Composiciones de un natural en k partes en *Coq*

Reporte de Proyecto Final
Verificación Formal
Luis Felipe Benítez Lluis
LL11

Resumen

Se define una k-composición de un número n como una k-tupla de naturales que sumen n. El problema de generar todas las composiciones es una tarea que puede aparecer en diversos problemas. Particularmente a la hora de enumerar todas las k-tuplas. Se dedica esta segunda parte del proyecto a plantear una biblioteca en *Coq* donde se trabajen estos conceptos además de ofrecer funciones hasNext y Next con la que se pueden generar todas las composiciones de un número de forma iterativa.

1. Introducción

La generación de estructuras combinatorias suele ser una tarea que surge de resolver problemas más grandes. La generación de k-composiciones es una de estas estructuras combinatorias que surge en problemas como desarrollar la expresión del producto de varios polinomios como:

$$\Big(\sum_{i=0}^{n} a_i x^i\Big)\Big(\sum_{i=0}^{n} c_i x^i\Big)\Big(\sum_{i=0}^{n} c_i x^i\Big) = \sum_{i=0}^{3m}\Big(\sum_{j+k+l=i} a_j b_k c_l\Big)x^i,$$

donde (j,k,l) representa una 3-composición de i. Generando todas las composiciones, podemos obtener efectivamente una representación de este producto de polinomios. Otra aparición de un problema de precisa obtener las k-composiciones de un número proviene como una tarea de dar una enumeración de tuplas de naturales en naturales.

La necesidad de dar una enumeración de tuplas de naturales proviene de querer generar todos los programas en el modelo de cómputo imp ,planteado en [1]. Esto con el fin de dar una aproximación a la distribución universal, medida propuesta por Solomonoff [2]. Esto se pretende realizar a través de la generación sistemática de programas cortos en el modelo planteado anteriormente, como parte del proyecto de doctorado del M. en C. Vladimir Lemus Yáñez [3].

La idea es codificar dichos programas en tuplas la estructura sintáctica de estos. Siendo así, una enumeración de tuplas corresponderá en una enumeración de programas. No obstante, se busca que estas enumeraciones comiencen por valores pequeños para paulatinamente alcanzar valores grandes. Esto se hace generalizando el algoritmo de enumeración de Cantor.

En esta enumeración se ordenan las parejas en una tabla infinita de manera que la pareja (i,j) se encuentre en el renglón j y la columna i y se enumeran las parejas por diagonales de manera que cada diagonal es siempre finita. Una versión de esta enumeración se ve como $0 \mapsto (0,0)$,$1 \mapsto (0,1)$, $2 \mapsto (1,0)$, $3 \mapsto (0,2)$, $4 \mapsto (1,1)$,$5 \mapsto (2,0)$,$6 \mapsto (0,3), \ldots$, donde las diagonales son $\{(0,0)\}, \{(0,1),(1,0)\}, \{(0,2),(1,1),(2,0)\}, \ldots$, como se puede apreciar en la tabla 1.

Se puede observar que todas las n-adas en una misma diagonal siempre suman lo mismo, esto es las n-ésima diagonal representa el conjunto de todas las composiciones de n en dos partes. Buscando generalizar este algoritmo a dimensión k de manera que se preserve la propiedad de que enumere tuplas con valores pequeños primero, se generaliza la noción de la n-ésima diagonal como el conjunto de k-composiciones de n.

	0	1	2	3	4	5	6	⋯
0	(0,0)	(1,0)	(2,0)	(3,0)	(4,0)	(5,0)	(6,0)	⋯
1	(0,1)	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)		
2	(0,2)	(1,2)	(2,2)	(3,2)	(4,2)			
3	(0,3)	(1,3)	(2,3)	(3,3)				
4	(0,4)	(1,4)	(2,4)					
5	(0,5)	(1,5)						
6	(0,6)							
⋮	⋮							

(a) Tabla de parejas ordenadas

	0	1	2	3	4	5	6	⋯
0	**0**	2	5	9	14	20	27	⋯
1	**1**	4	8	13	19	26		
2	**3**	7	12	18	25			
3	**6**	11	17	24				
4	**10**	16	23					
5	**15**	22						
6	**21**							
⋮	⋮							

(b) RANK en las parejas ordenadas

Tabla 1: Esquema de enumeración de parejas ordenadas

Para que la enumeración tenga sentido dentro de estas diagonales se requiere dar un ordenamiento de estas composiciones así como una forma de generarlas en tal orden. De aquí la necesidad de trabajar las k-composiciones.

