UNIVERSIDAD
DE GRANADA

Escuela Técnica Superior de
Ingenierías Informática y de Telecomunicación
Facultad de Ciencias

Doble Grado en
Ingeniería Informática y Matemáticas

TRABAJO DE FIN DE GRADO

Una perspectiva algorítmica a la incompletitud de las Matemáticas

Presentado por:
Miguel Ángel Fernández Gutiérrez

Tutor:
Serafín Moral Callejón
Departamento de Ciencias de la Computación e Inteligencia Artificial

Curso académico 2022/2023

Una perspectiva algorítmica a la incompletitud de las Matemáticas

Miguel Ángel Fernández Gutiérrez

Miguel Ángel Fernández Gutiérrez,
Una perspectiva algorítmica a la incompletitud de las Matemáticas.

Trabajo de fin de Grado. Curso académico 2022/2023.

Responsable de tutorización

Serafín Moral Callejón
Departamento de Ciencias de la Computación e Inteligencia Artificial

Doble Grado en Ingeniería Informática y Matemáticas

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Facultad de Ciencias

Universidad de Granada

Declaro explícitamente que el trabajo presentado como Trabajo de Fin de Grado (TFG), correspondiente al curso académico 2022/2023, es original, entendida esta, en el sentido de que no he utilizado para la elaboración del trabajo fuentes sin citarlas debidamente.

En Granada a 20 de mayo de 2023

Fdo: Miguel Ángel Fernández Gutiérrez

Índice general

Summary

WIP

Resumen

WIP

Parte I.

Introducción

Hablaremos de la motivación que subyace en este trabajo, y daremos un fundamento histórico para ... WIP

1. Motivación

El objetivo de la computación es el conocimiento, no los números.

— Richard Hamming, [17]

2. Breve reseña histórica

> *Bien las matemáticas son demasiado grandes para el cerebro humano, o bien la mente del hombre es algo más que una simple máquina.*
>
> — Kurt Gödel, [14]

La historia de este trabajo mezcla dos campos de las matemáticas: la lógica y las ciencias de la computación.

La *lógica* es una de las ramas más antiguas de la filosofía, desarrollada en China, India, Grecia y en el mundo islámico. Las teorías de Aristóteles fueron incluyentes durante milenios, mientras que estoicos como Crisipo de Solos comenzaron el desarrollo de la lógica de predicados. No es hasta la Europa del siglo XVIII que se intentan formalizar las operaciones de la lógica mediante los trabajos de Leibniz y Lambert, que permanecen aislados y desconocidos. [24]

Las *ciencias de la computación* no surgirán hasta más tarde. Aunque una aproximación a la noción de algoritmo ya había sido estudiada por milenios, no fue hasta mediados del siglo XIX cuando se diseñaron las primeras formas de dispositivos de computación, con los trabajos de Charles Babbage y Ada Lovelace. [13]

Paralelamente, George Boole y Agusutus De Morgan presentan tratados sistemáticos de lógica, extendiendo la doctrina lógica aristotélica para formar un nuevo campo de estudio de las matemáticas.

Poco más tarde, se descubren fallos en los axiomas de la geometría de Euclides. Además de la independencia del postulado de las paralelas, establecido por Nikolai Lobachevsky en 1826 [5], se descubre que algunos de los teoremas considerados demostrados por Euclides no eran, de hecho, probables a partir de los axiomas. Entre estos teoremas encontramos resultados como que una línea contiene al menos dos puntos, o que dos círculos del mismo radio cuyos centros están a ese radio de distancia deben intersecar. A finales de siglo, Hilbert [24] publica en *Grundlagen der Geometrie ("los fundamentos de la geometría")* un conjunto de axiomas completos para la geometría, basados en el trabajo previo de Moritz Pasch.

Es en estos momentos cuando Giuseppe Peano publica un conjunto de axiomas para la aritmética (que veremos en el capítulo 5).

El éxito de Hilbert le lleva motiva a buscar axiomatizaciones completas de otras áreas de las matemáticas, como la teoría de números. Este será el germen de una de las áreas de investigación más populares a principios del siglo XX.

En 1900, este mismo matemático publica su famosa lista de 23 problemas para el nuevo siglo (ahora conocidos como los *problemas de Hilbert*) en el II Congreso Internacional de Matemáticos de París. El segundo de esta lista se pregunta sobre si los axiomas de la aritmética son consistentes. [20]

En el año 1910 Bertrand Russell y Alfred Whitehead publican el primer volumen de *Principia Mathematica* [46], considerado uno de los trabajos más influyentes del siglo XX.

En 1928, Hilbert reformula su segundo problema en el Congreso Internacional de Bolonia, planteando tres preguntas:

(1) ¿Son completas las matemáticas? Esto es, ¿puede probarse o no cada sentencia matemática?

(2) ¿Son consistentes las matemáticas? Esto es, ¿no es posible probar simultáneamente una afirmación y su negación?

(3) ¿Son decidibles las matemáticas? Esto es, ¿existe un método automático que pueda aplicarse a cualquier afirmación matemática, y que determine si es cierta? Este problema se conoce como *Entscheidungsproblem*.

Exploramos los conceptos de completitud, consistencia y decidibilidad en el capítulo 5. El objetivo de Hilbert era encontrar un sistema matemático completo y consistente en el que las afirmaciones puedan plantearse con precisión, y donde puedan ser demostradas de forma automática.

En 1931, Kurt Gödel destruye los sueños de Hilbert publicando *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I ("sobre proposiciones formalmente indecidibles de Principia Mathematica y sistemas relacionados")* [15]. En este artículo, Gödel prueba la incompletitud de unos ciertos sistemas. Este resultado establece severas limitaciones en las fundaciones axiomáticas de las matemáticas. Hilbert, sin embargo, no reconoce la importancia de este resultado hasta pasado un tiempo. Poco más tarde, Gödel introduce una de las primeras nociones de definiciones recursivas (las hoy llamadas μ-recursivas).

La primera prueba de la no decidibilidad del *Entscheidungsproblem* la da Alonzo Church en 1936 [8], usando la noción de función λ-calculable. En él, Church demuestra la equivalencia con las funciones μ-recursivas de Gödel, y aventura que estas funciones serán las únicas calculables mediante una tesis, que lleva su nombre: la *tesis de Church*.

Un año más tarde, Alan Turing, en su publicación *On Computable Numbers With an Application to the Entscheidungsproblem* [44] introduce un artefacto matemático que más tarde será conocido como *máquina de Turing*, introduciendo tres problemas no decidibles: el problema de "satisfacibilidad", el problema de "impresión" y el *Entscheidungsproblem*. La prueba de Turing difiere de la de Church al introducir la noción de computabilidad mediante una máquina. También aventura que la clase de funciones calculables mediante un algoritmo corresponde a las calculables mediante su máquina, lo que se conoce como *tesis de Turing*. Aunque esto no se pueda probar, Turing da en su publicación un gran número de argumentos a favor.

Más adelante, Church, Turing y Stephen Kleene [27] prueban que los tres modelos de computación introducidos (funciones μ-recursivas, funciones λ-calculables y máquinas de Turing) son equivalentes, reforzando las tesis ya mencionadas e introduciendo la *tesis de Church-Turing*.

El siglo XX vio un desarrollo excepcional de las ciencias de la computación y el nacimiento de la *informática*, llegando a la revolución de los ordenadores personales y de los dispositivos móviles a día de hoy. Todo esto fue posible gracias al trabajo de, entre otros, los matemáticos que mencionamos en esta breve reseña.

Parte II.

Fundamentos

Crearemos un fundamento matemático que nos permitirá usar programas en Python como parte de demostraciones formales, y comentaremos la relación que esto tiene con una de las tesis principales de la teoría de la computación. Hablaremos sobre problemas de decisión, universalidad y el problema de parada. Finalmente, probaremos la no decidibilidad del problema de parada.

3. Programas en Python y máquinas de Turing

> *Hemos perfeccionado el ordenador de propósito general para cálculos científicos. Ahora intentamos emplear el mismo dispositivo con pequeñas modificaciones en una gran variedad de situaciones diferentes para las cuales no fue diseñado en primer lugar. En estas circunstancias, si llegara a resultar que la lógica básica de una máquina diseñada para la solución numérica de ecuaciones diferenciales coincide con la lógica básica de una máquina destinada a hacer facturas para una tienda departamental, consideraría esto como la coincidencia más asombrosa que haya encontrado.*
>
> — Howard Aiken, [2]

La máquina de Turing, inventada por Alan Turing en 1937 [44], es un modelo matemático sencillo que pretende representar la noción de algoritmo, y del que hablaremos en detalle en la sección 3.2. Esta máquina es una de las herramientas más relevantes en las ciencias de la computación, pues su simplicidad permite resolver potentes resultados, como la no decibilidad del problema de parada (proposición 4.6), a partir del cual podremos adentrarnos en la cuestión principal de nuestro trabajo: la incompletitud de las matemáticas. Sin embargo, esta potencia formal tiene una desventaja, que comprobaremos a partir de un simple ejemplo (ejemplo 3.1): definir programas sencillos en este modelo es complejo y tedioso.

Este inconveniente es el que motiva este capítulo: en la sección 3.3 probaremos la equivalencia entre los programas de ordenador (en nuestro caso, los programas en Python, que exploraremos en la sección 3.1) y las máquinas de Turing. Esto nos permitirá ver un programa en Python como una representación de una máquina de Turing, eliminando el principal inconveniente y quedándonos con la potencia matemática ya comentada.

Es importante resaltar que los resultados que aquí aparecen son perfectamente generalizables a cualquier lenguaje de programación. Sin embargo, decidimos usar este lenguaje por su facilidad de comprensión y de uso.

Tanto el resultado de equivalencia que probamos en este capítulo como la observación anterior son especialmente interesantes, ya que guardan una fuerte relación con la tesis de Church-Turing, de la que hablaremos brevemente en la sección 3.4.

3.1. Programas en Python

Python [32] es un lenguaje de programación de alto nivel, interpretado, orientado a objetos y *open source*, ampliamente usado en la actualidad para una gran variedad de aplicaciones. Una de sus principales características es su legibilidad y facilidad de uso: cualquier persona experimentada en otros lenguajes podrá comprender y redactar programas en Python con mucha sencillez.

Usando Python, podemos escribir una gran variedad de programas. Para el propósito de nuestro estudio, nos centraremos en una clase de programas específica: aquella que tiene únicamente una cadena de caracteres (*string*) como entrada y una cadena de caracteres como salida. Los llamaremos *programas SISO* (del inglés, *Single Input-Single Output*, ver definición 3.2). [31] En breve, comprobaremos cómo esta clase no es restrictiva: estos tipos de programas son tan expresivos como cualquier otro programa. De hecho, en este trabajo, usaremos los términos "programa" y "programa SISO" indistintamente, y en ambas ocasiones nos referiremos a un programa en Python de tipo SISO.

Definición 3.1 (Función SISO)**.** Definimos una *función SISO* como una función de Python que acepta un único parámetro de tipo *string* como entrada, y que devuelve un *string*.

Definición 3.2 (Programa en Python de tipo SISO)**.** Respecto a un sistema informático de referencia C, definimos un *programa en Python de tipo SISO* como una cadena de caracteres P, de modo que:

- P es código en Python sintácticamente correcto;
- P define al menos una función, y la primera de ellas es de *tipo SISO*. A esta función la llamaremos *función main*.

Observemos que en la definición anterior se especifica un sistema informático C, pero no se define de forma explícita. Podemos pensar en C como cualquier sistema informático, con cualquier configuración de *software* y *hardware*, y con tanta memoria como sea necesaria.

También podemos preguntarnos por qué queremos decir cuando nos referimos a un programa "sintácticamente correcto", dado que esto dependerá de varios factores, como la versión de Python instalada en C, así como el sistema operativo, arquitectura, etc. Es posible que un programa se ejecute en un sistema C y falle o incluso de una salida diferente en otro sistema C'. Sin embargo, y como veremos más adelante, usaremos estos programas como representaciones de máquinas de Turing. Los programas que presentamos aquí son independientes de arquitectura y de memoria, y las demostraciones que figuran en este trabajo son independientes de todas estas características.

También necesitaremos definir cuál es la salida de un programa en Python.

Definición 3.3 (Salida de un programa)**.** Sea P un programa en Python respecto a un sistema informático C. Sea M la función *main* de P, e I la *entrada* de P, una cadena de caracteres. La *salida de P respecto a la entrada I*, $P(I)$, se produce usando C para ejecutar P con entrada I, y se define como:

- Si M devuelve una cadena de caracteres O, entonces $P(I) = O$.
- Si M devuelve cualquier otro objeto en Python, entonces $P(I)$ no está definido.
- Si M lanza una excepción,[1] entonces $P(I)$ no está definido.
- Si M no termina, entonces $P(I)$ no está definido.

[1] Nos referimos a las excepciones que se pueden definir implícitamente en el lenguaje o que pueden resultar de errores en su uso, dado que en la definición de P el programa es correcto sintácticamente.

Nota. La definición anterior puede ser generalizada fácilmente para n entradas. Reemplazando I por $I_1, I_2, ..., I_n$ obtenemos la definición correspondiente para $P(I_1, I_2, ..., I_n)$.

En otras palabras, la salida de un programa es la cadena de caracteres devuelta por su función *main* en caso de que lo haga, y no estará definida en caso contrario.

Al definir los programas en Python que usaremos en este trabajo, nos centramos en el uso de cadenas de caracteres (también llamadas *strings*) como entrada y salida, y evitamos el uso de otros objetos de Python que no sean *strings*. Inicialmente, podemos pensar que esto impone una limitación sobre los programas en Python que podemos escribir, pero esto realmente no es así. Para ello, basta observar que que cualquier objeto en Python puede ser codificado debidamente mediante una cadena de caracteres. Existen liberías nativas que permiten hacer esto, como `pickle`.[2] Un ejemplo de esto se muestra en el programa 3.1. La función `funcion_con_objeto_codificado` es una función *main*.

```
1  import pickle

3  def funcion_con_objeto_codificado(codificado):
4      descodificado = pickle.loads(codificado)
5      # realizamos las operaciones que queramos con el objeto descodificado
6      # el objeto descodificado puede ser de cualquier tipo

8  # creamos un objeto, por ejemplo, una matriz
9  adyacencias = {1:2, 3:4, 5:3, 4:2}

11 # serializamos (codificamos) el objeto a un string
12 adyacencias_codificado = pickle.dumps(adyacencias)

14 # podemos ahora ejecutar este objeto dentro de una función SISO
15 funcion_con_objeto_codificado(adyacencias_codificado)
```

Programa 3.1: `funcion_con_objeto_codificado.py`

En la línea 1 importamos la librería `pickle`, que incorpora dos funciones:

- `dumps` convierte un objeto cualquiera a una cadena de caracteres. En la línea 12, serializamos el objeto `adyacencias` a un *string*.
- `loads` convierte la cadena de caracteres al objeto correspondiente. En la línea 4, "descodificamos" la cadena de caracteres, obteniendo en este caso el objeto `adyacencias`.

