SUMA 21
febrero 1996, pp. 21-33

El problema P-S de McCarthy y otros acertijos

Blas C. Ruiz
Francisco Gutiérrez
José E. Gallardo

Enunciado del problema

Un problema es elemental cuando su enunciado es comprensible para aquellos que tengan un conocimiento básico de la disciplina (por ejemplo, en matemáticas, la arit-

Blas C. Ruiz
Francisco Gutiérrez
José E. Gallardo

Enunciado del problema

Un problema es elemental cuando su enunciado es comprensible para aquellos que tengan un conocimiento básico de la disciplina (por ejemplo, en matemáticas, la aritmética); la historia de la Matemática aparece profundamente ilustrada con problemas elementales cuya solución necesita de un conocimiento profundo de alguna rama, como la Teoría Analítica de los Números (un ejemplo claro es el problema de Fermat); ello significa que ciertos problemas *elementales* no deben ser despreciados en modo alguno. Veremos pues cómo un problema elemental es resuelto, afortunadamente, con un conocimiento elemental de la Matemática, y con ayuda de un lenguaje (de programación) funcional moderno, cercano a la notación usual del matemático. El enunciado es:

Usualmente, los problemas de ingenio (puzles) han sido considerados ejemplos motivadores para la enseñanza de la programación. Muchos autores han defendido el lenguaje PROLOG como un primer acercamiento a la Programación y a las Ciencias de la Computación ya que con un conocimiento elemental de éste se pueden

Un árbitro elige dos números distintos entre 2 y 99, calcula el producto y la suma, los escribe en dos papeles distintos, entrega el papel con el producto a una persona PROD y el otro a SUM; dichas personas, después de leer los contenidos de los sobres, mantienen el siguiente diálogo:

PROD.- No puedo adivinar tu suma.

SUM.- Ya lo sabía; yo tampoco puedo adivinar tu producto.

PROD.- ¡Ajá!, entonces ya sé tu suma.

otras *versiones* del problema en las cuales un *observador* debe adivinar los datos de dos personas que establecen un diálogo del estilo anterior:

En una urna hay siete bolas, tres de color rojo, dos de color verde y dos de color negro (R,R,R,V,V,N,N); un primer jugador P coge cuatro bolas de esta urna y un segundo jugador S dos; a continuación se entabla el siguiente diálogo:

P.- No puedo adivinar tus colores.

S.- Ya lo sabía; yo tampoco puedo adivinar los tuyos.

P.- Entonces tienes una roja.

S.- Entonces tienes dos negras.

¿Qué combinación de bolas llevan ambos jugadores?

El observador puede utilizar la siguiente estrategia para resolver el problema: representar el *espacio de búsqueda* (todas las combinaciones posibles) y, con cada afirmación, cribar sucesivamente este espacio; se trata de una *estrategia hacia adelante.*

En la primera columna de la figura 1 aparece este espacio completo. De la primera afirmación, el árbitro puede deducir que los colores que tiene el primer jugador son tales que la combinación que puede tener el segundo no es única. Por ello tachamos la primera posibilidad (VVNN-RR). La segunda afirmación se descompone en dos:

(a) *ya lo sabía, y*

(b) *tampoco puedo adivinar los tuyos.*

Por *(a)* el segundo jugador tiene una combinación tal que todas las posibles combinaciones que puede tener el primero tienen varias combinaciones compatibles, por lo que eliminamos RR y todas sus compatibles (si el segundo tuviese RR el primero podría tener VVNN, y en ese caso la primera parte de la segunda afirmación no hubiese podido ser enunciada); *(b)* no elimina ninguna posibilidad. Con la tercera afirmación eliminamos todos los casos en los que alguna de las combinaciones compatibles para el segundo no tenga una bola roja. Por último, con la cuarta afirmación eliminamos la posibilidad RVVN-RN; la única posibilidad que queda (RVNN-RV) es la solución al problema.

El espacio de búsqueda para el problema *P-S* de McCarthy puede verse en las figuras 2 y 3 (con $t = 14$, si los números elegidos por el árbitro verifican $2 \leq x < y \leq t$). Usando la estrategia anterior, tras la primera afirmación, el espacio queda reducido a 31 posibilidades; tras la segunda quedan 4: (18,11), (24,11), (28,11), (30,11); la tercera afirmación no cambia las cosas y tras la cuarta no queda ninguna posibilidad; luego el problema es inconsistente para $t = 14$.

Con objeto de describir una estrategia hacia atrás, veamos en primer lugar una brevísima introducción al lenguaje que utilizaremos (pueden verse otras introducciones en (Bird, 1988; Hudak, 1992; Gallardo, 1995). Posteriormente, describiremos la estrategia hacia atrás, y otros problemas equivalentes; analizaremos seguidamente cómo tal estrategia puede ser traducida a *PROLOG*, para terminar de nuevo con la estrategia hacia adelante.

(Hamond, 1995). Exponemos como este lenguaje puede ser utilizado para modelar los conceptos elementales de la Matemática; cada concepto es representado en el lenguaje, y éste sirve de modelo denotacional. Nuestra experiencia avala que, en la mayoría de los casos (aún más si la solución a los problemas debe sugerir cierta extrategia), las soluciones descritas en un lenguaje funcional moderno son más expresivas y convincentes que las obtenidas con las versiones más recientes de PROLOG. Para mostrarlo, resolveremos en el presente trabajo un problema de ingenio, dando una esfrategia hacia atrás para su solución y su implementación en *Haskell.*