Como se puede observar en diversos algoritmos de generación de distintas estructuras combinatorias en [4], las composiciones pueden a su vez codificar otras estructuras combinatorias como multiconjuntos. De hecho, en este trabajo se codifican las composiciones mediante su representación por separadores para un manejo más amigable en el proceso de generación. Este tipo de traducciones de una estructura combinatoria en otra, son comunes en el estudio de la generación de las mismas. Consecuentemente, con esta biblioteca de generación de composiciones, queda abierta la posibilidad de representar otras estructuras combinatorias en *Coq*, para las cuales sólo se requieran funciones de traducción.

Siendo así, este proyecto se delimita a plantar una biblioteca básica en *Coq* para la representación de las composiciones de un número, así como una manera de generarlas. Esto se logra mediante funciones HASNEXT y NEXT con las que se pueden generar las composiciones. La función NEXT representa la función sucesor para composiciones provisto de un orden específico.

El proyecto medio de esta misma clase se encargó de realizar la tarea para composiciones en dos y tres partes, el presente proyecto se emplea para generalizar estas bibliotecas a que acoplen k-composiciones para una k provista. Es importante mencionar que hay diferencias entre las bibliotecas de dos y tres composiciones respecto a k-composiciones tanto en la forma de nombrar ciertas relaciones como en la forma de ordenar las composiciones. Esto se hizo para facilitar la integración de las ideas de generación a estructuras ya dadas en *Coq*.

2. Esbozo de la solución

Procedamos a mostrar cuál es el planteamiento para ordenar las composiciones, lo que se hará mediante la representación por separadores de una composición. Se planteará cómo obtener el sucesor de una composición dado el orden provisto. Es importante notar que habrá diferencias en el orden de las composiciones respecto a cómo se trabajó en las bibliotecas para 2 y 3-composiciones.

Definición 1. Sean $n, k \in \mathbb{N}$ con $k \geq 0$.

1. Definimos una $k+1$-*composición* o una *composición en* $k+1$ *partes* como una $k+1$-tupla $(x_1, \ldots, x_{k+1})$ de naturales tales que $\sum_{i=1}^{k+1} x_i = n$
2. La n-ésima $k+1$-*diagonal* como $\text{Diag}_{k+1}(n) := \{(x_1, \ldots, x_k) \in \mathbb{N}^{k+1} \mid \sum_{i=1}^{k+1} x_i = n\}$.

Es importante hacer hincapié en que las composiciones no pueden ser vacías.

Dado que en una composición es fundamental que los valores sumen n, podemos descomponer a este número en unidades (expresadas en estrellas). Al representarlo de esta manera podemos generar una composición sencillamente particionando estas unidades en subconjuntos. Cada subconjunto representará un sumando de la composición. Dado que es relevante el orden en que se presentan los sumandos y no importa si hay sumandos repetidos, bastará con alinear las unidades y poner separadores (barras) en estas unidades. Por ejemplo, la representación en barras y estrellas de la terna ordenada $(2, 1, 3)$ es ★★ | ★ | ★★★. Podemos observar que la tarea de generar todas las composiciones se reduce a seleccionar todas las posibles colocaciones de los separadores.

La cantidad de estrellas determina el número a componer n y la cantidad de barras más uno representa la cantidad de partes $k + 1$. Dos ejemplos de esto se pueden apreciar en la figura 1.

