MATEMÁTICA COMPUTABLE

ANTONIO MONTALBÁN

Introducción

Este artículo está basado en un curso de tres días dado durante el Segundo Coloquio Uruguayo de Matemática en Diciembre del 2009. El objetivo del curso, y de este artículo, es transmitir las ideas básicas de la teoría de la computabilidad y la matemática computable. Todos sabemos que en matemática hay construcciones o demostraciones que son más complicadas que otras. Matemática computable es el área de la teoría de la computabilidad donde uno estudia la complejidad de construcciones, procesos, demostraciones y estructuras matemáticas. Hay varias formas de medir la complejidad de una construcción. La que usaremos en este artículo se basa en la idea de que un proceso que es computable, es decir uno que puede ser efectuado de forma totalmente mecánica, es un proceso *simple*, y que los procesos más complejos son los que no pueden ser efectuados por computadores. Esta noción es adecuada para estudiar la complejidad de objetos infinitos. En algunas situaciones, uno puede estar interesado en otras nociones de complejidad, como por ejemplo en ciencias de la computación donde los procesos que son considerados simples son los que son computables en tiempo polinomial.

Este artículo esta dividido en tres secciones. En la primera desarrollamos los conceptos básicos de la teoría de la computabilidad. En las siguientes secciones estudiamos la complejidad de construcciones concretas en combinatoria y en álgebra. En la segunda sección estudiaremos problemas relacionados al problema de la detención, y en la tercera sección estudiaremos el lema de König sobre caminos en árboles binarios.

Hemos incluido varias demostraciones, pero no de todos los resultados que mencionamos. Además, muchas de las demostraciones omiten varios detalles que dejamos para que el lector verifique.

1. Computabilidad

La noción de *algoritmo* ya era conocida por los griegos cuando se preguntaban si ciertas construcciones geométricas, o de teoría de números, pueden hacerse mecánicamente. Un ejemplo de algoritmo, ya usado por lo griegos hace más de 2000 años, es el algoritmo de Euclides para calcular el máximo común divisor de dos números. Un algoritmo no es más que una receta de cocina. Es simplemente un método claramente determinado por una serie de instrucciones para calcular algo. Esta idea intuitiva es clara para la mayoría de los matemáticos: todos reconocemos un algoritmo cuando lo vemos.

Para poder trabajar con esta idea intuitiva deberíamos explicitar claramente a qué nos referimos con "un algoritmo totalmente mecánico." Hay varias definiciones posibles, y la mayoría son un poco técnicas, por lo cual no nos detendremos en detalles. Recomendamos al lector que desea ver una definición formal consultar los primeros capítulos de [Soa87] o [AK00]. Para los que ya conocen algún lenguaje de programación, imaginen que por algoritmo nos referimos a cualquier programa que pueda escribirse en ese lenguaje, y que tenemos tanto tiempo y memoria como sea necesario para correr el programa. La elección del lenguaje de programación a

[0] Saved: 3 - Dic. - 2010
Compiled: 10 de enero de 2011

usar no es importante, ya que todos son equivalentes desde nuestro punto de vista. La definición más usada en los cursos de complejidad se basa en el lenguaje de las máquinas de Turing. Éstas son máquinas muy simples, y son universales en el sentido que pueden hacer lo mismo que cualquier otra máquina mecánica. El que puedan hacer los mismos cálculos que cualquier otra máquina mecánica es la conocida *tesis de Church-Turing*. En la década del 30, Turing propuso una justificación para su tesis, pero la tesis no se puede demostrar rigurosamente, ya que no tenemos una definición rigurosa de "máquina mecánica." Actualmente, la tesis de Church-Turing es globalmente aceptada.

1.1. Conjuntos computables.

Definición 1.1. Un conjunto $A \subseteq \mathbb{N}$ es *computable* si existe un algoritmo totalmente mecánico que, dado $n \in \mathbb{N}$, decide si $n \in A$ o no. Osea, el algoritmo recibe n como entrada, y produce una salida de 1 o 0, dependiendo de si $n \in A$ o no.

Ejemplo 1.2. Los siguientes conjuntos son computables:

- El conjunto de los números pares;
- El conjunto de los números primos;
- El conjunto de los programas escritos correctamente de acuerdo a las reglas del lenguaje.

¿Por qué nos restringimos solamente a subconjuntos de $\mathbb{N}$? Porque es suficiente. Todo objeto finito puede ser codificado con un número natural. Y nos referimos a cualquier tipo de objeto finito, como un grafo, un complejo simplicial, un grupo, una representación finita de un grupo infinito, etc. La codificación pude ser hecha de varias maneras, y es irrelevante cual método se usa en la codificación. Todos sabemos que las computadoras modernas codifican todas los objetos que manejan en código binario: el mismo método serviría, cualquiera que sea.

Ahora presentamos tres ejemplos de conjuntos no computables.

1.1.1. El problema de la palabra. Consideremos el conjunto de grupos que pueden ser definidos usando un conjunto finito G de generadores y un conjunto finito R de relaciones entre los generadores. Por ejemplo, si tenemos dos generadores $G = \{a, b\}$, y la relación $R = \{aba^{-1}b^{-1} = 1\}$, tenemos el grupo $\mathbb{Z}^2$. El conjunto de pares (G, R) de generadores-relaciones que determinan un grupo no-trivial (i.e. con más de un elemento) no es computable. La razón es que para saber si un elemento del grupo (y en particular un generador) es equivalente a la identidad, uno tiene que aplicar las relaciones dadas de alguna manera hasta llegar a 1, pero no hay forma de anticipar el número de aplicaciones que uno puede llegar a necesitar.

1.1.2. Variedades simplemente conexas. Este problema esta directamente conectado con el anterior. Como ya dijimos, cada complejo simplicial finito se puede representar con un número natural. El conjunto de lo números que representan complejos simpliciales simplemente conexos no es computable. Si un complejo simplicial es simplemente conexo, entonces en un número finito de pasos podemos encontrar las homotopías de todos las curvas generadoras con la identidad. Pero si no, no hay forma de saber que hemos revisado todas la homotopías posibles.

1.1.3. El décimo problema de Hilbert. El conjunto de los polinomios con coeficientes enteros, y en varias variables, que tienen raíces enteras no es computable. Este era el décimo problema de la famosa lista que Hilbert propuso en el año 1900, antes de que existiese una noción formal de que significa ser computable. Este problema fue resulto en 1970 por Matiyasevich, usando resultados previos de M. Davis, Putnam y Robinson.

1.2. Funciones parciales computables. Todo programa define una función parcial. Por lo tanto, nos sera útil estudiar las funciones parciales que son computables.

