Universidad Nacional Autónoma de México

Posgrado en Ciencia e Ingeniería de la Computación

Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas

Teoría de la Computación

Enumeración de programas

T E S I S

QUE PARA OPTAR POR EL GRADO DE:

M A E S T R O E N C I E N C I A S D E L A C O M P U T A C I Ó N

P R E S E N T A:

E D U A R D O A C U Ñ A Y E O M A N S

Directores de tesis:

Dr. Francisco Hernández Quiroz

Facultad de Ciencias — UNAM

Dr. Hector Zenil Chávez

Oxford Immune Algorithmics

Ciudad Universitaria, Cd. Mx., Enero 2022

Enumeración de programas

por

Eduardo Acuña Yeomans

Tesis presentada para optar por el grado de

Maestro en Ciencias de la Computación

en el

Posgrado de Ciencias e Ingeniería de la Computación

Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas

Universidad Nacional Autónoma de México

Ciudad Universitaria, Cd. Mx., Enero 2022

Resumen

El trabajo descrito en esta tesis se centra en algoritmos que enumeran programas de lenguajes de programación. Un lenguaje imperativo y simple llamado IMP es utilizado como caso de estudio en el desarrollo de los algoritmos de enumeración, los cuales son posteriormente generalizados a partir de una colección de enumeraciones primitivas y operadores que combinan enumeraciones simples para producir otras mas complejas. Se hace hincapié en las propiedades cuantitativas y cualitativas de las enumeraciones así como en su utilidad en la generación sistemática de programas.

Palabras clave: Enumeración, Lenguajes de programación, Combinadores de enumeraciones.

Agradecimientos

Este trabajo pudo ser elaborado gracias al apoyo y colaboración de diversas personas e instituciones. En particular agradezco:

A los colaboradores del proyecto del cuál este trabajo forma parte, Luis Benítez Lluis, Victor Zamora Gutiérrez, Marco Vladimir Lemus Yáñez, Francisco Hernández Quiroz y Hector Zenil. A la Universidad Nacional Autónoma de México (UNAM), el Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas (IIMAS) y al Posgrado en Ciencia e Ingeniería en Computación (PCIC). Al Consejo Nacional de Ciencia y Tecnología (CONACYT), por el apoyo económico brindado en el transcurso de la maestría y para la realización de este trabajo. A mis compañeros y profesores del posgrado. A mis progenitores y hermanos.

Índice general

Introducción

Este trabajo tiene como objetivo principal el desarrollo de algoritmos de enumeración para la generación sistemática de programas. Se busca que estos algoritmos sean lo suficientemente flexibles para explorar el espacio de programas desde varias perspectivas y no limitarse a listar objetos en un orden particular.

Es de especial interés la enumeración de programas de un lenguaje de programación simple e imperativo llamado IMP. Se pretende utilizar su enumeración para generar una gran cantidad de programas a ser posteriormente analizados y ejecutados con la finalidad de aproximar la distribución universal como parte del proyecto de doctorado de Lemus Yáñez [15].

Al ser este un proyecto en curso, es deseable que las técnicas de enumeración puedan ser adaptadas a otros lenguajes y a otras aplicaciones. En este contexto se contemplan variaciones del lenguaje IMP y métodos alternativos de generación de programas como el muestreo aleatorio.

Este trabajo se conforma de cuatro capítulos y de un sistema que implementa todos los algoritmos descritos en este documento.

En el capítulo 1 se describe la motivación, el contexto histórico y la organización del proyecto para la aproximación de la distribución universal, del cual este trabajo forma parte. Se describe formalmente el lenguaje IMP y se describe el rol que juegan las enumeraciones en la generación de programas.

En el capítulo 2 se detalla un procedimiento para derivar enumeraciones del programas IMP basado en una especificación sintáctica del lenguaje. Posteriormente se proponen problemas y soluciones relacionados con la enumeración al incorporar una métrica de tamaño para las expresiones del lenguaje. Finalmente se presentan maneras en las que se puede analizar IMP utilizando los algoritmos de enumeración descritos previamente.

En el capítulo 3 se deja a un lado el lenguaje IMP para abordar el problema de enumerar estructuras sintácticas en general. Se conceptualiza las enumeraciones con base en operaciones que deben satisfacer tres propiedades para inducir una

biyección entre naturales y un conjunto de objetos. Se definen enumeraciones particulares, de distintos niveles de complejidad, que cumplen las restricciones del modelo propuesto. Posteriormente se definen mecanismos que permiten combinar enumeraciones simples para producir otras mas complejas. Esto resulta en una familia de operaciones que permiten describir la enumeración de programas IMP de forma declarativa.

En el capítulo 4 se presentan tres aplicaciones de los métodos desarrollados en los anteriores capítulos. Se detalla como se pueden utilizar para generar y almacenar programas IMP, luego como se puede modificar la enumeración del lenguaje para alterar el orden de los programas y explorar un subconjunto de las enumeraciones. Finalmente se presenta la exploración realizada por el grupo de trabajo para determinar la forma mas conveniente de definir parte de la enumeración de IMP del capítulo 2.

La implementación del sistema de enumeración es una colección de programas literarios que contemplan:

- La verificación formal de la biyección descrita en el capítulo 2 utilizando el asistente de pruebas Coq.
- La implementación en el lenguaje de programación Common Lisp de la enumeración IMP, así como el conteo y selección aleatoria de programas por tamaño.
- La implementación en el lenguaje de programación Common Lisp de las operaciones y enumeraciones descritas en el capítulo 3, incluyendo un mecanismo para la compilación dinámica de enumeraciones.
- Ejemplos del uso del sistema de enumeración, incluyendo algunas variantes de IMP propuestas en el capítulo 4.

Capítulo 1

Antecedentes

1.1 Distribución universal

En 1997 se publica en la revista de divulgación *The Mathematical Intelligencer* un artículo de Kirchherr, Li y Vitányi [11] donde plantean que el razonamiento inductivo contempla hacer predicciones sobre el comportamiento futuro basado en observaciones previas y cómo este razonamiento es una herramienta fundamental en el planteamiento de hipótesis en el quehacer científico.

El problema central que describen es cómo determinar métodos apropiados para formular estas hipótesis, y posteriormente, cómo determinar qué hipótesis es adecuada cuando las observaciones tienen mas de una explicación.

A lo largo del artículo presentan una perspectiva algorítmica particular, transversal a conceptos de probabilidad, teoría de la información y computabilidad, introduciendo a los lectores lo que ellos llaman *la milagrosa distribución universal*.

En la década de los sesenta, Ray Solomonoff sentó las bases del área de *probabilidad algorítmica* [22, 23], donde se estudia la distribución conocida como distribución universal. Esta distribución describe la probabilidad de que una cadena de símbolos x sea producida por algún programa computacional.

Sea U una máquina universal, p un programa y $|p|$ el tamaño del programa, se define la distribución universal D_U como

$$D_U(x) = \sum_{U(p)=x} 2^{-|p|}$$

De forma independiente, Andréi Kolmogorov desarrolló la llamada *complejidad algorítmica* al estudiar la naturaleza de las secuencias aleatorias [13]. La

complejidad algorítmica de una cadena de símbolos x es la longitud de un programa más corto, que al ser ejecutado produce x. De acuerdo con este enfoque, entre mayor complejidad tenga una cadena, más aleatoria es y entre menor complejidad, mayor su regularidad.

Considerando la notación utilizada en la definición de la distribución universal, se define la complejidad de Kolmogorov K_U como

$$K_U(x) = \min\{|p| : U(p) = x\}$$

A pesar de que el énfasis de Solomonoff fuera la inferencia inductiva y el de Kolmogorov fueran las propiedades matemáticas de la complejidad, ambas nociones se relacionan en el sentido que los programas que influyen en mayor medida la probabilidad algorítmica son los programas que tienen menor complejidad algorítmica, tal como lo explica Solomonoff en la conferencia impartida en 2003 al ser galardonado con la medalla Kolmogorov [24].

En el contexto de razonamiento inductivo, una observación es una cadena de símbolos, mientras que las predicciones de qué hipótesis explica una observación son programas que generan dichas cadenas. La probabilidad algorítmica permite estudiar qué tan probable es que esta cadena haya sido generada por un mecanismo computacional, mientras que la complejidad algorítmica permite estudiar qué tan aleatoria es la cadena. La suposición implícita en ambos planteamientos es que los procesos detrás las observaciones realizadas pueden ser descritos con mecanismos computacionales.

Tanto la probabilidad como la complejidad algorítmica son incomputables, es decir, no existe un método efectivo para calcular estas medidas. Sin embargo, se han realizado esfuerzos para aproximarlas, en particular utilizando algoritmos de compresión. En una reciente publicación de Zenil [29] se presenta un panorama de los distintos métodos, enfoques y esfuerzos dedicados a esta tarea.

El presente trabajo de tesis, así como los trabajos de Luis Benítez Lluis [2] y Victor Zamora Gutiérrez [28] forman parte del proyecto de investigación de Vladimir Lemus Yáñez [15] bajo la asesoría de Francisco Hernández Quiroz y Héctor Zenil. El proyecto busca aproximar la distribución universal utilizando técnicas alternativas a la compresión, como el *Coding Theorem Method* [7, 21] y el *Block Decomposition Method* [30].

En contraste con esfuerzos similares que utilizan modelos computacionales basados en máquinas de Turing [7] o sistemas dinámicos discretos [31] como autómatas celulares o redes complejas, este proyecto propone el uso de un lenguaje de programación imperativo de alto nivel con una sintaxis y semántica simple.

Como primera aproximación a solucionar el problema en cuestión, se propone generar programas de forma sistemática y ejecutarlos para determinar su salida, de tal manera que al medir el tamaño de los programas generados se puede almacenar su aporte a la distribución universal.

El trabajo descrito en esta tesis y la de Benítez Lluis [2] aborda la primer parte de esta propuesta, conceptualizando la generación de programas como un problema de enumeración. La segunda parte de la propuesta se aborda en el trabajo de tesis de Zamora Gutiérrez [28], donde se describe la implementación de un intérprete de programas diseñado para suspender su ejecución en caso que se determine probable que el programa no se vaya a detener.

1.2 Lenguaje de programación IMP

El lenguaje de alto nivel utilizado en el proyecto de investigación es llamado IMP. Los programas de este lenguaje se construyen a partir de expresiones y estructuras presentes en una gran cantidad de lenguajes imperativos: condicionales, iteraciones, asignaciones y concatenación[1] de programas.

Hay una amplia tradición de capturar la esencia de programas de alto nivel con estas mismas unidades básicas. Desde la década de los sesenta se han utilizado estos lenguajes para estudiar formalmente propiedades de programas [8] y para estudiar conceptos fundamentales de diseño de lenguajes de programación [26], en particular para la especificación de la semántica [18]. Recientemente se han utilizado lenguajes como IMP como herramienta didáctica para el estudio de técnicas de verificación formal y asistentes de prueba [16, 17].

El lenguaje IMP se describe formalmente utilizando una gramática libre de contexto para su sintaxis y reglas de semántica operacional estructurada para su semántica. Las referencias [9] y [18] pueden ser útiles para familiarizarse con estos formalismos. El apéndice A contiene una descripción completa y condensada de la definición de IMP presente en esta sección.

Las categorías sintácticas principales del lenguaje son los números naturales **N**, las localidades de memoria **X**, las expresiones aritméticas **A**, las expresiones booleanas **B** y los programas **P**.

Para toda categoría sintáctica **C**, se denota $\alpha \in \mathbf{C}$ si la cadena de símbolos α puede ser derivada a partir de **C**. En particular, se dice que una cadena de símbolos

[1] En este trabajo nos referimos por *concatenación* de programas a la estructura sintáctica que representa la ejecución secuencial de dos programas, por lo que este tipo de programas también se les puede llamar *secuencias*.

α es un programa sintácticamente válido de IMP cuando $\alpha \in \mathbf{P}$.

La ejecución de programas IMP ocurre en el contexto de un estado de memoria denotado σ. La memoria es modelada como una función que asocia localidades a números naturales. La semántica del lenguaje es definida en términos de relaciones de transición de la forma

$$\langle \alpha, \sigma \rangle \rightarrow x$$

donde α es una cadena de símbolos, σ un estado de memoria y x el resultado de evaluar operacionalmente la expresión α en el estado σ.

Los números naturales son uno de los tipos de datos fundamentales de IMP ya que son almacenados en la memoria, son utilizados para referirse a localidades de memoria y forman parte de los cálculos realizados mediante expresiones aritméticas. La categoría sintáctica de números naturales **N** tiene asociadas las producciones (1.1).

$$\begin{aligned} \mathbf{N} &\rightarrow \mathbf{D_0} \mid \mathbf{D_1}\mathbf{D_0}^+ \\ \mathbf{D_0} &\rightarrow 0 \mid \mathbf{D_1} \\ \mathbf{D_1} &\rightarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{aligned} \tag{1.1}$$

De acuerdo a estas producciones, la estructura sintáctica de naturales **N** consiste de cadenas de dígitos del cero al nueve tal que las cadenas con más de un dígito no comienzan con 0.

Sea $n \in \mathbf{N}$, se denota como $\#\mathsf{n}$ al valor numérico de n del conjunto de naturales $\mathbb{N}$. La evaluación operacional de la cadena n se especifica con la relación de transición (1.2).

$$\langle \mathsf{n}, \sigma \rangle \rightarrow_{\mathbf{N}} \#\mathsf{n} \tag{1.2}$$

Las localidades de memoria son expresiones que nos permiten referirnos a un valor almacenado en un estado de memoria. Sintácticamente se describen con la producción (1.3).

$$\mathbf{X} \rightarrow \mathsf{x}_{\mathbf{N}} \tag{1.3}$$

Se denota $\sigma(\mathsf{n}_\mathsf{n})$ al natural en $\mathbb{N}$ almacenado en el estado de memoria σ para la localidad x_n. La evaluación operacional de x_n modela la lectura de su valor almacenado en memoria y se especifica con la relación de transición (1.4).

$$\langle \mathsf{x}_\mathsf{n}, \sigma \rangle \rightarrow_{\mathbf{X}} \sigma(\mathsf{x}_\mathsf{n}) \tag{1.4}$$

El cálculo de números naturales se realiza a partir de expresiones aritméticas **A**, las cuáles se construyen a partir de naturales **N**, localidades **X**, o bien operaciones

de suma, resta y multiplicación de otras expresiones aritméticas. La estructura sintáctica de estas expresiones se describe con las producciones (1.5).

$$\mathbf{A} \to \mathbf{N} \mid \mathbf{X} \mid (\mathbf{A} + \mathbf{A}) \mid (\mathbf{A} - \mathbf{A}) \mid (\mathbf{A} \times \mathbf{A}) \tag{1.5}$$

El resultado de evaluar operacionalmente una expresión aritmética es un número natural en $\mathbb{N}$. Cuando la cadena a ser evaluada es derivada de **N** o **X** se aplican las reglas de inferencia (1.6).

$$\frac{\langle \mathsf{n}, \sigma \rangle \to_{\mathbf{N}} n}{\langle \mathsf{n}, \sigma \rangle \to_{\mathbf{A}} n} \qquad \frac{\langle \mathsf{x}_i, \sigma \rangle \to_{\mathbf{X}} n}{\langle \mathsf{x}_i, \sigma \rangle \to_{\mathbf{A}} n} \tag{1.6}$$

Para las operaciones de suma, resta y multiplicación, se denota como $\oplus$ al símbolo terminal correspondiente a la operación y $\oplus_{\mathbb{N}}$ al operador binario correspondiente para la operación con naturales $\mathbb{N}$. Las tres evaluaciones se describen con la regla de inferencia (1.7).

$$\frac{\langle a_1, \sigma \rangle \to_{\mathbf{A}} n_1 \quad \langle a_2, \sigma \rangle \to_{\mathbf{A}} n_2 \quad n_1 \oplus_{\mathbb{N}} n_2 = n}{\langle (a_1 \oplus a_2), \sigma \rangle \to_{\mathbf{A}} n} \tag{1.7}$$

La suma y multiplicación de naturales se interpretan de forma usual, pero en el caso de la resta de dos cadenas n_1 y n_2 de **N**, cuando $\#\mathsf{n}_2 > \#\mathsf{n}_1$ se considera que $\#\mathsf{n}_1 -_{\mathbb{N}} \#\mathsf{n}_2 = 0$.

A partir de valores de verdad true (verdadero) y false (falso), el lenguaje IMP permite construir expresiones booleanas **B** más complejas utilizando las operaciones de conjunción, disyunción y negación, así como comparaciones entre expresiones aritméticas.

Ya que las expresiones aritméticas se pueden conformar por valores almacenados en memoria, el resultado de una expresión booleana también puede depender del valor asociado a las localidades de memoria.

La estructura sintáctica de las expresiones booleanas **B** se describe con las producciones (1.8).

$$\mathbf{B} \to \mathsf{true} \mid \mathsf{false} \mid (\mathbf{A} = \mathbf{A}) \mid (\mathbf{A} < \mathbf{A}) \mid \neg\, \mathbf{B} \mid (\mathbf{B} \vee \mathbf{B}) \mid (\mathbf{B} \wedge \mathbf{B}) \tag{1.8}$$

El resultado de evaluar operacionalmente una expresión booleana es un valor de verdad true o false. Estas dos expresiones son evaluadas a si mismas como se

describe en las reglas de inferencia (1.9).

$$\langle \mathsf{true}, \sigma \rangle \rightarrow_{\mathbf{B}} \mathsf{true} \qquad \langle \mathsf{false}, \sigma \rangle \rightarrow_{\mathbf{B}} \mathsf{false} \tag{1.9}$$

Dos expresiones aritméticas pueden ser comparadas para calcular si corresponden al mismo valor numérico con la operación de *igual que* de acuerdo a las reglas de inferencia (1.10).

$$\begin{gathered} \frac{\langle a_1, \sigma \rangle \rightarrow_{\mathbf{A}} n_1 \quad \langle a_2, \sigma \rangle \rightarrow_{\mathbf{A}} n_2 \quad n_1 = n_2}{\langle (a_1 = a_2), \sigma \rangle \rightarrow_{\mathbf{B}} \mathsf{true}} \\ \frac{\langle a_1, \sigma \rangle \rightarrow_{\mathbf{A}} n_1 \quad \langle a_2, \sigma \rangle \rightarrow_{\mathbf{A}} n_2 \quad n_1 \neq n_2}{\langle (a_1 = a_2), \sigma \rangle \rightarrow_{\mathbf{B}} \mathsf{false}} \end{gathered} \tag{1.10}$$

De forma similar, se calcula si una expresión aritmética es numéricamente menor a otra con la operación de *menor que* de acuerdo a las reglas de inferencia (1.11).

$$\begin{gathered} \frac{\langle a_1, \sigma \rangle \rightarrow_{\mathbf{A}} n_1 \quad \langle a_2, \sigma \rangle \rightarrow_{\mathbf{A}} n_2 \quad n_1 < n_2}{\langle (a_1 < a_2), \sigma \rangle \rightarrow_{\mathbf{B}} \mathsf{true}} \\ \frac{\langle a_1, \sigma \rangle \rightarrow_{\mathbf{A}} n_1 \quad \langle a_2, \sigma \rangle \rightarrow_{\mathbf{A}} n_2 \quad n_1 \geq n_2}{\langle (a_1 < a_2), \sigma \rangle \rightarrow_{\mathbf{B}} \mathsf{false}} \end{gathered} \tag{1.11}$$

La negación de una expresión booleana es operacionalmente evaluada al valor de verdad opuesto de acuerdo a las reglas de inferencia (1.12).

$$\frac{\langle b, \sigma \rangle \rightarrow_{\mathbf{B}} \mathsf{false}}{\langle \neg\, b, \sigma \rangle \rightarrow_{\mathbf{B}} \mathsf{true}} \qquad \frac{\langle b, \sigma \rangle \rightarrow_{\mathbf{B}} \mathsf{true}}{\langle \neg\, b, \sigma \rangle \rightarrow_{\mathbf{B}} \mathsf{false}} \tag{1.12}$$

Para las conjunciones y disyunciones se describe su evaluación operacional con la regla de inferencia (1.13) donde $\oplus$ se refiere al símbolo terminal de la operación y $\oplus_{\mathbb{B}}$ se refiere a la operación correspondiente descrita en la tabla 1.1.

$$\frac{\langle b_1, \sigma \rangle \rightarrow_{\mathbf{B}} v_1 \quad \langle b_2, \sigma \rangle \rightarrow_{\mathbf{B}} v_2 \quad v_1 \oplus_{\mathbb{B}} v_2 = v}{\langle (b_1 \oplus b_2), \sigma \rangle \rightarrow_{\mathbf{B}} v} \tag{1.13}$$

Los programas del lenguaje IMP son expresiones que permiten modificar el estado de la memoria y controlar el flujo de ejecución. La estructura sintáctica de un programa **P** se describe con las producciones (1.14).

$$\mathbf{P} \rightarrow \mathsf{skip} \mid \mathbf{X} := \mathbf{A} \mid (\mathsf{while}\ \mathbf{B}\ \mathsf{do}\ \mathbf{P}) \mid (\mathbf{P}\ ;\ \mathbf{P}) \mid (\mathsf{if}\ \mathbf{B}\ \mathsf{then}\ \mathbf{P}\ \mathsf{then}\ \mathbf{P}) \tag{1.14}$$

v_1	v_2	$v_1 \vee v_2$	$v_1 \wedge v_2$
false	false	false	false
false	true	true	false
true	false	true	false
true	true	true	true

Tabla 1.1: Operaciones booleanas binarias.

El resultado de evaluar operacionalmente un programa es un estado de memoria, es decir, los programas IMP transforman un estado de memoria a otro.

El programa sintácticamente más simple es skip, su ejecución no realiza cambios en el estado de la memoria. Su evaluación operacional se describe con la relación de transición (1.15).

$$\langle \mathsf{skip}, \sigma \rangle \rightarrow_{\mathbf{P}} \sigma \tag{1.15}$$

Las asignaciones de memoria consisten en cambiar un natural almacenado en memoria por otro. Ya que los estados de memoria son modelados como funciones, se denota como $\sigma[n/\mathsf{x}_i]$ al estado de memoria que asocia x_i al natural n y cualquier otra localidad al natural asociado en σ. Formalmente se define con la ecuación (1.16).

$$\sigma[n/\mathsf{x}_i](\mathsf{x}_j) = \begin{cases} n & i = j \\ \sigma(\mathsf{x}_j) & i \neq j \end{cases} \tag{1.16}$$

La evaluación operacional de las asignaciones de memoria se describen con la regla de inferencia (1.17).

$$\frac{\langle a, \sigma \rangle \rightarrow_{\mathbf{A}} n}{\langle \mathsf{x}_i := a, \sigma \rangle \rightarrow_{\mathbf{P}} \sigma[n/\mathsf{x}_i]} \tag{1.17}$$

Para efectuar una secuencia de transformaciones al estado de memoria, el lenguaje IMP contiene expresiones para concatenar programas cuya evaluación operacional consiste en la ejecución secuencial de un programa después de otro con la regla de inferencia (1.18).

$$\frac{\langle p_1, \sigma \rangle \rightarrow_{\mathbf{P}} \sigma_1 \quad \langle p_2, \sigma_1 \rangle \rightarrow_{\mathbf{P}} \sigma_2}{\langle (p_1 \,;\, p_2), \sigma \rangle \rightarrow_{\mathbf{P}} \sigma_2} \tag{1.18}$$

Al igual que otros lenguajes de programación imperativos, IMP utiliza expresiones while para describir la repetida ejecución de un programa mientras una expresión booleana sea evaluada a true. Para definir la evaluación operacional de estas expresiones se considera primero la regla de inferencia (1.19) donde la expresión booleana asociada tiene valor false.

$$\frac{\langle b, \sigma\rangle \rightarrow_{\mathbf{B}} \mathsf{false}}{\langle(\mathsf{while}\ b\ \mathsf{do}\ p), \sigma\rangle \rightarrow_{\mathbf{P}} \sigma} \tag{1.19}$$

Cuando la expresión booleana es evaluada a true, se evalúa el cuerpo de la iteración para obtener un nuevo estado de memoria σ_1. Posteriormente se evalúa la expresión while con σ_1 para obtener el estado de memoria final σ_2, como se muestra en la regla de inferencia (1.20).

$$\frac{\langle b, \sigma\rangle \rightarrow_{\mathbf{B}} \mathsf{true} \quad \langle p, \sigma\rangle \rightarrow_{\mathbf{P}} \sigma_1 \quad \langle(\mathsf{while}\ b\ \mathsf{do}\ p), \sigma_1\rangle \rightarrow_{\mathbf{P}} \sigma_2}{\langle(\mathsf{while}\ b\ \mathsf{do}\ p), \sigma\rangle \rightarrow_{\mathbf{P}} \sigma_2} \tag{1.20}$$

El último tipo de programa corresponde a las estructuras de control condicionales if compuestas de una expresión booleana y dos programas, el primero a ser ejecutado cuando la expresión booleana es evaluada a true y el segundo cuando es evaluada a false como se muestra en las reglas de inferencia (1.21).

$$\frac{\langle b, \sigma\rangle \rightarrow_{\mathbf{B}} \mathsf{true} \quad \langle p_1, \sigma\rangle \rightarrow_{\mathbf{P}} \sigma'}{\langle(\mathsf{if}\ b\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2), \sigma\rangle \rightarrow_{\mathbf{P}} \sigma'} \qquad \frac{\langle b, \sigma\rangle \rightarrow_{\mathbf{B}} \mathsf{false} \quad \langle p_2, \sigma\rangle \rightarrow_{\mathbf{P}} \sigma'}{\langle(\mathsf{if}\ b\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2), \sigma\rangle \rightarrow_{\mathbf{P}} \sigma'} \tag{1.21}$$

Para determinar cuál es la *salida* de un programa se considera una función de interpretación de memoria f la cual a partir de un estado cualquiera σ calcula una cadena binaria α. También se considera un estado de memoria vacío σ_0 el cual asocia a todas las localidades con el natural cero. Formalmente, para cualquier programa p, si $\langle p, \sigma_0\rangle \rightarrow_{\mathbf{P}} \sigma$ y $f(\sigma) = \alpha$, entonces α es la salida del programa p.

El trabajo de tesis [28] presenta varias alternativas para la función de interpretación de memoria.

1.3 Enumeraciones

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.

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 métodos que la producen?.

En las áreas de computabilidad y complejidad computacional [1, 5, 20], 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 cual se recorre sobre todos los elementos de un conjunto de forma ordenada hasta encontrar uno que satisface alguna propiedad deseada.

En este trabajo se exploran las enumeraciones desde una perspectiva similar al área de combinatoria. Donald Knuth menciona en [12] 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 esta terminología argumentando que enumerar es usualmente asociada al acto de *contar* las posibilidades y que listar implica imprimir o almacenar todas las posibilidades.

Kreher y Stinson plantean una distinción similar en [14] 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.

En el trabajo preliminar titulado "Combinatorial Generation" [19], Frank Ruskey plantea la exploración de cuatro diferentes problemas de enumeración:

1. Generar objetos a partir de un primer elemento y un operador sucesor.
2. Seleccionar de forma aleatoria un elemento de un conjunto de posibilidades.
3. Calcular la posición ocupada por un objeto en un ordenamiento.
4. Calcular el objeto que ocupa una posición en un ordenamiento.

Los últimos dos problemas corresponden a procesos inversos conocidos como *ranking* y *unranking* respectivamente. El *rank* (o rango) de un objeto es la posición que ocupa, al contar las posiciones a partir del cero, se puede interpretar también como la cantidad de objetos que lo preceden en el ordenamiento. En este trabajo se utilizan operaciones rank y unrank como los mecanismos principales para trabajar con enumeraciones.

El enfoque de los algoritmos descritos en [12, 14, 19] consiste en partir de un universo finito de objetos combinatorios elementales como permutaciones, combinaciones, particiones, tuplas o árboles, para luego describir algoritmos que producen distintos ordenamientos, analizar sus diferencias y presentar las ventajas o desventajas dependiendo del contexto.

Las ideas presentes en estos algoritmos de enumeraciones finitas influyeron los métodos para las enumeraciones descritas en este trabajo, las cuáles son en su mayoría enumeraciones de una infinidad de objetos cuyas estructuras son similares a árboles de sintaxis.

La generalización de los algoritmos de enumeración de IMP en el capítulo 3 está inspirada en técnicas para reconocimiento sintáctico, en particular el conceptualizar las enumeraciones como objetos a ser combinados se basa en la técnica de combinadores de análisis sintáctico [27, 10] comúnmente empleada en programación funcional.

Capítulo 2

Enumeración de programas IMP

En este capítulo se describe el proceso para construir una enumeración de programas del lenguaje IMP. Posteriormente se muestra cómo las técnicas empleadas para lograr este fin pueden ser utilizadas para resolver otros problemas relacionados a la enumeración como el conteo de programas por tamaño y la selección aleatoria. Finalmente se muestra cómo estos algoritmos pueden ser usados para analizar con herramientas estadísticas el lenguaje y el ordenamiento inducido por la enumeración.

2.1 Construcción de un algoritmo de enumeración

La enumeración de todos los programas del lenguaje de programación IMP es modelada a partir de dos funciones. La función $\text{rank}_{\mathbf{P}}$ asocia a cada programa en **P** una posición en $\mathbb{N}$, mientras que la función $\text{unrank}_{\mathbf{P}}$ es la inversa de rank, es decir, asocia posiciones a programas.

Estas dos funciones deben inducir una biyección entre los naturales que representan posiciones y el conjunto de todos los programas IMP, es decir, a cada programa le corresponde una única posición y a cada posición le corresponde un único programa.

En esta sección se describe un proceso para definir una enumeración del lenguaje IMP. Primero se describe una especificación de la sintaxis del lenguaje. Posteriormente se definen relaciones entre posiciones y los distintos tipos de expresiones de la especificación. Finalmente se construyen algoritmos para las funciones rank y unrank para cada biyección.

A lo largo de este capítulo se le llama *enumeración* o *funciones de enumeración*

a parejas rank y unrank.

Especificación de IMP

La especificación de IMP utilizada en la enumeración de programas es similar a la gramática libre de contexto descrita en la anterior sección, sin embargo, presente algunas diferencias importantes.

Primero, las producciones asociadas a cada categoría sintáctica tienen un orden fijo el cual nos permite procesar un tipo de expresión en el lenguaje de acuerdo al orden que tiene en las producciones. Segundo, las derivaciones de una categoría sintáctica resultan en árboles de sintaxis en lugar de cadenas de símbolos. Tercero, el conjunto de terminales no es necesariamente finito, en particular se considera que los naturales en la especificación son naturales en $\mathbb{N}$ y no cadenas de dígitos.

Sea $\mathbf{C}$ una categoría sintáctica, se denota la cantidad de producciones asociadas a ella como $|\mathbf{C}|$ y la n-ésima producción como $\mathbf{C}(n)$. Una producción de la forma $\mathbf{C} \longrightarrow (\mathsf{R}\ \mathbf{C_1}\ \mathbf{C_2}\ \ldots\ \mathbf{C_n})$ denota que a partir de $\mathbf{C}$ es posible derivar un árbol de sintaxis con raíz R cuyos hijos pueden ser derivados a partir de $\mathbf{C_1}, \ldots, \mathbf{C_n}$ respectivamente.

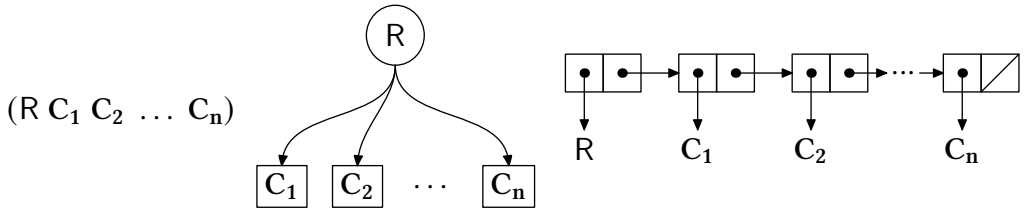


Figura 2.1: Representaciones de árboles de sintaxis.

Computacionalmente se considera que la estructura de los árboles sintácticos es una lista enlazada simple donde el primer elemento corresponde a la raíz y el resto de los elementos corresponden a los hijos. La relación entre la notación prefijo utilizada, la representación pictórica de los árboles y su estructura como listas se muestra en la figura 2.1.

La especificación de IMP es descrita similar a una gramática y el orden de las producciones corresponde al orden en que aparecen en (2.1). Se nombran las raíces de acuerdo al tipo de árbol sintáctico que representan: set para asignaciones de memoria, conc para concatenación de programas, add para la suma, sub para la resta, mul para la multiplicación, equal para la igualdad, less para menor que,

not para la negación, or para la disyunción, and para la conjunción y loc para las localidades de memoria. La producción de la categoría sintáctica **N** consiste de todos los números naturales.

$$
\begin{aligned}
\mathbf{P} &\to \mathsf{skip} \mid (\mathsf{set}\ \mathbf{X}\ \mathbf{A}) \mid (\mathsf{while}\ \mathbf{B}\ \mathbf{P}) \mid (\mathsf{conc}\ \mathbf{P}\ \mathbf{P}) \mid (\mathsf{if}\ \mathbf{B}\ \mathbf{P}\ \mathbf{P}) \\
\mathbf{B} &\to \mathsf{true} \mid \mathsf{false} \mid (\mathsf{equal}\ \mathbf{A}\ \mathbf{A}) \mid (\mathsf{less}\ \mathbf{A}\ \mathbf{A}) \\
&\quad \mid (\mathsf{and}\ \mathbf{B}\ \mathbf{B}) \mid (\mathsf{or}\ \mathbf{B}\ \mathbf{B}) \mid (\mathsf{not}\ \mathbf{B}) \\
\mathbf{A} &\to \mathbf{N} \mid \mathbf{X} \mid (\mathsf{add}\ \mathbf{A}\ \mathbf{A}) \mid (\mathsf{sub}\ \mathbf{A}\ \mathbf{A}) \mid (\mathsf{mul}\ \mathbf{A}\ \mathbf{A}) \\
\mathbf{X} &\to (\mathsf{loc}\ \mathbf{N}) \\
\mathbf{N} &\to \mathbb{N}
\end{aligned}
\tag{2.1}
$$

Utilizar esta especificación también permite prescindir de algunos aspectos de la sintaxis del lenguaje que son decorativos como then y else en las condicionales if o do en las iteraciones while.

Enumeración de naturales

Se comienza la construcción de la enumeración de programas IMP a partir de tipos de expresiones más simples. La categoría sintáctica más simple de la especificación (2.1) es **N**, y ya que no depende de ningún otra categoría sintáctica, es la primer enumeración que se desarrolla.

Tanto las posiciones como los objetos en una enumeración de **N** son números naturales, por lo que una forma simple de definir una biyección entre estos dos conjuntos es mediante su identidad, es decir, cada natural $n \in \mathbf{N}$ es asociado con la posición n y viceversa como se muestra en la figura 2.2.

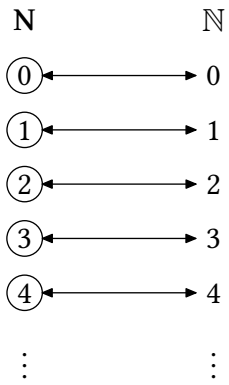


Figura 2.2: Relación entre posiciones y objetos en **N**.

A pesar de tener definiciones triviales, se precisa de forma explícita la definición de la enumeración de naturales a partir de las funciones $\text{rank}_{\mathbf{N}}$ y $\text{unrank}_{\mathbf{N}}$. Ambas son definidas sobre todos los naturales y su valor es por lo tanto un natural.

$$\text{rank}_{\mathbf{N}}(n) = n \tag{2.2}$$

$$\text{unrank}_{\mathbf{N}}(k) = k \tag{2.3}$$

Enumeración de localidades

Una vez definida la enumeración de naturales, se procede a construir la enumeración de localidades de memoria **X**.

De acuerdo a la especificación (2.1) los objetos derivables de **X** son árboles con raíz loc y un único hijo natural **N**. Una estrategia empleada a lo largo de este trabajo es definir la enumeración de objetos compuestos en términos de las enumeraciones de sus objetos constituyentes, por lo que $\text{rank}_{\mathbf{X}}$ y $\text{unrank}_{\mathbf{X}}$ son definidas en términos de $\text{rank}_{\mathbf{N}}$ y $\text{unrank}_{\mathbf{N}}$ respectivamente.

La figura 2.3 muestra la relación entre la forma de los árboles de sintaxis para localidades de memoria y la estructura computacional del árbol como lista enlazada donde el primer elemento es la raíz del árbol loc y el segundo elemento el natural asociado a la localidad.

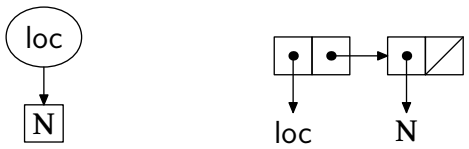


Figura 2.3: Árboles de sintaxis para localidades **X**.

Esta correspondencia entre localidades y naturales permite construir una enumeración simple para **X**, en donde el natural de una localidad corresponde a la posición del objeto, tal como se muestra en la figura 2.4.

Se define la enumeración de localidades con la función $\text{rank}_{\mathbf{X}}$ sobre todos los árboles de la forma $(\text{loc}\ n)$ con $n \in \mathbb{N}$ y la función $\text{unrank}_{\mathbf{X}}$ sobre todos los naturales $k \in \mathbb{N}$.

$$\text{rank}_{\mathbf{X}}((\text{loc}\ n)) = \text{rank}_{\mathbf{N}}(n) \tag{2.4}$$

$$\text{unrank}_{\mathbf{X}}(k) = (\text{loc}\ \ \text{unrank}_{\mathbf{N}}(k)) \tag{2.5}$$

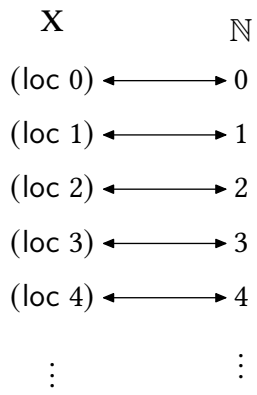


Figura 2.4: Relación entre posiciones y objetos en **X**.

Computacionalmente estas funciones se pueden implementar procesando estructuras de listas cuya primer componente es el identificador asociado al tipo de expresión.

Enumeración de expresiones aritméticas

Una vez construidas las enumeraciones de naturales y localidades, se procede a abordar el problema de enumerar las expresiones aritméticas **A**.

Como se muestra en la especificación (2.1) existen cinco tipos de árboles que conforman a las expresiones aritméticas, una por cada producción: naturales, localidades, sumas, restas y multiplicaciones. El orden de las producciones se muestra en la tabla 2.1.

i	$\mathbf{A}(i)$
0	**N**
1	**X**
2	(add **A** **A**)
3	(sub **A** **A**)
4	(mul **A** **A**)

Tabla 2.1: Orden de producciones de **A**.

La estrategia utilizada para construir esta enumeración consiste en *distribuir* todas las posiciones sobre las producciones de **A**. Ya que hay una infinidad de naturales **N** y localidades **X**, también habrá una infinidad de sumas, restas y

multiplicaciones. Cada una de estas producciones $\mathbf{A}(i)$ se asocia a un conjunto de naturales $\mathbb{N}_i$ los cuáles forman una partición de todas las posiciones. Los conjuntos $\mathbb{N}_i$ son disjuntos y su unión es $\mathbb{N}$.

Conceptualmente se particionan las posiciones de la siguiente forma: Primero se asocia la posición 0 a un objeto derivado de $\mathbf{A}(0)$, luego la posición 1 a un objeto derivado de $\mathbf{A}(1)$, luego la posición 2 a un objeto derivado de $\mathbf{A}(2)$, luego la posición 3 a un objeto derivado de $\mathbf{A}(3)$ hasta asociar la posición 4 a un objeto derivado de $\mathbf{A}(4)$.