En resumen, hemos visto cómo la condición de ser SISO para un programa en Python no es realmente una restricción. Sin embargo, esta clase de problemas nos resultará muy conveniente, como veremos más adelante.

[2] La librería `pickle` puede codificar y descodificar una gran variedad de tipos de datos, incluyendo todos los nativos. Tiene algunas limitaciones para objetos más complejos pero, dado que todos los objetos se pueden reducir estos tipos de datos, podemos afirmar que cualquier objeto puede ser codificado. [30]

3.2. Máquinas de Turing

Las máquinas de Turing, primero descritas por Alan Turing en 1936 [44], son dispositivos computacionales abstractos y simples creados para ayudar a investigar qué problemas pueden ser computados. Las "máquinas automáticas" –tal como Turing las llamó en 1936– fueron específicamente diseñadas para computar números reales. Estas máquinas no fueron nombradas "máquinas de Turing" hasta 1937, cuando Alonzo Church [7] revisó la publicación original de Turing. Hoy se consideran uno de los modelos fundamentales de la teoría de la computación, pues son la representación matemática de un algoritmo [10].

Antes de comprender cómo funcionan estas máquinas, haremos unas definiciones previas.

Definición 3.4 (Alfabeto, símbolo, palabra, longitud, palabra vacía)**.** Un *alfabeto*, A, es un conjunto finito. Sus elementos se llamarán *símbolos* (o *letras*).

Una *palabra* sobre el alfabeto A es una sucesión finita de elementos de A. Es decir, u es una palabra de A si y solo si $u = a_1\, a_2 \ldots a_n$, con $a_i \in A \;\; \forall i = 1, 2, ..., n$.

El conjunto de todas las palabras sobre un alfabeto se denota A^*.[3]

La *longitud* de una palabra u es el número de símbolos que contiene, y se denota $\|u\|$. Para $u = a_1\, a_2 \ldots a_n$, es $\|u\| = n$.

La *palabra vacía* es la palabra de longitud cero. Es la misma para todos los alfabetos, y se denota como ϵ.

La primera definición presentada por Turing en 1936 [44] dista de la que se presenta en la definición 3.5. Sin embargo, ambas son equivalentes.

Definición 3.5 (Máquina de Turing)**.** Una *máquina de Turing* es una séptupla $M = (Q, A, B, \delta, q_0, \sqcup, F)$, en la que:

- $Q = \{q_0, ...\}$ es un conjunto (finito) de *estados*.
- A es un *alfabeto de entrada*.
- B es un *alfabeto de símbolos de la cinta* (o *alfabeto de trabajo*), que incluye a A ($B \subset A$).
- $\delta : Q \times B \longrightarrow (Q \times B \times \{I, D, S\}) \cup \varnothing$ es la *función de transición* que asigna a cada estado $q \in Q$ y símbolo $b \in B$, el valor $\delta(q, b)$ que puede ser:
  - $\varnothing$ en caso de no estar definido, o
  - una tripleta (p, c, M), donde $p \in Q$, $c \in B$, $M \in \{I, D, S\}$.
- $q_0 \in Q$ es el *estado inicial*.
- $\sqcup \in B \backslash A$ es el *símbolo blanco*.
- $F \subseteq Q$ es el conjunto de *estados finales*.

[3] Esta notación proviene de la operación de clausura de Kleene.

La definición anterior es un tanto abstracta, y podemos visualizarla mejor en la figura 3.1. En ella, vemos las tres partes "físicas" de la máquina: la *unidad de control*, el *cabezal* y la *cinta*. La cinta contiene *casillas*, cada una de ellas puede almacenar un símbolo. Inicialmente, los símbolos son los del alfabeto de entrada, pero conforme la máquina va ejecutando instrucciones se puede usar el conjunto de símbolos del alfabeto de trabajo. Estos símbolos incluyen un símbolo blanco, que denotamos como ␣. Estos símbolos se encuentran en todas las posiciones de la cinta que no codifican la palabra de entrada. El cabezal puede moverse por la cinta a la izquierda y a la derecha (o quedarse en el sitio), informando a la unidad de control de los símbolos que ve. El símbolo al que apunta el cabezal lo llamaremos *símbolo escaneado*. El cabezal puede reemplazar el símbolo escaneado por cualquier otro del alfabeto de trabajo, incluyendo el símbolo blanco. La unidad de control manda *transiciones* al cabezal, y cambia entre los diferentes estados en función de los símbolos leídos por el cabezal.

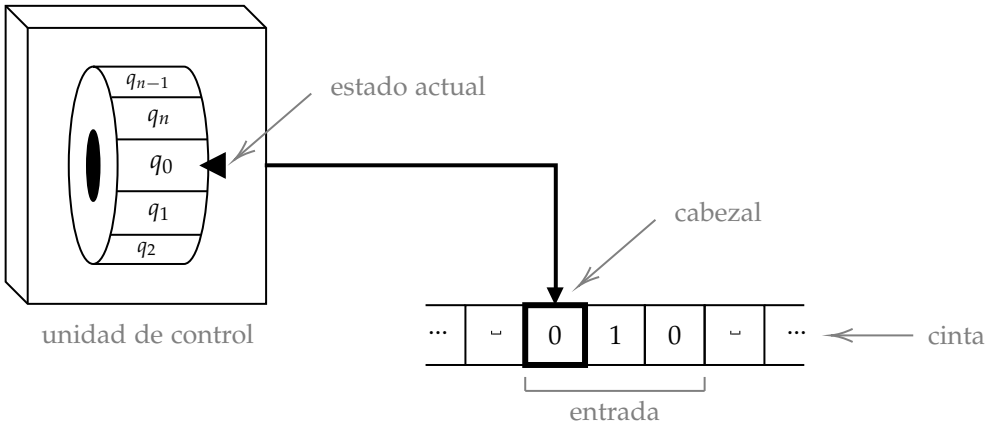


Figura 3.1.: Configuración inicial de una máquina de Turing para la entrada 010

Es importante aclarar que la máquina de Turing realmente no tiene ninguna "unidad de control" o "cabezal": es una descripción matemática. Lo que hemos hecho es proporcionar una especificación intuitiva de cómo se ejecuta el algoritmo que representa tal máquina.

Para describir la máquina en cada momento, necesitaremos el concepto de *configuración*.

Definición 3.6 (Configuración de una máquina de Turing)**.** Especificaremos la *configuración* de una máquina de Turing mediante dos parámetros:

- El estado $q \in Q$ en el que se encuentra la máquina.
- La palabra $w \in B^*$ presente en la *cinta* y la posición del *cabezal de lectura*, que representaremos recuadrando el símbolo donde se encuentra el cabezal. Dado que el cabezal se extiende infinitamente a ambos lados, eliminamos los caracteres blancos a la izquierda y a la derecha hasta llegar a la posición del cabezal.

Representaremos una configuración mediante

$$q : w$$

con uno de los símbolos de w debidamente recuadrado.

Lo único que nos queda es precisar la noción de "transición" que comentamos anteriormente. En cada paso, aplicamos una transición y cambiamos la configuración de la máquina. Esto es lo que denominamos *proceso de cálculo*, y formalizamos a continuación.

Definición 3.7 (Proceso de cálculo de una máquina de Turing)**.** Dada una máquina de Turing $M = (Q, A, B, \delta, q_0, \sqcup, F)$, y dada una entrada inicial $u = u_1\, u_2 \ldots u_n \in A^*$, definimos su *configuración inicial* como:

$$q_0 \ : \ \boxed{u_1}\, u_2 \ldots u_n$$

A continuación, definimos un *paso de cálculo* para cada configuración de la máquina:

$$q \ : \ b_{-n} \ldots b_{-2}\, b_{-1}\, \boxed{b_0}\, b_1\, b_2 \ldots b_m$$

con $q \in Q, b = b_{-n} \ldots b_{-2}\, b_{-1}\, b_0\, b_1\, b_2 \ldots b_m \in B^*$, en función del valor de la función de transición $\delta(q, b)$:

- Si $\delta(q, b)$ no está definido ($\delta(q, b) = \varnothing$), la máquina para.
- Si $\delta(q, b) = (p, c, M)$ con $p \in Q, c \in B, M \in \{I, D, S\}$, la máquina reemplazará el símbolo b presente en la posición de la cinta en la que se encuentra el cabezal por c, moverá el cabezal una posición de acuerdo a M (a la izquierda, derecha o no se moverá si es I, D o S, respectivamente), y cambiará su estado a p:

$$\begin{aligned} M = I \quad &\Rightarrow \quad p \ : \ b_{-n} \ldots b_{-2}\, \boxed{b_{-1}}\, c\, b_1\, b_2 \ldots b_m \\ M = D \quad &\Rightarrow \quad p \ : \ b_{-n} \ldots b_{-2}\, b_{-1}\, c\, \boxed{b_1}\, b_2 \ldots b_m \\ M = S \quad &\Rightarrow \quad p \ : \ b_{-n} \ldots b_{-2}\, b_{-1}\, \boxed{c}\, b_1\, b_2 \ldots b_m \end{aligned}$$

El *proceso de cálculo* se define como la consecución de pasos de cálculo, comenzando por la configuración inicial.

Es posible que la máquina pare cuando llega a una configuración tal que la función de transición no está definida.[4] Si esto no ocurre, la sucesión de pasos de cálculo será infinita y la máquina no parará nunca.

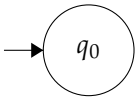
En definitiva, vemos que la máquina, en función del estado en el que se encuentre y el símbolo escaneado, ejecutará una cierta transición, definida en δ, que le dirá a qué estado pasar, por qué símbolo reemplazar el escaneado y el movimiento que tiene que realizar sobre la cinta.

Una nota importante respecto a las transiciones no definidas. A la hora de describir una función de transición, definiremos únicamente las transiciones para los estados $q \in Q$ y símbolos $b \in B$ para los que están definidas. En caso contrario, podemos asumir que $\delta(q, b) = \varnothing$. Puedes ver esto en el ejemplo 3.1, que comentaremos en detalle más adelante.

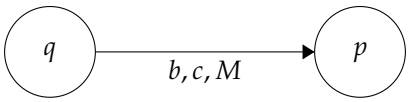
[4] Nótese que esta definición permite que la máquina siga cambiando su configuración aun habiendo llegado a un estado final. Normalmente, las máquinas de Turing paran llegado al estado final, y esto es algo de lo que nos podemos asegurar simplemente haciendo que $\delta(q, b) = \varnothing \ \forall q \in F, b \in B$.

Introducimos una notación gráfica para la función de transición. La definiremos mediante un grafo dirigido, representaremos los estados como nodos (uno para cada estado) y las transiciones como arcos dirigidos.

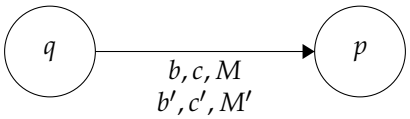
El estado inicial se simbolizará, además de en la definición de la máquina, con una flecha apuntando al estado (no es un arco, sino que no proviene de ningún nodo):



Para cada transición $\delta(q, b) = (p, c, M)$, insertaremos un arco dirigido conectando los nodos correspondientes a los estados q y p como sigue:



En caso de que tengamos varias funciones de transición entre ambos estados ($\delta(q, b) = (p, c, M)$ y $\delta(q, b') = (p, c', M')$) colocaremos varias líneas en el arco para simbolizarlo:



Finalmente, también indicaremos los estados finales rodeando dos veces el arco:

Observemos cómo una máquina de Turing, dada una entrada, puede:

- parar y aceptar,
- parar y rechazar, o
- no parar (*ciclar*).

La máquina para cuando llega a una configuración para la que no hay una transición definida. Cuando ocurre esto, podemos definir una salida con los símbolos que quedan en la cinta.

Definición 3.8 (Salida de una máquina de Turing)**.** Para una máquina de Turing M, definimos su *salida* como los contenidos de la cinta una vez que la máquina ha parado, omitiendo los símbolos blancos a ambos lados.

Definición 3.9 (Palabra aceptada por una máquina de Turing)**.** Dada una máquina de Turing $M = (Q, A, B, \delta, q_0, \sqcup, F)$, y dada una entrada inicial $u \in A^*$, decimos que M *acepta* u si M para con entrada u en un estado $q \in F$. Si lo hace en un estado $q \notin F$, decimos que M *rechaza* u.

Al haber definido qué palabras acepta una máquina determinada, introducimos el concepto de *lenguaje*. Un lenguaje no es más que un conjunto de palabras.

Definición 3.10 (Lenguaje aceptado por una máquina de Turing)**.** Un lenguaje L sobre un alfabeto A es un conjunto de palabras sobre A.

Si $M = (Q, A, B, \delta, q_0, \sqcup, F)$ es una máquina de Turing, el *lenguaje aceptado*, denotado como $L(M)$, es el conjunto de palabras aceptadas por M, es decir,

$$u \in L(M) \iff M \text{ acepta } u$$

Concretemos todos los conceptos que hemos introducido mediante un ejemplo.

