Cálculo de Programas

Javier Blanco Silvina Smith Damián Barsotti

CÁLCULO DE PROGRAMAS

Javier Blanco, Silvina Smith, Damián Barsotti.
Facultad de Matemática, Astronomía y Física,
Universidad Nacional de Córdoba.

Idea de tapa: foto de monumento a Franz Kafka, Praga, por Karina Moroni. Composición fotográfica por Damián Barsotti.

ISBN N:

Impreso en Argentina.

Índice General

Prefacio

Existen diversas maneras de introducir los conceptos y técnicas elementales de programación. Dos maneras bastante exploradas y que han dado lugar a materiales didácticos muy buenos son la de construcción sistemática de programas imperativos [Dij76, Gri81, DF88, Kal90, Bac03] y la de programación funcional [Tho96, Fok96, Hoo89, Bir98]. En este libro nos basaremos en ambas, tratando de mostrar que si bien los paradigmas de programación y los modelos computacionales son diferentes, a la hora de desarrollar programas las técnicas usadas se parecen más y se complementan. Más aún, es posible usar el paradigma funcional (el cual es más abstracto) en el desarrollo de programas imperativos. No se explotan en este libro todas las posibilidades de ambos paradigmas (por ejemplo, se usan muy poco las funciones de alto orden en programación funcional y no se usan los procedimientos y las funciones imperativas) sino que se trata de usar un núcleo mínimo que sea suficiente para resolver una familia interesante de problemas, pudiendo enfocarnos así en los métodos de resolución de problemas.

La primera versión de este libro se basó en notas escritas por Silvina Smith a partir de clases de Algoritmos y Estructuras de Datos dictadas por Javier Blanco en la Facultad de Matemática, Astronomía y Física de la Universidad Nacional de Córdoba en 1998. Estas notas fueron publicadas en 1999 en la *S*erie C, Trabajos de Informática, editada por dicha Facultad [BS99]. Desde entonces se utilizan en el dictado de asignaturas de la carrera de Licenciatura en Ciencias de la Computación. El material fue posteriormente revisado y ampliado, introduciéndose nuevos capítulos, ya con la participación de Damián Barsotti.

Versiones preliminares de esta edición comenzaron a utilizarse en 2001 en la Universidad Nacional de Río IV y en 2003 en la Universidad Nacional de Rosario como material bibliográfico para el dictado de asignaturas afines al contenido de la misma.

Es nuestro deseo que la primera edición de este libro sirva de guía para quienes deseen sumarse a nuestra propuesta de introducir la programación.

Agradecimientos

Agradecemos en primer lugar a todas las personas que colaboraron con sugerencias para mejorar este trabajo.

Agradecemos también la financiación parcial con fondos provenientes del Programa de Apoyo a las Tecnicaturas en Informática.

Por último, y no menos importante, un agradecimiento especial a todos nuestros afectos.

Capítulo 1

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*

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, creativa e interesante.

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.

Los programas son fórmulas lógicas tan complejas y voluminosas que fue difícil reconocerlos como tales. Los estudios de lógica matemática se centraron en analizar su poder expresivo y su adecuación al razonamiento humano, entre sus objetivos no estaba el de usarla como una herramienta efectiva para hacer demostraciones con fórmulas muy grandes. La deducción natural, por ejemplo, es un sistema de pruebas muy adecuado para comprender ciertos aspectos del razonamiento matemático pero dado su estilo de escribir las demostraciones como árboles se torna

muy engorroso trabajar ya con fórmulas de tamaño mediano. En lo que sigue presentaremos un estilo de lógica orientado a trabajar de manera más adecuada con fórmulas de mayor tamaño.

1.1 Resolución de ecuaciones

Como ya mencionamos en la introducción anterior, la lógica matemática es una herramienta indispensable para el desarrollo de programas. Para comenzar ejemplificando el tipo de presentación de la lógica que vamos a usar en este libro, comenzaremos aplicándola a una clase de problemas más simple y conocida como es el de la resolución de ecuaciones aritméticas.

En los libros de texto de matemática utilizados en las escuelas secundarias, la resolución de ecuaciones es presentada en general como un proceso mecánico el cual consiste en "despejar la incógnita". Sumado a esto, la lógica suele ser enseñada como un objeto de estudio separado y no como una herramienta para resolver estos problemas matemáticos. Es así que, el proceso mecánico de despejar la incógnita es poco comprendido por los alumnos debido en gran medida a la falta de un formalismo notacional claro y uniforme. En particular, los casos especiales en los cuales una ecuación admite infinitas o ninguna solución son tratados como anomalías. Además, cuando se introducen ecuaciones con soluciones múltiples el mecanismo notacional debe adaptarse o directamente dejarse de lado.

A continuación presentaremos una serie de ejemplos de resolución de ecuaciones utilizando la lógica como herramienta, no solo para resolverlas si no también para comprender y justificar los resultados obtenidos.

Ejemplo 1.1
Supongamos que se nos propone resolver la siguiente ecuación lineal:

$$3 * x + 1 = 7$$

Pensemos primero en el significado de esta ecuación. Está constituida por dos expresiones aritméticas unidas por un símbolo de igualdad. La presencia de la igualdad induce a pensar que toda la ecuación tiene cierto valor de verdad, es decir que será verdadera o falsa. Pero, la ecuación tal cual está escrita, no tiene un valor de verdad fijo. La presencia de la variable x indica que el valor de verdad de la ecuación dependerá del valor que tome esta variable. Para algunos valores será verdadera y para otros será falsa. Por ejemplo, si x toma el valor 1 la ecuación es falsa, pero si x es igual a 2 la misma es verdadera. La resolución de la ecuación consiste en identificar el conjunto de valores de x para los cuales la ecuación es verdadera. La manera usual de hacer esto es obtener otra ecuación equivalente a la primera pero suficientemente simple como para que el conjunto de soluciones sea visible de manera inmediata. Introduciremos a través de este ejemplo el formato de demostración que luego usaremos en todo el libro.

$$
\begin{array}{ll}
 & 3 * x + 1 = 7 \\
\equiv & \{ \text{ restamos 1 a cada término } \} \\
 & 3 * x = 6 \\
\equiv & \{ \text{ dividimos por 3 cada término } \} \\
 & x = 2
\end{array}
$$

En la demostración, las leyes de la aritmética aseguran la correción de cada paso, esto es que cada una de las sucesivas ecuaciones tiene el mismo conjunto de soluciones o, dicho de otra manera, son verdaderas para los mismos valores de x.

Como resultado de la demostración podemos afirmar que el valor de verdad (verdadero o falso) de la primera ecuación, para un x fijo, es el mismo que el de la última, para el mismo valor de x. O lo que es lo mismo, el conjunto de valores de x para los cuales la ecuación $3 * x + 1 = 7$ es verdadera, es exactamente el mismo para los que la ecuación $x = 2$ es verdadera. De esto último se desprende que el único valor para la variable x que hace verdadera la ecuación $3 * x + 1 = 7$ es 2. Llamaremos al conjunto de valores que hace verdadera una ecuación como el conjunto de soluciones de esa ecuación.

Observación: El tamaño de los pasos de una derivación dependerá del objetivo de la demostración que se esté realizando. En este caso, se está presentando de manera detallada la resolución de una ecuación lineal, por lo cual los pasos son particularmente detallados. Si se está trabajando por ejemplo en la derivación de programas, donde el enfásis está en otro lado, la demostración precedente puede abreviarse como

$$
\begin{array}{ll}
 & 3 * x + 1 = 7 \\
\equiv & \{ \text{ aritmética } \} \\
 & x = 2
\end{array}
$$

donde se asume que el lector de la demostración puede comprender el paso sin ningún problema. De todas maneras frente a la duda conviene ser lo más explícito posible.

Ejemplo 1.2
Consideremos la resolución de la siguiente ecuación:

$$
\begin{array}{ll}
 & 3 * x + 1 = 3 * (x + 1) - 2 \\
\equiv & \{ \text{ disributiva } \} \\
 & 3 * x + 1 = 3 * x + 3 - 2 \\
\equiv & \{ \text{ aritmética } \} \\
 & 3 * x + 1 = 3 * x + 1
\end{array}
$$

$\equiv$ { reflexividad de la igualdad }
$True$

Usamos la palabra $True$ para denotar la proposición que siempre es verdadera, independientemente de los valores de las variables que se estén usando. Esto significa que para cualquier valor de la variable x la ecuación es verdadera o, dicho de otro modo, que el conjunto de soluciones de la última ecuación es el de todos los enteros (o naturales, o reales, según como se haya planteado el problema). Por lo tanto, como resultado de la demostración, el conjunto de soluciones de la ecuación $3 * x + 1 = 3 * (x + 1) - 2$ es el de todos los números enteros.

Ejemplo 1.3
Consideremos ahora la resolución de la siguiente ecuación:

$3 * x + 1 = 3 * (x + 1)$
$\equiv$ { disributiva }
$3 * x + 1 = 3 * x + 3$
$\equiv$ { restamos $3 * x$ en cada miembro }
$1 = 3$
$\equiv$ { igualdad de enteros }
$False$

Simétricamente al ejemplo anterior, la proposición $False$ es la que es falsa para cualquier valor de x. Se dice en este caso que la ecuación no tiene solución o equivalentemente que su conjunto de soluciones es vacío. Por lo tanto, como resultado de la demostración, el conjunto de soluciones de la ecuación $3 * x + 1 = 3 * (x + 1)$ es vacío, o directamente, no tiene solución.

Ejemplo 1.4
Resolveremos ahora la siguiente ecuación cuadrática

$x * (x - 2) = 3 * (x - 2)$
$\equiv$ { restamos $3 * (x - 2)$ a ambos términos }
$x * (x - 2) - 3 * (x - 2) = 0$
$\equiv$ { distributiva }
$(x - 3) * (x - 2) = 0$
$\equiv$ { producto igual a 0: $a * b = 0 \equiv a = 0 \vee b = 0$ }
$x - 3 = 0 \vee x - 2 = 0$
$\equiv$ { sumamos 3 al primer término y 2 al segundo }
$x = 3 \vee x = 2$

Se ha llegado entonces a una fórmula lógica que comprende dos ecuaciones. El conectivo $\vee$ denota la diyunción inclusiva de dos fórmulas lógicas. El uso de conectivos lógicos permite mantener el formato unificado de demostración aún en los casos en los cuales hay más de una solución. Esto hace más facil entender la conexión que hay entre las dos ecuaciones obtenidas ($x = 3$ y $x = 2$). La última fórmula tiene claramente como conjunto de soluciones a los valores 3 y 2, y por la demostración, es el mismo conjunto de soluciones que para la ecuación $x * (x - 2) = 3 * (x - 2)$.

Observación: Un producto de dos reales es cero si alguno de los factores lo es. La presentación de esta regla usando una equivalencia no es la usual en los libros de matemática pese a que, como se ve en el ejemplo, ayuda a simplificar los cálculos.

Ejemplo 1.5
Como último ejemplo consideremos la siguiente ecuación cubica:

$$
\begin{aligned}
& (x-1) * (x^2+1) = 0 \\
\equiv\ & \{ \text{ producto igual a 0: } a * b = 0 \equiv a = 0 \vee b = 0 \ \} \\
& x - 1 = 0 \vee x^2 + 1 = 0 \\
\equiv\ & \{ \text{ sumamos 1 al primer término y restamos 1 al segundo } \} \\
& x = 1 \vee x^2 = -1 \\
\equiv\ & \{ \text{ el cuadrado nunca es negativo } \} \\
& x = 1 \vee \mathit{False} \\
\equiv\ & \{ \mathit{False} \text{ es neutro para el } \vee \ \} \\
& x = 1
\end{aligned}
$$

El último paso implica un conocimiento de las propiedades de los conectivos lógicos. En capítulos posteriores se verán estas propiedades detalladamente.

1.2 Resolución de desigualdades

A continución veremos el uso de la misma notación para resolver desigualdades. Nos vamos a concentrar en obtener la validez de desigualdades que contengan raíces cuadradas. Como veremos, utilizando el fórmalismo lógico, esta tarea se resuelve de manera muy simple, mostrando la potencia que tiene esta herramienta.

Ejemplo 1.6
Supongamos que queremos determinar si el número $\sqrt{2}+\sqrt{7}$ es menor que $\sqrt{3}+\sqrt{5}$ sin usar una calculadora. Notemos, que ambas expresiones no dependen de ninguna variable, con lo cual la desigualdad $\sqrt{2} + \sqrt{7} < \sqrt{3} + \sqrt{5}$ posee un valor de verdad fijo (verdadero o falso). La idea es aprovechar las propiedades de las desigualdades y transformar la expresión a resolver en una más simple cuyo valor de verdad sea inmediato.

$$
\begin{aligned}
& \sqrt{2}+\sqrt{7} < \sqrt{3}+\sqrt{5} \\
\equiv & \ \{\ a < b \equiv a^2 < b^2 \text{ para } a, b \text{ positivos }\ \} \\
& (\sqrt{2}+\sqrt{7})^2 < (\sqrt{3}+\sqrt{5})^2 \\
\equiv & \ \{\ \text{cuadrado de un binomio }\ \} \\
& 2+2*\sqrt{2}*\sqrt{7}+7 < 3+2*\sqrt{3}*\sqrt{5}+5 \\
\equiv & \ \{\ \text{aritmética }\ \} \\
& 9+2*\sqrt{2}*\sqrt{7} < 8+2*\sqrt{3}*\sqrt{5} \\
\equiv & \ \{\ \text{restamos 8 a ambos miembros }\ \} \\
& 1+2*\sqrt{2}*\sqrt{7} < 2*\sqrt{3}*\sqrt{5} \\
\equiv & \ \{\ \text{elevamos al cuadrado }\ \} \\
& 1+2*2*\sqrt{2}\sqrt{7}+4*2*7 < 4*3*5 \\
\equiv & \ \{\ \text{aritmética }\ \} \\
& 57+2*2*\sqrt{2}\sqrt{7} < 60 \\
\equiv & \ \{\ \text{restamos 57 a ambos miembros }\ \} \\
& 2*2*\sqrt{2}\sqrt{7} < 3 \\
\equiv & \ \{\ \text{elevamos al cuadrado }\ \} \\
& 16*2*7 < 3*3 \\
\equiv & \ \{\ \text{aritmética }\ \} \\
& 224 < 9 \\
\equiv & \ \{\ \text{aritmética }\ \} \\
& \mathit{False}
\end{aligned}
$$

A partir de la demostración vemos que la fórmula $\sqrt{2}+\sqrt{7} < \sqrt{3}+\sqrt{5}$ tiene el mismo valor de verdad que la fórmula $False$, por lo tanto no es el caso que $\sqrt{2}+\sqrt{7} < \sqrt{3}+\sqrt{5}$.

Podríamos haber intentado probar que ambos términos de la desigualdad anterior son iguales o que el primero es mayor que el segundo (estos son los únicos casos posibles restantes). Dado que todas las propiedades usadas preservan tanto la relación de menor, la de igualdad como la de mayor, la demostración puede fácilmente adaptarse para mostrar que $\sqrt{2}+\sqrt{7} > \sqrt{3}+\sqrt{5}$ es equivalente a $224 > 9$ lo cual es obviamente cierto. De la misma forma, para el caso de la igualdad llegaremos a un resultado negativo ya que no se cumple $224 = 9$. Con ello logramos obtener cual es la relación que hay entre los dos términos.

1.3 ¿Qué se puede aprender de una torta?

Presentaremos ahora un ejemplo para aclarar los conceptos. El mismo está tomado de 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 inspirada en una discusión similar en [Coh90].

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

Al igual que cuando trabajamos con aritmética y desigualdades, 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 son 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 }

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

$$
\begin{aligned}
&= \{ \text{ un punto de intersección interna cada 4 puntos en la circunferencia } \} \\
&\quad 1 + \binom{N}{2} + \binom{N}{4} \\
&= \{ \text{ álgebra } \} \\
&\quad 1 + \frac{N^4 - 6N^3 + 23N^2 - 18N}{24}
\end{aligned}
$$

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.

1.4 Expresiones aritméticas

A continuación profundizaremos algunos conceptos utilizados en las secciones anteriores aunque no fueron nombrados de forma explícita. Comenzaremos con los de estado, evaluación y sustitución. Los mismos tienen sentido para diferentes conjuntos de expresiones, por ejemplo expresiones aritméticas, booleanas, de conjuntos, etc. En particular, en las secciones anteriores se manipularon 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 $((3 + x) * 2)$.

De aquí en adelante utilizaremos letras mayúsculas para denotar expresiones y letras minúsculas para las variables.

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 1.1 (Asignación de valores o estado)
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.

Un estado en particular será escrito como un conjunto de pares. El primer elemento de cada par será el nombre de la variable y el segundo su valor. En el conjunto habrá un solo par para cada variable que aparezca en la expresión. Por ejemplo, un estado posible de la expresión $x * 5 + y$ es $\{(x, 3), (y, 8)\}$. En el ejemplo 1.1 el único estado que hace verdadera la ecuación es $\{(x, 2)\}$.

Definición 1.2 (Evaluación)
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.

1.5 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.

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).

1.6 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. Este operador ya fue utilizado en los ejemplos de la sección 1.1 para definir ecuaciones a partir de dos expresiones aritméticas. Como vimos en estos ejemplos, 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. Si los valores son distintos su valor final será $False$. La expresión $E = F$ se considerará verdadera (o equivalente a la expresión $True$) si su evaluación en

todo estado posible produce el valor $True$. Si su evaluación en *todo estado posible* produce el valor $False$ la expresión será falsa (o equivalente a la expresión $False$). Notar, como en el primer ejemplo de la sección 1.1, la existencia de expresiones que no son verdaderas ni falsas. En estos casos la expresión en cuestión se evaluará a distintos valores según el estado considerado, por lo que no se podrá asignar un valor de verdad a la misma.

Observación: De lo dicho anteriormente, debe quedar claro la diferencia entre una expresión y un valor. Una expresión es una entidad de carácter sintáctico. Su definición puede expresarse de manera puramente formal dando la manera de construirla y su precedencia, como se hizo con las expresiones aritméticas en la sección 1.4. Pero el concepto de valor asociado a una expresión solo tiene sentido si partimos de un estado dado. Esta diferencia conceptual entre una expresión y su valor ya fue establecida en la sección 1.1, al referirnos al conjunto de soluciones de una ecuación aritmética. Este conjunto contiene justamente los estados para los cuales la ecuación se evalúa al valor $True$. Sin este conjunto la ecuación por si sola no tiene nesesariamente un valor.

Observación: También debe quedar claro que existen expresiones que tienen el mismo valor independiente del estado. Este es el caso de las desigualdes tratadas en la sección 1.2, o las expresiones "$True$", "$False$", "$1 = 2$" o "$x - x$".

Observación: Se habrá podido observar también que hemos usado indistintamente la notación $True$ y $False$ para indicar los valores y las expresiones que denotan los mismos. También sucederá lo mismo para expresiones aritméticas que tienen asociado directamente un valor, como por ejemplo "1", "2", "3" (esta asociación será discutida en el capítulo 8). El lector podrá discernir cuando se esté hablando de uno u otro concepto según el contexto en el que se esté hablando.

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 (como vimos con las leyes de la aritmética en las secciones 1.1 y 1.2) 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:

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

donde X, Y y E son cualquier expresión.

La forma de leer la regla anterior es la siguiente: si la expresion por encima de la barra se evalúa al valor $True$ para todo estado posible, entonces la expresión de abajo también lo hace. O dicho de otra manera, si la expresion de arriba es verdadera (equivalente a la expresión $True$) entonces la expresión de abajo también lo es.

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

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

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.

Ejemplo 1.7 (Máximo entre dos números)
Veamos ahora un ejemplo donde aplicaremos una serie de reglas nuevas. Los axiomas presentados aquí no son necesariamente los mas simples ni completos. Luego de trabajar con el cálculo proposicional y con la noción de equivalencia lógica presentaremos una versión más simple y completa del máximo y mínimo (ver sección 4.6).

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 1.3 (Conmutatividad)

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

Axioma 1.4 (Asociatividad)

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

Axioma 1.5 (Idempotencia)

$$P \max P = P$$

Axioma 1.6 (Distributividad de + con respecto a max)

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

Axioma 1.7 (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 próxima propiedad a demostrar.

Usando estas reglas se demostrará la siguiente propiedad:

Teorema 1.8

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

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

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

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:

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

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

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

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 concluye que $E_0 = E_n$.

Debido a que (casi) todos los pasos de la demostración son explícitos, esta 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, 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 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 $\max$ }
$Y\ \max\ Z + W\ \max\ X$

1.7 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 usualmente 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

$$
\begin{array}{ll}
 & f.2 \\
= & \{ \text{ aplicación de } f \ \} \\
 & 2^3 \\
= & \{ \text{ aritmética } \} \\
 & 8
\end{array}
$$

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. Esto puede hacerse para cualquier expresión, no necesariamente para constantes. Por ejemplo

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

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 7.

1.8 Breve descripción de lo que sigue

El estilo de demostración presentado en las secciones precedentes tiene la ventaja de ser bastante autocontenido y de ser explícito cuando se usan resultados

de otras áreas. 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. Esto 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 estos 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 correcto 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

1.9 Ejercicios

Ejercicio 1.1
Resolver las siguientes ecuaciones:

- $2 * x + 3 = 5$

- $2 * x + 3 = 2 * x$

Ejercicio 1.2
Simplificar la expresión $(n+1)^2 - n^2$ de manera que no aparezcan potencias.

Ejercicio 1.3
Demostrar que $\sqrt{3} + \sqrt{11} > \sqrt{5} + \sqrt{7}$

Ejercicio 1.4
Determinar el orden entre $\sqrt{3} + \sqrt{13}$ y $\sqrt{5} + \sqrt{11}$

Ejercicio 1.5
Describiremos ahora un algoritmo para determinar si una suma de raíces cuadradas $\sqrt{a} + \sqrt{b}$ es menor que otra $\sqrt{c} + \sqrt{d}$, donde a, b, c, d son todos números naturales. El algoritmo no funciona para cualesquiera a, b, c, d. Encontrar un contraejemplo y corregir el algoritmos para que funcione siempre:

Paso 1. Elevar al cuadrado ambos miembros y simplificar la desigualdad para que quede de la forma $p + \sqrt{q} < r + \sqrt{s}$.

Paso 2. Restar p a ambos miembros y simplificar el miembro derecho de manera que la desigualdad quede de la forma $\sqrt{q} < t + \sqrt{s}$.

Paso 3 Elevar ambos miembros al cuadrado. Simplificar el miembro derecho para que quede de la forma $q < u + \sqrt{v}$.

Paso 4. Restar u a ambos miembros y simplificar dejando la desigualdad de la forma $w < \sqrt{t}$.

Paso 5. Elevar de nuevo al cuadrado y simplificar. Quedan dos números enteros. Si el primero es menor que el segundo, entonces la desigualdad es verdadera.

Ejercicio 1.6
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?

Ejercicio 1.7
Dar el conjunto de estados para los cuales la expresión $\dfrac{x-1}{x^2-1}$ está bien definida.

Ejercicio 1.8
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 1.9
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 1.10
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 1.11
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 1.12
Definir el valor absoluto en términos del máximo y demostrar la desigualdad triangular:
$|X| + |Y| \geq |X + Y|$

Capítulo 2

Lógica proposicional

> "Who did you pass on the road?" the King went on, holding out his hand to the Messenger for some hay.
>
> "Nobody," said the Messenger.
>
> "Quite right," said the King: "this young lady saw him too. So of course Nobody walks slower than you."
>
> "I do my best," the Messenger said in a sullen tone. "I'm sure nobody walks much faster than I do!".
>
> "He can't do that," said the King, "or else he'd have been here first. However, now you've got your breath, you may tell us what happened in the town."
>
> Lewis Carroll: *Through the Looking-Glass*

2.1 Introducción

Se suele definir a la lógica como el estudio de los métodos y principios usados para distinguir los razonamientos correctos de los incorrectos. En este sentido, la lógica estudia básicamente la forma de estos razonamientos, y determina si un razonamiento es válido o inválido pero no si la conclusión de este es verdadera o no. Lo único que la lógica puede afirmar de un razonamiento correcto es que si se partió de premisas verdaderas la conclusión será verdadera, pero en el caso que alguna de las premisas sea falsa nada se sabrá del valor de verdad de la conclusión.

Uno de los conceptos básicos de lógica es el de *proposición*. Suele definirse una proposición como una sentencia declarativa de la cual puede decirse que es verdadera o falsa. Así se las distingue de otros tipos de oraciones como las interrogativas,

exclamativas o imperativas, dado que de una pregunta, una exlamación o una orden no puede decirse que sea verdadera o falsa: su función en el lenguaje no es postular o establecer hechos.

En los textos de lógica suele distinguirse entre una oración o sentencia y su significado, reservándose la palabra proposición para este último. Por ejemplo

> Está lloviendo afuera.
> Afuera está lloviendo.

serán consideradas como dos oraciones diferentes pero no se las distinguirá como proposiciones, dado que su significado es obviamente el mismo.

Otro ejemplo usual de diferentes oraciones que dan lugar a la misma proposición es el de las oraciones en diferentes idiomas, por ejemplo

> Llueve.
> Il pleut.
> It rains.
> Het regent.

serán consideradas iguales como proposiciones.

Por supuesto que la relación entre oraciones y proposiciones es un fenómeno complejo el cual involucra, entre otras cosas, temas tan controversiales como la noción de semántica del lenguaje natural. No avanzaremos más en este punto, quedándonos con una noción preteórica de significado.

Las proposiciones serán usadas esencialmente en *razonamientos*. Entendemos por razonamiento a un conjunto de proposiciones de las cuales se afirma que una de ellas se deriva de las otras. En este sentido, un razonamiento no es cualquier conjunto de proposiciones sino que tiene una estructura. La *conclusión* de un razonamiento es una proposición que se deriva de las otras, llamadas *premisas*. Se supone que las premisas sirven como justificación o evidencia de la conclusión. Si el razonamiento es válido, no puede aceptarse la verdad de las premisas sin aceptar también la validez de la conclusión.

Ejemplos de razonamientos válidos

Como es usual escribimos las premisas una debajo de la otra, luego trazamos una raya, la cual podría leerse como "por lo tanto" o "luego" y debajo de ella escribimos la conclusión.

Ejemplo 2.1
El primero es uno de los más usados a lo largo de la historia de la lógica.

> Todos los hombres son mortales.
> Sócrates es hombre.
> ---
> Sócrates es mortal

Ejemplo 2.2
El siguiente razonamiento tiene premisas falsas, pero la forma es correcta: si fueran ciertas las premisas debería serlo también la conclusión. Esta forma de razonamiento recibe el nombre de *silogismo* en la lógica clásica.

Las arañas son mamíferos.
Los mamíferos tienen seis patas.

Las arañas tienen seis patas.

Ejemplo 2.3
En los capítulos siguientes veremos cómo puede mostrarse la validez de un razonamiento. Este último ejemplo de razonamiento correcto es más simple en este sentido que los anteriores, dado que las herramientas teóricas necesarias para demostrarlo son más elementales.

Si tuviera pelo sería feliz.
No soy feliz.

No tengo pelo.

Ejemplos de razonamientos inválidos

Ejemplo 2.4

Si tuviera pelo sería feliz.
No tengo pelo.

No soy feliz.

Obviamente un pelado feliz puede creer en la primera premisa, por lo cual serían válidas ambas y no la conclusión. Otra forma de ver que no es válido es construir otro razonamiento de la misma forma, por ejemplo

Si me tomara una damajuana de vino me emborracharía.
No me tomé una damajuana de vino.

No estoy borracho.

Bueno, podría haberme tomado un barril de cerveza.

Ejemplo 2.5
Otro ejemplo más sutil es el siguiente:

Todos los hombres son bípedos.
Todos los hombres son mortales.

Algunos bípedos son mortales.

A primera vista parece correcto, pero no lo es dado que la conclusión tiene lo que se llama contenido existencial, es decir que afirma que efectivamente existe algún bípedo y que es mortal, mientras que las premisas dicen que en caso de haber algún hombre este sería mortal. Un razonamiento de la misma forma que

obviamente permite obtener una conclusión falsa de premisas verdaderas es el siguiente:

Todos los unicornios son equinos.
Todos los unicornios son azules.

Algunos equinos son azules.

Las premisas son ciertas dado que no hay ningún unicornio que no sea azul o que no sea un equino, mientras que la conclusión es falsa, al menos dentro de nuestros conocimientos zoológicos.

Desde hace ya 2.500 años la filosofía está buscando y proponiendo respuestas a las siguientes preguntas:

- ¿Cómo puede determinarse si un razonamiento es válido?
- ¿Cómo puede determinarse si una conclusión es consecuencia de un conjunto de premisas?, y de ser así, ¿cómo puede demostrarse que lo es?
- ¿Qué características de las estrucuturas del mundo, del lenguaje y de las relaciones entre palabras, cosas y pensamientos hacen posible el razonamiento deductivo?

Para responder al menos parcialmente a estas preguntas, es necesario postular una teoría del razonamiento deductivo.

2.2 Expresiones Booleanas

Los razonamientos presentados más arriba fueron hechos en castellano (o en inglés, en el caso del rey de Alice). Cuando se usa un lenguaje natural, se suele caer en ambigüedades y confusiones que no tienen que ver con problemas lógicos. En el razonamiento presentado al inicio del capítulo es la confusión (la cual ya aparece en La Odisea de Homero) entre la palabra "nadie" y una persona que el rey asume se llama "Nadie". Para evitar este tipo de confusiones y concentrarnos en los problemas centrales de la lógica, se creará un *lenguaje artificial* libre de este tipo de ambigüedades, al cual puedan después traducirse los razonamientos anteriores.

Las *expresiones booleanas* constituirán (una parte de) ese lenguaje. Al igual que en el capítulo anterior para el caso de las expresiones aritméticas, construiremos estas expresiones usando constantes, en este caso las constantes $True$ y $False$, variables proposicionales representadas en general por letras, el operador unario $\neg$ y los operadores binarios $\equiv, \not\equiv, \vee, \wedge, \Rightarrow$ y $\Leftarrow$. En los estados o asignaciones podrán asignarse a las variables proposicionales solo los valores $True$ o $False$, los cuales no deben ser confundidos con las respectivas constantes, las cuales son solo expresiones

sintácticas. Estas expresiones denotan respectivamente los valores de verdad $True$ y $False$, por lo cual se usa la misma palabra para ambos. Los únicos valores a los cuales las expresiones booleanas pueden evaluar en un estado dado serán también $True$ y $False$.

No presentamos aquí una definición inductiva completa de las fórmulas booleanas, dado que esta definición no presenta ninguna dificultad siendo análoga a la de expresiones aritméticas dada en el capítulo precedente. Queda dicha definición como ejercicio elemental para el lector.

2.3 Tablas de verdad

Dado que el dominio de cada operador es finito ($True$ o $False$), puede definirse cómo se evalúa cada uno de ellos enumerando los valores que toman para cada posible combinación de los valores de sus argumentos. Para los operadores binarios las posibles combinaciones de los valores de los argumentos son cuatro, mientras que para el operador unario son dos. Una forma esquemática de describir estos valores son las llamadas *tablas de verdad*. En estas tablas se colocan en la primera columna las posibles combinaciones de valores de los argumentos y debajo de cada expresión el valor que corresponde al estado descripto por una fila dada. El caso de la negación es el más simple dado que solo son necesarias dos filas:

p	$\neg p$
$True$	$False$
$False$	$True$

A continuación describimos en una tabla de verdad la semántica de todos los operadores binarios.

p	q	$p \vee q$	$p \Leftarrow q$	$p \Rightarrow q$	$p \equiv q$	$p \wedge q$	$p \not\equiv q$
$True$	$True$	$True$	$True$	$True$	$True$	$True$	$False$
$True$	$False$	$True$	$True$	$False$	$False$	$False$	$True$
$False$	$True$	$True$	$False$	$True$	$False$	$False$	$True$
$False$	$False$	$False$	$True$	$True$	$True$	$False$	$False$

Ejemplo 2.6
Las tablas de verdad serán útiles entre otras cosas para evaluar expresiones booleanas en un estado dado. Por ejemplo, la evaluación de la expresión

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

en el estado $\{(p, True\}, (q, True)\}$ puede calcularse de la siguiente manera: si q es verdadera, en la tabla del $\neg$ puede verse que $\neg q$ es falsa. Luego, en la tabla del $\wedge$

se ve que cuando alguno de los argumentos es falso (en este caso $\neg q$), entonces la conjunción es falsa. Por último, en la tabla de $\Rightarrow$ se ve que si el primer argumento (el *antecedente*) es falso, la implicación es verdadera, no importa cual sea el valor de verdad del segundo argumento (llamado el *consecuente*).

Describiremos ahora sucintamente los operadores considerados. Más adelante los usaremos para interpretar oraciones del lenguaje natural y consideraremos los aspectos más sutiles o controversiales de cómo usar la lógica matemática para razonar acerca de argumentos presentados en el lenguaje corriente.

Equivalencia. El operador $\equiv$ simbolizará la igualdad de valores de verdad. Puede por lo tanto usarse también el operador usual de la igualdad para este caso. Sin embargo, ambos operadores tendrán propiedades distintas, dado que, como es usual, la igualdad múltiple se interpreta como "conjuntiva", es decir que si se escribe $a = b = c$ se interpreta como $a = b$ y $b = c$, mientras que la equivalencia es asociativa, es decir que $a \equiv b \equiv c$ se usará para referirse equivalentemente a $(a \equiv b) \equiv c$ o a $a \equiv (b \equiv c)$. Otra diferencia esencial es la precedencia asociada a ambos operadores. La del $=$ es más alta que la de cualquier operador lógico, mientras que la de la equivalencia es la más baja. Esto es cómodo para evitar paréntesis. Por ejemplo, la transitividad de la igualdad (con un $True$ redundante, es cierto) podría escribirse como

$$a = b \wedge b = c \Rightarrow a = c \equiv True$$

lo cual es mucho más simple que la expresión con todos los paréntesis

$$(((a = b) \wedge (b = c)) \Rightarrow (a = c)) \equiv True$$

Negación. El operador $\neg$ representa la negación usual.

Discrepancia. El operador $\not\equiv$ es la negación de la equivalencia. Tendrá para expresiones booleanas el mismo significado que el operador $\neq$, pero también tendrá las mismas diferencias que la igualdad con la equivalencia. En algunos libros de lógica se lo suele llamar *disyunción exclusiva* u operador *xor*, dado que es verdadero cuando exactamente uno de sus argumentos lo es.

Disyunción. El operador $\vee$ representará el 'o' usual del lenguaje. Se lo suele llamar *disyunción inclusiva*, dado que es verdadera cuando algún argumento lo es.

Conjunción. El operador $\wedge$ representa el 'y' del lenguaje, dado que es cierto solo cuando ambos argumentos lo son.

Implicación. El operador $\Rightarrow$ es uno de los más controversiales respecto de su relación con el lenguaje. La expresión $p \Rightarrow q$ será verdadera en todos los casos excepto cuando p sea verdadera y q falsa. Este operador captura la idea de consecuencia lógica. De la tabla de verdad se desprende uno de los hechos que resulta menos intuitivo, que es que de falso se puede deducir cualquier cosa.

Consecuencia. El operador $\Leftarrow$ es el converso de $\Rightarrow$, en el sentido de que son equivalentes $p \Rightarrow q$ con $q \Leftarrow p$. Se introduce en nuestras expresiones debido a su utilidad en el proceso de construcción de demostraciones.

Definición 2.1 (Precedencia)
Para cada operador nuevo se definirá también la *precedencia* la cual nos permite eliminar paréntesis. Por ejemplo, si el operador $\oplus$ tiene precedencia sobre el operador $\otimes$, entonces puede escribirse $X \oplus Y \otimes Z$ para indicar $(X \oplus Y) \otimes Z$; en cambio en la expresión $X \oplus (Y \otimes Z)$ no pueden eliminarse los paréntesis sin cambiar su sentido.

Los operadores lógicos tendrán la siguiente precedencia:

1. $\neg$ (la precedencia más alta).
2. $\wedge$ y $\vee$.
3. $\Rightarrow$ y $\Leftarrow$.
4. $\equiv$ y $\not\equiv$.

Por ejemplo, la fórmula

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

puede simplificarse como

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

Uso de las tablas de verdad. Las tablas de verdad pueden usarse para calcular los posibles valores de verdad de expresiones más complejas. Un estado estará descripto por una fila y el valor asociado a una expresión en ese estado será el valor en esa fila y en la columna de la fórmula. Por ejemplo:

p	q	r	$\neg q$	$p \wedge \neg q$	$p \wedge \neg q \Rightarrow r$
$True$	$True$	$True$	$False$	$False$	$True$
$True$	$True$	$False$	$False$	$False$	$True$
$True$	$False$	$True$	$True$	$True$	$True$
$True$	$False$	$False$	$True$	$True$	$False$
$False$	$True$	$True$	$False$	$False$	$True$
$False$	$True$	$False$	$False$	$False$	$True$
$False$	$False$	$True$	$True$	$False$	$True$
$False$	$False$	$False$	$True$	$False$	$True$

2.4 Tautologías y contradicciones

Definición 2.2 (Tautología)
Una *tautología* es una expresión booleana cuya evaluación en cualquier estado es siempre verdadera (cualesquiera sean los valores de verdad que se asignen a las variables proposicionales que aparecen en ella). Una tautología se reconoce porque en su columna en la tabla de verdad todos los valores son $True$. Ejemplos de tautologías son $p \Rightarrow p$ o $p \wedge p \equiv p$.

Definición 2.3 (Contradicción)
Una *contradicción* es una expresión booleana cuya evaluación en cualquier estado es siempre falsa. La negación de una tautología es una contradicción y viceversa. Ejemplos de contradicciones son $p \wedge \neg p$ o $p \not\equiv p$.

Ejercicio 2.1
No siempre es obvio si una expresión booleana es una tautología o una contradicción. Por ejemplo, se deja como ejercicio comprobar que la siguiente fórmula es efectivamente una tautología

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

2.5 Lenguaje y lógica

La lógica matemática surgió como una manera de analizar los razonamientos escritos en lenguaje natural. Los operadores presentados en la sección anterior fueron pensados como contrapartidas formales de operadores del lenguaje. Si se tiene cierto cuidado puede traducirse una proposición escrita en lenguaje natural a lenguaje simbólico. Esta traducción nos permitirá dos cosas: por un lado resolver ambigüedades del lenguaje natural; por otro, manipular y analizar las expresiones así obtenidas usando las herramientas de la lógica. Algunas de esas herramientas serán introducidas en las secciones que siguen.

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

Por ejemplo, la oración

> Hamlet defendió el honor de su padre pero no la felicidad de su madre o la suya propia.

puede analizarse como compuesta de básicamente tres proposiciones elementales:

p: Hamlet defendió el honor de su padre.

q: Hamlet defendió la felicidad de su madre.

r: Hamlet defendió su propia felicidad.

Usando estas variables proposicionales puede traducirse la proposición como

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

Nótese que la palabra *pero* se traduce como una conjunción (dado que afirma ambas componentes). Por otro lado la disyunción usada es inclusiva, dado que podría haber defendido la felicidad de su madre, la propia o ambas.

2.6 Negación, conjunción y disyunción en el lenguaje

La operación de negación aparece usualmente en el lenguaje insertando un "no" en la posición correcta del enunciado que se quiere negar; alternativamente, puede anteponerse la frase "es falso que" o la frase "no se da el caso que" o hacer construcciones sintácticas algo más complejas. Por ejemplo el enunciado

> Todos los unicornios son azules.

puede negarse de las siguientes maneras:

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

Si simbolizamos con p a la primera proposición, usaremos $\neg p$ para cualquiera de las variantes de negación propuestas.

La conjunción se representa paradigmáticamente por la palabra *y* uniendo dos proposiciones. Otras formas gramaticales de la conjunción se interpretarán del mismo modo, por ejemplo las palabras *pero* o *aunque*. Así, si tomamos como proposiciones simples las siguientes

p: Llueve.

q: Hace frío.

interpretaremos a las siguientes oraciones como representadas por la proposición $p \wedge \neg q$.

Llueve y no hace frío.
Llueve pero no hace frío.
Aunque llueve no hace frío.

Hay que tener en cuenta, sin embargo, que la palabra *y* no siempre representa una conjunción. Si bien lo hace en la frase

Joyce y Picasso fueron grandes innovadores en el lenguaje del arte.

no es este el caso en

Joyce y Picasso fueron contemporáneos.

donde simplemente se usa para expresar una relación entre ambos.

La palabra usual para representar la disyunción es *o*, aunque existen variantes del estilo *o bien ... o bien*, etc. El caso de la disyunción es un poco más problemático que los anteriores, dado que existen en el lenguaje dos tipos de disyunción, la llamada *inclusiva* y la *exclusiva*. Ambos tipos difieren en el caso en que las proposiciones intervinientes sean ambas verdaderas. La disyunción inclusiva considera a este caso como verdadero, como por ejemplo en:

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

obviamente, teniendo el apoyo de ambos se está también en condiciones de ser emperador. Esta proposición se simboliza como

$$p \vee q$$

donde las proposiciones elementales son:

p: Para ser emperador hace falta el apoyo de la nobleza.

q: Para ser emperador hace falta el apoyo del pueblo.

Un ejemplo de disyunción exclusiva es:

El consejero del emperador pertenece a la nobleza o al pueblo.

donde se interpreta que no puede pertenecer a ambos a la vez. Esta proposición se simboliza como

$$p \not\equiv q$$

donde las proposiciones elementales son

p: El consejero del emperador pertenece a la nobleza.

q: El consejero del emperador pertenece al pueblo.

En latín existen dos palabras diferentes para la disyunción inclusiva y exclusiva. La palabra "vel" se usa para la disyunción inclusiva mientras que para la exclusiva se usa la palabra "aut". El símbolo usado en lógica para la disyunción proviene precisamente de la inicial de la palabra latina.

2.7 Implicación y equivalencia

La implicación lógica suele representarse en el lenguaje natural con la construcción *si ... entonces ...* Es uno de los conectivos sobre los que menos acuerdo hay y al que más alternativas se han propuesto. Casi todo el mundo está de acuerdo en que cuando el antecedente p es verdadero y el consecuente q es falso, entonces $p \Rightarrow q$ es falsa. Por ejemplo, el caso de la afirmación

Si se reforman las leyes laborales entonces bajará el desempleo.

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

Si dos más dos es igual a cinco, entonces yo soy el Papa.

Para la lógica clásica que estamos presentando aquí, la frase anterior es verdadera (mirando la tabla de verdad del $\Rightarrow$, vemos que si ambos argumentos son falsos entonces la implicación es verdadera).

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

Si la economía mejoró con esos ajustes, yo soy Gardel.

es una manera estilizada de decir que la economía no mejoró con esos ajustes.

Otra forma de representar la implicación (y la consecuencia) en el lenguaje es a través de las palabras *suficiente* y *necesario*. Por ejemplo la siguiente proposición:

Es suficiente que use un preservativo para no contagiarme el sida

puede simbolizarse con $p \Rightarrow \neg q$ donde las proposiciones simples son:

p: Uso un preservativo.

q: Me contagio el sida.

Otro ejemplo donde entra la idea de necesidad lógica es

Para obtener una beca es necesario tener un buen promedio en la carrera.

puede simbolizarse como $p \Leftarrow q$ o equivalentemente como $q \Rightarrow p$ donde

p: Tener buen promedio.

q: Obtener una beca.

Nótese que puede darse el caso usual de que un estudiante tenga buen promedio pero no consiga ninguna beca.

Algunas veces la construcción lingüística *si ... entonces ...* o *si ..., ...* no se traduce como una implicación sino como una equivalencia. En matemática es común presentar definiciones como

Una función es biyectiva si es inyectiva y suryectiva.

lo cual podría en primera instancia traducirse como $p \wedge q \Rightarrow r$, donde p dice que una función es inyectiva, q que es suryectiva y r que es biyectiva. Si bien esa afirmación es indudablemente cierta, la definición matemática está diciendo en realidad que $p \wedge q \equiv r$. Obviamente si tengo una función biyectiva puedo asumir que es tanto inyectiva como suryectiva, lo cual no podría inferirse de la primer formulación como implicación. Una alternativa en el lenguaje es usar la construcción *si y solo si* para representar la equivalencia. De todas maneras no existe ninguna construcción del lenguaje que represente fielmente la equivalencia lógica. Es esta quizá la principal razón por la cual la equivalencia suele tener un lugar secundario en los cursos de lógica. En este libro no seguiremos esa tradición, dado que para el uso de la lógica en el desarrollo de programas la equivalencia tiene un rol fundamental. Como último ejemplo de la inadecuación del lenguaje para parafrasear la equivalencia, tomamos esta frase (verdadera) de [DS90] acerca de las capacidades visuales de Juan:

Juan ve bien si y solo si Juan es tuerto si y solo si Juan es ciego.

Dado que, como veremos más adelante, la equivalencia es asociativa, puede representarse esta proposición como $p \equiv q \equiv r$ ya que da lo mismo dónde se pongan los paréntesis. Dado que exactamente una de las tres proposiciones es cierta (p, q, r representan las proposiciones simples obvias), dejamos como ejercicio construir la tabla de verdad y comprobar que la proposición es verdadera (que en las filas donde exactamente una variable proposicional toma el valor $True$, el valor de la expresión booleana completa es también $True$).

2.8 Análisis de razonamientos

En la sección 2.1 se presentaron ejemplos de razonamientos en lenguaje natural. Ya estamos en condiciones de analizar la validez de algunas formas de razonamiento.

Hemos dicho que un razonamiento es válido si cada vez que todas las premisas sean verdaderas la conclusión también tiene que serlo. Una forma de analizar un razonamiento, entonces, es traducirlo a expresiones booleanas y analizar el caso en el cual todas las premisas son verdaderas. Si se tiene el siguiente razonamiento (ya traducido a expresiones booleanas):

$$\begin{array}{l} p_0 \\ \vdots \\ \underline{p_n} \\ q \end{array}$$

el mismo se considera válido si siempre que $p_0, \ldots, p_n$ sean todas verdaderas entonces también lo es q. Dada las tablas de verdad de la conjunción y de la implicación, esto ocurre si y solo si la siguiente fórmula es una tautología:

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

Por caso, el razonamiento del pelado que no es feliz (ejemplo 2.4) puede traducirse como

p: Tengo pelo.

q: Soy feliz.

$$\begin{array}{l} p \Rightarrow q \\ \underline{\neg q} \\ \neg p \end{array}$$

Se puede ahora comprobar la validez del razonamiento contruyendo la tabla de verdad asociada.

p	q	$p \Rightarrow q$	$\neg q$	$(p \Rightarrow q) \wedge \neg q$	$\neg p$	$(p \Rightarrow q) \wedge \neg q \Rightarrow \neg p$
$True$	$True$	$True$	$False$	$False$	$False$	$True$
$True$	$False$	$False$	$True$	$False$	$False$	$True$
$False$	$True$	$True$	$False$	$False$	$True$	$True$
$False$	$False$	$True$	$True$	$True$	$True$	$True$

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

De manera similar, puede verse que el segundo razonamiento capilar no es válido traduciéndolo a su forma simbólica

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

Y comprobando con la tabla de verdad asociada que no es una tautología.

p	q	$p \Rightarrow q$	$\neg p$	$(p \Rightarrow q) \wedge \neg p$	$\neg q$	$(p \Rightarrow q) \wedge \neg p \Rightarrow \neg q$
$True$	$True$	$True$	$False$	$False$	$False$	$True$
$True$	$False$	$False$	$False$	$False$	$True$	$True$
$False$	$True$	$True$	$True$	$True$	$False$	$False$
$False$	$False$	$True$	$True$	$True$	$True$	$True$

Más aún, mirando la tabla de verdad es posible saber cuándo las premisas son verdaderas y la conclusión es falsa. En este caso, cuando p es falsa y q es verdadera, o volviendo a nuestro ejemplo, cuando tenemos un pelado feliz.

Si bien este método para analizar razonamientos es efectivo para razonamientos con pocas variables proposicionales, a medida que la cantidad de variables crece, el tamaño de las tablas de verdad crece de manera exponencial (hay 2^n filas en una tabla de verdad con n variables proposicionales). Esta limitación práctica al uso de tablas de verdad no ocurre solo para analizar razonamientos, pero en este caso es más obvio el problema dado que en los razonamientos suelen ser necesarias una gran cantidad de variables proposicionales.

En los capítulos siguientes veremos métodos alternativos para resolver la validez de razonamientos y de expresiones booleanas en general, los cuales son a veces mucho más cortos que las tablas de verdad.

2.9 Resolución de acertijos lógicos

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

Ejemplo 2.7
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?

La solución de este problema es bastante simple. Supongamos que A es un pícaro. Luego la afirmación hecha es falsa, y por lo tanto ninguno de ellos podría ser un pícaro. Pero esto contradice la suposición acerca de A, por lo tanto A es un caballero. Ahora, dado que es un caballero, su afirmación es cierta. Pero entonces entre él y B hay al menos un pícaro. Ya habíamos visto que A solo podía ser un caballero. Por lo tanto B es un pícaro.

Ejemplo 2.8
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?

A no puede ser un caballero, dado que estaría diciendo que es un pícaro (y algo más), frase que no puede provenir de la boca de un caballero. Luego A es un pícaro, por lo cual su frase es falsa. Luego, dado que el es efectivamente un pícaro, es necesariamente falsa la otra mitad de la conjunción, o sea que B no lo es. Por lo tanto, B también es un pícaro.

2.10 Ejercicios

Ejercicio 2.2
Evaluar las siguientes expresiones en el estado $\{(p, True), (q, False), (r, False)\}$.

1. $(p \vee q) \wedge r$
2. $(p \wedge q) \vee r$
3. $p \vee (q \wedge r)$
4. $p \equiv (q \equiv r)$
5. $(p \equiv q) \equiv r$
6. $(p \Rightarrow q) \Rightarrow r$
7. $p \Rightarrow (q \Rightarrow r)$
8. $p \wedge q \Rightarrow r$

Ejercicio 2.3
Escribir las tablas de verdad para las siguientes expresiones, determinando los casos en los cuales sean tautologías o contradicciones.

1. $(p \vee q) \vee \neg q$
2. $(p \wedge q) \Leftarrow \neg q$
3. $(p \not\equiv q) \wedge (p \vee q)$
4. $p \equiv (q \equiv p)$
5. $(p \not\equiv p) \not\equiv p$
6. $(p \Rightarrow q) \Rightarrow p$
7. $\neg q \wedge \neg p \equiv (q \Rightarrow p) \Rightarrow q$

8. $(p \equiv p \vee \neg p) \Rightarrow p$

Ejercicio 2.4
En la isla de los pícaros y los caballeros, A dice 'O bien yo soy un pícaro o bien B es un caballero' ¿Qué son A y B?

Ejercicio 2.5
Traducir los siguientes razonamientos presentados en lenguaje natural al cálculo proposicional y analizar su validez.

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

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

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

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

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

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

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

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 estas 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 elementos:

- 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, este coincidirá con el definido en los capítulos precedentes, solo 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. Solo son fórmulas las construidas con las tres reglas precedentes.

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

Transitividad:	$\dfrac{P \equiv Q, Q \equiv R}{P \equiv R}$
Leibniz:	$\dfrac{P \equiv Q}{E(r := P) = E(r := Q)}$
Sustitución:	$\dfrac{P}{P(q := R)}$

donde P, Q, R y E son fórmulas booleanas arbitrarias. De aquí en adelante usaremos letras mayúsculas para denotar expresiones , diferenciándolas así de las variables.

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 1 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 al igual que lo hicimos en el capítulo 1.

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 este último.

Veremos a continuación una serie de leyes o axiomas y de teoremas que se demuestran a partir de estos. A veces se omiten las demostraciones, sobreentendiéndose que estas quedan como ejercicios para el lector. Se recomienda hacer estos ejercicios para desarrollar la habilidad de manipular formalmente el cálculo proposicional, 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 los siguientes axiomas:

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)
$p \equiv True \equiv p$

La asociatividad permite la omisión de paréntesis. Por ejemplo: el tercer axioma debería haberse escrito como $(p \equiv True) \equiv p$. 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

$$
\begin{array}{ll}
 & p \equiv p \\
\equiv & \{\ 3.3 \text{ aplicado a la primera } p\ \} \\
 & (p \equiv True) \equiv p
\end{array}
$$

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

$$
\begin{array}{ll}
 & E_0 \\
\equiv & \{ \text{ justificación de } E_0 \equiv E_1 \ \} \\
 & E_1 \\
 & \vdots \\
 & E_{n-1} \\
\equiv & \{ \text{ justificación de } E_{n-1} \equiv E_n \ \} \\
 & E_n
\end{array}
$$

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

$$
\begin{array}{ll}
 & True \\
\equiv & \{ \text{ reflexividad de la equivalencia } \} \\
 & E_0 \equiv E_0 \\
\equiv & \{ \text{ justificación de } E_0 \equiv E_1 \ \} \\
 & E_0 \equiv E_1 \\
 & \vdots \\
 & E_0 \equiv E_{n-1} \\
\equiv & \{ \text{ justificación de } E_{n-1} \equiv E_n \ \} \\
 & E_0 \equiv E_n
\end{array}
$$

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 $\neg$, es el operador unario que cumple los siguientes axiomas:

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
$False \equiv \neg True$

La negación tiene mayor precedencia que cualquier otro operador del cálculo. A partir de los axiomas se pueden demostrar los siguientes teoremas (se deja como ejercicios para el lector):

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

Teorema 3.10 (Contrapositiva)
$p \equiv False \equiv \neg p$

3.4 La discrepancia

La discrepancia es el operador que satisface el siguiente axioma:

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

La precedencia de la discrepancia es la misma que la de la equivalencia. A partir de este axioma se pueden demostrar los siguientes teoremas:

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

Notar que el teorema de asociatividad mutua permite eliminar paréntesis en expresiones que contienen equivalencias y discrepancias en el mismo nivel.

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).

$$
\begin{aligned}
& p \not\equiv False \\
\equiv\ & \{\ 3.11 \text{ con } q := false\ \} \\
& \neg(p \equiv False) \\
\equiv\ & \{\ 3.8\ \} \\
& \neg(p \equiv \neg True) \\
\equiv\ & \{\ 3.7\ \} \\
& \neg\neg(p \equiv True) \\
\equiv\ & \{\ \text{doble negación, } True \text{ neutro para } \equiv\ \} \\
& p
\end{aligned}
$$

Teorema 3.16 (Contrapositiva)
$p \equiv q \equiv \neg p \equiv \neg q$

Teorema 3.17 (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 $\vee$, es un operador binario de mayor precedencia que la equivalencia, es decir que $p \vee q \equiv r$ debe entenderse como $(p \vee q) \equiv r$. Este operador está definido por los siguientes axiomas:

Axioma 3.18 (Asociatividad)
$(p \vee q) \vee r \equiv p \vee (q \vee r)$

Axioma 3.19 (Conmutatividad)
$p \vee q \equiv q \vee p$

Axioma 3.20 (Idempotencia)
$p \vee p \equiv p$

Axioma 3.21 (Distributividad con la equivalencia)
$p \vee (q \equiv r) \equiv (p \vee q) \equiv (p \vee r)$

Axioma 3.22 (Tercero excluido)
$p \vee \neg p$

Demostraremos algunas propiedades de la disyunción.

Teorema 3.23
$p \vee (q \vee r) \equiv (p \vee q) \vee (p \vee r)$

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

$$
\begin{aligned}
& (p \vee q) \vee (p \vee r) \\
\equiv\ & \{ \text{ asociatividad de } \vee \ \} \\
& p \vee q \vee p \vee r \\
\equiv\ & \{ \text{ conmutatividad de } \vee \ \} \\
& p \vee p \vee q \vee r \\
\equiv\ & \{ \text{ idempotencia de } \vee \ \} \\
& p \vee q \vee r \\
\equiv\ & \{ \text{ asociatividad de } \vee \ \} \\
& p \vee (q \vee r)
\end{aligned}
$$

Teorema 3.24 (Elemento absorbente)
$p \vee True \equiv True$

Demostración

$$
\begin{aligned}
& p \vee True \\
\equiv\ & \{ \ p \equiv True \equiv p, \text{ interpretada como } (p \equiv p) \equiv True \ \} \\
& p \vee (q \equiv q) \\
\equiv\ & \{ \text{ distributividad de } \vee \text{ con respecto a } \equiv \ \} \\
& (p \vee q) \equiv (p \vee q) \\
\equiv\ & \{ \ p \equiv p \ \} \\
& True
\end{aligned}
$$

Teorema 3.25 (Elemento neutro)
$p \vee False \equiv p$

3.6 La conjunción

La conjunción, que denotaremos con el símbolo $\wedge$, es un operador binario con la misma precedencia que la disyunción, lo cual 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 definir conjunción.

Axioma 3.26 (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 3.27 (Asociatividad)
$p \wedge (q \wedge r) \equiv (p \wedge q) \wedge r$

Demostración

$$
\begin{aligned}
& p \wedge (q \wedge r) \\
\equiv\ & \{ \text{ regla dorada, interpretada como } p \wedge q \equiv (p \equiv q \equiv p \vee q) \ \} \\
& p \equiv (q \wedge r) \equiv p \vee (q \wedge r) \\
\equiv\ & \{ \text{ regla dorada, dos veces}^1 \ \} \\
& p \equiv q \equiv r \equiv q \vee r \equiv p \vee (q \equiv r \equiv q \vee r) \\
\equiv\ & \{ \text{ axioma 3.21 } \} \\
& p \equiv q \equiv r \equiv q \vee r \equiv p \vee q \equiv p \vee r \equiv p \vee q \vee r \\
\equiv\ & \{ \text{ axiomas 3.2, 3.1 } \} \\
& (p \equiv q \equiv p \vee q) \equiv r \equiv (p \vee r \equiv q \vee r \equiv p \vee q \vee r) \\
\equiv\ & \{ \text{ axioma 3.21 } \} \\
& (p \equiv q \equiv p \vee q) \equiv r \equiv (p \equiv q \equiv p \vee q) \vee r
\end{aligned}
$$

[1] De ahora en más no explicitaremos la interpretación, a menos que lo juzguemos necesario.

$\equiv$ { regla dorada }

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

$\equiv$ { regla dorada }

$(p \wedge q) \wedge r$

Teorema 3.28 (Conmutatividad)

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

Teorema 3.29 (Idempotencia)

$p \wedge p \equiv p$

Teorema 3.30 (Neutro)

$p \wedge True \equiv p$

Demostración

$p \wedge True$

$\equiv$ { regla dorada }

$p \equiv True \equiv p \vee True$

$\equiv$ { teorema 3.24 }

$p \equiv True \equiv True$

$\equiv$ { $p \equiv p$; $True$ 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: $p := False$.

Teorema 3.31
$p \vee q \equiv p \vee \neg q \equiv p$

Teorema 3.32 (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 (i)

$\neg p \wedge \neg q$
$\equiv$ { regla dorada }
$\neg p \equiv \neg q \equiv \neg p \vee \neg q$
$\equiv$ { teorema 3.31 }
$\neg p \equiv p \vee \neg q$
$\equiv$ { negación de $\equiv$ }
$\neg(p \equiv p \vee \neg q)$
$\equiv$ { teorema 3.31 }
$\neg(p \vee q)$

La demostración de (ii) es análoga y se deja como ejercicio para el lector.

Teorema 3.33 (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 $\Rightarrow$, es 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 3.34
$p \Rightarrow q \equiv p \vee q \equiv q$

Teorema 3.35
$p \Rightarrow p$

Demostración

$p \Rightarrow p$

$\equiv$ { axioma 3.34 }

$p \vee p \equiv p$

y este resultado es el axioma 3.20.

Teorema 3.36
$p \Rightarrow q \equiv \neg p \vee q$

Demostración

$p \Rightarrow q$

$\equiv$ { definición de $\Rightarrow$ }

$p \vee q \equiv q$

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

$p \vee q \equiv False \vee q$

$\equiv$ { 3.21 }

$(p \equiv False) \vee q$

$\equiv$ { $p \equiv False \equiv \neg p$ }

$\neg p \vee q$

Teorema 3.37
$\neg(p \Rightarrow q) \equiv p \wedge \neg q$

Teorema 3.38
$p \Rightarrow True$

Teorema 3.39
$p \wedge q \Rightarrow p$

Teorema 3.40
$p \Rightarrow p \vee q$

Teorema 3.41 (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 3.42 (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 la recíproca de la regla para nuestro lenguaje de expresiones booleanas y aritméticas es válida). El siguiente contraejemplo muestra que para el axioma no es este el caso.

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

$$E(z := True) = E(z := False)$$

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

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 $\Leftarrow$. Es un operador binario que tiene la misma precedencia que la implicación y que satisface:

Axioma 3.43
$p \Leftarrow q \equiv p \vee q \equiv p$

Puede ahora demostrarse fácilmente que

Teorema 3.44
$p \Leftarrow q \equiv q \Rightarrow p$

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.

Para fijar ideas, comenzaremos con una derivación matemática que involucra desigualdades además de igualdades [BvW08].

Ejemplo 3.2
Sean m y n dos enteros positivos tales que $m > n$ entonces $m^2 - n^2 > 3$.