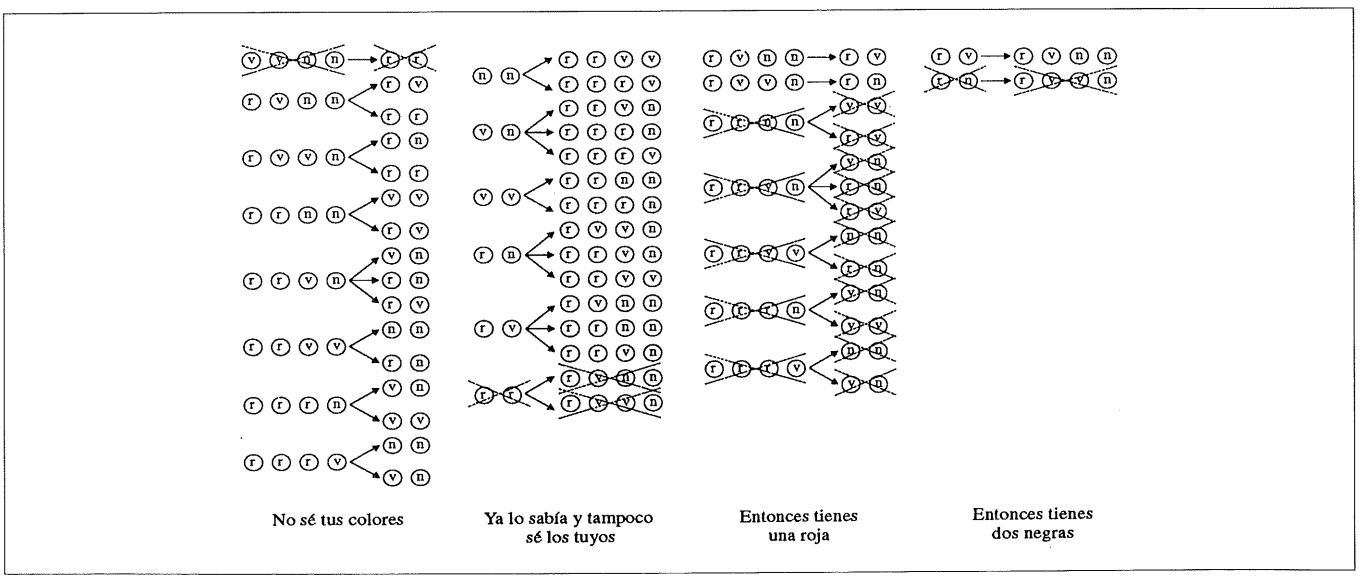

Figura 1. El juego de la urna

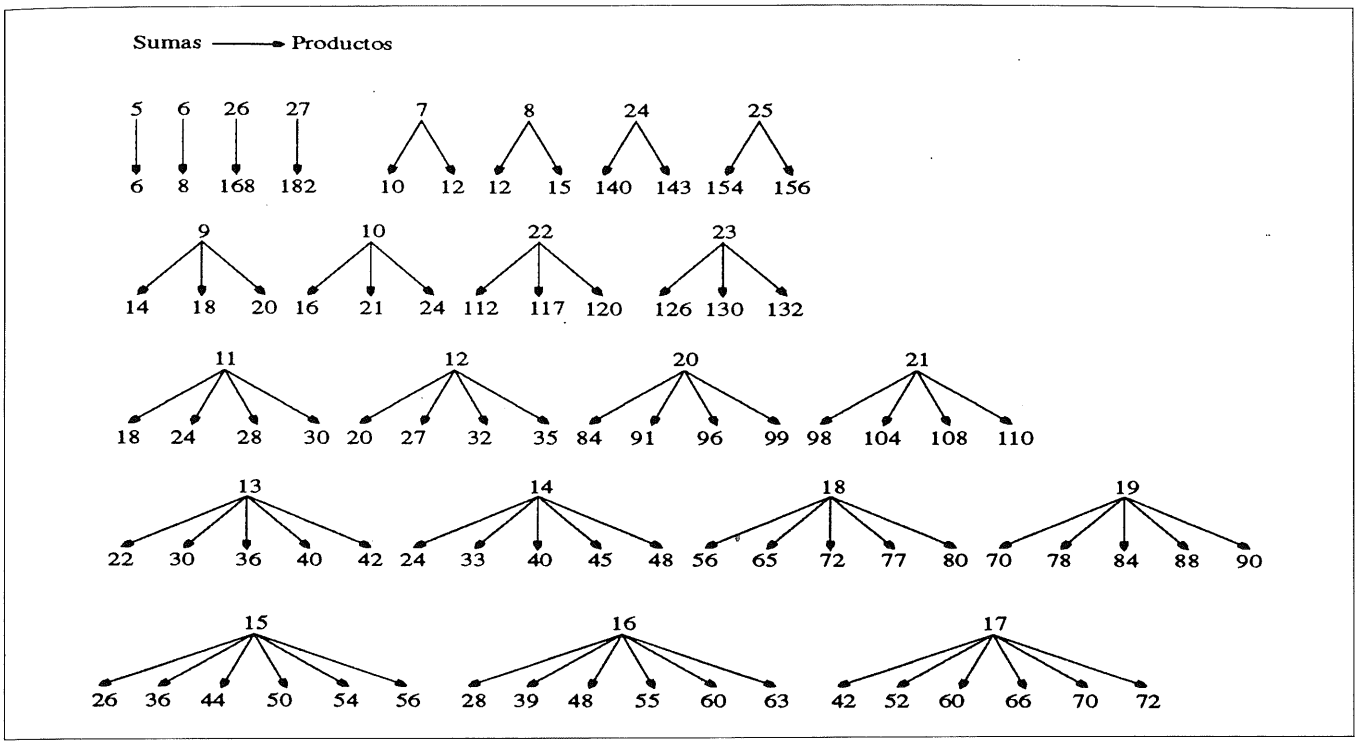

Figura 2. Productos compatibles con cada suma, para t=14

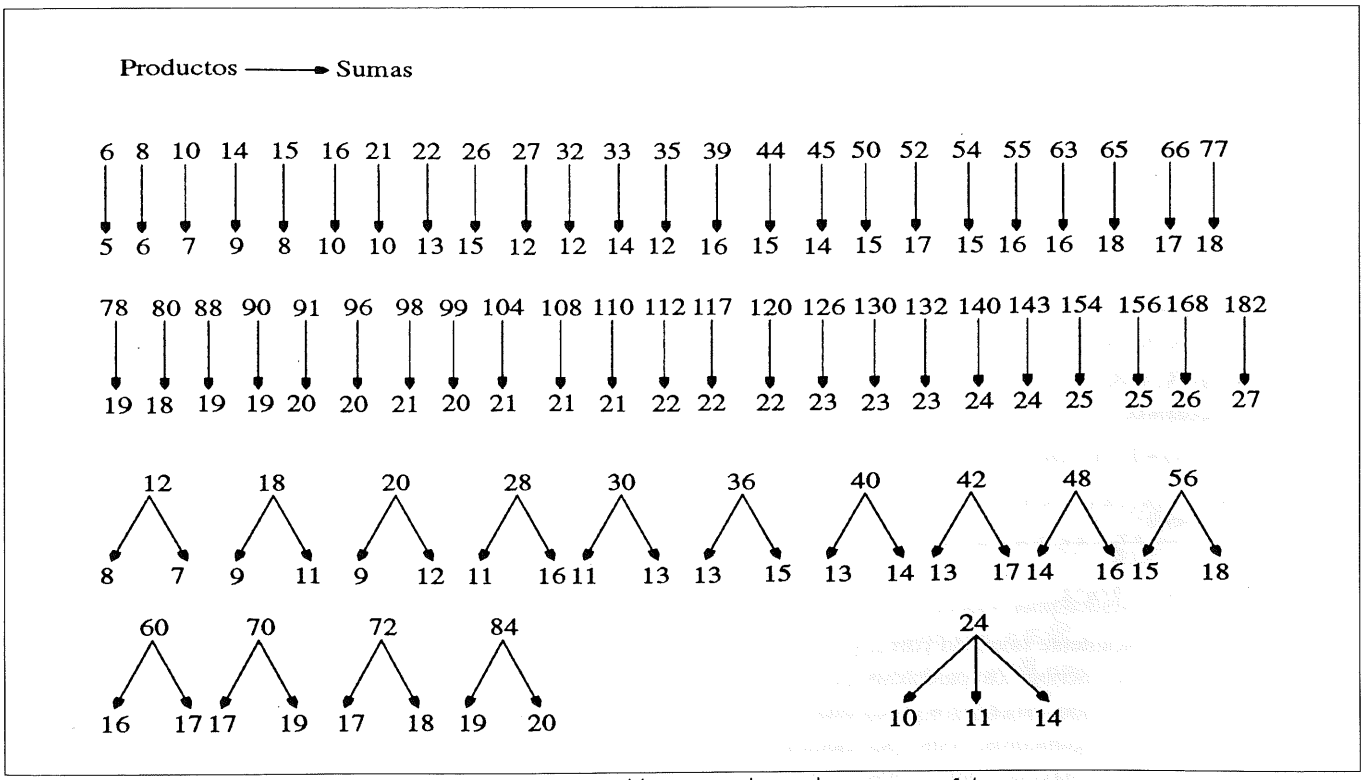

Figura 3. Sumas compatibles con cada producto, para t=14

1. Breve introducción a Haskell

Es usual en la Matemática encontrarnos con notaciones tales como

$$\pi : \mathbb{R}$$
$$\pi = 3.1415925\ldots$$

de forma que π es vista como una función constante. También es habitual encontrarnos con declaraciones de ecuaciones definidas *por partes* tal como

$$f : \mathbb{R} \to \mathbb{R}$$
$$f(x) = \begin{cases} x\ sen\frac{1}{x} & , si\ x \neq 0 \\ 0 & , si\ x = 0 \end{cases}$$

