Reporte final

Enumeración de programas IMP

Universidad Nacional Autónoma de México
Posgrado en Ciencia e Ingeniería de la Computación
Verificación Formal
Semestre 2021–1

Eduardo Acuña Yeomans †

Resumen. En este trabajo se explora desde una perspectiva de verificación formal la enumeración de programas de un lenguaje simple e imperativo llamado IMP. El demostrador interactivo de teoremas Coq es utilizado para implementar los planteamientos presentes en este documento, los cuáles forman parte de el proyecto de tesis del autor.

1. Contexto. El concepto de *enumeración* es utilizado para referirse a distintos procesos u objetos dependiendo del contexto. Desde la perspectiva de teoría de conjuntos, una enumeración es una relación biyectiva entre un conjunto de elementos y los números naturales. [1]

A pesar de proveer una definición clara y precisa, la perspectiva de teoría de conjuntos dice poco sobre cómo trabajar con enumeraciones desde un enfoque algorítmico: ¿Cómo se representa una enumeración en la computadora? ¿Qué procesos computacionales pueden manipular estas representaciones? ¿Qué podemos aprender de la enumeración a partir de algoritmos que la producen?.

En las áreas de computabilidad y complejidad computacional, el término enumeración es asociado a una amplia variedad de formalismos, mientras que el *acto* de enumerar se utiliza en estos contextos para describir el proceso mediante el cuál se recorre sobre todos los elementos de un conjunto de forma ordenada hasta encontrar uno que satisface alguna propiedad deseada. [2, 3, 4]

En este trabajo se exploran las enumeraciones desde una perspectiva algorítmica, basado en algunas perspectivas del área de combinatoria. Donald Knuth describe que algunos autores se refieren al proceso de recorrer todas las posibilidades de algún universo combinatorio como "enumerar" o "listar" todas las posibilidades, sin embargo se opone a estos usos argumentando que enumerar es usualmente asociada al acto de *contar* las posibilidades y que listar implica imprimir o almacenar todas las posibilidades. [5]

Kreher y Stinson plantean una distinción similar y al igual que Knuth proponen como alternativa el término de *generación* de objetos combinatorios, complementando este concepto con el de *visitarlos*, es decir, procesar un objeto sin tener que generarlos todos. [6]

Frank Ruskey, en su trabajo preliminar titulado "Combinatorial Generation", plantea la exploración de cuatro diferentes problemas en el contexto de generación de posibilidades. [7] El primero consiste en generar objetos combinatorios en orden a partir de un primer elemento y una operación sucesor. Otro consiste en seleccionar de forma aleatoria

† eduardo.acye@gmail.com

un elemento entre todas las posibilidades. Finalmente, los últimos dos problemas corresponden a dos procesos complementarios, el de *ranking* y *unranking*, los cuáles se abordan en este trabajo como mecanismos para codificar las enumeraciones.

A partir de un ordenamiento de objetos, el *rank* (o rango) de un objeto es la posición que el objeto ocupa en el ordenamiento. Al contar a partir del cero, esta posición es también la cantidad de objetos que lo preceden en el ordenamiento. El proceso que describe cómo encontrar el rank de un objeto es llamado *ranking* y el proceso inverso, es decir, el mecanismo para encontrar el objeto que ocupa cierta posición, es llamado *unranking*.

Frank Ruskey menciona que uno de los usos principales de los algoritmos de ranking es como funciones hash perfectas, mientras que los algoritmos de unranking pueden ser utilizados para implementar algoritmos paralelos para generar objetos combinatorios cuando se incorpora el procedimiento sucesor. Finalmente menciona que en general, el ranking y el unranking solo son posibles con algunos objetos combinatorios elementales. [7]

En este trabajo, el término enumeración se refiere a la existencia de la función computable e invertible rank, así como a la relación biyectiva inducida por rank sobre los naturales.

2. Lenguaje de programación IMP. La finalidad de este trabajo es enumerar programas del lenguaje de programación IMP. Este lenguaje es simple e imperativo, sus programas $\boldsymbol{P}$ se construyen a partir de localidades de memoria $\boldsymbol{X}$, expresiones aritméticas $\boldsymbol{A}$ y expresiones booleanas $\boldsymbol{B}$ de acuerdo a la siguiente gramática.

$$
\begin{aligned}
\boldsymbol{P} &\to \mathit{skip} \mid \boldsymbol{X} := \boldsymbol{A} \mid (\mathit{if}\ \boldsymbol{B}\ \mathit{then}\ \boldsymbol{P}\ \mathit{else}\ \boldsymbol{P}) \\
&\quad \mid (\mathit{while}\ \boldsymbol{B}\ \mathit{do}\ \boldsymbol{P}) \mid (\boldsymbol{P}\,;\boldsymbol{P}) \\
\boldsymbol{A} &\to \mathbb{N} \mid \boldsymbol{X} \mid (\boldsymbol{A}+\boldsymbol{A}) \mid (\boldsymbol{A}-\boldsymbol{A}) \mid (\boldsymbol{A}\times\boldsymbol{A}) \\
\boldsymbol{B} &\to \mathit{true} \mid \mathit{false} \mid (\boldsymbol{A}=\boldsymbol{A}) \mid (\boldsymbol{A}<\boldsymbol{A}) \\
&\quad \mid \neg\boldsymbol{B} \mid (\boldsymbol{B}\vee\boldsymbol{B}) \mid (\boldsymbol{B}\wedge\boldsymbol{B}) \\
\boldsymbol{X} &\to x_{\mathbb{N}}
\end{aligned}
$$

Su semántica es bastante intuitiva, las localidades de memoria se interpretan como el valor numérico almacenado en ellas, $x_n := a$ almacena en la n-ésima localidad el valor de la expresión a, el programa *skip* no realiza operación alguna y $(P_1\ ;\ P_2)$ denota la ejecución secuencial del programa P_1 y posteriormente de P_2.

La gramática de IMP describe un conjunto no-acotado de cadenas de símbolos, este conjunto puede ser ordenado de una gran cantidad de maneras para implementar la enumeración. En el proyecto del cuál deriva este trabajo, un aspecto fundamental para la enumeración del lenguaje es que su orden corresponda a alguna métrica de tamaño asociada a las cadenas, en particular, para cualesquiera dos programas P_1 y P_2 tales que $\mathrm{rank}(P_1) = n_1$ y $\mathrm{rank}(P_2) = n_2$, el tamaño de P_1 es menor o igual al tamaño de P_2 si y solo si $n_1 \leq n_2$.