$(1, 1, 0, 3)$	$\longrightarrow$	★	|	★	|	|	★	★	★	$\longrightarrow$ $(4, 3, 3)$
		1	2	3	4	5	6	7	8	
$(2, 0, 2, 1)$	$\longrightarrow$	★	★	|	|	★	★	|	★	$\longrightarrow$ $(3, 3, 1)$
		1	2	3	4	5	6	7	8	

(a) 4-tuplas (b) Representación barras y estrellas (c) Separadores

Figura 1: Ejemplos de representaciones de 4-tuplas

Ahora bien, sabiendo que los separadores determinan completamente una k-composición de n, basta con codificar las colocaciones de los separadores. Una forma adecuada es almacenando en una tupla la cantidad de estrellas posteriores a un separador. En la figura 1c se ven ejemplos de la representación por separadores de las tuplas en 1a.

Cabe mencionar que bien se pudo tomar la representación por separadores como la cantidad de estrellas que anteceden a un separador dado (que de hecho es como se manejó en un principio). Teóricamente, ambas representaciones son igual de válidas y producen órdenes distintos pero homólogos. Se optó por cambiar a la segunda representación pues facilitaba su implementación en *Coq* al trabajar con cabezas de listas en lugar de con el último elemento de éstas.

La representación por separadores ordenadas lexicográficamente resulta en un orden de lo más natural, además de simplificar el cómo se enumeran las composiciones.

La representación por separadores fija la cantidad de partes de la composición pero no es única si se permite variar el número a componer. Por ejemplo $(3, 2, 1)$ es una representación por separadores de $\mathbf{5} + 1 + 1 + 1 = 8$ como de $\mathbf{6} + 1 + 1 + 1 = 9$. Sin embargo, si fijamos la n a componer, la representación se vuelve única y la longitud determina la cantidad de partes de la composición.

Observemos que la representación por separadores es una tupla de naturales decrecientes todos menores que n (con la otra definición serían crecientes).

Definición 2. Sean $n, k \in \mathbb{N}$ con $k \geq 0$ y $(x_1, \ldots, x_k, x_{k+1})$ una $k+1$-composición de n. Decimos que $(a_1, \ldots, a_k) \in \mathbb{N}^k$ es la *representación por separadores* de $(x_1, \ldots, x_{k+1})$ si para cualquier $j \in \{2, \ldots, k+1\}$ se cumple que $a_j = \sum_{i=j}^{k+1} x_j$.

No es complicado ver que si tomamos a los separadores como en la definición 2 estaremos en el caso de una tupla decreciente de valores acotados por n. En *Coq* le daremos la vuelta a

este hecho, y se definirá una representación por separadores como una simple lista decreciente de naturales. Sabiendo que se tiene esta equivalencia, se definirán las funciones con las que traduciremos una representación por separadores en una composición y viceversa, siguiendo las siguientes especificaciones.

Si $(a_1, \ldots, a_k)$ es una representación por separadores de n, entonces

$$\textsc{ToCom}((a_1, \ldots, a_k), n) := (a_i - a_{i+1})_{i=0}^{k} \text{ donde } a_0 = n \text{ y } a_{k+1} = 0. \tag{1}$$

Si $(x_1, \ldots, x_{k+1})$ es una $k+1$-composición de n, entonces

$$\textsc{ToSep}(x_1, \ldots, x_{k+1}) := \Big(\sum_{i=j}^{k+1} x_i\Big)_{i=1}^{k}. \tag{2}$$

En el proyecto se definen estas funciones y se muestra que son una inversa de la otra y cuya imagen es siempre una representación por separadores o una composición respectivamente.