Tales notaciones pueden ser vistas como declaraciones: la primera declara el *tipo* de la función, mientras que la segunda declara el método de cómputo necesario para calcular el valor de la función en cada punto (se trata de una *declaración de ecuación*). *Haskell* permitirá describir tales declaraciones con una notación próxima a la anterior, tanto sintácticamente como conceptualmente. En *Haskell* tenemos la siguiente traducción de las declaraciones anteriores:

```
pi      :: Float
pi      = 3.1415927
f       :: Float -> Float
f x     = x*sin(1.0/x), if x/=0.0
        = 0.0,          if x==0.0
```

Una misma función puede aparecer a la izquierda de varias ecuaciones:

```
not             :: Bool -> Bool
not True        = False
not False       = True
```

y la elección de la ecuación a aplicar se determina por *comparación de patrones* (*pattern matching*). Algo parecido ocurre con la definición típica de la Matemática

$$g : \mathbb{R} \to \mathbb{R}$$
$$g(0) = 0$$
$$g(x) = 1 / x^2, \textit{ en otro caso}$$

cuya traducción sería

```
g 0.0   = 0.0
g x     = 1.0/x^2
```

Haskell utiliza sistemáticamente notación parcializada, en la cual no es necesario escribir los paréntesis para los argumentos en una aplicación; así (`f(x)` ≡ `f x`). Una función puede tener varios argumentos, como por ejemplo, la siguiente, que calcula el máximo común divisor de dos números:

```
mcd :: Int -> Int -> Int
mcd x y = x,                    if y == 0
        = mcd y (x `mod` y), otherwise
```

cuya ecuación utiliza el nombre de la propia función (recursividad). En este caso hay cierta diferencia entre la notación empleada en *Haskell* para describir una función con dos argumentos y la utilizada en la Matemática.

$$mcd : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$$
$$mcd(x, y) = \begin{cases} x & , si\ y = 0 \\ mcd(y, x\ mod\ y) & , en\ otro\ caso \end{cases}$$

Un tipo de datos extraordinariamente útil en programación funcional es la lista (o secuencia), cuya declaración *Haskell* es

```
data [a] = [] | a : [a]
```

Los objetos `[]` y `(:)` son constructores de datos; el primero representará la lista vacía; la declaración anterior se lee: una lista es, o bien la lista vacía `[]`, o bien `a:[a]`, donde `(:)` se interpreta como un operador-constructor infijo (el primer operando es un objeto de tipo `a` –la cabeza de la lista– y el segundo una lista del mismo tipo –la cola). Las listas modelan la descripción de los conjuntos y los *multiconjuntos* de la Matemática. Las siguientes funciones comprueban si una lista es vacía, tiene un solo elemento, o varios, y las restantes `and` y `or` calculan la conjunción y disyunción de una lista de valores lógicos:

```
no_vacia (_:_)  = True
no_vacia _      = False

uno [_]         = True
uno _           = False

varios (_:_:_)  = True
varios _        = False

and []          = True
and (b:bs)      = b && and bs

or []           = False
or (b:bs)       = b || or bs
```

Haskell utiliza sistemáticamente notación parcializada, en la cual no es necesario escribir los paréntesis para los argumentos en una aplicación...

donde los operadores `(&&)` y `(||)` representan la conjunción y la disyunción, respectivamente. Se incluyen distintas formas de expresar una lista:

```
1:2:3: [] ≠ [11,2,3]
[1..]     ≠ [1,2,3,...]
[1,4..20] ≠ [1,4,7,10,13,16,19]¶
```

Una función muy interesante es

```
map :: (a->b) -> [a] -> [b]
map f []= []
map f (x:xs) = f x : map f xs
```

que no es más que la representación *Haskell* de una función entre conjuntos; por ejemplo, si

$$f: A \to B \doteq \texttt{f::a->b}$$

(la anterior notación se lee: "f es representada en el lenguaje con la función `f`"), entonces, si $(\text{si } X \doteq \texttt{xs})$, el conjunto f(X) = { $f(x)$ | $x \in X$ } viene representado por `map f xs`. También, en matemáticas usamos sistemáticamente la descripción de *conjuntos por comprensión* en la forma $\{x \mid c\}$, donde c es una expresión que puede denotar un predicado, como $x \leq 100$, un generador como $x \in \mathbb{Z}$, o mezcla de ambos

A = { $g(x)$ | $x \in$ N, $x \leq 100$, x es par }

En forma similar, *Haskell* utiliza una notación para describir tales conjuntos en forma de *lista por comprensión*, a través de la función map; por ejemplo, el conjunto $\{g(x) \mid x \in \mathbb{N}\}$ se representa con `[g x | x<-[0..]]`, que equivale a `map g [0..]`. El anterior conjunto A viene descrito por:

```
a =[g x| x<-[0..], x<=100, par x]
   where par y = y `mod` 2 == 0
```

La función `map` verifica una serie de propiedades interesantes, como por ejemplo `map(f.g)= map f.map g`, donde `(.)` representa la composición de funciones: (f.g)x = f(g x), o también, para listas de booleanos: `or.map not = not.and`.

Si los conjuntos se modelan con listas sin repetición, veamos cómo modelar el cálculo de predicados con cuantificadores dentro del lenguaje; consideremos las siguientes expresiones *Haskell* para modelar el cuantificador existencial $(\text{si } X \doteq \texttt{xs}, p \doteq \texttt{p})$

$$\begin{aligned}
\exists x.\, x \in X.\, p(x) &\doteq \texttt{no_vacia[x|x<-xs,px]} \\
\exists' x.\, x \in X.\, p(x) &\doteq \texttt{uno} \quad\quad \texttt{[x|x<-xs,px]} \\
\exists'' x.\, x \in X.\, p(x) &\doteq \texttt{varios} \quad \texttt{[x|x<-xs,px]}
\end{aligned}$$

También podemos escribir

$$\exists x.\, x \in X.\, p(x) \doteq \texttt{or[px|x<-xs]}$$

(se demuestra fácilmente) y, podemos traducir directamente el cuantificador $\forall$:

$$\begin{aligned}
&\forall x.\, x \in X.\, p(x) \\
=\ & \quad \text{!cálculo de predicados} \\
&\neg(\exists x.\, x \in X.\, \neg p(x)) \\
\doteq\ & \\
&\texttt{(not.or)[not(p x)| x<-xs]} \\
=\ & \quad \text{!def. de listas por comprensión} \\
&\texttt{(not.or)[map (not.p) xs)} \\
=\ & \quad \text{!lema0, map(f.g)= map f.map g} \\
&\texttt{(not.not.and) (map p xs)} \\
=\ & \quad \text{!not.not = id --identidad} \\
&\text{and } \texttt{[p x| x<-xs]}
\end{aligned}$$

