ Z) \\ = & \ \{ \text{ Conmutatividad del max , con } P := W, Q := Y \max\ Z \ \} \\ & (Y \max\ Z+W) \max\ (X+Y \max\ Z) \\ = & \ \{ \text{ Conmutatividad del max , con } P := X, Q := Y \max\ Z \ \} \\ & (Y \max\ Z+W) \max\ (Y \max\ Z+X) \\ = & \ \{ \text{ Distributividad del + respecto del max , con } P := Y \max\ Z, Q := W, R := X \ \} \\ & Y \max\ Z+W \max\ X \\ = & \ \{ \text{ Conmutatividad del + } \} \\ & W \max\ X+Y \max\ Z \end{aligned}$$

El formato de la demostración anterior es el siguiente:

$$\begin{aligned} & E_0 \\ = & \ \{ \text{ Ley aplicada o justificación de } E_0 = E_1 \ \} \\ & E_1 \\ & \vdots \\ & E_{n-1} \end{aligned}$$

$= \{$ Ley aplicada o justificación de $E_{n-1} = E_n$ $\}$
E_n

luego, usando transitividad, se concluye que $E_0 = E_n$.

Debido a que (casi) todos los pasos de la demostración son explícitos, ésta es quizás más larga de lo esperado. Dado que en informática es necesario usar la lógica para calcular programas, es importante que las demostraciones no sean demasiado voluminosas y menos aún tediosas. Afortunadamente, ésto puede conseguirse sin sacrificar formalidad, adoptando algunas convenciones y demostrando algunos *metateoremas*, es decir, teoremas acerca del sistema formal de la lógica. Este último punto será tratado en el capítulo 3.

Hemos ya adoptado la convención de no escribir los paréntesis cuando un operador es asociativo con el consiguiente ahorro de pasos de demostración. De la misma manera, cuando un operador sea conmutativo, intercambiaremos libremente los términos sin hacer referencia explícita a la regla. Además, cuando no se preste a confusión juntaremos pasos similares en uno y no escribiremos la sustitución aplicada cuando ésta pueda deducirse del contexto. Así, por ejemplo, la demostración anterior quedaría como sigue:

$(W + Y) \max (W + Z) \max (X + Y) \max (X + Z)$
$= \{$ Distributividad del + respecto del max $\}$
$(W + Y \max Z) \max (X + Y \max Z)$
$= \{$ Distributividad del + respecto del max $\}$
$Y \max Z + W \max X$

2.4. Funciones

Una función es una regla para computar un valor a partir de otro u otros. Por ejemplo, la función que dado un número lo eleva al cubo se escribe usualemte en matemática como

$$f(x) = x^3$$

Para economizar paréntesis (y por otras razones que quedarán claras más adelante) 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 $f.x$. Por otro lado, para evitar confundir el predicado de 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

$$f.x \doteq x^3$$

Si ahora quiere evaluarse esta función en un valor dado, por ejemplo en (el valor) 2 obtenemos

$f.2$
$= \{$ aplicación de f $\}$
2^3
$= \{$ aritmética $\}$
8

Puede notarse que el primer paso justificado como "aplicación de f" es simplemente una sustitución de la variable x por la constante 2 en la expersión que define a f. Ésto puede hacerse para cualquier expresión, no necesariamente para constantes. Por ejemplo

$$
\begin{aligned}
& f.(z+1) \\
= & \ \{ \text{ aplicación de } f \ \} \\
& (z+1)^3 \\
= & \ \{ \text{ aritmética } \} \\
& z^3 + 3 * z^2 + 3 * z + 1
\end{aligned}
$$

La aplicación de funciones a argumentos puede definirse entonces usando sustitución. Si se tiene una función f definida como

$$f.x \doteq E$$

para alguna expresión E, entonces la aplicación de f a una expresión cualquiera X estará dada por

$$f.X = E(x := X)$$

La idea de sustitución de iguales por iguales (regla de Leibniz) y la de función están íntimamente ligadas, por lo cual puede postularse la siguiente regla (derivada de la regla usual de Leibniz y la definición de aplicación de funciones), la cual llamaremos, para evitar la proliferación de nombres, *regla de Leibniz*:

$$\frac{X = Y}{f.X = f.Y}$$

Las funciones serán uno de los conceptos esenciales de este libro. Volveremos sobre ellas en el capítulo 8.

2.5. Ejercicios

Ejercicio 2.1
Realizar las siguientes sustituciones eliminando los paréntesis innecesarios.

1. $(x + 2)(x := 6)$
2. $(x + 2)(x := x + 6)$
3. $(x * x)(x := z + 1)$
4. $(x + z)(y := z)$
5. $(x * (z + 1))(x := z + 1)$

Ejercicio 2.2
Realizar las siguientes sustituciones simultáneas eliminando los paréntesis innecesarios.

1. $(x + y)(x, y := 6, 3 * z)$
2. $(x + 2)(x, y := y + 5, x + 6)$

3. $(x * (y - z))(x, y := z + 1, z)$
4. $(x + y)(y, x := 6, 3 * z)$
5. $(x * (z + 1))(x, y, z := z, y, x)$

Ejercicio 2.3
Realizar las siguientes sustituciones eliminando los paréntesis innecesarios.

1. $(x + 2)(x := 6)(y := x)$
2. $(x + 2)(x := y + 6)(y := x - 6)$
3. $(x + y)(x := y)(y := 3 * z)$
4. $(x + y)(y := 3 * z)(x := y)$
5. $(x * (z + 1))(x, y, z := z, y, x)(z := y)$
6. $(4 * x * x + 4 * y * x + y * y)(x, y := y, x)(y := 3)$

Ejercicio 2.4
Más arriba mencionamos que Leibniz había propuesto como parte de la definición de igualdad la conversa de la que nosotros llamamos regla de Leibniz, es decir que si para todo estado y expresión E vale que $E(x := X) = E(x := Y)$, entonces $X = Y$. Demostrar que la conversa vale para las expresiones aritméticas.

Ejercicio 2.5
Demostrar las siguientes desigualdades.

1. $X\, max\, -X + Y\, max\, -Y \geq (X + Y)\, max\, -(X + Y)$
2. $(X + Z)\, max\, (Y + Z) + (X - Z)\, max\, (Y - Z) \geq X + Y$

Ejercicio 2.6
Definir el valor absoluto en términos del máximo y demostrar la desigualdad triangular:
$|X| + |Y| \geq |X + Y|$

Capítulo 3

Cálculo Proposicional

> One philosopher was shocked when Bertrand Russell told him that a false proposition implies any proposition. He said 'you mean that from the statement that two plus two equals five it follows that you are the Pope?' Russell replied 'yes.' The philosopher asked, 'Can you prove this?' Russell replied 'Certainly,' and contrieved the following proof on the spot:
>
> 1. Suppose $2 + 2 = 5$.
> 2. Substracting two from both sides of the equation, we get $2 = 3$.
> 3. Transposing, we get $3 = 2$.
> 4. Substracting one from both sides, we get $2 = 1$
>
> Now, the Pope and I are two. Since two equals one, then the Pope and I are one. Hence I am the Pope.
>
> Raymond Smullyan: *What Is the Name of This Book?*

En este capítulo se presenta una alternativa a las evaluaciones (tablas de verdad) para razonar con expresiones booleanas. En los capítulos precedentes se determinaba si una fórmula era una tautología construyendo su tabla de verdad; ahora en cambio se usarán métodos algebraicos para demostrar que una expresión dada es un *teorema*. Esta manera de trabajar presenta una buena alternativa a las tablas de verdad, dado que éstas pueden ser extremadamente largas si el número de variables es grande, en cambio una demostración puede ser mucho más corta. La tabla de verdad se construye de manera mecánica, a diferencia de la demostración para la cual es necesario desarrollar cierta habilidad en la manipulación sintáctica de fórmulas (y aún así no puede garantizarse que se encontrará una demostración). Por otro lado, la manipulación sintáctica de fórmulas suele tener más aplicaciones y es una de las habilidades requeridas más importantes a la hora de construir programas de computadora.

Con este fin introduciremos un sistema para construir demostraciones que involucren expresiones de la lógica proposicional. El estilo utilizado se conoce como *cálculo proposicional*. Este cálculo está orientado a la programación, por lo cual provee herramientas para el manejo efectivo de fórmulas lógicas de tamaño considerable.

La necesidad de este cálculo se fue haciendo cada vez más notoria a medida que se desarrollaba el cálculo de programas. Pueden encontrarse ya sus primeros elementos en el trabajo de

E.W. Dijkstra y W.H.J. Feijen [DF88]. Una exposición completa figura en [DS90] y en [GS93]. Puede consultarse también [Coh90], donde la exposición es más elemental.

3.1. Sistemas formales

El cálculo proposicional se presentará como un sistema formal. El objetivo de un sistema formal es explicitar un lenguaje en el cual se realizarán demostraciones y las reglas para realizarlas. Esto permite tener una noción muy precisa de lo que es una demostración, así también como la posibilidad de hablar con precisión de la sintaxis y la semántica. En este libro no estudiaremos estos temas en profundidad.

Un *sistema formal* consta de cuatro elelmentos:

- Un conjunto de símbolos llamado *alfabeto*, a partir del cual las expresiones son construidas.
- Un conjunto de *expresiones bien formadas*, es decir aquellas palabras construidas usando los símbolos del alfabeto que serán consideradas correctas. Una expresión bien formada no necesariamente es verdadera.
- Un conjunto de *axiomas*, los cuales son las fórmulas básicas a partir de las cuales todos los teoremas se derivan.
- Un conjunto de *reglas de inferencia*, las cuales indican cómo derivar fórmulas a partir de otras ya derivadas.

Como todo sistema formal, nuestro cálculo consistirá de un conjunto de expresiones construidas a partir de elementos de un alfabeto dado siguiendo reglas explícitas de buena formación. Si bien en este capítulo se vuelve a definir el lenguaje de las expresiones booleanas, éste coincidirá con el definido en los capítulos precedentes, sólo que aquí es presentado de manera más formal. Por otro lado, el cálculo estará constituido por un conjunto de axiomas, los cuales irán introduciéndose paulatinamente, y un conjunto de reglas de inferencia. A diferencia de otros sistemas formales para la lógica proposicional, nuestro cálculo da un lugar fundamental al conectivo de equivalencia lógica, por lo cual las reglas de inferencia usadas serán esencialmente las de Leibniz, transitividad y sustitución.

El sistema formal bajo consideración consta de los siguientes elementos:

alfabeto: Las *expresiones booleanas* – las cuales constituirán la sintaxis de nuestro cálculo – se construirán usando el siguiente alfabeto:

constantes: Las constantes *True* y *False* que se usarán para denotar los valores verdadero y falso respectivamente.

variables: Un conjunto infinito. Se usarán típicamente las letras p, q, r como variables proposicionales.

operadores unarios: 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 puedan construir de acuerdo a las siguientes prescripciones:

1. Las variables proposicionales y las constantes son fórmulas.
2. Si E es una fórmula, entonces $(\neg E)$ también lo es.
3. Si E y F son fórmulas y $\oplus$ es un operador binario ($\equiv$, $\vee$, etc.), entonces $(E \oplus F)$ también es una fórmula.
4. Sólo son fórmulas las construidas con las tres reglas precedentes.

reglas de inferencia: Las reglas de inferencia usadas serán las ya presentadas para la igualdad. En este caso, la equivalencia lógica entre expresiones booleanas satisfará las propiedades de la igualdad:

$$\text{Transitividad: } \frac{P \equiv Q, Q \equiv R}{P \equiv R}$$

$$\text{Leibniz: } \frac{P \equiv Q}{E(r := P) = E(r := Q)}$$

$$\text{Sustitución: } \frac{P}{P(q := R)}$$

axiomas: Los axiomas del cálculo se introducirán gradualmente. El cálculo proposicional tal como se presenta aquí no aspira a la minimalidad en el número de reglas de inferencia y axiomas, dado que uno de los principales objetivos de dicho cálculo es su uso para realizar demostraciones reales y no ser meramente un objeto de estudio en sí mismo, como generalmente ocurre. Si se usan un mínimo de axiomas y de reglas de inferencia, las demostraciones de teoremas elementales resultan en general demasiado largas.

En el capítulo siguiente se verán algunas aplicaciones básicas del cálculo proposicional. Su principal aplicación es la derivación y verificación de programas, lo cual se desarrollará en los capítulos subsiguientes.

Un formato cómodo para demostraciones (usando las reglas de inferencia ya presentadas) es el que usamos en el capítulo 2 para demostrar igualdades. Primero definimos un *teorema* como generado por las siguientes reglas:

1. un axioma es un teorema,

2. la conclusión de una regla de inferencia cuyas premisas son teoremas (ya demostrados) es un teorema,
3. una expresión booleana que se demuestra equivalente (ver más adelante) a un teorema es también un teorema.

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 se seguirá por aplicación de la regla de transitividad.

Una generalización usada usualmente en lógica e indispensable para cualquier cálculo que pretenda ser útil, es que los pasos de demostración puedan ser, además de las reglas elegidas para el cálculo, teoremas previamente demostrados. Más adelante se introducirán otras generalizaciones. La justificación de esta generalización (y de otras luego) es que de manera elemental se puede reconstruir una demostración "pura" a partir de la generalizada. En este caso, la demostración del nuevo teorema se obtiene simplemente "pegando" la demostración hasta el teorema viejo con la demostración (ya realizada) de éste último.

Veremos a continuación una serie de leyes o axiomas y de teoremas que se demuestran a partir de éstos. A veces se omiten las demostraciones, sobreentendiéndose que éstas quedan como ejercicios para el lector. Se recomienda hacer estos ejercicios para desarrollar la habilidad de manipular formalmente el cálculo de predicados, habilidad que será indispensable para realizar exitosamente derivaciones de programas.

3.2. La equivalencia

La equivalencia, que denotaremos con el símbolo $\equiv$, se define como el operador binario que satisface las siguientes condiciones:

Axioma 3.1 (Asociatividad)
$((p \equiv q) \equiv r) \equiv (p \equiv (q \equiv r))$

Axioma 3.2 (Conmutatividad)
$p \equiv q \equiv q \equiv p$

Axioma 3.3 (Neutro)
$p \equiv \mathit{True} \equiv p$

La asociatividad permite la omisión de paréntesis. Por ejemplo: la tercera condición debería haberse escrito como $(p \equiv \mathit{True}) \equiv p$. La asociatividad permite interpretarla en cada caso particular de la forma más conveniente. Como además es conmutativa, consideraremos también irrelevante el orden de los términos en una equivalencia. Por ejemplo, podemos usar el último axioma para reemplazar $p \equiv p$ por True.

La reflexividad de la equivalencia puede demostrarse como sigue:

Teorema 3.4 (Reflexividad)
$p \equiv p$

Demostración:

$p \equiv p$
$\equiv$ { 3.3 aplicado a la primera p }
$(p \equiv True) \equiv p$

Ponemos los paréntesis en la última expresión para remarcar cuál fue la sustitución realizada. Al ser la última expresión un axioma, el teorema queda demostrado.

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

Teorema 3.5
True

Cuando se quiere demostrar que dadas dos expresiones lógicas E y F vale que $E \equiv F$, en lugar de hacer una demostración que comience con la equivalencia y termine en *True* (o en algún otro teorema o axioma), podemos también comenzar con E y llegar a través de equivalencias a F. Una justificación de esta generalización es la siguiente. Dada una demostración de la forma

E_0
$\equiv$ { justificación de $E_0 \equiv E_1$ }
E_1
$\vdots$
E_{n-1}
$\equiv$ { justificación de $E_{n-1} \equiv E_n$ }
E_n

la misma puede transformarse mecanicamente en una demostración pura de la forma

True
$\equiv$ { reflexividad de la equivalencia }
$E_0 \equiv E_0$
$\equiv$ { justificación de $E_0 \equiv E_1$ }
$E_0 \equiv E_1$
$\vdots$
$E_0 \equiv E_{n-1}$
$\equiv$ { justificación de $E_{n-1} \equiv E_n$ }
$E_0 \equiv E_n$

Estos resultados acerca del cálculo se llamarán *metateoremas*, para distinguirlos de los teoremas que son fórmulas del cálculo. El objetivo de los metateoremas es proveer herramientas para poder usar el cálculo de manera efectiva para la construcción de demostraciones en la "vida real".

La (meta)demostración del siguiente metateorema se deja como ejercicio (nótese que hay que decir cómo se construye una demostración a partir de otra u otras).

Metateorema 3.6
Dos teoremas cualesquiera son equivalentes.

3.3. La negación

La negación, que denotaremos con el símbolo $\neg$, se define como el operador unario de más precedencia que cualquier otro operador del cálculo.

Axioma 3.7 (Negación y equivalencia)
$\neg(p \equiv q) \equiv \neg p \equiv q$

Además, definimos la constante *False* con el siguiente axioma

Axioma 3.8
$False \equiv \neg True$

Teorema 3.9 (Doble negación)
$\neg\neg p \equiv p$

Teorema 3.10
$p \equiv False \equiv \neg p$

3.4. La discrepancia

La discrepancia, que denotaremos con el símbolo $\not\equiv$, se define como un operador binario con precedencia igual a la equivalencia, lo cual implica que habría que poner paréntesis si aparecen ambos conectivos al mismo nivel en una expresión. Sin embargo, el teorema 3.14 nos permite sacarlos.

Axioma 3.11 (Definición de $\not\equiv$)
$p \not\equiv q \equiv \neg(p \equiv q)$

Teorema 3.12 (Asociatividad)
$((p \not\equiv q) \not\equiv r) \equiv (p \not\equiv (q \not\equiv r))$

Teorema 3.13 (Conmutatividad)
$(p \not\equiv q) \equiv (q \not\equiv p)$

Teorema 3.14 (Asociatividad mutua)
$p \equiv (q \not\equiv r) \equiv (p \equiv q) \not\equiv r$

Teorema 3.15
$p \not\equiv False \equiv p$

Demostración: Para demostrar propiedades de un operador nuevo definido en términos de otros (como en este caso la discrepancia en términos de la negación y la equivalencia) un método frecuentemente exitoso es reemplazar al operador por su definición y manipular los operadores "viejos", volviendo a obtener el operador nuevo si es necesario (no es el caso en esta demostración).

$$\begin{array}{ll} & p \not\equiv False \\ \equiv & \{ \text{ 3.11 con } q := false) \} \\ & \neg(p \equiv False) \end{array}$$

$\equiv$ { 3.8 }
$\neg(p \equiv \neg True)$
$\equiv$ { 3.7 }
$\neg\neg(p \equiv True)$
$\equiv$ { doble negación, *True* neutro para $\equiv$ }
p

Teorema 3.16 (Intercambiabilidad)
$p \equiv q \not\equiv r \equiv p \not\equiv q \equiv r$

3.5. La disyunción

La disyunción, que denotaremos con el símbolo $\vee$, se define como un operador binario de mayor precedencia que la equivalencia, es decir que $p \vee q \equiv r$ debe entenderse como $(p \vee q) \equiv r$.

Axioma 3.17 (Asociatividad)
$(p \vee q) \vee r \equiv p \vee (q \vee r)$

Axioma 3.18 (Conmutatividad)
$p \vee q \equiv q \vee p$

Axioma 3.19 (Idempotencia)
$p \vee p \equiv p$

Axioma 3.20 (Distributividad con la equivalencia)
$p \vee (q \equiv r) \equiv (p \vee q) \equiv (p \vee r)$

Axioma 3.21 (Tercero excluido)
$p \vee \neg p$

Demostraremos algunas propiedades de la disyunción.

Teorema 3.22
$p \vee (q \vee r) \equiv (p \vee q) \vee (p \vee r)$

En general, conviene comenzar a demostrar una equivalencia a partir de la expresión más compleja, dado que las posibilidades de simplificar una expresión suelen estar más acotadas que las de "complejizarla", lo cual hace que la demostración sea más fácil de encontrar.

Demostración:

$(p \vee q) \vee (p \vee r)$
$\equiv$ { asociatividad de $\vee$ }
$p \vee q \vee p \vee r$
$\equiv$ { conmutatividad de $\vee$ }
$p \vee p \vee q \vee r$

$\equiv$ { idempotencia de $\vee$ }

$p \vee q \vee r$

$\equiv$ { asociatividad de $\vee$ }

$p \vee (q \vee r)$

Teorema 3.23 (Elemento absorbente)
$p \vee \mathit{True} \equiv \mathit{True}$

Demostración:

$p \vee \mathit{True}$

$\equiv$ { $p \equiv \mathit{True} \equiv p$, interpretada como $(p \equiv p) \equiv \mathit{True}$ }

$p \vee (q \equiv q)$

$\equiv$ { distributividad de $\vee$ con respecto a $\equiv$ }

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

$\equiv$ { $p \equiv p$ }

True

Teorema 3.24 (Elemento neutro)
$p \vee \mathit{False} \equiv p$

3.6. La conjunción

La conjunción, que denotaremos con el símbolo $\wedge$, se define como un operador binario con la misma precedencia que la disyunción. Esto hace necesario el uso de paréntesis en las expresiones que involucran a ambos operadores. Por ejemplo: no es lo mismo $(p \vee q) \wedge r$ que $p \vee (q \wedge r)$. Esto nos dice que no podemos escribir $p \vee q \wedge r$, pues esta expresión no está asociada de manera natural a ninguna de las dos anteriores. Se introducirá un único axioma para la conjunción.