Teniendo claro las dos estructuras combinatorias con las que se modela una composición, se utiliza el orden lexicográfico en la representación por separadores para dotar a las composiciones. Es sobre esta representación en la que se plantean las funciones hasNext y Next.

d	(a_1, a_2)	BarrEst	(x, y, z)
0	(0,0)	★★★★ ||	(4,0,0)
1	(1,0)	★★★ | ★ |	(3,1,0)
2	(1,1)	★★★ || ★	(3,0,1)
3	(2,0)	★★ | ★★ |	(2,2,0)
4	(2,1)	★★ | ★ | ★	(2,1,1)
5	(2,2)	★★ || ★★	(2,0,2)
6	(3,0)	★ | ★★★ |	(1,3,0)
7	(3,1)	★ | ★★ | ★	(1,2,1)
8	(3,2)	★ | ★ | ★★	(1,1,2)
9	(3,3)	★ || ★★★	(1,0,3)
10	(4,0)	| ★★★★ |	(0,4,0)
11	(4,1)	| ★★★ | ★	(0,3,1)
12	(4,2)	| ★★ | ★★	(0,2,2)
13	(4,3)	| ★ | ★★★	(0,1,3)
14	(4,4)	|| ★★★★	(0,0,4)

Tabla 2: Ordenamiento de las ternas en $\text{Diag}_3(4)$.

Se verifica que para n y k fijas, todas las tuplas descendientes acotadas por n de tamaño k corresponden con todas las $k+1$-composiciones de n. Más aún, las tuplas descendientes con el orden lexicográfico es un orden total con mínimo y máximo. Esto último se verifica dentro del proyecto.

Podemos también observar que la función hasNext devuelve un booleando que nos acerta si estamos en la tupla máxima. Esto lo hace verificando si la entrada no es la tupla $(4, 4)$ corroborando si la cabeza es menor estrictamente a n; y en el caso de ser igual; verificar si la cola de la tupla cumple recursivamente esta cuota. Dicha función toma como parámetro una tupla y un valor n con el cual verificar si no se sobrepasa.

Para la función Next, se verifica si la tupla en efecto satisface tener sucesor mediante la función hasNext. En caso de sí poseer un sucesor, se busca en el separador más a la derecha e intenta moverlo un espacio a la izquierda. Si el separador, se topa con al menos un separador (pueden ser varios) entre estos separadores, mueve el más a la izquierda un lugar y regresa todos

los anteriores a su colocación más a la derecha. De esta forma garantizamos que se mueven todos los separadores y ninguno se rebasa, evitando así repeticiones. El procedimiento se puede ver visualmente en la tabla 2. Se puede plantear este procedimiento en la representación de separadores con sólo hacer comparaciones entre las entradas, que es como se plantea en el proyecto.

En [4] se pueden ver algoritmos generativos de este tipo de estructuras cuyos patrones son un tanto similares al empleado para generar las composiciones.

3. Modelado en Coq

Se utilizó las listas de la biblioteca de *List* para modelar tanto las composiciones como las representaciones aprovechando así, muchos de los teoremas ya probados. También se usó la biblioteca de *Peano nat* para la representación de los números naturales ya que no se precisó explotar alguna representación distinta a la usual.

Listas decrecientes

Para empezar a modelar la representación por separadores se planteo una relación que afirma si una lista de naturales es decreciente. En esta sección también se define la relación de orden lexicográfico con el cual se utilizará en la enumeración de composiciones. Esto en los archivos *Defs_des.v*, *Props_des.v* y *Props_Orddes.v*. Algunas de las definiciones importantes son

- *Des*: Relación que afirma si una lista es de valores descendientes.
- *allZeros*: Relación que afirma si una lista contiene puros ceros.
- *alln*: Relación que afirma si una lista contiene puros valores iguales.
- *leDes*: Relación de orden lexicográfico en listas en su versión reflexiva.

Se muestra *leDes* corresponde con un orden es total reflexivo en los teoremas *leDes_refl*, *leDes_antisym*, *leDes_trans* y *leDes_dich*.

Listas de separadores

Como se explicó anteriormente, una lista decreciente corresponde con una representación por separadores y se observa que una representación puede codificar múltiples composiciones. La forma de forzar la unicidad de composición se logra dando parámetros n y k que indiquen el número que se compone así como en cuantas partes respectivamente. Siendo así, en los archivos *Defs_des.v*,*Props_ssep.v* y *Props_Ordsep.v* se implementan estas restricciones.