Es decir, hemos demostrado:

$$\forall x.\, x \in X.\, p(x) \doteq \texttt{and [p x| x<-xs]}$$

La estrategia hacia atrás

Resolveremos en primer lugar el problema *P-S* original y para ello introducimos algunas notaciones; supongamos que los números x e y elegidos por el árbitro, verifican $2 \leq x < y \leq t$; por consiguiente, un producto p de tales números deberá verificar $6 \leq p \leq t(t\text{-}1)$ y, una suma s, $5 \leq s \leq 2t\text{-}1$; para tales sumas y productos definimos los conjuntos de compatibles:

$$\begin{aligned}
\oplus p &= \{x + y \mid 2 \leq x < y \leq t, xy = p\} \\
\otimes s &= \{xy \mid 2 \leq x < y \leq t, x + y = s\}
\end{aligned}$$

Si $s \in \oplus p$ escribiremos simplemente $s \oplus p$ y recíprocamente, $p \otimes s$ significará $p \in \otimes s$; por consiguiente $\otimes$ y $\oplus$ son dos relaciones sobre $\mathbb{N}$, una inversa de la otra.

Para resolver el problema, consideramos cuatro funciones de conjuntos,

$$V_1, V_2, V_3, V_4 : \mathbb{N} \to 2^{\mathbb{N}}$$

...Haskell utiliza una notación para describir [...] conjuntos en forma de lista por comprensión, *a través de la función map*

Cada valor posible p del producto de la primera persona (*PROD*) determina un conjunto de valores s, para los que la afirmación *no puedo adivinar tu suma* es cierta

$$\begin{aligned} V_1(p) &= S, \text{ si } card(S) \geq 2, \text{ donde } S = \oplus p \\ &= \varnothing, \text{ en otro caso} \end{aligned}$$

donde *card* denota el cardinal. Obsérvese que $V_1(p)$ es vacío para cada número primo p, ya que éste se descompone de forma única en un producto xy $(1 = x < y = p)$ pero tal descomposición no es válida (ya que $x \neq 2$); de igual forma, p no puede ser producto de primos. La segunda observación es que si (p,s) es solución, debe tenerse $s \in V_1(p)$, pero no recíprocamente.

Veamos ahora como construir un conjunto

$$V_2(s) = \{p \mid \ldots.\}$$

cuyos elementos sean los posibles productos compatibles con s que certifican la segunda afirmación: *ya lo sabía; además tampoco puedo adivinar tu producto*, que la descomponemos como conjunción de dos:

(a) *ya lo sabía,*
que es independiente de la primera afirmación (puesto que con su suma s, SUM ya sabía que podía afirmarlo), y

(b) *tampoco puedo adivinar tu producto,*
que depende de la primera afirmación

A la condición *(b)* le asociamos el predicado

$$card\{\, p \mid p \otimes s,\ s \in V_1(p)\} \geq 2$$

ya que la suma de *SUM* debe ser alguna de las que *PROD* pensó (en $V_1(p)$ aparecen solamente las múltiples sumas que puede tener el segundo orador); en definitiva, tal predicado viene a decir: como *PROD* dijo la verdad, *SUM* tiene una suma dentro del conjunto $V_1(p)$ ya que la solución debe estar en el conjunto

$$\bigcup_{p=6}^{t(t-1)} \{p\} \times V_1(p)$$

Por el contrario, *a* no depende de la primera afirmación y viene a decir: *sé que no puedes adivinar mi suma.* Para cierta suma s consideremos la afirmación contraria *puedes adivinar mi suma*; ello significa que debe existir un producto p compatible $(p \otimes s)$ con una única suma compatible (que sería precisamente s); y recíprocamente, si existiera un p con una única suma compatible, *PROD* podría adivinar, con tal producto, su suma; es decir, el predicado correspondiente a *puedes adivinar mi suma* es

$$\exists p.\ p \otimes s.\ card(\oplus p) = 1$$

Ahora es fácil escribir el predicado correspondiente a la afirmación *no puedes adivinar mi suma*, con un simple cálculo

$$\begin{aligned} & \neg(\exists p.\ p \otimes s.\ card(\oplus p) = 1) \\ = \quad & !\, c.\ p. \\ & \forall p.\ p \otimes s.\ card(\oplus p) \neq 1 \\ = \quad & !\, ya\ que\ p \otimes s \Rightarrow \oplus p \neq 0 \\ & \forall p.\ p \otimes s.\ card(\oplus p) \geq 2 \end{aligned}$$

En definitiva, a la segunda afirmación del diálogo le corresponde el conjunto

$$\begin{aligned} V_2(s) &= \otimes s, \text{ si } (\forall p.\ p \otimes s.\ card(\oplus p) \geq 2) \\ &\quad \wedge \\ &\quad card\{p \mid p \otimes s,\ s \in V_1(p)\} \geq 2 \\ &= \varnothing, \text{ en otro caso} \end{aligned}$$

Según la conjetura de Christian Goldbach (anunciada en 1742), si s es una suma par se puede descomponer en suma de dos primos x e y; pero el producto xy solo se descompone en esta forma, por lo que si tuviera tal producto el primer orador, sabría su suma; por otro lado, obsérvese que tales sumas quedan excluidas al no verificar el predicado anterior (si p es producto de dos primos, $card(\oplus p)=1$).

A la tercera afirmación (*entonces ya se tu suma*) le asociamos otro conjunto

$$\begin{aligned} V_3(p) &= S, \text{ si } card(S) = 1 \\ &= \varnothing, \text{ en otro caso} \\ &\text{donde } S = \{s \mid s \oplus p,\ p \in V_2(s)\} \end{aligned}$$

y finalmente, a la cuarta afirmación (*y yo tu producto*) le corresponde el conjunto

$$\begin{aligned} V_4(s) &= P, \text{ si } card(P) = 1 \\ &= \varnothing, \text{ en otro caso} \\ &\text{donde } P = \{p \mid p \otimes s,\ s \in V_3(p)\} \end{aligned}$$

En consecuencia, si Σ_t es el conjunto de sumas posibles

$$\Sigma_t = \{x + y \mid 2 \leq x < y \leq t\}$$

el conjunto de pares que verifican el diálogo es

$$S_t = \{(p,s) \mid s \in \Sigma_t,\ p \in V_4(s)\}$$

Un programa Haskell

La descripción por comprensión del conjunto S_t es una buena situación de partida para escribir un programa que calcule las soluciones; el único inconveniente es traducir a listas los conjuntos $V_1, \ldots, V_4$. En primer lugar veamos la descripción de los conjuntos $\oplus p$ y $\otimes s$

Lema 1. Sea $t \in \mathbb{N}$, $t \geq 3$; entonces:

$$(a)\quad \oplus p = \{x+y \mid \max(2,\lfloor p/t \rfloor) \leq x \leq \lfloor \sqrt{p} \rfloor,\ (y \equiv \lfloor p/x \rfloor), xy = p, x < y, y \leq t\}$$

$$(b)\quad \otimes s = \{xy \mid \max(2, s-t) \leq x \leq \lfloor (s-1)/2 \rfloor,\ (y \equiv s-x), x \neq y\}$$