Tanto en la tesis como en el presente trabajo se considera una restricción menos fuerte para la enumeración: Si una secuencia de programas $P_1, \ldots, P_n$ se combinan sintácticamente para formar un programa P, el rank de cada P_i debe ser estrictamente menor al rank de P. Esta restricción permite implementar el orden de manera independiente a

cualquier métrica de tamaño y también es un principio de composicionalidad útil al diseñar los algoritmos de rank y unrank.

Los algoritmos de rank y unrank se desarrollaron para ser parametrizados por la especificación del lenguaje. Esta especificación difiere de gramáticas libres de contexto en algunos puntos importantes:

- El orden en que aparecen las reglas de producción asociada a cada variable sintáctica es relevante.
- Las reglas de producción que describen una cantidad finita de cadenas deben aparecer al inicio en este orden.
- En lugar de cadenas de símbolos, la especificación describe la construcción de árboles de sintaxis, es decir, el azúcar sintáctico es descartado y en su lugar se asocia una etiqueta única con la que se identifican los vértices del árbol de sintaxis.

La especificación de IMP se presenta en el siguiente esquema con un estilo similar a la gramática. Cada árbol sintáctico se representa textualmente utilizando una notación prefija de lista cuyo primer elemento denota la etiqueta asociada a la regla de producción.

$$
\begin{aligned}
\boldsymbol{P} &\to \mathit{skip} \mid (\mathit{set}\ \boldsymbol{X}\ \boldsymbol{A}) \mid (\mathit{if}\ \boldsymbol{B}\ \boldsymbol{P}\ \boldsymbol{P}) \\
&\quad \mid (\mathit{while}\ \boldsymbol{B}\ \boldsymbol{P}) \mid (\mathit{seq}\ \boldsymbol{P}\ \boldsymbol{P}) \\
\boldsymbol{A} &\to \mathbb{N} \mid \boldsymbol{X} \mid (\mathit{plus}\ \boldsymbol{A}\ \boldsymbol{A}) \mid (\mathit{minus}\ \boldsymbol{A}\ \boldsymbol{A}) \mid (\mathit{mult}\ \boldsymbol{A}\ \boldsymbol{A}) \\
\boldsymbol{B} &\to \mathit{true} \mid \mathit{false} \mid (\mathit{equal}\ \boldsymbol{A}\ \boldsymbol{A}) \mid (\mathit{less}\ \boldsymbol{A}\ \boldsymbol{A}) \\
&\quad \mid (\mathit{not}\ \boldsymbol{B}) \mid (\mathit{or}\ \boldsymbol{B}\ \boldsymbol{B}) \mid (\mathit{and}\ \boldsymbol{B}\ \boldsymbol{B}) \\
\boldsymbol{X} &\to (\mathit{loc}\ \mathbb{N})
\end{aligned}
$$

Implementar los algoritmos de rank y unrank en términos de este formalismo permite cierto grado de control sobre el orden de la enumeración resultante. Un cambio en el orden de las producciones describe el mismo conjunto de árboles de sintaxis, sin embargo, induce un orden distinto en la enumeración.

Adicionalmente, ya que los programas se codifican con sus árboles de sintaxis, se evita el proceso de reconocimiento léxico y sintáctico en caso que se desee interpretar o compilar un programa.

3. Algoritmo de enumeración. La enumeración de la especificación de IMP se representa con la función $\mathrm{rank}_{\boldsymbol{P}}$, la cuál describe la posición de cada programa. Esta función a su vez se define en términos de las enumeraciones asociadas a localidades de memoria $\mathrm{rank}_{\boldsymbol{X}}$, expresiones aritméticas $\mathrm{rank}_{\boldsymbol{A}}$ y expresiones booleanas $\mathrm{rank}_{\boldsymbol{B}}$. De tal forma que para cada variable sintáctica que represente una cantidad no-acotada de árboles de sintaxis, se define una enumeración asociada.

La estrategia para implementar las funciones rank y unrank consiste en distribuir los números naturales en las reglas de producción de una variable sintáctica de acuerdo a los siguientes pasos.

1. Sea F la lista de reglas de producción que describen una cantidad finita de objetos y R la lista de reglas de producción que describen una cantidad no-acotada de objetos.
2. Sea n la cantidad de objetos que producen las reglas en F y r la cantidad de reglas en R.

3. Los objetos producidos por F son asociados a los naturales del 0 al $n-1$.
4. Para cada i entre 0 y $r-1$, se le asocia a la i-ésima regla de producción en R los naturales k tales que $i \equiv (k-n) \bmod r$.

Para ilustrar estos pasos en el contexto de la especificación de IMP, se muestra a continuación cómo se distribuyen los naturales sobre las reglas de producción de cada variable sintáctica. Del lado izquierdo de describe la perspectiva de la operación rank y del lado derecho la perspectiva de la operación unrank, en ambos casos es fácil identificar la estructura sintáctica de un árbol E o la divisibilidad de un natural k.

$$
\begin{array}{rcl}
\underline{\mathrm{rank}_{\boldsymbol{P}}(E) = k} & & \underline{\mathrm{unrank}_{\boldsymbol{P}}(k) = E} \\
\mathit{skip} & \iff & 0 \equiv k \\
(\mathit{set}\ \boldsymbol{X}\ \boldsymbol{A}) & \iff & 0 \equiv (k-1) \bmod 4 \\
(\mathit{if}\ \boldsymbol{B}\ \boldsymbol{P}\ \boldsymbol{P}) & \iff & 1 \equiv (k-1) \bmod 4 \\
(\mathit{while}\ \boldsymbol{B}\ \boldsymbol{P}) & \iff & 2 \equiv (k-1) \bmod 4 \\
(\mathit{seq}\ \boldsymbol{P}\ \boldsymbol{P}) & \iff & 3 \equiv (k-1) \bmod 4
\end{array}
$$

$$
\begin{array}{rcl}
\underline{\mathrm{rank}_{\boldsymbol{A}}(E) = k} & & \underline{\mathrm{unrank}_{\boldsymbol{A}}(k) = E} \\
\mathbb{N} & \iff & 0 \equiv k \bmod 5 \\
\boldsymbol{X} & \iff & 1 \equiv k \bmod 5 \\
(\mathit{plus}\ \boldsymbol{A}\ \boldsymbol{A}) & \iff & 2 \equiv k \bmod 5 \\
(\mathit{minus}\ \boldsymbol{A}\ \boldsymbol{A}) & \iff & 3 \equiv k \bmod 5 \\
(\mathit{mult}\ \boldsymbol{A}\ \boldsymbol{A}) & \iff & 4 \equiv k \bmod 5
\end{array}
$$