Definición 1.3. Todo programa de computadora (i.e. algoritmo totalmente mecánico) p, cuya entrada y salida es un número natural, representa a una función parcial $p\colon \mathbb{N} \rightharpoonup \mathbb{N}$, donde el valor de $p(n)$ puede estar indefinido en caso que el programa p con entrada n no se detenga nunca. Nótese que hay programas que entran en loops infinitos, o que se quedan haciendo cálculos para siempre, y nunca se detienen. En este caso escribimos $p(n)\uparrow$. Si p, con entrada n, sí se detiene con salida m, luego, obviamente $p(n) = m$. En este caso escribimos $p(n)\downarrow$, o $p(n)\downarrow = m$, o simplemente $p(n) = m$.

Una función parcial $f\colon \mathbb{N} \rightharpoonup \mathbb{N}$ (o sea una función que puede estar indefinida en algunos valores) es una *función parcial computable* si está asociada a un programa p como en el parágrafo anterior. Cuando nos referimos a *función computable*, nos referiremos a una función *total* (i.e. definida en todos los valores), que es computable.

Como los programas son secuencias finitas de caracteres, podemos ordenarlos en una lista (digamos en orden lexicográfico). Sea $\varphi_0, \varphi_1, \varphi_2, \ldots.$ una enumeración de todos los programas que uno puede escribir que tienen como entrada y como salida un número natural. A cada programa φ_e asociamos la función parcial $\varphi_e\colon \mathbb{N} \rightharpoonup \mathbb{N}$. Como la enumeración de los programas es totalmente efectiva, podemos escribir un programa p, cuya entrada son dos números naturales, tal que $p(e, n) = \varphi_e(n)$. Osea que p primero busca el e-ésimo programa de la lista, y luego lo corre con entrada n. Usando una biyección computable entre $\mathbb{N} \times \mathbb{N}$ y $\mathbb{N}$, (como por ejemplo $\frac{1}{2}((n+m)^2 + 3n + m)$), podemos suponer que la entrada de p es sólo un número natural y no dos.

1.2.1. El problema de la detención. El ejemplo más usado de conjunto no computable es el problema de la detención.

Definición 1.4. Llamamos el *problema de la detención* al conjunto K de los programas que, con entrada 0, se detienen en algún momento.

$$K = \{e \in \mathbb{N} : \varphi_e(0)\downarrow\}.$$

Teorema 1.5. *El problema de la detención no es computable.*

La única manera mecánica de saber si un programa eventualmente se detiene o no es correrlo. El problema es que si el programa nunca se detiene, nunca lo sabremos.

Demostración. Supongamos que K es computable, y que el programa $p\colon \mathbb{N} \to \{0, 1\}$ es tal que $p(e) = 1$ si y sólo si el e-ésimo programa φ_e con entrada 0 se detiene eventualmente. Consideremos el siguiente programa q:

> Con entrada i, primero buscamos un programa φ_e tal que $\varphi_e(n) = \varphi_i(i+n)$, y luego si $p(e) = 0$, paramos el programa $q(i)$ dando alguna respuesta (digamos $q(i) = 0$), y si $p(e) = 1$, dejamos que $q(i)$ entre en un loop infinito. Nótese que $p(e) = 1$ si y sólo si $\varphi_i(i)$ se detiene. Osea que $q(i)$ se detiene si y sólo si $\varphi_i(i)$ no se detiene.

Como q es un programa, existe e tal que $\varphi_e = q$. Luego, por definición de q, $\varphi_e(e)$ se detiene si y sólo si $\varphi_e(e)$ no se detiene. Esta contradicción muestra que K no puede ser computable. □

1.3. Computabilidad relativa. Sea B un conjunto, preferiblemente no computable, al que llamamos *oráculo*. Supongamos ahora que cuando queremos computar un conjunto o una función, podemos usar B en nuestro algoritmo. Es decir, dentro del algoritmo tenemos una función, que asumimos como primitiva, tal que dado n nos responde si $n \in B$ o no, y

dependiendo de la respuesta continuamos el algoritmo de una forma u otra. Ésto nos permite computar una cantidad mayor de conjuntos.

Definición 1.6. Dados $A, B \subseteq \mathbb{N}$, decimos que A *es computable en* B (y escribimos $A \leq_T B$) si hay un programa que, con entrada n y usando a B como oráculo, responde si $n \in A$ o no. Decimos que A y B son *Turing equivalentes* (y escribimos $A \equiv_T B$) si $A \leq_T B$ y $B \leq_T A$.

Ejemplo 1.7. Los siguientes conjuntos son Turing-equivalentes.

- El problema de la detención;
- El problema de la palabra;
- El conjunto de complejos simpliciales finitos simplemente conexos;
- El conjunto de los polinomios con coeficientes enteros, y en varias variables, que tienen raíces enteras.

Luego, por ejemplo, llamémosle T al conjunto de los pares (G, R) de generadores y relaciones que generan grupos de torsión. Este conjunto es mayor, en el sentido de $\leq_T$, que todos estos conjuntos recién mencionados, pero no puede ser computado en ninguno de ellos. El conjunto (que llamamos $Th(\mathbb{N})$) de todas las sentencias de lógica de primer orden que son verdaderas sobre los números naturales es aún $\leq_T$-mayor.

Definición 1.8. Las clases de equivalencia de la relación $\equiv_T$ se llaman *grados de Turing*. Nótese que los grados de Turing forman un orden parcial, al ser ordenados por $\leq_T$.

$$\mathbf{D} = \frac{2^{\mathbb{N}}}{\equiv_T} \quad \text{y} \quad \mathcal{D} = (\mathbf{D}, \leq_T).$$

Para cada $A \in 2^{\mathbb{N}}$, el conjunto $\{B \in 2^{\mathbb{N}} : B \leq_T A\}$ es numerable, porque hay una cantidad numerable de programas que uno puede escribir. Por lo tanto cada clase de equivalencia es numerable, y $\mathbf{D}$ tiene tantas clases de equivalencia como $\mathbb{R}$ tiene números reales. La segunda observación importante sobre $\mathcal{D}$ es que tiene un grado que es $\leq_T$-menor que todos los demás. Este grado, que denotamos 0, es la clase de equivalencia de los conjuntos computables: los conjuntos computables son Turing-equivalentes entre ellos y son computables en cualquier otro conjunto.

