Licenciatura en Ciencias de la Computación

Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario (Argentina)

⋆ Tesina de Grado ⋆

Reducción de los Esquemas de Recursión

Autor:
Daniel E. Severin

Director:
Dra. Gabriela Argiroffo

23 de noviembre de 2006

Índice general

1. Introducción a las funciones recursivas **1**
- 1.1. Motivación . . . 1
- 1.2. Nociones preliminares . . . 1
- 1.3. Las funciones recursivas primitivas . . . 3
- 1.4. Sintaxis de las funciones recursivas primitivas . . . 5
- 1.5. Ejemplos de funciones recursivas primitivas . . . 5
- 1.6. Funciones doblemente recursivas . . . 8
- 1.7. Minimización . . . 11

2. Reducción de los esquemas de recursión **13**
- 2.1. Funciones de parificación . . . 13
- 2.2. Estructuras de datos recursivas primitivas . . . 16
- 2.3. Reducción de algunos esquemas de recursión . . . 20
- 2.4. Reducción de $\overline{\mathrm{FDR}}$ a $\overline{\mathrm{FRG}}$ . . . 23
- 2.5. Notas con respecto a $\overline{\mathrm{FRG}}$ . . . 28

3. Reducción a funciones unarias **30**
- 3.1. Las funciones unarias primitivas . . . 30
- 3.2. Ejemplos de funciones unarias primitivas . . . 33
- 3.3. Reducción a funciones unarias primitivas . . . 35
- 3.4. Inversión y funciones unarias generales . . . 42
- 3.5. Notas sobre las funciones unarias generales . . . 46

4. Reducciones sobre las funciones unarias primitivas **48**
- 4.1. Introducción . . . 48
- 4.2. La función Q . . . 49
- 4.3. Funciones Rt y Sq . . . 54
- 4.4. Otras bases . . . 54
- 4.5. Breve reseña histórica . . . 57

5. Caracterización de las funciones recursivas con funciones generadoras **60**
- 5.1. Funciones generadoras para las funciones recursivas primitivas . . . 60
- 5.2. Ejemplos de funciones generadoras tradicionales . . . 61
- 5.3. Composición usando funciones generadoras tradicionales . . . 63
- 5.4. Conclusión . . . 64

Agradecimientos

Doy las gracias principalmente a mi familia por brindarme su apoyo y alentarme durante todos estos años. También agradezco a los profesores, ayudantes y director de la carrera por haberme formado y capacitado, aportando a mi crecimiento académico y personal.

Agradezco especialmente la colaboración de la Dra. Gabriela Argiroffo de la Universidad Nacional de Rosario (Argentina), que guió y supervisó pacientemente mi trabajo. Y también reconozco la ayuda incondicional de los profesores Stefano Mazzanti de la Università Iuav di Venezia (Italia), István Szalkai de la Pannon Egyetem (Hungría) y Cristian Calude de la University of Auckland (Nueva Zelanda). Gracias a ellos, dispongo de los artículos más importantes que utilicé en el desarrollo de mi tesina.

Dedico este trabajo a mi padre Aurelio Severin.

Capítulo 1

Introducción a las funciones recursivas

1.1. Motivación

En este trabajo estudiaremos en profundidad ciertas clases de funciones (que enumeraremos luego), bien conocidas en el ámbito de la *Teoría de la Computabilidad*. A estas clases de funciones las encapsularemos en un concepto más general, que llamaremos *formalismo*. Un formalismo es, básicamente, un conjunto de funciones que se definen inductivamente. Los formalismos conocidos que estudiaremos son el de las funciones recursivas primitivas y las funciones recursivas generales. Además, definiremos otros formalismos nuevos y demostraremos propiedades que los vinculan.
En este capítulo vamos a estudiar las funciones recursivas primitivas y las funciones recursivas generales. La ventaja de estas últimas es *que poseen el mismo poder expresivo que las máquinas de Turing*.
En el capítulo 2 analizaremos estructuras de datos que se pueden construir con las funciones recursivas primitivas y veremos algunos métodos de reducción (i.e. conversión de ciertos esquemas de recursión a esquemas más simples).
En el capítulo 3 introduciremos las *funciones unarias primitivas* que resultarán tener el mismo poder expresivo que las funciones recursivas primitivas. También presentaremos a las *funciones unarias generales* que, como en el caso anterior, tendrán el mismo poder expresivo que las funciones recursivas generales. Dado que son endomorfismos de los números naturales (i.e. funciones de la forma $\mathbb{N} \to \mathbb{N}$), estas funciones presentan una estructura de monoide peculiar. Esto permite el poder analizarlas desde un punto de vista algebraico además del aritmético como es el caso de las funciones recursivas primitivas y generales.
En el capítulo 4 propondremos otro formalismo que puede representar a las funciones unarias primitivas, y algunas variantes posibles del mismo.
Finalmente, en el capítulo 5 analizaremos una posible caracterización de las funciones recursivas utilizando funciones generadoras.

1.2. Nociones preliminares

A continuación enumeraremos algunos conceptos básicos que requeriremos durante el resto de este desarrollo.

⋆ Usaremos el conjunto de los números naturales con el cero incluido. Es decir,

$$\mathbb{N} = \{0, 1, 2, 3, 4, ...\}.$$

- ⋆ Cuando debamos representar funciones parciales (i.e. que no estén definidas para todos sus valores de entrada) nos servirá el siguiente conjunto extendido de los naturales:

$$\hat{\mathbb{N}} = \mathbb{N} \cup \{\bot\},$$

donde el símbolo $\bot$ es llamado *bottom* y determina un valor no definido.

- ⋆ Usaremos un estilo *estricto* (esta definición se puede consultar en [Bir00]) para evaluar funciones. Si, al menos, uno de los argumentos de la función resulta ser $\bot$, entonces la función devolverá $\bot$ también. Es decir,

$$f(x_1, x_2, ..., \bot, ..., x_n) = \bot.$$

DEFINICIÓN 1.2.A. *Sea A un conjunto. Llamaremos* formalismo *a un conjunto $\overline{X}$ de funciones de la forma $A \to A$ en donde cada función es generada a partir de ciertos constructores, previamente definidos para tal conjunto.*

Es inmediato ver que un formalismo sólo dependerá de A y de los constructores. Un ejemplo muy sencillo de formalismo es el de las funciones naturales constantes. Basta con tomar $A = \mathbb{N}$ y los operadores cero (que siempre devuelve la función nula) y sucesor (que, dada una función natural, genera otra similar a la anterior, pero cuyas imágenes están incrementadas en una unidad). Así, por ejemplo, la función que retorna 5 se obtiene con cinco aplicaciones del operador sucesor sobre el operador cero.

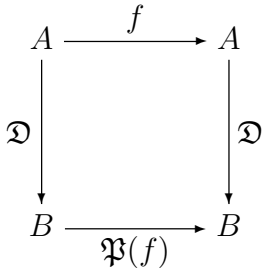
DEFINICIÓN 1.2.B. *Diremos que un formalismo $\overline{X}$ (de funciones de la forma $A \to A$)* puede ser representado *por otro formalismo $\overline{Y}$ (de funciones de la forma $B \to B$) cuando existen dos funciones $\mathfrak{D}$ y $\mathfrak{P}$ tales que:*

$$\mathfrak{D} : A \to B \quad \textit{inyectiva},$$

$$\mathfrak{P} : \overline{X} \to \overline{Y},$$

$$\forall f \in \overline{X}, a \in A \bullet \mathfrak{P}(f)(\mathfrak{D}(a)) = \mathfrak{D}(f(a)). \tag{1.2.1}$$

$\mathfrak{D}$ es una función que transforma los dominios de ambos formalismos y $\mathfrak{P}$ es otra función que permite generar una función en un formalismo a partir de cómo es en el otro. La ecuación (1.2.1) determina que cualquier función f en el formalismo $\overline{X}$ tiene una contrapartida en el $\overline{Y}$ que, bajo alguna transformación de los valores de entrada, ejecuta la misma operación que f.
Para comprender mejor la idea, podemos representar la ecuación (1.2.1) como un diagrama conmutativo[1]:



[1]No es raro usar un diagrama conmutativo, dada la similitud que tiene este concepto con los *funtores* empleados en teoría de categorías.

Notamos como $\overline{X}\dot{\subset}\overline{Y}$ al hecho de que $\overline{X}$ pueda *ser representado* por $\overline{Y}$. Cuando $\overline{X}\dot{\subset}\overline{Y}$ y $\overline{Y}\dot{\subset}\overline{X}$ simultaneamente, entonces diremos que ambos formalismos *tienen el mismo poder de expresión*, y escribiremos $\overline{X} \doteq \overline{Y}$. Es inmediato comprobar que $\dot{\subset}$ es una relación de orden y que $\doteq$ es una relación de equivalencia, como se resume a continuación:

(a) **Reflexión de** $\dot{\subset}$: proponemos $\mathfrak{P}(x) = x, \mathfrak{D}(a) = a \implies X\dot{\subset}X$

(b) **Transitividad de** $\dot{\subset}$: $X\dot{\subset}Y \wedge Y\dot{\subset}Z \implies$
$\exists\mathfrak{P}_1(x) = y, \mathfrak{D}_1(a) = b, \mathfrak{P}_2(y) = z, \mathfrak{D}_2(b) = c$
proponemos $\mathfrak{P} = \mathfrak{P}_2 \circ \mathfrak{P}_1, \mathfrak{D} = \mathfrak{D}_2 \circ \mathfrak{D}_1 \implies X\dot{\subset}Z$

(c) **Reflexión de** $\doteq$: es una consecuencia de la reflexión de $\dot{\subset}$.

(d) **Simetría de** $\doteq$: inmediata, a partir de la definición de $\doteq$.

(e) **Transitividad de** $\doteq$: es una consecuencia de la transitividad de $\dot{\subset}$.

Se pueden consultar ideas similares en [GM88] (que estudia el monoide $Prim(\mathbb{N}, \mathbb{N})$ de las funciones recursivas primitivas de un parámetro) y [GM91] (idem. para el caso de $Comp(\mathbb{N}, \mathbb{N})$ de las funciones recursivas generales de un parámetro).

1.3. Las funciones recursivas primitivas

A continuación vamos a exponer cómo formar las *funciones recursivas primitivas* (véase capítulo 9 de [Kle52], capítulo 4 de [Bro93], o también [DSW94, Goo57]) que nos servirán de base para estudiar otras clases de formalismos.

Definición 1.3.A. *Las* funciones recursivas primitivas *son funciones naturales (de la forma* $\varphi : \mathbb{N}^n \to \mathbb{N}$*) formadas a partir de los siguientes constructores (y sólo de ellos):*
La función cero *es una función sin argumentos que siempre devuelve el valor cero:*

$\zeta : \mathbb{N}^0 \to \mathbb{N}$,
$\zeta() = 0$.

La función sucesor *es una función unaria que devuelve el argumento incrementado en uno:*

$\sigma : \mathbb{N}^1 \to \mathbb{N}$,
$\sigma(x) = x + 1$.

Las funciones de proyección *son funciones n-arias que devuelven uno de sus argumentos. Si* $1 \leq i \leq n$ *entonces:*

$\pi_i^n : \mathbb{N}^n \to \mathbb{N}$,
$\pi_i^n(x_1, x_2, \ldots, x_n) = x_i$.

Dada una función recursiva primitiva $\varphi : \mathbb{N}^n \to \mathbb{N}$ *(donde* $n > 0$*) y un conjunto de n funciones recursivas primitivas* $\omega_i : \mathbb{N}^m \to \mathbb{N}$ *(donde* $1 \leq i \leq n$*) se define la* composición *(también llamada substitución) de ellas como:*

$\phi[\varphi, \omega_1, \omega_2, \ldots, \omega_n] : \mathbb{N}^m \to \mathbb{N}$,
$\phi[\varphi, \omega_1, \omega_2, \ldots, \omega_n](x_1, x_2, \ldots, x_m) = \varphi(\omega_1(x_1, x_2, \ldots, x_m),$
$\omega_2(x_1, x_2, \ldots, x_m), \ldots, \omega_n(x_1, x_2, \ldots, x_m))$.

Dada las funciones recursivas primitivas $\beta : \mathbb{N}^n \to \mathbb{N}$ *y* $\varphi : \mathbb{N}^{n+2} \to \mathbb{N}$ *(aquí* $n \in \mathbb{N}$*) se define la* recursión primitiva *de ambas como:*

$\rho[\beta, \varphi] : \mathbb{N}^{n+1} \to \mathbb{N}$,
$\rho[\beta, \varphi](x_1, x_2, \ldots, x_n, 0) = \beta(x_1, x_2, \ldots, x_n)$,
$\rho[\beta, \varphi](x_1, x_2, \ldots, x_n, y + 1) = \varphi(x_1, x_2, \ldots, x_n, y, \rho[\beta, \varphi](x_1, x_2, \ldots, x_n, y))$.

Denotaremos al conjunto de las funciones recursivas primitivas como $\overline{\mathrm{FRP}}$.

Se utilizan los corchetes para diferenciar los argumentos que corresponden a la construcción de las FRPs de los que son parámetros de la función (en cuyo caso se utilizan los paréntesis). Para representar una FRP no es necesario hacer referencia a los parámetros; simplemente se puede definir de la siguiente forma:

$$\texttt{NombreFunción} \equiv \texttt{Fórmula sin paréntesis}.$$

EJEMPLO 1.3.B. *Supongamos que queremos realizar una función ternaria que retorne la constante dos. Es decir,*

$$\varphi(x, y, z) = 2.$$

La FRP correspondiente sería

$$\varphi \equiv \phi[\phi[\sigma, \phi[\sigma, \rho[\zeta, \pi_2^2]]], \pi_1^3].$$

Vamos a desglosar esta fórmula de adentro hacia afuera. La función $\rho[\zeta, \pi_2^2]$ representa una función unaria que retorna el cero. Si $\zeta' \equiv \rho[\zeta, \pi_2^2]$,

$$\zeta'(0) = \zeta() = 0, \tag{1.3.1}$$
$$\zeta'(y+1) = \pi_2^2(y, \zeta'(y)) = \zeta'(y). \tag{1.3.2}$$

La prueba de esto es inductiva. Para el caso en que $y = 0$ es trivial, como lo muestra la ecuación (1.3.1). Para el caso inductivo, consideremos la hipótesis de inducción $\zeta'(y) = 0$. Luego $\zeta'(y+1) = 0$ también, por la ecuación (1.3.2).
Ahora tenemos que $\phi[\sigma, \phi[\sigma, \zeta']]$ es una función unaria que retorna el 2 (es, simplemente, la aplicación del sucesor exactamente dos veces). Para convertirla en ternaria debemos componerla con una función que ya sea ternaria. Cualquier proyección de una terna es factible, en particular π_1^3.

Una característica interesante de las FRPs es que son *totales* (i.e. para cualquier entrada siempre retornan un valor de salida). Dicha característica está estrechamente vinculada al hecho de que las funciones se evaluan en una cantidad finita de pasos. Para el caso en que son funciones iniciales, o composición de funciones totales, es trivial. La recursión, en cambio, se computa con un bucle *while* acotado por una variable. Un ejemplo en pseudo-código es:

```
i := 0
z := f(x)
while i < y do z := g(x, i, z); i := i + 1
return z
```

Otra observación importante es que $\overline{\text{FRP}}$ es un formalismo, pues define funciones en un dominio[2]

$$A = \bigcup_{n \in \mathbb{N}} \mathbb{N}^n = \mathbb{N}^0 \cup \mathbb{N}^1 \cup \mathbb{N}^2 \cup \dots$$

[2]Si deseamos ser estrictos, deberíamos especificar qué sucede cuando intentamos calcular una función de n argumentos con m valores de entrada, donde $m \neq n$. Convenimos en que dicha función devolverá cero en estos casos:
$f : \mathbb{N}^n \to \mathbb{N}$,
$f(x_1, x_2, ..., x_m) = 0$.

1.4. Sintaxis de las funciones recursivas primitivas

No todas las cadenas compuestas por los símbolos 'ζ', 'σ', 'π_i^n', 'ϕ', 'ρ', '[', ']' y ',' forman una función recursiva primitiva. En primer lugar, la cadena debe estar sujeta a una gramática. La siguiente gramática (en formato BNF) es una manera de representar funciones recursivas primitivas sintácticamente válidas:

$$f ::= \zeta \mid \sigma \mid \pi_i^n \mid \phi[f, fl] \mid \rho[f, f] \tag{1.4.1}$$

$$l ::= \mid l, f \tag{1.4.2}$$

En (1.4.1) se establece cómo representar una FRP, mientras que (1.4.2) representa una lista, posiblemente vacía, de FRPs. Notemos que la composición siempre está obligada a tomar, al menos, dos FRPs.

Pero esto no es suficiente. Todavía hay cadenas que se ajustan a la gramática anterior y no tienen sentido, como $\phi[\zeta, \sigma]$. La función ζ no admite parámetros, pero la composición supone que debe admitir uno. Este *problema de tipos* se debe a que los esquemas de composición y recursión primitiva relacionan la *aridad* (i.e. cantidad de parámetros) de las funciones que están presentes.
Supongamos que, dada una cadena de símbolos, la gramática anterior genera una estructura como la siguiente:

$\zeta : \overline{\text{FRP}}$,
$\sigma : \overline{\text{FRP}}$,
$\pi : \mathbb{N} \times \mathbb{N} \to \overline{\text{FRP}}$,
$\phi : List^{\geq 2}(\overline{\text{FRP}}) \to \overline{\text{FRP}}$,
$\rho : \overline{\text{FRP}} \times \overline{\text{FRP}} \to \overline{\text{FRP}}$,

donde π_i^n toma el par (i, n) y ϕ toma una lista de FRPs de al menos 2 elementos. El siguiente esquema[3] proporciona una manera de calcular la cantidad de parámetros de una FRP,

$arity : \overline{\text{FRP}} \to \mathbb{N}$
$arity\ \zeta = 0$
$arity\ \sigma = 1$
$arity\ \pi_i^n =$ if $1 \leq i \leq n$ then n else **Error**
$arity\ \phi[\varphi, \omega_1, \omega_2, \ldots, \omega_n] =$ if $(arity\ \varphi = n) \wedge$ $(arity\ \omega_1 = arity\ \omega_2 = ... = arity\ \omega_n)$ then $arity\ \omega_1$ else **Error**
$arity\ \rho[\beta, \varphi] =$ if $(arity\ \beta + 2 = arity\ \varphi)$ then $arity\ \beta + 1$ else **Error**

dando error en caso de que la FRP no esté bien formada.

1.5. Ejemplos de funciones recursivas primitivas

A continuación vamos a ver representaciones de operaciones conocidas en términos de funciones recursivas primitivas. Más funciones recursivas primitivas aparecen en [Kle52] y [Goo57]: tetración, suma de series geométricas, división y resto, máximo común divisor, n-ésimo número primo y otras tantas.

[3] A estos esquemas los escribiremos de manera recursiva, al estilo de los lenguajes funcionales como *Haskell* y *ML*. En ellos, la primera línea declara el tipo.

(a) **Función constante**: $\varsigma_k^n(x_1, x_2, ..., x_n) = k$.
Para generar las constantes, sólo es necesario componer la función sucesor k veces con la función cero,

$$\begin{aligned}
&\varsigma_1^0() = \sigma(\zeta()) = \sigma(0) = 1 \implies \varsigma_1^0 \equiv \phi[\sigma, \zeta], \\
&\varsigma_2^0() = \sigma(\sigma(\zeta())) = \sigma(1) = 2 \implies \varsigma_2^0 \equiv \phi[\sigma, \varsigma_1^0], \\
&\quad \dots
\end{aligned}$$

En términos generales,

$$\boxed{\varsigma_k^0 \equiv \begin{cases} \zeta & \text{si } k = 0 \\ \phi[\sigma, \varsigma_{k-1}^0] & \text{si } k > 0 \end{cases}}$$

Para calcular las funciones constantes restantes utilizaremos un argumento similar al del ejemplo 1.3.B,

$$\boxed{\varsigma_k^1 \equiv \rho[\varsigma_k^0, \pi_2^2]}$$

$$\boxed{\varsigma_k^n \equiv \phi[\varsigma_k^1, \pi_1^n]}$$

donde $n > 1$. Observemos que

$$\begin{aligned}
\varsigma_k^1(0) &= \varsigma_k^0() = k, \\
\varsigma_k^1(x+1) &= \pi_2^2(x, \varsigma_k^1(x)) = \varsigma_k^1(x),
\end{aligned}$$

donde la igualdad $\varsigma_k^1(x) = k$ se prueba para toda x por inducción.

(b) **Función suma**: $\Sigma(x, y) = x + y$.
La suma puede ser generada con la recursión primitiva

$$\begin{aligned}
\Sigma(x, 0) &= x + 0 = x = \pi_1^1(x), \\
\Sigma(x, y+1) &= x + (y+1) = (x+y) + 1 = \Sigma(x, y) + 1 = \sigma(\pi_3^3(x, y, \Sigma(x, y))).
\end{aligned}$$

$$\boxed{\Sigma \equiv \rho[\pi_1^1, \phi[\sigma, \pi_3^3]]}$$

(c) **Función producto**: $\Pi(x, y) = xy$.
El producto puede ser generada con una recursión primitiva sobre la suma,

$$\begin{aligned}
\Pi(x, 0) &= x0 = 0 = \varsigma_0^1(x), \\
\Pi(x, y+1) &= x(y+1) = xy + x = \Sigma(\Pi(x, y), x) \\
&= \Sigma(\pi_3^3(x, y, \Pi(x, y)), \pi_1^3(x, y, \Pi(x, y))).
\end{aligned}$$

$$\boxed{\Pi \equiv \rho[\varsigma_0^1, \phi[\Sigma, \pi_3^3, \pi_1^3]]}$$

(d) **Función doble**: $\mathsf{Dbl}(x) = 2x$.

$$\mathsf{Dbl}(x) = x + x = \Sigma(\pi_1^1(x), \pi_1^1(x)).$$

$$\boxed{\mathsf{Dbl} \equiv \phi[\Sigma, \pi_1^1, \pi_1^1]}$$

(e) **Función cuadrado**: $\mathsf{Sq}(x) = x^2$.

$$\mathsf{Sq}(x) = x^2 = xx = \Pi(\pi_1^1(x), \pi_1^1(x)).$$

$$\boxed{\mathsf{Sq} \equiv \phi[\Pi, \pi_1^1, \pi_1^1]}$$

(f) **Función potencia de dos**: $\mathsf{Pot}(x) = 2^x$.

$$\begin{aligned}\mathsf{Pot}(0) &= 2^0 = 1 = \varsigma_1^0(),\\ \mathsf{Pot}(x+1) &= 2^{x+1} = 2^x 2 = \mathsf{Dbl}(\mathsf{Pot}(x)) = \mathsf{Dbl}(\pi_2^2(x, \mathsf{Pot}(x))).\end{aligned}$$

$$\boxed{\mathsf{Pot} \equiv \rho[\varsigma_1^0, \phi[\mathsf{Dbl}, \pi_2^2]]}$$

(g) **Función predecesor**: $\mathsf{P}(x) = x \dot{-} 1 = \begin{cases} 0 & \text{si } x = 0,\\ x-1 & \text{si } x > 0.\end{cases}$

$$\begin{aligned}\mathsf{P}(0) &= 0 = \zeta(),\\ \mathsf{P}(x+1) &= x + 1 - 1 = x = \pi_1^2(x, \mathsf{P}(x)).\end{aligned}$$

$$\boxed{\mathsf{P} \equiv \rho[\zeta, \pi_1^2]}$$

(h) **Función resta truncada**: $\mathsf{Mon}(x, y) = x \dot{-} y = \begin{cases} 0 & \text{si } x \leq y,\\ x-y & \text{si } x > y.\end{cases}$

$$\begin{aligned}\mathsf{Mon}(x, 0) &= x \dot{-} 0 = x = \pi_1^1(x),\\ \mathsf{Mon}(x, y+1) &= x \dot{-} (y+1) = (x \dot{-} y) \dot{-} 1\\ &= \mathsf{P}(\mathsf{Mon}(x, y)) = \mathsf{P}(\pi_3^3(x, y, \mathsf{Mon}(x, y))).\end{aligned}$$

$$\boxed{\mathsf{Mon} \equiv \rho[\pi_1^1, \phi[\mathsf{P}, \pi_3^3]]}$$

(i) **Función distancia**: $\mathsf{Dist}(x, y) = |x - y|$.

$$\mathsf{Dist}(x, y) = |x - y| = (x \dot{-} y) + (y \dot{-} x) = \Sigma(\mathsf{Mon}(x, y), \mathsf{Mon}(\pi_2^2(x, y), \pi_1^2(x, y))).$$

$$\boxed{\mathsf{Dist} \equiv \phi[\Sigma, \mathsf{Mon}, \phi[\mathsf{Mon}, \pi_2^2, \pi_1^2]]}$$

(j) **Función distintor**[4]: $\mathsf{D}(x) = 1 \dot{-} x = \begin{cases} 1 & \text{si } x = 0,\\ 0 & \text{si } x > 0.\end{cases}$

$$\begin{aligned}\mathsf{D}(0) &= 1 = \varsigma_1^0(),\\ \mathsf{D}(x+1) &= 0 = \varsigma_0^2(x, \mathsf{D}(x)).\end{aligned}$$

$$\boxed{\mathsf{D} \equiv \rho[\varsigma_1^0, \varsigma_0^2]}$$

[4]Algunos autores usan 0^x para notar esta función, otros usan $\overline{sg}$. Para denotar la función $\mathsf{D}(\mathsf{D}(x))$ usan *sgn* y *sg* respectivamente.

(k) **Función alternación**: $\mathsf{Alt}(x) = x \bmod 2 = \begin{cases} 0 & \text{si } x \text{ es par,} \\ 1 & \text{si } x \text{ es impar.} \end{cases}$

$$\begin{aligned} \mathsf{Alt}(0) &= 0 = \zeta(), \\ \mathsf{Alt}(x+1) &= \mathsf{D}(\mathsf{Alt}(x)) = \mathsf{D}(\pi_2^2(x, \mathsf{Alt}(x))). \end{aligned}$$

$$\boxed{\mathsf{Alt} \equiv \rho[\zeta, \phi[\mathsf{D}, \pi_2^2]]}$$

(l) **Función mitad**: $\mathsf{Hf}(x) = \lfloor \frac{x}{2} \rfloor$.

$$\begin{aligned} \mathsf{Hf}(0) &= 0 = \zeta(), \\ \mathsf{Hf}(x+1) &= \mathsf{Hf}(x) + \mathsf{Alt}(x) = \Sigma(\pi_2^2(x, \mathsf{Hf}(x)), \mathsf{Alt}(\pi_1^2(x, \mathsf{Hf}(x)))). \end{aligned}$$

$$\boxed{\mathsf{Hf} \equiv \rho[\zeta, \phi[\Sigma, \pi_2^2, \phi[\mathsf{Alt}, \pi_1^2]]]}$$

(m) **Función raíz cuadrada**: $\mathsf{Rt}(x) = \lfloor \sqrt{x} \rfloor$.

$$\begin{aligned} \mathsf{Rt}(0) &= 0 = \zeta(), \\ \mathsf{Rt}(x+1) &= \mathsf{Rt}(x) + \mathsf{D}((\mathsf{Rt}(x)+1)^2 - (x+1)) = \\ &= \Sigma(\pi_2^2(x, \mathsf{Rt}(x)), \mathsf{D}(\mathsf{Mon}(\mathsf{Sq}(\sigma(\pi_2^2(x, \mathsf{Rt}(x)))), \sigma(\pi_1^2(x, \mathsf{Rt}(x)))))). \end{aligned}$$

$$\boxed{\mathsf{Rt} \equiv \rho[\zeta, \phi[\Sigma, \pi_2^2, \phi[\mathsf{D}, \phi[\mathsf{Mon}, \phi[\mathsf{Sq}, \phi[\sigma, \pi_2^2]], \phi[\sigma, \pi_1^2]]]]]}$$

La prueba se basa en la relación

$$\lfloor \sqrt{x+1} \rfloor = \begin{cases} \lfloor \sqrt{x} \rfloor + 1 & \text{si } x+1 \text{ es cuadrado perfecto,} \\ \lfloor \sqrt{x} \rfloor & \text{si no,} \end{cases} \tag{1.5.1}$$

y en el hecho de que $\mathsf{D}((\lfloor \sqrt{x} \rfloor + 1)^2 - (x+1)) = 1$ sólo cuando $x+1$ es cuadrado perfecto (si $x+1$ no es cuadrado perfecto entonces $(\lfloor \sqrt{x} \rfloor + 1)^2 > x+1$).

