8 INDUCCIÓN

Los números naturales -enteros positivos más el `0`- tienen una estructura de construcción sencilla de comprender y de estudiar. A partir de los Axiomas de Peano, explicados en **7**, se observa que todo número natural es construible a partir del `0` y de la función sucesor `S`, que no hace otra cosa que aumentar en `1` el número que recibe como dato.

Considérese la siguiente situación hipotética[1]: supóngase una escalera en la que una persona se puede parar en el primer peldaño. Además, si la persona se para en un peldaño cualquiera, puede pararse también en el siguiente, si éste existe. Entonces, parece plausible concluir que la persona puede pararse en cualquier peldaño de la escalera, incluso, si la escalera es infinita. Se puede probar que el primer peldaño es accesible, después el segundo, luego el tercero, etc. Si la escalera es finita (tiene finitos peldaños), este proceso terminará, demostrando así que todo peldaño es alcanzable. Sin embargo, cuando hay infinitos peldaños, la prueba misma debería tener una longitud infinita y no sería viable de realizar.

Cuando la escalera es infinita, ella misma se puede asimilar a los números naturales, denotando con `0` el primer peldaño y con `S.n` el peldaño siguiente al que se haya denotado con `n`. En primera instancia, puede comprobarse que todo peldaño de la escalera se denota con algún número. Para esto, hay que observar que los peldaños se denotan sucesivamente con `0`, `S0`, `SS0`, etc. Si se llama `A` al conjunto de los números naturales que denotan peldaños alcanzables, se observa que $0 \in A$ y que, además, para todo $n \in A$, también $S.n \in A$. Entonces el Axioma **e** de Peano permite concluir que `A=`**`nat`**. Es decir, los peldaños alcanzables son todos los que se pueden denotar.

El razonamiento utilizado en la prueba anterior utiliza una forma de argumentación que se conoce como *inducción matemática*. En términos generales, se puede intentar usar esta clase de argumentos cuando se trate de mostrar una afirmación sobre un conjunto infinito y discreto[2], como lo es **`nat`**. En lo que sigue del capítulo se explican esquemas de pruebas por inducción que permiten demostrar, de manera sistemática, algunas afirmaciones que valen sobre subconjuntos de **`nat`**.

8.1 INDUCCIÓN SIMPLE

El cálculo de predicados puede extenderse con la siguiente regla de inferencia:

$$\text{Inducción:}\quad \frac{p.0,\ (\forall k |\ p.k : p(k+1))}{(\forall n |: p.n)} \;\Big|\; p: \mathbf{nat} \to \mathbf{bool}$$

La corrección de esta regla se puede justificar con el Axioma de Peano **e** de **7**. En otras palabras, esta regla de inferencia es consistente con el cálculo de predicados estándar aumentado con los Axiomas de Peano.

La introducción de la nueva regla da lugar a nuevas formas de demostración que usen, de manera específica, la inducción. Aparentemente, la regla solo sirve para mostrar afirmaciones universales sobre todo **`nat`**; sin embargo, se mostrará más adelante que también se puede usar para mostrar afirmaciones universales sobre ciertos subconjuntos de **`nat`**.

---

[1] A partir de un ejemplo en [Mey2010].

[2] Un conjunto *discreto* es uno cuyos elementos se pueden enumerar con números naturales. No tiene que tener la estructura de los naturales. Hay conjuntos que no son discretos, v.gr., el intervalo `[0,1]` de números reales (aunque no es trivial mostrar que no lo sea).

Se llamará *inducción simple* a esta, en la que se comprueba que el predicado vale para `0` y, después, se comprueba que vale para `k+1`, a partir de la suposición de que ya se ha probado para `k≥0`.

8.1.1 Esquema de prueba por inducción

Supóngase que se quiere probar una afirmación de la forma

$$(\forall n{:}\mathbf{nat}|:\ p.n)$$

donde p es un predicado sobre **`nat`**.

Para escribir demostraciones que usen inducción se procede de la siguiente forma:
(0) Definir p (si todavía no se ha definido) como *predicado de inducción*.
(1) Mostrar que vale `p.0`. Esto se denomina *probar el caso base*.
(2) Suponer que `p.k` vale , para un `k≥0`. En este caso, `p.k` es la *hipótesis de inducción*.
(3) Mostrar, a partir de la hipótesis de inducción, que también vale `p(k+1)`. Esto se denomina *probar el caso inductivo*.

Formalmente, el cumplimiento de estas condiciones permite que, usando la nueva regla de inferencia de inducción, se pueda concluir la verdad de la afirmación de que `p.n` vale para cualquier `n∈`**`nat`**.

Cuando se quiere ser formal, se usará el siguiente esquema de prueba:

Teo : $(\forall n|:\ p.n)$
Dem:
Inducción sobre **`nat`**.
Predicado de inducción: p.n ≡ …

Caso base: p.0
  ⟨Demostración de p.0⟩

Caso Inductivo: p(n+1)
HI: p.n, n≥0
  ⟨Demostración de p(n+1)⟩

□

El esquema anterior señala los elementos que debe incluir una prueba por inducción. Ocasionalmente se puede evidenciar que estos elementos se dan, así no se guarde con rigor el formato de prueba mencionado, lo que daría lugar a una demostración menos formal pero, igualmente, aceptable[3].

Ejemplo A

1 *Suma de los* n *primeros números*

Teo: $(+k|\ 0\le k<n\ :\ k) = n(n-1)/2$
Dem:
Inducción sobre **`nat`**.
Predicado de inducción: $p.n \equiv (+k|\ 0\le k<n\ :\ k) = n*(n-1)/2,\ \ n\ge 0$.

Caso base: p.0
  $(+k|\ 0\le k<0\ :\ k)$

[3] Naturalmente, se trata de escribir demostraciones correctas y una prueba rigurosamente formal puede ser más "aburrida", aunque también, más segura (en cuanto a no dejar deslizar errores).

```
=     ⟨rango vacío⟩
   0
=     ⟨aritmética⟩
   0*(0-1)/2

Caso Inductivo: p(n+1)
HI: p.n, n≥0
   (+k| 0≤k<n+1 : k)
=     ⟨Partir rango⟩
   (+k| 0≤k<n : k) + n
=     ⟨HI: (+k| 0≤k<n : k) = n*(n-1)/2⟩
   n*(n-1)/2 + n
=     ⟨aritmética⟩
   (n+1)*n/2
```

□

2 *Un resultado sobre divisibilidad*

Se mostrará que n^3-n es divisible por 3, para $n\geq 0$.

Teo: $(\forall n|: n^3-n =_3 0)$

Dem:

Inducción sobre **nat**.

