Capítulo 4

Inducción

Un tipo común de problema es demostrar que un objeto *x* tiene alguna propiedad *P.* La notación matemática para esto es *P*(*x*), donde *P* significa predicado (o propiedad). Por ejemplo, si *x* es 6 y *P*(*x*) es el predicado '*x* es un número par', entonces podríamos expresar el enunciado '6 es un número par' con el enunciado matemático abreviado *P*(6).

Muchas de las aplicaciones de la informática nos obligan a demostrar que *todos* los elementos de un conjunto *S* tienen una propiedad determinada *P.* La declaración '*todo* elemento *x* en el conjunto *S* que tiene el predicado *P* para *x*' se puede escribir como:

$$\forall x \in S \,.\, P(x)$$

Se pueden usar declaraciones como esta para afirmar propiedades de datos. Por ejemplo, podríamos afirmar que cada elemento de una base de datos tiene una determinada propiedad. También se pueden usar para describir el comportamiento de los programas de computadora. Por ejemplo, *P*(*n*) podría ser una declaración de que un ciclo calcula el resultado correcto si termina después de *n* iteraciones, y la declaración $\forall n \in N \,.\, P(n)$ dice que el ciclo es correcto si termina después de *cualquier* número de iteraciones. *N* denota el conjunto de números naturales, por lo que $n \in N$ solo dice que *n* es un número natural, y la expresión completa $\forall n \in N.\ P(n)$ dice que para cualquier número natural *n*, el predicado *P*(*n)* se cumple.

Un enfoque para probar una afirmación sobre todos los elementos de un conjunto es escribir una prueba directa separada para cada elemento del conjunto. Eso estaría bien si el conjunto fuera pequeño. Por ejemplo, para demostrar que todos los elementos del conjunto *{*4*,*6*}* son pares, podría probar que 4 es par y también que 6 es par. Sin embargo, este enfoque se vuelve rápidamente tedioso para conjuntos grandes: para demostrar que todos los elementos de *{*2*,*4*,*6*, ...,*1000*}* son pares, ¡necesitaría 500 pruebas separadas! Peor aún, este método de fuerza bruta no funciona en absoluto para conjuntos infinitos porque una demostración no puede ser infinita.

La inducción es un método poderoso para probar que cada elemento de un conjunto tiene una determinada propiedad. La inducción es una técnica valiosa, porque no es tediosa incluso si el conjunto es grande, y funciona incluso para

conjuntos infinitos. La inducción se usa con frecuencia en aplicaciones informáticas. Este capítulo muestra cómo funcionan las pruebas inductivas y ofrece muchos ejemplos, incluyendo aplicaciones tanto matemáticas como informáticas. Utilizaremos dos formas de inducción: inducción matemática para probar propiedades sobre números naturales e inducción estructural para probar propiedades sobre listas.

4.1 El principio de inducción matemática

La inducción se utiliza para demostrar que una propiedad *P(x)* se mantiene para cada elemento de un conjunto *S*. En esta sección, restringimos *S* al conjunto *N* de números naturales; luego generalizamos la inducción para manejar conjuntos más generales. En lugar de probar $P(x)$ por separado para cada x, la inducción se basa en un procedimiento sistemático: sólo hay que probar algunos hechos sobre la propiedad P, y luego un teorema llamado **Principio de inducción matemática** permite concluir $\forall x \in N\,.\,P(x)$.

La idea básica es definir un procedimiento sistemático para probar que *P se* cumple para un elemento arbitrario. Para hacer esto, debemos probar dos afirmaciones:

1. El *caso base* $P(0)$ dice que la propiedad P es válida para el elemento base 0.
2. El *caso inductivo* $P(i) \rightarrow P(i + 1)$ dice que si P es válida para un elemento arbitrario i del conjunto, entonces también debe ser válida para el elemento sucesor i+1. El símbolo → se lee como *implica*.