Entender la forma y las propiedades del orden parcial de los grados de Turing es uno de los temas de investigación de la Teoría de la Computabilidad. En los años 60 se buscaba caracterizar este orden parcial como el único orden con alguna propiedad de densidad o alguna otra propiedad. Luego de varios años, todos los intentos de encontrar una caracterización de esa forma fueron fallando. En los años 80 hubo un cambio de dirección, y en vez de buscar alguna propiedad que muestre la simpleza de los grados de Turing, se empezaron a buscar propiedades que muestren que este orden parcial es muy complicado, lo más complicado que puede llegar a ser.

1.4. Conjuntos computablemente enumerables.

Una clase importante de conjuntos, es la de los conjuntos computablemente enumerables.

Definición 1.9. Un conjunto $A \subseteq \mathbb{N}$ es *computablemente enumerables (c.e.)* si existe una función computable $f : \mathbb{N} \to \mathbb{N}$ cuya imagen es A (i.e. $A = \{f(0), f(1), f(2), ...\}$). El conjunto vacío también es considerado un conjunto c.e. a pesar de no cumplir esta condición.

Los conjuntos c.e. no tienen por que ser computables. Por ejemplo, el problema de la detención es c.e. ya que podemos correr todos los programas, una cantidad finita a la vez, y cada vez que encontramos uno que se detiene lo enumeramos. Nótese que cuando enumeramos un conjunto $A = \{f(0), f(1), f(2), ...\}$, no tenemos porque hacerlo en orden creciente.

Teorema 1.10. *Sea $A \subseteq \mathbb{N}$. Los siguientes enunciados son equivalentes.*

- *A es c.e.*
- *Existe un programa φ tal que para $n \in \mathbb{N}$, $n \in A$ si y sólo si φ se detiene con entrada n.*
- *Existe un programa φ tal que para $n \in \mathbb{N}$, $n \in A$ si y sólo si existe m tal que $\varphi(m)$ se detiene y $\varphi(m) = n$.*

Teorema 1.11. *$A \subseteq \mathbb{N}$ es computable si y sólo si A y $\mathbb{N} \setminus A$ son ambos c.e. (donde $\mathbb{N} \setminus A$ es el complemento de A).*

Esta demostraciones quedan como ejercicio para el lector.

El problema de la detención K no es un conjunto c.e. cualquiera: es universal entre los conjuntos c.e.

Teorema 1.12. *Para todo conjunto c.e. A existe una función computable $f\colon \mathbb{N} \to \mathbb{N}$ tal que $\forall n \in \mathbb{N}$,*

$$n \in A \iff f(n) \in K.$$

Por lo tanto, todo conjunto c.e. es computable en K.

1.5. El salto de Turing. La definición del problema de la detención puede hacerse relativa a cualquier oráculo.

Definición 1.13. Dado $B \subseteq \mathbb{N}$, definimos B' como el conjunto de índices e de los programas que, con entrada 0 y oráculo B, se detienen. Si denotamos como φ_e^B el e-ésimo programa con oráculo B, y usamos $\varphi_e^B(n) \downarrow$ para decir que φ_e^B con entrada n se detiene, tenemos que

$$B' = \{e \in \mathbb{N} : \varphi_e^B(0) \downarrow\}.$$

B' es llamado el *salto de Turing* de B.

Teorema 1.14. *Para todos $A, B \subseteq \mathbb{N}$,*

- *$A \leq_T B$ implica $A' \leq_T B'$.*
- *$B <_T B'$.*

La demostración de este teorema también queda como ejercicio.

Con respecto a los ejemplos de 1.7, tenemos que $K \equiv_T 0'$, que $T \equiv_T 0''$, y que $Th(\mathbb{N}) >_T 0^{(n)}$ para todo n, donde $0^{(n)}$ es la n-ésima iteración del salto de Turing. De aquí en adelante usaremos $0'$ en lugar de K.

2. El problema de la detención en matemáticas

Ya mencionamos el problema de la detención en la Secciones 1.2.1 y 1.5. El aspecto más interesante de este problema es su complejidad. Conjuntos con la misma complejidad aparecen de forma natural en muchas áreas de matemática.

Teorema 2.1. *Sea $A \subseteq \mathbb{N}$. Los siguientes enunciados son equivalentes.*

1. *$0' \leq_T A$;*
2. *El oráculo A puede decidir si un grupo dado por (generadores, relaciones) es no-trivial;*
3. *El oráculo A puede decidir si un polinomio en $\mathbb{Z}[X_1, X_2, ...]$ tiene raíces en $\mathbb{Z}$;*
4. *El oráculo A puede computar la componente conexa de un vértice dado en cualquier grafo computable G;*
5. *Todo anillo computable tiene un ideal maximal computable en A;*
6. *Toda secuencia creciente acotada y computable de números racionales tiene limite cuya representación decimal es computable en A;*

Demostraremos la equivalencia entre (1), (4) y (5).

Como ya dijimos, el objetivo de la matemática computable es estudiar la complejidad de de las construcciones, procesos, demostraciones y estructuras que manejamos regularmente en matemática. Los resultados más interesantes son los que relacionan propiedades computacionales con propiedades estructurales o algebraicas.

La mejor forma de entender a que nos referimos con "estudiar la complejidad" de construcciones matemáticas es viendo ejemplos. Empezaremos estudiando algunas construcciones básicas sobre grafos y demostrando la equivalencia entre (1) y (4).

2.1. Grafos. Un grafo consiste de dos conjuntos (V, E) donde $E \subseteq V^2$. Los elementos de V se llaman *vértices*, y los de E *aristas*. Sólo vamos a considerar grafos no dirigidos (i.e. $(\forall u, v \in V)(u, v) \in E \to (v, u) \in E)$) y sin lazos (i.e. $(\forall v \in V)(v, v) \notin E$). Todos los grafos que mencionaremos son numerables infinitos, y como son numerables podemos suponer que $V \subseteq \mathbb{N}$. Decimos que un grafo es *computable* si los conjuntos $V \subseteq \mathbb{N}$ y $E \subseteq \mathbb{N}^2$ son computables.

El primer problema que estudiaremos es qué tan complicado es calcular la componente conexa de un vértice en un grafo infinito. Dado un vértice $v \in V$, la *componente conexa de* v es

$$C(v) = \{w \in V : \exists x_1, ..., x_k \in V \\ (v, x_0) \in E \ \& \ (x_0, x_1) \in E \ \& \ ... \ \& \ (x_{k-1}, x_k) \in E \ \& \ (x_k, w) \in E\}.$$

Supongamos que G es computable. ¿Es $C(v)$ computable? ¿Hay algún oráculo que calcule $C(v)$ para todo G computable?