1.6. Funciones doblemente recursivas

A principios del siglo XX existía la conjetura de que cualquier función total computable podía escribirse como una función recursiva primitiva (esto equivaldría a decir, en términos de formalismos, que $\overline{\mathrm{FRP}} \doteq \overline{\mathrm{MT}}$ donde $\overline{\mathrm{MT}}$ es el formalismo de las máquinas de Turing). A patir de esta conjetura, Hilbert[Hil26] planteó las siguientes preguntas:

- ⋆ ¿Existen esquemas de recursión que no son reducibles a recursión primitiva?
- ⋆ ¿Se puede escribir una función que sea recursiva y que no pertenezca a $\overline{\mathrm{FRP}}$?

las cuales fueron respondidas por Ackermann[Ack28][5]. En efecto, la función recursiva

$$\begin{aligned}\varphi(a,b,0) &= a+b,\\ \varphi(a,0,n+1) &= \alpha(a,n),\\ \varphi(a,b+1,n+1) &= \varphi(a,\varphi(a,b,n+1),n),\end{aligned}$$

donde $\alpha(a,0)=0$, $\alpha(a,1)=1$ y $\alpha(a,n+2)=a$, no pertenece a $\overline{\mathrm{FRP}}$ y, sin embargo, es computable y total. En su artículo, Ackermann afirma que si φ fuese recursiva primitiva, también lo sería la función $\xi(a)=\varphi(a,a,a)$ (de hecho, $\xi \equiv \phi[\varphi,\pi^1_1,\pi^1_1,\pi^1_1]$). Pero $\xi(a)$ crece más rápido que cualquier función recursiva primitiva (similarmente a cómo 2^x crece más rápido que cualquier polinomio en x).
R. Péter[Pét35] demostró también que hay funciones recursivas que no están en $\overline{\mathrm{FRP}}$ utilizando un método de diagonalización: Consideremos las funciones recursivas primitivas de un solo argumento

$$f_0(x), f_1(x), f_2(x), \ldots$$

Dado que estas funciones se pueden enumerar, puede verificarse que $f_x(x)+1$ no forma parte de esta enumeración. Lo que hizo Péter además, es escribir la función $p(x,z)=f_z(x)$ de manera recursiva (a este tipo de funciones se las conoce como *universales*[6]; véase p. 6 de [Grz53]).
Posteriormente, R. M. Robinson[Rob48] simplificó el esquema de Ackermann, demostrando que la función

$$\begin{aligned}\mathsf{A}(0,y) &= y+1,\\ \mathsf{A}(x+1,0) &= \mathsf{A}(x,1),\\ \mathsf{A}(x+1,y+1) &= \mathsf{A}(x,\mathsf{A}(x+1,y)),\end{aligned}$$

tampoco es recursiva primitiva. También es interesante el artículo [Tat03] en donde se extiende la función doblemente recursiva de Robinson a muchas variables.
No obstante, todas las funciones anteriores (e.g. φ, p, A) se pueden reducir a un esquema que veremos a continuación.

Definición 1.6.A. *Las* funciones doblemente recursivas *son el resultado de agregar a las funciones recursivas primitivas el esquema de doble recursión, que se muestra a continuación. Dada las funciones doblemente recursivas* $\beta_1 : \mathbb{N}^{n+1} \to \mathbb{N}$, $\beta_2 : \mathbb{N}^{n+2} \to \mathbb{N}$, $\beta_3 : \mathbb{N}^{n+1} \to \mathbb{N}$, $\beta_4 : \mathbb{N}^{n+4} \to \mathbb{N}$ *y* $\beta_5 : \mathbb{N}^{n+3} \to \mathbb{N}$ *(aquí* $n \in \mathbb{N}$*) se define la doble recursión de todas ellas como* $\varphi \equiv \delta[\beta_1,\beta_2,\beta_3,\beta_4,\beta_5]$ *donde*

$\varphi : \mathbb{N}^{n+2} \to \mathbb{N}$,
$\varphi(x_1,\ldots,x_n,0,z) = \beta_1(x_1,\ldots,x_n,z)$,
$\varphi(x_1,\ldots,x_n,y+1,0) = \beta_2(x_1,\ldots,x_n,y,\varphi(x_1,\ldots,x_n,y,\beta_3(x_1,\ldots,x_n,y)))$,

[5]No obstante, el primero que descubrió una función doblemente recursiva que no era recursiva primitiva fue Gabriel Sudan[Sud27] (Sudan y Ackermann eran estudiantes de Hilbert). La función es

$$\begin{aligned}\varphi_0(a,b) &= a+b,\\ \varphi_{n+1}(a,0) &= a,\\ \varphi_{n+1}(a,b+1) &= \varphi_n(\varphi_{n+1}(a,b),\varphi_{n+1}(a,b)+b+1).\end{aligned}$$

Para más información, se puede consultar [CMT79].

[6]Este nombre viene heredado de las máquinas de Turing universales (i.e. máquinas de Turing que pueden ejecutar cualquier máquina de Turing).

$$\varphi(x_1,\ldots,x_n,y+1,z+1)=\beta_4(x_1,\ldots,x_n,y,z,\varphi(x_1,\ldots,x_n,y+1,z),$$
$$\varphi(x_1,\ldots,x_n,y,\beta_5(x_1,\ldots,x_n,y,z,\varphi(x_1,\ldots,x_n,y+1,z)))).$$

Si denotamos al conjunto de las funciones doblemente recursivas como $\overline{\mathrm{FDR}}$ *entonces*

(a) $\alpha\in\overline{\mathrm{FRP}}\implies\alpha\in\overline{\mathrm{FDR}}$,

(b) $\beta_1,\beta_2,\beta_3,\beta_4,\beta_5\in\overline{\mathrm{FDR}}\implies\delta[\beta_1,\beta_2,\beta_3,\beta_4,\beta_5]\in\overline{\mathrm{FDR}}$,

siempre que se respeten las aridades de las funciones.

Ejemplo 1.6.B. *Ahora mostraremos que* $\mathsf{A}\in\overline{\mathrm{FDR}}$.

$$\begin{aligned}\mathsf{A}(0,y)&=\sigma(y),\\ \mathsf{A}(x+1,0)&=\pi^2_2(x,\mathsf{A}(x,\varsigma^1_1(x))),\\ \mathsf{A}(x+1,y+1)&=\pi^4_4(x,y,\mathsf{A}(x+1,y),\mathsf{A}(x,\pi^3_3(x,y,\mathsf{A}(x+1,y)))).\end{aligned}$$

Formalmente,

$$\boxed{\mathsf{A}\equiv\delta[\sigma,\pi^2_2,\varsigma^1_1,\pi^4_4,\pi^3_3]}$$

Al igual que las FRPs, las funciones doblemente recursivas también constituyen un formalismo. Luego, resulta posible comparar este formalismo con el de las funciones recursivas primitivas: Dado que $\mathsf{A}\notin\overline{\mathrm{FRP}}$, es natural pensar que $\overline{\mathrm{FRP}}\dot{\subset}\overline{\mathrm{FDR}}$ (donde $\mathfrak{D}$ y $\mathfrak{P}$ son funciones identidad apropiadas) de manera estricta.

Es posible hacer recursiones de mayores órdenes (e.g. *triples recursiones*). Péter[Pét36] estudió esta clase de esquemas, llamadas en inglés *k-fold recursions*, en donde aparece la siguiente jerarquía: Sea F_1 el conjunto de las funciones recursivas primitivas, F_2 el de las funciones doblemente recursivas, F_3 el de las funciones triplemente recursivas, etc. Entonces $F_1\dot{\subset}F_2\dot{\subset}F_3\dot{\subset}\cdots\dot{\subset}F_k\dot{\subset}\cdots$ (estrictamente). Lamentablemente, existen todavía funciones recursivas que no pertenecen a ninguno de estos conjuntos.

Teorema 1.6.C. *Sea* $\mathscr{C}$ *un conjunto enumerable de constructores tales que todas las funciones que puedan generar son totales y de la forma* $\mathbb{N}^n\to\mathbb{N}$. *Sea* $\mathscr{F}$ *el conjunto de funciones generadas a partir de tales constructores (y sólo de ellos). Entonces existe una función total computable que no pertenece a este conjunto.*

Demostración. Lo que sigue es sólo un bosquejo: Por la enumerabilidad del conjunto $\mathscr{C}$, se pueden enumerar también todas las funciones que se pueden construir a partir de ellos (como para dar un ejemplo notemos que una función se puede escribir como una secuencia finita de símbolos que la representa, pero este conjunto es enumerable siempre que lo sea el conjunto de símbolos que utilizamos) de manera que $\mathscr{F}$ es enumerable.
Luego, sea $f_0,f_1,f_2,...$ la sucesión de funciones generadas. Entonces podemos pensar en la función $g(x)$ que busca la x-ésima función, la aplica (usando x en cada entrada) y le suma 1,

$$g(x)=f_x(x,x,...,x)+1.$$

Claramente g es una función total computable, pero si g pudiera construirse a partir de los constructores dados en $\mathscr{C}$, entonces existiría un n tal que $g(x)=f_n(x)$. Ahora, en particular

$$f_n(n)=g(n)=f_n(n)+1,$$

que resulta una contradicción. Entonces $g\notin\mathscr{F}$. □

1.7. Minimización

A partir del Teorema 1.6.C surge la necesidad de insertar un constructor que genere funciones parciales, aunque las funciones que deseemos construir sean totales.

DEFINICIÓN 1.7.A. *Sea una función, posiblemente parcial,* $f : \hat{\mathbb{N}}^{n+1} \to \hat{\mathbb{N}}$. *Llamamos* minimización de f *(y la denotamos* μf*) a la función* $g : \hat{\mathbb{N}}^n \to \hat{\mathbb{N}}$ *definida por*

$$g(x_1, x_2, \ldots, x_n) = \min_{y \in \mathbb{N}} \{ f(x_1, x_2, \ldots, x_n, y) = 0 \}.$$

El mínimo $y \in \mathbb{N}$ *para el cual* $f(\mathbf{x}, y)$ *se anula es el valor de* $g(\mathbf{x})$ *(aquí* $\mathbf{x} \in \hat{\mathbb{N}}^n$*). Sin embargo, vamos a convenir en que* $g(\mathbf{x}) = \perp$ *siempre que se satisfaga alguna de las siguientes condiciones:*

- $\star$ $\nexists y \in \mathbb{N} \bullet f(\mathbf{x}, y) = 0$,
- $\star$ $\exists y \in \mathbb{N} \bullet (f(\mathbf{x}, y) = 0 \wedge \exists y' < y \bullet f(\mathbf{x}, y') = \perp)$.

El operador μ *se llama minimizador.*

Es decir, $g(\mathbf{x})$ estará definida sólo cuando $f(\mathbf{x}, y)$ se anule para algún y natural, y además $f(\mathbf{x}, y')$ está definida y no se anula para todos los valores de y' entre cero e $y - 1$. Estas imposiciones sobre cómo debe comportarse el minimizador están ligadas a la implementación del mismo usando un bucle while (aquí es donde entra en juego la aparición del símbolo $\perp$, que indica que el programa está atrapado en un lazo infinito):

```
y := 0
while f(x, y) not zero do y := y + 1
return y
```

Definiciones alternativas del minimizador se encuentran en [Kle52], en donde se proporciona además un estudio completo sobre las funciones parciales.
Por el momento, nos interesa trabajar con el siguiente formalismo, también definido por Kleene[Kle52].

DEFINICIÓN 1.7.B. *Las* funciones recursivas generales *son el resultado de agregar a las funciones recursivas primitivas el esquema de minimización siguiente.*
Dada la función recursiva general $\varphi : \hat{\mathbb{N}}^{n+1} \to \hat{\mathbb{N}}$ *(aquí* $n \in \mathbb{N}$*) se define su minimización como:*
$\mu\varphi : \hat{\mathbb{N}}^n \to \hat{\mathbb{N}}$
$\mu\varphi(x_1, x_2, \ldots, x_n) = \min_{y \in \mathbb{N}} \{ \varphi(x_1, x_2, \ldots, x_n, y) = 0 \}$
Denotamos al conjunto de las funciones recursivas generales como $\overline{\text{FRG}}$.

Un descubrimiento importante en la teoría de la computación, conocido como Tesis de Church, fue que *las funciones recursivas generales tienen el mismo poder expresivo que las máquinas de Turing* (es decir, que $\overline{\text{FRG}} \doteq \overline{\text{MT}}$). De hecho, cualquier función que se pueda escribir usando esquemas recursivos (i.e. mediante un sistema de ecuaciones donde la función pueda estar definida en términos de si mísma.) pertenece a $\overline{\text{FRG}}$. Para mostrar una idea de los procesos que se deben llevar a cabo para la transformación de cualquier esquema recursivo en FRGs, nosotros probaremos que $\overline{\text{FDR}} \dot{\subset} \overline{\text{FRG}}$ en el capítulo siguiente.

Kleene demostró, además, que las FRGs se pueden *normalizar*. Es decir, cualquier FRG se puede escribir como $\phi[\alpha, \mu\gamma]$ (llamada *forma normal de Kleene*) donde α y γ son FRPs tales que $arity\ \alpha = 1$ y $arity\ \gamma \geq 1$. La aridad de la FRG generada será, por supuesto, $arity\ \gamma - 1$. Para mayor información, véase [Kle36a].

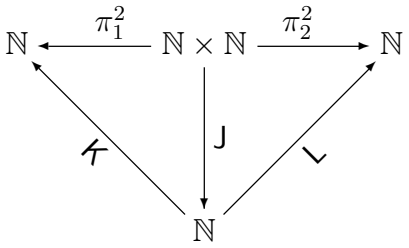
A pesar de que la minimización pueda generar funciones parciales, **nosotros nos limitaremos a demostrar propiedades sobre funciones totales**. Es decir, trabajaremos con el subconjunto de funciones totales de $\overline{\text{FRG}}$.

Capítulo 2

Reducción de los esquemas de recursión

2.1. Funciones de parificación

Una de las propiedades más importantes del conjunto de números naturales es que, al ser infinito, puede ponerse en correspondencia con ciertos subconjuntos propios de él. Más aun, puede ponerse en correspondencia con el producto cartesiano de si mísmo. Estos hechos fueron estudiados por Cantor entre 1874 y 1897, quien creó una función biyectiva $\mathsf{J} : \mathbb{N}^2 \to \mathbb{N}$ que codificaba pares de naturales con la suficiente información como para poder recuperarlos. Esta función, junto con sus inversas $\mathsf{K}, \mathsf{L} : \mathbb{N} \to \mathbb{N}$, conforman el siguiente diagrama conmutativo.



Nuestro interés será, por el momento, analizar esta clase de funciones.

Definición 2.1.A. *Diremos que J, K, L son* funciones de parificación *si satisfacen*[1]

$$\forall x, y \in \mathbb{N} \bullet K(J(x, y)) = x \wedge L(J(x, y)) = y. \tag{2.1.1}$$

Definición 2.1.B. *Diremos que que J, K, L son* funciones de parificación completas *si, además de ser funciones de parificación, satisfacen*

$$\forall x \in \mathbb{N} \bullet J(K(x), L(x)) = x. \tag{2.1.2}$$

Algunos ejemplos de funciones (recursivas primitivas) de parificación son

- ⋆ $J(x, y) = 2^x 3^y$,
  $K(x) =$ número de veces que x divide a 2,
  $L(x) =$ número de veces que x divide a 3.

[1] Cada autor utiliza una notación distinta para estas funciones. Por ejemplo, [DSW94] usa $\langle x, y \rangle$, $l(x)$ y $r(x)$. Nosotros adoptaremos la notación original de Robinson.

- $J(x,y) = (x+y)^2 + x$,
  $K(x) = x - \lfloor\sqrt{x}\rfloor^2$,
  $L(x) = \lfloor\sqrt{x}\rfloor \stackrel{\bullet}{-} K(x)$.

- $J(x,y) = ((x+y)^2 + x)^2 + y$,
  $K(x) = L(\lfloor\sqrt{x}\rfloor)$,
  $L(x) = x - \lfloor\sqrt{x}\rfloor^2$.

Algunos ejemplos de funciones (recursivas primitivas) de parificación completas son

- Utilizando notación binaria:
  $J(\cdots b_3^x b_2^x b_1^x b_0^x, \cdots b_3^y b_2^y b_1^y b_0^y) = \cdots b_3^y b_3^x b_2^y b_2^x b_1^y b_1^x b_0^y b_0^x$,
  $K(\cdots b_5 b_4 b_3 b_2 b_1 b_0) = \cdots b_4 b_2 b_0$,
  $L(\cdots b_5 b_4 b_3 b_2 b_1 b_0) = \cdots b_5 b_3 b_1$.

- $J(x,y) = 2^x(2y+1) - 1$,
  $K(x) =$ número de veces que $x+1$ divide a 2,
  $L(x) = \dfrac{\frac{x+1}{2^{K(x)}} - 1}{2}$.

Para nuestro desarrollo será suficiente utilizar las funciones originales de Cantor, quien propuso barrer diagonalmente los pares de números naturales, como se indica en la tabla:

$y\downarrow \; x\rightarrow$	0	1	2	3	4	5
0	0	2	5	9	14	20
1	1	4	8	13	19	26
2	3	7	12	18	25	33
3	6	11	17	24	32	41
4	10	16	23	31	40	50
5	15	22	30	39	49	60

A las funciones de Cantor las denotaremos con J, K y L. Para referirnos a cualquier otra terna de funciones que sean de parificación, lo haremos con J, K y L. La definición de J es:

$$\mathsf{J}(x,y) = \frac{(x+y)^2 + 3x + y}{2}.$$

Claramente, para cualquier $y \in \mathbb{N}$, $\mathsf{J}(0,y)$ es un número triangular (recordemos que los números triangulares son $\sum_{n=0}^{y} n = \frac{y^2+y}{2}$). Estos números pueden calcularse con la función recursiva primitiva $\blacktriangle$, donde

$$\blacktriangle(x) = \left\lfloor \frac{x^2+x}{2} \right\rfloor, \tag{2.1.3}$$

$$\blacktriangle \equiv \phi[\mathsf{Hf}, \phi[\Sigma, \mathsf{Sq}, \pi_1^1]].$$

Aplicando la resolvente de segundo grado, podemos hallar la inversa de esta función: $\frac{\sqrt{8x+1}-1}{2}$. Esta última función resultará de gran relevancia si la calculamos usando partes enteras (las restas que aparecen pueden ser reemplazas por cualquier función de resta, como Mon o Dist, dado que el sustraendo siempre es mayor o igual al minuendo).

$$\blacktriangledown(x) = \left\lfloor \frac{\lfloor\sqrt{8x+1}\rfloor - 1}{2} \right\rfloor, \tag{2.1.4}$$

$$\blacktriangledown \equiv \phi[\mathsf{Hf}, \phi[\mathsf{P}, \phi[\mathsf{Rt}, \phi[\sigma, \phi[\mathsf{Dbl}, \phi[\mathsf{Dbl}, \mathsf{Dbl}]]]]]].$$

Si bien $\blacktriangledown(\blacktriangle(x)) = x$, la función $\blacktriangle(\blacktriangledown(x))$ retorna *el mayor número triangular menor o igual a* x. Análogamente $\blacktriangle(\blacktriangledown(x)+1)$ es el menor número triangular mayor o igual a x (estos hechos se pueden comprobar fácilmente). Observemos de la tabla que devuelve los valores de J, que

$$\blacktriangle(\blacktriangledown(\mathsf{J}(x,y)) = \mathsf{J}(0,y) \leq \mathsf{J}(x,y) \leq \mathsf{J}(x,0) = \blacktriangle(\blacktriangledown(\mathsf{J}(x,y)) + 1) - 1.$$

De aquí podemos inferir las ecuaciones

$$\mathsf{J}(x,y) = \blacktriangle(x+y) + x, \tag{2.1.5}$$
$$\mathsf{K}(x) = x - \blacktriangle(\blacktriangledown(x)), \tag{2.1.6}$$
$$\mathsf{L}(x) = \blacktriangle(\blacktriangledown(x)+1) - x - 1. \tag{2.1.7}$$

Todas ellas son recursivas primitivas, ya que

$$\mathsf{J} \equiv \phi[\Sigma, \phi[\blacktriangle, \Sigma], \pi^2_1],$$
$$\mathsf{K} \equiv \phi[\mathsf{Mon}, \pi^1_1, \phi[\blacktriangle, \blacktriangledown]],$$
$$\mathsf{L} \equiv \phi[\mathsf{P}, \phi[\mathsf{Mon}, \phi[\blacktriangle, \phi[\sigma, \blacktriangledown]], \pi^1_1]].$$

Teorema 2.1.C. *Las funciones* $\mathsf{J}, \mathsf{K}, \mathsf{L}$ *son de parificación completas.*

Demostración. Lo más importante es que $\mathsf{J}(x,y)$ está entre dos números triangulares consecutivos,

$$\blacktriangle(x+y) \leq \mathsf{J}(x,y) \leq \blacktriangle(x+y+1),$$

pues

$$\frac{(x+y)^2+x+y}{2} \leq \frac{(x+y)^2+3x+y}{2} \leq \frac{(x+y)^2+3x+3y+2}{2}.$$

Luego $\blacktriangledown(\mathsf{J}(x,y)) = x+y$, y usando (2.1.6) y (2.1.7),

$$\mathsf{K}(\mathsf{J}(x,y)) = \mathsf{J}(x,y) - \blacktriangle(x+y) = \frac{(x+y)^2+3x+y}{2} - \frac{(x+y)^2+x+y}{2} = x,$$
$$\begin{aligned}\mathsf{L}(\mathsf{J}(x,y)) &= \blacktriangle(x+y+1) - \mathsf{J}(x,y) - 1 \\ &= \frac{(x+y)^2+3x+3y+2}{2} - \frac{(x+y)^2+3x+y}{2} - 1 = y.\end{aligned}$$

Para que sean funciones de parificación completas es preciso que $\mathsf{J}(\mathsf{K}(n), \mathsf{L}(n)) = n$ para todo n. Separaremos la prueba para el caso en que n es triangular (en cuyo caso, lo llamaremos t). Si t es un número triangular, existe un x tal que $t = \blacktriangle(x)$ (respectivamente $x = \blacktriangledown(t)$), y

$$\mathsf{J}(\mathsf{K}(t), \mathsf{L}(t)) = \mathsf{J}(t - \blacktriangle(x), \blacktriangle(x+1) - t - 1) = \mathsf{J}(0, \frac{(x+1)^2+x+1}{2} - \frac{x^2+x}{2} - 1),$$

de donde resulta $\mathsf{J}(\mathsf{K}(t), \mathsf{L}(t)) = \mathsf{J}(0,x) = \blacktriangle(x) = t$.
Ahora, sea n un natural acotado entre dos números triangulares, es decir

$$t = \blacktriangle(x) < n < \blacktriangle(x+1).$$

Entonces

$$\begin{aligned}\mathsf{J}(\mathsf{K}(n), \mathsf{L}(n)) &= \mathsf{J}(n - \blacktriangle(x), \blacktriangle(x+1) - n - 1) = \mathsf{J}(n-t, x+t-n) \\ &= \frac{x^2+3(n-t)+(x+t-n)}{2} = \frac{x^2+x}{2} + n - t = n.\end{aligned}$$

□

2.2. Estructuras de datos recursivas primitivas

Ya pudimos representar pares ordenados con números naturales, ¿será posible representar estructuras más complejas? Según Cantor, es posible poner en correspondencia cualquier conjunto enumerable con $\mathbb{N}$. Esto implica que cualquier estructura que pueda enumerar sus elementos, también tiene una codificación usando números naturales. Esto es útil si recordamos que las funciones que manipulamos son a valores naturales. Lo que haremos en esta sección es mostrar algunas estructuras de datos (que llamaremos *tipos de datos abstracto*) y su implementación usando funciones recursivas primitivas.
Un *tipo de datos abstracto*, o más simplemente TAD, es un conjunto de funciones que comparten algún tipo X en común, y están vinculadas a través de un sistema de ecuaciones funcionales que definen la semántica del TAD. El siguiente método (llamado *especificación algebráica de un TAD*) para definir TADs es bien conocido en lenguajes funcionales como Haskell y ML, y en lenguajes de especificación como Coq y Larch. Para representar un TAD X se utiliza un recuadro

`signatura de funciones (sintaxis)`
`axiomas (semántica)`

Notemos que el recuadro consta de dos campos. En el campo *signatura de funciones* se escriben todas las funciones pertenecientes al TAD con sus respectivos tipos (uno de ellos será X) y en el campo *axiomas* se escriben todas las ecuaciones que relacionan las funciones. Las funciones pueden clasificarse en constructores, observadores y operadores (véase cualquier documentación sobre TADs).
A continuación definimos los TADs que usaremos en el resto del capítulo.

(a) Tipo de datos *Bool*: Este TAD sirve para representar valores booleanos y realizar operaciones lógicas con ellos. Los constructores son $false$ y $true$, el observador es if, y los operadores son *not*, *and* y *or* como se muestra en el recuadro:

$false : Bool$
$true : Bool$
$if : Bool \times X \times X \to X$
$not : Bool \to Bool$
$and : Bool \times Bool \to Bool$
$or : Bool \times Bool \to Bool$
$if(false, x, y) = y$
$if(true, x, y) = x$
$not(x) = if(x, false, true)$
$and(x, y) = if(x, y, false)$
$or(x, y) = if(x, true, y)$

(b) Tipo de datos *Nat*: Este TAD sirve para representar números naturales y realizar operaciones aritméticas con ellos. Los constructores son $zero$ y suc, los observadores son $iszero$ y $pred$, y los operadores son sum y $prod$ como se muestra en el recuadro:

$zero : Nat$ $suc : Nat \to Nat$ $iszero : Nat \to Bool$ $pred : Nat \to Nat$ $sum : Nat \times Nat \to Nat$ $prod : Nat \times Nat \to Nat$
$iszero(zero) = true$ $iszero(suc(x)) = false$ $pred(suc(x)) = x$ $sum(x, y) = if(iszero(x), y, suc(sum(pred(x), y)))$ $prod(x, y) = if(iszero(x), zero, sum(x, prod(pred(x), y)))$

(c) Tipo de datos *Pair*: Este TAD sirve para encapsular dos valores de tipos de datos arbitrarios en uno sólo, al estilo de un producto cartesiano. El constructor es *pair*, y los observadores son fst y *snd* como se muestra en el recuadro:

$pair : X \times Y \to Pair$ $fst : Pair \to X$ $snd : Pair \to Y$
$fst(pair(x, y)) = x$ $snd(pair(x, y)) = y$

(d) Tipo de datos *List*: Este TAD sirve para representar listas de valores de un tipo de datos en particular. Los constructores son *nil* y *cons*, los observadores son *isnull*, *head* y *tail*, y los operadores son *concat*, *reverse* y *length* como se muestra en el recuadro:

$nil : List$ $cons : X \times List \to List$ $isnull : List \to Bool$ $head : List \to X$ $tail : List \to List$ $concat : List \times List \to List$ $reverse : List \to List$ $length : List \to Nat$
$isnull(nil) = true$ $isnull(cons(x, y)) = false$ $head(cons(x, y)) = x$ $tail(cons(x, y)) = y$ $concat(x, y) = if(isnull(x), y, cons(head(x), concat(tail(x), y)))$ $reverse(x) = if(isnull(x), nil, concat(reverse(tail(x)), cons(head(x), nil)))$ $length(x) = if(isnull(x), zero, suc(length(tail(x))))$

Los TADs presentados (a excepción de *Bool*) pueden ponerse en correspondencia uno-a-uno con los números naturales. En Nat la correspondencia es trivial, en $Pair$ también es obvia si usamos funciones de parificación completas, y en $List$ se puede usar el siguiente esquema:

$$nil \longleftrightarrow 0$$
$$cons(x, y) \longleftrightarrow \mathsf{J}(x, y) + 1$$

Aunque sólo basta con encontrar una correspondencia inyectiva entre cada elemento del TAD y el conjunto de los naturales, el hecho de que sea biyectiva nos permite comparar las estructuras usando la igualdad de números naturales. Así, dos pares (o dos listas) serán iguales si y sólo si los números que las representan también son iguales. Una función apropiada para comparar dos elementos de un TAD es la *delta de Kronecker*,

$$\delta(x,y) = \mathsf{D}(\mathsf{Dist}(x,y)) = \begin{cases} 0 & \text{si } x \neq y, \\ 1 & \text{si } x = y. \end{cases}$$

Ahora veremos las implementaciones de cada TAD con funciones recursivas primitivas.

(a) Tipo de datos *Bool*: Hacemos corresponder $false$ con el cero, y $true$ con cualquier número no nulo[2]. Aclaramos que esta correspondencia permite ver a δ como una relación matemática (i.e. $\delta : X \times X \to Bool$, en vez de $\delta : X \times X \to \mathbb{N}$). La implementación es

$$\boxed{\begin{array}{l} false \equiv \varsigma_0^n \\ true \equiv \varsigma_1^n \\ if \equiv \phi[\rho[\pi_2^2, \pi_1^4], \pi_2^3, \pi_3^3, \pi_1^3] \\ not \equiv \mathsf{D} \\ and \equiv \Pi \\ or \equiv \Sigma \end{array}}$$

en donde if verifica la siguiente fórmula[3]:

$$if(x,y,z) = \begin{cases} z & \text{si } x = 0, \\ y & \text{si } x > 0. \end{cases}$$

(b) Tipo de datos *Nat*: La correspondencia es trivial. A $zero$ le corresponde el cero, y a $suc(x)$ el sucesor de x. La implementación es

$$\boxed{\begin{array}{l} zero \equiv \varsigma_0^n \\ suc \equiv \sigma \\ iszero \equiv \mathsf{D} \\ pred \equiv \mathsf{P} \\ sum \equiv \Sigma \\ prod \equiv \Pi \end{array}}$$

Observemos que $pred(zero)$ no está especificado, así que podemos asignarle cualquier valor de Nat. Nosotros optamos por $zero$.

(c) Tipo de datos *Pair*. Se utilizan las funciones de parificación de Cantor.

$$\boxed{\begin{array}{l} pair \equiv \mathsf{J} \\ fst \equiv \mathsf{K} \\ snd \equiv \mathsf{L} \end{array}}$$

[2]Algunos autores representan a $true$ con el cero, y a $false$ con cualquier otro número. Respectivamente, *and* resulta ser la suma y *or* el producto. Además $\delta \equiv \mathsf{Dist}$.