Predicado de inducción: $d.n \equiv n^3-n =_3 0,\ n\geq 0$

```
Caso Base: d.0
   0³-0
=     ⟨Aritmética⟩
   0
=₃    ⟨=ₘ reflexiva⟩
   0

Caso Inductivo: d(n+1)
HI: d.n, n≥0
   (n+1)³-(n+1)
=     ⟨Aritmética⟩
   n³+3n²+3n+1-n-1
=     ⟨Aritmética⟩
   n³-n + 3*(n²+n)
=₃    ⟨HI: n³-n =₃ 0⟩
   3*(n²+n)
=₃    ⟨m*x =ₘ 0⟩
   0
```

□

8.1.2 Errores en pruebas usando inducción

Las pruebas por inducción pueden llevarse a cabo equivocadamente, cuando alguna de las condiciones que exige la regla de inferencia no es demostrada a cabalidad. Los errores típicos son, entonces:

(1) No se prueba $p.0$.

(2) No se prueba adecuadamente que $p.k \Rightarrow p(k+1)$, para $k\geq 0$.

Recuérdese que una prueba errada no significa que lo que se quiera probar sea falso. Simplemente, quiere decir que la supuesta demostración no ha seguido las reglas de prueba consideradas legales. Cuando lo que se demuestra es cierto es difícil ver que la prueba puede tener un error; sin embargo, es claro que un resultado contrario a la realidad (una contradicción) evidencia un error en la prueba misma, que debería descubrirse.

Ejemplo A

1 "La suma de los números impares hasta `2*n-1` *es* n^2"

Dem: Si $1+3+\ldots+(2*n-1) = n^2$, sumar el siguiente impar $(2*n+1)$, a ambos lados de la ecuación, permite afirmar que

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

□

El resultado es correcto, pero la demostración no lo es. Falta mostrar el caso base, i.e.,

$$1+3+\ldots+(2*0-1) = 0^2$$

el cual es verdadero, aunque la notación no ayuda mucho a ver por qué (es una suma vacía de términos). Sería mejor escribir la hipótesis de inducción como

$$P.n \equiv (+k|\ 0\le k<n:\ 2*k+1) = n^2,\ n\ge 0.$$

Entonces el caso base `P.0` es verdadero, ya que:

```
   P.0
=
   (+k| 0≤k<0: 2*k+1)
=     ⟨rango vacío⟩
   0
=
   0².
```

La discusión anterior sirve, además, para preferir las pruebas más formales, en el sentido de que se ajusten a los esquemas de prueba establecidos.

2 "Todas las niñas son rubias"

Supóngase que se quiere probar que todas las niñas son rubias. Otra manera de enunciar lo mismo es probar que, para cualquier grupo de n niñas, donde n es un número natural, todas ellas son rubias.

La siguiente demostración pretende probar el resultado.

Dem: Inducción sobre **`nat`**.
Predicado de inducción: `r.n` ≡ "En un grupo de n niñas, todas son rubias"
Caso base: `r.0`

```
   r.0
=
    "En un grupo de 0 niñas, todas son rubias"
=     ⟨rango vacío⟩
   true
```

Caso Inductivo: `r(n+1)`
`HI: r.n, n≥0`
Sea $G_{n+1} = \{x_0,\ldots,x_{n-1},x_n\}$ grupo de n+1 niñas.

Nótese que: $G_{n+1} = G_n \cup \{x_n\}$, donde $G_n = \{x_0,\ldots,x_{n-1}\}$.
Por HI, todas las niñas en G_n son rubias. En particular, x_{n-1} es rubia.
Sea $H_n = (G_n \setminus \{x_{n-1}\}) \cup \{x_n\}$.
Nótese que:
(1) H_n tiene n elementos
(2) $G_{n+1} = H_n \cup \{x_{n-1}\}$
Por HI, H_n está compuesto por sólo rubias.
Por (2), G_{n+1} también está conformado por solo rubias.

□

El resultado es claramente contrario a la realidad. Pero, ¿qué está mal en la prueba?

El caso base está correctamente planteado, ya que es un afirmación sobre el conjunto vacío. Si, por ejemplo, se tiene que todo grupo de 99 niñas está constituido por rubias, el argumento explicado sirve para mostrar que un grupo de 100 niñas también deberá estar constituido por solo rubias. Pero algo debe estar mal, de nuevo, porque el resultado es claramente falso.

El paso inductivo de 99 a 100 es correcto. Pero ¿qué pasa cuando se trata de probar P.1? La demostración afirma que G_0 tiene un elemento para intercambiar con x_1, pero aquí está el error, ya que $G_0 = \varnothing$.

Es decir: el paso inductivo "$p.k \Rightarrow p(k+1)$" no se prueba adecuadamente, para todo $k \geq 0$.

□

8.1.3 Un ejemplo gráfico

El siguiente ejemplo, basado en [Ros2007] , ilustra cómo se puede aplicar la inducción en contextos que no se circunscriben puramente a los números enteros ("suma de números", "suma de impares", "divisibilidad", etc.). Lo que importa es que se esté tratando con entidades que estén relacionadas con los números naturales y que lo que se quiera probar corresponda a una afirmación universal sobre números naturales.

El problema en cuestión se plantea en una situación en la que hay un patio cuadrado de dimensiones $2^n \times 2^n$, para cierto $n \geq 0$. Esto da lugar a una cuadrícula de lado 2^n. En uno de los cuadrados centrales se coloca una estatua. El patio se debe embaldosar con elementos de 3 unidades de área, en forma de L, i.e.,

Se debe demostrar que la labor se puede hacer sin usar fracciones de baldosas, para $n \geq 0$. Para fijar ideas, esto se llamará "L-embaldosar" el patio.

Curiosamente, resulta más fácil demostrar un resultado más general que el que originalmente se pide. De hecho, se probará:

Teorema A

Un patio cuadrado de dimensiones $2^n \times 2^n$, con $n \geq 0$, en el que en alguno de sus elementos se ha colocado una estatua, se puede L-embaldosar.

Dem: Inducción sobre **nat**.
Predicado de inducción:

`p.n` ≡ "se puede L-embaldosar un patio de lado 2^n que tenga una estatua en cualquier `(i,j)`, $0 \le i,j < 2^n$", $n \ge 0$.

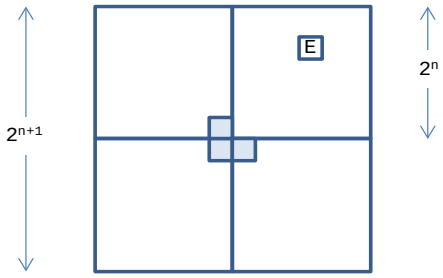
Caso base: `p.0`

El patio tiene lado `1`. La estatua se coloca en el centro y el resto del patio (nada!) se puede L-embaldosar usando `0` baldosas:

Caso inductivo: `p(n+1)`, $n \ge 0$.
`HI: p.n`.

Se considera un patio de lado 2^{n+1}. El patio se puede partir en cuatro cuadrantes, que pueden denominarse NE, NW, SE, SW, según su orientación.