Ejemplo 3.1. Sea $M_{\#a>\#b}$ una máquina de Turing tal que:

$$M_{\#a>\#b} = (\{q_0, q_a, q_b, q_R, q_F\}, \{a, b\}, \{a, b, X, \sqcup\}, \delta, q_0, \sqcup, \{q_F\})$$

con δ, su función de transición, descrita mediante el grafo:

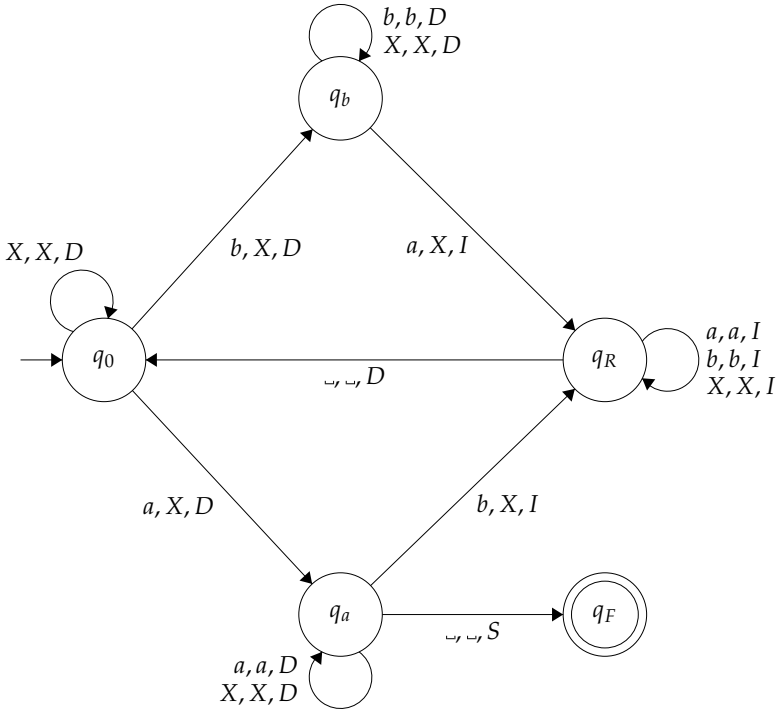


Figura 3.2.: La función de transición δ de $M_{\#a>\#b}$

Equivalentemente podemos definir la función de transición explícitamente:

$$\begin{array}{llll} \delta(q_0,a)=(q_a,X,D) & \delta(q_0,b)=(q_b,X,D) & \delta(q_0,X)=(q_0,X,D) & \\ \delta(q_a,a)=(q_a,a,D) & \delta(q_a,b)=(q_R,X,I) & \delta(q_a,X)=(q_a,X,D) & \delta(q_a,\text{␣})=(q_F,\text{␣},S) \\ \delta(q_b,a)=(q_R,X,I) & \delta(q_b,b)=(q_b,b,D) & \delta(q_b,X)=(q_b,X,D) & \\ \delta(q_R,a)=(q_R,a,I) & \delta(q_R,b)=(q_R,b,I) & \delta(q_R,X)=(q_R,X,I) & \delta(q_R,\text{␣})=(q_0,\text{␣},D) \end{array}$$

Veamos qué hace esta máquina. Empieza al principio de la cinta, y escanea los símbolos uno a uno, de izquierda a derecha. Hay tres posibilidades, dependiendo de si vemos a, b o ␣:

1. Si se lee una b:
   1.1. Reemplazar la b con una X.
   1.2. Continuar escaneando hacia la derecha.
      1.2.1. Si encontramos una a, reemplazarla con una X. Habremos reemplazado el mismo número de a que de b, así que nos movemos al inicio de la cinta y reiniciamos el algoritmo.
      1.2.2. Si no encontramos una a, tenemos una b de más, así que *rechazamos*.
2. Si se lee una a:
   2.1. Reemplazar la a con una X.
   2.2. Continuar escaneando hacia la derecha.
      2.2.1. Si encontramos una b, reemplazarla con una X. Habremos reemplazado el mismo número de a que de b, así que nos movemos al inicio de la cinta y reiniciamos el algoritmo.
      2.2.2. Si no encontramos una b, tenemos una a de más, así que *aceptamos*.
3. Si no encontramos a o b antes del ␣ al final de la entrada: el número de a reemplazados equivale al número de b reemplazados, así que *rechazamos*.

El lenguaje aceptado por esta máquina es el conjunto de palabras de $\{a,b\}^*$ con un número de a mayor que de b:[5]

$$L(M_{\#a>\#b}) = \{u \in \{a,b\}^* \ : \ \#(u,a) > \#(u,b)\}$$

Veamos un ejemplo de palabra aceptada: *abbaa*. Los pasos que da la máquina de Turing configuración tras configuración serán los siguientes:

$$\begin{array}{rcccccccccc} q_0 & : & \boxed{a}\,b\,b\,a\,a & \to & q_a & : & X\,\boxed{b}\,b\,a\,a & \to & q_R & : & \boxed{X}\,X\,b\,a\,a \\ \to & q_R & : & \text{␣}\,X\,X\,b\,a\,a & \to & q_0 & : & \boxed{X}\,X\,b\,a\,a & \to & q_0 & : & X\,\boxed{X}\,b\,a\,a \\ \to & q_0 & : & X\,X\,\boxed{b}\,a\,a & \to & q_b & : & X\,X\,X\,\boxed{a}\,a & \to & q_R & : & X\,X\,\boxed{X}\,X\,a \\ \to & q_R & : & X\,\boxed{X}\,X\,X\,a & \to & q_R & : & \text{␣}\,X\,X\,X\,X\,a & \to & q_0 & : & \boxed{X}\,X\,X\,X\,a \\ \to & q_0 & : & X\,\boxed{X}\,X\,X\,a & \to & q_0 & : & X\,X\,\boxed{X}\,X\,a & \to & q_0 & : & X\,X\,X\,\boxed{X}\,a \\ \to & q_0 & : & X\,X\,X\,X\,\boxed{a} & \to & q_a & : & X\,X\,X\,X\,X\,\text{␣} & \to & q_F & : & X\,X\,X\,X\,X\,\text{␣} \end{array}$$

Como la máquina para en el estado $q_F \in F$, la palabra es aceptada.

[5] Usamos la notación: $\#(u,a)$ es igual al número de apariencias del símbolo a en u.

Veamos un ejemplo de una palabra no aceptada, como es *abb*.

$$\begin{aligned} q_0 : \boxed{a}\, b\, b \;&\to\; q_a : X \boxed{b}\, b \;\to\; q_R : \boxed{X}\, X\, b \;\to\; q_R : \boxed{\sqcup}\, X\, X\, b \\ &\to\; q_0 : \boxed{X}\, X\, b \;\to\; q_0 : X \boxed{X}\, b \;\to\; q_0 : X\, X \boxed{b} \\ &\to\; q_b : X\, X\, X \boxed{\sqcup} \end{aligned}$$

Como la máquina para en el estado $q_b \notin F$, la palabra no es aceptada.

La máquina también permite la entrada vacía (ϵ, que definimos en definición 3.4), en cuyo caso la sucesión de configuraciones será:

$$q_0 : \boxed{\sqcup}$$

La máquina para inmediatamente en $q_0 \notin F$, y ϵ no es aceptada. Esto es esperado, pues ϵ tiene exactamente el mismo número de a que de b.

Estos ejemplos quedan resumidos en la tabla 3.1, junto con el comportamiento de la máquina para otras palabras. Observa cuándo cada palabra es aceptada, así como el número de transiciones tras los que la máquina para.

Una vez que hemos descrito con precisión qué es una máquina de Turing, procederemos a probar el resultado más importante de este capítulo (el teorema 3.1), que ya anticipamos en la introducción: la equivalencia entre las máquinas de Turing recién expuestas y los programas en Python que definimos en la sección 3.1.

Entrada	Config. inicial		Config. al parar	¿Aceptada?	N.º transiciones
abbaa	$q_0 : \boxed{a}\, b\, b\, a\, a$	$\to$	$q_F : X\, X\, X\, X\, X \boxed{\sqcup}$	sí	18
abb	$q_0 : \boxed{a}\, b\, b$	$\to$	$q_b : X\, X\, X \boxed{\sqcup}$	no	7
ϵ	$q_0 : \boxed{\sqcup}$	$\to$	$q_0 : \boxed{\sqcup}$	no	0
abab	$q_0 : \boxed{a}\, b\, a\, b$	$\to$	$q_0 : X\, X\, X\, X \boxed{\sqcup}$	no	16
ababb	$q_0 : \boxed{a}\, b\, a\, b\, b$	$\to$	$q_b : X\, X\, X\, X\, X \boxed{\sqcup}$	no	17
ababa	$q_0 : \boxed{a}\, b\, a\, b\, a$	$\to$	$q_F : X\, X\, X\, X\, X \boxed{\sqcup}$	sí	18
abaaab	$q_0 : \boxed{a}\, b\, a\, a\, b$	$\to$	$q_F : X\, X\, X\, X\, a\, X \boxed{\sqcup}$	sí	23

Tabla 3.1: Ejemplos del comportamiento de $M_{\#a>\#b}$ con diversas entradas

3.3. Equivalencia

Nuestro objetivo ahora es comprender por qué las máquinas de Turing tienen la misma potencia de cálculo que los programas en Python. Esto quiere decir que los problemas que podemos resolver en ambos son los mismos.[6]

Para probar la equivalencia entre ambos modelos de cálculo, recurriremos al concepto de *simulación*. Es normal para diferentes modelos computacionales simularse unos a otros. Un ejemplo de esto es la máquina virtual de Java (JVM), que es un modelo abstracto capaz de ejecutar cualquier programa en Java. [11] Otro ejemplo es el sistema operativo Linux, que es capaz de ejecutar la JVM. Por lo tanto, un ordenador Linux es capaz de ejecutar cualquier programa en Java. Esto es posible mediante la cadena de simulaciones:

$$\text{Linux} \rightsquigarrow \text{JVM} \rightsquigarrow \text{programa en Java}$$

Usamos la notación $A \rightsquigarrow B$ para indicar que A simula a B. El hecho de que A simule B quiere decir, sencillamente, que es posible realizar el procedimiento de cálculo de B usando exclusivamente el procedimiento de cálculo de A.

Procederemos a probar la equivalencia mostrada mediante la cadena de simulaciones que aparece en la figura 3.3.

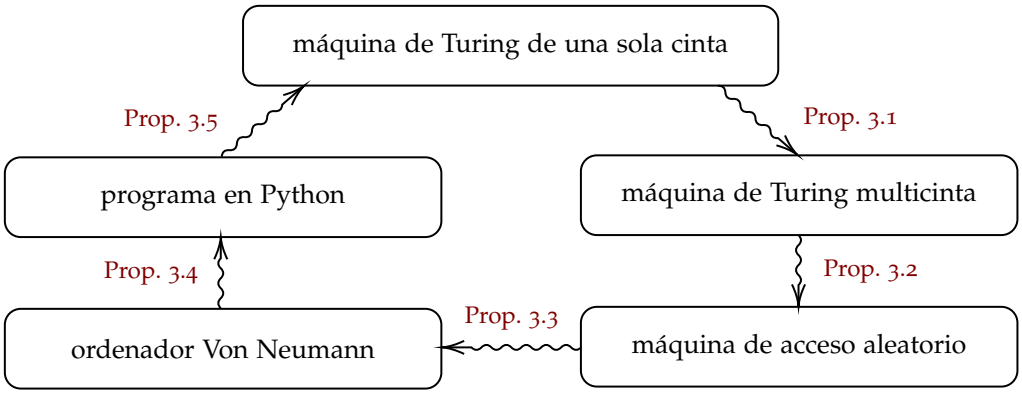


Figura 3.3.: Cadena de simulaciones

El trabajo que haremos en esta sección es arduo, y necesitaremos introducir diversos modelos computacionales para llegar a nuestro objetivo. Sin embargo, una vez probemos todas las simulaciones, sabremos que todas ellas son equivalentes, y podremos usar el modelo que queramos a conveniencia.

El primer modelo computacional que estudiaremos lo obtendremos añadiendo más cintas y cabezales a la máquina de Turing que definimos en la sección 3.2. [40]

[6] No introduciremos el concepto de "problema" con precisión hasta el capítulo 4 – por ahora, nos quedaremos con la noción intuitiva de problema, sin mayor explicación.

Definición 3.11 (Máquina de Turing multicinta)**.** Una *máquina de Turing multicinta* con k cintas es una máquina de Turing convencional en la que tenemos k cintas y k cabezales de lectura. En este sentido, la configuración de una máquina de Turing multicinta se describe como:

$$q \ : \ w_1 \ , \ w_2 \ , \ ... \ , \ w_k$$

donde cada una de las palabras de la cinta tiene uno de sus símbolos recuadrados, simbolizando la posición del cabezal en cada cinta.

La función $\delta : Q \times B^k \longrightarrow \Big(Q \times (B \times \{I, D, S\})^k\Big) \cup \varnothing$ será de la forma:

$$\delta(q, b_1, b_2, ..., b_k) = \varnothing$$

para las transiciones no definidas, y para las definidas:

$$\delta(q, b_1, b_2, ..., b_k) = (p, c_1, M_1, c_2, M_2, ..., c_k, M_k),$$
$$\text{con } q, p \in Q, b_i, c_i \in B, M_i \in \{I, D, S\}, \ \forall i \in \{1, 2, ..., k\}$$

El proceso de cálculo será análogo al de una máquina de Turing de una sola cinta, teniendo δ en cuenta que el símbolo b_i es el leído por el cabezal i-ésimo, que c_i es el símbolo por el que se cambia b_i en el cabezal i-ésimo, y M_i es el movimiento que seguirá el cabezal i-ésimo.

La entrada de la máquina se colocará en la primera cinta, con el cabezal en el primer símbolo de la palabra. El resto de cabezales apuntarán al símbolo vacío, dado que todas las cintas excepto la primera estarán llenas de símbolos vacíos.

Las palabras aceptadas por la máquina procederán de forma análoga al caso de una sola cinta, teniendo en cuenta si el estado donde la máquina para es final o no (está en F).

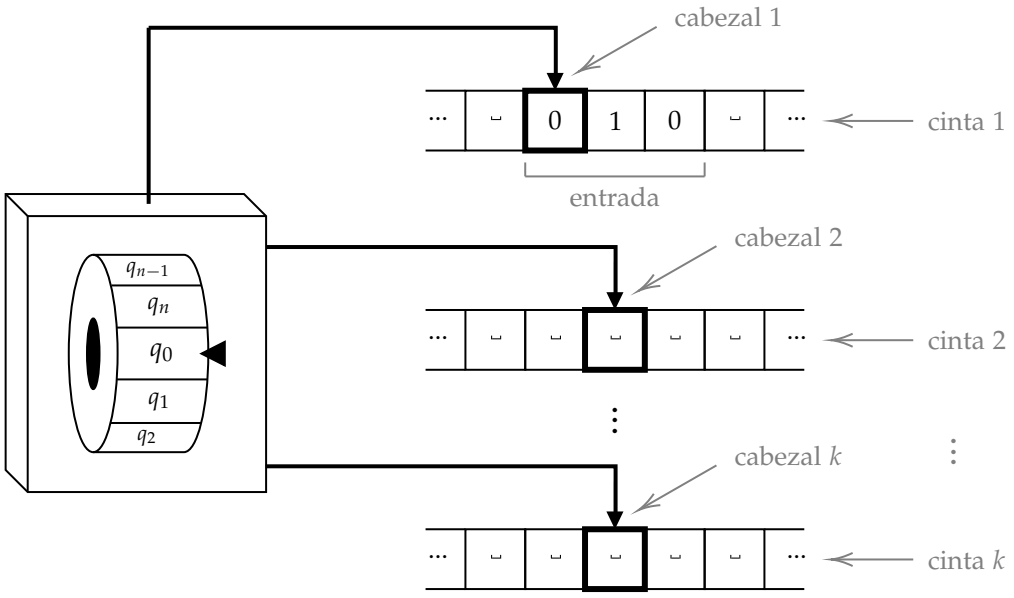


Figura 3.4.: Diagrama de una máquina de Turing multicinta en su configuración inicial

Esta definición es intuitiva y esperable. Simplemente incorporamos más cintas y cabezales. De ahora en adelante, también usaremos el término "máquina de Turing de una sola cinta" para referirnos a la máquina de Turing que definimos en primer lugar.

Uno puede pensar que estas máquinas tienen más poder computacional que las de una sola cinta, pero, sin embargo, son equivalentes. Es evidente que una máquina multicinta simula una máquina de una sola cinta (la definición 3.5 es la misma que la definición 3.11 para $k = 1$). En la proposición 3.1 vemos cómo el recíproco también es cierto.