[3]Una alternativa es: $if(x,y,z) = \mathsf{D}(x)z + \mathsf{D}(\mathsf{D}(x))y$.
Formalmente, $if \equiv \phi[\Sigma, \phi[\Pi, \phi[\mathsf{D}, \pi_1^1], \pi_1^3], \phi[\Pi, \phi[\mathsf{D}, \phi[\mathsf{D}, \pi_1^1]], \pi_1^2]]$.

(d) Tipo de datos *List*. Hacemos corresponder *nil* con el cero, y una lista no vacía con números no nulos.

$$\boxed{\begin{array}{l} nil_n \equiv \varsigma_0^n \\ cons \equiv \phi[\sigma, \mathsf{J}] \\ isnull \equiv \mathsf{D} \\ head \equiv \phi[\mathsf{K}, \mathsf{P}] \\ tail \equiv \phi[\mathsf{L}, \mathsf{P}] \end{array}}$$

Las funciones *concat*, *reverse* y *length* requieren mayor análisis. De la especificación podemos inferir:

$$concat \begin{cases} concat'(x, 0) & = x, \\ concat'(x, y+1) & = \mathsf{J}(\mathsf{K}(y), concat'(x, \mathsf{L}(y))) + 1, \\ concat(x, y) & = concat'(y, x). \end{cases} \tag{2.2.1}$$

$$reverse \begin{cases} reverse(0) & = 0, \\ reverse(x+1) & = concat(reverse(\mathsf{L}(x)), \mathsf{J}(\mathsf{K}(x), 0) + 1). \end{cases} \tag{2.2.2}$$

$$length \begin{cases} length(0) & = 0, \\ length(x+1) & = length(\mathsf{L}(x)) + 1. \end{cases} \tag{2.2.3}$$

Cada función está definida en términos de si mísma, pero la recursión no es primitiva. Por ejemplo, en (2.2.3), $length(x+1)$ se evalúa usando $length(\mathsf{L}(x))$ y no $length(x)$. En los otros casos sucede algo similar. Aunque estos casos no sean explícitamente recursivos primitivos, se pueden reducir a recursión primitiva. Estos esquemas reciben el nombre de *Course-of-values recursion* y se tratarán en la sección siguiente. Por el momento, sólo agregaremos que la reducción es posible debido a que $x \geq \mathsf{L}(x)$ para todo x.

Algunas de las estructuras de datos tienen asociados tipos genéricos X e Y que, en la implementación, acaban siendo números naturales (que es lo único que podemos manipular). Esto nos permite componer las estructuras. Por ejemplo, podemos armar pares ordenados de naturales y booleanos, o incluso listas de listas. Además, *siempre que compongamos estructuras que tienen correspondencia uno-a-uno, propagaremos esta propiedad*. Así, por ejemplo, si armamos listas de pares de naturales, cada una de ellas tendrá asociada un único número natural. Por ello, será posible compararlas entre sí.

Existen otras estructuras que pueden representarse y operarse en $\overline{\mathrm{FRP}}$. Todo conjunto finito de naturales se puede poner en correspondencia uno-a-uno con $\mathbb{N}$ utilizando la representación binaria de un número (i.e. $\cdots b_3 b_2 b_1 b_0$ representa un conjunto C en donde $n \in C$ si y sólo si $b_n = 1$; en particular el cero representa al conjunto vacío) y también se pueden construir números enteros y racionales, y estructuras más complejas como los árboles.
Las estructuras se pueden, además, comparar usando la delta de Kronecker siempre que tengan correspondencia uno-a-uno con los números naturales. Cuando no suceda esto, se puede utilizar una *normalización*. Por ejemplo, supongamos tener dos resultados booleanos b y b'. Como estos valores se representan con números naturales, podemos suponer que $b = 3$ y $b' = 5$. Según la convención que elegimos, b y b' representan a *true*. Sin embargo,

$\delta(b, b')$ es cero. Para que δ pueda comparar satisfactoriamente dos booleanos utilizamos la normalización

$$\begin{aligned} 0 &\to 0, \\ n &\to 1, \quad n > 0. \end{aligned}$$

La función $\phi[\mathsf{D}, \mathsf{D}]$ es útil para este caso. Ahora, $\delta(\mathsf{D}(\mathsf{D}(b)), \mathsf{D}(\mathsf{D}(b'))) = 1$ siempre que b y b' sean *true*.

En resumen, las funciones recursivas primitivas permiten operar con una amplia variedad de estructuras de datos. Supongamos que $\mathscr{L}$ sea un lenguaje imperativo que manipule las estructuras vistas y además contenga condicionales (construcción *if-then-else*) y bucles (*do-loop* o *while*) acotados por alguna constante o variable (o función recursiva primitiva[4]). Entonces, aunque no lo desarrollaremos aquí, es posible probar que $\mathscr{L} \doteq \overline{\mathrm{FRP}}$.
Un ejemplo de lenguaje imperativo que además es recursivo primitivo es el introducido por Hofstadter[Hof03] con el nombre de *BuD*. El utilizó este lenguaje para probar, de un modo más didáctico, que $\overline{\mathrm{FRP}}$ tiene menor poder computacional que $\overline{\mathrm{FRG}}$.

2.3. Reducción de algunos esquemas de recursión

A continuación, enumeraremos algunos esquemas de recursión reducibles a recursión primitiva.
Esquema 1. Consideremos la siguiente recursión, investigada inicialmente por R. Péter (se puede consultar dos desarrollos distintos en p. 271 [Kle52] o en la sección 6.1 del libro de Goodstein[Goo57]; nosotros seguiremos el primero), y conocida como *Recursion with parameter substitution*:

$$\mathscr{E}_1 \begin{cases} \varphi(x, 0) & = x, \\ \varphi(x, y+1) & = \varphi(\gamma(x, y), y), \end{cases}$$

donde $\gamma \in \overline{\mathrm{FRP}}$.
Claramente, el esquema anterior no es una recursión primitiva. Observemos que $\varphi(x, y)$ arroja la secuencia

$$\begin{aligned} &\varphi(x, 0) = x, \\ &\varphi(x, 1) = \varphi(\gamma(x, 0), 0) = \gamma(x, 0), \\ &\varphi(x, 2) = \varphi(\gamma(x, 1), 1) = \gamma(\gamma(x, 1), 0), \\ &\varphi(x, 3) = \varphi(\gamma(x, 2), 2) = \gamma(\gamma(\gamma(x, 2), 1), 0), \\ &\qquad \ldots \end{aligned}$$

Sin embargo, es recursiva primitiva la funcion

$$\begin{aligned} \varphi'(x, y, 0) &= x, \\ \varphi'(x, y, z+1) &= \gamma(\varphi'(x, y, z), y \dot{-} (z+1)), \end{aligned}$$

y se puede comprobar fácilmente por inducción sobre y que $\varphi(x, y) = \varphi'(x, y, y)$. Así que

$$\varphi \equiv \phi[\rho[\pi_1^2, \phi[\gamma, \pi_4^4, \phi[\mathsf{Mon}, \pi_2^4, \phi[\sigma, \pi_3^4]]]], \pi_1^2, \pi_2^2, \pi_2^2].$$

[4]El hecho de que los bucles deban estar acotados se estudia en [MR67]. Distintas caracterizaciones de los bucles acotados se encuentran en [GM88] y [Maz05].

Esquema 2. Esta es la recursión más importante, conocida como *Course-of-values recursion* (véase p. 231 [Kle52] o p. 119 [Goo57]). Se trata de que cualquier recursión de la forma

$$\mathscr{E}_2 \begin{cases} \varphi(x,0) & = A(x), \\ \varphi(x,y+1) & = B(x,y,\varphi(x,f_1(x,y)),\varphi(x,f_2(x,y)),\ldots,\varphi(x,f_n(x,y))), \end{cases}$$

puede ser reducida a recursion primitiva, siempre que $f_i(x,y) \leq y$ para toda $i : 1,\ldots,n$. La idea de este esquema es que, al momento de evaluar $\varphi(x,y+1)$ tiene en cuenta todos los valores generados por la recursión (es decir $\varphi(x,0)$, $\varphi(x,1)$, ..., $\varphi(x,y)$) mientras que en el esquema de recursión primitiva sólo tiene en cuenta el último valor generado (justamente $\varphi(x,y)$). Sin embargo, es posible reducir este esquema a recursión primitiva, almacenando en el último valor generado, una lista de todos los valores generados por la recursión.
Sea φ' una función que retorna la lista $[\varphi(x,y),\varphi(x,y-1),\ldots,\varphi(x,0)]$. Esta función se puede definir como

$$\begin{aligned} \varphi'(x,0) &= cons(A(x),nil()), \\ \varphi'(x,y+1) &= cons(B(x,y,index(\varphi'(x,y),f_1(x,y),y),index(\varphi'(x,y),f_2(x,y),y), \\ &\quad \ldots,index(\varphi'(x,y),f_n(x,y),y)),\varphi'(x,y)), \end{aligned}$$

donde $index(\varphi'(x,y),i,y) = \varphi(x,i)$ (recordar que $i \leq y$). Para lograr esto, definimos

$$\begin{aligned} extract(l,0) &= l, \\ extract(l,i+1) &= tail(extract(l,i)), \end{aligned}$$

e $index(l,i,y) = head(extract(l, y \dot{-} i))$. Luego, $\varphi(x,y) = head(\varphi'(x,y))$.
Utilizando esta estrategia, vamos a escribir las funciones que nos quedaron pendientes en la sección anterior:

$$\begin{aligned} concat &\equiv \phi[\phi[head,\rho[\phi[cons,\pi_1^1,nil_1],\phi[cons,\phi[\sigma,\phi[\mathsf{J},\phi[\mathsf{K},\pi_2^3], \\ &\quad \phi[index,\pi_3^3,\phi[\mathsf{L},\pi_2^3],\pi_1^3]]],\pi_3^3]]],\pi_2^2,\pi_1^2]. \\ reverse &\equiv \phi[head,\rho[\phi[cons,\varsigma_0^0,nil_0],\phi[cons,\phi[concat,\phi[index,\pi_2^2,\phi[\mathsf{L},\pi_1^2],\pi_1^2], \\ &\quad \phi[\sigma,\phi[\mathsf{J},\phi[\mathsf{K},\pi_1^2],\varsigma_0^2]]],\pi_2^2]]]. \\ length &\equiv \phi[head,\rho[\phi[cons,\varsigma_0^0,nil_0],\phi[cons,\phi[\sigma,\phi[index,\pi_2^2,\phi[\mathsf{L},\pi_1^2],\pi_1^2]],\pi_2^2]]]. \\ index &\equiv \phi[head,\phi[\rho[\pi_1^1,\phi[tail,\pi_3^3]],\pi_1^3,\phi[\mathsf{Mon},\pi_3^3,\pi_2^3]]]. \end{aligned}$$

Esquema 3. Un caso particular del esquema anterior es el llamado *Recursion from a double basis* que permite hallar varias sucesiones (una de ellas, la de Fibonacci).

$$\mathscr{E}_3 \begin{cases} \varphi(x,0) & = A(x), \\ \varphi(x,1) & = B(x), \\ \varphi(x,y+2) & = C(x,y,\varphi(x,y),\varphi(x,y+1)). \end{cases}$$

Una implementación sencilla es recordar los valores de la iteración precedente con un par ordenado (el par está constituido por $\varphi(x,y+1)$ y $\varphi(x,y)$). La siguiente función es recursiva primitiva y genera dichos pares:

$$\begin{aligned} \varphi'(x,0) &= \mathsf{J}(B(x),A(x)), \\ \varphi'(x,y+1) &= \mathsf{J}(C(x,y,\mathsf{L}(\varphi'(x,y)),\mathsf{K}(\varphi'(x,y))),\mathsf{K}(\varphi'(x,y))). \end{aligned}$$

Ahora redefinimos φ con otra recursión primitiva:

$$\begin{aligned}\varphi(x,0) &= A(x),\\ \varphi(x,y+1) &= \mathsf{K}(\varphi'(x,y)).\end{aligned}$$

En consecuencia,

$$\varphi \equiv \rho[A, \phi[\mathsf{K}, \phi[\rho[\phi[\mathsf{J}, B, A], \phi[\mathsf{J}, \phi[C, \pi^3_1, \pi^3_2, \phi[\mathsf{L}, \pi^3_3], \phi[\mathsf{K}, \pi^3_3]], \phi[\mathsf{K}, \pi^3_3]]], \pi^3_1, \pi^3_2]]].$$

Esquema 4. Otro esquema útil es el *Simultaneous recursion* y se define como sigue.

$$\mathscr{E}_4 \begin{cases}\varphi(x,0) & = A(x),\\ \varphi'(x,0) & = B(x),\\ \varphi(x,y+1) & = C(x,y,\varphi(x,y),\varphi'(x,y)),\\ \varphi'(x,y+1) & = D(x,y,\varphi(x,y),\varphi'(x,y)).\end{cases}$$

Vamos a ver que tanto φ como φ' son FRPs. Para lograrlo, introduciremos una función auxiliar que retornará el par $(\varphi(x,y), \varphi'(x,y))$,

$$\begin{aligned}\varphi''(x,0) &= \mathsf{J}(A(x), B(x)),\\ \varphi''(x,y+1) &= \mathsf{J}(C(x,y,\mathsf{K}(\varphi''(x,y)),\mathsf{L}(\varphi''(x,y))), D(x,y,\mathsf{K}(\varphi''(x,y)),\mathsf{L}(\varphi''(x,y)))).\end{aligned}$$

Luego, resulta $\varphi(x,y) = \mathsf{K}(\varphi''(x,y))$ y $\varphi'(x,y) = \mathsf{L}(\varphi''(x,y))$. La definición formal de φ'' es

$$\varphi'' \equiv \rho[\phi[\mathsf{J}, A, B], \phi[\mathsf{J}, \phi[C, \pi^3_1, \pi^3_2, \phi[\mathsf{K}, \pi^3_3], \phi[\mathsf{L}, \pi^3_3]], \phi[D, \pi^3_1, \pi^3_2, \phi[\mathsf{K}, \pi^3_3], \phi[\mathsf{L}, \pi^3_3]]]].$$

Los esquemas vistos aplican una metodología particular: todos ellos utilizan funciones de parificación. Cuando un procedimiento involucra alguna conversión en la entrada de datos (o en su procesamiento) que requieran funciones de parificación, se le suele llamar *estrategia de parificación*. Veremos más estrategias de parificación en el capítulo siguiente.

Para finalizar la sección, investigaremos una función (relativa a listas) muy conocida en el ámbito de la programación funcional. Sea ℓ una lista de naturales, e y q números naturales, y $\otimes$ una función binaria entre naturales (i.e. $arity\ \otimes\ = 2$). La función *foldr* se define recursivamente, según p. 102 [Bir00], como

$foldr_{e;\otimes} : List(\mathbb{N}) \to \mathbb{N}$
$foldr_{e;\otimes}\ nil = e$
$foldr_{e;\otimes}\ cons(q,\ell) = q \otimes (foldr_{e;\otimes}\ \ell)$

No obstante, sería útil que *foldr* aceptara una entrada adicional para permitir la parametrización de e y $\otimes$. Nosotros usaremos la siguiente versión modificada:

$foldr_{e;\otimes} : \mathbb{N} \times List(\mathbb{N}) \to \mathbb{N}$
$foldr_{e;\otimes}\ x\ nil = e(x)$
$foldr_{e;\otimes}\ x\ cons(q,\ell) = \otimes(x, q, foldr_{e;\otimes}\ x\ \ell)$

Caracterizando a la lista de entrada como un número natural, probaremos que *foldr* es recursiva primitiva (de manera similar a cómo lo hicimos con $length$[5]). Lo que sigue es una Course-of-value recursion.

$$foldr_{e;\otimes} \begin{cases} foldr_{e;\otimes}(x,y) & = e(x),\\ foldr_{e;\otimes}(x,y+1) & = \otimes(x, \mathsf{K}(y), foldr_{e;\otimes}(x, \mathsf{L}(y))).\end{cases}$$

[5]De hecho, *length* es un caso particular ($length \equiv \phi[foldr_{\varsigma^1_0;\phi[\sigma,\pi^3_1]}, \pi^1_1, \pi^1_1]$). *concat* y *reverse* también se pueden escribir en términos de un *foldr*.

Concluimos que

$$foldr_{e;\otimes} \equiv \phi[head, \rho[\phi[cons, e, nil_1], \phi[cons, \phi[\otimes, \pi^3_1, \phi[\mathsf{K}, \pi^3_2], \phi[index, \pi^3_3, \phi[\mathsf{L}, \pi^3_2], \pi^3_2]], \pi^3_3]]].$$

2.4. Reducción de $\overline{\mathrm{FDR}}$ a $\overline{\mathrm{FRG}}$

El propósito de esta sección es mostrar que las funciones doblemente recursivas pueden ser representadas mediante funciones recursivas generales. En otras palabras, probaremos que existe un algoritmo $\mathfrak{P} : \overline{\mathrm{FDR}} \to \overline{\mathrm{FRG}}$ apropiado.
Si $\mathfrak{D}$ es una identidad, entonces

$\mathfrak{P} : \overline{\mathrm{FDR}} \to \overline{\mathrm{FRG}}$
$\mathfrak{P}\ \zeta = \zeta$
$\mathfrak{P}\ \sigma = \sigma$
$\mathfrak{P}\ \pi^n_i = \pi^n_i$
$\mathfrak{P}\ \phi[\varphi, \omega_1, \omega_2, \ldots, \omega_n] = \phi[\mathfrak{P}\ \varphi, \mathfrak{P}\ \omega_1, \mathfrak{P}\ \omega_2, \ldots, \mathfrak{P}\ \omega_n]$
$\mathfrak{P}\ \rho[\beta, \varphi] = \rho[\mathfrak{P}\ \beta, \mathfrak{P}\ \varphi]$

Sólo falta representar el esquema de doble recursión (el hecho de que el recuadro no esté cerrado en la parte inferior fue realizado a propósito para recalcar que $\mathfrak{P}$ no ha sido definida completamente; el recuadro se cerrará cuando representemos el esquema de doble recursión). Recordemos que, si $\varphi \equiv \delta[\beta_1, \beta_2, \beta_3, \beta_4, \beta_5]$ entonces

$$\begin{aligned}
\varphi(\mathbf{x}, 0, z) &= \beta_1(\mathbf{x}, z),\\
\varphi(\mathbf{x}, y+1, 0) &= \beta_2(\mathbf{x}, y, \varphi(\mathbf{x}, y, \beta_3(\mathbf{x}, y))),\\
\varphi(\mathbf{x}, y+1, z+1) &= \beta_4(\mathbf{x}, y, z, \varphi(\mathbf{x}, y+1, z), \varphi(\mathbf{x}, y, \beta_5(\mathbf{x}, y, z, \varphi(\mathbf{x}, y+1, z)))),
\end{aligned}$$

donde $\mathbf{x} \in \mathbb{N}^n$ (en el caso que $n = 0$, este vector desaparece de la definición). Para poder llevar a cabo esta tarea, será necesario introducir dos estructuras de datos adicionales. Una de ellas nos permitirá representar tuplas de números naturales, y la otra funciones naturales finitas.

Tupla de números naturales. Sea $n \geq 2$. La construcción

$$\langle \_, \_, \cdots, \_ \rangle : \overline{\mathrm{FRP}}^n \to \overline{\mathrm{FRP}},$$

junto con sus proyecciones (donde i varía entre 1 y n)

$$\tau^n_i : \mathbb{N} \to \mathbb{N},$$

verifican $\tau^n_i \langle \pi^n_1, \pi^n_2, \ldots, \pi^n_n \rangle (x_1, x_2, \ldots, x_n) = x_i$. Estas funciones resultan recursivas primitivas, pues pueden implementarse con

$$\begin{aligned}
\langle F_1, F_2, \ldots, F_n \rangle &\equiv \phi[\mathsf{J}, F_1, \phi[\mathsf{J}, F_2, \ldots \phi[\mathsf{J}, F_{n-1}, F_n] \ldots]],\\
\tau^n_1 &\equiv \mathsf{K},\\
\tau^n_2 &\equiv \phi[\mathsf{K}, \mathsf{L}],\\
\tau^n_3 &\equiv \phi[\mathsf{K}, \phi[\mathsf{L}, \mathsf{L}]],\\
&\cdots\\
\tau^n_{n-1} &\equiv \phi[\mathsf{K}, \phi[\mathsf{L}, \ldots \phi[\mathsf{L}, \mathsf{L}] \ldots]],\\
\tau^n_n &\equiv \phi[\mathsf{L}, \phi[\mathsf{L}, \ldots \phi[\mathsf{L}, \mathsf{L}] \ldots]],
\end{aligned}$$

donde F_1, F_2, ... F_n son FRPs de la misma aridad, y L aparece $n-2$ y $n-1$ veces respectivamente en las dos últimas fórmulas. Estas funciones son una generalización de las funciones de parificación; hacen corresponder biunívocamente a cada tupla $(x_1, x_2, \ldots, x_n)$ con un número natural. La demostración se puede realizar por inducción sobre n, y usando (2.1.1) y (2.1.2).

Funciones naturales finitas. Sea $f : \mathbb{N} \to \mathbb{N}$ una función en donde una cantidad finita de valores x verifican $f(x) \neq 0$ (naturalmente, para el resto de los valores la función se anula). Llamaremos a f una función natural finita.
Representaremos a f mediante una lista de pares ordenados, $[(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)]$, donde los valores x_j no necesariamente son distintos entre sí (aquí $1 \leq j \leq m$). Supongamos que deseamos evaluar $f(x)$. Si $x \neq x_j$ para todo j, entonces convenimos que $f(x) = 0$. Si existe un único j para el cual $x = x_j$, entonces $f(x) = y_j$. Y, en el caso que existan varios x_j iguales a x, o más precisamente, que exista un conjunto de índices I tal que $x = x_k$ para todo $k \in I$, se emplea la fórmula

$$f(x) = \sum_{k \in I} y_k.$$

Con esta correspondencia se puede representar cualquier función natural finita. Sin embargo, no hay una biyección entre el conjunto de funciones naturales finitas y el conjunto de listas de pares ordenados. Por ejemplo, una función f que se anula en todo su dominio salvo para el valor 5, en cuyo caso devuelve 3, puede ser representada por las siguientes listas:

$[(5, 3)]$,
$[(5, 1), (5, 2)]$,
$[(5, 1), (5, 0), (5, 1), (5, 1)]$,
$[(5, 0), (38, 0), (5, 3), (47, 0)]$,
...

Sea ℓ la lista que representa una función f. Definimos la función de evaluación *eval* como

$$eval(x, \ell) = f(x).$$

Esta función es recursiva primitiva, pues puede implementarse como

$eval : \mathbb{N} \times List(\mathbb{N} \times \mathbb{N}) \to \mathbb{N}$
$eval\ x\ nil = 0$
$eval\ x\ cons((xj, yj), \ell) = \mathsf{D}(\mathsf{Dist}(x, xj)).yj + eval\ x\ \ell$

$$eval \equiv foldr_{\varsigma_0^1;\phi[\Sigma,\phi[\Pi,\phi[\mathsf{D},\phi[\mathsf{Dist},\pi_1^3,\phi[\mathsf{K},\pi_2^3]]]],\phi[\mathsf{L},\pi_2^3]],\pi_3^3]}.$$

En la definición recursiva de *eval*, se comienza con un valor inicial cero y se van sumando términos con el valor yj sólo cuando $x = xj$.
Otra función que será de utilidad es

$$dom(\ell) = [x_1, x_2, \ldots, x_m],$$

que se implementa como

$dom : List(\mathbb{N} \times \mathbb{N}) \to List(\mathbb{N})$
$dom\ nil = nil$
$dom\ cons((xj, yj), \ell) = cons(xj, dom\ \ell)$

$$dom \equiv \phi[foldr_{nil_1;\phi[cons,\pi_2^3,\pi_3^3]}, \pi_1^1, \pi_1^1].$$

En la definición recursiva de dom, se comienza con una lista vacía y se van agregando los valores xj para cada par que se consume en la entrada.
Finalmente, otra función que usaremos es

$$find(x, \ell) = \begin{cases} 0 & \text{si } x \text{ aparece en } dom(\ell), \\ 1 & \text{si no,} \end{cases}$$

y se implementa como

$find : \mathbb{N} \times List(\mathbb{N} \times \mathbb{N}) \to \mathbb{N}$
$find\ x\ nil = 1$ $find\ x\ cons((xj, yj), \ell) = \mathsf{D}(\mathsf{D}(\mathsf{Dist}(x, xj).find\ x\ \ell))$

$$find \equiv foldr_{\varsigma_0^1;\phi[\phi[\mathsf{D},\mathsf{D}],\phi[\Pi,\phi[\mathsf{Dist},\pi_1^3,\phi[\mathsf{K},\pi_2^3]],\pi_3^3]]}.$$

En la definición recursiva de $find$, se comienza con un valor inicial uno, que se mantiene mientras $x \neq xj$. Si $x = xj$, el resultado es cero.

Podemos mezclar ambas estructuras de datos para representar funciones finitas de la forma $f : \mathbb{N}^n \to \mathbb{N}$, las cuales quedan identificadas con la lista

$$[(\langle x_{11}, x_{12}, \ldots, x_{1n}\rangle, y_1), (\langle x_{21}, x_{22}, \ldots, x_{2n}\rangle, y_2), \ldots, (\langle x_{m1}, x_{m2}, \ldots, x_{mn}\rangle, y_m)],$$

la cual tiene asociada un número particular. De esta manera, a cada número natural le corresponde una función finita (no necesariamente distinta). Lamentablemente, es imposible representar cualquier función arbitraria con un número natural, por una cuestión de cardinalidad. No obstante, la siguiente definición nos permitirá trabajar con funciones arbitrarias:

Definición 2.4.A. *Sea g una función natural y ℓ un número natural que representa una lista que, a su vez, representa una función natural finita f. Diremos que ℓ* representa una parte finita de *g cuando $f(x) = g(x)$ para todo x contenido en $dom(\ell)$.*

Nuestro problema principal es calcular $\varphi(\mathbf{x}, y, z)$. Si bien, no es posible representar a φ con un número natural, sí podemos hallar un número ℓ que represente una parte finita de φ tal que $(\mathbf{x}, y, z)$ esté en $dom(\ell)$. El ejemplo que presentaremos puede aclarar este nuevo concepto de **representar una parte finita de φ con un número**.

Ejemplo 2.4.B. *Representaremos una parte finita de la función de Ackermann con un número natural. En primer lugar, observemos que*

$$\begin{aligned} &\mathsf{A}(0,1) = \sigma(1) = 2, \\ &\mathsf{A}(0,2) = \sigma(2) = 3, \\ &\mathsf{A}(0,3) = \sigma(3) = 4, \\ &\mathsf{A}(1,0) = \mathsf{A}(0,1) = 2, \\ &\mathsf{A}(1,1) = \mathsf{A}(0,\mathsf{A}(1,0)) = \mathsf{A}(0,2) = 3, \\ &\mathsf{A}(1,2) = \mathsf{A}(0,\mathsf{A}(1,1)) = \mathsf{A}(0,3) = 4. \end{aligned}$$

Sea $[(\langle 0,1\rangle, 2), (\langle 0,2\rangle, 3), (\langle 0,3\rangle, 4), (\langle 1,0\rangle, 2), (\langle 1,1\rangle, 3), (\langle 1,2\rangle, 4)]$ la lista que representa a A. Dicha lista es representada por un número ℓ. Para hallar $\mathsf{A}(1,2)$ simplemente calculamos $eval(\langle 1,2\rangle, \ell)$. Además, la lista dada contiene todos los valores de A necesarios para calcular $\mathsf{A}(1,2)$ (más precisamente, la mínima cantidad de valores necesarios). Con esto último queremos decir que los elementos de la lista satisfacen las siguientes reglas:

$\star\ x = 0 \implies eval(\langle x, y\rangle, \ell) = y + 1,$

$\star\ x > 0 \wedge y = 0 \implies eval(\langle x, y\rangle, \ell) = eval(\langle x - 1, 1\rangle, \ell),$

$\star\ x > 0 \wedge y > 0 \implies eval(\langle x, y\rangle, \ell) = eval(\langle x - 1, eval(\langle x, y - 1\rangle, \ell)\rangle, \ell).$

Ellas afirman que ℓ representa realmente una parte de A*, pues son una consecuencia directa de las reglas de reescritura de la función de Ackermann.*
Finalmente, el número ℓ se puede obtener como sigue:

$$\begin{aligned}
&\ell = [(\langle 0,1\rangle, 2), (\langle 0,2\rangle, 3), (\langle 0,3\rangle, 4), (\langle 1,0\rangle, 2), (\langle 1,1\rangle, 3), (\langle 1,2\rangle, 4)] = \\
&[(\mathsf{J}(0,1), 2), (\mathsf{J}(0,2), 3), (\mathsf{J}(0,3), 4), (\mathsf{J}(1,0), 2), (\mathsf{J}(1,1), 3), (\mathsf{J}(1,2), 4)] = \\
&[\mathsf{J}(\mathsf{J}(0,1), 2), \mathsf{J}(\mathsf{J}(0,2), 3), \mathsf{J}(\mathsf{J}(0,3), 4), \mathsf{J}(\mathsf{J}(1,0), 2), \mathsf{J}(\mathsf{J}(1,1), 3), \mathsf{J}(\mathsf{J}(1,2), 4)] = \\
&cons(\mathsf{J}(\mathsf{J}(1,2), 4), cons(\mathsf{J}(\mathsf{J}(1,1), 3), cons(\mathsf{J}(\mathsf{J}(1,0), 2), \\
&cons(\mathsf{J}(\mathsf{J}(0,3), 4), cons(\mathsf{J}(\mathsf{J}(0,2), 3), cons(\mathsf{J}(\mathsf{J}(0,1), 2), nil)))))) = \\
&\mathsf{J}(\mathsf{J}(\mathsf{J}(1,2), 4) + 1, \mathsf{J}(\mathsf{J}(\mathsf{J}(1,1), 3) + 1, \mathsf{J}(\mathsf{J}(\mathsf{J}(1,0), 2) + 1, \\
&\mathsf{J}(\mathsf{J}(\mathsf{J}(0,3), 4) + 1, \mathsf{J}(\mathsf{J}(\mathsf{J}(0,2), 3) + 1, \mathsf{J}(\mathsf{J}(\mathsf{J}(0,1), 2) + 1, 0)))))) = \\
&722136133944426599720787838580500415045809752718 04.
\end{aligned}$$

También es posible calcular la función de Ackermann mediante una minimización. Para el caso de $\mathsf{A}(1,2)$, sea $w' = \mathsf{J}(\langle 1,2\rangle, \ell)$ y supongamos contar con las funciones

$$\alpha(w) = eval(\mathsf{K}(w), \mathsf{L}(w)),$$

$$\gamma(x,y,w) = \begin{cases} 0 & \text{si } \mathsf{K}(w) = \langle x, y\rangle \text{ y } \mathsf{L}(w) \text{ representa una parte de } \mathsf{A} \\ & \text{y además existe, al menos, un par } (\langle x, y\rangle, \_) \text{ en } \mathsf{L}(w), \\ \neq 0 & \text{si no.} \end{cases}$$

Claramente, $\alpha(w') = \mathsf{A}(1,2)$ y $\gamma(1,2,w') = 0$. O, más generalmente, $\alpha(w) = \mathsf{A}(x,y)$ y $\gamma(x,y,w) = 0$, para algún $w \in \mathbb{N}$. Aquí es donde entra en juego el minimizador: podemos hacer una búsqueda del valor w que satisfaga $\gamma(x,y,w) = 0$. Resulta luego, $\mathsf{A} \equiv \phi[\alpha, \mu\gamma]$. La prueba de que el minimizador devuelve un valor natural (y no $\perp$) se debe a que A es total.