Se coloca la estatua en algún cuadrado `(i,j)`. Sin pérdida de generalidad, se puede elegir el del cuadrante NE (¿qué hacer si se elige otro cuadrante?). Entonces, se usa una baldosa en L en los cuadrados centrales cubriendo una esquina de cada uno de los cuadrantes diferentes del NE y, claro, sin tocar el cuadrado central del cuadrante NE.



Ahora, cada cuadrante tiene dimensiones $2^n \times 2^n$ y un elemento ya cubierto (por la estatua, en el NE, y por la baldosa puesta, en los otros tres). Por HI, cada uno de estos cuadrantes se puede L-embaldosar.

Es decir, el cuadrado de lado 2^{n+1} se puede L-embaldosar.

□

Corolario A.1:
El teorema vale, si se pide que la estatua esté en el centro siempre.

□

El ejemplo anterior muestra cómo debilitar la hipótesis de inducción puede facilitar la prueba, de modo que se logra demostrar algo más general que lo originalmente planteado. Esta clase de situaciones es menos rara de lo que, en principio, puede parecer, pero no es evidente cómo reconocer que se puede dar y tampoco qué tanto se puede pensar en debilitar la hipótesis de inducción.

Ejercicios 8.1

1 Pruebe que el Principio de Inducción Simple es equivalente al Principio del Buen Orden.

2 Demuestre las siguientes afirmaciones:

a $(+k \mid 0 \le k \le n : k^2) = n(n+1)(2n+1)/6$

b $(+k \mid 0 \le k \le n : 2k) = n(n+1)$

c $(+k \mid 0 \le k \le n : k^3) = n^2(n+1)^2/4$

d $(+k|\ 0\le k\le n\ :\ 2^k) = 2^{n+1} - 1$

e $(+k|\ 0\le k\le n\ :\ 3^k) = (3^{n+1} - 1)/2$

f $n^2 + n$ es par, para $n\ge 0$.

g $2^{2n}-1$ es divisible por 3, para $n\ge 0$.

3 Considere el problema de L-embaldosar un patio de cuadrado de dimensiones 8×8. Las posiciones del patio se denotan (i,j) , con $0\le i,j<8$. Indique cómo colocar L-baldosas para cubrir el patio, si se pone una estatua en $(3,7)$.

8.2 INDUCCIÓN FUERTE

La forma en que se adelante una prueba por inducción puede variar dependiendo de lo que se quiera probar y de su rango de validez.

En una situación más general se puede suponer que se tiene un predicado p**:int** $\to$ **int** y que se quiere mostrar, para cierto entero z_0:

$$(\forall z|\ z_0\le z\ :\ p.z)$$

La recurrencia con la que se define a p determina que pueden tener que definirse varios casos base, digamos, $p.z_0$, $p(z_0+1)$,...,$p(z_0+m-1)$. De todas formas, el número de estos casos es finito.

También la recurrencia que define a p puede requerir, al definir $p.z$ que se tengan ya definidos uno o más valores en el intervalo entero $z_0..z-1$.

En este sentido, se puede formular una regla de inferencia adicional, que permita llevar a cabo esta clase de pruebas inductivas:

Inducción fuerte:
$$\frac{(\forall k:\textbf{int}|\ (\forall j|:\ z_0\le j<k\ :\ p.j):\ p.k)}{(\forall n:\textbf{int}|\ z_0\le n:\ p.n)} \quad \Big|\ p:\ \textbf{int}\to \text{bool}$$

Esta regla incluye, claramente, la de inducción simple. Es decir, toda demostración que se pueda hacer con inducción simple se puede hacer con inducción fuerte. Al contrario, a pesar de parecer no tan evidente en un principio, no es muy complicado de probar: toda prueba que se puede realizar con inducción fuerte también se puede realizar con inducción simple. Entonces, con el uso de inducción fuerte no se añade ningún poder de prueba al que ya se tiene con la inducción simple, aunque sí puedan simplificarse algunas demostraciones.

Ejemplo A

1 $(+k|\ 0<k\le n\ :\ 1/k^2) < 2-1/n,\ n>1$.

Dem: Inducción sobre $n>1$, $n\in$**nat**.
Predicado de inducción: $s.n \equiv (+k|\ 0<k\le n\ :\ 1/k^2) < 2-1/n,\ n>1$

Caso base: $s.2$

$s.2$
= ⟨Def s⟩
$(+k|\ 0<k\le 2\ :\ 1/k^2) < 2-1/2$
= ⟨...⟩
$1/1^2 + 1/2^2 < 2-1/2$
= ⟨aritmética⟩
true

Caso inductivo: s(n+1)
HI: s.n, $n \geq 2$

$(+k \mid 0<k\leq n+1 : 1/k^2)$
= ⟨partir rango⟩
$(+k \mid 0<k\leq n : 1/k^2) + 1/(n+1)^2$
< ⟨HI⟩
$2-1/n + 1/(n+1)^2$
< ⟨Lema⟩
$2-1/(n+1)$

Lema: $-1/n + 1/(n+1)^2 < -1/(n+1)$
Dem: …

□

2 Cualquier cantidad superior a 3 centavos se puede representar con monedas de 2 y de 5 centavos

En primer lugar, se establece formalmente lo que se quiere demostrar. Una cantidad de centavos n es representable con monedas de 2 y 5 centavos si existen h, k, enteros no negativos, tales que n=2*h+5*k. En otras palabras, se quiere mostrar:

Teorema
$(\forall n \mid n>3 : (\exists h,k \mid 2*h+5*k = n))$

Dem: Inducción sobre n>3.
Predicado de inducción: $P.n \equiv (\exists h,k:\textbf{nat} \mid 2*h+5*k = n)$, n>3.

Caso base: P.4
$2*2+5*0 = 4$
⇒ ⟨∃-introducción⟩
$(\exists h,k \mid 2*h+5*k = 4)$

Caso inductivo: P(n+1)
HI: n>3, $2*h_0+5*k_0 = n$ //(se cambia el ∃ por una versión con testigos)

Caso $k_0>0$:
n+1
= ⟨Hip: $2*h_0+5*k_0 = n$⟩
$2*h_0+5*k_0+1$
= ⟨Caso: $k_0>0$⟩
$2*h_0+5*(k_0-1)+5+1$
= ⟨Aritmética⟩
$2*(h_0+3)+5*(k_0-1)$
⇒ ⟨∃-introducción⟩
$(\exists h,k \mid 2*h+5*k = 4)$

Caso $k_0=0$:
n+1
= ⟨Hip: $2*h_0+5*k_0 = n$⟩
$2*h_0+5*k_0+1$
= ⟨Hip: $k_0=0$⟩
$2*h_0+1$
= ⟨Hip: n>3; n>3 ⇒ $h_0 \geq 2$⟩
$2*(h_0-2)+4+1$
= ⟨Aritmética⟩
$2*(h_0-2)+5$

```
⇒       ⟨∃-introducción⟩
    (∃h,k | 2*h+5*k = 4)
```

□

Es usual, en demostraciones por inducción de predicados que incluyen existenciales, apoyarse en la teorema del testigo y denotar los elementos que existen para poder manipularlos en la prueba del caso inductivo. Al final, para ser rigurosos, debería terminarse con un paso de ∃-introducción -como aquí se hizo- aunque, en muchas ocasiones esto se omite esperando que el lector de la prueba quede satisfecho sabiendo cómo se construyen los nuevos testigos en el paso inductivo.