Debido a que cada elemento del conjunto de números naturales se puede alcanzar comenzando con 0 y sumando repetidamente 1, se puede establecer que *P se* cumple para cualquier elemento en particular, usando una secuencia finita de pasos. Dado que $P(0)$ se cumple (el caso base) y que $P(0) \rightarrow P(1)$ (una instancia del caso inductivo), podemos concluir que $P(1)$ también se cumple. Este es un ejemplo simple de *inferencia lógica*, que se estudiará con más detalle en los capítulos sobre lógica. El mismo procedimiento puede usarse sistemáticamente para probar $P(i)$ para *cualquier* elemento de los números naturales. Por ejemplo, $P(4)$ puede probarse utilizando los siguientes pasos (donde el símbolo $\wedge$ indica *y*):

Conclusión	**Justificación**
P(0)	el caso base
P(1)	$P(0) \rightarrow P(1) \wedge P(0)$
P(2)	$P(1) \rightarrow P(2) \wedge P(1)$
P(3)	$P(2) \rightarrow P(3) \wedge P(2)$
P(4)	$P(3) \rightarrow P(4) \wedge P(3)$

Dado cualquier elemento *k* de *N*, puede probar *P*(*k*) usando esta estrategia, y la prueba será *k*+1 líneas de largo. Todavía no hemos usado el Principio de Inducción Matemática. Acabamos de utilizar el razonamiento lógico ordinario para probar *P(k)* para una k arbitraria. Pero, probar *P*(*k*) para una k arbitraria no es lo mismo que probar $\forall k \in N.\ P(k)$, porque hay un número infinito de valores de *k*, por lo que la prueba sería infinitamente larga. El tamaño de una prueba siempre debe ser finito.

Lo que el Principio de Inducción Matemática nos permite concluir es que *P* es válido para *todos* los elementos de *N*, incluso si hay un número infinito de ellos. Y lo permite con una prueba finita. Por lo tanto, la inducción introduce algo nuevo: no es solo una macro que se expande en una larga prueba.

Teorema 5 (Principio de inducción matemática)**.** Sea *P*(*x*) un predicado que sea verdadero o falso para cada elemento del conjunto *N* de números naturales. Si *P*(0) es verdadero y $P(n) \rightarrow P(n + 1)$ se cumple para todos $n \geq 0$, entonces $\forall x \in N\ .\ P(x)$ es verdadero.

Este teorema se puede probar usando la teoría de conjuntos axiomáticos, que está más allá del alcance de este libro. Para la mayoría de las aplicaciones en ciencias de la computación, es suficiente tener una comprensión intuitiva de lo que dice el teorema y tener una comprensión funcional de cómo usarlo para probar otros teoremas.

La prueba del caso base generalmente resultará ser un cálculo sencillo. La expresión $P(n) \rightarrow P(n + 1)$ significa que 'si *P*(*n*) es verdadero, entonces también lo es *P*(*n* + 1)'. Podemos establecer esto asumiendo temporalmente *P*(*n*) y luego, en el contexto de este supuesto, demostrando *P*(*n* + 1). El supuesto *P*(*n*) se llama *hipótesis inductiva*. ¡Es importante entender que *asumir la hipótesis inductiva no significa que sea verdad!* Todo lo que importa es que *si* la suposición *P*(*n*) nos permite probar *P*(*n*+1), *entonces* la implicación $P(n) \rightarrow P(n + 1)$ es válida. Estudiaremos la inferencia lógica con más detalle en el Capítulo 6.

4.2 Ejemplos de inducción sobre números naturales

Existe una historia tradicional sobre Gauss, uno de los mejores matemáticos de la historia. Un día, en la escuela, la maestra le dijo a la clase que calculara la suma 1 + 2 +···+ 100. Después de pensar un poco, Gauss dio la respuesta correcta, 5050, mucho antes de que hubiera sido posible calcular todas las adiciones. Había notado que esta suma se puede organizar en pares de números, de esta manera:

$$(1 + 100) + (2 + 99) + (3 + 98) + \cdots + (50 + 51)$$

El total de cada par es 101, y hay 50 de los pares, por lo que el resultado es 50 × 101 = 5050.