Axioma 3.25 (Regla dorada)
$p \wedge q \equiv p \equiv q \equiv p \vee q$

La regla dorada aprovecha fuertemente la asociatividad de la equivalencia. En principio, la interpretaríamos 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.

Teorema 3.26 (Asociatividad)
$p \wedge (q \wedge r) \equiv (p \wedge q) \wedge r$

Demostración:

$p \wedge (q \wedge r)$
$\equiv$ { regla dorada, interpretada como $p \wedge q \equiv (p \equiv q \equiv p \vee q)$ }
$p \equiv (q \wedge r) \equiv p \vee (q \wedge r)$
$\equiv$ { regla dorada, dos veces[1] }
$p \equiv q \equiv r \equiv q \vee r \equiv p \vee (q \equiv r \equiv q \vee r)$
$\equiv$ { 3.20 }
$p \equiv q \equiv r \equiv q \vee r \equiv p \vee q \equiv p \vee r \equiv p \vee q \vee r$
$\equiv$ { 3.2, 3.1 }
$(p \equiv q \equiv p \vee q) \equiv r \equiv (p \vee r \equiv q \vee r \equiv p \vee q \vee r)$
$\equiv$ { 3.20 }
$(p \equiv q \equiv p \vee q) \equiv r \equiv (p \equiv q \equiv p \vee q) \vee r$
$\equiv$ { regla dorada }
$(p \wedge q) \equiv r \equiv (p \wedge q) \vee r$
$\equiv$ { regla dorada }
$(p \wedge q) \wedge r$

Teorema 3.27 (Conmutatividad)
$p \wedge q \equiv q \wedge p$

Teorema 3.28 (Idempotencia)
$p \wedge p \equiv p$

Teorema 3.29 (Neutro)
$p \wedge \mathit{True} \equiv p$

Demostración:

$p \wedge \mathit{True}$
$\equiv$ { regla dorada }
$p \equiv \mathit{True} \equiv p \vee \mathit{True}$
$\equiv$ { 3.23 }
$p \equiv \mathit{True} \equiv \mathit{True}$
$\equiv$ { $p \equiv p$; *True* es neutro de $\equiv$ }
p

La disyunción es distributiva con respecto a la equivalencia por definición. La conjunción *no* lo es. Veamos:

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

$\equiv$ { regla dorada en cada miembro }

$p \equiv q \equiv p \vee q \equiv p \equiv r \equiv p \vee r$

$\equiv$ { conmutatividad de $\equiv$ }

$p \equiv q \equiv r \equiv p \vee q \equiv p \vee r \equiv p$

$\equiv$ { distributividad de $\vee$ con respecto a $\equiv$ }

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

$\equiv$ { asociatividad de $\equiv$ }

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

$\equiv$ { regla dorada }

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

y esto *no* es equivalente a $p \wedge (q \equiv r)$; ejemplo: $p := \mathit{False}$.

Teorema 3.30
$p \vee q \equiv p \vee \neg q \equiv p$

Teorema 3.31 (de Morgan)

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

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

Demostración de (i):

$\neg p \wedge \neg q$

$\equiv$ { regla dorada }

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

$\equiv$ { 3.30 }

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

$\equiv$ { negación de $\equiv$ }

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

$\equiv$ { 3.30 }

$\neg(p \vee q)$

Teorema 3.32 (Distributividad de $\wedge$ con respecto a $\not\equiv$)
$p \wedge (q \not\equiv r) \equiv p \wedge q \not\equiv p \wedge r$

3.7. La implicación

La implicación, que denotaremos con el símbolo $\Rightarrow$, se define como el operador binario que precede a la equivalencia, pero es precedido por la disyunción (y por lo tanto también por la conjunción).

Para definirlo alcanza con el siguiente axioma:

Axioma 3.33
$p \Rightarrow q \equiv p \vee q \equiv q$

Teorema 3.34
$p \Rightarrow p$

Demostración:

$$\begin{aligned} & p \Rightarrow p \\ \equiv & \ \{\ 3.33\ \} \\ & p \vee p \equiv p \\ \equiv & \ \{\ 3.19;\ p \equiv p\ \} \\ & \mathit{True} \end{aligned}$$

Teorema 3.35
$p \Rightarrow q \equiv \neg p \vee q$

Demostración:

$$\begin{aligned} & p \Rightarrow q \\ \equiv & \ \{\ \text{definición de} \Rightarrow\ \} \\ & p \vee q \equiv q \\ \equiv & \ \{\ p \vee \mathit{False} \equiv p\ \} \\ & p \vee q \equiv \mathit{False} \vee q \\ \equiv & \ \{\ 3.20\ \} \\ & (p \equiv \mathit{False}) \vee q \\ \equiv & \ \{\ p \equiv \mathit{False} \equiv \neg p\ \} \\ & \neg p \vee q \end{aligned}$$

Teorema 3.36
$\neg(p \Rightarrow q) \equiv p \wedge \neg q$

Teorema 3.37
$p \Rightarrow \mathit{True}$

Teorema 3.38
$p \wedge q \Rightarrow p$

Teorema 3.39
$p \Rightarrow p \vee q$

Teorema 3.40 (Transitividad)
$(p \Rightarrow q) \wedge (q \Rightarrow r) \Rightarrow (p \Rightarrow r)$

Un axioma que será útil es la siguiente versión axiomática de la regla de Leibniz:

Axioma 3.41 (Leibniz)
$e = f \Rightarrow E(z := e) = E(z := f)$

La forma de este axioma es diferente a la de los anteriores, ya que no es estrictamente hablando una expresión booleana sino un esquema, dado que para cada E diferente será un axioma diferente. Alternativamente, podría demostrarse el axioma de Leibniz como un metateorema de nuestro cálculo de predicados. La demostración puede hacerse por inducción en la estructura de E.

Si bien este axioma está inspirado en la regla de Leibniz su significado es diferente, dado que la regla de Leibniz dice que si dos expresiones son iguales *en cualquier estado* entonces sustituir esas expresiones en una expresión dada también producirá expresiones iguales en cualquier estado. El axioma de Leibniz dice, en cambio, que si dos expresiones son iguales *en un estado dado*, entonces sustituirlas en una expresión dada producirá expresiones iguales *en ese estado*.

Una diferencia importante con la regla de Leibniz es que la recíproca del axioma no es cierta (mientras que para la regla vimos que para nuestro lenguaje de expresiones booleanas y aritméticas es válida). El siguiente contraejemplo muestra que para el axioma no es éste el caso.

Ejemplo 3.1
Tomemos como E a la expresión $\mathit{True} \vee z$. Luego se da que

$$E(z := \mathit{True}) = E(z := \mathit{False})$$

dado que ambos son verdaderos, pero obviamente $\mathit{True} \neq \mathit{False}$.

En el caso de la regla de inferencia de Leibniz, se puede pensar que la recíproca es válida dado que se supone una cuantificación sobre todas las expresiones posibles.

3.8. La consecuencia

La consecuencia es el operador dual de la implicación. La denotaremos con el símbolo $\Leftarrow$ y puede definirse como un operador binario que tiene la misma precedencia que la implicación y que satisface

Axioma 3.42
$p \Leftarrow q \equiv p \vee q \equiv p$

Puede ahora demostrarse fácilmente que

Teorema 3.43
$p \Leftarrow q \equiv q \Rightarrow p$

A partir de este teorema pueden "dualizarse" las propiedades de la implicación.

En las secciones siguientes permitiremos usar la implicación o la consecuencia como nexo en las demostraciones (pero no ambas en la misma demostración). Esto se justifica con la transitividad de estos operadores.

Cada vez que hemos introducido un operador, hemos mencionado su nivel de precedencia. La existencia de estas convenciones permite eliminar el uso de paréntesis, facilitando de esta manera la escritura y lectura de expresiones booleanas. Pero no debemos olvidar que cuando se involucran operadores con la misma precedencia los paréntesis sí son necesarios. La tabla que sigue resume los niveles de precedencia que hemos establecido, de mayor a menor:

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

3.9. Generalización del formato de demostración

Se desea tener en el cálculo de predicados una herramienta poderosa para resolver problemas concretos. Los principales problemas a resolver provienen del cálculo formal de programas. Tanto para estos problemas como para los tratados en el próximo capítulo, es cómodo disponer de un formato de demostración más laxo.

La principal generalización que realizaremos es permitir que en las demostraciones dos pasos se conecten no sólo con una equivalencia sino también con una implicación. Por otro lado, se usarán premisas para modularizar las demostraciones. Obviamente, para poder realizar esta generalización es necesario demostrar un metateorema que muestre como transformar una demostración generalizada en una pura. No se probará este metateorema dado que su demostración, si bien no es dificil, es bastante larga y engorrosa. Se ilustrará con un ejemplo como puede realizarse dicha transformación.

Ejemplo 3.2 (tomado de [GS93])
Se quiere demostrar la siguiente proposición: dado que $p \equiv q$ y $q \Rightarrow r$ obtener como conlusión $p \Rightarrow r$.

La demostración generalizada tendrá la siguiente forma:

$$
\begin{array}{ll}
 & p \\
\equiv & \{ \text{ razón por la cual } p \equiv q \ \} \\
 & q \\
\Rightarrow & \{ \text{ razón por la cual } q \Rightarrow r \ \} \\
 & r
\end{array}
$$

La demostración pura asociada será la siguiente (nótese que las razones de los pasos de demostración de las premisas son las mismas que en la generalizada):

$$
\begin{array}{ll}
 & True \\
\equiv & \{ \text{ transitividad } \} \\
 & (p \equiv q) \wedge (q \Rightarrow r) \Rightarrow (p \Rightarrow r) \\
\equiv & \{ \text{ razón por la cual } p \equiv q \ \} \\
 & True \wedge (q \Rightarrow r) \Rightarrow (p \Rightarrow r)
\end{array}
$$

$\equiv$ { razón por la cual $q \Rightarrow r$ }

$\quad$ $\mathit{True} \wedge \mathit{True} \Rightarrow (p \Rightarrow r)$

$\equiv$ { idempotencia, *True* es neutro a izquierda de $\Rightarrow$ }

$\quad$ $(p \Rightarrow r)$

3.10. Ejercicios

Ejercicio 3.1
(Largo) Construir las tablas de verdad para todos los axiomas introducidos en este capítulo.

Ejercicio 3.2
(Muy largo) Demostrar todos los teoremas enunciados en esté capítulo.

Ejercicio 3.3
Demostrar que *False* es neutro para la disyunción:

$$p \vee \mathit{False} \equiv p$$

Ejercicio 3.4
Demostrar que *False* es absorbente para la conjunción:

$$p \wedge \mathit{False} \equiv \mathit{False}$$

Ejercicio 3.5
Demostrar las leyes de absorción:

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

Ejercicio 3.6
Demostrar las leyes de absorción:

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

Ejercicio 3.7
Demostrar:

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

Ejercicio 3.8
Demostrar:

1. La conjunción distribuye con respecto a la conjunción: $p \wedge (q \wedge r) \equiv (p \wedge q) \wedge (p \wedge r)$.
2. La conjunción distribuye con respecto a la disyunción: $p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$.
3. La disyunción es distributiva con respecto a la conjunción: $p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$

4. En general, la conjunción no es distributiva con respecto a la equivalencia, pero cuando la cantidad de signos $\equiv$ es par, sí lo es: $s \wedge (p \equiv q \equiv r) \equiv s \wedge p \equiv s \wedge q \equiv s \wedge r$

Ejercicio 3.9
Demostrar:

1. $p \wedge q \Rightarrow p$.
2. $p \Rightarrow (q \Rightarrow r) \equiv p \wedge q \Rightarrow r$.
3. La implicación distribuye con respecto a la equivalencia. $p \Rightarrow (q \equiv r) \equiv p \Rightarrow q \equiv p \Rightarrow r$.
4. Doble implicación. $(p \Rightarrow q) \wedge (q \Rightarrow p) \equiv p \equiv q$
5. Contrarecíproca. $p \Rightarrow q \equiv \neg q \Rightarrow \neg p$
6. $p \Rightarrow q \equiv p \wedge q \equiv p$
7. $(p \wedge q \Rightarrow r) \equiv (p \Rightarrow \neg q \vee r)$
8. Modus ponens. $p \wedge (p \Rightarrow q) \Rightarrow q$
9. Transitividad. $(p \Rightarrow q) \wedge (q \Rightarrow r) \Rightarrow (p \Rightarrow r)$
10. Monotonía. $(p \Rightarrow q) \Rightarrow (p \wedge r \Rightarrow q \wedge r)$
11. Monotonía. $(p \Rightarrow q) \Rightarrow (p \vee r \Rightarrow q \vee r)$

Ejercicio 3.10
Demostrar:

- $p \wedge (p \Rightarrow q) \equiv p \wedge q$
- $p \wedge (q \Rightarrow p) \equiv p$
- $p \vee (p \Rightarrow q) \equiv \mathit{True}$
- $p \vee (q \Rightarrow p) \equiv q \Rightarrow p$
- $\mathit{True} \Rightarrow p \equiv p$
- $p \Rightarrow \mathit{False} \equiv \neg p$
- $\mathit{False} \Rightarrow p \equiv \mathit{True}$

Capítulo 4

Aplicaciones del cálculo proposicional

> Mi unicornio azul por fin te encontré...
>
> Leo Masliah. *La recuperación del unicornio*

La lógica es una herramienta fundamental para resolver problemas. En ciencias de la computación es la herramienta fundamental para construir especificaciones y programas. Si bien esta disciplina ha dado un gran impulso para el desarrollo de la lógica, sobre todo en su uso de la como herramienta concreta, la lógica se sigue usando para decidir acerca de la validez de los razonamientos y para proveer soluciones a problemas diversos, como por ejemplo la construcción de circuitos digitales combinatorios.

En este capítulo se presentarán algunas aplicaciones elementales del cálculo proposicional: análisis de argumentos lógicos y resolución de problemas de ingenio.

4.1. Análisis de argumentaciones

En el capítulo **??** se usaron las tablas de verdad para comprobar la validez de ciertas formas de razonamiento. Si bien ese método es efectivo, cuando crece el número de variables proposicionales su costo se vuelve prohibitivo, dado que el tamaño de las tablas de verdad crece exponencialemente. Para el caso de los silogismos el método es factible dado que por su forma el número de proposiciones involucradas es fijo. Sin embargo, cuando se quieren analizar razonamientos de la "vida real" es conveniente recurrir a métodos sintácticos, es decir a la manipulación de fórmulas sin interpretación semántica. En el primer ejemplo consieraremos un razonamiento con seis variables proposicionales, lo cual hubiera dado lugar a una tabla de verdad con 64 filas.

Ejemplo 4.1
Considérese por ejemplo la siguiente argumentación (adaptada del libro [Cop73]):

> Si Dios fuera incapaz de evitar el mal no sería omnipotente; si 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.

Para representarlo en el cálculo proposicional elegimos letras proposicionales para cada una de las proposiciones elementales en el razonamiento, luego encontramos las fórmulas que representan las proposiciones más complejas y mostramos que efectivamente la fórmula asociada a la conclusión se deduce formalmente de las fórmulas asociadas a las premisas.

Se ve que si uno se expresa con precisión, la argumentación precedente no es en si misma un "razonamiento" sino un teorema a ser demostrado.

Elegimos las siguientes letras proposicionales:

q: Dios quiere evitar el mal.

c: Dios es capaz de evitar el mal.

o: Dios es omnipotente.

m: Dios es malévolo.

e: El mal existe.

d: Dios existe.

Las premisas se expresan de la siguiente manera:

1. $\neg c \Rightarrow \neg o$
2. $\neg q \Rightarrow m$
3. $e \Rightarrow \neg q \vee \neg c$
4. e
5. $d \Rightarrow o \wedge \neg m$

Y la conclusión obviamente es:

$\neg d$

Una demostración de la corrección de este razonamiento es la siguiente:

$\neg d$

$\Leftarrow$ { premisa 5, contrapositivo }

$\neg(\neg m \wedge o)$

$\equiv$ { de Morgan }

$m \vee \neg o$

$\Leftarrow$ { premisa 1, 2, monotonía dos veces }

$\neg q \vee \neg c$

$\Leftarrow$ { premisa 3 }

e

$\equiv$ { premisa 4 }

True

Ejemplo 4.2
Considérese el siguiente argumento cuya forma se denomina *dilema* en la lógica clásica ([Cop73]). Si bien en este caso la tabla de verdad podría haber sido construída (tiene sólo 16 filas), la demostración sintáctica es más corta y elegante.

> Si el general era leal, habría obedecido las órdenes, y si era inteligente las habría comprendido. O el general desobedeció las órdenes o no las comprendió. Luego, el general era desleal o no era inteligente.

Elegimos las siguientes letras proposicionales:

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:

1. $l \Rightarrow o$
2. $i \Rightarrow c$
3. $\neg o \vee \neg c$

Y la conclusión es:

$\neg l \vee \neg i$

Una demostracion posible es:

$\neg l \vee \neg i$

$\Leftarrow$ { premisa 2, contrapositiva, monotonía }

$\neg l \vee \neg c$

$\Leftarrow$ { premisa 1, contrapositiva, monotonía }

$\neg o \vee \neg c$

$\equiv$ { premisa 3 }

True

4.2. Resolución de acertijos lógicos

En esta sección usaremos el cálculo de predicados para la resolución de acertijos lógicos. La mayor economía de este cálculo nos permite encontrar soluciones más elegantes a problemas como los planteados en **??**. Veremos primero soluciones sintácticas a dichos problemas los cuales fueron resueltos usando (implícitamente) tablas de verdad.

Ejemplo 4.3
Tenemos dos personas, A y B, habitantes de la isla. A hace la siguiente afirmación: 'Al menos uno de nosotros es un 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 desarrolladas en los capítulos anteriores podemos obtener

$$\begin{aligned}
& a \equiv (\neg a \vee \neg b) \\
\equiv \; & \{ \text{ de Morgan } \} \\
& a \equiv \neg(a \wedge b) \\
\equiv \; & \{ \text{ def. de } \neg \} \\
& \neg(a \equiv (a \wedge b)) \\
\equiv \; & \{ \text{ def. de } \Rightarrow \} \\
& \neg(a \Rightarrow b) \\
\equiv \; & \{ \text{ negación de una implicación } \} \\
& a \wedge \neg b
\end{aligned}$$

Se concluye entonces que A es un caballero y B un pícaro

Ejemplo 4.4
Otra vez nos cruzamos con dos personas, A y B, (no necesariamente los mismos del ejercicio anterior). 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

$$a \equiv \neg a \wedge b$$

Manipulando esta expresión obtenemos

$a \equiv \neg a \wedge b$

$\equiv$ { regla dorada }

$a \equiv \neg a \equiv b \equiv \neg a \vee b$

$\equiv$ { negación y equivalencia }

$False \equiv b \equiv \neg a \vee b$

$\equiv$ { *False*y equivalencia }

$\neg(b \equiv \neg a \vee b)$

$\equiv$ { def $\Rightarrow$ }

$\neg(\neg a \Rightarrow b)$

$\equiv$ { negación de la implicación }

$\neg a \wedge \neg b$

Se concluye entonces que ambos son pícaros.

4.3. Ejercicios

Ejercicio 4.1
Analizar los razonamientos del ejercicio **??** usando las herramientas introducidas en esta sección.

Ejercicio 4.2
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.

Ejercicio 4.3
De la isla de los caballeros y los pícaros.

1. Nos enocntramos con dos personas, A y B. A dice 'Al menos uno de nosotros es un pícaro ¿Que son A y B?

2. A dice 'Yo soy un pícaro o B es un caballero' ¿Qué son A y B?

3. A dice 'Yo soy un pícaro pero B no' ¿Que son A y B?

4. Dos perrsonas se dicen del mismo tipo si son ambas caballeros o ambas pícaros. Tenemos tres personas, A, B y C. A y B dicen lo siguiente

   A: B es un pícaro.

   B: A y C son del mismo tipo.

   ¿ Que es C?

5. A dice 'Si soy un caballero entonces B también lo es' ¿Que son A y B?

6. Le preguntan a A si es un caballero. A responde 'Si soy un caballero entonces me comeré el sombrero. Demostrar que A tiene que comerse el sombrero.

7. A realiza (por separado) las siguientes dos afirmaciones.

   a) Amo a María.

   b) Si amo a María, entonces amo a Yolanda.

   ¿Que es A?

Capítulo 5

Aplicaciones del cálculo proposicional 2: Piso y Techo

Mi unicornio azul por fin te encontré...

Leo Masliah. *La recuperación del unicornio*

El estilo de cálculo proposicional con el cual estamos trabajando es particularmente cómodo para trabajar con problemas matemáticos. En este capítulo trabajaremos con propiedades de algunas funciones matemáticas que aparecen en varios problemas de programación.

5.1. La función piso

En esta sección trabajaremos con una función cuyas propiedades no han sido demasiado estudiadas.