Después de esta primera ronda se asocia la posición 5 a un objeto de la producción $\mathbf{A}(0)$, continuando con las posiciones 6, 7, 8 y 9 asociadas a objetos derivados de $\mathbf{A}(1)$, $\mathbf{A}(2)$, $\mathbf{A}(3)$ y $\mathbf{A}(4)$ respectivamente. Este proceso continua hasta asociar cada posición a una única producción de $\mathbf{A}$.

Formalmente, se caracteriza la distribución de posiciones en producciones mediante la división euclidiana: Dados dos naturales a y b, con b distinto a cero, existe una pareja única de naturales q y r, tal que $a = bq + r$ donde $r < b$. El valor a es llamado *dividendo*, b es llamado *divisor*, q es llamado *cociente* y r es llamado *residuo*. En las ecuaciones (2.6) se muestran las relaciones producto de esta división, donde div corresponde a la división entera y mod a su residuo.

$$\begin{aligned} a &= bq + r \\ q &= a \text{ div } b \\ r &= a \bmod b \end{aligned} \tag{2.6}$$

En el contexto de la enumeración de expresiones aritméticas, se considera que una posición cualquiera k es el dividendo, mientras que la cantidad de producciones $|\mathbf{A}|$ es el divisor. De tal forma que el índice de la producción asociada a k es $i = k \bmod |\mathbf{A}|$. Otra forma de analizar esta relación es considerar que todas las posiciones asociadas a la producción $\mathbf{A}(i)$ son de la forma $k = 5k' + i$. La figura 2.5 muestra la relación entre cada producción y las posiciones que les corresponden.

Desde la perspectiva algorítmica, dada una posición k se calcula $i = \bmod|\mathbf{A}|$ y $k' = k$ div $|\mathbf{A}|$ para determinar que la k-ésima expresión aritmética corresponde al k'-ésimo objeto enumerado por la producción $\mathbf{A}(i)$.

El proceso inverso consiste en partir de una expresión aritmética cualquiera. Inspeccionando su estructura se determina a qué producción $\mathbf{A}(i)$ corresponde, luego se calcula la posición k' del objeto con respecto a $\mathbf{A}(i)$ y finalmente se calcula la posición final $k = |\mathbf{A}|k' + i$.

Una posición de la forma $5k'$ corresponde al k'-ésimo natural en $\mathbf{N}$ y una posición de la forma $5k' + 1$ corresponde a la k'-ésima localidad en $\mathbf{X}$. El problema

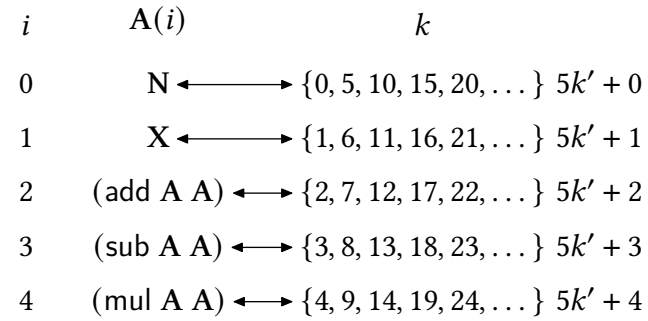
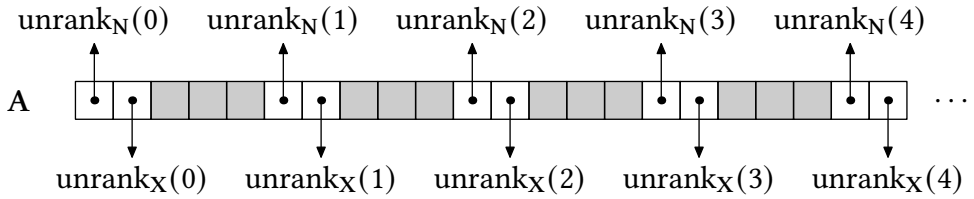


Figura 2.5: Relación entre posiciones y producciones en **A**.

se reduce a construir las enumeraciones de las tres producciones restantes.



Para enumerar las suma, resta y multiplicación de **A** se parte de la observación que estas tres enumeraciones son muy similares, en particular su única diferencia es el identificador de la raíz de los objetos enumerados.

La estrategia empleada para enumerar estos objetos compuestos consiste en utilizar una enumeración de duplas de naturales con funciones rank_2 y unrank_2. Esta enumeración permite relacionar posiciones k' a una pareja de posiciones k_1 y k_2. A partir de esta enumeración de duplas se asocia la posición $5k' + 2$ a la suma

$$(\mathsf{add}\ \ \text{unrank}_{\mathbf{A}}(k_1)\ \ \text{unrank}_{\mathbf{A}}(k_2)),$$

la posición $5k' + 3$ a la resta

$$(\mathsf{sub}\ \ \text{unrank}_{\mathbf{A}}(k_1)\ \ \text{unrank}_{\mathbf{A}}(k_2))$$

y la posición $5k' + 4$ a la multiplicación

$$(\mathsf{mul}\ \ \text{unrank}_{\mathbf{A}}(k_1)\ \ \text{unrank}_{\mathbf{A}}(k_2)).$$

Existen una infinidad de maneras de definir enumeraciones entre $\mathbb{N}$ y $\mathbb{N}^2$, sin embargo en este trabajo se considera que la enumeración utilizada satisface las

siguientes propiedades.

$$\text{rank}_2((k_1, k_2)) = k \implies k_1 \leq k \text{ y } k_2 \leq k \tag{2.7}$$
$$\text{unrank}_2(k) = (k_1, k_2) \implies k_1 \leq k \text{ y } k_2 \leq k \tag{2.8}$$

Al igual que con las localidades de memoria, los árboles de sintaxis de sumas, restas y multiplicaciones se codifican como listas enlazadas cuyo primer elemento es la raíz del árbol add, sub y mul respectivamente, el segundo y tercer elemento son a su vez árboles de sintaxis de expresiones aritméticas cualquiera como se muestra en la figura 2.6.

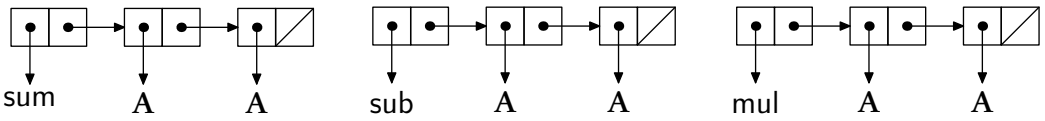


Figura 2.6: Árboles de sintaxis para expresiones aritméticas **A**.

Se define la enumeración de expresiones aritméticas con la función $\text{rank}_\mathbf{A}$ sobre todos objetos derivados de **A** y la función $\text{unrank}_\mathbf{A}$ sobre todas las posiciones naturales $k \in \mathbb{N}$.

$$\begin{aligned}
\text{rank}_\mathbf{A}(n) &= 5\,\text{rank}_\mathbf{N}(n) + 0 & \text{unrank}_\mathbf{A}(5k' + 0) &= \text{unrank}_\mathbf{N}(k') \\
\text{rank}_\mathbf{A}((\text{loc } n)) &= 5\,\text{rank}_\mathbf{X}((\text{loc } n)) + 1 & \text{unrank}_\mathbf{A}(5k' + 1) &= \text{unrank}_\mathbf{X}(k') \\
\text{rank}_\mathbf{A}((\text{add } a_1\ a_2)) &= 5k' + 2 & \text{unrank}_\mathbf{A}(5k' + 2) &= (\text{add } a_1\ a_2) \\
\text{rank}_\mathbf{A}((\text{sub } a_1\ a_2)) &= 5k' + 3 & \text{unrank}_\mathbf{A}(5k' + 3) &= (\text{sub } a_1\ a_2) \\
\text{rank}_\mathbf{A}((\text{mul } a_1\ a_2)) &= 5k' + 4 & \text{unrank}_\mathbf{A}(5k' + 4) &= (\text{mul } a_1\ a_2)
\end{aligned}$$

donde

$$\begin{aligned}
k_i &= \text{rank}_\mathbf{A}(a_i) & a_i &= \text{unrank}_\mathbf{A}(k_i) \\
k' &= \text{rank}_2((k_1, k_2)) & (k_1, k_2) &= \text{unrank}_2(k')
\end{aligned}$$

La enumeración de n-tuplas de naturales utilizada en este trabajo es descrita a detalle en [2]. Geométricamente estas enumeraciones asocian posiciones a cada celda de una cuadrícula n-dimensional a lo largo de diagonales. En la figura 2.7 se muestra la cuadrícula y las posiciones asociadas a cada celda para duplas.

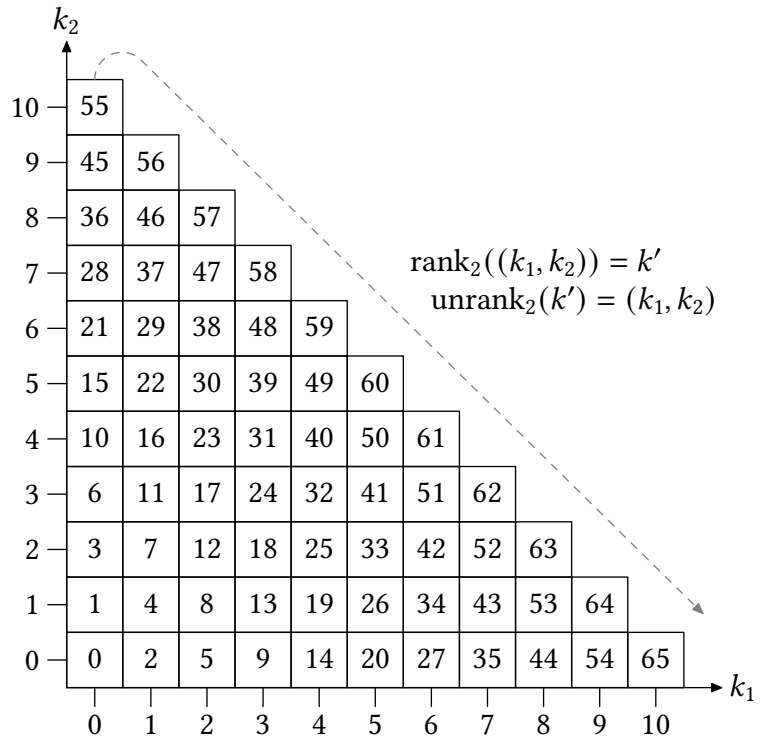


Figura 2.7: Enumeración de duplas.

Enumeración de expresiones booleanas

El problema de construir una enumeración para expresiones booleanas **B** es similar al de expresiones aritméticas, algunas producciones son sintácticamente simples y otras son compuestas.

De acuerdo a la especificación (2.1) las expresiones booleanas se derivan a partir de siete producciones, dos de ellas derivan un objeto, dos de ellas se construyen a partir de expresiones aritméticas y las tres restantes se construyen a partir de otras expresiones booleanas. El orden de las producciones se muestra en la tabla 2.2.

El método de distribución de posiciones utilizado para **A** no puede aplicarse directamente a este problema ya que $\mathbf{B}(0)$ y $\mathbf{B}(1)$ derivan un único objeto, por lo que deben relacionarse con una posición cada uno.

A partir de esta observación se propone resevar la posición 0 para true y la posición 1 para false, distribuyendo el resto de las posiciones sobre $\mathbf{B}(2)$, $\mathbf{B}(3)$, $\mathbf{B}(4)$, $\mathbf{B}(5)$ y $\mathbf{B}(6)$ usando la división euclidiana. La figura 2.8 muestra la relación

i	$\mathbf{B}(i)$
0	true
1	false
2	(equal **A** **A**)
3	(less **A** **A**)
4	(and **B** **B**)
5	(or **B** **B**)
6	(not **B**)

Tabla 2.2: Orden de producciones de **B**.

entre cada producción y las posiciones que les corresponden.

i	$\mathbf{B}(i)$	k	
0	true	$\{0\}$	0
1	false	$\{1\}$	1
2	(equal **A** **A**)	$\{2, 7, 12, 17, 22, \dots\}$	$5k' + 2$
3	(less **A** **A**)	$\{3, 8, 13, 18, 23, \dots\}$	$5k' + 3$
4	(and **B** **B**)	$\{4, 9, 14, 19, 24, \dots\}$	$5k' + 4$
5	(or **B** **B**)	$\{5, 10, 15, 20, 25, \dots\}$	$5k' + 5$
6	(not **B**)	$\{6, 11, 16, 21, 26, \dots\}$	$5k' + 6$

Figura 2.8: Relación entre posiciones y producciones en **B**.

Para aplicar el método de división euclidiana en este contexto se considera una posición $k \geq 2$ y la cantidad de producciones sobre las cuales particionar las posiciones. El dividendo es $k - 2$ y el divisor es $|\mathbf{B}| - 2$, por lo que

$$k' = (k - 2) \text{ div } (|B| - 2)$$
$$i = (k - 2) \text{ mod } (|B| - 2).$$

Con estas ecuaciones se obtiene que el valor de $i \in [0, |B| - 2) = [0, 5)$, sin embargo, debido a que las producciones sobre las que se distribuyen las posiciones son aquellas donde $i \in [2, 7)$, esta corrección al valor de i se obtiene al considerar

la ecuación

$$\begin{aligned} k - 2 &= (|B| - 2)k' + i \\ k &= (|B| - 2)k' + (i + 2) \end{aligned}$$

por lo que las posiciones asociadas a la producción $\mathbf{B}(i + 2)$ con son de la forma $5k' + i + 2$.

El proceso inverso consiste en partir de una expresión booleana cualquiera. Inspeccionando su estructura se determina a qué producción $\mathbf{B}(i)$ corresponde. Para $\mathbf{B}(0)$ la posición asociada es cero, para $\mathbf{B}(1)$ la posición asociada es uno. En otro caso, se calcula la posición k' del objeto con respecto a $\mathbf{B}(i)$ y finalmente se calcula la posición final $k = (|\mathbf{B}| - 2)k' + i$.

Se define la enumeración de expresiones booleanas con la función $\text{rank}_\mathbf{B}$ sobre todos los objetos derivados de $\mathbf{B}$ y la función $\text{unrank}_\mathbf{B}$ sobre todas las posiciones naturales $k \in \mathbb{N}$.

$$\begin{aligned} \text{rank}_\mathbf{B}(\mathsf{true}) &= 0 & \text{unrank}_\mathbf{B}(0) &= \mathsf{true} \\ \text{rank}_\mathbf{B}(\mathsf{false}) &= 1 & \text{unrank}_\mathbf{B}(1) &= \mathsf{false} \\ \text{rank}_\mathbf{B}((\mathsf{equal}\ a_1\ a_2)) &= 5k' + 2 & \text{unrank}_\mathbf{B}(5k' + 2) &= (\mathsf{equal}\ a_1\ a_2) \\ \text{rank}_\mathbf{B}((\mathsf{less}\ a_1\ a_2)) &= 5k' + 3 & \text{unrank}_\mathbf{B}(5k' + 3) &= (\mathsf{less}\ a_1\ a_2) \\ \text{rank}_\mathbf{B}((\mathsf{and}\ b_1\ b_2)) &= 5k' + 4 & \text{unrank}_\mathbf{B}(5k' + 4) &= (\mathsf{and}\ b_1\ b_2) \\ \text{rank}_\mathbf{B}((\mathsf{or}\ b_1\ b_2)) &= 5k' + 5 & \text{unrank}_\mathbf{B}(5k' + 5) &= (\mathsf{or}\ b_1\ b_2) \\ \text{rank}_\mathbf{B}((\mathsf{not}\ b)) &= 5k' + 6 & \text{unrank}_\mathbf{B}(5k' + 6) &= (\mathsf{not}\ b) \end{aligned}$$

para las definiciones correspondientes a $\mathbf{B}(2)$ y $\mathbf{B}(3)$ se considera

$$\begin{aligned} k_i &= \text{rank}_\mathbf{A}(a_i) & a_i &= \text{unrank}_\mathbf{A}(k_i) \\ k' &= \text{rank}_2((k_1, k_2)) & (k_1, k_2) &= \text{unrank}_2(k') \end{aligned}$$

para las definiciones correspondientes a $\mathbf{B}(4)$ y $\mathbf{B}(5)$ se considera

$$\begin{aligned} k_i &= \text{rank}_\mathbf{B}(b_i) & b_i &= \text{unrank}_\mathbf{B}(k_i) \\ k' &= \text{rank}_2((k_1, k_2)) & (k_1, k_2) &= \text{unrank}_2(k') \end{aligned}$$

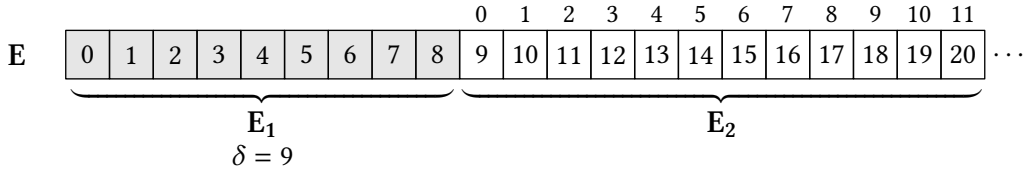
y para las definiciones correspondientes a $\mathbf{B}(6)$ se considera

$$k' = \text{rank}_\mathbf{B}(b) \qquad b = \text{unrank}_\mathbf{B}(k')$$

Este mecanismo combina el método de la división euclidiana con casos particulares para construir una enumeración se puede conceptualizar como una combinación de dos enumeraciones.

Sea $\mathbf{E_1}$ una enumeración correspondiente a las producciones que derivan conjuntos finitos de objetos y $\mathbf{E_2}$ una enumeración correspondiente a producciones que derivan conjuntos infinitos de objetos, se asignan posiciones como casos particulares a los objetos de $\mathbf{E_1}$ y se utiliza el método de división euclidiana para construir $\mathbf{E_2}$ donde sus posiciones son desplazadas $\delta = |\mathbf{E_1}|$ posiciones.

La siguiente figura considera como ejemplo las posiciones de $\mathbf{E_1}$ y $\mathbf{E_2}$, así como las posiciones de la enumeración combinada, cuando $\delta = 9$.



Enumeración de programas

Con las estrategias empleadas en la enumeración de expresiones booleanas se construye una enumeración para programas **P**. En este caso se tienen cinco producciones, como se muestra en la tabla 2.3, la producción $\mathbf{P}(0)$ deriva únicamente al objeto skip.

i	$\mathbf{P}(i)$
0	skip
1	(set **X** **A**)
2	(while **B** **P**)
3	(conc **P** **P**)
4	(if **B** **P** **P**)

Tabla 2.3: Orden de producciones de **P**.

Para distribuir las posiciones sobre las producciones de **P** se reserva la posición 0 para skip y se distribuyen el resto de las posiciones sobre el resto de las producciones usando la división euclidiana (2.6) como se muestra en la figura 2.9.

Para enumerar las asignaciones de memoria set, las iteraciones while y las concatenaciones de programas conc se utiliza la enumeración de duplas tal y

i	$\mathbf{P}(i)$	k	
0	skip	$\{0\}$	0
1	(set **X A**)	$\{1, 5, 9, 13, 17, 21, \dots\}$	$4k' + 1$
2	(while **B P**)	$\{2, 6, 10, 14, 18, 22, \dots\}$	$4k' + 2$
3	(conc **P P**)	$\{3, 7, 11, 15, 19, 23, \dots\}$	$4k' + 3$
4	(if **B P P**)	$\{4, 8, 12, 16, 20, 24, \dots\}$	$4k' + 4$

Figura 2.9: Relación entre posiciones y producciones en **P**.

como se aborda en las anteriores enumeraciones. Sin embargo, para enumerar las condicionales if se debe encontrar una forma de enumerar tripletas ya que estas expresiones se conforman de una expresión booleana y de dos programas.

Se considera la enumeración de tripletas con las funciones rank_3 y unrank_3 definidas en [2]. Estas funciones satisfacen las propiedades (2.7) y (2.8) para tres componentes, es decir, la posición de una tripleta es una cota superior de sus componentes. La figura 2.10 muestra las primeras tres diagonales tridimensionales sobre las cuales se ordenan las tripletas representadas por cubos numerados con la posición y ubicados en las coordenadas que establecen sus componentes.

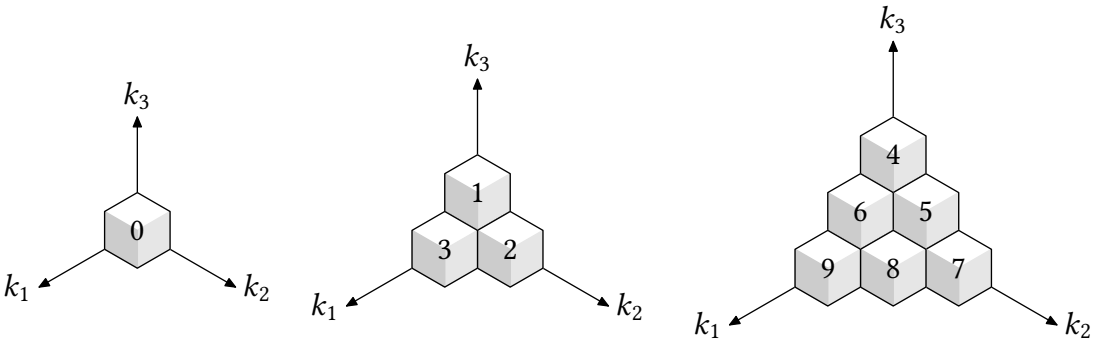


Figura 2.10: Enumeración de tripletas.

Se define la enumeración de programas con la función $\text{rank}_{\mathbf{P}}$ sobre todos los objetos derivados de **P** y la función $\text{unrank}_{\mathbf{P}}$ sobre todas las posiciones naturales $k \in \mathbb{N}$.

$$\text{rank}_{\mathbf{P}}(\text{skip}) = 0 \qquad \text{unrank}_{\mathbf{P}}(0) = \text{skip}$$

Para las asignaciones de memoria

$$\begin{aligned} k_1 &= \text{rank}_{\mathbf{X}}(x) & x &= \text{unrank}_{\mathbf{X}}(k_1) \\ k_2 &= \text{rank}_{\mathbf{A}}(a) & a &= \text{unrank}_{\mathbf{A}}(k_2) \\ k' &= \text{rank}_2((k_1, k_2)) & (k_1, k_2) &= \text{unrank}_2(k') \\ \text{rank}_{\mathbf{P}}((\text{set } x\ a)) &= 4k' + 1 & \text{unrank}_{\mathbf{P}}(4k' + 1) &= (\text{set } x\ a) \end{aligned}$$

Para iteraciones

$$\begin{aligned} k_1 &= \text{rank}_{\mathbf{B}}(b) & b &= \text{unrank}_{\mathbf{B}}(k_1) \\ k_2 &= \text{rank}_{\mathbf{P}}(p) & p &= \text{unrank}_{\mathbf{P}}(k_2) \\ k' &= \text{rank}_2((k_1, k_2)) & (k_1, k_2) &= \text{unrank}_2(k') \\ \text{rank}_{\mathbf{P}}((\text{while } b\ p)) &= 4k' + 2 & \text{unrank}_{\mathbf{P}}(4k' + 2) &= (\text{while } b\ p) \end{aligned}$$

Para concatenaciones de programas

$$\begin{aligned} k_i &= \text{rank}_{\mathbf{P}}(p_i) & p_i &= \text{unrank}_{\mathbf{P}}(k_i) \\ k' &= \text{rank}_2((k_1, k_2)) & (k_1, k_2) &= \text{unrank}_2(k') \\ \text{rank}_{\mathbf{P}}((\text{conc } p_1\ p_2)) &= 4k' + 3 & \text{unrank}_{\mathbf{P}}(4k' + 3) &= (\text{conc } p_1\ p_2) \end{aligned}$$

Finalmente, para condicionales

$$\begin{aligned} k_1 &= \text{rank}_{\mathbf{B}}(b) & b &= \text{unrank}_{\mathbf{B}}(k_1) \\ k_2 &= \text{rank}_{\mathbf{P}}(p_1) & p_1 &= \text{unrank}_{\mathbf{P}}(k_2) \\ k_3 &= \text{rank}_{\mathbf{P}}(p_2) & p_2 &= \text{unrank}_{\mathbf{P}}(k_3) \\ k' &= \text{rank}_3((k_1, k_2, k_3)) & (k_1, k_2, k_3) &= \text{unrank}_3(k') \\ \text{rank}_{\mathbf{P}}((\text{if } b\ p_1\ p_2)) &= 4k' + 4 & \text{unrank}_{\mathbf{P}}(4k' + 4) &= (\text{if } b\ p_1\ p_2) \end{aligned}$$

Para ilustrar el cálculo de posiciones y la construcción de objetos en la enumeración de IMP se considera el programa de la figura 2.11. La expresión del lado izquierdo corresponde a la representación textual del programa de acuerdo a la gramática libre de contexto descrita en 1.2. La expresión del lado derecho corresponde a la representación textual del árbol de sintaxis asociado de acuerdo a la especificación (2.1).

La ejecución de este programa resulta en un estado de memoria donde la localidad x_1 almacena el valor $5! = 120$.

```
((x0 := 0 ;                    (conc (conc (set (loc 0) 0)
  x1 := 1) ;                             (set (loc 1) 1))
 (while (x0 < 5) do                (while (less (loc 0) 5)
   (x0 := (x0 + 1) ;                       (conc (set (loc 0) (add (loc 0) 1))
    x1 := (x0 × x1)))                             (set (loc 1) (mul (loc 0) (loc 1))))))
```

Figura 2.11: Programa IMP para el cálculo de 5!.

Las definiciones de las funciones rank y unrank para las categorías sintácticas de IMP describen un proceso recursivo en donde la posición de una expresión se calcula a partir de las posiciones de las expresiones que la conforman. Para este ejemplo se propone sustituir las subexpresiones con posiciones conocidas utilizando la notación $n_{\mathbf{C}}$ que denota el n-ésimo objeto de la enumeración $\mathbf{C}$, por lo que la posición final se calcula desde las hojas del árbol hacia la raíz.

1. Utilizando la definición (2.2), se calcula la posición de cada natural n en el programa.

```
(conc (conc (set (loc 0_N) 0_N)
            (set (loc 1_N) 1_N))
      (while (less (loc 0_N) 5_N)
             (conc (set (loc 0_N) (add (loc 0_N) 1_N))
                   (set (loc 1_N) (mul (loc 0_N) (loc 1_N))))))
```

2. Utilizando la definición (2.4), se tiene que

$$\mathrm{rank}_{\mathbf{X}}((\mathsf{loc}\ \mathsf{n})) = \mathrm{rank}_{\mathbf{N}}(\mathsf{n}) = n,$$

por lo que se prosigue a calcular la posición de cada localidad en el programa.

```
(conc (conc (set 0_X 0_N)
            (set 1_X 1_N))
      (while (less 0_X 5_N)
             (conc (set 0_X (add 0_X 1_N))
                   (set 1_X (mul 0_X 1_X)))))
```

3. Los números naturales en las expresiones set, less y add corresponden a expresiones aritméticas. De acuerdo a la enumeración de $\mathbf{A}$ se tiene que

$$\mathrm{rank}_{\mathbf{A}}(n) = 5\,\mathrm{rank}_{\mathbf{N}}(n),$$

por lo que cada $n_{\mathbf{N}}$ en el programa se sustituye por $5n_{\mathbf{A}}$.

(conc (conc (set $0_{\mathbf{X}}$ $0_{\mathbf{A}}$)
(set $1_{\mathbf{X}}$ $5_{\mathbf{A}}$))
(while (less $0_{\mathbf{X}}$ $25_{\mathbf{A}}$)
(conc (set $0_{\mathbf{X}}$ (add $0_{\mathbf{X}}$ $5_{\mathbf{A}}$))
(set $1_{\mathbf{X}}$ (mul $0_{\mathbf{X}}$ $1_{\mathbf{X}}$)))))

4. Las localidades en las expresiones less, add y mul también corresponden a expresiones aritméticas. De acuerdo a la enumeración de **A** se tiene que

$$\mathrm{rank}_{\mathbf{A}}(n_{\mathbf{X}}) = 5n + 1,$$

por lo que cada $n_{\mathbf{X}}$ en el programa se sustituye por $(5n + 1)_{\mathbf{A}}$.

(conc (conc (set $0_{\mathbf{X}}$ $0_{\mathbf{A}}$)
(set $1_{\mathbf{X}}$ $5_{\mathbf{A}}$))
(while (less $1_{\mathbf{A}}$ $25_{\mathbf{A}}$)
(conc (set $0_{\mathbf{X}}$ (add $1_{\mathbf{A}}$ $5_{\mathbf{A}}$))
(set $1_{\mathbf{X}}$ (mul $1_{\mathbf{A}}$ $6_{\mathbf{A}}$)))))

5. Ahora se procede a calcular la posición de las expresiones aritméticas add y mul. Se utiliza la enumeración de duplas para combinar las posiciones de los operandos y se obtiene

$$\mathrm{rank}_2((1, 5)) = 22,$$
$$\mathrm{rank}_2((1, 6)) = 29.$$

De acuerdo a la enumeración de **A** se tiene que

$$\mathrm{rank}_{\mathbf{A}}((\mathsf{add}\ 1_{\mathbf{A}}\ 5_{\mathbf{A}})) = 5 \times 22 + 2 = 112$$
$$\mathrm{rank}_{\mathbf{A}}((\mathsf{mul}\ 1_{\mathbf{A}}\ 6_{\mathbf{A}})) = 5 \times 29 + 4 = 149$$

por lo que (add $1_{\mathbf{A}}$ $5_{\mathbf{A}}$) se sustituye por $112_{\mathbf{A}}$ y (mul $1_{\mathbf{A}}$ $6_{\mathbf{A}}$) se sustituye por $149_{\mathbf{A}}$.

(conc (conc (set $0_{\mathbf{X}}$ $0_{\mathbf{A}}$)
(set $1_{\mathbf{X}}$ $5_{\mathbf{A}}$))
(while (less $1_{\mathbf{A}}$ $25_{\mathbf{A}}$)
(conc (set $0_{\mathbf{X}}$ $112_{\mathbf{A}}$)
(set $1_{\mathbf{X}}$ $149_{\mathbf{A}}$))))

6. La posición de la expresión booleana (less $1_{\mathbf{A}}$ $25_{\mathbf{A}}$) se calcula de forma similar. Primero se utiliza la enumeración de duplas para combinar las posiciones de los operandos y se obtiene

$$\text{rank}_2((1, 25)) = 352.$$

De acuerdo a la enumeración de **B** se tiene que

$$\text{rank}_{\mathbf{B}}((\text{less } 1_{\mathbf{A}}\ 25_{\mathbf{A}})) = 5 \times 352 + 3 = 1763$$

por lo que (less $1_{\mathbf{A}}$ $25_{\mathbf{A}}$) se sustituye por $1763_{\mathbf{B}}$.

(conc (conc (set $0_{\mathbf{X}}$ $0_{\mathbf{A}}$)
(set $1_{\mathbf{X}}$ $5_{\mathbf{A}}$))
(while $1763_{\mathbf{B}}$
(conc (set $0_{\mathbf{X}}$ $112_{\mathbf{A}}$)
(set $1_{\mathbf{X}}$ $149_{\mathbf{A}}$))))

7. En este punto del cálculo todas las asignaciones de memoria están escritas en términos de la posición de su localidad y la posición de su expresión aritmética, por lo que se puede calcular directamente la posición de cada asignación. Utilizando la enumeración de duplas se tiene que

$$\begin{aligned}
\text{rank}_2((0,\ 0)) &= 0 \\
\text{rank}_2((1,\ 5)) &= 22 \\
\text{rank}_2((0,\ 112)) &= 6328 \\
\text{rank}_2((1,\ 149)) &= 11{,}326
\end{aligned}$$

De acuerdo a la enumeración de **P** se tiene que

$$\begin{aligned}
\text{rank}_{\mathbf{P}}((\text{set } 0_{\mathbf{X}}\ 0_{\mathbf{A}})) &= 4 \times 0 + 1 = 1 \\
\text{rank}_{\mathbf{P}}((\text{set } 1_{\mathbf{X}}\ 5_{\mathbf{A}})) &= 4 \times 22 + 1 = 89 \\
\text{rank}_{\mathbf{P}}((\text{set } 0_{\mathbf{X}}\ 112_{\mathbf{A}})) &= 4 \times 6328 + 1 = 25{,}313 \\
\text{rank}_{\mathbf{P}}((\text{set } 1_{\mathbf{X}}\ 149_{\mathbf{A}})) &= 4 \times 11{,}326 + 1 = 45{,}305
\end{aligned}$$

por lo que se sustituyen las asignaciones de memoria por su posición calculada.

(conc (conc $1_{\mathbf{P}}$
$89_{\mathbf{P}}$)
(while $1763_{\mathbf{B}}$

(conc $25{,}313_{\mathbf{P}}$
$45{,}305_{\mathbf{P}}$)))

8. Este mismo proceso se realiza para las concatenaciones internas.

$$\text{rank}_2((1,\ 89)) = 4096$$
$$\text{rank}_2((25{,}313,\ 45{,}305)) = 2{,}493{,}511{,}584$$

De acuerdo a la enumeración de **P** se tiene que

$$\text{rank}_{\mathbf{P}}((\text{conc } 1_{\mathbf{P}}\ 89_{\mathbf{P}})) = 4 \times 4096 + 3 = 16{,}387$$
$$\text{rank}_{\mathbf{P}}((\text{conc } 25{,}313_{\mathbf{P}}\ 45{,}305_{\mathbf{P}})) = 4 \times 2{,}493{,}511{,}584 + 3 = 9{,}974{,}046{,}339$$

por lo que se sustituyen las concatenaciones por sus posiciones calculada.

(conc $16{,}387_{\mathbf{P}}$
(while $1763_{\mathbf{B}}$
$9{,}974{,}046{,}339_{\mathbf{P}}$))

9. Se procede realizando el cálculo correspondiente a la iteración.

$$\text{rank}_2((1763,\ 9{,}974{,}046{,}339)) = 49{,}740{,}817{,}775{,}491{,}927{,}016$$

De acuerdo a la enumeración de **P** se tiene que

$$\begin{aligned}\text{rank}_{\mathbf{P}}((\text{while } 1763_{\mathbf{B}}\ 9{,}974{,}046{,}339_{\mathbf{P}})) &= 4 \times 49{,}740{,}817{,}775{,}491{,}927{,}016 + 2 \\ &= 198{,}963{,}271{,}101{,}967{,}708{,}066\end{aligned}$$

por lo que se sustituye la iteración por su posición calculada.

(conc $16{,}387_{\mathbf{P}}$
$198{,}963{,}271{,}101{,}967{,}708{,}066_{\mathbf{P}}$)

10. Finalmente se calcula la posición del programa en general realizando el cálculo con la concatenación restante.

$$\begin{aligned}&\text{rank}_2(16{,}387,\ 198{,}963{,}271{,}101{,}967{,}708{,}066) \\ &= 19{,}793{,}191{,}623{,}797{,}552{,}498{,}893{,}755{,}813{,}612{,}450{,}953{,}218\end{aligned}$$

De acuerdo a la enumeración de **P** se tiene que

$$\begin{aligned}
&\text{rank}_{\mathbf{P}}((\text{conc } 16{,}387_{\mathbf{P}}\ 198{,}963{,}271{,}101{,}967{,}708{,}066_{\mathbf{P}}))\\
&= 4 \times 19{,}793{,}191{,}623{,}797{,}552{,}498{,}893{,}755{,}813{,}612{,}450{,}953{,}218 + 3\\
&= 79{,}172{,}766{,}495{,}190{,}209{,}995{,}575{,}023{,}254{,}449{,}803{,}812{,}875
\end{aligned}$$

Por lo que el programa ejemplo para el cálculo de 5! está presente en la enumeración IMP en la posición

$$79{,}172{,}766{,}495{,}190{,}209{,}995{,}575{,}023{,}254{,}449{,}803{,}812{,}875.$$

Para calcular el proceso inverso se siguen pasos similares, pero utilizando las funciones unrank en lugar de rank. Por ejemplo, sea k de la posición final calculada, se determina la estructura parcial del programa de la siguiente manera.

Ya que $k \neq 0$ entonces el programa no es skip, para determinar la producción correspondiente realizan los siguientes cálculos.

$$\begin{aligned}
i &= 1 + (k-1) \bmod 4 = 1 + 2 = 3\\
k' &= (k-1) \text{ div } 4\\
&= 19{,}793{,}191{,}623{,}797{,}552{,}498{,}893{,}755{,}813{,}612{,}450{,}953{,}218
\end{aligned}$$

Ya que la producción correspondiente es $\mathbf{P}(i) = \mathbf{P}(3) = (\text{conc } \mathbf{P}\ \mathbf{P})$ se debe distribuir k' en dos posiciones, las cuáles se determinan a partir de $\text{unrank}_2(k')$.

$$\begin{aligned}
&\text{unrank}_2(19{,}793{,}191{,}623{,}797{,}552{,}498{,}893{,}755{,}813{,}612{,}450{,}953{,}218)\\
&= (16{,}387,\ 198{,}963{,}271{,}101{,}967{,}708{,}066)
\end{aligned}$$

el primer componente de la dupla corresponde a la posición del primer programa concatenado y el segundo componente de la dupla a la posición del segundo programa concatenado. Por lo que

$$\begin{aligned}
&\text{unrank}_{\mathbf{P}}(79{,}172{,}766{,}495{,}190{,}209{,}995{,}575{,}023{,}254{,}449{,}803{,}812{,}875)\\
&= (\text{conc unrank}_{\mathbf{P}}(16{,}387)\\
&\qquad \text{unrank}_{\mathbf{P}}(198{,}963{,}271{,}101{,}967{,}708{,}066))
\end{aligned}$$

Este procedimiento continua hasta obtener el árbol de sintaxis completo.

2.2 Conteo y selección aleatoria por tamaño

La modelación de enumeraciones con funciones rank y unrank permite explorar el espacio de programas IMP de acuerdo a su estructura sintáctica. A partir de una posición k se pueden analizar los programas en su vecindad numérica con $\text{unrank}_{\mathbf{P}}(k-1)$ y $\text{unrank}_{\mathbf{P}}(k+1)$, y a partir de un programa p se pueden analizar las posiciones en su vecindad sintáctica cambiando partes de las expresiones que lo constituyen y calculando su $\text{rank}_{\mathbf{P}}$.

Los métodos y estrategias utilizadas para construir estas enumeraciones, en particular el uso de n-tuplas de naturales [2] y la división euclidiana (2.6), permiten también explorar el espacio de programas desde otras perspectivas.

En esta sección se presentan mecanismos para trabajar con la enumeración del lenguaje IMP a partir de una noción de tamaño de programas y de las expresiones que los conforman.

Métrica de tamaño

Existen diversas formas de definir métricas de tamaño para los árboles de sintaxis enumerados, en este trabajo se propone definir el tamaño a partir de tres tipos de árboles.

1. Números naturales,
2. Hojas de identificadores como skip, true o false,
3. Objetos compuestos de una raíz y uno o más hijos.

El tamaño de números naturales se define como la cantidad de dígitos que lo representan, el tamaño de las hojas de identificadores se define como 1 y el tamaño de los objetos compuestos se define como uno más que la suma de los tamaños de sus hijos. Esta métrica consiste entonces en contar la cantidad de vértices del árbol considerando que un natural se conforma por tantos vértices como dígitos.

Para cada categoría sintáctica C del lenguaje se define la función $\text{size}_{\mathbf{C}}$ que asocia objetos derivados de C a su tamaño.

