Introducción a los Algoritmos - 2do. cuatrimestre 2010

Guía 1: Razonamiento ecuacional, inducción y recursión

Docentes: Araceli Acosta, Laura Alonso i Alemany, Luciana Benotti, Paula Estrella

El objetivo general de esta guía es que te familiarices con la metodología del cálculo ecuacional, que utilizaremos como forma de escribir demostraciones a lo largo de toda la materia. Este formato de prueba, que incluye ***justificaciones exhaustivas para cada paso****, resulta muy útil para evitar cometer errores, en particular cuando las demostraciones se van haciendo más largas y complejas. Introducimos además dos temas muy importantes: las definiciones recursivas, y la inducción como sistema de demostración asociado a ellas. Un objetivo secundario es comenzar a utilizar un interprete de* `haskell` *como herramienta que te asista en los ejercicios.*

Razonamiento ecuacional, *validez* y *satisfacción*

El objetivo de estos ejercicios es introducir el razonamiento ecuacional sobre un dominio familiar: la aritmética. De esta forma podemos centrarnos en el ***método de prueba*** *mas que en las propias demostraciones, que esperamos no te presenten demasiadas dificultades. Además introducimos brevemente nociones que distinguen cuando una fórmula es siempre verdadera (la llamamos* ***válida****), o verdadera para algunos valores (la llamamos* ***satisfactible****); y sus contrarios, cuando es falsa para algunos valores (la llamamos* ***no válida****) y falsa para todos los valores (la llamamos* ***no satisfactible****).*

1. Evaluá las siguientes expresiones, subrayando la subexpresión resuelta en cada paso justificado. Luego usá un intérprete de `haskell` (por ej. `ghci` o `hugs`) para verificar los resultados.

   Por ejemplo:

   $$\begin{aligned} & \underline{10 * (6+7)} \\ =\ & \{ \text{ def. de } + \} \\ & \underline{10 * 13} \\ =\ & \{ \text{ def. de } * \} \\ & 130 \end{aligned}$$

   a) $5 * 27 + 2 * (5 + 7 * 25 + (4 - \sqrt[3]{27})) * 2$

   b) $\frac{2*\sqrt{4}}{2^2 - \sqrt{\sqrt{16}}}$

   c) $(((\sqrt{25} * 2^3)\,\text{máx}(4^2)) + 10)\,\text{mín}\,22$

   En `haskell` los distintos operadores se pueden escribir así:

x^2	`x^2`
x^n	`x^n` (con n entero)
x^p	`x**p` (con p racional)
$\sqrt{x}$	`sqrt x`
$\sqrt[r]{x}$	`x**(1/r)`
$x\,\text{máx}\,y$	`x ‘max‘ y` (o bien `max x y`)
$x\,\text{mín}\,y$	`x ‘min‘ y` (o bien `min x y`)
$x = y$	`x == y`
$x \leqslant y$	`x <= y`
$x \geqslant y$	`x >= y`

2. ¿Qué valor de la variable x *satisface* las siguientes ecuaciones, es decir, qué valor hace que las fórmulas sean verdaderas? Por ejemplo para la fórmula $6 * x + 8 = x + 3$:

   $$\begin{aligned} & 6 * x + 8 = x + 3 \\ \equiv\ & \{ \text{ restar } x \text{ en ambos miembros } \} \\ & \underline{6 * x} + 8\ \underline{-x} = \underline{\underline{x}} + 3\ \underline{\underline{-x}} \\ \equiv\ & \{ \text{ aritmética } \} \\ & 5 * x + 8 = 3 \\ \equiv\ & \{ \text{ restar 8 en ambos miembros } \} \\ & 5 * x + \underline{8 - 8} = \underline{\underline{3 - 8}} \\ \equiv\ & \{ \text{ aritmética } \} \\ & 5 * x = -5 \\ \equiv\ & \{ \text{ dividir por 5 en ambos miembros, aritmética } \} \\ & x = -1 \end{aligned}$$

   Luego la respuesta es que solamente $x = -1$ hace verdadera la fórmula $6 * x + 8 = x + 3$. En otros casos, puede haber más de un valor para x que satisfaga la fórmula; en ese caso hay que encontrarlos a todos. *Notá que cuando realizamos dos pasos a la vez, utilizamos un subrayado diferente para cada subexpresión involucrada.*

a) $x^2 - 2 * x = 0$

b) $x^2 - 1 = 0$

c) $4 * x + 14 = 2 * (2 * x + 5) + 4$

d) $4 * x + 3 = 2 * (2 * x + 5) + 3$

¿Cómo se interpretan los resultados de los items *c*) y *d*)?

3. Un *predicado* sobre `x` es una función que para cada valor de `x` devuelve `Verdadero` o `Falso`. Definí en `haskell` las ecuaciones del ejercicio anterior como predicados sobre `x` y verificá los resultados obtenidos, en los casos que sea posible. Recordá que es necesario utilizar un nombre para cada ecuación y nombrar las variables. Por ejemplo, en `haskell` , en un archivo de texto que llamaremos `Ej3.hs`, se puede definir el predicado `p`

```
p x = (6 * x + 8 == x + 3)
```

y evaluando p en `ghci` con diferentes *argumentos* obtenemos:

```
Prelude> :load Ej3.hs
Main> p (-1)
True
Main> p 5
False
```

4. Demostrá que las siguientes (in)ecuaciones son *válidas*, es decir que las fórmulas son verdaderas para cualquier valor que tome la variable x. Por ejemplo:

$$
\begin{array}{ll}
 & 4 * x + 14 = \underline{2 * (2 * x + 5)} + 4 \\
\equiv & \{ \text{ distributividad de } * \text{ con } +, \text{ def. de } * \ \} \\
 & 4 * x + 14 = 4 * x + \underline{10 + 4} \\
\equiv & \{ \text{ def. de } + \ \} \\
 & 4 * x + 14 = 4 * x + 14 \\
\equiv & \{ \text{ restar } 4 * x + 14 \text{ en ambos miembros } \} \\
 & \underline{4 * x + 14 - (4 * x + 14) = 4 * x + 14 - (4 * x + 14)} \\
\equiv & \{ \text{ reflexibidad de } = (x = x) \ \} \\
 & \textit{True}
\end{array}
$$

Podés ver otro ejemplo en la pág. 7 del libro.

a) $(x - 1) * (x + 3) * (x - 2) = x^3 + (-7) * x + 6$

b) $\sqrt{3} + \sqrt{11} > \sqrt{5} + \sqrt{7}$

c) $\frac{x^2}{2} + \sqrt{24} > 3 * x$

5. Las siguientes (in)ecuaciones no son válidas, es decir que para al menos algún valor de la variable x cada fórmula es falsa. Justificá dando un *contraejemplo* o una demostración de que es equivalente a `Falso` cuando sea posible. Notá que ambas formas son correctas para demostrar su no-validez. Además, indicá cuáles son satisfactibles y qué valores las satisfacen.