Proposición 3.1. *Toda máquina de Turing multicinta puede ser simulada por una máquina de Turing de una sola cinta.*

Demostración.[7] Sea $M = (Q, A, B, \delta, q_0, \llcorner, F)$ una máquina de Turing con k cintas. Vamos a crear una máquina $\widetilde{M} = (\widetilde{Q}, A, \widetilde{B}, \widetilde{\delta}, q_0, [\llcorner^k], F)$ con una sola cinta de modo que $L(M) = L(\widetilde{M})$.

Para simular k cintas y k cabezales en un solo cabezal y una sola cinta, deberemos ampliar el alfabeto de trabajo $\widetilde{B}$. En primer lugar, vamos a introducir nuevos símbolos para poder saber la posición de cada uno de los cabezales. Definimos:

$$\breve{B} = \{\breve{b} : b \in B\}$$

Y, hecho esto, definimos $\widetilde{B}$, el alfabeto de $\widetilde{M}$, como:

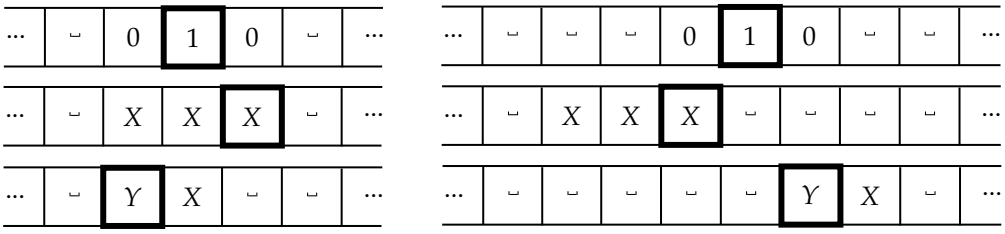
$$\widetilde{B} = \{[b_1\, b_2 \ldots b_k] : b \in B \cup \breve{B}\} \cup \{\#\}$$

Observemos que los símbolos de $\widetilde{B}$ son bien un delimitador #, bien un grupo de caracteres de $B \cup \widetilde{B}$. De este modo, podemos simular k cintas y k cabezales en una sola cinta con un solo cabezal. Un ejemplo: si en M tenemos la configuración:

$$q \;:\; 0\,\boxed{1}\,0\;,\; X\,X\,\boxed{X}\;,\; \boxed{Y}\,X$$

Una de las posibles configuraciones en $\widetilde{M}$ sería:

$$q \;:\; \boxed{\#}\,[0\,X\,\breve{Y}]\,[\breve{1}\,X\,X]\,[0\,\breve{X}\,\llcorner]\,\#$$



$q \;:\; \boxed{\#}\,[0\,X\,\breve{Y}]\,[\breve{1}\,X\,X]\,[0\,\breve{X}\,\llcorner]\,\#$ $\qquad$ $q \;:\; \boxed{\#}\,[\llcorner\,X\,\llcorner]\,[\llcorner\,X\,\llcorner]\,[0\,\breve{X}\,\llcorner]\,[\breve{1}\,\llcorner\llcorner]\,[0\,\llcorner\,\breve{Y}]\,[\llcorner\llcorner\,X]\,\#$

Figura 3.5.: Dos configuraciones de $\widetilde{M}$ para una configuración de M

[7] Basada en la demostración del teorema 3.13 de [40].

Observemos que varias configuraciones de $\widetilde{M}$ son posibles, dado que las k cintas son independientes (no importa cómo las "apilamos", y esto no tiene efecto en el proceso de cálculo). Esta característica de nuestra codificación se evidencia en la figura 3.5. Sin embargo, esta independencia hace que esta peculiaridad no nos afecte: simplemente dejamos que la máquina vaya computando y vamos aplicando las transiciones y modificando las transiciones según se dicte.

Describir de forma exacta el conjunto de estados $\widetilde{Q}$ y la función de transición $\widetilde{\delta}$ es un proceso arduo, que se detalla en el apéndice B (véase la proposición B.1). En esta demostración, nos limitaremos a explicar su comportamiento.

La máquina comienza con la configuración inicial para la entrada: $u = u_1\, u_2 \ldots u_n \in A^*$:

$$q_0 \ : \ \boxed{\#}\,[\widetilde{u_1}\ \underbrace{\check{\sqcup}\ldots\check{\sqcup}}_{k-1)}]\,[u_2\ \underbrace{\sqcup\ldots\sqcup}_{k-1)}]\ldots[u_n\ \underbrace{\sqcup\ldots\sqcup}_{k-1)}]\,\#$$

A continuación, va aplicando los movimientos de δ cinta a cinta y cabezal a cabezal. Llamaremos a estas cintas y cabezales "cintas virtuales" y "cabezales virtuales", respectivamente.

Para ello, comienza escaneando toda la cinta desde el delimitador # izquierdo, y almacena los valores de los cabezales virtuales en estados de $\widetilde{Q}$. El conjunto $\widetilde{Q}$ es ampliado para poder efectuar esta acción de "memorización". A continuación, volvemos al principio de la cinta, y vamos efectuando cada una de las acciones de M, por cada cinta virtual. Si en algún momento, al desplazar cada cabezal virtual nos lo coloca sobre un delimitador #, reemplazamos la posición de la cinta por el símbolo blanco de $\widetilde{M}$, $[\sqcup^k]$, y desplazamos # a la izquierda o derecha según sea necesario.

Una vez hecho esto, cambiamos el estado inicial por el estado correspondiente según δ y así sucesivamente. El conjunto F es el mismo dado que vamos pasando por los mismos estados de Q en $\widetilde{Q}$.

Es fácil ver que $L(M) = L(\widetilde{M})$, dado que hemos creado una máquina que imita a la perfección el proceso de cálculo de M, y en caso de parar, para en un estado de Q. Las palabras aceptadas serán aquellas en las que, una vez que la máquina haya parado, el estado esté en F. □

Pensemos un momento en las consecuencias de que estos dos modelos de cálculo sean equivalentes. En primer lugar, nos da una idea del alcance de la máquina de Turing monocinta: introduciendo más complejidad en su descripción, no podemos aspirar a crear algoritmos más complejos, pues siempre podremos encontrar un algoritmo equivalente en una máquina de una sola cinta. Esto, como comentábamos al comienzo del capítulo, nos da una pista de una de las tesis más relevantes de las ciencias de la computación en la actualidad, y que discutimos en la sección 3.4.

En segundo lugar, esto nos da una ventaja a la hora de querer resolver problemas: poder describir un programa en un modelo que nos da una mayor flexibilidad es mucho más sencillo.

Al avanzar en esta sección, es importante mantener en mente estas dos observaciones. Ahora, introduciremos un nuevo modelo computacional: la máquina de acceso aleatorio. [31]

Definición 3.12 (Máquina de Turing de acceso aleatorio)**.** Una *máquina de Turing de acceso aleatorio* con k cintas es una máquina de Turing multicinta de k cintas con dos cintas adicionales, limitadas a la izquierda:

- Una *cinta de dirección*, que almacena un número natural $n \in \mathbb{N}$.[8]
- Una cinta llamada *RAM*, que almacena símbolos en cada una de sus pistas.

Denotamos como RAM$[n]$ a la n-ésima pista de la RAM.

La función $\delta : Q \times \mathbb{N} \times B \times B^k \longrightarrow \left(Q \times \mathbb{N} \times B \times (B \times \{I, D, S\})^k\right) \cup \varnothing$ será de la forma:

$$\delta(q, b_1, b_2, ..., b_k) = \varnothing$$

para las transiciones no definidas, y para las definidas:

$$\delta(q, n, r, b_1, b_2, ..., b_k) = (p, m, s, c_1, M_1, c_2, M_2, ..., c_k, M_k),$$
$$\text{con } q, p \in Q, n, m, \in \mathbb{N}, r, s, b_i, c_i \in B, M_i \in \{I, D, S\},\ \forall i \in \{1, 2, ..., k\}$$

La transición definida se ejecutará si el valor de RAM$[n] = r$, y tras ejecutarla el valor de RAM$[n]$ cambiará a s, y el valor de la cinta de dirección cambiará de n a m. Es decir, en una única transición, es capaz de acceder a la posición de la RAM, modificarla, y cambiar la posición de la RAM.

Una configuración de la máquina se especificará mediante:

$$q \ : \ r \, ; \, w_1 \, , \, w_2 \, , \, \dots \, , \, w_k$$

con un símbolo de cada cinta $w_1, w_2, ..., w_k$ debidamente recuadrado, y con un símbolo de r recuadrado, concretamente, aquel al que apunte la cinta de dirección.

Podemos ver un ejemplo de tal configuración en la figura 3.6.

Probar que una máquina de Turing de acceso aleatorio puede ser simulada mediante una máquina de Turing multicinta es sencillo pero tedioso. En la proposición 3.2 se explican las rutinas que serían necesarias para tal simulación.

Proposición 3.2. *Toda máquina de Turing de acceso aleatorio de k puede ser simulada por una máquina de Turing multicinta.*

Demostración. Para probar esto, necesitaremos una máquina de Turing de $k + 2$ cintas. Las dos últimas cintas serán la cinta de dirección ($k + 1$-ésima cinta) y la RAM ($k + 2$-ésima cinta). A continuación, vemos que:

(1) Podemos crear una rutina que reemplace el símbolo de la posición del cabezal en la cinta $k + 2$ por otro.

(2) Podemos crear una rutina que modifique la dirección de la cinta $k + 1$.

(3) Podemos crear una rutina que, dada una dirección en la cinta $k + 1$, mueva el cabezal de la cinta $k + 2$ a la posición correspondiente. Podríamos implementar una rutina que vaya contando hasta el número de la cinta $k + 1$ y, cada vez que va contando, va moviendo el cabezal de la cinta $k + 2$ una posición a la derecha, comenzando al principio.

[8] Suponemos $0 \notin \mathbb{N}$.

Implementar la operación adicional atómica de la máquina de Turing de acceso aleatorio en esta máquina multicinta es sencillo: podemos acceder al valor de RAM$[n]$, r, viendo dónde se encuentra el cabezal en la cinta $k+2$. En función de esto, ejecutamos la transición necesaria. Tras esto, modificamos RAM$[n]$ a s ejecutando la rutina (1), y cambiamos la dirección de la cinta $k+1$ ejecutando (2). Finalmente, modificamos la posición del cabezal en la cinta $k+2$ ejecutando la rutina (3). □

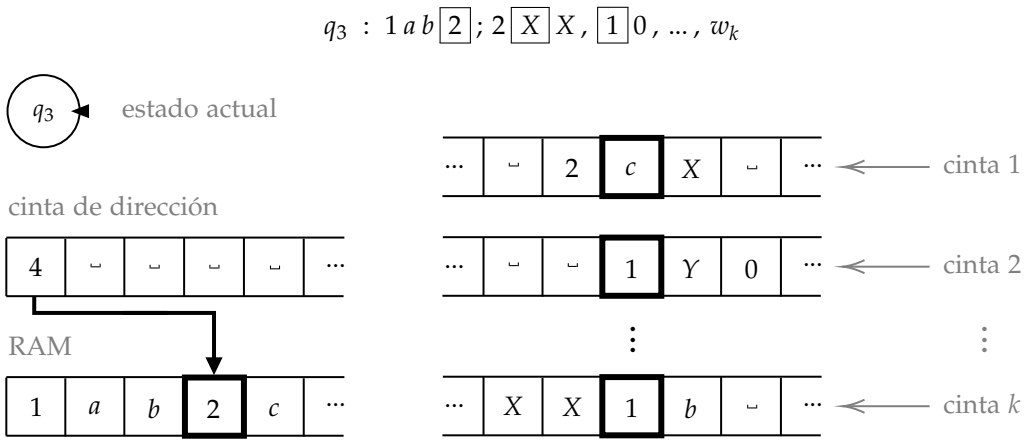


Figura 3.6.: Máquina de Turing de acceso aleatorio

A continuación, continuaremos con la cadena de simulaciones. Para ello, introduciremos una definición de *ordenador*. Ésta es una simplificación de la arquitectura de un ordenador pero, a efectos prácticos, todos los ordenadores modernos siguen la estructura que se especifica en la definición 3.13. [35]

Definición 3.13 (Ordenador moderno)**.** Definimos un *ordenador moderno* como aquel que dispone de los siguientes elementos:

- **Registros.** Un registro permite almacenar exactamente una unidad de información, en binario. Los registros tienen un tamaño fijo (en la mayoría de ordenadores actuales es de 64 bits). Esta unidad de información puede representar caracteres, instrucciones, etc.
- **RAM.** La RAM o memoria de acceso aleatorio (*Random Access Memory*) es el hardware del ordenador que almacena la información en uso por el ordenador.
- **ROM**. La ROM o memoria de solo lectura (*Read-Only Memory*) es el hardware del ordenador que almacena información, pero que no puede ser modificada. Normalmente, la ROM almacena programas fijos que indican al ordenador acciones básicas tales como cargar los programas en el encendido.
- **Repertorio de instrucciones.** Cada CPU contiene un conjunto fijo de acciones que puede ejecutar. Cada una de estas acciones se llama *instrucción*, y el conjunto de todas las instrucciones posibles es el *repertorio* de instrucciones. Estas instrucciones suelen ser

operaciones aritméticas y condicionales con los registros, cargar de la RAM a los registros, etc.

Para nuestro trabajo, saber detalles sobre qué tipos de instrucciones puede ejecutar un ordenador es irrelevante. Lo importante es saber que, independientemente del lenguaje de programación en el que esté escrito, un programa puede ser "traducido"[9] a este conjunto de instrucciones.

- **Disco.** Un ordenador tiene uno o varios discos (por simplicidad, hablaremos sólo de uno). El disco permite lectura y escritura, y almacena las entradas y salidas a los programas, así como los programas que quieran ejecutarse.

Simular este ordenador mediante una máquina de Turing de acceso aleatorio es inmediato: podemos simular todos los componentes mediante cintas y la RAM.

Proposición 3.3. *Un ordenador moderno puede ser simulado por una máquina de Turing de acceso aleatorio.*

Demostración. Veamos cómo podemos simular cada uno de los elementos del ordenador C mediante una máquina de Turing de acceso aleatorio M. En la figura 3.7 se simbolizan las afirmaciones siguientes.

- La ROM de C puede ser simulada incluyendo los estados y transiciones necesarios en M.
- Cada instrucción de C puede implementarse del mismo modo, mediante transiciones y estados.
- Los registros de C que sean usados para guardar direcciones de memoria pueden ser simulados por la cinta de direcciones de M.
- Otros registros de propósito general de C pueden ser simulados incorporando cintas adicionales a M.
- La RAM de C puede ser simulada con la RAM de M.
- El disco de C puede ser simulado con una cinta en M (que podemos llamar *cinta de entrada/salida*).