A continuación, introduciremos más funciones recursivas primitivas ($\alpha, \gamma, valid, computer$) a fin de poder alcanzar nuestro objetivo inicial: el cómputo de φ. Buscaremos funciones α y γ tales que

$$\phi[\alpha, \mu\gamma](x_1, x_2, \ldots, x_n, y, z) = \varphi(x_1, x_2, \ldots, x_n, y, z),$$

similares a las que propusimos para el caso de A. De hecho, $\alpha(w) = eval(\mathsf{K}(w), \mathsf{L}(w))$ pues considera un valor w que representa un par (t, ℓ), donde t representa la tupla de entrada para la función φ y ℓ representa una parte finita de φ. Por otra parte, la función γ se debe anular (i.e. $\gamma(x_1, x_2, \ldots, x_n, y, z, w) = 0$) cuando se satisfacen simultáneamente las igualdades $\mathsf{K}(w) = \langle x_1, x_2, \ldots, x_n, y, z\rangle$, $valid(\mathsf{L}(w), dom(\mathsf{L}(w))) = 0$ y $find(\mathsf{K}(w), \mathsf{L}(w)) = 0$. Es decir,

$$\gamma(\mathbf{x}, y, z, w) = \mathsf{Dist}(\mathsf{K}(w), \langle \mathbf{x}, y, z\rangle) + valid(\mathsf{L}(w), dom(\mathsf{L}(w))) + find(\mathsf{K}(w), \mathsf{L}(w)).$$

Formalmente,

$$\begin{aligned}
&\alpha \equiv \phi[eval, \mathsf{K}, \mathsf{L}], \\
&\gamma \equiv \phi[\Sigma, \phi[\mathsf{Dist}, \phi[\mathsf{K}, \pi^{n+3}_{n+3}], \langle \pi^{n+3}_1, \pi^{n+3}_2, \ldots, \pi^{n+3}_{n+2}\rangle], \\
&\quad \phi[\phi[\Sigma, \phi[\phi[valid, \pi^1_1, dom], \mathsf{L}], \phi[find, \mathsf{K}, \mathsf{L}]], \pi^{n+3}_{n+3}]].
\end{aligned}$$

La función $valid(\ell, d)$, la cual es la encargada de afirmar si ℓ representa realmente una parte finita de φ, retorna cero sólo si todas las tuplas (de la forma $\langle \mathbf{x}, y, z\rangle$) que aparecen en la lista d satisfacen las reglas que generan a φ. Esto es que

$$eval(\langle \mathbf{x}, y, z\rangle, \ell) = compute(\langle \mathbf{x}, y, z\rangle, \ell),$$

donde *compute* se define como

$$compute(\langle \mathbf{x}, y, z\rangle, \ell) = \begin{cases} \beta_1(\mathbf{x}, z) & \text{si } y = 0, \\ \beta_2(\mathbf{x}, y-1, eval(\langle \mathbf{x}, y-1, \beta_3(\mathbf{x}, y-1)\rangle, \ell)) & \text{si } y > 0 \text{ y } z = 0, \\ \beta_4(\mathbf{x}, y-1, z-1, eval(\langle \mathbf{x}, y, z-1\rangle, \ell), & \\ \quad eval(\langle \mathbf{x}, y-1, \beta_5(\mathbf{x}, y-1, z-1, & \\ \quad\quad eval(\langle \mathbf{x}, y, z-1\rangle, \ell))\rangle, \ell)) & \text{si } y > 0 \text{ y } z > 0. \end{cases}$$

La versión algorítmica de *valid* es

$valid : List(\mathbb{N} \times \mathbb{N}) \times List(\mathbb{N}) \to \mathbb{N}$
$valid\ \ell\ nil = 0$
$valid\ \ell\ cons(t, d) = \mathsf{Dist}(eval(t, \ell), compute(t, \ell)) + valid\ \ell\ d$

mientras que *compute* es de la forma

$$\begin{aligned} compute(t, \ell) = {} & \mathsf{D}(\tau_{n+1}^{n+2}(t)).A(t, \ell) \\ & + \mathsf{D}(\mathsf{D}(\tau_{n+1}^{n+2}(t)) + \tau_{n+2}^{n+2}(t)).B(t, \ell) \\ & + \mathsf{D}(\mathsf{D}(\tau_{n+1}^{n+2}(t).\tau_{n+2}^{n+2}(t))).C(t, \ell), \end{aligned}$$

donde A, B y C son funciones que calculan *compute* para cada caso particular. Por ejemplo, $A(\langle \mathbf{x}, y, z\rangle, \ell) = \beta_1(\mathbf{x}, z)$. Para B y C se procede análogamente.
Las funciones *valid* y *compute* son recursivas primitivas[6],

$$\begin{aligned} valid &\equiv foldr_{\varsigma_0^1;\phi[\Sigma,\phi[\phi[\mathsf{Dist},eval,compute],\pi_2^3,\pi_1^3],\pi_3^3]}, \\ compute &\equiv \phi[\Sigma, \phi[\Pi, \phi[\mathsf{D}, \phi[\tau_{n+1}^{n+2}, \pi_1^2]], A], \ldots], \\ A &\equiv \phi[\beta_1, \langle \tau_1^{n+2}, \tau_2^{n+2}, \ldots, \tau_n^{n+2}, \tau_{n+2}^{n+2}\rangle]. \end{aligned}$$

Hay que tener presente que la función *compute* es, en realidad, una construcción parametrizada por el natural n y las funciones β_1, β_2, β_3, β_4, y β_5. Por este motivo, *valid* y γ también son construcciones. Si consideramos que $\gamma_{n;\beta_1;\beta_2;\beta_3;\beta_4;\beta_5}$ es un algoritmo que genera la función γ adecuada, entonces

$$\mathfrak{P}\ \delta[\beta_1, \beta_2, \beta_3, \beta_4, \beta_5] = \phi[\alpha, \mu\gamma_{arity\ \beta_1 - 1;\mathfrak{P}\ \beta_1;\mathfrak{P}\ \beta_2;\mathfrak{P}\ \beta_3;\mathfrak{P}\ \beta_4;\mathfrak{P}\ \beta_5}]$$

Ahora, podemos asegurar la validez del siguiente teorema.

Teorema 2.4.C. *Las funciones doblemente recursivas pueden ser representadas con funciones recursivas generales. Más específicamente, si $\varphi \in \overline{\mathrm{FDR}}$ entonces existe $\varphi' \in \overline{\mathrm{FRG}}$ tal que $\varphi(\mathbf{x}) = \varphi'(\mathbf{x})$ para todo $\mathbf{x} \in \mathbb{N}^n$. Además, en cada ocurrencia de una doble recursión en φ le corresponde una minimización en φ'.*

Se pueden extender los conceptos vistos para representar funciones triplemente recursivas (o de grado mayor) con FRGs. Incluso se puede probar que $\mathscr{L} \doteq \overline{\mathrm{FRG}}$ si $\mathscr{L}$ es un lenguaje de propósito general (i.e. que puede representar cualquier función computable, véase [GM91]).

[6]No escribiremos formalmente *compute* (ni sus componentes B y C) porque es larga y tediosa. Sin embargo, su fórmula puede deducirse mecánicamente a partir de su definición.

2.5. Notas con respecto a $\overline{\text{FRG}}$

En la sección previa, hemos demostrado que podíamos calcular una doble recursión con una minimización. Pero debíamos usar tantas minimizaciones como doble recursiones aparezcan en la fórmula. En realidad, basta con tener sólo una minimización.
En la sección 1.7, citamos el artículo [Kle36a] donde se afirma que cualquier función recursiva general φ (o mejor, cualquier función natural Turing-computable) puede calcularse con $\phi[\alpha, \mu\gamma]$. Nosotros probaremos a continuación que esta fórmula se puede reducir a la forma más simple $\phi[\mathsf{K}, \mu\psi]$ donde ψ es una FRP con la misma aridad que γ. Pero sólo tendrá validez si φ es total.

Sea $\varphi \equiv \phi[\alpha, \mu\gamma]$ una función total, donde $\alpha, \gamma \in \overline{\text{FRP}}$. Aplicaremos una estrategia de parificación para dar la fórmula alternativa $\phi[\mathsf{K}, \mu\psi]$ a φ. Para ello, propondremos

$$\psi(x_1, x_2, \ldots, x_n, y) = \begin{cases} 0 & \text{si } \gamma(x_1, x_2, \ldots, x_n, \mathsf{L}(y)) = 0 \text{ y } \mathsf{K}(y) = \alpha(\mathsf{L}(y)), \\ \neq 0 & \text{si no,} \end{cases}$$

$$\psi \equiv \phi[\Sigma, \phi[\gamma, \pi_1^{n+1}, \pi_2^{n+1}, \ldots, \pi_n^{n+1}, \phi[\mathsf{L}, \pi_{n+1}^{n+1}]], \phi[\mathsf{Dist}, \phi[\mathsf{K}, \pi_{n+1}^{n+1}], \phi[\alpha, \phi[\mathsf{L}, \pi_{n+1}^{n+1}]]]].$$

Haremos la demostración de esta fórmula para el caso en que $n = 1$, que no es menos general que para $n > 1$.
En principio, $\varphi(x) = \alpha\big(\min_{z\in\mathbb{N}}\{\gamma(x, z) = 0\}\big)$. Además, sabemos que para un cierto x, existe un valor para z que satisface $\gamma(x, z) = 0$ (por ser φ total). Luego, $\varphi(x) = \alpha(z)$. Si hacemos $y = \mathsf{J}(\varphi(x), z)$, resulta

$$\psi(x, y) = \gamma(x, \mathsf{L}(y)) + |\mathsf{K}(y) - \alpha(\mathsf{L}(y))| = \gamma(x, z) + |\varphi(x) - \alpha(z)| = 0.$$

Por lo tanto,

$$\mathsf{K}\left(\min_{y\in\mathbb{N}}\{\psi(x, y) = 0\}\right) = \mathsf{K}(y) = \varphi(x).$$

Sólo falta asegurar que *al mínimo valor de z le corresponde el mínimo valor de y*. Esto se verifica usando las propiedades de monotonía de J:
Dado $a \in \mathbb{N}$ fijo,

$$b < c \implies \mathsf{J}(a, b) < \mathsf{J}(a, c),$$
$$b < c \implies \mathsf{J}(b, a) < \mathsf{J}(c, a).$$

Sean z_1 y z_2 dos valores que satisfacen $\gamma(x, z) = 0$. Respectivamente, sean $y_1 = \mathsf{J}(\varphi(x), z_1)$ e $y_2 = \mathsf{J}(\varphi(x), z_2)$. Si $z_1 < z_2$, entonces $y_1 < y_2$.
Para demostrar las propiedades de monotonía de J, se pueden deducir primero que para todo x e y, $\mathsf{J}(x, y) < \mathsf{J}(x, y + 1)$ y $\mathsf{J}(x, y) < \mathsf{J}(x + 1, y)$ (para cada caso, se deberá utilizar una doble inducción).

Un comentario final acerca de las FRGs es que se puede prescindir de la recursión primitiva, si disponemos de ciertas funciones iniciales. Según Kleene[Kle36b], cualquier función recursiva general puede ser representada en términos de las proyecciones $\pi_i^n(x_1, x_2, \ldots, x_n) = x_i$, la función delta de Kronecker $\delta(x, y) = \mathsf{D}(\mathsf{Dist}(x, y))$, la suma $\Sigma(x, y) = x + y$, el producto

$\Pi(x,y) = xy$, la composición de funciones ϕ, y la minimización μ. Por ejemplo,

$$\begin{aligned}
\varsigma_1^1 &\equiv \phi[\delta, \pi_1^1, \pi_1^1], \quad \text{(función uno: } f(x) = \delta(x,x)) \\
\sigma &\equiv \phi[\Sigma, \pi_1^1, \varsigma_1^1], \quad \text{(función sucesor: } f(x) = x+1) \\
\varsigma_0^1 &\equiv \phi[\delta, \pi_1^1, \sigma], \quad \text{(función cero: } f(x) = \delta(x, x+1)) \\
\mathsf{D} &\equiv \phi[\delta, \pi_1^1, \varsigma_0^1], \quad \text{(función distintor: } f(x) = \delta(x,0)) \\
\mathsf{Hf} &\equiv \mu\phi[\phi[\mathsf{D}, \phi[\Sigma, \phi[\delta, \pi_1^2, \pi_2^2], \phi[\delta, \pi_1^2, \phi[\sigma, \pi_2^2]]]], \pi_1^2, \phi[\Sigma, \pi_2^2, \pi_2^2]]. \\
&\text{(función mitad: } f(x) = \min_{y \in \mathbb{N}}\{x = 2y \vee x = 2y+1\})
\end{aligned}$$

Capítulo 3

Reducción a funciones unarias

3.1. Las funciones unarias primitivas

Nuestro objetivo principal es estudiar métodos para simplificar los esquemas de recursión (recursión primitiva y minimización principalmente). Para ello nos basaremos en el artículo de R. Robinson[Rob47] acerca de las FRPs, y en el de J. Robinson[Rob50] acerca de las FRGs. Los métodos propuestos pueden servir para probar propiedades sobre las funciones recursivas (e.g. en [Rob48] se aplican estos métodos para probar que existe una función que no es recursiva primitiva).
A continuación vamos a exponer cómo se pueden construir las funciones unarias primitivas, las cuales resultarán tan expresivas como las FRPs.

DEFINICIÓN 3.1.A. *Las* funciones unarias primitivas *son funciones naturales (de la forma $F : \mathbb{N} \to \mathbb{N}$) formadas a partir de los constructores (y sólo de ellos) que enumeraremos a continuación.*
La función distintor, cuadrado y raíz cuadrada (que ya fueron definidas en el Capítulo 1),

$$\mathsf{D}(x) = 1 \dot{-} x, \quad \mathsf{Sq}(x) = x^2, \quad \mathsf{Rt}(x) = \lfloor \sqrt{x} \rfloor.$$

Dadas dos funciones unarias primitivas F y G se definen:

- ⋆ *Suma de F y G. Es una función que retorna $F(x) + G(x)$. La notamos como $(F + G)$.*
- ⋆ *Resta de F y G. Es una función que retorna $F(x) - G(x)$. La notamos como $(F - G)$. La resta que usaremos estará definida siempre que $F(x) \geq G(x)$ (y, de hecho, **sólo la usaremos cuando se cumpla esta condición**). Técnicamente $x - y$ dará el valor correcto si $x \geq y$, y cualquier natural arbitrario (incluso $\perp$) de otro modo*[1].
- ⋆ *Composición de F y G. Es una función que retorna $F(G(x))$. La notamos como (FG) (como si fuese el producto entre ambas).*
- ⋆ *Iteración de F. Es una función que retorna la composición de F consigo misma x veces. Esto es, para $x = 0$ se obtiene cero, para $x = 1$ se obtiene $F(0)$, para $x = 2$ se obtiene $F(F(0))$ y así. La notamos como $(F^{\square})$. Más formalmente,*

$$F^{\square}(x) = \begin{cases} 0 & si\ x = 0, \\ F(F^{\square}(x-1)) & si\ x > 0. \end{cases} \tag{3.1.1}$$

[1] Algunas operaciones que pueden substituir a la resta son Mon (resta truncada) y Dist (función distancia), ya que sus comportamientos coinciden cuando $x \geq y$.

Denotaremos al conjunto de las funciones unarias primitivas como $\overline{\mathrm{FUP}}$.

Una acotación importante sobre estas funciones es que, al igual que las FRPs, siempre son totales. También poseen la característica de conformar una estructura de monoide sobre la composición:
es cerrada (i.e. $F, G \in \overline{\mathrm{FUP}} \implies FG \in \overline{\mathrm{FUP}}$),
asociativa (i.e. $F, G, H \in \overline{\mathrm{FUP}} \implies F(GH) = (FG)H$),
y existe una función identidad X (i.e. $F \in \overline{\mathrm{FUP}} \implies F\mathsf{X} = \mathsf{X}F = F$;
pruébese con $\mathsf{X} \equiv \mathsf{RtSq}$).
Para no escribir paréntesis redundantes haremos algunas convenciones tipográficas:

⋆ Los paréntesis más externos se pueden eliminar,

$$(F(G + H)) \quad \dashleftarrow \quad F(G + H).$$

⋆ Asociaremos las operaciones a la izquierda,

$$\begin{aligned}(F + G) + H &\dashleftarrow F + G + H,\\ (F - G) - H &\dashleftarrow F - G - H,\\ (FG)H &\dashleftarrow FGH.\end{aligned}$$

⋆ La precedencia entre las operaciones es similar a la utilizada habitualmente en las expresiones aritméticas. La composición toma el lugar del producto y la iteración el lugar de la potencia (el órden sería: suma y resta - composición - iteración),

$$F + (G(H^{\square})) \dashleftarrow F + GH^{\square}.$$

Con estas indicaciones se puede definir, sin mayores inconvenientes, una gramática para las FUPs. Además no es necesario efectuar ninguna verificación semántica (como era el caso de las FRPs en donde hacíamos uso de la función *arity*). Recordemos que todas las funciones toman siempre un argumento.

Analicemos, ahora, la causa de la elección de los constructores que forman las FUPs. La primer función inicial es el distintor. Sin ésta no es posible generar funciones tales que $F(0) \neq 0$ como afirma el siguiente teorema.

Teorema 3.1.B. *Supongamos contar con la suma, resta, composición e iteración y un conjunto finito de funciones iniciales* F_i. *Si cada una de ellas pasa por el origen (i.e.* $F_i(0) = 0$*) entonces todas las funciones generadas heredan esta misma propiedad.*

Demostración. La prueba es por inducción sobre los constructores. Supongamos que F y G son funciones iniciales o generadas, y que $F(0) = G(0) = 0$. Luego $F(0) + G(0) = 0$, $F(0) - G(0) = 0$, $F(G(0)) = 0$ y $F^{\square}(0) = 0$ (esta última es por definición de la iteración). □

Dado que Rt y Sq pasan por el origen, queda claro que D es necesaria. Pero todavía nos preguntamos si podrían haber otras funciones que la sustituyan. Si contáramos con la función uno ($\mathsf{U}(x) = 1$) entonces D surge de

$$\mathsf{D} \equiv \mathsf{U} - \mathsf{U}^{\square}.$$

En efecto, sea $F \equiv \mathsf{U}^{\square}$. Por definición de iteración $F(0) = 0$ y $F(x+1) = \mathsf{U}(F(x)) = 1$. Luego $1 - F(x)$ resulta ser el distintor. Observemos que hemos tenido cuidado de que $F(x) \leq 1$ para poder formar la resta apropiadamente. Siempre que restemos, verificaremos que el sustraendo sea mayor que el minuendo.

Por lo tanto, U puede reemplazar a D. Sin embargo, existen infinitas funciones que pueden reemplazar, a su vez, a U. Sea F cualquier función tal que $F(0) > 0$. Luego, $F(G - G)$ es una función constante que retorna el valor $F(0)$ (donde G puede ser cualquier función total). Notemos que Rt tiene dos puntos fijos, a saber $\mathsf{Rt}^{\infty}(0) = 0$ y $\mathsf{Rt}^{\infty}(n) = 1$ cuando $n > 0$ (usamos Rt^k para indicar composición, no potencia). La idea es aplicar una cantidad finita de veces Rt sobre la constante para llevarla a uno. Por ejemplo, si $F(0) = 1000$ y optamos por $G \equiv \mathsf{Sq}$, tenemos $\mathsf{U} \equiv \mathsf{RtRtRtRt}F(\mathsf{Sq} - \mathsf{Sq})$.

Las otras dos funciones inciales impuestas tienen una característica particular: llamemos F a Sq y G a Rt respectivamente. Compuestas en un sentido construyen la función identidad (GF), pero en el otro sentido nos dan una función que halla *el mayor cuadrado perfecto menor o igual al parámetro elegido* (FG). Otros pares de funciones apropiadas podrían haber sido

$F(x) = x^3$ y $G(x) = \lfloor \sqrt[3]{x} \rfloor$,
$F(x) = 2^x$ y $G(x) = \lfloor \log_2 x \rfloor$,
$F(x) = \blacktriangle(x)$ y $G(x) = \blacktriangledown(x)$.

En ellos se explota una propiedad fundamental: la función FG crece de manera escalonada, en donde la longitud de cada escalón es estrictamente creciente (véase la sección 1 de [Maz97]). En el caso de Sq y Rt la longitud de los escalones está dada por la separación entre los cuadrados perfectos.

No obstante, se pueden elegir otras variantes de funciones iniciales. Por ejemplo, podemos cambiar la terna de funciones iniciales $\{\mathsf{D}, \mathsf{Sq}, \mathsf{Rt}\}$ por $\{\mathsf{S}, \mathsf{Rt}\}$ donde S es, simplemente, el sucesor. Para ello, es necesario representar D y Sq. Notemos, por un lado, que la identidad puede ser reemplazada por $\mathsf{S}^{\square}$ y, por el otro, que $\mathsf{Sq}(0) = 0$ y $\mathsf{Sq}(x+1) = \mathsf{Sq}(x) + 2x + 1 = \mathsf{S}(\mathsf{Sq}(x)) + 2\mathsf{Rt}(\mathsf{Sq}(x))$. Entonces,

$$\begin{aligned} \mathsf{D} &\equiv (\mathsf{S} - \mathsf{S}^{\square}) - (\mathsf{S} - \mathsf{S}^{\square})^{\square}, \\ \mathsf{Sq} &\equiv (\mathsf{S} + \mathsf{Rt} + \mathsf{Rt})^{\square}. \end{aligned}$$

También podríamos haber optado utilizar otras construcciones (en vez de la suma y la resta), o eliminar alguna de ellas. R. Robinson[Rob55b] demostró que es suficiente con poseer $\{\mathsf{S}, \mathsf{K}\}$, la construcción $\mathsf{J}(F(x), G(x))$, la composición y la iteración. Para ver que tienen el mismo poder de expresión que las FUPs basta con que puedan representar cada una de sus operaciones. Observando que $\mathsf{Dbl} \equiv (\mathsf{SS})^{\square}$ tenemos que

$$\begin{aligned} F - G &\equiv \mathsf{K\ Dbl\ Dbl\ J(SSS\ K\ Dbl\ Dbl\ J(SSS\ K\ Dbl\ Dbl\ J}(G, \mathsf{Dbl\ Dbl}F), F), \mathsf{Dbl}G), \\ F + G &\equiv \mathsf{J}(F, G) - ((\mathsf{J}(F, G) - F) - G), \\ \mathsf{D} &\equiv (\mathsf{S} - \mathsf{S}^{\square}) - (\mathsf{S} - \mathsf{S}^{\square})^{\square}, \\ \mathsf{Sq} &\equiv \mathsf{Dbl}(\mathsf{SS}(\mathsf{S}^{\square} + \mathsf{K}))^{\square} - (\mathsf{SSS})^{\square}, \\ \mathsf{Rt} &\equiv \mathsf{Dbl} - (\mathsf{S} + \mathsf{DDKS})^{\square}. \end{aligned}$$

En el capítulo siguiente veremos que es suficiente contar con solamente el sucesor, la construcción $\mathsf{Dist}(F(x), G(x))$, la composición y la iteración.

3.2. Ejemplos de funciones unarias primitivas

Los siguientes teoremas nos permitirán ver alguna de las operaciones que se pueden representar en términos de funciones unarias primitivas:

TEOREMA 3.2.A. *Las siguientes funciones son unarias primitivas.*

(a) **Función cero***:* $\mathsf{Z}(x) = 0$.

(b) **Función uno***:* $\mathsf{U}(x) = 1$.

(c) **Función identidad***:* $\mathsf{X}(x) = x$.

(d) **Función sucesor***:* $\mathsf{S}(x) = x + 1$.

(e) **Función doble***:* $\mathsf{Dbl}(x) = 2x$.

(f) **Función predecesor***:* $\mathsf{P}(x) = x \dot{-} 1$.

(g) **Función mitad***:* $\mathsf{Hf}(x) = \lfloor \frac{x}{2} \rfloor$.