Por ejemplo para la ecuación $3 * x + 1 = 3 * (x + 1)$.

Contraejemplo: $x = 5$ *falsifica* la fórmula ya que

$$
\begin{array}{ll}
 & 3 * x + 1 = \underline{3 * (x + 1)} \\
\Rightarrow & \{ \text{ tomando } x = 5 \ \} \\
 & \underline{3 * 5} + 1 = \underline{3 * 5} + 3 \\
\equiv & \{ \text{ def. de } * \ \} \\
 & \underline{15 + 1} = \underline{15 + 3} \\
\equiv & \{ \text{ def. de } + \ \} \\
 & 16 = 18 \\
\equiv & \{ \text{ igualdad de números } \} \\
 & \textit{False}
\end{array}
$$

Demostración de que equivale a `Falso`:

$$
\begin{aligned}
& 3 * x + 1 = \underline{3 * (x + 1)} \\
\equiv & \{ \text{ distributividad de } * \text{ con } + \} \\
& \underline{3 * x} + 1 = \underline{3 * x} + 3 \\
\equiv & \{ \text{ restamos } 3 * x \text{ en cada miembro } \} \\
& 1 = 3 \\
\equiv & \{ \text{ igualdad de enteros } \} \\
& \mathit{False}
\end{aligned}
$$

Esta ecuación no es *satistactible*. Es decir, ningún valor asignado a la variable x la puede hacer verdadera.

a) $x + (y * z) = (x + y) * (x + z)$

b) $x^2 + 2 * x + 2 = (x + 1)^2$

c) $\sqrt{3} + \sqrt{13} < \sqrt{5} + \sqrt{7}$

d) $\frac{x^2+52*x}{169} - 2 * x > -121$

6. Teniendo en cuenta los conceptos de *validez* y *satistactibilidad*

   a) ¿Qué diferencia existe entre dar un ejemplo y demostrar la validez de una fórmula?

   b) ¿Qué diferencia existe entre dar un contraejemplo y demostrar que una fórmula es equivalente a `Falso`? ¿Cuándo se puede utilizar cada *estrategia*?

7. Decidí si son *válidas* o *no válidas* y *satisfactibles* o *no satisfactibles* las siguientes (in)ecuaciones. Justificá en cada caso.

   a) $\sqrt{x} + \sqrt{y} = \sqrt{x + y}$

   b) $(a - b) * (a + b) * ((a + b)^2 - 2 * a * b) = a^4 - b^4$

   c) $\sqrt{5 + 7 + 11} > \sqrt{5} + \sqrt{7}$

En los ejercicios 2, 4, 5 y 7 trabajamos sobre (in)ecuaciones centrándonos en diferentes aspectos: algunas veces buscando valores que satisfacen las ecuaciones, otras demostrando que las ecuaciones son siempre verdaderas independientemente de los valores que tomen las variables. En el primer caso decimos que la fórmula es ***satisfactible****; en el segundo, decimos que la fórmula es* ***valida****. Es importante tener bien presente estas diferencias. Nuestro principal interés es distinguir cuando una fórmula es válida y cuando no. Como la validez es independiente de los valores de las variables involucradas, si queremos demostrar la validez de una fórmula no podemos hacer* ***ninguna suposición*** *sobre los valores de las variables. Por el contrario, cuando queremos demostrar que una fórmula es no valida es suficiente con encontrar al menos un valor que haga que la fórmula sea falsa: esto es un contraejemplo. Algunas veces además es posible demostrar directamente que la fórmula es falsa para* ***todos los valores*** *de las variables. En este caso decimos que la fórmula es* ***no satisfactible****, porque no existe ningún valor posible que haga que sea verdadera.*

8. De ejemplos de una fórmula:

   a) válida (y por lo tanto satisfactible).

   b) satisfactible pero no válida.

   c) no satisfactible (y por lo tanto no válida).

Precedencia y tipado

*La **precedencia** de los operadores nos permite escribir fórmulas grandes de manera simple y más legible. Cuando un operador tiene mayor precedencia que otro, podemos escribir una fórmula que involucra a ambos sin necesidad de poner paréntesis, y a pesar de esto, la fórmula tiene un sentido único. Un ejemplo conocido por todos es el caso de la suma respecto a la multiplicación. El operador $*$ tiene mayor precedencia que $+$ y por ello es que normalmente interpretamos la expresión $2 + 4 * 3$ como $2 + (4 * 3)$. Esta regla es la que nos permitía en la primaria "separar en términos" una expresión algebraica. Al mismo tiempo, y por la misma regla, sabemos que no es lo mismo la expresión $(2 + 4) * 3$ que $2 + 4 * 3$. No siempre los paréntesis pueden sacarse indiscriminadamente sin cambiar el sentido de la expresión.*

tabla de precedencia	*más arriba → mayor precedencia*
$\sqrt{}, (\cdot)^2$	*raíces y potencias*
$*, /$	*producto y división*
máx, mín	*máximo y mínimo*
$+, -$	*suma y resta*
$=, <, \leqslant, > \geqslant$	*operadores de comparación*

*De la misma manera, cuando un operador **asociativo** se aplica múltiples veces es posible eliminar paréntesis, ya que el orden en que se efectúa la operación no altera el resultado. Por ejemplo $(2+4)+7$ es igual a $2+(4+7)$ y por lo tanto podemos escribir simplemente $2+4+7$.*

Los operadores $+$, $*$, mín, máx son asociativos.