- *SepL n k*: Relación que afirma si una lista L corresponde con la representación por separadores de una $k+1$ composición de n. Particularmente

$$Sep\ L\ n\ k := Des\ L \wedge \mathrm{hd}\ L \leq n \wedge \mathrm{length}\ L = k.$$

- *leSep*: Relación de orden reflexivo resultante de restringir el orden en *Des* para el caso de separadores. Particularmente

$$leSep\ A\ B\ n\ k := SepA\ n\ k \wedge Sep\ B\ n\ k \wedge A \leq_{\mathsf{Des}} B.$$

Las propiedades más relevantes para esta sección son las que afirman que este orden es total reflexivo que posee mínimo y máximo. Éstas corresponden exactamente con los teoremas *leSep_refl*, *leSep_antisym*, *leSep_trans*, *leSep_dich*, *leSep_Min* y *leSep_Max*. Se muestra que una lista de puros valores cero, es decir que cumpla con *allZeros* es mínima y análogamente que una lista que cumpla con *alln* n es máxima.

Propiedades auxiliares

Para poder plantear las funciones que transforman una representación por separadores en una composición y viceversa se precisan de ciertas definiciones y propiedades auxiliares. Éstas se trabajan en los archivos *Defs_aux.v* y *Props_aux.v*. Algunas de las definiciones y teoremas más importantes son:

- *Sum* L : Función que suma todos los elementos de una lista L.
- *zeros* k: Función que devuelve una lista de longitud k con puros valores 0.
- *enn* n k: Función que devuelve una lista de longitud k con puros valores repetidos en n.
- *zeros_Sep_Min*: Teorema que afirma que la lista *zeros* k, es mínima en el orden $\leq_{Sep}$ respecto a k.
- *enn_Sep_Max*: Teorema que afirma que la lista *enn* n k, es máxima en el orden $\leq_{Sep}$ respecto a n y k.

Funciones next y hasNext

Una vez planteadas las relaciones que determinan si una lista corresponde con una $k+1$-composición de un valor n. Resulta importante plantear la función sucesor NEXT en este orden. Recodando que este orden es finito en , no todos los elementos tienen sucesor, por lo que se define la función HASNEXT que hace esta verificación y que es indispensable para el correcto cálculo del sucesor. Estas funciones se plantean en los archivos *Defs_next.v* y *Props_next.v*.

- HASNEXTL n: Función booleana diseñada para listas descendientes que devuelve verdadero cuando L tiene sucesor respecto a n. Particularmente

```
Fixpoint hasNext( l : list nat)(n:nat):bool :=
   match l with
         | [] ⇒ false
         | h :: t ⇒ (h <?n)||((h =?n)&&(hasNext t h))
         end.
```

- NEXTL: Función que devuelve una lista correspondiente al sucesor en el orden lexicográfico de listas de valores descendientes.

```
Fixpoint NEXT( l :list nat): list nat:=
   match l with
         | [] ⇒ []
         | h :: t ⇒ match (hasNEXT t h) with
                | true ⇒  h :: (NEXTt)
                | false ⇒ (S h) :: (zeros(length t))
               end
        end.
```

Nota 1. Estas funciones están definidas para todas las listas. En el caso de que no se apliquen a listas que implementen una representación por separadores, las funciones hasNext y next tienen comportamientos distintos a los deseados en general.

Algunos de los teoremas importantes sobre las propiedad de estas funciones son las siguientes.

- *next_is_Sep_hN L n k*: si l es una lista de longitud k que tenga sucesor será Next$L$ será una representación por separadores.
- *next_leSep_hN L*: toda lista l es menor o igual a su Next.
- *next_is_zeros_iff_nil*: toda lista A con parámetro k cuyo next sea vacío es vacío y $k = 0$.
- *next_not_idem*: toda lista no vacía es distinta a sus sucesor.
- *next_sandwich*: para toda pareja de listas A y B que sean descendientes, de misma longitud, si B es mayor o igual a A pero menor o igual la sucesor de A entonces B tiene que ser alguno de los dos.
- *next_inj*: el operador sucesor para listas es inyectivo.