□

La demostración anterior no es estricta, y hacemos esto a propósito: definir de forma concreta cómo implementar cada uno de los componentes del ordenador queda fuera del objeto de este trabajo, y sería más relevante si hablásemos de arquitectura de computadores.

Ejemplo 3.2. Vamos a simular una instrucción de un ordenador moderno en una máquina de Turing de acceso aleatorio. Sea `ADD` la instrucción de suma:

```
ADD R0, R1, R2
```

[9] Mediante procesos de compilación y/o interpretación.

Esta instrucción toma los números en binario de los registros `R0` y `R1` y guarda su suma en el registro `R2`. Por ejemplo, si hacemos

```
ADD r4, r5, r9
```

almacenaremos la suma de los registros `r4` y `r5` en el registro `r9`.

Antes de realizar la simulación, observemos una cosa: si el tamaño de cada uno de los registros es de n bits, podemos hacer que la RAM esté compuesta de m registros de n bits y que la cinta de dirección haga que el cabezal de la RAM apunte al primer bit del registro correspondiente. Dicho esto, para realizar la simulación basta seguir el siguiente proceso:

1. Colocamos en la cinta de dirección la dirección correspondiente al registro `R0`. De este modo, el cabezal de la RAM apuntará al primer bit del registro `R0`.
2. Copiamos los contenidos de la RAM (los n bits a partir de la posición del cabezal de la RAM) en una cinta nueva, a la que llamaremos C_0.
3. Repetimos los pasos anteriores para copiar el registro `R1` en la cinta C_1.
4. Vamos repitiendo los pasos siguientes:
   a) Comprobamos si en la cinta C_1 el número almacenado es 0. En tal caso, seguimos al paso 5.
   b) Si no lo es, decrementamos en 1 el número almacenado en la cinta C_1 e incrementamos en 1 el número almacenado en la cinta C_0. Sumar y restar 1 a un número en binario en una cinta es una operación fácilmente implementable en una máquina de Turing.
5. Llegado a este punto, en la cinta C_1 tendremos almacenado 0 y en la cinta C_0 tendremos el resultado de la suma.
6. Ahora colocamos en la cinta de dirección la dirección correspondiente al registro `R2`. De este modo, el cabezal de la RAM apunta al registro `R2`.
7. Copiamos los contenidos de la cinta C_0 a la RAM (los n bits a partir de la posición del cabezal de la RAM, que apuntarán al registro `R2`).

Nótese que la simulación que hemos hecho no modifica los registros `R0` y `R1`, únicamente modifica `R2`.

El resultado que buscamos está a la vuelta de la esquina. Las proposiciones 3.4 y 3.5 cierran el círculo y permiten probar el teorema 3.1.

Proposición 3.4. *Un programa en Python puede ser simulado por un ordenador moderno.*

Demostración. La prueba es inmediata: en el momento en el que podemos simular un ordenador moderno de un solo núcleo, podemos simular cualquier software, incluyendo programas en Python.

Otra forma de ver esto es imaginar que configuramos un ordenador para ejecutar un programa en Python justo al iniciarse. De este modo, podemos pensar en el ordenador como un conjunto de hardware que simula el programa en Python. □

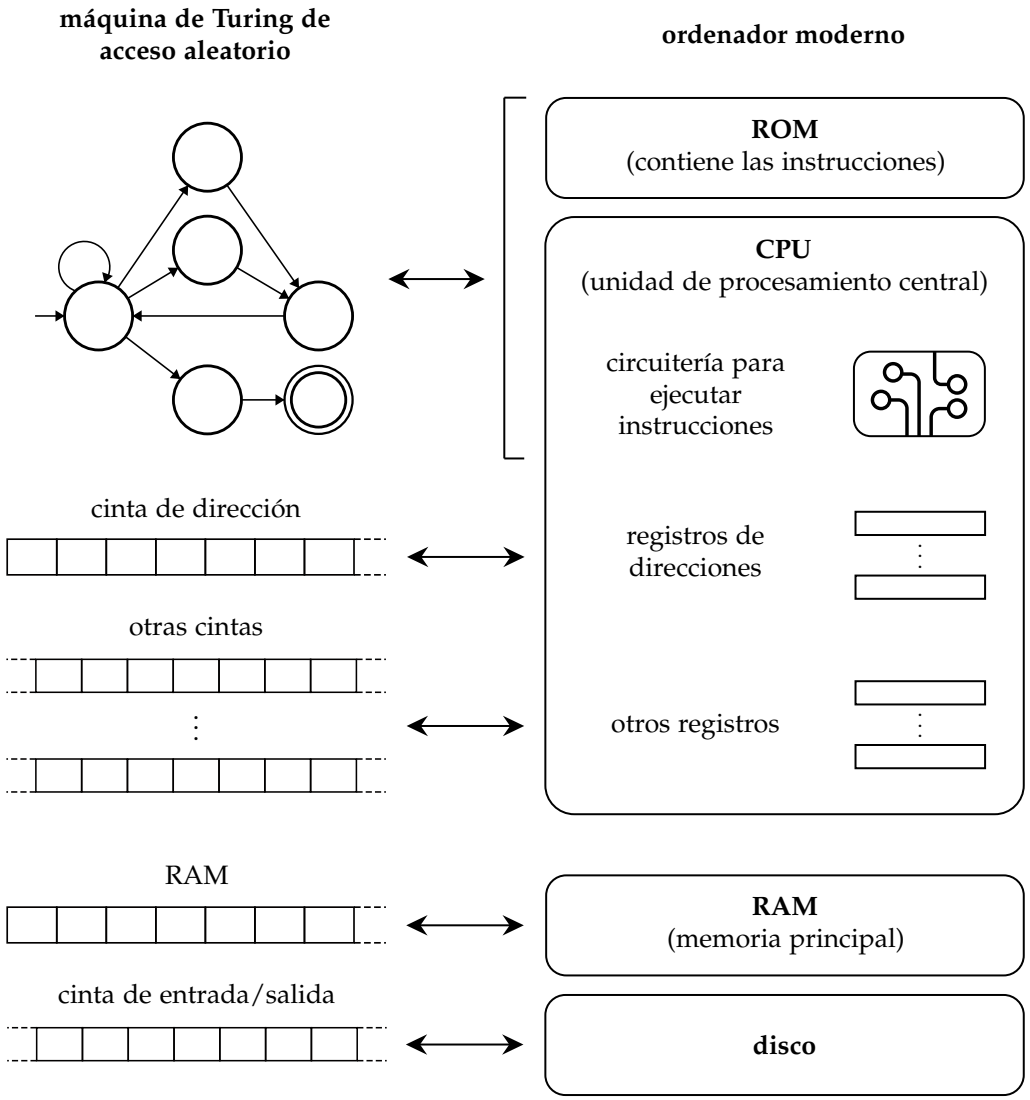


Figura 3.7.: Simulación de un ordenador moderno a través de una máquina de Turing de acceso aleatorio

Proposición 3.5. *Una máquina de Turing de una sola cinta puede ser simulada por un programa en Python.*

Demostración. Para probar esto, deberemos de crear un programa en Python que, dada la descripción de una máquina de Turing de una sola cinta $M =$, sea capaz de aplicar todos los pasos de cálculo y devolver la configuración en la que la máquina para.

```
1  import utilidades
2  from turing import Turing

4  def simula_turing(entrada):
5      codificacion_maquina, entrada_maquina = utilidades.UAM(entrada)

7      # creamos la máquina de Turing a partir de la entrada de la función SISO
8      maquina_turing = Turing(codificacion_maquina, entrada_maquina)

10     # ejecutamos la máquina
11     maquina_turing.ejecutar()

13     # si llegamos aquí, la máquina ha parado: devolvemos su configuración
14     return str(maquina_turing)
```

Programa 3.2: `simula_turing.py`

Podemos ver la simulación en el programa 3.2. Hay más información sobre dónde encontrar y cómo usar el código de este trabajo en el apéndice A. Explicaremos cada una de las líneas.

En primer lugar, en la línea 2, se importa la clase `Turing` del paquete `turing`. Puedes encontrar el código de `turing.py` en la carpeta `codigo`. Esta clase se encarga de instanciar una máquina de Turing, e incorpora las funciones necesarias para realizar la simulación.

En la línea 4 tenemos nuestra función *main*, que acepta un *string* como entrada. Para poder codificar varias entradas, recurrimos a las funciones `MAU` y `UAM` de la librería `utilidades`. Dado una lista de *strings*, la función `MAU` (de *Múltiple A Uno*) los convierte en un único *string* usando la cadena `'::'` como separador. La función `UAM` realiza el proceso contrario, y convierte un *string* en una lista de *strings*, separando por la cadena `'::'`.

De este modo, el *string* de entrada tendrá la forma

$$\texttt{CODIFICACIÓN} + \texttt{'::'} + \texttt{ENTRADA}$$

Es decir, estará compuesto de los *strings* correspondientes a la codificación de la máquina de Turing y a la entrada unidos por `'::'`.

Codificar una máquina de Turing M mediante un *string* es sencillo. Lo haremos mediante archivos, que no son más que cadenas de caracteres. Codificaremos cada uno de los estados y símbolos con *strings*, y crearemos una cadena de caracteres uniendo las siguientes cadenas de caracteres mediante saltos de línea, en este orden:

- Los estados, separados por espacios.

- Los símbolos del alfabeto de entrada, separados por espacios.
- Los símbolos del alfabeto de trabajo, separados por espacios.
- El símbolo blanco.
- El estado inicial.
- Los estados finales, separados por espacios.
- Una línea para cada transición

$$\delta(\texttt{EA}, \texttt{SL}) = (\texttt{ES}, \texttt{SR}, \texttt{M})$$

(donde `EA` es el estado actual, `SL` es el símbolo leído, `ES` es el estado siguiente, `SR` es el símbolo de reemplazo y `M` es el movimiento, pudiendo `M` ser `I`, `D` o `S`), con el siguiente formato:

```
EA, SL : ES, SR, M
```

En la línea 5, separamos la entrada y guardamos la codificación de la máquina en `codificacion_maquina` y la entrada en `entrada_maquina`, mediante la función `UAM` anteriormente explicada.

En la línea 8, creamos una instancia de la clase `Turing` con la codificación de la máquina y la entrada. Al crear la instancia, se ejecutarán funciones que comprueban si la descripción es correcta. En caso de haber un error, la máquina no ejecutará ningún paso y la salida de la línea 14 será de tipo error.

En la línea 11, ejecutamos la máquina, aplicando transición tras transición hasta que pare. En caso de no parar, el programa en Python tampoco parará.

En la línea 14, devolvemos la configuración de la máquina al haber parado, y si acepta o rechaza.

De esta forma hemos simulado M mediante un programa en Python. □

El programa `simula_turing` está implementado y se encuentra disponible en el repositorio de este trabajo (véase el apéndice A). Mostraremos un ejemplo de la codificación explicada, y cómo usarlo, mediante el ejemplo 3.3.

Ejemplo 3.3. Una codificación de la máquina $M_{\#a>\#b}$ del ejemplo 3.1 puede ser:[10]

```
1  q_0 q_a q_b q_R q_F   # estados
2  a b                   # alfabeto de entrada
3  a b X _               # alfabeto de trabajo
4  _                     # símbolo blanco
5  q_0                   # estado inicial
6  q_F                   # estados finales

8  # transiciones
9  q_0, a : q_a, X, D
```

[10] Incluimos comentarios y líneas vacías, que ignoraremos

```
10 q_0, b : q_b, X, D
11 q_0, X : q_0, X, D
12 q_a, a : q_a, a, D
13 q_a, b : q_R, X, I
14 q_a, X : q_a, X, D
15 q_a, _ : q_F, _, S
16 q_b, a : q_R, X, I
17 q_b, b : q_b, b, D
18 q_b, X : q_b, X, D
19 q_R, a : q_R, a, I
20 q_R, b : q_R, b, I
21 q_R, X : q_R, X, I
22 q_R, _ : q_0, _, D
```

Esta codificación se encuentra en el archivo `codigo/maquinas_turing/mas_a_que_b.mt`. Para probar `simula_turing.py`, basta abrir el intérprete de Python en la carpeta `codigo` y ejecuta las siguientes sentencias:

```
>>> from utilidades import leer, MAU
>>> from simula_turing import simula_turing
>>> codificacion = leer('./maquinas_turing/mas_a_que_b.mt')
>>> entrada = 'aababb'
```

En este caso hemos puesto *aababb* como entrada, pero puedes reemplazarla por cualquier entrada de $\{a, b\}^*$. Para ejecutar la función `simular`, simplemente ejecuta:

```
>>> simula_turing(MAU(codificacion, entrada))
'q_0 : X X X X X X [_] (rechaza)'
```

Vemos que el programa Python devuelve la cadena de caracteres

```
'q_0 : X X X X X X [_] (rechaza)'
```

Es decir, la máquina de Turing $M_{\#a>\#b}$ para en el estado

$$q_0 \ : \ X\,X\,X\,X\,X\,X\,\boxed{\sqcup}$$

que no es un estado final ($q_0 \notin F$), y por tanto la entrada *aababb* es rechazada.

Puedes probar con cualquier entrada y con otras máquinas de Turing de la carpeta `codigo/maquinas_turing`. Algunos de los archivos corresponden a codificaciones incorrectas, para que puedas ver cómo `simula_turing.py` es un programa en Python tal y como lo hemos definido (SISO, sin errores, etc.).

Una vez hemos probado la secuencia de simulaciones de la figura 3.3, el teorema siguiente queda demostrado.

Teorema 3.1 (Equivalencia entre programas en Python y máquinas de Turing)**.** *Dado un programa en Python P, existe una máquina de Turing (de una sola cinta) M que computa la misma función que P.*

Acabamos de probar que las máquinas de Turing y los programas en Python son iguales en términos de los programas que podemos escribir en ellos. Esto nos será útil cuando definamos el concepto de *problema* en el capítulo 4.

3.4. Relación con la tesis de Church-Turing

El trabajo que hemos realizado en este capítulo ha sido arduo, pero el resultado que hemos obtenido es extremadamente interesante. Sin embargo, debemos recordar que la elección del lenguaje de programación Python ha sido completamente arbitraria: podríamos sustituir a Python por cualquier otro lenguaje de programación capaz de codificar una máquina de Turing.

¿Qué nos dice esto de lo que puede ser computado? Podemos verlo de dos formas: por una parte, los programas en Python (así como los ordenadores en sí) no son más que máquinas de Turing con "muchos atajos" y extensiones. Por otra (y de forma más interesante), la máquina de Turing, que inicialmente puede parecer inocente por la sencillez de su descripción, tiene un potencial de cálculo enorme, pues puede ejecutar cualquier programa de ordenador.

En el capítulo 2 ya expusimos la *tesis de Church-Turing*. Por conveniencia, la referenciamos de nuevo a continuación.