Definición 5.1
Dado un número real x se define la función *piso* aplicada a x como el entero que satisface para todo entero m la siguiente propiedad:

$$\lfloor\ \rfloor : \mathbb{R} \mapsto Int$$

$$n \leq \lfloor x \rfloor \equiv n \leq x$$

Habría que mostrar que esta propiedad es suficiente para definir una función, es decir que $\lfloor x \rfloor$ está definida de manera única para cada x. Esto quedará demostrado por la proiedad 5.1 demostrada más abajo.

Propiedad 5.1
Instanciando n en la definición de piso con el entero $\lfloor x \rfloor$, obtenemos la siguiente propiedad:

$$\lfloor x \rfloor \leq x$$

Propiedad 5.2
Instanciando ahora x en la definición de piso con n (que es también un real), obtenemos la siguiente propiedad:

$$n \leq \lfloor n \rfloor$$

Propiedad 5.3
De las dos propiedades anteriores se sigue que para todo entero n

$$\lfloor n \rfloor = n$$

Propiedad 5.4
Podemos usar también la propiedad contrapositiva de la equivalencia **??**, y usando el hecho que $\neg p \leq q \equiv q < p$ obtenemos la siguiente propiedad, que puede pensarse como una definición alternativa de la función piso, para todo n entero:

$$\lfloor x \rfloor < n \equiv x < n$$

Propiedad 5.5
Instanciando n en la propiedad anterior con el entero $\lfloor x \rfloor + 1$, obtenemos la siguiente propiedad:

$$x < \lfloor x \rfloor + 1$$

Propiedad 5.6
De las propiedades anteriores puede deducirse que para todo x real

$$\lfloor x \rfloor \leq x < \lfloor x \rfloor + 1$$

y por lo tanto puede verse que la función piso es efectivamente una función cuya definición alternativa podría ser:

$$\lfloor\ \rfloor : \mathbb{R} \mapsto Int$$

$$n = \lfloor x \rfloor \equiv n \leq x < n + 1$$

La existencia y unicidad de un tal n es una propiedad conocida de los enteros.

Lema 5.1
La función piso es monótona, esto es, para todo par de reales x, y

$$x \leq y \Rightarrow \lfloor x \rfloor \leq \lfloor y \rfloor$$

Para demostrar el lema partimos del consecuente y mostramos que es consecuencia del antecedente

$\lfloor x \rfloor \leq \lfloor y \rfloor$

$\equiv$ { definiición de piso }

$\lfloor x \rfloor \leq y$

$\Leftarrow$ { transitividad }

$\lfloor x \rfloor \leq x \wedge x \leq y$

$\equiv$ { 5.1 }

$x \leq y$

5.2. Igualdad indirecta

Una de las maneras usuales de demostrar una igualdad entre dos números m y n cuando se conocen sólo desigualdades entre ellos es demostrar que tanto $m \leq n$ como $n \leq m$. Sin embargo, en el caso de la función piso este método a veces no funciona, dado que la función aparece en su definición sólo del lado derecho de la desigualdad. Otro método que introduciremos aquí como un teorema es el método de la *igualdad indirecta*

Teorema 5.2
Dos números p y q son iguales si y sólo si para cualquier otro número n del mismo tipo que p y q vale que

$$n \leq p \equiv n \leq q$$

Observación 5.3

Si bien este teorema es también una equivalencia lógica, uno de los miembros de dicha equivalencia no puede expresarse con los medios estudiados hasta ahora como una fórmula lógica. Hace falta usar cuantificación para poder expresar el teorema como una equivalencia simple, lo cual se hará en el capítulo 6.

Demostración: Si $p = q$ entonces la equivalencia vale por regla de Leibniz.

Supongamos ahora que la equivalencia es válida para todo n. En particular es válida cuando $n = p$. De aquí se deduce que $p \leq p \equiv p \leq q$, o, usando la reflexividad del $\leq$, que $p \leq q$. Simétricamente, se demuestra que $q \leq p$, instanciando a n en este caso con q.

Observación 5.4

La condición del teorema de que n es del mismo tipo que p y q es indispensable para poder realizar las instanciaciones en la demostración. Si este requisito no fuera necesario podría deducirse a partir de la definición de piso que $\lfloor x \rfloor = x$ por igualdad indirecta, lo cual claramente no es cierto.

Ejemplo 5.1
Como aplicación de la regla de igualdad indirecta resolveremos una propiedad secilla de la función piso. Demostraremos que para todo entero n vale que

$$\lfloor x + n \rfloor = \lfloor x \rfloor + n$$

Tomamos un entero k arbitrario

$k \leq \lfloor x + n \rfloor$
$\equiv$ { definición de piso }
$k \leq x + n$
$\equiv$ { aritmética }
$k - n \leq x$
$\equiv$ { definición de piso }
$k - n \leq \lfloor x \rfloor$
$\equiv$ { aritmética }
$k \leq \lfloor x \rfloor + n$

5.3. La función techo

De manera dual puede definirse la función techo, la cual dado un número real devuelve el entero más chico mayor o igual que él.

Definición 5.2
Dado un número real x se define la función *techo* aplicada a x como el entero que satisface para todo entero m la siguiente propiedad:

$$\lceil\ \rceil : \mathbb{R} \mapsto Int$$

$$\lceil x \rceil \leq n \equiv x \leq n$$

Ejemplo 5.2 (Máximo entre dos números [Coh90])
Consideremos el operador binario max , el cual aplicado a dos números devuelve el mayor. El operador max tendrá más precedencia que el +, o sea que $P + Q \max R = P + (Q \max R)$. Satisfará además los siguientes axiomas:

Axioma 5.5
Conmutatividad. $P \max Q = Q \max P$

Axioma 5.6
Asociatividad. $P \max (Q \max R) = (P \max Q) \max R$

Axioma 5.7
Idempotencia. $P \max P = P$

Axioma 5.8
Distributividad de + con respecto a max . $P + (Q \max R) = (P + Q) \max (P + R)$

Axioma 5.9
Conexión entre max y $\geq$. $P \max Q \geq P$

La forma axiomática de trabajar es asumir que los axiomas son ciertos y demostrar las propiedades requeridas a partir de ellos y de las *reglas de inferencia*, las cuales nos permiten inferir proposiciones válidas a partir de otras. Para el caso del máximo, asumiremos las reglas de inferencia de la igualdad, las de orden del $\geq$, y la siguiente regla de sustitución, la cual dice que si tenemos una proposición que sabemos verdadera (e.g. un axioma o un teorema previamente demostrado), podemos sustituir sus variables por cualquier expresión y seguiremos teniendo una expresión verdadera.

$$\textbf{Sustitución: } \frac{P}{P(x := X)}$$

Dado que el operador max es asociativo y conmutativo, evitaremos el uso de paréntesis cuando sea posible, por ejemplo en la propiedad a demostrar.

Usando estas reglas se demostrará la siguiente propiedad:

Teorema 5.10

$$W \max X + Y \max Z = (W+Y) \max (W+Z) \max (X+Y) \max (X+Z)$$

La demostración se realiza paso a paso aplicando Leibniz combinado a veces con la regla de sustitución. Un paso de demostración de una igualdad tendrá en general la forma

$$\begin{array}{ll} & E(x := X) \\ = & \{ X = Y \} \\ & E(x := Y) \end{array}$$

Trataremos de transformar la expresión más compleja en la más simple.

Puede justificarse el formato de demostración mostrando que en realidad es simplemente una forma estilizada de aplicar las reglas de inferencia definidas antes. Por ejemplo, la primera igualdad del teorema a demostrar puede justificarse como sigue:

$$\cfrac{\cfrac{\cfrac{P+(Q \max R)=(P+Q) \max (P+R)}{(W+Y \max Z)=(W+Y) \max (W+Z)}}{(W+Y \max Z) \max (X+Y) \max (X+Z)=(W+Y) \max (W+Z) \max (X+Y) \max (X+Z)}}{(W+Y) \max (W+Z) \max (X+Y) \max (X+Z)=(W+Y \max Z) \max (X+Y) \max (X+Z)}$$

Donde el primer paso es una aplicación de la regla de sustitución, el segundo de Leibniz y el último de simetría de la igualdad.

$$\begin{array}{ll} & (W+Y) \max (W+Z) \max (X+Y) \max (X+Z) \\ = & \{ \text{Distributividad del } + \text{ respecto del } \max, \text{ con } P := W, Q := Y, R := Z \} \\ & (W+Y \max Z) \max (X+Y) \max (X+Z) \\ = & \{ \text{Distributividad del } + \text{ respecto del } \max, \text{ con } P := X, Q := Y, R := Z \} \\ & (W+Y \max Z) \max (X+Y \max Z) \\ = & \{ \text{Conmutatividad del } \max, \text{ con } P := W, Q := Y \max Z \} \\ & (Y \max Z + W) \max (X+Y \max Z) \\ = & \{ \text{Conmutatividad del } \max, \text{ con } P := X, Q := Y \max Z \} \\ & (Y \max Z + W) \max (Y \max Z + X) \\ = & \{ \text{Distributividad del } + \text{ respecto del } \max, \text{ con } P := Y \max Z, Q := W, R := X \} \\ & Y \max Z + W \max X \\ = & \{ \text{Conmutatividad del } + \} \\ & W \max X + Y \max Z \end{array}$$

El formato de la demostración anterior es el siguiente:

$$\begin{array}{ll} & E_0 \\ = & \{ \text{Ley aplicada o justificación de } E_0 = E_1 \} \\ & E_1 \\ & \vdots \\ & E_{n-1} \end{array}$$

$=$ { Ley aplicada o justificación de $E_{n-1} = E_n$ }

E_n

luego, usando transitividad, se infiere que $E_0 = E_n$.

Debido a que (casi) todos los pasos de la demostración son explícitos, ésta es quizá más larga de lo esperado. Dado que en informática es necesario usar la lógica para calcular programas, es importante que las demostraciones no sean demasiado voluminosas y menos aún tediosas. Afortunadamente, esto puede conseguirse sin sacrificar formalidad, adoptando algunas convenciones y demostrando algunos *metateoremas*, es decir, teoremas acerca del sistema formal de la lógica. Este último punto será tratado en el capítulo 3.

Hemos adoptado ya la convención de no escribir los paréntesis cuando un operador es asociativo, con el consiguiente ahorro de pasos de demostración. De la misma manera, cuando un operador sea conmutativo, intercambiaremos libremente los términos sin hacer referencia explícita a la regla. Además, cuando no se preste a confusión juntaremos pasos similares en uno y no escribiremos la sustitución aplicada cuando ésta pueda deducirse del contexto. Así, por ejemplo, la demostración anterior quedaría como sigue:

$(W + Y) \max (W + Z) \max (X + Y) \max (X + Z)$

$=$ { Distributividad del + respecto del max }

$(W + Y \max Z) \max (X + Y \max Z)$

$=$ { Distributividad del + respecto del max }

$Y \max Z + W \max X$

5.4. Ejercicios

Ejercicio 5.1
Demostrar.

1. $\lfloor x/m \rfloor = \lfloor \lfloor x \rfloor / m \rfloor$
2. $\lfloor \sqrt{x} \rfloor = \lfloor \sqrt{\lfloor x \rfloor} \rfloor$
3. $\lfloor \lfloor x \rfloor \rfloor = \lfloor x \rfloor$
4. $\lceil \lfloor x \rfloor \rceil = \lfloor x \rfloor$

Ejercicio 5.2
¿ Qué está mal con la siguiente demostración (n, k enteros, $0 < n$ y x real) ?

$k \leq \lfloor n * x \rfloor$

$\equiv$ { definición de piso }

$k \leq n * x$

$\equiv$ { aritmética }

$\frac{k}{n} \leq x$

$\equiv$ { definición de piso }

$\frac{k}{n} \leq \lfloor x \rfloor$

$\equiv$ { aritmética }

$k \leq n * \lfloor x \rfloor$

Por lo tanto, por igualdad indirecta vale que $\lfloor n * x \rfloor = n * \lfloor x \rfloor$
Encontrar un contraejemplo a esta última ecuación

Ejercicio 5.3
¿ Qué está mal con la siguiente demostración (m, n, k enteros, $n \neq 0$) ?

$\lceil \frac{m}{n} \rceil \leq k$

$\equiv$ { definición de techo }

$\frac{m}{n} \leq k$

$\equiv$ { aritmética }

$\frac{m}{n} < k + 1$

$\equiv$ { definición de piso, contrapositiva }

$\lfloor \frac{m}{n} \rfloor < k + 1$

$\equiv$ { aritmética }

$\lfloor \frac{m}{n} \rfloor \leq k$

Por lo tanto, por igualdad indirecta vale que $\lfloor \frac{m}{n} \rfloor = \lceil \frac{m}{n} \rceil$

Ejercicio 5.4
Definir una regla de *desigualdad indirecta* análoga a la de igualdad indirecta. Demostrarla. Usarla para demostrar que para cualquier par de reales a y b, vale que $\lfloor a \rfloor + \lfloor b \rfloor \leq \lfloor a + b \rfloor$

Capítulo 6

Expresiones cuantificadas

> CHEREA: ... J'ai le goût et la besoin de la sécurité. La plupart des hommes sont comme moi. Ils sont incapables de vivre dans un univers où la pensée la plus bizarre peut en une seconde entrer dans la réalité...
>
> CALIGULA: La sécurité et la logique ne vont pas ensemble
>
> Albert Camus: *Caligula*

Una notación muy útil usada en matemática es la que nos permite aplicar operaciones una secuencia de expresiones las cuales dependen de alguna variable. Ejemplos paradigmáticos de esta notación son la sumatoria y la productoria, así también como la definición de conjuntos por comprensión o las cuantificaciones en lógica.

Por ejemplo, es usual escribir expresiones como

$$\sum_{i=0}^{n-1} 2 * i + 1$$

la cual puede leerse como la suma de los primeros n numeros impares.

En esta expresión pueden distinguirse varias componentes. Por un lado es operador usado (la sumatoria), correspondiente a un operador binario (la suma), por otro están las variables (en este caso sólo i) las cuales tienen asociado un rango de variación (i puede tomar valores entre 0 y $n-1$), y por último tenemos la expresión que indica cuales van a ser los términos de la sumatoria ($2*i+1$ en el ejemplo). Para un n dado, la sumatoria puede reducirse a una expresión aritmética de la forma

$$1 + 3 + \cdots + 2 * (n-1) + 2 * n$$

Nótese sin embargo la ventaja de usar la sumatoria respecto de la "expresión" con los puntos suspensivos, dado que esta última no estaría bien definida si por ejemplo n es 0 o 1. Usualmente en matemática se asume que el lector puede discernir estos casos sin inconvenientes. Sin embargo estas ambigüedades son mas perniciosas en el desarrollo formal de programas dado que las expresiones son más complejas y es por lo tanto importante tener más cuidado con los casos límite. Por otro lado, como se verá en este capítulo, pueden proverse reglas explícitas para el manejo de expresiones del estilo de la sumatoria, las cuales nos asguran la corrección de las

operaciones realizadas con ellas y, más importante aún, nos ayudarán a encontrar programas a partir de especificaciones escritas usando estas expresiones. Por otro lado, vamos a generalizar este mecanismo de definición de expresiones a cualquier operador binario asociativo y conmutativo. Esto permite que las reglas provistas se apliquen a un conjunto grande de expresiones, las cuales serán suficientes para especificar casi todos los problemas presentados en este libro.

Lo esencial de este mecanismo para nuestros fines es que provee reglas para realizar cálculos de manera suficientemente simple. Sumado esto a que la mayor parte de las reglas son generales para cualquier operador binario (por supuesto que también hay algunas reglas específicas), hace que el cálculo presentado en este libro pueda aspirar a ser un método eficiente para el desarrollo de programas. Los lectores podrán juzgar por si mismos acerca de la validez de esta afirmación. Diferentes versiones de cálculos que usan expresiones cuantificadas pueden encontrarse en [GS93, DF88, DS90, Kal90, Coh90].

6.1. Introducción

En diversos contextos -aritmética, lógica, teoría de conjuntos, lenguajes de programación, etc.- aparece cierta noción de cuantificación, entendiéndos por esto al uso de variables formales con un alcance delimitado explícitamente las cuales pueden usarse para construir expresiones dependientes de éstas pero sólo dentro de ese alcance.

Se propondrá una notación unificada para estas expresiones. Esta notación debe tener en cuenta al operador con el cual se cuantifica (en nuestro ejemplo la suma), las variables que van a usarse para crear las expresiones (i en el ejemplo), el rango de variación de estas variables ($0 \leq i < n$) y la expresión dependiente de las variables que define los términos de la cuantificación (en nuestro ejemplo $2 * i + 1$).

Una **expresión cuantificada** será entonces de la siguiente forma:

$$\langle \oplus i \ : R : \ T \rangle ,$$

donde $\oplus$ designa un operador asociativo y conmutativo (por ejemplo, $+$, $\vee$, $\wedge$, Max, Min, etc.), R es un predicado que se denomina *rango de especificación* y T es una función de i denominada *término* de la cuantificación. Usaremos i para denotar una secuencia de variables en la cual no importa el orden, usando la expresión $V.i$ para denotar al conjunto de todas las variables de i. La ocurrencia de i junto al operador suele llamarse *variable de cuantificación* o *dummy*. La *variable cuantificada* i (en las expresiones R o T) sólo tiene sentido dentro de los paréntesis cuando necesitemos hacer referencia explícita a la variable cuantificada escribiremos $R.i$ y $T.i$ para el rango y el término respectivamente.

Con esta notación la sumatoria del ejemplo se escribirá como sigue

$$\langle \sum i \ : 0 \leq i < n : \ 2 * i + 1 \rangle ,$$

o también como

$$\langle + i \ : 0 \leq i < n : \ 2 * i + 1 \rangle ,$$

Esta notación tiene varias ventajas sobre la usual de matemática las cuales son esenciales para el uso sistemático de las expresiones cuantificadas en el desarrollo de programas. Por un lado los paréntesis determinan exactamente el alcance de la variable. Por otro lado, en matemática es bastante engorroso escribir rangos que no sean intervalos de número naturales. Las expresiones cuantificadas presentadas aquí admiten cualquier expresión booleana como rango, permitiendo

además usar más de una variable cuantificada de manera natural, lo cual es casi imposible de escribir con la notación tradicional.

El tipo de la variable puede en general inferirse del contexto, en caso contrario se lo definirá explícitamente. Para los fines de la cuantificación la propiedad de que una variable sea de un tipo dado es equivalente a la de pertenecer al conjunto de valores de ese tipo. En este sentido, vamos a considerar a los tipos como un predicado más.

En algunas ocasiones no se desea restringir el rango de especificación al conjunto de variables que satisfacen un predicado R, como hemos escrito más arriba. En estos casos escribiremos $\langle \oplus i \ :: \ T.i \rangle$, entendiéndose que el rango abarca a todos los elementos del tipo de i.

Existe una serie de reglas que sirven para manipular expresiones cuantificadas. Las enunciaremos para un operador general $\oplus$ y luego las ejemplificaremos aplicándolas a una serie de operadores usuales. En lo que sigue, usaremos *True* y *False* para indicar los predicados constantemente iguales a "verdadero" y "falso" respectivamente. Por lo dicho anteriormente, $\langle \oplus i \ :: \ T.i \rangle \equiv \langle \oplus i \ : \mathit{True} : \ T.i \rangle$.

Uno de los conceptos esenciales para comprender y manejar a las expresiones cuantificadas es el de *variable ligada*. Asociados a este concepto estan el complementario de *variable libre* y el de *alcance*.

Consideremos otra vez el ejemplo de la sumatoria $\langle \sum i \ : 0 \leq i < n : \ 2 * i + 1 \rangle$. Es claro que el valor de esta sumatoria depende de n y para diferentes estados va a tomar diferentes valores. Sin embargo su valor no depende del valor de i en un estado dado, y puede cambiarse este nombre por otro sin que su valor cambie, por ejemplo $\langle \sum j \ : 0 \leq j < n : \ 2 * j + 1 \rangle$. Vamos a decir que las ocurrencias de la variable i (o j en la segunda versión) en el rango y el término están ligadas (a la variable que aparece junto a la sumatoria). Ahora, en una expresión más complicada de la forma

$$i + \langle \sum i \ : 0 \leq i < n : \ 2 * i + 1 \rangle$$

la ocurrencia de i de antes de la suma no tiene nada que ver con las ocurrencias dentro del alcance de la expresión cuantificada. Sólo esa primera ocurrencia de i tomará su valor de algún estado dado. Las ocurrencias internas a la sumatoria tienen ya su rango especificado. Por ejemplo, en un estado en el cual ambas i y n tomen el valor 6, el valor de la sumatoria va a ser

$$6 + \langle \sum i \ : 0 \leq i < 6 : \ 2 * i + 1 \rangle$$

o sea

$$6 + 1 + 3 + 5 + 7 + 9 + 11$$

Definición 6.1 ((variable libre))
Se define inductivamente cuando una variable está libre en una expresión.