Demostración. Las pruebas se enumeran a continuación, encerrando las fórmulas dentro de una caja:

(a) **Función cero**: Restando una función consigo misma produce la función nula.

$$\boxed{\mathsf{Z} \equiv \mathsf{D} - \mathsf{D}}$$

(b) **Función uno**: Notemos que $\mathsf{U}(x) = \mathsf{D}(x) + \mathsf{D}(\mathsf{D}(x))$ pues, si $x = 0$ entonces $\mathsf{D}(x) + \mathsf{D}(\mathsf{D}(x)) = 1 + 0 = 1$. Y si $x > 0$ entonces $\mathsf{D}(x) + \mathsf{D}(\mathsf{D}(x)) = 0 + 1 = 1$.

$$\boxed{\mathsf{U} \equiv \mathsf{D} + \mathsf{DD}}$$

(c) **Función identidad**: Aprovechamos que Sq y Rt son inversas en un sentido.

$$\boxed{\mathsf{X} \equiv \mathsf{RtSq}}$$

(d) **Función sucesor**.

$$\boxed{\mathsf{S} \equiv \mathsf{X} + \mathsf{U}}$$

(e) **Función doble**.

$$\boxed{\mathsf{Dbl} \equiv \mathsf{X} + \mathsf{X}}$$

(f) **Función predecesor**: Notemos que

$$\begin{aligned} \mathsf{P}(0) &= \mathsf{X}(0) - \mathsf{D}(\mathsf{D}(0)) = 0 - 0 = 0, \\ \mathsf{P}(x) &= \mathsf{X}(x) - \mathsf{D}(\mathsf{D}(x)) = x - 1, \text{ si } x > 0. \end{aligned}$$

$$\boxed{\mathsf{P} \equiv \mathsf{X} - \mathsf{DD}}$$

(g) **Función mitad**: Para el cálculo de esta función procederemos como R. Robinson[Rob47], definiendo previamente la función $\mathsf{Mod}_3(x) = x \bmod 3$, y probando que $\mathsf{Mod}_3 \in \overline{\mathrm{FUP}}$. Función Mod_3: Consideremos la función

$$F(x) = \begin{cases} 1 & \text{si } x = 0, \\ 2 & \text{si } x = 1, \\ 0 & \text{si } x = 2. \end{cases}$$

La aplicación sucesiva de F genera la sucesión $0, 1, 2, 0, 1, 2, 0, \ldots$ de manera que Mod_3 puede ser definida iterando F. Por otra parte, observemos que

$$\begin{aligned} F(0) &= \mathsf{D}(0) + 2\mathsf{D}(\mathsf{P}(0) + \mathsf{D}(0)) = 1 + 2\mathsf{D}(0+1) = 1 + 0 = 1, \\ F(1) &= \mathsf{D}(1) + 2\mathsf{D}(\mathsf{P}(1) + \mathsf{D}(1)) = 0 + 2\mathsf{D}(0+0) = 0 + 2 = 2, \\ F(2) &= \mathsf{D}(2) + 2\mathsf{D}(\mathsf{P}(2) + \mathsf{D}(2)) = 0 + 2\mathsf{D}(1+0) = 0 + 0 = 0. \\ &\therefore F(x) = \mathsf{D}(x) + 2\mathsf{D}(\mathsf{P}(x) + \mathsf{D}(x)). \end{aligned}$$

$$\boxed{\mathsf{Mod}_3 \equiv (\mathsf{D} + \mathsf{Dbl}\ \mathsf{D}(\mathsf{P} + \mathsf{D}))^{\square}}$$

Función Hf: Consideremos $F(x) = x + \mathsf{Mod}_3(x) + 1$. La aplicación sucesiva de F genera la sucesión $0, 1, 3, 4, 6, 7, 9, 10, 12, \ldots$ de manera que $G(x) = x + \lfloor \frac{x}{2} \rfloor$ puede ser definida iterando F. Lo que sigue es una prueba por inducción de este hecho (donde $x \in \mathbb{N}$):

$$\begin{aligned} G(0) &= 0, \quad (\ = 0 + \left\lfloor \frac{0}{2} \right\rfloor\ ) \\ G(2x+1) &= F(G(2x)) = F(2x + \left\lfloor \frac{2x}{2} \right\rfloor) = F(3x) \\ &= 3x + \mathsf{Mod}_3(3x) + 1 = 3x + 1, \quad (\ = 2x + 1 + \left\lfloor \frac{2x+1}{2} \right\rfloor\ ) \\ G(2x+2) &= F(G(2x+1)) = F(2x + 1 + \left\lfloor \frac{2x+1}{2} \right\rfloor) = F(3x+1) \\ &= 3x + 1 + \mathsf{Mod}_3(3x+1) + 1 = 3x + 3. \quad (\ = 2x + 2 + \left\lfloor \frac{2x+2}{2} \right\rfloor\ ) \end{aligned}$$

Finalmente, $\mathsf{Hf}(x) = G(x) - x$.

$$\boxed{\mathsf{Hf} \equiv (\mathsf{S} + \mathsf{Mod}_3)^{\square} - \mathsf{X}}$$

□

Teorema 3.2.B. *Sean $F, G \in \overline{\mathrm{FUP}}$. Entonces el producto de ellas también es una función unaria primitiva. A esta función la notaremos $\Pi(F, G)$.*

Demostración. Observemos la siguiente igualdad

$$2F(x)G(x) = (F(x) + G(x))^2 - F(x)^2 - G(x)^2.$$

Luego, dividiendo por 2 ambos términos, obtenemos

$$\Pi(F, G) \equiv \mathsf{Hf}(\mathsf{Sq}(F + G) - \mathsf{Sq}F - \mathsf{Sq}G).$$

□

Teorema 3.2.C. *Sean* $F, G \in \overline{\mathrm{FUP}}$. *Entonces* $\mathsf{J}(F(x), G(x))$ *también es una función unaria primitiva. Además* $\mathsf{K}, \mathsf{L} \in \overline{\mathrm{FUP}}$ *(*J*,* K *y* L *eran las funciones de parificación de Cantor).*

Demostración. En primer lugar debemos probar que las funciones $\blacktriangle$ y $\blacktriangledown$ son unarias primitivas. Las fórmulas

$$\blacktriangle \equiv \mathsf{Hf}(\mathsf{Sq} + \mathsf{X}),$$
$$\blacktriangledown \equiv \mathsf{Hf}\ \mathsf{P}\ \mathsf{Rt}\ \mathsf{S}\ \mathsf{Dbl}\ \mathsf{Dbl}\ \mathsf{Dbl},$$

satisfacen las definiciones (2.1.3) y (2.1.4). Y de las definiciones de (2.1.5), (2.1.6) y (2.1.7) surgen

$$\mathsf{J}(F, G) \equiv \blacktriangle(F + G) + F,$$
$$\mathsf{K} \equiv \mathsf{X} - \blacktriangle\blacktriangledown,$$
$$\mathsf{L} \equiv \mathsf{P}(\blacktriangle\mathsf{S}\blacktriangledown - \mathsf{X}).$$

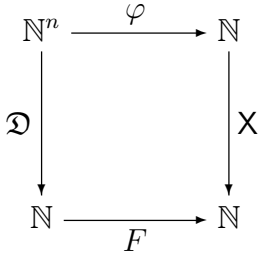
□

3.3. Reducción a funciones unarias primitivas

En esta sección presentaremos un algoritmo para reducir las funciones recursivas primitivas a funciones unarias primitivas, a fin de poder demostrar que $\overline{\mathrm{FRP}} \doteq \overline{\mathrm{FUP}}$.
En un sentido es trivial, pues cualquier FUP puede ser representada con una FRP sin mayor inconveniente. Las funciones iniciales $\mathsf{D}, \mathsf{Sq}, \mathsf{Rt}, \Sigma$ son FRPs y la iteración es un caso particular de la recursión primitiva. Para reemplazar la resta se puede utilizar Mon o Dist. De esta manera, $\mathfrak{D}$ es una identidad y $\mathfrak{P}$ es

$\mathfrak{P} : \overline{\mathrm{FUP}} \to \overline{\mathrm{FRP}}$
$\mathfrak{P}\ \mathsf{D} = \mathsf{D}$
$\mathfrak{P}\ \mathsf{Sq} = \mathsf{Sq}$
$\mathfrak{P}\ \mathsf{Rt} = \mathsf{Rt}$
$\mathfrak{P}\ (F + G) = \phi[\Sigma, \mathfrak{P}\ F, \mathfrak{P}\ G]$
$\mathfrak{P}\ (F - G) = \phi[\mathsf{Mon}, \mathfrak{P}\ F, \mathfrak{P}\ G]$
$\mathfrak{P}\ (FG) = \phi[\mathfrak{P}\ F, \mathfrak{P}\ G]$
$\mathfrak{P}\ (F^{\square}) = \rho[\zeta, \phi[\mathfrak{P}\ F, \pi_2^2]]$

originando $\overline{\mathrm{FRP}} \dot{\supset} \overline{\mathrm{FUP}}$.
Pasemos a la demostración de $\overline{\mathrm{FRP}} \dot{\subset} \overline{\mathrm{FUP}}$. Vamos a utilizar el tratamiento dado por R. Robinson[Rob47] (un método similar se puede consultar en el capítulo 7 de [Goo57]). Dado que las FRPs cuentan con funciones de más de un argumento, el primer paso es reducir los argumentos a sólo uno, empleando una estrategia de parificación,



donde

$\mathfrak{D} : \mathbb{N}^n \to \mathbb{N}$
$\mathfrak{D}\ () = 0$
$\mathfrak{D}\ (x) = x$
$\mathfrak{D}\ (x_1, x_2, \ldots, x_n) = \mathsf{J}(x_1, \mathfrak{D}\ (x_2, \ldots, x_n))$

En el resto de la sección trataremos de hallar una F apropiada que verifique

$$F(\mathsf{J}(x_1, \mathsf{J}(x_2, \mathsf{J}(x_3, \ldots)))) = \varphi(x_1, x_2, x_3, \ldots).$$

Las funciones iniciales surgen naturalmente. Si $2 \leq i \leq n$, entonces

$\mathfrak{P} : \overline{\mathrm{FRP}} \to \overline{\mathrm{FUP}}$
$\mathfrak{P}\ \zeta = \mathsf{Z}$
$\mathfrak{P}\ \sigma = \mathsf{S}$
$\mathfrak{P}\ \pi_1^1 = \mathsf{X}$
$\mathfrak{P}\ \pi_1^n = \mathsf{K}$
$\mathfrak{P}\ \pi_i^n =$ if $i = n = 2$ then L else $(\mathfrak{P}\ \pi_{i-1}^{n-1})\mathsf{L}$

Y la composición se efectúa de manera análoga a la parificación de la entrada,

$$\left|\ \mathfrak{P}\ \phi[\varphi, \omega_1, \omega_2, \ldots, \omega_n] = (\mathfrak{P}\ \varphi)(\mathfrak{D}'\ (\mathfrak{P}\ \omega_1, \mathfrak{P}\ \omega_2, \ldots, \mathfrak{P}\ \omega_n))\ \right|$$

donde

$\mathfrak{D}' : \mathit{List}^{\geq 1}(\overline{\mathrm{FUP}}) \to \overline{\mathrm{FUP}}$
$\mathfrak{D}'\ [\omega] = \omega$
$\mathfrak{D}'\ [\omega_1, \omega_2, \ldots, \omega_n] = \mathsf{J}(\omega_1, \mathfrak{D}'\ (\omega_2, \ldots, \omega_n))$

Ahora estudiaremos la forma de reducir $\rho[\beta, \varphi]$. Dado que hay que realizar varias reducciones antes de alcanzar las FUPs, será conveniente enriquecer a las FRPs con ciertas funciones de parificación.

$$\left|\begin{array}{l} \mathfrak{P}\ \langle \omega_1, \omega_2, \ldots \omega_n \rangle = \mathfrak{D}'\ (\mathfrak{P}\ \omega_1, \mathfrak{P}\ \omega_2, \ldots, \mathfrak{P}\ \omega_n) \\ \mathfrak{P}\ \tau_1^n = \mathsf{K} \\ \mathfrak{P}\ \tau_i^n = \text{if } i = n = 2 \text{ then } \mathsf{L} \text{ else } (\mathfrak{P}\ \tau_{i-1}^{n-1})\mathsf{L} \end{array}\right|$$

donde $2 \leq i \leq n$ y $\tau_i^n \langle x_1, x_2, ..., x_n \rangle = x_n$.
También agregaremos la *composición prohibida* que convierte una función constante φ dada en unaria:

$\phi[\varphi] : \mathbb{N} \to \mathbb{N}$,
$\phi[\varphi](x) = \varphi()$.

Esto se refleja en $\mathfrak{P}$ como

$$\left|\ \mathfrak{P}\ \phi[\varphi] = \mathfrak{P}\ \varphi\ \right|$$

Aunque la adjunción de estos nuevos esquemas formen un superconjunto de las funciones recursivas primitivas, no amplian su poder de expresión.

La corrección de los algoritmos que presentaremos se pueden demostrar usando inducción estructural y la propiedad que verifican J, K y L dada en (2.1.1). Para simplificar los esquemas de recursión, seguimos los siguientes pasos.

Paso 1. El esquema general de recursión primitiva es

$$\mathscr{R}\begin{cases} F(x_1, x_2, \ldots, x_n, 0) = \beta(x_1, x_2, \ldots, x_n), \\ F(x_1, x_2, \ldots, x_n, y+1) = \varphi(x_1, x_2, \ldots, x_n, y, F(x_1, x_2, \ldots, x_n, y)), \end{cases}$$

y se denota con $\rho[\beta, \varphi]$. Utilizando una estrategia de parificación, podemos eliminar los parámetros $x_2, \ldots, x_n$, y llevar el esquema $\mathscr{R}$ a

$$\mathscr{R}_1\begin{cases} F_1(x, 0) = A(x), \\ F_1(x, y+1) = B(x, y, F_1(x, y)), \end{cases}$$

que se denota con $\rho_1[A, B]$. Si hacemos $x = \mathfrak{D}(x_1, x_2, ..., x_n)$ entonces

$$\begin{aligned} A(x) &= \beta(x_1, x_2, \ldots, x_n), \\ B(x, y, z) &= \varphi(x_1, x_2, \ldots, x_n, y, z), \\ F_1(x, y) &= F(x_1, x_2, \ldots, x_n, y), \end{aligned}$$

siempre que A y B resulten adecuadas. Consideremos el caso en que $n > 1$. Evaluamos $A = \mathfrak{P}^A\beta$ (en este contexto $\mathfrak{P}^A$ es un algoritmo para generar A a partir de φ, no debe confundirse con $\mathfrak{P}$). Si $1 \leq i \leq n$ entonces

$\mathfrak{P}^A : \overline{\text{FRP}} \to \overline{\text{FRP}}$
$\mathfrak{P}^A\ \pi_i^n = \tau_i^n$
$\mathfrak{P}^A\ \phi[\varphi, \omega_1, \omega_2, \ldots, \omega_n] = \phi[\varphi, \mathfrak{P}^A\ \omega_1, \mathfrak{P}^A\ \omega_2, \ldots, \mathfrak{P}^A\ \omega_n]$
$\mathfrak{P}^A\ \alpha = \alpha$ otherwise

Respectivamente, $B = \mathfrak{P}^B\varphi$. Si $1 \leq i \leq n$ entonces

$\mathfrak{P}^B : \overline{\text{FRP}} \to \overline{\text{FRP}}$
$\mathfrak{P}^B\ \pi_i^{n+2} = \phi[\tau_i^n, \pi_1^3]$
$\mathfrak{P}^B\ \pi_{n+1}^{n+2} = \pi_2^3$
$\mathfrak{P}^B\ \pi_{n+2}^{n+2} = \pi_3^3$
$\mathfrak{P}^B\ \phi[\varphi, \omega_1, \omega_2, \ldots, \omega_n] = \phi[\varphi, \mathfrak{P}^B\ \omega_1, \mathfrak{P}^B\ \omega_2, \ldots, \mathfrak{P}^B\ \omega_n]$
$\mathfrak{P}^B\ \alpha = \alpha$ otherwise

Por lo tanto el algoritmo

$\mathfrak{P}_1 : \overline{\text{FRP}} \to \overline{\text{FRP}}$
$\mathfrak{P}_1\ \phi[\varphi, \omega_1, \omega_2, \ldots, \omega_n] = \phi[\mathfrak{P}_1\ \varphi, \mathfrak{P}_1\ \omega_1, \mathfrak{P}_1\ \omega_2, \ldots, \mathfrak{P}_1\ \omega_n]$
$\mathfrak{P}_1\ \rho[\beta, \varphi] = \phi[\rho_1[\mathfrak{P}^A\ \mathfrak{P}_1\ \beta, \mathfrak{P}^B\ \mathfrak{P}_1\ \varphi], \langle \pi_1^{n+1}, \pi_2^{n+1}, \ldots, \pi_n^{n+1} \rangle, \pi_{n+1}^{n+1}]$
where $n = arity\ \beta$
$\mathfrak{P}_1\ \alpha = \alpha$ otherwise

convierte todos los esquemas $\mathscr{R}$ a $\mathscr{R}_1$.

Los casos $n = 0$ y $n = 1$ se deben agregar al esquema de $\mathfrak{P}_1$. De hecho, si $n = 1$ resulta trivialmente

$$\mathfrak{P}_1\ \rho[\beta, \varphi] = \rho_1[\mathfrak{P}_1\ \beta, \mathfrak{P}_1\ \varphi].$$

Si $n = 0$, podemos agregar una variable para que coincida con $\mathscr{R}_1$: $A \equiv \phi[\beta]$ es la función unaria que retorna la constante β, y $B \equiv \phi[\varphi, \pi_2^3, \pi_3^3]$ es la función ternaria que representa a φ. Entonces,

$$\begin{aligned} A(x) &= \beta(), \\ B(x, y, z) &= \varphi(y, z). \end{aligned}$$

Luego, $\rho_1[A, B]$ es una función binaria que representa a $\rho[\beta, \varphi]$, salvo por la variable adicional. Así que hay que aplicarle una proyección, resultando

$$\mathfrak{P}_1\ \rho[\beta, \varphi] = \phi[\rho_1[\phi[\mathfrak{P}_1\ \beta], \phi[\mathfrak{P}_1\ \varphi, \pi_2^3, \pi_3^3]], \pi_1^1, \pi_1^1].$$

Paso 2. Utilizando una estrategia de parificación similar, podemos eliminar el parámetro x de B, y llevar el esquema $\mathscr{R}_1$ a

$$\mathscr{R}_2 \begin{cases} F_2(x, 0) = C(x), \\ F_2(x, y+1) = D(y, F_2(x, y)), \end{cases}$$

que se denota con $\rho_2[C, D]$. Esta vez, haremos uso de la equivalencia

$$F_2(x, y) = \langle x, F_1(x, y)\rangle.$$

Debemos tener presente que $\langle x, y\rangle = \mathsf{J}(x, y)$, $\tau_1^2(z) = \mathsf{K}(z)$ y $\tau_2^2(z) = \mathsf{L}(z)$. Entonces,

$$\begin{aligned} F_2(x, 0) &= \langle x, A(x)\rangle, \\ F_2(x, y+1) &= \langle x, B(x, y, F_1(x, y))\rangle \\ &= \langle \tau_1^2\langle x, F_1(x, y)\rangle, B(\tau_1^2\langle x, F_1(x, y)\rangle, y, \tau_2^2\langle x, F_1(x, y)\rangle)\rangle \\ &= \langle \tau_1^2 F_2(x, y), B(\tau_1^2 F_2(x, y), y, \tau_2^2 F_2(x, y))\rangle. \end{aligned}$$

Es decir,

$$\begin{aligned} C(x) &= \langle x, A(x)\rangle, \\ D(y, z) &= \langle \tau_1^2 z, B(\tau_1^2 z, y, \tau_2^2 z)\rangle, \\ F_1(x, y) &= \tau_2^2 F_2(x, y). \end{aligned}$$

Por lo tanto el algoritmo

$\mathfrak{P}_2 : \overline{\mathrm{FRP}} \to \overline{\mathrm{FRP}}$
$\mathfrak{P}_2\ \phi[\varphi, \omega_1, \omega_2, \ldots, \omega_n] = \phi[\mathfrak{P}_2\ \varphi, \mathfrak{P}_2\ \omega_1, \mathfrak{P}_2\ \omega_2, \ldots, \mathfrak{P}_2\ \omega_n]$ $\mathfrak{P}_2\ \rho_1[A, B] = \phi[\tau_2^2, \rho_2[\langle \pi_1^1, \mathfrak{P}_2\ A\rangle, \langle \phi[\tau_1^2, \pi_2^2], \phi[\mathfrak{P}_2\ B, \phi[\tau_1^2, \pi_2^2], \pi_1^2, \phi[\tau_2^2, \pi_2^2]]\rangle]]$ $\mathfrak{P}_2\ \alpha = \alpha$ otherwise

convierte todos los esquemas $\mathscr{R}_1$ a $\mathscr{R}_2$.

Paso 3. Una reducción sencilla es eliminar la función C, llevando el esquema $\mathscr{R}_2$ a

$$\mathscr{R}_3 \begin{cases} F_3(x, 0) = x, \\ F_3(x, y+1) = E(y, F_3(x, y)), \end{cases}$$

que se denota con $\rho_3[E]$. Haciendo $E(y, z) = D(y, z)$ resulta

$$F_2(x, y) = F_3(C(x), y).$$

Es decir, el algoritmo

$\mathfrak{P}_3 : \overline{\mathrm{FRP}} \to \overline{\mathrm{FRP}}$
$\mathfrak{P}_3\ \phi[\varphi, \omega_1, \omega_2, \ldots, \omega_n] = \phi[\mathfrak{P}_3\ \varphi, \mathfrak{P}_3\ \omega_1, \mathfrak{P}_3\ \omega_2, \ldots, \mathfrak{P}_3\ \omega_n]$ $\mathfrak{P}_3\ \rho_2[C, D] = \phi[\rho_3[\mathfrak{P}_3\ D], \phi[\mathfrak{P}_3\ C, \pi_1^2], \pi_2^2]$ $\mathfrak{P}_3\ \alpha = \alpha$ otherwise

convierte todos los esquemas $\mathscr{R}_2$ a $\mathscr{R}_3$.

Paso 4. Este es el paso más difícil. Tenemos que reducir el esquema $\mathscr{R}_3$ a

$$\mathscr{R}_4 \begin{cases} F_4(0) = 0, \\ F_4(x+1) = U(x, F_4(x)), \end{cases}$$

que se denota con $\rho_4[U]$. Para nuestro fin, será útil agregar funciones de parificación que presenten una característica adicional. Se debe verificar $J(0,0) = 0$ y

$$L(x+1) \neq 0 \implies K(x+1) = K(x) \wedge L(x+1) = L(x) + 1. \tag{3.3.1}$$

Analicemos mejor la última propiedad: sean $x_1, x_2 \in \mathbb{N}$ tales que verifican $x_1 < x_2$ y $L(x_1) = L(x_2) = 0$. Asumiremos que todos los valores intermedios no anularán L, i.e. $L(x) \neq 0$ para toda x entre x_1 y x_2. Luego, los pares

$$(K(x_1), 0), (K(x_1), 1), (K(x_1), 2), \ldots, (K(x_1), x_2 - x_1 - 1)$$

son números consecutivos al aplicársele J.
Antes de continuar con la reducción de $\mathscr{R}_3$ a $\mathscr{R}_4$, haremos un paréntesis para mostrar funciones J, K y L apropiadas.

Teorema 3.3.A. *Las funciones*

$$\begin{aligned} J(x,y) &= ((x+y)^2 + x)^2 + y, \\ K(x) &= \lfloor \sqrt{x} \rfloor - \left\lfloor \sqrt{\lfloor \sqrt{x} \rfloor} \right\rfloor^2, \\ L(x) &= x - \lfloor \sqrt{x} \rfloor^2, \end{aligned}$$

además de ser unarias primitivas, son de parificación y verifican $J(0,0) = 0$ y la propiedad (3.3.1) antes mencionada.

Demostración. Sea $z = J(x,y)$. Este valor está entre dos cuadrados perfectos,

$$((x+y)^2 + x)^2 \leq z < ((x+y)^2 + x + 1)^2.$$

Luego $\lfloor \sqrt{z} \rfloor = (x+y)^2 + x$, el cual está, a su vez, entre dos cuadrados perfectos,

$$(x+y)^2 \leq \lfloor \sqrt{z} \rfloor < (x+y+1)^2.$$

Por lo tanto, $\lfloor \sqrt{\lfloor \sqrt{z} \rfloor} \rfloor = x + y$. Ahora es más sencillo probar la propiedad (2.1.1).

$$\begin{aligned} K(J(x,y)) = K(z) &= \lfloor \sqrt{z} \rfloor - \left\lfloor \sqrt{\lfloor \sqrt{z} \rfloor} \right\rfloor^2 = (x+y)^2 + x - (x+y)^2 = x, \\ L(J(x,y)) = L(z) &= z - \lfloor \sqrt{z} \rfloor^2 = ((x+y)^2 + x)^2 + y - ((x+y)^2 + x)^2 = y. \end{aligned}$$

Ahora probemos (3.3.1). Supongamos que $L(z+1) \neq 0$, luego $z+1$ no es cuadrado perfecto. En consecuencia, $\lfloor \sqrt{z+1} \rfloor = \lfloor \sqrt{z} \rfloor$ y

$$\begin{aligned} K(z+1) &= (x+y)^2 + x - (x+y)^2 = x = K(z), \\ L(z+1) &= ((x+y)^2 + x)^2 + y + 1 - ((x+y)^2 + x)^2 = y + 1 = L(z) + 1. \end{aligned}$$

Si F y G son funciones unarias primitivas, entonces

$$\begin{aligned} J(F,G) &\equiv \mathsf{Sq}(\mathsf{Sq}(F+G)+F)+G, \\ K &\equiv (\mathsf{X} - \mathsf{Sq}\ \mathsf{Rt})\mathsf{Rt}, \\ L &\equiv \mathsf{X} - \mathsf{Sq}\ \mathsf{Rt}. \end{aligned}$$

□

Sea $F_3(x,y) = F_4(J(x,y))$. Si, en particular, $x = y = 0$ entonces ambas funciones retornan cero. Para el resto de los casos vamos a proponer

$$U(x,y) = \begin{cases} K(x+1) & \text{si } L(x+1) = 0, \\ E(L(x), y) & \text{si } L(x+1) \neq 0. \end{cases}$$

Probemos ahora que $F_4(J(x,0)) = F_3(x,0)$. Sea $z+1 = J(x,0)$, luego $L(z+1) = 0$ y

$$F_4(z+1) = U(z, F_4(z)) = K(z+1) = x = F_3(x,0).$$

Suponiendo que $F_4(J(x,y)) = F_3(x,y)$, probemos que $F_4(J(x,y+1)) = F_3(x,y+1)$. Sea $z+1 = J(x,y+1)$, luego $L(z+1) \neq 0$ verificándose la propiedad (3.3.1) (recordar que los pares eran consecutivos, i.e. $J(x,y+1) = J(x,y)+1$, de donde $z = J(x,y)$):

$$\begin{aligned} F_4(z+1) &= U(z, F_4(z)) = E(L(z), F_4(z)) \\ &= E(y, F_4(J(x,y))) = E(y, F_3(x,y)) = F_3(x, y+1). \end{aligned}$$

De esta forma, U resulta apropiada. Una posible representación de la misma es

$$U(x,y) = \mathsf{D}(L(\sigma(x))).K(\sigma(x)) + \mathsf{D}(\mathsf{D}(L(\sigma(x)))).E(L(x), y).$$

Sin embargo, hemos introducido algunas funciones auxiliares[2]: D, J, K, L, Σ y Π. Adjuntémoslas al formalismo $\overline{\mathrm{FRP}}$ (que, como sabemos, no aumentará su poder de expresión).

$$\left| \begin{array}{l} \mathfrak{P}\ \mathsf{D} = \mathsf{D} \\ \mathfrak{P}\ J[\omega_1, \omega_2] = J(\mathfrak{P}\ \omega_1, \mathfrak{P}\ \omega_2) \\ \mathfrak{P}\ K = K \\ \mathfrak{P}\ L = L \\ \mathfrak{P}\ \Sigma[\omega_1, \omega_2] = \mathfrak{P}\ \omega_1 + \mathfrak{P}\ \omega_2 \\ \mathfrak{P}\ \Pi[\omega_1, \omega_2] = \Pi(\mathfrak{P}\ \omega_1, \mathfrak{P}\ \omega_2) \end{array} \right|$$

El algoritmo

[2] $\Sigma[\omega_1,\omega_2](x_1,x_2,\ldots,x_n) = \omega_1(x_1,x_2,\ldots,x_n) + \omega_2(x_1,x_2,\ldots,x_n)$,
$\Pi[\omega_1,\omega_2](x_1,x_2,\ldots,x_n) = \omega_1(x_1,x_2,\ldots,x_n).\omega_2(x_1,x_2,\ldots,x_n)$.

$\mathfrak{P}_4 : \overline{\mathrm{FRP}} \to \overline{\mathrm{FRP}}$
$\mathfrak{P}_4\ \phi[\varphi, \omega_1, \omega_2, \ldots, \omega_n] = \phi[\mathfrak{P}_4\ \varphi, \mathfrak{P}_4\ \omega_1, \mathfrak{P}_4\ \omega_2, \ldots, \mathfrak{P}_4\ \omega_n]$ $\mathfrak{P}_4\ \rho_3[E] = \phi[\rho_4[\Sigma[\Pi[\phi[\mathsf{D}, \phi[L, \phi[\sigma, \pi_1^2]]], \phi[K, \phi[\sigma, \pi_1^2]]],$ $\quad \Pi[\phi[\mathsf{D}, \phi[\mathsf{D}, \phi[L, \phi[\sigma, \pi_1^2]]]], \phi[\mathfrak{P}_4\ E, \phi[L, \pi_1^2], \pi_2^2]]]], J[\pi_1^2, \pi_2^2]]$ $\mathfrak{P}_4\ \alpha = \alpha \quad \text{otherwise}$

convierte todos los esquemas $\mathscr{R}_3$ a $\mathscr{R}_4$.

Paso 5. Finalmente, podemos eliminar el parámetro x de U, y llevar el esquema $\mathscr{R}_4$ a una iteración

$$\mathscr{R}_5 \begin{cases} F_5(0) = 0, \\ F_5(x+1) = V(F_5(x)), \end{cases}$$

que se denota con $\rho_5[V]$. Vemos que $F_5(x) = \langle x, F_4(x)\rangle$ puede ser definida usando

$$V(x) = \langle \tau_1^2(x) + 1, U(\tau_1^2(x), \tau_2^2(x))\rangle$$

siempre que $\langle 0, 0\rangle = 0$ (las funciones de parificación de Cantor lo cumplen). Para el caso inductivo, observemos que si $F_5(x) = \langle x, F_4(x)\rangle$ entonces

$$F_5(x+1) = V(F_5(x)) = V(\langle x, F_4(x)\rangle) = \langle x+1, U(x, F_4(x))\rangle = \langle x+1, F_4(x+1)\rangle$$

de manera que se puede obtener F_4 extrayendo la segunda componente que otorga F_5.
Por lo tanto el algoritmo

$\mathfrak{P}_5 : \overline{\mathrm{FRP}} \to \overline{\mathrm{FRP}}$
$\mathfrak{P}_5\ \phi[\varphi, \omega_1, \omega_2, \ldots, \omega_n] = \phi[\mathfrak{P}_5\ \varphi, \mathfrak{P}_5\ \omega_1, \mathfrak{P}_5\ \omega_2, \ldots, \mathfrak{P}_5\ \omega_n]$ $\mathfrak{P}_5\ \rho_4[U] = \phi[\tau_2^2, \rho_5[\langle \phi[\sigma, \tau_1^2], \phi[\mathfrak{P}_5\ U, \tau_1^2, \tau_2^2]\rangle]]$ $\mathfrak{P}_5\ \alpha = \alpha \quad \text{otherwise}$

convierte todos los esquemas $\mathscr{R}_4$ a $\mathscr{R}_5$. Y, dado que $\mathscr{R}_5$ es simplemente una iteración, se deduce

$$\boxed{\mathfrak{P}\ \rho_5[V] = (\mathfrak{P}\ V)^{\square}}$$

En resumen, el algoritmo

$$\mathfrak{P} \circ \mathfrak{P}_5 \circ \mathfrak{P}_4 \circ \mathfrak{P}_3 \circ \mathfrak{P}_2 \circ \mathfrak{P}_1 : \overline{\mathrm{FRP}} \to \overline{\mathrm{FUP}}$$

transforma funciones recursivas primitivas en funciones unarias primitivas, verificando

$$(\mathfrak{P}\ \mathfrak{P}_5\ \mathfrak{P}_4\ \mathfrak{P}_3\ \mathfrak{P}_2\ \mathfrak{P}_1\ \varphi)(\mathfrak{D}(x_1, x_2, \ldots, x_n)) = \varphi(x_1, x_2, \ldots, x_n).$$

Hemos probado, entonces, el siguiente teorema.