*Por otro lado, la noción de **tipado** nos permite distinguir expresiones que están "bien escritas", es decir que tienen sentido, que representan algún valor. El tipo de un operador o función nos dice cual es la clase de valores que toma y cual es la clase del valor que devuelve. En las expresiones algebraicas en general todos los valores son de tipo número, que denotamos con Num (y los **subtipos** Nat, Int, ...). Como ejemplo tomemos el caso de la suma: tanto a la izquierda del símbolo "+" como a la derecha debe haber expresiones que representen números, y el resultado total también es un número.*

Además del tipo Num utilizaremos el tipo de los valores booleanos, que denotamos con Bool. Este tipo incluye las constantes True y False que representan las nociones de verdadero y falso respectivamente. Los valores booleanos son importantes porque son el resultado de los operadores de comparación $=, <, \leqslant, >, \geqslant$.

Las nociones de precedencia y tipado son importantes para asegurar que escribimos expresiones que tienen un sentido único y bien definido. Los siguientes ejercicios tratan sobre estas nociones en el marco de las expresiones algebraicas.

9. Sacá todos los paréntesis que sean *superfluos* según las reglas de precedencia y asociatividad de los operadores aritméticos. Por ejemplo:

$$
\begin{array}{l}
\quad (8-6)*x = \underline{(6*(x^2))} + 3 \\
\equiv \{ \text{ precedencia de } * \text{ por sobre } + \} \\
\quad (8-6)*x = 6*\underline{(x^2)} + 3 \\
\equiv \{ \text{ precedencia de } {}^2 \text{ sobre } * \} \\
\quad (8-6)*x = 6*x^2 + 3
\end{array}
$$

   a) $(-(5+x) + ((3*6)/(4*5))) * (8*5)$

   b) $(((2)^2 + 5) - (4*2)/(4)) + 1 = ((5*x)/8) + (3*5)^{(3^2)}$

10. Introducí paréntesis para hacer *explícita* la precedencia.

   a) $5 * 3^2 + 4^5 \geqslant 7 - 7 + 3$

   b) $3 + 4^0 * x = 4$

11. Escribí el *tipo* de los siguientes operadores y funciones: $*$, $/$, ^{2}, $\leqslant$, $\geqslant$, $=$. ¿Cuál es el tipo de esos mismos operadores en `haskell`? ¿Qué sucede con el operador $-$?

    Como ejemplo presentamos el caso del operador $+$. Este operador toma dos números y devuelve otro número. Podemos escribir su tipo en *notación funcional* listando el tipo de los parámetros y a continuación el tipo resultado:

$$+ : Num \to Num \to Num$$

Otra forma de escribir el tipo de este operador, útil para armar el árbol de tipado de una expresión, es en *notación de árbol*:

$$\frac{Num + Num}{Num}$$

En `haskell` el operador `(+)` tiene el mismo tipo. En `ghci` podemos corroborarlo de la siguiente manera:

```
Main> :t (+)
(+) :: Num a => a -> a -> a
```

Esta respuesta se interpreta de la siguiente manera. El tipo del operador `(+)` es lo que está a la derecha de `=>`, es decir $a \to a \to a$, donde la variable a indica cualquier tipo. Por otro lado, lo que que esta a la izquierda de `=>` nos indica que el tipo a es un subtipo de *Num*, es decir, *Nat*, *Int*.

Un árbol de tipado es una herramienta que nos permite verificar que una expresión está bien escrita, es decir, que respeta la sintaxis del lenguaje. Consiste en ir armando los tipos de cada subexpresión de la manera en que se iría resolviendo, es decir teniendo en cuenta la presedencia de los distintos operadores.

12. Completar los siguientes árboles de tipado con los tipos que corresponden a cada operación.

a)

$$\frac{\dfrac{\dfrac{(\ x\ +\ (\underline{2}\ /\ \underline{y}))}{+} \geqslant \dfrac{(\ x\ +\ \underline{1}\ )}{+}}{/}}{+}$$

b)

$$\frac{\dfrac{\dfrac{(\underline{8}\ -\ \underline{6})}{-} * \underline{x}}{*} = \dfrac{(\underline{6}\ *\ \underline{x^2})}{*} + \underline{3}}{=}$$

13. ¿Están bien escritas las siguientes expresiones? Justificá a través de un árbol de tipado utilizando las reglas que escribiste en el ejercicio 11 y eligiendo el tipo para cada variable. Para evitar errores, hace explícita la precedencia introduciendo paréntesis, y construí una tabla del tipo de las variables. Por ejemplo, para la expresión $x + 2/y \geqslant x + 1$ podemos armar el árbol de tipado de la siguiente manera:

Primero escribimos el tipo de las cosas sobre las que estamos seguros: las constantes.

$$(x + (\underset{Num}{2} \,/\, y)) \geqslant (x + \underset{Num}{1})$$

Luego asignamos tipos a las variables, según las reglas de tipado de los operadores que operan sobre ellas. Luego escribimos el tipo del resultado de estos operadores.

$$\underset{Num}{(\ x\ + (\underset{Num}{2} \,/\, \underset{Num}{y}))} \geqslant \underset{Num}{(\underset{Num}{x} + \underset{Num}{1})}$$

Var	Tipo
x	Num
y	Num

Sucesivamente completamos el resto de los operadores, hasta dar el tipo de la expresión completa

$$\frac{\underset{Num}{(\ x\ + (\underset{Num}{2} \,/\, \underset{Num}{y}))} \geqslant \underset{Num}{(\underset{Num}{x} + \underset{Num}{1})}}{Bool}$$

Var	Tipo
x	Num
y	Num

a) $(x-1)*(x+3)*(x-2) = x^2 + -7*x + 6$

b) $[2*(2*x/y) - 2 = 4*x/y - 2] \leqslant 4*x/y$

c) $y - 1 + (16 * \sqrt{x})/y$

Operadores máximo y valor absoluto

Hasta aquí hemos venido utilizando en las demostraciones una colección de propiedades aritméticas que no hemos listado exhaustivamente. En este sentido hemos sido informales al permitir utilizar reglas que si bien todos sabemos son válidas, no forman un conjunto fijo de axiomas. De aquí en más extenderemos paulatinamente el lenguaje de nuestras fórmulas, incorporando funciones, tipos complejos, como listas, valores booleanos, etc. Cada una de estas extensiones irá acompañada de un conjunto de axiomas y definiciones que nos permitirán razonar sobre las fórmulas, y servirán como justificaciones en los pasos de las demostraciones. Tal es el caso del operador de máximo, cuyos axiomas se listan en la sección 1.6 del libro.