Lema 2.2. *Para todo G computable y $v \in V$, $C(v) \leq_T 0'$.*

Demostración. Dado v, w y G, construiremos un programa φ, tal que, con entrada 0, el programa $\varphi(w)$ se detiene si y sólo si $w \in C(v)$. Si e es el índice de este programa (i.e. $\varphi = \varphi_e$), tendríamos que $w \in C(v) \iff e \in 0'$. Por lo que $C(v)$ seria computable en $0'$.

El programa $\varphi(0)$ hace lo siguiente: Una por una, enumera todas las tuplas $(x_1, ..., x_k)$ de vértices y chequea si forman un camino de v a w. Si sí, el programa para y no sigue enumerando más tuplas. Si no, el programa continua chequeando las siguientes tuplas indefinidamente. □

Lema 2.3. *Existe un grafo computable G y $v \in V$, tal que $0' \leq_T C(v)$.*

Antes de probar este lema, precisamos definir la siguiente notación: $\varphi_{e,s}(i) \downarrow$ significa que el programa φ_e con entrada i se detiene en menos de s pasos. Nótese que dados e, i, s podemos decidir si $\varphi_{e,s}(i) \downarrow$ automáticamente, ya que sólo necesitamos buscar el e-ésimo programa de la lista, correr φ_e por s pasos y ver si se detiene.

Demostración. Consideremos el siguiente grafo: $G = (V, E)$. Sea $V = \mathbb{N}$. Definimos E de la siguiente manera. Los números impares están todos por aristas: o sea $\forall n, m(2n+1, 2m+1) \in E$. No hay ninguna arista entre dos número pares: o sea $\forall n, m(2n, 2m) \notin E$. Entre los números pares y los impares tenemos las siguientes aristas:

$$(2n, 2m + 1) \in E \iff \varphi_{n,m}(0) \downarrow,$$

o sea si $\varphi_n(0)$ se detiene en menos de m pasos. Este grafo es claramente computable, es decir, dado cualquier par de vértices podemos decidir automáticamente si están conectados por una arista o no.

Ahora probaremos que $C(1)$ es Turing-equivalente a $0'$. Sabemos que $C(1)$ contiene a todos los números impares, y probaremos que $2n \in C(1) \iff n \in 0'$. La única forma de que $2n$ se conecte con un número impar es si $\varphi_{n,m}(0) \downarrow$ para algún $m \in \mathbb{N}$. Esto sólo ocurre si y sólo si, $\varphi_n(0) \downarrow$, o, equivalentemente, si $n \in 0'$. □

Juntando los dos lemas anteriores obtenemos el siguiente teorema.

Teorema 2.4. $0'$ *es necesario y suficiente para calcular* $C(v)$ *para todo grafo* G *computable y todo* $v \in V$.

2.2. Ideales maximales. Sea $(R; 0, 1, +_R, \times_R)$ un anillo conmutativo infinito numerable. Podemos suponer $R \subseteq \mathbb{N}$, y por lo tanto $+_R \colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ y $\times_R \colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, aunque estas operaciones no tiene nada que ver con la suma y el producto de números naturales. Todos los anillos que mencionaremos son conmutativos.

Definición 2.5. R es *computable* si el conjunto $R \subseteq \mathbb{N}$ y las funciones $+_R \colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ y $\times_R \colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ son computables.

Definición 2.6. Un *ideal* en R es un subconjunto $I \subseteq R$ tal que

- para todos $x, y \in I$, $x +_R y \in I$,
- para todo $x \in I$ y para todo $y \in R$, $x \times_R y \in I$.

Definición 2.7. $I \subsetneq R$ es un *ideal maximal* si no existe ningún ideal J tal que $I \subsetneq J \subsetneq R$. $I \subsetneq R$ es un *ideal primo* si $x \times_R y \in I$ implica que $(x \in I)$, o $(y \in I)$.

Estudiaremos la siguiente pregunta. ¿Qué tan difícil es encontrar un ideal maximal o primo en un anillo computable? En esta sección nos concentraremos ideales maximales, y estudiaremos ideales primos en 3.2. Los resultados de esta sección fueron obtenidos por a Friedman, Simpson y Smith.

Lema 2.8. *Todo anillo conmutativo computable tiene un ideal maximal computable en* $0'$.

Demostración. Sea R un anillo computable. Construiremos un ideal maximal $I \subseteq R$ por pasos usando $0'$ como oráculo. En el paso s, decidiremos si $s \in I$ o no (recordemos que $R \subseteq \mathbb{N}$). Si $s \notin R$, entonces $s \notin I$. Supongamos ahora que $s \in R$ y supongamos también que para todo $t < s$ ya decidimos si $t \in I$ o no. Sea I_s el conjunto de todos estos $t < s$ que ya enumeramos en I. Para saber si agregar s a I o no, debemos saber si $\langle I_s \cup \{s\}\rangle$, el ideal generado por $I_s \cup \{s\}$, contiene a 1_R o no. ¿Que tan difícil es saber si 1_R esta en $\langle I_s \cup \{s\}\rangle$? Escribimos un programa que busca por todas las combinaciones lineales de los elementos de $I_s \cup \{s\}$ con coeficientes en R, y se detiene si encuentra una tal combinación que es igual a 1_R. Si nunca la encuentra es porque 1_R no pertenece a $\langle I_s \cup \{s\}\rangle$. El oráculo $0'$ puede responder si este programa se detiene o no, por lo tanto sabe si $1_R \in \langle I_s \cup \{s\}\rangle$. Si sí, entonces dejamos a s afuera de I. Si no, entonces agregamos s a I.

Dejamos al lector verificar que el conjunto I construido de esta manera es un ideal propio, y que es maximal. □

Ahora que sabemos que si conocemos $0'$ podemos encontrar ideales maximales en cualquier anillo computable, nos preguntamos si $0'$ es necesario, o si hay alguna forma más fácil de construir ideales maximales. La respuesta es que sí, que $0'$ es necesario, y ésto es lo que dice el próximo lema.

Lema 2.9. *Existe un anillo computable* R *tal que* $0'$ *es computable en cualquier ideal maximal de* R.