Métodos como este a menudo se pueden usar para ahorrar tiempo en la informática, por lo que vale la pena encontrar una solución al caso general de este problema, que es la sumatoria

$$\sum_{i=1}^{n} i = 1 + 2 + \cdots + n.$$

Figura 4.1: Interpretación geométrica de la fórmula

Si n es par, obtenemos n / 2 pares que suman n+1, por lo que el resultado es $(n/2)\times(n+1) = (n\times(n + 1)) / 2$. Si n es impar, podemos comenzar desde 0 en lugar de 1; por ejemplo,

$$\sum_{i=0}^{7} i = (0 + 7) + (1 + 6) + (2 + 5) + (3 + 4).$$

En este caso, hay (n+1)/2 pares, cada uno de los cuales suman, por lo que el resultado es nuevamente $(n \times (n + 1)) / 2$. Para cualquier número natural n, terminamos con el resultado

$$\sum_{i=0}^{n} i = (n \times(n+1)) / 2.$$

Esta fórmula es útil porque reduce el tiempo de cálculo requerido. Por ejemplo, si n es 1000, se necesitan 999 adiciones para calcular la suma por la fuerza bruta, pero la fórmula siempre requiere sólo una suma, una multiplicación y una división.

La Figura 4.1 muestra otra forma de entender la fórmula. El rectángulo está cubierto mitad por puntos y mitad por estrellas. El número de estrellas es 1 + 2 + 3 + 4, y el área del rectángulo es 4 × (4 + 1), por lo que el número total de estrellas es (4 × (4 + 1)) / 2.

Hasta ahora solo hemos adivinado la fórmula general para $\sum_{i=0}^{n} i$ y hemos considerado dos formas de entenderlo. El siguiente paso es *demostrar* que esta fórmula siempre da la respuesta correcta, y la inducción proporciona una forma de hacerlo.

Teorema 6.

$$\forall n \in N. \sum_{i=0}^{n} i = (n \times (n + 1)) / 2.$$

Prueba. Defina la propiedad *P de la* siguiente manera:

$$P(n) = (\sum_{i=0}^{n} i = (n \times (n + 1)) / 2).$$

Por lo tanto, el objetivo es demostrar que

$$\forall n \in N \,.\, P(n)$$

y procedemos por inducción en n.

Caso base. Necesitamos probar $P(0)$.

$$\sum_{i=0}^{0} i$$
$$= 0$$
$$= 0 \times (0 + 1).$$

Por lo tanto, hemos establecido la propiedad $P(0)$.

Caso inductivo. El objetivo es demostrar que $P(n) \rightarrow P(n + 1)$, y probaremos esta implicación suponiendo que $P(n)$ (esto se llama *hipótesis inductiva*) y luego usando esa suposición para probar $P(n + 1)$. Comenzamos escribiendo con todo detalle la suposición y el objetivo de la prueba. Suponga para un número natural arbitrario n que

$$\sum_{i=0}^{n} i = (n \times (n + 1)) / 2 \,.$$

El objetivo es mostrar (para este valor particular de n) que

$$\sum_{i=0}^{n+1} i = ((n + 1) \times (n + 2)) / 2 \,.$$

Hacemos esto comenzando con el lado izquierdo de la ecuación y usando álgebra para transformarlo en el lado derecho. El primer paso utiliza la suposición dada anteriormente. Esto se llama *hipótesis inductiva*.

$$\sum_{i=0}^{n+1} i$$
$$= (\sum_{i=0}^{n} i) + (n + 1)$$
$$= (n \times (n + 1)) / 2 + (n + 1)$$
$$= ((n + 1) \times (n + 2)) / 2.$$

Ahora hemos establecido que

$$(\sum_{i=0}^{n} i = (n \times (n + 1)) / 2) \rightarrow (\sum_{i=0}^{n+1} i = ((n + 1) \times (n + 2)) / 2).$$

Es decir,

$$P(n) \rightarrow P(n + 1).$$

Uso del principio de inducción. Para resumir, hemos probado el caso base $P(0)$, y hemos probado el caso inductivo $P(n) \rightarrow P(n + 1)$. Por lo tanto, el principio de inducción nos permite concluir que el teorema $\forall n \in N \,.\, P(n)$ es verdadero.

Ejercicio 1. Sea *un* número real arbitrario. Demuestre, para todos los números naturales *m* y *n*, que $a^{m \times n} = (a^m)^n$.

Ejercicio 2. Demuestre que la suma de los primeros *n* números impares positivos es n^2.