El objetivo de estos ejercicios es comenzar a realizar las demostraciones de una manera más ***formal****, utilizando solo los axiomas permitidos y propiedades aritméticas elementales. El conjunto de axiomas para los siguientes ejercicios son los del operador máx (pág. 16 del libro)*

14. Demostrá justificando cada paso con un axioma, las siguientes fórmulas que involucran al máximo. Como ejemplo te referimos a la pág. 16 del libro.

    a) $x \operatorname{máx}(-x) + y \operatorname{máx}(-y) \geqslant (x+y) \operatorname{máx} -(x+y)$

    b) $(x+z) \operatorname{máx}(y+z) + (x-z) \operatorname{máx}(y-z) \geqslant x + y$

15. El operador de valor absoluto $|\ | : Num \to Num$, que dado un número positivo o negativo devuelve su valor (sin signo), tiene una propiedad muy conocida llamada "desigualdad triangular", que se enuncia como sigue:

    $$|x| + |y| \geqslant |x + y|$$

    ¿Cuál es la relación entre esta propiedad y el ítem *a*) del ejercicio anterior? ¿Cómo podemos definir $|\ |$?

16. Demostrá que $|x| \geqslant 0$. Para este ejercicio, deberás haber resuelto el anterior.

Listas

A partir de esta sección extendemos el lenguaje de nuestras fórmulas con el tipo de las listas, denotado como List. Una lista (o secuencia) es una colección ordenada de valores, que deben ser todos del mismo tipo. Denotamos lista vacía con $[\,]$. *El operador* $\triangleright$ *(llamado "pegar") es fundamental (se lo denomina constructor) ya que permite construir listas arbitrarias a partir de la lista vacía.* $\triangleright$ *toma un elemento* x *(a izquierda) y una lista* xs *y devuelve una lista con primer elemento* x *seguido de los elementos de* xs. *Por ejemplo* $3 \triangleright [\,] = [3]$, *y* $1 \triangleright [2,3] = [1,2,3]$. *Para denotar listas no vacías utilizamos expresiones de la forma* $[x, y, \ldots, z]$, *que son abreviaciones de* $x \triangleright (y \triangleright \ldots \triangleright (z \triangleright [\,]))$. *Como el operador* $\triangleright$ *es asociativo a derecha, es lo mismo escribir* $x \triangleright (y \triangleright \ldots \triangleright (z \triangleright [\,]))$ *que* $x \triangleright y \triangleright \ldots \triangleright z \triangleright [\,]$. *Otros operadores sobre listas son los siguientes:*

- $\triangleleft$ *toma una lista* xs *(a izquierda) y un elemento* y *devuelve una lista con todos los elemento de* xs *seguidos por* x *como último elemento. Ej:* $[1,2] \triangleleft 3 = [1,2,3]$. *Este operador, llamado "pegar a izquierda", es asociativo a izquierda, luego es lo mismo* $([\,] \triangleleft z) \ldots \triangleleft y) \triangleleft x$ *que* $[\,] \triangleleft z \ldots \triangleleft y \triangleleft x$.
- $\#$, *llamado cardinal, toma una lista* xs *y devuelve su cantidad de elementos. Ej:* $\#.[1,2,0,5] = 4$
- $.$ *toma una lista* xs *(a izquierda) y un natural* n *que indica una posición, y devuelve el elemento de la lista que se encuentra en la posición* n *(contando a partir de la posición* 0*). Ej:* $[1,3,3,6,2,3,4,5].4 = 2$. *Este operador, llamado índice, asocia a izquierda, por lo tanto* $xs.n.m$ *se interpreta como* $(xs.n).m$.
- $\uparrow$ *toma una lista* xs *(a izquierda) y un natural* n *que indica una cantidad, y devuelve la sublista con los primeros* n *elementos de* xs. *Ej:* $[1,2,3,4,5,6] \uparrow 2 = [1,2]$. *Este operador, llamado tomar, asocia a izquierda, por lo tanto* $xs \uparrow n \uparrow m$ *se interpreta como* $(xs \uparrow n) \uparrow m$.
- $\downarrow$ *toma una lista* xs *(a izquierda) y un natural* n *que indica una cantidad, y devuelve la sublista sin los primeros* n *elementos de* xs. *Ej:* $[1,2,3,4,5,6] \downarrow 2 = [3,4,5,6]$. *Este operador, llamado tirar, se comporta igual al anterior, interpretando* $xs \downarrow n \downarrow m$ *como* $(xs \downarrow n) \downarrow m$.
- $+\!\!+$ *toma una lista* xs *(a izquierda) y otra* ys, *y devuelve la lista con todos los elementos de* xs *seguidos de los elementos de* ys. *Ej:* $[1,2,4] +\!\!+ [1,0,7] = [1,2,4,1,0,7]$. *Este operador, llamado concatenación, es asociativo por lo que podemos escribir sin ambigüedad expresiones sin paréntesis, como* $xs +\!\!+ ys +\!\!+ zs$.

Existen además dos funciones fundamentales sobre listas que listamos a continuación. Notar que como son funciones se hace explícita la aplicación de función con el símbolo (.).