> **Tesis de Church-Turing.** *Toda función calculable (algoritmo) puede calcularse mediante una máquina de Turing*

Como hemos dicho, se trata de una tesis, es decir, es una afirmación que se supone pero que no se demuestra. Probar este resultado es prácticamente imposible, pero hay evidencias que lo refuerzan, como lo que acabamos de hacer: hemos visto cómo un modelo de computación, en apariencia, mucho más complejo que la máquina de Turing, resulta ser equivalente a ella.

Sin embargo, siempre existe la posibilidad de que avances en la comprensión de cómo se comporta el universo nos abran las puertas a nuevos y más potentes modelos de cálculo. Este no es el caso de la computación cuántica, que puede acelerar la resolución de algunos problemas, pero no amplía la gama de funciones calculables [25].

Esta tesis tiene profundas implicaciones. Concluiremos este capítulo con algunas preguntas.

1. ¿Es la mente humana una máquina de Turing?
2. ¿Es el universo una máquina de Turing? Esto es, ¿son las leyes que rigen el universo (las leyes de la física) computables?
3. En caso de que esto no sea así, ¿es posible crear una máquina más poderosa que la de Turing?

Respecto a la primera pregunta, es importante realizar una puntualización histórica: Turing afirmaba que la respuesta era afirmativa, es decirm que la mente humana podría simularse en un ordenador. Esta tesis fue central en sus trabajos de Inteligencia Artificial [43], en los que defendía que el desarrollo de esta disciplina solo era cuestión de tiempo.

4. El problema de la parada

> *La ciencia de la computación no trata sobre ordenadores de la misma manera en que la astronomía no trata sobre telescopios.*
>
> — Edsger W. Dijkstra, [16]

En el capítulo anterior, mencionamos que las máquinas de Turing y los programas en Python son equivalentes en términos de los "problemas" que pueden resolver. En este capítulo, definiremos formalmente la noción de problema y nos adentraremos en uno de los ejes centrales de la teoría de la computación: la decidibilidad (sección 4.1).

A continuación, procederemos a introducir la máquina universal (sección 4.2), un concepto que introdujo Turing al presentar su modelo de computación [44]. Esta máquina nos servirá para encontrar problemas no decidibles (sección 4.3). Finalmente, introduciremos el concepto de reducción (sección 4.5), que nos permitirá encontrar aun más problemas no decidibles.

Haremos todo esto para deducir el resultado principal de este capítulo: la no decidibilidad del problema de la parada (sección 4.6). Este problema es extremadamente relevante, no solo a nivel histórico, sino será uno de los pilares de la demostración prinicipal de este trabajo, que haremos en el capítulo 6.

4.1. Problemas decidibles

Comenzamos, como hemos prometido, definiendo de forma precisa qué es un problema.

Definición 4.1 (Problema computacional)**.** Un *problema computacional* (o *problema*) F[1] es una función $F : X \longrightarrow Y$, donde:

- X es el conjunto de *entradas*. Un elemento $x \in X$ se llama una *entrada*.
- Y es el conjunto de *soluciones*. Un elemento $y \in Y$ se llama una *solución*.

Dado un $x \in X$ llamamos a $F(x)$ la *solución* de x.

Un problema no es más que una función que mapea entradas a salidas, a las que llamamos soluciones. Cualquier problema puede reducirse a esta terminología, por muy complejo que sea. Veamos la primera descripción de un problema.

[1] Llamamos a los problemas computacionales F y no P para distinguirlos de los programas en Python.

CicloHamiltoniano

- **Entrada:** un grafo G.
- **Solución:** los ciclos hamiltonianos presentes en G.

Problema 4.1: CicloHamiltoniano

Vamos a definir los conjuntos de entrada y salida del problema 4.1. La entrada a nuestro problema es un grafo G, que no es más que un conjunto de vértices V y una matriz de adyacencia A representando sus arcos ($G = (V, A)$). Como salida, tendremos un conjunto de n caminos en G, $\pi_1, \pi_2, ..., \pi_n$ (es posible que un grafo tenga ningún, uno o varios ciclos hamiltonianos). Podemos especificar los caminos como subconjuntos ordenados de V.

Figura 4.1.: Esquema de CicloHamiltoniano

Para probar los resultados de nuestro trabajo, utilizaremos un subconjunto de problemas: aquellos cuya salida es únicamente *'sí'* o *'no'*. Los llamaremos *problemas de decisión*.

Definición 4.2 (Problema de decisión)**.** Dado un alfabeto A, un *problema de decisión*[2] es un problema computacional $F : X \longrightarrow Y$ en el que $Y = \{\text{'sí'}, \text{'no'}\}$.

Cualquier problema computacional puede "transformarse" en un problema de decisión preguntándonos por la existencia de soluciones. Un ejemplo de esto lo vemos en el problema 4.2.

HayCicloHamiltoniano

- **Entrada:** codificación de un grafo G.
- **Solución:** si hay al menos un ciclo hamiltoninano en G.

Problema 4.2: HayCicloHamiltoniano

Este problema es de decisión, pues no pregunta explícitamente por *cuáles* son los ciclos hamiltonianos presentes en un determinado grado, sino por si *existe* un ciclo hamiltoniano en él.

[2] Al referirnos a un "problema" sin especificar si es computacional o de decisión, no entraremos en ambigüedad: basta ver las salidas para saber si es de decisión.

Evidentemente, este tipos de problemas son más restrictivos que los problemas computacionales generales, pero tienen mucho más interés a la hora de hablar de decidibilidad, como veremos más adelante.

Figura 4.2.: Esquema de HayCicloHamiltoniano

En el caso de los problemas de decisión, tiene sentido hablar de su complementario.

Definición 4.3 (Problema complementario)**.** Dado un problema de decisión D, definimos su *problema complementario* (o *problema contrario*), denotado $\overline{D}$, como:

$$\overline{D}(a) = \text{'sí'} \iff D(a) = \text{'no'}, \qquad \overline{D}(a) = \text{'no'} \iff D(a) = \text{'sí'}$$

En algunas ocasiones, usaremos la notación C- para referirnos al problema complementario: dado un problema Problema, llamaremos C-Problema a su problema contrario.

Para continuar, es necesario comentar algunas particularidades de los programas en Python para tratar con problemas. Como bien sabemos, los programas sólo aceptan *strings* como entrada y *strings* como solución. Podemos pensar en un programa en Python P como en una función

$$P : S \longrightarrow S$$

donde S es el conjunto de todos los *strings* posibles en Python. Dependiendo de la versión de Python, este conjunto será mayor o menor.[3]

Para que un ordenador pueda resolver un problema, necesitaremos codificar las entradas y las soluciones del problema mediante un formato que pueda entender: las cadenas de caracteres (*strings*). Tal y como se demuestra en la proposición B.2, para cualquier problema podremos suponer que las entradas y soluciones están codificadas mediante *strings*.

Esto no supone una restricción: las entradas y las soluciones pueden ser codificadas como cadenas de caracteres. Deberemos, sin embargo, tener en cuenta que es posible que haya entradas que no sean una codificación correcta de una entrada del problema original. En tal caso, asignaremos la salida `'no'` a tal entrada.

Vemos un ejemplo de esto en el problema 4.3. Este problema tendrá asociada una cierta codificación para las entradas y las soluciones. Observa que este problema es idéntico al problema 4.1, y que podemos traducir las entradas y las soluciones de uno y otro conociendo la codificación elegida.

[3] En las últimas versiones de Python, es posible trabajar con UTF-8.

CicloHamiltonianoCodificación

- **Entrada:** codificación de un grafo G.
- **Solución:** codificación de los ciclos hamiltonianos presentes en G.

Problema 4.3: CicloHamiltonianoCodificación

En el caso de los problemas de decisión, las soluciones siempre serán `'sí'` y `'no'`. Nótese que `'no'` será solución tanto para las instancias negativas del problema como para las codificaciones incorrectas.

De aquí en adelante, todos los problemas tendrán cadenas de caracteres como entradas y soluciones.

A continuación, concretaremos la relación entre programas y problemas, que ya veníamos intuyendo desde el capítulo 3.

Definición 4.4 (Resolver y decidir)**.** Dado un problema $F : S \longrightarrow S$ y un programa P, diremos que P *resuelve* F si, para cada entrada $I \in S$, P para y devuelve un *string* tal que $P(I) = F(I)$.

Si F es un problema de decisión resuelto por P, diremos que P *decide* F.

Un problema computacional describe lo que queremos resolver. Un programa describe la forma particular de computar la solución a un problema.

Es muy importante destacar que, en la definición anterior, es necesario que P *pare* para todas las entradas. O, lo que es lo mismo, si P cicla para alguna entrada, no resuelve F. Esta condición puede relajarse dando lugar a problemas *semidecidibles* (véase la sección 4.4).

Esta característica de "resolver" es realmente una propiedad del problema, y no del programa. En el caso de los problemas de decisión, esto nos conduce a la definición de *computabilidad*.

Definición 4.5 (Problema computable y problema decidible)**.** Sea F un problema computacional. Si existe un programa P que lo resuelve, decimos que el problema es *computable*.

En el caso en el que D sea un problema de decisión, si existe un programa P que lo decida decimos que el problema es *decidible*.

Veamos un problema más sencillo que el problema 4.1. El problema 4.4, MásAQueB, tiene como entrada una palabra del alfabeto {`'a'`,`'b'`}, y como solución `'sí'` en caso de que la palabra tenga más `'a'` que `'b'`, y `'no'` en caso contrario o en el caso de que la codificación sea incorrecta. Para este problema, la codificación es incorrecta si la entrada es una cadena de caracteres que no conforma una palabra del alfabeto {`'a'`,`'b'`}.

Evidentemente, MásAQueB es un problema de decisión. Ahora nos queda preguntarnos si este problema es decidible. La respuesta es afirmativa, y se demuestra en detalle en la proposición 4.1.

MásAQueB

- **Entrada:** una cadena de caracteres del alfabeto {'a','b'}.
- **Salida:** si la cadena de caracteres tiene más 'a' que 'b'.

Problema 4.4: MásAQueB

Si queremos saber si el problema es decidible, deberemos escribir un programa que lo resuelva y nunca cicle, y que tenga en cuenta las codificaciones incorrectas. Esto es precisamente lo que hace el programa a continuación:

```
1  import utilidades
2
3  def mas_a_que_b(palabra):
4      # comprobamos si la codificación es correcta, es decir (si es del
           alfabeto {'a', 'b'}), si no es correcta devolvemos 'no'
5      if not utilidades.en_alfabeto(palabra, {'a', 'b'}):
6          return 'no'
7
8      if palabra.count('a') > palabra.count('b'):
9          return 'sí'
10
11     return 'no'
```

Programa 4.1: mas_a_que_b.py

Explicamos el programa brevemente. Vemos en la línea 3 la función *main* del programa, que acepta una palabra como entrada. En la línea 5 comprobamos que la palabra pertenezca al alfabeto {'a','b'} mediante la función en_alfabeto de la librería utilidades (que importamos en la línea 1). En caso de que la palabra no pertenezca al alfabeto, devolvemos 'no', pues la codificación es incorrecta. Si pertenece al alfabeto, llegamos a la línea 8 y comprobamos si tiene más 'a' que 'b'. En caso de ser así, devuelve 'sí' en la línea 9 y, si no, devuelve 'no' en la línea 11. Vemos algunas de las salidas del programa mas_a_que_b.py en la tabla 4.1.

Comando	Salida
mas_a_que_b('abaab')	'sí'
mas_a_que_b('abbab')	'no'
mas_a_que_b('')	'no'
mas_a_que_b('abaabc')	'no'

Tabla 4.1: Ejemplos de salidas de mas_a_que_b.py

Dado que hemos encontrado un programa `mas_a_que_b.py` que decide MásAQueB, el problema MásAQueB es decidible.

Proposición 4.1. *El problema* MásAQueB *es decidible.*

Demostración. El programa 4.1 decide MásAQueB. En efecto, vemos que el programa para para todas las entradas $I \in S$ y devuelve:

- Si $I \notin \{\texttt{'a'}, \texttt{'b'}\}$, es $P(I) = \text{MásAQueB}(I) = \texttt{'no'}$ (entrada con codificación incorrecta).
- Si $I \in \{\texttt{'a'}, \texttt{'b'}\}$:
  - Si I tiene más `'a'` que `'b'`, es $P(I) = \text{MásAQueB}(I) = \texttt{'sí'}$.
  - Si I no tiene más `'a'` que `'b'`, es $P(I) = \text{MásAQueB}(I) = \texttt{'no'}$.

□

Una nota respecto a nuestra definición de decidibilidad. Normalmente, en los cursos de teoría de la computación, esta definición se hace respecto a máquinas de Turing. Sin embargo, al haber probado la equivalencia con los programas en Python en el capítulo 3, podemos definirlo directamente en términos de programas. Según la definición de máquinas de Turing, sin embargo, también podríamos probar que MásAQueB es decidible: basta ver que la máquina $M_{\#a>\#b}$ del ejemplo 3.1 decide el problema, pues nunca para y acepta las palabras que tienen más a que b.

4.2. Universalidad

En la sección 3.3 vimos cómo podemos simular una máquina de Turing mediante un programa en Python. Pero, ¿es posible simular un programa en Python mediante un programa en Python? La respuesta es sí, y no debería resultar sorprendente, pues muchas de las operaciones que ejecutamos en el día a día en nuestros ordenadores involucran este tipo de simulaciones.

Por ejemplo, al ejecutar un programa en Python cualquiera, estamos ejecutando el intérprete de Python, que es un programa en sí mismo.

Esta idea de "programas que ejecutan programas" puede parecer obvia, pero realmente fue revolucionaria en su tiempo, y es el motivo por el que los ordenadores personales son hoy la norma, llegando a encontrarse hasta en nuestros bolsillos. Quizás la cosa más increíble que puede hacer un ordenador es ejecutar *cualquier cosa* – no simplemente un conjunto de programas fijos preinstalados por el fabricante.

Para entender cómo esto puede ser posible en el caso de un programa en Python, deberemos de entender el funcionamiento de la función nativa `exec` [30]. Si probamos en el intérprete:

```
>>> exec("print('resultado:', 11+2)")
resultado: 13
```

Vemos cómo a `exec` pasamos una sentencia en Python en forma de *string*, y la ejecuta. De hecho, observa cómo ha ejecutado la suma $11 + 2$, y no simplemente ha imprimido `'11+2'`.