El tamaño de los naturales se define por casos. Para $n > 0$ se calcula el logaritmo base diez partiendo del hecho que el primer natural con d dígitos

es 10^{d-1}. Ya que $\log_{10}(10^{d-1}) + 1 = d$ se define $\text{size}_{\mathbf{N}}$ como en la ecuación (2.9).

$$\text{size}_{\mathbf{N}}(n) = \begin{cases} 1 & n = 0 \\ \lfloor \log_{10}(n) + 1 \rfloor & n > 0. \end{cases} \tag{2.9}$$

Las localidades de memoria son árboles con una raíz y un hijo natural, su tamaño se define

$$\text{size}_{\mathbf{X}}((\mathsf{loc}\ n)) = 1 + \text{size}_{\mathbf{N}}(n). \tag{2.10}$$

Para el caso de expresiones aritméticas, el tamaño es calculado por $\text{size}_{\mathbf{N}}$ y $\text{size}_{\mathbf{X}}$ para naturales y localidades respectivamente. Para sumas, restas y multiplicaciones se considera el aporte de la raíz del árbol en el tamaño.

$$\begin{aligned} \text{size}_{\mathbf{A}}(n) &= \text{size}_{\mathbf{N}}(n) \\ \text{size}_{\mathbf{A}}((\mathsf{loc}\ n)) &= \text{size}_{\mathbf{X}}((\mathsf{loc}\ n)) \\ \text{size}_{\mathbf{A}}((\mathsf{add}\ a_1\ a_2)) &= 1 + \text{size}_{\mathbf{A}}(a_1) + \text{size}_{\mathbf{A}}(a_2) \\ \text{size}_{\mathbf{A}}((\mathsf{sub}\ a_1\ a_2)) &= 1 + \text{size}_{\mathbf{A}}(a_1) + \text{size}_{\mathbf{A}}(a_2) \\ \text{size}_{\mathbf{A}}((\mathsf{mul}\ a_1\ a_2)) &= 1 + \text{size}_{\mathbf{A}}(a_1) + \text{size}_{\mathbf{A}}(a_2). \end{aligned} \tag{2.11}$$

Las expresiones boolenas true y false se consideran hojas de identificadores, por lo que su tamaño es unitario, en otro caso la expresión es un árbol con una raíz y uno o dos hijos, su tamaño se define

$$\begin{aligned} \text{size}_{\mathbf{B}}(\mathsf{true}) &= 1 \\ \text{size}_{\mathbf{B}}(\mathsf{false}) &= 1 \\ \text{size}_{\mathbf{B}}((\mathsf{equal}\ a_1\ a_2)) &= 1 + \text{size}_{\mathbf{A}}(a_1) + \text{size}_{\mathbf{A}}(a_2) \\ \text{size}_{\mathbf{B}}((\mathsf{less}\ a_1\ a_2)) &= 1 + \text{size}_{\mathbf{A}}(a_1) + \text{size}_{\mathbf{A}}(a_2) \\ \text{size}_{\mathbf{B}}((\mathsf{and}\ b_1\ b_2)) &= 1 + \text{size}_{\mathbf{B}}(b_1) + \text{size}_{\mathbf{B}}(b_2) \\ \text{size}_{\mathbf{B}}((\mathsf{or}\ b_1\ b_2)) &= 1 + \text{size}_{\mathbf{B}}(b_1) + \text{size}_{\mathbf{B}}(b_2) \\ \text{size}_{\mathbf{B}}((\mathsf{not}\ b)) &= 1 + \text{size}_{\mathbf{B}}(b). \end{aligned} \tag{2.12}$$

Finalmente, el programa skip se considera una hoja de identificador, mientras que el resto de los programas son árboles con una raíz y dos o tres hijos, su tamaño

se define

$$\begin{aligned}
\text{size}_{\mathbf{P}}(\mathsf{skip}) &= 1 \\
\text{size}_{\mathbf{P}}((\mathsf{set}\ x\ a)) &= 1 + \text{size}_{\mathbf{X}}(x) + \text{size}_{\mathbf{A}}(a) \\
\text{size}_{\mathbf{P}}((\mathsf{while}\ b\ p)) &= 1 + \text{size}_{\mathbf{B}}(b) + \text{size}_{\mathbf{P}}(p) \\
\text{size}_{\mathbf{P}}((\mathsf{conc}\ p_1\ p_2)) &= 1 + \text{size}_{\mathbf{P}}(p_1) + \text{size}_{\mathbf{P}}(p_2) \\
\text{size}_{\mathbf{P}}((\mathsf{if}\ b\ p_1\ p_2)) &= 1 + \text{size}_{\mathbf{B}}(b) + \text{size}_{\mathbf{P}}(p_1) + \text{size}_{\mathbf{P}}(p_2).
\end{aligned} \tag{2.13}$$

Una vez definida la métrica de tamaño, se plantean problemas relacionados a ella, el primero de estos que se aborda es calcular la cantidad de programas de tamaño m. La estrategia utilizada para realizar estos cálculos es similar a la utilizada en la construcción de las enumeraciones, abordando las categorías sintácticas de más simples a más complejas.

Conteo de programas

Para cada categoría sintáctica **C** del lenguaje se define la función $\text{count}_{\mathbf{C}}$ que asocia cada tamaño m a la cantidad de objetos derivados de **C** con tamaño m.

Ya que el tamaño de un natural es la cantidad de dígitos que lo conforman, para contar la cantidad de naturales de tamaño m se deben contar los naturales que se pueden construir con m dígitos. Cuando $m = 0$, corresponden cero naturales y cuando $m = 1$ corresponden diez naturales (del cero al nueve).

Para tamaños $m > 1$ el problema se reduce a contar la cantidad de elementos en el intervalo

$$[10^{m-1}, 10^m - 1],$$

ya que 10^{m-1} es el primer natural con m dígitos y 10^m es el primer natural con $m + 1$ dígitos. Este cálculo se realiza de la siguiente manera

$$\begin{aligned}
|[10^{m-1}, 10^m - 1]| &= |[0, 10^m - 1]| - |[0, 10^{m-1} - 1]| \\
&= (10^m - 1 - 0 + 1) - (10^{m-1} - 1 - 0 + 1) \\
&= 10^m - 10^{m-1} \\
&= 10 \times 10^{m-1} - 10^{m-1} \\
&= (10 - 1)10^{m-1} \\
&= 9 \times 10^{m-1}
\end{aligned}$$

por lo que la definición del conteo de naturales por tamaño es

$$\begin{aligned} \mathrm{count}_{\mathbf{N}}(0) &= 0 \\ \mathrm{count}_{\mathbf{N}}(1) &= 10 \\ \mathrm{count}_{\mathbf{N}}(m) &= 9 \times 10^{m-1}. \end{aligned} \tag{2.14}$$

La definición del conteo de localidades de memoria es más simple. No existen localidades de tamaño cero ya que la raíz se cuenta como una unidad del tamaño. Si una localidad tiene tamaño m entonces el natural de la localidad debe tener tamaño $m-1$.

$$\begin{aligned} \mathrm{count}_{\mathbf{X}}(0) &= 0 \\ \mathrm{count}_{\mathbf{X}}(m) &= \mathrm{count}_{\mathbf{N}}(m-1). \end{aligned} \tag{2.15}$$

Las expresiones aritméticas pueden ser naturales, localidades u operaciones binarias. Para definir su tamaño se aborda el cálculo de cada producción de forma independiente, de tal manera que $\mathrm{count}_{\mathsf{add}}$, $\mathrm{count}_{\mathsf{sub}}$ y $\mathrm{count}_{\mathsf{mul}}$ calculan la cantidad de sumas, restas y multiplicaciones respectivamente.

$$\begin{aligned} \mathrm{count}_{\mathbf{A}}(m) = &\ \mathrm{count}_{\mathbf{N}}(m) \\ &+ \mathrm{count}_{\mathbf{X}}(m) \\ &+ \mathrm{count}_{\mathsf{add}}(m) \\ &+ \mathrm{count}_{\mathsf{sub}}(m) \\ &+ \mathrm{count}_{\mathsf{mul}}(m) \end{aligned} \tag{2.16}$$

Debido a su estructura sintáctica, la cantidad de sumas, restas y multiplicaciones de tamaño m es la misma. Estos tres tipos de expresión consisten de una raíz y dos hijos que son expresiones aritméticas. Para simplificar sus definiciones se considera el conjunto de todas las parejas de expresiones aritméticas (a_1, a_2) tales que $\mathrm{size}_{\mathbf{A}}(a_1) + \mathrm{size}_{\mathbf{A}}(a_1) = m - 1$.

Los algoritmos de enumeración de tuplas descritos en [2] resuelven un problema similar, como se observa en las figuras 2.7 y 2.10, el orden de las n-tuplas se establece a partir de diagonales sobre cuadrículas n-dimensionales, de tal forma que los componentes de las tuplas en la m-ésima diagonal suman m.

Sea $\mathcal{T}_n(m)$ el conjunto de n-tuplas tal que sus componentes suman m, se define $\mathrm{count}_{\mathbf{A}\times\mathbf{A}}$ como

$$\mathrm{count}_{\mathbf{A}\times\mathbf{A}}(m) = \sum_{(m_1,m_2)\in\mathcal{T}_2(m)} \mathrm{count}_{\mathbf{A}}(m_1) \times \mathrm{count}_{\mathbf{A}}(m_2) \tag{2.17}$$

A partir de estas definiciones se precisa el conteo de expresiones aritméticas de tamaño m como

$$\begin{aligned} \mathrm{count}_{\mathbf{A}}(0) &= 0 \\ \mathrm{count}_{\mathbf{A}}(m+1) &= \mathrm{count}_{\mathbf{N}}(m+1) \\ &\quad + \mathrm{count}_{\mathbf{X}}(m+1) \\ &\quad + 3 \times \mathrm{count}_{\mathbf{A}\times\mathbf{A}}(m) \end{aligned} \tag{2.18}$$

Para definir el conteo de expresiones booleanas por tamaño se utilizan las técnicas anteriores. Las hojas true y false solo se cuentan para tamaño $m = 1$, las negaciones se cuentan de forma similar a las localidades de memoria y las operaciones binarias de forma similar a las operaciones aritméticas. En esta definición se simplifican algunos cálculos considerando que no existen expresiones aritméticas o booleanas de tamaño cero, por lo que no existen árboles cuyos hijos son expresiones aritméticas o booleanas de tamaño uno.

$$\begin{aligned} \mathrm{count}_{\mathbf{B}}(0) &= 0 \\ \mathrm{count}_{\mathbf{B}}(1) &= 2 && (\mathsf{true}, \mathsf{false}) \\ \mathrm{count}_{\mathbf{A}}(m) &= 2 \times \mathrm{count}_{\mathbf{A}\times\mathbf{A}}(m-1) && (\mathsf{equal}, \mathsf{less}) \\ &\quad + 2 \times \mathrm{count}_{\mathbf{B}\times\mathbf{B}}(m-1) && (\mathsf{and}, \mathsf{or}) \\ &\quad + \mathrm{count}_{\mathbf{B}}(m-1) && (\mathsf{not}) \end{aligned}$$

Donde $\mathrm{count}_{\mathbf{B}\times\mathbf{B}}$ se define de forma similar a $\mathrm{count}_{\mathbf{A}\times\mathbf{A}}$:

$$\mathrm{count}_{\mathbf{B}\times\mathbf{B}}(m) = \sum_{(m_1,m_2)\in\mathcal{T}_2(m)} \mathrm{count}_{\mathbf{B}}(m_1) \times \mathrm{count}_{\mathbf{B}}(m_2)$$

Finalmente, el conteo de programas por tamaño se define siguiendo la estructura de las producciones como con las anteriores categorías sintácticas.

$$\begin{aligned} \mathrm{count}_{\mathbf{P}}(0) &= 0 \\ \mathrm{count}_{\mathbf{P}}(1) &= 1 \\ \mathrm{count}_{\mathbf{P}}(m) &= \mathrm{count}_{\mathbf{X}\times\mathbf{A}}(m-1) && (\mathsf{set}) \\ &\quad + \mathrm{count}_{\mathbf{B}\times\mathbf{P}}(m-1) && (\mathsf{while}) \\ &\quad + \mathrm{count}_{\mathbf{P}\times\mathbf{P}}(m-1) && (\mathsf{conc}) \\ &\quad + \mathrm{count}_{\mathbf{B}\times\mathbf{P}\times\mathbf{P}}(m-1) && (\mathsf{if}) \end{aligned}$$

Donde se define

$$\begin{aligned}
\mathrm{count}_{\mathbf{X}\times\mathbf{A}}(m) &= \sum_{(m_1,m_2)\in\mathcal{T}_2(m)} \mathrm{count}_{\mathbf{X}}(m_1)\times\mathrm{count}_{\mathbf{A}}(m_2)\\
\mathrm{count}_{\mathbf{B}\times\mathbf{P}}(m) &= \sum_{(m_1,m_2)\in\mathcal{T}_2(m)} \mathrm{count}_{\mathbf{B}}(m_1)\times\mathrm{count}_{\mathbf{P}}(m_2)\\
\mathrm{count}_{\mathbf{P}\times\mathbf{P}}(m) &= \sum_{(m_1,m_2)\in\mathcal{T}_2(m)} \mathrm{count}_{\mathbf{P}}(m_1)\times\mathrm{count}_{\mathbf{P}}(m_2)\\
\mathrm{count}_{\mathbf{B}\times\mathbf{P}\times\mathbf{P}}(m) &= \sum_{(m_1,m_2,m_3)\in\mathcal{T}_3(m)} \mathrm{count}_{\mathbf{B}}(m_1)\times\mathrm{count}_{\mathbf{P}}(m_2)\times\mathrm{count}_{\mathbf{P}}(m_3)
\end{aligned}$$

La tabla 2.4 muestra la cantidad de programas IMP para tamaños del cero al diez, junto con la suma acumulada de los conteos.

m	$\mathrm{count}_{\mathbf{P}}(m)$	$\sum_{i=0}^{m}\mathrm{count}_{\mathbf{P}}(i)$
0	0	0
1	1	1
2	0	1
3	3	4
4	4	8
5	324	332
6	6270	6602
7	109,811	116,413
8	1,726,376	1,842,789
9	26,271,032	28,113,821
10	392,978,090	421,091,911

Tabla 2.4: Tamaños de programas IMP.

Al implementar programas computacionales para estos conteos es importante realizar algunas consideraciones que pueden eficientar los cálculos, en particular para evitar cálculos redundantes. Por ejemplo, para calcular la cantidad de programas de tamaños entre cero y m de menor a mayor, el conteo de concatenaciones de cada tamaño repite el cálculo con todos los tamaños menores.

Para evitar estas redundancias se propone utilizar la técnica de *memoización* en cada función, de tal forma que una vez que el conteo se haya calculado, su valor es almacenado para ser utilizado como valor de retorno en futuras invocaciones

de la función.

Otra forma de reducir la cantidad de cálculos realizados es utilizar información de las estructuras sintácticas y la simetría de las sumatorias $\text{count}_{\mathbf{A}\times\mathbf{A}}$, $\text{count}_{\mathbf{B}\times\mathbf{B}}$ y $\text{count}_{\mathbf{P}\times\mathbf{P}}$. En particular las parejas de tamaño $(0, m)$ y $(m, 0)$ pueden ser descartadas ya que no aportan valor a la sumatoria debido a que no existen expresiones de tamaño cero, por otro lado, la segunda mitad de la sumatoria es equivalente a la primera por lo que se puede utilizar una multiplicación por dos en lugar de calcular la mitad de la sumatoria.

Por ejemplo, al considerar $\text{count}_{\mathbf{A}\times\mathbf{A}}(3)$ se tiene que

$$\begin{aligned}
\text{count}_{\mathbf{A}\times\mathbf{A}}(3) &= \sum_{(m_1,m_2)\in\mathcal{T}_2(3)} \text{count}_{\mathbf{A}}(m_1) \times \text{count}_{\mathbf{A}}(m_2) \\
&= \text{count}_{\mathbf{A}}(0) \times \text{count}_{\mathbf{A}}(3) + \text{count}_{\mathbf{A}}(1) \times \text{count}_{\mathbf{A}}(2) \\
&+ \text{count}_{\mathbf{A}}(2) \times \text{count}_{\mathbf{A}}(1) + \text{count}_{\mathbf{A}}(3) \times \text{count}_{\mathbf{A}}(0) \\
&= 0 \times \text{count}_{\mathbf{A}}(3) + \text{count}_{\mathbf{A}}(1) \times \text{count}_{\mathbf{A}}(2) \\
&+ \text{count}_{\mathbf{A}}(2) \times \text{count}_{\mathbf{A}}(1) + \text{count}_{\mathbf{A}}(3) \times 0 \\
&= \text{count}_{\mathbf{A}}(1) \times \text{count}_{\mathbf{A}}(2) + \text{count}_{\mathbf{A}}(2) \times \text{count}_{\mathbf{A}}(1) \\
&= 2 \times \text{count}_{\mathbf{A}}(1) \times \text{count}_{\mathbf{A}}(2)
\end{aligned}$$

Selección de programas

Una vez definido el conteo de expresiones del lenguaje, se presenta el problema de explorar el espacio de programas en función de su tamaño. Para resolver este problema se propone definir funciones $\text{sunrank}_{\mathbf{C}}(m, k)$ que construyen el k-ésimo objeto de tamaño m derivado de $\mathbf{C}$.

La función $\text{sunrank}_{\mathbf{C}}$ es similar a $\text{unrank}_{\mathbf{C}}$ ya que relacionan posiciones a objetos, sin embargo cuando se filtra del espacio de expresiones únicamente aquellas que tienen tamaño m, las posiciones k para las cuales $\text{sunrank}_{\mathbf{C}}$ es definida son aquellas que satisfacen $0 \leq k < \text{count}_{\mathbf{C}}(m)$.

Al igual que en la construcción de enumeraciones, de la métrica de tamaños y del conteo, la *selección de programas* por tamaños $\text{sunrank}_{\mathbf{P}}$ se define a partir de las categorías sintácticas más simples.

Para construir $\text{sunrank}_{\mathbf{N}}$ se considera utilizar la posición k como el desplaza-

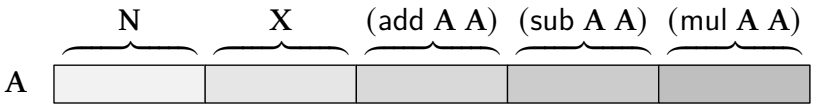
miento entre el primer natural con m dígitos y el k-ésimo natural de m dígitos.

$$\text{sunrank}_{\mathbf{N}}(1, k) = k$$
$$\text{sunrank}_{\mathbf{N}}(m, k) = k + 10^{m-1}$$

Para construir la función $\text{sunrank}_{\mathbf{X}}$ se parte de la observación que existen tantas localidades de tamaño m como naturales de tamaño $m - 1$. Se utiliza el orden establecido por $\text{sunrank}_{\mathbf{N}}$ para determinar el orden de las localidades.

$$\text{sunrank}_{\mathbf{X}}(m, k) = (\mathsf{loc}\ \ \text{sunrank}_{\mathbf{N}}(m - 1, k))$$

Ahora se aborda el problema de construir $\text{sunrank}_{\mathbf{A}}$. Se conceptualiza el orden de las expresiones de tamaño m como los objetos de tamaño m derivados de cada producción desde $\mathbf{A}(0)$ hasta $\mathbf{A}(4)$ como se muestra en la siguiente figura.



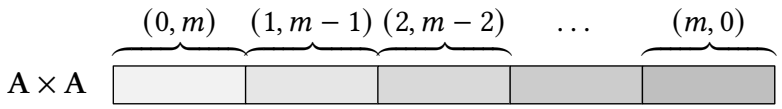
El algoritmo para construir la k-ésima expresión aritmética de tamaño m consiste en verificar a qué producción corresponde la posición k. Se utiliza la variable δ para almacenar la cantidad de objetos de tamaño m de la producción actual, cuando $k < \delta$ entonces k corresponde a la producción actual, de lo contrario se actualiza $k \leftarrow k - \delta$ y se verifica si esta posición corresponde a la siguiente producción.

En el siguiente algoritmo se considera que $\text{sunrank}_{\mathbf{A}\times\mathbf{A}}$ regresa una lista con dos expresiones aritméticas, la operación $(\mathsf{R}\ .\ \ell)$ denota una lista cuyo primer elemento es R y donde el resto de los elementos es la lista ℓ. De acuerdo a la estructura de los árboles de sintaxis utilizada en este trabajo, la lista resultante es equivalente a un árbol de sintaxis con raíz R e hijos ℓ.

$\text{sunrank}_{\mathbf{A}}(m, k) =$
  $\delta \leftarrow \text{count}_{\mathbf{N}}(m)$
  if $k < \delta$ **return** $\text{sunrank}_{\mathbf{N}}(m, k)$
  $k \leftarrow k - \delta$; $\delta \leftarrow \text{count}_{\mathbf{X}}(m)$
  if $k < \delta$ **return** $\text{sunrank}_{\mathbf{X}}(m, k)$
  $k \leftarrow k - \delta$; $\delta \leftarrow \text{count}_{\mathbf{A}\times\mathbf{A}}(m - 1)$
  if $k < \delta$ **return** $(\mathsf{add}\ .\ \text{sunrank}_{\mathbf{A}\times\mathbf{A}}(m - 1, k))$
  $k \leftarrow k - \delta$

if $k < \delta$ **return** (sub . $\text{sunrank}_{\mathbf{A}\times\mathbf{A}}(m-1, k)$)
$k \leftarrow k - \delta$
if $k < \delta$ **return** (mul . $\text{sunrank}_{\mathbf{A}\times\mathbf{A}}(m-1, k)$)

La función $\text{sunrank}_{\mathbf{A}\times\mathbf{A}}(m, k)$ construye la k-ésima pareja de expresiones booleanas donde la suma de sus tamaños es m. Para ordenar estas parejas se considera el orden de las parejas de naturales (m_1, m_2) tales que $m_1 + m_2 = m$ en el orden establecido por la enumeración de duplas [2].



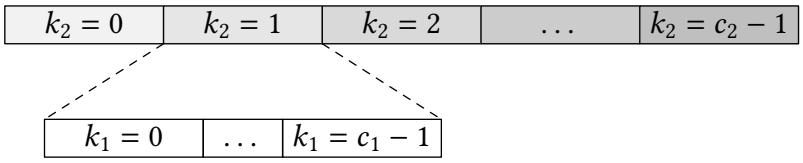
Cada pareja (m_1, m_2) representa las parejas de expresiones aritméticas a_1 de tamaño m_1 y a_2 de tamaño m_2. Se utiliza la estrategia de verificar a qué combinación de tamaños corresponde la posición k utilizando $\delta = \text{count}_{\mathbf{A}}(m_1) \times \text{count}_{\mathbf{A}}(m_2)$ y actualizando $k \leftarrow k - \delta$ cuando $k \geq \delta$.

Una vez que se determina los tamaños (m_1, m_2) correspondientes a la pareja de expresiones aritméticas en la posición k se determina qué posición k_1 corresponde a a_1 y qué posición k_2 corresponde a a_2.

Sea $c_1 = \text{count}_{\mathbf{A}}(m_1)$ y $c_2 = \text{count}_{\mathbf{A}}(m_2)$, la cantidad de parejas (a_1, a_2) de tamaños (m_1, m_2) es $c_1 \times c_2$. El orden de posiciones (k_1, k_2) se determina a partir de la división euclidiana (2.6) considerando k el dividendo y c_1 el divisor, de tal forma que

$$k_1 = k \bmod c_1$$
$$k_2 = k \operatorname{div} c_1$$

En la siguiente figura se muestra un esquema de este ordenamiento.

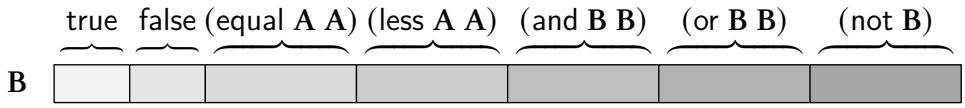


Partiendo de la propiedad $0 \leq k < c_1 \times c_2$ se tiene que k_1 toma valores tales que $0 \leq k_1 < c_1$, por lo que k_2 toma valores tales que $0 \leq k_2 < c_2$.

$\text{sunrank}_{\mathbf{A}\times\mathbf{A}}(m, k) =$
for (m_1, m_2) **in** $\mathcal{T}_2(m)$

$c_1 \leftarrow \text{count}_{\mathbf{A}}(m_1)$; $c_2 \leftarrow \text{count}_{\mathbf{A}}(m_2)$; $\delta \leftarrow c_1 \times c_2$
if $k < \delta$ **then**
$k_1 \leftarrow k \bmod c_1$; $k_2 \leftarrow k \operatorname{div} c_1$
return $(\text{sunrank}_{\mathbf{A}}(m_1, k_1)\ \ \text{sunrank}_{\mathbf{A}}(m_2, k_2))$
else
$k \leftarrow k - \delta$

La definición de $\text{sunrank}_{\mathbf{B}}$ utiliza la misma estrategia que $\text{sunrank}_{\mathbf{A}}$, siendo la diferencia principal el manejo de la construcción de hojas true, false las cuales se abordan como un caso particular cuando $m = 1$.



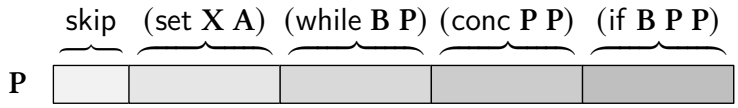
$\text{sunrank}_{\mathbf{B}}(m, k) =$
if $m = 1$ **then**
if $k = 0$ **return** true
if $k = 1$ **return** false
$k \leftarrow k - 2$
$\delta \leftarrow \text{count}_{\mathbf{A}\times\mathbf{A}}(m-1)$
if $k < \delta$ **return** $(\text{equal}\ .\ \text{sunrank}_{\mathbf{A}\times\mathbf{A}}(m-1, k))$
$k \leftarrow k - \delta$
if $k < \delta$ **return** $(\text{less}\ .\ \text{sunrank}_{\mathbf{A}\times\mathbf{A}}(m-1, k))$
$k \leftarrow k - \delta$; $\delta \leftarrow \text{count}_{\mathbf{B}\times\mathbf{B}}(m-1)$
if $k < \delta$ **return** $(\text{and}\ .\ \text{sunrank}_{\mathbf{B}\times\mathbf{B}}(m-1, k))$
$k \leftarrow k - \delta$
if $k < \delta$ **return** $(\text{or}\ .\ \text{sunrank}_{\mathbf{B}\times\mathbf{B}}(m-1, k))$
$k \leftarrow k - \delta$; $\delta \leftarrow \text{count}_{\mathbf{B}}(m-1)$
if $k < \delta$ **return** $(\text{not}\ \ \text{sunrank}_{\mathbf{B}}(m-1, k))$

La definición de $\text{sunrank}_{\mathbf{B}\times\mathbf{B}}$ es similar a $\text{sunrank}_{\mathbf{B}\times\mathbf{B}}$. En general, para toda pareja de categorías sintácticas $\mathbf{C_1}$ y $\mathbf{C_2}$ se define

$\text{sunrank}_{\mathbf{C}_1\times\mathbf{C}_2}(m, k) =$
for (m_1, m_2) **in** $\mathcal{T}_2(m)$
$c_1 \leftarrow \text{count}_{\mathbf{C}_1}(m_1)$; $c_2 \leftarrow \text{count}_{\mathbf{C}_2}(m_2)$
$\delta \leftarrow c_1 \times c_2$
if $k < \delta$ **then**

$k_1 \leftarrow k \bmod c_1; k_2 \leftarrow k \operatorname{div} c_1$
return $(\text{sunrank}_{\mathbf{C}_1}(m_1, k_1)\ \ \text{sunrank}_{\mathbf{C}_2}(m_2, k_2))$
else
$k \leftarrow k - \delta$

La definición de $\text{sunrank}_{\mathbf{P}}$ aborda los mismos casos que $\text{sunrank}_{\mathbf{B}}$, considerando skip como caso particular de $m = 1$ y siguiendo el orden del resto de las producciones en otros casos, como se muestra en la siguiente figura.



$\text{sunrank}_{\mathbf{P}}(m, k) =$
if $m = 1$ **then**
if $k = 0$ **return** skip
$k \leftarrow k - 1$
$\delta \leftarrow \text{count}_{\mathbf{X}\times\mathbf{A}}(m-1)$
if $k < \delta$ **return** $(\text{set} : \text{sunrank}_{\mathbf{X}\times\mathbf{A}}(m-1, k))$
$k \leftarrow k - \delta; \delta \leftarrow \text{count}_{\mathbf{B}\times\mathbf{P}}(m-1)$
if $k < \delta$ **return** $(\text{while} : \text{sunrank}_{\mathbf{B}\times\mathbf{P}}(m-1, k))$
$k \leftarrow k - \delta; \delta \leftarrow \text{count}_{\mathbf{P}\times\mathbf{P}}(m-1)$
if $k < \delta$ **return** $(\text{conc} : \text{sunrank}_{\mathbf{P}\times\mathbf{P}}(m-1, k))$
$k \leftarrow k - \delta; \delta \leftarrow \text{count}_{\mathbf{B}\times\mathbf{P}\times\mathbf{P}}(m-1)$
if $k < \delta$ **return** $(\text{if} : \text{sunrank}_{\mathbf{B}\times\mathbf{P}\times\mathbf{P}}(m-1, k))$

La definición de $\text{sunrank}_{\mathbf{B}\times\mathbf{P}\times\mathbf{P}}$ se utiliza la misma técnica que en el algoritmo con dos categorías sintácticas.

$\text{sunrank}_{\mathbf{B}\times\mathbf{P}\times\mathbf{P}}(m, k) =$
for (m_1, m_2, m_3) **such that** $m_1 + m_2 + m_3 = m$
$c_1 \leftarrow \text{count}_{\mathbf{B}}(m_1); c_2 \leftarrow \text{count}_{\mathbf{P}}(m_2); c_3 \leftarrow \text{count}_{\mathbf{P}}(m_3)$
$\delta \leftarrow c_1 \times c_2 \times c_3$
if $k < \delta$ **then**
$k_1 \leftarrow k \bmod c_1; k \leftarrow k \operatorname{div} c_1$
$k_2 \leftarrow k \bmod c_2; k_3 \leftarrow k \operatorname{div} c_2$
return $(\text{sunrank}_{\mathbf{B}}(m_1, k_1)\ \ \text{sunrank}_{\mathbf{P}}(m_2, k_2)\ \ \text{sunrank}_{\mathbf{P}}(m_3, k_3))$
else
$k \leftarrow k - \delta$

Estas definiciones de sunrank permiten explorar el espacio de programas IMP de dos formas adicionales a la enumeración.

Primero se considera la *selección aleatoria* por tamaño. A partir de un generador de números aleatorios $R(n) \in [0, n)$ con distribución uniforme se obtiene una posición k con la cuál construir el k-ésimo programa de tamaño m. Este procedimiento permite muestrear el espacio de programas de tamaño fijo.

$$\text{srandom}_{\mathbf{P}}(m) = \text{sunrank}_{\mathbf{P}}(m, R(\text{count}_{\mathbf{P}}(m)))$$

La segunda forma de explorar el espacio de programas es considerar los programas $\text{sunrank}_{\mathbf{P}}(m, k)$ para toda m y k ordenados de menor a mayor tamaño. Se define la función $\text{sortedunrank}_{\mathbf{P}}$ definida sobre todos los naturales k.

$\text{sortedunrank}_{\mathbf{P}}(k) =$
  for m **from** 0
    $\delta \leftarrow \text{count}_{\mathbf{P}}(m)$
    if $k < \delta$ **return** $\text{sunrank}_{\mathbf{P}}(m, k)$
    $k \leftarrow k - \delta$

2.3 Análisis del lenguaje y su enumeración

A partir de los mecanismos de enumeración y las propiedades sintácticas de los programas tales como su tamaño o su tipo de expresión, se pueden utilizar herramientas estadísticas para analizar el lenguaje y la enumeración.

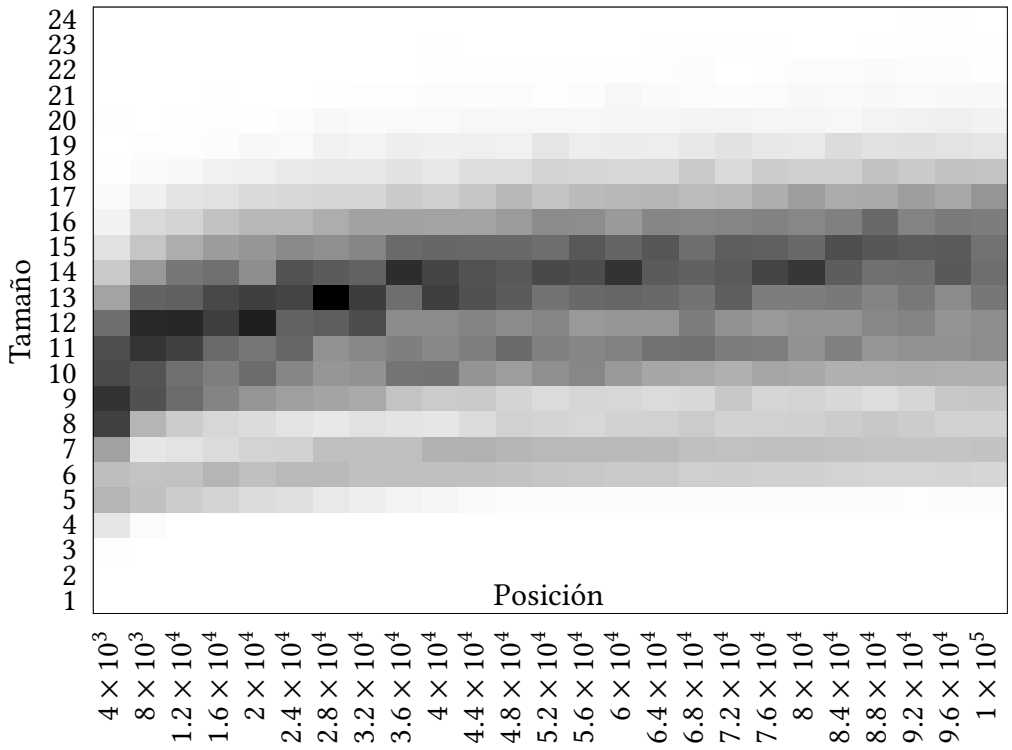
Se considera como primer ejemplo la figura 2.12, donde se muestra cómo se distribuyen los primeros 10^5 programas de acuerdo con su posición en el eje horizontal y su tamaño en el eje vertical. Las posiciones son agrupadas en 25 bloques de tamaño 4×10^3.

Las celdas con color más obscuro indican que hay una mayor cantidad de programas en el bloque de posiciones y tamaño correspondientes, mientras que un color totalmente blanco indica que hay cero programas para el bloque de posiciones y tamaño correspondientes.

En esta gráfica, la distribución de los tamaños parece caracterizarse por dos franjas crecientes, una más ancha donde se concentra la mayor cantidad de programas y otra mas delgada en la parte inferior.

Estudiar la distribución los tamaños de programas para un segmento inicial de la enumeración puede permitir estimar cómo se distribuyen estos tamaños para posiciones más grandes.

Figura 2.12: Tamaños de los primeros 10^5 programas IMP.



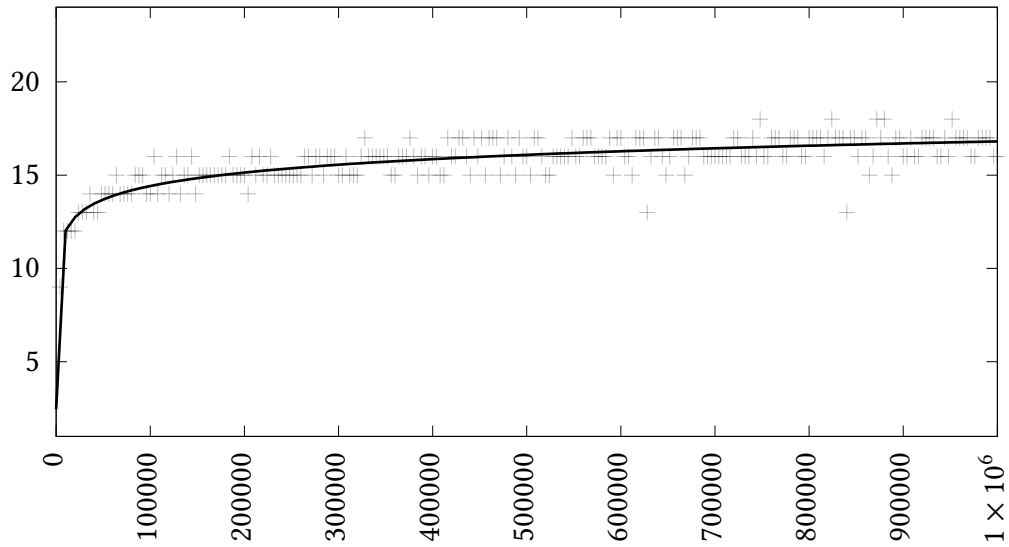
En la figura 2.13 se grafican puntualmente los tamaños con mayor cantidad de programas con el mismo tamaño de bloque pero para el primer millón de programas. A partir de un ajuste logaritmico de estos datos puede conjeturarse que una extrapolación es un buen estimado de los tamaños que contienen la mayor cantidad de programas para bloques después del primer millón de posiciones.

Una forma alternativa de estudiar estos datos es haciendo énfasis en un renglón de la figura 2.12, es decir, analizar la distribución de posiciones para programas de un cierto tamaño.

Como complemento a la tabla 2.4, en la gráfica de la figura 2.14 se muestra cómo crece la cantidad de programas en función a su tamaño. Ya que el comportamiento de la curva es lineal en representación semilogarítmica, el crecimiento de los datos es exponencial.

Este crecimiento de la cantidad de programas por tamaño puede hacer que sea computacionalmente costoso analizar cada programa, por ejemplo, al considerar

Figura 2.13: Tamaños con más programas para posiciones $k < 10^6$.



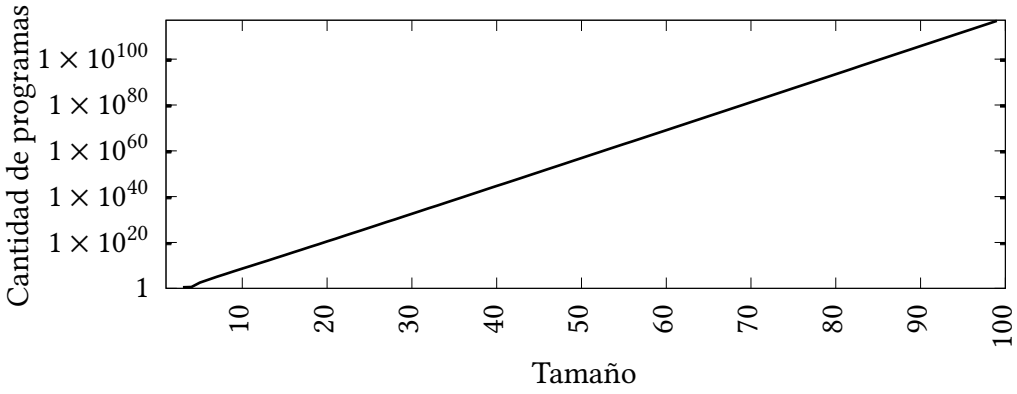
el tamaño $m = 10$ se calcula que existen más de 10^{100} programas. Para poder hacer un análisis aproximado de la distribución de posiciones, se utiliza la selección aleatoria de programas para muestrear el espacio de programas de tamaño m en lugar de calcular la posición de todos los programas de tamaño m.

En la figura 2.15 se muestra la distribución de posiciones para los programas de tamaños 6, 7 y 8 con diagramas de cajas utilizando una escala logarítmica en el eje vertical. Estas posiciones se obtienen al muestrear 10^3 programas para cada tamaño de forma aleatoria y calculando su rank en la enumeración. Como se observa en la figura, el tamaño del cuartil superior aparenta crecer más rápido que el cuartil inferior conforme aumenta el tamaño de los programas.

Finalmente, también se pueden utilizar otras funciones de conteo para estudiar la distribución de posiciones a un mayor nivel de detalle. En la figura 2.16 se muestra una gráfica similar a la anterior pero para un muestreo de 10^4 programas de tamaño 8 clasificados por el tipo de expresión: asignaciones set, iteraciones while, concatenaciones conc y condicionales if.

En esta figura se muestra como las asignaciones de memoria de tamaño 8 se encuentran ubicadas en un rango de posiciones muy pequeño a comparación del resto de las expresiones del mismo tamaño. Por otro lado, las condicionales if ocupan el rango de posiciones más amplio de todas las expresiones a pesar de que

Figura 2.14: Cantidad de programas por tamaño.



las condicionales conforman al rededor del 5% de todos los programas.