3 Los números de Fibonacci son mayores que n, para n>5.

Los números de Fibonacci se definen de forma recurrente:

F.0 = 0
F.1 = 1
F(n+2) = F.n + F(n+1), n≥0.

Entonces:

F.2=1, F.3=2, F.4=3, F.5=5, F.6=8, ...

Se quiere mostrar un resultado muy sencillo de verificar, pero que exige más de un caso base en su prueba, lo que corresponde a usar inducción fuerte.

Teorema
F.n > n, n>5

Dem: Inducción sobre n>5.
Predicado de inducción: M.n ≡ F.n > n, n>5.

Caso base: M.6

```
   F.6>6
=
   8>6
=
   true
```

Caso base: M.7

```
   F.7>7
=
   13>7
=
   true
```

Caso inductivo: M(n+2)
HI: M.n, M(n+1), n>5

```
   F(n+2)
=     ⟨Def F⟩
   F.n + F(n+1)
>     ⟨HI: M.n, M(n+1)⟩
   n+n+1
>     ⟨HI: n>5⟩
   n+2
```

□

3 Todo número es factorizable unívocamente en factores primos

El Principio de Inducción Fuerte permite realizar una prueba directa y sencilla del Teorema Fundamental de la Aritmética utilizando el hecho de que un número es primo (y entonces es producto de primos) o es producto de dos números menores que él. En el segundo caso, hay descomposición en primos para los factores, lo que lleva a la conclusión del teorema.

□

Ejercicios 8.2

1 Pruebe que el Principio de Inducción Simple es equivalente al Principio de Inducción Fuerte.

2 Sea `f.n` el número de bits en `1` en la notación binaria estándar de `n`. Formalmente, se puede definir `f: nat → nat` por los siguientes axiomas:

$f.0 = 0$
$f(2*n) = f.n \ , \ 0<k<2^n$
$f(2*n+1) = 1+f.n, \ 0<k<2^n$

a Muestre que $f(2^n) = 1$, para $0 \leq n$.

b Muestre que $f(2^n-1) = n$, para $0 \leq n$.

3 Use inducción fuerte para probar:

a Si $a_0=1$, $a_2=2$ y $a_{n+2}=2a_{n+1}+a_n$, entonces $a_n \leq (5/2)^n$

b Si F_n es el n-ésimo número de Fibonacci, $F_n<2^n$

c Si $a_1=5$, $a_2=13$ y $a_{n+2}=5a_{n+1}-6a_n$, $n \geq 1$ entonces $a_n=2^n+3^n$

8.3 DEFINICIONES RECURSIVAS

Es usual encontrar definiciones de colecciones de objetos matemáticos en los que objetos complejos son definidos en términos de objetos más simples. Por ejemplo, la clásica definición de la función factorial:

`.!: nat → nat`
`0! = 1`
$(n+1)! = (n+1)*n! \ , \ n \geq 0$

La buena definición de una definición recursiva siempre puede cuestionarse y, en rigor, debería mostrarse que sí es una buena definición, en el sentido de que lo que se define se apoya en conceptos que ya se tienen como bien definidos. En este sentido, se puede probar el siguiente resultado:

Teorema A
`n!` está bien definido, $n \geq 0$
Dem:
Inducción sobre **`nat`**.
Predicado de inducción: `def(n)` ≡ "`n!` está bien definido", $n \geq 0$.

Caso base: `def.0`
`0!` está bien definido, porque se define como la constante `1`.

Caso Inductivo: `def(n+1)`
HI: `def.n` , $n \geq 0$.
Ahora, `(n+1)!` se define como el producto de `n!` (que está bien definido, por HI) por `n+1`, un valor conocido.

Es decir, `(n+1)!` está bien definido, y vale `def(n+1)`.

□

Tales demostraciones se omiten en la práctica, porque usualmente la buena definición se entiende y se acepta sin mayores discusiones. En cambio se pueden formular recetas para garantizar que una definición recursiva sea una buena definición. por ejemplo, si se quiere definir una función

$$f: \mathbf{nat} \to \mathbf{nat}$$

se puede proceder de la siguiente forma:

(1) Definir $f.0,\ldots,f.n_0$, para un $n_0 \in \mathbf{nat}$.

(2) Definir $f.k$, usando valores anteriores $f.0,\ldots,f(k-1)$, para $n_0 < k$.

Este es el caso de `n!` o de `F.n` (cf. 8.2 Ejemplo A.3). En general, es el caso de sucesiones (funciones cuyo dominio es **nat**) cuya definición se entiende de manera recursiva.

Es usual, también, estudiar y demostrar propiedades de los objetos que se definen recursivamente. Cuando esto sucede, la demostración debe apelar a un argumento inductivo que, seguramente, corresponde a la forma en que se construyen los objetos.

Ejemplo A

1 Dado un número natural $a>0$, considérese la función g definida así:

$g: \mathbf{nat} \to \mathbf{nat}$
$g.0 = 1$
$g(2*n) = (g.n)*(g.n) \quad , n\geq 0$
$g(2*n+1) = g(2*n)*a \quad , n\geq 0$

La definición de g es buena: se define en `0` y, para $n\geq 0$ está bien definida, considerando el caso en que n sea par o impar. Cuando n es par la definición se apoya en la de `g(n/2)`; cuando es impar, en la de `g(n-1)`.

Se puede mostrar que $g.n = a^n$, $n\geq 0$. Nótese que se necesita una inducción fuerte que, además, sigue el esquema de casos que está presente en la definición de g. Como ya se dijo, esto no es casual.

Dem: Inducción sobre $n\geq 0$.
Predicado de inducción: $C.n \equiv g.n = a^n$, $n\geq 0$.