Teorema 3.3.B. *Las funciones unarias primitivas tienen el mismo poder expresivo que las funciones recursivas primitivas.*

3.4. Inversión y funciones unarias generales

A continuación vamos a introducir un operador para poder exponer las *funciones unarias generales*, las cuales resultarán tan expresivas como las FRGs.

Definición 3.4.A. *Dada la función $F : \hat{\mathbb{N}} \to \hat{\mathbb{N}}$ se define su* inversa *como*

$$F^-(x) = \min_{y\in\mathbb{N}}\big\{F(y) = x\big\}.$$

Si, en particular, F es biyectiva entonces $F^-(x) = F^{-1}(x)$ como era de esperarse.
La inversión también puede generar $\perp$ si nunca se cumple la proposición $F(y) = x$ para toda y, o si antes de que se cumpla se verifica $F(y') = \perp$ (aquí $y' < y$).
A la inversión de F la notamos como (F^-).

En las fórmulas que escribiremos se adoptarán las convenciones tipográficas dadas para las FUPs, en donde la inversión tendrá la misma precedencia que la iteración. Para trabajar con este nuevo operador, será útil el siguiente teorema.

Teorema 3.4.B. *Sea $F : \hat{\mathbb{N}} \to \hat{\mathbb{N}}$. Si F^- es total entonces $F(F^-(x)) = x$ para toda x. Y si G es una función total y natural tal que*

$$F(G(x)) = x \quad \textit{para toda } x, \tag{3.4.1}$$
$$F(y') \neq x \quad \textit{para toda } y' < G(x), \tag{3.4.2}$$

entonces $F^-(x) = G(x)$.

Demostración. Por definición, $F^-(x) = y$, donde y es el mínimo valor tal que $F(y) = x$. Pero entonces,

$$F(F^-(x)) = F(y) = x,$$
$$F(y') \neq x,$$

donde $y' < y$. Luego, si G verifica (3.4.1) y (3.4.2) entonces $G(x) = y$. □

Desarrollaremos un ejemplo para mostrar el uso de la inversión y del Teorema 3.4.B. La función $\mathsf{E}(x) = x - \lfloor\sqrt{x}\rfloor^2$, llamada *exceso sobre los cuadrados perfectos*, es unaria primitiva pues

$$\mathsf{E} \equiv \mathsf{RtSq} - \mathsf{SqRt}.$$

Ejemplo 3.4.C. *Propondremos una fórmula para la inversa de E, aunque sea un tanto compleja. Si $z = \left\lfloor \dfrac{x+3}{2} \right\rfloor$, entonces*

$$\mathsf{E}^-(x) = z^2 - 2z + x + 1.$$

En efecto, sean $y = z^2 - 2z + x + 1$, $y' < y$. Debemos probar que
1) $\mathsf{E}(y) = x$ para toda x impar,
2) $\mathsf{E}(y) = x$ para toda x par,
3) $\mathsf{E}(y') < x$ para toda x impar,
4) $\mathsf{E}(y') < x$ para toda x par.

Sólo mostraremos la prueba de (1). Las restantes se deducen de manera análoga.
Si x es impar, entonces $z = \left\lfloor \frac{x+3}{2} \right\rfloor = \frac{x+3}{2}$. *Luego,*

$$y = \frac{(x+3)^2}{4} - (x+3) + x + 1 = \frac{x^2+6x+1}{4}.$$

Este valor está entre dos cuadrados perfectos consecutivos,

$$\left(\frac{x+1}{2}\right)^2 = \frac{x^2+2x+1}{4} < y < \frac{x^2+6x+9}{4} = \left(\frac{x+3}{2}\right)^2.$$

Por lo tanto, $\lfloor \sqrt{y} \rfloor = \frac{x+1}{2}$, *y finalmente*

$$\mathsf{E}(y) = y - \lfloor \sqrt{y} \rfloor^2 = \frac{x^2+6x+1}{4} - \frac{x^2+2x+1}{4} = x.$$

DEFINICIÓN 3.4.D. *Las* funciones unarias generales *son el resultado de agregar a las funciones unarias primitivas el esquema de inversión definido en 3.4.A. Denotaremos al conjunto de las funciones unarias generales como* $\overline{\text{FUG}}$.

Claramente, cualquier función unaria primitiva es una función unaria general. Es decir, sean $F, G \in \overline{\text{FUG}}$; si $H \in \overline{\text{FUG}}$ entonces sólo se puede satisfacer una de las seis condiciones siguientes para H.

$$\begin{aligned} &H \in \overline{\text{FUP}}, && H \equiv (F+G), \\ &H \equiv (F-G), && H \equiv (FG), \\ &H \equiv (F^{\square}), && H \equiv (F^{-}). \end{aligned}$$

Lo que desarrollaremos en esta sección es que las FUGs pueden representar a las FRGs, o más específicamente, que *las funciones unarias generales totales pueden representar a las funciones recursivas generales totales.* Esto tiene una notable relevancia, pues estamos otorgando un modelo computacional con el mismo poder de expresión que las máquinas de Turing[3]. Para llevar a cabo nuestro objetivo, debemos extender el algoritmo $\mathfrak{P}$ propuesto en la sección 3.3.
En primer lugar, $\overline{\text{FRG}} \dot{\supset} \overline{\text{FUG}}$ pues existen $\mathfrak{D}$ y $\mathfrak{P}$ apropiados. $\mathfrak{D}$ será una identidad, y $\mathfrak{P}$ será

$\mathfrak{P} : \overline{\text{FUG}} \to \overline{\text{FRG}}$
$\mathfrak{P}\ (F+G) = \phi[\Sigma, \mathfrak{P}\ F, \mathfrak{P}\ G]$
$\mathfrak{P}\ (F-G) = \phi[\mathsf{Mon}, \mathfrak{P}\ F, \mathfrak{P}\ G]$
$\mathfrak{P}\ (FG) = \phi[\mathfrak{P}\ F, \mathfrak{P}\ G]$
$\mathfrak{P}\ (F^{\square}) = \rho[\zeta, \phi[\mathfrak{P}\ F, \pi_2^2]]$
$\mathfrak{P}\ (F^{-}) = \mu\phi[\mathsf{Dist}, \phi[\mathfrak{P}\ F, \pi_2^2], \pi_1^2]$
$\mathfrak{P}\ F =$ if $F \in \overline{\text{FUP}}$ then $\mathfrak{P} < \overline{\text{FUP}} >\ F$

[3]Cabe aclarar que, aunque sólo probemos el resultado para funciones totales, esto no reduce el poder expresivo de las FUGs. Aunque esta verificación no sea trivial, no la expondremos.

La penúltima línea del algoritmo muestra que la inversión es un caso particular de la minimización. Y la última línea revela que las FUGs contienen a las FUPs. La nomenclatura $\mathfrak{P} < \overline{\text{FUP}} >$ indica que debemos usar $\mathfrak{P} : \overline{\text{FUP}} \to \overline{\text{FRP}}$ (ya definido cuando probamos $\overline{\text{FRP}} \dot{\supset} \overline{\text{FUP}}$).
Notemos que no hemos introducido el elemento $\perp$. De hecho, si F es total pero $F^-(x) = \perp$ para algún x, entonces $(\mathfrak{P}\ F^-)(x) = \perp$ también. Pero no aseguramos nada respecto a la suma, resta, composición, iteración e inversión de funciones parciales.

Pasemos a la demostración de $\overline{\text{FRG}} \dot{\subset} \overline{\text{FUG}}$. Vamos a utilizar un tratamiento similar al de J. Robinson[Rob50]. Dado que las FRGs cuentan con funciones de más de un argumento, debemos reducir los argumentos a sólo uno, empleando las estrategias de parificación que aparecen en la sección 3.3. Por eso *reutilizaremos* los algoritmos $\mathfrak{P}_1$, $\mathfrak{P}_2$, $\mathfrak{P}_3$, $\mathfrak{P}_4$ y $\mathfrak{P}_5$ con la salvedad que en cada uno de ellos deberemos agregar la línea (donde $1 \leq j \leq 5$)

$$\mid\ \mathfrak{P}_j\ \mu\varphi = \mu(\mathfrak{P}_j\ \varphi)\ \mid$$

para hacer las conversiones correspondientes por cada minimización. Para referirnos a la función $\mathfrak{P} : \overline{\text{FRP}} \to \overline{\text{FUP}}$ emplearemos la nomenclatura $\mathfrak{P} < \overline{\text{FRP}} >$. Entonces,

$\mathfrak{D} : \hat{\mathbb{N}}^n \to \hat{\mathbb{N}}$
$\mathfrak{D}\ () = 0$
$\mathfrak{D}\ (x) = x$
$\mathfrak{D}\ (x_1, x_2, \ldots, x_n) =$ if $x_1 = \perp \vee y = \perp$ then $\perp$ else $\mathsf{J}(x_1, y)$
where $y = \mathfrak{D}\ (x_2, \ldots, x_n)$

y

$\mathfrak{P} : \overline{\text{FRG}} \to \overline{\text{FUG}}$
$\mathfrak{P}\ \phi[\varphi, \omega_1, \omega_2, \ldots, \omega_n] = (\mathfrak{P}\ \varphi)(\mathfrak{D}'\ (\mathfrak{P}\ \omega_1, \mathfrak{P}\ \omega_2, \ldots, \mathfrak{P}\ \omega_n))$
$\mathfrak{P}\ \langle \omega_1, \omega_2, \ldots \omega_n \rangle = \mathfrak{D}'\ (\mathfrak{P}\ \omega_1, \mathfrak{P}\ \omega_2, \ldots, \mathfrak{P}\ \omega_n)$
$\mathfrak{P}\ \rho_5[V] = (\mathfrak{P}\ V)^{\square}$
$\mathfrak{P}\ \varphi =$ if $\varphi \in \overline{\text{FRP}}$ then $\mathfrak{P} < \overline{\text{FRP}} >\ \varphi$

donde $\mathfrak{D}'$ ya fue definido en la sección 3.3. Para completar $\mathfrak{P}$, basta con representar adecuadamente al minimizador con una inversión. En primer lugar, debemos reducir la cantidad de argumentos del minimizador a sólo los necesarios.

El esquema general de minimización es

$$\mathscr{M} : \quad F(x_1, x_2, \ldots, x_n) = \min_{y \in \mathbb{N}} \big\{ \varphi(x_1, x_2, \ldots, x_n, y) = 0 \big\},$$

que se denota con $\mu\varphi$. Utilizando una estrategia de parificación podemos eliminar los parámetros $x_2, \ldots, x_n$, y llevar el esquema $\mathscr{M}$ a

$$\hat{\mathscr{M}} : \quad \hat{F}(x) = \min_{y \in \mathbb{N}} \big\{ G(x, y) = 0 \big\},$$

que se denota con $\hat{\mu}G$. Para el caso $n = 1$, la conversión es trivial. Y, en el caso $n = 0$, la función F resultará una constante. Estos casos se pueden realizar de manera análoga a la

planteada cuando definimos $\mathfrak{P}_1$.
Si hacemos $x = \mathfrak{D}(x_1, x_2, ..., x_n)$ entonces

$$G(x, y) = \varphi(x_1, x_2, \ldots, x_n, y),$$
$$\hat{F}(x) = F(x_1, x_2, \ldots, x_n),$$

siempre que G resulte adecuada. El siguiente algoritmo retorna la función G para un φ dado (donde $1 \leq i \leq n$).

$\mathfrak{P}^G : \overline{\mathrm{FRG}} \to \overline{\mathrm{FRG}}$
$\mathfrak{P}^G\ \pi_i^{n+1} = \phi[\tau_i^n, \pi_1^2]$
$\mathfrak{P}^G\ \pi_{n+1}^{n+1} = \pi_2^2$
$\mathfrak{P}^G\ \phi[\varphi, \omega_1, \omega_2, \ldots, \omega_n] = \phi[\varphi, \mathfrak{P}^G\ \omega_1, \mathfrak{P}^G\ \omega_2, \ldots, \mathfrak{P}^G\ \omega_n]$
$\mathfrak{P}^G\ \alpha = \alpha$ otherwise

Luego, el algoritmo

$\mathfrak{P}_6 : \overline{\mathrm{FRG}} \to \overline{\mathrm{FRG}}$
$\mathfrak{P}_6\ \phi[\varphi, \omega_1, \omega_2, \ldots, \omega_n] = \phi[\mathfrak{P}_6\ \varphi, \mathfrak{P}_6\ \omega_1, \mathfrak{P}_6\ \omega_2, \ldots, \mathfrak{P}_6\ \omega_n]$
$\mathfrak{P}_6\ \rho_5[V] = \rho_5[\mathfrak{P}_6\ V]$
$\mathfrak{P}_6\ \mu\varphi = \hat{\mu}(\mathfrak{P}^G\ \mathfrak{P}_6\ \varphi)$
$\mathfrak{P}_6\ \alpha = \alpha$ otherwise

convierte todos los esquemas $\mathscr{M}$ a $\hat{\mathscr{M}}$.

Nuestro siguiente paso es llevar el esquema $\hat{\mathscr{M}}$ a una inversión

$$\hat{\hat{\mathscr{M}}} : \quad \hat{\hat{F}}(x) = \min_{z \in \mathbb{N}} \big\{ H(z) = x \big\},$$

que se denota con $\hat{\mu}H$. Verificaremos que, si se propone $H(z) = \mathsf{D}(G(\mathsf{K}(z), \mathsf{L}(z))).\mathsf{S}(\mathsf{K}(z))$, entonces

$$\hat{F}(x) = \mathsf{L}(\hat{\hat{F}}(x+1)), \tag{3.4.3}$$

siempre que $\hat{F}$ y G no sean parciales.
Supongamos que existe una $y \in \mathbb{N}$ mínima tal que $G(x, y) = 0$ (y $\hat{F}(x) = y$). Luego, $\mathsf{D}(G(\mathsf{K}(z), \mathsf{L}(z))) = 1$ cuando $z = \mathsf{J}(x, y)$ (dado que a una y mínima le corresponde una $z = \mathsf{J}(x, y)$ mínima para un valor de x fijo, por las propiedades de monotonía de J). Para tal z, $H(z) = x + 1$ y $\hat{\hat{F}}(x + 1) = z$, verificándose (3.4.3).
El algoritmo

$\mathfrak{P}_7 : \overline{\mathrm{FRG}} \to \overline{\mathrm{FRG}}$
$\mathfrak{P}_7\ \phi[\varphi, \omega_1, \omega_2, \ldots, \omega_n] = \phi[\mathfrak{P}_7\ \varphi, \mathfrak{P}_7\ \omega_1, \mathfrak{P}_7\ \omega_2, \ldots, \mathfrak{P}_7\ \omega_n]$
$\mathfrak{P}_7\ \rho_5[V] = \rho_5[\mathfrak{P}_7\ V]$
$\mathfrak{P}_7\ \hat{\mu}G = \phi[\mathsf{L}, \phi[\hat{\hat{\mu}}\Pi[\phi[\mathsf{D}, \phi[G, \mathsf{K}, \mathsf{L}]], \phi[\sigma, \mathsf{K}]], \sigma]]$
$\mathfrak{P}_7\ \alpha = \alpha$ otherwise

convierte todos los esquemas $\hat{\mathscr{M}}$ a $\hat{\hat{\mathscr{M}}}$. Y, dado que $\hat{\hat{\mathscr{M}}}$ es simplemente una inversión, se deduce

$$\boxed{\mathfrak{P}\ \hat{\mu}H = (\mathfrak{P}\ H)^-}$$

En resumen, el algoritmo

$$\mathfrak{P} \circ \mathfrak{P}_7 \circ \mathfrak{P}_6 \circ \mathfrak{P}_5 \circ \mathfrak{P}_4 \circ \mathfrak{P}_3 \circ \mathfrak{P}_2 \circ \mathfrak{P}_1 : \overline{\mathrm{FRG}} \to \overline{\mathrm{FUG}}$$

transforma funciones recursivas generales (totales) en funciones unarias generales, verificando

$$(\mathfrak{P}\ \mathfrak{P}_7\ \mathfrak{P}_6\ \mathfrak{P}_5\ \mathfrak{P}_4\ \mathfrak{P}_3\ \mathfrak{P}_2\ \mathfrak{P}_1\ \varphi)(\mathfrak{D}(x_1, x_2, \ldots, x_n)) = \varphi(x_1, x_2, \ldots, x_n).$$

Por lo tanto, obtenemos el siguiente resultado.

TEOREMA 3.4.E. *Las funciones unarias generales tienen el mismo poder expresivo que las funciones recursivas generales.*

COROLARIO 3.4.F. *Cualquier función recursiva general puede representarse con una función unaria general, en donde la inversión ocurre sólo una vez en su fórmula.*

Demostración. En virtud del teorema de Kleene[Kle36a] (conocido como *Normal form theorem*), cualquier FRG puede normalizarse a la forma $\phi[\alpha, \mu\gamma]$ (donde $\alpha, \gamma \in \overline{\mathrm{FRP}}$). Si convertimos esta forma en una FUG, obtendremos la forma FG^- (donde $F, G \in \overline{\mathrm{FUP}}$). □

3.5. Notas sobre las funciones unarias generales

Es posible eliminar algunos constructores de $\overline{\mathrm{FUG}}$ sin perder poder de expresión. Supongamos, en esta sección, que las funciones iniciales distintor y raíz cuadrada,

$$\mathsf{D}(x) = 1 \dot{-} x, \quad \mathsf{Rt}(x) = \lfloor \sqrt{x} \rfloor,$$

y los esquemas de suma, resta, composición, e inversión conforman a $\overline{\mathrm{FUG}}$ (en otras palabras, supongamos que no contamos con Sq ni la iteración). A modo de ejemplo, veremos que las funciones que aparecen en la sección 3.2 también pertenecen a $\overline{\mathrm{FUG}}$. Ciertamente $\overline{\mathrm{FUP}} \dot{\subset} \overline{\mathrm{FUG}}$, pues se puede proponer $\overline{\mathrm{FUP}} \dot{\subset} \overline{\mathrm{FRP}} \dot{\subset} \overline{\mathrm{FRG}} \dot{\subset} \overline{\mathrm{FUG}}$, donde la contención de la derecha se prueba en la sección 3 de [Rob50], y las contenciones restantes son triviales.

EJEMPLO 3.5.A. *Antes de comenzar, sería necesario probar que* $\mathsf{Sq} \in \overline{\mathrm{FUG}}$. *Se puede ver fácilmente, usando el Teorema 3.4.B, que* Sq *es la inversa de* Rt *(i.e.* $\mathsf{Sq} \equiv \mathsf{Rt}^-$*). Si* $F(x) = \lfloor \sqrt{x} \rfloor$ *y* $G(x) = x^2$ *entonces* $F(G(x)) = x$ *y* $F(y') < x$ *para toda* $y' < G(x)$.
Explorando las fórmulas que se proponen en la sección 3.2, podemos observar que todas ellas se pueden escribir como FUGs, a excepción de Hf *que usa dos iteraciones.*
Recordemos que $\mathsf{E} \equiv \mathsf{RtSq} - \mathsf{SqRt}$ *y*

$$\mathsf{E}^-(x) = z^2 - 2z + x + 1,$$

donde $z = \left\lfloor \dfrac{x+3}{2} \right\rfloor$ *(véase el ejemplo 3.4.C). Si formamos la ecuación cuadrática*

$$z^2 - 2z + (x + 1 - \mathsf{E}^-(x)) = 0,$$

encontraremos la raíz positiva $z = 1 + \sqrt{\mathsf{E}^-(x) - x}$. *Por lo tanto,*

$$\left\lfloor \frac{x+4}{2} \right\rfloor = 1 + \sqrt{\mathsf{E}^-(x+1) - (x+1)},$$

$$\mathsf{Hf}(x) = \left\lfloor \frac{x+4}{2} \right\rfloor - 2 = \sqrt{\mathsf{E}^-(x+1) - (x+1)} - 1 = \mathsf{P}(\mathsf{Rt}(\mathsf{E}^-(\mathsf{S}(x)) - \mathsf{S}(x))).$$

Luego,

$$\mathsf{Hf} \equiv \mathsf{P}\ \mathsf{Rt}(\mathsf{E}^- - \mathsf{X})\mathsf{S}.$$

Analicemos, ahora, la causa de la elección de los constructores que forman las FUGs. Ya hemos probado que el distintor era necesario, y esto sigue siendo válido pues se puede extender el Teorema 3.1.B con el razonamiento *"Si* $F(0) = 0$ *entonces* $F^-(0) = 0$*"* que es trivial. Pero, no obstante, se puede sustituir D por U usando la fórmula[4]

$$\mathsf{D} \equiv \mathsf{U} - \mathsf{E}(\mathsf{Sq} + \mathsf{U}).$$

El hecho de que se puede sustituir D por cualquier función F que verifique $F(0) \neq 0$ ya fue tratado anteriormente, en la sección 3.1.

No solamente es posible cambiar las funciones iniciales, sinó también los constructores. Es suficiente contar con $\{\mathsf{U}, \mathsf{L}\}$, la suma, la composición y la inversión para obtener cualquier FUG: en primer lugar $\mathsf{S} \equiv \mathsf{LL}^- + \mathsf{U}$, $\mathsf{K} \equiv \mathsf{L}(\mathsf{LL}^- + \mathsf{SL} + \mathsf{SL})$ y $\mathsf{P} \equiv \mathsf{K}(\mathsf{KS})^-$. Entonces, según p. 711 [Rob50],

$$\begin{aligned} F - G &\equiv \mathsf{K}(\mathsf{U} + \mathsf{S}F + \mathsf{S}F + \mathsf{K}^-(F + G)), \\ \mathsf{D} &\equiv \mathsf{U} - (\mathsf{LL}^- - \mathsf{P}), \\ \mathsf{Sq} &\equiv (\mathsf{K}^- + \mathsf{K}^-) - (\mathsf{LL}^- + \mathsf{LL}^- + \mathsf{LL}^-), \\ \mathsf{Rt} &\equiv \mathsf{L}(\mathsf{Sq}\ \mathsf{L} + \mathsf{K})^-. \end{aligned}$$

[4]La función $\mathsf{E}(x^2 + 1)$ retorna cero cuando $x = 0$ (que se puede verificar trivialmente), y uno cuando $x > 0$. Para probar este último caso, $x^2 + 1$ es un valor acotado entre dos cuadrados perfectos (i.e. x^2 y $(x + 1)^2$). Luego, $\lfloor \sqrt{x^2 + 1} \rfloor = x$ y $\mathsf{E}(x^2 + 1) = x^2 + 1 - x^2 = 1$.

Capítulo 4

Reducciones sobre las funciones unarias primitivas

4.1. Introducción

En el capítulo anterior vimos que es posible representar cualquier función recursiva primitiva con tan sólo las funciones iniciales D, Sq y Rt, y los esquemas de suma, resta, composición e iteración. Lo que vamos a desarrollar a continuación es la posibilidad de sustituir los esquemas de suma y resta por el de distancia, y utilizar solamente la función inicial S. Para alcanzar este objetivo nos basaremos en los artículos [Gla71, Sev06] (una descripción mas profunda acerca de estos artículos se encuentra en la sección 4.5).

La única función inicial con la que contamos es

$$\mathsf{S}(x) = x + 1.$$

Además, dadas dos funciones F y G se definen los esquemas:

- ⋆ Distancia de F y G. Es una función que devuelve $\mathsf{Dist}(F(x), G(x))$. La notamos como $|F - G|$ y le corresponde la misma precedencia que la suma en las FUPs.
- ⋆ Composición de F y G. Es una función que devuelve $F(G(x))$. La notamos como (FG) y mantiene la misma precedencia que en las FUPs.
- ⋆ Iteración de F. Es una función que devuelve $F^{\square}(x)$ (definida en (3.1.1)). La notamos como $(F^{\square})$ y mantiene la misma precedencia que en las FUPs.

Nuestra meta es demostrar que, con este nuevo formalismo, se puede calcular el distintor, el cuadrado y la raíz cuadrada, y se pueden sumar o restar dos funciones. Uno de estos objetivos ya es trivial. De hecho, siempre que se verifique $F(x) \geq G(x)$, tenemos

$$F(x) - G(x) = |F(x) - G(x)|.$$

Es decir,

$$\boxed{F - G \equiv |F - G|}$$

En lo posible, vamos a intentar usar la resta en lugar de la distancia.

Ahora escribiremos las fórmulas de un grupo de funciones ya conocidas. Algunas de ellas ya fueron probadas, y otras se pueden deducir trivialmente.

$$\begin{aligned} \mathsf{Z} &\equiv \mathsf{S} - \mathsf{S}, \\ \mathsf{U} &\equiv \mathsf{SZ}, \\ \mathsf{X} &\equiv \mathsf{S} - \mathsf{U}, \\ \mathsf{Dbl} &\equiv (\mathsf{SS})^{\square}, \\ \mathsf{P} &\equiv \mathsf{X} - \mathsf{U}^{\square}, \end{aligned}$$

$$\boxed{\mathsf{D} \equiv \mathsf{U} - \mathsf{U}^{\square}}$$

También utilizaremos la potencia de un número (i.e. $\mathsf{Pot}(x) = 2^x$). Si $F(x) = 2x+1$ entonces, con una sencilla inducción, $F^{\square}(x) = 2^x - 1$. De manera que

$$\mathsf{Pot} \equiv \mathsf{S}(\mathsf{S}\ \mathsf{Dbl})^{\square}.$$

Ahora es posible efectuar $F + G$. En primer lugar, sean x e y números naturales tales que $x > y$, entonces

$$|2^x - 2^y| = 2^x - 2^y \geq 2^{x-1} = \frac{2^x}{2}.$$

Análogamente, si $x < y$, entonces

$$|2^x - 2^y| = 2^y - 2^x \geq 2^{y-1} = \frac{2^y}{2}.$$

De estas dos inecuaciones resulta

$$|2^x - 2^y| \geq \frac{2^{\max(x,y)}}{2}, \tag{4.1.1}$$

siempre que $x \neq y$. Consideremos $\vartheta(x, y) = |2^{2x+1} - 2^{2y+2}|$. En virtud de (4.1.1),

$$\vartheta(x, y) \geq \frac{2^{\max(2x+1,2y+2)}}{2} \geq 2^{2\max(x,y)} \geq 2\max(x, y) \geq x + y.$$

Luego $x + y = \vartheta(x, y) - ((\vartheta(x, y) - x) - y)$ y

$$F(x) + G(x) = |2^{2F(x)+1} - 2^{2(G(x)+1)}| - ((|2^{2F(x)+1} - 2^{2(G(x)+1)}| - F(x)) - G(x)).$$

En notación formal,

$$\vartheta(F, G) \equiv |\mathsf{Pot}\ \mathsf{S}\ \mathsf{Dbl}\ F - \mathsf{Pot}\ \mathsf{Dbl}\ \mathsf{S}\ G|,$$

$$\boxed{F + G \equiv \vartheta(F, G) - ((\vartheta(F, G) - F) - G)}$$

4.2. La función Q

Q es la función característica de los cuadrados perfectos. Su definición es

$$\mathsf{Q}(x) = \begin{cases} 1 & \text{si } x \text{ es un cuadrado perfecto,} \\ 0 & \text{si no.} \end{cases}$$

A la función Q, le dedicaremos una sección entera. Con esta función será sencillo hallar Rt y Sq (que son las que nos quedan pendientes). Para llegar a representar Q formalmente, deberemos definir otras funciones que se muestran a continuación.

Función 1. Comenzaremos definiendo al *distintor genérico* que nos permite distinguir algún número $n \geq 1$ en particular (para $n = 0$ disponemos de D),

$$\mathsf{D}_n(x) = \begin{cases} 1 & \text{si } x = n, \\ 0 & \text{si no.} \end{cases}$$

En efecto,

$$\mathsf{D}_1 \equiv \mathsf{D}(\mathsf{D} + \mathsf{P}),$$
$$\mathsf{D}_{n+1} \equiv \mathsf{D}_n\mathsf{P}.$$

Observemos que, para el caso base,

$$\mathsf{D}_1(0) = \mathsf{D}(\mathsf{D}(0) + \mathsf{P}(0)) = \mathsf{D}(1) = 0,$$
$$\mathsf{D}_1(1) = \mathsf{D}(\mathsf{D}(1) + \mathsf{P}(1)) = \mathsf{D}(0) = 1,$$
$$\mathsf{D}_1(x+2) = \mathsf{D}(\mathsf{D}(x+2) + \mathsf{P}(x+2)) = \mathsf{D}(x+1) = 0,$$

y que, para el caso inductivo, $\mathsf{D}_{n+1}(0) = \mathsf{D}_n(0) = 0$ y $\mathsf{D}_{n+1}(x+1) = \mathsf{D}_n(x)$.
Otra fórmula que también satisface la definición es $\mathsf{D}_n \equiv \mathsf{D}|\mathsf{X} - n|$, pero recordemos que íbamos a intentar evitar el uso de la distancia.