$$
\begin{array}{rcl}
\underline{\mathrm{rank}_{\boldsymbol{B}}(E) = k} & & \underline{\mathrm{unrank}_{\boldsymbol{B}}(k) = E} \\
\mathit{true} & \iff & 0 \equiv k \\
\mathit{false} & \iff & 1 \equiv k \\
(\mathit{equal}\ \boldsymbol{A}\ \boldsymbol{A}) & \iff & 0 \equiv (k-2) \bmod 5 \\
(\mathit{less}\ \boldsymbol{A}\ \boldsymbol{A}) & \iff & 1 \equiv (k-2) \bmod 5 \\
(\mathit{not}\ \boldsymbol{B}) & \iff & 2 \equiv (k-2) \bmod 5 \\
(\mathit{or}\ \boldsymbol{B}\ \boldsymbol{B}) & \iff & 3 \equiv (k-2) \bmod 5 \\
(\mathit{and}\ \boldsymbol{B}\ \boldsymbol{B}) & \iff & 4 \equiv (k-2) \bmod 5
\end{array}
$$

$$
\begin{array}{rcl}
\underline{\mathrm{rank}_{\boldsymbol{X}}(E) = k} & & \underline{\mathrm{unrank}_{\boldsymbol{X}}(k) = E} \\
(\mathit{loc}\ \mathbb{N}) & \iff & 0 \equiv (k-0) \bmod 1
\end{array}
$$

Este primer análisis permite identificar la forma de los naturales asociados a una regla de producción en R. Por ejemplo, en el caso de las iteraciones *while* se asocia a ellas todos los naturales k tales que $(k-1) \bmod 4$ es 2, en orden creciente son $3, 7, 11, 15, 19, 23, 27, \ldots$, este conjunto de números puede ser a su vez enumerado con operaciones rank y unrank.

$$
\mathrm{rank}_{[i \equiv (k-n) \bmod r]}(k) = \frac{k-n-i}{r} \qquad \mathrm{unrank}_{[i \equiv (k-n) \bmod r]}(k') = rk' + n + i
$$

Estas definiciones se obtienen a partir de unrank, observando que para todo natural k', si $k = rk' + n + i$ entonces $i \equiv (k - n) \bmod r$.

$$\begin{aligned} i \equiv (k-n) \bmod r &\implies i \equiv (rk' + n + i - n) \bmod r \\ &\implies i \equiv (rk' + i) \bmod r \\ &\implies 0 \equiv (rk') \bmod r \end{aligned}$$

La definición de rank se define como la inversa de unrank.

Hasta este punto del algoritmo, si aún no se ha determinado la asociación entre un árbol sintáctico y un natural, entonces se cuenta con la producción en R correspondiente al natural k, así como los valores i, n, r y k'. En la siguiente etapa del algoritmo de enumeración, se analiza la cantidad de subexpresiones que se deben enumerar. La especificación de IMP presenta tres casos que se deben contemplar.

- Cuando el árbol de sintaxis se construye a partir de una expresión que no se ha procesado, como es el caso de las producciones $\boldsymbol{A} \to \mathbb{N}$, $\boldsymbol{A} \to \boldsymbol{X}$ y $\boldsymbol{B} \to (\mathit{loc}\ \mathbb{N})$.
- Cuando el árbol de sintaxis se construye a partir de dos expresiones que no se han procesado, como es el caso de la mayoría de las producciones.
- Cuando el árbol de sintaxis se construye a partir de tres expresiones que no se han procesado, como es el caso de la producción $\boldsymbol{P} \to (\mathit{if}\ \boldsymbol{B}\ \boldsymbol{P}\ \boldsymbol{P})$.

El formalismo de la especificación no establece un límite en la cantidad de subexpresiones en una misma producción, por lo tanto el algoritmo debe capturar estos casos de una forma general.

La estrategia en la segunda etapa del algoritmo se basa en enumeraciones de tuplas de naturales, para cada $j > 0$ se utilizan operaciones rank_j y unrank_j. Estas enumeraciones son utilizadas para partir o combinar el natural k' en tantas subexpresiones como sea necesario para la producción correspondiente. De tal manera que para todo natural k', $\text{unrank}_j(k')$ es una j-tupla de naturales tal que cada componente de la j-tupla es a lo más k'.

Este mecanismo permite distribuir en el algoritmo unrank, la posición de una expresión en sus subexpresiones de acuerdo a los siguientes pasos. El valor de k' es conocido y el valor de E es calculado.

1. Sea k' el natural asociado a una expresión compuesta E con etiqueta t.
2. Sea j la cantidad de subexpresiones E_1, E_2, ... , E_j necesarias para construir E, cada una de tipo T_1, T_2, ... , T_j.
3. Sea $\text{unrank}_k(k')$ la j-tupla de naturales $(k_1, k_2, \ldots, k_j)$.
4. Se construye $E = (t\ \text{unrank}_{T_1}(k_1)\ \text{unrank}_{T_2}(k_2)\ \ldots\ \text{unrank}_{T_j}(k_j))$.

Para el algoritmo rank, se calcula la posición de una expresión a partir de las posiciones de sus subexpresiones de acuerdo a los siguientes pasos. El valor de E es conocido y el valor de k' es calculado.

1. Sea E la expresión con etiqueta t asociada al natural k'.
2. Sean E_1, E_2, ... , E_j las subexpresiones tales que $E = (t\ E_1\ E_2\ \ldots\ E_j)$, cada una de tipo T_1, T_2, ... , T_j.
3. Se calcula $k' = \text{rank}_j(\text{rank}_{T_1}(E_1), \text{rank}_{T_2}(E_2), \ldots, \text{rank}_{T_j}(E_j))$.

Para ilustrar estos pasos en el contexto de la especificación de IMP, se muestra a continuación cómo se distribuye la posición k' sobre las subexpresiones de cada regla de producción para cada variable sintáctica.