- *head, llamada cabeza, toma una lista* xs *y devuelve su primer elemento. Ej:* $head.[1,2,3] = 1$
- *tail, llamada cola, toma una lista* xs *y devuelve la sublista que resulta de eliminar el primer elemento. Ej:* $tail.[1,2,3] = [2,3]$

La aplicación de función asocia a izquierda, por lo tanto en general es necesario utilizar paréntesis para que la expresión quede bien tipada. Si se quiere escribir la expresión $tail.(tail.xs)$ *no se pueden eliminar los paréntesis, puesto que* $tail.tail.xs$ *(que se interpreta como* $(tail.tail).xs$*) no tiene sentido.*

A continuación, listamos los niveles de precedencia de estos operadores. Los que están más arriba tienen mayor precedencia. Cuando hay más de un operador en un nivel de precedencia, es necesario poner paréntesis para evitar la ambigüedad. Por ejemplo $x \triangleright xs \uparrow n$ *se interpreta como* $x \triangleright (xs \uparrow n)$, *pero la expresión* $tail.xs.n$ *no tiene sentido si no se ponen paréntesis (o bien es* $(tail.xs).m$ *o bien* $tail.(xs.n)$*).*

., #, head y tail	*aplicación de función e índice, cardinal, head y tail*
$\uparrow, \downarrow$	*tomar y tirar elementos de una lista*
$\triangleright, \triangleleft$	*pegar a derecha y pegar a izquierda*
$+\!\!+$	*concatenar dos listas*

Te recomendamos leer las secciones 7.2, 7.4, 7.5, 7.6, 7.8, 7.10, del libro para profundizar en estos conceptos.

El objetivo de los siguientes ejercicios es familiarizarse con el tipo de listas y extender el método para justificar el tipado de expresiones, considerando expresiones más complejas que las que veníamos trabajando.

17. Utilizá las definiciones intuitivas de los operadores de listas para evaluar las siguientes expresiones. Subrayá la subexpresión resuelta en cada paso justificado. Luego usá un intérprete de `haskell` para verificar los resultados. Por ejemplo:

$$\begin{array}{ll} & \underline{[23,45,6].(head.[1,2,3,10,34])} \\ \equiv & \{ \text{ def. de } head \text{ } \} \\ & \underline{[23,45,6].1} \\ \equiv & \{ \text{ def. de . } \} \\ & 45 \end{array}$$

En `haskell` los distintos operadores se pueden escribir así:

$head.xs$	`head xs`
$tail.xs$	`tail xs`
$x \triangleright xs$	`x:xs`
$xs \triangleleft x$	`xs ++ [x]`
$xs \uparrow n$	`take n xs`
$xs \downarrow n$	`drop n xs`
$xs +\!\!+ ys$	`xs ++ ys`
$\#xs$	`length xs`
$xs.n$	`xs !! n`

a) $\#[5,6,7]$

b) $[5,3,57].1$

c) $[0,11,2,5] \triangleright [\,]$

d) $[5,6,7] \uparrow 2$

e) $[5,6,7] \downarrow 2$

f) $head.(0 \triangleright [1,2,3])$.

g) $([3,5] \triangleright [[14,16],[1,3]]).1$

h) $([1,2] +\!\!+ [3,4]) \triangleleft (2+3)$.

i) $(([[1]] +\!\!+ [[2]]) \triangleleft [3]) \uparrow 2$.

j) $(([[\,]] +\!\!+ [[\,]]) \triangleleft ([\,] +\!\!+ [\,])) \uparrow \#([\,] \triangleright [[\,]])$.

18. Teniendo en cuenta la definición intuitiva de los operadores de listas de la introducción a esta sección, escribí las reglas de tipado de cada uno de ellos. Por ejemplo, el operador *head* toma una lista y devuelve el primer elemento de ella. La lista puede contener cualquier otro tipo, ya sean números, valores booleanos, otras listas, etc. Para denotar esta situación utilizamos variables (en mayúsculas). Entonces podemos decir que el operador *head* toma una lista de tipo $[A]$, donde la variable A representa cualquier tipo $(Num, Bool, [Num], \ldots)$ y devuelve un elemento de esa lista, por lo tanto debe ser un elemento de tipo A. Esto lo escribimos en *notación funcional* (izq.) o en *notación de árbol* (der.):

$$head : [A] \to A \qquad\qquad \frac{head.[A]}{A}$$

19. Decidí si las siguientes expresiones están bien escritas, agregando paréntesis para hacer explícita la precedencia y la asociatividad, y justificando con el árbol del tipado correspondiente. Usá un intérprete de `haskell` para verificar los resultados.

a) $[3,5].2 \triangleright ([14,16] \triangleleft \sqrt{2})$.
b) $-45 \triangleright [1,2,3]$.
c) $[1,5,False]$.
d) $0 \triangleleft [1,2,3]$.
e) $([1,2] +\!\!+ [3,4]) \triangleleft 5$.
f) $([1] +\!\!+ [2]) \triangleleft [3]$.
g) $head.[[5]]$.
h) $head.[True, False] +\!\!+ [False]$.
i) $(\#[True, False]) \triangleright [3,4]$.
j) $[[0,1],[False, False]]$.
k) $tail.([[2],[4,5]].0)$.
l) $([1,2] \uparrow 3) \downarrow 2$.

20. Decidí si es posible asignar tipos a las variables $x, y, z, \ldots$ de forma que las expresiones queden bien tipadas. Justificá con un árbol de tipado y una tabla de tipos de las variables.

a) $x \triangleright y \triangleright z$.
b) $x \triangleright (y \triangleleft z)$.
c) $(x \triangleright y) \triangleright z$.
d) $(x +\!\!+ y) \triangleright (z +\!\!+ w)$.
e) $head.xs.n = head.(xs.n)$.
f) $x \triangleright [x]$.
g) $x \triangleright [[x]]$.
h) $x \triangleright [[True]] \triangleright y$.
i) $x \triangleright [[True]] \triangleright y$.
j) $xs \uparrow ys.4$.
k) $xs \uparrow [1,2].0$.
l) $xs \downarrow xs.2$.

Funciones, inducción y recursión

Una técnica poderosa para demostrar propiedades sobre un dominio inductivo, como son los naturales o las listas, es usar el ***principio de inducción****. La idea que rige este principio consiste en demostrar dos cosas. Por un lado verificar que la propiedad se satisface para los elementos "más chicos" del dominio (por ejemplo el* 0*, o la lista* $[\,]$*). Por otro lado demostrar para un elemento arbitrario del dominio (por ejemplo* $n+1$*, o la lista* $x \triangleright xs$*) que si suponemos que la propiedad es cierta para todos los elementos más chicos que él (por ejemplo* n*, o* xs*), 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 que la propiedad es satisfecha por todos los elementos del dominio, y por lo tanto es válida.*

El objetivo de los siguientes ejercicios es introducirnos en la ***programación funcional****, es decir, al desarrollo de programas como funciones (generalmente recursivas), y a la demostración (por inducción) de propiedades sobre estos programas.*

21. Definí las funciones simples que describimos a continuación, luego implementálas en `haskell` . Por ejemplo:

    Enunciado: *signo* : $Int \to Int$, que dado un entero retorna su signo, de la siguiente forma: retorna 1 si x es positivo, -1 si es negativo y 0 en cualquier otro caso.