Esto podemos llevarlo un paso más allá: dentro de `exec` podemos insertar varias líneas e incluso definir funciones, variables, etc.

```
>>> exec("def hola(nombre):\n\tprint(f'Hola, {nombre}')\nhola('TFG')")
Hola, TFG
```

Una vez conocemos esto, podemos crear el siguiente programa:

```
1  import utilidades

3  def maquina_universal(programa, entrada):
4      # esto define las funciones del programa, pero no las ejecuta
5      try:
6          exec(programa)
7      except Exception as excepcion:
8          return 'error: ' + str(excepcion)

10     # extraemos una referencia a la función main, que está definida localmente
11     main = utilidades.extraer_main(programa, locals())

13     # invocamos la función
14     return main(entrada)
```

Programa 4.2: `maquina_universal.py`

Veamos qué es lo que hace el programa 4.2. En primer lugar, vemos en la línea 3 que la función *main* admite dos parámetros: un programa y una entrada, ambos *string*. En la línea 6 ejecutamos el programa definido en `programa`, tal y como explicamos anteriormente. Esta línea está dentro de un bloque `try-except` dado que es posible que el programa esté mal definido, en cuyo caso se lanza una excepción. Esta excepción la cazamos, dado que según la definición 3.2, el programa no debe lanzar ninguna excepción. En la línea 7 vemos que, de haber una excepción, entraremos a la línea 8, que devuelve un *string* con los detalles del error.

Si no hay error, la línea 6 se ejecutará satisfactoriamente, y las funciones que se definan en `programa` serán accesibles desde la función `universal` (línea 3). En concreto, todas las definiciones de `programa` serán accesibles desde `locals()`. Aprovechamos esto en la línea 11, donde extraemos una referencia a la función *main* de `programa`, que finalmente ejecutamos con la entrada `entrada` en la línea 14.

Hemos logrado crear un programa en Python que es capaz de ejecutar otro programa. Esto nos permitirá ejecutar cualquiera de los problemas que hemos creado, sin más que pasar el programa a un *string*. En la librería `utilidades` se encuentra la función `leer`, que dada una ruta a un archivo devuelve el archivo en *string*.

Algunos ejemplos de ejecución de `maquina_universal.py` se especifican en la tabla siguiente.

Comando	Salida
`maquina_universal('no es un programa', 'una entrada cualquiera')`	`'error...'`
`maquina_universal(leer('./mas_a_que_b.py'), 'abaab')`	`'sí'`
`maquina_universal(leer('./mas_a_que_b.py'), leer('./mas_a_que_b.py'))`	`'no'`
`maquina_universal(leer('./mas_a_que_b_v2.py'), leer('./mas_a_que_b.py'))`	`'sí'`
`maquina_universal(leer('./si.py'), leer('./mas_a_que_b.py'))`	`'sí'`
`maquina_universal(leer('./simula_turing.py'), utilidades.MAU(leer('./maquinas_turing/mas_a_que_b.mt'), 'abaab'))`	`'sí'`

Tabla 4.2: Ejemplos de salidas de `maquina_universal.py`

En la primera fila, podemos ver que, al insertar un programa inválido como entrada, la salida de `maquina_universal.py` será de error. En la segunda fila, ejecutamos el programa `mas_a_que_b.py` con la entrada `'aaba'`. El resultado es exactamente el esperado (observa la primera fila de la tabla 4.1).

En la tercera fila, usamos el propio código de `mas_a_que_b.py` como entrada para sí mismo. Esto tiene sentido, dado que un programa, como hemos comentado, no es más que un *string*. Recuerda que el programa 4.1 trata entradas fuera del alfabeto {`'a'`,`'b'`} como una entrada incorrecta, y es por ello por lo que la salida es `'no'`. Modificamos esto en el programa 4.3, en el que acpetamos cualquier entrada y contamos su número de `'a'` y `'b'`. Vemos cómo, en la cuarta fila, la salida es `'sí'`, dado que en este caso la codificación es correcta.

```
1  def mas_a_que_b_v2(palabra):
2      if palabra.count('a') > palabra.count('b'):
3          return 'sí'

5      return 'no'
```

Programa 4.3: `mas_a_que_b_v2.py`

En la quinta fila, utilizamos el programa 4.4. Este programa siempre devuelve `'sí'`, independientemente de la entrada que pasemos. Evidentemente, este programa es el que decide el problema 4.5 (es decir, Sí es decidible).

```
1  def si(entrada):
2      # siempre devuelve 'sí'
3      return 'sí'
```

Programa 4.4: `si.py`

Sí

- **Entrada:** una cadena de caracteres cualquiera.
- **Salida:** `'sí'`.

Problema 4.5: Sí

En la sexta y última fila vemos cómo `maquina_universal.py` puede ejecutar cualquier programa en Python, incluso el programa 3.2, que simula máquinas de Turing. Como entrada al programa pasamos la codificación de una máquina de Turing y la entrada. En efecto, la salida sigue siendo `'sí'`.

Esta capacidad de los programas de ejecutarse a sí mismos es extremadamente interesante y es una característica que nos causará muchos problemas, como veremos en la sección 4.3.

Llamamos a una máquina *universal* si es capaz de simular una máquina de Turing, dado que este concepto lo introdujo Turing en su famosa publicación [44]. Por el resultado de equivalencia (teorema 3.1), vemos que programa 4.2 es, en efecto, una máquina universal (algo que podíamos intuir con el nombre del programa).

Como vimos al principio, el concepto de universalidad es una constante en los ordenadores que usamos: el hardware es universal, porque puede ser usado para ejecutar cualquier máquina de Turing, del mismo modo que lo es un sistema operativo, pues puede compilar programas que simulan máquinas de Turing. El intérprete de Python también es universal por este mismo motivo, así como la máquina virtual de Java. Incluso un programa en sí mismo (programa 4.2) puede ser universal.

4.3. Problemas no decidibles

En la sección 4.1 mostramos cómo probar que un problema es decidible, lo cual es sencillo: basta encontrar un programa que lo decida. Sin embargo, probar que un problema no lo es es mucho más complicado. En este apartado encontraremos varios problemas no decidibles.

Universal

- **Entrada:** un programa P y una entrada I.
- **Salida:** `'sí'` si P está bien definido y $P(I) =$ `'sí'`, `'no'` en caso contrario.

Problema 4.6: Universal

Comencemos por el problema UNIVERSAL.[4] Claramente, se trata de un problema de decisión, pues devuelve `'sí'` o `'no'`. En esta sección, probaremos que escribir un programa que decida UNIVERSAL es imposible. En caso de que fuese posible hacerlo, vemos en la tabla 4.3 las posibles salidas que tendría.

Comando	Salida
`universal('no es un programa', 'una entrada cualquiera')`	`'no'`
`universal(leer('./mas_a_que_b_v2.py'), 'aaba')`	`'sí'`
`universal(leer('./mas_a_que_b_v2.py'), leer('./mas_a_que_b_v2.py')`	`'sí'`
`universal(leer('./si.py'), 'aaba')`	`'sí'`
`universal(leer('./si.py'), leer('./si.py'))`	`'sí'`

Tabla 4.3: Ejemplos de salidas de `universal.py`

En la primera fila vemos cómo, para codificaciones incorrectas, el programa devuelve `'no'`. Las filas siguientes son evidentes, y similares a las de la tabla 4.2.

Introduciremos dos problemas más. El primero de ellos, DIAGONAL, podemos obtenerlo a partir de UNIVERSAL (problema 4.6) con $I = P$.

DIAGONAL

- **Entrada:** un programa P.
- **Salida:** `'sí'` si $P(P)$ está bien definido y $P(I) =$ `'sí'`, `'no'` en caso contrario.

Problema 4.7: DIAGONAL

En caso de existir un programa que decida este problema, vemos en la tabla siguiente las posibles salidas que tendría.

Comando	Salida
`diagonal('no es un programa')`	`'no'`
`diagonal(leer('./mas_a_que_b_v2.py'))`	`'sí'`
`diagonal(leer('./si.py'))`	`'sí'`
`diagonal(leer('./diagonal.py'))`	¿`'sí'` o `'no'`?

Tabla 4.4: Ejemplos de salidas de `diagonal.py`

[4] Este problema no tiene, inicialmente, relación alguna con la máquina universal que definimos en el programa 4.2.

Las tres primeras salidas no requiren más explicación. Sin embargo, centrémonos en la última: ¿cuál sería el resultado de ejecutar `diagonal.py` con el código de `diagonal.py`?

Parece ser que tanto `'sí'` como `'no'` son ambas respuestas correctas – basta ver la definición del problema 4.7. Tomando I como `diagonal.py` y P como el mismo programa, entonces, si $P(I) =$ `'sí'`, el programa devolverá `'sí'`, mentras que si $P(I) =$ `'no'`, entonces el programa devolverá `'no'`.

Llegado a este punto deberíamos sospechar que las cosas no van a ir bien. Terminemos este "circo recursivo" con un último problema, C-Diagonal.[5] En caso de existir un programa que lo decida, `c_diagonal.py`, vemos sus posibles salidas en la tabla 4.5.

C-Diagonal

- **Entrada:** un programa P.
- **Salida:** `'no'` si $P(P)$ está bien definido y $P(I) =$ `'sí'`, `'sí'` en caso contrario.

Problema 4.8: C-Diagonal

Comando	Salida
`c_diagonal('no es un programa')`	`'sí'`
`c_diagonal(leer('./mas_a_que_b_v2.py'))`	`'no'`
`c_diagonal(leer('./si.py'))`	`'no'`
`c_diagonal(leer('./c_diagonal.py'))`	**¿**`'sí'` o `'no'`**?**

Tabla 4.5: Ejemplos de salidas de `c_diagonal.py`

Volvemos a repetir la misma pregunta que nos hicimos para el caso de Diagonal (problema 4.7). ¿Qué salida tendría un hipotético programa que lo resolviese, si se usa a sí mismo como entrada? En este caso, esta pregunta es justamente la que nos demuestra que escribir tal programa es imposible. Esto nos indica que el problema C-Diagonal no es decidible. Veamos una demostración más precisa de este hecho.

Proposición 4.2. *El problema* C-Diagonal *es no decidible.*

Demostración. Supongamos que C-Diagonal es decidible. Entonces, existe un programa en Python `c_diagonal.py` que resuelve C-Diagonal. Entonces, podemos preguntarnos por la salida del comando:

```
c_diagonal(leer('./c_diagonal.py'))
```

Este comando pregunta:

[5] Notar que C-Diagonal $= \overline{\text{Diagonal}}$

¿Es cierto que `c_diagonal.py` *no* devuelve `'sí'` cuando se ejecuta a sí mismo?

La respuesta a la pregunta anterior puede ser afirmativa o negativa. Veremos que ninguna de ambas puede ser posible.

Si la respuesta es afirmativa, entonces sabemos que `c_diagonal.py` no devuelve `'sí'` cuando se ejecuta a sí mismo, luego la respuesta debería haber sido negativa – una contradicción, es decir, la respuesta no puede ser afirmativa.

En caso de que sea la respuesta negativa, entonces `c_diagonal.py` devuelve `'no'` cuando se ejecuta a sí mismo, luego la respuesta debería haber sido afirmativa – de nuevo una contradicción, en otras palabras, la respuesta tampoco puede ser negativa.

Concluimos, por tanto, que es imposible escribir el programa `c_diagonal.py`. Como no podemos escribir un programa que decida C-Diagonal, el problema es no decidible. □

Acabamos de encontrar nuestro primer problema no decidible: C-Diagonal. Este hecho es de especial relevancia pues, a partir de él, podemos probar que Universal tampoco lo es.

Proposición 4.3. *El problema* Universal *(problema 4.6) es no decidible.*

Demostración. Procederemos por contradicción. Asumiremos que Universal es decidible. Entonces, podemos usarlo para escribir los programas 4.5 y 4.6, que resuelven (deciden) Diagonal (problema 4.7) y C-Diagonal (problema 4.8), respectivamente. Pero esto contradice la no decidibilidad de C-Diagonal probada en la proposición 4.2. □

```python
from universal import universal

def diagonal(programa):
    return universal(programa, programa)
```

Programa 4.5: `diagonal.py`

```python
from diagonal import diagonal

def c_diagonal(programa):
    salida = diagonal(programa)

    if salida == 'sí':
        return 'no'
    else:
        return 'sí'
```

Programa 4.6: `c_diagonal.py`

Podemos ver la cadena de problemas que hemos usado en la proposición 4.3 en la figura 4.3.

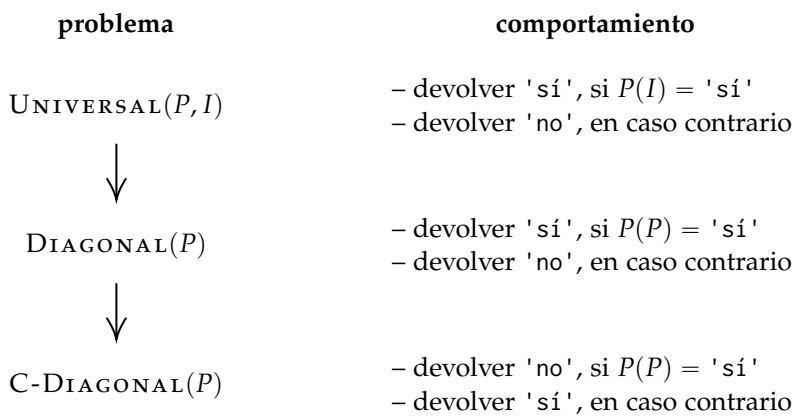


Figura 4.3.: Una cadena de problemas no decidibles

Probar la no decidibilidad de UNIVERSAL ha sido un proceso largo, que ha requerido de otros problemas. Sería posible demostrar este resultado usando un único programa, pero, usando las propias palabras de Turing, *«[tal] prueba, aunque perfectamente sólida, [tendría] la desventaja de que puede dejar al lector con la sensación de que 'debe haber algo mal'»* [44].

Ya hemos demostrado que dos de los tres problemas de la figura 4.3 no son decidibles. El siguiente resultado nos permitirá probar que el último de ellos tampoco lo es.

Proposición 4.4. *Sea D un problema decidible. Entonces, $\bar{D}$ es decidible.*

Demostración. Si D es un problema decidible, tenemos un programa `d.py` que lo decide. Entonces, podemos escribir el programa 4.8,[6] que decide $\bar{D}$. □

```
1  from d import d

3  def c_d(entrada):
4      salida = d(entrada)

6      if salida == 'sí':
7          return 'no'
8      else:
9          return 'sí'
```

Programa 4.7: `c_d.py`

Corolario 4.1. *Sea D un problema no decidible. Entonces, $\bar{D}$ es no decidible.*

Por el corolario 4.1, al saber que C-DIAGONAL (problema 4.8) es no decidible (proposición 4.2), tenemos que DIAGONAL (problema 4.7) es no decidible.

[6] Observa que el programa 4.6 usa la estructura del programa 4.8.

Saber que estos problemas no son decidibles es muy útil, pues nos permiten probar lo que buscamos en este capítulo: que el problema de la parada, que introduciremos en la sección 4.6, tampoco es decidible.

Para poder demostrar esto, sin embargo, deberemos usar una nueva herramienta, que desarrollaremos en la sección 4.5. Antes, comentaremos otra clase de problemas: aquellos que "casi" son decidibles.