Función 2. Sea $n \geq 2$ un valor constante. Otra función interesante que emplearemos es $\mathsf{Mul}_n(x) = nx$. Simplemente la definimos de manera análoga a Dbl (usamos el exponente para indicar que S se compone varias veces, por ejemplo $\mathsf{S}^3 \equiv \mathsf{SSS}$),

$$\mathsf{Mul}_n \equiv (\mathsf{S}^n)^{\square}.$$

Notemos que si $F(x) = x + n$, se deduce $F^{\square}(x) = nx$. Para $x = 0$, $F^{\square}$ se anula, y para el caso inductivo tenemos que $F^{\square}(x+1) = F(F^{\square}(x)) = F(nx) = nx + n = n(x+1)$.

Función 3. Ahora vamos a representar la función

$$\mathsf{Circle}_n(x) = \begin{cases} x+1 & \text{si } x < n-1, \\ 0 & \text{si no.} \end{cases}$$

Es fácil darse cuenta de que $\mathsf{Circle}_n(x)$ genera la secuencia $1, 2, \ldots, n-1, 0, 0, \ldots$ así que se puede proponer $\mathsf{Circle}_n(x) = \mathsf{D}(x) + 2\mathsf{D}_1(x) + 3\mathsf{D}_2(x) + \ldots + (n-1)\mathsf{D}_{n-2}(x)$. Por lo tanto,

$$\mathsf{Circle}_2 \equiv \mathsf{D},$$
$$\mathsf{Circle}_{n+1} \equiv \mathsf{Circle}_n + \mathsf{Mul}_n\mathsf{D}_{n-1}.$$

Función 4. La función que nos permite calcular el resto de la división por n es Mod_n (i.e. $\mathsf{Mod}_n(x) = x \bmod n$), y la podemos calcular iterando Circle_n. Esta técnica se ha utilizado en la demostración de Hf en el Teorema 3.2.A.

$$\mathsf{Mod}_n(0) = 0,$$
$$\mathsf{Mod}_n(x+1) = \mathsf{Circle}_n(\mathsf{Mod}_n(x)),$$
$$\mathsf{Mod}_n \equiv \mathsf{Circle}_n^{\square}.$$

En particular, $\mathsf{Alt} \equiv \mathsf{Mod}_2$.

Función 5. Sea $F(x) = x + \lfloor x/n \rfloor$. Notemos que satisface la recurrencia

$$F(x+1) = x + 1 + \left\lfloor \frac{x+1}{n} \right\rfloor = \begin{cases} x + \lfloor x/n \rfloor + 2 & \text{si } (x + \lfloor x/n \rfloor + 2) \bmod (n+1) = 0, \\ x + \lfloor x/n \rfloor + 1 & \text{si no.} \end{cases}$$

El caso inductivo se puede escribir como

$$F(x+1) = F(x) + 1 + \mathsf{D}(\mathsf{Mod}_{n+1}(F(x)+2)).$$

Además $F(0) = 0$, de manera que es posible calcularlo con una iteración,

$$F \equiv (\mathsf{S} + \mathsf{D}\ \mathsf{Mod}_{n+1}\ \mathsf{SS})^{\square}. \tag{4.2.1}$$

Sea $\mathsf{Div}_n(x) = \lfloor \frac{x}{n} \rfloor$. Con F resulta sencillo, pues $\mathsf{Div}_n(x) = F(x) - x$. En virtud de la ecuación (4.2.1) obtenemos

$$\mathsf{Div}_n \equiv (\mathsf{S} + \mathsf{D}\ \mathsf{Mod}_{n+1}\ \mathsf{SS})^{\square} - \mathsf{X}.$$

En particular, $\mathsf{Hf} \equiv \mathsf{Div}_2$.

Función 6. Esta vez vamos a construir un operador, muy útil para hacer funciones con guardas. Este operador toma dos funciones F y G, y genera la función

$$(F \to G)(x) = \begin{cases} G(x) & \text{si } F(x) = 0, \\ 0 & \text{si no.} \end{cases}$$

- ⋆ Sea $A(x) = 2^{x+1+\mathsf{Alt}(x)} - 2^{x+1}$. Cuando x es par, $A(x) = 2^{x+1} - 2^{x+1} = 0$. Y cuando x es impar, $A(x) = 2^{x+2} - 2^{x+1} = 2^{x+1}(2-1) = 2^{x+1}$. Es decir,

$$A \equiv \mathsf{Pot}(\mathsf{S} + \mathsf{Alt}) - \mathsf{Pot}\ \mathsf{S}.$$

- ⋆ Sea $B(x) = |A(x) - (x + 2^x)|$. Cuando x es par, $B(x) = (x + 2^x) - 0 = 2^x + x$. Y cuando x es impar, $B(x) = 2^{x+1} - (x + 2^x) = 2^x - x$. Es decir,

$$B \equiv |A - (\mathsf{X} + \mathsf{Pot})|.$$

- ⋆ Sea $C(x) = (B(x) + x) - 2^x$. Cuando x es par, $C(x) = 2^x + x + x - 2^x = 2x$. Y cuando x es impar, $C(x) = 2^x - x + x - 2^x = 0$. Es decir,

$$C \equiv (B + \mathsf{X}) - \mathsf{Pot}.$$

Ahora, observemos el comportamiento de $C(2G(x) + \mathsf{D}(\mathsf{D}(F(x))))$. Si $F(x) = 0$,

$$C(2G(x) + \mathsf{D}(\mathsf{D}(F(x)))) = C(2G(x)) = 4G(x).$$

Si, en cambio, $F(x) > 0$,

$$C(2G(x) + \mathsf{D}(\mathsf{D}(F(x)))) = C(2G(x) + 1) = 0.$$

Luego[1],

$$(F \to G)(x) = \frac{C(2G(x) + \mathsf{D}(\mathsf{D}(F(x))))}{4},$$
$$(F \to G) \equiv \mathsf{Hf}\ \mathsf{Hf}((|(\mathsf{Pot}(\mathsf{S} + \mathsf{Alt}) - \mathsf{Pot}\ \mathsf{S}) - (\mathsf{X} + \mathsf{Pot})| + \mathsf{X}) - \mathsf{Pot})(\mathsf{Dbl}G + \mathsf{DD}F).$$

Función 7. Tal vez esta sea la función más difícil de analizar. La detallaremos a continuación.

$$\mathsf{W}(x) = \begin{cases} 2 & \text{si } x = 0, \\ \lfloor 3x/2 \rfloor & \text{si } x \neq 0, \text{ es par y múltiplo de 5}, \\ \lfloor 2x/5 \rfloor & \text{si } x \neq 0, \text{ es impar y múltiplo de 5}, \\ \lfloor 2x/3 \rfloor & \text{si } x \neq 0, \text{ es múltiplo de 3 pero no de 5}, \\ \lfloor 15x/2 \rfloor & \text{si } x \neq 0, \text{ no es múltiplo de 3 ni de 5}. \end{cases}$$

Aunque la función original fue ideada por Gladstone (p. 664 [Gla71]), nosotros vamos a usar la variante dada por Georgieva (p. 129 [Geo76b]). Asumamos que la entrada es $x = 2^a 3^b 5^c$ (de hecho, la entrada es una terna (a, b, c) codificada, con $a, b, c \in \mathbb{N}$). La evaluación $y = \mathsf{W}(x)$ está sometida a las siguientes reglas.

(a) Si $a > 0$ y $c > 0$ entonces $y = 2^{a-1}3^{b+1}5^c$.

(b) Si $a = 0$ y $c > 0$ entonces $y = 2^{a+1}3^b5^{c-1}$.

(c) Si $b > 0$ y $c = 0$ entonces $y = 2^{a+1}3^{b-1}5^c$.

(d) Si $b = 0$ y $c = 0$ entonces $y = 2^{a-1}3^{b+1}5^{c+1}$.

Y, si iteramos W, obtenemos la sucesión $0, 2^13^05^0, 2^03^15^1, 2^13^15^0, 2^23^05^0, 2^13^15^1, 2^03^25^1, 2^13^25^0, 2^23^15^0, 2^33^05^0$ que, en general, tiene el comportamiento que se muestra a continuación.

Iteración	Salida	Regla aplicada
0	0	def. $\square$
1	$2^13^05^0$	$x = 0$
2	$2^03^15^1$	(d)
3	$2^13^15^0$	(b)
4	$2^23^05^0$	(c)
$\cdots$	$\cdots$	$\cdots$

Para valores mayores, el patrón formado es

[1] Si, en vez de Dist debiéramos usar Mon, se simplifica la fórmula para C, pues $C(x) = 2x \stackrel{\bullet}{-} A(x)$.

Iteración	Salida	Regla aplicada	Comentario
n^2	$2^n3^05^0$	(c)	1. Cuadrado perfecto
n^2+1	$2^{n-1}3^15^1$	(d)	
n^2+2	$2^{n-2}3^25^1$	(a)	
$\cdots$	$\cdots$	$\cdots$	
n^2+n-1	$2^13^{n-1}5^1$	(a)	
n^2+n	$2^03^n5^1$	(a)	2. Número triangular por 2
n^2+n+1	$2^13^n5^0$	(b)	
n^2+n+2	$2^23^{n-1}5^0$	(c)	
$\cdots$	$\cdots$	$\cdots$	
n^2+2n	$2^n3^15^0$	(c)	
n^2+2n+1	$2^{n+1}3^05^0$	(c)	3. Próximo cuadrado (i.e. $n+1$)
n^2+2n+2	$2^n3^15^1$	(d)	
$\cdots$	$\cdots$	$\cdots$	

Entre (1) y (2), las potencias de 2 disminuyen (se decrementa a) y las de 3 aumentan (se incrementa b). Entre (2) y (3) el proceso se invierte. Notemos que la potencia de 5 actúa de *flag*: nos indica si estamos en la primer fase del proceso (entre (1) y (2) es $c=1$), o en la segunda (entre (2) y (3) es $c=0$). De la tabla anterior podemos inferir las propiedades

- ⋆ para todo $n>0$, $\mathsf{W}^{\square}(n^2)=2^n$,
- ⋆ para todo $x>0$ que no sea cuadrado perfecto, $\mathsf{W}^{\square}(x) \bmod 3 = 0$,

que serán útiles para generar Q.
Pasemos a su implementación. Para cada guarda, podemos realizar una función en particular, en donde la condición de cada guarda se evalúa a cero sólo cuando todos los sumandos que la componen se anulan. En particular, en la segunda guarda, la condición *es par y múltiplo de 5* se puede sustituir por *es múltiplo de 10*.

$$\begin{aligned}
\mathsf{W}_1(x) &= \mathsf{Dbl}(\mathsf{D}(x)),\\
\mathsf{W}_2(x) &= (\mathsf{D}(x)+\mathsf{Mod}_{10}(x) \to \mathsf{Hf}(\mathsf{Mul}_3(x))),\\
\mathsf{W}_3(x) &= (\mathsf{D}(x)+\mathsf{D}(\mathsf{Alt}(x))+\mathsf{Mod}_5(x) \to \mathsf{Div}_5(\mathsf{Dbl}(x))),\\
\mathsf{W}_4(x) &= (\mathsf{D}(x)+\mathsf{Mod}_3(x)+\mathsf{D}(\mathsf{Mod}_5(x)) \to \mathsf{Div}_3(\mathsf{Dbl}(x))),\\
\mathsf{W}_5(x) &= (\mathsf{D}(x)+\mathsf{D}(\mathsf{Mod}_3(x))+\mathsf{D}(\mathsf{Mod}_5(x)) \to \mathsf{Hf}(\mathsf{Mul}_{15}(x))).
\end{aligned}$$

Como las condiciones son mutuamente excluyentes, entonces

$$\mathsf{W}(x) = \mathsf{W}_1(x)+\mathsf{W}_2(x)+\mathsf{W}_3(x)+\mathsf{W}_4(x)+\mathsf{W}_5(x).$$

O, más formalmente,

$$\begin{aligned}
\mathsf{W} \equiv\ & \mathsf{Dbl}\ \mathsf{D} + (\mathsf{D}+\mathsf{Mod}_{10} \to \mathsf{Hf}\ \mathsf{Mul}_3)\\
&+ (\mathsf{D}+\mathsf{D}\ \mathsf{Alt}+\mathsf{Mod}_5 \to \mathsf{Div}_5\ \mathsf{Dbl})\\
&+ (\mathsf{D}+\mathsf{Mod}_3+\mathsf{D}\ \mathsf{Mod}_5 \to \mathsf{Div}_3\ \mathsf{Dbl})\\
&+ (\mathsf{D}+\mathsf{D}\ \mathsf{Mod}_3+\mathsf{D}\ \mathsf{Mod}_5 \to \mathsf{Hf}\ \mathsf{Mul}_{15}).
\end{aligned}$$

Función 8. Con la función W debería ser claro cómo generar Q. Para cualquier $x>0$, $\mathsf{W}^{\square}(x)$ es múltiplo de 3 si y sólo si x no es cuadrado perfecto. Así, $\mathsf{D}(\mathsf{D}(\mathsf{Mod}_3(\mathsf{W}^{\square}(x))))$ es la

función característica de los cuadrados perfectos. Solamente falta agregar el caso $x = 0$ (que también es cuadrado perfecto),

$$\mathsf{Q}(x) = \mathsf{D}(\mathsf{D}(\mathsf{Mod}_3(\mathsf{W}^{\square}(x)))) + \mathsf{D}(x),$$
$$\mathsf{Q} \equiv \mathsf{DD}\ \mathsf{Mod}_3\ \mathsf{W}^{\square} + \mathsf{D}.$$

4.3. Funciones Rt y Sq

Antes de obtener las funciones cuadrado y raíz cuadrada, vamos a definir una función $\mathsf{T}(x) = x + 2\lfloor\sqrt{x}\rfloor$. Observemos que esta función verifica la recurrencia

$$\mathsf{T}(x+1) = x + 1 + 2\lfloor\sqrt{x+1}\rfloor = x + 2\lfloor\sqrt{x}\rfloor + 1 + 2\mathsf{Q}(x+1) = \mathsf{T}(x) + 1 + 2\mathsf{Q}(x+1),$$

en donde se aplica la propiedad (1.5.1): $\lfloor\sqrt{x+1}\rfloor = \lfloor\sqrt{x}\rfloor + \mathsf{Q}(x+1)$.
Ahora, si $x+1$ es un cuadrado perfecto, existe un $y \in \mathbb{N}$ tal que $(y+1)^2 = x+1$. Entonces $x = y^2 + 2y$ y, en consecuencia,

$$\mathsf{T}(x) = (y^2 + 2y) + 2\lfloor\sqrt{(y^2+2y)}\rfloor = y^2 + 2y + 2y = y^2 + 4y.$$

Luego $\mathsf{T}(x)+4 = y^2+4y+4 = (y+2)^2$ es también un cuadrado perfecto. Y recíprocamente, si $\mathsf{T}(x)+4$ es un cuadrado perfecto, también lo es $x+1$. Por lo tanto,

$$\mathsf{Q}(x+1) = \mathsf{Q}(\mathsf{T}(x)+4),$$
$$\mathsf{T}(x+1) = \mathsf{T}(x) + 1 + 2\mathsf{Q}(\mathsf{T}(x)+4).$$

Como $\mathsf{T}(0) = 0$, podemos escribir T mediante una iteración,

$$\mathsf{T} \equiv (\mathsf{S} + \mathsf{Dbl}\ \mathsf{Q}\ \mathsf{SSSS})^{\square}.$$

Sea, ahora, $F(x) = \mathsf{T}(x) + 1 = x + 2\lfloor\sqrt{x}\rfloor + 1$. Vamos a probar por inducción que, efectivamente, $F^{\square}(x) = x^2$.
Si $x = 0$, entonces $F^{\square}(0) = 0 = 0^2$.
Supongamos que se verifica $F^{\square}(x) = x^2$. Para $x+1$ resulta

$$F^{\square}(x+1) = F(F^{\square}(x)) = F(x^2) = x^2 + 2\lfloor\sqrt{x^2}\rfloor + 1 = x^2 + 2x + 1 = (x+1)^2,$$

$$\boxed{\mathsf{Sq} \equiv (\mathsf{ST})^{\square}}$$

Finalmente, notemos que $\mathsf{T}(x) - x = 2\lfloor\sqrt{x}\rfloor$. Dividiendo por 2 ambos miembros obtenemos la función faltante,

$$\boxed{\mathsf{Rt} \equiv \mathsf{Hf}(\mathsf{T} - \mathsf{X})}$$

4.4. Otras bases

Para referirnos a los formalismos de una manera compacta, los simbolizaremos con un conjunto. Los elementos de este conjunto serán los constructores con los que parte el formalismo. En particular, $\overline{\mathrm{FUP}} = \{\mathsf{D}, \mathsf{Sq}, \mathsf{Rt}, F+G, F-G, FG, F^{\square}\}$. Y el formalismo que desarrollamos en este capítulo es $\{\mathsf{S}, |F-G|, FG, F^{\square}\}$. A estos conjuntos se los suele llamar *bases*. En todos los casos, F y G no denotarán ninguna función en particular. Sólo sirven

para mostrar los constructores de manera adecuada.

Base 1. En el capítulo anterior vimos que la base $\{\mathsf{S}, \mathsf{Rt}, F+G, F-G, FG, F^{\square}\}$ bastaba para generar todas las funciones unarias primitivas. Lo mismo ocurre con la base $\{\mathsf{S}, \mathsf{Sq}, F+G, F-G, FG, F^{\square}\}$ como lo veremos a continuación.
Las funciones Z, U, X, Dbl, D y P son obtenibles según la sección 4.1. Y las funciones D_n, Mul_n, Circle_n, Mod_n y Div_n (entre ellas, Hf) también son obtenibles según la sección 4.2. Solamente nos falta demostrar que Rt se puede generar.
Resulta posible anotar el producto entre dos funciones, como lo vimos en el Teorema 3.2.B,

$$\Pi \equiv \mathsf{Hf}(\mathsf{Sq}(F+G) - \mathsf{Sq}F - \mathsf{Sq}G).$$

Y el operador para construir guardas resulta trivial,

$$(F \to G) \equiv \Pi(\mathsf{D}F, G).$$

Luego, las funciones W, Q, T y Rt también son obtenibles.
Incluso, es posible eliminar la suma $F+G$ de los constructores, pero no lo desarrollaremos aquí. Se debe usar la sucesión B_n definida en (4.5.1) (véase lema 6.2 de [Sev06]).

Base 2. Vamos a exponer el formalismo original de Raphael Robinson (Teorema 3 de [Rob47]), en donde la base $\{\mathsf{S}, \mathsf{E}, F+G, FG, F^{\square}\}$ alcanza para generar todas las funciones unarias primitivas. Nuevamente, sólo necesitamos generar D, Sq, Rt y $F-G$. Para hallar el distintor, ni siquiera es necesaria la suma. En primer lugar, $\mathsf{X} \equiv \mathsf{S}^{\square}$, $\mathsf{Dbl} \equiv (\mathsf{SS})^{\square}$, $\mathsf{Z} \equiv \mathsf{X}^{\square}$ y $\mathsf{U} \equiv \mathsf{SZ}$. Vamos a probar que

$$\mathsf{D}(x) = \mathsf{E}(\mathsf{S}(\mathsf{S}(\mathsf{Dbl}(\mathsf{U}^{\square}(x)))))$$

es cierta. Para $x=0$, $\mathsf{U}^{\square}(x)=0$ y $\mathsf{D}(x)=\mathsf{E}(2)=1$. Y para $x>0$, $\mathsf{U}^{\square}(x)=1$ y $\mathsf{D}(x)=\mathsf{E}(4)=0$. Por lo tanto, $\mathsf{D} \equiv \mathsf{E\ S\ S\ Dbl\ U}^{\square}$.
La función característica de los cuadrados perfectos es $\mathsf{D}(\mathsf{E}(x))$ pues E se anula cuando x es un cuadrado perfecto. Así que $\mathsf{Q} \equiv \mathsf{DE}$. Según la sección 4.3, las funciones T y Sq son obtenibles. Ahora, es factible escribir la resta

$$F(x) - G(x) = \mathsf{E}(\mathsf{S}((F(x)+G(x))^2 + 3F(x) + G(x))).$$

Consideremos $y = F(x)$ y $z = G(x)$. Si $y \geq z$, la cuenta $(y+z)^2 + 3y + z + 1$ da un valor que está, ciertamente, entre dos cuadrados perfectos (i.e. $(y+z+1)^2$ y $(y+z+2)^2$),

$$(y+z)^2 + 2y + 2z + 1 \leq (y+z)^2 + 3y + z + 1 \leq (y+z)^2 + 4y + 4z + 4.$$

Luego $\lfloor \sqrt{(y+z)^2 + 3y + z + 1} \rfloor = y + z + 1$, resultando

$$\mathsf{E}((y+z)^2 + 3y + z + 1) = ((y+z)^2 + 3y + z + 1) - ((y+z)^2 + 2y + 2z + 1) = y - z.$$

Entonces, $F - G \equiv \mathsf{E\ S}(\mathsf{Sq}(F+G) + F + \mathsf{Dbl}\ F + G)$. Y de aquí podemos generar la raíz cuadrada, $\mathsf{Rt} \equiv \mathsf{Hf}(\mathsf{T} - \mathsf{X})$, a partir de la prueba de Hf dada en el Teorema 3.2.A (la cual, a su vez, requiere del predecesor $\mathsf{P} \equiv \mathsf{X} - \mathsf{U}^{\square}$).
En [Sev06] se halla una simplificación, pues se demuestra que con la base $\{\mathsf{S}, \mathsf{E}, F+\mathsf{X}, FG, F^{\square}\}$ es suficiente (nótese que la suma ahora es un operador unario que toma por argumento a F

solamente, dejando como único operador binario a la composición).

Base 3. En este capítulo vimos que $\{\mathsf{S}, |F-G|, FG, F^{\square}\}$ permite representar cualquier función unaria primitiva. Sin embargo, sería interesante eliminar la distancia, sustituyéndola por algún operador unario. Julia Robinson[Rob50] desarrolló algunas investigaciones con respecto a esta posibilidad. Sean J, K y L funciones de parificación arbitrarias. Definimos el operador *estrella* como

$$F^*(x) = J(L(x), F(K(x))).$$

El proceso para calcular la distancia entre dos funciones F y G dadas se basa en la siguiente estrategia de parificación:

$$x \xrightarrow{J(X,X)} J(x,x) \xrightarrow{F^* \equiv J(L,FK)} J(x,F(x)) \xrightarrow{G^* \equiv J(L,GK)} J(F(x),G(x)) \xrightarrow{|K-L|} |F(x)-G(x)|$$

Por lo tanto,

$$|F-G| \equiv |K-L| G^* F^* J(X,X).$$

Luego, $\{\mathsf{S}, J(X,X), |K-L|, F^*, FG, F^{\square}\}$ consta de un solo operador binario: la composición. Debería ser claro que cualquier operador binario $\oplus$ que se desee adjuntar al formalismo, lo puede sustituir una función inicial $K \oplus L$. Esta es una técnica para reducir todas las operaciones binarias a funciones inciales.

Base 4. Una reducción ulterior es minimizar la cantidad de funciones iniciales de una base a sólo dos, una de ellas K. El truco aquí es que con K y la función

$$M \equiv J(L, J(F_0, J(F_1, J(F_2, \ldots))))$$

podemos obtener las funciones originales F_0, F_1, F_2, etc. Es decir,

$$\begin{aligned}
L(x) &= K(J(L(x), \ldots)) = K(M(x)),\\
F_0(x) &= K(L(J(L(x), J(F_0(x), \ldots)))) = K(L(M(x))),\\
F_1(x) &= K(L(L(J(L(x), J(F_0(x), J(F_1(x), \ldots)))))) = K(L(L(M(x)))),\\
F_2(x) &= K(L(L(L(J(L(x), J(F_0(x), J(F_1(x), J(F_2(x), \ldots))))))))) = K(L(L(L(M(x))))),\\
&\ldots
\end{aligned}$$

Nosotros recurriremos a una técnica similar. Partamos de la base 3 y parifiquemos las funciones iniciales usando

$$M \equiv J(\mathsf{S}, J(J(X,X), |K-L|)).$$

Luego,

$$\begin{aligned}
L \equiv KK^* &\equiv K\ J(L, KK),\\
\mathsf{S} &\equiv KM,\\
J(X,X) &\equiv KLM,\\
|K-L| &\equiv LLM.
\end{aligned}$$

Y la base $\{K, M, F^*, FG, F^{\square}\}$ también es factible.

Base 5. A pesar de ser sencilla la base 4, la función M resulta algo complicada de calcular. Podemos beneficiarnos de las propiedades de algún grupo de funciones de parificación particular para simplificar las funciones iniciales. Recordemos que en la sección 3.1 investigamos,

muy resumidamente, que la base $\{\mathsf{S}, \mathsf{K}, \mathsf{J}(F,G), FG, F^{\square}\}$ era suficiente para representar a las FUPs. Aquí, propondremos $\{\mathsf{S}, \mathsf{K}, F^{\dagger}, FG, F^{\square}\}$ como base, probando que $\mathsf{J}(F,G)$ puede deducirse. El operador

$$F^{\dagger}(x) = \mathsf{J}(\mathsf{L}(x), F(x))$$

es más simple que el operador estrella, y éste último puede obtenerse del anterior,

$$F^{*} \equiv \mathsf{J}(\mathsf{L}, F\mathsf{K}) \equiv (F\mathsf{K})^{\dagger}.$$

Exploremos, ahora, la siguiente estrategia:

$$x \xrightarrow{\mathsf{X}^{\dagger} \equiv \mathsf{J}(\mathsf{L},\mathsf{X})} \mathsf{J}(\mathsf{L}(x), x) \xrightarrow{\mathsf{L}^{\dagger} \equiv \mathsf{J}(\mathsf{L},\mathsf{L})} \mathsf{J}(x,x) \xrightarrow{(F\mathsf{K})^{\dagger}} \mathsf{J}(x, F(x)) \xrightarrow{(G\mathsf{K})^{\dagger}} \mathsf{J}(F(x), G(x))$$

Resulta entonces,

$$\mathsf{J}(F,G) \equiv (G\mathsf{K})^{\dagger}(F\mathsf{K})^{\dagger}\mathsf{L}^{\dagger}\mathsf{X}^{\dagger},$$

donde $\mathsf{X} \equiv \mathsf{S}^{\square}$ y $\mathsf{L} \equiv \mathsf{K}\mathsf{S}^{\dagger}$.

Notas. Los operadores F^{*} y $F^{\dagger}$ tienen la misma precedencia que la iteración. También hay que remarcar que Julia Robinson, en su trabajo, utilizó un operador estrella ligeramente distinto: $F^{*} \equiv \mathsf{J}(F\mathsf{K}, \mathsf{L})$. Ella, además, redujo el operador estrella a ciertas funciones iniciales, composición e iteración.

4.5. Breve reseña histórica

En algunos resultados aparece la siguiente iteración más genérica (donde la notación $F^{\square(a)}$ se debe a Szalkai[Sza85]),

$$F^{\square(a)}(x) = \begin{cases} a & \text{si } x = 0, \\ F(F^{\square(a)}(x-1)) & \text{si } x > 0. \end{cases}$$

En particular, $F^{\square} \equiv F^{\square(0)}$. No obstante, podemos calcular esta iteración si somos capaces de hallar[2]

$$G(x) = \begin{cases} a+1 & \text{si } x = 0, \\ F(x-1)+1 & \text{si } x > 0. \end{cases}$$

Luego, resulta $F^{\square(a)}(x) = G^{\square}(x+1) - 1$.

Péter[Pét34, Pét35] introdujo, por primera vez, métodos para la eliminación de parámetros y reducción a un esquema similar al de la iteración utilizando propiedades de los números primos. Inspirados en ellos, el prestigioso matemático R. Robinson[Rob47] tabuló distintos esquemas de recursión (entre ellos el de la iteración) y formuló métodos de reducción. El demostró, entre otros resultados, que $\{\mathsf{S}, \mathsf{E}, F+G, FG, F^{\square}\}$ y $\{\mathsf{S}, \mathsf{Q}, |F-G|, FG, F^{\square}\}$ tienen el mismo poder de expresión que las FRPs.

Posteriormente, su esposa J. Robinson[Rob50] formuló métodos de reducción para las funciones recursivas generales, inspirada en dos artículos escritos por Kleene[Kle36a, Kle36b]. Ella probó que $\{\mathsf{S}, \mathsf{E}, F+G, FG, F^{-}\}$, $\{\mathsf{S}, \mathsf{K}, F+G, FG, F^{-}\}$ y $\{\mathsf{S}, \mathsf{L}, F+G, FG, F^{-}\}$[3] tienen

[2]Una posibilidad es que $G \equiv \Pi(a+1, \mathsf{D}) + \Pi(\mathsf{S}F\mathsf{P}, \mathsf{D}\mathsf{D})$ y $F^{\square(a)} \equiv \mathsf{P}G^{\square}\mathsf{S}$.

[3]Nosotros hemos introducido, brevemente, una base muy similar (intercambiando U por S, aprovechando que $\mathsf{S} \equiv \mathsf{L}\mathsf{L}^{-} + \mathsf{U}$) en la sección 3.5.

el mismo poder de expresión que las FRGs. También probó que se puede eliminar la suma si se cambian las funciones iniciales por otras, una de ellas un tanto complicada,

$$M \equiv \mathsf{J}(\mathsf{L}, \mathsf{J}(\mathsf{J}(\mathsf{K} + \mathsf{KL}, \mathsf{X}), \mathsf{J}(\mathsf{J}(\mathsf{LK}, \mathsf{KL}), \mathsf{J}(\mathsf{X}, \mathsf{U})))).$$

La base $\{\mathsf{K}, M, FG, F^{-}\}$ también tiene el mismo poder de expresión que las FRGs. Para cerrar el tema, ella probó, a su vez, que no es posible obtener todas las FRGs con sólo una función inicial, la composición y la inversión.