Para demostrar este resultado, dado que m y n son enteros positivos se satisface que $m + n \geq 3$ (esto se deduce de que $n \geq 1$ y $m \geq 2$). Por otro lado, es obvio que $m - n \geq 1$. Calculemos ahora

$$
\begin{array}{ll}
 & m^2 - n^2 \\
= & \{\ \text{diferencia de cuadrados}\ \} \\
 & (m+n) * (m-n) \\
\geq & \{\ \text{propiedades de } m \text{ y } n\ \} \\
 & 3 * 1 \\
= & \{\ \text{aritmética}\ \} \\
 & 3
\end{array}
$$

En el ejemplo, se aplica en uno de los pasos una desigualdad. Dado que la igualdad la implica y que es transitiva el resultado final se deduce de esta demostración. Hay que tener en cuenta, sin embrago, que el reemplazo de iguales por iguales, la regla de Leibniz, no es siempre válida para las desigualdades. En este caso sí, dado

que para enteros positivos la multiplicación es monótona (no es el caso para enteros negativos). Esto implica que cuando se extiende el formato con desigualdades hay que ser mucho más cuidadoso al realizar un paso de demostración.

La principal generalización que realizaremos es permitir que en las demostraciones dos pasos se conecten no solo 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 cómo transformar una demostración generalizada en una pura. No se probará este metateorema dado que su demostración, si bien no es difícil, es bastante larga y engorrosa. Se ilustrará con un ejemplo cómo puede realizarse dicha transformación.

Ejemplo 3.3
[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):

$$
\begin{array}{ll}
 & True \\
\equiv & \{ \text{ transitividad } \} \\
 & (p \equiv q) \wedge (q \Rightarrow r) \Rightarrow (p \Rightarrow r) \\
\equiv & \{ \text{ razón por la cual } p \equiv q \ \} \\
 & True \wedge (q \Rightarrow r) \Rightarrow (p \Rightarrow r) \\
\equiv & \{ \text{ razón por la cual } q \Rightarrow r \ \} \\
 & True \wedge True \Rightarrow (p \Rightarrow r) \\
\equiv & \{ \text{ idempotencia, } True \text{ es neutro a izquierda de } \Rightarrow \ \} \\
 & (p \Rightarrow r)
\end{array}
$$

Hay que tener cuidado con las demostraciones usando implicaciones ya que estas no son preservadas en conextos arbitrarios, como sí es el caso para la equivalencia (regla de Leibniz). Por ejemplo, de $p \Rightarrow q$ no se deduce ni $(p \Rightarrow r) \Rightarrow (q \Rightarrow r)$ ni tampoco $\neg p \Rightarrow \neg q$.

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 este capítulo.

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

$$p \vee False \equiv p$$

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

$$p \wedge False \equiv False$$

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:

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

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 nos permite construir especificaciones y programas. Si bien la informática ha dado un gran impulso para el desarrollo de la lógica, sobre todo en su uso 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. Por último veremos un ejemplo de uso del estilo de cálculo proposicional para problemas matemáticos. Las definiciones de piso y techo y varias de sus propiedades están inspiradas en el libro de Roland Backhouse [Bac03]. La presentación del máximo y mínimo usando propiedades universales según nuestro conocimiento apareció por primera vez en el artículo de Wim Feijen [Fei90].

4.1 Análisis de argumentaciones

En el capítulo 2 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 razonamientos con dos premisas como los presentados en ese capítulo el método es adecuado, dado que el número de variables proposicionales es muy pequeño. 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 consideraremos un razonamiento con seis variables proposicionales, lo cual hubiera dado lugar a una tabla de verdad con 64 filas.

Ejemplo 4.1
Considérese 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 solo 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:

$$
\begin{array}{ll}
 & \neg d \\
\Leftarrow & \{ \text{ premisa 5, contrarecíproca (ejercicio 3.9) } \} \\
 & \neg(\neg m \wedge o) \\
\equiv & \{ \text{ de Morgan } \} \\
 & m \vee \neg o \\
\Leftarrow & \{ \text{ premisa 1, 2, monotonía dos veces } \} \\
 & \neg q \vee \neg c \\
\Leftarrow & \{ \text{ premisa 3 } \} \\
 & e \\
\equiv & \{ \text{ premisa 4 } \} \\
 & True
\end{array}
$$

Hemos demostrado entonces que asumiendo que todas las premisas son válidas, $True \Rightarrow \neg d$, lo cual es lo mismo que afirmar $\neg d$ dado que $True$ es neutro a izquierda de una implicación (como ya se demostró en el capítulo anterior).

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 construida (tiene solo 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:

$$
\begin{array}{ll}
 & \neg l \vee \neg i \\
\Leftarrow & \{ \text{ premisa 2, contrarecíproca, monotonía } \} \\
 & \neg l \vee \neg c \\
\Leftarrow & \{ \text{ premisa 1, contrarecíproca, monotonía } \} \\
 & \neg o \vee \neg c \\
\equiv & \{ \text{ premisa 3 } \} \\
 & True
\end{array}
$$

Hemos usado la consecuencia en el contexto de una disyunción en este ejemplo, lo cual es válido dado que la disyunción es monótona (demostrado en el capítulo previo).

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 2.9. 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 de caballeros y pícaros. 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 solo 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 de la isla de caballeros y pícaros, 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 solo si A es un caballero, nos queda como dato del problema que

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

Manipulando esta expresión obtenemos

$$\begin{aligned}
& a \equiv \neg a \wedge b \\
\equiv & \ \{ \text{ regla dorada } \} \\
& a \equiv \neg a \equiv b \equiv \neg a \vee b \\
\equiv & \ \{ \text{ negación y equivalencia } \} \\
& False \equiv b \equiv \neg a \vee b
\end{aligned}$$

$\equiv$ { $False$ 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 La función piso

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

Definición 4.1
Dado un número real x se define la función *piso* (o parte entera) aplicada a x como el entero que satisface, para todo entero n, la siguiente propiedad:

$$\frac{\lfloor\ \rfloor : \mathbb{R} \mapsto Int}{n \leq \lfloor x \rfloor \equiv n \leq x}$$

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 propiedad 4.1.

Propiedad 4.2
Instanciando n en la definición de piso con el entero $\lfloor x \rfloor$, obtenemos la siguiente propiedad:

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

Propiedad 4.3
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 4.4
De las dos propiedades anteriores se sigue que para todo entero n

$$\lfloor n \rfloor = n$$

Propiedad 4.5
Podemos usar también la propiedad contrapositiva de la equivalencia (teorema 3.16) y el hecho que $\neg p \leq q \equiv q < p$ para obtener la siguiente propiedad para todo n entero:

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

Esta propiedad puede pensarse como una definición alternativa de la función piso.

Propiedad 4.6
Instanciando n con el entero $\lfloor x \rfloor + 1$ en la propiedad anterior, obtenemos la siguiente propiedad:

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

Propiedad 4.7
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

$$\left| \begin{array}{l} \lfloor\ \rfloor : \mathbb{R} \mapsto Int \\ \hline n = \lfloor x \rfloor \equiv n \leq x < n + 1 \end{array} \right.$$

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

Lema 4.8
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

$$\begin{array}{ll} & \lfloor x \rfloor \leq \lfloor y \rfloor \\ \equiv & \{\ \text{definición de piso}\ \} \\ & \lfloor x \rfloor \leq y \\ \Leftarrow & \{\ \text{transitividad}\ \} \\ & \lfloor x \rfloor \leq x \wedge x \leq y \\ \equiv & \{\ \text{propiedad 4.2}\ \} \\ & x \leq y \end{array}$$

La aplicación de la transitividad parece sacada de una galera, pero si se tiene en vista la meta de la demostración (el antecedente $x \leq y$), su introducción se vuelve natural.

4.4 Igualdad indirecta

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

Teorema 4.9

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

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

Observación: 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: 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 4.5
Como aplicación de la regla de igualdad indirecta resolveremos una propiedad sencilla 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

$$\begin{array}{ll} & k \leq \lfloor x + n \rfloor \\ \equiv & \{ \text{ definición de piso } \} \\ & k \leq x + n \end{array}$$

$\equiv$ { aritmética }

$k - n \leq x$

$\equiv$ { definición de piso }

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

$\equiv$ { aritmética }

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

4.5 La función techo

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

Definición 4.10
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:

$$\frac{\lceil\ \rceil : \mathbb{R} \mapsto Int}{\lceil x \rceil \leq n \equiv x \leq n}$$

La función techo tiene propiedades duales a la función piso (ver ejercicio 4.3)

Ejemplo 4.6 (Redondeo [Bac03])
En algunos lenguajes de programación solo se dispone de la función piso y en algunos casos hace falta usar también la función techo (por ejemplo para redondear al entero más cercano). Resolveremos el problema para números racionales. Es decir, dados dos enteros a y b se propone encontrar enteros p y q tales que

$$\left\lfloor \frac{p}{q} \right\rfloor = \left\lceil \frac{a}{b} \right\rceil$$

Antes de seguir leyendo, recomendamos a los lectores intentar resolver el problema.

Para encontrar de manera sistemática los enteros p y q, procederemos usando la igualdad indirecta. Tomemos un k entero arbitrario y partiendo de la desigualdad

$$k \leq \left\lceil \frac{a}{b} \right\rceil$$

intentaremos llegar a través de equivalencias lógicas a una desigualdad similar pero donde la expresión de la derecha sea una aplicación de la función piso en lugar de techo. Por la regla de igualdad indirecta, podremos entonces encontrar p y q con la propiedad deseada.

$$
\begin{array}{ll}
& k \leq \lceil \frac{a}{b} \rceil \\
\equiv & \{ \text{ desigualdad de enteros } \} \\
& k - 1 < \lceil \frac{a}{b} \rceil \\
\equiv & \{ \text{ definición de techo, contrapositiva } \} \\
& k - 1 < \frac{a}{b} \\
\equiv & \{ \text{ aritmética, asumiendo } 0 < b \} \\
& b * (k - 1) < a \\
\equiv & \{ \text{ desigualdad de enteros } \} \\
& b * (k - 1) + 1 \leq a \\
\equiv & \{ \text{ aritmética, asumiendo } 0 < b \} \\
& k \leq \frac{a+b-1}{b} \\
\equiv & \{ \text{ definición de piso } \} \\
& k \leq \lfloor \frac{a+b-1}{b} \rfloor
\end{array}
$$

De esta manera conseguimos una definición de la función techo en términos de la función piso para números racionales. Si bien es requerido que el denominador sea positivo, esto no es un problema dado que todo racional puede escribirse de esa manera sin mayores inconvenientes.

4.6 Máximo y mínimo

En esta sección presentamos una definición para el máximo (y dualmente para el mínimo) en un estilo similar al usado para las funciones piso y techo. El tipo de propiedad usado para definir estos operadores es conocido como propiedad universal, e involucra la existencia y unicidad de cierto operador. En la sección 1.7 introdujimos axiomas para el máximo y demostramos propiedades útiles en un estilo ecuacional más cómodo que los análisis por casos. Aquí demostraremos (o propondremos como ejercicios) que todos esos axiomas son consecuencias de la definición universal del máximo.

Definición 4.11
Dados dos números a y b se define el operador de *máximo* de ambos como el número que satisface, para todo número n del mismo tipo que a y b, la siguiente propiedad:

$$
\begin{array}{|l}
\hline
\max \ : Num \times Num \mapsto Num \\
\hline
a \max b \leq n \equiv a \leq n \wedge b \leq n \\
\hline
\end{array}
$$

Los axiomas presentados en 1.7 pueden todos derivarse usando esta regla (ejercicio 4.8).

Demostraremos una propiedad esperable del máximo la cual no puede derivarse fácilmente a partir de los axiomas:

Propiedad 4.12

$$x \max y = x \vee x \max y = y$$

$$
\begin{array}{ll}
 & x \max y = x \vee x \max y = y \\
\equiv & \{ \text{ antisimetría de } \leq \} \\
 & (x \max y \leq x \wedge x \leq x \max y) \vee (x \max y \leq y \wedge y \leq x \max y) \\
\equiv & \{ \text{ axioma 1.7 } \} \\
 & (x \max y \leq x \wedge True) \vee (x \max y \leq y \wedge True) \\
\equiv & \{ \text{ cálculo proposicional, def. de máximo } \} \\
 & (x \leq x \wedge y \leq x) \vee (x \leq y \wedge y \leq y) \\
\equiv & \{ \text{ reflexividad de } \leq, \text{ cálculo proposicional } \} \\
 & x \leq y \vee y \leq x \\
\equiv & \{ \text{ tricotomía } \} \\
 & True
\end{array}
$$

4.7 Ejercicios

Ejercicio 4.1
Analizar los razonamientos del ejercicio 2.5 usando las herramientas introducidas en esta sección.

Ejercicio 4.2
Utilizando el cálculo proposicional, resolver los siguientes acertijos, donde A y B son habitantes de la isla de los caballeros y los pícaros.

1. Nos encontramos con dos personas, A y B. A dice 'Al menos uno de nosotros es un pícaro ¿Qué 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' ¿Qué son A y B?
4. Dos personas 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.

¿Qué es C?

5. A dice 'Si soy un caballero entonces B también lo es' ¿Qué 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.

   ¿Qué es A?

Ejercicio 4.3
Enunciar y demostrar las propiedades que satisface la función techo análogas a las presentadas para la función piso en la sección 4.3.

Ejercicio 4.4
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 4.5
¿Qué está mal en la siguiente demostración (n, k enteros, $0 < n$ y x real)?

$$
\begin{array}{ll}
 & k \leq \lfloor n * x \rfloor \\
\equiv & \{ \text{ definición de piso } \} \\
 & k \leq n * x \\
\equiv & \{ \text{ aritmética } \} \\
 & \frac{k}{n} \leq x \\
\equiv & \{ \text{ definición de piso } \} \\
 & \frac{k}{n} \leq \lfloor x \rfloor \\
\equiv & \{ \text{ aritmética } \} \\
 & k \leq n * \lfloor x \rfloor
\end{array}
$$

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

Ejercicio 4.6
¿Qué está mal en la siguiente demostración (m, n, k enteros, $n \neq 0$)?

$$
\begin{aligned}
& \lceil \tfrac{m}{n} \rceil \leq k \\
\equiv & \ \{ \text{ definición de techo } \} \\
& \tfrac{m}{n} \leq k \\
\equiv & \ \{ \text{ aritmética } \} \\
& \tfrac{m}{n} < k + 1 \\
\equiv & \ \{ \text{ definición de piso, contrapositiva } \} \\
& \lfloor \tfrac{m}{n} \rfloor < k + 1 \\
\equiv & \ \{ \text{ aritmética } \} \\
& \lfloor \tfrac{m}{n} \rfloor \leq k
\end{aligned}
$$

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

Ejercicio 4.7
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 $\lfloor a \rfloor + \lfloor b \rfloor \leq \lfloor a + b \rfloor$

Ejercicio 4.8
Demostrar los axiomas del máximo presentados en 1.7 a partir de la definición 4.6.

Ejercicio 4.9
Demostrar que

$$\lfloor x \text{ min } y \rfloor = \lfloor x \rfloor \text{ min } \lfloor y \rfloor$$

Capítulo 5

Cálculo de predicados

> (a) Socrates is a man. (b) All men are mortal. (c) All men are Socrates. That means all men are homosexuals.
>
> Woody Allen:*Love and Death*

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 2.1. Uno ejemplo típico de silogismo es el siguiente.

Las vacas son mamíferos.
Hay animales que no son mamíferos.

Hay animales que no son vacas.

Si quisiéramos analizar este razonamiento usando el cálculo proposicional nos quedaría claramente una forma inválida; cada premisa y la conclusión serían proposiciones elementales distintas,

vm**:** Las vacas son mamíferos.

am**:** Hay animales que no son mamíferos.

av**:** Hay animales que no son vacas.

y el razonamiento sería $vm \wedge am \Rightarrow av$, el cual no es válido dado que puede tener premisas verdaderas y conclusión falsa.

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

5.1 Predicados

El cálculo proposicional desarrollado en el capítulo 3 nos permitió razonar sobre una clase restringida de expresiones booleanas denominadas *fórmulas proposicionales*. Estas fueron construidas a partir de constantes ($True$ y $False$), variables y operadores booleanos ($\equiv$, $\neg$, $\vee$, etc.). Por lo tanto, la expresividad de la lógica estuvo restringida a sentencias que podían ser modeladas con estas fórmulas. Ya vimos algunos ejemplos de expresiones booleanas más generales en el capítulo 1. Allí, se utilizaron expresiones booleanas conjuntamente con expresiones aritméticas.

En este capítulo desarrollaremos el *cálculo de predicados*. Este nos permitirá razonar sobre una clase más extensa y expresiva de expresiones booleanas. De manera general, una fórmula del cálculo de predicados es una fórmula proposicional en donde algunas variables booleanas pueden ser reemplazadas por:

- *Predicados*, los cuales son funciones booleanas (el concepto de función fue introducido en 1.7) cuyos argumentos pueden ser expresiones no booleanas. Ejemplos de predicados son '$igual.\,(x-y).\,4$' y '$mayor.\,2+x.\,y$'. Los nombres de función (por ejemplo '$igual$', '$mayor$') son llamados *símbolos de predicados*. A menudo también se usa en los predicados la notación infija como '$a < b$' o 'a max b'.

  Los argumentos de los predicados pueden ser expresiones de distintos tipos. En el capítulo 1 vimos esto para el caso particular de las expresiones aritméticas. De esta forma los argumentos pueden tener variables no necesariamente booleanas. Estos argumentos se denominan *términos*.

- *Cuantificación universal* y *existencial*, como se verá más adelante en este capítulo.

Dado un conjunto de símbolos de predicado, el *cálculo puro de predicados* tendrá como axiomas y reglas de inferencia las del cálculo proposicional (ya vistos en el capítulo 3) y las de las cuantificaciones. En este cálculo, los símbolos de predicado no estarán interpretados (excepto para la igualdad $=$). Esto quiere decir que se hará abstracción de su significado, y la lógica no brindará reglas específicas para manipularlos. Por ejemplo, para razonar sobre la sentencia dada en la introducción, deberemos tener los símbolos de predicado 'es_vaca' y '$es\_mamifero$'. Una fórmula posible será entonces '$es\_vaca.\,m \Rightarrow es\_mamifero.\,m$', pero no se darán las reglas para deducir que esta fórmula es verdadera. A pesar de esta restricción, el cálculo nos permitirá razonar sobre sentencias, sin tener en cuenta el significado de los símbolos de predicados.

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.

5.2 El cuantificador universal

Enumeramos a continuación los axiomas que definen la cuantificación universal. Esta surge cuando se formaliza la noción de "para todo x"; por ejemplo, cuando uno dice en matemática

$2 * n$ es par, con n entero.

lo que en realidad está diciendo es

Para todos los n enteros, $2 * n$ es par.

Esta misma afirmación se escribe usando nuestra notación de la siguiente manera:

$$\langle \forall n \ : n \in \mathbb{Z} : \ \text{par} \rangle \, (2 * n) \tag{5.1}$$

donde "$n \in \mathbb{Z}$" es $True$ si y solo si n es entero, y "par.n" es $True$ si y solo si n es un entero par.

Lo que se ha indicado aquí es el significado ("semántica") que modela el comportamiento formal ("sintáctico") de la cuantificación universal. Una vez que se presentan los axiomas, uno puede trabajar sin preocuparse por el significado de cada expresión utilizada, siempre que se manipule usando las reglas *exactamente* como han sido especificadas.

El formato general de una cuantificación universal es el siguiente:

$$\langle \forall \text{variable de cuantificación} \ : \text{rango} : \ \text{término} \rangle$$

donde "rango" y "término" son predicados, los cuales toman como argumento a n es decir, que pueden devolver solo los valores $True$ o $False$ (ejemplo: "$n \leq 2$" es una tal expresión; para cada n es verdadera o falsa, en cambio "n^2" **no** lo es). Si en el rango figura el predicado constante $True$, lo omitiremos, lo cual queda expresado en el siguiente axioma:

Axioma 5.1 (Rango $True$)
$\langle \forall x \ :: \ f.x \rangle \equiv \langle \forall x \ : True : \ f.x \rangle$

Definición 5.2 (Variable libre)
Se define inductivamente cuándo una variable está libre en una expresión (el cuantificador existencial se define más adelante).

- 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 no aparece en la secuencia de variables x, entonces lo está también en $\langle \forall x : F : E \rangle$, en $\langle \forall x : E : F \rangle$, en $\langle \exists x : F : E \rangle$ y en $\langle \exists x : E : F \rangle$.

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

Uno de los axiomas más importantes es el que da significado al rango, el cual llamaremos *intercambio entre rango y término*:

Axioma 5.3 (Intercambio entre rango y término)
$\langle \forall x : r.x : f.x \rangle \equiv \langle \forall x :: r.x \Rightarrow f.x \rangle$

Usando el ejemplo de más arriba, este axioma nos dice que es lo mismo escribir "Para todos los n enteros, $2*n$ es par" que "Para todos los n, si n es entero entonces $2 * n$ es par".

Dado que el cuantificador universal es una generalización de la conjunción, algunas propiedades de esta pasan al primero, por ejemplo, la conmutatividad, la asociatividad y la distributividad de $\vee$ con respecto a $\wedge$:

Axioma 5.4 (Regla del término)
$\langle \forall x :: f.x \rangle \wedge \langle \forall x :: g.x \rangle \equiv \langle \forall x :: f.x \wedge g.x \rangle$

Axioma 5.5 (Distributividad de $\vee$ con $\forall$)
$X \vee \langle \forall x :: f.x \rangle \equiv \langle \forall x :: X \vee f.x \rangle$, siempre que x no ocurra en X.

Si bien estos dos axiomas han sido enunciados para el caso que el rango sea exactamente $True$, podemos demostrar que el segundo vale en general para cualquier rango, es decir:

Teorema 5.6
$X \vee \langle \forall x : r.x : f.x \rangle \equiv \langle \forall x : r.x : X \vee f.x \rangle$, donde x no ocurre en $FV.X$

Demostración

$X \vee \langle \forall x : r.x : f.x \rangle$

$\equiv$ { Intercambio entre rango y término }

$X \vee \langle \forall x :: r.x \Rightarrow f.x \rangle$

$\equiv$ { Caracterización de $\Rightarrow$: $p \Rightarrow q \equiv \neg p \vee q$ aplicado al término }

$X \vee \langle \forall x :: \neg r.x \vee f.x \rangle$

$\equiv$ { Distributividad de $\forall$ con $\vee$ (x no ocurre en X) }

$\langle \forall x :: X \vee \neg r.x \vee f.x \rangle$

$\equiv$ { Conmutatividad de $\vee$, Caracterización de $\Rightarrow$ }

$\langle \forall x :: r.x \Rightarrow X \vee f.x \rangle$

$\equiv$ { Intercambio entre rango y término }

$\langle \forall x : r.x : X \vee f.x \rangle$

El teorema que sigue formaliza la idea de que cuantificar universalmente sobre una unión de elementos es equivalente a cuantificar sobre cada uno de los conjuntos (no importa si no son disjuntos).

Teorema 5.7 (Partición de rango)

$\langle \forall x : r.x : f.x \rangle \wedge \langle \forall x : s.x : f.x \rangle \equiv \langle \forall x : r.x \vee s.x : f.x \rangle$

Demostración

$\langle \forall x : r.x : f.x \rangle \wedge \langle \forall x : s.x : f.x \rangle$

$\equiv$ { Intercambio entre rango y término, caracterización de $\Rightarrow$, dos veces }

$\langle \forall x :: \neg r.x \vee f.x \rangle \wedge \langle \forall x :: \neg s.x \vee f.x \rangle$

$\equiv$ { Distributividad de $\forall$ con $\wedge$ }

$\langle \forall x :: (\neg r.x \vee f.x) \wedge (\neg s.x \vee f.x) \rangle$

$\equiv$ { Distributividad de $\vee$ con $\wedge$ [1]}

$\langle \forall x :: (\neg r.x \wedge \neg s.x) \vee f.x \rangle$

$\equiv$ { de Morgan }

$\langle \forall x :: \neg(r.x \vee s.x) \vee f.x \rangle$

$\equiv$ { Caracterización de $\Rightarrow$ e intercambio entre rango y término }

$\langle \forall x : r.x \vee s.x : f.x \rangle$

Aplicando este teorema a la cuantificación universal (5.1), podemos tomar $r.n$ igual a "n es un entero positivo o cero", $s.n$ igual a "n es un entero negativo" y obtenemos la equivalencia entre

> Para todos los n enteros, $2 * n$ es par.

y

> Para todos los n enteros positivos o cero $2 * n$ es par **y** para todos los n enteros negativos $2 * n$ es par.

Esto funciona ya que decir $r.n \vee s.n$ ("n es un entero positivo o cero o n es un entero negativo") es una forma de decir "n es un entero".

El axioma de *Rango unitario* dice que si hay un *único* x posible, digamos X, decir que algo vale para todos los x equivale a decir que vale para ese único valor:

[1] Notar que indicamos quién distribuye con quién.

Axioma 5.8 (Rango unitario)
$\langle \forall\, x \ : x = X : \ f.x \,\rangle \equiv f.X$
Por ejemplo, "para todo n tal que n es 3, $2 * n$ es par" es equivalente a "$2 * 3$ es par".

Cuando uno dice que $f.x$ vale para todos los x, debería valer en particular para cualquier x arbitrario que uno tome, verbigracia, $x = Y$. Esta idea se resume en el siguiente teorema, que denominamos *Regla de Instanciación.*

Teorema 5.9 (Instanciación)
$\langle \forall\, x \ :: \ f.x \,\rangle \Rightarrow f.Y$

Demostración
Usando la definición dual de $\Rightarrow$, el teorema dice $\langle \forall\, x \ :: \ f.x \,\rangle \wedge f.Y \equiv \langle \forall\, x \ :: \ f.x \,\rangle$. Probemos esto último:

$$
\begin{array}{ll}
 & \langle \forall\, x \ :: \ f.x \,\rangle \\
\equiv & \{ \text{ Rango } True \ \} \\
 & \langle \forall\, x \ : True : \ f.x \,\rangle \\
\equiv & \{ \text{ Absorbente del } \vee \ \} \\
 & \langle \forall\, x \ : True \ \vee \ x = Y : \ f.x \,\rangle \\
\equiv & \{ \text{ Partición de rango } \} \\
 & \langle \forall\, x \ : True : \ f.x \,\rangle \wedge \langle \forall\, x \ : x = Y : \ f.x \,\rangle \\
\equiv & \{ \text{ Rango } True \text{ y Rango unitario (notar que valen las hipótesis) } \} \\
 & \langle \forall\, x \ :: \ f.x \,\rangle \wedge f.Y
\end{array}
$$

En algunas ocasiones interviene más de una cuantificación. Nuestro primer ejemplo no sirve, pero podemos reemplazarlo por otro igualmente simple: decir que "para todos los m se da que para todo n, $2 * (m - n)$ es par" es lo mismo que decir "para todos los n se da que para todo m, $2 * (m - n)$ es par". El siguiente axioma formaliza y generaliza este ejemplo.

Axioma 5.10 (Intercambio de cuantificadores)
$\langle \forall\, x \ :: \ \langle \forall\, y \ :: \ f.x.y \,\rangle \,\rangle \equiv \langle \forall\, y \ :: \ \langle \forall\, x \ :: \ f.x.y \,\rangle \,\rangle$

En virtud de que en castellano cualquiera de las proposiciones anteriores se resumen diciendo "para todo m y n, $2*(m-n)$ es par", de ahora en más escribiremos los cuantificadores "anidados" del siguiente modo:

Axioma 5.11 (Anidado)
$\langle \forall\, x, y \ :: \ f.x.y \,\rangle \equiv \langle \forall\, x \ :: \ \langle \forall\, y \ :: \ f.x.y \,\rangle \,\rangle$

El teorema que sigue recibe el nombre de *Cambio de variable*:

Teorema 5.12 (Cambio de variable)
$\langle\, \forall\, x \,:\, r.x :\; f.x \,\rangle \equiv \langle\, \forall\, y \,:\, r.y :\; f.y \,\rangle$, si y no ocurre en $f.x$ o en $r.x$ y x no ocurre en $f.y$ o en $r.y$.

Demostración

$\langle\, \forall\, y \,:\, r.y :\; f.y \,\rangle$

$\equiv$ { Rango unitario, aplicado al término tomando $Y := y$ }

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

$\equiv$ { Intercambio entre rango y término en $\forall y$, caracterización de $\Rightarrow$ }

$\langle\, \forall\, y \,::\, \neg r.y \vee \langle\, \forall\, x \,:\, x = y :\; f.x \,\rangle \,\rangle$

$\equiv$ { Distributividad de $\vee$ con $\forall$, x no ocurre en $\neg r.y$ }

$\langle\, \forall\, y \,::\, \langle\, \forall\, x \,:\, x = y :\; \neg r.y \vee f.x \,\rangle \,\rangle$

$\equiv$ { Intercambio entre rango y término en $\forall x$, caracterización de $\Rightarrow$ }

$\langle\, \forall\, y \,::\, \langle\, \forall\, x \,::\, \neg x = y \vee \neg r.y \vee f.x \,\rangle \,\rangle$

$\equiv$ { Intercambio de cuantificadores }

$\langle\, \forall\, x \,::\, \langle\, \forall\, y \,::\, \neg x = y \vee \neg r.y \vee f.x \,\rangle \,\rangle$

$\equiv$ { Caracterización de $\Rightarrow$, intercambio entre rango y término en $\forall x$ }

$\langle\, \forall\, x \,::\, \langle\, \forall\, y \,:\, x = y :\; \neg r.y \vee f.x \,\rangle \,\rangle$

$\equiv$ { Rango unitario (del $\forall y$, obvio –con el otro no se puede–) }

$\langle\, \forall\, x \,::\, \neg r.x \vee f.x \,\rangle$

$\equiv$ { Caracterización de $\Rightarrow$, intercambio entre rango y término en $\forall x$ }

$\langle\, \forall\, x \,:\, r.x :\; f.x \,\rangle$

Aplicado a nuestro primer ejemplo, el teorema afirma que es exactamente lo mismo decir "Para todos los n enteros, $2 * n$ es par" que decir "Para todos los x enteros, $2 * x$ es par", y asimismo reemplazando x por cualquier otra variable. Por esa razón, a las variables de cuantificación se las denomina "variables bobas", porque lo único que hacen es indicar qué se cuantifica, pero (por ejemplo) no pueden ser reemplazadas por un valor. No tiene sentido decir

$$\langle\, \forall\, 10 \,:\, \textbf{enteros}.10 :\; \text{par}.10 \,\rangle$$

que se obtiene reemplazando n por 10 en (5.1) ("Para todo 10" ya suena bastante ridículo). Otro ejemplo de variable boba es el dado por las variables de derivación: para decir "integral de f", uno escribe

$$\int f(x)dx \qquad \int f(y)dy$$

y es lo mismo, pero no tiene sentido decir

$$\int f(5)d5.$$

5.3 El cuantificador existencial

La cuantificación existencial está caracterizada por un solo axioma. A partir de este (el cual lo relaciona con el cuantificador universal) pueden deducirse como teoremas todas sus propiedades.

El cuantificador existencial afirma que hay algún individuo en un rango dado el cual satisface una propiedad determinada. Esto contrasta con el cuantificador universal en dos aspectos: por un lado, alcanza con que un individuo satisfaga la propiedad para que sea válido; por otro, se afirma la existencia de al menos un individuo, algo que la cuantificación universal no hace. Esto se refleja en el teorema que muestra que una cuantificación existencial sobre un conjunto vacío es falsa.

El formato general de una cuantificación existencial es análogo al de la cuantificación universal:

$$\langle\, \exists \,\text{variable de cuantificación}\ :\text{rango}:\ \text{término}\,\rangle$$

donde "rango" y "término" son predicados los cuales toman como argumento a n.

El axioma para el cuantficador existencial el cual lo relaciona con el cuantificador universal, es el siguiente:

Axioma 5.13 (Cuantificación existencial)
$\langle\, \exists x\ : r.x :\ f.x\,\rangle \equiv \neg\langle\, \forall x\ : r.x :\ \neg f.x\,\rangle$

Los siguientes teoremas son análogos a los axiomas o teoremas del cuantificador universal. Las demostraciones se dejan para el lector.

Uno de los teoremas más importantes es el que da significado al rango. La diferencia con el cuantificador universal es consistente con la interpretación del cuantificador.

Teorema 5.14 (Intercambio entre rango y término)
$\langle\, \exists x\ : r.x :\ f.x\,\rangle \equiv \langle\, \exists x\ ::\ r.x \wedge f.x\,\rangle$

Dado que el cuantificador existencial es una generalización de la disyunción, algunas propiedades de esta pasan al primero, por ejemplo, la conmutatividad, la asociatividad y la distributividad de $\wedge$ con respecto a $\vee$:

Teorema 5.15 (Regla del término)
$\langle\, \exists x\ ::\ f.x\,\rangle \vee \langle\, \exists x\ ::\ g.x\,\rangle \equiv \langle\, \exists x\ ::\ f.x \vee g.x\,\rangle$

Teorema 5.16 (Distributividad de $\wedge$ con $\exists$)
$X \wedge \langle\, \exists x\ ::\ f.x\,\rangle \equiv \langle\, \exists x\ ::\ X \wedge f.x\,\rangle$, siempre que x no ocurra en $FV.X$.

El teorema que sigue es la versión de la partición de rango para el caso existencial.

Teorema 5.17 (Partición de rango)
$\langle\, \exists x\ : r.x :\ f.x\,\rangle \vee \langle\, \exists x\ : s.x :\ f.x\,\rangle \equiv \langle\, \exists x\ : r.x \vee s.x :\ f.x\,\rangle$

5.4 Propiedades de las cuantificaciones universal y existencial

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

Teorema 5.18

Siempre que $i \notin FV.P$ y que el rango no sea vacío (es decir que $\langle \exists i :: R \rangle$) 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 5.19

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

Teorema 5.20 (Fortalecimiento por término)

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

Teorema 5.21 (Fortalecimiento por rango)

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

Teorema 5.22 (Monotonía)

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

Metateorema 5.23

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

Demostración

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

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

$$\begin{array}{ll} & P \\ \equiv & \{ \text{ Razón 1 } \} \\ & P_1 \\ & \vdots \\ & P_n \end{array}$$

$\equiv$ { Razón $n+1$ }
$True$

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 $n+1$ }
$\langle \forall i :: True \rangle$
$\equiv$ { Término constante }
$True$

Teorema 5.24
Siempre que $i \notin FV.P$ y que el rango no sea vacío (es decir que $\langle \exists i :: R \rangle$) vale la siguiente ley distributiva

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

Teorema 5.25 (Debilitamiento por término)

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

Teorema 5.26 (Debilitamiento por rango)

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

Teorema 5.27 (Monotonía)

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

Teorema 5.28 (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 5.29 (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.

Demostración

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

Aplicando el metateorema 5.23 de la cuantificación universal, vemos que la última línea es un teorema si y solo si $P(i := k) \Rightarrow Q$ lo es.

5.5 Aplicaciones del cálculo de predicados

Una de las aplicaciones tradicionales del cálculo de predicados es el análisis de razonamientos en lenguaje natural. Esto extiende el análisis realizado en el capítulo 4 a casos para los cuales el razonamiento proposicional es insuficiente. No solo usaremos letras proposicionales para representar proposiciones atómicas, sino que analizaremos la estructura de estas proposiciones usando predicados y cuantificadores los cuales permiten expresar relaciones que exceden al cálculo proposicional.

Tomemos primero un ejemplo que puebla casi todos los libros de lógica clásica.

Todos los hombres son mortales.
Sócrates es hombre

Sócrates es mortal

Para modelar este razonamiento es necesario introducir predicados que describen las categorías necesarias así también como una constante (para representar al individuo Socrates).

$H.x$: x es hombre.

$M.x$: x es mortal.

s : Sócrates.

El razonamiento puede codifcarse entonces como sigue:

$$\frac{\langle\, \forall\, x \;::\; H.x \Rightarrow M.x \,\rangle \quad H.s}{M.s}$$

Ahora podemos traducir este razonamiento a la implicación correspondiente (recordemos que tenemos que ver que no se da el caso de que las premisas puedan ser verdaderas y la conclusión falsa). Luego demostramos que esa implicación es un teorema (del cálculo de predicados).

$$\begin{array}{ll} & \langle\, \forall\, x \;::\; H.x \Rightarrow M.x \,\rangle \wedge H.s \Rightarrow M.s \\ \equiv & \{\ \text{instanciación}\ \} \\ & \langle\, \forall\, x \;::\; H.x \Rightarrow M.x \,\rangle \wedge (H.s \Rightarrow M.s) \wedge H.s \Rightarrow M.s \\ \equiv & \{\ \text{modus ponens}\ \} \\ & \langle\, \forall\, x \;::\; H.x \Rightarrow M.x \,\rangle \wedge H.s \wedge M.s \Rightarrow M.s \\ \equiv & \{\ \text{debilitamiento}\ \} \\ & True \end{array}$$

Un tipo de razonamientos que tuvo gran popularidad y fue considerado el paradigma de los razonamientos válidos es el de los silogismos. Estos son ejemplos de razonamientos correctos a los cuales Aristóteles pesaba que se podían reducir todos. Si bien se sabe que no es así, estos constituyen un ejemplo interesante para trabajar nuestros métodos. Mostraremos que esencialmente todas las formas de silogismos correctos (dieciséis) se reducen a variantes menores de dos formas, las cuales demostraremos correctas.

La primera forma a considerar es la que tiene dos premisas universales y conclusión también universal. El siguiente razonamiento la ejemplifica:

$$\frac{\text{Las arañas son mamíferos.} \quad \text{Los mamíferos tienen seis patas.}}{\text{Las arañas tienen seis patas.}}$$

donde

$a.x$**:** x es araña.

$m.x$**:** x es mamífero.

$s.x$**:** x tiene seis patas.

La forma de este silogismo puede esquematizarse como sigue:

$$\frac{\langle\, \forall\, x \;::\; a.x \Rightarrow m.x \,\rangle \quad \langle\, \forall\, x \;::\; m.x \Rightarrow s.x \,\rangle}{\langle\, \forall\, x \;::\; a.x \Rightarrow s.x \,\rangle}$$

Demostraremos ahora la implicación asociada a este razonamiento. Para ello partiremos de la conjunción de las premisas y llegaremos a la conclusión.

$$
\begin{array}{ll}
 & \langle\, \forall\, x \;::\; a.x \Rightarrow m.x \,\rangle \wedge \langle\, \forall\, x \;::\; m.x \Rightarrow s.x \,\rangle \\
\equiv & \{\ \text{regla del término}\ \} \\
 & \langle\, \forall\, x \;::\; (a.x \Rightarrow m.x) \wedge (m.x \Rightarrow s.x) \,\rangle \\
\Rightarrow & \{\ \text{transitividad (usando la monotonía del } \wedge \text{ y del } \forall\ \} \\
 & \langle\, \forall\, x \;::\; a.x \Rightarrow s.x \,\rangle
\end{array}
$$

Consideremos el ejemplo que abre este capítulo:

Las vacas son mamíferos.
Hay animales que no son mamíferos.

Hay animales que no son vacas.

el cual puede esquematizarse como sigue:

$$
\begin{array}{l}
\langle\, \forall\, x \;::\; v.x \Rightarrow m.x \,\rangle \\
\langle\, \exists\, x \;::\; a.x \wedge \neg m.x \,\rangle \\
\hline
\langle\, \exists\, x \;::\; a.x \wedge \neg v.x \,\rangle
\end{array}
$$

La demostracón de corrección la dejamos como ejercicio para el lector (sugerimos distribuir la conjunción con el existencial y luego aplicar la instanciación adecuada y la contrarecíproca).

5.6 Algunas conclusiones

En el capítulo 2 planteamos 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 uno 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 al razonamiento pertinente y puede afirmarse que es una tautología si y solo 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 estructuras del mundo, 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. Puede pensarse que las constantes de nuestra lógica denotan objetos y los predicados relaciones o propiedades. Sin embargo, cómo se establece esta relación de denotación es aún un misterio.

5.7 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 contemporáneo con San Anselmo.

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 tú sepas que es bueno para mí - de que tú existes, tal como nosotros creemos, y que tú 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" él 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 este exista. Cuando un pintor considera de antemano lo que va a pintar ya lo tiene en su entendimiento, pero él 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 solo en el entendimiento. Porque si realmente está solo 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 solo 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 él 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 está 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 solo 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. Básicamente 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).

Presentaremos ahora una demostración de que existe un unicornio, la cual se basa en la misma idea y veremos 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 excluido, 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 el 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.

5.8 Ejercicios

Ejercicio 5.1
Probar

1. $\langle \exists x \ : r.x : \ p.x \rangle \Rightarrow \langle \exists x \ :: \ r.x \rangle$.
2. $\neg \langle \exists x \ :: \ r.x \rangle \Rightarrow \langle \forall x \ : r.x : \ p.x \rangle$.
3. $\langle \exists x \ :: \ p.x \rangle \wedge (\forall x :: q.x) \Rightarrow \langle \exists x \ :: \ p.x \wedge q.x \rangle$.
4. $\langle \exists x \ : p.x : \ q.x \rangle \wedge \langle \forall x \ : q.x : \ r.x \rangle \Rightarrow \langle \exists x \ : q.x : \ p.x \wedge r.x \rangle$.
5. $\langle \exists x \ : p.x : \ q.x \rangle \wedge \langle \forall x \ : q.x : \ r.x \rangle \Rightarrow \langle \exists x \ : p.x : \ r.x \rangle$.
6. $\langle \forall x \ : R.x : \ T.x \equiv U.x \rangle \Rightarrow (\langle \forall x \ : R.x : \ T.x \rangle \equiv \langle \forall x \ : R.x : \ U.x \rangle)$.
7. $\langle \forall x \ : R.x : \ T.x \rangle \vee \langle \forall y \ : R.y : \ U.y \rangle \Rightarrow \langle \forall x \ : R.x : \ T.x \vee U.x \rangle$.

Ejercicio 5.2
Dar un ejemplo que pruebe que el $\Rightarrow$ no puede ser reemplazado con $\equiv$ en los ejercicios 6 y 7.

Ejercicio 5.3
Demostrar $\langle \forall\, x \;::\; P.x \rangle \Rightarrow \langle \exists\, x \;::\; P.x \rangle$.

Probar que **no es cierto** si se agrega rango, es decir, probar que en general **no es cierto** que
$\langle \forall\, x \;:\; R.x :\; T.x \rangle \Rightarrow \langle \exists\, x \;:\; R.x :\; T.x \rangle$.

Ejercicio 5.4
Demostrar $\langle \forall\, x \;:\; R.x :\; T.x \rangle \Rightarrow (\langle \forall\, x \;::\; R.x \rangle \Rightarrow \langle \forall\, x \;::\; T.x \rangle)$.

Dar un ejemplo que pruebe que el $\Rightarrow$ principal no puede ser reemplazado con $\equiv$. Es decir, probar que **no es cierto** que
$\langle \forall\, x \;:\; R.x :\; T.x \rangle \equiv (\langle \forall\, x \;::\; R.x \rangle \Rightarrow \langle \forall\, x \;::\; T.x \rangle)$.

Ejercicio 5.5 (Análisis de Razonamientos usando Cálculo de Predicados)
Entenderemos que "ningún R es P" es lo mismo que decir que"todo R no es P", y decir que "algún R es P" es negar que "ningún R es P".

1. Probar que el razonamiento "Algun mamífero es ovíparo, por lo tanto algún ovíparo es mamífero", es válido, probando que
   $\langle \exists\, x \;:\; r.x :\; p.x \rangle \doteq \langle \exists\, x \;:\; p.x :\; r.x \rangle$.
2. Probar que $\neg \langle \exists\, x \;:\; r.x :\; p.x \rangle \doteq \neg \langle \exists\, x \;:\; p.x :\; r.x \rangle$.

   Ayuda: recordar el ejercicio anterior.
3. Probar que el razonamiento "Ningún hombre es un ángel, por lo tanto ningún ángel es un hombre", es válido.
4. Probar que el siguiente razonamiento es válido: "Si algunos unicornios no están en mi casa, entonces existe algún unicornio".
5. Probar que el razonamiento "No existe ningún unicornio, por lo tanto todo unicornio tiene dos cuernos" es válido.
6. *a*) Probar que $\langle \forall\, x \;:\; r.x :\; p.x \rangle \Rightarrow (\forall x : r.x \wedge q.x : p.x \wedge q.x)$.
   b) Concluir que el siguiente razonamiento es válido: "Todo elefante es un animal, luego todo elefante gris es un animal gris".
   c) ¿Por qué no es válido el siguiente razonamiento, aparentemente igual al anterior?: "Todo elefante es un animal, luego todo elefante pequeño es un animal pequeño"?
7. Analizar los siguientes razonamientos, traduciendo cada paso a notación lógica, y descubrir el paso en el cual fallan, pero destacando aquellos pasos que están bien.

a) "Ningún matemático ha logrado cuadrar el círculo. Luego, nadie que haya cuadrado el círculo es matemático. Por lo tanto, todos los que han cuadrado el círculo son no-matemáticos. Entonces, ciertamente, algunos de los que han cuadrado el círculo son no-matemáticos. Luego, algún no-matemático ha cuadrado el círculo."

b) "Es verdad que ningún unicornio está en mi casa. Luego, es falso que todos los unicornios estén en mi casa. Por lo tanto, es verdad que algunos unicornios no están en mi casa. Con lo que concluimos que existe algún unicornio."

Ejercicio 5.6

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 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 a 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 las cuales fueron analizadas en el capítulo precedente.

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 números impares.

En esta expresión pueden distinguirse varias componentes. Por un lado el operador usado (la sumatoria), correspondiente a un operador binario (la suma); por otro están las variables (en este caso solo 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 cuáles serán 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 ó 1. Usualmente en matemática se asume que el lector puede discernir estos casos sin inconvenientes. Sin embargo estas ambigüedades son más 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 proveerse reglas explícitas para el manejo de expresiones del estilo de la sumatoria, las cuales nos aseguran 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, generalizaremos 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 sí 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, entendiéndose por esto al uso de variables formales con un alcance delimitado explícitamente las cuales pueden usarse para construir expresiones dependientes de estas pero solo 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:

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

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) solo tiene sentido dentro de los símbolos $\langle \ \ \rangle$. 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, $\langle \oplus i \ :: \ T.i \rangle \equiv \langle \oplus i \ : True : \ T.i \rangle$.

Uno de los conceptos esenciales para comprender y manejar a las expresiones cuantificadas es el de *variable ligada*. Asociados a este concepto están 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 tomará 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 $\langle \sum j \ : 0 \leq j < n : \ 2 * j + 1 \rangle$. Diremos 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. Solo 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 será

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

o sea

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

Generalizamos a continuación una definición similar a la dada en el capítulo precedente para el caso de los cuantificadores lógicos.

Definición 6.1 (Variable libre)
Se define inductivamente cuándo 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 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 $\langle \oplus x \ : E : \ F \rangle$.

Notación: 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 tal que $i \in V.x$, luego la variable i está ligada (a la variable de cuatificación correspondiente) en las expresiones $\langle \oplus x \ : E : \ F \rangle$ y $\langle \oplus x \ : F : \ E \rangle$.

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

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

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

6.2 Revisión de 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 se obtiene una expresión obviamente diferente de las anteriores (en particular será siempre igual a 0, sin importar el valor de n en el estado).

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

Extenderemos la sustitución a expresiones cuantificadas a través del siguiente axioma:

Axioma 6.3 (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 6.12.

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 rápidamente 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 se demostrarán algunas propiedades que serán 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 qué versión de la regla estará siendo usada.

Axioma 6.4 (Rango vacío)

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

$$\langle \oplus i \ : False : \ T \rangle = e$$

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

Axioma 6.5 (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 6.6 (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 6.7 (Partición de rango generalizada)
Si el operador $\oplus$ es idempotente y el rango de especificación es una cuantificación existencial, 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 6.8 (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 6.9 (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 6.10 (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 absorbente 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 6.11 (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 6.12 (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 6.13 (Cambio de variable)
Si f es una función que tiene inversa (es 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 inversa puede escribirse como

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

Demostración
Comenzamos la demostración con la expresión más complicada

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

Teorema 6.14 (Separación de un término)

$$\langle \oplus i \ : 0 \leq i < n+1 : \ T.i \rangle = T.0 \oplus \langle \oplus i \ : 0 \leq i < n : \ T.(i+1) \rangle$$

Teorema 6.15 (Separación de un término)

$$\langle \oplus i \ : 0 \leq i < n+1 : \ T.i \rangle = \langle \oplus i \ : 0 \leq i < n : \ T.i \rangle \oplus T.n$$

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: $\langle \sum i \ : False : \ T.i \rangle = 0$.

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: veamos la aplicación de esta regla en el último paso de la siguiente demostración.

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

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
$\langle \sum i \ : R.i : \ T.i * x \rangle = \langle \sum i \ : R.i : \ T.i \rangle * x$.

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

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 1.6, ejercicio 1.7).

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:

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

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

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. Este 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:

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

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 6.8 y sección 7.5).

6.5 Expresiones cuantificadas para conjuntos

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

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

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 comprensión):

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

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:

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

La ventaja de pensar a los conjuntos definidos por comprensión como se indica más arriba es que al hacerlo, estos 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:

Regla de pertenencia: $x \in \langle \cup\, i \,:R.i :\, T.i\, \rangle \equiv \langle\, \exists\, i \,:R.i :\, x \in T.i\, \rangle \quad .$

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 esta es:

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

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

6.6 Cuantificadores lógicos

En el capítulo anterior trabajamos el cálculo de predicados. Los cuantificadores existencial y universal son ejemplos de expresiones cuantificadas y satisfacen todas las propiedades de estas, las cuales fueron presentadas como axiomas o teoremas. Las enumeramos nuevamente a continuación.

El cuantificador universal

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

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

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
$\langle \forall i \ : R.i : \ T.i \vee x \rangle \equiv \langle \forall i \ : R.i : \ T.i \rangle \vee x$.

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

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:

$$\begin{array}{lcl} \textbf{\textit{Regla de intercambio}}\textbf{:} \ \langle \forall i \ : R.i : \ T.i \rangle & \equiv & \langle \forall i \ : True : \ \neg R.i \vee T.i \rangle \\ & \equiv & \langle \forall i \ :: \ R.i \Rightarrow T.i \rangle \ . \end{array}$$

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.

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

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

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:

$$\textbf{\textit{Reglas de De Morgan:}} \quad \begin{array}{lcl} \neg \langle \forall\, i \,:R.i:\, T.i \rangle & \equiv & \langle \exists\, i \,:R.i:\, \neg T.i \rangle \\ \neg \langle \exists\, i \,:R.i:\, T.i \rangle & \equiv & \langle \forall\, i \,:R.i:\, \neg T.i \rangle \ . \end{array}$$

6.7 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
Existe otra notación muy usual para la cuantificación universal y existencial que suprime el rango de cuantificación de forma explícita:

Cuantificador universal $\quad (\forall i \,.\, P)$

Cuantificador existencial $\quad (\exists i \,.\, P)$

Su equivalencia con la notación que ya conocemos puede ser expresada (gracias a las reglas de intercambio) por las siguientes equivalencias:

$$\begin{array}{lcl} (\forall i \,.\, P) & \equiv & \langle \forall\, i \,::\, P \rangle \\ (\exists i \,.\, P) & \equiv & \langle \exists\, i \,::\, P \rangle \\ \langle \forall\, i \,:R:\, T \rangle & \equiv & (\forall i \,.\, R \Rightarrow T) \\ \langle \exists\, i \,:R:\, T \rangle & \equiv & (\exists i \,.\, R \wedge T) \ . \end{array}$$

Demostrar

$$(\forall i \,.\, P) \Rightarrow (\exists i \,.\, P)$$

y que en nuestra notación en general **no** se cumple

$$\langle \forall\, i \,:R:\, T \rangle \Rightarrow \langle \exists\, i \,:R:\, T \rangle \ .$$

Ejercicio 6.7
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 6.8
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 6.9
[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 6.10
[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

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

Ejercicio 6.11
[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) $R.y.z \not\equiv False$ 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 6.12
[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 6.13
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

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 este capítulo definiremos 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 esta. 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].

7.1 Funciones

Las funciones serán uno de los conceptos esenciales en los cuales se basa el formalismo básico. Como ya indicamos en el capítulo 1, 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 esta 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 llamamos **regla de Leibniz**, la cual ya fue presentada en el capítulo 1. Usando expresiones cuantificadas podemos expresar 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 7.1

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

$$\begin{array}{|l} \forall\ p\ primo\ \wedge \forall\ x, y \in Nat, \\ \hline \quad\quad f.p \ = \ 1 \\ \quad f.(x * y) \ = \ f.x + f.y \end{array}$$

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 satisfaga 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 $x, y \in \mathbb{N}$, $y \neq 0$.

$$\begin{array}{ll} & \sqrt{p} \text{ racional} \\ \equiv & \{ \text{ definición de racional } \} \\ & \left\langle \exists\, x, y \ ::\ \sqrt{p} = \frac{x}{y} \right\rangle \end{array}$$

$$
\begin{array}{ll}
\equiv & \{ \text{ álgebra } \} \\
& \langle\, \exists\, x, y \;::\; p * y^2 = x^2 \,\rangle \\
\Rightarrow & \{ \text{ regla de Leibniz } \} \\
& \langle\, \exists\, x, y \;::\; f.(p * y^2) = f.x^2 \,\rangle \\
\equiv & \{ \text{ propiedad de } f \} \\
& \langle\, \exists\, x, y \;::\; f.p + f.y + f.y = f.x + f.x \,\rangle \\
\equiv & \{ \text{ álgebra, } f.p = 1 \} \\
& \langle\, \exists\, x, y \;::\; 1 + 2 * f.y = 2 * f.x \,\rangle \\
\equiv & \{ \text{ números pares e impares son diferentes } \} \\
& False
\end{array}
$$

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

7.2 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 solo 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 estas 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 estas.

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 esta exista (profundizaremos estos conceptos en el capítulo 8).

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 constituirá 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 permitirá distinguir una definición (las cuales serán nuestros programas) de una propiedad. Las reglas de plegado/desplegado de la sección 7.3 mostrarán que el signo de definición y el signo igual están íntimamente 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 su posterior reducción a su forma normal, donde los nombres definidos en el programa pueden ocurrir en la expresión (veremos esto más detenidamente en el capítulo 8).

7.3 Reglas para el cálculo con definiciones

El formalismo básico nos permitirá no solo 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 solo "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:

$$\frac{(\circ) : (B \mapsto C) \mapsto (A \mapsto B) \mapsto (A \mapsto C)}{(f \circ g).x \doteq f.(g.x)}$$

Nótese que estamos usando *notación infija* para el operador de composición. Usaremos la convención de que dado un operador infijo la función 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:

$$\frac{(\circ) : (B \mapsto C) \mapsto (A \mapsto B) \mapsto (A \mapsto C)}{(\circ).f.g.x \doteq f.(g.x)}$$

En esta defínción es más claro que tanto f y g como x son solo 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:

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

7.4 Definiciones locales

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

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

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, $disc$ solo 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 solo tienen sentido dentro de la definición de $raiz1$.

En caso de haber más de una definición local separaremos las mismas con comas, dentro de los corchetes.

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

7.5 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

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

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 será el valor de alguna E_i tal que B_i es cierta.

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

$$\begin{array}{|lcl} \overline{max.a.b} & \doteq & (\quad a \leq b \quad \to \quad b \\ & & \Box\quad b \geq a \quad \to \quad a \\ & &) \end{array}$$

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, cuál 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 7.2
La función factorial puede definirse por casos como sigue

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

Regla para análisis por casos: 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 7.3
Para definir la regla de rango unitario para el cuantificador numérico N es necesario introducir una función n definida por casos:

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

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 7.1
Demostrar la siguiente regla usando análisis por casos.

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

donde e es el neutro de $\oplus$.

7.6 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 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 (incluido 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 solo 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 ó 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

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

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

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

7.7 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 $Num \times Num \mapsto Num$.

7.8 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: $+, -, *, /, \hat{}$, 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.

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 solo 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".

Ejercicio 7.2
Determinar las propiedades que tienen las funciones *div* y *mod*.

7.9 Tuplas

Una manera de formar un nuevo tipo combinando los tipos básicos es tomar estos 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:

$$\boxed{f.(x, y, z) \doteq \ldots}$$

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

$$(a_0, \ldots, a_n).i = a_i \ , \qquad 0 \leq i \leq n \ .$$

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

Ejemplo 7.4
Los números racionales pueden ser representados por pares de numeros enteros. La función que suma dos racionales puede ser definida como:

$$\begin{array}{|l} \hline SumRat : (Num, Num) \mapsto (Num, Num) \mapsto (Num, Num) \\ \hline SumRat.(a, b).(c, d) = (a * d + b * c\ ,\ b * d) \\ \hline \end{array}$$

7.10 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 solo consideraremos listas finitas). Cuando son finitas, se las denota entre corchetes, con sus elementos separados por comas.

Ejemplos: $[0, 1, 2, 3]$

$[(\text{“Juan”}, True), (\text{“Jorge”}, False)]$

[‘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 $[Num]$

$[(\text{“Juan”}, True), (\text{“Jorge”}, False)]$ 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. Por 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. Por ejemplo, [1,1] es una lista de dos elementos. Dos listas son iguales solo 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.

Constructores de listas

Una lista se puede generar con los siguientes constructores:

1. $[\,]$ lista vacía.

2. $\rhd$ agrega un elemento a una lista por la izquierda.

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

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

3. $\lhd$ agrega un elemento a una lista por la derecha.

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

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

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

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 cómo agregar un elemento por la izquierda).

Proposición 7.1
Sean x, y de tipo A y xs, ys de tipo $[A]$. Entonces

$$(x \rhd xs) = (y \rhd ys) \;\Leftrightarrow\; x = y \wedge xs = ys.$$

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

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

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

longitud

$$\#[\,] \quad \doteq \quad 0$$

$$\begin{aligned} \#(xs +\!\!+ ys) &\doteq \#xs + \#ys \\ \#(xs{\uparrow}n) &= n \min \#xs \\ \#(xs{\downarrow}n) &= (\#xs - n) \max 0 \end{aligned}$$

tomar n

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

tirar n

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

indexar

$$\begin{aligned} (x \rhd xs).0 &\doteq x \\ (x \rhd xs).(n+1) &\doteq xs.n \end{aligned}$$

7.11 Ejercicios

Ejercicio 7.3
Definir la función $sgn : Int \to Int$ que dado un número devuelve 1, 0 ó -1 en caso que el número sea positivo, cero o negativo respectivamente.

Ejercicio 7.4
Definir una función $abs : Int \to Int$ que calcule el valor absoluto de un número.

Ejercicio 7.5
Definir el predicado $bisiesto : Nat \to Bool$ que determina si un año es bisiesto, Recordar que los años bisiestos son aquellos que son divisibles por 4 pero no por 100, a menos que también lo sean por 400. Por ejemplo 1900 no es bisiseto pero 2000 sí lo es.

Ejercicio 7.6
Definir los predicados *equi* e *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 7.7
Definir la función $edad : (Nat, Nat, Nat) \to (Nat, Nat, Nat) \to Int$ que dadas dos fechas indica los años transcurridos entre ellas. Por ejemplo

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

Ejercicio 7.8
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:

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

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

Ejercicio 7.9
Definir la función *raices*, que devuelve las raíces de un polinomio de segundo grado.

Ejercicio 7.10
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 7.11
Reescribir la 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 7.12
Reescribir la 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 7.13
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 8

Modelo computacional

> -Aún queda fuego en la chimenea –dijo Paracelso–. Si arrojaras esta rosa a las brasas, creerías que ha sido consumida y que la ceniza es verdadera. Te digo que la rosa es eterna y que solo su apariencia puede cambiar. Me bastaría una palabra para que la vieras de nuevo.
>
> -¿Una palabra? –dijo con extrañeza el discípulo–. El atanor está apagado y están llenos de polvo los alambiques. ¿Qué harías para que resurgiera?
>
> Paracelso le miró con tristeza.
>
> Jorge Luis Borges: *La rosa de Paracelso*

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. Consecuentemente se estableció una relación entre lenguaje y programas pero tratando de abstraer algunas de sus características con la finalidad de simplificar la concepción 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. De esta forma se describe cómo los programas se ejecutan, en vez de analizar simplemente el resultado de esta ejecución.

La interpretación operacional se utiliza mayormente como guía para implementar el lenguaje en la computadora (en la forma de compiladores e intérpretes) y

para probar que esta implementación es correcta. Además, sirve para analizar la complejidad de funciones definidas en el formalismo básico y para obtener algunos resultados teóricos referentes al mismo (en la sección 8.6 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.

8.1 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.

Como hemos mencionado en la sección 7.2, es importante distinguir entre un valor y su representación por medio de expresiones. Sea la función *cuadrado* : $Num \to Num$ 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".

funciones: Dadas las funciones

$$\begin{array}{lcl} f.x.y & \doteq & (\quad x \geq y \quad \to \quad x \\ & & \square \quad x \leq y \quad \to \quad y \\ & &) \end{array} ,$$

$$\begin{array}{lcl} g.x.y & \doteq & (\quad x > y \quad \to \quad x \\ & & \square \quad x < y \quad \to \quad y \\ & & \square \quad x = y \quad \to \quad x \\ & &) \end{array} ,$$

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.

8.2 Forma normal

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, ¿cuál 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, a lo sumo una que llamaremos *expresión canónica* de ese valor. Además, llamaremos a la expresión canónica que representa el valor de una expresión la *forma normal* de esa expresión. Con esta restricción pueden quedar expresiones sin forma normal.

Aparte del requerimiento mencionado al principio del párrafo anterior, no hay ninguna otra restricción objetiva para elegir la forma normal 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 más 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 $15 * 15$.