Demostración. Consideremos el anillo computable $\mathbb{Q}[X_1, X_2, ...]$ de polinomios en $\mathbb{Q}$ sobre las indeterminadas $X_1, X_2, ...$ Este anillo es computable, ya que sumar y multiplicar polinomios es claramente computable. Recordemos que $0' = \{e \in \mathbb{N} : \varphi_e(0) \downarrow\}$. Sea

$$P = \langle \{X_i : i \notin 0'\}\rangle,$$

el ideal de todos los polinomios cuyos monomios tienen al menos una variable X_i con $i \notin 0'$. Observemos que P es un ideal primo. Este ideal no es computable, pero su complemento

$M = \mathbb{Q}[X_1, X_2, ...] \setminus P$ es computablemente enumerable, es decir que hay un programa que enumera todos sus miembros (ya que $0'$ es computablemente enumerable). Como P es primo, M es un conjunto multiplicativo (es decir que el producto de sus elementos se mantiene en M) . Por lo tanto podemos localizar con respecto a M. Sea

$$R = \{\frac{p}{q} : p \in \mathbb{Q}[X_1, X_2, ...], q \in M\} \subseteq \mathbb{Q}(X_1, X_2, ...).$$

Sea

$$P_R = \{\frac{p}{q} : p \in P, q \in M\}.$$

No es difícil ver que P_R es un ideal de R. Más aún, todo elemento de $R \setminus P_R$ es invertible en R, y, por lo tanto, no sólo P_R es un ideal maximal, si no que es el único ideal maximal de R. Obsérvese que $P_R \geq_T 0'$, ya que $n \in 0'$ si y sólo si $X_n \notin P_R$. Por lo tanto, $0'$ es computable en todo ideal maximal de R (el cual sólo puede ser P_R).

La prueba no termina aquí, ya que R no es un anillo computable–pero casi. Dijimos que M es c.e., y por lo tanto también lo es R. Sea $\{r_0, r_1, ...\}$ una enumeración computable de los elementos de R. Esta secuencia nos da una biyección $r\colon \mathbb{N} \to R$. A través de esta biyección podemos definir un anillo $A = (\mathbb{N}, +_A, \times_A)$ haciendo el pull-back de las operaciones de R. Osea $x +_A y = f^{-1}(f(x) +_R f(y))$ y $x \times_A y = f^{-1}(f(x) \times_R f(y))$. Como f es computable, el anillo A es computable. Como A y R son isomorfos, $f^{-1}(P)$ es el único ideal maximal de A, y también tenemos que $f^{-1}(P) \equiv_T 0'$ ya que $n \in 0'$ si y sólo si para el único i tal que $f(i) = (X_n)$ tenemos $i \notin f^{-1}(P)$. □

Juntando estos dos lemas obtenemos el siguiente teorema.

Teorema 2.10. $0'$ *es necesario y suficiente para calcular ideales maximales en anillos conmutativos computables.*

3. El Lema de König

En esta sección estudiaremos una serie de problemas cuya complejidad es menor a la del problema de la detención.

Teorema 3.1. *Sea $A \subseteq \mathbb{N}$. Los siguientes son equivalentes.*

1. *Todo árbol binario infinito computable tiene un camino computable en A;*
2. *Todo anillo computable tiene un ideal primo computable en A;*
3. *(Compacidad de $[0,1]$) Si $\{(a_i, b_i) : i \in \mathbb{N}\}$ es una familia de intervalos en $[0,1]$ con $a_i, b_i \in \mathbb{Q}$ tal que $\forall n, [0,1] \not\subseteq \bigcup_{i<n}(a_i, b_i)$, existe un número real $r \in [0,1], r \notin \bigcup_{i\in\mathbb{N}}(a_i, b_i)$ cuya representación decimal es computable en A.*

En el resto de esta sección demostraremos la equivalencia entre (1) y (2), y demostraremos que estos problemas son menos complejos que el problema de la detención, pero que tampoco son computables.

3.1. Caminos en árboles binarios. Un string binario es un secuencia finita de ceros y unos. Sea $2^{<\omega}$ el conjunto de los strings binarios finitos. Dado un string binario σ, sea $\sigma(i)$ la i-ésimo término de la secuencia, empezando por 0, y sea $|\sigma|$ el largo de la secuencia. Osea que $\sigma = (\sigma(0), \sigma(1), ..., \sigma(|\sigma| - 1))$. Dado un string binario finito σ, y dado $n \leq |\sigma|$, sea $\sigma \upharpoonright n = (\sigma(0), \sigma(1), ..., \sigma(n-1))$, el segmento inicial de σ de largo n. Dados dos strings σ y τ, decimos que $\sigma \subset \tau$ si $\sigma = \tau \upharpoonright n$ para algún n. Dados dos strings σ y τ, sea $\sigma^\frown\tau$ la concatenación de σ seguido por τ. Dada un función $f\colon \mathbb{N} \to \{0,1\}$ y $n \in \mathbb{N}$, sea $f \upharpoonright n = (f(0), ..., f(n-1)) \in 2^{<\omega}$.

Definición 3.2. Un *árbol binario* es un subconjunto $T \subseteq 2^{<\omega}$ tal que si $\tau \in T$, y $\sigma \subseteq \tau$ entonces $\sigma \in T$. Una función $f\colon \mathbb{N} \to \{0,1\}$ es un *camino* de T si $(\forall n)\ f \upharpoonright n \in T$.

Teorema 3.3 (König)**.** *Todo árbol binario infinito tiene un camino.*

Demostración. Sea T un árbol infinito. Definimos un camino f por recursión. Dado $\sigma \in T$, sea

$$T_\sigma = \{\tau \in 2^{<\omega} : \sigma^\frown \tau \in T\}.$$

Como T es infinito, sabemos que o $T_{(0)}$ es infinito, o $T_{(1)}$ es infinito, o ambos. Elijamos $f(0)$ tal que $T_{(f(0))}$ es infinito. Luego, de la misma manera, podemos definir $f(1)$ de forma que $T_{(f(0),f(1))}$ es infinito. Por recursión, siempre podemos elegir $f(n)$ de forma que $T_{f\upharpoonright n+1}$ sea infinito. □

La prueba de este lema no es efectiva, ya que no podemos decidir mecánicamente en cada paso cual de las dos ramas del árbol es infinita.

Un árbol $T \subset 2^{<\omega}$ es *computable* si es computable como conjunto. (Usando una biyección efectiva entre $\mathbb{N}$ y $2^{<\omega}$ (como por ejemplo vía representaciones binarias) podemos traducir la noción de subconjunto de $\mathbb{N}$ computable a subconjunto de $2^{<\omega}$ computable.) Decimos que un árbol es b.i.c. si es binario infinito y computable.

Lema 3.4. *Todo árbol b.i.c. tiene un camino computable en* $0'$*.*

Demostración. Solamente tenemos que observar que la prueba del Teorema 3.3 produce un camino $f \leq_T 0'$. Para ver que el proceso es computable en $0'$, tenemos que observar que $0'$ puede decidir si hay un árbol T_σ es infinito o no. Todo lo que tenemos que hacer es escribir un programa que, con entrada σ, busca un número $n > |\sigma|$ tal que ninguna extensión de σ de largo n pertenece a T. Si el programa encuentra un tal n, se detiene y sabemos que el árbol T_σ es finito. Si no, el el árbol es infinito, y el programa continua buscando para siempre. □

La pregunta natural luego de este lema es si $0'$ es necesario para computar caminos en árboles b.i.c. Antes de responder esta pregunta, el siguiente lema muestra que no hay ninguna forma mecánica de hallar un camino en un árbol infinito.