En la tabla 2.5 se presenta la cantidad de programas muestreados por cada tipo de expresión así como el total de programas utilizando las funciones de conteo. Estas cifras son acompañadas del porcentaje correspondiente a la muestra y al total. Como se observa en la tabla, las proporciones de cada tipo de programa se preservan en la muestra en comparación con el total.

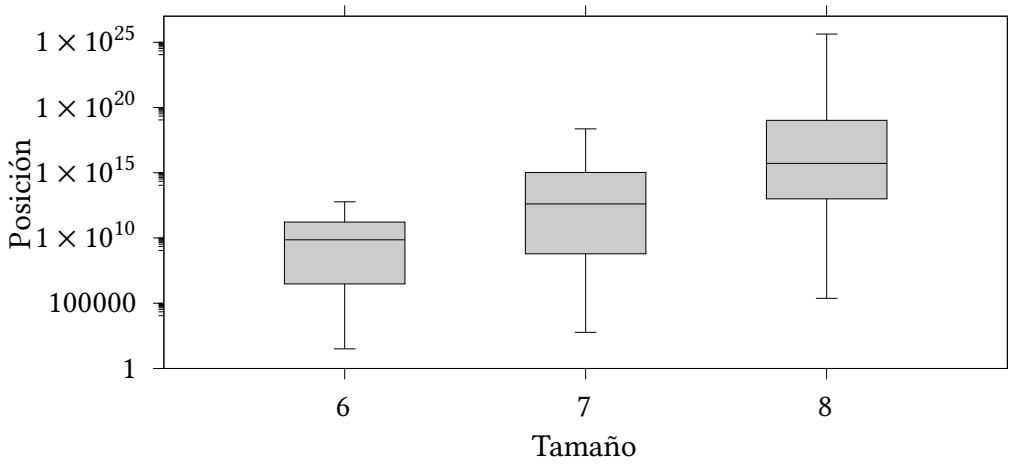
Tipo	Muestra	(%)	Total	(%)
set	1996	(19.96%)	342,000	(19.81%)
while	7476	(74.76%)	1,291,708	(74.82%)
conc	76	(0.76%)	12,564	(0.73%)
if	452	(4.52%)	80,104	(4.64%)
Total	10,000		1,726,376	

Tabla 2.5: Cantidad y proporción de programas muestrados por tipo de expresión.

El uso de herramientas estadísticas para estudiar el lenguaje IMP y sus enumeraciones asociadas permite entender desde una perspectiva general cómo la enumeración ordena los programas y por consecuente, cómo la enumeración permite explorar el espacio de programas.

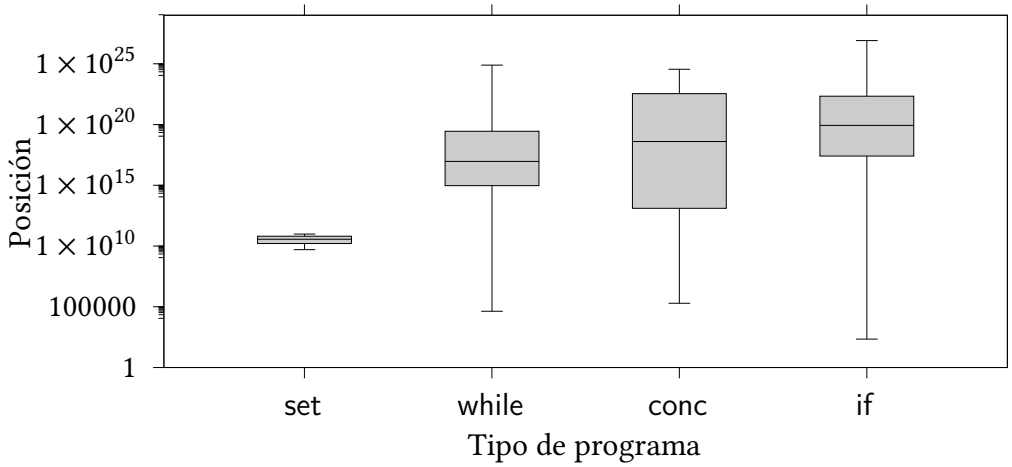
Los resultados descritos en esta sección contemplan únicamente la enumeración y las propiedades sintácticas del lenguaje. Sin embargo, al considerar la semántica de IMP descrita en la sección 1.2, se pueden analizar otros aspectos del lenguaje.

Figura 2.15: Distribución de posiciones por tamaño de programa.



El trabajo descrito en [28] presenta la implementación de un intérprete para IMP y como parte del proyecto del grupo de trabajo [15] se propone utilizar este intérprete y la enumeración de programas para analizar la detención de programas.

Figura 2.16: Distribución de posiciones por tipo de expresión ($m = 8$).



Capítulo 3

Combinadores de enumeraciones

El objeto de estudio del capítulo anterior son las estructuras sintácticas del lenguaje IMP y un conjunto de técnicas que permiten enumerar estos objetos. Las funciones de enumeración definidas para los distintos tipos de expresiones del lenguaje son similares entre sí, sin embargo el proceso para adaptar estos algoritmos a otros lenguajes puede ser tedioso.

El objeto de estudio de este capítulo son las enumeraciones vistas como objetos computacionales abstractos, por lo que no tienen una estructura particular ni dependen de especificaciones del lenguaje. Las enumeraciones se definen en términos de un conjunto de propiedades y operaciones correspondientes a un *protocolo de enumeración*.

Primero se describe a detalle el protocolo, posteriormente se definen un conjunto de enumeraciones simples llamadas *enumeraciones primitivas*. Finalmente se describen una serie de mecanismos para combinar y construir nuevas enumeraciones, estos son operadores llamados *combinadores de enumeraciones*.

Las enumeraciones que pueden ser construidas por estos combinadores son al menos tan complejas como la enumeración de programas IMP.

3.1 Protocolo de enumeración

Una enumeración es una biyección entre un conjunto de objetos X y los naturales en el rango $[0, |X|)$ denominados *posiciones*. Sea E una enumeración del conjunto de objetos X, se denota $|E| = |X|$ y $x \in E$ cuando $x \in X$. Esta notación permite trabajar con el conjunto X de forma implícita.

Se propone clasificar las enumeraciones a partir de su cardinalidad. Cuando

una enumeración E contiene tantos objetos como naturales se dice que es una enumeración *no-acotada* y se denota $|E| = \infty$, por otro lado, cuando E contiene una cantidad fija de objetos se dice que es una enumeración *acotada* por lo que existe un natural n tal que $|E| = n$.

Se denotan las enumeraciones en general con la letra mayúscula E, posiblemente utilizando subíndices para distinguir dos o más enumeraciones diferentes. Las enumeraciones acotadas se denotan con la letra B (del inglés *bounded*) y las no-acotadas con la letra U (del inglés *unbounded*).

En este trabajo, se considera que un objeto E es una enumeración cuando tiene definidas cuatro operaciones particulares. Primero, su cardinalidad $|E|$ debe poder ser calculada y corresponder a un natural o bien al símbolo ∞ que denota un objeto mayor a cualquier natural. Segundo, para cualquier objeto x se debe poder determinar si $x \in E$, es decir, si es enumerado por E o pertenece a E. Finalmente, la biyección entre posiciones y objetos enumerados debe definirse a partir de operaciones rank_E y unrank_E.

Estas cuatro operaciones se relacionan entre si a partir de tres propiedades, llamadas en este trabajo *propiedades de enumerabilidad* (abreviadas como pe) descritas a continuación.

Propiedad 3.1. *Para todo $x \in E$ existe una posición k tal que si $\text{rank}_E(x) = k$, entonces $0 \leq k < |E|$.*

Propiedad 3.2. *Para toda posición k donde $0 \leq k < |E|$, existe un objeto x tal que si $\text{unrank}_E(k) = x$, entonces $x \in E$.*

Propiedad 3.3. *Para todo objeto x y posición k, $\text{rank}_E(x)$ y $\text{unrank}_E(k)$ son inversas entre sí, es decir*

$$\text{rank}_E(x) = k \iff \text{unrank}_E(k) = x$$

Las operaciones de cardinalidad, pertenencia, rank y unrank, junto con pe 3.1, pe 3.2 y pe 3.3, conforman lo que se llama *protocolo de enumeración*. Independientemente de la estructura computacional de un objeto cualquiera, si satisface el protocolo, entonces es considerado una enumeración.

Como consecuencia de las propiedades de enumerabilidad, todas las enumeraciones también satisfacen las siguientes propiedades.

Proposición. *La operación rank_E es inyectiva, es decir, para cualesquiera objetos x_1 y x_2 en E, si $\text{rank}_E(x_1) = \text{rank}_E(x_2)$ entonces $x_1 = x_2$.*

Prueba. Sea $k = \text{rank}_E(x_1)$, por el antecedente de la implicación $\text{rank}_E(x_2) = k$. Por pe 3.3 se tiene que $\text{unrank}_E(k) = x_1$ y $\text{unrank}_E(k) = x_2$, por lo que $x_1 = x_2$. □

Proposición. *La operación* unrank_E *es inyectiva, es decir, para cualesquiera posiciones* k_1 *y* k_2 *en el intervalo* $[0, |E|)$*, si* $\text{unrank}_E(k_1) = \text{unrank}_E(k_2)$ *entonces* $k_1 = k_2$*.*

Prueba. Sea $x = \text{unrank}_E(k_1)$, por el antecedente de la implicación se tiene que $\text{unrank}_E(k_2) = x$. Por pe 3.3 se tiene que $\text{rank}_E(x) = k_1$ y $\text{rank}_E(x) = k_2$, por lo tanto $k_1 = k_2$. □

Proposición. *La operación* rank_E *es sobreyectiva, es decir, para toda posición* $k \in [0, |E|)$ *existe un objeto* $x \in E$ *tal que* $\text{rank}(x) = k$*.*

Prueba. Por pe 3.2 existe un objeto $x = \text{unrank}_E(k) \in E$ y por pe 3.3 $\text{rank}_E(x) = k$. □

Proposición. *La operación* unrank_E *es sobreyectiva, es decir, para todo objeto* $x \in E$ *existe una posición* $k \in [0, |E|)$ *tal que* $\text{unrank}_E(k) = x$*.*

Prueba. Por pe 3.1 existe un objeto $k = \text{rank}_E(x) \in [0, |E|)$ y por pe 3.3 $\text{unrank}_E(k) = x$. □

A partir de un conjunto de objetos X se pueden definir distintos ordenes para especificar una enumeración. Por ejemplo, cuando $|X| = n$ existen $n!$ distintos ordenes y para cada uno de estos ordenes se puede definir rank y unrank con diversos métodos y algoritmos. En este trabajo se definen enumeraciones a partir de la construcción de rank y unrank por lo que el orden de los objetos enumerados es consecuencia de la enumeración.

En ocasiones es útil definir el algoritmo rank en términos de unrank o viceversa, de tal manera que pe 3.3 se satisface por construcción, para eso se utilizan las igualdades

$$\text{rank}_E(\text{unrank}_E(k)) = k$$
$$\text{unrank}_E(\text{rank}_E(x)) = x.$$

3.2 Enumeraciones triviales

En esta sección se definen algunas enumeraciones primitivas que, a pesar de ser triviales, son útiles para entender el proceso de construcción de enumeraciones que satisfacen el protocolo y además son utilizadas en enumeraciones más complejas.

Se considera primero la *enumeración vacía* denotada $\emptyset$, la cual corresponde a la enumeración del conjunto vacío. Las operaciones de cardinalidad y pertenencia son triviales de definir, ya que $|\emptyset| = 0$ y para todo objeto x se tiene que $x \notin \emptyset$.

Es indistinta la definición de las operaciones $\text{rank}_\emptyset$ y $\text{unrank}_\emptyset$, ya que las propiedades del protocolo de enumeración se satisfacen por vacuidad, es decir, no existen objetos $x \in \emptyset$ ni posiciones $0 \leq k < 0$, por lo tanto $\text{rank}_\emptyset$ no es definida para ningún objeto y $\text{unrank}_\emptyset$ no es definida para ninguna posición.

La enumeración vacía es utilizada en este trabajo como la enumeración resultante de casos particulares de enumeraciones más complejas.

En ocasiones es conveniente definir una familia de enumeraciones y mostrar que todos sus elementos satisfacen el protocolo de enumeración. Una de las familias más simples es la que consiste de todas las enumeraciones con un único objeto, llamadas *enumeraciones constantes*.

Para definir esta familia de enumeraciones se utiliza la función $\text{K}(x)$ como mecanismo de construcción, es decir $\text{K}(x)$ es la enumeración tal que $|\,\text{K}(x)| = 1$ y $x \in \text{K}$. La biyección se define de la siguiente manera.

$$\begin{aligned} \text{rank}_{\text{K}(x)}(x) &= 0 \\ \text{unrank}_{\text{K}(x)}(0) &= x \end{aligned}$$

Estas operaciones satisfacen las propiedades pe 3.1 y pe 3.2 por definición, ya que

$$\begin{aligned} 0 \leq \text{rank}_{\text{K}(x)}(x) &< 1 \\ \text{unrank}_{\text{K}(x)}(0) &\in \text{K}(x) \end{aligned}$$

también se satisface pe 3.3 ya que para todo objeto y y posición k, si $\text{rank}_{\text{K}(x)}(y) = k$ entonces $y = x$ y si $\text{unrank}_{\text{K}(x)}(k) = y$ entonces $k = 0$.

$$\begin{aligned} &\text{rank}_{\text{K}(x)}(y) = k \\ &\implies y = x \wedge k = 0 \\ &\implies \text{unrank}_{\text{K}(x)}(0) = x \\ &\implies \text{unrank}_{\text{K}(x)}(k) = y \end{aligned} \qquad \begin{aligned} &\text{unrank}_{\text{K}(x)}(k) = y \\ &\implies k = 0 \wedge y = x \\ &\implies \text{rank}_{\text{K}(x)}(x) = 0 \\ &\implies \text{rank}_{\text{K}(x)}(y) = k \end{aligned}$$

La familia de enumeraciones K contiene algunas enumeraciones del lenguaje IMP, en particular todas las raíces de árboles y hojas de identificadores se representan con las enumeraciones K(skip), K(set), K(while), etc.

3.3 Enumeración de naturales

Algunos conjuntos de objetos son usualmente asociados a cierto orden, tal es el caso de los números naturales, donde su posición en la enumeración corresponde al valor numérico del natural. Se denota la enumeración de naturales como Nat y se definen las operaciones de cardinalidad y pertenencia como

$$|\,\mathrm{Nat}\,| = \infty$$
$$x \in \mathrm{Nat} = x \in \mathbb{N}$$

La definición de las operaciones $\mathrm{rank}_{\mathrm{Nat}}$ y $\mathrm{unrank}_{\mathrm{Nat}}$, es la función identidad definida sobre los naturales, por lo que

$$\mathrm{rank}_{\mathrm{Nat}}(x) = x$$
$$\mathrm{unrank}_{\mathrm{Nat}}(k) = k$$

La enumeración Nat satisface las propiedades del protocolo trivialmente ya que las posiciones y los objetos son el mismo conjunto y se relacionan con la función identidad, la cuál es su propia inversa.

Se considera ahora una familia de enumeraciones para intervalos de naturales. Se denota $\mathrm{Nat}(a, b)$ a la una enumeración que ordena naturales desde a hasta b de acuerdo a las siguientes reglas.

- Si $a \in \mathbb{N}$ y $b = \infty$ la enumeración resultante es no-acotada y asocia posiciones a naturales consecutivos a partir de a de menor a mayor. Como caso particular se tiene que $\mathrm{Nat} = \mathrm{Nat}(0, \infty)$.
- Si $a \in \mathbb{N}$ y $b \in \mathbb{N}$ con $a \leq b$, entonces la enumeración resultante es acotada con cardinalidad $b - a + 1$ y asocia posiciones a naturales $n \in [a, b]$ de menor a mayor. Por ejemplo $\mathrm{Nat}(5, 10)$ enumera en orden 5, 6, 7, 8, 9, 10. Cuando $a = b$ la enumeración resultante es equivalente a $\mathrm{K}(a)$.
- Si $a \in \mathbb{N}$ y $b \in \mathbb{N}$ con $b < a$, entonces la enumeración resultante describe los mismos objetos que $\mathrm{Nat}(b, a)$ pero en orden inverso, comenzando con el máximo a y terminando con el mínimo b. Por ejemplo, $\mathrm{Nat}(10, 5)$ enumera en orden 10, 9, 8, 7, 6, 5.

Las operaciones del protocolo de enumeración se definen en base a estas tres reglas. En particular, se definen las operaciones de cardinalidad y pertenencia de

la siguiente manera.

$$|\operatorname{Nat}(a,b)| = \begin{cases} \infty & b = \infty \\ b - a + 1 & a \leq b \\ a - b + 1 & b < a \end{cases} \tag{3.1}$$

$$x \in \operatorname{Nat}(a,b) = \begin{cases} a \leq x & b = \infty \\ a \leq x \leq b & a \leq b \\ b \leq x \leq a & b < a \end{cases} \tag{3.2}$$

Se definen $\operatorname{rank}_{\operatorname{Nat}(a,b)}$ y $\operatorname{unrank}_{\operatorname{Nat}(a,b)}$ considerando que las primeras dos reglas describen un orden creciente y la tercera un orden decreciente.

$$\operatorname{rank}_{\operatorname{Nat}(a,b)}(x) = \begin{cases} x - a & b = \infty \\ x - a & a \leq b \\ a - x & b < a \end{cases} \tag{3.3}$$

$$\operatorname{unrank}_{\operatorname{Nat}(a,b)}(k) = \begin{cases} k + a & b = \infty \\ k + a & a \leq b \\ a - k & b < a \end{cases} \tag{3.4}$$

Se muestra que $\operatorname{Nat}(a,b)$ es una enumeración cuando $a \in \mathbb{N}$ y $b \in \mathbb{N} \cup \{\infty\}$ ya que satisface las propiedades del protocolo de enumeración.

Proposición (pe 3.1)**.** *Para todo natural* $x \in \operatorname{Nat}(a,b)$ *existe una posición* k *tal que si* $\operatorname{rank}_{\operatorname{Nat}(a,b)}(x) = k$*, entonces* $0 \leq k < |\operatorname{Nat}(a,b)|$.

Prueba. Primero se consideran las primeras dos reglas donde $a \leq b$ y se muestra que se satisfacen las dos desigualdades de forma independiente. Primero, $\operatorname{rank}_{\operatorname{Nat}(a,b)}(x)$ siempre es mayor a cero ya que por (3.2), si $x \in \operatorname{Nat}(a,b)$ entonces $a \leq x$, por lo tanto

$$\begin{aligned} a \leq x &\implies 0 \leq x - a \\ &\implies 0 \leq \operatorname{rank}_{\operatorname{Nat}(a,b)}(x) \end{aligned}$$

Por otro lado, la segunda desigualdad se satisface trivialmente cuando $b = \infty$,

por lo que se considera que $x \leq b < \infty$.

$$\begin{aligned} x \leq b &\implies x - a \leq b - a \\ &\implies x - a < b - a + 1 \\ &\implies \operatorname{rank}_{\operatorname{Nat}(a,b)} < |\operatorname{Nat}(a,b)| \end{aligned}$$

Finalmente, si $b < a$ se tiene que x pertenece a la enumeración cuando $b \leq x < a$, por lo tanto

$$\begin{aligned} b \leq x \leq a &\implies -a \leq -x \leq -b \\ &\implies 0 \leq a - x \leq a - b \\ &\implies 0 \leq \operatorname{rank}_{\operatorname{Nat}(a,b)}(x) < a - b + 1 \\ &\implies 0 \leq \operatorname{rank}_{\operatorname{Nat}(a,b)}(x) < |\operatorname{Nat}(a,b)| \end{aligned}$$

□

Proposición (pe 3.2)**.** *Para toda posición k donde $0 \leq k < |\operatorname{Nat}(a,b)|$, existe un objeto x tal que si $\operatorname{unrank}_{\operatorname{Nat}(a,b)}(k) = x$, entonces $x \in \operatorname{Nat}(a,b)$.*

Prueba. Cuando $b = \infty$ se tiene que $k \in [0, \infty)$, por lo tanto

$$\begin{aligned} 0 \leq k &\implies a \leq k + a \\ &\implies a \leq \operatorname{unrank}_{\operatorname{Nat}(a,\infty)}(k) \\ &\implies \operatorname{unrank}_{\operatorname{Nat}(a,\infty)}(k) \in \operatorname{Nat}(a,\infty) \end{aligned}$$

Cuando $a \leq b < \infty$ se tiene que $0 \leq k < b - a + 1$, por lo tanto

$$\begin{aligned} 0 \leq k < b - a + 1 &\implies 0 \leq k \leq b - a \\ &\implies a \leq k + a \leq b \\ &\implies a \leq \operatorname{unrank}_{\operatorname{Nat}(a,b)}(k) \leq b \\ &\implies \operatorname{unrank}_{\operatorname{Nat}(a,b)}(k) \in \operatorname{Nat}(a,b) \end{aligned}$$

Finalmente, cuando $b < a$ se tiene que $0 \leq k < a - b + 1$, por lo tanto

$$\begin{aligned} 0 \leq k < a - b + 1 &\implies 0 \leq k \leq a - b \\ &\implies b - a \leq -k \leq 0 \\ &\implies b \leq a - k \leq a \\ &\implies b \leq \mathrm{unrank}_{\mathrm{Nat}(a,b)}(k) \leq a \\ &\implies \mathrm{unrank}_{\mathrm{Nat}(a,b)}(k) \in \mathrm{Nat}(a,b) \end{aligned}$$

□

Proposición (pe 3.3)**.** *Para todo objeto x y posición k,* $\mathrm{rank}_{\mathrm{Nat}(a,b)}$ *y* $\mathrm{unrank}_{\mathrm{Nat}(a,b)}$ *son inversas entre sí.*

Prueba. Ya que las definiciones de $\mathrm{rank}_{\mathrm{Nat}(a,b)}$ y $\mathrm{unrank}_{\mathrm{Nat}(a,b)}$ son iguales cuando $b = \infty$ o $a \leq b$ se consideran como un mismo caso.

$$\begin{aligned} \mathrm{rank}_{\mathrm{Nat}(a,b)}(x) = k &\implies x - a = k \\ &\implies x = k + a \\ &\implies x = \mathrm{unrank}_{\mathrm{Nat}(a,b)}(k) \\ \mathrm{unrank}_{\mathrm{Nat}(a,b)}(k) = x &\implies k + a = x \\ &\implies k = x - a \\ &\implies k = \mathrm{rank}_{\mathrm{Nat}(a,b)}(x) \end{aligned}$$

Finalmente, cuando $b < a$ se tiene que

$$\begin{aligned} \mathrm{rank}_{\mathrm{Nat}(a,b)}(x) = k &\implies a - x = k \\ &\implies a - k = x \\ &\implies \mathrm{unrank}_{\mathrm{Nat}(a,b)}(k) = x \\ \mathrm{unrank}_{\mathrm{Nat}(a,b)}(k) = x &\implies a - k = x \\ &\implies a - x = k \\ &\implies \mathrm{rank}_{\mathrm{Nat}(a,b)}(x) = k \end{aligned}$$

□

En el contexto del lenguaje IMP, la enumeración **N** corresponde a $\mathrm{Nat}(0, \infty)$. Para otros lenguajes, cuando a y b son menores a ∞, $\mathrm{Nat}(a, b)$ puede ser utilizada para representar un conjunto finito de valores, por ejemplo, en algunos lenguajes

los numeros 0 y 1 son utilizados para representar falso y verdadero, por lo que $\mathrm{Nat}(0, 1)$ puede corresponder a la enumeración de los valores de verdad.

3.4 Enumeración de cadenas binarias

En esta sección se presenta una enumeración de todas las cadenas binarias, denotada Bits. Tanto la estructura de los objetos como los algoritmos para su enumeración son los más complejos que se abordan en este trabajo.

A pesar de que esta enumeración no es utilizada para construir enumeraciones del lenguaje IMP, se presenta la construcción de Bits ya que se abordan métodos que son adaptados en el resto del capítulo.

Una cadena binaria α es una secuencia finita de bits, es decir, símbolos 0 y 1. Se denota con $|\alpha|$ al tamaño o la cantidad de bits en α y la cadena vacía ε es la única cadena de tamaño cero.

Para construir una cadena binaria a partir de otra se utiliza la operación $\alpha \cdot b$ que resulta en una cadena binaria de tamaño $|\alpha| + 1$ con el bit b concatenado al final de α.

La enumeración Bits es no-acotada ya que existen tantos naturales como tamaños de cadena, por lo que $|\,\mathrm{Bits}\,| = \infty$. La pertenencia de un objeto en la enumeración se define a partir de la estructura de la cadena, $\varepsilon \in \mathrm{Bits}$ y si $\alpha \in \mathrm{Bits}$ y $b \in \{0, 1\}$ entonces $\alpha \cdot b \in \mathrm{Bits}$.

Se busca que el orden de las cadenas binarias inducido por $\mathrm{rank}_{\mathrm{Bits}}$ y $\mathrm{unrank}_{\mathrm{Bits}}$ sea de menor a mayor tamaño. De tal forma que las cadenas de tamaño n preceden a las cadenas de tamaño $n + 1$, siendo ε el primer objeto en Bits.

Antes de abordar la construcción de las funciones $\mathrm{rank}_{\mathrm{Bits}}$ y $\mathrm{unrank}_{\mathrm{Bits}}$ se propone definir como paso intermedio la familia de enumeraciones $\mathrm{Bits}(n)$ que relacionan posiciones a cadenas binarias de tamaño n.

La operación de pertenencia se define para todo natural n considerando la construcción de cadenas binarias mencionada previamente.

$$\begin{gathered} \varepsilon \in \mathrm{Bits}(0) \\ \alpha \cdot b \in \mathrm{Bits}(n+1) = \alpha \in \mathrm{Bits}(n) \wedge b \in \{0, 1\} \end{gathered} \tag{3.5}$$

La operación de cardinalidad se define para todo natural n considerando que existe una única cadena de tamaño cero y que todas las cadenas de tamaño $n + 1$

se pueden formar concatenando a las cadenas de tamaño n los bits 0 o 1.

$$\begin{aligned} |\operatorname{Bits}(0)| &= 1 \\ |\operatorname{Bits}(n+1)| &= 2 \times |\operatorname{Bits}(n)| \end{aligned} \tag{3.6}$$

A partir de esta recurrencia, se concluye que la cardinalidad se puede definir de forma alternativa como

$$|\operatorname{Bits}(n)| = 2^n.$$

Para definir la operación $\operatorname{rank}_{\operatorname{Bits}(n)}$ se considera que ε precede al resto de las cadenas y que $\alpha_1 \cdot b$ precede a $\alpha_2 \cdot b$ cuando α_1 precede a α_2, mientras que $\alpha \cdot 0$ precede a $\alpha \cdot 1$. Sea $|\alpha| = n$, se define

$$\begin{aligned} \operatorname{rank}_{\operatorname{Bits}(0)}(\varepsilon) &= 0 \\ \operatorname{rank}_{\operatorname{Bits}(n+1)}(\alpha \cdot 0) &= 2 \times \operatorname{rank}_{\operatorname{Bits}(n)}(\alpha) \\ \operatorname{rank}_{\operatorname{Bits}(n+1)}(\alpha \cdot 1) &= 2 \times \operatorname{rank}_{\operatorname{Bits}(n)}(\alpha) + 1 \end{aligned} \tag{3.7}$$

Ya que b toma valores 0 o 1, una definición alternativa es

$$\operatorname{rank}_{\operatorname{Bits}(n+1)}(\alpha \cdot b) = 2 \operatorname{rank}_{\operatorname{Bits}(n)}(\alpha) + b.$$

Esta definición se basa en el método de división euclidiana (2.6), en donde $\operatorname{rank}_{\operatorname{Bits}(n+1)}$ corresponde al dividendo, el valor 2 corresponde a divisor, $\operatorname{rank}_{\operatorname{Bits}(n)}$ corresponde al cociente y b corresponde al residuo.

Proposición (pe 3.1)**.** *Para toda cadena binaria* $\alpha \in \operatorname{Bits}(n)$ *existe una posición* k *tal que si* $\operatorname{rank}_{\operatorname{Bits}(n)}(\alpha) = k$*, entonces* $0 \leq k < |\operatorname{Bits}(n)|$.

Prueba. Se muestra que se satisface la propiedad por inducción sobre n. En el caso base, la única cadena en la enumeración es ε, por lo que

$$\begin{aligned} &\operatorname{rank}_{\operatorname{Bits}(0)}(\varepsilon) = k \\ \implies &0 = k \\ \implies &0 \leq k < 1 = |\operatorname{Bits}(0)| \end{aligned}$$

Ahora se considera que la proposición se satisface para n por hipótesis de inducción y se muestra que también se satisface para $n+1$. Se abordan las dos desigualdades de forma independiente considerando que $|\alpha| = n$ y $b \in \{0, 1\}$.

Para mostrar que la posición calculada por $\operatorname{rank}_{\operatorname{Bits}(n+1)}$ es mayor o igual a

cero se parte de la hipótesis de inducción.

$$\begin{aligned} & 0 \leq \operatorname{rank}_{\operatorname{Bits}(n)}(\alpha) \\ \Longrightarrow\ & 0 \leq 2 \operatorname{rank}_{\operatorname{Bits}(n)}(\alpha) \\ \Longrightarrow\ & 0 \leq b \leq 2 \operatorname{rank}_{\operatorname{Bits}(n)}(\alpha) + b \\ \Longrightarrow\ & 0 \leq \operatorname{rank}_{\operatorname{Bits}(n+1)}(\alpha \cdot b) \end{aligned}$$

Para mostrar que la posición calculada es estrictamente menor a la cardinalidad de la enumeración, se tiene que

$$\begin{aligned} & \operatorname{rank}_{\operatorname{Bits}(n)}(\alpha) < 2^n \\ \Longrightarrow\ & \operatorname{rank}_{\operatorname{Bits}(n)}(\alpha) \leq 2^n - 1 \\ \Longrightarrow\ & 2 \operatorname{rank}_{\operatorname{Bits}(n)}(\alpha) \leq 2(2^n - 1) \\ \Longrightarrow\ & 2 \operatorname{rank}_{\operatorname{Bits}(n)}(\alpha) \leq 2^{n+1} - 2 \\ \Longrightarrow\ & 2 \operatorname{rank}_{\operatorname{Bits}(n)}(\alpha) + 1 \leq 2^{n+1} - 1 \\ \Longrightarrow\ & 2 \operatorname{rank}_{\operatorname{Bits}(n)}(\alpha) + b < 2^{n+1} \\ \Longrightarrow\ & \operatorname{rank}_{\operatorname{Bits}(n+1)}(\alpha \cdot b) < |\operatorname{Bits}(n+1)| \end{aligned}$$

□

La operación $\operatorname{unrank}_{\operatorname{Bits}(n)}$ es definida como inversa de $\operatorname{rank}_{\operatorname{Bits}(n)}$ por construcción, de tal forma que satisface pe 3.3. Al considerar $k' = \operatorname{rank}_{\operatorname{Bits}(n)}(\alpha)$ se tiene que $\operatorname{unrank}_{\operatorname{Bits}(n)}(k') = \alpha$, por lo tanto

$$\begin{aligned} & \operatorname{rank}_{\operatorname{Bits}(n+1)}(\alpha \cdot b) = 2 \operatorname{rank}_{\operatorname{Bits}(n)}(\alpha) + b \\ \Longrightarrow\ & \alpha \cdot b = \operatorname{unrank}_{\operatorname{Bits}(n+1)}(2 \operatorname{rank}_{\operatorname{Bits}(n)}(\alpha) + b) \\ \Longrightarrow\ & \operatorname{unrank}_{\operatorname{Bits}(n)}(k') \cdot b = \operatorname{unrank}_{\operatorname{Bits}(n+1)}(2k' + b) \end{aligned}$$

A partir de esta derivación y las relaciones de la división euclidiana (2.6) se define la operación $\operatorname{unrank}_{\operatorname{Bits}(n)}$ como

$$\begin{aligned} \operatorname{unrank}_{\operatorname{Bits}(0)}(0) &= \varepsilon \\ \operatorname{unrank}_{\operatorname{Bits}(n+1)}(k) &= \operatorname{unrank}_{\operatorname{Bits}(n)}(k \operatorname{div} 2) \cdot k \bmod 2 \end{aligned} \tag{3.8}$$

de tal forma que para posiciones pares la cadena asociada termina en 0, mientras que para posiciones impares la cadena asociada termina en 1.

Proposición (pe 3.2)**.** *Para toda posición* k *donde* $0 \leq k < |\operatorname{Bits}(n)|$*, existe una cadena binaria* α *tal que si* $\operatorname{unrank}_{\operatorname{Bits}(n)}(k) = \alpha$*, entonces* $\alpha \in \operatorname{Bits}(n)$*.*

Prueba. Se muestra que se satisface la propiedad por inducción sobre n. En el caso base, se tiene que $|\operatorname{Bits}(0)| = 1$ por lo tanto $k = 0$

$$\begin{aligned} & \operatorname{unrank}_{\operatorname{Bits}(0)}(0) \in \operatorname{Bits}(0) \\ \Longrightarrow\ & \varepsilon \in \operatorname{Bits}(0) \end{aligned}$$

Ahora se considera que la proposición se satisface para n por hipótesis de inducción y se muestra que también se satisface para $n+1$. Se parte de la hipótesis $0 \leq k < |\operatorname{Bits}(n+1)|$

$$\begin{aligned} & 0 \leq k < 2^{n+1} \\ \Longrightarrow\ & 0 \leq 2(k \operatorname{div} 2) \leq k < 2^{n+1} \\ \Longrightarrow\ & 0 \leq k \operatorname{div} 2 < 2^n \\ \Longrightarrow\ & \operatorname{unrank}_{\operatorname{Bits}(n)}(k \operatorname{div} 2) \in \operatorname{Bits}(n) \\ \Longrightarrow\ & \operatorname{unrank}_{\operatorname{Bits}(n)}(k \operatorname{div} 2) \cdot k \bmod 2 \in \operatorname{Bits}(n+1) \\ \Longrightarrow\ & \operatorname{unrank}_{\operatorname{Bits}(n+1)}(k) \in \operatorname{Bits}(n+1) \end{aligned}$$

□

Para construir la enumeración Bits se considera que la posición de las cadenas de tamaño n preceden a las de tamaño $n+1$, si dos cadenas tienen el mismo tamaño n su orden es determinado por $\operatorname{Bits}(n)$. Una forma de garantizar este ordenamiento es utilizando la cantidad de cadenas que preceden al primer objeto en $\operatorname{Bits}(n)$, es decir, todas las cadenas de tamaños menores a n

$$\sum_{i=0}^{n-1} |\operatorname{Bits}(i)| = \sum_{i=0}^{n-1} 2^i = 2^n - 1 \tag{3.9}$$

por lo que para toda cadena α de tamaño n

$$\operatorname{rank}_{\operatorname{Bits}}(\alpha) = \operatorname{rank}_{\operatorname{Bits}(n)}(\alpha) + 2^n - 1.$$

A partir de este resultado se propone definir $\operatorname{rank}_{\operatorname{Bits}}$ en términos de sí misma en lugar de utilizar la familia $\operatorname{Bits}(n)$ y la ecuación (3.9). En el caso base

$\text{rank}_{\text{Bits}}(\varepsilon) = 0$, de lo contrario, sea $|\alpha| = n$

$$\begin{aligned}
\text{rank}_{\text{Bits}}(\alpha \cdot b) &= \text{rank}_{\text{Bits}(n+1)}(\alpha \cdot b) + 2^{n+1} - 1 \\
&= 2\,\text{rank}_{\text{Bits}(n)}(\alpha) + b + 2^{n+1} - 1 \\
&= 2\,\text{rank}_{\text{Bits}(n)}(\alpha) + b + 2^{n+1} - 2 + 1 \\
&= 2\,\text{rank}_{\text{Bits}(n)}(\alpha) + b + 2(2^n - 1) + 1 \\
&= 2\left(\text{rank}_{\text{Bits}(n)}(\alpha) + 2^n - 1\right) + b + 1 \\
&= 2\,\text{rank}_{\text{Bits}(n)}(\alpha) + b + 1
\end{aligned}$$

por lo tanto, la posición asociada a una cadena binaria cualquiera se define

$$\begin{aligned}
\text{rank}_{\text{Bits}}(\varepsilon) &= 0 \\
\text{rank}_{\text{Bits}}(\alpha \cdot b) &= 2\,\text{rank}_{\text{Bits}(n)}(\alpha) + b + 1
\end{aligned} \tag{3.10}$$

Proposición (pe 3.1)**.** *Para toda cadena binaria* $\alpha \in \text{Bits}$ *existe una posición* k *tal que si* $\text{rank}_{\text{Bits}}(\alpha) = k$*, entonces* $0 \leq k < \infty$*.*

Prueba. Sea $|\alpha| = n$, por las propiedades de enumerabilidad de $\text{Bits}(n)$ se tiene que

$$\begin{aligned}
& 0 \leq \text{rank}_{\text{Bits}(n)}(\alpha) < \infty \\
\implies & 0 \leq 2^n - 1 \leq \text{rank}_{\text{Bits}(n)}(\alpha) + 2^n - 1 < \infty \\
\implies & 0 \leq \text{rank}_{\text{Bits}}(\alpha) < \infty
\end{aligned}$$

□

Finalmente, la operación $\text{unrank}_{\text{Bits}}$ es definida como inversa de $\text{rank}_{\text{Bits}}$ por construcción, de tal forma que satisface pe 3.3. Al considerar $k' = \text{rank}_{\text{Bits}}(\alpha)$ se tiene que $\text{unrank}_{\text{Bits}}(k') = \alpha$, por lo tanto

$$\begin{aligned}
& \text{rank}_{\text{Bits}}(\alpha \cdot b) = 2\,\text{rank}_{\text{Bits}}(\alpha) + b + 1 \\
\implies & \alpha \cdot b = \text{unrank}_{\text{Bits}}(2\,\text{rank}_{\text{Bits}}(\alpha) + b + 1) \\
\implies & \text{unrank}_{\text{Bits}}(k') \cdot b = \text{unrank}_{\text{Bits}}(2k' + b + 1)
\end{aligned}$$

A partir de esta derivación y las relaciones de la división euclidiana (2.6) se

define la operación $\text{unrank}_{\text{Bits}}$ como

$$\begin{aligned} \text{unrank}_{\text{Bits}}(0) &= \varepsilon \\ \text{unrank}_{\text{Bits}}(k+1) &= \text{unrank}_{\text{Bits}}(k \text{ div } 2) \cdot k \bmod 2 \end{aligned} \tag{3.11}$$

de tal forma que para posiciones impares la cadena asociada termina en 0, mientras que para posiciones pares mayores a cero la cadena asociada termina en 1.

Proposición (pe 3.2)**.** *Para toda posición k donde $0 \leq k < \infty$, existe una cadena binaria α tal que si $\text{unrank}_{\text{Bits}}(k) = \alpha$, entonces $\alpha \in \text{Bits}$.*

Prueba. Se muestra por inducción fuerte sobre k. En el caso base $k = 0$, por lo que

$$\text{unrank}_{\text{Bits}}(0) = \varepsilon \in \text{Bits}\,.$$

En el paso inductivo, se supone que la proposición se satisface para toda posición menor o igual a k y se muestra que se satisface para $k+1$.

$$\begin{aligned} & k \text{ div } 2 \leq k < k+1 \\ \implies & \text{unrank}_{\text{Bits}}(k \text{ div } 2) \in \text{Bits} \\ \implies & \text{unrank}_{\text{Bits}}(k \text{ div } 2) \cdot k \bmod 2 \in \text{Bits} \\ \implies & \text{unrank}_{\text{Bits}}(k+1) \in \text{Bits} \end{aligned}$$

□

La tabla 3.1 muestra las primeras quince cadenas binarias de acuerdo a Bits, al costado de la tabla se identifican las cadenas correspondientes a Bits(0), Bits(1), Bits(2) y Bits(3). El orden inducido por esta enumeración es lexicográfico creciente.