Otra cuestión importante es que, aunque se defina este conjunto de forma tal que existan expresiones sin forma normal, el conjunto de expresiones que sí 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 *expresión 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 *expresión canónica* del valor π.

Hasta ahora hemos visto valores que no tienen expresión canónica (funciones, número π). También puede suceder que una expresión no tenga forma normal. Por ejemplo dada la definición de función:

$$inf \doteq inf + 1 \tag{8.1}$$

La expresión inf (de tipo número) no tiene forma normal dado que puede demostrarse que su valor es diferente al de cualquier forma normal.

Sea n la forma normal que representa el mismo valor que la expresión inf (n es la forma normal 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 normal) se cumple $n \in \mathbb{R}$. Suponiendo esto (será nuestra hipótesis) podemos demostrar:

$$\begin{array}{ll} & n \\ = & \{\ \text{hipótesis}\ \} \\ & inf \end{array}$$

$$
\begin{aligned}
&= \{ \text{ definición de } \mathit{inf} \ \} \\
&\quad \mathit{inf} + 1 \\
&= \{ \text{ hipótesis } \} \\
&\quad n + 1
\end{aligned}
$$

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 normal.

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 normal 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 normal, $n \in \mathbb{R}$. Suponiendo esto:

$$
\begin{aligned}
&\quad 0 \\
&= \{ \text{ elemento neutro de la multiplicación en } \mathbb{R} \text{ y } n \in \mathbb{R} \ \} \\
&\quad 0 * n \\
&= \{ \text{ conmutabilidad de la multiplicación en } R \text{ y } n \in \mathbb{R} \ \} \\
&\quad n * 0 \\
&= \{ \text{ hipótesis } \} \\
&\quad \frac{1}{0} * 0 \\
&= \{ \text{ inverso multiplicativo de 0 } \} \\
&\quad 1
\end{aligned}
$$

En consecuencia concluimos que $0 = 1$. Razonando por el absurdo n no puede ser la forma normal de $\frac{1}{0}$ y por lo tanto esta expresión no tiene forma normal.

8.3 Evaluación

A partir de todo lo dicho, la ejecución de un programa será el proceso de encontrar su forma normal. 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 normal 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

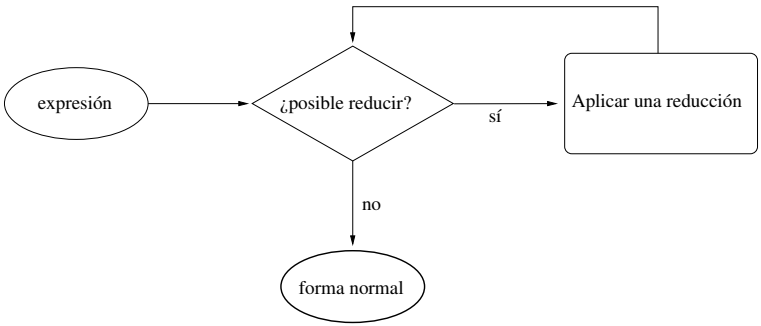


Figura 8.1: sistema de evaluación

dentro de una computadora está hecho de forma tal que cuando ya no es posible reducir la expresión es porque se ha llegado a la forma normal (ver figura 8.1).

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 $cuadrado.(3 * 5)$:

$$
\begin{aligned}
& cuadrado.(3 * 5) \\
= & \ \{ \text{ aritmética } \} \\
& cuadrado.15 \\
= & \ \{ \text{ definición de } cuadrado \ \} \\
& 15 * 15 \\
= & \ \{ \text{ aritmética } \} \\
& 225
\end{aligned}
$$

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:

$$
\begin{aligned}
& cuadrado.(3 * 5) \\
= & \ \{ \text{ definición de } cuadrado \ \} \\
& (3 * 5) * (3 * 5) \\
= & \ \{ \ * \text{ aplicada al primer factor } \} \\
& 15 * (3 * 5) \\
= & \ \{ \ * \text{ aplicada al segundo factor } \} \\
& 15 * 15
\end{aligned}
$$

$$
\begin{aligned}
&= \{ \text{ aritmética } \} \\
&\quad 225
\end{aligned}
$$

En el primer paso de reducción hemos elegido la subexpresión de más 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:

$$
\begin{aligned}
&\quad inf \\
&= \{ \text{ definición de } inf \text{ } \} \\
&\quad inf + 1 \\
&= \{ \text{ definición de } inf \text{ } \} \\
&\quad (inf + 1) + 1 \\
&= \{ \text{ asociatividad de la suma } \} \\
&\quad inf + (1 + 1) \\
&= \{ \text{ aritmética } \} \\
&\quad inf + 2 \\
&= \{ \text{ definición de } inf \text{ } \} \\
&\quad (inf + 1) + 2 \\
&= \{ \text{ asociatividad de la suma } \} \\
&\quad inf + (1 + 2) \\
&= \{ \text{ aritmética } \} \\
&\quad inf + 3 \\
&= \{ \text{ definición de } inf \text{ } \} \\
&\quad (inf + 1) + 3 \\
&\quad \ldots
\end{aligned}
$$

Obviamente se puede seguir aplicando pasos de reducción indefinidamente. Esto no es necesariamente un problema ya que la expresión *inf* no tiene forma normal.

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 normal 3. Veamos que sucede si aplicamos pasos de reducción a esta expresión eligiendo siempre la subexpresión de más adentro:

$$
\begin{aligned}
&\quad K.3.inf \\
&= \{ \text{ definición de } inf \text{ } \} \\
&\quad K.3.(inf + 1)
\end{aligned}
$$

$$
\begin{aligned}
&= \{ \text{ definición de } \mathit{inf} \ \} \\
&\quad K.3.((\mathit{inf} + 1) + 1) \\
&= \{ \text{ asociatividad de la suma } \} \\
&\quad K.3.(\mathit{inf} + (1 + 1)) \\
&= \{ \text{ aritmética } \} \\
&\quad K.3.(\mathit{inf} + 2) \\
&= \{ \text{ definición de } \mathit{inf} \ \} \\
&\quad K.3.((\mathit{inf} + 1) + 2) \\
&\quad \ldots
\end{aligned}
$$

Obviamente este proceso no termina. Pero ahora elijamos las subexpresión de más afuera:

$$
\begin{aligned}
&\quad K.3.\mathit{inf} \\
&= \{ \text{ definición de } K \ \} \\
&\quad 3
\end{aligned}
$$

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 está basado) que avala estas pruebas.

Teorema 8.1

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, se arribará a la forma normal, siempre que ella exista. El orden (más afuera y más a la izquierda) de reducción se denomina *orden normal* de reducción.

En los ejemplos donde no se pudo lograr la convergencia a una forma normal se utilizó la estrategia que siempre elige la subexpresión de más adentro y más a la izquierda. Este orden de reducción se denomina *orden aplicativo*.

8.4 Un modelo computacional más eficiente

Según el teorema 8.1 nuestro modelo computacional deberá usar una estrategia de reducción normal si se desea encontrar la forma normal, siempre que exista. Pero esta estrategia es, en algunos casos, menos eficiente. Veamos cómo se evalúa la expresión $\mathit{cuadrado}.(\mathit{cuadrado}.3)$ utilizando la misma.

$$
\begin{aligned}
& \mathit{cuadrado}.(\mathit{cuadrado}.3) \\
= & \ \{ \text{ definición de cuadrado } \} \\
& (\mathit{cuadrado}.3) * (\mathit{cuadrado}.3) \\
= & \ \{ \text{ definición de cuadrado } \} \\
& (3 * 3) * (\mathit{cuadrado}.3) \\
= & \ \{ \text{ aritmética } \} \\
& 9 * (\mathit{cuadrado}.3) \\
= & \ \{ \text{ definición de cuadrado } \} \\
& 9 * (3 * 3) \\
= & \ \{ \text{ aritmética } \} \\
& 9 * 9 \\
= & \ \{ \text{ aritmética } \} \\
& 81
\end{aligned}
$$

En seis pasos de reducción logramos encontrar la forma normal. Pero ahora no utilicemos esta estrategia.

$$
\begin{aligned}
& \mathit{cuadrado}.(\mathit{cuadrado}.3) \\
= & \ \{ \text{ definición de cuadrado } \} \\
& \mathit{cuadrado}.(3 * 3) \\
= & \ \{ \text{ aritmética } \} \\
& \mathit{cuadrado}.9 \\
= & \ \{ \text{ definición de cuadrado } \} \\
& 9 * 9 \\
= & \ \{ \text{ aritmética } \} \\
& 81
\end{aligned}
$$

y solo se realizaron cuatro pasos de reducción.

Analicemos más detenidamente lo que sucedió. Notemos que en la expresión inicial $\mathit{cuadrado}.(\mathit{cuadrado}.3)$ la función *cuadrado* aparece dos veces. En los ejemplos se puede ver que en el primero se "copió" la subexpresión $\mathit{cuadrado}.3$ 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:

$$\begin{array}{ll} & cuadrado.(cuadrado.3) \\ = & \{ \text{ definición de cuadrado } \} \\ & x * x \\ & \quad [\![x = cuadrado.3]\!] \\ = & \{ \text{ definición de cuadrado } \} \\ & x * x \\ & \quad [\![x = 3 * 3]\!] \\ = & \{ \text{ aritmética } \} \\ & x * x \\ & \quad [\![x = 9]\!] \\ = & \{ \text{ definición local } \} \\ & 9 * 9 \\ = & \{ \text{ aritmética } \} \\ & 81 \end{array}$$

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 subexpresiones, se denomina generalmente en la literatura *modelo computacional lazy con evaluación "call-by-need"*.

Como ya dijimos en la introducción de este capítulo, 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 como por ejemplo Haskell [Has06]. Por lo general es implementado representando las expresiones como un grafo, denominado *grafo de reducción*. Para más información sobre implementación de lenguajes funcionales ver [JL91].

Otro modelo computacional es el denominado *modelo computacional estricto*. Este modelo implementa estrategias de evaluación del tipo aplicativas (las expresiones se evalúan desde más adentro), y es el que se utiliza en algunos lenguajes funcionales como por ejemplo ML, OCaml y Scheme y para la evaluación de funciones en lenguajes imperativos como C, C++ y Java.

En la próxima sección veremos una forma analítica de obtener una medida de la eficiencia de las funciones recursivas definidas en el formalismo básico.

8.5 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 así 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 8.1
Calculemos $Tfac$. Como

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

resulta

$$\left\{\begin{array}{rcl} Tfac.0 & = & 1 \\ Tfac.(n+1) & = & 1 + Tfac.n \end{array}\right.$$

Ahora hay que resolver estas ecuaciones. Si tomamos $Tfac.n = n+1$, 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 8.2
Calculemos Tf, donde f está dada por

$$\boxed{\begin{array}{rcl} f.0 & \doteq & 0 \\ f.(n+1) & \doteq & f.n + g.n \\ & & [\![g.i = X^i]\!] \end{array}}$$

Para el costo de g tenemos

$$\left\{ \begin{array}{rcl} Tg.0 & = & 1 \\ Tg.(n+1) & = & 1 + Tg.n \end{array} \right.$$

de donde resulta $Tg.n = n + 1$ para todo n. Entonces, para f tenemos

$$\left\{ \begin{array}{rcl} Tf.0 & = & 1 \\ Tf.(n+1) & = & 1 + Tf.n + Tg.n \\ & = & 2 + n + Tf.n \end{array} \right.$$

Se puede demostrar que si se define

$$Tf.n = \frac{n * (n+3)}{2} \qquad \forall\ n$$

se satisfacen todas las ecuaciones. Por lo tanto el costo de evaluar f es cuadrático.

8.6 Problema de la forma normal

Ya hemos dicho que 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

$$nf.e \ \equiv\ e \text{ tiene forma normal} \tag{8.2}$$

Construyamos ahora la definición

$$\boxed{\begin{array}{rcccl} P & \doteq & (& nf.P & \to \quad inf \\ & & \Box & \neg nf.P & \to \quad 0 \\ & &) & & \end{array}}$$

Como ya lo demostramos en la sección 8.2, inf no tiene forma normal. Por lo tanto, usando el modelo computacional, es obvio que P tiene forma normal (igual a 0) si y solo si $\neg nf.P$. Luego

$$\begin{array}{ll} & P \text{ tiene forma normal} \\ \equiv & \{ \text{ modelo computacional } \} \\ & \neg nf.P \end{array}$$

$\equiv$ { especificación de nf }
$\neg(P$ tiene forma normal$)$

absurdo. Luego, no puede existir un predicado nf que satisfaga la especificación 8.2.

8.7 Ejercicios

Ejercicio 8.1
Mostrar los pasos de reducción hasta llegar a la forma normal de la expresión:

$$2 * cuadrado.(head.[2, 4, 5, 6, 7, 8])$$

1. utilizando el orden de reducción aplicativo.
2. utilizando el orden de reducción normal.

Ejercicio 8.2
Dada la definición:

$$\boxed{linf \doteq 1 \triangleright linf}$$

mostrar los pasos de reducción hasta llegar a la forma normal de la expresión:

$$head.linf$$

1. utilizando el orden de reducción aplicativo.
2. utilizando el orden de reducción normal.

Comparar ambos resultados.

Ejercicio 8.3
Dada la definición:

$$\boxed{\begin{array}{rcl} f.x.0 & \doteq & x \\ f.x.(n+1) & \doteq & cuadrado.(f.x.n) \end{array}}$$

mostrar los pasos de reducción hasta llegar a la forma normal de la expresión:

$$f.2.3$$

1. utilizando la estrategia de reducción normal, sin utilizar definiciones locales.
2. utilizando el modelo computacional lazy con evaluación “call-by-need”.

Comparar ambos resultados.

Ejercicio 8.4
Dadas las definiciones

$$
\begin{array}{|l}
\hline
cond : Bool \mapsto A \mapsto A \\
\hline
cond.p.x.y \quad \doteq \quad \begin{array}{llll} (& p & \rightarrow & x \\ \Box & \neg p & \rightarrow & y \\ ) & & & \end{array} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
fac : Num \mapsto Num \\
\hline
fac.n \quad \doteq \quad cond.(n = 0).1.(n * fac.(n-1)) \\
\hline
\end{array}
$$

mostrar los pasos de redución de la expresión

$fac.2$

1. utilizando el orden de reducción aplicativo.
2. utilizando el modelo computacional lazy.

Responder a la pregunta: ¿Qué sucede si traslado este algoritmo al lenguaje Java y ejecuto el programa?

Capítulo 9

El proceso de construcción de programas

> Un coup de dés jamais n'abolira le hasard.
>
> Stéphane Mallarmé: *Un Coup de Dés*

En [Par90] se define al *desarrollo de software* como el proceso de

> *Dado un problema, encontrar un programa (o un conjunto de programas) que lo resuelvan (eficientemente).*

Este proceso involucra tanto cuestiones técnicas como organizativas. En este libro trataremos solo las primeras.

Una de las dificultades esenciales de este proceso radica en el hecho que las descripciones de los problemas a resolver suelen no ser precisas ni completas. Sin embargo, esta dificultad no cobró importancia hasta el final de la década del '60, dado que hasta ese momento las computadoras solo eran usadas para la solución de problemas científicos, los cuales estaban expresados en un lenguaje suficientemente preciso. Con el abaratamiento de los costos, las computadoras comenzaron a usarse para problemas originados en otros ámbitos (en general problemas administrativos), los cuales resultaron mucho más difíciles de resolver. Esta situación dio lugar a la llamada *crisis del software* [Gib94], la cual produjo un decaimiento en la confiabilidad del software y por lo tanto un cuestionamiento de los métodos usados para su desarrollo. A partir de haber identificado a estos métodos como esencialmente inadecuados, se comenzó a desarrollar un estilo de programación en el cual la demostración de corrección del software respecto de la especificación es

construida al mismo tiempo que el programa; más aún, la demostración es usada como guía para el desarrollo del programa.

El principal método hasta entonces usado era simplemente partir del problema, el cual estaba expresado de manera informal y poco detallada, y obtener un programa que, por definición, es muy detallado y está escrito en una notación formal (ver figura 9.1). Este salto demasiado grande fue una de las principales razones de dicha crisis. Volviendo a la analogía de la torta (sección 1.3), sería como intentar encontrar la fórmula final sin realizar todos los pasos intermedios.

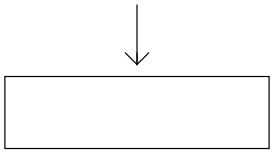


Figura 9.1: Desarrollo del programa directamente a partir del problema

Para comprobar que el programa era correcto se realizaban entonces ensayos con conjuntos de datos para los cuales se conocía el resultado, y si estos daban los resultados esperables se terminaba la tarea. En el caso frecuente en que los resultados no fueran correctos, se procedía a modificar el programa para intentar corregirlo. Esta tarea era sumamente difícil, dado que no se sabía a priori si el resultado inesperado se debía a meros errores de programación o a una concepción inadecuada del problema.

En la actualidad es ampliamente aceptado (tanto en la academia como en la industria) que el proceso de construcción de programas debe dividirse en al menos dos etapas: la etapa de *especificación* del problema y la etapa de desarrollo del programa o de *programación* (ver figura 9.2).

El resultado de la primera etapa es una **especificación formal** del problema, la cual seguirá siendo abstracta (poco detallada) pero estará escrita con precisión en algún lenguaje cuya semántica esté definida rigurosamente. La segunda etapa dará como resultado un programa y una demostración de que el programa es *correcto* respecto de la especificación dada. Nótese que no tiene sentido hablar de un programa correcto *per se*; que la noción de *corrección* de un programa siempre está referida a una especificación dada.

En el trabajo pionero de Hoare [Hoa69] (basado en trabajos anteriores de R.W.Floyd) se establecen axiomas y reglas de inferencia para la demostración formal de que un programa satisface una especificación, escrita esta en una lógica

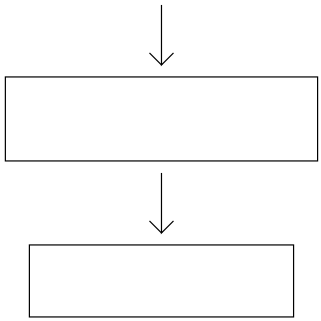


Figura 9.2: Especificación formal del problema previa al desarrollo del programa

de primer orden. Dicho trabajo muestra que es posible obtener una certeza equivalente a la provista por una demostración matemática de que un programa es correcto. Puede también verse en el trabajo que la confección de dicha prueba es una labor relativamente ardua. Más aún, este sistema de demostración permite solo hacer demostraciones *a posteriori* de la corrección del programa, pero no ayuda a construirlo.

Es un libro de E.W.Dijkstra, [Dij76], el que finalmente sienta las bases para la aplicación de la lógica de Hoare al desarrollo sistemático de programas. No solo se construye a través de estos métodos una demostración de corrección al mismo tiempo que el programa, sino que la misma demostración que se está contruyendo sirve como guía para construir el programa. Estas técnicas serán tratadas en el capítulo 16, cuando comencemos a trabajar con la programación imperativa.

La separación en dos pasos permite discernir ahora si un programa cuyos resultados no son los esperados es incorrecto o si, en cambio, es la especificación la que no describe al problema de manera adecuada. Es importante resaltar que la propiedad de *adecuación* no puede demostrarse formalmente, debido a la naturaleza informal de los problemas. De todas maneras, dado que una especificación es más abstracta que un programa, es mucho más fácil convencerse de que una especificación expresa nuestra idea informal de un problema que determinar esto a partir del código de un programa.

En lo que sigue de esta sección ilustraremos a través de ejemplos cómo pueden construirse especificaciones para problemas relativamente simples (pero para nada triviales). Todo el resto del curso estará dedicado a la construcción de programas

a partir de especificaciones formales.

9.1 Especificaciones

En un sentido general podemos definir una especificación como *una descripción formal de la tarea que un programa tiene que realizar*. Un slogan usado con referencia a las especificaciones es que estas responden a la pregunta *¿qué* hace el programa? mientras que el programa mismo, usualmente llamado la *implementación*, reponde a la pregunta *¿cómo* se realiza la tarea?

En el libro [BEKV94] se presenta a una especificación como un *contrato* entre el programador y el potencial usuario. Este contrato establece de manera precisa cuál es el comportamiento del producto que el programador debe proveer al usuario. Usualmente las especificaciones dicen que si el programa se ejecuta para un valor de un conjunto dado, el resultado satisfará una cierta propiedad. Una de las consecuencias de dicho contrato es que el usuario se compromete a usar el programa solo para valores en ese conjunto predeterminado y el programador a asegurar que el programa producirá un resultado satisfactorio. En principio si el usuario ingresa al programa datos que no están en el conjunto de datos aceptables, el comportamiento del programa puede ser cualquiera sin con ello violar el contrato establecido por la especificación.

En el contexto de nuestro formalismo básico podemos ser más precisos al definir el concepto de especificación. Diremos simplemente que una **especificación** es un predicado sobre el espacio de valores. En general estaremos especificando funciones, que como ya dijimos, son valores como cualesquiera otros. La tarea de programación de un problema especificado por un predicado dado puede describirse ahora simplemente como encontrar un valor (una función) que satisfaga dicho predicado.

En este libro no nos encargaremos de cualquier clase de problema, pues algunos son triviales y otros demasiado complejos o definidos muy vagamente.

Tomemos por ejemplo el problema de construir un sistema para la administración de los alumnos de una materia. A primera vista parece un problema fácil, pero si se examina el problema con un poco más de detalle puede notarse que su formulación es demasiado imprecisa. Si se da ese enunciado a ocho programadores diferentes, es probable que se obtengan ocho soluciones a problemas diferentes. No resolveremos problemas formulados de manera tan vaga.

Consideremos un problema más específico. Dado el conjunto de los alumnos de un curso se pide obtener el alumno con el segundo promedio más alto. Este problema parece mucho más simple, pero aún quedan varias cuestiones por resolver. Por ejemplo, puede ser que dicho alumno no exista (en caso de que todos los alumnos del curso tengan el mismo promedio) o que haya más de uno. Si nos quedamos solo con el enunciado informal, la solución para estos casos vuelve a quedar librada al criterio del programador.

En lo que resta del capítulo usaremos las herramientas introducidas anteriormente para escribir especificaciones precisas. Para ello interpretaremos las fórmulas del cálculo de predicados y las expresiones cuantificadas como es usual y construiremos con ellas y con el formalismo básico las especificaciones formales. Esto nos permitirá resolver las ambigüedades en la formulación de los problemas. Por supuesto, siempre queda la pregunta de si la especificación formal construida efectivamente expresa el enunciado del problema a resolver. Esta pregunta es inevitable y es bueno responderla de manera completa lo más temprano posible en el proceso de construcción de programas. Por otro lado, no siempre una especificación puede ser satisfecha por un programa o la construcción de este puede ser muy costosa o su ejecución muy ineficiente. En estos casos es necesario cambiar la especificación.

Ejemplo 9.1
El presente ejemplo ha sido adaptado de uno de [BEKV94]. El usuario propone al programador el problema de calcular la raíz cuadrada de un número real dado.

Primer intento. El programador, conociendo el dominio del álgebra, propone resolver el problema solo para reales que no sean negativos. El usuario acepta la propuesta por lo cual el contrato entre ambos (la especificación) queda establecido como sigue:

El programador se compromete a construir una función sqrt, tal que

$$\begin{array}{|l} sqrt : Num \mapsto Num \\ \hline \langle\, \forall\, x \;:\; 0 \leq x :\; (sqrt.x)^2 = x \,\rangle \end{array}$$

Puede pensarse a la especificación como una ecuación a resolver donde la incógnita es *sqrt*, es decir, el problema es encontrar una función *sqrt* tal que se satisfaga la especificación. Nótese que esta función no necesariamente es única, en particular en este ejemplo, dado que hay por lo menos dos soluciones posibles para la raíz cuadrada de un número estrictamente positivo. Esto da lugar a que haya infinitas soluciones para el problema planteado (dado que puede en principio elegirse para qué valores la raíz será positiva y para cuales negativa).

Una forma estilizada de escribir la misma especificación es separar la *precondición* de la *postcondición*. La primera condición habla de las restricciones que determinan los datos aceptables, mientras que la segunda establece las propiedades que satisfará el resultado. Nuestro ejemplo podría escribirse como sigue.

$$\begin{array}{|ll} sqrt : Num \mapsto Num & \\ \hline \text{pre:} & 0 \leq x \\ \text{post:} & (sqrt.x)^2 = x \end{array}$$

Nótese que la cuantificación de la variable x queda implícita. Esto será una práctica usual. En general se supondrá que las variables que aparecen como argumentos de la función especificada están cuantificadas universalmente. La especificación original puede entonces escribirse también simplemente como

$$\begin{array}{|l} sqrt : Num \mapsto Num \\ \hline 0 \leq x \Rightarrow (sqrt.x)^2 = x \end{array}$$

Segundo intento. Un fenómeno usual es que el programador por alguna razón no pueda satisfacer la especificación acordada. Por ejemplo, puede descubrir cuando está intentando resolver el problema que el sistema de cómputo en el cual va a implementar su programa solo maneja aproximaciones finitas a los números irracionales, por lo que es imposible encontrar por ejemplo una solución exacta de $sqrt.2$. El problema es que la especificación requiere que la solución sea exacta. Dada esta situación, el programador tiene que cambiar el contrato con el usuario, quien probablemente no va a sentirse satisfecho, aún en el caso usual en que el usuario es el mismo programador. Una posible especificación es

$$\begin{array}{|ll} sqrt : Num \mapsto Num & \\ \hline \text{pre:} & 0 \leq x \\ \text{post:} & |(sqrt.x)^2 - x| < \epsilon \end{array}$$

donde ϵ es una constante a ser negociada.

Escrita de esta manera, es obvio que la especificación es más favorable para el programador, en el sentido de que la postcondición es más débil y por lo tanto mas fácil de satisfacer. Si se escribe la especificación como

$$\begin{array}{|l} sqrt : Num \mapsto Num \\ \hline 0 \leq x \Rightarrow |(sqrt.x)^2 - x| < \epsilon \end{array}$$

queda claro que la especificación realizada en el primer intento implica lógicamente a esta.

Cambios en la especificación

Como se vio en el ejemplo anterior, frecuentemente es necesario modificar las especificaciones. En este caso la especificación fue *debilitada*, es decir se escribió otra especificación la que requería menos del programa. La forma de debilitarla fue debilitar la postcondición. Otra forma de debilitar una especificación es fortaleciendo la precondición, es decir permitiendo que el programa pueda ser solo usado con un conjunto más chico de valores de entrada. Por ejemplo, podemos poner

como requisito que los argumentos para *sqrt* sean cuadrados perfectos (con lo cual es posible encontrar valores exactos para la raíz cuadrada). La especificación de esta (no demasiado útil) función sería:

$$
\begin{array}{|l}
sqrt : Num \mapsto Num \\
\hline
\quad \text{pre:} \left\langle \exists\, y \ : y \in Nat : \ y^2 = x \right\rangle \\
\quad \text{post:} (sqrt.x)^2 = x
\end{array}
$$

Como contrapartida, a veces es necesario *fortalecer* la especificación. Esto ocurre en general dado que el usuario puede darse cuenta que necesita un programa con mejores propiedades que el anterior, o poder usarlo para un conjunto de datos de entrada más amplio que el que había pensado. En el ejemplo de la raíz cuadrada, puede por ejemplo requerirse que el resultado sea siempre positivo, con lo cual se está fortaleciendo la postcondición y por lo tanto la especificación como un todo, pudiendo escribirse como sigue.

$$
\begin{array}{|l}
sqrt : Num \mapsto Num \\
\hline
0 \leq x \Rightarrow |(sqrt.x)^2 - x| < \epsilon \wedge 0 \leq sqrt.x
\end{array}
$$

Ejemplo 9.2
Consideremos ahora la especificación de una función que calcule el segundo mejor promedio de un curso. La manera más simple de construir esta especificación es usando un par de especificaciones auxiliares que expresen cuál es el mayor promedio y cuál es el conjunto de aquellos estudiantes que tienen este promedio.

$$
\begin{array}{|l}
maxProm : \{Alumno\} \mapsto Num \\
mejProm : \{Alumno\} \mapsto \{Alumno\} \\
\hline
maxProm.A = \langle \text{Max } a \ : a \in A : \ prom.a \rangle \\
mejProm.A = \{a \ : \ a \in A \wedge prom.a = maxProm.A \ : \ a\}
\end{array}
$$

La especificación de la función requerida es ahora:

$$
\begin{array}{|l}
segProm : \{Alumno\} \mapsto \{Alumno\} \\
\hline
segProm.A = \{a \ : \ a \in A \wedge prom.a = maxProm.(A \setminus mejProm.A) \ : \ a\}
\end{array}
$$

donde estamos suponiendo que la función *prom* nos da el promedio de un alumno dado.

Las funciones *maxProm* y *mejProm* son funciones auxiliares que se usan para especificar la función requerida *segProm*, y no es en principio necesario contruir

un programa para ellas. Durante el proceso de construcción del programa para *segProm* se verá si es necesario construir también un programa para las especificaciones auxiliares o eventualmente para algunas otras que aparezcan (usando básicamente la técnica de *modularización*, la cual se verá en el capítulo 11). Si quisiéramos ser completamente precisos podríamos por un lado cuantificar universalmente la variable A y por otro indicar que la "incógnita" de la ecuación a resolver es *segProm*. Dejaremos esto implícito siempre que no lleve a confusión. No debe confundirse entonces a la especificación con una definición en el formalismo básico. El signo igual es aquí solo un predicado, el cual no necesariamente dará lugar a una definición en el formalismo básico.

Esta especificación nos permite resolver las dudas que teníamos, dado que si no hay ningún alumno con el segundo mejor promedio la función *segProm* tiene que devolver un conjunto vacío, y si hay más de uno, entonces devuelve el conjunto con todos estos. Nótese que hubo que tomar la decisión de calcular el conjunto de todos los que tienen el segundo mejor promedio dado que es una propiedad intrínseca del problema el hecho que el alumno con el segundo mejor promedio puede no ser único.

9.2 Ejemplos

Dado que no existen reglas de aplicación general para la especificación de problemas, incluimos en esta sección algunos ejemplos. La clave está en saber expresar los problemas planteados en lenguaje coloquial de manera concisa, usando un lenguaje formal.

Escribiremos el problema a especificar entre comillas. Cabe aclarar que con frecuencia los problemas pueden especificarse de diversas maneras, por lo que no deben tomarse las especificaciones dadas a continuación como las únicas posibles.

Ejemplo 9.3
"Dados $x \geq 0$ e $y > 0$ enteros, calcular el cociente y el resto de la división entera de x por y".

El algoritmo de división en los enteros nos asegura la existencia de un *único* entero q cociente de la división entera de x por y y un *único* entero r resto de esta división. Estos números están caracterizados por las propiedades $x = q * y + r$ y $0 \leq r < y$. Por lo tanto, una función que resuelva este problema puede especificarse como sigue

$$\begin{array}{|l} \hline divMod.x.y : Nat \to Nat \to (Nat \times Nat) \\ \hline x \geq 0 \wedge y > 0 \Rightarrow divMod.x.y = (q, r) \equiv (x = q * y + r \wedge 0 \leq r < y) \\ \hline \end{array}$$

Ejemplo 9.4
"Dado un entero $x \geq 0$, *sqrtInt*.x es la raíz cuadrada entera de x".

La raíz cuadrada entera de un número está definida como el mayor entero positivo cuyo cuadrado es menor o igual que el número en cuestión. Esto puede expresarse en lenguaje formal de la siguiente manera:

$$\begin{array}{|l} sqrtInt : Num \mapsto Num \\ \hline 0 \leq x \Rightarrow sqrtInt.x = \langle \text{Max } z \ : 0 \leq z \wedge z^2 \leq x : \ z \rangle \end{array}$$

alternativamente, puede evitarse el cuantificador máximo de la siguiente manera.

$$\begin{array}{|l} sqrtInt : Num \mapsto Num \\ \hline 0 \leq x \Rightarrow 0 \leq sqrtInt.x \wedge (sqrInt.x)^2 \leq x \wedge x < (sqrtInt.(x) + 1)^2) \end{array}$$

Ejemplo 9.5
Sea xs una lista no vacía de enteros. "m es el mínimo de xs".

$$\begin{array}{|l} m : Num \\ \hline m = \langle \text{Min } i \ : 0 \leq i < \#xs : \ xs.i \rangle \end{array}$$

Otra especificación posible se obtiene al describir con más detalle lo que significa mínimo:

$$\begin{array}{|l} m : Num \\ \hline \langle \forall i \ : 0 \leq i < \#xs : \ xs.i \geq m \rangle \wedge \langle \exists i \ : 0 \leq i < \#xs : \ xs.i = m \rangle \end{array}$$

Una manera más usual es definir una función que tome a la lista como parámetro (no como en el caso previo en el cual la constante m esta especificada solo para una lista particular xs). Consideramos la función *minlist* que dada una lista cualquiera devuelve el valor mínimo de la lista

$$\begin{array}{|l} minlist : [Num] \mapsto Num \\ \hline minlist.xs = \langle \text{Min } i \ : 0 \leq i < \#xs : \ xs.i \rangle \end{array}$$

Nótese que en esta última especificación la variable xs está implícitamente cuantificada (no así en el caso anterior, en el cual era una constante)

Ejemplo 9.6
Sea xs una lista no vacía. "Todos los elementos de xs son iguales".

Una manera de especificar este problema es expresar la condición de igualdad de todos los elementos diciendo que deben ser iguales dos a dos:

$$\overline{\vert\ \langle \forall\, i\ : 0 \leq i < \#xs :\ \langle \forall\, j\ : 0 \leq j < \#xs :\ xs.i = xs.j \,\rangle \,\rangle}$$

Pero también podríamos expresar que todos son iguales diciendo que cada uno de ellos es igual a uno fijo, por ejemplo, el primero (sabemos que existe pues suponemos que la lista no es vacía):

$$\overline{\vert\ \langle \forall\, i\ : 0 \leq i < \#xs :\ xs.i = xs.0 \,\rangle}$$

Ejemplo 9.7
Sea xs una lista de enteros. "xs está ordenada en forma creciente".

Si una lista está ordenada en forma creciente, cada elemento debe ser mayor que el anterior:

$$\overline{\vert\ \langle \forall\, i\ : 0 < i < \#xs :\ xs.i > xs.(i-1) \,\rangle}$$

Notar que fue necesario excluir el cero del rango de cuantificación, para que el término esté siempre definido.

También puede especificarse este problema expresando la relación de desigualdad correspondiente entre cualquier par de índices:

$$\overline{\vert\ \langle \forall\, i\ : 0 \leq i < \#xs :\ \langle \forall\, j\ : i < j < \#xs :\ xs.i < xs.j \,\rangle \,\rangle}$$

Ejemplo 9.8
"La función $f : Int \mapsto [Num] \mapsto Bool$ determina si el k-ésimo elemento de la lista xs aloja el mínimo valor de xs".

Notemos que la función f debe aplicarse a dos argumentos, una lista y un entero que indica una posición en la lista. Para que la evaluación tenga sentido, el entero debe estar comprendido en el rango que corresponde al tamaño de la lista. Entonces podemos escribir:

$$\left\vert \begin{array}{l} f : Num \mapsto [Num] \mapsto Bool \\ \hline f.k.xs = (0 \leq k < \#xs \wedge xs.k = \langle \text{Min}\ i\ : 0 \leq i < \#xs :\ xs.i \,\rangle) \end{array} \right.$$

o bien

$$\left\vert \begin{array}{l} f : Num \mapsto [Num] \mapsto Bool \\ \hline f.k.xs = (0 \leq k < \#xs \wedge \langle \forall\, i\ : 0 \leq i < \#xs :\ xs.k \leq xs.i \,\rangle) \end{array} \right.$$

Ejemplo 9.9
"La función $minout : [Nat] \mapsto Nat$ selecciona el menor natural que no está en xs".

La función *minout* extrae el mínimo de un conjunto, por lo que resulta claro que la especificación usará el cuantificador Min. El conjunto del cual se extrae el mínimo deberá especificarse en el rango de la cuantificación:

$$\begin{array}{|l} minout : [Nat] \mapsto Nat \\ \hline minout.xs = \langle \text{Min } i \ : 0 \leq i \wedge \langle \forall j \ : 0 \leq j < \#xs : \ xs.j \neq i \rangle : \ i \rangle \end{array}$$

Ejemplo 9.10
"La función $lmax : [A] \mapsto Num$ calcula la longitud del máximo intervalo de la forma $[0..k]$ que verifica $xs.i = 0$ para todo $i \in [0..k]$".

Podemos dar una especificación similar a la del ejemplo anterior:

$$\begin{array}{|l} lmax : [A] \mapsto Num \\ \hline lmax.xs = \langle \text{Max } k \ : 0 \leq k \leq \#xs \wedge \langle \forall i \ : 0 \leq i < k : \ xs.i = 0 \rangle : \ k \rangle \end{array}$$

Esto también se puede escribir usando listas en lugar del entero k:

$$\begin{array}{|l} lmax : [A] \mapsto Num \\ \hline lmax.xs = \langle \text{Max } as, bs \\ \qquad\qquad : xs = as +\!\!+ bs \wedge \langle \forall i \ : 0 \leq i < \#as : \ xs.i = 0 \rangle : \ \#as \rangle \end{array}$$

Ejemplo 9.11
"Dada una lista de números, la función P determina si algún elemento de la lista es igual a la suma de todos los anteriores a él".

Evidentemente, la función P debe tomar como argumento una lista y devolver un booleano. El problema se puede especificar fácilmente con la ayuda del cuantificador existencial y de la suma:

$$\begin{array}{|l} P : [Num] \mapsto Bool \\ \hline P.xs = \langle \exists i \ : 0 \leq i < \#xs : \ xs.i = \langle \sum j \ : 0 \leq j < i : \ xs.j \rangle \rangle \end{array}$$

Ejemplo 9.12
"Dado un polinomio $p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$, la función ev realiza la evaluación del polinomio en un valor dado de la variable independiente".

En principio, la función ev se aplica a dos argumentos, el polinomio $p(x)$ y el punto en el que se desea evaluarlo. Para especificar la función debemos, entonces, elegir una manera adecuada de representar el polinomio. Como un polinomio queda unívocamente determinado por sus coeficientes, una posibilidad es formar con dichos coeficientes una lista: $xs = [a_0, a_1, \ldots, a_{n-1}, a_n]$. Más aún, cada lista de enteros puede pensarse como la lista de coeficientes de un polinomio. Así, podemos especificar ev como:

$$\begin{array}{|l} ev : [Num] \mapsto Num \mapsto Bool \\ \hline ev.xs.y = \langle \sum i \ : 0 \leq i < \#xs : \ xs.i * y^i \ \rangle \end{array}$$

Ejemplo 9.13
"Dada una lista de booleanos, la función *bal* debe indicar si en la lista hay igual cantidad de elementos $True$ que $False$".

El cuantificador aritmético N nos permite contar la cantidad de $True$ y de $False$ que hay en la lista. Lo que nos interesa es el resultado de la comparación de estas dos cantidades:

$$\begin{array}{|l} bal : [Bool] \mapsto Bool \\ \hline bal.xs = (\langle N\, i \ : 0 \leq i < \#xs : \ xs.i = True \ \rangle = \\ \qquad\qquad \langle N\, i \ : 0 \leq i < \#xs : \ xs.i = False \ \rangle) \end{array}$$

o equivalentemente

$$\begin{array}{|l} bal : [Bool] \mapsto Bool \\ \hline bal.xs = (\langle N\, i \ : 0 \leq i < \#xs : \ xs.i \ \rangle = \langle N\, i \ : 0 \leq i < \#xs : \ \neg xs.i \ \rangle) \end{array}$$

Ejemplo 9.14
"Dada una lista xs de booleanos, la función *balmax* debe indicar la longitud del máximo segmento de xs que satisface *bal*".

Dada una lista xs, un segmento de ella es una lista cuyos elementos están en xs, en el mismo orden y consecutivamente. Por ejemplo: si $xs = [2, 4, 6, 8]$, entonces $[2, 4]$, $[4, 6, 8]$, $[\,]$, son segmentos, mientras que $[4, 2]$ o $[2, 6, 8]$ no lo son.

En problemas como este, suele ser conveniente expresar la lista como una concatenación. Si escribimos $xs = as +\!\!+ bs +\!\!+ cs$, entonces as, bs y cs son segmentos de xs, por lo que podemos usarlos para expresar las condiciones que necesitamos que se satisfagan. Observemos que de los tres mencionados, el *más general* es bs, puesto que as necesariamente comienza en la posición cero de xs, mientras que cs termina necesariamente en la última posición de xs. El segmento bs puede estar en

cualquier parte, pues tanto as como cs pueden ser vacíos. Parece razonable, pues, usar bs para especificar el problema. Además de pedir que sea segmento, necesitamos que satisfaga bal; esto debe figurar en el rango. En el término debemos poner lo que nos interesa de cada elemento considerado en el rango, esto es, la longitud. Como queremos obtener la máxima longitud, usamos el cuantificador Max. Así, podemos especificar el problema de la siguiente manera:

$$\begin{array}{|l} balmax : [Bool] \mapsto Num \\ \hline balmax.xs = \langle \text{Max } bs : \langle\, \exists\, as, cs : : xs = as +\!\!+ bs +\!\!+ cs \,\rangle \wedge bal.bs : \#bs \,\rangle \end{array}$$

o equivalentemente (pero más simple)

$$\begin{array}{|l} balmax : [Bool] \mapsto Num \\ \hline balmax.xs = \langle \text{Max } as, bs, cs \ : xs = as +\!\!+ bs +\!\!+ cs \wedge bal.bs : \ \#bs \,\rangle \end{array}$$

9.3 Ejercicios

Ejercicio 9.1
Expresar en lenguaje formal las oraciones entre comillas:

1. Dado un entero $N > 0$: "x es el mayor entero que vale a lo sumo N y es una potencia de 2".
2. Dado un conjunto B: "si B no es vacío, x es el mayor elemento de B; en caso contrario, x es igual a 0".

Ejercicio 9.2
Sea xs una lista no vacía, con elementos enteros. Expresar en lenguaje formal:

1. Suponiendo que $\#xs > 1$ y que existen al menos dos valores distintos en xs: "x es el segundo valor más grande de xs". (Ejemplo: si $xs = [3, 6, 1, 7, 6, 4, 7]$, debe resultar $x = 6$).
2. "s es la suma de los elementos de xs".
3. Dado un entero X: "n es la cantidad de veces que X aparece en xs".
4. "Todos los valores de xs son distintos".
5. "Si el 1 está en xs, entonces también el 0 está en xs".
6. "p es el producto de todos los valores positivos de xs".
7. Dado un entero x: "p es un booleano cuyo valor de verdad coincide con el de la afirmación $x \in xs$".

Ejercicio 9.3
Sea xs una lista no vacía, con elementos booleanos, tal que al menos un elemento de xs es $True$. Expresar en lenguaje formal:

1. "n es el menor entero tal que $xs.n \equiv True$".
2. "n indica la posición del último elemento de la lista que es equivalente a $True$"

Ejercicio 9.4
Sea xs una lista no vacía. Expresar las siguientes especificaciones en lenguaje común:

1. $\langle \forall\, i \ : 0 \leq i < N \ \wedge \ N \leq \#xs : \ xs.i \geq 0 \,\rangle$
2. $\langle \exists\, i \ : 0 < i < N \ \wedge \ N \leq \#xs : \ xs.(i-1) < xs.i \,\rangle$
3. $\langle \exists\, i \ : 0 \leq i < N \ \wedge \ N \leq \#xs : \ xs.i = 0 \,\rangle$
4. $\langle \forall\, p, q \ : 0 \leq p \ \wedge \ 0 \leq q \ \wedge \ p + q = N - 1 \ \wedge \ N \leq \#xs : \ xs.p = xs.q \,\rangle$

Ejercicio 9.5
Sea xs una lista no vacía.

1. Especificar: "xs es creciente".
2. Especificar y demostrar: "si xs escreciente, entonces el primer elemento es el menor".

Ejercicio 9.6
Especificar las siguientes funciones:

1. $f : [Num] \mapsto Bool$

   $f.xs$ determina si xs contiene igual cantidad de elementos pares que impares.

2. $cp : [Num] \mapsto Num$

   $cp.xs$ determina la cantidad de números pares que contiene xs.

3. $g : Num \mapsto [Num] \mapsto Bool$

   $g.k.xs$ determina si el k-ésimo elemento de xs aloja el máximo valor de xs.

4. $h : Num \mapsto [Num] \mapsto Bool$

   $h.k.xs$ determina si el k-ésimo elemento de xs aloja la primera ocurrencia del máximo valor de xs.

5. $nonulo : [Num] \mapsto Bool$

   $nonulo.xs$ es $True$ si y solo si xs no contiene elementos nulos.

6. $prod : Num \mapsto [Num] \mapsto Bool$

   $prod.x.xs$ es $True$ si y solo si x contiene el producto de los elementos de xs.

7. $meseta : [Num] \mapsto Num \mapsto Num \mapsto Bool$

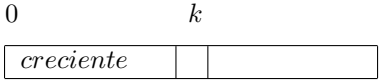
   $meseta.xs.i.j$ determina si todos los valores de la lista xs que están entre los índices i y j (ambos incluidos) son iguales.

8. $ordenada : [Num] \mapsto Num \mapsto Num \mapsto Bool$

   $ordenada.xs.i.j$ determina si la lista xs está ordenada entre los índices i y j (ambos incluidos). Notar que "ordenado" puede ser creciente o decreciente.

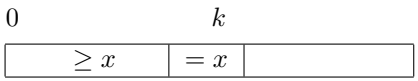
Ejercicio 9.7
Sea xs una lista de naturales. Especificar los siguientes problemas:

1. Los elementos de xs verifican:

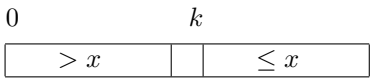


2. Dado un número natural x, los elementos de xs verifican:

   (i)



   (ii)



Capítulo 10

Inducción y recursión

> *Induction.* L'être aimé est desiré parce qu'un autre ou d'autres ont montré au sujet qu'il est désriable: tout spécial qu'il soit, le désir amoureux se découvre par induction.
>
> Roland Barthes:*Fragments d'un discourse amoureux*

Una técnica poderosa para demostrar propiedades sobre un dominio inductivo, como por ejemplo los naturales o las listas, es usar el *principio de inducción*. Este principio provee una técnica de demostración muy efectiva y fácil de usar. La idea de las demostraciones por inducción consiste en demostrar que una propiedad vale para los elementos "más chicos" del dominio (por ejemplo el cero en los naturales), y demostrar para un elemento arbitrario que si suponemos que la propiedad es cierta para todos los elementos más chicos que él entonces la propiedad también es satisfecha por ese elemento. Dado que todo elemento de un dominio inductivo puede ser construido a partir de elementos más simples, este procedimiento demuestra la propiedad para todos los elementos del dominio.

Por otro lado, es posible usar este principio para definir funciones de manera inductiva. Este tipo de definiciones consiste esencialmente en definir una función pudiendo usar para ello los valores de la función aplicada a elementos "más chicos" del dominio en cuestión. Esta forma de definir funciones, además de su significado operacional, tiene al principio de inducción (correspondiente al dominio inductivo considerado) como manera natural de demostrar propiedades. Este principio nos servirá no solo para demostrar propiedades de funciones ya definidas sino también para derivar definiciones de funciones a partir de especificaciones no necesariamente ejecutables o para derivar funciones a partir de otras ya definidas para obtener programas que cumplan con requisitos determinados, por ejemplo eficiencia en la ejecución.

10.1 Inducción matemática

El dominio inductivo más familiar es el de los números naturales. El principio de inducción asociado a este dominio se conoce con el nombre de **inducción matemática**. Sea por ejemplo la proposición

$$P.n : \langle \Sigma\, i \ : 0 \leq i \leq n : \ i \,\rangle = \frac{n * (n+1)}{2}$$

Una manera usual de demostrar que P es cierta para todo número natural es demostrar que vale $P.0$ y que suponiendo que vale para un número natural n dado, entonces también vale para el siguiente $(n+1)$. Esta última afirmación puede escribirse como

$$\langle\, \forall\, n \ : 0 \leq n : \ P.n \Rightarrow P.(n+1) \,\rangle \tag{10.1}$$

Ahora, si ambas cosas son ciertas, puede verse cómo construir una demostración de P para un número N dado: La demostración de $P.0$ es trivial. Usando la propiedad (10.1) e instanciación con $n = 0$ puede demostrarse por Modus Ponens y $P.0$ que vale $P.1$. Luego, instanciando (10.1) con $n = 1$, puede demostrarse de manera análoga $P.2$, etcétera, hasta llegar a $P.N$. Si tenemos suficiente tiempo y ganas, no importa cuan grande sea N, se llegará en algún momento.

El principio de inducción, el cual podría plantearse como un metateorema, permite inferir a partir de $P.0$ y de (10.1) que vale P para todo número natural.

Consideremos de nuevo el ejemplo. Demostraremos primero el *caso base* $P.0$.

$$\begin{aligned}
& P.0 \\
\equiv\ & \{\ \text{instanciación}\ \} \\
& \langle \textstyle\sum i \ : 0 \leq i \leq 0 : \ i \,\rangle = \dfrac{0 * (0+1)}{2} \\
\equiv\ & \{\ \text{aritmética}\ \} \\
& \langle \textstyle\sum i \ : i = 0 : \ i \,\rangle = \dfrac{0}{2} \\
\equiv\ & \{\ \text{rango unitario, aritmética}\ \} \\
& 0 = 0 \\
\equiv\ & \{\ \text{reflexividad de} = \} \\
& True
\end{aligned}$$

Demostraremos ahora el *caso inductivo*, suponiendo $P.n$ válida demostraremos $P.(n+1)$.

$$P.(n+1)$$

$$
\begin{array}{ll}
\equiv & \{ \text{ instanciación } \} \\
& \langle \sum i \ : 0 \leq i \leq n+1 : \ i \rangle = \dfrac{(n+1) * ((n+1)+1)}{2} \\
\equiv & \{ \text{ aritmética } \} \\
& \langle \sum i \ : 0 \leq i \leq n \vee i = n+1 : \ i \rangle = \dfrac{(n+1) * (n+2)}{2} \\
\equiv & \{ \text{ partición de rango } \} \\
& \langle \sum i \ : 0 \leq i \leq n : \ i \rangle + \langle \sum i \ : i = n+1 : \ i \rangle = \dfrac{(n+1) * (n+2)}{2} \\
\equiv & \{ \text{ rango unitario } \} \\
& \langle \sum i \ : 0 \leq i \leq n : \ i \rangle + n + 1 = \dfrac{(n+1) * (n+2)}{2} \\
\equiv & \{ \text{ suponemos } P.n \text{ (hipótesis inductiva), Leibniz } \} \\
& \dfrac{n * (n+1)}{2} + n + 1 = \dfrac{(n+1) * (n+2)}{2} \\
\equiv & \{ \text{ aritmética } \} \\
& \dfrac{n * (n+1) + 2 * (n+1)}{2} = \dfrac{(n+1) * (n+2)}{2} \\
\equiv & \{ \text{ aritmética } \} \\
& True
\end{array}
$$

El formato de demostración puede hacerse más (o menos) económico, como se ilustrará a continuación. El interés en estos formatos no está solo en la elegancia y la consiguiente facilidad para comprender o realizar las demostraciones, sino que también se van a usar estos formatos para construir nuevos objetos (en general funciones) los cuales van a satisfacer ciertas propiedades predeterminadas. La demostración (inductiva) de que el objeto construido satisfará estas propiedades se construye paso a paso junto con el objeto. Para ejemplificar esto introduciremos primero una definición formal del principio de inducción.

Con la notación que estamos usando, el principio de inducción puede escribirse de la siguiente manera:

$$
\begin{array}{ll}
& \langle \forall i \ : 0 \leq i : \ P.i \rangle \\
\Leftarrow & \{ \text{ inducción } \} \\
& P.0 \wedge \langle \forall j \ : 0 \leq j : \ P.j \Rightarrow P.(j+1) \rangle
\end{array}
$$

En el contexto de nuestro formalismo básico el principio de inducción se usará de dos maneras: para demostrar que una definición recursiva dada satisface cierta definición explícita, o -inversamente- para encontrar una definición recursiva correspondiente a una definición explícita dada. Veamos esto con un ejemplo.
Sea f definida recursivamente por

$$\left|\begin{array}{rcl} f.0 & \doteq & 0 \\ f.(i+1) & \doteq & f.i + 2 * i + 1 \end{array}\right.$$

Queremos demostrar que f satisface la propiedad $\langle\, \forall\, i \,:\, i \in Nat :\ f.i = i^2 \,\rangle$.

La primera demostración va a usar el principio de inducción en su forma "cruda".

$$\begin{array}{ll} & \langle\, \forall\, i \,:\, i \in Nat :\ f.i = i^2 \,\rangle \\ \Leftarrow & \{\ \text{inducción}\ \} \\ & f.0 = 0^2 \wedge \langle\, \forall\, j \,:\, j \in Nat :\ (f.j = j^2 \Rightarrow f.(j+1) = (j+1)^2) \,\rangle \\ \equiv & \{\ \text{álgebra}\ \} \\ & f.0 = 0 \wedge \langle\, \forall\, j \,:\, j \in Nat :\ (f.j = j^2 \Rightarrow f.(j+1) = j^2 + 2 * j + 1) \,\rangle \\ \Leftarrow & \{\ \text{Leibniz (axioma)}\ \} \\ & f.0 = 0 \wedge \langle\, \forall\, j \,:\, j \in Nat :\ (f.j = j^2 \Rightarrow f.(j+1) = f.j + 2 * j + 1) \,\rangle \\ \Leftarrow & \{\ (p \Rightarrow q) \Leftarrow q\text{, monotonía}\ \} \\ & f.0 = 0 \wedge \langle\, \forall\, j \,:\, j \in Nat :\ f.(j+1) = f.j + 2 * j + 1 \,\rangle \\ \equiv & \{\ \text{definición de } f\ \} \\ & True \end{array}$$

Como se vio en el primer ejemplo, en los problemas que usan inducción, suele separarse el "caso base" del resto. Lo anterior podría escribirse de la siguiente manera:

Caso base	**Resto**
$f.0 = 0^2$	$f.(i+1) = (i+1)^2$
$\equiv$ { álgebra }	$\equiv$ { álgebra }
$f.0 = 0$	$f.(i+1) = i^2 + 2 * i + 1$
$\equiv$ { definición de f }	$\equiv$ { hipótesis inductiva }
$True$	$f.(i+1) = f.i + 2 * i + 1$
	$\equiv$ { definición de f }
	$True$

Obviamente sería necesario demostrar un metateorema que indique cómo reconstruir una demostración "pura" a partir de la anterior. No vamos a demostrarlo aquí, dejando para el lector interesado esta tarea. Para el ejemplo considerado la traducción realizada es obvia.

Una forma más compacta de escribir esta demostración es evitar repetir el lado izquierdo de la igualdad en cada paso, dado que este no cambia. La demostración quedará por lo tanto de la siguiente forma:

Caso base	**Paso inductivo**
$f.0$	$f.(i+1)$
$= \{$ definición de f $\}$	$= \{$ definición de f $\}$
0	$f.i + 2 * i + 1$
$= \{$ álgebra $\}$	$= \{$ hipótesis inductiva $\}$
0^2	$i^2 + 2 * i + 1$
	$= \{$ álgebra $\}$
	$(i+1)^2$

Por lo tanto f satisface la identidad $\langle \forall\, i \ : i \in Nat : \ f.i = i^2 \rangle$. Lo que hemos hecho se llama **verificación de programa**, dado que estamos comprobando que un programa dado (una definición recursiva) satisface su especificación.

Si bien la técnica de verificación es muy útil, dado que permite tener una certeza mucho mayor (tanto como lo permite la matemática) de que el programa construido es correcto, queda por verse cómo se encuentra este programa. Por otro lado, la demostración de que un programa satisface una especificación dada no es una tarea trivial, por lo que es conveniente realizarla, al menos en parte, a medida que se construye el programa. Esta construcción conjunta de programa y demostración tiene la ventaja adicional de que la misma demostración nos va guiando en la construcción del programa.

Volviendo a nuestro ejemplo, tomemos ahora la propiedad demostrada antes a modo de especificación de f:

$$\langle \forall\, i \ : i \in Nat : \ f.i = i^2 \rangle \ .$$

Obviamente, existe una forma trivial de encontrar una función que satisfaga la especificación:

$$f.x = x * x \ .$$

Si bien es inmediato que esta definición satisface la especificación y que además es una definición extremadamete eficiente, a modo de ejemplo trataremos de encontrar una definición recursiva de f. Para hacer esto, algunos de los pasos de derivación (en general los primeros) van a usar la especificación como si fuera cierta. En realidad lo que se quiere construir es, además del programa, una demostración de que efectivamente el programa satisface dicha especificación. Puede pensarse por analogía que una especificación de una función f es una ecuación a resolver con incógnita f. Luego el proceso de derivación consistirá en "despejar" f

para saber qué valor toma (obviamente puede tener varias soluciones posibles). El proceso de despejar nos asegura que si ahora reemplazamos el valor obtenido en la ecuación original tendremos un enunciado verdadero. Veamos esto a través del ejemplo.

Caso base

$$
\begin{array}{ll}
 & f.0 \\
= & \{ \text{ especificación de } f \ \} \\
 & 0^2 \\
= & \{ \text{ álgebra } \} \\
 & 0
\end{array}
$$

Paso inductivo

$$
\begin{array}{ll}
 & f.(i+1) \\
= & \{ \text{ especificación de } f \ \} \\
 & (i+1)^2 \\
= & \{ \text{ álgebra } \} \\
 & i^2 + 2 * i + 1 \\
= & \{ \text{ hipótesis inductiva } \} \\
 & f.i + 2 * i + 1
\end{array}
$$

Por lo tanto, definiendo recursivamente

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

(la cual es casualmente la función con la cual comenzamos el ejemplo) se satisface la propiedad requerida. Lo que acabamos de hacer se llama **derivación de programa**, dado que hemos construido un programa que satisface la especificación y la demostración de que lo hace. Para ver esto último, nótese que es inmediato cómo recuperar la tercera demostración por inducción de que la función f satisface la propiedad requerida, simplemente invirtiendo la demostración, poniendo en el segundo paso el último (y como justificación en lugar de "especificación de f" se pone "definición de f" o regla de desplegado para f) y como conclusión el segundo paso de la derivación. En general el primer paso de una derivación será la propiedad que finalmente se quiere que la función cumpla. En ejemplos más complejos el formato de derivación puede ser ligeramente diferente.

En general, nos concentraremos en el problema de la derivación de programas más que en la verificación de los mismos. Veamos otro ejemplo sencillo de derivación: sea fac una función que satisface la propiedad

$$\langle \forall n \ : n \in Nat : \ fac.n = \langle \prod i \ : 0 < i \leq n : \ i \rangle \rangle \ ,$$

es decir, fac calcula el factorial de un número. Se desea una definición recursiva

de fac.

Caso base

$$
\begin{array}{ll}
 & fac.0 \\
= & \{ \text{ especificación de } f \ \} \\
 & \langle \prod i \ : 0 < i \leq 0 : \ i \, \rangle \\
= & \{ \text{ rango vacío } \} \\
 & 1
\end{array}
$$

Paso inductivo

$$
\begin{array}{ll}
 & fac.(n+1) \\
= & \{ \text{ especificación de } f \ \} \\
 & \langle \prod i \ : 0 < i \leq n+1 : \ i \, \rangle \\
= & \{ \text{ partición de rango } \} \\
 & \langle \prod i \ : 0 < i \leq n : \ i \, \rangle * \\
 & \quad \langle \prod i \ : n < i \leq n+1 : \ i \, \rangle \\
= & \{ \text{ hipótesis inductiva; rango unitario } \} \\
 & fac.n * (n+1)
\end{array}
$$

Por lo tanto una definición recursiva de fac es:

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

10.2 Inducción generalizada

Existe una generalización natural del principio de inducción matemática, la cual consiste en usar para demostrar el caso inductivo $P.(n+1)$ no solo $P.n$ sino también $P.i$, con $0 \leq i < n$. La justificación de que demostrando esta propiedad (más débil) y el caso base implica que P vale para todo natural es la misma que para la inducción presentada antes, dado que al demostrar $P.(n+1)$ ya tenemos demostrados todos los anteriores, no solo $P.n$. Afortunadamente ambos principios de inducción son equivalentes.

El principio de inducción generalizada puede formularse como sigue

$$
\begin{array}{ll}
 & \langle \forall i \ : 0 \leq i : \ P.i \, \rangle \\
\Leftarrow & \{ \text{ inducción } \} \\
 & \langle \forall j \ : 0 \leq j : \ \langle \forall k \ : 0 \leq k < j : \ P.k \, \rangle \Rightarrow P.j \, \rangle
\end{array}
$$

Nótese que no hace falta separar el caso base, dado que está implícito en el rango vacío del cuantificador interno para el caso $n = 0$.