- Una variable i está libre en la expresión i.
- Si la variable i está libre en E entonces lo está también en (E).
- Si la variable i está libre en E y f es una operación válida en el tipo de E, entonces i también está libre en $f.(\ldots, E, \ldots)$.
- Si la variable i está libre en E y i no aparece en la secuencia de variables x ($i \notin V.x$), entonces lo está también en $\langle \oplus x \ : F : \ E \rangle$ y en $\langle \oplus x \ : E : \ F \rangle$.

Dada una expresión E, el conjunto de las variables libres de E se denotará con $FV.E$.

Definición 6.2 ((variable ligada))
Sea i una variable libre en la expresión E y $i \in V.x$, luego la variable i está ligada (a la variable de cuatificación correspondiente) en las expresiones $\langle \oplus x \ : E : \ F \rangle$ y en $\langle \oplus x \ : F : \ E \rangle$

Se extiende la definición inductivamente. Si i está ligada en E, también lo estará (a la misma variable de cuantificación) en (E), $f.(\ldots, E, \ldots)$, $\langle \oplus x \ : F : \ E \rangle$ y en $\langle \oplus x \ : E : \ F \rangle$.

Dada una expresión E, el conjunto de las variables ligadas de E se denotará con $BV.E$.

Ejemplo 6.1
Consideremos la expresión

$$E = i * k + \langle \sum i \ : 0 \leq i < n : \ 2 * i + 1 \rangle$$

luego, $FV.E = \{i, k, n\}$ y $BV.E = \{i\}$.

6.2. Revisión de la sustitución y la regla de Leibniz

Las variables ligadas tienen su alcance delimitado de manera explícita y ligadas a una variable de cuantificación. Si se cambian ambas por un nombre "fresco" (que no aparezca dentro del alcance), el significado de la expresión no cambiará, como lo ejemplifica la siguiente igualdad:

$$\langle \sum i \ : 0 \leq i < n : \ 2 * i + 1 \rangle = \langle \sum j \ : 0 \leq j < n : \ 2 * j + 1 \rangle$$

Debe tenerse cuidado sin embargo con las colisiones de nombres, dado que si por ejemplo se reemplaza a i por n nos da una expresión la cual es obviamente diferente de las anteriores (en particular será siempre igual a 0, sin importar el valor de n en el estado).

$$\langle \sum n \ : 0 \leq n < n : \ 2 * n + 1 \rangle$$

para extender la sustitución a expresiones cuantificadas a través del siguiente axioma:

Axioma 6.1 (Sustitución para expresiones cuantificadas)

$$V.y \cap (V.x \cup FV.E) = \emptyset \Rightarrow \langle \oplus y \ : R : \ T \rangle (x := E) = \langle \oplus y \ : R(x := E) : \ T(x := E) \rangle$$

La condición sobre las variables es necesaria para evitar las colisiones de nombres. Si la condición no se cumple, es necesario renombrar la variable de cuantificación. Esto se verá en la sección siguiente, axioma 6.10.

Regla de Leibniz.

El objetivo de la regla de Leibniz es permitir el reemplazo de iguales por iguales. Sin embargo, con su formulación actual, esto no siempre es posible cuando aparecen expresiones cuantificadas. Por ejemplo, es esperable que aplicando la regla de Leibniz pueda deducirse que

$$\langle \sum i \ : 0 \leq i < n : \ 2 * i + 1 \rangle = \langle \sum i \ : 0 \leq i < n : \ 2 * (i + 1) - 1 \rangle$$

Recordemos rapidamente la versión actual de la regla de Leibniz

$$\textbf{Leibniz: } \frac{X = Y}{E(x := X) = E(x := Y)}$$

La forma en que podría usarse para demostrar la igualdad deseada es la siguiente

$$\frac{2 * i + 1 = 2 * (i + 1) - 1}{\langle \sum i \ : 0 \leq i < n : \ y \rangle (y := 2 * i + 1) = \langle \sum i \ : 0 \leq i < n : \ y \rangle (y := 2 * (i + 1) - 1)}$$

Pero, dado que la variable i aparece necesariamente en la lista de variables de cuantificación, el resultado de $\langle \sum i \ : 0 \leq i < n : \ y \rangle (y := 2 * i + 1)$ será $\langle \sum j \ : 0 \leq j < n : \ 2 * i + 1 \rangle$ y no el esperado.

Proponemos entonces generalizar la regla de Leibniz para las expresiones cuantificadas, agregando las siguentes dos reglas (para el rango y para el término).

$$\frac{X = Y}{\langle \oplus i \ : R(i := X) : \ T \rangle = \langle \oplus i \ : R(i := Y) : \ T \rangle}$$

$$\frac{X = Y}{\langle \oplus i \ : R : \ T(i := X) \rangle = \langle \oplus i \ : R : \ T(i := X) \rangle}$$

6.3. Reglas generales para las expresiones cuantificadas

En esta sección se enunciarán axiomas y demostrarán algunas propiedades que van a ser útiles en el desarrollo de programas. Se le darán nombres a estos axiomas y propiedades para poder mencionarlos luego en las demostraciones o derivaciones pertinentes. En los casos en que una propiedad sea una generalización de un axioma, conservaremos en general el nombre del axioma ya que quedará claro a partir del contexto que versión de la regla estará siendo usada.

Axioma 6.2 (Rango vacío)
Cuando el rango de especificación es vacío, la expresión cuantificada es igual al elemento neutro e del operador $\oplus$:

$$\langle \oplus i \ : \mathit{False} : \ T \rangle = e$$

Si $\oplus$ no posee elemento neutro, la expresión no está bien definida.

Axioma 6.3 (Rango unitario)
Si el rango de especificación consiste en un solo elemento, la expresión cuantificada es igual al término evaluado en dicho elemento:

$$\langle \oplus i \ : i = N : \ T \rangle = T(i := N)$$

Otra forma de escribir esta regla es hacer explícita la dependencia de T de la variable i (lo cual obviamente no significa que i puede no aparecer en T).

$$\langle \oplus i \ : i = N : \ T.i \rangle = T.N$$

Axioma 6.4 (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,

– los predicados R y S son disjuntos,

la expresión cuantificada puede reescribirse como sigue:

$$\langle \oplus i \ : R \vee S : \ T \rangle = \langle \oplus i \ : R : \ T \rangle \oplus \langle \oplus i \ : S : \ T \rangle$$

El hecho que la igualdad sea una relación simétrica, permite indistintamente reemplazar cualquiera de los dos miembros por el otro, vale decir que la regla de partición de rango puede leerse también de derecha a izquierda. Lo mismo ocurre con todas las reglas que siguen.

Axioma 6.5 (Partición de rango generalizada)
Si el operador $\oplus$ es idempotente y el rango de especificación es una cuantificación existencial (ver 6.7), entonces:

$$\langle \oplus i \ : \langle \exists j \ : S.i.j : \ R.i.j \rangle : \ T.i \rangle = \langle \oplus i, j \ : S.i.j \wedge R.i.j : \ T.i \rangle$$

Axioma 6.6 (Regla del término)
Cuando el operador $\oplus$ aparece en el término de la cuantificación, la expresión cuantificada puede reescribirse de la siguiente manera:

$$\langle \oplus i \ : R : \ T \oplus G \rangle = \langle \oplus i \ : R : \ T \rangle \oplus \langle \oplus i \ : R : \ G \rangle$$

Axioma 6.7 (Regla del término constante)
Si el término de la cuantificación es constantemente igual a C, la variable cuantificada i no aparece en C, el operador $\oplus$ es idempotente y el rango de especificación es no vacío, entonces:

$$\langle \oplus i \ : R : \ C \rangle = C$$

Axioma 6.8 (Distributividad)
Si $\otimes$ es distributivo a izquierda con respecto a $\oplus$ 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 absorvente para $\otimes$,

entonces:

$$\langle \oplus i \ : R : \ x \otimes T \rangle = x \otimes \langle \oplus i \ : R : \ T \rangle$$

Análogamente, si $\otimes$ es distributivo a derecha con respecto a $\oplus$ y se cumple al menos una de las condiciones anteriores, entonces:

$$\langle \oplus i \ : R : \ T \otimes x \rangle = \langle \oplus i \ : R : \ T \rangle \otimes x$$

Axioma 6.9 (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:

$$\langle \oplus i, j \ : R.i \wedge S.i.j : \ T.i.j \rangle = \langle \oplus i \ : R.i : \ \langle \oplus j \ : S.i.j : \ T.i.j \rangle \rangle$$

Axioma 6.10 (Regla de cambio de variable)
Si $V.j \cap (FV.R \cup FV.T) = \emptyset$ pueden renombrarse las variables

$$\langle \oplus i \ : R : \ T \rangle = \langle \oplus j \ : R(i := j) : \ T(i := j) \rangle$$

Puede también escribirse usando una referencia explícita a i

$$\langle \oplus i \ : R.i : \ T.i \rangle = \langle \oplus j \ : R.j : \ T.j \rangle$$

Todas estas reglas pueden particularizarse, refiriéndose a operadores concretos. Es lo que haremos en adelante, con los operadores más usuales.

Teorema 6.11 (Regla de cambio de variable)
Si f es una función que tenga inversa (sea biyectiva) en el rango de especificación y j es una variable que no aparece en R ni en T, las variables cuantificadas pueden renombrarse como sigue:

$$\langle \oplus i \ : R.i : \ T.i \rangle = \langle \oplus j \ : R.(f.j) : \ T.(f.j) \rangle$$

La inversa en el rango considerado se denotará con f^{-1}. La propiedad de ser ineversa puede escribirse como

$$\left\langle \forall i, j \ : R.i \wedge R.(f.j) : \ f.i = j \equiv i = f^{-1}.j \right\rangle$$

Comenzamos la demostración con la expresión más complicada

$$\begin{aligned}
& \langle \oplus j \ : R.(f.j) : \ T.(f.j) \rangle \\
= & \ \{ \text{ Rango unitario (introducción de la cuantificación sobre } i) \ \} \\
& \langle \oplus j \ : R.(f.j) : \ \langle \oplus i \ : i = f.j : \ T.i \rangle \rangle \\
= & \ \{ \text{ Anidado } \} \\
& \langle \oplus i, j \ : R.(f.j) \wedge i = f.j : \ T.i \rangle \\
= & \ \{ \text{ Leibniz } \} \\
& \langle \oplus i, j \ : R.i \wedge i = f.j : \ T.i \rangle \\
= & \ \{ \text{ Anidado } \} \\
& \langle \oplus i \ : R.i : \ \langle \oplus j \ : i = f.j : \ T.i \rangle \rangle \\
= & \ \{ \text{ Inversa } \} \\
& \left\langle \oplus i \ : R.i : \ \left\langle \oplus j \ : j = f^{-1} : \ T.i \right\rangle \right\rangle \\
= & \ \{ \text{ Rango unitario, } j \notin FV.T \ \} \\
& \langle \oplus i \ : R.i : \ T.i \rangle
\end{aligned}$$

Teorema 6.12 (Separación de un término)
$\langle \oplus i \ : 0 \leq i < n : \ T.i \rangle = T.0 \oplus \langle \oplus i \ : 0 \leq i < n : \ T.(i+1) \rangle$

Teorema 6.13 (Separación de un término)
$\langle \oplus i \ : 0 \leq i < n : \ T.i \rangle = \langle \oplus i \ : 0 \leq i < n : \ T.i \rangle \oplus T.n$

6.4. Cuantificadores aritméticos

Sumatoria y productoria.

Dos cuantificadores aritméticos usuales son los que provienen de los operadores suma y producto, denotados Σ y Π respectivamente. Enunciaremos las reglas para la sumatoria, dejando las de la productoria como ejercicio.

El operador $+$ tiene por elemento neutro al cero y no es idempotente. Teniendo esto en cuenta, obtenemos las siguientes reglas:

Rango vacío: $\langle \sum i \ : \mathit{False} : \ T.i \rangle = 0$.

Rango unitario: $\langle \sum i \ : i = N : \ T.i \rangle = T.N$.

Partición de rango: si R y S son disjuntos,

$$\langle \sum i \ : R.i \vee S.i : \ T.i \rangle = \langle \sum i \ : R.i : \ T.i \rangle + \langle \sum i \ : S.i : \ T.i \rangle .$$

Ejemplo:

$$\begin{aligned} & \langle \sum i \ : 0 < i \leq 2 * n : \ i \rangle \\ = & \ \{ \text{ reescritura del rango para lograr una disyunción de predicados disjuntos } \} \\ & \langle \sum i \ : 0 < i \leq n \vee n < i \leq 2 * n : \ i \rangle \\ = & \ \{ \text{ partición de rango } \} \\ & \langle \sum i \ : 0 < i \leq n : \ i \rangle + \langle \sum i \ : n < i \leq 2 * n : \ i \rangle \end{aligned}$$

Regla del término: $\langle \sum i \ : R.i : \ T.i + G.i \rangle = \langle \sum i \ : R.i : \ T.i \rangle + \langle \sum i \ : R.i : \ G.i \rangle$.

Distributividad: como $*$ es distributivo con respecto a $+$ a derecha y a izquierda, si R es no vacío,
$\langle \sum i \ : R.i : \ x * T.i \rangle = x * \langle \sum i \ : R.i : \ T.i \rangle$
y
$\langle \sum i \ : R.i : \ T.i * x \rangle = \langle \sum i \ : R.i : \ T.i \rangle * x$.

Regla de anidado: $\langle \sum i, j \ : R.i \wedge S.i.j : \ T.i.j \rangle = \langle \sum i \ : R.i : \ \langle \sum j \ : S.i.j : \ T.i.j \rangle \rangle$.

Regla de cambio de variable: para toda f biyectiva y para cualquier j que no aparezca en R ni en T,

$\langle \sum i \ : R.i : \ T.i \rangle = \langle \sum j \ : R.(f.j) : \ T.(f.j) \rangle$.

Las reglas de partición de rango generalizada y del término constante no se aplican porque $+$ no es idempotente.

Máximo y mínimo.

Otros operadores aritméticos que resultan de gran utilidad para especificar programas son Max y Min (ver la definición en la sección 2.3, ejercicio 5.2).

Pueden tomarse las siguientes propiedades como definiciones de las versiones cuantificadas del máximo y del mínimo:

$$z = \langle \text{Max}\, i \ : R.i : \ F.i \rangle \equiv \langle \exists\, i \ : R.i : \ z = F.i \rangle \wedge \langle \forall\, i \ : R.i : \ F.i \leq z \rangle$$

$$z = \langle \text{Min}\, i \ : R.i : \ F.i \rangle \equiv \langle \exists\, i \ : R.i : \ z = F.i \rangle \wedge \langle \forall\, i \ : R.i : \ z \leq F.i \rangle$$

Una consecuencia de estas propiedades es la siguiente:

$$F.x = \langle \text{Max}\, i \ : R.i : \ F.i \rangle \equiv R.x \wedge \langle \forall\, i \ : R.i : \ F.i \leq F.x \rangle$$

$$F.x = \langle \text{Min}\, i \ : R.i : \ F.i \rangle \equiv R.x \wedge \langle \forall\, i \ : R.i : \ F.x \leq F.i \rangle$$

Ninguno de los dos operadores tiene un neutro en los enteros. Por conveniencia, extenderemos los enteros con dos constantes que denotaremos con ∞ y $-\infty$, las cuales serán, por definición, neutros para el mínimo y el máximo respectivamente. Las operaciones aritméticas usuales no estarán definidas para estas constantes. En determinados casos es posible elegir otros elementos neutros. Por ejemplo, cuando se usa el operador de máximo para números naturales es posible tomar al 0 como neutro.

Se deja como ejercicio para el lector el enunciado de todas las reglas para la cuantificación de los operadores Max y Min.

Operador de conteo.

Hasta aquí hemos mencionado únicamente cuantificadores que provienen de un operador conmutativo y asociativo. Pero también es posible definir nuevos cuantificadores a partir de otros ya definidos. Éste es el caso del cuantificador N, el cual cuenta la cantidad de elementos en el rango de especificación que satisfacen el término de la cuantificación:

$$\langle N\, i \ : R.i : \ T.i \rangle \doteq \langle \textstyle\sum i \ : R.i \wedge T.i : \ 1 \rangle$$

Por ejemplo, $\langle N\, x \ : x \in S : \ par.x \rangle$ cuenta la cantidad de elementos pares que hay en el conjunto S. Nótese que T es una función booleana.

Las propiedades del cuantificador N pueden calcularse a partir de las de la suma (ver ejercicio 6.10 y sección 8.5).

6.5. Expresiones cuantificadas para conjuntos

La unión de conjuntos es un operador conmutativo y asociativo, por lo tanto podemos cuantificarlo:

$$\langle \cup\, i \ : R.i : \ C.i \rangle \,,$$

donde para todo i la expresión $C.i$ denota un conjunto.

El operador $\cup$ es idempotente y posee elemento neutro. Queda como ejercicio para el lector escribir las reglas correspondientes a la expresión cuantificada de la unión de conjuntos.

Los conjuntos definidos por comprensión pueden verse también como una cuantificación a través de la siguiente definición (usando la notación usual de matemática para los conjuntos definidos por comrensión):

$$\{e.i \,|\, R.i\} \doteq \langle \cup\, i\ : R.i :\ \{e.i\}\ \rangle$$

Las variables de cuantificación quedan implícitas (no se escriben), y en matemática pueden en general deducirse a partir del contexto. Para evitar confusiones y no tener que usar expresiones con la unión (las cuales son ligeramente más engorrosas), usaremos una notación coherente con los otros cuantificadores.

$$\{i\ :\ R.i : \ e.i\} \doteq \langle \cup\, i\ : R.i :\ \{e.i\}\ \rangle$$

La ventaja de pensar a los conjuntos definidos por comprensión como se indica más arriba es que al hacerlo, éstos heredan algunas propiedades de la expresión cuantificada de la unión.

Ejercicio 6.1
Dar las reglas que satisfacen los conjuntos definidos por comprensión.

Una regla particular para expresiones cuantificadas para conjuntos es la siguiente:

$$x \in \langle \cup\, i\ : R.i :\ T.i\ \rangle \equiv \langle\, \exists\, i\ : R.i :\ x \in T.i\ \rangle$$

Una consecuencia de esta regla es:

$$x \in \{i\ :\ R.i\ :\ e.i\} \equiv \langle\, \exists\, i\ : R.i :\ x = e.i\ \rangle$$

y un caso particular de ésta es:

$$x \in \{y\ :\ R.y\ :\ y\} \equiv R.x$$

Ejercicio 6.2
Demostrar que las dos últimas reglas se derivan a partir de la primera.

6.6. El cuantificador universal

Uno de ellos es el asociado a la conjunción, que es un operador booleano asociativo y conmutativo:

$$\langle \wedge\, i\ : R.i :\ T.i\ \rangle$$

Esta operación es muy conocida en lógica, por lo que usaremos una notación particular:

$$\langle\, \forall\, i\ : R.i :\ T.i\ \rangle \doteq \langle \wedge\, i\ : R.i :\ T.i\ \rangle\,,$$

la cual recibe el nombre de *cuantificación universal*. Nótese que la expresión $T.i$ es un predicado.

El operador $\wedge$ es idempotente y posee elemento neutro *True*. Así, al aplicar las reglas enunciadas anteriormente, obtenemos:

Rango vacío: $\langle \forall i : False : T.i \rangle \equiv True$.

Rango unitario: $\langle \forall i : i = N : T.i \rangle \equiv T.N$.

Partición de rango: $\langle \forall i : R.i \vee S.i : T.i \rangle \equiv \langle \forall i : R.i : T.i \rangle \wedge \langle \forall i : S.i : T.i \rangle$.

Partición de rango generalizada:

$$\langle \forall i : \langle \exists j : S.i.j : R.i.j \rangle : T.i \rangle \equiv \langle \forall i, j : S.i.j \wedge R.i.j : T.i \rangle$$

Regla del término: $\langle \forall i : R.i : T.i \wedge G.i \rangle \equiv \langle \forall i : R.i : T.i \rangle \wedge \langle \forall i : R.i : G.i \rangle$.

Regla del término constante: si el rango es no vacío, $\langle \forall i : R.i : C \rangle \equiv C$.

Distributividad: como $\vee$ es distributivo con respecto a $\wedge$ a izquierda y a derecha,
$\langle \forall i : R.i : x \vee T.i \rangle \equiv x \vee \langle \forall i : R.i : T.i \rangle$
y
$\langle \forall i : R.i : T.i \vee x \rangle \equiv \langle \forall i : R.i : T.i \rangle \vee x$.

Regla de anidado: $\langle \forall i, j : R.i \wedge S.i.j : T.i.j \rangle \equiv \langle \forall i : R.i : \langle \forall j : S.i.j : T.i.j \rangle \rangle$.

Regla de cambio de variable: para toda f biyectiva y para cualquier j que no aparezca en R ni en T, $\langle \forall i : R.i : T.i \rangle \equiv \langle \forall j : R.(f.j) : T.(f.j) \rangle$.

Para el cuantificador universal hay una regla extra, cuya importancia radica en el hecho que provee un modo de "pasar del rango al término" y viceversa:

***Regla de intercambio*:**

$$\begin{aligned} \langle \forall i : R.i : T.i \rangle &\equiv \langle \forall i : True : \neg R.i \vee T.i \rangle \\ &\equiv \langle \forall i :: R.i \Rightarrow T.i \rangle \end{aligned}$$

En el caso particular del cuantificador universal, muchas de las reglas son derivables a partir de las otras, en general usando la regla de intercambio y el cálculo de predicados (ver por ejemplo [DS90] y ejercicio 6.7).