Caso base: `C.0`

```
   g.0
=      ⟨Def g⟩
   1
=      ⟨a⁰ = 1⟩
   a⁰
```

Caso inductivo: `C(n+1)`

`HI: C.0, C.1,..., C(n-1), n≥0`
`Casos: n=2*k, n=2*k+1  // n par, n impar (k es un testigo de la paridad)`

```
Caso: n+1=2*k
   g(n+1)
=      ⟨caso: n+1=2*k⟩
   g(2*k)
=      ⟨Def g⟩
   (g.k)*(g.k)
=      ⟨HI: C.k (porque k=(n+1)/2, 0≤k<n+1)⟩
```

```
   a^k*a^k
=
   a^(2*k)
=     ⟨caso: n+1=2*k⟩
   a^(n+1)

Caso: n+1=2*k+1
   g(n+1)
=     ⟨caso: n+1=2*k+1⟩
   g(2*k+1)
=     ⟨Def g⟩
   g(2*k)*a
=     ⟨HI: C(2*k) (porque 2*k=n, 0≤2*k<n+1)⟩
   a^(2*k)*a
=
   a^(2*k+1)
=     ⟨caso: n+1=2*k+1⟩
   a^(n+1)
```

□

8.4 INDUCCIÓN SOBRE OTRAS ESTRUCTURAS

Los números naturales son la estructura numérica por excelencia para definir funciones recursivas y probar propiedades sobre ellas usando inducción, como ya se vio en las secciones anteriores. Sin embargo, la noción de definición recursiva no es exclusiva de los números naturales. Otros dominios pueden ser dotados de un orden que permita efectuar pruebas inductivas de manera análoga a la que se hace sobre **nat**. Esto se logra con el concepto de orden bien fundado introducido en 6.6.

Recuérdese que una relación de orden es bien fundada cuando el orden estricto correspondiente no permite construir cadenas descendentes infinitas[4]. Cuando se tiene un conjunto ordenado bien fundado $(A,\preccurlyeq)$, se puede pensar en tener definiciones recursivas basadas en el orden de A. Si se definen objetos o propiedades $p.a$, para $a \in A$, se procede así:

- definir $p.a$ para $a \in A$ que no tengan predecesores (estos a se llaman *minimales*);
- definir $p.a$ que tengan predecesores suponiendo que ya se conocen las definiciones de éstos.

La buena fundación del orden permite entender la definición de $p.a$ en un número finito de pasos, ya que nunca se podrá armar una cadena de predecesores de tamaño infinito.

Las demostraciones por inducción en estas estructuras deben seguir esquemas análogos a los ya explicados para $(\mathbf{nat},\leq)$ que, al fin y al cabo, es un conjunto bien fundado.

Formalmente, la discusión anterior se puede resumir en aumentar el cálculo deductivo con la siguiente regla de inferencia:

Inducción bien fundada:

$$\frac{(\forall a:A \mid a \text{ minimal} : p.a) \quad (\forall a:A \mid (\forall b:A \mid : b \prec a : p.b) : p.a)}{(\forall a:A \mid : p.a)} \qquad \begin{array}{l} p: A \to \text{bool} \\ (A,\preccurlyeq) \text{ obf} \end{array}$$

[4] O mejor, de longitud arbitrariamente grande.

8.4.1 TIPOS ABSTRACTOS DE DATOS ECUACIONALES

En informática se utilizan estructuras de datos como arreglos unidimensionales y listas. Cada dato se guarda en una posición de la estructura y, en estos casos, los datos se pueden considerar indexados por un conjunto asimilable a los números naturales. Entonces se puede pensar en efectuar demostraciones inductivas sobre esta clase de estructuras, siguiendo esquemas similares a los ya explicados para $(\mathbf{nat}, \leq)$.

Más generalmente, se puede pensar en construir objetos de manera recursiva y, después, probar propiedades de estos objetos apoyándose en la recursividad de la construcción. Esto da lugar a una técnica conocida como *inducción estructural*, en la que el dominio de definición de los objetos es el conjunto en el que se realiza la inducción y el orden bien fundado que permite hacer la inducción es el mismo que sustenta la buena definición de los objetos en cuestión.

Los *tipos abstractos ecuacionales* (TADs) son estructuras algebraicas -conjuntos con operaciones- que representan de manera genérica (independiente de la implementación) estructuras de datos que se usan en informática. Pueden considerarse como representaciones abstractas de clases de un programa escrito en un lenguaje orientado por objetos. Tanto la construcción de los objetos como la semántica de las operaciones se puede explicar de manera recursiva, con axiomas adicionales en forma de ecuación. No se pretende ser exhaustivo en la explicación de todo lo que se puede hacer con TADs; en cambio, se mostrarán ejemplos que expliquen cómo la inducción es una herramienta fundamental para la representación y la manipulación de esta clase de estructuras de datos[5].

8.4.1.1 Construcción recursiva de objetos

La idea que se tiene, para construir objetos recursivamente, es la de comenzar con unos objetos dados por entendidos y, a partir de tener ya definidos algunos objetos, poder definir la construcción de objetos más complicados que, de algún modo, incluyan a los ya entendidos.

En realidad, se construye el conjunto de los objetos al tiempo que se construyen los objetos mismos. Se mostrará un par de ejemplos que ilustran cómo puede llevarse a cabo esta construcción, de manera que sea razonable y aceptable.

Listas

Para definir el TAD `Lista[X]`, el conjunto de las listas de elementos de un conjunto `X`, se puede proceder a declarar dos operaciones de construcción, `vac` e `ins` (se escribe `Lista` como simplificación de `Lista[X]`):

* `vac:` $\rightarrow$ `Lista`
* `ins: Lista` $\times$ `X` $\rightarrow$ `Lista`

El * que precede los símbolos denota que son estas operaciones las que servirán para construir objetos. La función `vac` denota, en realidad, una constante de tipo `Lista`. La idea es que éste sea el nombre con que uno pueda referirse a una lista vacía. La función `ins` es una que construye una lista nueva dada agregándole un elemento a otra que ya esté construida. Así, `ins(l,x)`es una lista como `l` a la que, además, se ha agregado el elemento `x`.

En este punto conviene tener una representación informal de las listas con elementos en `X`. Por ejemplo representar con

$$\langle a_1, a_2, \ldots, a_r \rangle$$

[5] Véase [Car1993] si se quiere entender de manera detallada el concepto de TAD y los alcances de esta herramienta de modelaje y de especificación.

una lista de r elementos, $0 \leq r$, que están en cola de manera que el primero de ellos es a_1,el segundo a_2, etc. El último elemento en la lista es a_r. Nótese que, si l es la lista $\langle a_1, a_2, \ldots, a_r \rangle$, $ins(l,x)$ debe ser la lista $\langle a_1, a_2, \ldots, a_r, x \rangle$.

Por ejemplo, si X=**nat**, la lista $\langle 23,12,0,4,12 \rangle$ se puede construir con las operaciones vac e ins así:

$$ins(ins(ins(ins(ins(vac,23),12),0),4),12).$$

Formalmente, la declaración de los encabezados de las operaciones se puede entender como una taquigrafía de la siguiente construcción inductiva:

$$Lista_0 = \{vac\}$$

$$Lista_k = Lista_{k-1} \cup \{ins(l,x) \mid l \in Lista_{k-1} \wedge x \in X\},\ k>0$$

$$Lista = (\cup k \mid k \geq 0 : Lista_k).$$

Árboles binarios

Los árboles binarios con elementos en un conjunto X se pueden construir tomando como constante base un árbol vacío, que aquí se denominará $arvac$ y una operación constructora $cons$. Esta última tendrá tres argumentos, dos árboles binarios -ya construidos- y un elemento de X. El resultado de la construcción será un árbol nuevo, cuya *raíz* sera el elemento que llega como dato y cuyos *hijos izquierdo* y *derecho* serán los árboles que llegan como dato.