Ejercicio 10.1
Demostrar que el principio de inducción generalizada es equivalente al siguiente

$$\langle \forall i \ : 0 \leq i : \ P.i \, \rangle$$

$$
\begin{array}{ll}
\Leftarrow & \{ \text{ inducción } \} \\
& P.0 \wedge \langle \forall\, j \ : 0 \leq j : \ \langle \forall\, k \ : 0 \leq k \leq j : \ P.k \,\rangle \Rightarrow P.(j+1) \,\rangle
\end{array}
$$

Ejemplo 10.1
La sucesión de Fibonacci se genera de la siguiente manera:

$$
\left\lceil
\begin{array}{rcl}
fib.0 & \doteq & 0 \\
fib.1 & \doteq & 1 \\
fib.(n+2) & \doteq & fib.n + fib(n+1)
\end{array}
\right.
$$

Esta sucesión tiene propiedades muy interesantes, algunas de las cuales se presentan en los ejercicios. Vamos a demostrar ahora una muy simple:

$$\langle \forall\, n \ : 0 \leq n : \ fib.n < 2^n \,\rangle$$

Dada la definición de fib, parece conveniente considerar dos casos base, cuando $n = 0$ y cuando $n = 1$. El caso inductivo podrá entonces considerarse para $n \geq 2$.

Ejercicio 10.2
Demostrar que el principio de inducción generalizada es equivalente al siguiente

$$
\begin{array}{ll}
& \langle \forall\, i \ : 0 \leq i : \ P.i \,\rangle \\
\Leftarrow & \{ \text{ inducción } \} \\
& P.0 \wedge P.1 \wedge \langle \forall\, j \ : 0 \leq j : \ \langle \forall\, k \ : 0 \leq k \leq j+1 : \ P.k \,\rangle \Rightarrow P.(j+2) \,\rangle
\end{array}
$$

Retomando el ejemplo, los dos casos base se dejan como ejercicio (fácil) para el lector. Consideremos el caso inductivo (usaremos inducción generalizada). Sea $P.n$ la proposición $fib.n < 2^n$. Partiremos de $fib.(n+2)$ y llegaremos a 2^{n+2}.

$$
\begin{array}{ll}
& fib.(n+2) \\
= & \{ \text{ definición de } fib \ \} \\
& fib.n + fib.(n+1) \\
< & \{ \text{ hipótesis inductiva } (P.n) \text{ , álgebra } \} \\
& 2^n + fib.(n+1) \\
< & \{ \text{ hipótesis inductiva } (P.(n+1)) \text{ , álgebra } \} \\
& 2^n + 2^{n+1} \\
< & \{ \text{ la exponenciación es creciente } \} \\
& 2^{n+1} + 2^{n+1} \\
= & \{ \text{ álgebra } \} \\
& 2^{n+2}
\end{array}
$$

10.3 Ejercicios

Ejercicio 10.3
Probar:

$$fib.n = \frac{\Phi^n - \hat{\Phi}^n}{\sqrt{5}}$$

donde $\Phi = \dfrac{1+\sqrt{5}}{2}$ y $\hat{\Phi} = \dfrac{1-\sqrt{5}}{2}$.

Ejercicio 10.4
Determinar el error en el siguiente razonamiento, el cual demuestra que un grupo cualquiera de gatos está compuesto solo por gatos negros.

> La demostración es por inducción en el número de gatos de un grupo dado. Para el caso de 0 gatos el resultado es inmemdiato, dado que obviamente todo gato del grupo es negro. Consideremos ahora un grupo de $n+1$ gatos. Tomemos un gato cualquiera del grupo. El grupo de los gatos restantes tiene n gatos, y por hipótesis inductiva son todos negros. Luego tenemos n gatos negros y uno que no sabemos aún de qué color es. Intercambiemos ahora el gato que sacamos con uno del grupo, el cual sabemos que es negro. Luego tenemos un gato negro y un grupo de n gatos. Por hipótesis inductiva los n gatos son negros, pero como el que tenemos aparte también es negro, entonces tenemos que los $n+1$ gatos son negros.

Ejercicio 10.5
Determinar el error en el siguiente razonamiento, el cual demuestra que en un grupo cualquiera de personas si una es comunista, todas lo son.

> Lo demostraremos por inducción en el número de personas del grupo. El caso de 0 personas es trivialmente cierto (el antecedente de la implicación es falso). Para un grupo de 1 persona también es inmediato, dado que si esta persona es comunista se satisface el consecuente de la implicación y si no lo es entonces el antecedente es falso. Tomemos un grupo de $n+2$ personas en el cual asumimos que alguna es comunista (en caso contrario la implicación vale trivialmente), con n mayor o igual a cero. Sacamos una persona cualquiera del grupo. Si esta persona es comunista, entonces la intercambiamos con cualquier otra del grupo para asegurarnos que en el grupo de $n+1$ personas haya al menos una que sea comunista. Luego, por hipótesis inductiva tenemos un grupo de $n+1$ personas todas comunistas y una afuera de la cual no conocemos aún sus simpatías políticas. Intercambiamos entonces a esta persona con cualquiera de los que ya sabemos que son comunistas, y nos queda un grupo de $n+1$ personas de las cuales todas

menos una son comunistas. Por hipótesis inductiva entonces tienen que ser todas comunistas, y como la que está aparte también lo es, entonces son todos comunistas.

Ejercicio 10.6
"Demuestre" usando ideas análogas a las de los dos ejercicios anteriores que si en un grupo de personas hay dos con la misma cantidad de pelos en la cabeza, entonces todas las personas en el grupo tienen la misma cantidad de pelos en la cabeza.

Ejercicio 10.7 (Principio de las casillas)
Demostrar que dados $xs : [Nat]$ y $M : Nat$, $M > 0$,

$$\langle \sum i \ : 0 \leq i < \#xs : \ xs.i \rangle < M \ \vee \ \langle \exists i \ : 0 \leq i < \#xs : \ xs.i * \#xs \geq M \rangle$$

Ejercicio 10.8
Sean $xs : [Nat]$ y $M : Nat$. Demostrar:

1. $\langle \exists i \ : 0 \leq i < \#xs : \ xs.i \geq M \rangle \equiv \langle \text{Max } i \ : 0 \leq i < \#xs : \ xs.i \rangle \geq M$
2. $\langle \forall i \ : 0 \leq i < \#xs : \ xs.i \geq M \rangle \equiv \langle \text{Min } i \ : 0 \leq i < \#xs : \ xs.i \rangle \geq M$

Ejercicio 10.9
Demostrar que las siguientes proposiciones son equivalentes:

1. $\langle \forall n \ : n \geq 0 : \ P.n \rangle \equiv P.0 \wedge \langle \forall n \ : n \geq 0 : \ P.n \Rightarrow P.(n+1) \rangle$
2. $\langle \forall n \ : n \geq 0 : \ P.n \rangle \equiv \langle \forall n \ : n \geq 0 : \ P.n \vee \langle \exists i \ : 0 \leq i < n : \ \neg P.i \rangle \rangle$
3. Dado $S \subset Nat$,
   $\langle \exists x \ : x \geq 0 : \ x \in S \rangle \equiv \langle \exists x \ : x \geq 0 : \ x \in S \wedge \langle \forall y \ : 0 \leq y < x : \ y \notin S \rangle \rangle$

Ejercicio 10.10 (Torres de Hanoi)
Se tienen tres postes -0, 1 y 2- y n discos de distinto tamaño. En la situación inicial se encuentran todos los discos ubicados en el poste 0 en forma decreciente según el tamaño (el más grande en la base).

El problema consiste en llevar todos los discos al poste 2, con las siguientes restricciones:

(i) Se puede mover solo un disco por vez (el que está más arriba en algún poste).

(ii) No se puede apoyar un disco sobre otro de menor tamaño.

Sea $B = \{0, 1, 2\}$. Definir una función $f : B \mapsto B \mapsto B \mapsto Nat \mapsto [(B, B)]$ tal que $f.a.b.c.n$ calcule la secuencia de movimientos para llevar n discos del poste a al poste c, pasando por el poste b.

Ejemplo: $f.0.1.2.2 = [(0, 1), \ (0, 2), \ (1, 2)]$

Capítulo 11

Técnicas elementales para la programación

The Road goes ever on and on
    Down from the door where it began.
Now far ahead the road has gone,
    And I must follow, if I can,
Pursuing it with eager feet,
    Until it joins some larger way
Where many paths and errands meet.
    And wither then? I cannot say.

J.R.R.Tolkien:
The Lord of the Rings

Si bien un paso importante en la solución de un problema planteado es escribir una especificación correcta del mismo, una vez que se tiene esta hay que manipularla adecuadamente, a fin de transformarla en algo que sea factible de implementar en un programa eficiente. Veremos a continuación algunas técnicas que nos permitirán llevar esto a cabo.

11.1 Definiciones recursivas

La técnica más usada –y sobre la cual reposan las otras– fue ya presentada en el capítulo sobre inducción y consiste en obtener una definición recursiva de una

función a partir de su especificación. Consideraremos aquí otro ejemplo simple para repasar el funcionamiento de esta técnica.

Ejemplo 11.1
Dada una lista de enteros se desea obtener su suma. Primero especificaremos formalmente esta función.

$$sum.xs = \langle \sum i \ : 0 \leq i < \#xs : \ xs.i \, \rangle$$

A partir de esta especificación calcularemos inductivamente un definición recursiva. Primero consideraremos el caso base.

Caso base $(xs = [\,])$

$$\begin{array}{ll} & sum.[\,] \\ = & \{ \text{ especificación de } sum \ \} \\ & \langle \sum i \ : 0 \leq i < \#[\,] : \ [\,].i \, \rangle \\ = & \{ \text{ definición de } \# \ \} \\ & \langle \sum i \ : 0 \leq i < 0 : \ [\,].i \, \rangle \\ = & \{ \text{ rango vacío } \} \\ & 0 \end{array}$$

El primer paso es el que diferencia una derivación de una verificación. En la demostración el objetivo es llegar a la fórmula obtenida en ese paso, mientras que el objetivo de la derivación es simplificar esta fórmula hasta obtener una expresión del formalismo básico que pueda tomarse como definición, en este caso la constante 0. La demostración de corrección de la definición obtenida habrá sido ya construida en la derivación misma.

Consideremos ahora el caso inductivo

Paso inductivo $(x \triangleright xs)$

$$\begin{array}{ll} & sum.(x \triangleright xs) \\ = & \{ \text{ especificación de } sum \ \} \\ & \langle \sum i \ : 0 \leq i < \#(x \triangleright xs) : \ (x \triangleright xs).i \, \rangle \\ = & \{ \text{ definición de } \# \ \} \\ & \langle \sum i \ : 0 \leq i < 1 + \#xs : \ (x \triangleright xs).i \, \rangle \\ = & \{ \text{ separación de un término } \} \\ & (x \triangleright xs).0 + \langle \sum i \ : 0 \leq i < \#xs : \ (x \triangleright xs).(i+1) \, \rangle \\ = & \{ \text{ definición de . } \} \\ & x + \langle \sum i \ : 0 \leq i < \#xs : \ xs.i \, \rangle \end{array}$$

$= \{$ hipótesis inductiva $\}$

$x + sum.xs$

El paso de separación de un término es un lema ya demostrado el cual consiste en la aplicación sucesiva de partición de rango, rango unitario y cambio de variables para volver a obtener un rango del tipo de la hipótesis inductiva.

A partir de esta derivación hemos obtenido el siguiente programa para sum.

$$
\begin{array}{|l}
sum : [Num] \mapsto Num \\
\hline
\quad\quad sum.[\,] \;\doteq\; 0 \\
\quad sum.(x \triangleright xs) \;\doteq\; x + sum.xs
\end{array}
$$

11.2 Reemplazo de constantes por variables

Consideremos el siguiente problema: dados N y X, calcular

$$\langle \sum i \;:\; 0 \leq i < N :\; X^i \rangle \quad .$$

Una forma de abordar este problema es usar la técnica de **reemplazo de constantes por variables**. El objetivo es cambiar el problema original por uno más general, que luego se resolverá de la manera en que hemos trabajado en la sección anterior.

Las constantes que aparecen en nuestro problema son 0, N y X, y cualquiera de ellas puede ser reemplazada por una variable, dando lugar a una nueva especificación del problema:

Si reemplazamos la constante N por la variable n, la nueva especificación es

$$\langle \forall n \;:\; n \in Nat :\; f.n = \langle \sum i \;:\; 0 \leq i < n :\; X^i \rangle \rangle \tag{11.1}$$

y la solución del problema estará dada por $f.N$.

Si reemplazamos la constante 0 por la variable n, la nueva especificación es

$$\langle \forall n \;:\; n \in Nat :\; f.n = \langle \sum i \;:\; n \leq i < N :\; X^i \rangle \rangle$$

y la solución del problema estará dada por $f.0$.

Si reemplazamos la constante X por la variable x, la nueva especificación es

$$\langle \forall x \;:\; x \in \mathbb{R} :\; f.x = \langle \sum i \;:\; 0 \leq i < N :\; x^i \rangle \rangle$$

y la solución del problema estará dada por $f.X$.

La ventaja de las dos primeras frente a la tercera es que al variar n sobre los naturales es posible aplicar el principio de inducción.

Trabajemos con la primera. Tratemos de encontrar una definición recursiva de f:

Caso base $(n = 0)$

$$
\begin{array}{ll}
 & f.0 \\
= & \{ \text{ especificación (11.1) } \} \\
 & \langle \sum i \ : 0 \leq i < 0 : \ X^i \rangle \\
= & \{ \text{ rango vacío } \} \\
 & 0
\end{array}
$$

Paso inductivo $(n + 1)$

$$
\begin{array}{ll}
 & f.(n+1) \\
= & \{ \text{ especificación (11.1) } \} \\
 & \langle \sum i \ : 0 \leq i < n+1 : \ X^i \rangle \\
= & \{ \text{ cálculo de predicados } \} \\
 & \langle \sum i \ : 0 \leq i < n \vee n \leq i < n+1 : \ X^i \rangle \\
= & \{ \text{ partición de rango; } (n \leq i < n+1) \equiv (i = n) \ \} \\
 & \langle \sum i \ : 0 \leq i < n : \ X^i \rangle + \langle \sum i \ : i = n : \ X^i \rangle \\
= & \{ \text{ hipótesis inductiva y rango unitario } \} \\
 & f.n + X^n
\end{array}
$$

Por lo tanto, una definición recursiva de f es

$$
\left|
\begin{array}{l}
f : Num \mapsto Num \\
\hline
\begin{array}{rcl}
f.0 & \doteq & 0 \\
f.(n+1) & \doteq & f.n + X^n
\end{array}
\end{array}
\right.
$$

Si el lenguaje de programación permite calcular X^n, hemos terminado. En caso contrario, es necesario desarrollar un programa para el cálculo de X^n.

11.3 Modularización

En algunas ocasiones, como en el ejemplo que se acaba de analizar, la solución del problema original requiere la solución de un "sub-problema". En estos casos suele ser conveniente no atacar ambos problemas simultáneamente, sino "por módulos", cada uno de los cuales debe ser independiente de los demás. Esta técnica se denomina **modularización**. En el ejemplo anterior, si modularizamos, podemos escribir la definición recursiva de f de la siguiente manera:

$$
\begin{array}{|l}
f : Num \mapsto Num \\
\hline
\begin{array}{rcl}
f.0 & \doteq & 0 \\
f.(n+1) & \doteq & f.n + g.n \\
 & & [\![g.i \doteq X^i]\!]
\end{array}
\end{array}
$$

La independencia de módulos significa que una vez resuelta g, debe quedar resuelta f. Derivemos la función g:

Caso base $(n = 0)$

$$
\begin{array}{ll}
 & g.0 \\
= & \{ \text{ especificación de } g \ \} \\
 & X^0 \\
= & \{ \text{ álgebra } \} \\
 & 1
\end{array}
$$

Paso inductivo $(n + 1)$

$$
\begin{array}{ll}
 & g.(n+1) \\
= & \{ \text{ especificación de } g \ \} \\
 & X^{(n+1)} \\
= & \{ \text{ álgebra } \} \\
 & X * X^n \\
= & \{ \text{ hipótesis inductiva } \} \\
 & X * g.n
\end{array}
$$

Es decir,

$$
\begin{array}{|l}
g : Num \mapsto Num \\
\hline
\begin{array}{rcl}
g.0 & \doteq & 1 \\
g.(n+1) & \doteq & X * g.n
\end{array}
\end{array}
$$

Teniendo una definición explícita de g, el problema inicial queda resuelto.

La definición recursiva de una función puede no ser única. En nuestro ejemplo, una forma diferente de dividir el rango conduce a otra definición recursiva de f. Veamos:

$$
\begin{array}{ll}
 & f.(n+1)
\end{array}
$$

$$
\begin{array}{ll}
= & \{ \text{ especificación (11.1) } \} \\
& \langle \sum i \ : 0 \leq i < n+1 : \ X^i \rangle \\
= & \{ \text{ cálculo de predicados } \} \\
& \langle \sum i \ : 0 \leq i < 1 \vee 1 \leq i < n+1 : \ X^i \rangle \\
= & \{ \text{ partición de rango } \} \\
& \langle \sum i \ : i = 0 : \ X^i \rangle + \langle \sum i \ : 1 \leq i < n+1 : \ X^i \rangle \\
= & \{ \text{ rango unitario } \} \\
& X^0 + \langle \sum i \ : 1 \leq i < n+1 : \ X^i \rangle \\
= & \{ \text{ reemplazando i por j+1 } \} \\
& X^0 + \langle \sum j \ : 1 \leq j+1 < n+1 : \ X^{j+1} \rangle \\
= & \{ \text{ álgebra } \} \\
& 1 + \langle \sum j \ : 0 \leq j < n : \ X * X^j \rangle \\
= & \{ \text{ distributividad de * con respecto a + } \} \\
& 1 + X * \langle \sum j \ : 0 \leq j < n : \ X^j \rangle \\
= & \{ \text{ hipótesis inductiva } \} \\
& 1 + X * f.n
\end{array}
$$

Por lo tanto otra definición recursiva de f es

$$
\begin{array}{|l}
f : Num \mapsto Num \\
\hline
\quad f.0 \doteq 0 \\
f.(n+1) \doteq 1 + X * f.n
\end{array}
$$

11.4 Uso de tuplas

En ocasiones, es posible reducir el costo de un programa implementando una especificación más eficiente. Una manera de lograr esto es usar la denominada **técnica de pares** (más generalmente, técnica de tuplas; también suele llamársela inmersión de eficiencia). Veamos cómo funciona con un ejemplo:

Ejemplo 11.2
Consideremos la función que calcula la sucesión de Fibonacci derivada en el ejemplo 10.1 (pág. 168).

$$
\begin{array}{|l}
fib : Nat \mapsto Nat \\
\hline
\quad fib.0 \doteq 0 \\
\quad fib.1 \doteq 1 \\
fib.(n+2) \doteq fib.n + fib.(n+1)
\end{array}
$$

La evaluación de este programa para un numero n realiza un número de reducciones que crece exponencialmente con n. Esto implica que ya para números chicos se vuelve impracticable su evaluación. La ineficiencia radica en que en el caso inductivo se evalúa dos veces por separado la función fib. La evaluación del segundo sumando $fib.(n+1)$ calculará, entre otras cosas, $fib.n$, pero este valor será recalculado por el primer sumando. Este fenómeno hace que muchos valores se calculen un gran número de veces. La técnica de tuplas se basa en la idea de calcular ambos sumandos a la vez evitando así repetir cálculos. Esta mejora permite reducir drásticamente la complejidad de este programa.

Usando la técnica de tuplas se especifica

$$\frac{g : Nat \mapsto (Nat, Nat)}{g.n = (fib.n,\ fib.(n+1))}$$

La función podría definirse como $g.n \doteq (fib.n,\ fib.(n+1))$, pero nada se ganaría en eficiencia. Calcularemos ahora una definición recursiva directa para g. Nótese que es fácil definir ahora fib como $g.0$, el primer elemento del par.

El caso base es simple.

Caso base $(n = 0)$

$$\begin{aligned} & g.0 \\ = & \ \{ \text{ especificación de } g \ \} \\ & (fib.0, fib.1) \\ = & \ \{ \text{ definición de } fib \ \} \\ & (0, 1) \end{aligned}$$

El caso inductivo requiere más cuidado

Paso inductivo $(n+1)$

$$\begin{aligned} & g.(n+1) \\ = & \ \{ \text{ especificación de } g \ \} \\ & (fib.(n+1), fib.(n+2)) \\ = & \ \{ \text{ definición de } fib \ \} \\ & (fib.(n+1), fib.n + fib.(n+1)) \end{aligned}$$

en este punto de la derivación vemos el motivo de la ineficiencia de la definición original en la repetición del cálculo de $fib.(n+1)$. Para evitar esta ineficiencia hacemos uso de definiciones locales que nos permiten reemplazar repetidas veces un valor. Nótese que el uso de pattern matching da lugar a una definición muy fácil de leer.

$$
\begin{array}{ll}
 & (fib.(n+1), fib.n + fib.(n+1)) \\
= & \{ \text{ introducción de } a, b \text{ como definiciones locales } \} \\
 & (b, a+b) \\
 & \quad |[a \doteq fib.n, \\
 & \quad\ b \doteq fib.(n+1)]| \\
= & \{ \text{ igualdad de pares } \} \\
 & (b, a+b) \\
 & \quad |[(a,b) \doteq (fib.n, fib.(n+1))]| \\
= & \{ \text{ hipótesis inductiva } \} \\
 & (b, a+b) \\
 & \quad |[(a,b) \doteq g.n]|
\end{array}
$$

Por lo tanto la definición recursiva de g es

$$
\begin{array}{|l}
\hline
g : Nat \mapsto (Nat, Nat) \\
\hline
\quad\quad g.0 \doteq (0,1) \\
\quad g.(n+1) \doteq (b, a+b) \\
\quad\quad\quad\quad\quad |[(a,b) \doteq g.n]| \\
\hline
\end{array}
$$

y la función que calcula Fibonacci queda definida como

$$
\begin{array}{|l}
\hline
fib : Nat \mapsto Nat \\
\hline
\quad fib.n \doteq g.n.0 \\
\hline
\end{array}
$$

donde $g.n.0$ es la primera componente del par devuelto por la función g.

Ejemplo 11.3
Continuamos ahora con el ejemplo de la suma de las primeras n potencias de un número dado (sección 11.2). En la derivación realizada usando modularización se obtiene un programa de complejidad cuadrática.

$$
\begin{array}{|l}
\hline
\quad f : Num \mapsto Num \\
\quad g : Num \mapsto Num \\
\hline
\quad\quad f.0 \doteq 0 \\
\quad f.(n+1) \doteq f.n + g.n \\
\quad\quad g.0 \doteq 1 \\
\quad g.(n+1) \doteq X * g.n \\
\hline
\end{array}
$$

Calculemos inductivamente una función h que satisfaga la siguiente especificación

$$\left|\begin{array}{l} h : Num \mapsto (Num, Num) \\ \hline h.n = (f.n, g.n) \end{array}\right.$$

Derivemos ahora de manera análoga al caso de Fibonacci una definición recursiva para h

Caso base $(n = 0)$

$$\begin{array}{ll} & h.0 \\ = & \{ \text{ especificación de } h \text{ } \} \\ & (f.0, g.0) \\ = & \{ \text{ definición de } f \text{ y } g \text{ } \} \\ & (0, 1) \end{array}$$

Paso inductivo $(n + 1)$

$$\begin{array}{ll} & h.(n+1) \\ = & \{ \text{ especificación de } h \text{ } \} \\ & (f.(n+1), g.(n+1)) \\ = & \{ \text{ definición de } f \text{ y } g \text{ } \} \\ & (f.n + g.n, X * g.n) \\ = & \{ \text{ introducimos } a,\ b \text{ } \} \\ & (a + b, X * b) \\ & \qquad [\![a \doteq f.n \\ & \qquad\ \ b \doteq g.n]\!] \\ = & \{ \text{ igualdad de pares } \} \\ & (a + b, X * b) \\ & \qquad [\![(a, b) \doteq (f.n, g.n)]\!] \\ = & \{ \text{ hipótesis inductiva } \} \\ & (a + b, X * b) \\ & \qquad [\![(a, b) \doteq h.n]\!] \end{array}$$

Por lo tanto

$$\left|\begin{array}{l} h : Num \mapsto (Num, Num) \\ \hline \begin{array}{rcl} h.0 & \doteq & (0, 1) \\ h.(n+1) & \doteq & (a + b, X * b) \\ & & [\![(a, b) \doteq h.n]\!] \end{array} \end{array}\right.$$

Observemos que lo que se hace es calcular al mismo tiempo las funciones f y g, de modo tal de tener ambas disponibles a medida que se van necesitando. De esta manera, $f.n = h.n.0$ (la primera componente del par) y se logra que el costo resulte lineal.

Ejercicio: Probar que el costo de calcular f usando la función h es lineal.

11.5 Generalización por abstracción

Una técnica muy poderosa es la **generalización por abstracción** (por algunos autores llamada inmersión). Esta técnica será usada para distintos objetivos en este curso. En este capítulo la utilizaremos como un medio para resolver una derivación cuando la hipótesis inductiva no puede aplicarse de manera directa. La idea consiste en buscar una especificación más general uno de cuyos casos particulares sea la función en cuestión. Para encontrar la generalización adecuada se introducen parámetros nuevos los cuales servirán para dar cuenta de las subexpresiones que no permiten aplicar la hipótesis inductiva en la derivación original.

Ejemplo 11.4
Consideremos un predicado que determina si todas las sumas parciales de una lista son mayores o iguales a cero. La especificación está dada por:

$$\begin{array}{|l} P : [Num] \mapsto Bool \\ \hline P.xs \equiv \langle\, \forall\, i \;::\; 0 \leq i < \#xs :\; sum.(xs{\uparrow}i) \geq 0 \,\rangle \end{array}$$

El caso base es simple y lo dejamos como ejercicio para el lector (el resultado es $True$ por rango vacío del cuantificador).

Calculemos el caso inductivo.

Paso inductivo $(x \triangleright xs)$

$$\begin{array}{ll} & P.(x \triangleright xs) \\ = & \{\ \text{especificación de } P\ \} \\ & \langle\, \forall\, i \;:\; 0 \leq i < \#(x \triangleright xs) :\; sum.((x \triangleright xs){\uparrow}i) \geq 0 \,\rangle \\ = & \{\ \text{definición de } \#\ \} \\ & \langle\, \forall\, i \;:\; 0 \leq i < 1 + \#xs :\; sum.((x \triangleright xs){\uparrow}i) \geq 0 \,\rangle \\ = & \{\ \text{separación de un término}\ \} \\ & sum.((x \triangleright xs){\uparrow}0) \geq 0 \wedge \langle\, \forall\, i \;:\; 0 \leq i < \#xs :\; sum.((x \triangleright xs){\uparrow}(i+1)) \geq 0 \,\rangle \\ = & \{\ \text{definición de}{\uparrow}\} \\ & sum.[\,] \geq 0 \wedge \langle\, \forall\, i \;:\; 0 \leq i < \#xs :\; sum.(x \triangleright (xs{\uparrow}i)) \geq 0 \,\rangle \end{array}$$

$= \{$ definición de sum $\}$
$0 \geq 0 \wedge \langle \forall i \ : 0 \leq i < \#xs : \ x + sum.(xs{\uparrow}i) \geq 0 \rangle$

En este punto notamos que podría aplicarse la hipótesis inductiva de no ser por la x que está sumando. No parece haber ninguna forma de eliminar x por lo cual se propone derivar la definición de una función generalizada Q especificada como sigue.

$$\boxed{\begin{array}{l} Q : Num \mapsto [Num] \mapsto Bool \\ \hline Q.n.xs \equiv \langle \forall i \ : 0 \leq i < \#xs : \ n + sum.(xs{\uparrow}i) \geq 0 \rangle \end{array}}$$

Esta nueva especificación está inspirada en la última expresión de la derivación de P. Sin embargo a partir de saber cómo resolver P en términos de Q puede olvidarse la derivación de P y y realizar la de Q de manera independiente, pese a que esta será en general similar a la anterior. Que esta nueva derivación pueda llevarse adelante dependerá de las propiedades del dominio en cuestión (en este caso los números, en particular la asociatividad de la suma) y puede no llegar a buen puerto o tener que volver a generalizarse a su vez. La programación es una actividad creativa y esto se manifiesta en este caso en la elección de las posibles generalizaciones.

Lo primero que debe determinarse antes de comenzar la derivación del nuevo predicado Q es si este efectivamente generaliza a P. Esto puede determinarse a partir de la especificación dado que $P.xs = Q.0.xs$ debido a que 0 es el neutro de la suma. Puede entonces definirse

$$\boxed{P.xs \doteq Q.0.xs}$$

y derivar directamente Q

Calculemos el caso base.

Caso base $(xs = [\,])$

$Q.n.[\,]$
$= \{$ especificación de Q $\}$
$\langle \forall i \ : 0 \leq i < \#[\,] : \ n + sum.([\,]{\uparrow}i) \geq 0 \rangle$
$= \{$ definición de $\#$ $\}$
$\langle \forall i \ : 0 \leq i < 0 : \ n + sum.([\,]{\uparrow}i) \geq 0 \rangle$
$= \{$ rango vacío $\}$
$True$

Calculamos ahora el caso inductivo.

Paso inductivo $(x \triangleright xs)$

$$
\begin{array}{ll}
 & Q.(x \triangleright xs) \\
= & \{ \text{ especificación de } Q \ \} \\
 & \langle\, \forall\, i \ : 0 \leq i < \#(x \triangleright xs) : \ n + sum.((x \triangleright xs)\!\uparrow\! i) \geq 0 \,\rangle \\
= & \{ \text{ definición de } \# \ \} \\
 & \langle\, \forall\, i \ : 0 \leq i < 1 + \#xs : \ n + sum.((x \triangleright xs)\!\uparrow\! i) \geq 0 \,\rangle \\
= & \{ \text{ separación de un término } \} \\
 & n + sum.((x \triangleright xs)\!\uparrow\! 0) \geq 0 \ \wedge \\
 & \quad \langle\, \forall\, i \ : 0 \leq i < \#xs : \ n + sum.((x \triangleright xs)\!\uparrow\! (i+1)) \geq 0 \,\rangle \\
= & \{ \text{ definición de} \uparrow \} \\
 & n + sum.[\,] \geq 0 \wedge \langle\, \forall\, i \ : 0 \leq i < \#xs : \ n + sum.(x \triangleright (xs\!\uparrow\! i)) \geq 0 \,\rangle \\
= & \{ \text{ definición de } sum \ \} \\
 & n \geq 0 \wedge \langle\, \forall\, i \ : 0 \leq i < \#xs : \ n + (x + sum.(xs\!\uparrow\! i)) \geq 0 \,\rangle \\
= & \{ \text{ asociatividad de } + \ \} \\
 & n \geq 0 \wedge \langle\, \forall\, i \ : 0 \leq i < \#xs : \ (n + x) + sum.(xs\!\uparrow\! i)) \geq 0 \,\rangle \\
= & \{ \text{ hipótesis inductiva } \} \\
 & n \geq 0 \wedge Q.(n + x)xs
\end{array}
$$

En esta derivación hemos enfatizado el uso de la propiedad asociativa de la suma lo cual en posteriores derivaciones dejaremos implícito.

El resultado completo de esta derivación es el siguiente programa

$$
\begin{array}{|rcl}
P & : & [Num] \mapsto Bool \\
Q & : & Num \mapsto [Num] \mapsto Bool \\
\hline
P.xs & \doteq & Q.0.xs \\
Q.n.[\,] & \doteq & True \\
Q.n.(x \triangleright xs) & \doteq & n \geq 0 \wedge Q.(n + x)xs
\end{array}
$$

Ejemplo 11.5

Continuamos aquí con el ejemplo de la suma de las primeras n potencias de un número dado (sección 11.2). Intentaremos esta vez mejorar la eficiencia de los programas obtenidos, tratando de obtener un programa con costo logarítmico. Para ello analizaremos dos casos: cuando el parámetro de f es par y cuando es impar. De esta forma intentaremos obtener una definición de la función f que utilice pattern matching sobre los casos $f.(2 * n)$ y $f.(2 * n + 1)$ (ver sección 7.6) en términos de la expresión $f.n$

Volvamos a considerar la especificación

$$\langle\, \forall\, n\ : n \in Nat :\ f.n = \langle \sum i\ : 0 \le i < n :\ X^i \,\rangle \,\rangle$$

para el primer caso.

$$\begin{array}{ll} & f.(2*n) \\ = & \{\ \text{especificación de } f\ \} \\ & \langle \sum i\ : 0 \le i < 2*n :\ X^i \,\rangle \\ = & \{\ \text{partición de rango}\ \} \\ & \langle \sum i\ : 0 \le i < n :\ X^i \,\rangle + \langle \sum i\ : n \le i < 2*n :\ X^i \,\rangle \\ = & \{\ \text{reemplazando } i \leftarrow n+i \text{ en el segundo término}\ \} \\ & \langle \sum i\ : 0 \le i < n :\ X^i \,\rangle + \langle \sum i\ : n \le n+i < 2*n :\ X^{n+i} \,\rangle \\ = & \{\ \text{álgebra}\ \} \\ & \langle \sum i\ : 0 \le i < n :\ X^i \,\rangle + \langle \sum i\ : 0 \le i < n :\ X^n * X^i \,\rangle \\ = & \{\ \text{distributividad de * con respecto a +}\ \} \\ & \langle \sum i\ : 0 \le i < n :\ X^i \,\rangle + X^n * \langle \sum i\ : 0 \le i < n :\ X^i \,\rangle \\ = & \{\ \text{álgebra}\ \} \\ & (1 + X^n) * \langle \sum i\ : 0 \le i < n :\ X^i \,\rangle \\ = & \{\ \text{hipótesis inductiva}\ \} \\ & (1 + X^n) * f.n \end{array}$$

Hasta aquí hemos logrado escribir $f.(2*n) = (1+X^n) * f.n$, pero aún nos queda X^n en la expresión. Para su cálculo no podemos usar la función g obtenida anteriormente (sección 11.3), pues su costo es lineal, lo que haría imposible que el costo total resulte logarítmico. Más adelante veremos un ejemplo que muestra cómo definir una función para el cálculo de la exponencial con costo logarítmico.

Probemos con una partición diferente del rango.

$$\begin{array}{ll} & f.(2*n) \\ = & \{\ \text{especificación de } f\ \} \\ & \langle \sum i\ : 0 \le i < 2*n :\ X^i \,\rangle \\ = & \{\ \text{partición de rango}\ \} \\ & \langle \sum i\ : 0 \le i < 2*n \wedge i\ par :\ X^i \,\rangle + \langle \sum i\ : 0 \le i < 2*n \wedge i\ impar :\ X^i \,\rangle \\ = & \{\ \text{reemplazo } i \leftarrow 2*i \text{ en el primer término, } i \leftarrow 2*i+1 \text{ en el segundo}\ \} \\ & \langle \sum i\ : 0 \le 2*i < 2*n :\ X^{2*i} \,\rangle + \langle \sum i\ : 0 \le 2*i+1 < 2*n :\ X^{2*i+1} \,\rangle \end{array}$$

$$
\begin{array}{ll}
= & \{ \text{ álgebra } \} \\
& \left\langle \sum i \ : 0 \leq i < n : \ X^{2*i} \right\rangle + \left\langle \sum i \ : 0 \leq i < n : \ X * X^{2*i} \right\rangle \\
= & \{ \text{ distributividad de * con respecto a + } \} \\
& \left\langle \sum i \ : 0 \leq i < n : \ X^{2*i} \right\rangle + X * \left\langle \sum i \ : 0 \leq i < n : \ X^{2*i} \right\rangle \\
= & \{ \text{ álgebra } \} \\
& (1 + X) * \left\langle \sum i \ : 0 \leq i < n : \ X^{2*i} \right\rangle \\
= & \{ \text{ álgebra } \} \\
& (1 + X) * \left\langle \sum i \ : 0 \leq i < n : \ (X^2)^i \right\rangle
\end{array}
$$

Aquí nos encontramos con un problema, dado que no podemos aplicar la hipótesis inductiva, pues no aparece X sino X^2. Esto puede solucionarse otra vez generalizando la función f. Como mencionamos al comienzo de esta sección, la idea es definir una función que dependa de ciertos parámetros, de manera tal que f sea un caso particular de la nueva función (para ciertos valores fijos de los parámetros).

Sea g especificada por $\left\langle \forall x, n \ :: \ g.x.n = \left\langle \sum i \ : 0 \leq i < n : \ x^i \right\rangle \right\rangle$. La función g introduce el parámetro x. f es un caso particular de g que se obtiene cuando el parámetro x toma el valor fijo X, pues $g.X.n = f.n$.

Usando el razonamiento anterior donde simplemente se reemplaza X por x es posible obtener una definición recursiva para el caso par de g.

$$
\begin{array}{ll}
& g.x.(2 * n) \\
= & \{ \text{ especificación de } g \ \} \\
& \left\langle \sum i \ : 0 \leq i < n : \ x^i \right\rangle \\
= & \{ \text{ demostración anterior } \} \\
& (1 + x) * \left\langle \sum i \ : 0 \leq i < n : \ (x^2)^i \right\rangle
\end{array}
$$

Aquí podríamos aplicar hipótesis inductiva obteniendo

$$g.x.(2 * n) \doteq (1 + x) * g.(x * x).n \ .$$

Pero para hacerlo tenemos que garantizar que el segundo parámetro de g a la derecha de la definición sea menor que este mismo parámetro a la izquierda, o sea $n < 2 * n$ (ver inducción generalizada en sección 10.2). Esto es cierto únicamente si $n > 0$. Por lo tanto falta encontrar la definición de g para el caso base $n = 0$.

Ejercicio: Derivar la definición de g para el caso base.

Todavía falta encontrar una definición de g para el caso en que el parámetro sea impar.

$$
\begin{array}{ll}
& g.x.(2*n+1) \\
= & \{ \text{ especificación de } g \ \} \\
& \langle \sum i \ : 0 \le i < 2*n+1 : \ x^i \ \rangle \\
= & \{ \text{ reemplazando } i \leftarrow i+1 \ \} \\
& \langle \sum i \ : 0 \le i+1 < 2*n+1 : \ x^{i+1} \ \rangle \\
= & \{ \text{ álgebra } \} \\
& \langle \sum i \ : -1 \le i < 2*n : \ x^{i+1} \ \rangle \\
= & \{ \text{ separación de un término } \} \\
& 1 + \langle \sum i \ : 0 \le i < 2*n : \ x^{i+1} \ \rangle \\
= & \{ \text{ álgebra } \} \\
& 1 + \langle \sum i \ : 0 \le i < 2*n : \ x * x^i \ \rangle \\
= & \{ \text{ distributividad de * con respecto a + } \} \\
& 1 + x * \langle \sum i \ : 0 \le i < 2*n : \ x^i \ \rangle \\
= & \{ \text{ hipótesis inductiva } \} \\
& 1 + x * g.x.(2*n)
\end{array}
$$

Por lo tanto, una posible definición de g será

$$
\begin{array}{|l}
\hline
g : Num \mapsto Num \mapsto Num \\
\hline
\qquad\qquad\quad g.x.0 \doteq 0 \\
\qquad\quad g.x.(2*n) \doteq (1+x) * g.(x*x).n \quad \text{si } n > 0 \\
\quad g.x.(2*n+1) \doteq 1 + x * g.x.(2*n) \\
\hline
\end{array}
$$

o utilizando análisis por casos

$$
\begin{array}{|l}
\hline
g : Num \mapsto Num \mapsto Num \\
\hline
g.x.n \doteq (\ \ n = 0 \to 0 \\
\qquad\quad \Box\ n \neq 0 \to (\ \ n \ \mathsf{mod}\ 2 = 0 \to (1+x) * g.(x*x).(n \ \mathsf{div}\ 2) \\
\qquad\qquad\qquad\quad \Box\ n \ \mathsf{mod}\ 2 = 1 \to 1 + x * g.x.(n-1) \\
\qquad\qquad\qquad\quad) \\
\qquad\quad) \\
\hline
\end{array}
$$

Es claro que el costo de calcular g (y consecuentemente el de f) resulta logarítmico, pues cuando el segundo argumento es par $(2*n)$, la recursividad nos lleva a evaluar g en un par de argumentos tal que el segundo de ellos es exactamente la mitad del segundo argumento original (n), mientras que cuando el segundo argumento es impar $(2*n+1)$, la recursividad nos lleva a evaluar g en un par de argumentos tal que el segundo de ellos es par $(2*n)$.

Ejercicio: demostrar que el costo de g es logarítmico (depende básicamente de $\log_2 n$).

Ejemplo 11.6
Veamos ahora cómo calcular la función exponencial con costo logarítmico. Partiendo de la especificación $\langle \forall x, n : x \in \mathbb{R} \wedge n \in Nat : e.x.n = x^n \rangle$ y considerando por separado los casos n par y n impar con $n > 0$, tenemos

$$
\begin{array}{ll}
 & e.x.(2 * k) \\
= & \{ \text{ especificación de } e \ \} \\
 & x^{2*k} \\
= & \{ \text{ álgebra } \} \\
 & (x^k)^2 \\
= & \{ \text{ álgebra } \} \\
 & x^k * x^k \\
= & \{ \text{ hipótesis inductiva } \} \\
 & e.x.k * e.x.k \\
= & \{ \text{ introducción de } a \ \} \\
 & a * a \\
 & \quad [\![a = e.x.k]\!]
\end{array}
\qquad
\begin{array}{ll}
 & e.x.(2 * k + 1) \\
= & \{ \text{ especificación de } e \ \} \\
 & x^{2*k+1} \\
= & \{ \text{ álgebra } \} \\
 & x * (x^k)^2 \\
= & \{ \text{ álgebra } \} \\
 & x * x^k * x^k \\
= & \{ \text{ hipótesis inductiva } \} \\
 & x * e.x.k * e.x.k \\
= & \{ \text{ introducción de } a \ \} \\
 & x * a * a \\
 & \quad [\![a = e.x.k]\!]
\end{array}
$$

Ejercicio: Derivar la definición de e para el caso base.

Obtenemos así la siguiente definición recursiva para e:

$$
\begin{array}{|l}
\hline
e : Num \mapsto Nat \mapsto Num \\
\hline
\begin{array}{lll}
 & e.x.0 & \doteq \ 1 \\
n \neq 0 \Rightarrow & e.x.n & \doteq \ (\ n \ \mathsf{mod}\ 2 = 0 \rightarrow a * a \\
 & & \quad \square \ n \ \mathsf{mod}\ 2 = 1 \rightarrow x * a * a \\
 & & \quad) \\
 & & \qquad [\![a = e.x.(n \ \mathsf{div}\ 2)]\!]
\end{array}
\end{array}
$$

El costo de evaluar e usando esta definición es $\log_2(n + 2)$.

Observemos que si en lugar de escribir $x^{2*k} = (x^k)^2$ hacemos $x^{2*k} = (x^2)^k$, obtenemos otra expresión para la exponencial:

$$
\begin{array}{ll}
 & e.x.(2 * k) \\
= & \{ \text{ especificación de } e \ \} \\
 & x^{2*k}
\end{array}
$$

$$
\begin{array}{ll}
= & \{ \text{ álgebra } \} \\
& (x^2)^k \\
= & \{ \text{ hipótesis inductiva } \} \\
& e.x^2.k \\
= & \{ \text{ álgebra } \} \\
& e.(x * x).k
\end{array}
$$

y sabiendo como calcular la exponencial con exponente par, podemos usar esta definición para el caso impar, pues

$$
\begin{array}{ll}
& e.x.(2 * k + 1) \\
= & \{ \text{ especificación de } e \ \} \\
& x^{2*k+1} \\
= & \{ \text{ álgebra } \} \\
& x * x^{2*k} \\
= & \{ \text{ cálculo anterior } \} \\
& x * e.(x * x).k
\end{array}
$$

Con lo cual podemos dar la siguiente definición recursiva para el cálculo de la exponencial:

$$
\begin{array}{l}
e : Num \mapsto Nat \mapsto Num \\
\hline
\qquad\qquad e.x.0 \doteq 1 \\
n \neq 0 \Rightarrow \quad e.x.n \doteq (\ \ n \bmod 2 = 0 \rightarrow e.(x * x).(n \operatorname{div} 2) \\
\qquad\qquad\qquad\quad \square\ n \bmod 2 = 1 \rightarrow x * e.(x * x).((n-1) \operatorname{div} 2) \\
\qquad\qquad\qquad\quad)
\end{array}
$$

Finalmente, podemos usar análisis por casos para dar una definición de $e.x.n$ válida para todo n:

$$
\begin{array}{l}
e : Num \mapsto Nat \mapsto Num \\
\hline
e.x.n \doteq (\ \ n = 0 \rightarrow 1 \\
\qquad\quad \square\ n \neq 0 \rightarrow (\ \ n \bmod 2 = 0 \rightarrow e.(x * x).(n \operatorname{div} 2) \\
\qquad\qquad\qquad\qquad\ \ \square\ n \bmod 2 = 1 \rightarrow x * e.(x * x).((n-1) \operatorname{div} 2) \\
\qquad\qquad\qquad\qquad\ \ ) \\
\qquad\quad)
\end{array}
$$

11.6 Ejercicios

Ejercicio 11.1
Especificar y derivar la función $iguala : [A] \mapsto A \mapsto Bool$ que verifica si todos los elementos de una lista son iguales a un valor dado.

Ejercicio 11.2
Sea $m : [Int] \mapsto Int$ la función que devuelve el mínimo elemento de una lista de enteros. Obtener una definición recursiva para m.

Ejercicio 11.3
Calcular la función que eleva un número natural al cubo usando solo sumas. La especificación es la obvia: $f.x = x^3$. Sugerencia: usar inducción, modularización varias veces y luego técnica de tuplas para mejorar la eficiencia.

Ejercicio 11.4
[Kal90] Calcular la función de fibolucci especificada como sigue:

$$\begin{array}{|l} f : Nat \mapsto Nat \\ \hline f.n = \langle \sum i \ : 0 \leq i < n : \ fib.i * fib.(n-i) \, \rangle \end{array}$$

Ejercicio 11.5
Sea f la función que resuelve el problema de las torres de Hanoi (ver sección 10.3, ejercicio 10.10). Calcular una definición recursiva para la función

$$\begin{array}{|l} t : B \mapsto B \mapsto B \mapsto Nat \mapsto ([(B,B)],\ [(B,B)],\ [(B,B)]) \\ \hline t.a.b.c.n = (f.a.b.c.n,\ f.b.c.a.n,\ f.c.b.a.n) \end{array}$$

Ayuda: Además del caso base $t.a.b.c.0$, calcule $t.a.b.c.1$; luego calcule el caso inductivo $t.a.b.c.(n+1)$.

Capítulo 12

Ejemplos de derivación de programas funcionales

> ¿No sabías que para abrir un aujerito hay que ir sacando la tierra y tirándola lejos?
>
> Julio Cortázar: *Rayuela*

En este capítulo desarrollaremos algunos ejemplos completos de derivación de programas a partir de especificaciones, usando cuando sea necesario las diversas técnicas descriptas en el capítulo 11. En todos los casos, el objetivo es encontrar una definición recursiva de la función que se desea calcular.

12.1 Ejemplos numéricos

Ejemplo 12.1 (Evaluación de un polinomio)
Un problema que se presenta frecuentemente en matemática es el de evaluar un polinomio de la forma $p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$ en un valor dado y de la variable independiente. En el ejemplo 9.12 especificamos esta función:

$$\begin{array}{|l} ev : Num \mapsto [Num] \mapsto Num \\ \hline ev.xs.y = \langle \sum i \ : 0 \leq i < \#xs : \ xs.i * y^i \ \rangle \end{array}$$

Ahora estamos en condiciones de derivar una definición recursiva para ev, haciendo inducción en la lista de coeficientes del polinomio.

Caso base

$$
\begin{array}{ll}
 & ev.[\,].y \\
= & \{ \text{ especificación de } ev;\ \#[\,] = 0 \ \} \\
 & \langle \sum i \ : 0 \le i < 0 : \ [\,].i * y^i \ \rangle \\
= & \{ \text{ rango vacío } \} \\
 & 0
\end{array}
$$

Paso inductivo

$$
\begin{array}{ll}
 & ev.(x \triangleright xs).y \\
= & \{ \text{ especificación de } ev;\ \#(x \triangleright xs) = 1 + \#xs \ \} \\
 & \langle \sum i \ : 0 \le i < 1 + \#xs : \ (x \triangleright xs).i * y^i \ \rangle \\
= & \{ \text{ partición de rango } \} \\
 & \langle \sum i \ : 0 \le i < 1 : \ (x \triangleright xs).i * y^i \ \rangle + \\
 & \quad \langle \sum i \ : 1 \le i < 1 + \#xs : \ (x \triangleright xs).i * y^i \ \rangle \\
= & \{ \text{ rango unitario; } i \leftarrow i + 1 \text{ en el segundo término } \} \\
 & (x \triangleright xs).0 * y^0 + \langle \sum i \ : 1 \le i + 1 < 1 + \#xs : \ (x \triangleright xs).(i+1) * y^{i+1} \ \rangle \\
= & \{ \text{ álgebra; definición de indexar } \} \\
 & x + \langle \sum i \ : 0 \le i < \#xs : \ xs.i * y * y^i \ \rangle \\
= & \{ \text{ distributividad de } * \text{ con respecto a } + \ \} \\
 & x + y * \langle \sum i \ : 0 \le i < \#xs : \ xs.i * y^i \ \rangle \\
= & \{ \text{ hipótesis inductiva } \} \\
 & x + y * ev.xs.y
\end{array}
$$

Por lo tanto una definición recursiva para ev es:

$$
\begin{array}{|l}
\hline
ev : Num \mapsto [Num] \mapsto Num \\
\hline
\qquad\quad ev.[\,].y \ \doteq \ 0 \\
ev.(x \triangleright xs).y \ \doteq \ x + y * ev.xs.y \\
\hline
\end{array}
$$

Vale la pena notar que la definición obtenida corresponde al conocido esquema de Horner [Knu69, pág. 486], el cual resulta computacionalmente más eficiente que el cálculo de las sucesivas potencias y los productos por los coeficientes respectivos.

La definición de ev puede reescribirse usando análisis por casos y las operaciones indexar y tirar:

$$
\begin{array}{|l}
ev : Num \mapsto [Num] \mapsto Num \\
\hline
ev.xs.y \;\doteq\; (\;\; xs = [\,] \to 0 \\
\qquad\qquad \Box\;\; xs \neq [\,] \to xs.0 + y * ev.(xs{\downarrow}1).y \\
\qquad\qquad)
\end{array}
$$

Ejemplo 12.2 (Aproximación de la constante matemática e)
La constante matemática e se define a través de una serie:

$$e = \sum_{i \geq 0} \frac{1}{i!}$$

Se desea una función recursiva que calcule una aproximación de e usando N términos de la serie. En primer lugar, debemos especificar el problema. Tomando solamente N términos de la serie, obtenemos la expresión

$$e \approx \langle \textstyle\sum i \;:\; 0 \leq i < N :\; \frac{1}{i!} \rangle$$

la cual nos induce, reemplazando la constante N por la variable n, a especificar la función buscada por:

$$
\begin{array}{|l}
f : Num \mapsto Num \\
\hline
f.n = \langle \textstyle\sum i \;:\; 0 \leq i < n :\; \frac{1}{i!} \rangle
\end{array}
$$

A partir de esta especificación, derivaremos una definición recursiva para la función f, por inducción en la cantidad de términos de la serie.

Caso base

$$
\begin{array}{ll}
 & f.0 \\
= & \{ \text{ especificación de } f \text{ } \} \\
 & \langle \textstyle\sum i \;:\; 0 \leq i < 0 :\; \frac{1}{i!} \rangle \\
= & \{ \text{ rango vacío } \} \\
 & 0
\end{array}
$$

Paso inductivo

$$
\begin{array}{ll}
 & f.(n+1) \\
= & \{ \text{ especificación de } f \text{ } \} \\
 & \langle \textstyle\sum i \;:\; 0 \leq i < n+1 :\; \frac{1}{i!} \rangle
\end{array}
$$

$$
\begin{array}{ll}
= & \{ \text{ partición de rango } \} \\
& \langle \sum i \ : 0 \leq i < n : \ \frac{1}{i!} \rangle + \langle \sum i \ : n \leq i < n+1 : \ \frac{1}{i!} \rangle \\
= & \{ \text{ hipótesis inductiva; rango unitario } \} \\
& f.n + \frac{1}{n!}
\end{array}
$$

Por lo tanto f queda definida recursivamente por:

$$
\begin{array}{|l}
\hline
f : Num \mapsto Num \\
\hline
\quad\quad f.0 \ \doteq \ 0 \\
\quad f.(n+1) \ \doteq \ f.n + \dfrac{1}{n!}
\end{array}
$$

La definición que hemos obtenido es ineficiente, pues calcula el factorial en cada paso de recursión, lo cual hace que el costo de evaluar f sea cuadrático. Usando la técnica de tuplas, podemos obtener una definición recursiva que calcule simultáneamente e y el factorial, reduciendo de esta manera el costo. Sea

$$h.n = (f.n, g.n) \ ,$$

donde g es la función que calcula el factorial, es decir:

$$
\begin{array}{|l}
\hline
g : Num \mapsto Num \\
\hline
\quad\quad g.0 \ \doteq \ 1 \\
\quad g.(n+1) \ \doteq \ (n+1) * g.n
\end{array}
$$

Derivemos una definición recursiva para h.

Caso base

$$
\begin{array}{ll}
& h.0 \\
= & \{ \text{ especificación de } h \ \} \\
& (f.0, g.0) \\
= & \{ \text{ definición de } f \text{ y } g \ \} \\
& (0, 1)
\end{array}
$$

Paso inductivo

$$
\begin{array}{ll}
& h.(n+1) \\
= & \{ \text{ especificación de } h \ \} \\
& (f.(n+1), g.(n+1))
\end{array}
$$

$$
\begin{array}{ll}
= & \{ \text{ definición de } f \text{ y } g \ \} \\
& (f.n + \dfrac{1}{g.n}, (n+1) * g.n) \\
= & \{ \text{ introducimos } a,\ b \ \} \\
& (a + \dfrac{1}{b}, (n+1) * b) \\
& \qquad [\![(a,b) = (f.n, g.n)]\!] \\
= & \{ \text{ hipótesis inductiva } \} \\
& (a + \dfrac{1}{b}, (n+1) * b) \\
& \qquad [\![(a,b) = h.n]\!]
\end{array}
$$

De esta manera, h queda definida recursivamente por:

$$
\begin{array}{|l}
\hline
h : Num \mapsto (Num, Num) \\
\hline
\qquad\qquad h.0 \doteq (0,1) \\
h.(n+1) \doteq (a + \dfrac{1}{b}, (n+1) * b) \\
\qquad\qquad\qquad [\![(a,b) = h.n]\!] \\
\hline
\end{array}
$$

o bien, usando análisis por casos,

$$
\begin{array}{|l}
\hline
h : Num \mapsto (Num, Num) \\
\hline
h.n \doteq (\ \ n = 0 \to (0,1) \\
\qquad \Box\ n \neq 0 \to (a + \dfrac{1}{b},\ n * b) \\
\qquad\qquad\qquad\qquad [\![(a,b) = h.(n-1)]\!] \\
\qquad)
\end{array}
$$

Ejemplo 12.3 (búsqueda lineal)
Dada $f : Nat \mapsto Bool$ y suponiendo $\langle\, \exists\, n \ : 0 \leq n : \ f.n \,\rangle$, encontrar el mínimo natural x tal que $f.x$. La expresión que especifica este natural es la siguiente

$$\langle \text{Min } i \ : 0 \leq i \wedge f.i : \ i \,\rangle \ .$$

El método a usar para resolver este problema es el de reemplazo de constantes por variables. La única constante que puede ser reemplazada es 0.

Consideramos una función $g : Nat \mapsto Nat$ (suponiendo f ya dada) especificada como

$$g.n = \langle \text{Min } i \ : n \leq i \wedge f.i : \ i \,\rangle \ .$$

Obviamente se encuentra el valor requerido aplicando g al valor 0.

Para poder hacer inducción la única información que tenemos sobre f es que $f.N$ vale para algún N, por lo tanto, podemos trabajar usando inducción (generalizada) sobre $N - n$ haciendo análisis por casos.

$$
\begin{array}{ll}
 & g.n \\
= & \{ \text{ especificación de } g \ \} \\
 & \langle \text{Min } i \ : n \leq i \wedge f.i : \ i \rangle \\
= & \{ \text{ partición de rango } \} \\
 & \langle \text{Min } i \ : n = i \wedge f.i : \ i \rangle \ \text{min} \ \langle \text{Min } i \ : n+1 \leq i \wedge f.i : \ i \rangle \\
= & \{ \text{ Leibniz } \} \\
 & \langle \text{Min } i \ : n = i \wedge f.n : \ i \rangle \ \text{min} \ \langle \text{Min } i \ : n+1 \leq i \wedge f.i : \ i \rangle \\
= & \{ \text{ hipótesis inductiva } \} \\
 & \langle \text{Min } i \ : n = i \wedge f.n : \ i \rangle \ \text{min} \ g.(n+1)
\end{array}
$$

Ahora se hace necesario un análisis por casos de acuerdo al valor de $f.n$.

El caso en que el predicado es cierto se transformará en el caso base de la función.

Caso base $(f.n)$

$$
\begin{array}{ll}
 & \langle \text{Min } i \ : n = i : \ i \rangle \ \text{min} \ \ g.(n+1) \\
= & \{ \text{ rango unitario } \} \\
 & n \ \text{min} \ \ g.(n+1) \\
= & \{ \ n \leq g.(n+1) \ \} \\
 & n
\end{array}
$$

El caso en que es falso será el caso inductivo.

Caso inductivo $(\neg f.n)$

$$
\begin{array}{ll}
 & \langle \text{Min } i \ : n = i \wedge False : \ i \rangle \ \text{min} \ \ g.(n+1) \\
= & \{ \text{ rango vacío } \} \\
 & +\infty \ \ \text{min} \ \ g.(n+1) \\
= & \{ \text{ neutro del min } \} \\
 & g.(n+1)
\end{array}
$$

La definición de g quedará entonces como

$$
\begin{array}{|l}
\hline
g : Num \mapsto Num \\
\hline
g.n \doteq (\ \ f.n\ \ \rightarrow n \\
\qquad \Box\ \neg f.n \rightarrow g.(n+1) \\
\qquad)
\end{array}
$$

12.2 Ejemplos con listas

Las especificaciones sobre listas pueden hacerse de maneras diferentes. Una posibilidad es usar variables cuantificadas para hacer referencia a los índices y usar la funciones de indexar, tomar y tirar. Esta es la forma que se ha usado hasta ahora. Otra forma es usar variables cuantificadas de tipo lista para denotar sublistas de la original y eventualmente variables que denoten elementos de la lista. En este estilo se usan las operaciones de concatenación y de agregar elementos a una lista para describir la forma de los términos de la cuantificación. Esta idea quedará clara en los ejemplos.

Estos estilos diferentes de especificar dan lugar a distintas herramientas metodológicas para la construcción de los programas. Dependiendo del problema, alguno de estos estilos puede ser más simple que otro y por lo tanto preferible.

Ejemplo 12.4 (Un problema para listas de números)
Dada una lista de números, se debe determinar si alguno de los elementos de la lista es igual a la suma de todos los elementos que lo preceden. Este problema se puede especificar de la siguiente manera:

$$P.xs \equiv \langle\, \exists\, i\ : 0 \leq i < \#xs :\ xs.i = \langle \sum j\ : 0 \leq j < i :\ xs.j \,\rangle \,\rangle \quad .$$

Para abreviar la notación, usaremos la función para el cálculo de la sumatoria derivada en el ejemplo 11.1 (pág. 172).

Con la ayuda de esta función, podemos especificar el problema como:

$$P.xs \equiv \langle\, \exists\, i\ : 0 \leq i < \#xs :\ xs.i = sum.(xs{\uparrow}i) \,\rangle$$

Derivemos ahora una expresión recursiva para P, haciendo inducción en la longitud de la lista.

Caso base $(xs = [\,])$

$$
\begin{array}{ll}
 & P.[\,] \\
\equiv & \{\ \text{especificación de } P;\ \#[\,] = 0\ \} \\
 & \langle\, \exists\, i\ : 0 \leq i < 0 :\ [\,].i = sum.([\,]{\uparrow}i) \,\rangle
\end{array}
$$

$\equiv$ { rango vacío }

$False$

Paso inductivo $(x \triangleright xs)$

$P.(x \triangleright xs)$

$\equiv$ { especificación de P; definición de $\#$ }

$\langle\, \exists\, i \;:\; 0 \leq i < 1 + \#xs :\; (x \triangleright xs).i = sum.((x \triangleright xs)\!\uparrow\! i)\,\rangle$

$\equiv$ { separación de un término }

$(x \triangleright xs).0 = sum.(x \triangleright xs)\!\uparrow\! 0$

$\vee\ \langle\, \exists\, i \;:\; 0 \leq i < \#xs :\; (x \triangleright xs).(i+1) = sum.((x \triangleright xs)\!\uparrow\!(i+1))\,\rangle$

$\equiv$ { definiciones de . y de$\uparrow$}

$x = sum.[\,] \vee \langle\, \exists\, i \;:\; 0 \leq i < \#xs :\; xs.i = sum.x \triangleright (xs\!\uparrow\! i)\,\rangle$

$\equiv$ { definición de sum }

$x = 0 \vee \langle\, \exists\, i \;:\; 0 \leq i < \#xs :\; xs.i = x + sum.(xs\!\uparrow\! i)\,\rangle$

La presencia de "x" dentro del término de la cuantificación existencial hace que no podamos aplicar la hipótesis inductiva, por lo que es necesario generalizar la función P.