4.3 Inducción y recursión

La inducción es un método común para probar propiedades de funciones definidas recursivamente. La función *factorial*, que se define de forma recursiva, proporciona un buen ejemplo de una prueba de inducción.

factorial :: *Natural* →*natural*

Factorial 0 = 1

factorial (*n* +1) = (*n* + 1) ∗ *factorial n*

Un problema común en informática es demostrar que un programa calcula la respuesta correcta. Tal demostración requiere una especificación matemática abstracta del problema, y tiene que mostrar que para todas las entradas, el programa produce el mismo resultado definido por la especificación. El valor de *n*! (factorial de *n*) se define como el producto 1 a *n*; La notación estándar para esto es usando productoria $\Pi^n_{i=1}i$. El siguiente teorema dice que la definición recursiva anterior de la función *factorial* calcula correctamente el valor *n*!.

Teorema 7. Para todos los números naturales *n*, *factorial n* = $\Pi^n_{i=1}i$.

Prueba. Inducción sobre *n*. El caso base es

factorial 0

= 1	*{factorial.1}*
= $\Pi^0_{i=1}i$	*{def. de* Π *}*

Para el caso inductivo, es necesario demostrar que

$$(factorial\ n = \Pi^n_{i=1} i) \rightarrow (factorial\ (n + 1) = \Pi^{n+1}_{i=1} i).$$

Esto se hace asumiendo el lado izquierdo como la hipótesis inductiva: para un n particular, la hipótesis es *factorial n* = $\Pi^{n}_{i = 1} i$. Dado este supuesto, debemos demostrar que *factorial (n + 1)* = $\Pi^{n + 1}_{i = 1} i$.

factorial (n + 1)
= (n + 1) × factorial n *{factorial.2}*
= (n + 1) × $\Pi^{n}_{i = 1} i$ *{hipótesis}*
= $\Pi^{n + 1}_{i = 1} i$ *{def. Π}*

Ejercicio 4. (Este problema es de [12], donde se pueden encontrar muchos más) El *enésimo* número de Fibonacci se define de la siguiente manera:

```
fib :: Entero -> Entero
fib 0 = 0
fib 1 = 1
fib (n 2) = fib n + fib (n + 1)
```

los primeros números en esta famosa secuencia son *0,1,1,2,3,*5,.... Demuestre lo siguiente:

$$\sum^{n}_{i = 1} \text{fib i} = \text{fib (n + 2)} - 1.$$

4.4 Inducción en Peano Naturals

[No incluido.]

4.5 Inducción en listas

Las listas son una de las estructuras de datos más utilizadas en informática, y existe una gran familia de funciones para manipularlas. Estas funciones se definen típicamente de forma recursiva, con un caso base para listas vacías [] y un caso recursivo para listas no vacías, es decir, listas de la forma (*x*:*xs*). La inducción de listas es el método más común para probar las propiedades de tales funciones.

Antes de continuar, discutimos algunas técnicas prácticas que ayudan a hacer frente a los teoremas. Si no está familiarizado con ellos, las declaraciones matemáticas a veces parecen confusas, y es fácil desarrollar un mal hábito de saltarse la matemática al leer un libro. Aquí hay algunos consejos sobre mejores enfoques. Trataremos de ilustrar los consejos de manera concreta en esta sección, pero estos métodos darán resultado a lo largo de cualquier trabajo en informática, no solo en las demostraciones por inducción.

- Al enfrentarse a un nuevo teorema, intentar comprender lo que significa antes de intentar probarlo. Repetir la idea principal en español.

- Pensar en qué aplicaciones podría tener el teorema. Si dice que dos expresiones son iguales, ¿se puede pensar en situaciones en las que podría haber una ventaja práctica en reemplazar el lado izquierdo por el lado derecho, o viceversa? (Una situación común es que un lado de la ecuación es más natural de escribir y el otro lado es más eficiente).
- Probar el teorema en algunos ejemplos pequeños. Los teoremas a menudo se expresan como ecuaciones; inventar algunos datos de entrada adecuados y evaluar ambos lados de la ecuación para ver si son iguales.
- Verificar lo que sucede en casos límite. Si la ecuación dice algo sobre las listas, ¿qué sucede si la lista está vacía? ¿Qué pasa si la lista es infinita?
- ¿El teorema parece estar relacionado con otros? Los pequeños teoremas sobre las funciones, del tipo que generalmente se prueban por inducción, tienden a ser similares dentro de tipos de funciones. Darse cuenta de estas relaciones ayuda a comprender, recordar y aplicar los resultados.