    Solución:

    $sgn : Num \to Num$
    $sgn.x \doteq (\; 0 < x \to 1$
    $\quad \Box \, x < 0 \to -1$
    $\quad \Box \, x = 0 \to 0$
    $\quad)$

    ```
    sgn :: Int -> Int
    sgn x | 0<x   = 1
          | x<0   = -1
          | x==0  = 0
    ```

    En `haskell` los conectivos para las condiciones se pueden escribir así:

$\wedge$	`&&`
$\vee$	`||`
$\neg$	`not`

    a) *entre0y9* : $Int \to Bool$, que dado un entero devuelve *True* si el entero se encuentra entre 0 y 9.

    b) *segundo3* : $(Int, Int, Int) \to Int$, que dada una terna de enteros devuelve su segundo elemento.

    c) *mayor3* : $(Int, Int, Int) \to (Bool, Bool, Bool)$, que dada una una terna de enteros devuelve una terna de valores booleanos que indica si cada uno de los enteros es mayor que 3.
    Por ejemplo: $mayor3.(1, 4, 3) = (False, True, False)$; $mayor3.[5, 1984, 6] = (True, True, True)$

    d) *ordena* : $(Int, Int) \to (Int, Int)$, que dados dos enteros los ordena de menor a mayor.

    e) *rangoPrecio* : $Int \to String$, que dado un número que representa el precio de una computadora, retorne "muy barato" si el precio es menor a 2000, "demasiado caro" si el precio es mayor que 5000, "hay que verlo bien" si el precio está entre 2000 y 5000, y "esto no puede ser!" si el precio es negativo.

    f) *absoluto* : $Int \to Int$, que dado un entero retorne su valor absoluto.

    g) *esMultiplo2* : $Int \to Bool$, que dado un entero n devuelve *True* si n es múltiplo de 2.
    Ayuda: usar `mod`, el operador que devuelve el resto de la división.

    h) *rangoPrecioParametrizado* : $Int \to (Int, Int) \to String$ que dado un número x, que representa el precio de un producto, y un par $(menor, mayor)$ que represente el rango de precios que uno espera encontrar, retorne "muy barato" si x está por debajo del rango, "demasiado caro" si está por arriba del rango, "hay que verlo bien" si el precio está en el rango, y "esto no puede ser!" si x es negativo.

22. Definí recursivamente los operadores básicos de listas: $\#, \, . \, , \triangleleft, \uparrow, \downarrow, +\!\!+$. Para los operadores $\uparrow$, $\downarrow$ y . deberás hacer recursión en ambos parámetros, en el parámetro lista y en el parámetro numérico.

23. Demostrá por inducción las siguientes propiedades. Para ello necesitarás las definiciones recursivas de los operadores del ejercicio anterior.

    a) $xs +\!\!+ [\,] = xs$ (la lista vacía es el elemento neutro de la concatenación)

    b) $\#xs \geqslant 0$

    c) $xs +\!\!+ (ys +\!\!+ zs) = (xs +\!\!+ ys) +\!\!+ zs$ (la concatenación es asociativa)

d) $(xs +\!\!+ ys) \uparrow \#xs = xs$

e) $(xs +\!\!+ ys) \downarrow \#xs = ys$

f) $xs +\!\!+ (y \triangleright ys) = (xs \triangleleft y) +\!\!+ ys$

g) $xs +\!\!+ (ys \triangleleft y) = (xs +\!\!+ ys) \triangleleft y$

24. Definí funciones por recursión y/o composición para cada una de las siguientes descripciones. Luego evaluá manualmente la función para los valores de cada ejemplo justificando cada paso realizado. Programálas en `haskell` y verificá los resultados obtenidos. Por ejemplo:

Enunciado: *duplica* : $[Int] \to [Int]$, que dada una lista de enteros duplica cada uno de sus elementos.

Por ejemplo $duplica.[2, 5, 12] = [4, 10, 24]$

Solución:

$$\begin{aligned} duplica &: [Num] \to [Num] \\ duplica.[\,] &\doteq [\,] \\ duplica.(x \triangleright xs) &\doteq (x * 2) \triangleright duplica.xs \end{aligned}$$

```
duplica :: [\Int] -> [\Int]
duplica [] = []
duplica (x:xs) = (x*2): duplica xs
```

$$\begin{aligned} & \underline{duplica.[2, 5, 12]} \\ = & \{ \text{ def. de } duplica \text{ caso } (x \triangleright xs) \ \} \\ & \underline{(2 * 2)} \triangleright duplica.[5, 12] \\ = & \{ \text{ aritmética } \} \\ & 4 \triangleright \underline{duplica.[5, 12]} \\ = & \{ \text{ def. de } duplica \text{ caso } (x \triangleright xs) \ \} \\ & 4 \triangleright \underline{(5 * 2)} \triangleright duplica.[12] \\ = & \{ \text{ aritmética } \} \\ & 4 \triangleright 10 \triangleright \underline{duplica.[12]} \\ = & \{ \text{ def. de } duplica \text{ caso } (x \triangleright xs) \ \} \\ & 4 \triangleright 10 \triangleright \underline{(12 * 2)} \triangleright duplica.[\,] \\ = & \{ \text{ aritmética } \} \\ & 4 \triangleright 10 \triangleright 24 \triangleright \underline{duplica.[\,]} \\ = & \{ \text{ def. de } duplica \text{ caso } [\,] \ \} \\ & 4 \triangleright 10 \triangleright 24 \triangleright [\,] \\ = & \{ \text{ notación } \} \\ & [4, 10, 24] \end{aligned}$$

```
Main> duplica [2,5,12]
[4,10,24]
```

a) *multiplica* : $Int \to [Int] \to [Int]$, que dado un número n y una lista, multiplica cada uno de los elementos por n.