Lema 3.5. *Existe un árbol b.i.c. que no tiene caminos computables.*

Demostración. Recordemos que $\varphi_0, \varphi_1, \varphi_2, ...$ es una enumeración de todos las funciones parciales computables. Tenemos que construir un árbol binario infinito y computable tal que para todo e, φ_e no es un camino en T. Usaremos un método llamado *diagonalización*. Vamos a definir un árbol T tal que si f es un camino en T, entonces para todo e, $f(e) \neq \varphi_e(e)$. Con esto logramos que para todo e, $f \neq \varphi_e$, y por lo tanto f no es computable. El único detalle a tener en cuenta es que $\varphi_e(e)$ puede no estar definido debido a que el programa no se detiene, y es esto no lo podemos saber de una forma computable. Lo que haremos es que, si luego de s pasos vemos que $\varphi_e(e)$ se detiene, entonces no permitimos que ningún σ de largo s y tal que $\sigma(e) = \varphi_e(e)$ pertenezca a T. Recordemos la siguiente notación: decimos que $\varphi_{e,s}(n) \neq k$ si, o $\varphi_e(n)$ no se detiene en menos de s pasos, o sí se detiene en s pasos pero no toma el valor k. Nótese que, dados e, n, s y k, se puede decidir computablemente si $\varphi_{e,s}(n) \neq k$. Sea

$$T = \{\sigma \in 2^{<\omega} : (\forall e < |\sigma|) \;\; \varphi_{e,|\sigma|}(e) \neq \sigma(e)\}.$$

Dejamos al lector verificar que T es un árbol, que es infinito, que es computable, y que si f es un camino en T, f no es computable. □

Lema 3.6 (Jockusch, Soare)**.** *Todo árbol binario infinito computable tiene un camino* $f <_T 0'$*.*

Demostración. Sea T un árbol b.i.c. Ya sabemos que $0'$ puede encontrar un camino en T, pero ahora tenemos que probar que puede encontrar un camino f tal que $f \not\geq_T 0'$. Construiremos f usando $0'$ como oráculo. Al mismo tiempo construiremos una función g computable en $0'$

y usaremos el método de diagonalización para demostrar que $f \not\geq_T g$. Como $0' \geq_T g$, esto implica que $f \not\geq_T 0'$.

Para cada e, tenemos el siguiente requerimiento:

$$(R_e) \qquad \varphi_e^f(e) \neq g(e),$$

donde φ_e^f es el e-ésimo programa que usa f como oráculo, o sea que cada vez que el programa hace una pregunta al oráculo, estas preguntas van dirigidas a f. Como en las pruebas anteriores, vamos a estar interesados en $\varphi_{e,s}^f(n)$, que devuelve el resultado de este cálculo luego de s pasos. Sin perder generalidad, podemos suponer que en menos de s pasos sólo podemos preguntarle al oráculo sobre números menores a s (por ejemplo, en una máquina de Turing toma k pasos escribir el número k en la memoria para luego poder preguntarle al oráculo sobre k). Por lo tanto, si σ es un string de largo al menos s, tiene sentido escribir $\varphi_{e,s}^\sigma(n)$, ya que σ puede responder todas las preguntas que le hagamos al oráculo. Para simplificar la notación, escribiremos $\varphi_e^\sigma(n)$ en lugar de $\varphi_{e,|\sigma|}^\sigma(n)$, o sea que corremos $\varphi_e^\sigma(n)$ usando como máximo $|\sigma|$ pasos. Nótese que si $\varphi_e^\sigma(n)\downarrow$ y $\sigma \subseteq \tau$, tenemos que $\varphi_e^\tau(n)\downarrow = \varphi_e^\sigma(n)$. Por lo tanto, si fijamos e, n, k, el conjunto de todos los $\sigma \in 2^{<\omega}$ tales que $\varphi_e^\sigma(n)\uparrow$ es un árbol. (Recordemos que $\varphi_e^\sigma(n)\uparrow$ significa que $\varphi_e^\sigma(n)$ no se detiene en menos de $|\sigma|$ pasos.)

Volvamos ahora al la construcción de f y g computables en $0'$. Para construir f, vamos a construir una secuencia de árboles binarios infinitos computables $T_0 \supseteq T_1 \supseteq T_2 \supseteq \cdots$, y definiremos f como el único camino en común de todos estos árboles.

La construcción es por etapas. En la etapa número e, construiremos T_e y definiremos el valor de $g(e)$, con el objetivo de satisfacer el requerimiento (R_e). Supongamos que ya hemos definido T_{e-1} como un árbol b.i.c. Sea

$$S = \{\sigma \in T_{e-1} : \varphi_e^\sigma(e)\uparrow\}.$$

Como ya mencionamos, S es un árbol y es computable. Preguntémosle a $0'$ si S es infinito (o sea si $\exists n(S \cap 2^n = \emptyset)$ donde $2^n = \{\tau \in 2^{<\omega} : |\tau| = n\}$). Si sí, sea $T_e = S$. Si f es un camino en S, entonces $\varphi_e^f(e)\uparrow$, por que si $\varphi_e^f(e)$ se detuviese, usaría una cantidad finita del oráculo f, y por lo tanto para algún segmento inicial σ de f tendríamos $\varphi_e^\sigma(e)\downarrow$. Por lo que no importa que valor le damos a $g(e)$ – sea $g(e) = 0$. Si no, si S es finito, sea n tal que $S \cap 2^n = \emptyset$. Por lo tanto, para todo $\sigma \in T$ de largo n, $\varphi_e^\sigma(e)\downarrow$. Recorramos uno por uno todos estos σ hasta encontrar uno tal que $(T_{e-1})_\sigma = \{\tau \in T_{e-1} : \sigma^\frown\tau \in T_{e-1}\}$ es infinito, cosa que $0'$ puede responder. Una vez que encontramos un tal σ, definamos $T_e = (T_{e-1})_\sigma$ y $g(e) = (\varphi_e^\sigma(e)) + 1$. Cualquiera sea f, sólo por ser un camino en T_e tenemos que f es una extensión de σ y $\varphi_e^f(e) = \varphi_e^\sigma(e) \neq g(e)$. Si la respuesta es "no", sea $g(e) = 0$ y

$$T_e = \{\sigma \in T_{e-1} : \varphi_e^\sigma(e)\uparrow\}.$$

Como ya mencionamos, T_e es un árbol y es computable. Lo que no es tan inmediato es que es infinito. Si no fuese infinito, habría un nivel n tal para que todo $\sigma \in T_{e-1}$ de largo n, $\varphi_e^\sigma(e)\downarrow$. Como T_{e-1} es infinito, habría un tal σ con $(T_e)_\sigma$ infinito, y por lo tanto la respuesta a nuestra pregunta sería "sí".