6.7. El cuantificador existencial

Otro de los cuantificadores que aparece con frecuencia es el asociado a la disyunción, que también es un operador booleano asociativo y conmutativo. La expresión cuantificada que se obtiene a partir del operador $\vee$ es:

$$\langle \exists i : R.i : T.i \rangle \doteq \langle \vee i : R.i : T.i \rangle$$

y recibe el nombre de *cuantificación existencial*.

El operador $\vee$ es idempotente y tiene elemento neutro *False*, por lo cual al aplicar las reglas generales al cuantificador existencial obtenemos:

Rango vacío: $\langle \exists i \ : False : \ T.i \rangle \equiv False$.

Rango unitario: $\langle \exists i \ : i = N : \ T.i \rangle \equiv T.N$.

Partición de rango: $\langle \exists i \ : R.i \vee S.i : \ T.i \rangle \equiv \langle \exists i \ : R.i : \ T.i \rangle \vee \langle \exists i \ : S.i : \ T.i \rangle$.

Partición de rango generalizada:

$\langle \exists i \ : \langle \exists j \ : S.i.j : \ R.i.j \rangle : \ T.i \rangle \equiv \langle \exists i, j \ : S.i.j \wedge R.i.j : \ T.i \rangle$

Regla del término: $\langle \exists i \ : R.i : \ T.i \vee G.i \rangle \equiv \langle \exists i \ : R.i : \ T.i \rangle \vee \langle \exists i \ : R.i : \ G.i \rangle$.

Regla del término constante: si el rango es no vacío, $\langle \exists i \ : R.i : \ C \rangle \equiv C$.

Distributividad: como $\wedge$ es distributivo con respecto a $\vee$ a izquierda y a derecha,

$\langle \exists i \ : R.i : \ x \wedge T.i \rangle \equiv x \wedge \langle \exists i \ : R.i : \ T.i \rangle$
y
$\langle \exists i \ : R.i : \ T.i \wedge x \rangle \equiv \langle \exists i \ : R.i : \ T.i \rangle \wedge x$.

Regla de anidado: $\langle \exists i, j \ : R.i \wedge S.i.j : \ T.i.j \rangle \equiv \langle \exists i \ : R.i : \ \langle \exists j \ : S.i.j : \ T.i.j \rangle \rangle$.

Regla de cambio de variable: para toda f biyectiva y para cualquier j que no aparezca en R ni en T, $\langle \exists i \ : R.i : \ T.i \rangle \equiv \langle \exists j \ : R.(f.j) : \ T.(f.j) \rangle$.

También en este caso hay una regla extra, que relaciona el término de la cuantificación con el rango de especificación:

***Regla de intercambio*:** $\langle \exists i \ : R.i : \ T.i \rangle \ \equiv \ \langle \exists i \ :: \ R.i \wedge T.i \rangle$

Las cuantificaciones universal y existencial están vinculadas a través de dos reglas importantes, que son una generalización de las leyes de De Morgan:

$$\neg \langle \forall i \ : R.i : \ T.i \rangle \ \equiv \ \langle \exists i \ : R.i : \ \neg T.i \rangle$$
$$\neg \langle \exists i \ : R.i : \ T.i \rangle \ \equiv \ \langle \forall i \ : R.i : \ \neg T.i \rangle$$

6.8. Ejercicios

Ejercicio 6.3
Sea $\oplus$ un cuantificador asociado a un operador conmutativo y asociativo. Probar la siguiente regla de eliminación de una *dummy* (Z no depende de i ni de j):

$$\langle \oplus i, j \ : i = Z \wedge R.i.j : \ T.i.j \rangle \equiv \langle \oplus j \ : R.Z.j : \ T.Z.j \rangle$$

Ejercicio 6.4
Demostrar:

$$\langle \exists x, y \ : x = y : \ P.x.y \rangle \equiv \langle \exists x \ :: \ P.x.x \rangle$$

Ejercicio 6.5
Probar que la implicación es distributiva con respecto al cuantificador universal:

$$\langle \forall i \ : R.i : \ Z \Rightarrow T.i \rangle \equiv Z \Rightarrow \langle \forall i \ : R.i : \ T.i \rangle$$

Ejercicio 6.6
Probar las *reglas de instanciación* para la cuantificación universal y existencial:

1. $\langle \forall i \ :: \ f.i \rangle \Rightarrow f.x$

2. $f.x \Rightarrow \langle \exists i \ :: \ f.i \rangle$

Ejercicio 6.7
1. Probar la siguiente versión de la regla de intercambio para el cuantificador universal:

$$\langle \forall i \ : R.i \wedge S.i : \ T.i \rangle \equiv \langle \forall i \ : R.i : \ S.i \Rightarrow T.i \rangle$$

2. Suponiendo válidas las reglas del término, de intercambio, anidado, distributividad, De Morgan e instanciación, demostrar:

   (i) $\langle \forall i \ : R.i : \ \mathit{True} \rangle \equiv \mathit{True}$
   (ii) Partición de rango.
   (iii) Partición de rango generalizada.
   (iv) Cambio de variables. Sugerencia: demostrar las siguientes implicaciones:
   - $\langle \forall i \ : R.i : \ T.i \rangle \Rightarrow \langle \forall j \ : R.(f.j) : \ T.(f.j) \rangle$ para *cualquier* función f.
   - Usando lo anterior demostrar $\langle \forall i \ : R.i : \ T.i \rangle \Leftarrow \langle \forall j \ : R.(f.j) : \ T.(f.j) \rangle$ para f *invertible.*

Ejercicio 6.8
Probar la regla de intercambio para el cuantificador existencial usando la del cuantificador universal y De Morgan:

$$\langle \exists i \ : R.i : \ T.i \rangle \equiv \langle \exists i \ :: \ R.i \wedge T.i \rangle$$

Ejercicio 6.9
Demostrar la siguiente relación entre el máximo y el mínimo:

$$\langle \text{Min} \ \ i \ : R.i : \ -F.i \rangle = - \langle \text{Max} \ \ i \ : R.i : \ F.i \rangle$$

Ejercicio 6.10
El cuantificador aritmético N no está definido en base a un operador subyacente sino a través de la cuantificación de la suma:

$$\langle N i \ : R.i : \ T.i \rangle \doteq \langle \sum i \ : R.i \wedge T.i : \ 1 \rangle$$

1. Enunciar y demostrar la regla de partición de rango para N.

2. Ídem con la regla del rango vacío.

3. Probar: $\langle \sum i \ : R.i \wedge T.i : \ k \rangle = k * \langle N i \ : R.i : \ T.i \rangle$

Ejercicio 6.11
[Kal90] Suponiendo que:

(i) $x = \langle \sum i \ : R.i : \ f.i \rangle$

(ii) $R.i \not\equiv i = N$ para cualquier i

calcular una expresión libre de cuantificadores que sea equivalente a

$$\langle \sum i \ : R.i \vee i = N : \ f.i \rangle$$

Ejercicio 6.12
[Kal90] Suponiendo que:

(i) $x = \langle \text{Max}\, i, j \ : R.i.j \wedge j < N + 1 : \ f.i.j \rangle$

(ii) $y = \langle \text{Max}\, i \ : R.i.(N + 1) : \ f.i.(N + 1) \rangle$

calcular una expresión libre de cuantificadores que sea equivalente a

$$(\text{Max}\, i, j \ : \ R.i.j \wedge (j < N + 1 \vee j = N + 1) \ : \ f.i.j)$$

Ejercicio 6.13
[Kal90] Suponiendo que:

(i) $x = \langle \text{Max}\, i \ : R.i.N : \ f.i.N \rangle$

(ii) $R.y.(z + 1) = R.y.z \ \vee \ y = z + 1$ para cualquier y, z

(iii) $f.y.y = 0$ para cualquier y

(iv) $f.y.(z + 1) = f.y.z + g.z$ para cualquier y, z

(v) $R.y.z \not\equiv \mathit{False}$ para cualquier y, z

calcular una expresión libre de cuantificadores que sea equivalente a

$$\langle \text{Max}\, i \ : R.i.(N + 1) : \ f.i.(N + 1) \rangle$$

Ejercicio 6.14
[Kal90] Suponiendo que:

(i) $x \equiv \langle \forall i, j \ : R.i.j \wedge j < N : \ f.i \Rightarrow f.j \rangle$

(ii) $y \equiv \langle \forall i \ : R.i.N : \ \neg f.i \rangle$

calcular una expresión libre de cuantificadores que sea equivalente a:

$$\langle \forall i, j \ : R.i.j \wedge (j < N \vee j = N) : \ f.i \Rightarrow f.j \rangle$$

Ejercicio 6.15
Demostrar que para cualquier $\oplus$, R, S y T se cumple:

$$\langle \oplus i \ : R : \ T \rangle \oplus \langle \oplus i \ : S : \ T \rangle \equiv \langle \oplus i \ : R \vee S : \ T \rangle \oplus \langle \oplus i \ : R \wedge S : \ T \rangle$$

Capítulo 7

Cálculo de predicados

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 muy simples. Un caso paradigmático es el de los silogismos, tratados en la sección ??. Uno de los ejemplos considerados fue el siguiente.

Las arañas son mamíferos.
Los mamíferos tienen seis patas.

Las arañas tienen seis patas.

Si quisiéramos analizar este razonamiento usando el cálculo proposicional nos quedaría claramente una forma invalida, dado que cada premisa, - así también como la conclusión - es una proposición elemental distinta.

Para poder demostrar la validez de este tipo de razonamientos va a ser necesario disponer de herramientas que permitan analizar la estructura interna de las proposiciones elementales.

7.1. Predicados

Un *predicado* es una función con codominio en los booleanos, es decir, que toma valores verdadero o falso. Por ejemplo el predicado 'menor o igual' $\leq$ toma dos números como argumentos; el predicado 'es mujer' toma personas como argumentos.

Cuando se estudia formalmente el sistema de la lógica, es necesario especificar el lenguaje que se usa. Para ello se introducen de manera explícita los nombres o símbolos de predicados así también como los nombres o símbolos de función pertinentes. Por ejemplo, si se quiere trabajar con un sistema formal para la aritmética, se tendrán símbolos de predicado para la igualdad, las desigualdades, etc, y símbolos de función para el cero, el sucesor de un número, la suma, etc.

Los símbolos de predicado nos permitirán construir nuevas proposiciones elementales, extendiendo de esta manera el cálculo proposicional presentado en el capítulo 3. Por otro lado, se usará la cuantificación universal y existencial introducida en el capítulo precedente.

Dado un conjunto de símbolos de predicado y otro de símbolos de función, el cálculo puro de predicados tendrá como axiomas y reglas de inferencia a las del cálculo proposicional y las de las cuantificaciones.

Por otro lado, cuando se quiera trabajar en una *teoría* determinada (por ejemplo la aritmética) se introducirán axiomas que expresen las propiedades que deben satisfacer las operaciones y predicados que aparecen en el lenguaje.

7.2. Propiedades de las cuantificaciones universal y existencial

Muchas de las propiedades de las cuantificaciones fueron enunciads en el capítulo 6, especialente en las secciones 6.6 y 6.7 y en algunos ejercicios. Por otro lado, en el ejercicio 6.7 se ve que pueden axiomatizarse los cuantificadores lógicos con menos axiomas que otros (debido esencialmente a la regla de intercambio).

En esta sección enunciaremos algunas otras propiedades y metateoremas que van a ser útiles a la hora de derivar programas. Las demostraciones se omitirán casi siempre quedando como ejercicio para el lector.

Teorema 7.1
Siempre que $i \notin FV.P$ y que el rango no sea vacío (es decir que $\langle \exists i :: R \rangle$) entonce vale la siguiente ley distributiva

$$\langle \forall i : R : P \wedge Q \rangle \equiv P \wedge \langle \forall i : R : Q \rangle$$

Nótese que es necesario pedir que el rango no sea vacío, a diferencia de la distributividad de la disyunción, dado que el neutro de la conjunción (*True*) es absorbente para la disyunción.

Teorema 7.2
$\langle \forall i : R : P \equiv Q \rangle \equiv \langle \forall i : R : P \rangle \equiv \langle \forall i : R : Q \rangle$

Teorema 7.3 (Fortalecimiento por término)

$$\langle \forall i : R : P \wedge Q \rangle \Rightarrow \langle \forall i : R : P \rangle$$

Teorema 7.4 (Fortalecimiento por rango)

$$\langle \forall i : R \vee Q : P \rangle \Rightarrow \langle \forall i : R : P \rangle$$

Teorema 7.5 (Monotonía)

$$\langle \forall i : R : P \Rightarrow Q \rangle \Rightarrow (\langle \forall i : R : P \rangle \Rightarrow \langle \forall i : R : Q \rangle)$$

Metateorema 7.6
P es un teorema si y solo si $\langle \forall i :: P \rangle$ es un teorema

La demostración puede hacerse por doble implicación. Una de ellas es inmediata por la propiedad de instanciación. Para ver la otra mostraremos como transformar una demostración de P en una de $\langle \forall i :: P \rangle$.

Supongamos que tenemos una demostración de P de la siguiente forma.

$$\begin{array}{ll} & P \\ \equiv & \{ \text{ Razón 1 } \} \\ & P_0 \\ & \vdots \\ & P_n \end{array}$$

$\equiv$ { Razón n }
True

Podemos construir entonces de manera mecánica la siguiente demostración de $\langle \forall i :: P \rangle$

$\langle \forall i :: P \rangle$
$\equiv$ { Razón 1 }
$\langle \forall i :: P_0 \rangle$
$\vdots$
$\langle \forall i :: P_n \rangle$
$\equiv$ { Razón n }
$\langle \forall i :: True \rangle$
$\equiv$ { Término constante }
True

Teorema 7.7
Siempre que $i \notin FV.P$ y que el rango no sea vacío (es decir que $\langle \exists i :: R \rangle$) entonce vale la siguiente ley distributiva

$$\langle \exists i : R : P \vee Q \rangle \equiv P \vee \langle \exists i : R : Q \rangle$$

Nótese que es necesario pedir que el rango no sea vacío, a diferencia de la distributividad de la disyunción, dado que el neutro de la conjunción (*True*) es absorbente para la disyunción.

Teorema 7.8 (Debilitamiento. Término)

$$\langle \exists i : R : P \rangle \Rightarrow \langle \exists i : R : P \vee Q \rangle$$

Teorema 7.9
Debilitamiento.

$$\langle \exists i : R : P \rangle \Rightarrow \langle \exists i : R \vee Q : P \rangle$$

Teorema 7.10
Monotonía.

$$\langle \exists i : R : P \Rightarrow Q \rangle \Rightarrow (\langle \exists i : R : P \rangle \Rightarrow \langle \exists i : R : Q \rangle)$$

Teorema 7.11
Intercambio de cuantificadores. Si $i \notin FV.R$ y $j \notin FV.Q$, entonces

$$\langle \exists j : R : \langle \forall i : Q : P \rangle \rangle \Rightarrow \langle \forall i : Q : \langle \exists i : R : P \rangle \rangle$$

El siguiente metateorema nos permite usar un nombre de constante nuevo (el *testigo*) para referirse a un elemento cuya existencia es postulada por el cuantificador.

Metateorema 7.12
(testigo). Si $k \notin (FV.P \cup FV.Q)$ entonces $\langle \exists i :: P \rangle \Rightarrow Q$ si y solo si $P(i := k) \Rightarrow Q$ es un teorema.

$\langle \exists i :: P \rangle \Rightarrow Q$
$\equiv$ { implicaión }
$\neg \langle \exists i :: P \rangle \vee Q$
$\equiv$ { de Morgan }
$\langle \forall i :: \neg P \rangle \vee Q$
$\equiv$ { cambio de variables, $k \notin FV.P$ }
$\langle \forall k :: \neg P(i := k) \rangle \vee Q$
$\equiv$ { distributividad, $k \notin FV.Q$ }
$\langle \forall k :: \neg P(i := k) \vee Q \rangle$
$\equiv$ { implicación }
$\langle \forall k :: P(i := k) \Rightarrow Q \rangle$

Podemos aplicar ahora el metateorema 7.6 de la cuantificación universal y vemos que la última línea es un teorema si y solo si $P(i := k) \Rightarrow Q$ lo es.

7.3. Aplicaciones del cálculo de predicados

7.4. Algunas conclusiones

En el capítulo **??** nos propusimos algunas preguntas que una teoría del razonamiento deductivo debería intentar responder. Estamos ya en condiciones de responderlas, al menos parcialmente.

- *¿Cómo puede determinarse si un razonamiento es válido?*

  Podemos escribir todos los pasos de un razonamiento en notación formal y determinar si cada paso es un paso válido de acuerdo con las reglas definidas de manera explícita para el cálculo de predicados.

- *¿Cómo puede determinarse si una conclusión es consecuencia de un conjunto de premisas, y de ser así cómo puede demostrarse que lo es?*

  Para el caso de la lógica proposicional, esta pregunta puede resolverse con relativa facilidad. Se construye la tabla de verdad para la fórmula asociada el razonamiento pertinente y puede afirmarse usando el teorema de completitud que es una tautología si y sólo si la conclusión es consecuencia de las premisas.

  Para el caso de la lógica de predicados en cambio no existe ningún método mecánico para decidir si una conclusión puede deducirse de un conjunto de premisas. La teoría de la computación o computabilidad demuestra que un tal algoritmo no puede existir.

- *¿Qué características de las estrucuturas del mundo y del lenguaje y de las relaciones entre palabras, cosas y pensamientos hacen posible el razonamiento deductivo?*

  Esta pregunta no está aún resuelta de manera satisfactoria y existen numerosas teorías que proponen posibles respuestas. Los ejemplos que hemos considerado en este libro sugieren

que la relación entre lenguaje y mundo no es arbitraria en su totalidad y que cierta estructura del lenguaje refleja cierta estructura del mundo. Las constantes de nuestra lógica puede pensarse que denotan objetos, los predicados relaciones o propiedades. Sin embargo como se establece esta relación de denotación es aún un misterio.

7.5. Dios y la lógica*

En la historia de la filosofía se registra una cantidad considerable de intentos de demostraciones de la existencia de Dios. Si bien hoy son consideradas por los lógicos como meras falacias, algunas de ellas aún son enseñadas en las escuelas religiosas. Una de estas "demostraciones" tiene aún cierto interés desde el punto de vista de la lógica, pese a haber sido quizá el argumento más veces refutado en la historia. Es el llamado *Argumento Ontológico* de San Anselmo de Canterbury (1033-1109). La primera refutación fue realizada por un monje llamado Gaunilo quien fue contemporaneo con San Anselmo (aunque no canonizado).

La demostración es presentada de varias formas. En los siguientes dos párrafos, hay dos demostraciones diferentes;

> Y entonces, O Señor, dado que has dado entendimiento a la fe, dame el entendimento - siempre que tu sepas que es bueno para mi - de que tu existes, tal como nosotros creemos, y que tu eres lo que nosotros creemos que eres. Creemos que eres un ser del cual nada mayor puede ser pensado. O podría ser que no haya tal ser, dado que "el insensato ha dicho en su corazón 'No hay Dios' " (Psalmo 14.1; 53:1). Pero cuando el mismo insensato oye lo que digo - "Un ser del cual nada mayor puede ser pensado" el entiende lo que escucha, y lo que entiende está en su entendimiento aún si no entiende que esto exista. Porque una cosa es que un objeto esté en el entendimiento y otra cosa entender que éste exista. Cuando un pintor considera de antemano lo que va a pintar ya lo tiene en su entendimiento, pero el no supone que lo que aún no ha pintado ya existe. Pero una vez que lo ha pintado ambas cosas ocurren: está en su entendimiento y también entiende que lo que ha producido existe. Aún el insensato, entonces, debe estar convencido de que un ser del cual nada mayor puede ser pensado existe al menos en su entendimiento, dado que cuando oye esto lo entiende y aquello que se entiende está en el entendimiento. Sin embargo es claro que aquello de lo cual nada mayor puede ser pensado no puede existir sólo en el entendimiento. Porque si realmente está sólo en el entendimiento, puede ser pensado que existe en realidad y esto sería aún mayor. Por lo tanto, si aquello de lo cual nada mayor puede ser pensado está sólo en el entendimiento, esa misma cosa de la cual nada mayor puede ser pensado es una cosa de la cual algo mayor puede ser pensado. Pero obviamente esto es imposible. Sin ninguna duda, por lo tanto, existe tanto en el entendimiento como en la realidad, algo de lo cual nada mayor puede ser pensado.
>
> Dios no puede ser pensado como inexistente. Y ciertamente el existe de manera tan verdadera que no puede ser pensado como inexistente. Dado que puede ser pensado algo que existe lo cual no puede ser pensado como inexistente, y esto es mayor que aquello que puede ser pensado como no existente. Entonces si aquello de lo cual nada mayor puede ser pensado, puede ser pensado como inexistente, esa misma cosa de la cual nada mayor puede ser pensado *no es* algo de lo cual nada mayor pueda ser pensado. Pero esto es contradictorio. Luego, hay verdaderamente un ser del cual

nada mayor puede ser pensado - luego ciertamente no puede ni siquiera pensarse que no exista.