El TAD $Arbin[X]$, el conjunto de los árboles binarios de elementos de un conjunto X se puede construir inductivamente así (de nuevo, se escribe $Arbin$ como taquigrafía de $Arbin[X]$):

$$* \ arvac: \rightarrow Arbin$$

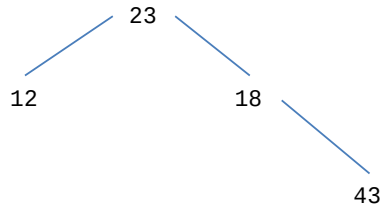
$$* \ cons : Arbin \times X \times Arbin \rightarrow Arbin$$

$$Arbin_0 = \{arvac\}$$

$$Arbin_k = Arbin_{k-1} \cup \{cons(a1,x,a2) \mid a1,a2 \in Arbin_{k-1} \wedge x \in X\},\ k>0$$

$$Arbin = (\cup k \mid k \geq 0 : Arbin_k)$$

Tomando otra vez a X=**nat**, un árbol binario de la forma



se construye con $cons(cons(arvac,12,arvac),23,cons(arvac,18,cons(arvac,43,arvac)))$.

La notación es complicada y se puede simplificar usando abreviaturas como, por ejemplo:

$[] \approx arvac$

$[a] \approx cons(arvac,a,arvac)$

$[h,a,h'] \approx cons(h,a,h')$

Con estas abreviaturas la construcción el árbol del ejemplo se denota

$$[[12],23,[[],18,[43]]]$$

que sigue siendo rara, pero puede ser más entendible[6].

8.4.1.2 Definición recursiva de operaciones

Como ya se indicó, los TADs son conjuntos (como `Lista` o `Arbin`) con operaciones que se definen según se necesite. En esta sección se explicará cómo se pueden definir operaciones sobre listas y sobre árboles binarios, usando recursivamente la misma construcción de los objetos que ya se explicó.

Listas

En una lista $l = \langle a_1, a_2, \ldots, a_r \rangle$ puede ser interesante definir:

- `esv.l`, un valor booleano que señale si la lista es o no vacía.
- `prim.l`, el primer elemento de la lista. Debe ser a_1.
- `resto.l`, la lista que resulta de `l` después de eliminar el primer elemento. Debe ser $\langle a_2, \ldots, a_r \rangle$.
- `tam.l`, el tamaño de la lista. Debe ser el número natural `r`.

Si hubiera otra lista $l' = \langle b_1, b_2, \ldots, b_s \rangle$ podría ser interesante

- `concat(l,l')`, lista resultado de concatenar `l` y `l'`. Debe ser $\langle a_1, a_2, \ldots, a_r, b_1, b_2, \ldots, b_s \rangle$.

La definición de operaciones puede hacerse ahora de manera recursiva, con la tranquilidad de que `Lista` ya se definió de la manera indicada con las operaciones `vac` e `ins`. Lo primero que hay que hacer, para cada operación es definir la funcionalidad correspondiente. Enseguida se dan axiomas que garanticen que la función se defina bien en su dominio. No hay una única forma de hacer esta definición axiomática y el elegir una mejor forma (más legible, más utilizable, etc.) depende de quien diseña el TAD.

Por ejemplo, la función `esv` se puede definir así:

```
esv: Lista → bool
[e1] esv.vac = true
[e2] esv(ins(l,x)) = false.
```

En este caso, los axiomas garantizan solo el argumento `vac` hace que `esv` dé `true` como resultado. Nótese, también, que el efecto de `esv` sobre cualquier `l:Lista` (su dominio) ha sido definido.

Ahora puede definirse `prim`. Nótese que la lista vacía no tiene primer elemento, de modo que, en este sentido `prim` es una función parcial:

```
prim: Lista → X
[p1] prim(ins(l,x)) = x.
```

La función `resto` podría ser definida así:

```
resto: Lista → Lista
[r1] resto.vac = vac
[r2] resto(ins(l,x)) = if esv.l then vac else ins(resto.l,x) fi
```

Hay varias observaciones al respecto de esta última definición:

---

[6] De todas maneras, nótese que la notación de construcción es una forma lineal de describir un objeto que se entiende mejor cuando se dibuja en dos dimensiones. El lograr una 'mejor notación' puede ser una pretensión difícil de cumplir.

- El definir que el resto de `vac` sea `vac` es cuestión de gusto del diseñador. Hay quienes prefieren decir que esto es un error y, como `prim.vac` , evitar definir `resto.vac`. Pero el suso de este axioma no da problemas prácticos y sí permite que `resto` se una función total.
- La función `esv` se usa en la definición de `resto`, porque ya se supone bien definida.
- El lado derecho del axioma `[r2]` corresponde a una definición por casos, para lo cual se permite una notación condicional **`if ... fi`**, con la semántica usual. Por supuesto, si el resultado de la operación debe ser de tipo `Lista`, por cualquiera de las ramas debe llegarse a un objeto de esta clase.

La función `tam` es muy fácil de definir:

```
[t1] tam.vac = 0
[t2] tam(ins(l,x)) = 1 + tam.l.
```

Por último, la función `concat` se puede definir así:

```
[c1] concat(l,vac) = l
[c2] concat(l,ins(l',x)) = ins(concat(l,l'),x).
```

Nótese que en esta definición se ha recurrido únicamente sobre el segundo argumento. Otra posible axiomatización hubiera podido ser:

```
[d1] concat(vac,vac) = vac
[d2] concat(ins(l,x),vac) = ins(l,x)
[d3] concat(vac,ins(l,x)) = ins(l,x)
[d4] concat(ins(l,x),ins(l',x')) = ins(concat(ins(l,x),l'))
```

Una mirada cuidadosa permite intuir[7] que los axiomas `[c1]` y `[c2]` definen la misma función que definen los axiomas `[d1]`,`[d2]`,`[d3]` y `[d4]`. Ya que esto es así, los primeros axiomas son más legibles y sencillos; se debería preferir esta axiomatización.

Árboles binarios

Como ejemplos de operaciones para árboles binarios se darán algunas que se explicarán a medida que que se definan.