A forma de ejemplo, se muestra a continuación la derivación de la cadena

	k	$\text{unrank}_{\text{Bits}}(k)$
$\text{Bits}(0)$	0	ε
$\text{Bits}(1)$	1	0
	2	1
$\text{Bits}(2)$	3	00
	4	01
	5	10
	6	11
$\text{Bits}(3)$	7	000
	8	001
	9	010
	10	011
	11	100
	12	101
	13	110
	14	111

Tabla 3.1: Primeras cadenas binarias enumeradas por Bits.

binaria en la posición 13 de la enumeración Bits.

$$\begin{aligned}
\text{unrank}_{\text{Bits}}(13) &= \text{unrank}_{\text{Bits}}(12+1) \\
&= \text{unrank}_{\text{Bits}}(2\times 6+0+1) \\
&= \text{unrank}_{\text{Bits}}(6)\cdot 0 \\
&= \text{unrank}_{\text{Bits}}(5+1)\cdot 0 \\
&= \text{unrank}_{\text{Bits}}(2\times 2+1+1)\cdot 0 \\
&= \text{unrank}_{\text{Bits}}(2)\cdot 1\cdot 0 \\
&= \text{unrank}_{\text{Bits}}(1+1)\cdot 1\cdot 0 \\
&= \text{unrank}_{\text{Bits}}(2\times 0+1+1)\cdot 1\cdot 0 \\
&= \text{unrank}_{\text{Bits}}(0)\cdot 1\cdot 1\cdot 0 \\
&= \varepsilon\cdot 1\cdot 1\cdot 0 \\
&= 110
\end{aligned}$$

3.5 Combinadores y recursividad

En las siguientes secciones se describen operadores llamados *combinadores* que construyen enumeraciones a partir de otras enumeraciones. De forma similar a los operadores K y Nat, los combinadores son parametrizados por valores que determinan tanto las propiedades como la relación de posiciones y objetos en la enumeración resultante. A diferencia de estos operadores, la parametrización de los combinadores consiste de enumeraciones cualquiera.

Un problema que surge a partir del uso de combinadores es la construcción de enumeraciones recursivas, es decir, enumeraciones que se definen en términos de sí mismas. Esta recursividad se puede identificar al analizar la gráfica de dependencias de una enumeración.

En la figura 3.1 se muestran algunas gráficas de dependencias que pueden surgir al construir enumeraciones con combinadores, donde cada vértice es una enumeración y cada arista $(\mathbf{C_1}, \mathbf{C_2})$ indica que $\mathbf{C_1}$ se define en términos de $\mathbf{C_2}$.

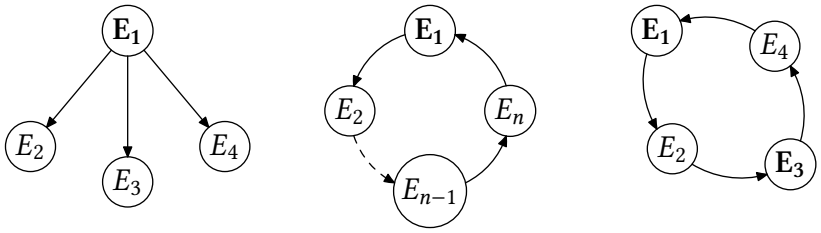


Figura 3.1: Tipos de dependencias entre enumeraciones.

En la primer gráfica se muestra la definición de una enumeración $\mathbf{E_1}$ que depende de tres enumeraciones E_2, E_3 y E_4. Ya que estas enumeraciones no dependen de otras, la definición de $\mathbf{E_1}$ no es recursiva.

En la segunda gráfica se muestra la definición de una enumeración $\mathbf{E_1}$ que depende de forma directa de una enumeración E_2, esta a su vez depende de otras enumeraciones. Esta relación de dependencia continúa hasta llegar a E_n, la cuál a su vez depende de $\mathbf{E_1}$. Por lo tanto, $\mathbf{E_1}$ se define en términos de sí misma y es recursiva.

En la tercer gráfica se muestra la definición de dos enumeraciones $\mathbf{E_1}$ y $\mathbf{E_3}$ las cuales dependen una de la otra de forma indirecta a partir de las enumeraciones E_2 y E_4. En este caso, se dice que $\mathbf{E_1}$ y $\mathbf{E_3}$ son mutuamente recursivas.

En general, las dependencias entre enumeraciones pueden ser más complejas que los ejemplos planteados, por lo que es importante identificar precisamente cuándo una enumeración depende de otra.

Se define el conjunto de dependencias de una enumeración $\mathcal{D}(E)$ a partir de la construcción de E. Si la enumeración E es primitiva, entonces se define $\mathcal{D}(E) = \emptyset$. Si la enumeración E es resultado de un combinador $f(E_1, \ldots, E_n)$, entonces se define

$$\mathcal{D}(E) = \bigcup_{i=1}^{n} (\mathcal{D}(E_i) \cup \{E_i\})$$

es decir, las dependencias de una enumeración combinada son los operandos utilizados para construirla junto con las dependencias de los operandos. Por lo tanto, una enumeración E es recursiva cuando $E \in \mathcal{D}(E)$.

Las gráficas y listas de dependencias de las enumeraciones son importantes para calcular la cardinalidad de las enumeraciones recursivas, así como en la compilación de enumeraciones a código ejecutable.

Para construir enumeraciones a partir de combinadores se considera que los operandos satisfacen el protocolo de enumeración. Sin embargo, esta suposición no es suficiente para garantizar que el objeto resultante es una enumeración bien formada.

Para ilustrar el tipo de problemas que pueden surgir se consideran las enumeraciones descritas en el capítulo 2, donde se definen funciones rank y unrank para cada categoría sintáctica del lenguaje IMP. En la figura 3.2 se muestra la gráfica de dependencias entre las enumeraciones del lenguaje.

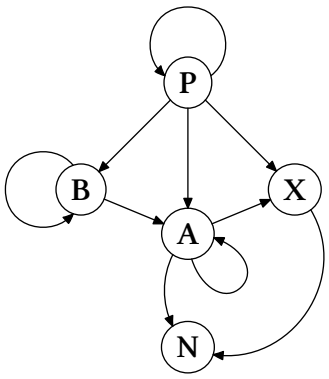


Figura 3.2: Dependencias entre enumeraciones de IMP.

Las enumeraciones **P**, **B** y **A** son recursivas y su definición depende de la especificación del lenguaje. Como se describe en el capítulo 2, las funciones rank y unrank de una categoría sintáctica se construyen a partir del orden de sus producciones en la especificación. De tal manera que cambiar el orden de las

producciones, cambia el orden de los objetos enumerados.

Se considera un orden alternativo $\mathbf{A}'$ para las producciones de $\mathbf{A}$ donde las operaciones de suma, resta y multiplicación preceden a los naturales y localidades de memoria. La tabla 3.2 muestra la diferencia entre estos dos ordenamientos.

i	$\mathbf{A}(i)$	$\mathbf{A}'(i)$
0	$\mathbf{N}$	(add $\mathbf{A}'$ $\mathbf{A}'$)
1	$\mathbf{X}$	(sub $\mathbf{A}'$ $\mathbf{A}'$)
2	(add $\mathbf{A}$ $\mathbf{A}$)	(mul $\mathbf{A}'$ $\mathbf{A}'$)
3	(sub $\mathbf{A}$ $\mathbf{A}$)	$\mathbf{N}$
4	(mul $\mathbf{A}$ $\mathbf{A}$)	$\mathbf{X}$

Tabla 3.2: Orden de producciones de $\mathbf{A}$ y $\mathbf{A}'$.

Ya que la estructura de las producciones es la misma, se pudiera pensar que el conjunto de objetos enumerados es el mismo, sin embargo este no es el caso. Para mostrar el problema con el orden alternativo, se considera el cálculo del primer objeto enumerado, es decir, $\text{unrank}_{\mathbf{A}'}(0)$.

De acuerdo al algoritmo utilizado, primero se determinan los valores k' e i tales que $i < 5$ y $k = 5k' + i$. En este ejemplo $k = 0$, por lo que tanto k' como i tienen valor 0. Esto significa que la producción correspondiente a esta posición es (add $\mathbf{A}'$ $\mathbf{A}'$) y que $k' = 0$ debe ser utilizado para determinar las posiciones de las subexpresiones aritméticas.

Basado en la figura 2.7 se tiene que $\text{unrank}_2(0) = (0, 0)$, por lo que la posición correspondiente a ambas subexpresiones aritméticas es 0. Sin embargo, en este punto se ha llegado a un ciclo. Para construir el objeto $\text{unrank}_{\mathbf{A}'}(0)$ se debe construir $\text{unrank}_{\mathbf{A}'}(0)$, por lo tanto la enumeración $\mathbf{A}'$ no tiene un primer elemento.

Para evitar este problema con las definiciones recursivas se consideran tres propiedades, la primera es utilizada como heurística en el cálculo de la cardinalidad y las otras dos son son proposiciones a ser verificadas adicionalmente a las propiedades de enumerabilidad.

Propiedad 3.4. *Para toda enumeración recursiva E, el cálculo de su cardinalidad $|E|$ se realiza suponiendo que toda aparición $|E| = \infty$.*

Propiedad 3.5. *Para todo $x \in E$ y toda aparición $\text{rank}_E(x')$ en el cálculo de $\text{rank}_E(x)$, el objeto x' es estructuralmente menor al objeto x.*

Propiedad 3.6. *Para toda posición k donde $0 \leq k < |E|$ y toda aparición* $\mathrm{unrank}_E(k')$ *en el cálculo de* $\mathrm{unrank}_E(k)$*, la posición k' es estrictamente menor a k.*

El razonamiento detrás de pe 3.4 es que la suposición permite resolver el problema de la recursividad en el cálculo de la cardinalidad sin obtener resultados erróneos. Sea E una enumeración cualquiera, si E es acotada y no-recursiva, la suposición nunca es utilizada en el cálculo, por otro lado si E es no-acotada y recursiva, entonces la suposición corresponde a la cardinalidad de la enumeración. El único caso en que esta suposición puede resultar incorrecta es cuando E es acotada y recursiva en cuyo caso se considera que E es malformada o tiene una definición equivalente que es contemplada por alguno de los otros dos casos.

Las propiedades pe 3.5 y pe 3.6 hacen referencia a la *aparición* de rank y unrank dentro de su cálculo y que el argumento de estos cálculos recursivos debe ser menor al del cálculo inicial. En ambos casos, esta condición es utilizada para mostrar que las propiedades pe 3.1, pe 3.2 y pe 3.3 se satisfacen por inducción.

En la definición de pe 3.5 se menciona el término "estructuralmente menor" como comparación entre dos objetos. Esta noción es similar a la relación "estrictamente menor" entre naturales, pero puede ser utilizada para otras estructuras definidas de forma inductiva. En particular este trabajo contempla el uso de árboles de sintaxis basados en listas.

Una lista se define ya sea como la lista vacía null, o como una pareja $(x\ .\ \ell)$ donde x es cualquier objeto y ℓ es una lista. Se dice que x es el *primer elemento* de la lista y ℓ es el *resto* de la lista. Se utiliza la notación usual de listas $(x_1, x_2, \ldots, x_{n-1}, x_n)$ para describir el valor de

$$(x_1\ .\ (x_2\ .\ \ldots (x_{n-1}\ .\ (x_n\ .\ \mathsf{null}))\ldots)).$$

Se define que un objeto y es estructuralmente menor a una lista $(x\ .\ \ell)$ cuando $y = x$, $y = \ell$, o bien cuando y es estructuralmente menor a x o estructuralmente menor a ℓ. Desde una perspectiva textual este criterio se satisface cuando y aparece como subtérmino de $(x\ .\ \ell)$. Al comparar dos naturales, se define que n es estructuralmente menor a m cuando $n < m$. Por este motivo se denota esta comparación con el mismo símbolo $<$.

3.6 Producto de enumeraciones

En esta sección se define un combinador llamado *producto*, debido a su relación con el producto cartesiano de conjuntos. Sean $E_1, \ldots, E_n$ enumeraciones,

el producto product$(E_1, \ldots, E_n)$ enumera todas las listas de la forma $(x_1, \ldots, x_n)$ donde $x_i \in E_i$, es decir, es una enumeración del producto cartesiano de los objetos enumerados por sus operandos.

Una estrategia utilizada para definir combinadores es primero abordar un problema más simple que permita resolver el problema original. En el caso del producto de enumeraciones, el problema simplificado consiste en definir un operador llamado *combinador de parejas*. Sean E_1 y E_2 enumeraciones, la enumeración de parejas $E_1 \times E_2$ enumera todas las parejas de la forma $(x_1 \ . \ x_2)$ donde $x_1 \in E_1$ y $x_2 \in E_2$.

La cardinalidad de las enumeraciones resultantes de este combinador se define a partir del hecho que para cada $x_1 \in E_1$ existen $|E_2|$ parejas en $E_1 \times E_2$, por lo tanto

$$|E_1 \times E_2| = |E_1| \times |E_2|. \tag{3.12}$$

Se considera que cuando alguno de los operandos tiene cardinalidad cero $|E_1| \times |E_2| = 0$, por lo tanto, en este caso se define

$$|E_1| = 0 \vee |E_2| = 0 \implies E_1 \times E_2 = \emptyset. \tag{3.13}$$

Por otro lado, cuando alguno de los operandos es no-acotado se considera que $|E_1| \times |E_2| = \infty$.

Para que un objeto sea enumerado por $E_1 \times E_2$, este debe ser una pareja cuyo primer elemento pertenece a E_1 y cuyo resto pertenece a E_2, es decir

$$(x_1 \ . \ x_2) \in E_1 \times E_2 = x_1 \in E_1 \text{ y } x_2 \in E_2 \tag{3.14}$$

Las funciones rank y unrank para parejas se definen haciendo un análisis de casos sobre las cardinalidades de E_1 y E_2. Cuando ambas son no-acotadas, entonces se utiliza la enumeración de duplas para relacionar posiciones con parejas de posiciones. Cuando alguno de los operandos es acotada, entonces se utiliza la división euclidiana (2.6) para determinar qué pareja de posiciones corresponden a las parejas de objetos enumerados.

Se considera que cuando alguno de los operandos es acotado su cardinalidad es mayor a cero, debido al caso base (3.13).

Primero se considera el caso en que el primer operando B es acotado, mientras que el segundo operando E puede o no ser acotado. Para utilizar la división euclidiana se considera $|B|$ como divisor, mientras que la posición de un objeto

$x_1 \in B$ es el residuo. Por pe 3.1 se tiene que

$$0 \leq \text{rank}_B(x_1) < |B|.$$

La posición de un objeto $x_2 \in E$ es utilizada como cociente en las relaciones de la división. Por lo tanto, la función que asocia parejas a posiciones se define como

$$\text{rank}_{B\times E}((x_1 \ . \ x_2)) = |B| \, \text{rank}_E(x_2) + \text{rank}_B(x_1) \tag{3.15}$$

En la figura 3.3 se muestra un diagrama de cómo se ordenan los objetos en este caso, al considerar B como una enumeración con cinco objetos y E como una enumeración no-acotada, el orden consiste primero en ordenar el primer elemento de E emparejado con cada elemento de B en orden, continuando con el segundo elemento de E emparejado de la misma manera y siguiendo esta relación para elementos sucesivos en E.

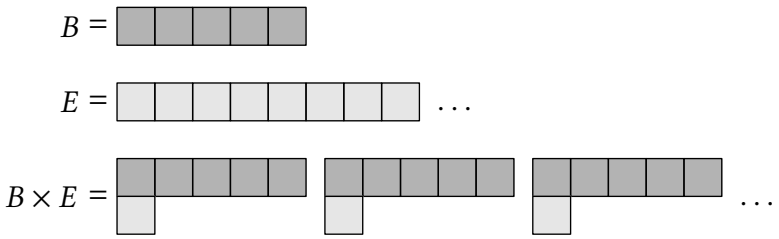


Figura 3.3: Orden de objetos en $B \times E$.

A partir de este método se tiene que $(x_1 \ . \ x_2)$ precede a $(y_1 \ . \ y_2)$ si x_2 precede a y_2 en E, o bien, si $x_2 = y_2$ y x_1 precede a y_1 en B.

Para obtener la función inversa $\text{unrank}_{B\times E}$ se consideran las relaciones de la división euclidiana. Sea $k = \text{rank}_{B\times E}((x_1 \ . \ x_2))$ se tiene que

$$\begin{aligned} k &= |B| \, \text{rank}_E(x_2) + \text{rank}_B(x_1) \\ \text{rank}_E(x_2) &= k \text{ div } |B| \\ \text{rank}_B(x_1) &= k \bmod |B| \end{aligned}$$

y ya que pe 3.3 se satisface para E y B se tiene que

$$\begin{aligned} x_2 &= \text{unrank}_E(k \text{ div } |B|) \\ x_1 &= \text{unrank}_B(k \bmod |B|) \end{aligned}$$

por lo que se define

$$\text{unrank}_{B\times E}(k) = (\text{unrank}_B(k \bmod |B|) \ . \ \text{unrank}_E(k \operatorname{div} |B|)) \tag{3.16}$$

Cuando el primer operando U es no-acotado y el segundo operando B es acotado, entonces se realiza el mismo procedimiento anterior utilizando como divisor a $|B|$, es decir

$$\text{rank}_{U\times B}((x_1 \ . \ x_2)) = |B|\,\text{rank}_U(x_1) + \text{rank}_B(x_2) \tag{3.17}$$

$$\text{unrank}_{U\times B}(k) = (\text{unrank}_U(k \operatorname{div} |B|) \ . \ \text{unrank}_B(k \bmod |B|)) \tag{3.18}$$

Finalmente, se considera el caso en que ambos operandos U_1 y U_2 son no-acotados. Ya que la división euclidiana requiere de un divisor natural, las cardinalidades de los operandos no pueden ser utilizados en este método. En estos casos se recurre a distribuir las posiciones utilizando la enumeración de duplas.

$$\text{rank}_{U_1\times U_2}((x_1 \ . \ x_2)) = \text{rank}_2((\text{rank}_{U_1}(x_1), \text{rank}_{U_2}(x_2))) \tag{3.19}$$

De acuerdo a pe 3.3 de U_1 y U_2 se tiene que

$$\begin{aligned} \text{rank}_{U_1}(x_1) = k_1 &\implies x_1 = \text{unrank}_{U_1}(k_1) \\ \text{rank}_{U_2}(x_2) = k_2 &\implies x_2 = \text{unrank}_{U_2}(k_2) \end{aligned}$$

Por lo que al considerar $\text{unrank}_2(k) = (k_1, k_2)$ se define

$$\text{unrank}_{U_1\times U_2}(k) = (\text{unrank}_{U_1}(k_1) \ . \ \text{unrank}_{U_2}(k_2)) \tag{3.20}$$

Ya que unrank es definida como inversa de rank para los tres casos, se considera que pe 3.3 se satisface para $E_1 \times E_2$ independientemente de la cardinalidad de los operandos.

Proposición (pe 3.1)**.** *Sea $E = E_1 \times E_2$, para toda pareja $(x_1 \ . \ x_2) \in E$ existe una posición k tal que si $\text{rank}_E((x_1 \ . \ x_2)) = k$, entonces $0 \leq k < |E|$.*

Prueba. Primero se aborda el caso $|E_1| < \infty$ mostrando que se satisfacen ambos lados de la desigualdad de forma independiente. Para mostrar que $0 \leq k$ se parte

de que E_2 satisface pe 3.1.

$$\begin{aligned}
& 0 \leq \mathrm{rank}_{E_2}(x_2) \\
\Longrightarrow\ & 0 \leq |E_1|\,\mathrm{rank}_{E_2}(x_2) \\
\Longrightarrow\ & \mathrm{rank}_{E_1}(x_1) \leq |E_1|\,\mathrm{rank}_{E_2}(x_2) + \mathrm{rank}_{E_1}(x_1) \\
\Longrightarrow\ & 0 \leq |E_1|\,\mathrm{rank}_{E_2}(x_2) + \mathrm{rank}_{E_1}(x_1) \\
\Longrightarrow\ & 0 \leq \mathrm{rank}_E((x_1\ .\ x_2))
\end{aligned}$$

Para mostrar que $k < |E|$ se tiene que cuando $|E_2| = \infty$ la desigualdad se satisface trivialmente ya que el cálculo de k resulta en un natural por lo que siempre tiene como cota superior ∞. Por otro lado, cuando $|E_2| < \infty$ se considera el siguiente razonamiento

$$\begin{aligned}
& \mathrm{rank}_{E_2}(x_2) < |E_2| \\
\Longrightarrow\ & \mathrm{rank}_{E_2}(x_2) \leq |E_2| - 1 \\
\Longrightarrow\ & |E_1|\,\mathrm{rank}_{E_2}(x_2) \leq |E_1| \times |E_2| - |E_1| \\
\Longrightarrow\ & |E_1|\,\mathrm{rank}_{E_2}(x_2) + \mathrm{rank}_{E_1}(x_1) \leq |E_1| \times |E_2| - |E_1| + \mathrm{rank}_{E_1}(x_1) \\
\Longrightarrow\ & |E_1|\,\mathrm{rank}_{E_2}(x_2) + \mathrm{rank}_{E_1}(x_1) \leq |E_1| \times |E_2| - |E_1| + |E_1| - 1 \\
\Longrightarrow\ & \mathrm{rank}_E((x_1\ .\ x_2)) \leq |E_1| \times |E_2| - 1 \\
\Longrightarrow\ & \mathrm{rank}_E((x_1\ .\ x_2)) < |E_1| \times |E_2|
\end{aligned}$$

Cuando $|E_1| = \infty$ pero $|E_2| < \infty$, se sigue el razonamiento anterior invirtiendo los roles de E_1 y E_2 en la demostración.

Finalmente, si $|E_1| = |E_2| = \infty$ se tiene que $|E| = \infty$, partiendo que pe 3.1 se satisface para ambos operandos se tiene que $0 \leq \mathrm{rank}_{E_1}(x_1)$ y $0 \leq \mathrm{rank}_{E_2}(x_2)$. A partir de la propiedad de enumeración de duplas (2.7) se concluye que

$$\begin{aligned}
0 \leq \mathrm{rank}_{E_1}(x_1) \leq \mathrm{rank}_2((\mathrm{rank}_{E_1}(x_1), \mathrm{rank}_{E_2}(x_2))) \\
0 \leq \mathrm{rank}_{E_2}(x_2) \leq \mathrm{rank}_2((\mathrm{rank}_{E_1}(x_1), \mathrm{rank}_{E_2}(x_2)))
\end{aligned}$$

por lo tanto

$$0 \leq \mathrm{rank}_E((x_1\ .\ x_2)) < \infty.$$

□

Proposición (pe 3.2). *Sea $E = E_1 \times E_2$, para toda posición k donde $0 \leq k < |E|$, existe una pareja $(x_1\ .\ x_2)$ tal que si $\mathrm{unrank}_E(k) = (x_1\ .\ x_2)$, entonces $(x_1\ .\ x_2) \in E$.*

Prueba. Cuando $|E_1| < \infty$, por la definición (3.16) se tiene que

$$x_1 = \text{unrank}_{E_1}(k \bmod |E_1|)$$
$$x_2 = \text{unrank}_{E_2}(k \operatorname{div} |E_1|)$$

Para mostrar que $(x_1 \,.\, x_2) \in E$ se debe mostrar que $x_1 \in E_1$ y $x_2 \in E_2$. Por definición de residuo $0 \leq k \bmod |E_1| < |E_1|$, por lo tanto $x_1 \in E_1$. Por otro lado, ya que $0 \leq k < |E_1| \times |E_2|$ se tiene que $0 \leq k \operatorname{div} |E_1| < |E_2|$, de tal forma que $x_2 \in E_2$.

Cuando $|E_1| = \infty$ y $|E_2| < \infty$, se sigue el razonamiento anterior invirtiendo los roles de E_1 y E_2 en la demostración.

Finalmente, si $|E_1| = |E_2| = \infty$, por la enumeración de duplas se tiene que si $\text{unrank}_2(k) = (k_1, k_2)$, entonces $0 \leq k_1$ y $0 \leq k_2$. Considerando la definición (3.20) se tiene que

$$\text{unrank}_E(k) = (\text{unrank}_{E_1}(k_1) \;.\; \text{unrank}_{E_2}(k_2)).$$

Ya que k_1 y k_2 satisfacen el antecedente de pe 3.2 para E_1 y E_2 respectivamente, se tiene que $\text{unrank}_{E_1}(k_1) \in E_1$ y $\text{unrank}_{E_2}(k_2) \in E_2$, por lo tanto $\text{unrank}_E(k) \in E$. □

En la tabla 3.3 se muestran las primeras posiciones de cuatro enumeraciones construidas con el combinador de parejas, cada enumeración es representativa de un caso correspondiente a la cardinalidad de los operandos.

Una vez definido el combinador de parejas, se puede definir la enumeración resultante de $\text{product}(E_1, E_2, \ldots, E_{n-1}, E_n)$ con la composición de parejas

$$E_1 \times (E_2 \times (\ldots (E_{n-1} \times (E_n \times \text{K}(\mathsf{null}))) \ldots)).$$

De acuerdo a la definición de pertenencia (3.14) y la definición inductiva de listas, se tiene que esta composición enumera todos los objetos de la forma $(x_1, \ldots, x_n)$ tal que $x_i \in E_i$.

En el contexto de la enumeración de IMP, se puede utilizar el producto para construir las enumeraciones de algunas subexpresiones, por ejemplo, un equivalente a la enumeración de las asignaciones de memoria sería

$$(\text{set } \mathbf{X}\ \mathbf{A}) = \text{product}(\text{K}(\text{set}), \mathbf{X}, \mathbf{A}),$$

Nat(1, 2) × Nat(1, 3)	
k	$\text{unrank}_E(k)$
0	(1 . 1)
1	(2 . 1)
2	(1 . 2)
3	(2 . 2)
4	(1 . 3)
5	(2 . 3)

Nat × Bits	
k	$\text{unrank}_E(k)$
0	(0 . ε)
1	(0 . 0)
2	(1 . ε)
3	(0 . 1)
4	(1 . 0)
5	(2 . ε)
6	(0 . 00)
7	(1 . 1)
8	(2 . 0)
9	(3 . ε)
10	(0 . 01)

Nat(1, 2) × Bits	
k	$\text{unrank}_E(k)$
0	(1 . ε)
1	(2 . ε)
2	(1 . 0)
3	(2 . 0)
4	(1 . 1)
5	(2 . 1)
6	(1 . 00)
7	(2 . 00)
8	(1 . 01)
9	(2 . 01)
10	(1 . 10)

Bits × Nat(1, 3)	
k	$\text{unrank}_E(k)$
0	(ε . 1)
1	(ε . 2)
2	(ε . 3)
3	(0 . 1)
4	(0 . 2)
5	(0 . 3)
6	(1 . 1)
7	(1 . 2)
8	(1 . 3)
9	(00 . 1)
10	(00 . 2)

Tabla 3.3: Objetos en las primeras posiciones de enumeraciones de parejas.

para las iteraciones while sería

$$(\text{while } \mathbf{B}\ \mathbf{P}) = \text{product}(\text{K}(\text{while}), \mathbf{B}, \mathbf{P}),$$

para las condicionales if sería

$$(\text{if } \mathbf{B}\ \mathbf{P}\ \mathbf{P}) = \text{product}(\text{K}(\text{if}), \mathbf{B}, \mathbf{P}, \mathbf{P}),$$

y de forma similar para otros tipos de expresión asociados a árboles de sintaxis.

Definir al producto como composición de parejas puede ser conveniente debido a su simplicidad y por que se garantiza satisfacer el protocolo de enumerabilidad. Sin embargo, esta definición presenta una desventaja en cuestión a las aplicaciones computacionales de las enumeraciones.

Uno de los principales usos de las enumeraciones es como mecanismos de exploración. Por ejemplo, para calcular y analizar programas a partir de la posición cero, o bien alrededor de una posición particular. En este contexto, no solo es importante garantizar que cualquier expresión puede ser encontrada en alguna posición. También es importante entender cómo se distribuyen las expresiones a lo largo de la enumeración.

Para abordar este tema se introduce la noción de *rangos de exploración*. Al considerar una enumeración $E = \text{product}(E_1, \ldots, E_n)$, el rango de exploración de

cada operando en las primeras k posiciones en E se refiere a las posiciones de los objetos x_i en las enumeraciones E_i. Por ejemplo, en la enumeración de expresiones if es de interés analizar cómo se distribuyen las posiciones exploradas para las condicionales, en contraste con las posiciones de los programas del consecuente y el alternante.

Para analizar estos rangos de exploración se propone enumerar las primeras k listas $(x_1, \ldots, x_n)$ de un producto, graficando la posición máxima calculada para cada posición k_i asociada a sus componentes x_i. Al no presuponer más relevantes unos componentes que otros, se considera *deseable* que las posiciones máximas sean similares para cada componente.

En la figura 3.4 se muestran dos gráficas, la primera consiste en los rangos de exploración de un producto de tres enumeraciones no-acotadas U_1, U_2 y U_3 de acuerdo a la definición basada en composición de parejas, la segunda gráfica muestra los rangos de exploración deseables para el producto de las mismas tres enumeraciones.

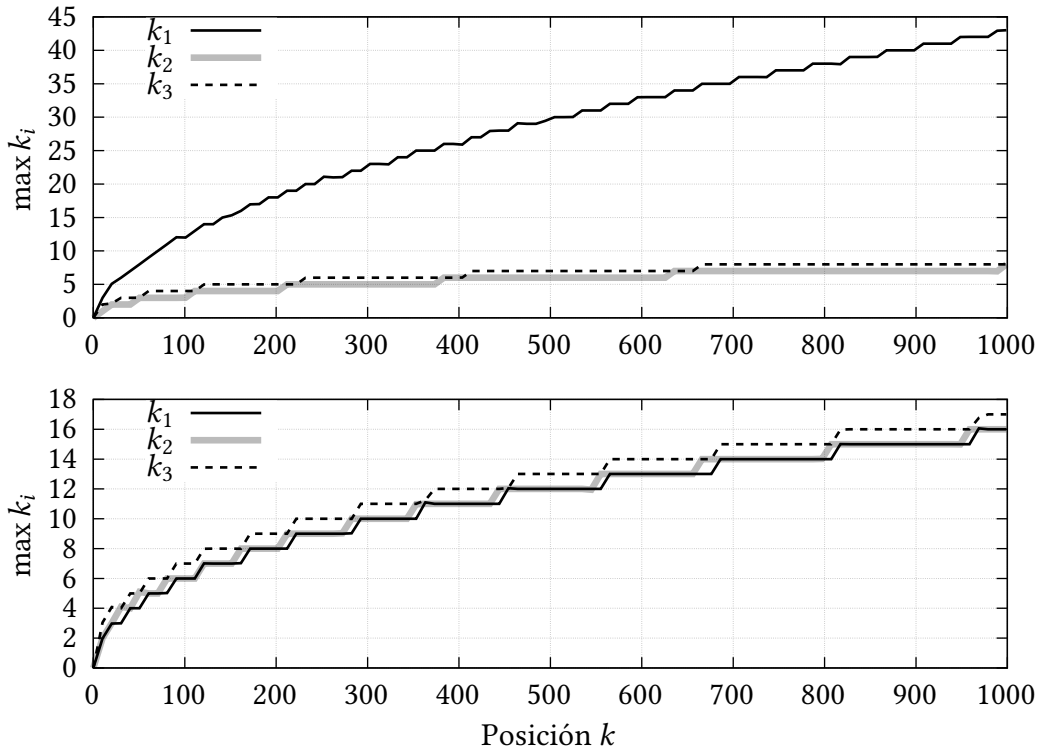


Figura 3.4: Posiciones máximas en productos de tres enumeraciones no-acotadas.

Como se observa en las gráficas, al utilizar la composición de parejas se explora una mayor cantidad de posiciones para el primer componente que para el resto. El problema de fondo con este sesgo es el uso de la enumeración de duplas de naturales en la distribución de la posición k.

Suponiendo que el rango de los valores de los componentes en la enumeración de duplas no es sesgado sobre un componente en particular, la composición de tres o más enumeraciones utilizando la enumeración de parejas resulta en una exploración sesgada de posiciones. Esto ocurre ya que de acuerdo a las definiciones (3.19) y (3.20), el cálculo de las posiciones k_1, k_2 y k_3 se puede resumir como

$$\begin{aligned} \text{unrank}_2(k) &= (k_1, k') \\ \text{unrank}_2(k') &= (k_2, k_3) \end{aligned}$$

Para obtener una exploración más uniforme como en la segunda gráfica de la figura 3.4 se utiliza la distribución de k en n posiciones a partir de la enumeración de n-tuplas, en el ejemplo propuesto esto corresponde a

$$\text{unrank}_3(k) = (k_1, k_2, k_3).$$

En general, las enumeraciones de n-tuplas de naturales descritas en [2] permiten explorar rangos de posiciones sin sesgos.

Definir los combinadores para minimizar los sesgos en la exploración de posiciones es conveniente cuando el objetivo es definir operadores genéricos. Sin embargo, en algunos contextos puede ser conveniente construir enumeraciones con sesgos.

Por ejemplo, al considerar las asignaciones de memoria $\mathbf{X} := \mathbf{A}$ del lenguaje IMP, pudiera ser deseable explorar programas donde haya una mayor variedad de expresiones aritméticas que de localidades de memoria, partiendo de la suposición que hay menos programas "interesantes" que utilizan muchas localidades con poca diversidad de valores almacenados en ellas. Estos criterios son subjetivos y es decisión del usuario de las enumeraciónes incorporar sesgos o no.

En este trabajo se procura definir combinadores con la menor cantidad de sesgos, por lo que la definición del combinador producto basada en composición de parejas de enumeraciones es descartada.

Para obtener un producto con menos sesgos, se define un combinador auxiliar que opera exclusivamente sobre enumeraciones no-acotadas $U_1, \ldots, U_n$ llamado *producto no-acotado*. En las definiciones relacionadas a este combinador se considera que $U = \text{uproduct}(U_1, \ldots, U_n)$.

Cuando $n = 0$ se define uproduct() = K(null), de lo contrario se definen las operaciones del protocolo suponiendo $n > 0$.

$$|U| = \infty \tag{3.21}$$

$$(x_1, \ldots, x_n) \in U = x_1 \in U_1 \wedge \cdots \wedge x_n \in U_n \tag{3.22}$$

Para construir las funciones rank y unrank se utiliza la enumeración de n-tuplas de naturales [2]. Sea $\text{unrank}_n(k) = (k_1, \ldots, k_n)$ se define

$$\text{rank}_U((x_1, \ldots, x_n)) = \text{rank}_n((\text{rank}_{U_1}(x_1), \ldots, \text{rank}_{U_n}(x_n))) \tag{3.23}$$

$$\text{unrank}_U(k) = (\text{unrank}_{U_1}(k_1), \ldots, \text{unrank}_{U_n}(k_n)) \tag{3.24}$$

Las enumeraciones construidas por uproduct satisfacen las propiedades del protocolo de enumeración, al estar basadas completamente en la enumeración de n-tuplas y en las enumeraciones U_i para las cuáles se supone que el protocolo se satisface como hipótesis.

Ahora se considera el producto general de enumeraciones product$(E_1, \ldots, E_n)$ donde cada E_i puede ser acotada o no-acotada. Cuando el producto no tiene operandos se define product() = K(null), es decir, el producto vacío enumera únicamente la lista vacía. Cuando alguno de los operandos en el producto tiene cardinalidad cero, el operando se considera equivalente a $\emptyset$, por lo que el producto resultante se considera equivalente a $\emptyset$.

Cuando el producto involucra al menos una enumeración y cada operando tiene cardinalidad mayor a cero, se utiliza tanto el combinador de parejas como el combinador de productos no-acotados para obtener la enumeración resultante. El primer paso para mezclar estos dos combinadores es categorizar los operandos en n_1 enumeraciones acotadas B_j y n_2 enumeraciones no-acotadas U_r, preservando su orden como enumeradores como se muestra en el algoritmo de la figura 3.5.

$j \leftarrow 0; r \leftarrow 0$
for i **from** 1 **upto** n
  if $|E_i| < \infty$ **then**
    $j \leftarrow j + 1; B_j \leftarrow E_i; b_j \leftarrow i$
  else
    $r \leftarrow r + 1; U_r \leftarrow E_i; u_r \leftarrow i$
$n_1 \leftarrow j; n_2 \leftarrow r$

Figura 3.5: Clasificación de operandos en el combinador product.

La variable j se utiliza como subíndice para identificar el orden de los operandos acotados, tomando valores entre 0 y n_1. Además de identificar los operandos acotados B_j se almacena la lista b de posiciones correspondientes a estos operandos, de tal forma que si la j-ésima enumeración acotada del producto aparece como el i-ésimo operando, entonces $b_j = i$.

De forma análoga la variable r se utiliza como subíndice para identificar el orden de los operandos no-acotados y toma valores entre 0 y n_2, mientras que la lista u contiene las posiciones de los operandos no-acotados.

Después de realizar esta clasificación se simplifica la definición del combinador product a partir de dos casos. Cuando $n_1 = 0$ todos los operandos son no-acotados, por lo que se define

$$\text{product}(E_1, \ldots, E_n) = \text{uproduct}(E_1, \ldots, E_n).$$

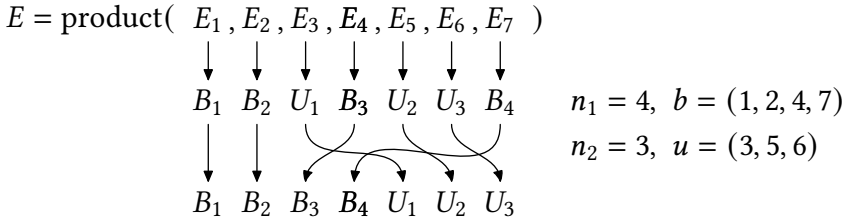
Cuando $n_2 = 0$ todos los operandos son acotados, por lo que se define

$$\text{product}(E_1, \ldots, E_n) = E_1 \times (\ldots \times (E_n \times \text{K}(\text{null})) \ldots).$$

En cualquier otro caso se tiene el producto de al menos un operando acotado y un operando no-acotado. Sea $E = \text{product}(E_1, \ldots, E_n)$, se define la enumeración auxiliar

$$E' = B_1 \times (\ldots \times (B_{n_1} \times \text{uproduct}(U_1, \ldots, U_{n_2})) \ldots). \tag{3.25}$$

La enumeración E' consiste de listas cuyos componentes son enumerados por los operandos de E, sin embargo los componentes de las listas tienen un orden distinto ya que la primer parte corresponde a operadores acotados y la segunda parte a operadores no-acotados. Para ilustrar esta diferencia se considera el ejemplo de la figura 3.6.



$$E' = B_1 \times (B_2 \times (B_3 \times (B_4 \times \text{uproduct}(U_1, U_2, U_3))))$$
$$= E_1 \times (E_2 \times (E_4 \times (E_7 \times \text{uproduct}(E_3, E_5, E_6))))$$

Figura 3.6: Ejemplo de clasificación de operandos en product.

Al analizar la diferencia del orden de los operandos en E y E' se tiene que una lista de objetos $(x_1, x_2, x_3, x_4, x_5, x_6, x_7)$ es enumerada por E si y solo si la lista de objetos $(x_1, x_2, x_4, x_7, x_3, x_5, x_6)$ es enumerada por E'. Por lo que el único problema que se debe resolver para definir las funciones rank y unrank para el producto general consiste en calcular dos operaciones de permutación de listas, una que transforme listas de E en listas de E' y otra que calcule el proceso inverso.

La enumeración asociada E' satisface las propiedades pe 3.1, pe 3.2 y pe 3.3 ya que se construye a partir de combinadores de parejas y del combinador uproduct. Ya que E es simplemente un reordenamiento de los operandos de E' también satisface las propiedades de enumerabilidad.