$$
\begin{array}{rcl}
\underline{\mathrm{rank}_{\boldsymbol{P}}(E) = k} & & \underline{k = 4k' + 1 + i} \\
E = (\mathit{set}\ X\ A) & \Longrightarrow & k' = \mathrm{rank}_2(\mathrm{rank}_{\boldsymbol{X}}(X), \mathrm{rank}_{\boldsymbol{A}}(A)) \\
E = (\mathit{if}\ B\ P_1\ P_2) & \Longrightarrow & k' = \mathrm{rank}_3(\mathrm{rank}_{\boldsymbol{B}}(B), \mathrm{rank}_{\boldsymbol{P}}(P_1), \mathrm{rank}_{\boldsymbol{P}}(P_2)) \\
E = (\mathit{while}\ B\ P) & \Longrightarrow & k' = \mathrm{rank}_2(\mathrm{rank}_{\boldsymbol{B}}(B), \mathrm{rank}_{\boldsymbol{P}}(P)) \\
E = (\mathit{seq}\ P_1\ P_2) & \Longrightarrow & k' = \mathrm{rank}_2(\mathrm{rank}_{\boldsymbol{P}}(P_1), \mathrm{rank}_{\boldsymbol{P}}(P_2))
\end{array}
$$

$$
\begin{array}{rcl}
\underline{\mathrm{unrank}_{\boldsymbol{P}}(k) = E} & & \underline{E = (t\ E_1\ \ldots\ E_j)} \\
k = 4k' + 1 & \Longrightarrow & \mathrm{unrank}_2(k') = (k_1, k_2) \\
& \Longrightarrow & E = (\mathit{set}\ \mathrm{unrank}_{\boldsymbol{X}}(k_1)\ \mathrm{unrank}_{\boldsymbol{A}}(k_2)) \\
k = 4k' + 2 & \Longrightarrow & \mathrm{unrank}_3(k') = (k_1, k_2, k_3) \\
& \Longrightarrow & E = (\mathit{if}\ \mathrm{unrank}_{\boldsymbol{B}}(k_1)\ \mathrm{unrank}_{\boldsymbol{P}}(k_2)\ \mathrm{unrank}_{\boldsymbol{P}}(k_3)) \\
k = 4k' + 3 & \Longrightarrow & \mathrm{unrank}_2(k') = (k_1, k_2) \\
& \Longrightarrow & E = (\mathit{while}\ \mathrm{unrank}_{\boldsymbol{B}}(k_1)\ \mathrm{unrank}_{\boldsymbol{P}}(k_2)) \\
k = 4k' + 4 & \Longrightarrow & \mathrm{unrank}_2(k') = (k_1, k_2) \\
& \Longrightarrow & E = (\mathit{seq}\ \mathrm{unrank}_{\boldsymbol{P}}(k_1)\ \mathrm{unrank}_{\boldsymbol{P}}(k_2))
\end{array}
$$

$$
\begin{array}{rcl}
\underline{\mathrm{rank}_{\boldsymbol{A}}(E) = k} & & \underline{k = 5k' + 0 + i} \\
E = N & \Longrightarrow & k' = \mathrm{rank}_1(\mathrm{rank}_{\mathbb{N}}(N)) \\
E = X & \Longrightarrow & k' = \mathrm{rank}_1(\mathrm{rank}_{\boldsymbol{X}}(X)) \\
E = (\mathit{plus}\ A_1\ A_2) & \Longrightarrow & k' = \mathrm{rank}_2(\mathrm{rank}_{\boldsymbol{A}}(A_1), \mathrm{rank}_{\boldsymbol{A}}(A_2)) \\
E = (\mathit{minus}\ A_1\ A_2) & \Longrightarrow & k' = \mathrm{rank}_2(\mathrm{rank}_{\boldsymbol{A}}(A_1), \mathrm{rank}_{\boldsymbol{A}}(A_2)) \\
E = (\mathit{mult}\ A_1\ A_2) & \Longrightarrow & k' = \mathrm{rank}_2(\mathrm{rank}_{\boldsymbol{A}}(A_1), \mathrm{rank}_{\boldsymbol{A}}(A_2))
\end{array}
$$

$$
\begin{array}{rcl}
\underline{\mathrm{unrank}_{\boldsymbol{A}}(k) = E} & & \underline{E = (t\ E_1\ \ldots\ E_j)} \\
k = 5k' + 0 & \Longrightarrow & \mathrm{unrank}_1(k') = (k_1) \\
& \Longrightarrow & E = \mathrm{unrank}_{\mathbb{N}}(k_1) \\
k = 5k' + 1 & \Longrightarrow & \mathrm{unrank}_1(k') = (k_1) \\
& \Longrightarrow & E = \mathrm{unrank}_{\boldsymbol{X}}(k_1) \\
k = 5k' + 2 & \Longrightarrow & \mathrm{unrank}_2(k') = (k_1, k_2) \\
& \Longrightarrow & E = (\mathit{plus}\ \mathrm{unrank}_{\boldsymbol{A}}(k_1)\ \mathrm{unrank}_{\boldsymbol{A}}(k_2)) \\
k = 5k' + 3 & \Longrightarrow & \mathrm{unrank}_2(k') = (k_1, k_2) \\
& \Longrightarrow & E = (\mathit{minus}\ \mathrm{unrank}_{\boldsymbol{A}}(k_1)\ \mathrm{unrank}_{\boldsymbol{A}}(k_2)) \\
k = 5k' + 4 & \Longrightarrow & \mathrm{unrank}_2(k') = (k_1, k_2) \\
& \Longrightarrow & E = (\mathit{mult}\ \mathrm{unrank}_{\boldsymbol{A}}(k_1)\ \mathrm{unrank}_{\boldsymbol{A}}(k_2))
\end{array}
$$

$$
\begin{array}{rcl}
\underline{\mathrm{rank}_{\boldsymbol{B}}(E) = k} & & \underline{k = 5k' + 2 + i} \\
E = (\mathit{equal}\ A_1\ A_2) & \Longrightarrow & k' = \mathrm{rank}_2(\mathrm{rank}_{\boldsymbol{A}}(A_1), \mathrm{rank}_{\boldsymbol{A}}(A_2)) \\
E = (\mathit{less}\ A_1\ A_2) & \Longrightarrow & k' = \mathrm{rank}_2(\mathrm{rank}_{\boldsymbol{A}}(A_1), \mathrm{rank}_{\boldsymbol{A}}(A_2)) \\
E = (\mathit{not}\ B) & \Longrightarrow & k' = \mathrm{rank}_1(\mathrm{rank}_{\boldsymbol{B}}(B)) \\
E = (\mathit{or}\ B_1\ B_2) & \Longrightarrow & k' = \mathrm{rank}_2(\mathrm{rank}_{\boldsymbol{B}}(B_1), \mathrm{rank}_{\boldsymbol{B}}(B_2)) \\
E = (\mathit{and}\ B_1\ B_2) & \Longrightarrow & k' = \mathrm{rank}_2(\mathrm{rank}_{\boldsymbol{B}}(B_1), \mathrm{rank}_{\boldsymbol{B}}(B_2))
\end{array}
$$