Podemos analizar primero el argumento del segundo párrafo el cual es más sencillo. Las premisas consideradas y conclusiones obtenidas son las siguientes:

Premisa 1: Un ser que no puede pensarse como inexistente es mayor que un ser que puede pensarse como inexistente.

Por lo tanto si Dios puede ser pensado como inexistente, entonces puede pensarse un ser mayor el cual no puede ser pensado como inexistente.

Premisa 2: Dios es un ser del cual nada mayor puede ser pensado.

Conclusión: No puede pensarse que dios no exista.

La conclusión sacada de la primer premisa necesita una suposición adicional de que es efectivamente posible pensar un ser el cual no puede ser pensado como inexistente.

El argumento enunciado en el primer párrafo puede ser parafraseado como sigue:

Premisa 1: Podemos concebir un ser del cual nada mayor puede ser concebido.

Premisa 2: Aquello que es concebido esta en el entendimiento de quien lo concibe.

Premisa 3: Aquello que existe en el entendimiento de alguien y también en la realidad es mayor que algo que sólo existe en el entendimiento.

Por lo tanto Un ser concebido tal que ninguno mayor puede ser pensado debe existir tanto en la realidad como en el entendimiento.

Premisa 4: Dios es un ser del cual nada mayor puede ser pensado.

Conclusión: Dios existe en la realidad.

Quinientos años después, una versión del mismo argumento fue propuesta por Descartes. Puede resumirse la esencia de esta nueva versión y de algunas de las anteriores en lo siguiente. Descartes define a dios como un ser que tiene todas las propiedades (al menos las propiedades buenas). Luego, tiene entre ellas la propiedad de la existencia. Luego, dios existe.

Las objeciones a estos argumentos son variadas. Basicamente la de Gaunilo y posteriormente de Kant se basan en la idea de que la existencia no es una propiedad. Otra objeción que puede hacerse al argumento de San Anselmo (que Descartes se esmera en rechazar en la Meditaciones Metafísicas) es que en ningún lado se prueba la unicidad de Dios, y puede claramente haber más de un ser con esas propiedades. Siguiendo esta línea de argumentación puede también probarse la existencia de otras cosas, por ejemplo de una isla perfecta (muchos pueden imaginarse una isla de la cual ninguna mejor pueda ser pensada).

Vamos a presentar ahora una demostración de que existe un unicornio la cual se basa en la misma idea y ver una de las objeciones más contundentes al argumento ontológico hecha por Raymond Smullyan en [Smu78].

En lugar de demostrar que existe un unicornio, vamos a demostrar la proposición posiblemente más fuerte de que existe un unicornio existente (la cual obviamente implica que existe un unicornio). Por la ley del tercero excluído, tenemos solamente dos posibilidades:

1. Un unicornio existente existe.

2. Un unicornio existente no existe.

La segunda posibilidad es obviamente contradictoria, dado que ningun ser existente puede no existir. De la misma manera que un unicornio azul tiene que ser necesariamente azul (aunque se haya perdido), un unicornio existente necesariamente tiene que existir. Luego, la primera proposición es verdadera.

La objeción de Kant es aplicable a esta demostración, pero también puede verse una confusión más elemental en el uso de la palabra 'un'. En algunos contextos 'un' significa 'todos' y en otros significa 'al menos uno'. Por ejemplo, en 'un gato es un felino' estamos diciendo que cualquier gato es un felino, mientras que en 'un gato se comió el pescado' nos estamos refiriendo a un gato en particular, el cual existe y además se comió nuestra cena. En la demostración de la existencia de un unicornio, estamos confundiendo ambos usos de la palabra 'un'. En la demostración usamos 'un' en el primer sentido. En este caso la conclusión es verdadera, dado que obviamente todo unicornio existente existe, pero la confusión viene de leer la conlusión usando la palabra 'un' en el segundo sentido. Si interpretáramos así a la palabra 'un' la demostración no sería correcta, dado que en ese caso la primera proposición sería falsa (como por ejemplo decir 'un unicornio azul se me perdió) y la segunda verdadera (no existe ningun unicornio, existente o no). Esta misma objeción puede aplicarse al argumento ontológico de San Anselmo y Descartes. Lo único que están demostrando es que todo dios que satisficiera las propiedades de la definición de Descartes obviamente existiría.

Para finalizar, presentamos una "objeción" de Borges en un cuento llamado sugerentemente *Argumentum Ornithologicum.*

> Cierro los ojos y veo una bandada de pájaros. La visión dura un segundo o acaso menos; no sé cuantos pájaros vi. ¿Era definido o indefinido su número? El problema involucra es de la existencia de Dios. Si Dios existe, el número es definido, porque Dios sabe cuantos pájaros vi. Si Dios no existe, el número es indefinido, porque nadie pudo llevar la cuenta. En tal caso, vi menos de diez pájaros (digamos) y más de uno, pero no vi nueve, ocho, siete, seis, cinco, cuatro, tres o dos pájaros, Vi un número entre diez y uno que no es nueve, ocho, siete, seis, cinco, etcétera. Ese número entero es inconcebible; *ergo,* Dios existe.

7.6. Ejercicios

Ejercicio 7.1

Demostrar la siguiente versión generalizada del metateorema del testigo.

Si $k \notin (FV.P \cup FV.Q \cup FV.R)$ entonces $\langle \exists i \ : R : \ P \rangle \Rightarrow Q$ si y solo si $P(i := k) \Rightarrow Q$ es un teorema.

Capítulo 8

El formalismo básico

> 41. Nada se edifica sobre la piedra, todo sobre la arena, pero nuestro deber es edificar como si fuera piedra la arena.
>
> Jorge Luis Borges:
> *Fragmentos de un evangelio apócrifo*

En esta sección definimos una notación simple y abstracta que nos permitirá escribir y manipular programas. Esta notación, que llamaremos el **formalismo básico**, se basa en la *programación funcional* (ver por ej. [Bir98, Tho96]), y puede pensarse como que representa la esencia de ésta. Al mismo tiempo, esta notación es mucho más simple que la mayoría de los lenguajes de programación y es suficientemente rica para expresar de manera sencilla programas funcionales. Por otro lado, las reglas que nos permiten manipularlas son explícitas y mantienen el estilo del cálculo de predicados y de las expresiones cuantificadas, lo cual permite integrarlas armónicamente en el proceso de construcción de programas.

Varias de estas ideas están inspiradas en la tesis de doctorado [Hoo89].

8.1. Funciones

Las funciones van a ser uno de los conceptos esenciales en los cuales se basa el formalismo básico. Como ya indicamos en el capítulo 2, la notación usual para la aplicación de funciones en matemática, $f(x)$ no será adecuada, por lo cual usaremos un operador explícito para la aplicación de funciones, el cual escribiremos como un punto.

En matemática es usual hablar de "la función $f.x$ tal que ...", para referirse a la función f (en cuya definición se usa probablemente la variable x). En matemática esto no suele traer problemas dado que las funciones son raramente usadas como valores ellas mismas, pero en nuestro caso debemos distinguir claramente entre la función f y la aplicación de ésta al valor x escrito como $f.x$.

La regla básica para la aplicación de funciones es la regla de "sustitución de iguales por iguales", que llamaremos **regla de Leibniz**, la cual ya fue presentada en el capítulo 2. Usando expresiones cuantificadas podemos expersar esta regla como sigue.

$$\langle \forall f, x, y \; :: \; x = y \Rightarrow f.x = f.y \rangle$$

El operador de aplicación de funciones tiene la máxima precedencia, es decir que para cualquier otro operador $\oplus$ vale

$$f.x \oplus y = (f.x) \oplus y$$

El valor de $f.x$ puede ser a su vez otra función, la cual puede ser aplicada a otro valor, por ejemplo, $(f.x).y$. Supondremos que el operador de aplicación asocia a izquierda, esto es

$$f.x.y = (f.x).y$$

y para todos los fines puede pensarse a una tal f como una función de dos argumentos.
Veamos ahora un ejemplo de aplicación de la regla de Leibniz.

Ejemplo 8.1
Supongamos que existe una función $f : Nat \mapsto Nat$ con la siguiente propiedad:

$$\forall\ p\ primo\ \wedge \forall\ x, y \in Nat,$$

$$\begin{aligned} f.p &= 1 \\ f.(x * y) &= f.x + f.y \end{aligned}$$

Se pide demostrar que $\forall\ p$ primo, $\sqrt{p}$ no es racional.

Una tal f es la función que cuenta la cantidad de factores primos de un número. Para la demostración que haremos cualquier otra función que satisiciera esta propiedad también serviría. Razonemos por el absurdo, es decir, supongamos que $\sqrt{p}$ es racional, con p primo y veamos que se deduce una contradicción (*False*). Omitimos los rangos, pero se asume que $x, y \in \mathbb{N}$, $y \neq 0$.

$\sqrt{p}$ racional

$\equiv$ { definición de racional }

$\left\langle \exists\, x, y\ ::\ \sqrt{p} = \frac{x}{y} \right\rangle$

$\equiv$ { álgebra }

$\langle \exists\, x, y\ ::\ p * y^2 = x^2 \rangle$

$\Rightarrow$ { regla de Leibniz }

$\langle \exists\, x, y\ ::\ f.(p * y^2) = f.x^2 \rangle$

$\equiv$ { propiedad de f }

$\langle \exists\, x, y\ ::\ f.p + f.y + f.y = f.x + f.x \rangle$

$\equiv$ { álgebra, $f.p = 1$ }

$\langle \exists\, x, y\ ::\ 1 + 2 * f.y = 2 * f.x \rangle$

$\equiv$ { números pares e impares son diferentes }

False

Por lo tanto, $\sqrt{p}$ es irracional.

8.2. Definiciones y expresiones

Uno de los elementos que constituirán nuestro formalismo básico es el conjunto de **expresiones**. Al igual que en la matemática las expresiones son usadas sólo para denotar valores. Estos valores pertenecerán a conjuntos que resulten útiles para expresar programas, por ejemplo, números, valores lógicos, caracteres, tuplas, funciones y listas. Las expresiones válidas de cada uno de dichos conjuntos se describirán al final de este capítulo.

Es importante distinguir claramente entre las expresiones y el valor que éstas denotan. Por ejemplo, la expresión $6 * 7$ denota el número abstracto cuarenta y dos, al igual que la expresión 42. En este sentido la igualdad $6 * 7 = 42$ significa que ambas expresiones denotan el mismo valor, pero no hay que confundir el número denotado por la expresión decimal 42 con la expresión misma. En matemática es usual identificar la expresión con el valor cuando esto no da lugar a confusión. En el contexto de la programación es necesario ser un poco más cuidadosos, dado que también nos interesará hablar de expresiones y hacer transformaciones sintácticas con éstas.

En el ejemplo anterior, la expresión 42 suele preferirse a la expresión $6*7$, debido a que es una expresión más "directa" del valor que ambas denotan (el número cuarenta y dos). En nuestro formalismo la única forma de referirse a un valor será a través de expresiones. Cuando sea posible (como en el caso de los números) elegiremos un conjunto de expresiones representantes, las cuales denominaremos **expresiones canónicas**. El proceso de cómputo consistirá, en general, en reducir una expresión dada a su forma canónica cuando ésta exista (ver el capítulo 9).
Algunos ejemplos de expresiones en dominios matemáticos conocidos son:

booleanas:	$False$, $True$, $\neg(3 = 2)$
numéricas:	42, 3*5, 3.14159
de caracteres:	'8', 'z', ','

El otro elemento que constiuirá nuestro formalismo básico será el conjunto de **definiciones**, las cuales nos permitirán introducir nuevos valores al formalismo y definir operaciones para manejarlos. En general, aunque no necesariamente, se usarán las definiciones para introducir valores del tipo función. Una definición es una asociación de una expresión a un nombre. La forma de una definición será

$$f.x \doteq E$$

donde las variables en x pueden ocurrir en E. Si la secuencia de variables x es vacía (es decir si la definición es de la forma $f \doteq E$) se dirá que f es una **constante**. Si f ocurre en E se dice que la definición es **recursiva** .

Es esencial notar aquí que usamos un signo ligeramente diferente al signo igual para las definiciones. Esto nos va a permitir distinguir una definición (las cuales van a ser nuestros programas) de una propiedad. Las reglas de plegado/desplegado de la sección 8.3 muestran que el signo de definición y el signo igual están intímamente relacionados.

Dado un conjunto de definiciones, es posible usar los nombres definidos para formar expresiones. Por ejemplo:

Son definiciones:	$pi \doteq 3.1416$
	$cuadrado.x \doteq x * x$
	$area.r \doteq pi * cuadrado.r$
Son expresiones:	pi
	$pi * cuadrado.(3 * 5)$
	$area.15$

Un **programa funcional** será un conjunto de definiciones. La ejecución de un programa funcional consistirá en la evaluación de una expresión y reducción a su forma canónica, donde los nombres definidos en el programa pueden ocurrir en ésta (ver capítulo 9).

8.3. Reglas para el cálculo con definiciones

El formalismo básico nos permitirá no sólo escribir programas sino también razonar con ellos (también nos permitirá escribir cosas que no son, estrictamente hablando, programas). Para poder razonar dentro del formalismo básico contaremos con todas las reglas asociadas a cada dominio matemático que usemos (números, funciones, listas, etc.) y algunas reglas específicas para el manejo de definiciones.

La reglas básicas para manejar expresiones en un contexto de definiciones dado son las reglas de **plegado** y **desplegado** (en inglés, folding y unfolding). Estas dos reglas son una la inversa de la otra.

Si se tiene la definición $f.x \doteq E$, entonces para cualquier expresión A,

$$f.A = E(x := A)$$

donde la expresión $E(x := A)$ denota a la expresión E en la que toda aparición del nombre x es reemplazado sintácticamente por la expresión A. Si x es una secuencia de variables, entonces A también debe serlo y debe tener la misma longitud. En tal caso, la sustitución se realiza de manera 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 sólo "definición de f" entendiéndose si es plegado o desplegado a partir del contexto.

Es importante no confundir una función con su definición. Una función es un valor abstracto, mientras que una definición es una entidad sintáctica. Es así que las definiciones $f.x \doteq 2 * x$ y $g.x \doteq x + x$ definen la misma función, por lo cual es verdadero que $f = g$. La regla que nos permite mostrar esto es la de **extensionalidad** (la cual no debe ser confundida con la regla de Leibniz).

$$\langle \forall f, g \ :: \ \langle \forall x \ :: \ f.x = g.x \rangle \Rightarrow f = g \rangle$$

Como ejemplo consideremos la composición de funciones (para la cual usaremos el símbolo $\circ$), cuya definición es:

$$(\circ) : (B \mapsto C) \mapsto (A \mapsto B) \mapsto (A \mapsto C)$$

$$(f \circ g).x \doteq f.(g.x)$$

Nótese que estamos usando *notación infija* para el operador de composición. vamos a usar la convención de que dado un operador infijo la fuunción de dos variables asociada se escribirá con el operador entre paréntesis. En este sentido vale la igualdad $(\circ).f.g = f \circ g$ y por lo tanto la definición del operador de composición podría escribirse en la notación introducida incialmente como sigue:

$$(\circ) : (B \mapsto C) \mapsto (A \mapsto B) \mapsto (A \mapsto C)$$

$$(\circ).f.g.x \doteq f.(g.x)$$

En esta defínción es más claro que tanto f y g como x son sólo parámetros en la definición de $\circ$.

Demostraremos que el operador de composición de funciones es asociativo. Para ello, por extensionalidad, alcanza con demostrarlo para la aplicación a un valor arbitrario:

$$
\begin{aligned}
& ((f \circ g) \circ h).x \\
= & \ \{ \text{ definición de } \circ \ \} \\
& (f \circ g).(h.x) \\
= & \ \{ \text{ definición de } \circ \ \} \\
& f.(g.(h.x)) \\
= & \ \{ \text{ definición de } \circ \ \} \\
& f.((g \circ h).x) \\
= & \ \{ \text{ definición de } \circ \ \} \\
& (f \circ (g \circ h)).x
\end{aligned}
$$

8.4. Definiciones locales

Es posible también incluir en una definición una o varias **definiciones locales**. Estas definiciones sólo tienen sentido en la expresión a la cual están asociadas, por ejemplo:

$$
\begin{array}{lcl}
raiz1.a.b.c & \doteq & (-b - sqrt.disc)/(2 * a) \\
 & & \|[disc \doteq b^2 - 4 * a * c]\|
\end{array}
$$

En la definición de $raiz1$, que calcula la menor raíz de la ecuación $ax^2 + bx + c = 0$, se incluye la variable $disc$, la cual está localmente definida, es decir, $disc$ sólo tiene sentido dentro de la definición de $raiz1$. En particular, en este caso la definición de $disc$ hace referenecia a nombres (b, a, c) que sólo tienen sentido dentro de la definición de $raiz1$.

Las definiciones locales pueden servir para mejorar la legibilidad de un programa pero también para mejorar la eficiencia, como veremos mas adelante.

8.5. Análisis por casos

Un mecanismo muy útil para construir expresiones y por lo tanto para definir funciones es el *análisis por casos*. Cuando se usa esta construcción, el valor de la expresión puede depender de los valores de verdad de ciertas expresiones booleanas que la componen. En general, una expresión E puede tomar la siguiente forma

$$
\begin{array}{rcl}
E & = & (\ \ B_0 \to E_0 \\
 & & \vdots \\
 & & \Box\ B_n \to E_1 \\
 & &)
\end{array}
$$

donde las B_i son expresiones boolenas y las E_i son expresiones del mismo tipo que E.

Se entiende que el valor de E va a ser el valor de alguna E_i tal que B_i es cierta.

Esta construcción es indispensable en la definición de algunas funciones, por ejemplo:

$$
\begin{array}{rcllll}
max.a.b & \doteq & (& a \leq b & \to & b \\
 & & \Box & b \geq a & \to & a \\
 & &) & & &
\end{array}
$$

Las condiciones booleanas reciben el nombre de **guardas** o protecciones (preferimos conservar la palabra "guarda" usada como un neologismo, del inglés *guard*). Esta expresión tomará valores de acuerdo al valor de a y b. Si bien las guardas no son disjuntas, esto no significa que la función pueda comportarse de manera diferente en dos evaluaciones, sino que no es relevante para demostrar que el programa satisface alguna especificación dada, cual de las dos guardas se elige en el caso que ambas sean verdaderas. Esto puede dar cierta libertad a la hora de implementar un programa escrito en nuestro formalismo en un lenguaje de programación.

Ejemplo 8.2
La función factorial puede definirse por casos como sigue

$$
\begin{array}{rcllll}
fac.n & \doteq & (& n = 0 & \to & 1 \\
 & & \Box & n \neq 0 & \to & n * fac.(n-1) \\
 & &) & & &
\end{array}
$$

Existe una regla que permite manipular expresiones que incluyen análisis por casos. Sea por ejemplo una expresión con dos alternativas

$$
\begin{array}{rcl}
E & = & (\ \ B_0 \to E_0 \\
 & & \Box\ B_1 \to E_1 \\
 & &)
\end{array}
$$

y supongamos que queremos demostrar que E satisface la propiedad P, esto es $P.E$. Para ello es suficiente con demostrar la conjunción de las siguientes

- $B_0 \vee B_1$
- $B_0 \Rightarrow P.E_0$
- $B_1 \Rightarrow P.E_1$

El primer punto requiere que al menos una de la guardas sea verdadera, mientras que el segundo y el tercero nos piden que, suponiendo la guarda verdadera, podamos probar el caso correspondiente. Esto nos da un método de demostración (y por lo tanto de derivación de programas) que será explotado más adelante.

Ejemplo 8.3
Para definir la regla de rango unitario para el cuantificador numérico N es necesario introducir una función n definida por casos:

$$
\begin{array}{l}
n : Bool \mapsto Int \\
\hline
n.b \doteq \begin{array}{lll} (& b & \rightarrow 1 \\ \Box & \neg b & \rightarrow 0 \\ ) & & \end{array}
\end{array}
$$

Ahora puede definirse la regla de rango unitario para el cuantificador N como sigue:

$$\langle N\, i\ : i = C :\ T.i \rangle = n.(T.C)$$

Ejercicio 8.1
Demostrar la regla usando la definición de N y análisis por casos.

$$\langle \oplus\, i\ : i = C \wedge R.i :\ T.i \rangle \equiv \begin{array}{lll} (& R.C & \rightarrow T.C \\ \Box & \neg R.C & \rightarrow e \\ ) & & \end{array}$$

donde e es el neutro de $\oplus$.

8.6. Pattern Matching

Vamos a introducir una abreviatura cómoda para escribir ciertos análisis por casos que ocurren frecuentemente. 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 ver si el argumento es igual a 0 o no. Una forma alternativa de escribir esa función es usar en los argumentos de la definición de la función un patrón o *pattern*, el cual permite distinguir si el número es 0 o no. Para ello se usa la idea de que un número natural es o bien 0 o bien de la forma $(n+1)$, con n cualquier natural (incuído el 0). Luego la función factorial podría escribirse como

$$
\begin{array}{rcl}
fac.0 & \doteq & 1 \\
fac.(n+1) & \doteq & (n+1) * fac.n
\end{array}
$$

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 (en este caso el número anterior).