Utilizando las posiciones originales b_j de los operandos acotados y u_r de los operandos no-acotados, se definen las funciones rank_E y unrank_E a partir de las operaciones de E' y un reordenamiento de los operandos. Los algoritmos de la figura 3.7 muestran cómo se pueden lograr estos cálculos con una simple iteración.

$\text{rank}_E((x_1, \ldots, x_n)) =$
  $j \leftarrow 1$
  $r \leftarrow 1$
  for i **from** 1 **upto** n
    if $b_j = i$ **then**
      $y_i \leftarrow x_i$; $j \leftarrow j + 1$;
    else
      $y_{r+n_1} \leftarrow x_i$; $r \leftarrow r + 1$
  return $\text{rank}_{E'}((y_1, \ldots, y_n))$

$\text{unrank}_E(k) =$
  $(y_1, \ldots, y_n) \leftarrow \text{unrank}_{E'}(k)$
  $j \leftarrow 1$; $r \leftarrow 1$;
  for i **from** 1 **upto** n
    if $i \leq n_1$ **then**
      $x_{b_j} \leftarrow y_i$; $j \leftarrow j + 1$;
    else
      $x_{u_r} \leftarrow y_i$; $r \leftarrow r + 1$
  return $(x_1, \ldots, x_n)$

Figura 3.7: Enumeración del producto general de enumeraciones.

Para ilustrar la relación entre posiciones y listas enumeradas por el producto general de enumeraciones se considera como ejemplo

$$E = \text{product}(\text{Nat}(1, 3), \text{Nat}, \text{K}(\text{foo}), \text{Bits}).$$

Los operandos $\text{Nat}(1, 3)$ y $\text{K}(\text{foo})$ son acotados, mientras que Nat y Bits son no-acotados, por lo que la enumeración auxiliar asociada es

$$E' = \text{Nat}(1, 3) \times (\text{K}(\text{foo}) \times \text{uproduct}(\text{Nat}, \text{Bits})).$$

La tabla 3.4 muestra las listas correspondientes a E' y E para posiciones $k \leq 30$.

$E = \text{product}(\text{Nat}(1,3), \text{Nat}, \text{K}(\text{foo}), \text{Bits})$ $E' = \text{Nat}(1,3) \times (\text{K}(\text{foo}) \times \text{uproduct}(\text{Nat}, \text{Bits}))$		
k	$\text{unrank}_{E'}(k)$	$\text{unrank}_{E}(k)$
0	$(1, \text{foo}, 0, \varepsilon)$	$(1, 0, \text{foo}, \varepsilon)$
1	$(2, \text{foo}, 0, \varepsilon)$	$(2, 0, \text{foo}, \varepsilon)$
2	$(3, \text{foo}, 0, \varepsilon)$	$(3, 0, \text{foo}, \varepsilon)$
3	$(1, \text{foo}, 0, 0)$	$(1, 0, \text{foo}, 0)$
4	$(2, \text{foo}, 0, 0)$	$(2, 0, \text{foo}, 0)$
5	$(3, \text{foo}, 0, 0)$	$(3, 0, \text{foo}, 0)$
6	$(1, \text{foo}, 1, \varepsilon)$	$(1, 1, \text{foo}, \varepsilon)$
7	$(2, \text{foo}, 1, \varepsilon)$	$(2, 1, \text{foo}, \varepsilon)$
8	$(3, \text{foo}, 1, \varepsilon)$	$(3, 1, \text{foo}, \varepsilon)$
9	$(1, \text{foo}, 0, 1)$	$(1, 0, \text{foo}, 1)$
10	$(2, \text{foo}, 0, 1)$	$(2, 0, \text{foo}, 1)$
11	$(3, \text{foo}, 0, 1)$	$(3, 0, \text{foo}, 1)$
12	$(1, \text{foo}, 1, 0)$	$(1, 1, \text{foo}, 0)$
13	$(2, \text{foo}, 1, 0)$	$(2, 1, \text{foo}, 0)$
14	$(3, \text{foo}, 1, 0)$	$(3, 1, \text{foo}, 0)$
15	$(1, \text{foo}, 2, \varepsilon)$	$(1, 2, \text{foo}, \varepsilon)$
16	$(2, \text{foo}, 2, \varepsilon)$	$(2, 2, \text{foo}, \varepsilon)$
17	$(3, \text{foo}, 2, \varepsilon)$	$(3, 2, \text{foo}, \varepsilon)$
18	$(1, \text{foo}, 0, 00)$	$(1, 0, \text{foo}, 00)$
19	$(2, \text{foo}, 0, 00)$	$(2, 0, \text{foo}, 00)$
20	$(3, \text{foo}, 0, 00)$	$(3, 0, \text{foo}, 00)$
21	$(1, \text{foo}, 1, 1)$	$(1, 1, \text{foo}, 1)$
22	$(2, \text{foo}, 1, 1)$	$(2, 1, \text{foo}, 1)$
23	$(3, \text{foo}, 1, 1)$	$(3, 1, \text{foo}, 1)$
24	$(1, \text{foo}, 2, 0)$	$(1, 2, \text{foo}, 0)$
25	$(2, \text{foo}, 2, 0)$	$(2, 2, \text{foo}, 0)$
26	$(3, \text{foo}, 2, 0)$	$(3, 2, \text{foo}, 0)$
27	$(1, \text{foo}, 3, \varepsilon)$	$(1, 3, \text{foo}, \varepsilon)$
28	$(2, \text{foo}, 3, \varepsilon)$	$(2, 3, \text{foo}, \varepsilon)$
29	$(3, \text{foo}, 3, \varepsilon)$	$(3, 3, \text{foo}, \varepsilon)$
30	$(1, \text{foo}, 0, 01)$	$(1, 0, \text{foo}, 01)$

Tabla 3.4: Listas enumeradas por un producto de enumeraciones.

3.7 Unión de enumeraciones

En esta sección se define un combinador llamado *unión*, debido a su relación con la unión de conjuntos. Sean $E_1, \ldots, E_n$ enumeraciones, union$(E_1, \ldots, E_n)$ enumera todos los objetos que enumeran sus operandos.

A diferencia del combinador producto y su relación con el producto cartesiano de conjuntos, el combinador unión no es una enumeración de la unión de los conjuntos de objetos enumerados por sus operandos, debido a que las enumeraciones resultantes de este combinador relacionan una posición única a cada objeto enumerado por cada operando, de tal forma que si dos operandos enumeran el mismo objeto x, entonces x ocupa dos posiciones en la enumeración.

Para garantizar que el combinador funcione como la unión de conjuntos, se debe mostrar que los operandos enumeran conjuntos disjuntos de objetos. A lo largo de este trabajo, se presupone que los operandos de la unión satisfacen esta propiedad.

Antes de definir la unión de n enumeraciones, se describe un combinador más simple, el combinador *suma*. Sean E_1 y E_2 enumeraciones, la suma $E_1 + E_2$ enumera todos los objetos $x_1 \in E_1$ y $x_2 \in E_2$, por lo que se define la cardinalidad y pertenencia de una suma de la siguiente manera.

$$|E_1 + E_2| = |E_1| + |E_2| \tag{3.26}$$

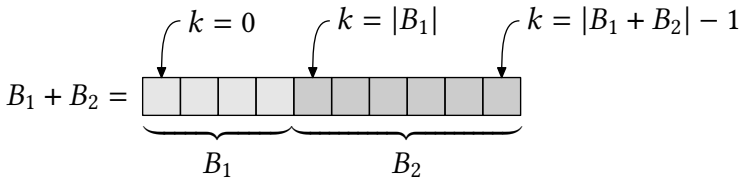
$$x \in E_1 + E_2 = x \in E_1 \text{ o } x \in E_2 \tag{3.27}$$

A partir de estas definiciones se considera que

$$|E_1| = 0 \implies E_1 + E_2 = E_2 \qquad |E_2| = 0 \implies E_1 + E_2 = E_1 \tag{3.28}$$

Las operaciones rank y unrank de sumas se definen a partir de las cardinalides de E_1 y E_2. Los mecanismos utilizados deben garantizar que todos los objetos enumerados por los operandos sean enumerados también por la suma.

Primero se considera el caso en que los dos operandos B_1 y B_2 son acotados. Ya que hay una cantidad finita de objetos a enumerar se propone preservar las posiciones de los objetos en B_1 y desplazar las posiciones de los objetos en B_2 por $|B_1|$, es decir, todo objeto en B_1 precede a todo objeto en B_2 como se muestra en el siguiente diagrama.



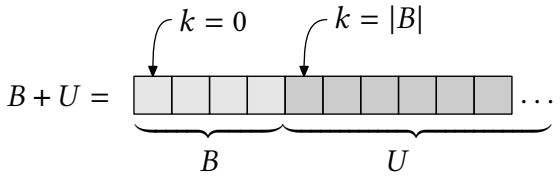
La operación $\text{rank}_{B_1+B_2}(x)$ se define entonces con un análisis de casos dependiendo de qué operando enumera a x.

$$\text{rank}_{B_1+B_2}(x) = \begin{cases} \text{rank}_{B_1}(x) & x \in B_1 \\ |B_1| + \text{rank}_{B_2}(x) & x \notin B_1 \text{ y } x \in B_2 \end{cases} \tag{3.29}$$

La operación $\text{unrank}_{B_1+B_2}(k)$ se define como inversa por construcción. Por pe 3.1 de B_1 se tiene que si $x \in B_1$ entonces su posición debe ser menor a $|B_1|$. En otro caso se tiene una posición $k \geq |E_1|$ la cuál es ajustada para posiciones en B_2.

$$\text{unrank}_{B_1+B_2}(k) = \begin{cases} \text{unrank}_{B_1}(k) & k < |B_1| \\ \text{unrank}_{B_2}(k - |B_1|) & k \geq |B_1| \end{cases} \tag{3.30}$$

En estas dos definiciones, la cardinalidad del segundo operando no es utilizada en los cálculos, por lo que también permiten resolver el caso de la suma con primero operando B acotado y segundo operando U no-acotado como se muestra en el siguiente diagrama.

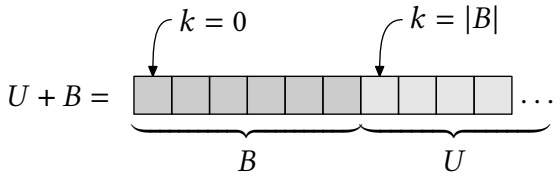


Para una suma con primer operando U no acotado y segundo operando B acotado se realizan cálculos similares a (3.29) y (3.30) al considerar que $U + B = B + U$.

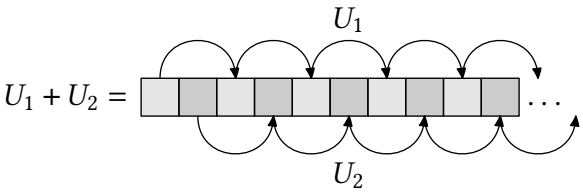
$$\text{rank}_{U+B}(x) = \begin{cases} \text{rank}_{B}(x) & x \in B \\ |B| + \text{rank}_{U}(x) & x \notin B \text{ y } x \in U \end{cases} \tag{3.31}$$

$$\text{unrank}_{U+B}(k) = \begin{cases} \text{unrank}_{B}(k) & k < |B| \\ \text{unrank}_{U}(k - |B|) & k \geq |B| \end{cases} \tag{3.32}$$

Estas definiciones corresponden a enumerar B seguido de U como se muestra en el siguiente diagrama.



Finalmente, para una suma con dos operandos no-acotados U_1 y U_2 se considera asociar las posiciones pares a objetos de U_1 y posiciones impares a objetos de U_2 como se muestra en el siguiente diagrama.



A partir de esta distribución de posiciones en pares e impares se define

$$\operatorname{rank}_{U_1+U_2}(x) = \begin{cases} 2\operatorname{rank}_{U_1}(x) & x \in U_1 \\ 2\operatorname{rank}_{U_2}(x) + 1 & x \notin U_1 \text{ y } x \in U_2 \end{cases} \tag{3.33}$$

La operación $\operatorname{unrank}_{U_1+U_2}(k)$ se construye como el proceso inverso, de tal forma que si k es par, entonces $k \bmod 2 = 0$ y $k \operatorname{div} 2$ corresponde a la posición en U_1, de lo contrario corresponde a la posición en U_2. Esto se puede corroborar considerando las relaciones de la división euclidiana (2.6) con divisor 2.

$$\operatorname{unrank}_{U_1+U_2}(k) = \begin{cases} \operatorname{unrank}_{U_1}(k \operatorname{div} 2) & k \bmod 2 = 0 \\ \operatorname{unrank}_{U_2}(k \operatorname{div} 2) & k \bmod 2 = 1 \end{cases} \tag{3.34}$$

Ya que en todos los casos se define unrank como inversa de rank por construcción, las sumas de enumeraciones satisfacen pe 3.3. A continuación se muestra que también satisfacen pe 3.1 y pe 3.2.

Proposición (pe 3.1)**.** *Sea $E = E_1 + E_2$, para todo objeto $x \in E$ existe una posición k tal que si $\operatorname{rank}_E(x) = k$, entonces $0 \leq k < |E|$.*

Prueba. Se aborda la demostración de acuerdo a las cardinalidades de los operandos. Cuando $|E_1| < \infty$ se considera la pertenencia de x en los operandos de forma independiente.

- Cuando $x \in E_1$ entonces, por pe 3.1 de E_1 se tiene que

$$\begin{aligned} & 0 \leq \operatorname{rank}_{E_1}(x) < |E_1| \\ \Longrightarrow\ & 0 \leq \operatorname{rank}_{E_1}(x) < |E_1| + |E_2| \\ \Longrightarrow\ & 0 \leq \operatorname{rank}_{E_1+E_2}(x) < |E_1| + |E_2| \end{aligned}$$

- En caso contrario, si $x \in E_2$, entonces por pe 3.1 de E_2 se tiene

$$\begin{aligned} & 0 \leq \operatorname{rank}_{E_2}(x) < |E_2| \\ \Longrightarrow\ & |E_1| \leq |E_1| + \operatorname{rank}_{E_2}(x) < |E_1| + |E_2| \\ \Longrightarrow\ & 0 \leq |E_1| + \operatorname{rank}_{E_2}(x) < |E_1| + |E_2| \\ \Longrightarrow\ & 0 \leq \operatorname{rank}_{E_1+E_2}(x) < |E_1| + |E_2| \end{aligned}$$

Para el caso en que $|E_1| = \infty$ pero $|E_2| < \infty$ se sigue el mismo razonamiento al caso anterior pero invirtiendo los roles de E_1 y E_2.

Finalmente, cuando $|E_1| = |E_2| = \infty$ se consideran de forma independiente cuando $x \in E_1$ y $x \in E_2$.

- Cuando $x \in E_1$ entonces

$$\begin{aligned} & 0 \leq \operatorname{rank}_{E_1}(x) \\ \Longrightarrow\ & 0 \leq 2 \operatorname{rank}_{E_1}(x) \\ \Longrightarrow\ & 0 \leq \operatorname{rank}_{E_1+E_2}(x) \end{aligned}$$

- Cuando $x \in E_2$ entonces

$$\begin{aligned} & 0 \leq \operatorname{rank}_{E_2}(x) \\ \Longrightarrow\ & 0 \leq 2 \operatorname{rank}_{E_1}(x) \\ \Longrightarrow\ & 1 \leq 2 \operatorname{rank}_{E_1}(x) + 1 \\ \Longrightarrow\ & 0 \leq \operatorname{rank}_{E_1+E_2}(x) \end{aligned}$$

□

Proposición (pe 3.2). *Sea $E = E_1 + E_2$, para toda posición k donde $0 \leq k < |E|$, existe un objeto x tal que si* $\operatorname{unrank}_E(k) = x$, *entonces $x \in E$.*

Prueba. Cuando $|E_1| < \infty$ se consideran los siguientes casos

- Cuando $k < |E_1|$ entonces el objeto calculado es $x = \text{unrank}_{E_1}(k)$, ya que $x \in E_1$ se concluye que $x \in E$.
- Cuando $k \geq |E_1|$ entonces el objeto calculado es $x = \text{unrank}_{E_2}(k - |E_1|)$. Para determinar que $x \in E$ se debe mostrar que $x \in E_2$. Por pe (3.2) de E_2 esto se cumple cuando $0 \leq k - |E_1| < |E_2|$. Si E_2 es un operando no-acotado esta desigualdad se satisface trivialmente, de lo contrario se sigue que

$$\begin{aligned} & |E_1| \leq k < |E_1| + |E_2| \\ \implies & 0 \leq k - |E_1| < |E_2| \end{aligned}$$

Cuando $|E_1| = \infty$ pero $|E_2| < \infty$ se considera el caso anterior invirtiendo los roles de E_1 y E_2. Finalmente, cuando $|E_1| = |E_2| = \infty$ se abordan los casos a partir del valor del residuo k mod 2.

- Cuando $k \bmod 2 = 0$, existe un natural k' tal que $k = 2k'$, por lo tanto $\text{unrank}_E(2k') = \text{unrank}_{E_1}(k') \in E_1$.
- Cuando $k \bmod 2 = 1$, existe un natural k' tal que $k = 2k' + 1$, por lo tanto $\text{unrank}_E(2k' + 1) = \text{unrank}_{E_2}(k') \in E_2$.

□

En la tabla 3.5 se muestran las primeras posiciones de cuatro enumeraciones construidas con el combinador de suma, cada una es representativa de los casos descritos previamente.

Así como el producto puede definirse como una composición de parjeas de enumeraciones, se puede definir la unión como una composición de sumas de enumeraciones.

$$\text{union}(E_1, E_2, \ldots, E_{n-1}, E_n) = E_1 + (E_2 + \cdots + (E_{n+1} + (E_n + \emptyset)) \ldots).$$

En el contexto de la enumeración de IMP, se puede utilizar la unión para enumerar categorías sintácticas compuestas de varios tipos de expresiones. Por ejemplo, la enumeración de programas se puede construir como

$$\begin{aligned} &\text{union}(\text{K}(\mathsf{skip}), \text{product}(\text{K}(\mathsf{set}), \mathbf{X}, \mathbf{A}), \\ &\qquad \text{product}(\text{K}(\mathsf{while}), \mathbf{B}, \mathbf{P}), \text{product}(\text{K}(\mathsf{conc}), \mathbf{P}, \mathbf{P}), \\ &\qquad \text{product}(\text{K}(\mathsf{if}), \mathbf{B}, \mathbf{P}, \mathbf{P})) \end{aligned}$$

Nat(1, 3) + Bits(2)	
k	$\text{unrank}_E(k)$
0	1
1	2
2	3
3	00
4	01
5	10
6	11

Bits(2) + Nat(1, 3)	
k	$\text{unrank}_E(k)$
0	00
1	01
2	10
3	11
4	1
5	2
6	3

Nat(1, 2) + Bits	
k	$\text{unrank}_E(k)$
0	1
1	2
2	ε
3	0
4	1
5	00
6	01
7	10
8	11
9	000

Nat + Bits	
k	$\text{unrank}_E(k)$
0	0
1	ε
2	1
3	0
4	2
5	1
6	3
7	00
8	4
9	01

Tabla 3.5: Objetos en las primeras posiciones de enumeraciones de suma.

Al igual que el combinador producto como composición de parejas, la definición de unión como composición de sumas puede resultar en una distribución de posiciones sesgada. Considerando los rangos de exploración en el ejemplo de programas, aproximadamente la mitad de las posiciones son asociadas a asignaciones de memoria, mientras que la otra mitad es asociada al resto de las expresiones. Esto resulta en que un cuarto de las posiciones sean asociadas a iteraciones while, un octavo son asociadas a concatenaciones y el octavo restante a condicionales if.

En la figura 3.8 se muestra la gráfica de los rangos de exploración de una unión de tres enumeraciones no-acotadas E_1, E_2 y E_3 de acuerdo a la definición basada en composición de sumas. Para cualquier posición k, el rango de exploración de E_1 es aproximadamente el doble que el rango de exploración de E_2 y E_3.

Este comportamiento proviene de asociar las posiciones pares al operando E_1 y las posiciones impares al resto, de tal manera que la $(2n)$-ésima posición impar corresponde a E_2 y la $(2n+1)$-ésima posición impar corresponde a E_3.

Para obtener una unión sin sesgo en la distribución de posiciones para operandos no-acotados se utiliza una definición alternativa de union. Sean $U_1, \ldots, U_n$ enumeraciones no-acotadas el combinador uunion$(U_1, \ldots, U_n)$, llamado *unión no-acotada*, construye una enumeración que distribuye posiciones de forma equitativa entre los operandos. En las definiciones relacionadas a este combinador se considera que $U = \text{uunion}(U_1, \ldots, U_n)$.

Cuando se construye una unión no-acotada con cero operandos se considera

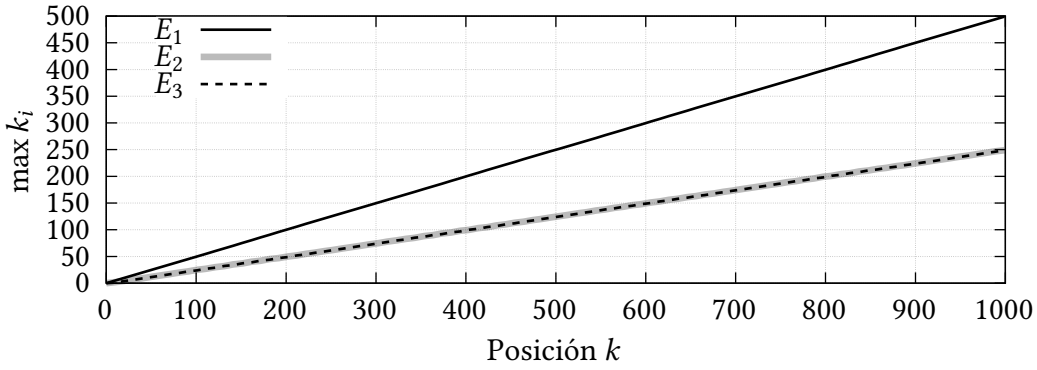


Figura 3.8: Comparación de sesgos para unión de tres enumeraciones.

que uunion() = ∅. De lo contrario, la cardinalidad y la pertenencia de una unión no-acotada con al menos un operando se define como

$$|U| = \infty \tag{3.35}$$

$$x \in U = x \in U_1 \vee \cdots \vee x \in U_n \tag{3.36}$$

Para definir funciones rank y unrank bajo estas consideraciones se utiliza la división euclidiana con divisor n, de tal forma que el residuo $k \bmod n$ determina el operando al que corresponde la posición $k \operatorname{div} n$.

$$\operatorname{rank}_U(x) = n \operatorname{rank}_{U_{i+1}}(x) + i \qquad x \in U_{i+1} \tag{3.37}$$

$$\operatorname{unrank}_U(k) = \operatorname{unrank}_{U_{i+1}}(k \operatorname{div} n) \qquad i = k \bmod n \tag{3.38}$$

Las enumeraciones construidas por uunion satisfacen las propiedades del protocolo de enumeración, al basarse en la división euclidiana 2.6 y en las enumeraciones de sus operandos.

Ahora se considera la unión de enumeraciones union$(E_1, \ldots, E_n)$ cuando cada operando E_i puede ser acotado o no-acotado. Al igual que la unión no-acotada, cuando la unión general no tiene operandos se define union() = ∅. Por otro lado, si alguno de los operandos tiene cardinalidad cero, se considera equivalente a ∅, por lo que puede ser omitido de la unión.

Cuando la unión involucra al menos un operando y cada uno de ellos tiene cardinalidad mayor a cero se utiliza tanto el combinador suma como la unión no-acotada. El primer paso para combinar estos dos operadores es categorizar los operandos en n_1 enumeraciones acotadas B_j y n_2 enumeraciones no-acotadas U_r

de acuerdo al algoritmo de la figura 3.5. A diferencia del procedimiento equivalente para product, no es necesario que el orden de los operandos se preserve, por lo que las variables b_j y u_r no son involucradas en los algoritmos de enumeración.

Una vez que los operandos acotados y no-acotados han sido identificados se simplifica la definición del combinador union a partir de dos casos. Cuando $n_1 = 0$ todos los operandos son no-acotados, por lo que se define

$$\text{union}(E_1, \ldots, E_n) = \text{uunion}(E_1, \ldots, E_n).$$

Cuando $n_2 = 0$ todos los operandos son acotados, por lo que se define

$$\text{union}(E_1, \ldots, E_n) = E_1 + (\cdots + (E_n + \emptyset) \ldots).$$

De lo contrario se tiene una unión con al menos un operando acotado de cardinalidad mayor a cero y un operando no-acotado, por lo que se define

$$\text{union}(E_1, \ldots, E_n) = B_1 + (\cdots + (B_{n_1} + \text{uunion}(U_1, \ldots, U_{n_2})) \ldots).$$

Ya que tanto la suma como uunion satisfacen las propiedades de enumerabilidad, union también las satisface.

Para ejemplificar la relación entre posiciones y objetos enumerados por una unión se considera la enumeración de expresiones aritméticas **A** y expresiones booleanas **B** definidas como

$$\begin{aligned}
\mathbf{A} = \text{union}(&\text{Nat},\\
&\text{product}(\text{K}(\mathsf{loc}), \text{Nat}),\\
&\text{product}(\text{K}(\mathsf{add}), \mathbf{A}, \mathbf{A}),\\
&\text{product}(\text{K}(\mathsf{sub}), \mathbf{A}, \mathbf{A}),\\
&\text{product}(\text{K}(\mathsf{mul}), \mathbf{A}, \mathbf{A}))
\end{aligned}$$

$$\begin{aligned}
\mathbf{B} = \text{union}(&\text{K}(\mathsf{true}), \text{K}(\mathsf{false}),\\
&\text{product}(\text{K}(\mathsf{equal}), \mathbf{A}, \mathbf{A}),\\
&\text{product}(\text{K}(\mathsf{less}), \mathbf{A}, \mathbf{A}),\\
&\text{product}(\text{K}(\mathsf{and}), \mathbf{B}, \mathbf{B}),\\
&\text{product}(\text{K}(\mathsf{or}), \mathbf{B}, \mathbf{B}),\\
&\text{product}(\text{K}(\mathsf{not}), \mathbf{B}))
\end{aligned}$$

La tabla 3.6 muestra los objetos correspondientes a **A** y **B** para las primeras 35 posiciones. Se utiliza la notación de lista usual $(\mathsf{R}, C_1, \ldots, C_n)$ para representar al árbol de raíz R con hijos C_i en lugar de la notación de árbol $(\mathsf{R}\ C_1\ \ldots\ C_n)$ usada en el capítulo 2.

k	$\text{unrank}_{\mathbf{A}}(k)$	$\text{unrank}_{\mathbf{B}}(k)$
0	0	true
1	(loc, 0)	false
2	(add, 0, 0)	(equal, 0, 0)
3	(sub, 0, 0)	(less, 0, 0)
4	(mul, 0, 0)	(and, true, true)
5	1	(or, true, true)
6	(loc, 1)	(not, true)
7	(add, 0, (loc, 0))	(equal, 0, (loc, 0))
8	(sub, 0, (loc, 0))	(less, 0, (loc, 0))
9	(mul, 0, (loc, 0))	(and, true, false)
10	2	(or, true, false)
11	(loc, 2)	(not, false)
12	(add, (loc, 0), 0)	(equal, (loc, 0), 0)
13	(sub, (loc, 0), 0)	(less, (loc, 0), 0)
14	(mul, (loc, 0), 0)	(and, false, true)
15	3	(or, false, true)
16	(loc, 3)	(not, (equal, 0, 0))
17	(add, 0, (add, 0, 0))	(equal, 0, (add, 0, 0))
18	(sub, 0, (add, 0, 0))	(less, 0, (add, 0, 0))
19	(mul, 0, (add, 0, 0))	(and, true, (equal, 0, 0))
20	4	(or, true, (equal, 0, 0))
21	(loc, 4)	(not, (less, 0, 0))
22	(add, (loc, 0), (loc, 0))	(equal, (loc, 0), (loc, 0))
23	(sub, (loc, 0), (loc, 0))	(less, (loc, 0), (loc, 0))
24	(mul, (loc, 0), (loc, 0))	(and, false, false)
25	5	(or, false, false)
26	(loc, 5)	(not, (and, true, true))
27	(add, (add, 0, 0), 0)	(equal, (add, 0, 0), 0)
28	(sub, (add, 0, 0), 0)	(less, (add, 0, 0), 0)
29	(mul, (add, 0, 0), 0)	(and, (equal, 0, 0)true)
30	6	(or, (equal, 0, 0)true)
31	(loc, 6)	(not, (or, true, true))
32	(add, 0, (sub, 0, 0))	(equal, 0, (sub, 0, 0))
33	(sub, 0, (sub, 0, 0))	(less, 0, (sub, 0, 0))
34	(mul, 0, (sub, 0, 0))	(and, true, (less, 0, 0))

Tabla 3.6: Objetos enumeradas por **A** y **B**.

3.8 Clausura de Kleene de enumeraciones

En esta sección se define un combinador llamado *clausura de Kleene*, debido a su relación con la operación del mismo nombre usada en lenguajes formales.

La clausura o estrella de Kleene de un conjunto de cadenas L es denotada L^* y representa el conjunto de aquellas cadenas que pueden ser formadas al tomar la concatenación de cualquier cantidad de cadenas de L, posiblemente con repeticiones [9].

A partir de esta definición se define el combinador kleene(E), el cuál enumera todas las listas de objetos en E, esto incluye la lista vacía, las listas que contienen un componente de E, las listas con dos componentes de E, etc.

Si el combinador union fuera definido para una infinidad de operandos, se pudiera definir kleene como

$$\begin{aligned}\text{kleene}(E) = \text{union}(&\text{product}(),\\ &\text{product}(E),\\ &\text{product}(E,E),\\ &\text{product}(E,E,E),\\ &\ldots)\end{aligned}$$

Ya que todos los combinadores son definidos para una cantidad finita de operandos, se puede modificar la definición anterior utilizando recursividad.

$$\text{kleene}(E) = \text{K}(\text{null}) + E \times \text{kleene}(E). \tag{3.39}$$

De tal forma que un objeto en kleene(E) es una lista vacía, o bien una pareja cuyo primer elemento es enumerado por E y cuyo resto es una lista enumerada por kleene(E).

Para mostrar que la definición (3.39) corresponde a una enumeración de acuerdo al protocolo, se debe mostrar que su cardinalidad puede ser calculada y que satisface pe 3.5 y pe 3.6.

Utilizando pe 3.4 como hipótesis se calcula la cardinalidad de kleene(E) a

partir de la cardinalidad de E analizando tres casos. Cuando $|E| = 0$ se tiene que

$$\begin{aligned} |\,\mathrm{kleene}(E)| &= |\,\mathrm{K}(\mathsf{null}) + E \times \mathrm{kleene}(E)| \\ &= |\,\mathrm{K}(\mathsf{null}) + \emptyset| && \text{por (3.13)} \\ &= |\,\mathrm{K}(\mathsf{null})| && \text{por (3.28)} \\ &= 1 \end{aligned}$$

Cuando E es una enumeración acotada con cardinalidad mayor a cero se tiene

$$\begin{aligned} |\,\mathrm{kleene}(E)| &= |\,\mathrm{K}(\mathsf{null}) + E \times \mathrm{kleene}(E)| \\ &= |\,\mathrm{K}(\mathsf{null})| + |E \times \mathrm{kleene}(E)| && \text{por (3.26)} \\ &= 1 + |E| \times |\,\mathrm{kleene}(E)| && \text{por (3.12)} \\ &= 1 + |E| \times \infty && \text{por pe 3.4} \\ &= \infty \end{aligned}$$

Cuando E es una enumeración no-acotada, su cardinalidad es ∞ y por el razonamiento anterior se tiene que $|\,\mathrm{kleene}(E)| = \infty$. Estos cálculos son consistentes con la definición de la clausura de Kleene en lenguajes formales donde el único lenguaje resultante con cardinalidad finita es $\emptyset^* = \{\varepsilon\}$.

Proposición (pe 3.5)**.** *Sea* $E = \mathrm{kleene}(E')$*, para toda lista* $\ell \in E$ *y toda aparición* $\mathrm{rank}_E(\ell')$ *en el cálculo de* $\mathrm{rank}_E(\ell)$*, la lista* ℓ' *es estructuralmente menor a* ℓ*.*

Prueba. Cuando $\ell = \mathsf{null}$ se tiene que $\ell \in \mathrm{K}(\mathsf{null})$, por lo que la posición resultante es $\mathrm{rank}_{\mathrm{K}(\mathsf{null})}(\mathsf{null})$. Ya que no hay apariciones internas de rank_E se satisface la propiedad por vacuidad.

Por otro lado, cuando $\ell = (x\ .\ \ell')$ con $x \in E'$ y $\ell' \in E$, se tiene que

$$\begin{aligned} \mathrm{rank}_E((x\ .\ \ell')) &= |\,\mathrm{K}(\mathsf{null})| + \mathrm{rank}_{E' \times E}((x\ .\ \ell')) \\ &= 1 + \mathrm{rank}_{E' \times E}((x\ .\ \ell')) \end{aligned}$$

Para proceder con este cálculo se debe analizar los casos en que E' es acotada y no-acotada.

- Cuando $|E'| < \infty$ se tiene que la posición correspondiente es

$$1 + |E'|\,\mathrm{rank}_E(\ell') + \mathrm{rank}_{E'}(x).$$

La lista ℓ' es el resto de ℓ, por lo tanto es estructuralmente menor a ℓ y se satisface la propiedad.

- Cuando $|E'| = \infty$ se tiene que la posición correspondiente es

$$1 + \mathrm{rank}_2((\mathrm{rank}_{E'}(x), \mathrm{rank}_E(\ell'))).$$

La lista ℓ' es el resto de ℓ, por lo tanto es estructuralmente menor a ℓ y se satisface la propiedad.

□

Proposición (pe 3.6)**.** *Sea* $E = \mathrm{kleene}(E')$*, para toda posición* k *donde* $0 \leq k < |E|$ *y toda aparición* $\mathrm{unrank}_E(k')$ *en el cálculo de* $\mathrm{unrank}_E(k)$*, la posición* k' *es estrictamente menor a* k*.*

Prueba. Cuando $k = 0$ se tiene que

$$\begin{aligned}\mathrm{unrank}_E(0) &= \mathrm{unrank}_{\mathrm{K(null)}+E'\times E}(0)\\ &= \mathrm{unrank}_{\mathrm{K(null)}}(0)\end{aligned}$$

ya que no hay apariciones internas de unrank_E se satisface la propiedad por vacuidad. Por otro lado, cuando $k = k' + 1$, se tiene que $|E| = \infty$

$$\begin{aligned}\mathrm{unrank}_E(k'+1) &= \mathrm{unrank}_{\mathrm{K(null)}+E'\times E}(k'+1)\\ &= \mathrm{unrank}_{E'\times E}(k'+1-|\,\mathrm{K(null)}|)\\ &= \mathrm{unrank}_{E'\times E}(k')\end{aligned}$$

Se abordan los casos dependiendo de si E' es acotada o no-acotada.

- Cuando $|E'| < \infty$ se tiene que

$$\mathrm{unrank}_{E'\times E}(k') = (\mathrm{unrank}_{E'}(k' \bmod |E'|)\ .\ \mathrm{unrank}_E(k' \operatorname{div} |E'|)).$$

Ya que $k' < k$, el cociente $k' \operatorname{div} |E'|$ también es estrictamente menor a k por lo que se satisface la propiedad.

- Cuando $|E'| = \infty$ se considera $(k_1, k_2) = \mathrm{unrank}_2(k')$, por lo tanto

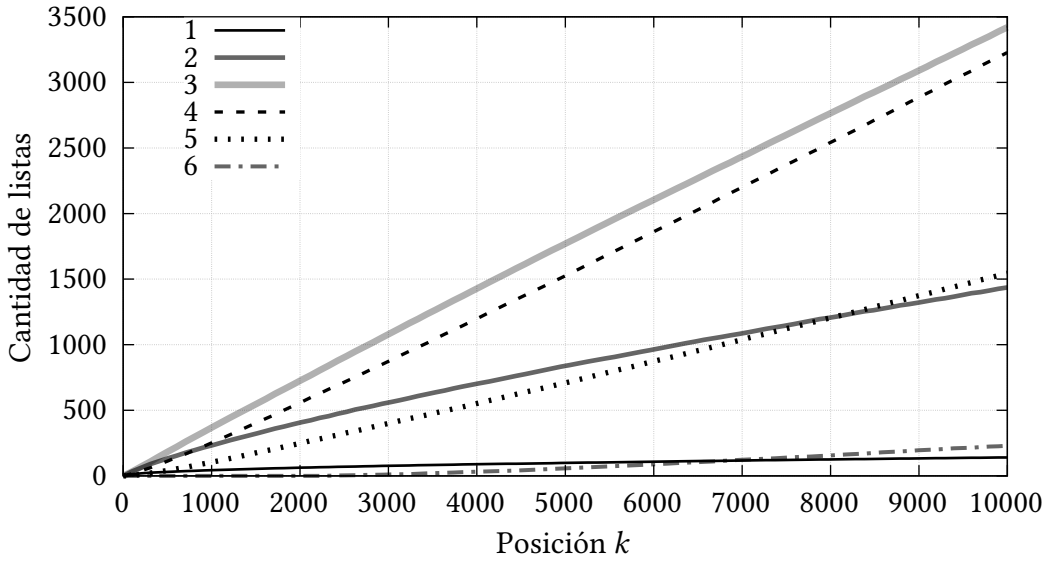
$$\mathrm{unrank}_{E'\times E}(k') = (\mathrm{unrank}_{E'}(k_1)\ .\ \mathrm{unrank}_E(k_2)).$$

Por la propiedad (2.7) se tiene que $k_2 \leq k'$ y ya que $k' < k$, entonces $k_2 < k$ por transitividad.

□

Una desventaja de definir kleene con la ecuación (3.39) es que la distribución de tamaños de listas en las enumeraciones resultantes es inusual. En la figura 3.9 se muestra la cantidad de listas enumeradas de acuerdo a su tamaño para las primeras 10^4 posiciones para la enumeración kleene(U) con U siendo una enumeración no-acotada.

Figura 3.9: Cantidad de listas por tamaño.



Como se observa en esta gráfica, las listas con tres componentes ocupan la mayor cantidad de posiciones, mientras las listas con un componente ocupan la menor cantidad de posiciones. Por otro lado, las listas con cuatro componentes ocupan más posiciones que las listas con cinco y seis componentes. Al igual que en la construcción del producto y unión de enumeraciones, se opta por reducir la cantidad de sesgos sobre la simplicidad de las definiciones.

En el caso de la clausura de Kleene de una enumeración no-acotada, para cada tamaño de lista hay una infinidad de objetos, por lo que no es posible distribuir las posiciones de forma equitativa sobre todos los tamaños. La estrategia que se

desarrolla en este trabajo es en construir un operador donde la distribución de posiciones sea más *predecible* que la definición (3.39).

Primero se considera la definición de kleene(E) cuando E es una enumeración acotada y posteriormente cuando E es una enumeración no-acotada. Como caso particular se considera

$$|E| = 0 \implies \text{kleene}(E) = \text{K}(\text{null}) \tag{3.40}$$

En la sección 3.4 se presenta una enumeración de todas las cadenas binarias, en lenguajes formales el conjunto de todas las cadenas binarias se puede definir utilizando la clausura de Kleene $\{0,1\}^*$. Por lo que una enumeración similar a Bits pero basada en listas de bits en lugar de cadenas de bits es kleene$(\text{Nat}(0,1))$.

Sea B una enumeración no-acotada con cardinalidad mayor a cero, se utiliza el razonamiento detrás de la definición de Bits para construir kleene(B).

Así como Bits se define en términos de Bits(n), la clausura con operando acotado se define en términos del combinador *potencia* B^n, el cuál enumera listas de n objetos en B.

En lugar de utilizar el combinador de parejas para construir un producto se propone utilizar un orden diferente procesando los componentes de las listas de derecha a izquierda al igual que en Bits. Sea ℓ una lista y x un objeto, se denota $\ell \cdot x$ a la lista que contiene los componentes de ℓ concatenados con x al final.