$$
\begin{array}{rcl}
\underline{\mathrm{unrank}_{\boldsymbol{B}}(k) = E} & & \underline{E = (t\ E_1\ \ldots\ E_j)} \\
k = 5k' + 2 + 0 & \Longrightarrow & \mathrm{unrank}_2(k') = (k_1, k_2) \\
& \Longrightarrow & E = (\mathit{equal}\ \mathrm{unrank}_{\boldsymbol{A}}(k_1)\ \mathrm{unrank}_{\boldsymbol{A}}(k_2)) \\
k = 5k' + 2 + 1 & \Longrightarrow & \mathrm{unrank}_2(k') = (k_1, k_2) \\
& \Longrightarrow & E = (\mathit{less}\ \mathrm{unrank}_{\boldsymbol{A}}(k_1)\ \mathrm{unrank}_{\boldsymbol{A}}(k_2)) \\
k = 5k' + 2 + 2 & \Longrightarrow & \mathrm{unrank}_1(k') = (k_1) \\
& \Longrightarrow & E = (\mathit{not}\ \mathrm{unrank}_{\boldsymbol{B}}(k_1)) \\
k = 5k' + 2 + 3 & \Longrightarrow & \mathrm{unrank}_2(k') = (k_1, k_2) \\
& \Longrightarrow & E = (\mathit{or}\ \mathrm{unrank}_{\boldsymbol{B}}(k_1)\ \mathrm{unrank}_{\boldsymbol{B}}(k_2)) \\
k = 5k' + 2 + 4 & \Longrightarrow & \mathrm{unrank}_2(k') = (k_1, k_2) \\
& \Longrightarrow & E = (\mathit{and}\ \mathrm{unrank}_{\boldsymbol{B}}(k_1)\ \mathrm{unrank}_{\boldsymbol{B}}(k_2))
\end{array}
$$

$$
\begin{array}{rcl}
\underline{\mathrm{rank}_{\boldsymbol{X}}(E) = k} & & \underline{k = k' + 0 + i} \\
E = (\mathit{loc}\ N) & \Longrightarrow & k' = \mathrm{rank}_1(\mathrm{rank}_{\mathbb{N}}(N))
\end{array}
$$

$$
\begin{array}{rcl}
\underline{\mathrm{unrank}_{\boldsymbol{X}}(k) = E} & & \underline{E = (t\ E_1\ \ldots\ E_j)} \\
k = k' + 0 + 0 & \Longrightarrow & \mathrm{unrank}_1(k') = (k_1) \\
& \Longrightarrow & E = (\mathit{loc}\ \mathrm{unrank}_{\mathbb{N}}(k_1))
\end{array}
$$

El proceso descrito se repite con las subexpresiones hasta reducir la posición a un objeto asociado en F.

4. Enumeración de tuplas. Describir a detalle algoritmos para la enumeración de tuplas queda fuera del alcance de este trabajo, uno de los objetivos que se plantean es definir el algoritmo de enumeración de programas de manera independiente al algoritmo específico utilizado para enumerar tuplas. Sin embargo, es necesario establecer algunas propiedades que deben satisfacer rank_j y unrank_j para que el algoritmo funcione correctamente y mostrar que existen dos funciones que satisfacen estas propiedades.

Al igual que otros procesos de enumeración, rank_j y unrank_j deben ser biyectivas e inversas una de la otra. Adicionalmente se incorpora la propiedad débil de orden descrita en la sección anterior, la cuál establece que las componentes de la tupla no deben ser mayores a la posición en el orden establecido.

En la práctica, es conveniente que la distribución de un natural en j partes sea relativamente uniforme, los algoritmos de enumeración de tuplas utilizados en el proyecto de

tesis garantizan adicionalmente que el orden de las tuplas es creciente con respecto a la suma de sus componentes. Sin embargo, para fines de la enumeración de programas, esta propiedad es meramente deseable, mas no necesaria.

La construcción de las funciones rank y unrank que satisfacen las propiedades descritas anteriormente se basa en separar los casos cuando $j = 1$, $j = 2$ y $j > 2$.

El caso trivial es enumerar 1-tuplas, ya que solo hay una componente se define $\text{rank}_1(k) = (k)$ y $\text{unrank}_1(k) = k$. Para enumerar 2-tuplas, se plantea primero analizar la operación de $unrank_2(k)$. Se representa a k en binario, y las componentes (k_1, k_2) de la tupla resultante son calculadas de la siguiente manera.

- k_1 es la cantidad de 1 consecutivos a partir del bit menos significativo.
- k_2 es la mitad del valor de la cadena de bits a partir del último 1 contemplado por k_1.

Otra forma de visualizar esta codificacieon es considerar el primer bit 0 menos significativo como un separador entre el valor unario de la primera componente y el valor binario de la segunda componente:

$$\begin{aligned}
\text{unrank}_2(0) &= \text{unrank}_2([\texttt{00000}]_2) = (0,0)\\
\text{unrank}_2(1) &= \text{unrank}_2([\texttt{0000}\underline{\texttt{1}}]_2) = (1,0)\\
\text{unrank}_2(2) &= \text{unrank}_2([\texttt{00010}]_2) = (0,1)\\
\text{unrank}_2(3) &= \text{unrank}_2([\texttt{000}\underline{\texttt{11}}]_2) = (2,0)\\
\text{unrank}_2(4) &= \text{unrank}_2([\texttt{00100}]_2) = (0,2)\\
\text{unrank}_2(5) &= \text{unrank}_2([\texttt{0010}\underline{\texttt{1}}]_2) = (1,1)\\
\text{unrank}_2(6) &= \text{unrank}_2([\texttt{00110}]_2) = (0,3)\\
\text{unrank}_2(7) &= \text{unrank}_2([\texttt{00}\underline{\texttt{111}}]_2) = (3,0)\\
\text{unrank}_2(8) &= \text{unrank}_2([\texttt{01000}]_2) = (0,4)\\
\text{unrank}_2(9) &= \text{unrank}_2([\texttt{0100}\underline{\texttt{1}}]_2) = (1,2)\\
\text{unrank}_2(10) &= \text{unrank}_2([\texttt{01010}]_2) = (0,5)\\
\text{unrank}_2(11) &= \text{unrank}_2([\texttt{010}\underline{\texttt{11}}]_2) = (2,1)\\
\text{unrank}_2(12) &= \text{unrank}_2([\texttt{01100}]_2) = (0,6)\\
\text{unrank}_2(13) &= \text{unrank}_2([\texttt{0110}\underline{\texttt{1}}]_2) = (1,3)\\
\text{unrank}_2(14) &= \text{unrank}_2([\texttt{01110}]_2) = (0,7)\\
\text{unrank}_2(15) &= \text{unrank}_2([\texttt{0}\underline{\texttt{1111}}]_2) = (4,0)
\end{aligned}$$

El proceso inverso de rank_2 es sencillo bajo esta distribución de naturales, a partir de k_1 y k_2 se calcula su posición $k = 2 \times 2^{k_1} \times k_2 + 2^{k_1} - 1$, esta operación aritmética corresponde a desplazar los bits de k_2 en representación binaria $k_1 + 1$ veces a la izquierda y activar los primeros k_1 bits menos significativos.