Otros patterns posibles para los naturales podrían ser por ejemplo 0, 1 y $n+2$, lo cual estaría asociado a tres guardas, dos que consideran los casos en que el argumento es 0 o 1 y uno para el caso en que el número considerado es 2 o más. Así una defininición de la forma

$$
\begin{array}{rcl}
f.0 & \doteq & E_0 \\
f.1 & \doteq & E_1 \\
f.(n+2) & \doteq & E_2
\end{array}
$$

se traduce a análisis por casos como

$$\begin{array}{lcll} f.n & \doteq & (& n = 0 \rightarrow E_0 \\ & & \Box & n = 1 \rightarrow E_1 \\ & & \Box & n \geq 2 \rightarrow E_2(n := n-2) \\ & &) & \end{array}$$

Otro posible patrón para los naturales podría ser separar pares e impares, es decir que una función

$$\begin{array}{rcl} f.(2*n) & \doteq & E_0 \\ f.(2*n+1) & \doteq & E_1 \end{array}$$

se traduce a análisis por casos, asumiendo un predicado *par* de significado obvio, como

$$\begin{array}{lcll} f.n & \doteq & (& par.n \rightarrow E_0(n := \frac{n}{2}) \\ & & \Box & \neg par.n \rightarrow E_1\left(n := \frac{n-1}{2}\right) \\ & &) & \end{array}$$

8.7. Tipos

Toda expresión tiene un tipo asociado. Hay tres tipos básicos, que son: ***Num***, que incluye todos los números; ***Bool***, para los valores de verdad *True* y *False*; y ***Char***, para los caracteres. A toda expresión correcta se le puede asignar un tipo, ya sea un tipo básico o un tipo compuesto, obtenido a partir de los tipos básicos. Si a una expresión no se le puede asignar un tipo, la misma será considerada incorrecta.

Ejemplos:	5	es de tipo Num
	'h'	es de tipo $Char$
	cuadrado	es de tipo $Num \mapsto Num$

En ocasiones no se desea especificar un tipo en particular. En estos casos, el tipo se indicará con una letra mayúscula. Ejemplo: la función $id.x = x$ tiene sentido como $id : Num \mapsto Num$, pero también como $id : Char \mapsto Char$ o $id : Bool \mapsto Bool$. Entonces, si no nos interesa especificar un tipo en particular, escribiremos $id : A \mapsto A$. La variable A se denomina **variable de tipo**. Cuando el tipo de una expresión incluye variables, diremos que es un **tipo polimórfico**.

La función *max* que definimos en la sección anterior tiene tipo $Num \mapsto Num \mapsto Num$, que no es exactamente lo mismo que $Num \times Num \mapsto Num$. El tipo que hemos indicado es más general. En ausencia de paréntesis, debe entenderse como $Num \mapsto (Num \mapsto Num)$. Para (casi) todos los temas desarrollados en este libro, no se pierde nada con pensar que $Num \mapsto Num \mapsto Num$ es una notación estilizada para $Num \times Num \mapsto Num$.

8.8. Tipos básicos

Los tipos básicos de nuestra notación de programas serán descriptos por las expresiones canónicas y las operaciones posibles entre elementos de ese tipo.

Tipo *Num*: Las expresiones canónicas son las constantes (números). Las operaciones usadas para procesar los elementos de este tipo son: $+, -, *, /, ^\wedge$, con las propiedades usuales, y las funciones *div* y *mod*, que devuelven, respectivamente, el cociente y el resto de la división entera de dos números enteros.

Ejercicio 8.2
Determinar las propiedades que tienen las funciones *div* y *mod*.

Tipo *Bool*: Hay dos expresiones canónicas para denotar los valores de verdad, que son *True* y *False*. Una función que retorna un valor de verdad se denomina predicado. Los booleanos son importantes porque son el resultado de los operadores de comparación: $=, \neq, >, <, \geq, \leq$. Estos operadores se aplican no sólo a números, sino también a otros tipos básicos ($Char$, $String$, etc.), siempre y cuando los dos argumentos a comparar tengan el mismo tipo. Es decir, cada operador de comparación es una función polimórfica de tipo $A \mapsto A \mapsto Bool$. Los booleanos se pueden combinar usando los operadores lógicos $\wedge, \vee, \neg$, etc. con las propiedades usuales.

Tipo *Char*: Constituido 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 ***String*** y se denota encerrada entre comillas dobles, por ejemplo: "hola".

8.9. Tuplas

Una manera de formar un nuevo tipo combinando los tipos básicos es tomar éstos de a pares. Por ejemplo: el tipo $(Num, Char)$ consiste de todos los pares en los que la primera componente es de tipo Num y la segunda de tipo $Char$. De la misma manera, se pueden construir ternas, cuaternas, etc.. En general, hablaremos de **tuplas**. Los elementos canónicos de una tupla son de la forma $(\ , \ , \ldots, \ )$, donde cada una de las coordenadas es un elemento canónico del tipo correspondiente. Ejemplo: podemos definir la función $raices$, que devuelve las raíces de un polinomio de segundo grado, de manera tal que el resultado sea un par, es decir, $raices : Num \mapsto Num \mapsto Num \mapsto (Num, Num)$.

La noción de pattern matching puede extenderse al caso de tuplas de manera natural. Por ejemplo, para definir una función f sobre ternas puede darse una definición de la siguiente forma:

$$f.(x, y, z) = \ldots$$

es más, hasta el momento es la única manera de definir una función sobre tuplas. Otra manera quizá menos elegante es tomar algún elemento de la tupla usando la función de indexación que se define a continuación.

Dada una n-upla, se supone definida la función

$$.i : (A_0 \times \ldots \times A_n) \mapsto A_i$$

la cual satisface la propiedad

$$(a_0, \ldots, a_n).i = a_i$$

esta última ecuación puede ser considerada su definición (usando pattern matching).

Ejemplo 8.4
Los números racionales pueden ser representados por pares de numeros enteros. La función que suma dos racionales puede ser definida como:

$$SumNat : (Num, Num) \mapsto (Num, Num)$$

$$SumRat.(a, b).(c, d) = (a * d + b * c, b * d)$$

8.10. Listas

Una **lista** (o secuencia) es una colección de valores ordenados, los cuales deben ser todos del mismo tipo. Las listas pueden ser finitas o infinitas (en este curso sólo consideraremos listas finitas). Cuando son finitas, se las denota entre corchetes, con sus elementos separados por comas.

Ejemplos: [0,1,2,3]

[(“*Juan*”, *True*), (“*Jorge*”, *False*)]

[‘*a*’] (lista con un elemento)

[ ] (lista vacía).

El tipo de una lista es $[A]$, donde A es el tipo de sus elementos.

Ejemplos: [0,1,2,3] es de tipo $[Num]$

[(“*Juan*”, *True*), (“*Jorge*”, *False*)] es de tipo $[(String, Bool)]$

[[1,2],[3,4]] es de tipo $[[Num]]$ (lista de listas de números).

La lista vacía podría considerarse de cualquier tipo, por eso se le asigna el tipo polimórfico $[A]$, a menos que quede claro que tiene un tipo en particular. Ejemplo: en $[[1, 2], [\,]]$ el tipo de $[\,]$ es $[Num]$, pues es un elemento de una lista cuyos elementos son de tipo $[Num]$.

A diferencia de los conjuntos, una lista puede contener elementos repetidos. Ejemplo: [1,1] es una lista de dos elementos. Dos listas son iguales sólo si tienen los mismos elementos en el mismo orden. Ejemplos: $[0, 1, 2, 3] \neq [3, 2, 1, 0]$, $[1, 1] \neq [1]$.

Notación: convencionalmente, se usa el símbolo x para indicar un elemento de una lista, xs para una lista, xss para una lista de listas, etcétera. Si bien esto no tiene sentido semántico, es decir, no es *necesario* usar esta nomenclatura, adoptaremos la convención.

8.10.1. 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 $[A]$, entonces $x \triangleright xs$ es una nueva lista de tipo $[A]$, cuyo primer elemento es x y el resto es xs.

   Ejemplo: $[0, 1, 2, 3] = 0 \triangleright [1, 2, 3] = 0 \triangleright (1 \triangleright (2 \triangleright (3 \triangleright [\,])))$.
3. $\triangleleft$ agrega un elemento a una lista por la derecha.

   Ejemplo: $[0, 1, 2, 3] = [0, 1, 2] \triangleleft 3 = ((([\,] \triangleleft 0) \triangleleft 1) \triangleleft 2) \triangleleft 3$

En general, los programas funcionales usan sólo los dos primeros constructores. Los tipos de estos constructores son:

$$\begin{aligned} [\,] &: [A] \\ \triangleright &: A \mapsto [A] \mapsto [A] \end{aligned}$$

Las operaciones sobre listas pueden definirse por pattern matching en estos dos constructores, o equivalentemente dando su definición para el caso vacío y no vacío (considerado como agregar un elemento por la izquierda).

Proposicion 8.1
Sean x, y de tipo A y xs, ys de tipo $[A]$. Entonces

$$(x \triangleright xs) = (y \triangleright ys) \iff x = y \wedge xs = ys.$$

8.10.2. Operaciones sobre listas

Hay cinco operaciones fundamentales sobre listas, que son las siguientes:

concatenar : $+\!\!+ : [A] \mapsto [A] \mapsto [A]$

El operador $+\!\!+$ toma dos listas del mismo tipo y devuelve una lista del mismo tipo, que consiste en las dos anteriores, puestas una inmediatamente después de la otra. Si xs e ys son listas del mismo tipo, la concatenación de ambas se denota $xs +\!\!+ ys$.

Ejemplos: $[0,1] +\!\!+ [2,3] = [0,1,2,3]$
$[0,1,2] +\!\!+ [\,] +\!\!+ [3] = [0,1,2,3]$
$[0,1] +\!\!+ [1,2] = [0,1,1,2]$

longitud : $\# : [A] \mapsto A$

El operador $\#$ devuelve la longitud de una lista, es decir, la cantidad de elementos que la misma contiene. Si xs es una lista, su longitud se denota $\#xs$.

Ejemplos: $\#[0,1,2,3] = 4$
$\#[\,] = 0$

tomar n : $\uparrow : [A] \mapsto Num \mapsto [A]$

El operador $\uparrow$ toma una lista y un número natural n, y devuelve la lista de los primeros n elementos de la lista original. Cuando n es mayor que la longitud de la lista, $\uparrow$ devuelve la lista completa. Si xs es una lista, la lista de sus primeros n elementos se denota $xs{\uparrow}n$.

Ejemplos: $[0,1,2,3]{\uparrow}2 = [0,1]$
$[0,1,2,3]{\uparrow}5 = [0,1,2,3]$
$[\,]{\uparrow}n = [\,]$

tirar n : $\downarrow : [A] \mapsto Num \mapsto [A]$

El operador $\downarrow$ toma una lista y un número natural n, y devuelve la lista que resulta al eliminar de la lista original los primeros n elementos. Si xs es una lista, la lista sin sus primeros n elementos se denota $xs{\downarrow}n$.

Ejemplos: $[0,1,2,3]{\downarrow}2 = [2,3]$
$[0,1,2,3]{\downarrow}5 = [\,]$
$[\,]{\downarrow}n = [\,]$

indexar : $\cdot : [A] \mapsto Num \mapsto A$

El operador $\cdot$ toma una lista y un número natural, y devuelve el elemento de la lista que se encuentra en la posición indicada por el número natural. Si xs es una lista, el elemento que ocupa el lugar i se denota $xs.i$. Tener en cuenta que el primer elemento de la lista ocupa la posición 0 y que $xs.i$ no está definido cuando $i > \#xs - 1$.

Ejemplos: $[0, 1, 2, 3].2 = 2$
$[4, 5, 6].0 = 4$
$[4, 5, 6].3$ indefinido

8.10.3. Propiedades de las operaciones

A continuación enumeramos algunas de las propiedades más importantes que poseen los operadores definidos en la sección anterior. Algunas propiedades usan el símbolo de definición ($\doteq$) en vez de la igualdad. Por la regla de fold/unfold, estas definiciones son también igualdades. Nótese que todas las operaciones pueden definirse por pattern matching sobre los constructores de listas.

concatenar

$$\begin{array}{rcl}
[\,] +\!\!+\; ys & \doteq & ys \\
(x \triangleright xs) +\!\!+\; ys & \doteq & x \triangleright (xs +\!\!+\; ys) \\
(xs +\!\!+\; ys) +\!\!+\; zs & = & xs +\!\!+\; (ys +\!\!+\; zs) \\
(xs +\!\!+\; ys).i & = & (\ i < \#xs \rightarrow xs.i \\
 & & \Box\ i \geq \#xs \rightarrow ys.(i - \#xs) \\
 & &) \\
(xs +\!\!+\; ys) = [\,] & \equiv & xs = [\,] \wedge ys = [\,]
\end{array}$$

longitud

$$\begin{array}{rcl}
\#[\,] & \doteq & 0 \\
\#(xs +\!\!+\; ys) & \doteq & \#xs + \#ys \\
\#(xs{\uparrow}n) & = & n \min \#xs \\
\#(xs{\downarrow}n) & = & (\#xs - n) \max\ 0
\end{array}$$

tomar n

$$\begin{array}{rcl}
xs{\uparrow}0 & \doteq & [\,] \\
[\,]{\uparrow}n & \doteq & [\,] \\
(x \triangleright xs){\uparrow}(n+1) & \doteq & x \triangleright (xs{\uparrow}n)
\end{array}$$

tirar n

$$\begin{array}{rcl}
xs{\downarrow}0 & \doteq & xs \\
[\,]{\downarrow}n & \doteq & [\,] \\
(x \triangleright xs){\downarrow}(n+1) & \doteq & xs{\downarrow}n
\end{array}$$

indexar

$$\begin{array}{rcl}
(x \triangleright xs).0 & \doteq & x \\
(x \triangleright xs).(n+1) & \doteq & xs.n
\end{array}$$

8.11. Ejercicios

Ejercicio 8.3
Definir la función $sgn : Int \rightarrow Int$ que dado un número devuelve 1, 0, o -1 en caso que el número sea positivo, cero o negativo respectivamente.

Ejercicio 8.4
Definir una función $abs : Int \rightarrow Int$ que calcule el valor absoluto de un número

Ejercicio 8.5
Definir el predicado $bisiesto : Nat \rightarrow Bool$ 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 bisiseto pero 2000 si lo es.

Ejercicio 8.6
Definir los predicados *equi* y *isoc* que toman tres números los cuales representan las longitudes de los lados de un triángulo y determinan respectivamente si dicho triángulo es equilátero o isóceles.

Ejercicio 8.7
Definir la función $edad : (Nat, Nat, Nat) \rightarrow (Nat, Nat, Nat) \rightarrow Int$ que dadas dos fechas indica los años transcurridos entre ellas. Por ejemplo

$$edad.(20, 10, 1968).(30, 4, 1987) = 18$$

Ejercicio 8.8
En un prisma rectangular, llamemos h a la altura, b al ancho y d a la profundidad. Completar la siguiente definición del área del prisma:

$$\begin{array}{rl} area.h.b.d = & 2 * frente + 2 * lado + 2 * arriba \\ & [\![\qquad\qquad\qquad\qquad\qquad]\!] \end{array}$$

donde "frente", "lado" y "arriba" son las caras frontal, lateral y superior del prisma respectivamente.

Ejercicio 8.9
Definir la función *raices*, que devuelve las raíces de un polinomio de segundo grado.

Ejercicio 8.10
Suponer el siguiente juego: m jugadores en ronda comienzan a decir los 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 $0 \leq n \leq 9999$.

Ayuda: definir una función *unidad*, que devuelva la cifra de las unidades de un número. Idem con las decenas, etc.; para ello usar las funciones *div* y *mod*.

Ejercicio 8.11
Reescribir las siguiente definición usando análisis por casos:

$$\begin{array}{rcl} f.0 & \doteq & 1 \\ f.(n+1) & \doteq & f.n + 2 * n + 1 \end{array}$$

Ejercicio 8.12
Reescribir las siguiente definición usando análisis por casos:

$$\begin{array}{rcl} g.x.0 & \doteq & 1 \\ g.x.(2*n+1) & \doteq & x*g.x.(2*n) \\ g.x.(2*n+2) & \doteq & g.(x*x).(n+1) \end{array}$$

Ejercicio 8.13
Definir las siguientes funciones:

1. $hd : [A] \mapsto A$ devuelve el primer elemento de una lista. (hd es la abreviatura de *head*).
2. $tl : [A] \mapsto [A]$ devuelve toda la lista menos el primer elemento. (tl es la abreviatura de *tail*).
3. $last : [A] \mapsto A$ devuelve el último elemento de una lista.
4. $init : [A] \mapsto [A]$ devuelve toda la lista menos el último elemento.

Capítulo 9

Modelo Computacional

9.1. Introducción

En el capítulo anterior definimos una notación abstracta para razonar sobre programas llamada *formalismo básico*. Como se pudo ver, este lenguaje hace referencia a los mismos y sirve para escribirlos y manipularlos. Es así que se estableció una relación entre lenguaje y programas pero tratando de ocultar deliberadamente algunas de las características de estos últimos con la finalidad de simplificar la noción que tenemos de ellos y poder así razonar con entidades más simples de tipo matemático. Esto facilita el proceso de desarrollo de programas a partir de especificaciones (como se verá en capítulos siguientes), abstrayendo nociones que podrían perjudicar el correcto desarrollo de esta labor.

Entre estas nociones están las que posibilitan una interpretación de las expresiones como programas ejecutables. Este tipo de interpretación es denominada *interpretación operacional* o *modelo computacional* asociado al formalismo y consiste en definir formalmente los pasos elementales de ejecución de un programa hasta llegar al resultado final. Se describe, de esta forma, como los programas se ejecutan, en vez de analizar simplemente el resultado de esta ejecución.

Este tipo de interpretaciones se utiliza mayormente como guía para implementar el lenguaje en la computadora y para probar que esta implementación es correcta. Además, sirve para analizar la complejidad de funciones definidas en el formalismo básico (esto se verá en capítulos siguientes) y para obtener algunos resultados teóricos referentes al mismo (en la sección 9.7 pag. 90 se verá un ejemplo de estos resultados). Estas interpretaciones no son útiles para demostrar la corrección total de los programas ya que habría que analizar todos los pasos elementales de ejecución para cada uno de los elementos del dominio de la función.

De todas formas, como se vio en el capítulo anterior es posible entender y utilizar el formalismo sin tener que recurrir a este tipo de interpretaciones. Es decir que, para el desarrollo de programas, no hace falta conocer el modelo computacional asociado al formalismo.

Esto no significa que el formalismo no permita una interpretación operacional. Podría suceder que nuestro lenguaje sea tan abstracto que no permita ser implementado en una computadora. Al contrario, el formalismo desarrollado permite este tipo de interpretaciones y sobre este tópico tratará este capítulo. De todas formas, es posible y conveniente utilizar el formalismo libre de estas connotaciones al momento de desarrollar programas.

9.2. Valores

En el formalismo básico, como en matemática, una expresión es utilizada para representar (*denotar*) un valor. Una expresión puede denotar distintos tipos de valores, como ser números, listas, pares, funciones, etc.

Es importante distinguir entre un valor y su representación por medio de expresiones. Sea la función *cuadrado* : $Num \rightarrow Num$ definida como:

$$cuadrado.x \doteq x * x$$

Las expresiones $cuadrado.(3 * 5)$, $cuadrado.15$ y 225 denotan (todas) el número "doscientos veinticinco" pero ninguna (inclusive la expresión 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 el valor booleano "falso".

pares: Las expresiones

- $(cuadrado.3 * 5, 4)$
- $((225, 2 + 2), True).0$
- $(225, 4)$

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".

función: Dadas las funciones

$$\begin{array}{lllll} f.x.y & \doteq & (& x >= y & \rightarrow & x \\ & & \Box & x <= y & \rightarrow & y \\ & &) & & & , \\ g.x.y & \doteq & (& x > y & \rightarrow & x \\ & & \Box & x < y & \rightarrow & y \\ & & \Box & x = y & \rightarrow & x \\ & &) & & & , \end{array}$$

las expresiones

- f
- g
- $id \circ f$
- $id \circ g$

denotan todas la “función mínimo de dos números”.

Las computadoras no manipulan valores, solo pueden manejar representaciones concretas de los mismos. Así, por ejemplo, las computadoras utilizan la representación binaria de números enteros para denotar este tipo de valores. Con los otros tipos de valores (números, listas, pares, funciones, etc.) sucede lo mismo; las computadoras solo manejan sus representaciones.

9.3. Forma Canónica

Volvamos a las expresiones $cuadrado.(3 * 5)$, $cuadrado.15$ y 225. Cuando uno ejecuta alguno de estos programas desea conocer, como resultado de esta ejecución (o evaluación), el valor denotado por 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, ¿cual de todas?. La computadora podría elegir como resultado de la ejecución la expresión misma. Por ejemplo si ejecutamos el programa $cuadrado.(3 * 5)$ la computadora podría mostrar como resultados la misma expresión $cuadrado.(3 * 5)$ haciendo de la evaluación una operación trivial. Pero esto no nos ayuda a reconocer el valor de la expresión de manera inmediata.