El principio de la inducción sobre listas establece una técnica para probar las propiedades de las listas. Es similar al principio de inducción matemática. La principal diferencia es que el caso base es generalmente la lista vacía (en lugar de 0) y el caso inductivo generalmente utiliza una lista con un elemento adicional (`x:xs`) en lugar de *n*+1. La inducción sobre listas es un caso especial de una técnica más general llamada *inducción estructural*. La inducción sobre listas se usa para demostrar que una proposición *P*(*xs*) se cumple para toda lista *xs*.

Teorema 14 (Principio de inducción de la lista)**.** Supongamos que *P*(*xs*) es un predicado en listas de tipo [*a*], para algún tipo *a*. Supongase que se demuestra que *P*([]) es verdadero (este es el caso base). Además, supongamos que se demuestra que si *P(xs)* se mantiene para *xs* ::*[a]* arbitrarias*,* entonces *P*(x:*xs)* también es válido para *x* :: *a* arbitrario*.* Entonces *P*(*xs*) se cumple para cada lista *xs* que tiene una longitud finita.

Así, el caso base es demostrar que el predicado se mantiene para la lista vacía, y el caso inductivo es demostrar que *si P* es cierta para una lista *xs,* entonces debe *también* mantener durante cualquier lista de la forma (x:*xs*). Cuando se demuestran la caso base y el caso inductivo, el principio de inducción nos permite concluir que el predicado es válido para todas las listas finitas.

El principio de la inducción de listas no puede usarse para probar teoremas sobre listas infinitas. Este punto se discute en la Sección 4.8.

Ahora trabajaremos a través de una serie de ejemplos donde la inducción se usa para probar teoremas sobre las propiedades de las funciones recursivas sobre las listas.

La función *sum* toma una lista de números y los suma. Definimos sum de una lista vacía como 0, y sum de una lista no vacía se calcula sumando el primer número al sum de todo el resto.

```
sum :: Num a => [a] -> a
sum [] = 0
sum (x: xs) = x + sum xs
```

El siguiente teorema establece un hecho útil sobre la relación entre dos funciones: *sum* y ++. Dice que si hay dos listas, digamos *xs* e *ys*, entonces hay dos formas diferentes de calcular el total combinado de todos los elementos. Se puede concatenar las listas con ++, y luego aplicar *sum*, o se puede aplicar *sum* independientemente a las dos listas y sumar los dos números resultantes.

Los teoremas de este tipo a menudo son útiles para transformar programas y para hacerlos más eficientes. Por ejemplo, suponga que tiene una lista muy larga para sumar y que hay dos computadoras disponibles. Podríamos usar paralelismo para reducir el tiempo de ejecución casi a la mitad. La idea es dividir la larga lista en dos más cortas, que se pueden sumar en paralelo. Una adición rápida será suficiente para obtener el resultado final. Esta técnica es un ejemplo de la estrategia *dividir y conquistar* para mejorar la eficiencia de los algoritmos. Obviamente, la computación paralela y la optimización de programas tienen más de lo que hemos cubierto en este párrafo, pero los teoremas como el que estamos considerando pueden ayudar a mostrar que las soluciones paralelizadas son equivalentes a las que no lo están.

Teorema 15. *sum* (*xs*++*ys*) = *sum xs* + *sum ys*

La prueba de este teorema es una inducción típica sobre las listas, y proporciona un buen modelo a seguir para problemas futuros. Las justificaciones utilizadas en los pasos de prueba se escriben entre llaves *{...}*. Muchas de las justificaciones citan la definición de una función, junto con el número de la ecuación en la definición; así *{* (++)*.*1 *}* significa 'este paso está justificado por la primera ecuación en la definición de ++'. El paso más crucial en una prueba de inducción es aquel en el que se utiliza la hipótesis inductiva; la justificación citada para ese paso es *{* hipótesis inductiva *}*.