Demostración. En primer lugar observamos

$$\begin{array}{l} 2 \leq x < y \leq t, xy = p \\ \Rightarrow \\ 2 \leq x, \quad x^2 < xy = p \leq tx \\ \Rightarrow \\ 2 \leq x \leq \lfloor \sqrt{p} \rfloor, \quad \lfloor p/t \rfloor \leq x \\ \Rightarrow \\ \max(2,\lfloor p/t \rfloor) \leq x \leq \lfloor \sqrt{p} \rfloor \end{array}$$

y esto probaría $\oplus p \subset \{\ldots\}$; la recíproca es evidente. Para probar *(b)* tenemos

$$\begin{array}{l} 2 \leq x < y \leq t, \quad x+y = s \\ = \\ 2 \leq x < s-x = y \leq t \\ = \\ 4 \leq 2x < s = x+y, \quad y \leq t, \quad s \leq t+x \\ = \quad !\, x, y \in \mathbb{N} \\ 4 \leq 2x \leq s-1, x+y = s, y \leq t, \quad s-t \leq x \\ = \\ \max(2, s-t) \leq x \leq \lfloor (s-1)/2 \rfloor, y = s-x \leq t \end{array}$$

de donde $\otimes s \subset \{\ldots\}$; la recíproca de nuevo es fácil. <c.q.d>

Del lema concluimos que, en nuestro lenguaje, los conjuntos de compatibles vienen descritos por las funciones (`t` es una constante del programa):

```
pro_de s = [x*y|x<-[max 2 (s-t)..(s-1)`div`2],
              y=s-x, x/=y]
sum_de p = [x+y|x<-[max 2 (p`div`t)..sqrt p],
              y = p`div`x, x/=y,y<=t,x*y==p]
```

ya que en las listas anteriores no existen elementos repetidos (*Haskell* no permite variables locales en listas por comprensión, como por ejemplo `y=s-x`, pero sí Gofer (Jones, 1992) , un subconjunto extendido de éste; mantenemos tal notación para simplificar el código). La descripción de las funciones V_1, V_2, V_3, V_4 y S_t es fácil:

```
v1 p = ss, varios ss
     = [], otherwise
   where ss = sum_de p

v2 s = ps, varios [ p | p<-ps, s `elem` v1 p]
          && and [varios(sum_de p) | p<-ps]
     = [], otherwise
   where ps = pro_de s

v3 p = ss, uno ss
     = [], otherwise
   where ss = [ s | s<-sum_de p, p `elem` v2 s]

v4 s = pp, uno pp
     = [], otherwise
   where pp = [ p | p<-pro_de s, s `elem` v3 p]

sol = [(p,s)|s<-[5..2*t-1], p<-v4 s]
```

(`` `elem` `` comprueba la pertenencia de un elemento a una lista). Si utilizamos la conjetura de Goldbach podemos reemplazar la lista `[5..2*t-1]` por la lista de números impares `[5,7..2*t-1]` e incluso podemos eliminar la relación `x/=y` en la función `pro_de`.

La descripción de la solución al problema utiliza una técnica muy conocida: la técnica de backtracking *o vuelta-atrás...*

La descripción de la solución al problema utiliza una técnica muy conocida: la técnica de *backtracking* o vuelta-atrás, en la cual, la búsqueda de la solución consiste en suponer un valor s para una solución (p,s) (o sea, una suma dentro de las posibles), y buscar los elementos de la lista `v4 s`; si ésta es vacía (s no está de acuerdo con la cuarta afirmación del diálogo) se produce la vuelta-atrás, eligiendo otra suma posible. Igualmente, en la descripción de `v4` se considera la lista

```
[p | p<-pro\_de s, s `elem` v3 p]
```

que se interpreta: «tomar un producto `p` compatible con `s`; si `s` es un elemento de `v3 p`, se tomará, en otro caso se elige otro»; pero tal lista, evaluada en forma perezosa, no debe calcularse completamente: en cuanto el evaluador descubra que hay más de uno dejará de calcular y devolverá la lista vacía. El lector que conozca con cierta destreza el lenguaje *PROLOG*, habrá observado un enorme parecido con la técnica utilizada en éste lenguaje; en el §5 volveremos sobre ello.

3. Revisión de la estrategia hacia atrás

Como puede verse en la función que describe la solución,

```
sol = [(p,s)|s<-[5..2*t-1], p<-v4 s]
```

para cada suma posible, se calcula el correspondiente conjunto `v4 s`, lo que conlleva en general mucho cálculo; se puede agilizar el proceso si tenemos en cuenta las observaciones del siguiente

Lema 2. Con las notaciones de §2,

$$(a)\ V_4(s) \subset V_2(s)$$
$$(b)\ V_3(p) \subset V_1(p)$$

El lema muestra la consistencia de las afirmaciones de los oradores (por ejemplo, *(a)* indica que la afirmación cuarta no contradice a la segunda). También, la propiedad *(a)* muestra que basta buscar las parejas que verifican $p \in V_2(s)$, por lo que la solución puede obtenerse también en la forma

```
sol = [(p,s)|s<-[5..2*t-1],
       no_vacio(v2 s), p<-v4 s]
```

que reduce el cómputo en una proporción del orden de $t/2$. La propiedad *(b)* indica que el conjunto $V_3(p)$ puede obtenerse en la forma

```
v3 p = ss, uno ss
     = [], otherwise
 where ss = [ s | s<-v1 p,
  p `elem` v2 s]
```

(donde hemos sustituido el generador `s<-sum_de p` por `s<-v1 p`)pero, aunque el cómputo de `v1 p` no conlleva demasiado cálculo, en general, es más eficiente la selección desde la lista `sum_de p`, ya que el número de sumas compatibles para un producto dado es pequeño, al contrario que el número de productos compatibles para una suma (como vemos en la tabla 1).

Demostración del lema:

$p \in V_4(s)$
$\Rightarrow$! en este caso $V_4(s) \neq \emptyset$ tiene un elemento
$p \otimes s, s \in V_3(p)$
$\Rightarrow$! también $V_3(p) \neq \emptyset$, con un elemento
$p \in V_3(p)$

La demostración de *(b)* es parecida

$s \in V_3(p)$
$\Rightarrow$! $V_3(p) \neq \emptyset$, con un solo elemento
$s \oplus p, p \in V_2(s)$
$\Rightarrow$! $V_2(s) \neq \emptyset$, y entonces debe darse
$\forall p'.\ p' \otimes s.\ \mathrm{card}(\oplus p') \geq 2$
$s \oplus p, card(\oplus p) \geq 2$
$\Rightarrow$! $V_1(p) \neq \emptyset$ y $s \in \oplus p$
$s \in V_1(p)$