La operación de cardinalidad se define como una relación de recurrencia con respecto al exponente.

$$\begin{aligned} |B^0| &= 1 \\ |B^{n+1}| &= |B| \times |B^n| \end{aligned} \tag{3.41}$$

De forma alternativa se puede utilizar la solución a la recurrencia $|B^n| = |B|^n$.

La operación de pertenencia se define de forma inductiva sobre la estructura de las listas en base a la concatenación por la derecha.

$$\begin{aligned} \text{null} &\in B^0 \\ \ell \cdot x \in B^{n+1} &= \ell \in B^n \text{ y } x \in B \end{aligned} \tag{3.42}$$

La asociación de listas y posiciones en B^{n+1} se define en términos de las asociaciones en B^n, ya que por cada lista de tamaño n existen $|B|$ listas de tamaño $n+1$. Se propone considerar que $\ell_1 \cdot x_1$ precede a $\ell_1 \cdot x_2$ cuando ℓ_1 precede a ℓ_2 en B^n, o bien, que $\ell \cdot x_1$ precede a $\ell \cdot x_2$ cuando x_1 precede a x_2 en B.

A partir de este planteamiento se define la operación rank_{B^n} utilizando el

método de división euclidiana (2.6) en el caso inductivo y considerando a ℓ como una lista de tamaño n.

$$\begin{aligned} \operatorname{rank}_{B^0}(\mathsf{null}) &= 0 \\ \operatorname{rank}_{B^{n+1}}(\ell \cdot x) &= |B| \operatorname{rank}_{B^n}(\ell) + \operatorname{rank}_B(x) \end{aligned} \tag{3.43}$$

Se define la operación $\operatorname{unrank}_{B^n}$ como inversa de $\operatorname{rank}_{B^n}$ por construcción.

$$\begin{aligned} \operatorname{unrank}_{B^0}(0) &= \mathsf{null} \\ \operatorname{unrank}_{B^{n+1}}(k) &= \operatorname{unrank}_{B^n}(k \operatorname{div} |B|) \cdot \operatorname{unrank}_B(k \bmod |B|) \end{aligned} \tag{3.44}$$

Estas definiciones son similares a las de productos basados en parejas de la sección 3.6 pero con los componentes de las listas en orden inverso.

Para combinar las enumeraciones B^n para todo tamaño de lista n se considera un orden donde las listas de tamaño n preceden a las de tamaño $n+1$ y cuando dos listas tienen la misma cantidad de componentes n, su orden es determinado por B^n.

Para incorporar este orden en la definición de la clausura de Kleene se utiliza la cardinalidad del operando B para calcular la cantidad de listas con menos de n componentes.

$$\sum_{i=0}^{n-1} |B^i| = \sum_{i=0}^{n-1} |B|^i = \frac{|B|^n - 1}{|B| - 1}.$$

Sea ℓ una lista enumerada por B^n, se define su posición en $\operatorname{kleene}(B)$ como

$$\operatorname{rank}_E(\ell) = \operatorname{rank}_{B^n}(\ell) + \frac{|B|^n - 1}{|B| - 1}.$$

En el contexto de la clausura de Kleene de una enumeración acotada B se denota $E = \operatorname{kleene}(B)$.

Al igual que en la construcción de Bits, se plantea una simplificación de la

definición que no depende de rank_{B^n}. Sea $\ell \in B^n$ y $x \in B$, se tiene que

$$\begin{aligned}
\text{rank}_E(\ell \cdot x) &= \text{rank}_{B^{n+1}}(\ell \cdot x) + \frac{|B|^{n+1} - 1}{|B| - 1} \\
&= |B| \, \text{rank}_{B^n}(\ell) + \text{rank}_B(x) + \frac{|B|^{n+1} - 1}{|B| - 1} \\
&= |B| \, \text{rank}_{B^n}(\ell) + \text{rank}_B(x) + \frac{|B|^{n+1} - 1}{|B| - 1} + \frac{|B|}{|B| - 1} - \frac{|B|}{|B| - 1} \\
&= |B| \, \text{rank}_{B^n}(\ell) + \text{rank}_B(x) + \frac{|B|^{n+1} - |B|}{|B| - 1} + \frac{|B|}{|B| - 1} - \frac{1}{|B| - 1} \\
&= |B| \, \text{rank}_{B^n}(\ell) + \text{rank}_B(x) + |B| \frac{|B|^n - 1}{|B| - 1} + \frac{|B| - 1}{|B| - 1} \\
&= |B| \left(\text{rank}_{B^n}(\ell) + \frac{|B|^n - 1}{|B| - 1} \right) + \text{rank}_B(x) + 1 \\
&= |B| \, \text{rank}_E(\ell) + \text{rank}_B(x) + 1
\end{aligned}$$

por lo que la definición simplificada de la operación rank es

$$\begin{aligned}
\text{rank}_E(\text{null}) &= 0 \\
\text{rank}_E(\ell \cdot x) &= |B| \, \text{rank}_E(\ell) + \text{rank}_B(x) + 1
\end{aligned} \tag{3.45}$$

Para construir la operación inversa unrank_E en el caso inductivo, se consideran las relaciones de la división euclidiana. Sea $k = |B| \, \text{rank}_E(\ell) + \text{rank}_B(x)$ se tiene

$$\begin{aligned}
k \text{ div } |B| = \text{rank}_E(\ell) &\implies \text{unrank}_E(k \text{ div } |B|) = \ell \\
k \text{ mod } |B| = \text{rank}_B(x) &\implies \text{unrank}_B(k \text{ mod } |B|) = x
\end{aligned}$$

Por lo tanto, se define la operación unrank_E sobre todo natural como

$$\begin{aligned}
\text{unrank}_E(0) &= \text{null} \\
\text{unrank}_E(k+1) &= \text{unrank}_E(k \text{ div } |B|) \cdot \text{unrank}_B(k \text{ mod } |B|)
\end{aligned} \tag{3.46}$$

La tabla 3.7 muestra la diferencia entre las primeras posiciones de esta definición de clausura de Kleene y la propuesta inicial de la ecuación (3.39).

Ahora se plantea el problema de construir una clausura de Kleene para enumeraciones no-acotadas con una distribución de tamaños de listas más predecible que (3.39). Sea U una enumeración con cardinalidad $|U| = \infty$. En las siguientes

$B = 0 + 1$ $E_1 = \mathsf{null} + B \times E_1$ $E_2 = \text{kleene}(B)$		
k	$\text{unrank}_{E_1}(k)$	$\text{unrank}_{E_2}(k)$
0	null	null
1	(0)	(0)
2	(1)	(1)
3	$(0, 0)$	$(0, 0)$
4	$(1, 0)$	$(0, 1)$
5	$(0, 1)$	$(1, 0)$
6	$(1, 1)$	$(1, 1)$
7	$(0, 0, 0)$	$(0, 0, 0)$
8	$(1, 0, 0)$	$(0, 0, 1)$
9	$(0, 1, 0)$	$(0, 1, 0)$
10	$(1, 1, 0)$	$(0, 1, 1)$
11	$(0, 0, 1)$	$(1, 0, 0)$
12	$(1, 0, 1)$	$(1, 0, 1)$
13	$(0, 1, 1)$	$(1, 1, 0)$
14	$(1, 1, 1)$	$(1, 1, 1)$

Tabla 3.7: Comparación de orden entre definiciones de kleene.

definiciones se describen procedimientos para enumerar $E = \text{kleene}(U)$.

Primero se considera que al igual que en el caso acotado, la posición cero es asociada a la lista vacía null. En caso contrario se tienen posiciones de la forma $k + 1$ con k natural.

Una forma de resolver el problema de distribuir el resto de las posiciones es utilizando la enumeración de duplas, de tal forma que el sesgo en los tamaños sea determinado por el algoritmo de duplas particular que sea utilizado.

Sea $k+1$ una posición mayor a cero, la pareja de naturales $(k', n) = \text{unrank}_2(k)$ permite asociar la posición $k + 1$ a la k'-ésima lista con $n + 1$ componentes de U. Por lo que el último cálculo consiste en construir la $(n + 1)$-tupla $\text{unrank}_n(k')$ y para cada componente k_i construir la lista con componentes $\text{unrank}_U(k_i)$.

Considerando la enumeración de duplas descrita en [2] e ilustrada en la figura 2.7, ya que no se presentan sesgos considerables entre el primer y segundo componente, se espera que el método anterior enumere tantas posiciones k' como

tamaños de lista $n + 1$.

Como alternativa a estos algoritmos para la enumeración de duplas, se propone construir una enumeración de naturales (k', n) donde el rango de valores para el primer componente sea exponencialmente mayor al rango del segundo componente.

Se denota con rank'_2 y unrank'_2 a las operaciones alternativas de enumeración de duplas. Para definir estas operaciones se propone utilizar una codificación binaria de las posiciones, utilizando el primer cero desde el bit menos significativo como un separador entre componentes. De tal forma que la cantidad de bits consecutivos 1 desde la posición menos significativa es el valor del segundo componente y el resto del número binario a la izquierda del separador es el valor del primer componente.

Para ilustrar esta relación de posiciones y duplas de naturales se considera el siguiente ejemplo.

$$\begin{aligned} k &= 1234567 \\ k \text{ base } 2 &= 10010110101101000111 \\ &= \underbrace{10010110101101000}_{77160}\ 0\ \underbrace{111}_{3} \\ \text{unrank}'_2(k) &= (77160, 3) \end{aligned}$$

Para realizar el proceso inverso, desde una perspectiva de manipulación de bits, se desplaza $n + 1$ bits a la izquierda el valor k' y se cambian los primeros n bits menos significativos a 1. El equivalente numérico a este procedimiento es

$$\text{rank}'_2((k', n)) = 2^{n+1} \times k' + 2^n - 1$$

Al utilizar una representación binaria para el primer componente y una representación unaria para el segundo componente, se espera que el rango de valores de una sea exponencialmente mayor a la otra. Por este motivo, se dice que esta enumeración de duplas tiene *sesgo exponencial*.

A partir de el método propuesto, es claro ver que cada posición tiene asociada una dupla única de naturales ya que cada natural tiene una única representación en base 2. Adicionalmente, se satisfacen las propiedades (2.7) y (2.7) ya que

$$\begin{aligned} 2^{n+1} \times k' &\leq \text{rank}'_2((k', n)) \\ 2^n - 1 &\leq \text{rank}'_2((k', n)) \end{aligned}$$

La tabla 3.8 muestra las primeras posiciones de esta enumeración alternativa.

$\text{unrank}'_2(k) = (k', n)$			
k	k base 2	k'	n
0	0	0	0
1	1	0	1
2	10	1	0
3	11	0	2
4	100	2	0
5	101	1	1
6	110	3	0
7	111	0	3
8	1000	4	0
9	1001	2	1
10	1010	5	0
11	1011	1	2
12	1100	6	0
13	1101	3	1
14	1110	7	0

Tabla 3.8: Duplas con sesgo.

En la figura 3.10 se muestran los algoritmos rank_E y unrank_E utilizando la enumeración alternativa de duplas para distribuir las posiciones sobre los tamaños de lista y la enumeración de n-tuplas [2] para obtener los componentes de las listas de tamaño n.

En la figura 3.11 se muestra la distribución de tamaños de listas siguiendo un procedimiento similar al de la figura 3.9 pero utilizando el algoritmo alternativo de duplas.

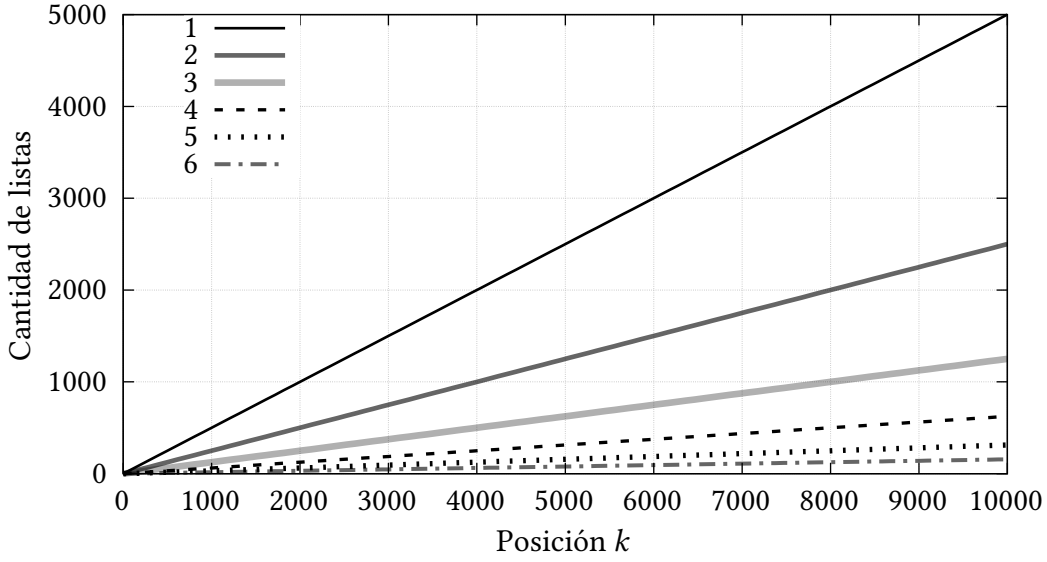
Como se observa en la gráfica, la mitad de las posiciones corresponden a listas con un componente, un cuarto a listas con dos componentes, un octavo a listas de tres componentes, etc. De tal forma que en general, para un rango de posiciones en $[0, k)$ se espera que aproximadamente 2^{-n} listas tengan tamaño n.

$\text{rank}_{\text{kleene}(U)}(\ell) =$
  if $\ell = \text{null}$ **return** 0
  $(x_1, \ldots, x_n, x_{n+1}) \leftarrow \ell$
  for i **from** 1 **upto** $n+1$
    $k_i \leftarrow \text{rank}_U(x_i)$
  $k' \leftarrow \text{rank}_{n+1}((k_1, \ldots, k_n, k_{n+1}))$
  return $1 + \text{rank}'_2((k', n))$

$\text{unrank}_{\text{kleene}(U)}(k) =$
  if $k = 0$ **return** null
  $(k', n) \leftarrow \text{unrank}'_2(k-1)$
  $(k_1, \ldots, k_n, k_{n+1}) \leftarrow \text{unrank}_{n+1}(k')$
  for i **from** 1 **upto** $n+1$
    $x_i \leftarrow \text{unrank}_U(k_i)$
  return $(x_1, \ldots, x_n, x_{n+1})$

Figura 3.10: Algoritmos de kleene para enumeraciones no-acotadas.

Figura 3.11: Cantidad de listas por tamaño.



3.9 Enumeración IMP

Para concluir este capítulo, se muestran definiciones de las enumeraciones de IMP descritas en 2 utilizando los combinadores propuestos en las anteriores secciones.

Primero se comienza con la enumeración de naturales **N** definida con la enumeración primitiva de la sección 3.3.

$$\mathbf{N} = \text{Nat}$$

Ahora se plantea la enumeración de localidades de memoria **X**, la cuál consiste de todas las listas de dos componentes donde el primero siempre es loc y el segundo es un natural.

$$\mathbf{X} = \text{product}(\text{K}(\mathsf{loc}), \mathbf{N})$$

Una vez construidas estas enumeraciones básicas, se definen **A**, **B** y **P**. Ya que estas enumeraciones son recursivas, se debe mostrar cómo calcular su cardinalidad y que se satisfacen las propiedades pe 3.5 y pe 3.6.

La enumeración de expresiones aritméticas **A** se define en términos de **N**, **X** y **A** de acuerdo a la siguiente definición.

$$\begin{aligned}\mathbf{A} = \text{union}(&\mathbf{N}, \mathbf{X},\\ &\text{product}(\text{K}(\mathsf{add}), \mathbf{A}, \mathbf{A}),\\ &\text{product}(\text{K}(\mathsf{sub}), \mathbf{A}, \mathbf{A}),\\ &\text{product}(\text{K}(\mathsf{mul}), \mathbf{A}, \mathbf{A}))\end{aligned}$$

Para mostrar que se puede calcular su cardinalidad considerando como hipótesis pe 3.4 se parte de la observación que Nat es no-acotada, por lo tanto también lo son **N** y **X**. En el caso de los productos de la definición se tiene que para todo identificar de operación op se sigue que

$$\begin{aligned}|\,\text{product}(\text{K}(\mathsf{op}), \mathbf{A}, \mathbf{A})| &= |\,\text{K}(\mathsf{op})| \times |\mathbf{A}| \times |\mathbf{A}|\\ &= 1 \times \infty \times \infty\\ &= \infty\end{aligned}$$

Por lo tanto, la unión consiste de operandos no-acotados, resultando en que la cardinalidad de **A** es ∞.

Para mostrar que esta enumeración **A** satisface pe 3.5 se parte de la definición rank de una unión no-acotada (3.37) y se abordan los casos en que un objeto x pertenece a cada uno de los operandos.

Cuando $x \in \mathbf{N}$ o $x \in \mathbf{X}$ el cálculo de su posición no contiene apariciones de $\text{rank}_{\mathbf{A}}$, por lo tanto la propiedad se satisface por vacuidad. De lo contrario se considera el caso cuando x es una suma. Ya que el primer operando es acotado y el resto son no-acotados se tiene que

$$\text{product}(\text{K}(\mathsf{add}), \mathbf{A}, \mathbf{A}) = \text{K}(\mathsf{add}) \times \text{uproduct}(\mathbf{A}, \mathbf{A})$$

Sea $x = (\mathsf{add}, a_1, a_2)$ su posición se calcula como

$$5\left(\text{rank}_{\text{uproduct}(\mathbf{A},\mathbf{A})}((a_1, a_2))\right) + 2 = 5\left(\text{rank}_2(\text{rank}_{\mathbf{A}}(a_1), \text{rank}_{\mathbf{A}}(a_2))\right) + 2$$

Ya que a_1 es el segundo elemento de x y a_2 el tercer elemento de x, ambas son estructuralmente menores a x, por lo que se satisface pe 3.5.

Cuando x es una resta o una multiplicación, se sigue el mismo razonamiento ya que únicamente cambia el segundo componente de la suma en el cálculo de la posición, para sumas es 2, para restas es 3 y para multiplicaciones es 4.

Para mostrar que la enumeración $\mathbf{A}$ satisface pe 3.5 se considera un análisis de casos a partir de una posición cualquiera k. Cuando $k = 5k'$ o $k = 5k' + 1$ la posición corresponde a naturales o localidades respectivamente, ya que en estos cálculos no aparece $\text{unrank}_{\mathbf{A}}$ se satisface trivialmente la propiedad.

Cuando $k = 5k' + 2$ la posición corresponde a la k'-ésima suma, de acuerdo a la ecuación (3.24) se tiene que si $\text{unrank}_2(k') = (k_1, k_2)$ entonces

$$\text{unrank}_{\mathbf{A}}(5k' + 2) = (\mathsf{add}, \text{unrank}_{\mathbf{A}}(k_1), \text{unrank}_{\mathbf{A}}(k_2))$$

Ya que k_1 y k_2 no son mayores a k' y $k' < 5k' + 2$, se satisface la propiedad para estas dos apariciones de $\text{unrank}_{\mathbf{A}}$.

Cuando $k = 5k' + 3$ o $k = 5k' + 4$ la posición corresponde a la k'-ésima resta o k'-ésima multiplicación respectivamente, ya que estas expresiones tienen una estructura similar a la suma, el razonamiento para mostrar la propiedad es similar.

La enumeración de expresiones booleanas $\mathbf{B}$ se define en términos de $\mathbf{A}$ y $\mathbf{B}$ de acuerdo a la siguiente definición.

$$\begin{aligned}\mathbf{B} = \text{union}(&\text{K}(\mathsf{true}), \text{K}(\mathsf{false}),\\ &\text{product}(\text{K}(\mathsf{equal}), \mathbf{A}, \mathbf{A}),\\ &\text{product}(\text{K}(\mathsf{less}), \mathbf{A}, \mathbf{A}),\\ &\text{product}(\text{K}(\mathsf{and}), \mathbf{B}, \mathbf{B}),\\ &\text{product}(\text{K}(\mathsf{or}), \mathbf{B}, \mathbf{B}),\\ &\text{product}(\text{K}(\mathsf{not}), \mathbf{B}))\end{aligned}$$

Para calcular la cardinalidad de esta enumeración se parte de que las enumeraciones constantes tienen cardinalidad 1 y las comparaciones aritméticas cardinalidad ∞. Por la hipótesis pe 3.4 se tiene que la cardinalidad de $\mathbf{B}$ en los productos correspondientes a la conjunción, disyunción y negación tienen cardinalidad ∞, por lo tanto

$$|\mathbf{B}| = 1 + 1 + \infty + \infty + \infty + \infty + \infty = \infty.$$

Para mostrar que se satisfacen las propiedades pe 3.5 y pe 3.6 se parte de la observación que únicamente los cálculos asociados a los tres últimos operandos tienen apariciones de $\text{rank}_{\mathbf{B}}$ y $\text{unrank}_{\mathbf{B}}$.

Sea $x = (\mathsf{and}, b_1, b_2)$, ya que x corresponde al tercer operando no-acotado se tiene que su posición es de la forma

$$2 + 5\,\text{rank}_{\text{uproduct}(\mathbf{B},\mathbf{B})}((b_1, b_2)) + 2 = 5\,(\text{rank}_2(\text{rank}_{\mathbf{B}}(b_1), \text{rank}_{\mathbf{B}}(b_2))) + 4$$

Las expresiones b_1 y b_2 son estructuralmente menores a x, por lo tanto se satisface pe 3.5. Este razonamiento es el mismo para las disyunciones ya que consideran el cálculo

$$2 + 5\,\text{rank}_{\text{uproduct}(\mathbf{B},\mathbf{B})}((b_1, b_2)) + 3 = 5\,(\text{rank}_2(\text{rank}_{\mathbf{B}}(b_1), \text{rank}_{\mathbf{B}}(b_2))) + 5$$

Por otro lado, las posiciones asociadas a conjunciones tienen la forma $k = 5k' + 4$, de acuerdo a la ecuación (3.24) se tiene que si $\text{unrank}_2(k') = (k_1, k_2)$ entonces

$$\text{unrank}_{\mathbf{B}}(5k' + 4) = (\mathsf{and}, \text{unrank}_{\mathbf{B}}(k_1), \text{unrank}_{\mathbf{B}}(k_2))$$

Ya que k_1 y k_2 no son mayores a k' y $k' < 5k' + 4$, se satisface la propiedad para estas dos apariciones de $\text{unrank}_{\mathbf{B}}$. Para las disyunciones se utiliza el mismo razonamiento a partir de la ecuación

$$\text{unrank}_{\mathbf{B}}(5k' + 5) = (\mathsf{or}, \text{unrank}_{\mathbf{B}}(k_1), \text{unrank}_{\mathbf{B}}(k_2))$$

En el caso de las negaciones se tiene que la posición calculada para una expresión de la forma $x = (\mathsf{not}, b)$ es

$$\text{rank}_{\mathbf{B}}(x) = 2 + 5\,\text{rank}_{\mathbf{B}}(b) + 4$$

Ya que b es el segundo elemento de la lista, es estructuralmente menor a x y por

lo tanto satisface pe 3.5. Por otro lado, las posiciones asociadas a negaciones son de la forma $k = 5k' + 6$, el cálculo para construir la expresión correspondiente es

$$\text{unrank}_{\mathbf{B}}(5k' + 6) = (\text{not}, \text{unrank}_{\mathbf{B}}(k')$$

y se tiene que $k' < 5k' + 6$, por lo que pe 3.6 también se satisface.

Finalmente, la enumeración de programas **P** se define en términos de **X**, **A**, **B** y **P** de acuerdo a la siguiente definición.

$$\begin{aligned}\mathbf{P} = \text{union}(&\text{K}(\text{skip}),\\ &\text{product}(\text{K}(\text{set}), \mathbf{X}, \mathbf{A}),\\ &\text{product}(\text{K}(\text{while}), \mathbf{B}, \mathbf{P}),\\ &\text{product}(\text{K}(\text{conc}), \mathbf{P}, \mathbf{P}),\\ &\text{product}(\text{K}(\text{if}), \mathbf{B}, \mathbf{P}, \mathbf{P}))\end{aligned}$$

El cálculo de su cardinalidad y la demostración de que satisface las propiedades pe 3.5 y pe 3.6 es similar a que en las expresiones booleanas.

De forma general se observa que si alguno de los operandos de la unión es no-acotado, entonces la cardinalidad de la unión es ∞.

Para analizar que las productos que tienen a **P** como operando funcionan de manera correcta, basta con observar que los productos por definición de pertenencia distribuyen sobre sus operandos partes de la expresión original, por lo tanto las apariciones de $\text{rank}_{\mathbf{P}}$ siempre usan un argumento que es estructuralmente menor.

A su vez, una posición k asociada a un operando que incluye **P** siempre es de la forma $k = 4k' + i$ con $i > 0$, por lo tanto al usar argumentos menores o iguales a k' como argumentos de $\text{unrank}_{\mathbf{P}}$ garantiza que se satisface pe 3.6.

Capítulo 4

Aplicaciones

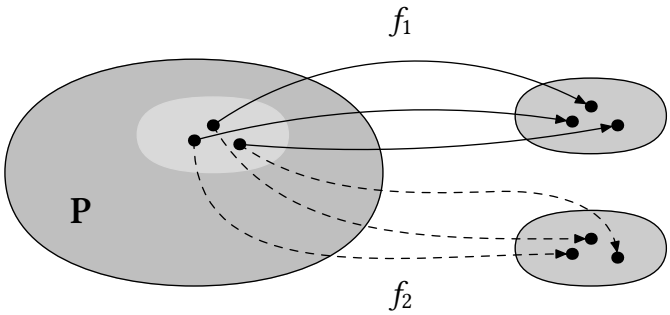
En este capítulo se presentan tres aplicaciones de las enumeraciones descritas en el capítulo 3 poniendo especial énfasis al lenguaje de programación IMP. Sin embargo, las ideas presentes en este trabajo pueden ser utilizadas con otros lenguajes de programación.

Primero se describe como utilizar la enumeración de IMP para generar, analizar y almacenar programas o propiedades asociadas a programas. Posteriormente se plantean distintas formas de trabajar con el lenguaje utilizando diferentes enumeraciones. Finalmente se desarrolla un caso de estudio que utiliza expresiones de lógica proposicional para estudiar posibles definiciones para la enumeración de expresiones booleanas.

4.1 Generación y almacenamiento de programas

El proyecto de investigación [15] describe una propuesta para aproximar a la distribución universal por medio de simulaciones exhaustivas, esto involucra ejecutar cada programa en un subconjunto de programas y determinar su salida en caso que el programa se detenga.

En general se consideran programas $p \in \mathbf{P}$ y funciones $f(p)$ que miden alguna propiedad de los programas, estas propiedades pueden ser sintácticas o semánticas como el tamaño del programa, el valor de su salida, o la cantidad de pasos realizados en su ejecución.



En esta sección es de interés explorar de qué formas se pueden generar subconjuntos de programas y cómo almacenar las relaciones entre programas y sus propiedades calculadas.

Con los métodos que se han descrito en este trabajo, se pueden generar estos subconjuntos de programas de diversas formas.

Comenzando con la operación $\text{unrank}_{\mathbf{P}}$, se pueden analizar los programas en cualquier rango de posiciones. Una forma simple de utilizar esta operación es calculando alguna propiedad para los primeros k programas como en el ejemplo de la figura 2.12, donde se analiza como se distribuyen los tamaños de los programas para un prefijo de la enumeración.

Al incorporar la operación $\text{rank}_{\mathbf{P}}$, en lugar de analizar un rango arbitrario de posiciones se pueden analizar vecindades de un programa. A partir de un programa cualquiera $p \in \mathbf{P}$, se calcula su posición $k = \text{rank}_{\mathbf{P}}(p)$ y posteriormente se utiliza $\text{unrank}_{\mathbf{P}}$ para construir los programas que ocupan las posiciones alrededor de k.

Como se describe al final de la sección 2.2, se pueden utilizar una métrica de tamaño para analizar el subconjunto de todos los programas de tamaño m con la operación $\text{sunrank}_{\mathbf{P}}$, o bien, muestrear una cantidad fija de programas de tamaño m de forma aleatoria con $\text{srandom}_{\mathbf{P}}$.

La métrica de tamaño utilizada en este trabajo se basa en la cantidad de vértices de los árboles de sintaxis, sin embargo, los métodos desarrollados a partir de este cálculo pueden ser adaptados a otras métricas, como por ejemplo la cantidad de bits en una codificación del programa en ASCII, la cantidad de localidades de memoria utilizadas en los programas o la cantidad de iteraciones while anidadas.

Una ventaja de generar programas a partir de operaciones rank y unrank es que una vez que el programa es generado y analizado, se pueden asociar y almacenar estos valores de forma eficiente.

En el contexto del uso de bases de datos las posiciones pueden ser utilizadas e indexadas como llave primaria debido a que cada programa es asociado a una única posición. Esto elimina también la necesidad de almacenar el programa

de forma textual o como árbol de sintaxis ya que la posición puede describir ambas representaciones en su totalidad. Desde esta perspectiva, las posiciones son versiones comprimidas de los programas.

En el contexto de enumeraciones finitas, Frank Ruskey [19] describe a la operación rank como una función hash perfecta debido a que es inyectiva, es decir, le asocia a cada objeto un natural único. Ya que las enumeraciones descritas en el capítulo 3 satisfacen el protocolo de enumeración y a su vez las propiedades de enumerabilidad garantizan que las operaciones rank y unrank sean biyectivas, cualquier enumeración acotada construida con los combinadores puede ser utilizada como función hash perfecta.

Para utilizar $\text{rank}_\mathbf{P}$ como función hash se considera el problema de almacenar alguna propiedad $f(p)$ para cada programa p en un rango de posiciones $[k_1, k_2]$.

Si el rango de posiciones es pequeño, basta con utilizar un arreglo v de tamaño $k_2 - k_1 + 1$ para representar la tabla (también llamadas *direct-address tables*), donde el valor $f(p)$ es almacenado en el componente $h(p)$ de v

$$\begin{aligned} h(p) &= \text{rank}_\mathbf{P}(p) - k_1 \\ v[h(p)] &\leftarrow f(p) \end{aligned}$$

de tal forma que la propiedad del programa en la posición k_1 es almacenado en el componente cero del arreglo y la propiedad del programa en la posición k_2 es almacenado en el último componente $k_2 - k_1$ del arreglo.

Cuando el rango de posiciones es tan grande que resulta conveniente perjudicar el tiempo de ejecución a favor de la reducción en uso de memoria, se puede utilizar una tabla hash que maneje colisiones con listas enlazadas. Si se desea utilizar una tabla de tamaño fijo n se puede modificar la función hash anterior para considerar las posiciones que exceden el tamaño usando aritmética modular

$$h(p) = (\text{rank}_\mathbf{P}(p) - k_1) \bmod n$$

ya que las posiciones de los programas son consecutivas en el rango $[k_1, k_2]$ la cantidad de colisiones en la tabla de tamaño n es mínima.

Cuando el tamaño de la tabla hash se ajusta de forma dinámica, por ejemplo al superar un umbral de colisiones, se puede simplificar la función hash como

$$h(p, n) = \text{rank}_\mathbf{P}(p) \bmod n$$

Una referencia para estas y otras estrategias para trabajar con tablas hash es

el libro “Introduction to Algorithms” [6].

4.2 Enumeraciones alternativas

En la anterior sección se describe como se pueden utilizar las operaciones rank y unrank para construir un subconjunto de objetos de una enumeración. Estas técnicas se basan principalmente en determinar subconjuntos de una enumeración a partir de posiciones.

En esta sección se presentan un conjunto de técnicas para obtener enumeraciones alternativas de IMP. Primero se describe cómo cambiar el orden de los programas y luego cómo enumerar un subconjunto de programas.

Modificación al orden en IMP

Primero se aborda cómo utilizar los métodos de este trabajo para construir enumeraciones de IMP con ordenes distintos a la enumeración canónica.

Suponiendo que las simulaciones exhaustivas descritas en [15] se realizan para los programas que ocupan las primeras k posiciones, el orden inducido por la enumeración de programas puede ser determinante. Si en el prefijo de posiciones $[0, k)$ una gran proporción de programas son triviales, como por ejemplo, concatenaciones de skip, es posible que las propiedades que se desean medir no sean útiles.

Ya que la enumeración de IMP contempla todos los programas y el lenguaje contiene una infinidad de estos programas “poco útiles” para fines de una simulación, es inevitable considerar algunos de ellos para un prefijo suficientemente grande de la enumeración.

La manera más facil de considerar ordenes alternativos de programas es utilizando un orden distinto para los operandos de los combinadores con que se construyen las enumeraciones. Sin embargo, no toda permutación de operandos resulta en una enumeración válida.

Como se menciona y ejemplifica en la sección 3.5, algunas permutaciones de operandos usados para construir enumeraciones recursivas no pueden satisfacer las propiedades pe 3.5 y pe 3.6, por lo tanto, no son permutaciones válidas.

En la figura 4.1 se muestran algunos ejemplos de permutaciones de operandos en la definición de la enumeración de programas **P**. Se denota $\mathbf{P}(i)$ al i-ésimo operando de la unión correspondiente a **P**. Cada permutación es etiquetada con una lista de posiciones cuyos valores describen el orden original.

i	$(0, 4, 3, 2, 1)$ $\mathbf{P}(i)$	$(0, 4, 1, 2, 3)$ $\mathbf{P}(i)$	$(0, 3, 1, 4, 2)$ $\mathbf{P}(i)$
0	K(skip)	K(skip)	K(skip)
1	product(K(if), **B**, **P**, **P**)	product(K(if), **B**, **P**, **P**)	product(K(conc), **P**, **P**)
2	product(K(conc), **P**, **P**)	product(K(set), **X**, **A**)	product(K(set), **X**, **A**)
3	product(K(while), **B**, **P**)	product(K(while), **B**, **P**)	product(K(if), **B**, **P**, **P**)
4	product(K(set), **X**, **A**)	product(K(conc), **P**, **P**)	product(K(while), **B**, **P**)

Tabla 4.1: Permutaciones válidas de operandos para **P**.

A pesar de que la unión distribuye de forma equitativa las posiciones sobre los operandos no-acotados, el cambio de orden tiene un efecto sobre la estructura recursiva de los objetos. Por ejemplo, al considerar la permutación $(0, 4, 1, 2, 3)$ de la tabla, las condicionales pasan a ser el primer operando no-acotado de tal forma que se espera encontrar expresiones if con otras expresiones if anidadas más al inicio de la enumeración en contraste con la permutación original.

Otra forma de reordenar los operandos de una unión y por consecuencia cambiar el orden de los objetos enumerados es compactar dos o mas operandos en uno.

Considerando como ejemplo la definición de la enumeración de expresiones aritméticas **A**, ya que todos los operandos son no-acotados, la unión distribuye de forma equitativa las posiciones sobre cada uno y por lo tanto al explorar un rango de posiciones a cada operando le corresponde una quinta parte del rango de exploración.

Para cambiar estas proporciones se pueden combinar los operandos **N** y **X** en uno solo utilizando el operador unión, resultando en una definición como la siguiente.

$$\begin{aligned}\mathbf{A} = \text{union}(&\text{union}(\mathbf{N}, \mathbf{X}),\\ &\text{product}(\text{K}(\text{add}), \mathbf{A}, \mathbf{A}),\\ &\text{product}(\text{K}(\text{sub}), \mathbf{A}, \mathbf{A}),\\ &\text{product}(\text{K}(\text{mul}), \mathbf{A}, \mathbf{A}))\end{aligned}$$

De esta forma, explorar un rango de posiciones en **A** resulta en tener un octavo asociadas a naturales y localidades respectivamente y un cuarto asociadas a sumas, restas y multiplicaciones respectivamente.

Al describir el combinador producto en la sección 3.6 se menciona cómo los sesgos en la distribución de posiciones pueden ser deseables en algunos contextos

y se plantea el ejemplo de una asignación de memoria donde los rangos de exploración de las localidades sea menor al de las expresiones aritméticas.

Los combinadores descritos en el capítulo 3 procuran no incorporar sesgos en las enumeraciones resultantes, sin embargo se utilizan algunos métodos que permiten incorporar sesgos de forma controlada. Por ejemplo, al describir el combinador de clausura de Kleene se utiliza una enumeración de duplas alternativa de tal forma que el rango de exploración del primer componente es en promedio exponencialmente mas grande que el del segundo componente.

Para utilizar este sesgo en las asignaciones de memoria se considera una enumeración de duplas donde el segundo componente tiende a ser exponencialmente mayor a la primera. Sea ℓ una lista, se denota con $\text{rev}(\ell)$ la lista que tiene los mismos elementos que ℓ pero en orden inverso.

$$\text{rank}''_2(\ell) = \text{rank}'_2(\text{rev}(\ell))$$
$$\text{unrank}''_2(k) = \text{rev}(\text{unrank}'_2(k))$$

Finalmente, basta con reemplazar el uso de rank_2 y unrank_2 por rank''_2 y unrank''_2 respectivamente en las operaciones de la enumeración $\text{product}(\text{K}(\mathsf{set}), \mathbf{X}, \mathbf{A})$ para obtener una enumeración con el sesgo exponencial deseado.

Estas modificaciones a las enumeraciones de IMP no cambian el lenguaje, por lo que todo programa enumerado por $\mathbf{P}$, también es enumerado por cualquier enumeración alternativa $\mathbf{P}'$. Esto permite relacionar las posiciones de estas dos enumeraciones de tal forma que se puede explorar el espacio de programas en la enumeración $\mathbf{P}'$ y asociar las propiedades calculadas a posiciones en la enumeración $\mathbf{P}$ o viceversa

$$k_2 = \text{rank}_{\mathbf{P}'}(\text{unrank}_{\mathbf{P}}(k_1)) \tag{4.1}$$
$$k_1 = \text{rank}_{\mathbf{P}}(\text{unrank}_{\mathbf{P}'}(k_2)) \tag{4.2}$$

Subconjuntos de IMP

Ahora se describen formas de construir enumeraciones no-acotadas de subconjuntos del lenguaje IMP.

Explorar los programas de una enumeración parcial del lenguaje IMP es útil cuando se desean analizar programas con una estructura sintáctica particular.

Una forma de lograr esto es usar un programa como *plantilla* para el subconjunto de programas a ser enumerados. Por ejemplo, si se desean analizar aquellos programas que comienzan con un valor particular almacenado en la localidad de

memoria 0 y luego realizan una iteración que se detiene cuando la localidad 0 tiene valor 0, se utiliza la plantilla

$$(\mathsf{conc}, (\mathsf{set}, (\mathsf{loc}, 0), \mathbf{A}), (\mathsf{while}, (\mathsf{less}, 0, (\mathsf{loc}, 0)), \mathbf{P}))$$

Esta lista de expresiones corresponde a programas de la forma

$$(\mathsf{x}_0 \coloneqq \mathbf{A}\ ;\ (\mathsf{while}\ (0 < \mathsf{x}_0)\ \mathsf{do}\ \mathbf{P}))$$

Para generar esta enumeración, basta con enumerar $\mathbf{A} \times \mathbf{P}$ y para cada pareja construir el programa conforme a la plantilla.

Sobre esta estrategia se pueden contemplar otras modificaciones, por ejemplo, para garantizar que la expresión while de la plantilla es la única iteración en los programas a ser explorados, se puede excluir de la definición de $\mathbf{P}$ al operando $\mathrm{product}(\mathrm{K}(\mathsf{while}), \mathbf{B}, \mathbf{P})$.

Si adicionalmente se desea que los programas enumerados involucren únicamente las primeras dos localidades de memoria, entonces se puede definir

$$\mathbf{X} = \mathrm{product}(\mathsf{loc}, \mathrm{Nat}(0, 1))$$

de tal forma que las asignaciones de memoria en $\mathbf{P}$ y las localidades en $\mathbf{A}$ se limiten a enumerar únicamente estos dos subexpresiones en lugar de cualquier localidad.