Prueba. Inducción sobre *xs*. El caso base es

sum ([] ++ *ys*)
= *sum ys* *{* (++)*.*1 *}*
= 0+ *sum ys* *{* 0 + *x* = *x}*
= *sum* [] + *sum ys* *{suma.*1 *}*

El caso inductivo es

sum ((*x* : *xs*) ++*ys*)
= *sum* (*x* : (*xs*++*ys*)) *{* (++)*.*2 *}*

= *x* + *sum* (*xs*++*ys*) *{ suma.2 }*
= *x* + (*sum xs* + *sum ys*) *{* hipótesis *}*
= (*x* + *sum xs*) + *sum ys* *{* +. es asociativo *}*
= *sum* (*x* : *xs*) + *sum ys* *{suma.*2 *}*

Muchos teoremas describen una relación entre dos funciones; el anterior trata sobre la combinación de *sum* y (++), mientras que este combina *length* con (++). Su prueba se deja como un ejercicio.

Teorema 16. *length* (*xs*++*ys*) = *length xs* + *length ys*

Ahora consideramos varios teoremas que establecen propiedades cruciales de la función de orden superior *map*. Estos teoremas se usan comúnmente en la transformación del programa y la implementación del compilador. También se usan con frecuencia para justificar pasos en pruebas de teoremas más complejos.

El siguiente teorema dice que el *map* 'preserva la longitud': cuando se aplica *map* a una lista, el resultado tiene la misma length que la lista de entrada.

Teorema 17. *length* (*map f xs*) = *length xs*

Prueba. Inducción sobre *xs*. El caso base es

length (*map f* [])
= *length* [] *{map.*1 *}*

Para el caso inductivo, suponga *length* (*map f xs*) = *length xs*. Entonces

length (*map f* (*x* : *xs*))
= *length* (*fx* : *map f xs*) *{map.*2 *}*
= 1+ *length* (*map f xs*) *{length.*2 *}*
= 1+ *length xs* *{* hipótesis *}*
= *length* (*x* : *xs*) *{length.*2 *}*

El siguiente teorema recuerda al Teorema 15; dice que puede obtener el mismo resultado con cualquiera de los dos métodos: (1) mapear una función en dos listas y luego sumar los resultados, y (2) agregar las listas de entrada y luego realizar un map más largo sobre el resultado. Su prueba es otro buen ejemplo de inducción, y se deja como ejercicio.

Teorema 18. *map f* (*xs*++*ys*) = *map f xs* ++ *map f ys*

Una de las propiedades más importantes del map se expresa precisamente por el siguiente teorema. Suponga que tiene dos cálculos que realizar en todos los elementos de una lista. Primero desea aplicar *g* a un elemento, obteniendo un

resultado intermedio al que desea aplicar *f*. Hay dos métodos para hacer el cálculo. El primer método usa dos bucles separados, uno para realizar *g* en cada elemento y el segundo bucle para realizar *f* en la lista de resultados intermedios. El segundo método es utilizar sólo un bucle, y cada iteración realiza las aplicaciones *G* y *F* en secuencia. Este teorema se usa comúnmente para optimizar compiladores, para transformaciones de programas (tanto manuales como automáticos), y también es de vital importancia en la programación paralela. Nuevamente, dejamos la prueba como un ejercicio.

Teorema 19. (*map f. map g*) *xs* = *map* (*fg*) *xs*

Para un cambio de ritmo, ahora consideramos un teorema intrigante. Una vez que entiendes lo que dice, sin embargo, se vuelve perfectamente intuitivo. (La notación (1+) denota una función que suma 1 a un número.)

Teorema 20. *sum* (*map* (1+) *xs*) = *length xs* + *sum xs*

Prueba. Inducción sobre *xs*. El caso base es

sum (*map* (1+) [])

= *sum* []	*{map.*1 *}*
= 0+ *sum* []	*{* 0 + *x* = *x}*
= *length* [] + *sum* []	*{length.*1 *}*

Para el caso inductivo, suponga *sum* (*map* (1+) *xs*) = *length xs* + *sum xs*. Entonces

sum (*map* (1+) (*x* : *xs*))

= *sum* ((1 + *x*): *map* (1+) *xs*)	*{map.*2 *}*
= (1 + *x*) + *sum* (*map* (1+) *xs*)	*{suma.*2 *}*
= (1 + *x*) + (*length xs* + *sum xs*)	*{* hipótesis *}*
= (1 + *length xs*) + (*x* + *sum xs*)	*{* (+). Álgebra *}*
= *length* (*x* : *xs*) + *sum* (*x* : *xs*)	*{length.*2*, suma.*2 *}*