Para $j > 2$ se realiza una composición de 2-tuplas en su segunda componente, es decir, k_2 es distribuida en dos naturales al igual que k, hasta obtener la cantidad deseada de componentes. Por ejemplo, al considerar $\text{unrank}_3(k)$ primero se calcula k_1 y k_2 resultantes de $\text{unrank}_2(k)$ y finalmente se calcula k_3 y k_4 resultantes de $\text{unrank}_2(k_2)$, con estos valores se obtiene la 3-tupla (k_1, k_3, k_4); al considerar $\text{rank}_3(k_1, k_3, k_4)$ primero se calcula k_2 resultante de $\text{rank}_2(k_3, k_4)$ y finalmente se calcula k resultante de $\text{rank}_2(k_1, k_2)$.

5. Modelado de enumeraciones en Coq. Las enumeraciones descritas en la anterior sección son modeladas en Coq desde dos perspectivas: como relaciones binarias y como funciones.

Las variables sintácticas en la especificación de IMP se asocian a tipos inductivos donde cada regla de producción corresponde a un constructor. Una enumeración para estos tipos inductivos se modela utilizando una relación binaria de tipo enum_rel la cuál debe satisfacer la propiedad de ser enumerable_type.

`EnumType_Defs.v`

Definition enum_rel (T : Type) := T → nat → Prop**.**

Definition enumerable_type {T : Type} (R : enum_rel T) :=
(**forall** x, **exists!** pos, R x pos) ∧
(**forall** pos, **exists!** x, R x pos)**.**

La enumeración de tuplas se modela como dos funciones computables rank_2 y unrank_2, las cuales son utilizadas como se describe en la anterior sección emulando sus equivalentes para j-tuplas en general.

`Tuples_Defs.v::1-3`

Definition rank_func := nat → nat → nat**.**
Definition unrank_func := nat → (nat × nat)**.**

Estas funciones deben satisfacen las propiedades básicas de las enumeraciones, en particular: ser biyectivas, inversa una de la otra y satisfacer la propiedad débil de orden.

`Tuples_Defs.v::5-30`

Definition rank_injective (f : rank_func) :=
forall a b, **exists** n, f a b = n**.**
Definition rank_surjective (f : rank_func) :=
forall n, **exists** a b, f a b = n**.**
Definition rank_bijective (f : rank_func) :=
(rank_injective f) ∧ (rank_surjective f)**.**

Definition unrank_injective (g : unrank_func) :=
forall n, **exists** a b, g n = (a, b)**.**
Definition unrank_surjective (g : unrank_func) :=
forall a b, **exists** n, g n = (a, b)**.**
Definition unrank_bijective (g : unrank_func) :=
(unrank_injective g) ∧ (unrank_surjective g)**.**

Definition rank_unrank_inv (f : rank_func) (g : unrank_func) :=
forall a b n, f a b = n ⟺ g n = (a, b)**.**

Definition rank_le (f : rank_func) :=
forall a b n, f a b = n ⟹ $a \leq n \wedge b \leq n$**.**
Definition unrank_le (g : unrank_func) :=
forall n a b, g n = (a, b) ⟹ $a \leq n \wedge b \leq n$**.**

Las relaciones para la enumeración de expresiones en IMP se definen en términos de dos funciones arank y aunrank. Por limitantes técnicas de Coq o ignorancia del autor, se declara su existencia como parámetros y la satisfacción de las propiedades correspondientes como axiomas.

Tuples_Props.v::39-52

Parameter (arank : rank_func)
(aunrank : unrank_func)**.**

Axiom arank_injective : (rank_injective arank)**.**
Axiom arank_surjective : (rank_surjective arank)**.**
Axiom arank_bijective : (rank_bijective arank)**.**
Axiom arank_le : (rank_le arank)**.**

Axiom aunrank_injective : (unrank_injective aunrank)**.**
Axiom aunrank_surjective : (unrank_surjective aunrank)**.**
Axiom aunrank_bijective : (unrank_bijective aunrank)**.**
Axiom aunrank_le : (unrank_le aunrank)**.**

Axiom arank_unrank_inv : (rank_unrank_inv arank aunrank)**.**

Este detalle de implementación no debe causar preocupación ya que se demuestra la existencia de dos funciones que pudieran reemplazar el uso de arank y aunrank.

Tuples_Props.v::5-10

Theorem tuple_enumeration : **exists** f g,
rank_bijective f $\wedge$
unrank_bijective g $\wedge$
rank_unrank_inv f g $\wedge$
rank_le f $\wedge$
unrank_le g**.**

6. Implementación de enumeración IMP. La enumeración más fácil de implementar es la de los naturales, se reutiliza el tipo inductivo nat y se define la relación trivial correspondiente.

IMP_Nats_Defs.v::3-4

Inductive enum_nat : (enum_rel nat) :=
| EN_Trivial (n : nat) : enum_nat n n**.**

La estrategia para verificar la propiedad (enumerable_type enum_nat) es separar por casos la conjunción y proponer la variable del cuantificador universal como evidencia para el cuantificador existencial, utilizando EN_Trivial para aseverar que todo natural se asocia a si mismo en la enumeración.

La enumeración de las localidades de memoria $\boldsymbol{X}$ es similar a la de los naturales, sin embargo, las localidades tienen una estructura sintáctica propia, definida en términos de nat. Por la definición de rank_1 y unrank_1, toda posición n se relaciona con la localidad de memoria $(loc\ n)$.

IMP_Locs_Defs.v::3-8

Inductive Loc : Type :=
| X (n : nat)**.**

Inductive enum_loc : Loc → nat → Prop :=
| EL_Bypass (n pos : nat) (H : enum_nat n pos) :
  enum_loc (X n) pos**.**

La estrategia para verificar la propiedad (enumerable_type enum_loc) es similar a la enumeración de naturales, incorporando el constructor X cuando es pertinente.

La enumeración de expresiones aritméticas $\boldsymbol{A}$ difiere de las enumeraciones simples mencionadas anteriormente, en particular, su estructura sintáctica involucra más de un constructor.