<c.q.d.>

El lema muestra la consistencia de las afirmaciones de los oradores

Puede mejorarse aún más la eficiencia del programa si observamos que en el cómputo de las funciones que calculan los conjuntos $V_1, \ldots, V_4$ siempre es necesario evaluar un predicado previo. Para V_1 el predicado es

$$a_1(p) = card(\oplus p) \geq 2$$

que se corresponde con la afirmación del primer orador, de forma que el conjunto $V_1(p)$ queda determinado por éste, y el segundo orador evaluaría el predicado

$$a_2(s) = (\forall p.\ p \otimes s.\ card(\oplus p) \geq 2) \wedge card\{p \mid p \otimes s, a_1(p)\} \geq 2$$

Para la afirmación tercera podemos intentar el predicado

$$a_3(p) = card\{s \mid s \oplus p, a_2(s)\} = 1$$

de forma que el conjunto $V_4(s)$ es reemplazado por

$$W_4(s) = P, \text{ si } card(P) = 1$$
$$= \emptyset, \text{ en otro caso}$$
$$\text{donde } P = \{p \mid p \otimes s, a_3(p)\}$$

Desgraciadamente, el conjunto

$$S'_t = \{(p,s) \mid s \in \Sigma_t, p \in W_4(s)\}$$

contiene pares que no son soluciones, y se observa que las no válidas aparecen porque no verifican el predicado a_2; en ese caso, introducimos tal predicado en el conjunto:

$$S'_t = \{(p,s) \mid s \in \Sigma_t, a_2(s), p \in W_4(s)\}$$

Probaremos que los pares así obtenidos son las soluciones; ello será consecuencia del siguiente

Lema 3.

$(a)\ s \in V_1(p) \Leftrightarrow s \oplus p \wedge a_1(p)$
$(b)\ p \in V_2(s) \Leftrightarrow p \otimes s \wedge a_2(s)$
$(c)\ s \in V_3(p) \Leftrightarrow s \oplus p \wedge a_3(p)$
$(d)\ s \in V_3(p) \Leftrightarrow s \oplus p \wedge a_3(p) \wedge a_2(s)$
$(e)\ p \in W_4(s) \wedge a_2(s) \Leftrightarrow p \in V_1(s)$

Antes de la demostración hagamos algunas observaciones; mientras las propiedades *(a)* y *(b)* son equivalencias, las restantes son implicaciones; por ejemplo, para el valor $t = 14$, la impli-

cación recíproca de *(c)* es falsa, ya que $V_3(18) = \{11\}$, $\oplus 18 = \{9,11\}$, $a_3(18)$ es cierto (ver figuras 2 y 3) pero sin embargo $9 \notin V_3(18)$ (esto ocurre porque $a_2(9) = Falso$). Por otro lado, la propiedad *(d)* asegura que toda solución obtenida con

```
sol = [(p,s)|s<-[5..2*t-1], a2 s, p<-w4 s]
```

verifica el diálogo, y recíprocamente. S_t es vacío para valores de t dentro del intervalo $3 \le t \le 61$. Para valores en el intervalo $62 \le t \le 1681$, dicho conjunto tiene un solo elemento, el par (52,17) que corresponde a $x = 4$ e $y = 13$: el números de *palos* y *cartas* (de cada palo) de la *baraja francesa*. Para valores mayores de t aparecen nuevos pares soluciones; conjeturamos que los pares soluciones de la forma $(2^k, y)$, donde y es un números primo, se conservan al aumentar t. Obsérvese la simpleza del programa resultante con tales predicados

```
a1 p = varios(sum_de p)
a2 s = varios[p|p<-ps, a1 p] &&
         and[varios(sum_de p)|p<-ps]
     where ps = pro_de s
a3 p = uno[s|s<-sum_de p, a2 s]
a4 s = pp, uno pp
     = [], otherwise
     where pp = [p|p<-pro_de s, a3 p]
```

Demostración del lema. *(a)* es trivial. Para probar *(b)*

$$
\begin{aligned}
& p \in V_2(s) \\
= \quad & \text{! definición de } V_2 \\
& p \otimes s \wedge (\forall p' . p' \otimes s . card(\oplus p') \ge 2) \wedge \\
& card\{p' \mid p' \otimes s, s \in V_1(p')\} \ge 2 \\
= \quad & \text{! cálculo, por } (a) s \in V_1(p') \equiv a_1(p') \wedge p' \otimes s \\
& p \otimes s \wedge (\forall p' . p' \otimes s . card(\oplus p') \ge 2) \wedge \\
& card\{p' \mid p' \otimes s, a_1(p')\} \ge 2 \\
= \quad & \\
& p \otimes s \wedge a_2(s)
\end{aligned}
$$

Es evidente que *(c')* $\Rightarrow$ *(c)*; veamos entonces *(c')*:

$$
\begin{aligned}
& s \in \oplus p, a_2(s'), a_3(p) \\
\Rightarrow \quad & ! \; a_3(p) \equiv card(\{s \in \oplus p, a_2(s)\}) = 1 \\
& s = s' \\
\Rightarrow \quad & \\
& \{s' \in \oplus p, p \in V_2(s')\} = \{s\}, \text{ y } s \in V_3(p)
\end{aligned}
$$

Finalmente veamos *(d)*,

$$
\begin{aligned}
& p \in W_4(s) \wedge a_2(s) \\
= \quad & \text{! existe un único elemento en } W_4(s) \\
& p \otimes s \wedge card\{p' \mid p' \otimes s, a_3(p')\} = 1 \wedge a_3(p) \wedge a_2(s) \\
= \quad & \text{! } (c') \\
& s \in V_3(p) \wedge card\{p' \mid p' \otimes s, a_3(p')\} = 1 \\
= \quad & \text{! por } (c') \\
& s \in V_3(p) \wedge card\{p' \mid p' \otimes s, s \in V_3(p')\} = 1 \\
= \quad & \\
& p \in V_4(s)
\end{aligned}
$$

<c.q.d.>

4. El problema de Freudenthal

El problema que hemos resuelto es el expuesto en Coehlo (1988), aunque, como demostraremos a continuación, es equivalente al propuesto por Hans Freudenthal en 1969 (ver referencias en Gardner, 1980), cuyo enunciado *original* es:

SUM.- No se cómo vas a adivinar mi suma.
PROD.- ¡Ajá!, entonces ya sé tu suma
SUM.- Y yo tu producto.

Es fácil comprobar que al diálogo anterior le corresponden los mismos predicados salvo

$$
\begin{aligned}
a_2(s) \quad = \quad & \forall p . p \otimes s . car(\oplus p) \ge 2 \\
& card\{p \mid p \otimes s, a_1(p)\} \ge 2
\end{aligned}
$$

que es reemplazado por

$$b_2(s) = \forall p . p \otimes s . card(\oplus p) \ge 2$$

Se observa también que podemos eliminar el predicado $a(p)$ del primer término de la conjunción de a_2 por lo que éste es equivalente a

$$a_2(s) = b_2(s) \wedge card(\otimes s) \ge 2$$

La equivalencia de los dos diálogos será consecuencia de la igualdad de los predicados a_2 y b_2, que a su vez será consecuencia del siguiente

Lema 4. Sea una suma $s \in \Sigma t$ $(t > 5)$ con un único producto; entonces $s \in \{5, 6, 2t\text{-}2, 2t\text{-}1\}$. Y recíprocamente, tales sumas tienen un único producto.