Pidamos que la representación del valor resultado de la evaluación sea única. De esta forma, seleccionemos de cada conjunto de expresiones que denoten el mismo valor como máximo una que llamaremos *expresión canónica* de ese valor. Además, llamaremos a la expresión canónica que representa el valor de una expresión *la forma canónica* de esa expresión. Con esta restricción pueden quedar expresiones sin forma canónica.

Aparte del requerimiento mencionado al principio del párrafo anterior, no hay ninguna otra restricción objetiva para elegir la forma canónica de una expresión. Una forma de comenzar a resolver esta cuestión es definiendo el *conjunto de expresiones canónicas* como un subconjunto propio del de todas las expresiones posibles. Al construirlo se deberán tener en cuenta algunas cuestiones de índole mas bien práctica. En este sentido será conveniente que sus elementos sean expresiones simples ya que de esta forma es más fácil identificar el valor que representan; por ejemplo, para identificar el valor “doscientos veinticinco” elijamos la expresión 225 y no a $15*15$.

Otra cuestión importante es que, aunque se defina este conjunto de forma tal que existan expresiones sin forma canónica, el conjunto de expresiones que si la tienen sea lo suficientemente abarcativo como para que se pueda reconocer el valor abstracto de la mayor parte de las mismas. Este punto será analizado con más detenimiento después que definamos el conjunto.

Definamos entonces el conjunto de expresiones canónicas para las expresiones de distintos tipos:

booleanas: *True* y *False*.

números: -1, 0, 1, 2, 3,15 (número 3,15), etc., o sea la representación decimal del número.

pares: (E_0, E_1)
donde E_0 y E_1 son expresiones canónicas.

listas: $[E_0, \cdots, E_{n-1}]$, donde $n \geq 0$ y $E_0, \cdots, E_{n-1}$ son expresiones canónicas.
Notar que aquí también definimos a la lista vacía ([ ]) como una expresión canónica cuando $n = 0$.

Notemos que no hemos incluido en el ejemplo expresiones canónicas que representen funciones. Matemáticamente hablando, los valores de tipo función pueden ser vistas como reglas que asocian cada elemento de un conjuntos de valores con un único elemento de otro conjunto. Uno puede encontrar varias expresiones que representan esta relación, pero no vamos a elegir a una expresión como *la forma canónica* de estos valores.

Otros valores tampoco tienen expresión canónica. Por ejemplo, el número π no tiene una representación decimal finita. Uno puede escribir la expresión que denota este valor mediante la expansión decimal de π, dígito por dígito, pero en el formalismo básico todas las expresiones son finitas. Por lo tanto no se puede elegir *la forma canónica*.

Hasta ahora hemos visto valores que no tienen expresión canónica (funciones, número π). También puede suceder que una expresión no tenga forma canónica. Por ejemplo dada la definición de función:

$$inf \doteq inf + 1 \tag{9.1}$$

La expresión *inf* (de tipo número) no tiene forma canónica dado que puede demostrarse que su valor es diferente al de cualquier forma canónica.

Sea n la forma canónica que representa el mismo valor que la expresión *inf* (n es la forma canónica de *inf*). Esta relación implica que la expresión $n = inf$ es cierta. Además, como n es la representación decimal de un número (ya que es una forma canónica) se cumple $n \in \mathbb{R}$. Suponiendo esto (será nuestra hipótesis) podemos demostrar:

$$\begin{aligned} & n \\ = & \{ \text{ hipótesis } \} \\ & inf \\ = & \{ \text{ definición de } inf \} \\ & inf + 1 \\ = & \{ \text{ hipótesis } \} \\ & n + 1 \end{aligned}$$

Con esto concluimos que $n = n + 1$ lo cual es falso si $n \in \mathbb{R}$. En consecuencia, razonando por el absurdo, nuestra hipótesis era falsa, o sea *inf* no tiene forma canónica.

Lo mismo sucede con la expresión $\frac{1}{0}$ que cumple con la propiedad:

$$\frac{1}{0} * 0 = 1 \text{ (inverso multiplicativo de 0)}$$

Sea n la forma canónica que representa el mismo número que $\frac{1}{0}$. Esta relación implica que la expresión $n = \frac{1}{0}$ es cierta. Además, como n es una forma canónica, $n \in \mathbb{R}$. Suponiendo esto:

$$
\begin{array}{ll}
 & 0 \\
= & \{ \text{ elemento neutro de la multiplicación en } \mathbb{R} \text{ y } n \in \mathbb{R} \} \\
 & 0 * n \\
= & \{ \text{ conmutabilidad de la multiplicación en } R \text{ y } n \in \mathbb{R} \} \\
 & n * 0 \\
= & \{ \text{ hipótesis } \} \\
 & \frac{1}{0} * 0 \\
= & \{ \text{ inverso multiplicativo de } 0 \} \\
 & 1
\end{array}
$$

En consecuencia concluimos que $0 = 1$. Razonando por el absurdo n no puede ser la forma canónica de $\frac{1}{0}$ y por lo tanto esta expresión no tiene forma canónica.

9.4. Evaluación

A partir de todo lo dicho, la ejecución de un programa será el proceso de encontrar su forma canónica. Veamos entonces más detenidamente 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 este 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 esta hecho de forma tal que cuando ya no es posible reducir la expresión es que se ha llegado a la forma canónica (ver figura 9.1).

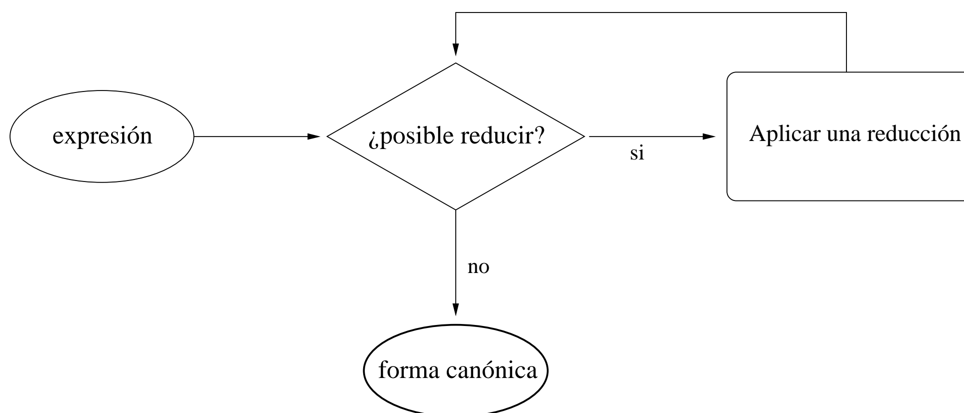

Figura 9.1: sistema de evaluación

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 reducción de la expresión $cuadrado.(3 * 5)$:

$$
\begin{array}{ll}
& cuadrado.(3*5) \\
= & \{ * \} \\
& cuadrado.15 \\
= & \{ \text{definición de } cuadrado \} \\
& 15*15 \\
= & \{ * \} \\
& 225
\end{array}
$$

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:

$$
\begin{array}{ll}
& cuadrado.(3*5) \\
= & \{ \text{definición de } cuadrado \} \\
& (3*5)*(3*5) \\
= & \{ * \text{ aplicada al primer término} \} \\
& 15*(3*5) \\
= & \{ * \text{ aplicada al segundo término} \} \\
& 15*15 \\
= & \{ * \} \\
& 225
\end{array}
$$

En el primer paso de reducción hemos elegido la subexpresión de mas afuera, decisión que no fue tomada en el proceso de evaluación anterior.

Puede suceder que el proceso de evaluación nunca termine. Por ejemplo tratemos de evaluar la expresión inf definida en la sección anterior:

$$
\begin{array}{ll}
& inf \\
= & \{ \text{definición de } inf \} \\
& inf+1 \\
= & \{ \text{definición de } inf \} \\
& (inf+1)+1 \\
= & \{ \text{asociatividad de la suma} \} \\
& inf+(1+1) \\
= & \{ + \} \\
& inf+2 \\
= & \{ \text{definición de } inf \} \\
& (inf+1)+2 \\
= & \{ \text{asociatividad de la suma} \} \\
& inf+(1+2)
\end{array}
$$

$= \{ + \}$
$\quad inf + 3$
$= \{$ definición de inf $\}$
$\quad (inf + 1) + 3$
$\quad \ldots$

Obviamente se puede seguir aplicando pasos de reducción indefinidamente. Esto no es necesariamente un problema ya que la expresión *inf* no tiene forma canónica.

Consideremos ahora la función definida por $K.x.y \doteq x$. ¿Qué sucede con la expresión $K.3.inf$? Claramente denota el valor "tres", luego se esperaría que reduzca a la forma canónica 3. Veamos que sucede si apliquemos pasos de reducción a esta expresión eligiendo siempre la subexpresión de más adentro:

$\quad K.3.inf$
$= \{$ definición de inf $\}$
$\quad K.3.(inf + 1)$
$= \{$ definición de inf $\}$
$\quad K.3.((inf + 1) + 1)$
$= \{$ asociatividad de la suma $\}$
$\quad K.3.(inf + (1 + 1))$
$= \{ + \}$
$\quad K.3.(inf + 2)$
$= \{$ definición de inf $\}$
$\quad K.3.((inf + 1) + 2)$
$\quad \ldots$

Obviamente este proceso no termina. Pero ahora elijamos las subexpresión de más afuera:

$\quad K.3.inf$
$= \{$ definición de K $\}$
$\quad 3$

Y con esta estrategia de reducción el proceso sí termina.

Afortunadamente existe un resultado teórico (demostrado para el cálculo lambda, un cálculo teórico de funciones en el cual el formalismo básico esta basado) que avala estas pruebas.

Teorema 9.1
Si la forma canónica existe, la estrategia de reducción que siempre elige la subexpresión de más afuera y más a la izquierda, conduce a la misma.

En otras palabras, si nuestro modelo computacional usa esta estrategia (denominada estrategia de *reducción normal*), va a llegar a encontrar la forma canónica, siempre que ella exista.

9.5. Un modelos computacional más eficiente

Según el teorema de la sección anterior nuestro modelo computacional deberá usar una estrategia de reducción normal si se desea encontrar la forma canónica, siempre que exista. Pero esta estrategia es, en algunos casos, menos eficiente. Veamos como se evalúa la expresión $cuadrado.(cuadrado.3)$ utilizando la misma.

$$
\begin{array}{ll}
 & cuadrado.(cuadrado.3) \\
= & \{ \text{ definición de cuadrado } \} \\
 & (cuadrado.3) * (cuadrado.3) \\
= & \{ \text{ definición de cuadrado } \} \\
 & (3 * 3) * (cuadrado.3) \\
= & \{ * \} \\
 & 9 * (cuadrado.3) \\
= & \{ \text{ definición de cuadrado } \} \\
 & 9 * (3 * 3) \\
= & \{ * \} \\
 & 9 * 9 \\
= & \{ * \} \\
 & 81
\end{array}
$$

En seis pasos de reducción logramos encontrar la forma normal. Pero ahora no utilicemos esta estrategia.

$$
\begin{array}{ll}
 & cuadrado.(cuadrado.3) \\
= & \{ \text{ definición de cuadrado } \} \\
 & cuadrado.(3 * 3) \\
= & \{ * \} \\
 & cuadrado.9 \\
= & \{ \text{ definición de cuadrado } \} \\
 & 9 * 9 \\
= & \{ * \} \\
 & 81
\end{array}
$$

y solo se realizaron cuatro pasos de reducción.

Analicemos más detenidamente lo que sucedió. Notemos que en la expresión inicial $cuadrado.(cuadrado.3)$ la función $cuadrado$ aparece solo dos veces. En los ejemplos se puede ver que en el primero se "copió" la subexpresión $cuadrado.3$ al ejecutar el primer paso de reducción y en el segundo esto no sucedió. Esto produjo tres desplegados de la función $cuadrado$ en el primer ejemplo, contra los dos esperados en el segundo. Además la subexpresión $3 * 3$ fue reducida una vez más en el primero.

A primera vista esta ineficiencia del modelo computacional puede no ser importante para la función que se esta analizando, pero si se trata de funciones más complejas se puede llegar a un desperdicio de recursos computacionales muy alto.

Este fenómeno de "copiado" de subexpresiones se debe a la forma en que fue definida la función cuadrado:

$$cuadrado.x \doteq x * x$$

donde el parámetro x fue repetido dos veces en la definición.

Una forma de solucionar este problema es cambiando nuestro modelo computacional para que cuando el sistema deba desplegar funciones que tengan este tipo de definición se utilicen definiciones locales. Así, por ejemplo, manteniendo la estrategia de reducción normal, la expresión $cuadrado.(cuadrado.3)$ se podría haber evaluado de esta forma:

$$\begin{array}{ll}
 & cuadrado.(cuadrado.3) \\
= & \{ \text{ definición de cuadrado } \} \\
 & x * x \\
 & \quad [\![x = cuadrado.3]\!] \\
= & \{ \text{ definición de cuadrado } \} \\
 & x * x \\
 & \quad [\![x = 3 * 3]\!] \\
= & \{ * \} \\
 & x * x \\
 & \quad [\![x = 9]\!] \\
= & \{ \text{ definición local } \} \\
 & 9 * 9 \\
= & \{ * \} \\
 & 81
\end{array}$$

y de esta forma solo 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 pueden compartir subexpreciones, se denomina generalmente en la literatura *modelo computacional lazy*.

Como ya dijimos en la introducción (pag. 81) la definición formal del modelo computacional sirve, entre otras cosas, para calcular la complejidad de las funciones definidas en nuestro formalismo. Es así que, si nuestra implementación del lenguaje está realizada en base al modelo lazy, puede suceder que funciones definidas con variables repetidas, sean más eficientes. En nuestro ejemplo la definición de $cuadrado \doteq x * x$ tiene la variable x repetida con lo cual las expresiones que instancien las mismas, serán reducidas solo una vez en este modelo, como ya se ha visto.

Este modelo es el que se utiliza en la mayoría de las implementaciones de los lenguajes funcionales puros. Por lo general es implementado representado las expresiones como un grafo, denominado *grafo de reducción*. Para más información sobre implementación de lenguajes funcionales ver [JL91].

En la próxima sección veremos una forma analítica de obtener una medida de la eficiencia de las funciones recursivas definidas en el formalismo básico.

9.6. Nociones de eficiencia de programas funcionales

En esta sección veremos una forma de calcular la complejidad de las funciones definidas en el formalismo básico para asi poder comparar la *eficiencia* de programas funcionales. La misma se medirá contando la cantidad de reducciones ejecutadas. En general, nos interesará únicamente el comportamiento asintótico, por eso en muchos casos no tendremos en cuenta las constantes.

Supongamos que para una función cualquiera f podemos definir $Tf \doteq$ costo de ejecutar f. Entonces el costo de reducir $f.E$, donde E es una expresión, será

$$T(f.E) = Tf + \textit{costo de reducir } E$$

La cuestión será, pues, definir Tf. No veremos ningún método para resolver esto sistemáticamente, sino que procederemos a través de ejemplos.

Ejemplo 9.1 (Calcular $Tfac$)

$$\begin{array}{rcl} fac.0 & \doteq & 1 \\ fac.(n+1) & \doteq & (n+1) * fac.n \end{array} \quad \Rightarrow \quad \left\{ \begin{array}{rcl} Tfac.0 & = & 1 \\ Tfac.(n+1) & = & 1 + Tfac.n \end{array} \right.$$

Ahora hay que resolver estas ecuaciones. Si tomamos $Tfac.n \doteq n+1$, tenemos una solución de las últimas ecuaciones; por lo tanto esta definición de $Tfac$ es apropiada. Así, el costo de calcular fac es lineal (depende de n).

Ejemplo 9.2 (Calcular Tf, donde f está dada por)

$$\begin{array}{rcl} f.0 & \doteq & 0 \\ f.(n+1) & \doteq & f.n + g.n \\ & & [\![g.i = X^i]\!] \end{array}$$

Para el costo de g tenemos

$$\left. \begin{array}{rcl} Tg.0 & = & 1 \\ Tg.(n+1) & = & 1 + Tg.n \end{array} \right\} \Rightarrow \quad Tg.n = n+1 \quad \forall\, n$$

Entonces, para f tenemos

$$\begin{array}{rcl} Tf.0 & = & 1 \\ Tf.(n+1) & = & 1 + Tf.n + Tg.n \\ & = & 2 + n + Tf.n \end{array}$$

A partir de estas ecuaciones se puede demostrar que si se define

$$Tf.n \doteq \frac{n*(n+3)}{2} \quad \forall\, n$$

se satisfacen las ecuaciones. Por lo tanto el costo de evaluar f es cuadrático.

9.7. Problema de la forma normal

Como se mencionó en la introducción (pag. 81) el desarrollo formal del modelo computacional se puede utilizar para obtener algunos resultados referentes a nuestro formalismo.

Supongamos que existe un predicado $nf : Num \to Bool$ en nuestro formalismo básico el cual determina si una expresión entera tiene forma normal. Su especificación es

$$nf.e \equiv e \text{ tiene forma normal} \tag{9.2}$$

Construyamos ahora la definición

$$
\begin{array}{|lllll}
P \doteq & (& nf.P & \rightarrow & P \\
 & \Box & \neg nf.P & \rightarrow & 0 \\
 &) & & &
\end{array}
$$

Usando el modelo computacional es obvio que P tiene forma normal (igual a 0) si y sólo si $\neg nf.P$. Luego

P tiene forma normal
$\equiv$ { modelo computacional }
$\neg nf.P$
$\equiv$ { especificación de nf }
$\neg$(P tiene forma normal)

absurdo. Luego, no puede existir un predicado nf que satisfaga la especificación 9.2.

9.8. Ejercicios

Ejercicio 9.1
Mostrar los pasos de reducción hasta llegar a la forma canónica de la expresión:

$$2 * cuadrado.(head.[2, 4, 5, 6, 7, 8])$$

Ejercicio 9.2
Dada la definición:

$$linf \doteq 1 \triangleright linf$$

mostrar los pasos de reducción hasta llegar a la forma canónica de la expresión:

$$head.linf$$

1. reduciendo primero las subexpresiones más internas.
2. utilizando la estrategia de reducción normal.

Comparar ambos resultados.

Ejercicio 9.3
Dada la definición:

$$
\begin{cases}
f.x.0 & \doteq x \\
f.x.(n+1) & \doteq cuadrado.(f.x.n)
\end{cases}
$$

mostrar los pasos de reducción hasta llegar a la forma canónica de la expresión:

$$f.2.3$$

1. utilizando la estrategia de reducción normal, sin utilizar definiciones locales.
2. utilizando el modelo computacional lazy.

Comparar ambos resultados.

Bibliografía

[BEKV94] K. Broda, S. Eisenbach, H. Khoshnevisan, and S. Vickers. *Reason programming.* Prentice Hall, 1994.

[Bir98] R. Bird. *Introduction to Functional Programming using Haskell.* Prentice Hall, 1998.

[Coh90] E. Cohen. *Programming in the 1990's. An Introduction to the Calculation of Programs.* Texts and Monographs in Computer Science. Springer Verlag, 1990.

[Cop73] Irving M. Copi. *Introducción a la Lógica.* Texts and Monographs in Computer Science. Eudeba, Buenos Aires, 1973.

[DF88] E.W. Dijkstra and W.H.J. Feijen. *A Method of Programming.* Addison Wesley, 1988.

[Dij76] E.W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[Dij89] E.W. Dijkstra. *Formal Development of Programs and Proofs.* Addison Wesley, 1989.

[DS90] E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics.* Texts and Monographs in Computer Science. Springer Verlag, 1990.

[Fok96] M. Fokkinga. *Werkcollege Functioneel Programmeren.* Universidad de Enschede, 1996.

[Gri81] D. Gries. *The Science of Programming.* Springer Verlag, 1981.

[GS93] D. Gries and F.B. Schneider. *A Logical Approach to Discrete Math.* Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Comm. of the ACM*, 12:576–580, 1969.

[Hoo89] R. Hoogerwoord. *The Design of Functional Programs: a Calculational Approach.* PhD thesis, Eindhoven University of Technology, Países Bajos, 1989.

[JL91] Simon L. Peyton Jones and Davis R. Lester. Implementing functional languages: a tutorial, February 1991. Este libro puede encontrarse en Internet `ftp://ftp.dcs.glasgow.ac.uk/pub/pj-lester-book`.

[Kal90] A. Kaldeweij. *Programming: the Derivation of Algorithms.* Prentice Hall, 1990.

[Par90] H.A. Partsch. *Specification and Transformation of Programs: a Formal Approach to Software Development.* Texts and Monographs in Computer Science. Springer Verlag, 1990.

[Smu78] R. Smullyan. *What is the Name of This Book?* Prentice Hall, New Jersey, 1978.

[Tho96] S. Thompson. *Haskell: The Craft of Functional Programming.* Addison Wesley, 1996.