Sea

$$Q.n.xs \equiv \langle\, \exists\, i \;:\; 0 \leq i < \#xs :\; xs.i = n + sum.(xs\!\uparrow\! i)\,\rangle$$

La función Q es efectivamente una generalización de P, pues $P.xs = Q.0.xs$. Derivemos ahora una definición recursiva para Q.

Caso base $(xs = [\,])$

$Q.n.[\,]$

$\equiv$ { especificación de Q; $\#[\,] = 0$ }

$\langle\, \exists\, i \;:\; 0 \leq i < 0 :\; [\,].i = n + sum.([\,]\!\uparrow\! i)\,\rangle$

$\equiv$ { rango vacío }

$False$

Paso inductivo $(x \triangleright xs)$

$Q.n.(x \triangleright xs)$

$\equiv$ { especificación de Q; definición de $\#$ }

$\langle\, \exists\, i \;:\; 0 \leq i < 1 + \#xs :\; (x \triangleright xs).i = n + sum.((x \triangleright xs)\!\uparrow\! i)\,\rangle$

$\equiv$ { separación de un término }

$(x \triangleright xs).0 = n + sum.(x \triangleright xs)\uparrow 0$

$\vee \ \langle \exists\, i \ : 0 \leq i < \#xs : \ (x \triangleright xs).(i+1) = n + sum.((x \triangleright xs)\uparrow(i+1)) \,\rangle$

$\equiv$ { definiciones de . y de$\uparrow$ }

$x = n + sum.[\,] \vee \langle \exists\, i \ : 0 \leq i < \#xs : \ xs.i = n + sum.(x \triangleright (xs\uparrow i)) \,\rangle$

$\equiv$ { definición de sum }

$x = n + 0 \vee \langle \exists\, i \ : 0 \leq i < \#xs : \ xs.i = n + x + sum.(xs\uparrow i) \,\rangle$

$\equiv$ { hipótesis inductiva }

$x = n \vee Q.(n+x).xs$

De esta manera, Q queda definida recursivamente por:

$$
\begin{array}{|l}
Q : Num \mapsto [Num] \mapsto Bool \\
\hline
\qquad\quad Q.n.[\,] \doteq False \\
Q.n.(x \triangleright xs) \doteq (x = n) \vee Q.(n+x).xs
\end{array}
$$

o bien, usando análisis por casos,

$$
\begin{array}{|l}
Q : Num \mapsto [Num] \mapsto Bool \\
\hline
Q.n.xs \doteq (\ \ xs = [\,] \to False \\
\qquad\quad \Box\ xs \neq [\,] \to (xs.0 = n) \vee Q.(n + xs.0).(xs{\downarrow}1) \\
\qquad\quad)
\end{array}
$$

Ejemplo 12.5 (El problema de la lista balanceada)
Dada una lista cuyos elementos pueden ser solamente de dos clases (por ejemplo, booleanos, números pares o impares, números positivos o negativos, etc.), se desea saber si la cantidad de elementos de cada clase que hay en la lista es la misma. Ya hemos considerado el caso de una lista de booleanos y hemos llamado *bal* al predicado que indica si en dicha lista hay igual cantidad de $True$ que de $False$ (ejemplo 9.13).

El problema puede especificarse fácilmente usando expresiones cuantificadas adecuadas, que permitan contar la cantidad de elementos de cada clase que hay en la lista. Sean ct y cf las funciones que cuentan la cantidad de $True$ y de $False$ respectivamente. Podemos especificar fácilmente estas funciones utilizando el cuantificador de conteo N (pág. 103):

$$
\begin{array}{|l}
ct : [Bool] \mapsto Num \\
\hline
ct.xs = \langle N\, i \ : 0 \leq i < \#xs : \ xs.i \,\rangle
\end{array}
$$

$$\begin{array}{|l} cf : [Bool] \mapsto Num \\ \hline cf.xs = \langle N\, i \;:\; 0 \leq i < \#xs :\; \neg xs.i \,\rangle \end{array}$$

La función *bal* puede especificarse entonces por:

$$\begin{array}{|l} bal : [Bool] \mapsto Num \\ \hline bal.xs \equiv (ct.xs = cf.xs) \end{array}$$

A fin de simplificar la derivación de una definición recursiva para *bal*, derivemos en primer término propiedades de las funciones ct y cf. Como de costumbre, haremos inducción en la longitud de la lista.

Caso base $(xs = [\,])$

$$\begin{array}{ll} & ct.[\,] \\ = & \{ \text{ especificación de } ct\text{; definición de } \# \} \\ & \langle N\, i \;:\; 0 \leq i < 0 :\; [\,].i \,\rangle \\ = & \{ \text{ rango vacío} \} \\ & 0 \end{array}$$

Paso inductivo $(x \triangleright xs)$

$$\begin{array}{ll} & ct.(x \triangleright xs) \\ = & \{ \text{ especificación de } ct\text{; definición de } \# \} \\ & \langle N\, i \;:\; 0 \leq i < 1 + \#xs :\; (x \triangleright xs).i \,\rangle \\ = & \{ \text{ partición de rango} \} \\ & \langle N\, i \;:\; i = 0 :\; (x \triangleright xs).i \,\rangle + \langle N\, i \;:\; 1 \leq i < 1 + \#xs :\; (x \triangleright xs).i \,\rangle \\ = & \{\; i \leftarrow i + 1 \text{ en el segundo término} \} \\ & \langle N\, i \;:\; i = 0 :\; (x \triangleright xs).i \,\rangle + \langle N\, i \;:\; 1 \leq i + 1 < 1 + \#xs :\; (x \triangleright xs).(i+1) \,\rangle \\ = & \{ \text{ álgebra; propiedad de indexar} \} \\ & \langle N\, i \;:\; i = 0 :\; (x \triangleright xs).i \,\rangle + \langle N\, i \;:\; 0 \leq i < \#xs :\; xs.i \,\rangle \\ = & \{ \text{ hipótesis inductiva} \} \\ & \langle N\, i \;:\; i = 0 :\; (x \triangleright xs).i \,\rangle + ct.xs \end{array}$$

El primer término es 1 si x es $True$ y 0 si x es $False$. Por lo tanto, la función ct satisface:

$$\begin{array}{|l} ct : [Bool] \mapsto Num \\ \hline \begin{array}{rl} ct.[\,] & \doteq 0 \\ ct.(True \triangleright xs) & \doteq 1 + ct.xs \\ ct.(False \triangleright xs) & \doteq ct.xs \end{array} \end{array}$$

Análogamente se demuestra que la función cf satisface:

$$\begin{array}{|l} cf : [Bool] \mapsto Num \\ \hline \begin{array}{rl} cf.[\,] & \doteq 0 \\ cf.(True \triangleright xs) & \doteq cf.xs \\ cf.(False \triangleright xs) & \doteq 1 + cf.xs \end{array} \end{array}$$

Ejercicio: Derivar la función cf.

Derivemos ahora una definición recursiva para la función bal.

Caso base $(xs = [\,])$

$$\begin{array}{ll} & bal.[\,] \\ \equiv & \{ \text{ especificación de } bal \} \\ & ct.[\,] = cf.[\,] \\ \equiv & \{ \text{ propiedad de } ct \text{ y } cf \} \\ & True \end{array}$$

Las propiedades de ct y cf nos inducen a derivar el paso inductivo haciendo un análisis por casos, según el valor del primer elemento de la lista.

Paso inductivo $(True \triangleright xs)$

$$\begin{array}{ll} & bal.(True \triangleright xs) \\ \equiv & \{ \text{ especificación de } bal \} \\ & ct.(True \triangleright xs) = cf.(True \triangleright xs) \\ \equiv & \{ \text{ propiedad de } ct \text{ y } cf \} \\ & 1 + ct.xs = cf.xs \end{array}$$

Del mismo modo se obtiene $bal.(False \triangleright xs) \equiv (ct.xs = 1 + cf.xs)$. Como no es posible aplicar la hipótesis inductiva, debemos generalizar la función y comenzar a derivar nuevamente a partir de la generalización. Sea:

$$\begin{array}{|l} g : Num \mapsto [Bool] \mapsto Num \\ \hline g.k.xs \equiv (k + ct.xs = cf.xs) \end{array}$$

La función g es efectivamente una generalización de bal, pues $bal.xs = g.0.xs$. Derivemos, entonces, una definición recursiva para g.

Caso base $(xs = [\,])$

$$\begin{array}{ll} & g.k.[\,] \\ \equiv & \{ \text{ especificación de } gbal \} \\ & k + ct.[\,] = cf.[\,] \\ \equiv & \{ \text{ propiedad de } ct \text{ y } cf \} \\ & k = 0 \end{array}$$

Paso inductivo $(True \triangleright xs)$

$$\begin{array}{ll} & g.k.(True \triangleright xs) \\ \equiv & \{ \text{ especificación de } g \} \\ & k + ct.(True \triangleright xs) = cf.(True \triangleright xs) \\ \equiv & \{ \text{ propiedad de } ct \text{ y } cf \} \\ & k + 1 + ct.xs = cf.xs \\ \equiv & \{ \text{ hipótesis inductiva } \} \\ & g.(k+1).xs \end{array}$$

El caso en que la lista comience con $False$ es análogo y se deja como ejercicio. Por lo tanto, una definición recursiva para g es:

$$\begin{array}{|l} g : Num \mapsto [Bool] \mapsto Num \\ \hline \begin{array}{rcl} g.k.[\,] & \doteq & k = 0 \\ g.k.(True \triangleright xs) & \doteq & g.(k+1).xs \\ g.k.(False \triangleright xs) & \doteq & g.(k-1).xs \end{array} \end{array}$$

12.3 Ejercicios

Ejercicio 12.1 (Lista de iguales)
Derivar una definición recursiva para la función $iguales : [A] \mapsto Bool$ que determina si los elementos de una lista dada son todos iguales entre sí.

Ejercicio 12.2 (Lista creciente)
Derivar una definición recursiva para la función $creciente : [Int] \mapsto Bool$ que determina si los elementos de una lista de enteros están ordenados en forma creciente.

Ejercicio 12.3
Derivar una definición recursiva para la función $f : Int \mapsto [Num] \mapsto Bool$ que determina si el k-ésimo elemento de una lista de números aloja el mínimo valor de la misma.

Ejercicio 12.4 (Función "*minout*")
Derivar una definición recursiva para la función $minout : [Nat] \mapsto Nat$ que, dada una lista de naturales, selecciona el menor natural que no está en la misma.

Ayuda: usar búsqueda lineal.

Ejercicio 12.5
Derivar una definición recursiva para la función $P : [Num] \mapsto Bool$ que, dada una lista de naturales, determina si algún elemento de la lista es igual a la suma de todos los otros elementos de la misma.

Ejercicio 12.6
Derivar una definición recursiva para la función f especificada como sigue:

$$\begin{array}{|l} f : [Num] \mapsto [Num] \mapsto Num \\ \hline f.xs.ys = \langle \text{Min } i, j \ : 0 \leq i < \#xs \wedge 0 \leq j < \#ys : \ |xs.i - ys.j| \ \rangle \end{array}$$

Ejercicio 12.7
Derivar una función recursiva a partir de la siguiente especificación

$$\begin{array}{|l} [\![\_]\!] : [Nat] \mapsto Nat \\ \hline [\![xs]\!] = \langle \sum i \ : 0 \leq i < \#xs : \ xs.i * (i+1)! \ \rangle \end{array}$$

El costo de calcular la función debe ser lineal en la longitud de la lista.

Ayuda: Para encontrar la función habrá que generalizar. Si no encuentra una generalización que resuelva el problema intente con la siguiente:

$$\boxed{[\![xs]\!]_m = \left\langle \sum i \ : 0 \leq i < \#xs : \ xs.i * \tfrac{(i+m)!}{m!} \right\rangle}$$

Ejercicio 12.8
Derivar una función recursiva a partir de la siguiente especificación

$$\begin{array}{|l} [\![\_]\!] : [Nat] \mapsto Nat \\ \hline [\![xs]\!] = \left\langle \sum i \ : 0 \leq i < \#xs : \ xs.i * \frac{1}{(i+2)!} \right\rangle \end{array}$$

El costo de calcular la función debe ser lineal en la longitud de la lista.

Nota: Esta función realiza la evaluación de un número fraccionario positivo escrito en notación factorial [Knu69, pág. 65]. Junto con el resultado del ejercicio anterior se podría representar cualquier número racional positivo mediante dos listas (una que represente la parte entera y otra la fraccionaria). La codificación tiene la ventaja que cualquier número racional puede ser representado de forma eficiente con una cantidad finita de dígitos, pero la cantidad de dígitos posibles no es acotada (a diferencia de la notación usual en una base dada).

Capítulo 13

Ejemplos con segmentos

> "No me haga caso... Lo que usted hace es más importante que lo que yo digo... No soy más que un profesor... No entiendo nada..."
>
> Gustave Moreau a su alumno Henri Matisse.

En este capítulo continuaremos mostrando ejemplos de derivación de programas funcionales enfocándonos en la técnica de especificación por segmentos de listas. En el ejemplo 9.14 esbozamos la utilidad de esta técnica para escribir especificaciones de forma clara y concisa. Ahora estudiaremos algunos ejemplos que muestran cómo derivar programas funcionales a partir de las mismas.

13.1 Búsqueda de un segmento

El siguiente ejemplo es de un predicado que se define inductivamente sobre dos listas.

Se especifica el predicado P que determina si una lista es un segmento de otra de la siguiente manera

$$\left|\begin{array}{l} P : [A] \mapsto [A] \mapsto Bool \\ \hline P.xs.ys = \langle\, \exists\, as, bs \;::\; ys = as +\!\!+ xs +\!\!+ bs \,\rangle \end{array}\right.$$

Para resolver esta función puede estructurarse la derivación inductiva de varias maneras. La más simple (aunque no necesariamente la más corta) es considerar los cuatro casos posibles (vacía-vacía, vacía-no vacía, etc.). Dado que la lista vacía es segmento de cualquier otra lista, analizaremos primero el caso en que xs sea vacía para cualquier valor posible de ys.

Caso base $(xs = [\,])$

$$
\begin{array}{ll}
 & P.[\,].ys \\
\equiv & \{ \text{ especificación de } P \} \\
 & \langle \exists as, bs \;::\; ys = as +\!\!+ [\,] +\!\!+ bs \rangle \\
\equiv & \{ \text{ def. de } +\!\!+ \} \\
 & \langle \exists as, bs \;::\; ys = as +\!\!+ bs \rangle \\
\equiv & \{ \text{ partición de rango } (as = [\,] \text{ o } as \neq [\,]) \} \\
 & \langle \exists as, bs \;:\; as = [\,] :\; ys = as +\!\!+ bs \rangle \vee \langle \exists as, bs \;:\; as \neq [\,] :\; ys = as +\!\!+ bs \rangle \\
\equiv & \{ \text{ anidado; rango unitario } \} \\
 & \langle \exists bs \;::\; ys = [\,] +\!\!+ bs \rangle \vee \langle \exists as, bs \;:\; as \neq [\,] :\; ys = as +\!\!+ bs \rangle \\
\equiv & \{ \text{ def. de } +\!\!+ \} \\
 & \langle \exists bs \;::\; ys = bs \rangle \vee \langle \exists as, bs \;:\; as \neq [\,] :\; ys = as +\!\!+ bs \rangle \\
\equiv & \{ \text{ intercambio } \} \\
 & \langle \exists bs \;:\; ys = bs :\; True \rangle \vee \langle \exists as, bs \;:\; as \neq [\,] :\; ys = as +\!\!+ bs \rangle \\
\equiv & \{ \text{ rango unitario } \} \\
 & True \vee \langle \exists as, bs \;:\; as \neq [\,] :\; ys = as +\!\!+ bs \rangle \\
\equiv & \{ \text{ absorbente para el } \vee \} \\
 & True
\end{array}
$$

El próximo paso será encontrar un definición inductiva para el caso xs distinto de lista vacía. Para ello volveremos a aplicar inducción pero sobre la lista ys.
El caso base es como sigue

Caso $(x \triangleright xs,\ ys = [\,])$

$$
\begin{array}{ll}
 & P.(x \triangleright xs).[\,] \\
\equiv & \{ \text{ especificación de } P \} \\
 & \langle \exists as, bs \;::\; [\,] = as +\!\!+ (x \triangleright xs) +\!\!+ bs \rangle \\
\equiv & \{ \text{ igualdad de listas } \} \\
 & \langle \exists as, bs \;::\; [\,] = as \wedge [\,] = (x \triangleright xs) \wedge [\,] = bs \rangle \\
\equiv & \{ \text{ igualdad de listas } \} \\
 & \langle \exists as, bs \;::\; [\,] = as \;\wedge\; False \;\wedge [\,] = bs \rangle \\
\equiv & \{ \text{ rango vacío } \} \\
 & False
\end{array}
$$

El caso inductivo será

Caso $(x \triangleright xs,\ y \triangleright ys)$

$$
\begin{array}{ll}
 & P.(x \triangleright xs).(y \triangleright ys) \\
\equiv & \{ \text{ especificación de } P \ \} \\
 & \langle\, \exists\, as, bs \ ::\ y \triangleright ys = as +\!\!+ (x \triangleright xs) +\!\!+ bs \,\rangle \\
\equiv & \{ \text{ partición de rango } (as = [\,] \text{ o } as \neq [\,]) \ \} \\
 & \langle\, \exists\, as, bs \ : as = [\,] :\ y \triangleright ys = as +\!\!+ (x \triangleright xs) +\!\!+ bs \,\rangle \\
 & \ \vee\ \langle\, \exists\, as, bs \ : as \neq [\,] :\ y \triangleright ys = as +\!\!+ (x \triangleright xs) +\!\!+ bs \,\rangle \\
\equiv & \{ \text{ anidado; rango unitario } \} \\
 & \langle\, \exists\, bs \ ::\ y \triangleright ys = [\,] +\!\!+ (x \triangleright xs) +\!\!+ bs \,\rangle \\
 & \ \vee\ \langle\, \exists\, as, bs \ : as \neq [\,] :\ y \triangleright ys = as +\!\!+ (x \triangleright xs) +\!\!+ bs \,\rangle \\
\equiv & \{ \text{ reemplazo } as \leftarrow a \triangleright as \text{ (válido por } as \neq [\,]) \ \} \\
 & \langle\, \exists\, bs \ ::\ y \triangleright ys = [\,] +\!\!+ (x \triangleright xs) +\!\!+ bs \,\rangle \\
 & \ \vee\ \langle\, \exists\, a, as, bs \ : a \triangleright as \neq [\,] :\ y \triangleright ys = (a \triangleright as) +\!\!+ (x \triangleright xs) +\!\!+ bs \,\rangle \\
\equiv & \{ \text{ def. } +\!\!+ \text{ ; igualdad de listas } (a \triangleright as \neq [\,] \equiv True) \ \} \\
 & \langle\, \exists\, bs \ ::\ y \triangleright ys = (x \triangleright xs) +\!\!+ bs \,\rangle \\
 & \ \vee\ \langle\, \exists\, a, as, bs \ ::\ y \triangleright ys = (a \triangleright as) +\!\!+ (x \triangleright xs) +\!\!+ bs \,\rangle \\
\equiv & \{ \text{ igualdad de listas } \} \\
 & \langle\, \exists\, bs \ ::\ y = x \ \wedge\ ys = xs +\!\!+ bs \,\rangle \\
 & \ \vee\ \langle\, \exists\, a, as, bs \ ::\ y = a \ \wedge\ ys = as +\!\!+ (x \triangleright xs) +\!\!+ bs \,\rangle \\
\equiv & \{ \text{ distributiva del } \wedge \text{ con el } \exists \ \} \\
 & x = y \ \wedge\ \langle\, \exists\, bs \ ::\ ys = xs +\!\!+ bs \,\rangle \\
 & \ \vee\ \langle\, \exists\, a, as, bs \ ::\ y = a \ \wedge\ ys = as +\!\!+ (x \triangleright xs) +\!\!+ bs \,\rangle \\
\equiv & \{ \text{ intercambio } \} \\
 & x = y \ \wedge\ \langle\, \exists\, bs \ ::\ ys = xs +\!\!+ bs \,\rangle \\
 & \ \vee\ \langle\, \exists\, a, as, bs \ : y = a :\ ys = as +\!\!+ (x \triangleright xs) +\!\!+ bs \,\rangle \\
\equiv & \{ \text{ anidado; rango unitario } \} \\
 & x = y \ \wedge\ \langle\, \exists\, bs \ ::\ ys = xs +\!\!+ bs \,\rangle \\
 & \ \vee\ \langle\, \exists\, as, bs \ ::\ ys = as +\!\!+ (x \triangleright xs) +\!\!+ bs \,\rangle \\
\equiv & \{ \text{ modularización; hipótesis inductiva } \} \\
 & (y = x \wedge Q.xs.ys) \vee P.(x \triangleright xs).ys
\end{array}
$$

Donde el predicado Q queda especificado por

$$
\begin{array}{|l}
\hline
Q : [A] \mapsto [A] \mapsto Bool \\
\hline
Q.xs.ys = \langle\, \exists\, bs \ ::\ ys = xs +\!\!+ bs \,\rangle \\
\hline
\end{array}
$$

Nótese que el predicado Q es más simple que P y que su especificación podría leerse en términos informales como '$Q.xs.ys$ determina si xs es un segmento inicial de ys', donde segmento inicial es un segmento al comienzo de la lista.

Derivaremos entonces una definición recursiva para Q. El patrón recursivo usado será similar al usado para P.

Caso base $(xs = [\,])$

$$
\begin{array}{ll}
 & Q.[\,].ys \\
\equiv & \{ \text{ especificación de } P \} \\
 & \langle\, \exists\, bs \;::\; ys = [\,] +\!\!+ bs \,\rangle \\
\equiv & \{ \text{ def. de } +\!\!+ \} \\
 & \langle\, \exists\, bs \;::\; ys = bs \,\rangle \\
\equiv & \{ \text{ intercambio } \} \\
 & \langle\, \exists\, bs \;:\; ys = bs :\; True \,\rangle \\
\equiv & \{ \text{ término constante } \} \\
 & True
\end{array}
$$

El caso base cuando la primera lista no es vacía será entonces

Caso $(x \triangleright xs,\, ys = [\,])$

$$
\begin{array}{ll}
 & Q.(x \triangleright xs).[\,] \\
\equiv & \{ \text{ especificación de } P \} \\
 & \langle\, \exists\, bs \;::\; [\,] = (x \triangleright xs) +\!\!+ bs \,\rangle \\
\equiv & \{ \text{ igualdad de listas } \} \\
 & \langle\, \exists\, bs \;::\; [\,] = (x \triangleright xs) \;\wedge\; [\,] = bs \,\rangle \\
\equiv & \{ \text{ igualdad de listas } \} \\
 & \langle\, \exists\, bs \;::\; False \;\wedge\; [\,] = bs \,\rangle \\
\equiv & \{ \text{ intercambio, rango vacío } \} \\
 & False
\end{array}
$$

y el caso inductivo sobre ys será

Caso $(x \triangleright xs,\, y \triangleright ys)$

$$
\begin{array}{ll}
 & Q.(x \triangleright xs).(y \triangleright ys) \\
\equiv & \{ \text{ especificación de } P \} \\
 & \langle\, \exists\, bs \;::\; y \triangleright ys = (x \triangleright xs) +\!\!+ bs \,\rangle
\end{array}
$$

$\equiv$ { definición de $+\!\!+$ }

$\langle\, \exists\, bs \;::\; y \triangleright ys = x \triangleright (xs +\!\!+ bs) \,\rangle$

$\equiv$ { igualdad de listas }

$\langle\, \exists\, bs \;::\; y = x \;\wedge\; ys = xs +\!\!+ bs \,\rangle$

$\equiv$ { distributiva del $\wedge$ con el $\vee$ }

$y = x \;\wedge\; \langle\, \exists\, bs \;::\; ys = xs +\!\!+ bs \,\rangle$

$\equiv$ { hipótesis inductiva }

$y = x \;\wedge\; Q.xs.ys$

El programa completo queda entonces

$$
\begin{array}{|rcl}
P & : & [Num] \mapsto [Num] \mapsto Bool \\
Q & : & [Num] \mapsto [Num] \mapsto Bool \\
\hline
P.[\,].ys & \doteq & True \\
P.(x \triangleright xs).[\,] & \doteq & False \\
P.(x \triangleright xs).(y \triangleright ys) & \doteq & (x = y \;\wedge\; Q.xs.ys) \;\vee\; P.(x \triangleright xs).ys \\
Q.[\,].ys & \doteq & True \\
Q.(x \triangleright xs).[\,] & \doteq & False \\
Q.(x \triangleright xs).(y \triangleright ys) & \doteq & x = y \;\wedge\; Q.xs.ys
\end{array}
$$

Nota: La función derivada implementa un algoritmo de búsqueda de cadenas cuyo costo es proporcional al producto de las longitudes de las listas. Los algoritmos de búsqueda de cadenas han sido muy estudiados. Algunos tienen costo proporcional a la longitud del texto sobre el que se quiere buscar pero utilizan estructuras intermedias para lograrlo (tablas, autómatas, etc.). Para una reseña de los más conocidos ver [Wik08].

13.2 Problema de los paréntesis equilibrados

Dada una expresión matemática, se pide determinar si los paréntesis están equilibrados, donde esto significa que debe haber igual cantidad de paréntesis izquierdos como derechos y que nunca puede aparecer un paréntesis derecho sin que antes haya habido uno izquierdo. Para simplificar el problema, consideraremos expresiones constituidas solo por paréntesis. Por ejemplo, "()", "(())", "()()" y "(()())" tienen los paréntesis equilibrados, mientras que "(", "(()", "())" y ")()()" no son expresiones con paréntesis equilibrados. El caso en el cual aparezcan otros caracteres en las expresiones (por ejemplo números y operadores aritméticos) es fácilmente generalizable.

Nótese la similitud de este problema con el de la lista balanceada (la expresión matemática puede pensarse como una lista de caracteres). En este caso, además de plantear que la cantidad de elementos de cada clase sea la misma, debemos exigir también que en todo momento la cantidad de paréntesis izquierdos sea mayor o igual que la cantidad de paréntesis derechos leídos.

La forma de resolver el problema es similar a la del ejemplo 12.5. Sean *abr* y *cie* las funciones que cuentan la cantidad de paréntesis izquierdos (abren) y derechos (cierran), respectivamente. Podemos especificar estas funciones de la siguiente manera:

$$\begin{array}{|l} abr : [Char] \mapsto Num \\ \hline abr.xs = \langle N\ i\ :\ 0 \leq i < \#xs\ :\ xs.i = \text{`}(\text{'} \rangle \end{array}$$

$$\begin{array}{|l} cie : [Char] \mapsto Num \\ \hline cie.xs = \langle N\ i\ :\ 0 \leq i < \#xs\ :\ xs.i = \text{`})\text{'} \rangle \end{array}$$

La definición inductiva de estas funciones es fácil de calcular y se deja como ejercicio.

$$\begin{array}{|rcl} abr : [Char] \mapsto Num & & \\ \hline abr.[\,] & \doteq & 0 \\ abr.(\text{`}(\text{'} \triangleright xs) & \doteq & 1 + abr.xs \\ abr.(\text{`})\text{'} \triangleright xs) & \doteq & abr.xs \end{array}$$

$$\begin{array}{|rcl} cie : [Char] \mapsto Num & & \\ \hline cie.[\,] & \doteq & 0 \\ cie.(\text{`}(\text{'} \triangleright xs) & \doteq & cie.xs \\ cie.(\text{`})\text{'} \triangleright xs) & \doteq & 1 + cie.xs \end{array}$$

La función *pareq* que resuelve el problema puede especificarse como sigue

$$\begin{array}{|l} pareq : [Char] \mapsto Bool \\ \hline pareq.xs \equiv \langle\, \forall\, i\ :0 \leq i < \#xs :\ abr.(xs{\uparrow}i) \geq cie.(xs{\uparrow}i)\, \rangle\ \wedge \\ \qquad\qquad\qquad abr.xs = cie.xs \end{array}$$

o bien usando el estilo de segmentos

$$
\begin{array}{|l}
\hline
pareq : [Char] \mapsto Bool \\
\hline
pareq.xs \equiv \langle\, \forall\, as, bs \;:\; xs = as +\!\!+ bs :\; abr.as \geq cie.as \,\rangle \;\wedge \\
\qquad\qquad\qquad\quad abr.xs = cie.xs \\
\hline
\end{array}
$$

Usaremos esta última especificación. Derivemos ahora una definición recursiva para la función *pareq*.

Caso base ($xs = [\,]$)

$$
\begin{array}{ll}
& pareq.[\,] \\
\equiv & \{\ \text{especificación de } pareq\ \} \\
& \langle\, \forall\, as, bs \;:\; [\,] = as +\!\!+ bs :\; abr.as \geq cie.as \,\rangle \;\wedge\; abr.[\,] = cie.[\,] \\
\equiv & \{\ \text{definición de } +\!\!+ \text{; propiedad de } abr \text{ y } cie\ \} \\
& \langle\, \forall\, as, bs \;:\; [\,] = as +\!\!+ bs \;\wedge\; as = [\,] \;\wedge\; bs = [\,] :\; abr.as \geq cie.as \,\rangle \;\wedge\; 0 = 0 \\
\equiv & \{\ \text{anidado; rango unitario; álgebra; cálculo proposicional}\ \} \\
& \langle\, \forall\, bs \;:\; [\,] = bs :\; abr.[\,] \geq cie.[\,] \,\rangle \\
\equiv & \{\ \text{rango unitario; definición de } abr \text{ y } cie\ \} \\
& True
\end{array}
$$

Para el paso inductivo analizaremos el caso en que el primer elemento de la lista sea un paréntesis abierto.

Paso inductivo ($`('\triangleright xs$)

$$
\begin{array}{ll}
& pareq.(`('\triangleright xs) \\
\equiv & \{\ \text{especificación de } pareq\ \} \\
& \langle\, \forall\, as, bs \;:\; `('\triangleright xs = as +\!\!+ bs :\; abr.as \geq cie.as \,\rangle \;\wedge\; abr.(`('\triangleright xs) = cie.(`('\triangleright xs) \\
\equiv & \{\ \text{partición de rango } (as = [\,] \text{ o } as \neq [\,])\ \} \\
& \langle\, \forall\, as, bs \;:\; `('\triangleright xs = as +\!\!+ bs \;\wedge\; as = [\,] :\; abr.as \geq cie.as \,\rangle \\
& \;\wedge\; \langle\, \forall\, as, bs \;:\; `('\triangleright xs = as +\!\!+ bs \;\wedge\; as \neq [\,] :\; abr.as \geq cie.as \,\rangle \\
& \;\wedge\; abr.(`('\triangleright xs) = cie.(`('\triangleright xs) \\
\equiv & \{\ \text{anidado; rango unitario; igualdad de listas}\ \} \\
& \langle\, \forall\, bs \;:\; `('\triangleright xs = bs :\; abr.[\,] \geq cie.[\,] \,\rangle \\
& \;\wedge\; \langle\, \forall\, as, bs \;:\; `('\triangleright xs = as +\!\!+ bs \;\wedge\; as \neq [\,] :\; abr.as \geq cie.as \,\rangle \\
& \;\wedge\; abr.(`('\triangleright xs) = cie.(`('\triangleright xs) \\
\equiv & \{\ \text{definición de } abr \text{ y } cie\ \} \\
& \langle\, \forall\, bs \;:\; `('\triangleright xs = bs :\; 0 \geq 0 \,\rangle \\
& \;\wedge\; \langle\, \forall\, as, bs \;:\; `('\triangleright xs = as +\!\!+ bs \;\wedge\; as \neq [\,] :\; abr.as \geq cie.as \,\rangle \\
& \;\wedge\; abr.(`('\triangleright xs) = cie.(`('\triangleright xs)
\end{array}
$$

$\equiv$ { término constante; cálculo proposicional }

$\langle\, \forall\, as, bs \;:\; `(' \triangleright xs = as +\!\!+ bs \;\wedge\; as \neq [\,] :\; abr.as \geq cie.as \,\rangle$
$\;\wedge\; abr.(`(' \triangleright xs) = cie.(`(' \triangleright xs)$

$\equiv$ { reemplazo $as \leftarrow a \triangleright as$ (válido por $as \neq [\,]$) }

$\langle\, \forall\, a, as, bs : `(' \triangleright xs = (a \triangleright as) +\!\!+ bs \wedge a \triangleright as \neq [\,] : abr.(a \triangleright as) \geq cie.(a \triangleright as) \rangle$
$\;\wedge\; abr.(`(' \triangleright xs) = cie.(`(' \triangleright xs)$

$\equiv$ { anidado; rango unitario; igualdad de listas }

$\langle\, \forall\, as, bs \;:\; xs = as +\!\!+ bs :\; abr.(`(' \triangleright as) \geq cie.(`(' \triangleright as) \,\rangle$
$\;\wedge\; abr.(`(' \triangleright xs) = cie.(`(' \triangleright xs)$

$\equiv$ { propiedad de *abr* y *cie* }

$\langle\, \forall\, as, bs \;:\; xs = as +\!\!+ bs :\; 1 + abr.as \geq cie.as \,\rangle$
$\;\wedge\; 1 + abr.xs = cie.xs$

Con esto hemos llegado a un punto donde resulta imposible aplicar la hipótesis inductiva. Probemos generalizar la función. Sea:

$$\begin{array}{|l} g : Num \mapsto [Char] \mapsto Bool \\ \hline g.k.xs \equiv \langle\, \forall\, as, bs \;:\; xs = as +\!\!+ bs :\; k + abr.as \geq cie.as \,\rangle \;\wedge \\ \qquad\qquad k + abr.xs = cie.xs \end{array}$$

Es claro que g generaliza a *pareq*, pues $g.0.xs = pareq.xs$. Derivemos una definición recursiva para g.

Caso base $(xs = [\,])$

$g.k.[\,]$

$\equiv$ { razonamiento análogo al anterior }

$k = 0$

Paso inductivo $(`(` \triangleright xs)$

$g.k.(`(` \triangleright xs)$

$\equiv$ { especificación de g }

$\langle\, \forall\, as, bs \;:\; `(` \triangleright xs = as +\!\!+ bs :\; k + abr.as \geq cie.as \,\rangle$
$\;\wedge\; k + abr.(`(` \triangleright xs) = cie.(`(` \triangleright xs)$

$\equiv$ { razonamiento análogo al anterior }

$\langle\, \forall\, bs \;:\; `(` \triangleright xs = bs :\; k + abr.[\,] \geq cie.[\,] \,\rangle$
$\;\wedge\; \langle\, \forall\, as, bs \;:\; xs = as +\!\!+ bs :\; k + abr.(`(` \triangleright as) \geq cie.(`(` \triangleright as) \,\rangle$
$\;\wedge\; k + abr.(`(` \triangleright xs) = cie.(`(` \triangleright xs)$

$\equiv$ { rango unitario; álgebra; propiedad de abr y cie }

$k \geq 0 \ \wedge \ \langle \forall\, as, bs \ : xs = as +\!\!+ bs : \ k + abr.(\text{`}(\text{`} \triangleright as) \geq cie.(\text{`}(\text{`} \triangleright as) \ \rangle$
$\wedge \ k + abr.(\text{`}(\text{`} \triangleright xs) = cie.(\text{`}(\text{`} \triangleright xs)$

$\equiv$ { propiedad de abr y cie }

$k \geq 0 \ \wedge \ \langle \forall\, as, bs \ : xs = as +\!\!+ bs : \ k + 1 + abr.as \geq cie.as \ \rangle$
$\wedge \ k + 1 + abr.xs = cie.xs$

$\equiv$ { hipótesis inductiva }

$k \geq 0 \ \wedge \ g.(k+1).xs$

El caso ‘)‘ $\triangleright xs$ es simétrico.

Ejercicio: Derivar el caso ‘)‘ $\triangleright xs$.

Finalmente, podemos dar la siguiente definición recursiva para la función g:

$$
\begin{array}{|l}
g : Num \mapsto [Char] \mapsto Bool \\
\hline
\qquad\qquad g.k.[\,] \ \doteq \ (k = 0) \\
g.k.(\text{`}(\text{`} \triangleright xs) \ \doteq \ k \geq 0 \ \wedge \ g.(k+1).xs \\
g.k.(\text{`})\text{`} \triangleright xs) \ \doteq \ k \geq 0 \ \wedge \ g.(k-1).xs
\end{array}
$$

13.3 Problema del segmento de suma mínima

Dada una lista de enteros, se desea hallar la suma del segmento de suma mínima de la lista. Por ejemplo, si la lista es $xs = [1, -4, -2, 1, -5, 8, -7]$, el segmento que da la suma mínima es [-4,-2,1,-5], cuya suma es -10. Si $xs = [1, 3, 5]$, el segmento que da la suma mínima es [ ], pues la suma de la lista vacía es cero.

En el ejemplo de la sección 13.1 se especificó y derivó una función que determina si una lista es un segmento de otra. Podría usarse esa definición para especificar el problema en cuestión como sigue.

$$
\begin{array}{|l}
f : [Num] \mapsto Num \\
\hline
f.xs = \langle \text{Min } bs \ : \langle \exists\, as, cs \ :: \ xs = as +\!\!+ bs +\!\!+ cs \ \rangle : \ sum.bs \ \rangle
\end{array}
$$

Es fácil demostrar que esta especificación es equivalente a una más simple con un solo cuantificador:

$$
\begin{array}{|l}
f : [Num] \mapsto Num \\
\hline
f.xs = \langle \text{Min } as, bs, cs \ : \ xs = as +\!\!+ bs +\!\!+ cs \ : \ sum.bs \rangle
\end{array}
$$

Derivaremos entonces una definición recursiva para f.

Caso base $(xs = [\,])$

$f.[\,]$

$= \{$ especificación de f $\}$

$\langle \text{Min } as, bs, cs \ : [\,] = as +\!\!+ bs +\!\!+ cs : \ sum.bs \rangle$

$= \{$ propiedad de $+\!\!+$ $\}$

$\langle \text{Min } as, bs, cs \ :$
$\quad [\,] = as +\!\!+ bs +\!\!+ cs \ \wedge \ as = [\,] \ \wedge \ bs = [\,] \ \wedge \ cs = [\,] : \ sum.bs \rangle$

$= \{$ anidado; rango unitario; término constante $\}$

$sum.[\,]$

$= \{$ definición de sum $\}$

0

Paso inductivo $(x \triangleright xs)$

$f.(x \triangleright xs)$

$= \{$ especificación de f $\}$

$\langle \text{Min } as, bs, cs \ : x \triangleright xs = as +\!\!+ bs +\!\!+ cs : \ sum.bs \rangle$

$= \{$ partición de rango ($as = [\,]$ o $as \neq [\,]$) $\}$

$\langle \text{Min } as, bs, cs \ : x \triangleright xs = as +\!\!+ bs +\!\!+ cs \ \wedge \ as = [\,] : \ sum.bs \rangle$
$\min \ \langle \text{Min } as, bs, cs \ : x \triangleright xs = as +\!\!+ bs +\!\!+ cs \ \wedge \ as \neq [\,] : \ sum.bs \rangle$

$= \{$ anidado en el primer término; reemplazo $as \leftarrow a \triangleright as$ en el segundo término (válido por $as \neq [\,]$) $\}$

$\langle \text{Min } as \ : as = [\,] : \ \langle \text{Min } bs, cs \ : x \triangleright xs = as +\!\!+ bs +\!\!+ cs : \ sum.bs \rangle \rangle$
$\min$
$\langle \text{Min } a, as, bs, cs \ : x \triangleright xs = (a \triangleright as) +\!\!+ bs +\!\!+ cs \ \wedge \ a \triangleright as \neq [\,] : \ sum.bs \rangle$

$= \{$ rango unitario; definición de $+\!\!+$ $\}$

$\langle \text{Min } bs, cs \ : x \triangleright xs = bs +\!\!+ cs : \ sum.bs \rangle$
$\min$
$\langle \text{Min } a, as, bs, cs \ : x \triangleright xs = (a \triangleright as) +\!\!+ bs +\!\!+ cs \ \wedge \ a \triangleright as \neq [\,] : \ sum.bs \rangle$

$= \{$ definición de $+\!\!+$ y de igualdad de listas $\}$

$\langle \text{Min } bs, cs \ : x \triangleright xs = bs +\!\!+ cs : \ sum.bs \rangle$
$\min \ \langle \text{Min } a, as, bs, cs \ : x = a \ \wedge \ xs = as +\!\!+ bs +\!\!+ cs : \ sum.bs \rangle$

$= \{$ anidado; rango unitario $\}$

$\langle \text{Min } bs, cs \ : x \triangleright xs = bs +\!\!+ cs : \ sum.bs \rangle$
$\min \ \langle \text{Min } as, bs, cs \ : xs = as +\!\!+ bs +\!\!+ cs : \ sum.bs \rangle$

$=$ { hipótesis inductiva }

$\langle \text{Min } bs, cs \ : x \triangleright xs = bs +\!\!+ cs : \ sum.bs \rangle$
$\min \ f.xs$

$=$ { introducción de $g.xs = \langle \text{Min } bs, cs \ : xs = bs +\!\!+ cs : \ sum.bs \rangle$ }

$g.(x \triangleright xs) \min \ f.xs$

Por lo tanto puede definirse

$$
\begin{array}{|l}
f : [Num] \mapsto Num \\
\hline
\begin{array}{rcl}
f.[\,] & \doteq & 0 \\
f.(x \triangleright xs) & \doteq & g.(x \triangleright xs) \min \ f.xs
\end{array}
\end{array}
$$

Aún falta dar una definición explícita para la función g introducida en el razonamiento anterior. Notemos que $g.xs = \langle \text{Min } bs, cs \ : \ xs = bs +\!\!+ cs \ : \ sum.bs \rangle$ da la suma del "segmento inicial" de suma mínima.

El caso base es similar al anterior.

Ejercicio: Derivar el caso base de g.

Veamos el caso inductivo.

Paso inductivo $(x \triangleright xs)$

$g.(x \triangleright xs)$

$=$ { especificación de g }

$\langle \text{Min } bs, cs \ : x \triangleright xs = bs +\!\!+ cs : \ sum.bs \rangle$

$=$ { partición de rango ($bs = [\,]$ o $bs \neq [\,]$) }

$\langle \text{Min } bs, cs \ : x \triangleright xs = bs +\!\!+ cs \ \wedge \ bs = [\,] : \ sum.bs \rangle$
$\min \ \langle \text{Min } bs, cs \ : x \triangleright xs = bs +\!\!+ cs \ \wedge \ bs \neq [\,] : \ sum.bs \rangle$

$=$ { anidado en el primer término; reemplazo $bs \leftarrow b \triangleright bs$ en el segundo término (válido por $bs \neq [\,]$) }

$\langle \text{Min } bs \ : bs = [\,] : \ \langle \text{Min } cs \ : x \triangleright xs = bs +\!\!+ cs : \ sum.bs \rangle \rangle$
$\min \ \langle \text{Min } b, bs, cs \ : x \triangleright xs = (b \triangleright bs) +\!\!+ cs \ \wedge (b \triangleright bs) \neq [\,] : \ sum.(b \triangleright bs) \rangle$

$=$ { rango unitario }

$\langle \text{Min } cs \ : x \triangleright xs = [\,] +\!\!+ cs : \ sum.[\,] \rangle$
$\min \ \langle \text{Min } b, bs, cs \ : x \triangleright xs = (b \triangleright bs) +\!\!+ cs \ \wedge (b \triangleright bs) \neq [\,] : \ sum.(b \triangleright bs) \rangle$

$=$ { término constante; definición de $sum.[\,]$ }

$0 \min \ \langle \text{Min } b, bs, cs \ : x \triangleright xs = (b \triangleright bs) +\!\!+ cs \ \wedge (b \triangleright bs) \neq [\,] : \ sum.(b \triangleright bs) \rangle$

$=$ { definición de $+\!\!+$ y de igualdad de listas }

$0 \text{ min } \langle \text{Min } b, bs, cs \ : x = b \wedge xs = bs +\!\!+ cs : \ sum.(b \triangleright bs) \rangle$

$=$ { anidado; rango unitario }

$0 \text{ min } \langle \text{Min } bs, cs \ : xs = bs +\!\!+ cs : \ sum.(x \triangleright bs) \rangle$

$=$ { definición de sum }

$0 \text{ min } \langle \text{Min } bs, cs \ : xs = bs +\!\!+ cs : \ x + sum.bs \rangle$

$=$ { distributividad de $+$ con respecto a min (el rango es no vacío) }

$0 \text{ min } (x + \langle \text{Min } bs, cs \ : xs = bs +\!\!+ cs : \ sum.bs \rangle)$

$=$ { hipótesis inductiva }

$0 \text{ min } (x + g.xs)$

Por lo tanto,

$$
\begin{array}{|l}
g : [Num] \mapsto Num \\
\hline
\quad g.[\,] \ \doteq \ 0 \\
g.(x \triangleright xs) \ \doteq \ 0 \text{ min } (x + g.xs)
\end{array}
$$

De esta manera, tenemos $f.(x \triangleright xs) = 0 \ min \ (x + g.xs) \ min \ f.xs$. Pero $f.xs \leq 0$ $\forall$ xs (si todos los elementos de la lista son positivos, el mínimo se obtiene con la lista vacía), por lo que resulta:

$$
\begin{array}{|l}
f : [Num] \mapsto Num \\
\hline
\quad f.[\,] \ \doteq \ 0 \\
f.(x \triangleright xs) \ \doteq \ (x + g.xs) \text{ min } f.xs
\end{array}
$$

Ejercicio: Probar que el costo de calcular g es lineal y el de f es cuadrático.

Aún podemos reducir el costo en el cálculo de la función f. Para lograrlo, usemos la técnica de tuplas: postulemos $h.xs = (f.xs, g.xs)$ y busquemos una definición recursiva para h.

Caso base $(xs = [\,])$

$h.[\,]$

$=$ { especificación de h }

$(f.[\,], g.[\,])$

$=$ { definición de f y g }

(0,0)

Paso inductivo $(x \triangleright xs)$

$$
\begin{array}{ll}
& h.(x \triangleright xs) \\
= & \{ \text{ especificación de } h \ \} \\
& (f.(x \triangleright xs),\ g.(x \triangleright xs)) \\
= & \{ \text{ definición de } f \text{ y } g \ \} \\
& ((x + g.xs) \text{ min } f.xs,\ 0 \text{ min } (x + g.xs)) \\
= & \{ \text{ introducimos } a,\ b \ \} \\
& ((x + b) \text{ min } a,\ 0 \text{ min } (x + b)) \\
& \quad [\![(a, b) = (f.xs, g.xs)]\!] \\
= & \{ \text{ hipótesis inductiva } \} \\
& ((x + b) \text{ min } a,\ 0 \text{ min } (x + b)) \\
& \quad [\![(a, b) = h.xs]\!]
\end{array}
$$

Entonces, para h tenemos:

$$
\left|
\begin{array}{l}
h : [Num] \mapsto (Num, Num) \\
\hline
\begin{array}{rcl}
h.[\,] & \doteq & (0, 0) \\
h.(x \triangleright xs) & \doteq & ((x + b) \text{ min } a,\ 0 \text{ min } (x + b)) \\
& & \quad [\![(a, b) = h.xs]\!]
\end{array}
\end{array}
\right.
$$

o bien, usando análisis por casos,

$$
\left|
\begin{array}{l}
h : [Num] \mapsto (Num, Num) \\
\hline
\begin{array}{rcl}
h.xs & \doteq & (\ \ xs = [\,] \to (0, 0) \\
& & \Box\ xs \neq [\,] \to ((xs.0 + b) \text{ min } a,\ 0 \text{ min } (xs.0 + b)) \\
& & \qquad\qquad [\![(a, b) = h.(xs{\downarrow}1)]\!] \\
& &)
\end{array}
\end{array}
\right.
$$

Ejercicio: Probar que el costo de calcular h es lineal.

13.4 Ejercicios

Ejercicio 13.1 (Cantidad de apariciones de un segmento)
Especificar y derivar la función que calcula la cantidad de apariciones de un segmento en una lista.

Ayuda: Ver cómo se especificó y derivó el ejemplo de la sección 13.1.

Ejercicio 13.2
Derivar una definición recursiva para la función l especificada como sigue:

$$\begin{array}{|l} l : [Char] \mapsto [Char] \mapsto Bool \\ \hline l.xs.ys = \\ \quad \langle \exists\, as, bs, c, cs : xs = as +\!\!+ bs \wedge ys = as +\!\!+ (c \triangleright cs) : bs = [\,] \vee bs.0 < c \,\rangle \end{array}$$

donde $<$ indica una relación de orden entre caracteres.
Decir en palabras cuándo $l.xs.ys$ es $True$.

Ejercicio 13.3 (Contar "la")
Derivar una función recursiva $cantla : String \mapsto Num$ tal que, dada una lista de caracteres, devuelva la cantidad de apariciones del segmento "la". El costo de evaluar la misma debe ser lineal en el tamaño de la lista.

Ejercicio 13.4
Derivar una función recursiva $maxiguala : [A] \mapsto A \mapsto Num$ especificada como

$$\begin{array}{|l} maxiguala : [A] \mapsto A \mapsto Num \\ \hline maxiguala.xs.y = \\ \quad \langle \mathrm{Max}\ as, bs, cs \ : xs = as +\!\!+ bs +\!\!+ cs \ \wedge \ iguala.bs.y : \ \#bs \,\rangle \end{array}$$

donde $iguala$ es la función a derivar en el ejercicio 11.1. El costo de la función resultado debe ser lineal en el tamaño de la lista.

Ayuda: Para obtener costo lineal puede ser necesario utilizar la técnica de tuplas.

Ejercicio 13.5 (Máxima longitud de iguales)
Derivar la función especificada como

$$\begin{array}{|l} maxigual : [A] \mapsto Num \\ \hline maxigual.xs = \langle \mathrm{Max}\ as, bs, cs : xs = as +\!\!+ bs +\!\!+ cs \ \wedge \ iguales.bs : \#bs \rangle \end{array}$$

donde $iguales$ es la función derivada en el ejercicio 12.1. La función obtenida debe tener costo lineal.

Ayuda: Para obtener costo lineal puede ser necesario utilizar la técnica de tuplas.

Capítulo 14

Especificaciones implícitas

> Estas ambigüedades, redundancias y deficiencias recuerdan las que el doctor Franz Kuhn atribuye a cierta enciclopedia china que se titula *Emporio celestial de conocimientos benévolos*. En sus remotas páginas está escrito que los animales se dividen en (a) pertenecientes al Emperador, (b) embalsamados, (c) amaestrados, (d) lechones, (e) sirenas, (f) fabulosos, (g) perros sueltos, (h) incluidos en esta clasificación, (i) que se agitan como locos, (j) innumerables, (k) dibujados con un pincel finísimo de pelo de camello, (l) etcétera, (m) que acaban de romper el jarrón (n) que de lejos parecen moscas.
>
> Jorge Luis Borges: *El Idioma analítico de John Wilkins*

> Dans l'émerveillement de cette taxinomie, ce qu'on rejoint d'un bond, ce qui à la faveur de l'apologue, nous est indiqué comme la charme exotique d'une autre pensée, c'est la limite de la nôtre: l'impossibilité nue de penser cela.
>
> Michel Foucault: *Les mots et les choses*

En este capítulo veremos un ejemplo de derivación de programas que nos introducirá a una nueva técnica para obtener algoritmos.

14.1 Aritmética de precisión arbitraria

Un problema común que se presenta cuando utilizamos algunos lenguajes de programación es el de poder manipular números enteros de gran magnitud. Por

lo general los lenguajes brindan una serie de operaciones aritméticas sobre números enteros, pero los números que se pueden manipular tienen solo una precisión limitada. Esta limitación suele estar determinada por la arquitectura del ordenador sobre la cual se ejecutan los programas; las arquitecturas de los ordenadores están basadas en tamaños fijos de registros para manipular los valores representados, por lo cual no será posible manipular enteros muy grandes directamente con mecanismos de hardware. Por ejemplo, la mayor parte de los compiladores que corren sobre arquitecturas tipo PC permiten representar números desde -2^{31} a $2^{31} - 1$. De esta forma será imposible representar números mayores a 2^{31} (por ejemplo $16384 * 20000 * 31321 - 3$) con las primitivas que brindan estos lenguajes sobre estas arquitecturas.

Una forma de solucionar el problema es creando nuestra propia representación de números enteros. Simplificando el problema, pensemos que queremos representar solo números enteros positivos con una precisión arbitraria. Una posibilidad es hacerlo con la lista de sus dígitos en una base dada. Estos últimos serán representados con el tipo entero que brinda el lenguaje. Así, por ejemplo podríamos representar el número 2147483648 con la lista $[648, 483, 147, 2]$ (con los dígitos en orden inverso) en base 1000. Notemos en este ejemplo que la base es grande en comparación a las usadas habitualmente aprovechando el hecho que los dígitos pueden representar valores hasta la precisión de los enteros dados por el lenguaje. Con esta elección las listas tendrán menor longitud y ocuparán menos espacio en memoria.

Para poder utilizar estas representaciones de números en un entorno de programación hará falta construir funciones para operar con ellas. De esta forma tendremos funciones para sumar, restar, multiplicar y comparar los números denotados por las mismas. Para llevar a cabo esta tarea deberemos derivar estas operaciones a partir de especificaciones, las cuales tendremos que encontrar.

Para especificar estas operaciones definamos la función

$$\frac{[\![\_]\!]_B : [Num] \mapsto Num}{[\![xs]\!]_B = \langle \sum i \ : 0 \leq i < \#xs : \ xs.i * B^i \ \rangle}$$

la cual, dada una lista de números, devuelve el que representa en la base B (con $B > 1$). Este tipo de funciones suelen denominarse **función de abstracción** y relaciona los valores concretos con los abstractos que queremos representar (en nuestro caso listas y números respectivamente).

La función tiene algunas propiedades que serán de utilidad al momento de encontrar las operaciones sobre las representaciones. Mencionaremos algunas de ellas, dejando las demostraciones como ejercicio para el lector.

Propiedades

P$_1$: $[\![[\,]]\!]_B = 0$

P$_2$: $[\![x \triangleright xs]\!]_B = x + B * [\![xs]\!]_B$

Como ejemplo encontremos la función $(\oplus) : [Num] \mapsto [Num] \mapsto [Num]$ que, dadas dos listas de números, devuelve otra que representa el valor suma de los dos números representados por las anteriores. Utilizando la función $[\![\_]\!]_B$ se puede especificar esto de una forma muy simple:

$$\frac{(\oplus) : [Num] \mapsto [Num] \mapsto [Num]}{[\![xs \oplus ys]\!]_B = [\![xs]\!]_B + [\![ys]\!]_B}$$

Tratemos de encontrar una definición recursiva de esta operación de la forma habitual. Comencemos con el caso base:

Caso base $(xs = [\,])$

$$\begin{aligned}
& [\![[\,] \oplus ys]\!]_B \\
= & \quad \{ \text{ especificación de } \oplus \} \\
& [\![[\,]]\!]_B + [\![ys]\!]_B \\
= & \quad \{ \text{ P}_1 \} \\
& 0 + [\![ys]\!]_B \\
= & \quad \{ \text{ álgebra } \} \\
& [\![ys]\!]_B
\end{aligned}$$

Con esto obtenemos la igualdad

$$[\![[\,] \oplus ys]\!]_B = [\![ys]\!]_B$$

con lo cual tenemos un problema ya que no obtenemos una definición para "$[\,] \oplus ys$" de la forma

$$[\,] \oplus ys = \cdots$$

sino una para $[\![[\,] \oplus ys]\!]_B$.

Olvidemos por un momento esta cuestión y sigamos derivando los otros casos.

Caso base $(ys = [\,])$

$$\begin{aligned}
& [\![xs \oplus [\,]]\!]_B \\
= & \quad \{ \text{ especificación de } \oplus \} \\
& [\![xs]\!]_B + [\![[\,]]\!]_B \\
= & \quad \{ \text{ P}_1 \} \\
& [\![xs]\!]_B + 0 \\
= & \quad \{ \text{ álgebra } \} \\
& [\![xs]\!]_B
\end{aligned}$$

Hipótesis inductiva: $[\![xs \oplus ys]\!]_B = [\![xs]\!]_B + [\![ys]\!]_B$

Paso inductivo $(x \triangleright xs,\ y \triangleright ys)$

$$
\begin{array}{ll}
 & [\![x \triangleright xs \oplus y \triangleright ys]\!]_B \\
= & \{ \text{ especificación de } \oplus \ \} \\
 & [\![x \triangleright xs]\!]_B + [\![y \triangleright ys]\!]_B \\
= & \{ \ \mathrm{P}_2 \ \} \\
 & x + B * [\![xs]\!]_B + y + B * [\![ys]\!]_B \\
= & \{ \text{ Conmutatividad y asociatividad de la suma } \} \\
 & (x + y) + B * [\![xs]\!]_B + B * [\![ys]\!]_B \\
= & \{ \text{ Distributividad de la suma respecto al producto } \} \\
 & (x + y) + B * ([\![xs]\!]_B + [\![ys]\!]_B) \\
= & \{ \text{ hipótesis inductiva } \} \\
 & (x + y) + B * [\![xs \oplus ys]\!]_B \\
= & \{ \ \mathrm{P}_2 \ \} \\
 & [\![(x + y) \triangleright (xs \oplus ys)]\!]_B
\end{array}
$$

Como fue mencionado, todavía no pudimos encontrar una definición recursiva de $\oplus$. A lo sumo demostramos, por el principio de inducción (capítulo 10), que si el operador cumple:

$$
\begin{array}{rcl}
[\![[\,] \oplus ys]\!]_B & = & [\![ys]\!]_B \\
[\![xs \oplus [\,]]\!]_B & = & [\![xs]\!]_B \\
[\![x \triangleright xs \oplus y \triangleright ys]\!]_B & = & [\![(x + y) \triangleright (xs \oplus ys)]\!]_B
\end{array}
\tag{14.1}
$$

entonces se satisface la especificación:

$$[\![xs \oplus ys]\!]_B = [\![xs]\!]_B + [\![ys]\!]_B \ .$$

Pero el problema se puede solucionar fácilmente; la propiedad 14.1 del operador $\oplus$ es implicada, usando Leibniz (sección 1.6), por la siguiente definición

$$
\begin{array}{|l}
(\oplus) : [Num] \mapsto [Num] \mapsto [Num] \\
\hline
\begin{array}{rcl}
[\,] \oplus ys & \doteq & ys \\
xs \oplus [\,] & \doteq & xs \\
x \triangleright xs \oplus y \triangleright ys & \doteq & (x + y) \triangleright (xs \oplus ys)
\end{array}
\end{array}
$$

y con esto logramos eliminar la función de abstracción obteniendo la definición recursiva buscada.

En la próxima sección veremos más detalladamente el proceso realizado.

14.2 Especificaciones implícitas

Analicemos de forma más abstracta el problema y la resolución que acabamos de ver.

La mayor parte de las especificaciones vistas en capítulos anteriores declaran explícitamente la función a derivar. Esto es, las especificaciones tienen la forma

$$\boxed{f.X = E.X}$$

donde f es la función a encontrar, X su lista de parámetros y E una expresión que depende de los mismos (generalmente una expresión cuantificada). Ejemplo de este tipo de especificaciones es la evaluación de un polinomio (pág. 189)

$$\boxed{ev.xs.y = \langle \sum i \ : 0 \leq i < \#xs : \ xs.i * y^i \ \rangle}$$

A esta clase de especificaciones las denominaremos **especificaciones explícitas**.

En el ejemplo anterior sucedió que fue más claro y conveniente hacer una especificación como una ecuación de la función buscada, con la forma

$$\boxed{g.(f.X) = E.X}$$

donde g es una función definida (en nuestro ejemplo la función de abstracción "$[\![\_]\!]_B$"), f es la función a encontrar ("$\oplus$" en el ejemplo), X su lista de parámetros y E una expresión que depende de los mismos. A esta clase de especificaciones las denominaremos **especificaciones implícitas** y una estrategia para encontrar la función f es, a partir del miembro izquierdo de la igualdad, reescribir el derecho $E.X$ en una expresión equivalente de la forma $g.(E'.X)$ (en el ejemplo aplicando inducción). Con esto obtenemos la igualdad $g.(f.X) = g.(E'.X)$ la cual cumple

$$\begin{array}{ll} & g.(f.X) = E.X \\ \Leftarrow & \{ \text{ Por derivación } \} \\ & g.(f.X) = g.(E'.X) \end{array}$$

concluyendo que la igualdad encontrada implica la especificación implícita desde la cual partimos.

Además, si definimos

$$f.X \doteq E'.X \ ,$$

por la regla de Leibniz se satisface

$$\begin{array}{ll} & g.(f.X) = g.(E'.X) \\ \Leftarrow & \{ \text{ Leibniz } \} \\ & f.X = E'.X \end{array}$$

Por lo tanto, juntando los dos pasos previos, obtenemos

$$
\begin{array}{ll}
& g.(f.X) = E.X \\
\Leftarrow & \{ \text{ Por derivación } \} \\
& g.(f.X) = g.(E'.X) \\
\Leftarrow & \{ \text{ Leibniz } \} \\
& f.X = E'.X
\end{array}
$$

y aplicando transitividad de $\Rightarrow$ llegamos a

$$f.X = E'.X \Rightarrow g.(f.X) = E.X$$

lo cual nos dice que si definimos la función incógnita f como

$$\boxed{f.X \doteq E'.X}$$

se satisfará su especificación implícita $g.(f.X) = E.X$ y con ello encontramos la función que estábamos buscando.