En cualquiera de los dos casos tenemos que $\varphi_e^f(e) \neq g(e)$.

Dejamos al lector verificar los detalles de esta demostración. □

El teorema original de Jockusch y Soare es que se puede encontrar un camino f tal que $f' \leq_T 0'$. La demostración de esta versión es parecida a la del lema anterior.

El siguiente lema dice que podemos encontrar un oráculo A que está estrictamente debajo de $0'$ con respecto a $\leq_T$, tal que A pude encontrar caminos en todos los árboles binarios infinitos computables simultáneamente.

Lema 3.7. *Existe $A \subseteq \mathbb{N}$, tal que $0 <_T A <_T 0'$, y que puede computar caminos en cualquier árbol b.i.c.*

Demostración. La prueba consiste en construir un árbol b.i.c. cuyos caminos computan caminos en todos los otros árboles b.i.c. y luego aplicar el lema anterior.

Primero tenemos que demostrar que podemos enumerar, computablemente, todos los árboles binarios computables. Para cada e, vamos a construir un árbol binario computable S_e tal que si φ_e es una función total y define un árbol $\{\tau \in 2^{<\omega} : \varphi_e(\tau) = 1\}$, entonces S_e y el árbol definido por φ_e tienen los mismos caminos. Lo interesante de esta definición es que si φ_e no es total, cosa que no podemos saber computablemente, igual obtenemos un árbol binario computable S_e.

$$S_e = \{\sigma \in 2^{<\omega} : (\forall \tau \subseteq \sigma)\varphi_{e,|\sigma|}(\tau) \neq 0\}.$$

Dejamos al lector verificar que S_e cumple las propiedades que mencionamos. Ahora, no todos los árboles S_e son infinitos, así que vamos a construir una secuencia de árboles b.i.c. $\{T_e\}_{e\in\mathbb{N}}$ tal que si S_e es infinito, es igual a T_e. Si S_e es finito, sea n el mayor largo posible de los strings de S_e, y sea $\sigma \in S_e$ el menor (con respecto al orden lexicográfico) string de S_e de largo n. Sea $T_e = S_e \cup \{\tau : \tau \supseteq \sigma\}$. Esta definición de la secuencia T_e no es computable, ya que tenemos que saber si S_e es finito o no. Dejamos al lector verificar que el conjunto $\{(e, \sigma) : \sigma \in T_e\}$ sí es computable.

Ahora tenemos que juntar todos los árboles T_e en un solo árbol. Recodemos que existe una biyección computable : $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$. Supongamos que el par (n, m) representa la imagen de esta biyección. Supongamos también que la biyección es tal que si $n_0 \leq n_1$ y $m_0 \leq m_1$, $(n_0, m_0) \leq (n_1, m_1)$. Dado $\sigma \in 2^{<\omega}$ sea $\sigma^i = (\sigma((i, 0)), \sigma((i, 1)), ..., \sigma((i, k)))$ donde k es el mayor número tal que $(i, k) < |\sigma|$. De la misma forma, dada $f\colon \mathbb{N} \to \{0, 1\}$ sea $f^i\colon \mathbb{N} \to \{0, 1\}$ tal que $f^i(k) = f((i, k))$. Ahora podemos definir

$$T = \{\sigma \in 2^{<\omega} : (\forall i < |\sigma|)\sigma^i \in T_i\}.$$

No es difícil probar que T es un árbol computable, y que f es un camino en T si y sólo si, para todo i, f^i es un camino en T_i. Como todos los T_i tienen caminos, T también tiene un camino. Como f^i es computable en f, f computa caminos en todos los arboles b.i.c., y usando el lema anterior obtenemos un camino $f <_T 0'$. Sea $A = \{n : f(n) = 1\}$. □

El siguiente lema examina la relación entre buscar caminos en árboles b.i.c. y computar conjuntos computablemente enumerable. Como ya mencionamos, el problema de la detención puede computar todos los conjuntos c.e. Por otro lado, lo más que pueden hacer los caminos en árboles b.i.c. es separar dos conjuntos c.e. Demostraremos solamente una dirección.

Lema 3.8. *Para todo árbol b.i.c., existen conjuntos c.e. disjuntos $A, B \subseteq \mathbb{N}$, tales que si $S \subseteq \mathbb{N}$ es tal que*

$$A \subseteq S \subseteq \mathbb{N} \setminus B,$$

entonces S computa un camino de T.

Demostración. Sea T un árbol b.i.c. Definiremos A y B como subconjuntos de $2^{<\omega}$. (Recordemos que hay una biyección computable entre $\mathbb{N}$ y 2^ω usando la expresión binaria de los números naturales.) El objetivo es definir A y B tal que si $T_{(\sigma^\frown 0)}$ es finito, entonces $\sigma \in A$, y si $T_{(\sigma^\frown 1)}$ es finito, entonces $\sigma \in B$. Si lográramos ésto, y S separara A de B, podríamos definir un camino f en T recursivamente de la siguiente forma: Dado $f \upharpoonright n$, sea

$$f(n) = \begin{cases} 0 & \text{si } (f \upharpoonright n) \notin S \\ 1 & \text{si } (f \upharpoonright n) \in S. \end{cases}$$

De esta forma, podemos demostrar por inducción que, para cada n, $T_{(f\restriction n)}$ es infinito: Si suponemos que $T_{(f\restriction n)}$ es infinito, luego, si $T_{(f\restriction n)^\frown 0}$ es finito, entonces $(f\restriction n)\in A\subseteq S$ y si $T_{(f\restriction n)^\frown 1}$ es finito, entonces $(f\restriction n)\in B\subseteq \mathbb{N}\setminus S$. Por lo tanto $T_{(f\restriction n+1)}$ sería infinito y podemos continuar con la inducción.