A = Número de productos con N sumas compatibles
B = Número de sumas con N productos compatibles

N	1	2	3	4	5	6	
A	47	14	1	0	0	0	$t = 14$
B	4	4	4	4	4	3	

N	1	2	3	4	5	6	7	...	18	19	
A	306	111	38	17	5	1	0	...	0	0	$t = 40$
B	4	4	4	4	4	4	4	...	4	3	

N	1	2	3	4	5	6	7	8	...	28	29	
A	685	212	96	36	20	7	4	0	...	0	0	$t = 60$
B	4	4	4	4	4	4	4	4	...	4	3	

N	1	2	3	4	5	6	7	8	9	...	38	39	
A	1216	347	155	73	42	23	6	3	0	...	0	0	$t = 80$
B	4	4	4	4	4	4	4	4	4	...	4	3	

Tabla 1. Número de sumas y productos compatibles

Demostración. Sea $6 < s < 2t\text{-}2$; si s par $(= 2k)$:

$$6 < 2k < 2t-2$$
$$=$$
$$4 \le k \le t-2$$
$$\Rightarrow$$
$$p = (k-1)(k+1) \otimes s,\ p' = (k-2)(k+2) \otimes s,\ p \neq p'$$

y para s impar $(= 2k+1)$,

$$6 < 2k+1 < 2t-2$$
$$=$$
$$3 \le k \le t-2$$
$$\Rightarrow$$
$$p = k(k+1) \otimes s,\ p' = (k-1)(k+2) \otimes s,\ p \neq p'$$

El recíproco se deduce de $\otimes 5 = \{6\}$, $\otimes 6 = \{8\}$, $\otimes(2t\text{-}1) = \{(t\text{-}1)t\}$ y $\otimes(2t\text{-}2) = \{(t\text{-}2)t\}$ <c.q.d.>

Corolario. Los siguientes predicados son equivalentes

$$b_2(s) = \forall p.\ p \otimes s.\ card(\oplus p) \ge 2$$
$$a_2(s) = b_2(s) \wedge card(\otimes s) \ge 2$$

Demostración. Probaremos que los valores lógicos $b_2(5)$, $b_2(6)$, $b_2(2t\text{-}2)$, $b_2(2t\text{-}1)$, son todos falsos; los dos primeros resultan falsos ya que $\oplus 6 = \{5\}$ y $\oplus 8 = \{6\}$; por otro lado, el producto $t(t\text{-}1)$ sólo admite tal descomposición, ya que

$$p = t(t-1) = xy,\ 2 \le x < y \le t$$
$$\Rightarrow \text{ ! lema 1}$$
$$\max(2, \lfloor p/t \rfloor) \le x \le \lfloor \sqrt{t(t-1)} \rfloor,$$
$$\Rightarrow\ !(t-1)^2 \le t(t-1) < t^2$$
$$t-1 \le x \le t-1$$

Igualmente $\otimes t(t\text{-}2) = \{2t\text{-}2\}$, ya que

$$p = t(t-2) = xy,\ 2 \le x < y \le t$$
$$\Rightarrow \text{ ! lema 1}$$
$$\max(2, \lfloor p/t \rfloor) \le x \le \lfloor \sqrt{t(t-2)} \rfloor,$$
$$\Rightarrow\ !(t-2)^2 \le t(t-2) < (t-1)^2$$
$$t-2 \le x \le t-2$$

Es decir, tenemos

$$b_2(s) \Rightarrow 6 < s < 2t-2 \Rightarrow^{\text{lema 4}} card(\otimes s) > 1$$

<c.q.d.>

El lema muestra a su vez que los apartados correspondientes de los lemas 2 y 3 siguen siendo válidos, por lo que podemos eliminar las funciones `v1` y `a1`, y reemplazar `v2` y `a2` por:

```
v2 s = ps, and [varios(sum_de p) | p<-ps]
     = [], otherwise
       where ps = pro_de s
a2 s = and [varios(sum_de p) | p<-pro_de s]
```

5. Un programa PROLOG

La estrategia hacia atrás descrita, así como sus refinamientos, puede ser utilizada para diseñar programas en el lenguaje PROLOG teniendo en cuenta las siguientes transformaciones (Ruiz 1993, Wadler 1985): (a) reemplazamos cada función de conjunto V por un predicado v cuyo conjunto de éxitos (sustituciones) se identifica precisamente con V, y (b) los predicados se reemplazan directamente. En la figura 4 aparece el programa PROLOG ya traducido (la "t" representa la cota máxima).

```
compatible(S,P) :- nonvar(S), !,
                   Xmax is (S-1) // 2, Smt is S-"t",
                   max(2,Smt,Xmin),genera(X,Xmin,Xmax),
                   Y is S-X, Y=\=X, P is X*(S-X).

compatible(S,P) :- nonvar(P), !,
                   Xmax is integer(sqrt(P)), Pdt is P // "t",
                   max(2,Pdt,Xmin),genera(X,Xmin,Xmax),
                   Y is P // X, Y=\=X,Y=<Max,P is X*Y,S is X+Y.

a2(S) :-setof(P,compatible(S,P),L),varios_prods(L).

varios_prods([],_,_) :-!.
varios_prods([P|R])  :- setof(S,compatible(S,P), [_,_|_]),varios_prods(R).

a3(P) :-setof( S,(compatible(S,P),a2(S)),[_]).

a4(S,Psol):-genera(S,5,"2t-1"),a2(S), setof(P,(compatible(S,P),a3(P)),[Psol]).

genera(I,I,_) :- !.
genera(I,L,M) :- L<M, L1 is L+1, genera(I,L1,M).

max(X,Y,X) :- X >= Y,!.
max(_,Y,Y).
```

Figura 4. La estrategia hacia atrás en PROLOG

D. Warren describe otro programa en (Coehlo, 1988) parecido al nuestro, pero cambiando los predicados a3 y a4 en la forma

```
a3(P,S_sol) :- setof( S,
       (genera(S,5,"2t-1"),
       a2(S),compatible(S,P)),[S_sol]).
a4(S,Psol) :- setof(P,a3(P,S),[Psol]).
```

El lector podrá observar la enorme dificultad de su comprensión (¿y su corrección?), aunque resulta más eficiente ya que con el predicado `a3`, a través de `setof`, se recupera *¡en memoria!* todo el espacio de búsqueda correspondiente a la segunda afirmación, por lo que el programa resultante es *técnicamente imperativo*. En el siguiente apartado veremos una solución también técnicamente imperativa, con una eficiencia similar.