14.3 Revisión del ejemplo

En el ejemplo inicial vimos cómo obtener la suma de dos números de precisión arbitraria representados como listas. Las listas contenían, en orden inverso, los dígitos del número en una base dada. Pero hemos olvidado un detalle; la especificación que dimos del operador $\oplus$ nada dice acerca que los dígitos del resultado deben estar entre 0 y la base. Una forma de hacerlo es definir un predicado I_B

$$\boxed{I_B.xs = \langle \forall\, i \ : 0 \leq i < \#xs : \ 0 \leq xs.i < B \,\rangle}$$

el cual requiere para su validez que todos los elementos de la lista se encuentren entre 0 y $B-1$. Con el mismo podemos especificar el operador $\oplus$ como

$$\left|\begin{array}{l} (\oplus) : [Num] \mapsto [Num] \mapsto [Num] \\ \hline [\![xs \oplus ys]\!]_B = [\![xs]\!]_B + [\![ys]\!]_B \ \wedge \ I_B.(xs \oplus ys) \end{array}\right.$$

y con ello requerimos que el resultado de la operación cumpla I_B.

Notemos que esta nueva especificación es más fuerte que la anterior (agrega un término conjuntivo). Podríamos debilitarla levemente solicitando que los parámetros xs y ys verifiquen el predicado I_B:

$$\left|\begin{array}{l} (\oplus) : [Num] \mapsto [Num] \mapsto [Num] \\ \hline I_B.xs \ \wedge \ I_B.ys \\ \ \Rightarrow [\![xs \oplus ys]\!]_B = [\![xs]\!]_B + [\![ys]\!]_B \ \wedge \ I_B.(xs \oplus ys) \end{array}\right.$$

Esta nueva especificación es acorde a los requerimientos del problema ya que supone que todas las representaciones de enteros que usaremos son listas de enteros de precisión acotada.

Veamos si nuestra definición del operador $\oplus$ (pág. 220) cumple con esta especificación. Según su derivación, esta definición satisface $[\![xs \oplus ys]\!]_B = [\![xs]\!]_B + [\![ys]\!]_B$. Restaría ver si cumple el predicado I_B. Para ello instrumentaremos la prueba de la forma usual: supondremos válido $I_B.xs \wedge I_B.ys$ e intentaremos demostrar $I_B.(xs \oplus ys)$ para el caso inductivo. Los casos bases lo satisfacen y se dejan como ejercicio al lector.

Hipótesis inductiva: $I_B.(xs \oplus ys)$

Suposiciones: $I_B.(x \triangleright xs)$, $I_B.(y \triangleright ys)$

Paso inductivo $(x \triangleright xs,\ y \triangleright ys)$

$$
\begin{array}{ll}
 & I_B.(x \triangleright xs \oplus y \triangleright ys) \\
\equiv & \{\ \text{definición de } \oplus\ \} \\
 & I.((x+y) \triangleright (xs \oplus ys)) \\
\equiv & \{\ \text{definición de } I_B\ \} \\
 & \langle \forall\, i\ : 0 \le i < \#((x+y) \triangleright (xs \oplus ys)) :\ 0 \le ((x+y) \triangleright (xs \oplus ys)).i < B \,\rangle \\
\equiv & \{\ \text{definición de } \#\text{; partición de rango; rango unitario}\ \} \\
 & 0 \le x+y < B \ \wedge \\
 & \quad \langle \forall\, i\ : 0 < i < \#(xs \oplus ys) + 1 :\ 0 \le ((x+y) \triangleright (xs \oplus ys)).i < B \,\rangle \\
\equiv & \{\ \text{reemplazo } i \leftarrow i+1\text{, aritmética; definición de } \triangleright\ \} \\
 & 0 \le x+y < B \ \wedge\ \langle \forall\, i\ : 0 \le i < \#(xs \oplus ys) :\ 0 \le (xs \oplus ys).i < B \,\rangle \\
\equiv & \{\ \text{definición de } I_B\ \} \\
 & 0 \le x+y < B \ \wedge\ I_B.(xs \oplus ys) \\
\equiv & \{\ \text{hipótesis inductiva}\ \} \\
 & 0 \le x+y < B
\end{array}
$$

En este punto no podemos avanzar más con la demostración; aunque por las suposiciones tenemos que $0 \le x < B$ y $0 \le y < B$, a partir de ellas no es cierto en general que $0 \le x + y < B$. Por lo tanto nuestra definición de $\oplus$ no cumple con la nueva especificación.

Intentemos obtener entonces otra definición de $\oplus$ volviendo a hacer la derivación de la página 219 pero ahora teniendo en cuenta las suposiciones y la restricción sobre el tamaño de los dígitos. Para los casos base dejaremos los mismos resultados ya que estos sí cumplen con el predicado I_B. Veamos el caso inductivo analizando dos posibilidades: $x + y < B$ y $x + y \ge B$.

Suposiciones: $I_B.(x \triangleright xs)$, $I_B.(y \triangleright ys)$

Caso: $x + y < B$

Paso inductivo $(x \triangleright xs,\ y \triangleright ys)$

$$
\begin{aligned}
& [\![x \triangleright xs \oplus y \triangleright ys]\!]_B \\
= & \ \{ \text{ por demostración anterior } \} \\
& [\![(x + y) \triangleright (xs \oplus ys)]\!]_B
\end{aligned}
$$

Como vemos, para este caso podemos dejar la misma definición de $\oplus$ obtenida anteriormente, ya que la condición $x+y < B$ asegura que los dígitos están acotados por la base.

Veamos el segundo caso

Caso: $x + y \geq B$

Paso inductivo $(x \triangleright xs,\ y \triangleright ys)$

$$
\begin{aligned}
& [\![x \triangleright xs \oplus y \triangleright ys]\!]_B \\
= & \ \{ \text{ por demostración anterior } \} \\
& (x + y) + B * ([\![xs]\!]_B + [\![ys]\!]_B) \\
= & \ \{ \text{ aritmética } \} \\
& (x + y - B) + B + B * ([\![xs]\!]_B + [\![ys]\!]_B) \\
= & \ \{ \text{ aritmética } \} \\
& (x + y - B) + B * ((1 + [\![xs]\!]_B) + [\![ys]\!]_B) \\
= & \ \{ \text{ definición de } [\![\_]\!]_B \ \} \\
& (x + y - B) + B * (([\![[1]]\!]_B + [\![xs]\!]_B) + [\![ys]\!]_B) \\
= & \ \{ \text{ hipótesis inductiva } \} \\
& (x + y - B) + B * ([\![[1] \oplus xs]\!]_B + [\![ys]\!]_B) \\
= & \ \{ \text{ hipótesis inductiva } \} \\
& (x + y - B) + B * ([\![[1] \oplus xs \oplus ys]\!]_B) \\
= & \ \{ \ \mathrm{P}_2 \ \} \\
& [\![(x + y - B) \triangleright ([1] \oplus xs \oplus ys)]\!]_B
\end{aligned}
$$

Por lo tanto

$$[\![x \triangleright xs \oplus y \triangleright ys]\!]_B = [\![(x + y - B) \triangleright ([1] \oplus xs \oplus ys)]\!]_B$$

$\Leftarrow$ { Leibniz }

$x \triangleright xs \oplus y \triangleright ys = (x + y - B) \triangleright ([1] \oplus xs \oplus ys)$

Nota: En la demostración anterior se aplicó dos veces hipótesis inductiva. En ambas se hizo inducción completa sobre la longitud de la concatenación de las dos listas parámetros del operador $\oplus$.

Ejercicio: Comprobar con esta hipótesis inductiva que la derivación es correcta.

A partir de las demostraciones anteriores hemos encontrado una nueva definición del operador $\oplus$:

$$
\begin{array}{|l}
\hline
(\oplus) : [Num] \mapsto [Num] \mapsto [Num] \\
\hline
\begin{array}{rcl}
[\,] \oplus ys & \doteq & ys \\
xs \oplus [\,] & \doteq & xs \\
x \triangleright xs \oplus y \triangleright ys & \doteq & (\ \ x + y < B \ \rightarrow\ (x + y) \triangleright (xs \oplus ys) \\
 & & \Box\ x + y \geq B \ \rightarrow\ (x + y - B) \triangleright ([1] \oplus xs \oplus ys) \\
 & &)
\end{array}
\end{array}
$$

Falta ahora demostrar que este resultado cumple el predicado I_B suponiendo que lo cumplen los parámetros. Los casos base son triviales y se dejan como ejercicios para el lector. El caso inductivo tendrá dos casos al igual que la derivación anterior y se utilizará la misma hipótesis inductiva que en la última derivación. Para el caso $x + y \geq B$ tendremos

Suposiciones: $I_B.(x \triangleright xs)$, $I_B.(y \triangleright ys)$

Caso: $x + y \geq B$

Paso inductivo $(x \triangleright xs,\ y \triangleright ys)$

$I_B.(x \triangleright xs \oplus y \triangleright ys)$

$\equiv$ { definición de $\oplus$ }

$I.((x + y - B) \triangleright ([1] \oplus xs \oplus ys))$

$\equiv$ { definición de I_B }

$\langle \forall\, i\ : 0 \leq i < \#((x + y - B) \triangleright ([1] \oplus xs \oplus ys)) :$
$\qquad 0 \leq ((x + y - B) \triangleright ([1] \oplus xs \oplus ys)).i < B \,\rangle$

$\equiv$ { definición de $\#$; partición de rango; rango unitario }

$0 \leq x + y - B < B \ \wedge \quad \langle \forall\, i\ : 0 < i < \#([1] \oplus xs \oplus ys) + 1 :$
$\qquad\qquad 0 \leq ((x + y - B) \triangleright ([1] \oplus xs \oplus ys)).i < B \,\rangle$

$\equiv$ { reemplazo $i \leftarrow i+1$, aritmética; definición de $\triangleright$ }

$0 \leq x + y - B < B \ \wedge$
$\langle \forall\, i \ : 0 \leq i < \#([1] \oplus xs \oplus ys) : \ 0 \leq ([1] \oplus xs \oplus ys).i < B \,\rangle$

$\equiv$ { definición de I_B }

$0 \leq x + y - B < B \ \wedge \ I_B.([1] \oplus xs \oplus ys)$

$\equiv$ { suposiciones; aritmética; cálculo proposicional }

$I_B.([1] \oplus xs \oplus ys)$

$\equiv$ { hipótesis inductiva dos veces }

$I_B.[1] \ \wedge \ I_B.xs \ \wedge \ I_B.ys$

$\equiv$ { suposiciones; cálculo proposicional }

$I_B.[1]$

$\equiv$ { definición de I_B; $B > 1$; aritmética }

$True$

El caso en que $x + y < B$ es trivial y se deja como ejercicio al lector.
Con las dos últimas demostraciones probamos respectivamente que

$$I_B.xs \ \wedge \ I_B.ys \Rightarrow [\![xs \oplus ys]\!]_B = [\![xs]\!]_B + [\![ys]\!]_B$$

y

$$I_B.xs \ \wedge \ I_B.ys \Rightarrow I_B.(xs \oplus ys) \ ,$$

por lo tanto hemos encontrado una definición de $\oplus$ que satisface su nueva especificación.

14.4 Especificaciones implícitas con invariante de representación

Veamos de forma más general el proceso de derivación realizado en el ejemplo anterior. En el mismo comenzamos con una especificación un tanto más complicada, donde además de una especificación implícita aparece un predicado I de la siguiente manera:

$$\boxed{\begin{array}{l}\langle \forall\, x \ : x \text{ parámetro en la lista } X : \ I.x \,\rangle \\ \ \Rightarrow g.(f.X) = E.X \ \wedge \ I.(f.X)\end{array}}$$

A este predicado lo llamaremos *invariante de representación*. En palabras, lo que esta especificación dice es que si el invariante de representación se cumple sobre los parámetros, entonces el resultado también debe cumplirlo (además de cumplir con la especificación implícita). Por esto mismo su nombre; es una condición que

se cumple en los parámetros y en el resultado, o sea que se mantiene invariante al aplicar la función. Este tipo de especificaciones funcionan claramente como un contrato (ver sección 9.1) entre el programador y el potencial usuario del programa. En él se establece que si el operador se ejecuta para cierto conjunto de valores, entonces el resultado satisfará la propiedad deseada.

Para simplificar la notación escribiremos $I.X$ en vez de la cuantificación universal sobre la lista de parámetro. Con ello la especificación quedará como

$$\boxed{\begin{array}{l} I.X \\ \Rightarrow g.(f.X) = E.X \ \wedge \ I.(f.X) \end{array}}$$

Para obtener una definición de la función incógnita f trabajaremos como usualmente hemos hecho cuando queremos demostrar implicaciones. Supondremos válido $I.X$ y trabajaremos con cada término de la conjunción (del lado derecho de la implicación) por separado.

El primer término de la conjunción es $g.(f.X) = E.X$ y tiene la forma de una especificación implícita. Por lo tanto repetiremos la metodología vista para las mismas aunque suponiendo que el invariante de representación se cumple para los parámetros.

Suposición: $I.X$

$$\begin{array}{ll} & g.(f.X) = E.X \\ \Leftarrow & \{ \text{ Por derivación suponiendo } I.X \ \} \\ & g.(f.X) = g.(E'.X) \\ \Leftarrow & \{ \text{ Leibniz } \} \\ & f.X = E'.X \end{array}$$

Como sucedió con las especificaciones implícitas, ya tenemos un candidato a definición de la función f. Pero todavía no demostramos que esta definición cumple el invariante. Solo demostramos

$$I.X \Rightarrow (f.X = E'.X \Rightarrow g.(f.X) = E.X)$$

o lo que es lo mismo (por cálculo proposicional)

$$f.X = E'.X \ \wedge \ I.X \Rightarrow g.(f.X) = E.X \ . \tag{14.2}$$

Esto es, si la función está definida como fue derivada y se cumple el invariante de representación sobre los parámetros, entonces se cumple la parte implícita de la especificación. Todavía resta demostrar que definiendo la función de esta forma, se cumple el invariante. En el último ejemplo realizamos esto probando:

$$f.X = E'.X \ \wedge \ I.X \Rightarrow I.(f.X) \ . \tag{14.3}$$

Esto es, si la función está definida como fue derivada y se supone $I.X$, entonces el resultado también cumple el invariante I (ver que en el ejemplo anterior se hizo justamente esto).

Por lo tanto, a partir de (14.2), (14.3) y aplicando cálculo proposicional, demostramos

$$f.X = E'.X \Rightarrow (I.X \Rightarrow g.(f.X) = E.X \wedge I.(f.X))$$

lo cual dice que la función derivada cumple con su especificación.

Es posible que esta demostración no pueda ser realizada, debido a que simplemente el resultado obtenido no cumple el invariante (como sucedió en el ejemplo). En este caso habrá que rever la derivación hecha a partir de la especificación implícita para obtener una nueva definición de la función f que la verifique. Con este nuevo resultado habrá que volver a intentar demostrar el invariante (ecuación (14.3)).

14.5 Ejercicios

Ejercicio 14.1
Derivar el operador multiplicación de enteros de precisión arbitraria. La especificación del mismo es la siguiente:

$$\begin{array}{|l} (\otimes) : [Num] \mapsto [Num] \mapsto [Num] \\ \hline I_B.xs \wedge I_B.ys \\ \quad \Rightarrow [\![xs \otimes ys]\!]_B = [\![xs]\!]_B * [\![ys]\!]_B \wedge I_B.(xs \otimes ys) \end{array}$$

Ejercicio 14.2
Derivar la función $conv_B$ que dado un número mayor o igual a cero, devuelve su representación en base B. Esto es

$$\begin{array}{|l} conv_B : Num \mapsto [Num] \\ \hline [\![conv_B.n]\!]_B = n \wedge I_B.(conv_B.n) \end{array}$$

Nota: La función $conv_B$ está definida como la inversa a derecha de la función de abstracción $[\![\_]\!]_B$. En particular, esto quiere decir que la función $[\![\_]\!]_B$ es sobreyectiva pero no necesariamente inyectiva. En caso que no lo sea buscar algunos contraejemplos en que la función evaluada en distintos valores devuelva como resultado el mismo número.

Ejercicio 14.3
Rehacer el ejemplo de la sección 14.3 generalizando el operador $\oplus$ con la función definida como

$$
\begin{array}{|l}
\hline
g : Num \mapsto [Num] \mapsto [Num] \mapsto [Num] \\
\hline
I_B.xs \ \wedge \ I_B.ys \\
\ \Rightarrow [\![g.k.xs.ys]\!]_B = k + [\![xs]\!]_B + [\![ys]\!]_B \ \wedge \ I_B.(g.k.xs.ys) \\
\hline
\end{array}
$$

Demostrar también que la función g especificada de esta forma, es una generalización del operador $\oplus$. Esto es, demostrar que si se define $\oplus$ como

$$
\boxed{xs \oplus ys \doteq g.0.xs.ys}
$$

entonces satisface su especificación.

Ejercicio 14.4
Los números representados en el ejemplo de aritmética de precisión arbitraria son siempre positivos. Una forma de implementar además los negativos es cambiando ligeramente la representación de los números:

$$
\begin{array}{|l}
\hline
[\![\_]\!]_B : (Bool, [Num]) \mapsto Num \\
\hline
[\![(s, xs)]\!]_B \ = \ (\quad s \rightarrow \quad \langle \sum i \ : 0 \leq i < \#xs : \ xs.i * B^i \rangle \\
\qquad\qquad\quad \Box \neg s \rightarrow - \langle \sum i \ : 0 \leq i < \#xs : \ xs.i * B^i \rangle \\
\qquad\qquad\quad) \\
\hline
\end{array}
$$

Esto es, si el booleano s es verdadero el número será positivo y en caso contrario negativo.

A partir de esta nueva función de abstracción especificar y derivar las operaciones de suma, resta y multiplicación de enteros de precisión arbitraria.

Ejercicio 14.5 (Notación factorial)
En el ejercicio 12.7 se planteó el problema de evaluar un número entero positivo representado en su notación factorial. La función de abstracción fue planteada como:

$$
\begin{array}{|l}
\hline
[\![\_]\!] : [Nat] \mapsto Nat \\
\hline
[\![xs]\!] = \langle \sum i \ : 0 \leq i < \#xs : \ xs.i * (i+1)! \rangle \\
\hline
\end{array}
$$

Además, los números así representados cumplen con el siguiente invariante de representación:

$$
\boxed{I.xs = \langle \forall i \ : 0 \leq i < \#xs : \ 0 \leq xs.i \leq i + 1 \rangle}
$$

esto es, los dígitos deben ser números positivos menores o iguales al número cuyo factorial los multiplica.

Algunos ejemplos de números en esta representación son:

$$\begin{array}{lcllcl} [\![[0,1]]\!] & = & 2 & [\![[1,2]]\!] & = & 5 \\ [\![[1,1,1]]\!] & = & 9 & [\![[0,2,1]]\!] & = & 10 \\ [\![[1,2,3]]\!] & = & 23 & [\![[0,0,0,1]]\!] & = & 24 \end{array}$$

Especificar y derivar la función que dado un número devuelve su representación factorial y el operador de suma en esta representación.

Ayuda: Para obtener la función que devuelva la representación factorial de un número será necesario generalizarla. Ver la generalización en el ejercicio 12.7 y plantear una inversa a derecha de aquella.

Capítulo 15

Recursión final

> Unwin, cansado, lo detuvo.
> - No multipliques los misterios - le dijo -. Estos deben ser simples. Recuerda la carta robada de Poe, recuerda el cuarto cerrado de Zangwill.
> - O complejos - replicó Dunraven -. Recuerda el universo
>
> Jorge Luis Borges:
> *Abenjacán el Bojarí muerto en su laberinto*

Existen algunos tipos de definiciones recursivas que gozan de buenas propiedades. Uno de los esquemas recursivos más importantes es el llamado de recursión final, o recursión a la cola (en inglés, tail recursion). Una de las propiedades importantes de dicho esquema es que las funciones así definidas pueden traducirse fácilmente a programas iterativos. Más aún, la prueba de corrección del programa iterativo puede obtenerse a partir de la definición recursiva de la función.

Diremos que una función es **recursiva final** si es posible definirla explícitamente de la siguiente manera:

$$H : A \mapsto B$$

$$\begin{array}{lcl} H.x & \doteq & (\quad b.x \rightarrow f.x \\ & & \Box \ \neg b.x \rightarrow H.(g.x) \\ & &) \end{array}$$

donde puede deducirse el tipo de las componentes:

$$\begin{array}{ll} b: & A \mapsto Bool \\ f: & A \mapsto B \\ g: & A \mapsto A \end{array}$$

Nótese el abuso de notación implícito en la función g. Si, por ejemplo, H es una función de dos parámetros, en realidad harán falta dos funciones -sean estas g_0, g_1- (o una función que devuelva pares), quedando entonces la definición explícita de la siguiente manera:

$$\begin{array}{lcl} H.x.y & \doteq & (\quad b.x.y \rightarrow f.x.y \\ & & \Box \ \neg b.x.y \rightarrow H.(g_0.x.y).(g_1.x.y) \\ & &) \end{array}$$

En la definición de $H.x$ la recursividad se da solo cuando no se cumple la condición $b.x$ y consiste en una aplicación de la función a una modificación del argumento original. El esquema de cáclulo de la función es modificar sucesivamente el argumento hasta llegar a un valor que satisface la condición b; en este punto se acaba la reducción.

Para asegurarnos que la condición b efectivamente se hace cierta, requeriremos también la existencia de una función

$$t : A \mapsto Int$$

la cual satisface

$$\begin{array}{lcl} \neg b.x & \Rightarrow & t.x > 0 \\ \neg b.x & \Rightarrow & t.(g.x) < t.x \end{array}$$

Ejemplo 15.1
La función mcd (máximo común divisor) se puede definir, para $x, y > 0$, como una recursión final:

$$\boxed{\begin{array}{lcl} mcd.x.y & \doteq & (\quad x = y \rightarrow x \\ & & \Box \quad x \neq y \rightarrow mcd.(max.x.y - min.x.y).(min.x.y) \\ & &) \end{array}}$$

En este caso, las funciones involucradas son:

$b.x.y \equiv (x = y)$
$f.x.y = x$
$g_0.x.y = max.x.y - min.x.y$
$g_1.x.y = min.x.y$

La función t puede ser, por ejemplo, $t.x.y = x + y$. Efectivamente, si $x \neq y$,

$$
\begin{array}{ll}
& t.x.y \\
= & \{ \text{ definición de } t \ \} \\
& x + y \\
> & \{ \ x, y > 0 \ \} \\
& 0
\end{array}
$$

$$
\begin{array}{ll}
& t.(g_0.x.y).(g_1.x.y) \\
= & \{ \text{ definición de } t \ \} \\
& max.x.y - min.x.y + min.x.y \\
= & \{ \text{ álgebra } \} \\
& max.x.y \\
< & \{ \ x, y > 0 \ \} \\
& x + y \\
= & \{ \text{ definición de } t \ \} \\
& t.x.y
\end{array}
$$

Ejemplo 15.2
La definición recursiva de *fac* que hemos dado en capítulos anteriores,

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

no es recursiva final (por la aparición del factor $n + 1$).

Muchas funciones recursivas pueden expresarse como recursivas finales. Una forma de lograrlo es usar la técnica de generalización por abstracción de la función. Veamos cómo hacerlo con la función *fac*:

Sea *gfac* especificada por

$$\langle \, \forall \, m, n \ :: \ gfac.m.n = m * fac.n \, \rangle \quad .$$

Es claro que *gfac* generaliza a *fac*, pues $fac.n = gfac.1.n$. Derivemos una definición recursiva final de *gfac*.

Caso base

$$
\begin{array}{ll}
& gfac.m.0 \\
= & \{ \text{ especificación de } gfac \ \} \\
& m * fac.0 \\
= & \{ \text{ definición de } fac \ \} \\
& m * 1 \\
= & \{ \text{ álgebra } \} \\
& m
\end{array}
$$

Paso inductivo

$$
\begin{array}{ll}
& gfac.m.(n+1) \\
= & \{ \text{ especificación de } gfac \ \} \\
& m * fac.(n+1) \\
= & \{ \text{ definición de } fac \ \} \\
& m * ((n+1) * fac.n) \\
= & \{ \text{ álgebra } \} \\
& (m * (n+1)) * fac.n \\
= & \{ \text{ hipótesis inductiva } \} \\
& gfac.(m * (n+1)).n
\end{array}
$$

Por lo tanto,

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

que es recursiva final, pues puede escribirse con el esquema descripto al comienzo del capítulo:

$$\begin{array}{rcl} gfac.m.n & \doteq & (\quad n = 0 \to m \\ & & \Box \; n \neq 0 \to gfac.(m*n).(n-1) \\ & &) \end{array}$$

La principal dificultad que se presenta al buscar una definición recursiva final de una función, es determinar la función generalizada a partir de la cual se derivará la expresión recursiva final. Si bien no daremos reglas generales para encontrar esta expresión, el problema puede generalmente resolverse usando la técnica de generalización por abstracción.

Ejemplo 15.3
Dar una definición recursiva final de la función especificada por:

$$g.x.n = \langle \textstyle\sum i \; : 0 \leq i < n : \; x^i \rangle \quad .$$

En la sección 11.5 dimos una definición recursiva para g:

$$\begin{array}{rcll} g.x.0 & \doteq & 0 & \\ g.x.(2*n) & \doteq & (1+x)*g.(x*x).n & si\ n > 0 \\ g.x.(2*n+1) & \doteq & 1 + x*g.x.(2*n) & \end{array}$$

Observando esta definición, vemos que su "forma general" es la siguiente: cuando el segundo argumento es par, el valor asignado es un múltiplo de g, mientras que cuando el segundo argumento es impar, el resultado es un múltiplo de g más una constante. Esto sugiere proponer una función generalizada de la siguiente forma:

$$h.z.y.x.n = z + y * g.x.n \quad .$$

Derivemos una definición recursiva final de h.

Caso base $(n = 0)$

$$\begin{array}{ll} & h.z.y.x.0 \\ = & \{ \text{ especificación de } h \; \} \\ & z + y * g.x.0 \end{array}$$

$$
\begin{array}{ll}
= & \{ \text{ definición de } g \ \} \\
& z + y * 0 \\
= & \{ \text{ álgebra } \} \\
& z
\end{array}
$$

Paso inductivo $(n = 2{*}k, k > 0)$

$$
\begin{array}{ll}
& h.z.y.x.(2 * k) \\
= & \{ \text{ especificación de } h \ \} \\
& z + y * g.x.(2 * k) \\
= & \{ \text{ definición de } g,\ k > 0 \ \} \\
& z + y * (1 + x) * g.(x * x).k \\
= & \{ \text{ álgebra } \} \\
& z + (y * (1 + x)) * g.(x * x).k \\
= & \{ \text{ hipótesis inductiva } \} \\
& h.z.(y * (1 + x)).(x * x).k
\end{array}
$$

Paso inductivo $(n = 2{*}k{+}1)$

$$
\begin{array}{ll}
& h.z.y.x.(2 * k + 1) \\
= & \{ \text{ especificación de } h \ \} \\
& z + y * g.x.(2 * k + 1) \\
= & \{ \text{ definición de } g \ \} \\
& z + y * (1 + x * g.x.(2 * k)) \\
= & \{ \text{ álgebra } \} \\
& z + y + y * x * g.x.(2 * k) \\
= & \{ \text{ álgebra } \} \\
& (z + y) + (y * x) * g.x.(2 * k) \\
= & \{ \text{ hipótesis inductiva } \} \\
& h.(z + y).(y * x).x.(2 * k)
\end{array}
$$

Por lo tanto,

$$
\boxed{
\begin{array}{rcll}
h.z.y.x.0 & \doteq & z & \\
h.z.y.x.(2 * k) & \doteq & h.z.(y * (1 + x)).(x * x).k & si\ k > 0 \\
h.z.y.x.(2 * k + 1) & \doteq & h.(z + y).(y * x).x.(2 * k) &
\end{array}
}
$$

que es una definición recursiva final, la cual puede escribirse como:

$$
\boxed{
\begin{array}{l}
h.z.y.x.n \ \doteq \ (\ \ n = 0 \rightarrow z \\
\qquad \Box\ n \neq 0 \rightarrow (\ \ n \ \mathsf{mod}\ 2 = 0 \rightarrow \\
\qquad\qquad\qquad\qquad h.z.(y * (1 + x)).(x * x).(n \ \mathsf{div}\ 2) \\
\qquad\qquad\quad \Box\ n \ \mathsf{mod}\ 2 \neq 0 \rightarrow \\
\qquad\qquad\qquad\qquad h.(z + y).(y * x).x.(2 * (n \ \mathsf{div}\ 2)) \\
\qquad\qquad\quad) \\
\qquad\quad)
\end{array}
}
$$

Este programa puede traducirse directamente a uno iterativo. Nótese que la derivación es esencial para asegurarse la corrección del mismo, dada la complejidad de las expresiones obtenidas.

15.1 Ejemplos de funciones recursivas finales

En el capítulo 12 desarrollamos una serie de ejemplos, derivando para cada uno de los problemas planteados una definición recursiva para el cálculo de la función deseada. En esta sección derivaremos definiciones recursivas finales para algunos de dichos ejemplos. Los demás serán tratados más adelante o se dejarán como ejercicio.

Ejemplo 15.4 (Evaluación de un polinomio)
Recordemos que el problema consiste en la evaluación de un polinomio de la forma $p(x) = a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$ en un valor dado de la variable independiente, digamos y. Para calcular $p(y)$ definimos recursivamente una función $ev.xs.y$, donde xs era la lista de los coeficientes de $p(x)$, de la siguiente manera:

$$\begin{array}{|rcl} ev.[\,].y & \doteq & 0 \\ ev.(x \triangleright xs).y & \doteq & x + y * ev.xs.y \end{array}$$

Buscamos ahora una definición recursiva final de ev. Para ello procederemos usando la técnica de generalización por abstracción. Sea

$$gev.xs.y.z.w = w + z * ev.xs.y \ .$$

Es claro que gev generaliza a ev, pues $ev.xs.y = gev.xs.y.1.0$. Derivemos ahora una definición recursiva para gev:

Caso base $(xs = [\,])$

$$\begin{array}{ll} & gev.[\,].y.z.w \\ = & \{ \text{ especificación de } gev \ \} \\ & w + z * ev.[\,].y \\ = & \{ \text{ definición de } ev \ \} \\ & w + z * 0 \\ = & \{ \text{ álgebra } \} \\ & w \end{array}$$

Paso inductivo $(x \triangleright xs)$

$$\begin{array}{ll} & gev.(x \triangleright xs).y.z.w \\ = & \{ \text{ definición de } gev \ \} \\ & w + z * ev.(x \triangleright xs).y \\ = & \{ \text{ definición de } ev \ \} \\ & w + z * (x + y * ev.xs.y) \\ = & \{ \text{ álgebra } \} \\ & (w + z * x) + (z * y) * ev.xs.y \\ = & \{ \text{ hipótesis inductiva } \} \\ & gev.xs.y.(z * y).(w + z * x) \end{array}$$

Es decir, tenemos la siguiente definición recursiva de gev:

$$\begin{array}{|rcl} gev.[\,].y.z.w & \doteq & w \\ gev.(x \triangleright xs).y.z.w & \doteq & gev.xs.y.(z * y).(w + z * x) \end{array}$$

la cual es recursiva final y puede escribirse de la siguiente manera:

$$\begin{array}{|l} \overline{gev.xs.y.z.w} \doteq (\ xs = [\,] \to w \\ \qquad\qquad\quad \Box\, xs \neq [\,] \to gev.(xs{\downarrow}1).y.(z * y).(w + z * xs.0) \\ \qquad\qquad\quad) \end{array}$$

Ejemplo 15.5 (Un problema para listas de números)
Dada una lista xs de números, la función $P.xs$ determina si alguno de los elementos de xs es igual a la suma de todos los elementos que lo preceden. Este problema se resolvió generalizando la función P con gP, para la cual se obtuvo la siguiente definición recursiva:

$$\begin{array}{|rl} \overline{\qquad gP.n.[\,]} & \doteq False \\ gP.n.(x \triangleright xs) & \doteq (x = n) \vee gP.(n + x).xs \end{array}$$

Para encontrar una definición recursiva final para gP usaremos nuevamente la técnica de generalización por abstracción. Sea

$$hP.b.n.xs = b \vee gP.n.xs\ .$$

La función hP generaliza a gP, pues $gP.n.xs = hP.\,False\,.n.xs$. Derivemos ahora una definición recursiva para hP:

Caso base $(xs = [\,])$

$$\begin{array}{ll} & hP.b.n.[\,] \\ = & \{\ \text{especificación de } hP\ \} \\ & b \vee gP.n.[\,] \\ = & \{\ \text{definición de } gP\ \} \\ & b \vee False \\ = & \{\ \text{cálculo proposicional}\ \} \\ & b \end{array}$$

Paso inductivo $(x \triangleright xs)$

$$\begin{array}{ll} & hP.b.n.(x \triangleright xs) \\ = & \{\ \text{definición de } hP\ \} \\ & b \vee gP.n.(x \triangleright xs) \\ = & \{\ \text{definición de } gP\ \} \\ & b \vee x = n \vee gP.n.xs \\ = & \{\ \text{hipótesis inductiva}\ \} \\ & hP.(b \vee x = n).n.xs \end{array}$$

Por lo tanto una definición recursiva final para el cálculo de P es

$$\begin{array}{|l} \overline{hP.b.n.xs} \doteq (\ xs = [\,] \to b \\ \qquad\qquad\quad \Box\, xs \neq [\,] \to hP.(b \vee xs.0 = n).n.(xs{\downarrow}1) \\ \qquad\qquad\quad) \end{array}$$

15.2 Recursión lineal y recursión final

Un esquema más general que la recursión final es el de recursión lineal. En este esquema, la referencia recursiva aparece a lo sumo una vez (en cada posible rama).

Diremos que una función es **recursiva lineal** si es posible definirla explícitamente como

$$F : A \mapsto B$$

$$\begin{array}{lcl} F.x & \doteq & (\quad b.x \ \rightarrow f.x \\ & & \Box \ \neg b.x \ \rightarrow h.x \oplus F.(g.x) \\ & &) \end{array}$$

donde $\oplus$ puede ser cualquier operador.

La función h es -estrictamente hablando- innecesaria (se puede evitar redefiniendo el operador $\oplus$), pero facilita el enunciado de las propiedades de la definición.

Nuevamente, se supone la existencia de una función de cota $t : A \mapsto Int$ de iguales propiedades que en el caso de la recursión final.

Ejemplo 15.6
La función *fac* es recursiva lineal, puesto que puede expresarse de la siguiente manera:

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

Las funciones f, g y h que aparecen en la definición de función recursiva lineal son en este caso las siguientes:
$f.n = 1 \ \ \forall \ n$
$g.n = n - 1$
$h.n = n$
y el operador $\oplus$ es la multiplicación.

El teorema que sigue es la generalización de la derivación hecha en el ejemplo del cálculo del factorial a una función recursiva lineal arbitraria.

Teorema 15.1
Toda función recursiva lineal en la cual el operador $\oplus$ es asociativo y tiene neutro a izquierda se puede expresar como una función recursiva final.

Demostración
Sea F una función recursiva lineal y sea G especificada por

$$\langle \, \forall \, y, x \ :: \ G.y.x = y \oplus F.x \, \rangle \quad .$$

La función G es una generalización de F, pues $F.x = G.e.x$, donde e es el elemento neutro a izquierda del operador $\oplus$. Veamos que G es recursiva final realizando un análisis por casos:

Caso $(b.x)$

$$\begin{array}{ll} & G.y.x \\ = & \{ \text{ especificación de } G \ \} \\ & y \oplus F.x \\ = & \{ \ b.x \ \} \\ & y \oplus f.x \end{array}$$

Caso $(\neg b.x)$

$$\begin{array}{ll} & G.y.x \\ = & \{ \text{ especificación de } G \ \} \\ & y \oplus F.x \\ = & \{ \ \neg b.x \ \} \\ & y \oplus (h.x \oplus F.(g.x)) \\ = & \{ \ \oplus \text{ es asociativo } \} \\ & (y \oplus h.x) \oplus F.(g.x) \\ = & \{ \text{ hipótesis inductiva } \} \\ & G.(y \oplus h.x).(g.x) \end{array}$$

Por lo tanto la función G es recursiva final:

$$\begin{array}{lcl} G.y.x & \doteq & (\quad b.x \rightarrow y \oplus f.x \\ & & \Box \ \neg b.x \rightarrow G.(y \oplus h.x).(g.x) \\ & &) \end{array}$$

Ejemplo 15.7
Usando el teorema, obtenemos la siguiente función recursiva final para el cálculo de *fac*:

$$\begin{array}{lcl} G.y.n & \doteq & (\quad n = 0 \rightarrow y \\ & & \Box \ n \neq 0 \rightarrow G.(y * n).(n-1) \\ & &) \end{array}$$

y $fac.n$ se calcula a través de $G.1.n$, pues 1 es el neutro de la multiplicación. Notar que se trata de la misma función que se obtuvo en el ejemplo 15.2.

La asociatividad del operador $\oplus$ puede ser un requisito demasiado restrictivo. A continuación enunciamos un teorema que no requiere como hipótesis la asociatividad de $\oplus$.

Teorema 15.2
Sean

$$\left\lceil \begin{array}{c} f.0 \doteq c \\ f.(n+1) \doteq n \oplus f.n \end{array} \right. \qquad \left\lceil \begin{array}{c} G.z.m.0 \doteq z \\ G.z.m.(n+1) \doteq G.(m \oplus z).(m+1).n \end{array} \right.$$

entonces

$$G.(f.m).m.n = f.(m+n) \ .$$

Notar que la función G es recursiva final. El teorema dice que cualquier función definida con el esquema de f se puede expresar como una recursiva final; más aún, nos brinda la definición explícita de dicha función recursiva final.

Demostración
Hacemos inducción sobre n.

Caso base $(n = 0)$

$$\begin{array}{ll} & G.(f.m).m.0 \\ = & \{ \text{ definición de } G \ \} \\ & f.m \\ = & \{ \text{ álgebra } \} \\ & f.(m+0) \end{array}$$

Paso inductivo

$$\begin{array}{ll} & G.(f.m).m.(n+1) \\ = & \{ \text{ definición de } G \ \} \\ & G.(m \oplus f.m).(m+1).n \\ = & \{ \text{ definición de } f \ \} \\ & G.(f.(m+1)).(m+1).n \\ = & \{ \text{ hipótesis inductiva } \} \\ & f.((m+1)+n) \\ = & \{ \text{ álgebra } \} \\ & f.(m+(n+1)) \end{array}$$

Corolario 15.3
Con f y G definidas como en el teorema, $G.c.0.n = f.n$.

El teorema y su corolario resultan útiles para encontrar funciones recursivas finales que permiten calcular funciones definidas con el esquema de la función f del teorema. Lo único que se requiere es identificar la constante c y el operador $\oplus$.

Ejemplo 15.8
Consideremos nuevamente la función *fac*, definida por

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

Para escribirla como recursiva final usando el último teorema, debemos identificar "c" y "$\oplus$" en este caso. Claramente, $c = 1$. Con $\oplus$ hay que ser cuidadosos, pues el primer factor en la expresión de $fac.(n+1)$ no es n. Para responder al esquema de la función f del teorema debemos considerar $\oplus$ definido por $x \oplus y = (x+1) * y$. Entonces la función G debe ser

$$\boxed{\begin{array}{rcl} G.z.m.0 & \doteq & z \\ G.z.m.(n+1) & \doteq & G.((m+1)*z).(m+1).n \end{array}}$$

y el teorema asegura que

$$G.(fac.m).m.n = fac.(m+n)$$

Así, según el corolario, tenemos otra definición recursiva final para el cálculo de *fac*:

$$\boxed{fac.n \doteq G.1.0.n}$$

15.3 Recursión final para listas

Cuando se definen funciones sobre listas, el requisito de asociatividad del operador usado para definir la función es innecesariamente estricto. En esta sección presentamos un teorema para obtener definiciones recursivas finales a partir de definiciones recursivas lineales que no requiere que el operador sea asociativo. A cambio, es necesario introducir una nueva función en el enunciado del teorema. Se trata de la función *rev*, que invierte el orden de los elementos de una lista:

$$\boxed{\begin{array}{rcl} rev.[\,] & \doteq & [\,] \\ rev.(x \triangleright xs) & \doteq & (rev.xs) \triangleleft x \end{array}}$$

Antes de enunciar el teorema, estudiemos un ejemplo. La función *sum* que hemos usado en secciones anteriores es recursiva lineal:

$$\boxed{\begin{array}{rcl} sum.[\,] & \doteq & 0 \\ sum.(x \triangleright xs) & \doteq & x + sum.xs \end{array}}$$

Se desea una expresión recursiva final que permita calcular *sum*. Como ya hemos hecho antes, primero planteamos una generalización de la función y luego buscamos una definición recursiva final de la generalización. Sea

$$G.y.xs = y + sum.xs \ .$$

La función G generaliza a sum, pues $sum.xs = G.0.xs$. Busquemos ahora una expresión recursiva final para G.

Caso base $(xs = [\,])$

$$
\begin{array}{ll}
 & G.y.[\,] \\
= & \{ \text{ especificación de } G \ \} \\
 & y + sum.[\,] \\
= & \{ \text{ definición de } sum \ \} \\
 & y + 0 \\
= & \{ \text{ álgebra } \} \\
 & y
\end{array}
$$

Paso inductivo $(x \triangleright xs)$

$$
\begin{array}{ll}
 & G.y.(x \triangleright xs) \\
= & \{ \text{ especificación de } G \ \} \\
 & y + sum.(x \triangleright xs) \\
= & \{ \text{ definición de } sum \ \} \\
 & y + (x + sum.xs) \\
= & \{ \text{ asociatividad de } + \ \} \\
 & (y + x) + sum.xs \\
= & \{ \text{ hipótesis inductiva } \} \\
 & G.(y + x).xs
\end{array}
$$

Esto es,

$$
\boxed{G.y.xs} \doteq (\ xs = [\,] \rightarrowtail y \ \Box\ xs \neq [\,] \rightarrowtail G.(y + xs.0).(xs{\downarrow}1) \ )
$$

Para encontrar la definición recursiva final de G hemos usado la asociatividad del operador $+$. Supongamos ahora que queremos dar una definición recursiva final de sum sin usar la asociatividad del operador $+$. Notemos que al poner $G.y.xs = y + sum.xs$, en y vamos acumulando sumas parciales (interpretación operacional). Tratemos de expresar esto:

$$
\langle\, \forall\, xs, ys, y \ : y = sum.ys : \ G.y.xs = sum.(xs +\!\!+ ys) \,\rangle \quad .
$$

Trabajemos ahora a partir de esta especificación.

Caso base $(xs = [\,])$

$$
\begin{array}{ll}
 & G.y.[\,] \\
= & \{ \text{ especificación de } G \ \} \\
 & sum.([\,] +\!\!+ ys) \\
= & \{ \text{ propiedad de } +\!\!+ \ \} \\
 & sum.ys \\
= & \{ \ y = sum.ys \ \} \\
 & y
\end{array}
$$

Paso inductivo $(x \triangleright xs)$

$$
\begin{array}{ll}
 & G.y.(x \triangleright xs) \\
= & \{ \text{ especificación de } G \ \} \\
 & sum.((x \triangleright xs) +\!\!+ ys) \\
= & \{ \text{ lema (ejercicio siguiente) } \} \\
 & sum.(xs +\!\!+ (x \triangleright ys)) \\
= & \{ \text{ introducción de } y'\text{; hipótesis inductiva } \} \\
 & G.y'.xs \\
 & \quad [\![y' = sum.(x \triangleright ys)]\!] \\
= & \{ \text{ definición de } sum \ \} \\
 & G.y'.xs \\
 & \quad [\![y' = x + sum.ys]\!] \\
= & \{ \ y = sum.ys \ \} \\
 & G.y'.xs \\
 & \quad [\![y' = x + y]\!] \\
= & \{ \text{ eliminación de } y' \ \} \\
 & G.(x + y).xs
\end{array}
$$

Ejercicio: Demostrar $sum.((x \triangleright xs) +\!\!+ ys) \equiv sum.(xs +\!\!+ (x \triangleright ys))$.

Si bien hemos arribado a la misma definición de G que antes, la forma de obtener este resultado merece ser tenida en cuenta, pues no hace uso de la asociatividad del operador. A cambio, usa la conmutatividad del operador en el lema que se usa en la derivación. La conmutatividad del operador no es estrictamente necesaria, como se verá en el siguiente resultado.

El ejemplo que acabamos de analizar se puede generalizar mediante el siguiente

Teorema 15.4
Sean

$$
\left| \begin{array}{rcl} f.[\,] & = & c \\ f.(x \triangleright xs) & = & x \oplus f.xs \end{array} \right. \qquad \left| \begin{array}{rcl} G.z.[\,] & = & z \\ G.z.(x \triangleright xs) & = & G.(x \oplus z).xs \end{array} \right.
$$

entonces

$$G.(f.ys).xs = f.(rev.xs +\!\!+ ys) \ .$$

Demostración

Caso base $(xs = [\,])$

$$
\begin{array}{ll}
 & G.(f.ys).[\,] \\
= & \{ \text{ definición de } G \} \\
 & f.ys \\
= & \{ \text{ propiedad de } +\!\!+ \} \\
 & f.([\,] +\!\!+ ys) \\
= & \{ \text{ definición de } rev \} \\
 & f.(rev.[\,] +\!\!+ ys)
\end{array}
$$

Paso inductivo $(x \triangleright xs)$

$$
\begin{array}{ll}
 & G.(f.ys).(x \triangleright xs) \\
= & \{ \text{ definición de } G \} \\
 & G.(x \oplus f.ys).xs \\
= & \{ \text{ definición de } f \} \\
 & G.(f.(x \triangleright ys).xs) \\
= & \{ \text{ hipótesis inductiva } \} \\
 & f.(rev.xs +\!\!+ (x \triangleright ys)) \\
= & \{ \text{ propiedad de } +\!\!+ \} \\
 & f.((rev.xs \triangleleft x) +\!\!+ ys) \\
= & \{ \text{ definición de } rev \} \\
 & f.(rev.(x \triangleright xs) +\!\!+ ys)
\end{array}
$$

Observación: La función G del teorema anterior es recursiva final.

Corolario 15.5
Con f y G como en el teorema, $G.c.xs = f.(rev.xs)$

El teorema y su corolario son particularmente útiles cuando la función f que se desea calcular es independiente del orden de los elementos de la lista a la cual se aplica, pues en este caso $f.(rev.xs) = f.xs$, de donde resulta $f.xs = G.c.xs$, es decir, el teorema nos indica cómo obtener una definición recursiva final para el cálculo de f.

Ejemplo 15.9 (El problema del segmento de suma mínima)
Recordemos que este problema consiste en determinar la suma del segmento de suma mínima de una lista de enteros. En la sección 13.3 resolvimos este problema utilizando la técnica de tuplas, arribando finalmente a la siguiente definición recursiva:

$$
\begin{array}{|l}
\hline
h : [Num] \mapsto (Num, Num) \\
\hline
h.xs \;\doteq\; (\;\; xs = [\,] \to (0,0) \\
\qquad\quad \square\; xs \neq [\,] \to ((xs.0 + b) \text{ min } a,\; 0 \text{ min } (xs.0 + b)) \\
\qquad\qquad\qquad\qquad [\![(a,b) = h.(xs{\downarrow}1)]\!] \\
\qquad\quad) \\
\hline
\end{array}
$$

Usemos el teorema para dar una definición recursiva final para h. Como antes, lo primero es identificar "c" y "$\oplus$". De la definición de h surge $c = (0, 0)$. Por la forma que debe tener la función G del teorema, debemos definir $p \oplus (q, r) = ((p + r) \min q, 0 \min (p + r))$. Entonces

$$h.xs = G.(0, 0).xs$$

es decir, hemos encontrado una definición recursiva final para el problema del segmento de suma mínima.

15.4 Ejercicios

Ejercicio 15.1
Sea $P : [Int] \mapsto Bool$ la función que determina si en una lista de enteros hay algún elemento que es igual a la suma de todos los demás. Dar una definición recursiva final para P.

Ejercicio 15.2
Dar una definición recursiva final para el problema del cálculo de una aproximación de la constante matemática e (ver sección 12.2).

Ejercicio 15.3
Dar una definición recursiva final para el problema de la lista balanceada (ver sección 12.5).

Ejercicio 15.4
Dar una definición recursiva final para el problema de los paréntesis equilibrados (ver ejemplo 13.2).

Ejercicio 15.5
Una lista de enteros se denomina *esferulada* si cumple las siguientes dos condiciones:

(i) La suma de cualquier segmento inicial es no negativa.

(ii) La suma de sus elementos es 0.

Se pide:

1. Especificar $esf : [Int] \mapsto Bool$.
2. Derivar una función recursiva que resuelva esf.
3. Calcular una definición recursiva final para esf.

Capítulo 16

La programación imperativa

I fear those big words, Stephen said, which make us so unhappy.

James Joyce: *Ulysses*

A partir de ahora cambiaremos el modelo de computación subyacente, y por lo tanto el formalismo para expresar los programas. Esto no significa que todo lo hecho hasta ahora no nos sea útil en lo que sigue. La utilidad puede verse al menos en dos aspectos. El primero es que la experiencia adquirida en el cálculo de programas, y la consecuente habilidad para el manejo de fórmulas, seguirá siendo esencial en este caso, mostrando así que pese a sus diferencias la programación funcional y la imperativa tienen bastante en común. El segundo se verá en los capítulos finales y es un método para obtener programas imperativos a partir de programas funcionales (recursivos finales o de cola).

A diferencia de la primera parte, el material aquí presentado está cubierto en numerosos libros cuya consulta se aconseja a los lectores [Dij76, Gri81, DF88, Kal90, Coh90].

16.1 Estados y predicados

Los problemas que se presentan a un programador están, en general, expresados de manera informal, lo cual resulta demasiado impreciso a la hora de programar. Por eso es necesario *precisar* los problemas por medio de *especificaciones formales*. En la primera parte del curso se especificaba, usando lógica y un cálculo de funciones, el valor que debía calcularse. Este tipo de especificación era adecuado para el desarrollo de programas funcionales. En el estilo de programación imperativa, la computación no se expresa como cálculo de valores sino como modificación de estados. Para desarrollar programas imperativos es por lo tanto deseable expresar

las especificaciones como relaciones entre estados iniciales y finales. Así es que un problema a resolver estará especificado por un *espacio de estados*, una *precondición* y una *postcondición*. El espacio de estados es el universo de valores que pueden tomar las variables. La pre y postcondición son predicados que expresan el problema dado en términos precisos.

La notación que usaremos para expresar la especificación de un programa imperativo es

$$\{P\}\, S\, \{Q\}$$

donde P y Q son predicados y S es un programa (o sentencia, o instrucción, o comando). Las sentencias dirán cómo se modifican los estados. La terna $\{P\}\, S\, \{Q\}$ se denomina *terna de Hoare* (en inglés, Hoare triple) y se interpreta como sigue:

> *"Cada vez que se ejecuta S comenzando en un estado que satisface P, se termina en un estado que satisface Q."*

Derivar un programa consiste en encontrar S que satisfaga una especificación dada.

Ejemplo 16.1
Derivar un programa que satisfaga $\{P\}\, S\, \{x = y\}$ consiste en encontrar S tal que, aplicado a un estado que satisfaga P, termine en un estado que satisfaga $x = y$.

En los lenguajes imperativos es necesario explicitar el tipo de las variables y constantes usadas en los programas, lo cual se hace a través de una *declaración de variables y constantes*. Esto completa la especificación del programa a la vez que indica implícitamente las operaciones que se pueden realizar con las variables y constantes declaradas.

La declaración de variables y constantes debe figurar al comienzo del programa:

$$\begin{array}{l} \|[\text{var } x, y : Int \\ \ \text{con } X, Y : Int \\ \ \{X > 0 \wedge Y > 0 \wedge x = X \wedge y = Y\} \\ \ S \\ \ \{x = mcd.X.Y\} \\ ]\| \end{array}$$

Aquí x e y están declaradas como variables de tipo entero, mientras que X e Y son constantes de tipo entero (y por lo tanto no pueden ser modificadas por S). Esto implica que el espacio de estados será $Int \times Int$. El programa debe ser tal que comenzando con $x = X$ e $y = Y$, siendo $X, Y > 0$, al finalizar su ejecución la variable x contenga el máximo común divisor entre los números X e Y.

En los predicados pueden aparecer tres tipos de nombres: las *constantes* X, Y las cuales pueden pensarse como variables universalmente cuantificadas cuyo objetivo principal es proveer una forma de referirse en la postcondición a los valores

iniciales de la computación; las *variables de programa* x, y que no aparecen dentro del rango de ningún cuantificador y que decriben el estado del programa; por último, usaremos también las *variables de cuantificación* (dummies) como ya veníamos haciendo para el caso de los programas funcionales.

Antes de continuar con el desarrollo de programas, veremos algunas reglas para la correcta interpretación y utilización de ternas de Hoare. Para indicar que un predicado es válido para cualquier estado, lo encerraremos entre corchetes [ ].

La primera regla, conocida como *ley de exclusión de milagros*, expresa

$$\boxed{\{P\}\, S\, \{False\} \;\equiv\; [P \equiv False]}$$

Esta regla dice que para cualquier programa S si se requiere que termine y que los estados finales satisfagan $False$, entonces ese programa no puede ejecutarse exitosamente para ningún posible estado inicial. O bien, leído de otra manera, no existe ningún estado tal que si se ejecuta un programa S comenzando en él se puede terminar en un estado que satisfaga $False$.

La expresión

$$\{P\}\, S\, \{True\}$$

indica que, cada vez que comienza en un estado que satisface P, la ejecución de S concluye en un estado que satisface el predicado $True$. Esto se interpreta diciendo que la ejecución de S *termina* cada vez que se comienza en un estado que satisface P.

Las reglas que siguen expresan el hecho que la precondición puede "reforzarse", mientras que la postcondición puede "debilitarse":

$$\boxed{\{P\}\, S\, \{Q\} \wedge [P_0 \Rightarrow P] \;\Rightarrow\; \{P_0\}\, S\, \{Q\}}$$

$$\boxed{\{P\}\, S\, \{Q\} \wedge [Q \Rightarrow Q_0] \;\Rightarrow\; \{P\}\, S\, \{Q_0\}}$$

La primera regla dice que si P_0 es más fuerte[1] que P y vale $\{P\}\, S\, \{Q\}$, entonces $\{P_0\}\, S\, \{Q\}$ también es válido.

La segunda regla dice que si Q_0 es más débil que Q y vale $\{P\}\, S\, \{Q\}$, entonces también vale $\{P\}\, S\, \{Q_0\}$.

Las dos reglas que siguen establecen equivalencias para los casos en que los estados inicial/final contienen una conjunción o una disyunción.

Si las ternas $\{P\}\, S\, \{Q\}$ y $\{P\}\, S\, \{R\}$ son válidas, entonces ejecutar S comenzando en un estado que satisface P, termina en un estado que satisface Q y también

[1] Se dice que P_0 es *más fuerte* que P sí y solo si $P_0 \Rightarrow P$.
Recíprocamente, se dice que P_0 es *más débil* que P sí y solo si $P \Rightarrow P_0$

R, es decir, satisface $Q \wedge R$. Recíprocamente, si la terna $\{P\}\, S\, \{Q \wedge R\}$ es válida, entonces ejecutar S comenzando en un estado que satisface P, termina en un estado que satisface $Q \wedge R$, es decir, satisface tanto a Q como a R.

$$\boxed{\{P\}\, S\, \{Q\} \wedge \{P\}\, S\, \{R\} \;\equiv\; \{P\}\, S\, \{Q \wedge R\}}$$

Si las ternas $\{P\}\, S\, \{Q\}$ y $\{R\}\, S\, \{Q\}$ son válidas, entonces ejecutar S comenzando en un estado que satisface o bien P o bien R, es decir, $P \vee R$, termina en un estado que satisface Q, por otro lado, si cada vez que se comienza con un estado que satisface $P \vee R$ al terminar vale Q, entonces a fortiori (fortalecimiento de la condición) vale que comenzando en un estado que satisface P (o R) al terminar valdrá Q.

$$\boxed{\{P\}\, S\, \{Q\} \wedge \{R\}\, S\, \{Q\} \;\equiv\; \{P \vee R\}\, S\, \{Q\}}$$

16.2 El transformador de predicados wp

Para cada comando S se puede definir la función

$$wp.S : Predicados \mapsto Predicados$$

tal que si Q es un predicado, $wp.S.Q$ representa el predicado más débil P para el cual vale $\{P\}\, S\, \{Q\}$. Para cada Q, $wp.S.Q$ se denomina *precondición más débil* de S con respecto a Q (en inglés, weakest precondition). En otras palabras, para todo predicado Q

$$[wp.S.Q = P] \quad \Longleftrightarrow \quad \begin{cases} (i) & \{P\}\, S\, \{Q\} \\ (ii) & \{P_0\}\, S\, \{Q\} \;\Rightarrow\; [P_0 \Rightarrow P] \end{cases}$$

La definición de $wp.S$ permite relacionar las expresiones $\{P\}\, S\, \{Q\}$ y $wp.S.Q$:

$$\boxed{\{P\}\, S\, \{Q\} \;\equiv\; [P \Rightarrow wp.S.Q]}$$

De esta manera, podemos decir que un programa es correcto con respecto a su especificación si $[P \Rightarrow wp.S.Q]$, lo cual nos permitirá, una vez definido wp para todos los posibles comandos, trabajar en el conocido campo del cálculo de predicados.

Observemos que $\{wp.S.Q\}\, S\, \{Q\}$ es correcto, es decir, $wp.S.Q$ siempre es una precondición admisible para un programa S con postcondición Q.

Las reglas enunciadas en la sección anterior se deducen a partir de las siguientes propiedades de $wp.S.Q$:

- $[wp.S.\, False \equiv False]$

- $[wp.S.Q \wedge wp.S.R \ \equiv \ wp.S.(Q \wedge R)]$
- $[wp.S.Q \vee wp.S.R \ \Rightarrow \ wp.S.(Q \vee R)]$

Observar que las propiedades de la disyunción involucran solo una implicación, no una equivalencia. Esto nos dice que la precondición más débil de S con respecto al predicado $Q \vee R$ no es suficiente para asegurar que se cumple al menos una de las dos precondiciones más débiles, $wp.S.Q$ o $wp.S.R$. Este fenómeno se debe a que aceptaremos programas no-determinísticos, esto es, donde puede haber más de un resultado. Por ejemplo, si consideramos el "programa" S que tira una moneda al aire y tomamos $Q \equiv$ *sale cara* y $R \equiv$ *sale ceca*, obviamente $wp.S.(Q \vee R) \equiv True$, pero $wp.S.Q \equiv False$ (no hay ningún estado que asegure que al tirar la moneda saldrá cara) y $wp.S.R \equiv False$ (análogamente).

16.3 Ejercicios

Ejercicio 16.1
Dadas las siguientes propiedades del transformador de predicados wp,

Exclusión de milagros:	$[wp.S.false \equiv false]$
Conjuntividad:	$[wp.S.(P \wedge Q) \equiv wp.S.P \wedge wp.S.Q]$

1. Demostrar monotonía:

$$[(P \Rightarrow Q) \ \Rightarrow \ (wp.S.P \Rightarrow wp.S.Q)] \ .$$

2. Usando la monotonía de wp, demostrar:

$$\{Q\}\, S\, \{A\} \wedge (A \Rightarrow R) \Rightarrow \{Q\}\, S\, \{R\} \ .$$

Ejercicio 16.2
Usando las propiedades de $wp.S.Q$, demostrar las reglas de la sección 16.1.

Capítulo 17

Definición de un lenguaje de programación imperativo

> And while I thus spoke, did there not cross your mind some thought of the *physical power of words?* Is not every word an impulse on the air?
> Edgar Allan Poe: *The power of words*

En esta sección veremos una serie de comandos que definen un lenguaje de programación imperativo básico. Para cada uno de ellos se dará la respectiva precondición más débil y se mostrarán ejemplos de su funcionamiento.

17.1 Skip

La sentencia *skip* no tiene ningún efecto sobre el estado válido antes de su ejecución (no lo modifica). La utilidad de una instrucción que "no hace nada" se entenderá más adelante, pero podemos adelantar que será tan útil como lo es el 0 en el álgebra.

Queremos caracterizar la sentencia *skip* usando ternas de Hoare. Comencemos estudiando $wp.skip.Q$. Claramente, el predicado más débil P tal que $\{P\}\ skip\ \{Q\}$ es Q, pues lo mínimo necesario para asegurar el estado final Q cuando "no se hace nada" es tener Q como estado inicial. Por lo tanto, la precondición más débil del comando *skip* con respecto a un predicado es dicho predicado:

$$\boxed{[wp.skip.Q \equiv Q]}$$

Esto podría inducirnos a caracterizar *skip* usando la terna $\{Q\}\, skip\, \{Q\}$ como axioma, pero como ya hemos visto que es posible reforzar la precondición, daremos una caracterización más general en terminos de una equivalencia:

$$\boxed{\{P\}\, skip\, \{Q\} \;\equiv\; [P \Rightarrow Q]}$$

Ejemplo 17.1
El programa
$[\![$var $x : Int$
$\{x \geq 1\}$
$skip$
$\{x \geq 0\}$
$]\!]$
es correcto, puesto que $[x \geq 1 \Rightarrow x \geq 0]$.