Esta técnica anterior puede utilizarse para construir enumeraciones parciales de todos los programas que utilizan n localidades, parametrizando la definición de la enumeración de localidades como

$$\mathbf{X}(n) = \begin{cases} \emptyset & n = 0 \\ \mathrm{product}(\mathsf{loc}, \mathrm{Nat}(0, n-1)) & n > 0 \end{cases}$$

Finalmente, se pueden utilizar enumeraciones de expresiones que sean sintácticamente distintas a programas IMP, pero que pueden ser transformadas a programas equivalentes. En algunos casos esto permite reducir la redundancia semántica en la generación de programas.

Se considera el ejemplo de la concatenación de programas. Sean p_1, p_2 y p_3 programas cualquiera de IMP, la sintaxis descrita en el capítulo 2 establece que

$$(p_1\ ;\ (p_2\ ;\ p_3)) \neq ((p_1\ ;\ p_2)\ ;\ p_3)$$

a pesar de que semánticamente son equivalentes. Por lo que cada concatena-

ción de mas de dos programas en la enumeración **P** ocupa distintas posiciones dependiendo de la estructura de la concatenación.

Al considerar la semántica de IMP esta redundancia se vuelve preocupante ya que que todas las concatenaciones de programas skip (o en general de programas que no afectan la memoria) tienen posiciones únicas en la enumeración **P** y son semánticamente equivalentes a skip.

En general para $n+1$ programas, la cantidad de concatenaciones en **P** que se pueden formar a partir de ellos se calcula como el n-ésimo número de Catalan[4][25] definido como

$$C_n = \frac{1}{n+1}\binom{2n}{n}$$

Por lo que entre más programas se involucren en una concatenación, hay una cantidad exponencialmente mayor de concatenaciones equivalentes.

Para mitigar este problema se propone definir la enumeración de programas de forma distinta, considerando las asignaciones de memoria, las condicionales y las iteraciones como programas simples **S**, de tal forma que los programas **P** se describen como la concatenación de cero o más programas simples utilizando la clausura de Kleene.

$$\begin{aligned}\mathbf{S} = \text{union}(&\text{product}(\text{K}(\mathsf{set}), \mathbf{X}, \mathbf{A}),\\ &\text{product}(\text{K}(\mathsf{while}), \mathbf{B}, \mathbf{P}'),\\ &\text{product}(\text{K}(\mathsf{if}), \mathbf{B}, \mathbf{P}', \mathbf{P}'))\end{aligned}$$

$$\mathbf{P}' = \text{K}(\mathsf{conc}) \times \text{kleene}(\mathbf{S})$$

Para todo programa enumerado por $\mathbf{P}'$ se obtiene su equivalente en **P** al transformar las concatenaciones de n programas a una composición de concatenaciones binarias con asociatividad a la derecha.

$$\begin{aligned}(\mathsf{conc}) &\underset{\mathbf{P}}{\Longrightarrow} \mathsf{skip}\\ (\mathsf{conc}, p_1) &\underset{\mathbf{P}}{\Longrightarrow} p_1\\ (\mathsf{conc}, p_1, p_2) &\underset{\mathbf{P}}{\Longrightarrow} (\mathsf{conc}, p_1, p_2)\\ (\mathsf{conc}, p_1, p_2, p_3) &\underset{\mathbf{P}}{\Longrightarrow} (\mathsf{conc}, p_1, (\mathsf{conc}, p_2, p_3))\\ (\mathsf{conc}, p_1, p_2, p_3, p_4) &\underset{\mathbf{P}}{\Longrightarrow} (\mathsf{conc}, p_1, (\mathsf{conc}, p_2, (\mathsf{conc}, p_3, p_4)))\\ &\dots\end{aligned}$$

Al explorar programas en $\mathbf{P}'$ se obtiene la ventaja de que todas las concatenaciones de los mismos n programas corresponden a una única expresión. Utilizando la ecuación (4.2) se pueden traducir todas las posiciones de estas enumeraciones alternativas a posiciones en la enumeración original, siempre y cuando las expresiones sean transformadas a un equivalente sintácticamente válido de IMP.

4.3 Orden de operandos en expresiones booleanas

En esta sección se presentan los resultados de una exploración que consiste en estudiar funciones booleanas de hasta tres variables a partir de la posición en que se enumeran expresiones booleanas. Para esto se comparan distintas permutaciones del orden de las expresiones de negación, disyunción y conjunción en la especificación de expresiones booleanas $\mathbf{B}$.

Esta parte del trabajo se realiza en colaboración con Luis Benítez Lluis [2] a partir del interés de elegir una especificación conveniente para la enumeración de booleanos del lenguaje IMP. Suponiendo que se enumeran las expresiones secuencialmente a partir de la posición cero, lo ideal es que el prefijo en donde se encuentran las funciones booleanas de dos y tres variables sea lo menor posible, es decir, que todas estas funciones boolenas sean "fáciles" de encontrar en la enumeración.

Se utiliza una simplificación de la enumeración de expresiones booleanas, de tal forma que en lugar de contemplar valores de verdad y comparaciones aritméticas, se utiliza una cantidad fija de variables, similares a localidades de memoria.

$$\begin{aligned}\mathbf{B} = \text{union}(&\mathbf{V_n},\\ &\text{product}(\text{K}(\mathsf{and}), \mathbf{B}, \mathbf{B}),\\ &\text{product}(\text{K}(\mathsf{or}), \mathbf{B}, \mathbf{B}),\\ &\text{product}(\text{K}(\mathsf{not}), \mathbf{B}))\end{aligned}$$

Se propone comparar las enumeraciones de todas las permutaciones de operandos utilizando distintas cantidades de variables. En particular se trabajan con enumeraciones que involucran una, dos y tres variables. Se define la enumeración de n variables booleanas $\mathbf{V_n} = \text{product}(\text{K}(\mathsf{var}), \text{Nat}(0, n-1))$, de tal forma que las variables son listas de la forma (var, i), pero se denotan v_i.

A continuación se muestran las enumeraciones para las seis permutaciones posibles. Se identifica cada una de ellas con el la primer letra del nombre de cada operación. Por ejemplo, cuando los operandos son not, or y and se identifica la enumeración como noa, mientras que para and, or y not se identifica como aon.

$$\mathbf{B} = \text{union}(\mathbf{V_n}, \text{product}(\text{K}(\text{not}), \mathbf{B}), \text{product}(\text{K}(\text{or}), \mathbf{B}, \mathbf{B}), \text{product}(\text{K}(\text{and}), \mathbf{B}, \mathbf{B}))$$

$$\mathbf{B} = \text{union}(\mathbf{V_n}, \text{product}(\text{K}(\text{or}), \mathbf{B}, \mathbf{B}), \text{product}(\text{K}(\text{not}), \mathbf{B}), \text{product}(\text{K}(\text{and}), \mathbf{B}, \mathbf{B}))$$

$$\mathbf{B} = \text{union}(\mathbf{V_n}, \text{product}(\text{K}(\text{or}), \mathbf{B}, \mathbf{B}), \text{product}(\text{K}(\text{and}), \mathbf{B}, \mathbf{B}), \text{product}(\text{K}(\text{not}), \mathbf{B}))$$

$$\mathbf{B} = \text{union}(\mathbf{V_n}, \text{product}(\text{K}(\text{not}), \mathbf{B}), \text{product}(\text{K}(\text{and}), \mathbf{B}, \mathbf{B}), \text{product}(\text{K}(\text{or}), \mathbf{B}, \mathbf{B}))$$

$$\mathbf{B} = \text{union}(\mathbf{V_n}, \text{product}(\text{K}(\text{and}), \mathbf{B}, \mathbf{B}), \text{product}(\text{K}(\text{not}), \mathbf{B}), \text{product}(\text{K}(\text{or}), \mathbf{B}, \mathbf{B}))$$

$$\mathbf{B} = \text{union}(\mathbf{V_n}, \text{product}(\text{K}(\text{and}), \mathbf{B}, \mathbf{B}), \text{product}(\text{K}(\text{or}), \mathbf{B}, \mathbf{B}), \text{product}(\text{K}(\text{not}), \mathbf{B}))$$

Para comparar estas enumeraciones se analizan sus expresiones booleanas a partir de la posición cero. Cada una de estas expresiones es evaluada para todos los posibles valores que toman sus variables para obtener la función booleana que representa la expresión. Una vez que todas las funciones booleanas hayan sido encontradas en la enumeración, o bien cuando se analicen 10^{10} expresiones, se detiene el proceso.

Para cada permutación de operandos y cada cantidad de variables $n \in \{1, 2, 3\}$, se registra la cantidad de expresiones que corresponden a cada función booleana de n variables, la posición de la expresión que representa una función booleana por primera vez y el orden en que se encontró cada función booleana.

Una función booleana de n variables se codifica como un natural menor a 2^n, cuya representación binaria determina el valor de la función para cada posible asignación de valores de verdad.

i	i base 2	Función
0	00	Contradicción
1	01	Negación
2	10	Identidad
3	11	Tautología

	$v_0 = 1$	$v_0 = 0$
$f_{00}(v_0)$	0	0
$f_{01}(v_0)$	0	1
$f_{10}(v_0)$	1	0
$f_{11}(v_0)$	1	1

Tabla 4.2: Funciones booleanas de una variable.

En la tabla 4.2 se muestra para cada función booleana de una variable el número que la identifica, su codificación binaria, el nombre de la función y su interpretación semántica.

Los bits en la función booleana leídos a partir de la posición menos significativa (de derecha a izquierda) representan el valor asignado a la combinación de valores de cada variable ordenados en orden lexicográfico creciente. Este ordenamiento es comúnmente utilizado en las tablas de verdad, por lo que para funciones con pocas variables, basta con conocer su codificación binaria para identificar su semántica.

La tabla de verdad 4.3 muestra la interpretación de la función booleana 13 de dos variables, representada en binario como 1101.

v_0	v_1	$f_{1101}(v_0, v_1)$
0	0	1
0	1	0
1	0	1
1	1	1

Tabla 4.3: Interpretación de función booleana 1101 de dos variables.

A pesar de que las enumeraciones de **B** son no-acotadas y por lo tanto enumeran tantas expresiones como naturales, el conjunto de funciones booleanas para una cantidad de variables fija siempre es finito.

Ya que cada variable puede tomar 2 valores de verdad, con dos variables se tienen 4 posibles combinaciones de valores, y ya que una función asocia a cada combinación un valor de verdad 0 o 1, existen 16 funciones booleanas de dos variables. En general para n variables se tienen 2^n combinaciones de valores y por lo tanto $2^{(2^n)}$ funciones booleanas.

A continuación se analizan los resultados obtenidos utilizando una sola variable. La tabla 4.4 muestra la cantidad de expresiones analizadas por cada función booleana. Con estos resultados se observan dos propiedades en común de todas las permutaciones.

La primera es que la función identidad es la que más repeticiones tiene en el rango explorado, por lo que aparenta ser la función más común en la enumeración. A partir de las definiciones de **B** es fácil ver cómo la identidad es la primer función que se encuentra en la enumeración, después de todo el primer operando es $\mathbf{V_n}$. Sin embargo, esto no es suficiente para explicar la abundancia de expresiones menos simples con el mismo valor semántico.

Función		Permutación noa	nao	ona	ano	aon	oan
Contradicción	00	1	1	1	1	1	1
Negación	01	1	1	3	3	5	5
Identidad	10	4	4	8	8	14	14
Tautología	11	1	1	1	1	1	1
	Total	7	7	13	13	21	21

Tabla 4.4: Conteo de expresiones por función booleana con $n = 1$.

La segunda propiedad es que en todas las permutaciones se encontraron la misma cantidad de contradicciones que de tautologías. Analizando el orden en que se encuentran las funciones, todas las permutaciones presentan el mismo patrón: primero se encuentra la identidad, luego la negación y al final la tautología o la contradicción.

Analizando las diferencias entre las permutaciones, se puede caracterizar la tabla de resultados de acuerdo a la posición de la negación como operando en cada enumeración. La única diferencia que se encuentra dentro de estas agrupaciones es que cuando la disyunción precede a la conjunción en los operandos, se encuentra la primer tautología una posición antes que la primer contradicción, y en caso contrario se encuentra la primer contradicción una posición antes que la primer tautología.

La suma de las entradas de cada columna en la tabla corresponde a la menor cantidad de posiciones que se deben enumerar para encontrar todas las funciones booleanas. Comparando los tres grupos de permutaciones entre sí, aparenta ser más conveniente elegir una permutación con la negación al inicio, ya que se necesita explorar una menor cantidad de posiciones para encontrar todas las funciones.

Ahora se continúa el análisis con los resultados para funciones booleanas de dos variables. En la tabla 4.5 se muestra la cantidad de expresiones analizadas para cada función booleana y se indican los nombres de las funciones comúnmente utilizadas en lógica proposicional.

A diferencia de los resultados con una variable, los datos muestran que las contradicciones y las tautologías son de las funciones que más se repiten en el rango explorado de la enumeración. En este caso, las funciones con menos repeticiones para todas las permutaciones son la disyunción exclusiva y su negación, la

Función		Permutación					
		noa	nao	ona	ano	aon	oan
Contradicción	0000	176	177	147	146	108	109
	0001	58	52	64	52	52	56
	0010	38	41	43	46	39	38
	0011	35	35	27	27	21	21
	0100	56	56	47	46	38	39
	0101	132	132	117	117	104	104
Disyunción exclusiva	0110	1	2	1	4	2	1
	0111	52	58	52	64	56	52
Conjunción	1000	124	128	133	151	135	124
Bicondicional	1001	2	1	4	1	1	2
	1010	173	173	233	233	233	233
Condicional material	1011	41	38	46	43	38	39
	1100	79	79	68	68	54	54
	1101	56	56	46	47	39	38
Disyunción	1110	128	124	151	133	124	135
Tautología	1111	177	176	146	147	109	108
	Total	1328	1328	1325	1325	1153	1153

Tabla 4.5: Conteo de funciones booleanas con $n = 2$.

cuál es equivalente al operador bicondicional.

También se puede observar que al agrupar las permutaciones por la posición del operando negación permite enfatizar algunos patrones. Aunque es un poco más difícil de ver en la tabla de datos, en cada grupo de permutaciones las cantidades de expresiones para una función booleana i son similares a la función booleana cuya codificación es resultado de negar e invertir los bits de i base 2.

Por ejemplo, la cantidad de expresiones que representan la función 0100 con la permutación noa es igual a la cantidad de expresiones que representan la función 1101 con la permutación nao, esto ocurre también al comparar ona con ano y aon con oan. Por otro lado, una función cuya inversa negada es igual a si misma, como por ejemplo 0011 o 1010, tienen valores únicos independientemente del orden relativo entre la disyunción y la conjunción en los operandos.

Este patrón posiblemente aparece en el conteo ya que el procedimiento de invertir y negar bits relaciona la función booleana de la conjunción con la dis-

yunción y viceversa, los cuales son los operadores utilizados en las expresiones booleanas enumeradas.

La diferencia más relevante para este análisis entre los datos para una y dos variables es el conteo total por permutación. El orden de estos valores en la tabla 4.5 es completamente opuesto al que aparece en la tabla 4.4. Con dos variables aparenta ser el caso que el grupo de permutaciones aon y oan es mejor a los otros dos, ya que necesita enumerar menos posiciones en el rango para encontrar todas las funciones booleanas.

Analizar estos datos de forma aislada puede llevar a conclusiones que no sean favorables en el contexto del lenguaje IMP. Un factor que se toma en cuenta al comparar las permutaciones es que la enumeración **B** es utilizada únicamente en programas if y while.

En las expresiones if es indistinto si la condicional es una tautología o contradicción, ya que se tiene la equivalencia semántica

$$(\mathsf{if}, \mathsf{true}, p_1, p_2) = (\mathsf{if}, \mathsf{false}, p_2, p_1)$$

Sin embargo, en el contexto de las expresiones while, hay una gran diferencia entre tener una tautología o una contradicción en la condicional, ya que en el primer caso el ciclo es infinito, mientras que en el segundo el ciclo es semánticamente equivalente a skip.

Al enumerar y ejecutar programas, puede ser difícil determinar para cada ciclo si la expresión booleana en su condicional es una tautología o una contradicción y ya que los ciclos infinitos son computacionalmente costosos, se opta por preferir permutaciones con más proporción de contradicciones que de tautologías.

Este último criterio apoya la hipótesis que el grupo de permutaciones mas conveniente para IMP es aon y oan. Sin embargo, se pueden observar datos que van en contra de esta hipótesis al analizar la primer posición en la que aparecen las funciones booleanas.

La tabla 4.6 muestra una tabla con estructura similar a 4.5 pero que muestra la cantidad de posiciones que se analizaron antes dela primera aparición de cada función booleana. Ya que las posiciones se cuentan a partir del cero, este valor es equivalente a la posición de la primera expresión que representa cada función booleana.

Los renglones sombreados corresponden a las funciones que fueron más difíciles de encontrar, las cuáles coinciden con las funciones con menos repeticiones en el rango de posiciones analizado.

Lo que vale la pena resaltar es que las primeras posiciones de estas dos

Función		Permutación noa	nao	ona	ano	aon	oan
Contradicción	0000	13	12	22	20	32	33
	0001	20	23	18	24	22	19
	0010	∗ 49	48	∗ 67	65	86	∗ 87
	0011	5	5	6	6	7	7
	0100	25	24	37	35	50	51
	0101	2	2	3	3	4	4
Disyunción exclusiva	0110	1327	1158	1324	998	995	1152
	0111	23	20	24	18	19	22
Conjunción	1000	7	6	7	5	5	6
Bicondicional	1001	1158	1327	998	1324	1152	995
	1010	0	0	0	0	0	0
Condicional material	1011	48	∗ 49	65	∗ 67	∗ 87	86
	1100	1	1	1	1	1	1
	1101	24	25	35	37	51	50
Disyunción	1110	6	7	5	7	6	5
Tautología	1111	12	13	20	22	33	32

Tabla 4.6: Posiciones analizadas antes de primera aparición con $n = 2$.

funciones son valores atípicos, ya que la diferencia en posiciones de estas funciones con el resto es de dos ordenes de magnitud. En la tabla se etiqueta con el símbolo '$*$' las posiciones máximas de cada columna excluyendo los valores atípicos.

Desde esta perspectiva, las permutaciones noa y nao solo deben enumerar las primeras 50 posiciones para haber encontrado todas las funciones booleanas "usuales", mientras que las permutaciones que aparentaban ser mejores, aon y oan, deben enumerar 88 posiciones.

Esta última observación puede ser poco relevante al considerar que una diferencia de alrededor de 40 posiciones es pequeña, sin embargo, es posible que el contraste sea cada vez mas grande conforme se involucran más variables.

Se plantea establecer prioridades de los distintos criterios mencionados, de mayor preferencia a menor preferencia se considera

1. La permutación encuentra más funciones booleanas,

2. Requiere analizar menos posiciones para encontrar todas las funciones,

3. Tiene una menor proporción de contradicciones y tautologías,

4. La proporción de tautologías es menor a la de contradicciones.

Bajo este orden de prioridad, las mejores permutaciones son aquellas que tienen la negación como último operando en la definición de la enumeración. Para justificar la elección final se analizan los datos para tres variables.

El tiempo de cómputo utilizado para registrar la información para el caso de una variable fue en promedio 0.000088 segundos, para el caso de dos variables fue en promedio 0.005395 segundos y para el caso de tres variables fue en promedio 1 día, 10 horas y 29 minutos[1]. Debido a este gran incremento en el tiempo de cómputo, se decide trabajar exclusivamente con las permutaciones donde la conjunción precede a la disyunción, es decir nao, ano y aon.

En estos tres casos, se evaluaron las primeras 10^{10} posiciones sin terminar de encontrar todas las funciones boolenas de tres variables, por lo que algunas funciones tienen su primer expresión booleana fuera de este rango. Ya que existen $2^8 = 256$ funciones booleanas de tres variables las tablas de datos se presentan parcialmente.

En la tabla 4.7 se muestran las funciones que no fueron encontradas o bien fueron encontradas cerca del umbral de posiciones 10^{10}. Cuando la función fue encontrada en una permutación se muestra un aproximado de su primera posición.

Función	nao	ano	aon
01101001	$> 10^{10}$	$> 10^{10}$	$> 10^{10}$
10010010	$> 10^{10}$	$> 10^{10}$	$> 10^{10}$
10010100	$> 10^{10}$	$> 10^{10}$	$> 10^{10}$
10010110	$> 10^{10}$	$> 10^{10}$	$> 10^{10}$
10011110	$> 10^{10}$	$> 10^{10}$	$\approx 8 \times 10^9$
10110110	$> 10^{10}$	$\approx 6 \times 10^9$	$\approx 7 \times 10^9$
11010110	$> 10^{10}$	$\approx 9 \times 10^9$	$> 10^{10}$

Tabla 4.7: Funciones booleanas más difíciles de encontrar con $n = 3$.

A partir de estos casos extremos, se determina que las permutaciones ano y aon son mejores que nao. Entre estos dos mejores candidatos se tiene que encuentran

[1]Estas mediciones de tiempo son aproximadas, los cálculos se realizaron en una computadora HP Z4G4 de cuatro procesadores Intel Xeon y 16 GB de RAM.

la misma cantidad de funciones booleanas, sin embargo presentan diferencias importantes. Por un lado con la permutación ano se encuentra la función 10110110 10^9 posiciones antes que con la permutación aon, pero por otro, la última función que se encuentra con la permutación ano es en 10^9 posiciones después que con la permutación aon.

La ventaja de aon sobre ano se observa al analizar la proporción de tautologías y contradicciones mostrada en la tabla 4.8. El porcentaje de estas dos funciones en las primeras 10^{10} posiciones es menor que en las otras permutaciones y también se tiene que la cantidad de tautologías es menor que la cantidad de contradicciones.

Permutación	Tautologías	Contradicciones	Porcentaje del total
nao	1,731,632,806	1,743,919,753	34.75%
ano	1,652,660,276	1,676,022,782	33.28%
aon	1,592,366,289	1,603,939,758	31.96%

Tabla 4.8: Cantidad de tautologías y contradicciones con $n = 3$.

La permutación aon es utilizada en la definición de expresiones booleanas de los capítulos 2 y 3. Sin embargo, este orden fue determinado posterior al análisis descrito en esta sección, mientras que en las primeras etapas del trabajo se utilizaba la permutación noa.

Capítulo 5

Conclusiones

En este trabajo se mostró como a partir de un lenguaje de programación se pueden definir operaciones rank y unrank para inducir una biyección entre expresiones del lenguaje y números naturales.

Las técnicas utilizadas para la definición de estas operaciones, como la división euclidiana y la enumeración de n-tuplas, permiten construir otros mecanismos para explorar el espacio de programas.

Esto se ejemplifica en la sección 2.2, donde se introduce una métrica de tamaño para programas IMP y se abordan problemas de enumeración en función del tamaño. Primero con métodos para contar la cantidad de programas para un tamaño particular, posteriormente para derivar la operación sunrank que enumera programas de acuerdo a su tamaño, finalmente se muestra como adaptar estas soluciones para muestrear de forma aleatoria el espacio de programas.

Todo el capítulo 3 es dedicado a modelar computacionalmente las enumeraciones utilizando un protocolo de enumeración, el cuál permite razonar sobre los algoritmos de enumeración independientemente del lenguaje con que se planea trabajar.

Se detallan procedimientos para definir enumeraciones primitivas, comenzando con enumeraciones triviales como $\emptyset$, pasando por la construcción de familias de enumeraciones como K y Nat, hasta llegar a definir la enumeración relativamente compleja Bits.

Se muestra la flexibilidad del protocolo de enumeración al construir los combinadores de enumeraciones product, union y kleene. En los tres casos se presentan operadores simplificados, se discuten sus desventajas en los órdenes que generan y se proponen mejoras para procurar tener una distribución uniforme de objetos en las enumeraciones resultantes.

Este capítulo termina presentando una enumeración de IMP basada en combinadores, la cuál no solo enumera los programas en el mismo orden que los algoritmos del capítulo 2, sino que también resulta en una equivalencia intencional, es decir, los cálculos realizados por rank y unrank son los mismos.

En términos de las aplicaciones, el capítulo 4 presenta una variedad de estrategias y técnicas para trabajar con las enumeraciones, incluyendo el uso de rank como función hash o llave primaria para el almacenamiento de información sobre programas y la exploración de un subespacio de programas a considerando las propiedades sintácticas y semánticas de IMP.

Finalmente se describe a detalle el proceso mediante el cual se analizaron distintas enumeraciones de expresiones booleanas para mejorar los resultados de la generación e interpretación de programas IMP al reducir la proporción de ciclos infinitos mediante el estudio de funciones booleanas para un conjunto finito de variables lógicas.

El sistema de enumeración implementado fue utilizado en cada parte de este trabajo escrito para verificar la viabilidad de las ideas desarrolladas y para experimentar con las enumeraciones como una herramienta para el estudio lenguajes de programación.

5.1 Trabajo futuro

En el contexto del proyecto de investigación y la colaboración con Lemus Yáñez, Benítez Lluis y Zamora Gutiérrez, el sistema de enumeración y el intérprete de IMP no han sido integrados para que funcionen de forma automática. Por un lado se enumera una cantidad fija de programas y se almacenan en archivos de texto, y por otro lado se leen y analizan sintácticamente estos archivos para evaluar los programas que contienen.

Ya que el operador unrank produce como resultado árboles de sintaxis, es redundante transcribir su estructura en texto plano para luego ser analizado sintácticamente. Estos dos sistemas pudieran ser fusionados para interpretar un programa inmediatamente después de su construcción.

El grupo de trabajo ha considerado investigar posibles soluciones a la simulación de programas ante el inevitable problema de la detención. Se han explorado métodos estadísticos para calcular parámetros de detención a partir de una muestra aleatoria de programas [3]. Es trabajo pendiente aplicar los algoritmos de selección aleatoria de la sección 2.2 a estos modelos estadísticos.

En el contexto del las enumeraciones basadas en combinadores, se realizaron

intentos de incorporar la noción de *enumeraciones parametrizadas* al protocolo descrito en la sección 3.1, sin embargo no se ha podido implementar esta clase de enumeraciones de forma eficiente en el sistema.

Una enumeración parametrizada extiende el protocolo de enumeración para que cada operación dependa de parámetros adicionales y calcule valores adicionales a los establecidos. Esto permite definir enumeraciones que cambian sus reglas de operación dependiendo de objetos enumerados previamente.

Una aplicación de la parametrización de enumeraciones para IMP es enumerar el conjunto de todos los programas donde las localidades de memoria se definen en orden a partir de $(\mathsf{loc}, 0)$ y solo se utilizan localidades de memoria a las que previamente se le asignó un valor.

Bibliografía

[1] Sanjeev Arora y Boaz Barak. *Computational Complexity: A Modern Approach*. 1st. USA: Cambridge University Press, 2009. ISBN: 0521424267.

[2] Luis Benítez Lluis. "Sobre la correctud de algoritmos de enumeración de tuplas para la enumeración de programas". Tesis no publicada. 2021.

[3] Cristian S. Calude y Monica Dumitrescu. "A statistical anytime algorithm for the Halting Problem". En: *Computability* 9 (2020). 2, págs. 155-166. ISSN: 2211-3576.

[4] Eugène Catalan. "Note on a finite difference equation". En: *Journal de Mathématiques Pures et Appliquées* 3 (1838), págs. 508-516.

[5] S. B. Cooper. *Computability theory*. Boca Raton: Chapman & Hall/CRC, 2004. ISBN: 9781584882374.

[6] Thomas H. Cormen y col. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844.

[7] Jean-Paul Delahaye y Hector Zenil. "Numerical evaluation of algorithmic complexity for short strings: A glance into the innermost structure of randomness". En: *Applied Mathematics and Computation* 219.1 (2012), págs. 63-77.

[8] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". En: *Commun. ACM* 12.10 (oct. de 1969), págs. 576-580. ISSN: 0001-0782.

[9] John E. Hopcroft, Rajeev Motwani y Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321455363.

[10] Graham Hutton. "Higher-order functions for parsing". En: *Journal of Functional Programming* 2.3 (1992), págs. 323-343.

[11] Walter Kirchherr, Ming Li y Paul Vitányi. “The miraculous universal distribution”. En: *The Mathematical Intelligencer* 19.4 (1997), págs. 7-15.

[12] Donald E. Knuth. *The Art of Computer Programming: Combinatorial Algorithms, Part 1*. 1st. Addison-Wesley Professional, 2011. ISBN: 0201038048.

[13] A. N. Kolmogorov. “Three approaches to the quantitative definition of information”. En: *International Journal of Computer Mathematics* 2.1-4 (1968), págs. 157-168.

[14] Donald L. Kreher y Douglas R. Stinson. *Combinatorial Algorithms: Generation, Enumeration, and Search*. English. Paperback. CRC Press, nov. de 2019, pág. 344. ISBN: 978-0367400156.

[15] Marco Vladimir Lemus Yáñez, Francisco Hernández Quiroz y Héctor Zenil. “Aproximación a la distribución universal por medio de simulaciones exhaustivas”. Proyecto de investigación. Tesis doct. UNAM, 2019.

[16] Benjamin C. Pierce y col. *Logical Foundations*. Ed. por Benjamin C. Pierce. Vol. 1. Software Foundations. Version 6.1, http://softwarefoundations.cis.upenn.edu. Electronic textbook, 2021.

[17] Benjamin C. Pierce y col. *Programming Language Foundations*. Ed. por Benjamin C. Pierce. Vol. 2. Software Foundations. Version 6.1, http://softwarefoundations.cis.upenn.edu. Electronic textbook, 2021.

[18] Gordon D. Plotkin. “A Structural Approach to Operational Semantics”. En: *The Journal of Logic and Algebraic Programming* 60 (2004), págs. 17-139.

[19] Frank Ruskey. “Combinatorial generation”. En: *Preliminary working draft. University of Victoria, Victoria, BC, Canada* 11 (2003), pág. 20.

[20] Michael Sipser. *Introduction to the theory of computation*. Boston, MA: Cengage Learning, 2013. ISBN: 978-1133187790.

[21] Fernando Soler-Toscano y col. “Calculating Kolmogorov complexity from the output frequency distributions of small Turing machines”. En: *PloS one* 9.5 (2014), e96223.

[22] R.J. Solomonoff. “A formal theory of inductive inference. Part I”. En: *Information and Control* 7.1 (1964), págs. 1-22. ISSN: 0019-9958.

[23] R.J. Solomonoff. “A formal theory of inductive inference. Part II”. En: *Information and Control* 7.2 (1964), págs. 224-254. ISSN: 0019-9958.

[24] Ray J. Solomonoff. “The Kolmogorov Lecture The Universal Distribution and Machine Learning”. En: *The Computer Journal* 46.6 (ene. de 2003), págs. 598-601. ISSN: 0010-4620.

[25] Richard P. Stanley. *Catalan Numbers*. Cambridge University Press, 2015.

[26] R. D. Tennent. *Principles of Programming Languages*. USA: Prentice Hall PTR, 1981. ISBN: 0137098731.

[27] Philip Wadler. “How to replace failure by a list of successes a method for exception handling, backtracking, and pattern matching in lazy functional languages”. En: *Functional Programming Languages and Computer Architecture*. Ed. por Jean-Pierre Jouannaud. Berlin, Heidelberg: Springer Berlin Heidelberg, 1985, págs. 113-128. ISBN: 978-3-540-39677-2.

[28] Victor Zamora Gutiérrez. “Diseño e implementación de un intérprete eficiente para ejecución masiva de programas”. Tesis no publicada. 2021.

[29] Hector Zenil. “A Review of Methods for Estimating Algorithmic Complexity: Options, Challenges, and New Directions”. En: *Entropy* 22.6 (2020). ISSN: 1099-4300.

[30] Hector Zenil y col. “A decomposition method for global evaluation of shannon entropy and local estimations of algorithmic complexity”. En: *Entropy* 20.8 (2018), pág. 605.

[31] Hector Zenil y col. “Causal deconvolution by algorithmic generative models”. En: *Nature Machine Intelligence* 1.1 (2019), págs. 58-66.

Apéndice A

Sintaxis y semántica de IMP

A.1 Gramática libre de contexto

$$
\begin{aligned}
\mathbf{P} &\to \mathsf{skip} \mid \mathbf{X} \coloneqq \mathbf{A} \mid (\mathsf{while}\ \mathbf{B}\ \mathsf{do}\ \mathbf{P}) \mid (\mathbf{P}\ ;\ \mathbf{P}) \mid (\mathsf{if}\ \mathbf{B}\ \mathsf{then}\ \mathbf{P}\ \mathsf{then}\ \mathbf{P}) \\
\mathbf{B} &\to \mathsf{true} \mid \mathsf{false} \mid (\mathbf{A} = \mathbf{A}) \mid (\mathbf{A} < \mathbf{A}) \mid \neg\, \mathbf{B} \mid (\mathbf{B} \vee \mathbf{B}) \mid (\mathbf{B} \wedge \mathbf{B}) \\
\mathbf{A} &\to \mathbf{N} \mid \mathbf{X} \mid (\mathbf{A} + \mathbf{A}) \mid (\mathbf{A} - \mathbf{A}) \mid (\mathbf{A} \times \mathbf{A}) \\
\mathbf{X} &\to \mathsf{x}_{\mathbf{N}} \\
\mathbf{N} &\to \mathbf{D_0} \mid \mathbf{D_1}\mathbf{D_0}^{+} \\
\mathbf{D_0} &\to 0 \mid \mathbf{D_1} \\
\mathbf{D_1} &\to 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}
$$

A.2 Semántica

$$\langle n, \sigma \rangle \to_{\mathbf{N}} \#n \qquad \langle \mathsf{x}_i, \sigma \rangle \to_{\mathbf{X}} \sigma(\mathsf{x}_i)$$

$$\frac{\langle \mathsf{n}, \sigma \rangle \to_{\mathbf{N}} n}{\langle \mathsf{n}, \sigma \rangle \to_{\mathbf{A}} n} \qquad \frac{\langle \mathsf{x}_i, \sigma \rangle \to_{\mathbf{X}} n}{\langle \mathsf{x}_i, \sigma \rangle \to_{\mathbf{A}} n}$$

Para cada $\oplus$ en $\{+, -, \times\}$

$$\frac{\langle a_1, \sigma\rangle \to_{\mathbf{A}} n_1 \quad \langle a_2, \sigma\rangle \to_{\mathbf{A}} n_2 \quad n_1 \oplus_{\mathbb{N}} n_2 = n}{\langle (a_1 \oplus a_2), \sigma\rangle \to_{\mathbf{A}} n}$$

$$\langle \mathsf{true}, \sigma\rangle \to_{\mathbf{B}} \mathsf{true} \qquad \langle \mathsf{false}, \sigma\rangle \to_{\mathbf{B}} \mathsf{false}$$

$$\frac{\langle a_1, \sigma\rangle \to_{\mathbf{A}} n_1 \quad \langle a_2, \sigma\rangle \to_{\mathbf{A}} n_2 \quad n_1 = n_2}{\langle (a_1 = a_2), \sigma\rangle \to_{\mathbf{B}} \mathsf{true}} \qquad \frac{\langle a_1, \sigma\rangle \to_{\mathbf{A}} n_1 \quad \langle a_2, \sigma\rangle \to_{\mathbf{A}} n_2 \quad n_1 \neq n_2}{\langle (a_1 = a_2), \sigma\rangle \to_{\mathbf{B}} \mathsf{false}}$$

$$\frac{\langle a_1, \sigma\rangle \to_{\mathbf{A}} n_1 \quad \langle a_2, \sigma\rangle \to_{\mathbf{A}} n_2 \quad n_1 < n_2}{\langle (a_1 < a_2), \sigma\rangle \to_{\mathbf{B}} \mathsf{true}} \qquad \frac{\langle a_1, \sigma\rangle \to_{\mathbf{A}} n_1 \quad \langle a_2, \sigma\rangle \to_{\mathbf{A}} n_2 \quad n_1 \geq n_2}{\langle (a_1 < a_2), \sigma\rangle \to_{\mathbf{B}} \mathsf{false}}$$

$$\frac{\langle b, \sigma\rangle \to_{\mathbf{B}} \mathsf{false}}{\langle \neg\, b, \sigma\rangle \to_{\mathbf{B}} \mathsf{true}} \qquad \frac{\langle b, \sigma\rangle \to_{\mathbf{B}} \mathsf{true}}{\langle \neg\, b, \sigma\rangle \to_{\mathbf{B}} \mathsf{false}}$$

Para cada $\oplus$ en $\{\wedge, \vee\}$

$$\frac{\langle b_1, \sigma\rangle \to_{\mathbf{B}} v_1 \quad \langle b_2, \sigma\rangle \to_{\mathbf{B}} v_2 \quad v_1 \oplus_{\mathbb{B}} v_2 = v}{\langle (b_1 \oplus b_2), \sigma\rangle \to_{\mathbf{B}} v}$$

		$v_1 \oplus_{\mathbb{B}} v_2$	
v_1	v_2	$v_1 \vee v_2$	$v_1 \wedge v_2$
false	false	false	false
false	true	true	false
true	false	true	false
true	true	true	true

$$\langle \mathsf{skip}, \sigma\rangle \to_{\mathbf{P}} \sigma$$

$$\frac{\langle a, \sigma\rangle \to_{\mathbf{A}} n}{\langle \mathsf{x}_i := a, \sigma\rangle \to_{\mathbf{P}} \sigma[n/\mathsf{x}_i]}$$

$$\frac{\langle p_1, \sigma\rangle \to_{\mathbf{P}} \sigma_1 \quad \langle p_2, \sigma_1\rangle \to_{\mathbf{P}} \sigma_2}{\langle (p_1 \,;\, p_2), \sigma\rangle \to_{\mathbf{P}} \sigma_2}$$

$$\frac{\langle b, \sigma\rangle \to_{\mathbf{B}} \mathsf{false}}{\langle (\mathsf{while}\ b\ \mathsf{do}\ p), \sigma\rangle \to_{\mathbf{P}} \sigma}$$

$$\frac{\langle b, \sigma\rangle \to_{\mathbf{B}} \mathsf{true} \quad \langle p, \sigma\rangle \to_{\mathbf{P}} \sigma_1 \quad \langle (\mathsf{while}\ b\ \mathsf{do}\ p), \sigma_1\rangle \to_{\mathbf{P}} \sigma_2}{\langle (\mathsf{while}\ b\ \mathsf{do}\ p), \sigma\rangle \to_{\mathbf{P}} \sigma_2}$$

$$\frac{\langle b, \sigma\rangle \to_{\mathbf{B}} \mathsf{true} \quad \langle p_1, \sigma\rangle \to_{\mathbf{P}} \sigma'}{\langle (\mathsf{if}\ b\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2), \sigma\rangle \to_{\mathbf{P}} \sigma'} \qquad \frac{\langle b, \sigma\rangle \to_{\mathbf{B}} \mathsf{false} \quad \langle p_2, \sigma\rangle \to_{\mathbf{P}} \sigma'}{\langle (\mathsf{if}\ b\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2), \sigma\rangle \to_{\mathbf{P}} \sigma'}$$

Apéndice B

Implementación

En el siguiente enlace se encuentra un repositorio con los programas literarios que conforman al sistema de enumeración descrito en este trabajo.

https://gitlab.com/eduardoacye/enumera

El repositorio contiene un archivo `README.org` con instrucciones de instalación y ejecución del sistema, así como algunos ejemplos de uso de los programas.

Los programas literarios consisten de archivos con extensión `.org` los cuáles son procesados para generar un documento con el mismo nombre pero con extensión `.pdf` y uno o más archivos de código en la carpeta `src/`. El listado de programas literarios es el siguiente:

- `coq-imp`: Describe una verificación de que la asociación entre posiciones y objetos en la enumeración de IMP es una biyección.
- `enumera`: Describe el sistema de enumeración basado en combinadores y enumeraciones primitivas.
- `imp`: Describe la enumeración, conteo por tamaño y selección aleatoria por tamaño de programas IMP.