Luego, R. Robinson[Rob55b] publicó un artículo donde prueba que $\{\mathsf{S}, \mathsf{K}, F+G, FG, F^{\square}\}$ $\{\mathsf{S}, \mathsf{L}, F+G, FG, F^{\square}\}$, $\{\mathsf{S}, \mathsf{K}, \mathsf{J}(F,G), FG, F^{\square}\}$ y $\{\mathsf{S}, \mathsf{L}, \mathsf{J}(F,G), FG, F^{\square}\}$ tienen el mismo poder de expresión que las FRPs. En ese mismo año, J. Robinson[Rob55a] publicó una prueba de que $\{\mathsf{K}, M, FG, F^{\square}\}$ tiene el mismo poder expresivo que las FRPs, donde

$$M \equiv \mathsf{J}(\mathsf{L}, \mathsf{J}(\mathsf{J}(\mathsf{K} + \mathsf{KL}, \mathsf{X}), \mathsf{J}(\mathsf{J}(\mathsf{LK}, \mathsf{KL}), \mathsf{J}(\mathsf{J}(\mathsf{X}, \mathsf{U}), \mathsf{J}(\mathsf{DD\Pi}(\mathsf{KL}, \mathsf{K}), \mathsf{L}))))).$$

Para finalizar, J. Robinson probó (al igual que con las FRGs) que no es posible obtener todas las FRPs con sólo una función inicial, la composición y la iteración más genérica $F^{\square(a)}$ (este hecho fue posteriormente simplificado por Szalkai[Sza85], donde descubre ciertas propiedades algebraicas de bases que contengan $\{FG, F^{\square(a)}\}$).

Pocos avances se realizaron luego: Gladstone[Gla67] simplificó más uno de los esquemas de recursión, pero la contribución mas importante fue su artículo [Gla71] donde, entre otras cosas, demuestra que $\{\mathsf{S}, |F-G|, FG, F^{\square(a)}\}$ es suficiente para representar a las FRPs. Y N. Georgieva[Geo76a] demuestra que $\{\mathsf{S}, F \dot{-} G, FG, F^{\square(a)}\}$ también es suficiente. Ella construye una sucesión creciente de funciones $f_k(x)$ (llamadas, por lo general, funciones de Ackermann) tal que para dos funciones dadas F y G se crean funciones f_k y f_l que las mayoren (i.e. $f_k(x) \geq F(x)$ por ejemplo). Luego,

$$F(x) + G(x) \leq f_k(x) + f_l(x) \leq 2f_m(x),$$

donde $m = \max(k, l)$. Entonces, mediante un truco similar al visto cuando se definió ϑ, es posible definir la suma y la distancia (aquí $\mathsf{Dbl} \equiv (\mathsf{SS})^{\square}$),

$$\begin{aligned} F + G &\equiv \mathsf{Dbl} f_m \dot{-} ((\mathsf{Dbl} f_m \dot{-} F) \dot{-} G), \\ |F - G| &\equiv (F \dot{-} G) + (G \dot{-} F), \end{aligned}$$

y aplicar el formalismo de Gladstone. Un resumen de funciones generadas a partir del formalismo de Georgieva se puede ubicar en [Nau83] en el Lema 1.

Una mejora a estos esquemas se muestra en [Sev06], pues se utiliza la iteración más simple. Aquí se prueba que $\{\mathsf{S}, |F-G|, FG, F^{\square}\}$ y $\{\mathsf{S}, F \dot{-} G, FG, F^{\square}\}$ son suficientes, entre otras cosas. Para generar la suma, se usa una sucesión que mayora a la sucesión de Ackermann (i.e. $B_n(x+1) \geq f_n(x)$), y es de la forma

$$\begin{cases} B_0 & \equiv \mathsf{S}, \\ B_{n+1} & \equiv (\mathsf{S}^{f_n(1)}\ B_n)^{\square}. \end{cases} \tag{4.5.1}$$

Ahora se puede formular $F + G \equiv \mathsf{Dbl} B_m \mathsf{S} \dot{-} ((\mathsf{Dbl} B_m \mathsf{S} \dot{-} F) \dot{-} G)$.

Paralelamente al desarrollo de las reducciones, también se definieron varias clasificaciones de las funciones recursivas primitivas. Generalmente, las FRPs se clasifican según la cantidad de veces que se anida el esquema de recursión primitiva. Son particularmente conocidas

las jerarquías $\mathscr{E}^n$ y $\mathscr{F}^n$ de Grzegorczyk[Grz53] (donde, entre otros resultados, ha probado que $\mathscr{E}^n \dot{\subset} \mathscr{E}^{n+1}$ estrictamente, y que $\bigcup_{n\in\mathbb{N}} \mathscr{E}^n \doteq \overline{\mathrm{FRP}}$), $\mathscr{K}^n$ de Axt[Axt65, Axt66], $\mathscr{G}^n$ de Ritchie[Rit65] (quien, además, probó que $\mathscr{F}^n \doteq \mathscr{E}^n$), $\mathscr{L}^n$ de Georgieva[Geo76b] y $\mathscr{I}^n$ de Naumović[Nau83]. Para todas ellas se cumple la *propiedad de Darboux* (conocida en análisis matemático) como se puede ver en los artículos presentados por Calude[Cal81, Cal82, CT83].

Un problema interesante es el que aparece en el *Scottish Book*[Mau81] como Problema 143 (cuyos autores son R. Robinson y S. Mazur). La idea es probar si se puede generar la función delta de Kronecker a partir de ciertos operadores que generan funciones binarias. Al final la respuesta es satisfactoria, pues tales operadores pueden generar cualquier función recursiva primitiva binaria.

Capítulo 5

Caracterización de las funciones recursivas con funciones generadoras

5.1. Funciones generadoras para las funciones recursivas primitivas

Sabemos bien que las funciones generadoras pueden caracterizar sucesiones de números naturales. Pero, por otra parte, una función natural f define una sucesión

$$f(0), f(1), f(2), \ldots$$

así que es una buena idea definir funciones generadoras que representen funciones naturales.

DEFINICIÓN 5.1.A. *Sea $f : \mathbb{N} \to \mathbb{N}$ una función natural total. Definimos las funciones generadoras de f como las funciones generadoras de la sucesión $a_n = f(n)$ donde $n \in \mathbb{N}$. Respectivamente,*

- ⋆ *la* función generadora tradicional *de f es*

$$\mathfrak{S}_f(z) = \sum_{n=0}^{\infty} f(n) z^n,$$

- ⋆ *la* función generadora exponencial *de f es*

$$\mathfrak{S}_f^E(z) = \sum_{n=0}^{\infty} \frac{f(n) z^n}{n!},$$

- ⋆ *la* función generadora de Dirichlet *de f es*

$$\mathfrak{S}_f^D(z) = \sum_{n=0}^{\infty} \frac{f(n)}{(n+1)^z}.$$

En la definición se resumen los tipos de funciones generadoras más conocidos, aunque sabemos que se pueden elegir otras bases $K_n(z)$ (i.e. funciones *kernel*) para las funciones generadoras $\sum_{n=0}^{\infty} f(n) K_n(z)$ siempre que la función generadora nula sea la única que representa

a la función $f(n) = 0$.

En el capítulo 3 vimos que las funciones unarias primitivas pueden ser generadas a través de la sucesiva aplicación de ciertos constructores (como la suma y composición) sobre ciertas funciones iniciales (como el distintor y el cuadrado). El desafío que propondremos en este capítulo es caracterizar las FUPs con funciones generadoras. Es decir, hallar $\mathfrak{S}_f$ para las funciones iniciales f, y $\mathfrak{S}_{F\otimes G}$ a partir de conocer $\mathfrak{S}_F$ y $\mathfrak{S}_G$ (aquí $\otimes$ representa un constructor binario).
En el resto del capítulo trabajaremos sólo con funciones generadoras tradicionales, dada su sencillez. Estas funciones generadoras presentan, sin embargo, algunas limitaciones. Cuando $\mathfrak{S}_f(z)$ diverge para toda z (exceptuando $z = 0$ donde converge trivialmente), esta función no ofrece demasiadas posibilidades para investigarla usando métodos analíticos. Esto ocurre, por lo general, cuando la función que se representa crece rápidamente (e.g. $f(x) = x!$).
Para conocer más sobre funciones generadoras, se puede consultar [Knu80] y [GKP97]. Un estudio completo sobre las funciones generadoras lo hace [Wil90].

5.2. Ejemplos de funciones generadoras tradicionales

Una función generadora puede caracterizar una función a valores naturales por el simple hecho de que existe una técnica para recuperar los valores de ésta última. Sea $F : \mathbb{N} \to \mathbb{N}$,

$$\begin{aligned} F(0) &= \mathfrak{S}_F(0), \\ F(1) &= \frac{\partial}{\partial z}(\mathfrak{S}_F)(0), \\ F(2) &= \frac{1}{2}\frac{\partial^2}{\partial z^2}(\mathfrak{S}_F)(0), \\ &\cdots \\ F(n) &= \frac{1}{n!}\frac{\partial^n}{\partial z^n}(\mathfrak{S}_F)(0). \end{aligned}$$

Será útil definir el operador

$$\begin{aligned} \_[[\_]] &: \mathcal{F} \times \mathbb{N} \to \mathcal{F}, \\ \mathfrak{S}_F[[n]] &= \frac{1}{n!}\frac{\partial^n}{\partial z^n}(\mathfrak{S}_F), \end{aligned}$$

donde $\mathcal{F}$ representa el espacio de funciones generadoras que caracterizan funciones naturales. El operador $\_[[\_]]$ estará bien definido siempre que todas las derivadas que aparezcan tengan sentido, y su acción será efectuar un corrimiento en la función que se caracteriza. En otras palabras $\mathfrak{S}_F[[n]](z) = \mathfrak{S}_G(z)$, donde $G(x) = F(x + n)$ para todo $x \in \mathbb{N}$. En particular, se verifica $\mathfrak{S}_F[[n]](0) = \mathfrak{S}_G(0) = G(0) = F(n)$. El operador puede ser calculado recursivamente con

$$\begin{aligned} \mathfrak{S}_F[[0]] &= \mathfrak{S}_F, \\ \mathfrak{S}_F[[n+1]] &= \frac{1}{n+1}\frac{\partial}{\partial z}(\mathfrak{S}_F[[n]]). \end{aligned}$$

Las funciones generadoras permiten también representar esquemas de sumas y restas muy fácilmente. En efecto, sean F y G dos funciones naturales. Entonces

$$\mathfrak{S}_{F+G}(z) = \mathfrak{S}_F(z) + \mathfrak{S}_G(z).$$

Si, además $F(x) \geq G(x)$ para toda x, entonces

$$\mathfrak{S}_{F-G}(z) = \mathfrak{S}_F(z) - \mathfrak{S}_G(z).$$

Esta es una de las razones por la cual hemos fomentado el uso de la resta, en vez de la resta truncada o la distancia.

A continuación mostraremos una tabla de funciones generadoras para algunas de las funciones unarias primitivas que hemos definido. Todas ellas se pueden deducir usando conceptos elementales de análisis matemático.

FUP	Función generadora	Forma cerrada
Z	$\sum_{n=0}^{\infty} 0z^n$	0
U	$\sum_{n=0}^{\infty} 1z^n$	$\frac{1}{1-z}$
X	$\sum_{n=0}^{\infty} nz^n$	$\frac{z}{(1-z)^2}$
S	$\sum_{n=0}^{\infty} (n+1)z^n$	$\frac{1}{(1-z)^2}$
Dbl	$\sum_{n=0}^{\infty} 2nz^n$	$\frac{2z}{(1-z)^2}$
D	$\sum_{n=0}^{\infty} \mathsf{D}(n)z^n$	1
Sq	$\sum_{n=0}^{\infty} n^2z^n$	$\frac{z^2+z}{(1-z)^3}$
Pot	$\sum_{n=0}^{\infty} 2^nz^n$	$\frac{1}{1-2z}$
P	$\sum_{n=0}^{\infty} (n \dot{-} 1)z^n$	$\frac{z^2}{(1-z)^2}$
Alt	$\sum_{n=0}^{\infty} (n \bmod 2)z^n$	$\frac{1}{1-z} - \frac{1}{1-z^2}$
Hf	$\sum_{n=0}^{\infty} \lfloor \frac{n}{2} \rfloor z^n$	$\frac{1}{2}\left[\frac{2z-1}{(1-z)^2} + \frac{1}{1-z^2}\right]$

Ahora estudiemos la función Q (i.e. la función característica de los cuadrados perfectos). La función generadora de Dirichlet para QS tiene una forma cerrada, a saber

$$\mathfrak{S}^D_{\mathsf{QS}}(z) = \sum_{n=0}^{\infty} \frac{\mathsf{Q}(n+1)}{(n+1)^z} = \frac{1}{1^z} + \frac{1}{4^z} + \frac{1}{9^z} + \ldots = \sum_{n=1}^{\infty} \frac{1}{(n^2)^z} = \sum_{n=1}^{\infty} \frac{1}{n^{2z}} = \zeta(2z),$$

donde ζ es la función zeta de Riemann, $\zeta(z) = \sum_{n=1}^{\infty} \frac{1}{n^z}$. Lamentablemente, la función generadora tradicional de Q (o QS que es una variante) no posee forma cerrada en términos de

funciones trascendentes elementales (de la misma manera que la integral $\int e^{x^2}dx$ no tiene primitiva[1]),

$$\mathfrak{S}_{\mathsf{Q}}(z) = \sum_{n=0}^{\infty} \mathsf{Q}(n)z^n = 1 + z + z^4 + z^9 + \ldots = \sum_{n=0}^{\infty} z^{(n^2)}.$$

Este problema se propaga a otras funciones que se puedan escribir en términos de Q. Por ejemplo, podemos aprovechar una propiedad[2] de las funciones generadoras para obtener $\mathfrak{S}_{\mathsf{Rt}}$,

$$\mathfrak{S}_{\mathsf{Rt}}(z) = \sum_{n=0}^{\infty}\left[\sum_{t=0}^{n} \mathsf{Q}(t) - 1\right]z^n = \sum_{n=0}^{\infty}\left[\sum_{t=0}^{n} \mathsf{Q}(t)\right]z^n - \sum_{n=0}^{\infty} z^n = \frac{\mathfrak{S}_{\mathsf{Q}}(z) - 1}{1-z},$$

pues $\mathsf{Rt}(x) = \mathsf{Q}(0) + \mathsf{Q}(1) + \ldots + \mathsf{Q}(x) - 1$.

5.3. Composición usando funciones generadoras tradicionales

En esta sección nos proponemos estudiar la función generadora de la composición entre funciones. Sean $\mathfrak{S}_F$ y $\mathfrak{S}_G$ las funciones generadoras de dos funciones naturales F y G, entonces

$$\mathfrak{S}_{FG}(z) = \sum_{n=0}^{\infty} F(G(n))z^n = \sum_{n=0}^{\infty}(\mathfrak{S}_F)[[\mathfrak{S}_G[[n]](0)]](0).$$

Sería interesante formular una forma cerrada para la expresión anterior. Hasta el momento (al menos de lo que he investigado), **no se conoce ninguna**, y posiblemente no haya ninguna que sea lo suficientemente cerrada como para ser útil. Pero no nos limitaremos a este hecho y trataremos de escribir algunos casos particulares.

⋆ Cuando se conoce F: Enumeraremos algunas funciones generadoras formadas a partir de componer una F en particular con una G arbitraria.

$$\begin{aligned}
\mathfrak{S}_{\mathsf{X}G}(z) &= \mathfrak{S}_G(z),\\
\mathfrak{S}_{\mathsf{S}G}(z) &= \mathfrak{S}_G(z) + \frac{1}{1-z},\\
\mathfrak{S}_{\mathsf{Z}G}(z) &= 0,\\
\mathfrak{S}_{\mathsf{U}G}(z) &= \frac{1}{1-z},\\
\mathfrak{S}_{\mathsf{Dbl}G}(z) &= 2\mathfrak{S}_G(z).
\end{aligned}$$

Más generalmente, si $F(x) = ax + b$ entonces $\mathfrak{S}_{FG}(z) = a\mathfrak{S}_G(z) + \dfrac{b}{1-z}$.

⋆ Cuando se conoce G: Enumeraremos algunas funciones generadoras formadas a partir de componer una F arbitraria con una G en particular. En los lugares donde aparezca

[1] Sin embargo, se puede extender el conjunto de funciones elementales para incluir nuevas integrales. En este caso, $\mathrm{erfi}(x) = \int e^{x^2}dx$ es conocida como *función de error imaginario*. No se descarta la posibilidad de que $\mathfrak{S}_{\mathsf{Q}}$ pueda ser escrita con estas funciones.

[2] Véase igualdad (7.21) en p. 334 [GKP97]. También, igualdad (7) en p. 94 [Knu80].

$\mathfrak{S}_F[[1]](z)$ podemos reemplazarlo por $\dfrac{\partial}{\partial z}(\mathfrak{S}_F)(z)$.

$$\begin{aligned}
\mathfrak{S}_{F\mathsf{X}}(z) &= \mathfrak{S}_F(z),\\
\mathfrak{S}_{F\mathsf{S}}(z) &= \frac{\mathfrak{S}_F(z)-\mathfrak{S}_F(0)}{z},\\
\mathfrak{S}_{F\mathsf{Z}}(z) &= \mathfrak{S}_F(0),\\
\mathfrak{S}_{F\mathsf{U}}(z) &= \frac{\mathfrak{S}_F[[1]](0)}{1-z},\\
\mathfrak{S}_{F\mathsf{Dbl}}(z) &= \frac{\mathfrak{S}_F(\sqrt{z})+\mathfrak{S}_F(-\sqrt{z})}{2},\\
\mathfrak{S}_{F\mathsf{D}}(z) &= \mathfrak{S}_F[[1]](0)+\frac{z\mathfrak{S}_F(0)}{1-z},\\
\mathfrak{S}_{F\mathsf{Alt}}(z) &= \frac{1}{1-z^2}\mathfrak{S}_F(0)+\left[\frac{1}{1-z}-\frac{1}{1-z^2}\right]\mathfrak{S}_F[[1]](0).
\end{aligned}$$

Ciertamente, es posible escribir $\mathfrak{S}_{FG}$ cuando $G(x)=nx+b$, pero hay que introducir números complejos cuando $n\geq 3$ (precisamente, raíces n-ésimas de la unidad, véase p. 95 [Knu80].).

Ejemplo 5.3.A. *Estos casos tienen aplicación cuando deseamos escribir funciones, aparentemente complicadas, pero que se pueden descomponer en otras funciones más simples. Sea* $H(x)=3(2x \overset{\bullet}{-} 1)+5$*, la cual resulta de componer* $F(x)=3x+5$*,* P *y* Dbl*. Entonces,*

$$\mathfrak{S}_H(z)=\mathfrak{S}_{F\mathsf{PDbl}}(z)=3\mathfrak{S}_{\mathsf{PDbl}}(z)+\frac{5}{1-z}=\frac{3}{2}\left[\mathfrak{S}_{\mathsf{P}}(\sqrt{z})+\mathfrak{S}_{\mathsf{P}}(-\sqrt{z})\right]+\frac{5}{1-z}.$$

5.4. Conclusión

En lo que sigue vamos a suponer que conocemos **una forma cerrada para representar las funciones generadoras de la composición e iteración de funciones unarias primitivas**. Sean $f,g\in\mathcal{F}$, notaremos con $\mathcal{C}[f;g]$ a la forma cerrada de la composición y $\mathcal{I}[f]$ a la forma cerrada de la iteración.
Podemos construir una función generadora para cada FUP, utilizando la base 1 de la sección 4.4, $\{\mathsf{S},\mathsf{Sq},F+G,F-G,FG,F^{\square}\}$:

$$\begin{aligned}
\mathfrak{S}_{\mathsf{S}}(z) &= \frac{1}{(1-z)^2}, & \mathfrak{S}_{\mathsf{Sq}}(z) &= \frac{z^2+z}{(1-z)^3},\\
\mathfrak{S}_{F+G}(z) &= \mathfrak{S}_F(z)+\mathfrak{S}_G(z), & \mathfrak{S}_{F-G}(z) &= \mathfrak{S}_F(z)-\mathfrak{S}_G(z),\\
\mathfrak{S}_{FG}(z) &= \mathcal{C}[\mathfrak{S}_F;\mathfrak{S}_G], & \mathfrak{S}_{F^{\square}}(z) &= \mathcal{I}[\mathfrak{S}_F].
\end{aligned}$$

Ejemplo 5.4.A. *Vamos a obtener una función generadora para la función identidad* X*. Esta función es unaria primitiva, pues se puede escribir (a partir de la base 1) como*

$$\mathsf{X}\equiv\mathsf{S}-\mathsf{S}(\mathsf{S}-\mathsf{S}).$$

Resulta entonces,

$$\begin{aligned}
\mathfrak{S}_{\mathsf{X}}(z)&=\mathfrak{S}_{\mathsf{S}-\mathsf{S}(\mathsf{S}-\mathsf{S})}(z)=\mathfrak{S}_{\mathsf{S}}(z)-\mathfrak{S}_{\mathsf{S}(\mathsf{S}-\mathsf{S})}(z)=\mathfrak{S}_{\mathsf{S}}(z)-\left(\mathfrak{S}_{\mathsf{S}-\mathsf{S}}(z)+\frac{1}{1-z}\right)\\
&=\mathfrak{S}_{\mathsf{S}}(z)-\left(\mathfrak{S}_{\mathsf{S}}(z)-\mathfrak{S}_{\mathsf{S}}(z)+\frac{1}{1-z}\right)=\frac{1}{(1-z)^2}-\frac{1}{1-z}=\frac{z}{(1-z)^2}.
\end{aligned}$$

Cabe aclarar que, en algún momento, $\mathcal{C}[f;g]$ y $\mathcal{I}[f]$ introducirán nuevas funciones en las formas cerradas que generen. Estas nuevas funciones deberán ser consideradas como elementales. Sabemos que este fenómeno ocurrirá, pues $\mathfrak{S}_{\mathbb{Q}}$ no puede escribirse en términos de funciones trascendentes elementales como vimos en la sección 5.2.

En resumen, podemos concluir que, cuando se conozcan $\mathcal{C}[f;g]$ y $\mathcal{I}[f]$ (si es que esto resulta posible), tendremos un algoritmo para construir funciones generadoras que permitan caracterizar cualquier función unaria primitiva, y por lo tanto, cualquier función recursiva primitiva. Otra area que queda por investigar es ver la factibilidad de poder caracterizar funciones parciales con funciones generadoras y, de ser aplicable, hallar una forma cerrada para la inversión. Es decir, $\mathfrak{S}_{F^-} = \mathcal{V}[\mathfrak{S}_F]$ donde $\mathcal{V}[f]$ es una forma cerrada para la inversión. Conociendo $\mathcal{V}[f]$, será posible construir funciones generadoras que permitan caracterizar cualquier función unaria general, y por lo tanto, cualquier función recursiva general. Bajo una adecuada representación, **una función generadora podría caracterizar cualquier función Turing-computable**.

Bibliografía

[Ack28] Wilhelm Ackermann. Zum Hilbertschen Aufbau der reellen Zahlen. *Math. Ann.*, 99:118–133, 1928.

[Axt65] Paul Axt. Iteration of primitive recursion. *Z. Math. Logik Grundlagen Math.*, 11:253–255, 1965.

[Axt66] Paul Axt. Iteration of relative primitive recursion. *Math. Ann.*, 167:53–55, 1966.

[Bir00] Richard Bird. *Introducción a la Programación Funcional con Haskell. Segunda Edición.* 2000.

[Bro93] J. G. Brookshear. *Teoría de la Computación: Lenguajes formales, autómatas y complejidad.* 1993.

[Cal81] Cristian Calude. Darboux property and primitive recursive functions. *Rev. Roumaine Math. Pures Appl.*, 26(9):1187–1192, 1981.

[Cal82] Cristian Calude. Note on a hierarchy of primitive recursive functions. *Rev. Roumaine Math. Pures Appl.*, 27(9):935–936, 1982.

[CMT79] Cristian Calude, Solomon Marcus, and Ionel Tevy. The first example of a recursive function which is not primitive recursive. *Historia Math.*, 9:380–384, 1979.

[CT83] Cristian Calude and Monica Tatarâm. Universal sequences of primitive recursive functions. *Rev. Roumaine Math. Pures Appl.*, 28(5):381–389, 1983.

[DSW94] Martin D. Davis, Ron Sigal, and Elaine J. Weyuker. *Computability, Complexity, and Languages (2^{nd} Ed.): Fundamentals of theoretical computer science.* 1994.

[Geo76a] Nadejda Georgieva. Another simplification of the recursion scheme. *Arch. Math. Logik*, 18:1–3, 1976.

[Geo76b] Nadejda Georgieva. Classes of one-argument recursive functions. *Z. Math. Logik Grundlagen Math.*, 22:127–130, 1976.

[GKP97] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A foundation for computer science.* 1997.

[Gla67] M. D. Gladstone. A reduction of the recursion scheme. *J. Symbolic Logic*, 32(4):505–508, 1967.

[Gla71] M. D. Gladstone. Simplifications of the recursion scheme. *J. Symbolic Logic*, 36(4):653–665, 1971.

[GM88] Giorgio Germano and Stefano Mazzanti. Primitive iteration and unary functions. *Ann. Pure Appl. Logic*, 40:217–256, 1988.

[GM91] Giorgio Germano and Stefano Mazzanti. General iteration and unary functions. *Ann. Pure Appl. Logic*, 54:137–178, 1991.

[Goo57] Reuben L. Goodstein. *Recursive Number Theory: A development of recursive arithmetic in a logic-free equation calculus*. 1957.

[Grz53] Andrzej Grzegorczyk. Some classes of recursive functions. *Rozprawy Matematyczne*, 4:1–44, 1953.

[Hil26] David Hilbert. Über das Unendliche. *Math. Ann.*, 95:161–190, 1926.

[Hof03] Douglas R. Hofstadter. *Gödel, Escher, Bach: Un Eterno y Grácil Bucle (8^{va} Ed.)*. 2003.

[Kle36a] Stephen C. Kleene. General recursive functions of natural numbers. *Math. Ann.*, 112:727–742, 1936.

[Kle36b] Stephen C. Kleene. A note on recursive functions. *Bull. Amer. Math. Soc.*, 42:544–546, 1936.

[Kle52] Stephen C. Kleene. *Introduction to Metamathematics*. 1952.

[Knu80] Donald E. Knuth. *El Arte de Programar Ordenadores I: Algoritmos Fundamentales*. 1980.

[Mau81] R. Mauldin. *The Scottish Problem Book*. 1981.

[Maz97] Stefano Mazzanti. Iterative characterizations of computable unary functions: a general method. *MLQ Math. Log. Q.*, 43:29–38, 1997.

[Maz05] Stefano Mazzanti. Bounded iteration and unary functions. *MLQ Math. Log. Q.*, 51:89–94, 2005.

[MR67] A. R. Meyer and D. M. Ritchie. The complexity of loop programs. In *Proceedings of the 22^{nd} National Conference*, pages 465–469. Thompson Book Company, 1967.

[Nau83] Jovan Naumović. A classification of the one-argument primitive recursive functions. *Arch. Math. Logik*, 23:161–174, 1983.

[Pét34] Rózsa Péter. Über den Zusammenhang der verschiedenen Begriffe der rekursiven Funktion. *Math. Ann.*, 110:612–632, 1934.

[Pét35] Rózsa Péter. Konstruktion nichtrekursiver Funktionen. *Math. Ann.*, 111:42–60, 1935.

[Pét36] Rózsa Péter. Über die mehrfache Rekursion. *Math. Ann.*, 113:489–527, 1936.

[Rit65] Robert W. Ritchie. Classes of recursive functions based on Ackermann's function. *Pacific J. Math.*, 15(3):1027–1044, 1965.

[Rob47] Raphael M. Robinson. Primitive recursive functions. *Bull. Amer. Math. Soc.*, 53(10):925–942, 1947.

[Rob48] Raphael M. Robinson. Recursion and double recursion. *Bull. Amer. Math. Soc.*, 54(10):987–993, 1948.

[Rob50] Julia Robinson. General recursive functions. *Proc. Amer. Math. Soc.*, 1(6):703–718, 1950.

[Rob55a] Julia Robinson. A note on primitive recursive functions. *Proc. Amer. Math. Soc.*, 6(4):667–670, 1955.

[Rob55b] Raphael M. Robinson. Primitive recursive functions II. *Proc. Amer. Math. Soc.*, 6(4):663–666, 1955.

[Sev06] Daniel E. Severin. Unary primitive recursive functions. *Manuscript, http://arxiv.org/abs/cs/0603063*, pages 1–16, 2006.

[Sud27] Gabriel Sudan. Sur le nombre transfini ω^ω. *Bull. Math. Soc. Sci. Math. Roumanie*, 30:11–30, 1927.

[Sza85] István Szalkai. On the algebraic structure of primitive recursive functions. *Z. Math. Logik Grundlagen Math.*, 31:551–556, 1985.

[Tat03] Monica Tatarâm. An extension of the double recursion theorem of R. M. Robinson. *Rev. Roumaine Math. Pures Appl.*, 48(1):105–111, 2003.

[Wil90] Herbert S. Wilf. *Generating functionology.* 1990.