17.2 Abort

La instrucción *abort* está especificada por

$$\boxed{\{P\}\, abort\, \{Q\} \;\equiv\; [P \equiv False]}$$

El programa *abort* representa un programa erróneo. La semántica dice que no existe ningún estado en el cual *abort* pueda ejecutarse y terminar.

En términos de la precondición más débil, tenemos

$$\boxed{[wp.abort.Q \equiv False]}$$

17.3 Asignación

La asignación se usa, como su nombre lo indica, para asignar valores a variables. La sintaxis es

$$x := E$$

donde x es una lista de variables separadas por comas y E es una lista de expresiones de la misma longitud, también separadas por comas, tal que la $n-$ésima expresión es del mismo tipo que la $n-$ésima variable de x. El efecto de la sentencia $x := E$ es reemplazar el valor de x por el de E. Ejemplos de asignaciones son:

$$x := x + 1 \qquad x, y := x - 2, x + y \qquad y, p := y/2, True$$

Para analizar el significado de $\{P\}\, x := E\, \{Q\}$ estudiemos en primer lugar $wp.(x := E).Q$. Lo mínimo que se requiere para que el predicado Q sea válido

luego de efectuar la asignación $x := E$, es el predicado $Q(x := E)$. En otras palabras, la precondición más débil de una asignación con respecto a un predicado es

$$\boxed{[wp.(x := E).Q \equiv Q(x := E)]}$$

Entonces, para que la terna $\{P\}\, x := E\, \{Q\}$ sea correcta, el predicado P debe ser más fuerte que el predicado $Q(x := E)$. De esta manera, la asignación está especificada por

$$\boxed{\{P\}\, x := E\, \{Q\} \;\equiv\; [P \Rightarrow Q(x := E)]}$$

Asumiremos que en E solo se admitirán expresiones bien definidas, sino habría que agregar el predicado $def.E$ (que es $True$ solo si la expresión E está bien definida) en el segundo miembro de la implicación:

$$\{P\}\, x := E\, \{Q\} \;\equiv\; [P \Rightarrow def.E \wedge Q(x := E)] \;.$$

Nótese que la expresión "$x := E$" del miembro izquierdo es una sentencia válida de nuestro lenguaje de programación, mientras que la del miembro derecho, en la sustitución $Q(x := E)$, es una operación sintáctica que se aplica a expresiones lógicas, por lo tanto no son iguales.

Ejemplo 17.2
Encontrar E tal que se satisfaga $\{True\}\, x, y := x + 1, E\, \{y = x + 1\}$

$$\begin{array}{ll}
 & [True \Rightarrow (y = x + 1)(x, y := x + 1, E)] \qquad \text{(esto es } [P \Rightarrow Q(x := E)]) \\
\equiv & \{\ (True \Rightarrow R) \equiv R\ \} \\
 & [(y = x + 1)(x, y := x + 1, E)] \\
\equiv & \{\ \text{sustitución en predicados}\ \} \\
 & [E = x + 1 + 1] \\
\equiv & \{\ \text{aritmética}\ \} \\
 & [E = x + 2]
\end{array}$$

Es decir, cualquier expresión E que valga lo mismo que $x+2$ es válida; en particular, podemos tomar $E = x + 2$.

Notemos que la definición de $wp.(x := E).Q$ nos permite escribir

$$\{P\}\, x := E\, \{Q\} \;\equiv\; [P \Rightarrow wp.(x := E).Q]$$

De aquí que una forma práctica de derivar las expresiones que se deben usar en las asignaciones es comenzar la derivación analizando $wp.(x := E).Q$ y usar en algún momento de la misma la validez de P.

Ejemplo 17.3
Desarrollar un programa que satisfaga

$$\{q = a * c \wedge w = c^2\}\, a, q := a + c, E\, \{q = a * c\} \quad .$$

Derivemos E a partir de la precondición más débil:

$$\begin{aligned}
& wp.(a, q := a + c, E).(q = a * c) \\
\equiv\ & \{\ \text{definición de } wp\ \} \\
& (q = a * c)(a, q := a + c, E) \\
\equiv\ & \{\ \text{sustitución en predicados}\ \} \\
& E = (a + c) * c \\
\equiv\ & \{\ \text{aritmética}\ \} \\
& E = a * c + c^2 \\
\equiv\ & \{\ \text{precondición (aquí se usa la validez de } P)\ \} \\
& E = q + w
\end{aligned}$$

A veces, programas que son aparentemente diferentes (por su sintaxis) son en realidad equivalentes en cuanto a los cambios de estados que producen. Esto motiva la siguiente definición:

Definición 17.1 (Equivalencia de programas)
Dos programas S y T se dicen equivalentes si solo si $\langle\, \forall\, Q\ ::\ wp.S.Q \equiv wp.T.Q\, \rangle$. Lo denotaremos $S = T$.

Ejercicio 17.1
Usando la definición de equivalencia de programas, demostrar:$(x := x) = skip$.

17.4 Concatenación o composición

La concatenación permite describir secuencias de acciones. La ejecución de dos sentencias S y T, una a continuación de la otra, se indicará separando las mismas con punto y coma:

$$S; T$$

siendo la de la izquierda la primera que se ejecuta.

La precondición más débil de la concatenación $S; T$ con respecto a un predicado Q se obtiene tomando primero la precondición más débil de T con respecto a Q $(wp.T.Q)$ y luego la precondición más débil de S con respecto a este último predicado:

$$\boxed{[wp.S; T.Q \ \equiv\ wp.S.(wp.T.Q)]}$$

Para demostrar $\{P\}\, S;T\, \{Q\}$ hay que encontrar un predicado "intermedio" R que sirva como postcondición de S y como precondición de T:

$$\boxed{\{P\}\, S;T\, \{Q\} \;\equiv \text{existe } R \text{ tal que } \{P\}\, S\, \{R\} \wedge \{R\}\, T\, \{Q\}}$$

Ejemplo 17.4
El programa

$[\![\text{var } x, y : Int$
$\{x > y\}$
$y := y + 1; x := x + 1$
$\{x > y\}$
$]\!]$

es correcto, puesto que $\{R :\ x \geq y\}$ es un predicado intermedio válido.

Ejercicio 17.2
Usando la definición de equivalencia de programas, demostrar:

(a) $S; skip = S$ y $skip; S = S$ (es decir, $skip$ es el elemento neutro de la concatenación).

(b) $S; abort = abort$ y $abort; S = abort$

17.5 Alternativa

Los comandos vistos hasta el momento solo permiten derivar programas muy simples. Un comando que amplía notablemente las posibilidades de desarrollar programas es el que permite una alternativa. La sintaxis es

$$\begin{array}{ll} \underline{\textbf{if}} & B_0 \to S_0 \\ \Box & B_1 \to S_1 \\ \vdots & \\ \Box & B_n \to S_n \\ \underline{\textbf{fi}} & \end{array}$$

donde para todo $0 \leq i \leq n$, B_i es una expresión booleana y S_i es una instrucción. Las expresiones B_i se llaman **guardas** o protecciones (en inglés, guards) y $B_i \to S_i$ se denominan comandos resguardados o protegidos (en inglés, guarded commands). Notar la analogía con lo expuesto en la sección 7.5.

La alternativa funciona de la siguiente manera: todas las guardas son evaluadas; si ninguna es $True$, se produce un $abort$; en caso contrario, se elige (de alguna manera) una guarda B_i que sea $True$ y se ejecuta el correspondiente S_i.

Ya hemos dicho que *abort* no es una sentencia ejecutable; por lo tanto, un **if** con ninguna guarda verdadera será considerado un error.

La especificación de la alternativa es la siguiente:

$$\begin{array}{lllll} \{P\} & \underline{\textbf{if}} & B_0 \to S_0 & \{Q\} & \equiv \quad [P \Rightarrow (B_0 \vee B_1 \vee \ldots \vee B_n)] \\ & \square & B_1 \to S_1 & & \quad \wedge \{P \wedge B_0\}\, S_0\, \{Q\} \\ & \vdots & & & \quad \wedge \{P \wedge B_1\}\, S_1\, \{Q\} \\ & \square & B_n \to S_n & & \quad \vdots \\ & \underline{\textbf{fi}} & & & \quad \wedge \{P \wedge B_n\}\, S_n\, \{Q\} \end{array}$$

La implicación $P \Rightarrow (B_0 \vee B_1 \vee \ldots \vee B_n)$ es necesaria para asegurarse que el programa no es erróneo (contiene un *abort*), mientras que $\{P \wedge B_i\}\, S_i\, \{Q\}$ asegura que ejecutar S_i cuando se satisface $P \wedge B_i$ conduce a un estado que satisface Q, es decir, cualquiera que sea la guarda verdadera, ejecutar el respectivo comando conduce al estado final deseado.

Asumiremos que las expresiones B_i están todas bien definidas, de otra manera habría que agregar esta condición en el segundo miembro:

$$[P \Rightarrow def.B_0 \wedge def.B_1 \wedge \ldots \wedge def.B_n \wedge (B_0 \vee B_1 \vee \ldots \vee B_n)] \ .$$

En cuanto a la precondición más débil de la alternativa con respecto a un predicado Q, notemos que lo mínimo requerido es que alguna de las guardas B_i sea verdadera y que cada una de ellas sea más fuerte que la precondición más débil del respectivo S_i con respecto a Q:

$$\boxed{wp.if.Q \equiv [(B_0 \vee B_1 \vee \ldots \vee B_n) \wedge (B_0 \Rightarrow wp.S_0.Q) \wedge \ldots \wedge (B_n \Rightarrow wp.S_n.Q)]}$$

Así, al derivar un **if** con precondición P y postcondición Q hay que probar:

(i) $[P \Rightarrow (B_0 \vee B_1 \vee \ldots \vee B_n)]$

(ii) $\{P \wedge B_i\}\, S_i\, \{Q\}$ o, equivalentemente, $[P \wedge B_i \Rightarrow wp.S_i.Q]$

Ejemplo 17.5
Determinar S tal que:

$$\{P:\ x = X \wedge y = Y\}\, S\, \{Q:\ (x = X \vee x = Y) \wedge x \geq X \wedge x \geq Y\} \ ,$$

donde X e Y son constantes.

Teniendo en cuenta la precondición, para lograr $(x = X \vee x = Y) \equiv True$, podemos hacer dos cosas: *skip* o $x := y$. Veamos lo que necesitamos para poder ejecutar la segunda:

$$
\begin{array}{ll}
 & wp.(x := y).((x = X \vee x = Y) \wedge x \geq X \wedge x \geq Y) \\
\equiv & \{ \text{ definición de } wp \ \} \\
 & ((x = X \vee x = Y) \wedge x \geq X \wedge x \geq Y)(x := y) \\
\equiv & \{ \text{ sustitución } \} \\
 & (y = X \vee y = Y) \wedge y \geq X \wedge y \geq Y \\
\equiv & \{ \ y = Y \text{ por precondición } \} \\
 & True \wedge y \geq X \wedge True \\
\equiv & \{ \ True \text{ elemento neutro de } \wedge \ \} \\
 & y \geq X \\
\equiv & \{ \ x = X \text{ por precondición } \} \\
 & y \geq x
\end{array}
$$

Esto nos dice que en caso de valer $y \geq x$, ejecutar $x := y$ conduce a un estado que satisface la postcondición. En otras palabras, $y \geq x$ es candidato a ser una guarda, bajo la cual el comando a ejecutar es $x := y$. Es importante notar que no podemos quedarnos con $y \geq X$ como guarda, puesto que X no es variable de programa y por lo tanto no puede figurar en una guarda.

Veamos ahora los requisitos para ejecutar *skip*:

$$
\begin{array}{ll}
 & wp.skip.((x = X \vee x = Y) \wedge x \geq X \wedge x \geq Y) \\
\equiv & \{ \text{ definición de } wp \ \} \\
 & (x = X \vee x = Y) \wedge x \geq X \wedge x \geq Y \\
\equiv & \{ \ x = X \text{ por precondición } \} \\
 & True \wedge True \wedge x \geq Y \\
\equiv & \{ \ True \text{ elemento neutro de } \wedge \ \} \\
 & x \geq Y \\
\equiv & \{ \ y = Y \text{ por precondición } \} \\
 & x \geq y
\end{array}
$$

Esto nos dice que en caso de valer $x \geq y$, ejecutar *skip* conduce a un estado que satisface la postcondición. En otras palabras, $x \geq y$ es candidato a ser una guarda, bajo la cual el comando a ejecutar es *skip*.

Hasta aquí hemos demostrado

$$[P \wedge (y \geq x) \Rightarrow wp.(x := y).Q] \quad \text{y} \quad [P \wedge (x \geq y) \Rightarrow wp.skip.Q] \ ,$$

lo que nos conduce a proponer el siguiente programa:

$$\begin{array}{ll} \underline{\textbf{if}} & y \geq x \rightarrow x := y \\ \square & x \geq y \rightarrow \mathit{skip} \\ \underline{\textbf{fi}} & \end{array}$$

Todavía falta probar $[P \Rightarrow (B_0 \vee B_1)]$. Pero $(y \geq x \vee x \geq y) \equiv True$, por lo tanto esta implicación también es cierta para cualquier P, con lo cual queda demostrado que el programa de arriba es correcto para la especificación dada.

Ejercicio 17.3
Demostrar que **if** $False \rightarrow S$ **fi** es equivalente a *abort*.

17.6 Repetición

La lista de comandos se completa con una sentencia que permitirá la ejecución repetida de una acción, lo que recibirá el nombre de ciclo. La sintaxis es

$$\begin{array}{ll} \underline{\textbf{do}} & B_0 \rightarrow S_0 \\ \square & B_1 \rightarrow S_1 \\ \vdots & \\ \square & B_n \rightarrow S_n \\ \underline{\textbf{od}} & \end{array}$$

donde para todo $0 \leq i \leq n$, B_i es una expresión booleana también llamada guarda y S_i es una instrucción.

El ciclo funciona de la siguiente manera: mientras haya guardas equivalentes a $True$, se elige (de alguna manera) una de ellas y se ejecuta la instrucción correspondiente; luego vuelven a evaluarse las guardas. Esto se repite hasta que ninguna guarda sea $True$. Si desde un principio ninguna guarda es $True$, el ciclo equivale a un *skip*.

La especificación de la repetición es

$$\begin{array}{lllll} \{P\} & \underline{\textbf{do}} \quad B_0 \rightarrow S_0 & \{Q\} & \equiv & [P \wedge \neg B_0 \wedge \neg B_1 \wedge \ldots \wedge \neg B_n \Rightarrow Q] \\ & \square \quad B_1 \rightarrow S_1 & & & \wedge \{P \wedge B_0\}\, S_0\, \{P\} \\ & \vdots & & & \wedge \{P \wedge B_1\}\, S_1\, \{P\} \\ & \square \quad B_n \rightarrow S_n & & & \vdots \\ & \underline{\textbf{od}} & & & \wedge \{P \wedge B_n\}\, S_n\, \{P\} \\ & & & & \wedge \text{ el ciclo termina} \end{array}$$

La implicación $P \wedge \neg B_0 \wedge \neg B_1 \wedge \ldots \wedge \neg B_n \Rightarrow Q$ asegura el cumplimiento de la postcondición cuando ninguna guarda es verdadera, es decir, cuando el ciclo termina, mientras que $\{P \wedge B_i\}\, S_i\, \{P\}$ asegura que ejecutar S_i cuando se satisface $P \wedge B_i$ conduce a un estado que satisface P, lo cual es necesario para que tenga sentido volver a evaluar las guardas. La condición adicional "el ciclo termina" es necesaria porque de otra manera el programa podría ejecutarse indefinidamente, lo cual es ciertamente indeseable. Más adelante formalizaremos esta condición.

Asumiremos que las expresiones B_i están todas bien definidas, de otra manera habría que agregar esta condición en el segundo miembro.

Un predicado P que satisface $\{P \wedge B_i\}\, S_i\, \{P\}$ recibe el nombre de **invariante**. La mayor dificultad de la programación imperativa consiste en la determinación de invariantes (notemos que esto es de vital importancia para el armado del ciclo). Hay varias técnicas para determinar invariantes, que analizaremos más adelante. Por ahora, intentaremos comprender el funcionamiento de los ciclos suponiendo que ya se ha determinado el invariante.

Ejemplo 17.6 (Máximo Común Divisor)
Desarrollar un algoritmo para calcular el máximo común divisor entre dos enteros positivos.

El programa tendrá la siguiente especificación:

$$
\begin{array}{l}
[\![\text{var } x, y : Int \\
\ \text{con } X, Y : Int \\
\ \{X > 0 \wedge Y > 0 \wedge x = X \wedge y = Y\} \\
\ S \\
\ \{x = mcd.X.Y\} \\
]\!]
\end{array}
$$

Para derivar S, recordemos las propiedades del mcd:

(1) $mcd.x.x = x$
(2) $mcd.x.y = mcd.y.x$
(3) $x > y \ \Rightarrow\ mcd.x.y = mcd.(x - y).y$
$y > x \ \Rightarrow\ mcd.x.y = mcd.x.(y - x)$

Postulamos el siguiente invariante: $\{P :\ x > 0 \wedge y > 0 \wedge mcd.x.y = mcd.X.Y\}$ (notemos que la precondición implica P, es decir, P es un predicado que efectivamente vale al comienzo de S).
Derivemos ahora el ciclo, usando la propiedad (3):

$$
\begin{array}{ll}
 & P \wedge x > y \\
\equiv & \{ \text{ reemplazando } P \ \} \\
 & x > 0 \wedge y > 0 \wedge mcd.x.y = mcd.X.Y \wedge x > y
\end{array}
$$

$\equiv$ { propiedad (3) }

$x > 0 \wedge y > 0 \wedge mcd.(x - y).y = mcd.X.Y \wedge x > y$ x

$\equiv$ { álgebra }

$x - y > 0 \wedge y > 0 \wedge mcd.(x - y).y = mcd.X.Y$

$\equiv$ { sustitución en predicados }

$P(x := x - y)$

Hemos demostrado que eligiendo como guarda $x > y$ y como instrucción $x := x-y$, luego de ejecutar la acción sigue valiendo P, es decir, $\{P \wedge x > y\}\, x := x - y\, \{P\}$ es correcto. Análogamente, partiendo de $P \wedge y > x$ se arriba a $P(y := y - x)$. Así, obtenemos el ciclo

$$
\begin{array}{l}
[\![\text{var } x, y : Int \\
\ \text{con } X, Y : Int \\
\ \{X > 0 \wedge Y > 0 \wedge x = X \wedge y = Y\} \\
\ \ \underline{\textbf{do}} \quad x > y \rightarrow x := x - y \\
\ \ \square \quad\ \ y > x \rightarrow y := y - x \\
\ \ \underline{\textbf{od}} \\
\ \{x = mcd.X.Y\} \\
]\!]
\end{array}
$$

Observemos que si $x = y$ el ciclo equivale a un *skip*, lo cual es correcto por propiedad (1) del *mcd*. En otras palabras,

$$(P \wedge \neg(x > y) \wedge \neg(y > x)) \ \Rightarrow\ \{x = mcd.X.Y\}\ .$$

Hasta este punto, hemos demostrado lo que se conoce como **corrección parcial** del ciclo. Esto es: suponiendo que el ciclo termina, demostrar

(i) $(P \wedge \neg B_0 \wedge \neg B_1 \wedge \ldots \wedge \neg B_n) \ \Rightarrow\ Q$

(ii) $\{P \wedge B_i\}\, S_i\, \{P\} \quad \forall\ 0 \leq i \leq n$

Ahora bien, aún no podemos asegurar que el programa derivado sea completamente correcto, pues no hemos demostrado que el ciclo termina. Esta demostración se conoce como prueba de **corrección total** y consiste en encontrar una función $t : Estados \mapsto Int$, acotada inferiormente y con la propiedad de decrecer en cada paso de la repetición. Esta función recibe el nombre de **función variante** o **función de cota**. De aquí en adelante frecuentemente la llamaremos simplemente **cota**.

Para demostrar que el ciclo termina, hay que encontrar alguna función $t : Estados \mapsto Int$ tal que:

(i) $[P \wedge (B_0 \vee B_1 \vee \ldots \vee B_n) \ \Rightarrow\ t \geq 0]$

(ii) $\{P \wedge B_i \wedge t = T\}\, S_i\, \{t < T\} \quad \forall\, 0 \leq i \leq n$

La condición (i) asegura que mientras alguna guarda es verdadera, la función de cota es no negativa; o bien, leído al revés, si la función de cota se hace negativa, entonces ninguna guarda es verdadera (puesto que P siempre es verdadero):

$$P \wedge t < 0 \;\Rightarrow\; \neg(B_0 \vee B_1 \vee \ldots \vee B_n) \;\; .$$

Por otra parte, la condición (ii) asegura el decrecimiento de la función cota con la ejecución de cualquier S_i. De esta manera, en algún momento la cota será negativa y por lo explicado más arriba queda claro que el ciclo habrá terminado.

En el ejemplo precedente, una elección posible para la función variante es $t = max.x.y$. Otra cota posible es $t = |x - y|$. Reiteramos que basta con encontrar *una* función de cota; damos aquí dos funciones diferentes solo a modo de ejemplo.

No hemos mencionado aún la precondición más débil de la repetición. La razón es que su expresión es complicada y no resulta de utilidad para la derivación de ciclos. Lo que se usa es el denominado *teorema de invariancia*, cuyo enunciado resume lo que hicimos en el ejemplo precedente. El teorema de invariancia asegura que si probamos corrección parcial y corrección total, el ciclo está bien diseñado.

Teorema 17.2 (Teorema de invariancia)

$$\begin{array}{lll} \{P\} \;\; \underline{\textbf{do}} \;\; B_0 \to S_0 \;\; \{Q\} & \equiv & [P \wedge \neg B_0 \wedge \neg B_1 \wedge \ldots \wedge \neg B_n \Rightarrow Q] \\ \quad \square \;\; B_1 \to S_1 & & \quad \wedge \;\; \{P \wedge B_i\}\, S_i\, \{P\} \quad \forall\, 0 \leq i \leq n \\ \quad \vdots & & \quad \wedge \; Existe\ t : Estados \mapsto Int\ tal\ que \\ \quad \square \;\; B_n \to S_n & & \quad\;\; (i)\;\; [P \wedge (B_0 \vee B_1 \vee \ldots \vee B_n) \;\Rightarrow\; t \geq 0] \\ \quad \underline{\textbf{od}} & & \quad\;\; (ii)\;\; \{P \wedge B_i \wedge t = T\}\, S_i\, \{t < T\} \\ & & \qquad\;\; \forall\, 0 \leq i \leq n \end{array}$$

Ejercicio 17.4
Demostrar que todo ciclo se puede reescribir como un ciclo con una sola guarda:

$$\begin{array}{lll} \underline{\textbf{do}} & B_0 \to S_0 & \equiv \\ \square & B_1 \to S_1 & \\ \underline{\textbf{od}} & & \end{array} \qquad \begin{array}{ll} \underline{\textbf{do}} & B_0 \vee B_1 \to \\ & \underline{\textbf{if}}\ B_0 \to S_0 \\ & \square\ B_1 \to S_1 \\ & \underline{\textbf{fi}} \\ \underline{\textbf{od}} & \end{array}$$

El enunciado del ejercicio resulta útil, pues no todos los lenguajes de programación admiten ciclos con varias guardas.

17.7 Ejercicios

Ejercicio 17.5
En cada uno de los siguientes casos, determinar la precondición más débil wp para que el programa sea correcto. Suponer que las variables $x,\ y,\ z,\ q,\ r$ son de tipo Int, las variables $i,\ j$ son de tipo Nat y las variables $a,\ b$ son de tipo $Bool$.

1. $\{wp\}\, x := 8\, \{x = 8\}$
2. $\{wp\}\, x := 8\, \{x \neq 8\}$
3. $\{wp\}\, x := 8\, \{x = 7\}$
4. $\{wp\}\, x := x + 2;\ y := y - 2\, \{x + y = 0\}$
5. $\{wp\}\, x := x + 1;\ y := y - 1\, \{x * y = 0\}$
6. $\{wp\}\, x := x + 1;\ y := y - 1\, \{x + y + 10 = 0\}$
7. $\{wp\}\, z := z * y;\ x := x - 1\, \{z * y^x = c\}$
8. $\{wp\}\, x, z, y := 1, c, d\, \{z * x^y = c^d\}$
9. $\{wp\}\, i, j := i + i, j;\ j := j + i\, \{i = j\}$
10. $\{wp\}\, x := (x - y) * (x + y)\, \{x + y^2 = 0\}$
11. $\{wp\}\, q, r := q + 1, r - y\, \{q * y + r = x\}$
12. $\{wp\}\, a := a \equiv b;\ b := a \equiv b;\ a := a \equiv b\, \{(a \equiv B) \wedge (b \equiv A)\}$

Ejercicio 17.6
[vdS93] En cada uno de los siguientes casos, determinar el predicado Q más fuerte para que el programa sea correcto. Suponer que las variables $x,\ y$ son de tipo Int.

1. $\{x = 10\}\, x := x + 1\, \{Q\}$
2. $\{x^2 > 45\}\, x := x + 1\, \{Q\}$
3. $\{x \geq 10\}\, x := x - 10\, \{Q\}$
4. $\{0 \leq x < 10\}\, x := x^2\, \{Q\}$
5. $\{x^3 = y\}\, x := |x|\, \{Q\}$

Ejercicio 17.7
Calcular expresiones E tales que:

1. $\{A = q * B + r\}\, q := E;\ r := r - B\, \{A = q * B + r\}$

2. $\{x * y + p * q = N\}\, x := x - p;\ q := E\, \{x * y + p * q = N\}$

Ejercicio 17.8
Demostrar: $\{x = A \wedge y = B\}\, x := x - y;\ y := x + y;\ x := y - x\, \{x = B \wedge y = A\}$

Ejercicio 17.9
Demostrar que los siguientes programas son correctos. En todos los casos $x,\ y : Int$ y $a,\ b : Bool$.

1. $\{True\}$
   if $x \geq 1 \rightarrow x := x + 1$
   $\square\ x \leq 1 \rightarrow x := x - 1$
   fi
   $\{x \neq 1\}$

2. $\{True\}$
   if $x \geq y \rightarrow skip$
   $\square\ x \leq y \rightarrow x, y := y, x$
   fi
   $\{x \geq y\}$

3. $\{True\}$
   $x, y := y * y, x * x$
   if $x \geq y \rightarrow x := x - y$
   $\square\ x \leq y \rightarrow y := y - x$
   fi
   $\{x \geq 0 \wedge y \geq 0\}$

4. $\{True\}$
   if $\neg a \vee b \rightarrow a := \neg a$
   $\square\ a \vee \neg b \rightarrow b := \neg b$
   fi
   $\{a \vee b\}$

5. $\{N \geq 0\}$
   $x := 0$
   do $x \neq N \rightarrow x := x + 1$
   od
   $\{x = N\}$

6. $\{N \geq 0\}$
   $x, y := 0, 0$
   do $x \neq 0 \rightarrow x := x - 1$
   $\square\ y \neq N \rightarrow x, y := N, y + 1$
   od
   $\{x = 0 \wedge y = N\}$

Ejercicio 17.10
Suponiendo que el programa de la izquierda es correcto, demostrar que el de la derecha también lo es, es decir, siempre se puede lograr que el **if** sea *determinístico* (solo *una* guarda verdadera).

$\{P\}$ **if** $B_0 \rightarrow S_0$ $\{Q\}$
$\square\ B_1 \rightarrow S_1$
fi

$\{P\}$ **if** $B_0 \wedge \neg B_1 \rightarrow S_0$ $\{Q\}$
$\square\ B_1 \rightarrow S_1$
fi

Ejercicio 17.11
Demostrar que el siguiente programa es equivalente a *Skip*:

do $False \rightarrow S$
od

Ejercicio 17.12
Demostrar las siguientes implicaciones:

1. $$\begin{array}{l}\{P\}\\ \underline{\textbf{if}}\quad B \to S\\ \underline{\textbf{fi}}\\ \{Q\}\end{array} \quad\Rightarrow\quad \begin{array}{l}\{P\}\\ S\\ \{Q\}\end{array}$$

2. $$\begin{array}{l}\{P\}\\ \underline{\textbf{if}}\quad B_0 \to S_0\\ \square\quad B_1 \to S_1\\ \underline{\textbf{fi}}\\ \{Q\}\end{array} \quad\Rightarrow\quad \begin{array}{l}\{P\}\\ \underline{\textbf{if}}\quad B_0 \to S_0\\ \square\quad \neg B_0 \to S_1\\ \underline{\textbf{fi}}\\ \{Q\}\end{array}$$

3. $$\begin{array}{l}\{P\}\\ \underline{\textbf{if}}\quad B_0 \to S_0\\ \square\quad B_1 \to S_1\\ \square\quad B_2 \to S_2\\ \underline{\textbf{fi}}\\ \{Q\}\end{array} \quad\Rightarrow\quad \begin{array}{l}\{P\}\\ \underline{\textbf{if}}\quad B_0 \to S_0\\ \square\quad \neg B_0 \to \underline{\textbf{if}}\quad B_1 \to S_1\\ \qquad\qquad\quad\ \square\quad B_2 \to S_2\\ \qquad\qquad\quad\ \underline{\textbf{fi}}\\ \underline{\textbf{fi}}\\ \{Q\}\end{array}$$

Ejercicio 17.13
[vdS93] En la sección 17.2 se ha especificado la sentencia *abort* mediante

$$\{P\}\, abort\, \{Q\} \;\equiv\; [P \equiv False]\ .$$

Considerar ahora una sentencia "inversa", es decir, especificada por:

$$\{P\}\, troba\, \{Q\} \;\equiv\; [Q \equiv False]\ .$$

¿Sería útil una sentencia como esta en un lenguaje de programación? ¿Es posible implementarla?

Ejercicio 17.14
[vdS93] Tomando como invariante $\{True\}$ y como función de cota $t.x = 2^x$, se puede demostrar que el programa

$$\{True\}\ \underline{\textbf{do}}\, True \to x := x - 1\ \underline{\textbf{od}}\ \{True\}$$

es correcto. ¿Cuál es el error?

Capítulo 18

Introducción al cálculo de programas imperativos

The Road goes ever on and on
    Down from the door where it began.
Now far ahead the road has gone,
    And I must follow, if I can,
Pursuing it with weary feet,
    Until it joins some larger way
Where many paths and errands meet.
    And wither then? I cannot say.

J.R.R.Tolkien:
The Lord of the Rings

En el capítulo anterior definimos los comandos básicos de un lenguaje de programación imperativo. Salvo en el caso de la repetición, para cada uno de ellos definimos la precondición más débil, la cual indica la forma en que se deben derivar programas que contengan estos comandos. En este capítulo nos concentraremos en la derivación de programas que incluyen repeticiones.

18.1 Derivación de ciclos

El teorema de invariancia proporciona las condiciones necesarias para demostrar que un ciclo es correcto. Ahora bien, ¿cómo desarrollar un ciclo a partir de la

especificación mediante pre y postcondición de un problema?

Pensemos, para simplificar, en un ciclo con una sola guarda. Podemos expresar el teorema de invariancia de la siguiente manera:

$$\left.\begin{array}{l}\{P \wedge B\}\, S\, \{P\} \\ \{P \wedge B \wedge t = T\}\, S\, \{t < T\} \\ P \wedge t \leq 0 \Rightarrow \neg B\end{array}\right\} \;\Rightarrow\; \{P\}\, \underline{\mathbf{do}}\ B \to S\ \underline{\mathbf{od}}\, \{P \wedge \neg B\}$$

Supongamos ahora que el problema a resolver está especificado por una precondición general R y una postcondición general Q. Comencemos por la postcondición: si se cumplen

- $\{P\}\, \underline{\mathbf{do}}\ B \to S\ \underline{\mathbf{od}}\, \{P \wedge \neg B\}$
- $P \wedge \neg B \;\Rightarrow\; Q$

entonces, por debilitamiento de la postcondición, se cumple

$\{P\}\, \underline{\mathbf{do}}\ B \to S\ \underline{\mathbf{od}}\, \{Q\}$

En otras palabras, si Q es la postcondición del problema, el invariante P y la guarda B deben elegirse de tal manera que $P \wedge \neg B \;\Rightarrow\; Q$.

Consideremos ahora la precondición: como P debe valer al inicio del ciclo, debemos derivar un programa S' tal que $\{R\}\, S'\, \{P\}$, proceso que se conoce como **inicialización**. De esta manera, tenemos

- $\{R\}\, S'\, \{P\}$
- $\{P\}\, \underline{\mathbf{do}}\ B \to S\ \underline{\mathbf{od}}\, \{Q\}$

y concatenando ambos, tenemos un programa con precondición R y postcondición Q, como deseábamos.

Concentrémonos ahora en el desarrollo del ciclo propiamente dicho, es decir en encontrar un programa S tal que $\{P\}\, \underline{\mathbf{do}}\ B \to S\ \underline{\mathbf{od}}\, \{P \wedge \neg B\}$, donde P es el invariante. Por el teorema de invariancia, S debe satisfacer

$\{P \wedge B\}\, S\, \{P\}$

$\{P \wedge B \wedge t = T\}\, S\, \{t < T\}$

En este punto del desarrollo, tanto el invariante P como la guarda B han sido determinados, por lo que no hay nada que agregar a la primera condición. Pero en la segunda condición interviene la cota t, que aún no ha sido determinada. El próximo paso es, entonces, elegir la función variante, lo cual debe hacerse de manera tal de satisfacer lo requerido por el teorema de invariancia, es decir, debe

cumplir $P \wedge t \leq 0 \Rightarrow \neg B$. Una vez determinada t, solo resta derivar el cuerpo del ciclo, S.

Sintetizando, dadas una precondición R y una postcondición Q, los pasos a seguir son:

1. Elegir el invariante P y la guarda B de manera que se satisfaga $P \wedge \neg B \Rightarrow Q$. Esto garantiza el cumplimiento de la postcondición al finalizar el ciclo.

2. Inicializar, desarrollando S' tal que $\{R\}\, S'\, \{P\}$. Esto garantiza el cumplimiento del invariante al comenzar el ciclo.

3. Elegir la cota t de manera que $P \wedge t \leq 0 \Rightarrow \neg B$, es decir, una función cuyo decrecimiento asegure que la guarda B sea falsa en algún momento. Esto garantiza la finalización del ciclo.

4. Derivar S de tal manera que $\{P \wedge B\}\, S\, \{P\}$ y $\{P \wedge B \wedge t = T\}\, S\, \{t < T\}$, es decir, un programa que mantenga el invariante y decrezca la función variante.

La elección del invariante resulta crucial para el desarrollo correcto del ciclo. El capítulo 19 está dedicado al estudio de algunas técnicas que se pueden aplicar para la determinación de invariantes, pero antes de introducirnos en este tema, sugerimos resolver los ejercicios que siguen, en los cuales el invariante está dado.

18.2 Ejercicios

Ejercicio 18.1
Derivar dos programas que calculen $r = X^Y$ a partir de cada una de las siguientes definiciones de la función exponencial:

$$\begin{array}{lll}
\text{(a)} & exp(x,y) & = \ (\ \ y = 0 \rightarrow 1 \\
 & & \ \square\ y \neq 0 \rightarrow x * exp(x, y-1) \\
 & & \ ) \\
\text{(b)} & exp(x,y) & = \ (\ \ y = 0 \rightarrow 1 \\
 & & \ \square\ y \neq 0 \rightarrow (\ \ y \ \mathsf{mod}\ 2 = 0 \rightarrow exp(x * x, y \ \mathsf{div}\ 2) \\
 & & \qquad\qquad\ \square\ y \ \mathsf{mod}\ 2 = 1 \rightarrow x * exp(x, y-1) \\
 & & \qquad\qquad\ ) \\
 & & \ )
\end{array}$$

Diseñar los dos programas a partir de:

Precondición R: $\{x = X \wedge y = Y \wedge x \geq 0 \wedge y \geq 0\}$

Postcondición Q: $\{r = X^Y\}$

Invariante P: $\{y \geq 0 \wedge r * x^y = X^Y\}$

Para cada programa usar una de las definiciones. Tener en cuenta las mismas a la hora de decidir la manera de achicar la cota.

Ejercicio 18.2

Dado $n > 0$, desarrollar un programa que devuelva en la variable k la mayor potencia de 2 menor o igual que n.

Precondición R: $\{n > 0\}$

Postcondición Q: $\{0 < k \leq n \wedge n < 2 * k \wedge \langle \exists j \ : 0 < j : \ k = 2^j \rangle\}$

Invariante P: $\{0 < k \leq n \wedge \langle \exists j \ : 0 < j : \ k = 2^j \rangle\}$

Capítulo 19

Técnicas para determinar invariantes

> In many of the more relaxed civilizations on the Outer Eastern Rim of the Galaxy, the *Hitchhiker's Guide* has already suplanted the great *Encyclopedia Galactica* as the standard repository of all knowledge and wisdom, for though it has many omissions and contains much that is apocryphal, or at least wildly inaccurate, it scores over the older, more pedestrian work in two important respects.
>
> First, it is slighty cheaper; and second it has the words DON'T PANIC inscribed in large friendly letters on its cover.
>
> Douglas Adams
> *The Hitchhiker's Guide to the Galaxy*

El primer paso en la derivación de un ciclo es la determinación del invariante P y la guarda B, los cuales, como hemos dicho, deben satisfacer $P \wedge \neg B \Rightarrow Q$, donde Q es la postcondición del problema. A continuación describiremos algunas técnicas para determinar invariantes.

19.1 Tomar términos de una conjunción

En algunas ocasiones la postcondición es una conjunción, digamos $Q = Q_0 \wedge Q_1$, o puede reescribirse como tal. En estos casos, la implicación $P \wedge \neg B \Rightarrow Q$ puede asegurarse fácilmente tomando como P a uno de los términos de la conjunción y como B a la negación del otro: $P = Q_0$ y $B = \neg Q_1$.

Ejemplo 19.1 (División entera)
Dados dos números, hay que encontrar el cociente y el resto de la división entera entre ellos.

Para enteros $x \geq 0$ e $y > 0$, el cociente q y el resto r de la división entera de x por y están caracterizados por $x = q * y + r \wedge 0 \leq r \wedge r < y$. Por lo tanto, debemos derivar un programa S que satisfaga

$$
\begin{array}{ll}
[\![\text{var } x, y, q, r : Int & \\
\{R: \quad x \geq 0 \wedge y > 0\} & \text{(precondición)} \\
S & \\
\{Q: \quad x = q * y + r \wedge 0 \leq r \wedge r < y\} & \text{(postcondición)} \\
]\!] &
\end{array}
$$

La poscondición es una conjunción de tres términos. El primero de ellos, $x = q * y + r$, debe ser válido en todo momento, ya que en cualquier paso intermedio de una división se verifica que el cociente obtenido hasta el momento multiplicado por el divisor más el resto que se tiene en ese momento es igual al dividendo. El término $0 \leq r$ también debe valer en todo momento, puesto que la división no admite restos negativos. El término restante, $r < y$, no es necesariamente válido en todo momento de la división; más aún, solo se verifica al completar la misma.

Por ello elegimos como invariante $\{P: \ x = q * y + r \wedge 0 \leq r\}$. La guarda es la negación del resto de la conjunción, es decir $\neg(r < y)$, de donde resulta $\{B: \ r \geq y\}$.

El próximo paso es establecer el invariante inicialmente. Como en P intervienen q y r, que no aparecen en la precondición R, la inicialización debe consistir en una asignación de valores a estas variables. Debemos elegir, pues, expresiones enteras E y F tales que $R \Rightarrow wp.(q, r := E, F).P$.

$$
\begin{array}{ll}
& wp.(q, r := E, F).P \\
\equiv & \{ \text{ definición de } wp \text{ y } P \ \} \\
& (x = q * y + r \wedge 0 \leq r)(q, r := E, F) \\
\equiv & \{ \text{ sustitución en predicados } \} \\
& x = E * y + F \wedge 0 \leq F
\end{array}
$$

$$
\begin{array}{ll}
\Leftarrow & \{ \text{ precondición y } E, F : \ Int \ \} \\
& E = 0 \wedge F = x
\end{array}
$$

Hasta aquí tenemos:

$$
\begin{array}{l}
[\![\text{var } x, y, q, r : Int \\
\ \{R : \ \ x \geq 0 \wedge y > 0\} \\
\ q, r := 0, x \\
\ \{P : \ \ x = q * y + r \wedge 0 \leq r\} \\
\ \underline{\textbf{do}}\ r \geq y \rightarrow S\ \underline{\textbf{od}} \\
\ \{Q : \ \ x = q * y + r \wedge 0 \leq r \wedge r < y\} \\
]\!]
\end{array}
$$

El paso siguiente es determinar la cota t. Sabemos que el decrecimiento de t debe asegurar que la guarda sea falsa en algún momento. Como la guarda es $r \geq y$, esto se lograría haciendo decrecer r o haciendo crecer y; pero lo segundo no es factible, por lo que elegimos lo primero. Como función variante, tomamos $t.r = r$.

Lo único que resta es derivar el cuerpo del ciclo. El análisis que hemos efectuado para determinar la cota nos indica que r debe decrecer. Postulemos, entonces, ejecutar $q, r := E, r - k$ y determinemos los valores adecuados de E y k, teniendo en cuenta que debemos mantener el invariante, es decir, se debe satisfacer $\{P \wedge B\}\,(q, r := E, r - k)\,\{P\}$. Esto queda garantizado demostrando $P \wedge B \Rightarrow wp.(q, r := E, r - k).P$.

$$
\begin{array}{ll}
& wp.(q, r := E, r - k).P \\
\equiv & \{ \text{ definición de } wp \ \} \\
& (x = q * y + r \wedge 0 \leq r)(q, r := E, r - k) \\
\equiv & \{ \text{ sustitución en predicados } \} \\
& x = E * y + (r - k) \wedge 0 \leq r - k \\
\equiv & \{ \ P \text{ y aritmética } \} \\
& q * y + r = E * y + r - k \wedge k \leq r \\
\equiv & \{ \text{ aritmética } \} \\
& E = q + \dfrac{k}{y} \wedge k \leq r
\end{array}
$$

Como E debe ser entero, k debe ser múltiplo de y. Además, k debe ser positivo, para que t decrezca. Podemos elegir, entonces, cualquier múltiplo positivo de y. Continuemos la derivación:

$$E = q + \frac{k}{y} \wedge k \leq r$$

$\Leftarrow$ { Elegimos $k = y$, el menor múltiplo positivo de y }
$E = q + 1 \wedge y \leq r$
$\equiv$ { $y \leq r$ por B }
$E = q + 1$

Finalmente obtenemos el programa completo:

$[\![\text{var } x, y, q, r : Int$
$q, r := 0, x$
$\underline{\textbf{do}}\ r \geq y \rightarrow q, r := q + 1, r - y\ \underline{\textbf{od}}$
$]\!]$

En algunas ocasiones, la postcondición no está dada inicialmente como una conjunción, pero puede reescribirse como tal.

Ejemplo 19.2 (Búsqueda lineal)
Dada $f : Nat \mapsto Bool$ y suponiendo $\langle \exists n : 0 \leq n : f.n \rangle$, encontrar el mínimo natural x tal que $f.x$.

Como precondición tenemos $\{R : \langle \exists n : 0 \leq n : f.n \rangle\}$.

La postcondición es $\{Q : x = \langle \text{Min } i : 0 \leq i \wedge f.i : i \rangle\}$. Como esta expresión no sirve para derivar el ciclo, busquemos una equivalente a ella que sea una conjunción, usando la definición de mínimo.

Tomemos $\{Q : 0 \leq x \wedge f.x \wedge \langle \forall i : 0 \leq i < x : \neg f.i \rangle\}$. El primer término de la conjunción establece que x debe ser natural, el segundo dice que debe valer $f.x$ y el tercero indica que $f.i$ no es válida para ningún natural i menor que x, con lo cual x resulta ser el mínimo natural i para el que vale $f.i$.

El término $f.x$ no puede formar parte del invariante, pues como este debe valer al comienzo del ciclo, deberíamos inicializar con la solución del problema, valor que desconocemos. Por eso elegimos como guarda la negación de este término y como invariante el resto de la conjunción:

$P : 0 \leq x \wedge \langle \forall i : 0 \leq i < x : \neg f.i \rangle$
$B : \neg f.x$

Como la única variable de estado que figura en el invariante es x, debemos inicializar asignando un valor a esta variable, es decir, debemos encontrar E tal que $R \Rightarrow wp.(x := E).P$

$wp.(x := E).(0 \leq x \wedge \langle \forall i : 0 \leq i < x : \neg f.i \rangle)$
$\equiv$ { definición de wp }
$0 \leq E \wedge \langle \forall i : 0 \leq i < E : \neg f.i \rangle$

$\Leftarrow$ { rango vacío asegura que el segundo término sea $True$ }

$E = 0$

Hasta aquí tenemos:

$\{\!|$var $x : Int$
$\{R : \ \langle \exists n \ : 0 \leq n : \ f.n \rangle\}$
$x := 0$
$\{P : \ \ 0 \leq x \wedge \langle \forall i \ : 0 \leq i < x : \ \neg f.i \rangle\}$
do $\neg f.x \rightarrow S$ **od**
$\{Q : \ \ 0 \leq x \wedge f.x \wedge \langle \forall i \ : 0 \leq i < x : \ \neg f.i \rangle\}$
$|\!\}$

La única información que tenemos sobre f es que $f.N$ vale para algún N, por lo que el candidato a cota es $t.x = N - x$.

El modo de hacer decrecer la cota es incrementar x. Propongamos la asignación $x := x + k$ y encontremos el valor adecuado de k, teniendo presente que se debe satisfacer $P \wedge B \ \Rightarrow \ wp.(x := x + k).P$

$wp.(x := x + k).(0 \leq x \wedge \langle \forall i \ : 0 \leq i < x : \ \neg f.i \rangle)$

$\equiv$ { definición de wp }

$0 \leq x + k \wedge \langle \forall i \ : 0 \leq i < x + k : \ \neg f.i \rangle$

$\equiv$ { partición de rango }

$0 \leq x+k \wedge \langle \forall i \ : 0 \leq i < x : \ \neg f.i \rangle \wedge \neg f.x \wedge \langle \forall i \ : x + 1 \leq i < x + k : \ \neg f.i \rangle$

$\equiv$ { el segundo y tercer términos son equivalentes a $True$ por P y B }

$0 \leq x + k \wedge \langle \forall i \ : x + 1 \leq i < x + k : \ \neg f.i \rangle$

Cualquier valor positivo de k asegura el cumplimiento del primer término. Dado que no conocemos ninguna propiedad de f que nos ayude, la única forma de asegurar que el segundo término sea $True$ es haciendo que el rango sea vacío, por lo que el único valor posible para k es 1.

$0 \leq x + k \wedge \langle \forall i \ : x + 1 \leq i < x + k : \ \neg f.i \rangle$

$\Leftarrow$ { $k = 1$ }

$0 \leq x + 1 \wedge \langle \forall i \ : x + 1 \leq i < x + 1 : \ \neg f.i \rangle$

$\equiv$ { $x + 1 \geq 0$ por P; rango vacío en el segundo término }

$True$

Finalmente estamos en condiciones de dar un programa para la búsqueda lineal:

$$
\begin{array}{l}
[\![\text{var } x : Int \\
\ \{R : \ \langle \exists n \ : 0 \leq n : \ f.n \rangle\} \\
\ x := 0 \\
\ \{P : \ \ 0 \leq x \wedge \langle \forall i \ : 0 \leq i < x : \ \neg f.i \rangle\} \\
\ \underline{\textbf{do}} \ \neg f.x \rightarrow x := x + 1 \ \underline{\textbf{od}} \\
\ \{Q : \ \ 0 \leq x \wedge f.x \wedge \langle \forall i \ : 0 \leq i < x : \ \neg f.i \rangle\} \\
]\!]
\end{array}
$$

Ejemplo 19.3 (División entera mejorada)
El costo de un ciclo se obtiene multiplicando la cantidad de operaciones realizadas dentro del ciclo por la cantidad de repeticiones del mismo. Por eso, es deseable repetir la menor cantidad de veces. Volvamos a considerar el problema de encontrar el cociente y el resto de la división entera de dos números. Al derivar este programa vimos que el cuerpo del ciclo debía ser de la forma $q, r := E, r - k$, donde E y k debían satisfacer $E = q + k/y \wedge k \leq r$. Además, por ser r la cota, k también debía satisfacer $0 < k$. De allí resultó que k debía ser un múltiplo positivo de y. En aquel momento elegimos $k = y$. Pensemos ahora en la posibilidad de considerar un valor más grande de k, lo que haría decrecer la cota más rápidamente, reduciendo la cantidad de repeticiones del ciclo y por ende su costo. Para ello dejemos expresado $k = d * y$, sin darle un valor específico a d aún.

$$
\begin{array}{ll}
& E = q + \dfrac{k}{y} \wedge 0 < k \wedge k \leq r \\
\equiv & \{\ k = d * y\ \} \\
& E = q + \dfrac{d * y}{y} \wedge 0 < d * y \wedge d * y \leq r \\
\equiv & \{\ y > 0 \text{ por precondición, aritmética}\ \} \\
& E = q + d \wedge 0 < d \wedge d * y \leq r \\
\equiv & \{\ \text{álgebra}\ \} \\
& E = q + d \wedge 1 \leq d \wedge d * y \leq r \\
\equiv & \{\ \text{suponiendo } 1 \leq d \wedge d * y \leq r\ \} \\
& E = q + d
\end{array}
$$

Esto nos permite escribir

$$
\begin{array}{l}
[\![\text{var } x, y, q, r, d : Int \\
\ \{R : \ \ x \geq 0 \wedge y > 0\} \\
\ q, r := 0, x \\
\ \{P : \ \ x = q * y + r \wedge 0 \leq r\}
\end{array}
$$

$\underline{\mathbf{do}}\ r \geq y \rightarrow S_0;\ q, r := q + d, r - d * y\ \underline{\mathbf{od}}$
$\{Q: \quad x = q * y + r \wedge 0 \leq r \wedge r < y\}$
$]\!]$

donde S_0 consiste en determinar d tal que $1 \leq d \wedge d * y \leq r$. Más aún, para evitar la multiplicación $d * y$ dentro del ciclo, podemos mantener invariante $dd = d * y$. De esta manera, debemos derivar el siguiente algoritmo:

$[\!]$[var $x, y, q, r, d, dd : Int$
$\{R: \quad x \geq 0 \wedge y > 0\}$
$q, r := 0, x$
$\{P: \quad x = q * y + r \wedge 0 \leq r\}$
$\underline{\mathbf{do}}\ r \geq y \rightarrow S_1;\ q, r := q + d, r - dd\ \underline{\mathbf{od}}$
$\{Q: \quad x = q * y + r \wedge 0 \leq r \wedge r < y\}$
$]\!]$

donde S_1 consiste en calcular d y dd tales que $1 \leq d \wedge d * y \leq r \wedge dd = d * y$. Para derivar S_1, llamemos

$R_0: \ 1 \leq d$
$R_1: \ d * y \leq r$
$R_2: \ dd = d * y$

Lo que buscamos es el máximo valor de d que satisface $R_0 \wedge R_1$, lo cual equivale a buscar el mínimo valor de d tal que $r < (d + 1) * y$. Como además sabemos que existe un d que satisface esta condición, podríamos pensar en resolver el problema usando búsqueda lineal.

Ejercicio: Determinar d usando búsqueda lineal.

Sin embargo, esta solución no es eficiente, pues cada repetición tendrá costo lineal. Para disminuir el costo, debemos encontrar una manera de incrementar d más rápido que linealmente. Pidamos, por ejemplo, que d sea una potencia de 2:

$R_3: \ d$ es potencia de 2

Notemos que al pedir esto estamos tomando una decisión de diseño, pues estamos restringiendo los candidatos a d; pero no es una restricción tan drástica como la que hicimos al tomar $d = 1$ $(k = y)$.

Además, pediremos que d sea la máxima potencia de 2 posible, lo que -debido a R_1- da lugar a una restricción más:

$R_4: \ r < 2 * d * y$

En otras palabras, S_1 tiene como postcondición $RR = R_0 \wedge R_1 \wedge R_2 \wedge R_3 \wedge R_4$ y su precondición es $P \wedge r \geq y$.

Derivemos ahora S_1 a partir de su especificación. Aprovechando que la post-condición es una conjunción, intentemos derivar un ciclo. El primer paso es elegir el invariante. Claramente, tanto R_0 como R_3 deben formar parte del invariante, pues contienen información acerca de d que no debe ser cambiada. Lo mismo ocurre con R_2, que es el único término de la conjunción que dice algo acerca de dd. De las otras dos, solo una puede usarse en la guarda, debiendo la otra quedar en el invariante. Eligiendo R_4 para la guarda (la elección es arbitraria), obtenemos

invariante: $PP: \ R_0 \wedge R_1 \wedge R_2 \wedge R_3$
guarda: $BB: \ \neg R_4$

Ahora debemos inicializar d y dd de manera tal que

$$\{P \wedge r \geq y\}\, d, dd := E, F\, \{PP\} \quad .$$

Ejercicio: Probar que la inicialización $d, dd := 1, y$ es corrrecta.

Hasta este punto, tenemos

$\{P \wedge r \geq y\}$
$d, dd := 1, y$
$\{PP\}$
do $r \geq 2 * d * y \rightarrow SS$ **od**
$\{RR\}$

Nos resta derivar SS. Lo único que podemos hacer para lograr que la guarda sea falsa en algún momento es incrementar d. Por lo tanto, elegimos como cota $t.r.d = r - d$. Como la variable dd depende del valor de d, también será necesario modificarla. En otras palabras, proponemos como SS la asignación $d, dd := E, F$. Ahora bien, como esta asignación debe mantener el invariante (en particular d debe ser una potencia de 2) la forma obvia de modificar d es $d := d{*}2$ y consecuentemente también habrá que hacer $dd := dd * 2$. Probemos que esto es correcto, es decir, que $\{PP \wedge BB\}\, d, dd := 2 * d, 2 * dd\, \{PP\}$:

$$
\begin{array}{ll}
 & wp.(d, dd := 2 * d, 2 * dd).PP \\
\equiv & \{ \text{ definición de } wp \text{ y } PP \ \} \\
 & (1 \leq d \wedge d * y \leq r \wedge dd = d * y \wedge \langle\, \exists\, k \ : k \geq 0 : \ d = 2^k \,\rangle)(d, dd := 2 * d, 2 * dd) \\
\equiv & \{ \text{ sustitución en predicados } \} \\
 & 1 \leq 2 * d \wedge 2 * d * y \leq r \wedge 2 * dd = 2 * d * y \wedge \langle\, \exists\, k \ : k \geq 0 : \ 2 * d = 2^k \,\rangle \\
\equiv & \{ \text{ aritmética, definición de } BB \ \} \\
 & 1 \leq 2 * d \wedge BB \wedge dd = d * y \wedge \langle\, \exists\, k \ : k \geq 0 : \ d = 2^{k-1} \,\rangle \\
\equiv & \{ \ d \geq 1 \text{ por } PP \ \} \\
 & 1 \leq 2 * d \wedge BB \wedge dd = d * y \wedge \langle\, \exists\, k \ : k \geq 1 : \ d = 2^{k-1} \,\rangle
\end{array}
$$

$\Leftarrow$ { aritmética }

$1 \leq d \wedge BB \wedge dd = d * y \wedge \langle\, \exists k \;:\; k \geq 0 :\; d = 2^k \,\rangle$

$\equiv$ { definición de PP }

$PP \wedge BB$

Finalmente, tenemos el programa completo:

$$
\begin{array}{l}
[\!|\text{var } x, y, q, r, d, dd : Int \\
\;q, r := 0, x \\
\;\underline{\textbf{do}}\; r \geq y \rightarrow \\
\qquad d, dd := 1, y \\
\qquad \underline{\textbf{do}}\; r \geq 2 * dd \rightarrow d, dd := 2 * d, 2 * dd \;\underline{\textbf{od}} \\
\qquad q, r := q + d, r - dd \\
\underline{\textbf{od}} \\
|\!]
\end{array}
$$

Observación: Las variables d y dd solo son necesarias en el ciclo interno, por lo que puede definírselas como variables locales:

$$
\begin{array}{l}
[\!|\text{var } x, y, q, r : Int \\
\;q, r := 0, x \\
\;\underline{\textbf{do}}\; r \geq y \rightarrow \\
\qquad [\!|\text{var } d, dd : Int \\
\qquad\; d, dd := 1, y \\
\qquad\; \underline{\textbf{do}}\; r \geq 2 * dd \rightarrow d, dd := 2 * d, 2 * dd \;\underline{\textbf{od}} \\
\qquad\; q, r := q + d, r - dd \\
\qquad |\!] \\
\underline{\textbf{od}} \\
|\!]
\end{array}
$$

Ejercicio: Derivar S_1 tomando como invariante $PP = R_0 \wedge R_2 \wedge R_3 \wedge R_4$ y como guarda $BB = \neg R_1$.

19.2 Reemplazo de constantes por variables

A veces es imposible expresar la postcondición como una conjunción. En estos casos, hay que apelar a otras técnicas para la determinación de invariantes. Una técnica muy simple, pero no por ello menos útil, es la de reemplazar en la postcondición constantes por variables.

Antes de dar un ejemplo, queremos introducir la noción de arreglos o vectores (en inglés, arrays), los que van a cumplir el rol de estructuras ordenadas, similares

a las listas que manejábamos al usar programación funcional. Un **arreglo** es una función definida sobre un segmento de los naturales. En general, un arreglo se declara de la siguiente manera:

$$a : \ array\ [p, q)\ of\ [A]$$

donde $p \leq q$ y A indica el tipo de los elementos del arreglo. Por ejemplo:

$a : array\ [0, 4)\ of\ Int$	declara un arreglo de 4 elementos de tipo entero.
$a : array\ [5, 7)\ of\ Bool$	declara un arreglo de 2 elementos de tipo $Bool$.

- La cantidad de elementos de un arreglo es $q - p$. Como permitimos $q = p$, el arreglo puede no contener elementos. En este caso, se dirá que es un arreglo vacío.
- No es necesario que el primer índice del arreglo sea 0.
- Un elemento de un arreglo a se referenciará por $a.n$, donde n solo tiene sentido en el rango $p \leq n < q$.
- Diremos que un valor v está en el arreglo si algún elemento del arreglo es igual a v.
- Es posible usar variables cuantificadas en relación a arreglos. Por ejemplo: si $a : array\ [p, q)\ of\ Int$, entonces $\langle \text{Max}\ i\ : p \leq i < q :\ a.i\ \rangle$ indica el máximo elemento del arreglo.

Ahora sí, veamos un ejemplo de determinación de invariantes mediante el reemplazo de constantes por variables en la postcondición.

Ejemplo 19.4 (Suma de los elementos de un arreglo)
Dado un arreglo de enteros, se debe devolver en una variable la suma de todos los elementos del arreglo.

Debemos derivar un programa que satisfaga la siguiente especificación:

$$
\begin{array}{l}
\|[\text{con } N : Int;\ a : array\ [0, N)\ of\ Int \\
\ \text{var } x : Int \\
\ \{R :\ \ N \geq 0\} \\
\ S \\
\ \{Q :\ \ x = \langle \sum i\ : 0 \leq i < N :\ a.i\ \rangle\} \\
]\|
\end{array}
$$

El cuantificador que aparece en la postcondición involucra las constantes 0 y N. Tomemos como invariante P lo que se obtiene al reemplazar en la postcondición la constante N por la variable n:

$$\{P :\ x = \langle \sum i\ : 0 \leq i < n :\ a.i\ \rangle\}\ .$$

Con esta definición de P, resulta $P \wedge (n = N) \Rightarrow Q$. De aquí que la guarda es $\neg(n = N)$, es decir, $n \neq N$. Ahora bien, no podemos permitir valores de n mayores que N, pues la evaluación $a.i$ que aparece en el invariante solo tiene sentido cuando $0 \leq i < N$. Por eso, al introducir variables nuevas, es generalmente necesario reforzar el invariante, a fin de evitar indefiniciones. El refuerzo consiste en agregar una cláusula indicando el rango para la variable recientemente introducida. En nuestro caso, debemos pedir

invariante: $P: \quad x = \langle \sum i \ : 0 \leq i < n : \ a.i \, \rangle \ \wedge \ 0 \leq n \leq N$

guarda: $B: \quad n \neq N$

Observar que el refuerzo del invariante permitiría tomar como guarda $n < N$ en lugar de $n \neq N$.

De aquí en más, la derivación se desarrolla del modo acostumbrado. El paso siguiente es inicializar de manera tal de asegurar $\{R\}$*inicialización*$\{P\}$.

Ejercicio: Probar que la inicialización $n, x := 0, 0$ es correcta.