Por ejemplo: $multiplica.3.[3, 0, -2] = [9, 0, -6]$

¿Qué relación existe con la función anterior?

b) *maximo* : $[Int] \to Int$, que calcula el máximo elemento de una lista de enteros.

Por ejemplo: $maximo.[2, 5, 1, 7, 3] = 7$

c) *esMultiploLista* : $Int \to [Int] \to [Bool]$, que dado un entero n y una lista de enteros xs devuelve una lista de booleanos que indica si n es múltiplo de cada uno de los elementos de xs.

Por ejemplo: $esMultiploLista.6.[2, 3, 5] = [True, True, False]$

d) *sum* : $[Num] \to Num$, que toma una lista de números y devuelve la suma de ellos.

Por ejemplo: $sum.[1, 2, 3] = 6$

e) *prod* : $[Num] \to Num$, que toma una lista de números y devuelve el producto de ellos.

Por ejemplo: $prod.[1, 2, 3, 4] = 24$

f) *promedio* : $[Real] \to Real$, que calcula el valor promedio de los valores de una lista de números reales.

Por ejemplo: $promedio.[6, 7, 9, 4, 10] = 7, 2$

g) *sumaPares* : $[(Num, Num)] \to Num$, que dada una lista de pares de números, devuelve la sumatoria de todos los números de todos los pares.

Por ejemplo: $sumaPares.[(1, 2)(7, 8)(11, 0)] = 29$

h) *todos0y1* : $[Int] \to Bool$, que dada una lista devuelve *True* si ésta consiste sólo de 0s y 1s.
Por ejemplo: $todos0y1.[1,0,1,2,0,1] = False$, $todos0y1.[1,0,1,0,0,1] = True$

i) *todosMenores10* : $[Int] \to Bool$, que dada una lista devuelve *True* si ésta consiste sólo de números menores que 10.
Por ejemplo: $todosMenores10.[2,-3,-5,4,,-11] = True$, $todosMenores10.[2,-4,3,11] = False$

j) *hay0* : $[Int] \to Bool$, que dada una lista decide si existe algún 0 en ella.
Por ejemplo: $hay0.[1,2] = False$, $hay0.[1,4,0,5] = True$.

k) *soloPares* : $[Int] \to [Int]$, que dada una lista de enteros xs devuelve una lista sólo con los números pares contenidos en xs, en el mismo orden y con las mismas repeticiones (si las hubiera).
Por ejemplo: $soloPares.[0,1,2,3,4,5,2,3,4] = [0,2,4,2,4]$

l) *quitar0s* : $[Int] \to [Int]$ que dada una lista de enteros devuelve una la lista pero quitando todos los ceros.
Por ejemplo $quitar0s.[2,0,3,4] = [2,3,4]$

m) *ultimo* : $[A] \to A$, que devuelve el último elemento de una lista.
Por ejemplo: $ultimo.[10,5,3,1] = 1$

n) *inicio* : $[A] \to A$, que devuelve todos los elementos de la lista menos el último.
Por ejemplo: $ultimo.[10,5,3,1] = [10,5,3]$

ñ) *pares* : $[A] \to [A] \to [(A,A)]$, que toma dos listas y devuelve una lista de pares, tal que el n-ésimo elemento es el par de los n-ésimos elementos de cada una de las listas.
Por ejemplo: $pares.[0,1,2,3,4].[10,20,30] = [(0,10),(1,20),(2,30)]$

o) *listasIguales* : $[A] \to [A] \to Bool$, que determina si dos listas son iguales, es decir, contienen los mismos elementos en las mismas posiciones respectivamente.
Por ejemplo: $listasIguales.[1,2,3].[1,2,3] = True$, $listasIguales.[1,2,3,4].[1,3,2,4] = False$

25. Considerando la función $sum : [Num] \to Num$ del ejercicio 24, demostrá que:

$$sum.(xs +\!\!+ ys) = sum.xs + sum.ys$$

26. Considerando la función $repetir : Nat \to Num \to [Num]$, que construye una lista de un mismo número repetido cierta cantidad de veces, definida recursivamente como:

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

demostrá que $\#repetir.n.x = n$.

27. Considerando las funciones *sum* y *duplica* del ejercicio 24, demostrá que:

$$sum.(duplica.xs) = 2 * sum.xs$$

28. Considerando la función $concat : [[A]] \to [A]$ que toma una lista de listas y devuelve la concatenación de todas ellas, definida recursivamente como:

$$\begin{array}{rcl} concat.[\,] & \doteq & [\,] \\ concat.(xs \triangleright xss) & \doteq & xs +\!\!+ concat.xss \end{array}$$

demostrá que $concat.(xss +\!\!+ yss) = concat.xss +\!\!+ concat.yss$

29. Considerando la función $rev : [A] \to [A]$ que toma una lista y devuelve una lista con los mismos elementos pero en orden inverso, definida recursivamente como:

$$\begin{array}{rcl} rev.[\,] & \doteq & [\,] \\ rev.(x \triangleright xs) & \doteq & rev.xs \triangleleft x \end{array}$$

demostrá que $rev.(xs +\!\!+ ys) = rev.ys +\!\!+ rev.xs$

30. Determiná el tipo de cada una de las funciones definidas a continuación, asignando un tipo apropiado a cada variables. Por ejemplo, consideremos la función *más* definida como:

$$más.xs.ys \doteq xs.0 + ys.0$$

Podemos construir el árbol de tipado para el *cuerpo* de la función (lo que está a la derecha de $\doteq$):

$$\dfrac{\dfrac{\dfrac{xs \quad . \quad 0}{[Num].Nat} \quad + \quad \dfrac{ys \quad . \quad 0}{[Num].Nat}}{Num \qquad + \qquad Num}}{Num}$$

Luego las variables xs y ys son de tipo $[Num]$ y el resultado es de tipo Num. Notar además que todas las variables del cuerpo están listadas como parámetros de la función. Por lo tanto, el tipo de la función *más* es:

$$más : [Num] \to [Num] \to Num$$

Podes verificar los resultados programando en `haskell` las funciones y consultando en el interprete. Por ejemplo, la función *más* la podemos definir como

```
más xs ys = (xs !! 0) + (ys !! 0)
```