IMP_Ariths_Defs.v::6-11

Inductive Arith : Type :=
| AN (n : nat)
| AX (x : Loc)
| APlus ($a1$ $a2$: Arith)
| AMinus ($a1$ $a2$: Arith)
| AMult ($a1$ $a2$: Arith)**.**

Este tipo inductivo sigue el esquema descrito en la especificación de IMP, la única diferencia notable es que los naturales y las localidades de memoria se asocian a un constructor propio de las expresiones aritméticas.

$$\boldsymbol{A} \to \mathbb{N} \mid \boldsymbol{X} \mid (\mathit{plus}\ \boldsymbol{A}\ \boldsymbol{A}) \mid (\mathit{minus}\ \boldsymbol{A}\ \boldsymbol{A}) \mid (\mathit{mult}\ \boldsymbol{A}\ \boldsymbol{A})$$

Como se describe en el algoritmo de enumeración de programas, cada producción en IMP se analiza de forma independiente, por lo que la relación de enumeración de Arith es implementada con una regla de inferencia por producción.

IMP_Ariths_Defs.v::13-41

Inductive enum_arith : (enum_rel Arith) :=
| EA_N `(* ... *)`
| EA_X `(* ... *)`
| EA_Plus `(* ... *)`
| EA_Minus `(* ... *)`
| EA_Mult `(* ... *).`

Los naturales y localidades se asocian a naturales k tales que (k mod 5) es 0 o 1 respectivamente.

| EA_N (n pos : nat)
  (H : enum_nat n pos) :
  enum_arith (AN n) (5 ∗ pos)
| EA_X (x : Loc)
  (pos : nat)
  (H : enum_loc x pos) :
  enum_arith (AX x) (S (5 ∗ pos))

La suma, resta y multiplicación corresponden a los naturales k tales que (k mod 5) es 2, 3 o 4, sin embargo, estas expresiones se conforman de dos subexpresiones, por lo que se utiliza la enumeración de tuplas para calcular la posición correspondiente.

```
| EA_Plus (a1 a2 : Arith)
          (pos1 pos2 : nat)
          (H1 : enum_arith a1 pos1)
          (H2 : enum_arith a2 pos2)
          (pos : nat)
          (H3 : arank pos1 pos2 = pos) :
    enum_arith (APlus a1 a2) (S (S (5 * pos)))
| EA_Minus (a1 a2 : Arith)
           (pos1 pos2 : nat)
           (H1 : enum_arith a1 pos1)
           (H2 : enum_arith a2 pos2)
           (pos : nat)
           (H3 : arank pos1 pos2 = pos) :
    enum_arith (AMinus a1 a2) (S (S (S (5 * pos))))
| EA_Mult (a1 a2 : Arith)
          (pos1 pos2 : nat)
          (H1 : enum_arith a1 pos1)
          (H2 : enum_arith a2 pos2)
          (pos : nat)
          (H3 : arank pos1 pos2 = pos) :
    enum_arith (AMult a1 a2) (S (S (S (S (5 * pos))))).
```

La verificación de (enumerable_type enum_arith), así como el resto de las enumeraciones, utiliza la estrategia de separar el problema en cuatro partes:

a. **La existencia de una posición para todo objeto de tipo Arith** se verifica por inducción estructural sobre el tipo Arith, utilizando las hipótesis de inducción y la propiedad inyectiva de arank para calcular la posición asociada por el algoritmo.

b. **La existencia de un objeto de tipo Arith para toda posición** se verifica por inducción fuerte sobre la posición y analizando los posibles residuos de (k mod 5) para determinar la producción correspondiente. Las subexpresiones utilizadas en la construcción de la expresión provienen de las componentes de aunrank aplicadas a la hipótesis de inducción utilizando la propiedad débil de orden para las 2-tuplas.

c. **La unicidad del objeto asociado a las posiciones** se verifica por inducción sobre las reglas de inferencia de la relación de enumeración. Ya que existe una regla de inferencia por cada producción, una producción por cada constructor de Arith y un valor de (k mod 5) asociado a cada constructor, se descartan todas menos una posibilidad de posición, cuya unicidad se verifica con las hipótesis de inducción.

d. **La unicidad de la posición asociada a los objetos** se verifica de forma similar a **c**, utilizando adicionalmente la unicidad en la enumeración de tuplas.

Las expresiones booleanas se definen de forma similar a las expresiones aritméticas, incorporando la estrategia de utilizar constructores de cero argumentos para los valores constantes *true* y *false*. Estas constantes tienen el efecto de *desplazar* la enumeración de

los objetos producidos por las producciones en R dos posiciones correspondientes al valor de $n = 2$ en el algoritmo de enumeración.

IMP_Bools_defs::5-48

```
Inductive Bool : Type :=
| BT
| BF
| BEq (a1 a2 : Arith)
| BLt (a1 a2 : Arith)
| BNot (b : Bool)
| BOr (b1 b2 : Bool)
| BAnd (b1 b2 : Bool).

Inductive enum_bool : (enum_rel Bool) :=
| EB_T : enum_bool BT 0
| EB_F : enum_bool BF 1
| EB_Eq (* ... *)
| EB_Lt (* ... *)
| EB_Not (* ... *)
| EB_Or (* ... *)
| EB_And (* ... *).
```

La enumeración de comparaciones booleanas utiliza la relación enum_arith sobre las subexpresiones y de igual forma que con las reglas de inferencia para expresiones aritméticas, se utiliza arank para asociar la posición correspondiente.

```
| EB_Eq (a1 a2 : Arith)
        (pos1 pos2 : nat)
        (H1 : enum_arith a1 pos1)
        (H2 : enum_arith a2 pos2)
        (pos : nat)
        (H3 : arank pos1 pos2 = pos) :
    enum_bool (BEq a1 a2) (S (S (5 * pos)))
| EB_Lt (a1 a2 : Arith)
        (pos1 pos2 : nat)
        (H1 : enum_arith a1 pos1)
        (H2 : enum_arith a2 pos2)
        (pos : nat)
        (H3 : arank pos1 pos2 = pos) :
    enum_bool (BLt a1 a2) (S (S (S (5 * pos))))
```

Las negaciones se implementan de forma similar a las localidades de memoria, ya que se construyen a partir de una sola subexpresión.

```
| EB_Not (b : Bool)
         (pos : nat)
         (H : enum_bool b pos) :
    enum_bool (BNot b) (S (S (S (S (5 * pos)))))
```

Las operaciones booleanas binarias se implementan de forma similar a la suma, resta y multiplicación.

```
| EB_Or (b1 b2 : Bool)
        (pos1 pos2 : nat)
        (H1 : enum_bool b1 pos1)
        (H2 : enum_bool b2 pos2)
        (pos : nat)
        (H3 : arank pos1 pos2 = pos) :
    enum_bool (BOr b1 b2) (S (S (S (S (S (5 * pos))))))
| EB_And (b1 b2 : Bool)
         (pos1 pos2 : nat)
         (H1 : enum_bool b1 pos1)
         (H2 : enum_bool b2 pos2)
         (pos : nat)
         (H3 : arank pos1 pos2 = pos) :
    enum_bool (BAnd b1 b2) (S (S (S (S (S (S (5 * pos)))))))).
```

La verificación de (enumerable_type enum_bool) sigue la estrategia descrita para expresiones aritméticas, incorporando los casos triviales para los valores de verdad quienes tienen una única asignación de posiciones.