La función *foldr* es importante porque expresa un patrón de bucle básico. Hay muchas propiedades importantes de *foldr* y funciones relacionadas. Aquí está uno de ellos:

Teorema 21. *foldr* (:) [] *xs* = *xs*

Algunos de los teoremas anteriores pueden ser fáciles de entender de un vistazo, ¡pero es poco probable que sea cierto para este! Recuerde que la función *foldr* separa una lista y combina sus elementos usando un operador. Por ejemplo,

foldr (+) 0 [1,2,3] = 1 + (2 + (3 + 0))

Ahora, ¿qué sucede si combinamos los elementos de la lista usando el operador cons (:) en lugar de la suma, y si usamos [] como valor inicial para la recursión en lugar de 0? La ecuación anterior se convierte en

foldr (:) [] [1,2,3] = 1: (2: (3: []))
= [1,2,3].

Terminamos con la misma lista con la que comenzamos, y el teorema dice que esto siempre sucederá, no solo con el ejemplo [1,2,3] utilizado aquí.

Prueba. Inducción sobre *xs*. El caso base es

foldr (:) [] []
= [] *{foldr.*1 *}*

Ahora suponga que *foldr* (:) [] *xs* = *xs*; entonces el caso inductivo es

foldr (:) [] (*x* : *xs*)
= *x* : *foldr* (:) [] *xs* *{foldr.*2 *}*
= *x* : *xs* *{* hipótesis *}*

Supongamos que tiene una lista de listas, de la forma $xss = [xs_0, xs_1, ..., xs_n]$. Todas las listas xs_i debe tener el mismo tipo [*a*], y el tipo de *xss* es [[*a*]]. Es posible que queramos aplicar una función $f :: a \rightarrow b$ a todos los elementos de todas las listas, y construir una lista de todos los resultados. Hay dos formas diferentes de organizar este cálculo:

- use la función *concat* para hacer una lista plana única del tipo [*a*] que contenga todos los valores, y luego aplique el *map f* para producir el resultado con el tipo [*b*].
- Aplique el *map f por* separado a cada xs_i, calculando el *map* (*map f*) *xss*, produciendo una lista de tipo [[*b*]]. Luego use *concat* para aplanarlos en una sola lista de tipo [*b*].

El siguiente teorema garantiza que ambos enfoques producen el mismo resultado. Esto es significativo porque hay muchas situaciones prácticas en las que es más conveniente escribir un algoritmo utilizando un enfoque, pero el otro es más eficiente.

Teorema 22. *map f* (*concat xss*) = *concat* (*map* (*map f*) *xss*)

Prueba. Prueba por inducción sobre *xss*. El caso base es

map f (*concat* [])

= *map f* [] *{ concat.*1 *}*
= [] *{ map.*1 *}*
= *concat* [] *{ concat.*1 *}*
= *concat* (*map* (*map f*) []) *{ map.*1 *}*

Suponga que *map f* (*concat xss*) = *concat* (*map* (*map f*) *xss*). El caso inductivo es

map f (*concat* (*xs* : *xss*))
= *map f* (*xs* ++ *concat xss*) *{ concat.*2 *}*
= *map f xs* ++ *map f* (*concat xss*) *{* Teorema 18 *}*
= *map f xs* ++ *concat* (*map* (*map f*) *xss*) *{* hipótesis *}*
= *concat* (*map f xs* : *map* (*map f*) *xss*) *{ concat.*2 *}*
= *concat* (*map* (*map f*) (*xs* : *xss*)) *{ map.*2 *}*

A veces no es necesario realizar una inducción, porque una técnica de prueba más simple ya está disponible. Aquí hay un ejemplo típico:

Teorema 23. *length* (*xs*++ (*y* : *ys*)) = 1 + *length xs* + *length ys*

Este teorema ciertamente podría probarse por inducción (¡y eso podría ser una buena práctica!) Pero ya tienen un teorema similar que dice que *length* (*xs*++*ys*) = *length xs* + *length ys*. En lugar de comenzar una nueva inducción completamente nueva, es más elegante llevar a cabo algunos pasos que nos permiten aplicar el teorema existente. Así como la reutilización de software es una buena idea, la reutilización de teoremas es un buen estilo en informática teórica.

length (*xs* ++ (*y* : *ys*))
= *length xs* + *length* (*y* : *ys*)
= *length xs* + (1+ *length ys*)
= 1+ *length xs* + *length ys*

Ejercicio 5. Prueba el teorema 16.