En primer lugar, puede ser interesante definir funciones como `raíz`, `hi` y `hd` que permitan calcular la raíz, el hijo izquierdo y el hijo derecho de un árbol binario. Como en el caso del primer elemento de una lista vacía, tampoco tiene sentido hablar de la raíz de una árbol vacío. Y, de la misma manera, no hay ningún problema en pedir que el hijo izquierdo y el hijo derecho de un árbol vacío se defina como un árbol vacío. A continuación, las definiciones de estas funciones:

```
raíz: Arbin → Arbin
[r1] raíz(cons(a,x,a')) = x

hi: Arbin → Arbin
[hi1] hi.arvac = arvac
[hi2] hi(cons(a,x,a')) = a


hd: Arbin → Arbin
[hd1] hd.arvac = arvac
[hd2] hd(cons(a,x,a')) = a'.
```

[7] En rigor, esta igualdad entre funciones debería probarse (cf. 8.4.1.3).

También puede interesar establecer el tamaño del árbol, como el número de nodos que tenga:

```
tama: Arbin → nat
[m1] tama.arvac = 0
[m2] tama(cons(a,x,a')) = 1 + tama.a + tama.a'.
```

La altura de un árbol puede calcularse así:

```
alt: Arbin → nat
[a1] alt.arvac = 0
[a2] alt(cons(a,x,a')) = 1 + max(alt.a,alt.a')
```

donde `max(x,y)` es el máximo de los números `x,y`.

8.4.1.3 Prueba de propiedades sobre TADs

La prueba de propiedades sobre objetos de TADs debe, necesariamente usar inducción. Esta inducción, apoyada en la construcción de los objetos, es una variante lo que se denomina en la literatura como *inducción estructural*. La idea es mostrar lo que se quiere para los casos base (los objetos que no se definen recursivamente) y después, usando como hipótesis de inducción que lo que se quiere probar vale sobre objetos más simples que el que se está observando, poder argumentar que el resultado también vale en el caso complejo.

De nuevo, se ilustrará lo anterior con ejemplos sobre listas y árboles binarios.

Listas

Supóngase que se quiere mostrar el siguiente resultado

Teorema A
Para `l,l': Lista`,
`tam(concat(l,l')) = tam.l + tam.l'`

Dem: Inducción sobre `Lista`.
Predicado de inducción: `T(l,l')` ≡ `tam(concat(l,l')) = tam.l + tam.l'`, con `l,l': Lista`.

Caso base: `T(l,vac)`.

```
   tam(concat(l,vac))
=     ⟨[c2]⟩
   tam.l
=     ⟨aritmética⟩
   tam.l + 0
=     ⟨[t1]⟩
   tam.l + tam.vac
```

Caso inductivo: `T(l,ins(l',x))`
`HI: T(l,la), la≼ins(l',x)`
// Vale con `la` " más simple que" `ins(l',x)` (i.e., que tiene menos elementos)

```
   tam(concat(l,ins(l',x)))
=     ⟨[c2]⟩
   tam(ins(concat(l,l'),x)))
=     ⟨[t2]⟩
   1 + tam(concat(l,l'))
=     ⟨HI: l'≼ins(l',x)⟩
```

```
   1 + tam.l + tam.l'
=     ⟨aritmética⟩
   tam.l + 1 + tam.l'
=     ⟨[t2]⟩
   tam.l + tam(ins(l',x))
```

□

La hipótesis de inducción se refiere a que el resultado se puede suponer verdadero para listas más pequeñas que `ins(l',x)`, que es la que se está concatenando en el caso inductivo. La formulación que se da corresponde a lo que podría llamarse una inducción fuerte; para este caso, bastaba suponer, más débilmente, que valía `T(l,l')`.

Árboles

Se quiere mostrar que un árbol binario de altura n puede tener, a lo sumo, 2^n-1 nodos. Formalmente:

Teorema B
Para `a:Arbin`,
$\texttt{tama.a} \leq 2^{\texttt{alt.a}} - 1$

Dem: Prueba con hipótesis e inducción sobre `Arbin`.
Predicado de inducción: $\texttt{Q.a} \equiv \texttt{tama.a} \leq 2^{\texttt{alt.a}}-1,\ \texttt{a:Arbin}$

Caso base: `Q.arvac`

```
   tam.arvac
=     ⟨[a1]⟩
   0
≤     ⟨aritmética⟩
   2^0-1
=     ⟨[t1]⟩
   2^alt.arvac-1
```

Caso inductivo: `Q.cons(h,x,h')`
`HI: Q.h, Q.h'` // Vale con árboles más simples que `cons(h,x,h')`, en particular, h y `h'`

```
   tama.cons(h,x,h')
=     ⟨[a2]⟩
   1 + tama.h + tama.h'
≤     ⟨HI⟩
   1 + 2^alt.h - 1 + 2^alt.h' - 1
≤     ⟨2^z es creciente⟩
   2^max(alt.h,alt.h') + 2^max(alt.h,alt.h') - 1
=     ⟨aritmética⟩
   2^1+max(alt.h,alt.h') – 1
=     ⟨[a2]⟩
   2^alt.cons(h,x,h')
```

□

8.4.2 TERMINACIÓN DE PROCESOS

Una ejecución de un proceso se puede entender como una sucesión de estados. En realidad, el proceso se describe explicando cómo se cambia de un estado a otro siguiente o, en otras palabras, comprendiendo

cuál puede ser la relación de "sucesor" entre los estados del proceso. Esta relación puede ser complicada de estudiar, de modo que la evolución del proceso resulte difícil de predecir.

En particular, puede ser interesante predecir si un proceso debe terminar o no. En informática es muy importante poder demostrar que un programa (un proceso de computación) termine o no[8]. La terminación de un proceso, a partir de un estado inicial, se entiende como la propiedad de que cualquier posible ejecución sea una sucesión finita de estados.

Se mostrará con un par de ejemplos como los argumentos de inducción pueden ayudar a mostrar la terminación de procesos y, eventualmente, afirmar propiedades que deban cumplirse al momento de terminar.

8.4.2.1 Bolas blancas y negras [Gri1993]

Una bolsa contiene B bolas blancas y N negras , $B+N>0$. El siguiente proceso se repite siempre que sea posible:

i Se extraen dos bolas al azar.
ii Si tienen el mismo color, se tiran, pero se entra una bola negra en la bolsa (hay bolas negras suficientes para hacer esto, si hace falta).
iii Si son de diferente color, se devuelve la bola blanca y se tira la negra.

En primer lugar, se quiere saber si el proceso debe terminar o si, por el contrario, existe la posibilidad de que continúe indefinidamente. Una segunda pregunta, en caso de que el proceso termine, es saber en qué condiciones pasa esto. El proceso acaba si no se pueden sacar dos bolas, y esto sucede si queda una o si no queda ninguna. Una tercera pregunta, si el proceso terminara porque solo hay una bola en la bolsa, es averiguar el color de esta bola.