Los programas se definen de forma similar a las expresiones booleanas, la constante *skip* desplaza la enumeración en una posición y las producciones en R distribuyen la posición dependiendo del valor de (k mod 4). La diferencia principal entre los programas y el resto de las expresiones es que las condicionales *if* se conforman de tres subexpresiones, ya que arank y aunrank codifican la enumeración de 2-tuplas, se incorpora el algoritmo para 3-tuplas dentro de la regla de inferencia EP_If.

IMP_Progs_Defs.v::7-45

```
Inductive Prog : Type :=
| PSkip
| PSet (x : Loc) (a : Arith)
| PConc (p1 p2 : Prog)
| PWhile (b : Bool) (p : Prog)
| PIf (b : Bool) (p1 p2 : Prog).

Inductive enum_prog : (enum_rel Prog) :=
| EP_Skip : enum_prog PSkip 0
| EP_Set (* ... *)
| EP_Conc (* ... *)
| EP_While (* ... *)
| EP_If (* ... *).
```

La asignación de memoria, la concatenación de programas y la iteración se implementan de forma similar al las operaciones aritméticas.

```
| EP_Set (x : Loc) (a : Arith)
        (pos1 pos2 : nat)
        (H1 : enum_loc x pos1)
        (H2 : enum_arith a pos2)
        (pos : nat)
        (H3 : arank pos1 pos2 = pos) :
    enum_prog (PSet x a) (S (4 * pos))
| EP_Conc (p1 p2 : Prog)
          (pos1 pos2 : nat)
          (H1 : enum_prog p1 pos1)
          (H2 : enum_prog p2 pos2)
          (pos : nat)
          (H3 : arank pos1 pos2 = pos) :
    enum_prog (PConc p1 p2) (S (S (4 * pos)))
| EP_While (b : Bool) (p : Prog)
           (pos1 pos2 : nat)
           (H1 : enum_bool b pos1)
           (H2 : enum_prog p pos2)
           (pos : nat)
           (H3 : arank pos1 pos2 = pos) :
    enum_prog (PWhile b p) (S (S (S (4 * pos))))
```

Para las condicionales, primero se obtiene el rank de las posiciones asociadas al consecuente y el alternante, y esta es a su vez combinada con el rank de la expresión booleana para obtener la enumeración de 3-tuplas que asocia la posición correspondiente al *if*.

```
| EP_If (b : Bool) (p1 p2 : Prog)
       (pos1 pos2 pos3 : nat)
       (H1 : enum_bool b pos1)
       (H2 : enum_prog p1 pos2)
       (H3 : enum_prog p2 pos3)
       (pos pos' : nat)
       (H4 : arank pos2 pos3 = pos')
       (H5 : arank pos1 pos' = pos) :
    enum_prog (PIf b p1 p2) (S (S (S (S (4 * pos))))).
```

La verificación de (enumerable_type enum_prog)sigue la misma estrategia descrita anteriormente.

7. Existencia de una enumeración de tuplas. El teorema tuple_enumeration plantea la existencia de dos funciones rank y unrank que satisfacen las propiedades necesarias para que la enumeración de programas funcione correctamente. Para demostrar este teorema se implementan las funciones rank_tuple y unrank_tuple basadas en el algoritmo propuesto en la sección 4.

`Ranking_Defs.v::4-6`

```
Definition rank_tuple (a b : nat) : nat := bnat_to_nat (mixed_rank a b).
Definition unrank_tuple (n : nat) : nat × nat := mixed_unrank (nat_to_bnat n).
```

Debido al manejo combinado de representaciones unarias y binarias de números naturales en el algoritmo, se define esta enumeración en función a una enumeración que difiere de las firmas rank_func y unrank_func en el sentido que las posiciones de las tuplas se codifican en binario utilizando numeros naturales por paridad y las componentes de las tuplas se codifican en unario utilizando el tipo inductivo nat.

Los naturales por paridad se implementan en el tipo de datos bnat, el cual define tres constructores: Z que representa al cero, (U k) que representa el número impar $2k+1$ y (D k) que representa el número par $2k+2$.

NatBNat_Defs::1-4

```
Inductive bnat : Type :=
| Z
| U (bn : bnat)
| D (bn : bnat).
```

Las funciones bnat_to_nat y nat_to_bnat permiten transformar naturales de una representación a la otra y son utilizados para que la implementación del algoritmo sea una enumeración de tuplas adecuada para la enumeración de programas.

MixedRanking_Defs.v::3-19

```
Fixpoint mixed_rank (a b : nat) : bnat :=
 match a with
 | O ⇒ match b with
        | O ⇒ Z
        | S b' ⇒ D (nat_to_bnat b')
        end
 | S a' ⇒ U (mixed_rank a' b)
 end.

Fixpoint mixed_unrank (bn : bnat) : nat × nat :=
 match bn with
 | Z ⇒ (O, O)
 | U x ⇒ match (mixed_unrank x) with
           | (a, b) ⇒ (S a, b)
           end
 | D x ⇒ (O, S (bnat_to_nat x))
 end.
```

Utilizando esta enumeración de tuplas con codificaciones mixtas para naturales, la verificación de las propiedades de tuple_enumeration es bastante directa, recurriendo únicamente a lemas de equivalencia entre ambas codificaciones e inducción sobre los tipos de datos nat y bnat.

Bibliografía

[1] Wikipedia, "Enumeration", *Wikipedia, la enciclopedia libre* (consulta Febrero 2021) `https://en.wikipedia.org/wiki/Enumeration`.

[2] Sanjeev Arora y Boaz Barak, "Computational Complexity: A Modern Approach", *Cambridge University Press* (2009).

[3] Barry S. Cooper, "Computability Theory", *CRC Press* (2004).

[4] Michael Sipser, "Introduction to the Theory of Computation", *Cengage Learning* (2013).

[5] Donald E. Knuth, "The Art of Computer Programming, Combinatorial Algorithms Volume 4A Part I", *Addison-Wesley* (2011).

[6] Donald L. Kreher y Douglas R. Stinson, "Combinatorial Algorithms: Generation, Enumeration, and Search", *CRC Press* (1998).

[7] Frank Ruskey, "Combinatorial Generation", *Preliminary working draft* (2003).