Como cota elegimos la función $t.n = N - n$. El cuerpo del ciclo consistirá, pues, en incrementar el valor de n. Naturalmente, una modificación en el valor de n producirá una modificación en el valor de x, pues este depende de n. Probemos con un incremento de n en una unidad. Debemos encontrar, entonces, una expresión E tal que $P \wedge B \Rightarrow wp.(n, x := n + 1, E).P$:

$$
\begin{array}{ll}
 & wp.(n, x := n + 1, E).(x = \langle \sum i \ : 0 \leq i < n : \ a.i \, \rangle \ \wedge \ 0 \leq n \leq N) \\
\equiv & \{ \text{ definición de } wp \ \} \\
 & E = \langle \sum i \ : 0 \leq i < n + 1 : \ a.i \, \rangle \wedge 0 \leq n + 1 \leq N \\
\equiv & \{ \text{ partición de rango y } P \ \} \\
 & E = \langle \sum i \ : 0 \leq i < n : \ a.i \, \rangle + a.n \wedge 0 \leq n < N \\
\equiv & \{ \text{ por } P \text{ y } B \ \} \\
 & E = x + a.n \wedge 0 \leq n < N
\end{array}
$$

Por lo tanto, arribamos al siguiente programa:

$$
\begin{array}{l}
[\![\text{con } N : Int; \ a : array \ [0, N) \ of \ Int \\
\ \text{var } x : Int \\
\ n, x := 0, 0 \\
\ \underline{\textbf{do}} \ n \neq N \rightarrow x, n := x + a.n, n + 1 \ \underline{\textbf{od}} \\
]\!]
\end{array}
$$

Ejercicio: Derivar el programa tomando como invariante lo que resulta al reemplazar en la postcondición la constante 0 por la variable n.

Ejemplo 19.5 (Exponenciación de enteros)
Dados dos enteros no negativos A y B, calcular A^B (suponer, para simplificar, $0^0 = 1$).

La especificación del programa es:

$[\![$con $A, B : Int$
var $r : Int$
$\{R : \quad A \geq 0 \wedge B \geq 0\}$
S
$\{Q : \quad r = A^B\}$
$]\!]$

En la postcondición aparecen las constantes A y B. Podríamos pensar en reemplazar solo una de ellas o bien ambas a la vez por una variable, es decir, los candidatos a invariante son: $r = A^x$, $r = x^B$, $r = x^y$. El segundo es inadecuado, dado que no podemos hacer inducción sobre x. El tercero es una generalización del primero. Tomemos, pues, el primero: $r = A^x \wedge x = B \Rightarrow Q$, de donde resulta que la guarda del ciclo debe ser $\neg(x = B)$. Reforcemos el invariante agregando un rango para x:

invariante: $P : \ r = A^x \wedge 0 \leq x \leq B$

guarda: $B : \ x \neq B$

Ejercicio: Probar que la inicialización $x, r := 0, 1$ es correcta.

La cota es $t.x = B - x$, que decrecerá mediante un incremento de la variable x, el cual, a su vez, producirá una modificación de la variable r. Nuevamente, el incremento natural es en una unidad.
Busquemos E tal que $P \wedge B \Rightarrow wp.(x, r := x + 1, E).P$:

$$
\begin{array}{ll}
 & wp.(x, r := x + 1, E).(r = A^x \wedge 0 \leq x \leq B) \\
\equiv & \{ \text{ definición de } wp \ \} \\
 & E = A^{x+1} \wedge 0 \leq x + 1 \leq B \\
\Leftarrow & \{ \text{ álgebra } \} \\
 & E = A^x * A \wedge 0 \leq x < B \\
\equiv & \{ \text{ por } P \text{ y } B \ \} \\
 & E = r * A
\end{array}
$$

De esta manera, el programa buscado es

$[\![$con $A, B : Int$
var $r : Int$
$x, r := 0, 1$
do $x \neq B \rightarrow x, r := (x + 1), r * A$ **od**
$]\!]$

Ejercicio: Después de leer la sección 19.3, resuelva el problema tomando como invariante $r = x^y$.

19.3 Fortalecimiento de invariantes

El método usado para construir un ciclo consiste en plantear la forma de este dejando algunas expresiones sin resolver, lo cual puede pensarse como ecuaciones con estas (meta)variables formales como incógnitas. Luego calculamos para "despejar" las incógnitas. Este proceso puede dar lugar a nuevas expresiones las cuales no admitan una expresión simple en términos de las variables de programa. Una forma de resolver esta situación es introducir nuevas variables que se mantengan invariantemente igual a algunas de estas subexpresiones. Esto resuelve el problema a costa de introducir la necesidad de mantener el nuevo invariante.

La técnica de fortalecimiento de invariantes es en algunos casos análoga a la aplicación en funcional de la técnica de modularización (junto con la técnica de tuplas para mejorar la eficiencia) y en otros a una forma particular de aplicación de la técnica de generalización.

Ejemplo 19.6 (Tabla de cubos)
Consideremos el problema de calcular $x = N^3$, con N natural, usando solamente sumas. Aplicando la técnica de reemplazo de constantes por variables, proponemos como invariante

$$P : \ P_0 \wedge P_1 \ ,$$

donde

$$P_0 : \ x = n^3 \ ; \quad P_1 : \ 0 \leq n \leq N \ ,$$

siendo la guarda del ciclo $n \neq N$ y la función de cota $t.n = N - n$.

Ejercicio: Probar que la inicialización $x, n := 0, 0$ es correcta.

Incrementemos n en una unidad y derivemos el programa que mantiene P_0 invariante (la invariancia de P_1 se deja como ejercicio para el lector):

$$\begin{array}{ll} & wp.(x, n := E, n+1).P_0 \\ \equiv & \{ \text{ def. de } wp \ \} \\ & E = (n+1)^3 \\ \equiv & \{ \text{ aritmética } \} \\ & E = n^3 + 3n^2 + 3n + 1 \end{array}$$

Para obtener el valor de E, es necesario sumar $3n^2 + 3n + 1$ al valor anterior de x. Pero esta suma no puede expresarse fácilmente en términos de n y x usando solo sumas. Por ello, introducimos una nueva variable que representa este valor y fortalecemos el invariante agregando un término que mantenga la invariancia de la nueva variable:

$$P : \ P_0 \wedge P_1 \wedge P_2 \ ,$$

donde

$$P_2 : \ y = 3n^2 + 3n + 1 \ .$$

Ejercicio: Probar que la inicialización $x, y, n := 0, 1, 0$ es correcta.

$$\begin{array}{ll} & wp.(x, y, n := E, F, n+1).(P_0 \wedge P_2) \\ \equiv & \{ \text{ def. de } wp \ \} \\ & E = (n+1)^3 \wedge F = 3(n+1)^2 + 3(n+1) + 1 \\ \equiv & \{ \text{ aritmética } \} \\ & E = n^3 + 3n^2 + 3n + 1 \wedge F = 3n^2 + 6n + 3 + 3n + 3 + 1 \\ \equiv & \{ \text{ por } P \ \} \\ & E = x + y \wedge F = y + 6n + 6 \end{array}$$

Aplicando nuevamente la misma estrategia, introducimos $z = 6n + 6$ y fortalecemos el invariante:

$$P : \ P_0 \wedge P_1 \wedge P_2 \wedge P_3 \ ,$$

donde

$$P_3 : \ z = 6n + 6 \ .$$

Ejercicio: Determinar la inicialización correcta.

Con una derivación análoga a la anterior, se obtiene finalmente el siguiente programa:

$$\begin{array}{l} [\![\text{ con } N : Int; \text{var } x, y, z : Int; \\ \quad \{N \geq 0\} \\ \quad x, y, z, n := 0, 1, 6, 0 \\ \quad \{P_0 \wedge P_1 \wedge P_2 \wedge P_3\} \\ \quad \underline{\textbf{do}} \ n \neq N \rightarrow x, y, z, n := x + y, y + z, z + 6, n + 1 \ \underline{\textbf{od}} \\ \quad \{x = N^3\} \\ ]\!] \end{array}$$

Ejemplo 19.7 (Fibonacci)
Consideremos nuevamente la función que calcula la sucesión de Fibonacci.

$$
\begin{array}{|l}
\hline
fib : Nat \mapsto Nat \\
\hline
\quad\quad fib.0 \doteq 0 \\
\quad\quad fib.1 \doteq 1 \\
\quad fib.(n+2) \doteq fib.n + fib.(n+1) \\
\hline
\end{array}
$$

Derivaremos un programa imperativo que calcule Fibonacci para un valor dado. El desarrollo presentado puede encontrarse en [Kal90]. La especificación del problema es la siguiente:

$$
\begin{array}{l}
[\![\text{ con } N : Int;\ \text{var } r : Int; \\
\quad \{N \geq 0\} \\
\quad S \\
\quad \{r = fib.N\} \\
]\!]
\end{array}
$$

Usando la técnica de reemplazo de constantes por variables, proponemos como invariante

$$P :\ P_0 \wedge P_1\ ,$$

donde

$$P_0 :\ r = fib.n \qquad P_1 :\ 0 \leq n \leq N\ ,$$

siendo la guarda $n \neq N$ y la función de cota $t.n = N - n$.

Ejercicio: Probar que la inicialización $r, n := 0, 0$ es correcta.

Proceder como lo hemos hecho antes, es decir, incrementar n en una unidad, conduce a la expresión $fib.(n+1)$, la cual no puede expresarse fácilmente en términos de r y n. Por ello agregamos una variable que represente este valor y fortalecemos el invariante introduciendo un término que se ocupe de mantener esta nueva variable invariantemente igual a $fib.(n+1)$:

$$P :\ P_0 \wedge P_1 \wedge P_2\ ,$$

donde

$$P_2 :\ y = fib.(n+1)\ .$$

Ejercicio: Probar que la inicialización $r, y, n := 0, 1, 0$ es correcta.

Incrementemos ahora n en una unidad y derivemos el programa que mantiene $P_0 \wedge P_2$ invariante:

$$
\begin{array}{ll}
& wp.(r, y, n := E, F, n+1).(P_0 \wedge P_2) \\
\equiv & \{ \text{ def. de } wp \ \} \\
& E = fib.(n+1) \wedge F = fib.(n+2) \\
\equiv & \{ \text{ definición de } fib,\ n \geq 0, \text{ validez de } P \ \} \\
& E = y \wedge F = E + r
\end{array}
$$

Por lo tanto, la asignación simultánea $r, y, n := y, y + r, n + 1$ asegura el invariante dentro del ciclo. De esta manera, obtenemos el siguiente programa:

$$
\begin{array}{l}
[\![\text{ con } N : Int; \text{var } r, y : Int; \\
\quad \{N \geq 0\} \\
\quad r, y, n := 0, 1, 0 \\
\quad \{P_0 \wedge P_1 \wedge P_2\} \\
\quad \underline{\textbf{do}}\ n \neq N \rightarrow r, y, n := y, y + r, n + 1\ \underline{\textbf{od}} \\
\quad \{r = fib.N\} \\
]\!]
\end{array}
$$

Ejemplo 19.8 (Un problema para listas de números)
Consideramos aquí un problema ya resuelto en programación funcional usando la técnica de generalización por abstracción. La especificación es ligeramente diferente para adaptarlo al caso imperativo (arreglos en lugar de listas).

Dado un arreglo $A[0, N)$ de enteros, se pide determinar si alguno de sus elementos es igual a la suma de todos los elementos que lo preceden. Este problema se puede especificar con la siguiente postcondición:

$$r \equiv \langle \exists\, i \ : 0 \leq i < N : \ A.i = \langle \sum j \ : 0 \leq j < i : \ A.j \,\rangle \,\rangle \,.$$

Usaremos la siguiente abreviatura para la sumatoria:

$$sum.i = \langle \sum j \ : 0 \leq j < i : \ A.j \,\rangle \quad .$$

Con la ayuda de esta función, podemos especificar el problema como:

$$r \equiv \langle \exists\, i \ : 0 \leq i < N : \ A.i = sum.i \,\rangle \quad .$$

Reemplazando la constante N por una variable n, obtenemos los siguientes invariantes y esquema de programa:

$$
\begin{array}{lcl}
P_0 & : & r = \langle \exists\, i \ : 0 \leq i < n : \ A.i = sum.i \,\rangle \\
P_1 & : & 0 \leq n \leq N
\end{array}
$$

$[\![$con $N : Int;\ A : array\ [0, N)\ of\ Int$
var $x : Int$
$\{N \geq 0\}$
$n, r := 0, False$
$\{P_0 \wedge P_1\}$
do $n \neq N \rightarrow r, n := E, n + 1$ **od**
$]\!]$

Tratemos ahora de encontrar E

$wp.(r, n := E, n + 1).P_0$
$\equiv$ { def. de wp }
$E \equiv \langle\, \exists\, i\ : 0 \leq i < n + 1 :\ A.i = sum.i \,\rangle$
$\equiv$ { separación de un término }
$E \equiv \langle\, \exists\, i\ : 0 \leq i < n :\ A.i = sum.i \,\rangle \vee A.n = sum.n$
$\equiv$ { P_0 }
$E \equiv r \vee A.n = sum.n$

En este punto introduciremos una nueva variable que nos permita eliminar la expresión $sum.n$ la cual no es una expresión válida del lenguaje de programación. Hay dos alternativas: una, usar una variable que se mantenga igual a todo el término de la disyunción $A.n = sum.n$. El problema es que será imposible mantener este nuevo invariante sin introducir nuevas variables (se invita al lector a intentar ese desarrollo). La opción que seguiremos aquí es la de introducir una variable s que se mantenga invariantemente igual a $sum.n$. Es elemental completar esta derivación, por lo cual lo dejamos como ejercicio, presentando solamente la solución final:

P_0 : $r = \langle\, \exists\, i\ : 0 \leq i < n :\ A.i = sum.i \,\rangle$
P_1 : $0 \leq n \leq N$
P_2 : $s = sum.n$

$[\![$con $N : Int;\ A : array\ [0, N)\ of\ Int$
var $x : Int$
$\{N \geq 0\}$
$n, r, s := 0, False, 0$
$\{P_0 \wedge P_1 \wedge P_2\}$
do $n \neq N \rightarrow r, n, s := r \vee A.n = s, n + 1, s + A.n$ **od**
$]\!]$

19.4 Problemas con los bordes

Los siguientes ejemplos presentan casos en los cuales la forma "obvia" de reforzar el invariante incluye una expresión que no está definida en algún punto, típicamente debido a algún índice fuera de rango en un arreglo. Las soluciones a estos problemas no son universales, cada caso debe ser considerado por separado.

Ejemplo 19.9 (Segmento de suma máxima)
Dado un arreglo de enteros, se debe guardar en una variable la suma del segmento de suma máxima del arreglo.

La especificación del programa es:

$$
\begin{array}{l}
[\![\text{con } N : Int,\ A : array[0, N)\ of\ Int \\
\ \text{var } r : Int \\
\ \{N \geq 0\} \\
\ S \\
\ \{r = \langle \text{Max } p, q\ : 0 \leq p \leq q \leq N :\ sum.p.q \,\rangle\} \\
\qquad\qquad [\![sum.p.q = \langle \textstyle\sum i\ : p \leq i < q :\ A.i \,\rangle]\!] \\
]\!]
\end{array}
$$

Reemplazando en la postcondición la constante N por la variable n, postulamos como invariante $P_0 \wedge P_1$, donde:

$$
\begin{array}{l}
P_0 :\ r = \langle \text{Max } p, q\ : 0 \leq p \leq q \leq n :\ sum.p.q \,\rangle \\
\qquad\quad [\![sum.p.q = \langle \textstyle\sum i\ : p \leq i < q :\ A.i \,\rangle]\!]
\end{array}
$$

$$P_1 :\ 0 \leq n \leq N$$

La guarda es $n \neq N$ y la función de cota es $t.n = N - n$. Inicializando $n, r := 0, 0$ aseguramos el invariante, por lo que tenemos

$$
\begin{array}{l}
[\![\text{con } N : Int,\ A : array[0, N)\ of\ Int \\
\ \text{var } r : Int \\
\ \{N \geq 0\} \\
\ n, r := 0, 0 \\
\ \{P_0 \wedge P_1\} \\
\ \underline{\textbf{do}}\ n \neq N \rightarrow S\ \underline{\textbf{od}} \\
\ \{r = \langle \text{Max } p, q\ : 0 \leq p \leq q \leq N :\ sum.p.q \,\rangle\} \\
\qquad\qquad [\![sum.p.q = \langle \textstyle\sum i\ : p \leq i < q :\ A.i \,\rangle]\!] \\
]\!]
\end{array}
$$

Incrementemos n en una unidad y derivemos el ciclo que mantiene el invariante:

$$
\begin{array}{ll}
 & wp.(n, r := n + 1, E).(P_0 \wedge P_1) \\
\equiv & \{\ \text{definición de } wp\ \} \\
 & E = \langle \text{Max } p, q\ : 0 \leq p \leq q \leq n + 1 :\ sum.p.q \,\rangle \wedge 0 \leq n + 1 \leq N \\
 & \qquad [\![sum.p.q = \langle \textstyle\sum i\ : p \leq i < q :\ A.i \,\rangle]\!]
\end{array}
$$

$\equiv$ { partición de rango en el primer término; $P_1 \wedge n \neq N$ }

$E = \langle \text{Max } p, q \ : 0 \leq p \leq q \leq n : \ sum.p.q \ \rangle$

$max \ \langle \text{Max } p, q \ : 0 \leq p \leq q = n+1 : \ sum.p.q \ \rangle$

$[\![sum.p.q = \langle \sum i \ : p \leq i < q : \ A.i \ \rangle]\!]$

$\equiv$ { P_0 y anidado }

$E = r \text{ max } \langle \text{Max } q \ : q = n+1 : \ \langle \text{Max } p \ : 0 \leq p \leq q : \ sum.p.q \ \rangle \ \rangle$

$[\![sum.p.q = \langle \sum i \ : p \leq i < q : \ A.i \ \rangle]\!]$

$\equiv$ { rango unitario }

$E = r \text{ max } \langle \text{Max } p \ : 0 \leq p \leq n+1 : \ sum.p.(n+1) \ \rangle$

$[\![sum.p.q = \langle \sum i \ : p \leq i < q : \ A.i \ \rangle]\!]$

$\equiv$ { introducimos $u = \langle \text{Max } p \ : 0 \leq p \leq n+1 : \ sum.p.(n+1) \ \rangle$ (*) }

$E = r \text{ max } u$

$[\![u = \langle \text{Max } p \ : 0 \leq p \leq n+1 : \ sum.p.(n+1) \ \rangle \ \ ,$

$sum.p.q = \langle \sum i \ : p \leq i < q : \ A.i \ \rangle]\!]$

Ahora bien, para poder asignar $r := r \text{ max } u$, la expresión para u debería formar parte del invariante (pues debe ser válida cada vez que se inicie el ciclo). Sin embargo, no podemos agregar en el invariante la expresión para u tal cual está, puesto que la misma no esta definida para $n = N$. Fortalezcamos el invariante agregando

$P_2 : \ u = \langle \text{Max } p \ : 0 \leq p \leq n : \ sum.p.n \ \rangle.$

Con la definición de P_2 dada, la expresión para u obtenida en el razonamiento anterior resulta ser $P_2(n := n+1)$. Así, la asignación $n, r := n+1, r \text{ max } u$ es correcta (uno puede convencerse recomenzando el cómputo de wp, pero ahora incluyendo P_2, es decir, calculando E tal que

$$(P_0 \wedge P_1 \wedge P_2 \wedge n \neq N) \ \Rightarrow \ wp.(n, r := n+1, E).(P_0 \wedge P_1 \wedge P_2) \ .$$

Pero para poder hacer $r := r \text{ max } u$ es necesario haber calculado antes u (más precisamente, necesitamos calcular $P_2(n := n+1)$). Por otra parte, también es necesario inicializar u de manera de satisfacer el invariante $P_0 \wedge P_1 \wedge P_2$. La inicialización adecuada es $u := 0$. Así, tenemos:

$$
\begin{array}{l}
[\![\text{con } N : Int,\ A : array[0, N)\ of\ Int \\
\ \text{var } r, u : Int \\
\ \{N \geq 0\} \\
\ n, r, u := 0, 0, 0 \\
\ \{P :\ P_0 \wedge P_1 \wedge P_2\} \\
\ \underline{\textbf{do}}\ n \neq N \rightarrow S_0; \\
\qquad\qquad \{P :\ P_0 \wedge P_1 \wedge P_2(n := n+1)\} \\
\qquad\qquad n, r := n+1, r \text{ max } u \\
\ \underline{\textbf{od}} \\
\ \{r = \langle \text{Max } p, q\ : 0 \leq p \leq q \leq N :\ sum.p.q \rangle \\
\qquad\qquad [\![sum.p.q = \langle \sum i\ : p \leq i < q :\ A.i \rangle]\!] \\
]\!]
\end{array}
$$

donde S_0 consistirá en calcular u. Notemos que luego de ejecutar S_0, debe satisfacerse $P_2(n := n+1)$, que es precondición necesaria para la asignación $r := r \text{ max } u$. Buscamos, entonces, una expresión E tal que

$$(P_0 \wedge P_1 \wedge P_2 \wedge n \neq N) \ \Rightarrow\ wp.(u := E).(P_2(n := n+1))\ .$$

$$
\begin{array}{ll}
 & wp.(u := E).(P_2(n := n+1)) \\
\equiv & \{\text{ definición de } wp \ \} \\
 & (P_2(n := n+1))(u := E) \\
\equiv & \{\text{ sustitución } \} \\
 & E = \langle \text{Max } p\ : 0 \leq p \leq n+1 :\ sum.p.(n+1) \rangle \\
\equiv & \{\text{ partición de rango } \} \\
 & E = \langle \text{Max } p\ : 0 \leq p \leq n :\ sum.p.(n+1) \rangle \\
 & \qquad max\ \langle \text{Max } p\ : 0 \leq p = n+1 :\ sum.p.(n+1) \rangle \\
\equiv & \{\text{ rango unitario } \} \\
 & E = \langle \text{Max } p\ : 0 \leq p \leq n :\ sum.p.(n+1) \rangle \text{ max } sum.(n+1).(n+1) \\
\equiv & \{\text{ definición de } sum \ \} \\
 & E = \langle \text{Max } p\ : 0 \leq p \leq n :\ sum.p.n + A.n \rangle \text{ max } 0 \\
\equiv & \{\ + \text{ distribuye con respecto a } max, \text{ pues el rango no es vacío } \} \\
 & E = (A.n + \langle \text{Max } p\ : 0 \leq p \leq n :\ sum.p.n \rangle) \text{ max } 0 \\
\equiv & \{\text{ usamos } P_2 \ \} \\
 & E = (A.n + u) \text{ max } 0
\end{array}
$$

Por lo tanto el programa se completa con la asignación $u := (A.n + u) \text{ max } 0$. Finalmente, obtenemos

$$[\![\text{con } N : Int,\ A : array[0, N)\ of\ Int$$

$\|[$ var $r, u : Int$
$\{N \geq 0\}$
$n, r, u := 0, 0, 0$
$\{P_0 \wedge P_1 \wedge P_2\}$
do $n \neq N \rightarrow u := (A.n + u) \max 0;\ n, r := n + 1, r \max u$ **od**
$\{r = \langle \text{Max } p, q \ : 0 \leq p \leq q \leq N : \ sum.p.q \rangle$
$\quad \|[sum.p.q = \langle \sum i \ : p \leq i < q : \ A.i \rangle]\|$
$]\|$

Observación: La cantidad de segmentos no vacíos de A es del orden de N^2 y calcular la suma de los elementos de un segmento requiere, en promedio, una cantidad de operaciones del orden de N. Por lo tanto, calcular la suma de todos los segmentos y luego tomar el máximo requeriría del orden de N^3 operaciones. El programa derivado requiere una cantidad de operaciones del orden de N.

Ejemplo 19.10

Consideremos el problema de tomar la máxima diferencia entre dos elementos de un arreglo (en orden, el primero menos el segundo). Este problema puede especificarse como sigue (nótese la precondición para N).

$\|[$con $N : Int,\ A : array[0, N)\ of\ Int$
var $r : Int$
$\{N \geq 1\}$
S
$\{r = \langle \text{Max } p, q \ : 0 \leq p < q < N : \ A.p - A.q \rangle\}$
$]\|$

Usando la técnica de reemplazo de constantes por variables obtenemos como invariante $P_0 \wedge P_1$, donde

$$P_0 \ : \ r = \langle \text{Max } p, q \ : 0 \leq p < q < n : \ A.p - A.q \rangle$$
$$P_1 \ : \ 0 \leq n \leq N$$

Tomaremos como cota la usual $t.n = N - n$ e incrementaremos n para decrementar la cota. Queda por lo tanto el siguiente esquema de programa

$\|[$con $N : Int,\ A : array[0, N)\ of\ Int$
var $r : Int$
$\{N \geq 1\}$
S_0
$\{P_0 \wedge P_1\}$
do $n \neq N \rightarrow r, n := E, n + 1$ **od**
$\{r = \langle \text{Max } p, q \ : 0 \leq p < q < N : \ A.p - A.q \rangle\}$
$]\|$

Calculemos E de manera tal que mantenga el invariante:

$$
\begin{array}{ll}
& wp.(r, n := E, n+1).P_0 \\
= & \{ \text{ def. } wp \ \} \\
& E = \langle \text{Max } p, q \ : 0 \leq p < q < n+1 : \ A.p - A.q \, \rangle \\
= & \{ \text{ partición de rango } \} \\
& E = \langle \text{Max } p, q \ : 0 \leq p < q < n : \ A.p - A.q \, \rangle \\
& \qquad \text{max } \langle \text{Max } p, q \ : 0 \leq p < q = n : \ A.p - A.q \, \rangle \\
= & \{ \ P_0, \text{ rango unitario } \} \\
& E = r \text{ max } \langle \text{Max } p \ : 0 \leq p < n : \ A.p - A.n \, \rangle
\end{array}
$$

Aquí estamos tentados de fortalecer el invariante con una nueva variable que se mantenga invariantemente igual a la expresión de la derecha, pero el problema es que $A.n$ no está definida cuando $n = N$.

Una alternativa es intentar simplificar la expresión, pero para esto es necesario asumir que el rango no es vacío. Esto puede conseguirse fortaleciendo ligeramente el invariante con el requisito de que n sea estrictamente positiva, lo cual solo complicará ligeramente la inicialización. Continuemos con la derivación:

$$
\begin{array}{ll}
& E = r \text{ max } \langle \text{Max } p \ : 0 \leq p < n : \ A.p - A.n \, \rangle \\
= & \{ \text{ distributiva usando que } 0 < n \ \} \\
& E = r \text{ max } (\langle \text{Max } p \ : 0 \leq p < n : \ A.p \, \rangle - A.n) \\
= & \{ \text{ introducción de } s \ \} \\
& E = r \text{ max } (s - A.n) \\
& \qquad [\![s = \langle \text{Max } p \ : 0 \leq p < n : \ A.p \, \rangle]\!]
\end{array}
$$

Por ahora queda entonces el siguiente programa donde falta encontrar F, el cual mantenga P_2 (dado que s no aparece en P_0 ni en P_1 estos se prservan)

$$
\begin{array}{lcl}
P_0 & : & r = \langle \text{Max } p, q \ : 0 \leq p < q < n : \ A.p - A.q \, \rangle \\
P_1 & : & 0 < n \leq N \\
P_2 & : & s = \langle \text{Max } p \ : 0 \leq p < n : \ A.p \, \rangle
\end{array}
$$

$$
\begin{array}{l}
[\![\text{con } N : Int, \ A : array[0, N) \ of \ Int \\
\ \text{var } r : Int \\
\ \{N \geq 1\} \\
\ S_0 \\
\ \{P_0 \wedge P_1 \wedge P_2\} \\
\ \underline{\textbf{do}} \ n \neq N \rightarrow r, s, n := r \text{ max } (s - A.n), F, n+1 \ \underline{\textbf{od}} \\
\ \{r = \langle \text{Max } p, q \ : 0 \leq p < q < N : \ A.p - A.q \, \rangle\} \\
]\!]
\end{array}
$$

Queda por resolver la inicialización y el cálculo de F. Dado que se asumió que el arreglo no es vacío puede inicializarse n en 1 y r y s de manera adecuada. El cálculo de F es elemental y se deja como ejercicio para el lector, así también como la corrección de la inicialización del ciclo. El programa completo queda entonces como sigue:

$$
\begin{array}{l}
\|[\text{con } N : Int,\ A : array[0, N)\ of\ Int \\
\ \text{var } r : Int \\
\ \{N \geq 1\} \\
\ r, s, n := -\infty, A.0, 1 \\
\ \{P_0 \wedge P_1 \wedge P_2\} \\
\ \underline{\textbf{do}}\ n \neq N \rightarrow r, s, n := r \max (s - A.n), s \max A.n, n + 1\ \underline{\textbf{od}} \\
\ \{r = \langle \text{Max } p, q\ : 0 \leq p < q < N :\ A.p - A.q \rangle\} \\
]\|
\end{array}
$$

19.5 Ejercicios

Ejercicio 19.1
Calcular un programa que, dados dos enteros positivos x e y, devuelva en una variable el mínimo común múltiplo de ambos.

Ayuda: el mínimo común múltiplo de dos enteros positivos se puede especificar por:
$mcm.x.y = \langle \text{Min } n\ : 1 \leq n \wedge n \bmod x = 0 \wedge n \bmod y = 0 :\ n \rangle$.

Ejercicio 19.2
Sea $N \geq 0$.

1. Derivar un programa que calcule el menor entero x que satisface $x^3 + x \geq N$.
2. Derivar un programa que calcule el mayor entero x que satisface $x^3 + x \leq N$.

Ejercicio 19.3
Sea A un arreglo de enteros.

1. Derivar un programa que determine si todos los elementos de A son positivos.
2. Derivar un programa que determine si algún elemento de A es positivo.

Ejercicio 19.4
Derivar un programa que guarde en una variable el máximo elemento de un arreglo de enteros.

Ejercicio 19.5
Derivar un programa que calcule la cantidad de elementos pares de un arreglo de enteros.

Ejercicio 19.6
Derivar un programa para la siguiente especificación:

$$
\begin{array}{l}
[\![\text{con } M : Int,\ A : array[0, M)\ of\ Int \\
\ \text{var } r : Int \\
\ \{M \geq 1\} \\
\ S \\
\ \{r = \langle N\, p, q\ : 0 \leq p < q < M :\ A.p * A.q \geq 0 \,\rangle\} \\
]\!]
\end{array}
$$

Ejercicio 19.7
Derivar un programa para la siguiente especificación:

$$
\begin{array}{l}
[\![\text{con } N : Int,\ A : array[0, N)\ of\ Int \\
\ \text{var } r : Int \\
\ \{N \geq 1\} \\
\ S \\
\ \{r = \big\langle \text{Max}\ p, q\ : 0 \leq p < q < N :\ (A.p - A.q)^2 \,\big\rangle\} \\
]\!]
\end{array}
$$

Ejercicio 19.8
Derivar un programa para la siguiente especificación:

$$
\begin{array}{l}
[\![\text{con } N : Int,\ A : array[0, N)\ of\ Int \\
\ \text{var } r : Int \\
\ \{N \geq 1\} \\
\ S \\
\ \{r \equiv \langle\, \exists\, p, q\ : 0 \leq p < q < N :\ A.p - A.q \leq 8 \,\rangle\} \\
]\!]
\end{array}
$$

Ayuda: escribir la postcondición usando el operador de mínimo.

Capítulo 20

Recursión final y programación imperativa

Still round the corner there may wait
A new road or a secret gate;
And though I oft have passed them by,
A day will come at last when I
Shall take the hidden paths that run
West of the Moon, East of the Sun.

J.R.R.Tolkien:
The Lord of the Rings

En este capítulo mostramos cómo construir un programa imperativo para resolver funciones que responden al esquema de recursión final. Este resultado brinda una interesante conexión entre la programación funcional y la imperativa, por cuanto permite derivar definiciones recursivas finales usando el formalismo naturalmente asociado a un lenguaje funcional y obtener luego un programa imperativo para el cómputo de dichas funciones.

Recordemos que una función recursiva final es de la forma

$$\begin{array}{lcl} H.x & \doteq & (\quad b.x \to f.x \\ & & \Box \ \neg b.x \to H.(g.x) \\ & &) \end{array}$$

donde estamos suponiendo que existe una función de cota t que decrece cada vez que $\neg b.x$, es decir, $t.(g.x) < t.x$.

Supongamos que el problema consiste en determinar $H.X$ para cierto valor X. Observemos el esquema repetitivo de cálculo de una función recursiva final: mientras no ocurre $b.x$, se sigue aplicando la función (variando los argumentos, claro está). En algún momento la condición $b.x$ se verifica, con lo cual el cómputo de la función finaliza simplemente con la asignación $H.x = f.x$. Este es precisamente el esquema que hemos descripto para los ciclos. Por lo tanto, para calcular $H.X$ debemos derivar un ciclo.

La postcondición del problema puede expresarse por medio de una variable que guarda el valor que se desea calcular:

$\{Q: \quad r = H.X\}$

En primer lugar, debemos elegir el invariante. Ya hemos dicho que este es un punto clave en la derivación de un ciclo. La táctica en este caso consiste en elegir

$$\{P: \quad H.x = H.X\}$$

La validez del invariante al comienzo del ciclo se establece con la inicialización obvia $x := X$.

Antes de determinar la guarda, observemos nuevamente que la evaluación de H finalizará cuando ocurra $b.x$ (para el x que haya quedado determinado), en cuyo caso el valor asignado a $H.x$ será $f.x$. Pero como el invariante debe mantenerse, debe valer $H.x = H.X$. Además, al finalizar el cómputo debe verificarse también $r = H.X$. Por lo tanto, al finalizar el programa debería satisfacerse $r = f.x$. Esto nos induce a escribir como última instrucción del programa la asignación a la variable r del valor $f.x$:

$[\![$var x : [*tipo adecuado*]
var r : [*tipo adecuado*]
$\{R\}$
$x := X$
$\{P: \quad H.x = H.X\}$
S
$\{f.x = H.X\}$
$r := f.x$
$\{Q: \quad r = H.X\}$
$]\!]$

Ahora debemos derivar S con precondición P (invariante) y postcondición $\{f.x = H.X\}$. Como $P \wedge b.x \Rightarrow f.x = H.X$, la guarda debe ser $\neg b.x$. Y para mantener el invariante, lo que debemos hacer en caso de satisfacerse la guarda es asignarle a x el valor $g.x$. Así, obtenemos:

$[\![$[var x : [*tipo adecuado*]
 var r : [*tipo adecuado*]
 $\{R\}$
 $x := X$
 $\{P : \ H.x = H.X\}$
 do $\neg b.x \to x := g.x$ **od**
 $\{f.x = H.X\}$
 $r := f.x$
 $\{Q : \ r = H.X\}$
$]\!]$

Para garantizar terminación, debemos encontrar una función entera t, acotada inferiormente, que decrezca en cada paso de la aplicación de H, es decir, una función t tal que

$\neg b.x \ \Rightarrow \ t.(g.x) < t.x$ y
$\neg b.x \ \Rightarrow \ t.x > 0$

Si la función H está bien definida (termina), esta función necesariamente tiene que existir.

Ejercicio: Demostrar, justificando cada paso, que S es **do** $\neg b.x \to x := g.x$ **od** .

Ejemplo 20.1
Obtener un programa que calcule la suma de los dígitos de la representación en base 10 de un número natural.

La función

$$\boxed{f.x \ \doteq \ (\ x = 0 \to 0 \ \square \ x \neq 0 \to x \ \mathsf{mod}\ 10 + f.(x \ \mathsf{div}\ 10)\ )}$$

devuelve, para cada natural x representado en base 10, la suma de sus dígitos. Pero f no es recursiva final, por lo que primero debemos transformarla en una función de este tipo.

Sea $H.y.x = y + f.x$, donde y y x son naturales. H es una generalización de f, pues $H.0.x = f.x$. Busquemos una definición recursiva final para H. Como H involucra a f, estudiemos por separado los casos $x = 0$ y $x \neq 0$:

Caso $(x = 0)$

$$
\begin{array}{ll}
 & H.y.x \\
= & \{ \text{ especificación de } H \ \} \\
 & y + f.x \\
= & \{ \text{ definición de } f \ \} \\
 & y + 0 \\
= & \{ \text{ aritmética } \} \\
 & y
\end{array}
$$

Caso $(x \neq 0)$

$$
\begin{array}{ll}
 & H.y.x \\
= & \{ \text{ especificación de } H \ \} \\
 & y + f.x \\
= & \{ \text{ definición de } f \ \} \\
 & y + (x \bmod 10 + f.(x \operatorname{div} 10)) \\
= & \{ \text{ asociatividad de } + \ \} \\
 & (y + x \bmod 10) + f.(x \operatorname{div} 10) \\
= & \{ \text{ hipótesis inductiva } \} \\
 & H.(y + x \bmod 10).(x \operatorname{div} 10)
\end{array}
$$

Por lo tanto, la definición recursiva final de H es

$$
\boxed{H.y.x} \ \doteq \ \begin{array}{l} (\quad x = 0 \to y \\ \Box \ x \neq 0 \to H.(y + x \bmod 10).(x \operatorname{div} 10) \\ ) \end{array}
$$

Si vemos que es posible definir una cota, podremos aplicar directamente el programa anterior. En este caso, la función $g.x$ que figura en la expresión general de función recursiva final es $x \operatorname{div} 10$. Como se cumple

$$
\begin{array}{l}
x \neq 0 \ \Rightarrow \ x \operatorname{div} 10 < x \quad \text{y} \\
x \neq 0 \ \Rightarrow \ x > 0 \ ,
\end{array}
$$

podemos usar el programa para calcular H. Entonces, si lo que deseamos es calcular $f.X$, evaluamos $H.0.X$ simplemente sustituyendo en el programa derivado anteriormente:

$$
\begin{array}{l}
\| [\text{var } x, y : Nat \\
\ y, x := 0, X \\
\ \{P : \ \ H.y.x = H.0.X\} \\
\ \underline{\textbf{do}} \ x \neq 0 \to y, x := y + x \bmod 10, x \operatorname{div} 10 \ \underline{\textbf{od}} \\
\ \{y = H.0.X\} \\
\ r := y \\
\ \{Q : \ \ r = H.0.X\} \\
] \|
\end{array}
$$

20.1 Programas imperativos sobre listas

En esta sección mostraremos cómo calcular un programa imperativo a partir de un programa funcional recursivo final, algunos de cuyos argumentos son listas.

En general, los lenguajes imperativos no proveen un manejo adecuado de listas. Existen sin embargo diversas maneras de implementar programas que usan listas en un lenguaje imperativo. Se presentará uno de los más usados, el cual consiste en la implementación de listas usando arreglos. Este método cuenta entre sus ventajas la eficiencia (en general las operaciones sobre arreglos son implementadas muy eficientemente por la mayoría de los lenguajes imperativos), la portabilidad (los arreglos están presentes en casi todos los lenguajes de programación, Fortran incluido) y sobre todo la claridad. Una desventaja que no nos afectará en los ejemplos a tratar, es que debe proveerse a priori una cota para el número máximo de elementos de un arreglo.

Ejemplo 20.2 (Problema del segmento de suma mínima)
Recordemos que la función h definida abajo se usó para calcular con costo lineal la suma del segmento de suma mínima de una lista (sección 13.3).

$$\begin{array}{|rcl} h.[\,] & \doteq & (0,0) \\ h.(x \triangleright xs) & \doteq & ((x+b) \text{ min } a,\ 0 \text{ min } (x+b)) \\ & & \quad \llbracket (a,b) = h.xs \rrbracket \end{array}$$

Usando el teorema de recursión final para listas, obtuvimos la siguiente definición (sección 15.3):

$$h.xs = G.(0,0).xs$$

donde G es recursiva final:

$$\begin{array}{|l} G.(a,b).xs \doteq (\ \ xs = [\,] \to (a,b) \\ \qquad\qquad \Box\ xs \neq [\,] \to G.((xs.0+b) \text{ min } a,\ 0 \text{ min } (xs.0+b)).(xs{\downarrow}1) \\ \qquad\qquad) \end{array}$$

Primero supondremos que el lenguaje imperativo posee primitivas que permiten el manejo de listas. De esta manera, la implementación es inmediata usando el teorema que nos permite construir programas imperativos a partir de funciones recursivas finales.

En este caso el invariante será

$$P: \quad G.(a,b).ys = G.(0,0).xs \ .$$

El programa se calcula fácilmente como sigue:

$$
\begin{array}{l}
[\![\text{var } ys : [Int] \\
\ \text{var } a, b, r : Int \\
\ \{R\} \\
\ a, b, ys := 0, 0, xs \\
\ \{P : \ \ G.(a,b).ys = G.(0,0).xs\} \\
\ \underline{\textbf{do}}\ ys \neq [\,] \rightarrow \\
\quad a, b, ys := (ys.0 + b) \text{ min } a,\ 0 \text{ min } (ys.0 + b), ys{\downarrow}1 \\
\ \underline{\textbf{od}} \\
\ \{(a,b) = G.(0,0).xs\} \\
\ r := a \\
\ \{Q : \ \ a = f.xs\} \\
]\!]
\end{array}
$$

Dado que en la mayoría de los lenguajes de programación imperativos no se dispone de un manejo de listas tan simple, mostramos ahora cómo pueden implementarse programas sobre listas usando arreglos. Para esto volveremos a usar el concepto de función de abstracción. Esta función se usará para definir la manera en que un arreglo (y una o dos variables enteras) representarán una lista dada. Las operaciones sobre listas serán entonces implementadas como funciones en la representación, que conmuten con la función de abstracción de la manera en que se detallará a continuación. Esta técnica puede en principio usarse para cualquier tipo de datos, no necesariamente para listas. Es por esto que en la definición general no hace falta referirse explícitamente a estas.

Dada una definición recursiva final $H : A \mapsto B$,

$$
\begin{array}{rcl}
H.x & \doteq & (\quad b.x \rightarrow f.x \\
 & & \Box\ \neg b.x \rightarrow H.(g.x) \\
 & &)
\end{array}
$$

consideramos una función de abstracción (ver capítulo 14)

$$[\![\ ]\!] : A' \mapsto A$$

y funciones

$$
\begin{array}{l}
\underline{b} : A' \mapsto Bool \\
\underline{f} : A' \mapsto B \\
\underline{g} : A' \mapsto A'
\end{array}
$$

que satisfacen, para $y \in A'$,

$$
\begin{array}{rcl}
\underline{b}.y & \equiv & b.[\![y]\!] \\
\underline{f}.y & = & f.[\![y]\!] \\
[\![\underline{g}.y]\!] & = & g.[\![y]\!]
\end{array}
$$

Se necesita también una constante X' (el valor inicial) que represente al valor inicial abstracto, esto es $[\![X']\!] = X$, siendo X el valor en el que se desea evaluar la función H. Entonces, el cálculo de H puede implementarse como un programa imperativo como sigue:

$$
\begin{array}{l}
[\![\text{var } y : A' \\
\quad y := X' \\
\quad \{P : \ \ H.[\![y]\!] = H.X\} \\
\quad \underline{\textbf{do}}\ \neg \underline{b}.y \rightarrow y := \underline{g}.y\ \underline{\textbf{od}} \\
\quad \{\underline{f}.y = H.X\} \\
\quad r := \underline{f}.y \\
\quad \{Q : \ \ r = H.X\} \\
]\!]
\end{array}
$$

Volvamos al ejemplo del segmento de suma mínima. Hemos dicho que la implementación de las listas se hará a través del uso de arreglos. Definimos la siguiente función de abstracción:

$$[\![(A, i)]\!] = [A.i,\ A.(i+1)\,,\ \cdots\,,\ A.(N-1)]$$

donde A es un arreglo. Por ejemplo, Si A es el *arreglo* [2,4,6,8,10], entonces $[\![(A, 2)]\!]$ es la *lista* [6,8,10].

Si ys es la lista $[\![(A, i)]\!]$, tenemos las siguientes equivalencias:

$$
\begin{array}{rcl}
(ys = [\,]) & \equiv & (i \geq N) \\
ys.0 & = & A.i \\
(ys := ys{\downarrow}1) & \equiv & (i := i + 1)
\end{array}
$$

Escribamos ahora el programa imperativo para el problema del segmento de suma mínima usando arreglos:

$$
\begin{array}{l}
[\![\text{con } N = \#xs : Nat \\
\quad \text{var } a, b, i : Int \\
\quad \text{var } A : array[0..N)\ of\ Int \\
\quad \{[\![(A, 0)]\!] = xs\} \\
\quad a, b, i := 0, 0, 0 \\
\quad \{P : \ \ G.(a, b).[\![(A, i)]\!] = G.(0, 0).xs\} \\
\quad \underline{\textbf{do}}\ i \neq N \rightarrow \\
\qquad a, b, i := (A.i + b) \text{ min } a,\ 0 \text{ min } (A.i + b), i + 1 \\
\quad \underline{\textbf{od}} \\
\quad \{(a, b) = G.(0, 0).xs\} \\
\quad r := a \\
\quad \{Q : \ \ a = f.xs\} \\
]\!]
\end{array}
$$

Observar que la guarda usada en el ciclo, $i \neq N$, es equivalente a la desigualdad $i < N$, puesto que se comienza el programa con la asignación $i := 0$ y en el cuerpo del ciclo se incrementa el valor de esta variable en 1.

Resulta instructivo comparar este programa con el que se obtuvo en la sección 19.2 al estudiar el problema del segmento de suma máxima, el cual es completamente análogo al problema resuelto más aquí.

Ejemplo 20.3 (El problema de los paréntesis equilibrados)
En la sección 13.2 derivamos la siguiente definición recursiva para la función que permite resolver el problema:

$$
\begin{array}{|l}
g : Num \mapsto [Char] \mapsto Bool \\
\hline
\qquad\quad g.k.[\,] \;\doteq\; (k = 0) \\
g.k.(\text{`(`} \triangleright xs) \;\doteq\; k \geq 0 \;\wedge\; g.(k+1).xs \\
g.k.(\text{`)`} \triangleright xs) \;\doteq\; k \geq 0 \;\wedge\; g.(k-1).xs
\end{array}
$$

A partir de la misma, se puede derivar la siguiente función recursiva final:

$$
\begin{array}{|l}
G : Bool \mapsto Num \mapsto [Char] \mapsto Bool \\
\hline
\qquad\quad G.b.k.[\,] \;\doteq\; b \wedge k = 0 \\
G.b.k.(\text{`(`} \triangleright xs) \;\doteq\; G.(k \geq 0).(k+1).xs \\
G.b.k.(\text{`)`} \triangleright xs) \;\doteq\; G.(k \geq 0).(k-1).xs
\end{array}
$$

Ejercicio: Derivar la función recursiva final anterior.

Escribamos ahora un programa imperativo para el cómputo de esta función usando el modelo del comienzo del capítulo y suponiendo que el lenguaje imperativo permite manejar listas.

La función $H.X$ del modelo es $G.True.0.XS$, por lo cual el invariante es $\{G.b.k.xs = G.True.0.XS\}$. Así, tenemos

$$
\begin{array}{l}
[\![\text{var } b : Bool,\ k : Int,\ xs : String \\
\quad b, k, xs := True, 0, XS \\
\quad \{G.b.k.xs = G.True.0.XS\} \\
\quad \underline{\textbf{do}}\ xs \neq [\,] \\
\qquad \underline{\textbf{if}}\ xs.0 = \text{`(`} \rightarrow b, k, xs := b \wedge k \geq 0, k+1,\ xs{\downarrow}1 \\
\qquad\quad\ \ xs.0 = \text{`)`} \rightarrow b, k, xs := b \wedge k \geq 0, k-1,\ xs{\downarrow}1 \\
\qquad \underline{\textbf{fi}} \\
\quad \underline{\textbf{od}} \\
\quad \{xs = [\,] \wedge G.b.k.xs = G.True.0.XS\} \\
\quad r := b \wedge k = 0 \\
\quad \{r = G.True.0.XS\} \\
]\!]
\end{array}
$$

Ejercicio: Probar que el valor asignado a r es equivalente a $pareq.XS$ (definición de $pareq$ en sección 13.2).

Implementando las listas por medio de arreglos a través de la función de abstracción que definimos antes,

$$[\![(A, i)]\!] = [A.i,\ A.(i+1)\ ,\ \cdots\ ,\ A.(N-1)]$$

podemos escribir el programa en lenguaje imperativo con arreglos:

$[\![$con $N = \#XS : Nat$
var $b : Bool,\ k, i : Int$
var $A : array[0..N)\ of\ Char$
$\{[\![(A, 0)]\!] = XS\}$
$b, k, i := True, 0, 0$
$\{P :\ G.b.k.[\![(A, i)]\!] = G.True\ .0.XS\}$
do $i \neq N$
  if $A.i = \text{`(`} \rightarrow b, k, i := b \wedge k \geq 0, k+1,\ i+1$
    $A.i = \text{`)`} \rightarrow b, k, i := b \wedge k \geq 0, k-1,\ i+1$
  fi
od
$\{(i = N\ \wedge\ G.b.k.[\![(A, i)]\!] = G.True\ .0.XS\}$
$r := b \wedge k = 0$
$\{r = G.True\ .0.XS\}$
$]\!]$

20.2 Ejercicios

Ejercicio 20.1
Escribir programas imperativos para cada uno de los siguientes problemas, cuya derivación se realizó en el capítulo 12. En un primer paso, suponer que el lenguaje dispone de primitivas para el manejo de listas; luego implementar las listas usando arreglos.

1. Evaluación de un polinomio.
2. Cálculo de una aproximación de la constante matemática e.
3. Problema de la lista balanceada.
4. Problema del máximo segmento balanceado.

Apéndice A

Operadores booleanos

La siguiente tabla, extraída de [Kal90], resume algunas de las propiedades importantes que involucran a los operadores booleanos definidos en el capítulo 5. En todos los casos, P, Q y R son predicados.

Idempotencia:	$P \wedge P \equiv P$
	$P \vee P \equiv P$
Conmutatividad:	$P \wedge Q \equiv Q \wedge P$
	$P \vee Q \equiv Q \vee P$
	$(P \equiv Q) \equiv (Q \equiv P)$
Asociatividad:	$(P \wedge Q) \wedge R \equiv P \wedge (Q \wedge R)$
	$(P \vee Q) \vee R \equiv P \vee (Q \vee R)$
	$(P \equiv Q) \equiv R \equiv P \equiv (Q \equiv R)$
Distributividad:	$P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge R)$
	$P \vee (Q \wedge R) \equiv (P \vee Q) \wedge (P \vee R)$
	$P \vee (Q \equiv R) \equiv P \vee Q \equiv P \vee R$
Absorción:	$P \wedge (P \vee Q) \equiv P$
	$P \vee (P \wedge Q) \equiv P$
Reglas verdadero-falso:	$P \wedge True \equiv P$
	$P \wedge False \equiv False$
	$P \vee True \equiv True$
	$P \vee False \equiv P$
Leyes de Morgan:	$\neg(P \wedge Q) \equiv \neg P \vee \neg Q$
	$\neg(P \vee Q) \equiv \neg P \wedge \neg Q$

Equivalencia:

$$P \equiv P$$
$$P \equiv P \equiv True$$
$$\neg P \equiv P \equiv False$$

Implicación:

$$P \Rightarrow Q \equiv \neg P \vee Q$$
$$P \Rightarrow Q \equiv P \wedge Q \equiv P$$
$$P \Rightarrow Q \equiv P \vee Q \equiv Q$$
$$P \Rightarrow P \vee Q$$
$$P \wedge Q \Rightarrow P$$
$$False \Rightarrow P$$
$$P \Rightarrow True$$
$$True \Rightarrow P \equiv P$$
$$P \Rightarrow False \equiv \neg P$$

Negación:

$$\neg\neg P \equiv P$$
$$P \wedge \neg P \equiv False$$
$$P \vee \neg P \equiv True$$
$$\neg(P \equiv Q) \equiv \neg P \equiv Q$$

Apéndice B

Sobre las implementaciones

El lenguaje de comandos protegidos o guardados (guarded commands) de Dijkstra, el cual hemos usado a lo largo de este curso, es una de las notaciones más adecuadas para expresar programas imperativos. Esto se debe esencialmente a su claridad y a tener una semántica bien definida y simple.

Pese a ser suficientemente cercano a los lenguajes de programación imperativos usuales (e.g. Pascal, C), cuando se quiere implementar un algoritmo escrito en esta notación hace falta tener algo de cuidado de preservar la corrección y la claridad de la solución. Los siguientes consejos pueden ser de utilidad.

1. En los lenguajes de programación suele haber más de una instrucción para implementar repeticiones. La supuesta "flexibilidad" que esto permite es en realidad fuente de confusiones y errores. Por ejemplo en Pascal se tienen tres variantes: el **while** equivalente a nuestro **do** con una sola guarda (lo cual sabemos que no es una restricción a la expresividad), el **repeat** que evalúa la guarda al final del ciclo y el **for** que decrementa automáticamente una variable que funciona como función de cota. Aconsejamos usar sólo el **while**, dado que el **repeat** es un caso particular de éste (el cual no es particularmente interesante) y el **for** se presta a toda clase de confusiones: nunca se sabe si la variable de cota se decrementa al comenzar o al terminar el ciclo y por lo tanto cual es la condición de terminación de éste.

2. Si bien en los algoritmos desarrollados en el apunte suponemos siempre que los datos de entrada satisfacen la precondición, no es mala idea controlar eso cuando no es demasiado caro hacerlo. Esto hace que los programas sean más *robustos*. Por ejemplo, en el algoritmo de la división se supone que el primer argumento es mayor o igual a cero y el segundo mayor estricto que cero. Un programa robusto controlaría esto al iniciarse y terminaría con un mensaje de error si dicha condición no se satisficiera.

3. La claridad de un programa se ve significativamente aumentada si se escriben como comentarios en el texto del mismo las aserciones más relevantes. Se esperaría que estuvieran escritas por lo menos las pre y post condiciones de cada función, así también como los invariantes de los ciclos. Con respecto a esto último, insistimos en que si no se puede expresar el invariante de un ciclo es que no se ha comprendido bien lo que éste hace.

4. Las instrucciones de repetición y alternativa disponibles en los lenguajes de programación imperativos son en general deterministas, i.e. sólo se permiten guardas disjuntas. Para implementar entonces una alternativa o repetición no-determinista es necesario fortalecer previamente las guardas. En general, la alternativa suele tener una guarda **else**, la cual se entiende como la negación de todas las otras. Sugerimos indicar como comentario cual es la guarda que le corresponde para evitar confusiones. En los casos de instrucciones de alternativa anidadas esto es aún más recomendable, dado que muy rápidamente se pierde de vista de que otras guardas es negación un **else**.

   Por ejemplo, el siguiente programa

   $$\begin{array}{ll} \underline{\textbf{if}} & y \geq x \rightarrow z := x \\ \Box & x \geq y \rightarrow z := y \\ \underline{\textbf{fi}} & \end{array}$$

   se transforma primero en el siguiente, al cual se le ha fortalecido una de las guardas (preservando obviamente la corrección).

   $$\begin{array}{ll} \underline{\textbf{if}} & y \geq x \rightarrow z := x \\ \Box & x < y \rightarrow z := y \\ \underline{\textbf{fi}} & \end{array}$$

   luego de lo cual se lo traduce por ejemplo a Pascal escribiendo la guarda del **else** como comentario. Agregamos también la del **then** para mayor claridad, aunque esto es menos necesario dada la proxmidad a la guarda (en una composición alternativa más compleja la guarda asociada al **else** puede estar decenas de renglones más arriba).

   $$\begin{array}{lll} \underline{\textbf{if}} & y \geq x & \\ & \underline{\textbf{then}} & z := x \\ & \underline{\textbf{else}}\{\mathrm{x} < \mathrm{y}\} & z := y \\ \underline{\textbf{fi}} & & \end{array}$$

5. En la mayoría de los lenguajes de progrmación no se dispone de asignación múltiple. En esos casos, ésta puede implementarse usando una secuencia de asignaciones simples. A veces es necesario usar variables auxiliares, siendo

recomendable entonces demostrar que la secuencia de asignaciones tiene la misma semántica que la asignación múltiple, dado que un error de este tipo es en general difícil de encontrar. Por ejemplo la asignación múltiple

$$S_0 : \quad x, y := 2 * y, x + y$$

puede ser implementada por el programa

$$\begin{array}{rl} S_1 : & x_0 := x \\ ; & x := 2 * y \\ ; & y := x_0 + y \end{array}$$

pero no por

$$\begin{array}{rl} S_2 : & x := 2 * y \\ ; & y := x + y \end{array}$$

Ejercicio: Demostrar que S_0 es equivalente (cuando no se considera el valor de x_0) a S_1 pero no a S_2

6. Existen opiniones divididas acerca de la utilidad de usar nombres mnemotécnicos para las variables. En este trabajo hemos preferido usar nombres cortos, lo cual vuelve más cortos los cálculos. A veces un nombre mnemotécnico en un programa ayuda a reconocer de que variable se esta hablando pero si no está definido con precisión puede confundir más que ayudar (¿qué significa por ejemplo la variable "prod"?). En el caso en que sí esté definido precisamente, la mnemotecnia se vuelve innecesaria. Aparentemente, los nombres mnemotécnicos aparecieron en los programas "artesanales" pero se vuelven superfluos cuando el programa es derivado formalmente.

Bibliografía

[Bac03] Roland Backhouse. *Program Construction: Calculating Implementations from Specifications*. John Wiley & Sons, Inc., New York, NY, USA, 2003.

[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.

[BS99] Javier O. Blanco and Silvina Smith. *Cálculo de Programas*. Facultad de Matemática Astronomía y Física (UNC), 1999.

[BvW08] Ralph-Johan Back and Joakim von Wright. *Mathematics with a little bit of logic: Structured derivation in high-school mathematics*. Draft, 2008.

[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.

[Fei90] W. H. Feijen. Exercises in formula manipulation. *Formal development programs and proofs*, pages 139–158, 1990.

[Fok96] M. Fokkinga. *Werkcollege Functioneel Programmeren.* Universidad de Enschede, 1996.

[Gib94] W. Wayt Gibbs. Software's chronic crisis. *Scientific American*, page 86, September 1994.

[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.

[Has06] Haskell home page. `http://www.haskell.org`, 2006.

[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.

[Knu69] D.E. Knuth. *The Art of Computer Programming, Volume 2, Seminumerical Algorithms.* Addison-Wesley, 1969.

[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.

[vdS93] Jan L.A. van de Snepscheut. *What Computing Is All About.* Springer Verlag, 1993.

[Wik08] Wikipedia. String searching algorithm — wikipedia, the free encyclopedia, 2008. [Online; accessed 11-March-2008].

Sobre los autores

Damián Barsotti

El primer contacto que tuvo con la informática fue a principios de los '80 a través de una TI-99 que trajo su tío de un viaje por Estados Unidos. A la edad de 15 años toma algunos cursos de programación en Basic y COBOL. Después de terminar la escuela secundaria ingresa a la carrera de Licenciatura en Física en FaMAF. Permanece allí por cuatro años pero sin abandonar su interés por la programación. En el año 1993 se abre la carrera de Informática en la misma facultad y decide cambiar de rumbo académico siguiendo su verdadera vocación. Se recibe junto con la primera promoción de licenciados en Informática del FaMAF en el año 1999. Desde ese año hasta la fecha, cumple tareas docentes en cursos iniciales de programación junto a su amigo Javier. En base a esta experiencia pudo escribir este libro. Actualmente se dedica además a la investigación en el área de la programación concurrente, dentro de la cual está realizando su trabajo de doctorado.

Javier Blanco

Estudió informática en la ESLAI (Escuela Superior Latinoamericana de Informática) hasta 1991 y se doctoró luego en la Universidad de Eindhoven, Holanda, en 1996. Trabaja desde 1997 como profesor en Famaf, en la Universidad Nacional de Córdoba. Sus intereses académicos fueron variando: primero fueron la teoría de tipos y teoría de categorías, luego las semánticas algebraicas de la concurrencia y ahora la construcción formal de programas, especialmente programas concurrentes y programas que manejan dinámicamente la memoria. También trabaja en filosofía y computación y en educación de informática. Aprovecha cada ocasión para la puesta en cuestión del lugar de la ciencia en la sociedad actual, tanto en discusiones académicas como en proyectos concretos.

Silvina Smith

Se recibió de Licenciada en Matemática en la Facultad de Matemática, Astronomía y Física (FaMAF) de la Universidad Nacional de Córdoba en 1989. Hizo un postgrado en matemática aplicada a la industria en la Johannes Kepler Universität Linz, Austria, 1994. Desarrolló su trabajo de tesis en el departamento Polyolefins and Elastomers, Research and Development de la compañía química Dow Benelux, Terneuzen, Holanda. Comenzó a vincularse con la temática de este libro en 1998 y desde entonces se ha mantenido relacionada con ella en el aspecto docente. En la actualidad, trabaja en educación en matemática. Es docente en FaMAF y en el Instituto Universitario Aeronáutico (IUA), Córdoba.

Se terminó de imprimir
en el mes de mayo de 2008
en el Taller General de Imprenta
de la UNC, Secretaría General.
Córdoba–Argentina

Contratapa

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. La presencia de errores es en estos casos más la regla que la excepción.

Existe, afortunadamente, otra forma de aproximarse a la programación. Los programas pueden ser desarrollados de manera metódica a partir de *especificaciones*, construyendo a la vez el programa y su demostración, con esta última guiando el proceso. Además de su utilidad práctica, este ejercicio intelectual es altamente instructivo y vuelve a la programación una tarea creativa e interesante.

En este punto la lógica matemática es una herramienta indispensable como ayuda para la especificación y desarrollo de programas. Los programas son fórmulas lógicas tan complejas y voluminosas que fue difícil reconocerlos como tales. Este libro comienza presentando un estilo de lógica orientado a trabajar de manera más adecuada con fórmulas de gran tamaño. A partir de su estudio se mostrará cómo desarrollar programas funcionales e imperativos utilizando este formalismo de manera unificada.