El único problema es que A y B definidos de esta forma no son necesariamente disjuntos, ya que si T_σ es finito, $T_{(\sigma^\frown 0)}$ y $T_{(\sigma^\frown 1)}$ también lo son. Lo que vamos a hacer es enumerar σ en A si $T_{(\sigma^\frown 0)}$ es "más finito" que $T_{(\sigma^\frown 1)}$:

$$\begin{aligned} A &= \{\sigma\in 2^{<\omega} : (\exists n)\ T_{(\sigma^\frown 0)}\cap 2^n=\emptyset \ \&\ T_{(\sigma^\frown 1)}\cap 2^n\neq\emptyset\},\\ B &= \{\sigma\in 2^{<\omega} : (\exists n)\ T_{(\sigma^\frown 0)}\cap 2^n\neq\emptyset \ \&\ T_{(\sigma^\frown 1)}\cap 2^n=\emptyset\},\end{aligned}$$

donde $2^n=\{\tau\in 2^{<\omega} : |\tau|=n\}$. De esta forma podemos demostrar que A y B son c.e. y disjuntos, y que si T_σ es infinito tenemos que si $T_{(\sigma^\frown 0)}$ es finito, $\sigma\in A$, y si $T_{(\sigma^\frown 1)}$ es finito, $\sigma\in B$. □

3.2. Ideales primos. Pasamos ahora al estudio de la complejidad de los ideales primos. No es difícil probar que todo ideal maximal es primo, y en general así es que se prueba la existencia de ideales primos. Veremos que hay métodos para construir ideales primos que son esencialmente más simples que que los métodos necesarios para construir ideales maximales. Los resultados de esta sección también fueron obtenidos por Friedman, Simpson y Smith.

Lema 3.9. *Para todo anillo computable R, existe un árbol b.i.c. cuyos caminos pueden calcular ideales primos en R.*

Demostración. Sea $R=\{r_0,r_1,....\}$ donde $r_0=0_R$ y $r_1=1_R$. Construiremos un árbol computable $T\subseteq 2^{<\omega}$ tal que, para todo camino f de T, el conjunto

$$I_f=\{r_i : i\in\mathbb{N}, f(i)=1\}\subseteq R \quad \text{es un ideal primo propio.}$$

Sea T el conjunto de los $\sigma\in 2^{<\omega}$ tales que

- Si $|\sigma|>0$, $\sigma(0)=1$.
- Si $|\sigma|>1$, $\sigma(1)=0$.
- Para todo $i,j,k<|\sigma|$ tales que $r_i+_R r_j=r_k$, si $\sigma(i)=\sigma(j)=1$, entonces $\sigma(k)=1$.
- Para todo $i,j,k<|\sigma|$ tales que $r_i\times_R r_j=r_k$, si $\sigma(i)=1$, entonces $\sigma(k)=1$.
- Para todo $i,j,k<|\sigma|$ tales que $r_i\times_R r_j=r_k$, si $\sigma(k)=1$, entonces o $\sigma(i)=1$, o $\sigma(j)=1$, o ambos.

Estas propiedades garantizan que, si $f\in[T]$, entonces I_f contiene a 0_R, no contiene a 1_R, que $I_f+I_f\subseteq I_f$, que $I_f\times R\subseteq I_f$ y que I_f es primo, respectivamente. Nótese que T es un árbol y que es computable. □

Corolario 3.10. *Existe $A<_T 0'$ que puede calcular ideales primos en todos los anillo computables.*

Por lo tanto, encontrar ideales primos es estrictamente más fácil que encontrar ideales maximales.

Lema 3.11. *Para todo árbol b.i.c. T, existe un anillo computable cuyos ideales primos pueden calcular caminos en T.*

Demostración. Usando el Lema 3.8 tenemos que existen funciones computables f y g tales que si $S\subseteq\mathbb{N}$ separa las imágenes de f y g (es decir $(\forall n) f(n)\in X \ \&\ g(n)\notin X$), entonces S computa un camino de T. Ahora construiremos un anillo computable A, cuyos ideales primos computan conjuntos que separan f y g. Consideremos de vuelta el anillo $\mathbb{Q}[X_1,X_2,....]$. Vamos a definir un ideal computable I en $\mathbb{Q}[X_1,....]$, y después definir A como el cociente de

$\mathbb{Q}[X_1, ...]$ módulo I. Sea I el ideal de $\mathbb{Q}[X_1, ...]$ generado por $\{X^n_{f(n)} : n \in \mathbb{N}\} \cup \{X^n_{g(n)} + 1 : n \in \mathbb{N}\}$. Para probar que este ideal es computable, observamos que todo $p \in \mathbb{Q}[X_1, ...]$ es equivalente, módulo I, a un polinomio p^* tal que si X^k_m ocurre en p^*, entonces $(\forall n \leq k)\ f(n) \neq m\ \&\ g(n) \neq m$. Para construir p^* efectivamente la idea es, cada vez que vemos un término X^k_m en p, chequeamos si $(\exists n \leq k) f(n) = m$, y si sí, lo reemplazamos por (X^{k-n}_m), y si $(\exists n \leq k) g(n) = m$, lo reemplazamos por $(X^k_m - (X^n_m + 1)^{k-n})$, y continuamos con este algoritmo hasta llegar al p^* deseado. Ahora, como I es computable, no es difícil representar $A = \mathbb{Q}[X_1, ...]/I$ computablemente: alcanza con encontrar un conjunto computable de representantes para las clases de equivalencia, y podemos hacerlo tomando dentro de cada clase de equivalencia el polinomio con menor índice en $\mathbb{N}$ (recordemos que estamos usando un subconjunto de $\mathbb{N}$ para representar $\mathbb{Q}[X_1, ...]$).

Ahora supongamos que P es un ideal primo propio de A. Si $f(n) = m$, entonces como $X^n_m = 0$ módulo I, y P es primo, $X_m \in P$. Si $f(n) = m$, entonces como $X^n_m = 1$ módulo I, y P es propio, $X_m \notin P$. Por lo tanto, el conjunto $S = \{m : X_m \in P\}$ separa las imágenes de f y g como queríamos. □

Si juntamos estos dos lemas obtenemos el siguiente teorema.

Teorema 3.12. *El problema de buscar ideales primos en anillos conmutativos computables es equivalente al problema de buscar caminos en árboles binarios infinitos computables.*

Referencias

[AK00] C.J. Ash and J. Knight. *Computable Structures and the Hyperarithmetical Hierarchy*. Elsevier Science, 2000.

[Soa87] Robert I. Soare. *Recursively enumerable sets and degrees*. Perspectives in Mathematical Logic. Springer-Verlag, Berlin, 1987. A study of computable functions and computably generated sets.

Department of Mathematics, University of Chicago, 5734 S. University Ave., Chicago, IL 60637, USA

E-mail address: antonio@math.uchicago.edu

URL: www.math.uchicago.edu/∼antonio