Ejercicio 6. Prueba Teorema 18.

Ejercicio 7. Demuestre el teorema 19.

Ejercicio 8. Recuerde el teorema 20, que dice

sum (*map* (1+) *xs*) = *length xs* + *sum xs.*

Explica en español lo que dice este teorema. Usando las definiciones de las funciones involucradas (*suma*, *length* y *map*), calcule los valores de los lados izquierdo y derecho de la ecuación usando *xs* = [1,2,3,4].

Ejercicio 9. Inventa un nuevo teorema similar al Teorema 20, donde (1+) se reemplaza por (*k*+). Pruébelo en uno o dos pequeños ejemplos. Entonces probar el teorema.

4.6 Igualdad funcional

[No incluido.]

4.7 Errores comunes

Después de un poco de práctica, la inducción puede parecer casi demasiado fácil. Simplemente plantear el caso base y el caso inductivo, aplicar el método y sale una prueba. Hay muchos tipos de malas pruebas inductivas. Sus fallas a menudo se deben a casos de base sospechosos, aunque también hay una variedad de formas dudosas para probar los casos de inducción. Aquí hay un ejemplo interesante.

4.7.1 Un caballo de otro color

El siguiente teorema es un clásico famoso.

Teorema 26. Todos los caballos son del mismo color.

Prueba. Defina $P(n)$ como 'en cualquier conjunto que contenga n caballos, todos tienen el mismo color'. Procedemos por inducción sobre n.

Caso base. Cada caballo tiene el mismo color que él mismo, por lo que $P(1)$ es cierto.

Caso inductivo. Suponga $P(n)$ y considere un conjunto que contenga $n + 1$ caballos; llámalos $h_1, h_2, ..., h_{n+1}$. Podemos definir dos subconjuntos $A = h_1, ..., h_n$ y $B = h_2, ... h_{n+1}$. Ambos conjuntos A y B contienen n caballos, por lo que todos los caballos en A son del mismo color (llamarlo $C_{A),}$y todos los caballos en B son del mismo color (llamarlo $C_{B).}$Escoja uno de los caballos que es un elemento de A y B. Claramente este caballo tiene el mismo color que él mismo; llámalo C_h. Así $C_A = C_h = C_{B.}$Por lo tanto, todos los caballos $h_1, ..., h_{n+1}$ tienen el mismo color.
Ahora hemos demostrado $P(n) \rightarrow P(n + 1)$, por lo que se deduce por inducción matemática que $\forall n \in \text{Nat} \; . \; P(n)$. Por lo tanto, todos los caballos son del mismo color.

Ejercicio 13. ¿Cuál es el defecto en la prueba dada anteriormente? (Intente resolverlo usted mismo y luego verifique la respuesta dada más abajo).

Respuesta. La primera línea del caso inductivo dice ‘Suponga P (n) y considere un conjunto que contiene n + 1 caballos; llámelos h_1, h_2,. . . , h_{n+1}. "Considere lo que sucede cuando n = 1. Nuestro conjunto de n + 1 caballos contiene solo dos de ellos,

h_1 y h_2. Por lo tanto, los dos subconjuntos resultan ser A = {h_1} y B = {h_2}. Todos los caballos en A tienen el mismo color, y todos los caballos en B tienen el mismo color. Hasta ahora, todo bien, pero la siguiente oración dice "Elija uno de los caballos que es un elemento de A y B", y es imposible hacerlo porque cuando n = 1, los conjuntos A y B son disjuntos. El resto de la prueba no es válida porque se basa en este caballo inexistente.

Hay dos lecciones útiles para aprender de esto.

- Siempre que una prueba diga que ‘escoja una x tal que. . . ", es esencial asegurarse de que tal x exista.
- Es útil trabajar en los detalles utilizando un ejemplo concreto.

Por cierto: la falla explicada anteriormente es el único error en esta prueba. Si la prueba funcionara para el caso n = 1, ---si todos los pares de caballos tuvieran el mismo color---, entonces *sería cierto* que todos los caballos son del mismo color.