La estrategia hacia adelante

De acuerdo a lo expuesto en §0 para el problema de las bolas, podemos programar directamente la *estrategia hacia adelante*; vamos a resolver, por simplicidad, la versión de Freundenthal. La idea es representar el espacio de búsqueda de alguna forma, y filtrar éste espacio sucesivamente y según las afirmaciones. Podemos intentar un espacio de búsqueda de la forma

```
type Suma = Int; type Prod = Int
type EspacioBusqueda = [(Prod,Suma)]
```

Es decir, una lista de pares de posibles soluciones; tal organización no es muy eficiente si tenemos en cuenta que en cierto momento del proceso de criba debemos eliminar las sumas compatibles con cierto producto, o los productos compatibles con cierta suma. Por ello, es más interesante mantener una estructura que facilite la poda; por ejemplo podemos pensar en la estructura:

```
type EspacioBusqueda = [([Prod],Suma)]
```

donde cada suma posible aparece una sola vez, junto con la lista de los productos compatibles con ella, o también

```
type EspacioBusqueda = [(Prod,[Suma])]
```

donde `[Suma]` es la lista de productos compatibles con `Producto`. Se puede observar que, en general, hay más sumas con muchos productos compatibles que productos con muchas sumas compatibles, por lo que la primera representación del espacio de búsqueda es más adecuada, ya que implementando un filtrado especial, nos permitirá mejorar la eficiencia del algoritmo. La generación del espacio de búsqueda inicial será:

```
espacio_inicial :: EspacioBusqueda
espcacio_inicial = [ (pro_de s,s) |
                     s<-posibles_sumas]
```

Debemos ahora cribar según el orden de las afirmaciones:

```
type FiltroP = Prod->EspacioBusqueda->Bool
-- Filtra con afirmaciones
-- del primer jugador
criba_P_con :: FiltroP->EspacioBusqueda
               ->EspacioBusqueda
criba_P_con f es =
 [(ps',s') | (ps,s')<-es,
             ps'=[p|p<-ps, f p es],
             ps' /= []]

type FiltroS = Suma->[Prod]->Bool
- Filtra con afirmaciones del segundo
criba_S_con :: FiltroS->EspacioBusqueda
               ->EspacioBusqueda
criba_S_con _ []=              []
criba_S_con f ((ps@(p:ps'),s):es')
   = (ps,s):criba_S_con f es', if f s ps
   = criba_S_con f es',         otherwise
```

La función `criba_P_con` aplica el filtro para cada suma y cada producto compatible de la lista. Sin embargo, `criba_S_con` aplica el filtro para una suma, y si no se cumple, elimina del espacio de búsqueda dicha suma junto con todos sus productos compatibles en una sola vez. Los filtros para las distintas afirmaciones serán:

```
f3     :: FiltroP
f2,f4 :: FiltroS

f2 s es =
 and [varios(sum_de p es) | p<-pro_de s es]
f3 p es = uno (sum_de p es)
f4 s es = uno es
```

donde las funciones pro_de y sum_de ahora tienen un tercer argumento (el espacio de búsqueda):

```
sum_de p es = [s | (ps,s)<-es, p `elem` ps]
pro_de s es = [p | (ps,s')<-es, s'==s, p<-ps]
```

La solución al problema será:

```
sol = criba_S_con f4 . criba_P_con f3
      criba_S_con f2 espacio_inicial
```

6. Otros acertijos

Otro tipo de problemas más complejos son aquellos acertijos en los cuales el número de personas que intervienen es mayor (tres o cuatro personas), cada una ve algunos datos de los restantes (pero no el suyo). Ejemplo de ello es la siguiente variante del *problema de los sombreros* (un curioso problema atribuido a J.Sebelik, 1983):

Tres personas -SEBELIK, PACO y BLAS - se encuentran en una habitación, de forma que SEBELIK ve a PACO y a BLAS, y PACO ve sólo a BLAS (BLAS -como casi siempre - no ve nada). En una mesa de otra habitación hay cinco sombreros, dos negros y tres blancos. Se apaga la luz; un "árbitro" toma tres sombreros de la mesa y los coloca en las cabezas de

las personas; después de encender la luz, se escucha el siguiente diálogo:

SEBELIK.- Yo no se el color de mi sombrero.

PACO.- En ese caso, ya sé el color del mío.

¿De qué color es su sombrero?

Otra variante del problema es para cuatro personas:

SEBELIK, PACO, PEPE y BLAS, (SEBELIK ve a todos, PACO también ve a todos, PEPE ve solo a BLAS, y, de nuevo, BLAS no ve nada); los sombreros ahora son dos blancos, uno negro, uno rojo y uno verde, y el diálogo es

SEBELIK.- Yo no sé el color de mi sombrero.

PACO.- En ese caso, yo tampoco.

PEPE- ¡Ajá!, ya sé el color del mio.

El lector puede comprobar que las técnicas descritas anteriormente son fácilmente generalizables a tales problemas; algunos de los resultados demostrados anteriormente seguirán siendo válidos con ligeros retoques.

7. Conclusiones

En este artículo hemos visto como un lenguaje funcional moderno puede ser utilizado para describir la solución a problemas de forma próxima a la notación del matemático. Ciertas conjeturas derivadas de la ejecución del programa con ligeros cambios conducen a un estudio detallado de éstos; y viceversa, un estudio detallado de la formulación matemática del problema permite una descripción eficiente de la solución. Esta fuerte conexión ayuda a los alumnos a considerar las *Ciencias de la Computación* como una disciplina dentro de *la matemática.*

Blas C. Ruiz
Francisco Gutiérrez
José E. Gallardo
Departamento de Lenguajes y Ciencias de la Computación.
Universidad de Málaga

8. Referencias bibliográficas

BIRD, R. y R. WADLER (1988): *Introduction to Functional Programming*, Prentice-Hall.

COELHO, H. y J. C. COTTa (1988): *Prolog by example*, Springer-Verlag.

GALLARDO, J. E., P. GARCÍA, F., GUTIÉRREZ y B. RUIZ (1995): *Programación Funcional en Haskell*, Universidad de Málaga.

GARDNER, M. (1980): «Juegos Matemáticos», Investigación y Ciencia, Febrero, Abril y Julio.

HAMMOND, K. (Editor) (1995): *Report on the Programming Language Haskell, A Non-Strict Purely Functional Language, Version 1.3.*

HUDAK, P. y J. H. Fasel (1992): «A Gentle introduction to Haskell». *SIGPLAN*, 27, 5.

JONES, M. (1992): *Gofer 2.28. Release Notes*, Technical Report, Dep. Comp. Science, Yale Universiy.

RUIZ, B. (1993): «Programación funcional á là Prolog», en *Actas de PRODE'93*, Gerona.

WADLER, P. (1985): «How to replace failure by a list of successes», *2'nd Symposium on Functional Programming Languages and Computer Architecture*, Nancy.

REGLAMENTO DE LA OLIMPIADA MATEMÁTICA DE LA FEDERACIÓN ESPAÑOLA DE SOCIEDADES DE PROFESORES DE MATEMÁTICAS

(Aprobado en Junta de Gobierno celebrada el 13 de septiembre de 1995)

1. La Federación Española de Sociedades de Profesores de Matemáticas organizará anualmente una Olimpiada cuyo nombre oficial es: *Olimpiada Matemática de la Federación Española de Sociedades de Profesores de Matemáticas.*
   El logotipo será el que fije la Federación.

Sobre los objetivos

2. Se pretende conseguir los siguientes objetivos:
   a) Potenciar la participación masiva de estudiantes y profesores en las fases previas al encuentro nacional, de acuerdo con los objetivos propuestos en los Estatutos de la Federación.
   b) Fomentar entre los estudiantes el gusto por las Matemáticas, así como presentar una visión de las mismas complementaria a la utilizada en el aula.
   c) Favorecer las relaciones de amistad y conocimiento entre jóvenes de diversas comunidades autónomas.

Sobre la organización

3. La Junta de Gobierno de la Federación encomendará la organización de cada edición a alguna de las sociedades federadas.
4. La sociedad que asuma la responsabilidad de organizar la Olimpiada propondrá a la Junta de Gobierno el nombramiento de un coordinador de la misma.