Una vez cargada la definición en el intérprete, invocando el comando `:t mas` obtenemos:

```
Main> :t más
más :: Num a => [a] -> [a] -> a
```

a) $f.xs.n.m \doteq xs.n.m$

b) $g.x.y \doteq x \triangleright y$

c) $h.x \doteq [head.x] +\!\!+ [head.x]$

d) $k.x.xs.y \doteq [x, y, xs.x + y]$

e) $duplica'.xs \doteq xs +\!\!+ xs$

f) $cuadruplica.xs \doteq duplica.(duplica.xs)$

g) $cabeza.xs \doteq (head.xs = 0)$

31. ¿Está bien escrita esta expresión? Decidí si es posible dar un tipo a la función f definida como:

$$f.i.j.N.xs.ys \doteq tail.((i \triangleright xs) \triangleright j \triangleright ys) \triangleleft (ys.N.i \triangleright j)$$

En caso afirmativo, da el tipo de la función justificando con un árbol de tipado y una tabla de tipo de las variables.

32. Considerando las funciones $bin2dec : [Int] \to Int$ y $repetirUnos : Int \to [Int]$ definidas recursivamente como:

$$\begin{array}{rcl@{\qquad}rcl} bin2dec.[\,] & \doteq & 0 & repetirUnos.0 & \doteq & [\,] \\ bin2dec.(x \triangleright xs) & \doteq & x + 2 * bin2dec.xs & repetirUnos.(n+1) & \doteq & 1 \triangleright repetirUnos.n \end{array}$$

demostrá que $bin2dec.(repetirUnos.n) = 2^n - 1$.

33. *[Torres de Hanoi]*. Se tienen tres postes numerados 0, 1 y 2, y n discos de distinto tamaño. Inicialmente se encuentran todos los discos ubicados en el poste 0, ordenados según el tamaño, con el disco más grande en la base. El problema consiste en llevar todos los discos al poste 2, con las siguientes restricciones:

a) Se puede mover sólo un disco a la vez

b) Sólo se puede mover el disco que se encuentra más arriba en algún poste.

c) No se puede colocar un disco sobre otro de menor tamaño.

Resolvé los siguientes items:

a) Sea $B = \{0, 1, 2\}$. Definí la función $hanoi : B \to B \to B \to Nat \to [(B, B)]$ tal que $hanoi.a.b.c.n$ calcule la secuencia de *movimientos* para llevar n discos desde el poste a hacia el poste c, utilizando posiblemente el poste b de forma auxiliar. Un *movimiento* es un par (B, B) cuya primer componente indica el poste de salida, y la segunda el poste de llegada. Luego programála en `haskell`.

Por ejemplo, *hanoi* para los postes 0, 1 y 2, con dos discos es: $hanoi.0.1.2.2 = [(0, 1),\ (0, 2),\ (1, 2)]$

b) Demostrá que $\#hanoi.a.b.c.n = 2^{n+1} - 1$

c) ¿En qué movimiento se cambia de poste por primera vez el disco de mayor tamaño?

Apéndice

Niveles de Precedencia

$E(x := a)$, .	sustitución y evaluación
$\sqrt{}, (\cdot)^2$	raíces y potencias
$*, /$	producto y división
máx, mín	máximo y mínimo
$+, -$	suma y resta
$=, \leqslant, \geqslant$	operadores de comparación
$\neg$	negación
$\vee \wedge$	disyunción y conjunción
$\Rightarrow \Leftarrow$	implicación y consecuencia
$\equiv \not\equiv$	equivalencia y discrepancia

Tipos de los operadores

$-x$	$- : Num \to Num$
x^y	$(-)^{(-)} : Num \to Num \to Num$
$\sqrt[r]{x}$	$\sqrt{} : Num \to Num \to Num$
$x * y$	$* : Num \to Num \to Num$
x/y	$/ : Num \to Num \to Num$
x máx y	máx : $Num \to Num \to Num$
x mín y	mín : $Num \to Num \to Num$
$x + y$	$+ : Num \to Num \to Num$
$x - y$	$- : Num \to Num \to Num$
$x > y$	$>: Num \to Num \to Bool$
$x \geqslant y$	$\geqslant: Num \to Num \to Bool$
$x < y$	$<: Num \to Num \to Bool$
$x \leqslant y$	$\leqslant: Num \to Num \to Bool$
$x = y$	$=: Num \to Num \to Bool$
$\neg p$	$\neg : Bool \to Bool$
$p \wedge q$	$\wedge : Bool \to Bool \to Bool$
$p \vee q$	$\vee : Bool \to Bool \to Bool$
$head.xs$	$head : [A] \to A$
$tail.xs$	$tail : [A] \to [A]$
$x \triangleright xs$	$\triangleright : A \to [A] \to [A]$
$xs \triangleleft x$	$\triangleleft : [A] \to A \to [A]$
$xs \uparrow n$	$\uparrow: [A] \to Int \to [A]$
$xs \downarrow n$	$\downarrow: [A] \to Int \to [A]$
$xs +\!\!+ ys$	$+\!\!+ : [A] \to [A] \to [A]$
$\#xs$	$\# : [A] \to Int$
$xs.n$	$(-).(-) : [A] \to Int \to A$

Operadores en `haskell`

x^2	`x^2`
x^n	`x^n` con n entero
x^p	`x**p`
$\sqrt{x}$	`sqrt x`
$\sqrt[r]{x}$	`x**(1/r)`
x máx y	`x ‘max‘ y` o bien `max x y`
x mín y	`x ‘min‘ y` o bien `min x y`
$x \geqslant y$	`x >= y`
$x \leqslant y$	`x <= y`
$x = y$	`x == y`
$\neg p$	`not p`
$p \wedge q$	`p && q`
$p \vee q$	`p || q`
$head.xs$	`head xs`
$tail.xs$	`tail xs`
$x \triangleright xs$	`x:xs`
$xs \triangleleft x$	`xs ++ [x]`
$xs \uparrow n$	`take n xs`
$xs \downarrow n$	`drop n xs`
$xs +\!\!+ ys$	`xs ++ ys`
$\#xs$	`length xs`
$xs.n$	`xs !! n`