Listas de composiciones

Hasta ahora se trabajó a las composiciones por su representación como separadores. Se plantea una relación que trabaja las composiciones de la forma usual como una tupla de longitud k que sume un valor n. En los archivos *Defs_com.v* y *Props_com.v*.

La relación *Com L n k* justo afirma que la suma de los elementos de L es n y además que tienen longitud $k + 1$. Algunas de las funciones, relaciones y teoremas importantes son los siguientes:

- *remove_head_is_com*: Si a una $k + 2$ composición n le retiramos su cabeza a nos queda una $k + 1$ composición de $n - a$.
- *ext_head_is_com*: Si a una $k+1$ composición de n le añadimos un elemento a a la cabeza tenemos una $k + 2$ composición de $n + a$.
- *exxl*: función que representa una composición extrema de n donde todos los valores de la tupla valen cero excepto el de la más izquierda.

- *exxr*función que representa una composición extrema de n donde todos los valores de la tupla valen cero excepto el de la más derecha.
- *exxr_is_com*: toda tupla extrema derecha es una composición.
- *exxl_is_com*: toda tupla extrema izquierda es una composición.

Funciones de transformación de representaciones

Finalmente, se muestra la equivalencia entre la representación por separadores y una composición a través de las funciones de trasformación. Las funciones *ToCom* y *ToSep* se definen en el archivo *Defs_com.v*. Las propiedades de estas funciones se plantean en el archivo *Props_transform.v*

- La implementación de la función planteada en la ecuación 2 que dada una lista correspondiente a una composición devuelve una lista correspondiente a su representación por separadores.

```
Fixpoint ToSep(l:list nat):list nat:=
   match l with
         |  []  → [](*si Com l, entonces caso imposible*)
         |  _ :: t  → match t with
               |  []  →  []
               |  _ :: _ → Sumt :: (ToSept)
              end
        end.
```

- Función auxiliar para transformar una representación por separadores en una composición.

```
Fixpoint  ToComAux (l:list nat):list nat: =
   match l with
         | [] → []
         | a :: t → match t with
               | [] → []
               |  b :: _  → (a − b) :: (  ToComAux (t))
              end
        end.
```

- Función que dada una representación por separadores un un natural devuelve la composición de tal natural correspondiente. Esta es la implemetación de la función planteada en la ecuación 1.

$$ToCom(l\ n) := ToComAux(n :: (l + +[0])).$$

Los teoremas relevantes sobre estas transformaciones son los siguientes:

- *ToCom_is_Com*: Aplicar la función *ToCom* a listas que sean representaciones por separadores arroja una composición con los mismos parámetros.

- *ToSep_is_sep*: Transformar una composición en su representación por separadores es, en efecto un representación por separadores.
- *ToCom_ToSep_inv*: La función *ToCom* es inversa izquierda de la función *ToSep*.
- *ToSep_ToCom_inv*: La función *ToSep* es inversa izquierda de la función *ToCom* mostrándose que son biyectivas y bien definidas.
- *ToSep_exxr_is_enn*: La función extremo derecho se traduce en el elemento máximo en la enumeración, esto es *enn*.
- *ToSep_exxl_is_zeros*:La función extremo izquierdo se traduce en el elemento mínimo en la enumeración, esto es *zeros*.

Referencias

[1] G. Winskel, *The formal semantics of programming languages: an introduction*. MIT press, 1993.

[2] R. J. Solomonoff, "A formal theory of inductive inference. Part I," *Information and control*, vol. 7, no. 1, pp. 1–22, 1964. Publisher: Elsevier.

[3] M. V. Lemus Yáñez, *Aproximación a la distribución universal por medio de simulaciones exhaustivas*. Proyecto de investgación para el Doctorado en Ciencias de la computación, Universidad Nacional Autónoma de México, 2019.

[4] D. Walden and B. Addison-Wesley, "An appreciation: The art of computer programming, volume 4a," *TUGboat-TeX Users Group*, vol. 32, no. 2, p. 230, 2011.