Cada posible ejecución del proceso se modelar como una sucesión de estados

$$\langle (b_0,n_0),(b_1,n_1),(b_2,n_2),\ldots,(b_k,n_k),\ldots \rangle$$

donde la pareja de números naturales (b_k,n_k) denota el número de bolas blancas y de bolas negras en la bolsa después de efectuar k iteraciones. Al empezar, $b_0=B$, $n_0=N$, y $b_0+n_0>0$. Se puede observar que el proceso puede pasar del estado k al $k+1$ si $b_k+n_k>1$ y, en este caso, (b_{k+1},n_{k+1}) puede tomar cualquiera de los siguientes tres valores:

(b_k-2,n_k+1) : si se sacaron 2 bolas blancas;
(b_k,n_k-1) : si se sacaron 2 bolas negras;
(b_k,n_k-1) : si se sacaron 2 bolas de diferente color.

Ahora se puede mostrar el siguiente resultado:

Lema A
Para $k\geq 0$, siempre que se pueda iterar (i.e., si $b_k+n_k>1$):

(1) $b_{k+1}+n_{k+1}>0$
(2) $b_{k+1}+n_{k+1} = b_k+n_k-1$, para $k\geq 0$
(3) $n_{k+1} =_2 n_k$, para $k\geq 0$.

Dem: Considérense los predicados, definidos para $k\geq 0$:
$P.k \equiv b_k+n_k>1 \Rightarrow b_k+n_k>0$

---

[8] En programas que calculan un resultado, se quiere mostrar que hay terminación. Pero, en programas que proveen un servicio permanente - como sucede con un sistema operacional- se quiere mostrar que no puede haber terminación, porque ésta conllevaría un comportamiento indeseado.

$Q.k \equiv b_k+n_k>1 \Rightarrow b_{k+1}+n_{k+1} = b_k+n_k-1$
$R.k \equiv b_k+n_k>1 \Rightarrow b_{k+1} =_2 b_k$

Supóngase que $b_k+n_k>1$, para $k\geq 0$. Entonces, (b_{k+1}, n_{k+1}) se define y puede ser uno de los dos valores:

- (b_k-2, n_k+1), en cuyo caso:
  $b_{k+1}+n_{k+1} = b_k-2+n_k+1 = b_k+n_k-1 > 0$. Por tanto, valen $P(k+1)$ y $Q(k+1)$.
  $b_{k+1} = b_k-2 =_2 b_k$. por tanto, vale $R(k+1)$.

- (b_k, n_k-1), en cuyo caso:
  $b_{k+1}+n_{k+1} = b_k+n_k-1 > 0$. Por tanto, valen $P(k+1)$ y $Q(k+1)$.
  $b_{k+1} = b_k =_2 b_k$. por tanto, vale $R(k+1)$.

Es decir, siempre que se pueda iterar, se cumplen los predicados P, Q y R en el siguiente estado.

□

Y entonces es fácil probar :

Teorema B
(1) El proceso termina.
(2) Se termina con una bola en la bolsa.
(3) La última bola es negra si y solo si $B =_2 0$.

Dem:

(1) La sucesión $\langle b_0+n_0, b_1+n_1, \ldots, b_k+n_k, \ldots \rangle$ es una cadena descendente en **nat**. Como $(\mathbf{nat}, \leq)$ es un obf, esta cadena no puede ser infinita, de modo que el proceso debe terminar.

(2) El resultado 2 del Lema permite probar por inducción que siempre hay bolas en la bolsa, incluso al final. Como se debe terminar, debe haber un estado final (b_f, n_f) para el que, además, debe valer $b_f+n_f\leq 1$. Es decir, se acaba en $(0,1)$ o en $(1,0)$.

(3) Sea f el índice del estado final. Inducción sobre el intervalo $0..f$. El predicado de inducción será $N.k \equiv b_k =_2 B,\ 0\leq k\leq f$.

Caso base: $N.0$
$b_0 =_2 B$
= $\langle b_0 = B \rangle$
$B =_2 B$
= $\langle =_m$ reflexividad$\rangle$
true

Caso inductivo: $N(k+1)$
HI: $N.k,\ 0\leq k<f$
$b_{k+1} =_2 B$
= $\langle b_{k+1} =_2 b_k$ Lema A.3$\rangle$
$b_k =_2 B$
= $\langle$HI$\rangle$
true

□

8.4.2.2 El patio de trenes

En una estación de ferrocarril hay un patio de parqueo que se desea desocupar. En el patio hay trenes, es decir, secuencias de uno o más vagones. El operario encargado de la tarea efectúa, al azar y mientras pueda, una de dos maniobras:

- separar un tren en dos trenes más cortos
- sacar un vagón (un tren de un vagón) del patio.

Se quiere mostrar que el proceso termina y, al final, el patio está desocupado.

Aunque parezca evidente, puede ser difícil explicar la terminación del proceso, de manera satisfactoria. Al igual que en el ejemplo de 8.4.2.1, la manera adecuada es definir estados del proceso y un obf en que los estados formen una cadena estrictamente descendente.

En este caso, el estado se puede representar con una pareja `(v,a)` de números naturales, que representen, respectivamente, el número total de vagones en el patio y el número total de amarres entre vagones en el patio. Nótese que las posibles operaciones hacen una transición de `(n,a)` a un estado `(n',a')`igual a

- `(n,a-1)` , si se separan dos trenes;
- `(n-1,a)` , si se saca un vagón.

Ahora, es claro que $(n,a) \succ_{lex} (n',a')$ usando el orden lexicográfico corriente sobre **nat**×**nat**. Por tanto, los estados forman una cadena estrictamente descendente en un obf, y el proceso debe terminar. El estado final debe ser `(0,0)` porque, de lo contrario, el operario siempre debería poder hacer algo.

Ejercicios 8.4

1 Enriquezca el TAD `Lista[`**nat**`]`con axiomas que definan las siguientes funciones

a `suma: Lista` → **nat**
donde `suma.l` es la suma de los elementos de la lista `l`.

b `mcd: Lista` → **nat**
donde `mcd.l` es el máximo común divisor de los elementos de la lista `l`.

2 Enriquezca el TAD `Arbin[X]`con axiomas que definan las siguientes funciones

a `completo: Arbin` → `bool`
donde `completo.a` si el árbol a es completo.

b `preorden: Arbin` → `Lista`
donde `preorden.a` es la lista que corresponde a la secuencia en preorden de los elementos del árbol a.

3 Considere las definiciones recursivas sobre **nat**, para detectar la paridad o no paridad de un número,

`par, impar:` **nat** → `bool`

con los axiomas:

```
[p1] par.0 = true
[p2] par(n+1) = impar.n
[i1] impar.0 = false
[i2] impar(n+1) = par.n
```

¿Definen estos axiomas funciones que calculen efectivamente los conceptos pretendidos? Es decir:

- Está el cálculo de `par.n` e `impar.n` bien definido?

4 Considere los Algoritmos de Euclides como se definieron en 7.3.1. Muestre que estos procesos terminan siempre que los datos de entrada satisfagan la precondición.