4.4. Problemas semidecidibles

En la definición 4.5, exigimos que el programa pare para cada entrada para poder decir que el problema es decidible. Si relajamos esta condición obtenemos una nueva categoría de problemas.

Definición 4.6 (Semidecidir)**.** Dado un problema de decisión $D : S \longrightarrow S$ y un programa P, diremos que P *semidecide* F si:

- para cada entrada $I \in S$ con $F(I) =$ `'sí'`, P para y $P(I) =$ `'sí'`,
- para cada entrada $I \in S$ con $F(I) =$ `'no'`, $P(I)$ es indefinido o $P(I) =$ `'no'`.

Decimos que un problema de decisión D es *semidecidible* si existe un programa que lo semidecide.

Es claro que todo problema decidible es semidecidible. El recíproco no es cierto. Veremos un ejemplo de esto más adelante, en el problema 5.3.

No profundizaremos en la noción de problema semidecidible, ya que no será necesario para nuestro estudio. Cuando sea necesario, recurriremos a la definición aquí especificada.

4.5. Reducciones

En la sección 4.3, hemos usado unos problemas para poder resolver otros o para probar que no pueden resolverse (véase la figura 4.3). En teoría de la computación, esta intuición se formaliza en el concepto de *reducción*. Cuando decimos que "F se reduce a G", nos referimos a que "F se puede resolver si G se puede resolver".

Definición 4.7 (Reducción de Turing)**.** Sean $F : S \longrightarrow S$ y $G : S \longrightarrow S$ dos problemas. Decimos que F se *Turing-reduce*, o simplemente *reduce*,[7] a G, y lo denotamos $F \leqslant_T G$, si existe un programa P_F que resuelve F tras asumir la existencia de un programa P_G que resuelve G.

[7] Existen otros tipos de reducciones, pero nosotros sólo usaremos esta. Es por ello por lo que usamos el término *reducción* en lugar de *reducción de Turing*, ya que no da lugar a ambigüedad.

Al programa P_G del que asumimos su existencia lo llamamos *oráculo*, por motivos históricos (el primer uso de este término se remonta a [23]).

Intuitivamente, si podemos reducir F a G, es porque G es al menos tan "difícil" como F o, recíprocamente, F es al menos tan "fácil" como G. Esta intuición se recoge en la proposición siguiente.

Proposición 4.5. *Sean F y G problemas, y sea $F \leqslant_T G$. Entonces:*

a) si G es decidible, F es decidible.

b) si F no es decidible, G no es decidible.

Demostración. La afirmación *a)* se deduce de la definición de reducción de Turing: si el problema G es decidible, podemos crear un programa P_G que lo resuelve. Al reducirse F a G, podemos crear un programa P_F que resuelve F a partir de P_G, por lo que es decidible.

La afirmación *b)* es inmediata a partir de *a)*. □

Esta proposición nos permite probar que un problema es decidible (o no) a partir de otro. Esto nos permite proceder por varias vías. En primer lugar, si queremos saber que un problema F es fácil, nos baste tener un problema decidible G que reduzca a F, $F \leqslant_T G$. Entonces, F será decidible.

Por otra parte, podemos probar que un problema G es difícil. Basta tener un problema F no decidible tal que $F \leqslant_T G$. Entonces, G tampoco será decidible.

Veamos un primer ejemplo de reducción. En la sección 4.3 ya vimos que DIAGONAL (problema 4.7) y UNIVERSAL (problema 4.6) son ambos no decidibles. Supongamos que únicamente conocemos que DIAGONAL es no decidible. Entonces, podremos probar la no decidibilidad de UNIVERSAL mediante una sencilla reducción.[8] Observa el programa siguiente.

```python
from universal import universal # oráculo

def diagonal(programa):
    salida = universal(programa, programa)

    if salida == 'sí':
        return 'sí'
    else:
        return 'no'
```

Programa 4.8: `diagonal_a_universal.py`

Es sencillo que ver que `diagonal_a_universal.py` es una reducción de DIAGONAL a UNIVERSAL. En efecto, el programa `diagonal_a_universal.py` resuelve DIAGONAL asumiendo la existencia de un programa `universal.py` que resuelva UNIVERSAL.

[8] Recuerda que en la página 38 ya dijimos que podíamos obtener DIAGONAL a partir de UNIVERSAL.

4.6. El problema de la parada

Dado un programa P y una entrada I, nos hemos preguntado por cuál es la salida de tal programa. Ahora, vamos a preguntar si el programa *para*.

Este problema, de reconocer si un programa para dada una entrada, es el conocido como *problema de la parada*. Los motivos por los que lo estudiamos son varios: en primer lugar, este problema tiene aplicación práctica. En caso de ser decidible, podríamos escribir un algoritmo que, para cualquier programa, nos comprobara si es capaz de terminar. Evidentemente, no todos los programas paran: algunos de ellos están diseñados para ejecutarse de forma indefinida, como es el caso de los servidores web o de un sistema operativo.

Por otra parte, este problema tiene una especial relevancia histórica. Fue uno de los primeros problemas en ser demostrados no computables, y es uno de los gérmenes de la teoría de la computación.

En nuestro caso, sin embargo, estudiar este problema nos permitirá deducir el resultado central de este trabajo.

Antes de hablar sobre este problema, debemos recordar que la noción de *parada* que definió Turing está relacionada con las máquinas de su mismo nombre: una máquina de Turing, dada una entrada, puede parar y aceptar, parar y rechazar, o ciclar.

En nuestro caso, diremos que un programa en Python P para dada una entrada I si la salida $P(I)$ está definida, de acuerdo a la definición 3.3. Un programa podría, de este modo, no parar por muchas razones: podría entrar en un bucle infinito, lanzar una excepción o devolver un objeto que no sea un *string*. En los dos últimos casos, puede parecer contraintuitivo decir que el programa no ha parado: al fin y al cabo, el programa para al lanzar una excepción o devolver un objeto que no sea un *string*. En estos casos, es mejor pensar que el programa se ha "congelado": no está ejecutando más código, pero no puede terminar satisfactoriamente. En este sentido, el programa "no ha parado".

Ahora estamos en condiciones de presentar la formulación del problema.

PARADA

- **Entrada:** un programa P y una entrada I.
- **Solución:** si el programa P para con entrada I.

Problema 4.9: PARADA

Nos preguntamos si este problema es decidible. La respuesta es negativa, y esperable teniendo en cuenta todos los problemas de este estilo que ya hemos probado no decidibles (problemas 4.6 a 4.8).

Proposición 4.6. *El problema* PARADA *es no decidible.*

Demostración. Haremos la reducción $\textsc{Universal} \leqslant_T \textsc{Parada}$ mediante el programa 4.10.

Para ello, transformaremos un programa P que devuelve `'sí'` / no devuelve `'sí'` en un programa P' que para / no para. En concreto, P será el programa `programa` que se inserta como entrada en la línea 4 del programa 4.10, y P' será el programa 4.9.

En concreto,

- Si `programa` devuelve `'sí'` con entrada `entrada` (esto quiere decir que $\textsc{Universal}$(`programa`, `entrada`) = `'sí'`), entonces `maquina_universal_parada` para con la entrada `entrada` ($\textsc{Parada}$(`maquina_universal_parada`, `entrada`) = `'sí'`). Observa cómo en la línea 9 del programa 4.9 devolvemos `'sí'`. El valor que devolvemos es irrelevante, lo importante es que el programa para.
- Si `programa` no devuelve `'sí'` con entrada `entrada` ($\textsc{Universal}$(`programa`, `entrada`) = `'no'`), entonces `maquina_universal_parada` no para (cicla) con la entrada `entrada` ($\textsc{Parada}$(`maquina_universal_parada`, `entrada`) = `'no'`). En la línea 11 del programa 4.9 vemos que usamos la función `ciclar` de la librería `utilidades`, que simplemente entra en un bucle infinito.

Hemos demostrado que el programa 4.10 es una reducción de $\textsc{Universal}$ a $\textsc{Parada}$. El resultado se sigue de las proposiciones 4.3 y 4.5. □

```
1  import utilidades
2  from maquina_universal import maquina_universal

4  def maquina_universal_parada(entrada_codificada):
5      (programa, entrada) = utilidades.UAM(entrada_codificada)
6      salida = maquina_universal(programa, entrada)

8      if salida == 'sí':
9          return 'sí'
10     else:
11         utilidades.ciclar()
```

Programa 4.9: `maquina_universal_parada.py`

```
1  import utilidades
2  from parada import parada # oráculo

4  def universal_a_parada(programa, entrada):
5      entrada_codificada = utilidades.MAU(programa, entrada)
6      maquina_universal_parada = utilidades.leer('maquina_universal_parada.py')

8      return parada(maquina_universal_parada, entrada_codificada)
```

Programa 4.10: `universal_a_parada.py`

En la demostración anterior hemos usado una técnica nueva: crear un programa nuevo a partir de otro. En concreto, modificamos `maquina_universal.py` (programa 4.2), la máquina universal

en Python que introdujimos en la sección 4.2, para crear una máquina universal que para para las salidas positivas y cicla para cualquier otra salida.

A continuación, introduciremos un nuevo problema de la parada, más sencillo que PARADA.[9]

PARADAENVACÍO

- **Entrada:** un programa P.
- **Solución:** si el programa P para con entrada vacía.

Problema 4.10: PARADAENVACÍO

Por tratarse de un problema "más sencillo", inicialmente no tiene por qué ser no decidible. Para probarlo, en este caso deberemos realizar un truco: ignorar la entrada.

```
1  import utilidades
2  from maquina_universal import maquina_universal

4  def ignora_entrada(entrada_ignorada):
5      programa = utilidades.leer('disco/programa.txt')
6      entrada = utilidades.leer('disco/entrada.txt')
7      return maquina_universal(programa, entrada)
```

Programa 4.11: `ignora_entrada.py`

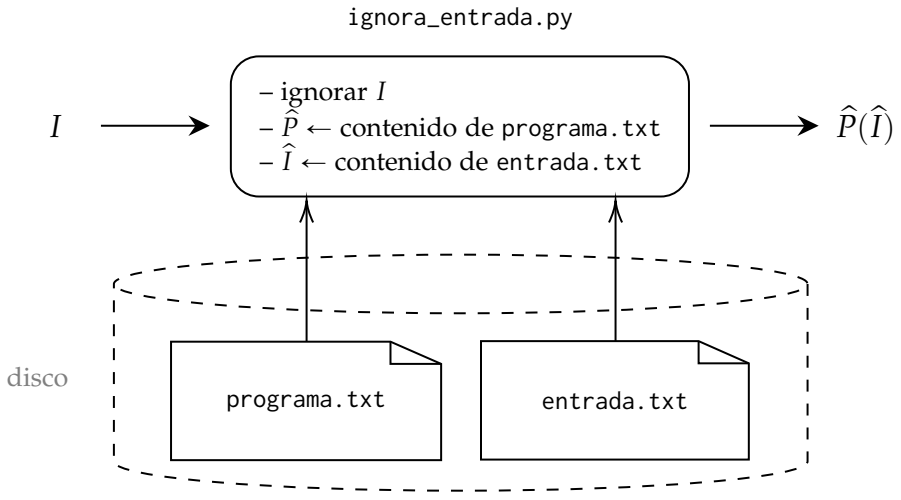


Figura 4.4.: Esquema del programa `ignora_entrada.py`

[9] En algunos libros, se enuncia este problema como el propio problema de la parada.

Para ello, usamos el programa 4.11. Este programa, en lugar de ejecutar la entrada `entrada_ignorada`, ejecuta la máquina universal sobre un programa y una entrada almacenada en disco. Concretamente, usa como programa el almacenado en `disco/programa.txt`, y como entrada la almacenada en `disco/entrada.txt`.

Una vez conocemos el funcionamiento de `ignora_entrada.py`, es sencillo encontrar una reducción de Parada a ParadaEnVacío.

Proposición 4.7. *El problema* ParadaEnVacío *es no decidible.*

Demostración. Basta hacer la reducción Parada $\leqslant_T$ ParadaEnVacío mediante el programa 4.12. En efecto, vemos que tal programa resuelve Parada, asumiendo la existencia de un programa `parada_en_vacio.py` que resuelve ParadaEnVacío. Dado un programa `programa` y una entrada `entrada`, `parada_a_parada_en_vacio` almacenará los valores de las variables en `disco/programa.txt` y `disco/entrada.txt`, respectivamente, en las líneas 5 y 6. A continuación, en la línea 8 se devuelve si el programa `ignora_entrada.py` para o no, lo que equivale a decir, por la forma en la que está escrito tal programa, si `programa` para con entrada `entrada`, resolviendo Parada.

El resultado se sigue de las proposiciones 4.5 y 4.6. □

```
1 import utilidades
2 from parada_en_vacio import parada_en_vacio # oráculo

4 def parada_a_parada_en_vacio(programa, entrada):
5     utilidades.escribir('disco/programa.txt', programa)
6     utilidades.escribir('disco/entrada.txt', entrada)

8     return parada_en_vacio(utilidades.leer('ignora_entrada.py'))
```

Programa 4.12: `parada_a_parada_en_vacio.py`

Para concluir, introducimos un último problema: AdivinaConsistente.

AdivinaConsistente

- **Entrada:** un programa P.
- **Solución:** `'sí'` si P devuelve `'sí'` con entrada vacía (acepta), `'no'` en caso contrario (rechaza o cicla).

Problema 4.11: AdivinaConsistente

La intuición nos indica que este problema, como los anteriores, tampoco es decidible. Lo probamos en la siguiente proposición.

Proposición 4.8. AdivinaConsistente *es no decidible.*

Demostración. Asumiremos que AdivinaConsistente es decidible, entonces existe un programa `adivina_consistente.py` que lo decide. En tal caso, podemos modificar `adivina_consistente.py` para crear un nuevo programa `modifica_adivina_consistente.py` (programa 4.13), al que llamaremos P, que, dada una entrada `programa`, almacena en la variable `salida` (línea 9):

- `'no'` en caso de que `programa(programa) == 'sí'`.
- `'sí'` en caso de que `programa(programa) == 'no'`.
- `'sí'` en caso de que `programa(programa)` cicle (no pare).

Ahora, pensemos qué ocurre al ejecutar el programa sobre sí mismo, es decir, si pasamos el propio programa como entrada. Observemos antes que P nunca ciclará, de acuerdo a la definición de AdivinaConsistente. Hay tres posibilidades:

- Si $P(P) =$ `'no'`, la única forma de que esto ocurra es si se verifica la condición de la línea 11, lo cual solo ocurre si $P(P) =$ `'sí'`. Por tanto, esta posibilidad se descarta.
- Si $P(P) =$ `'sí'`, la única forma de que esto pase es si se verifica la condición de la línea 13, lo cual solo ocurre si $P(P) =$ `'no'` o P cicla, pero como P nunca cicla, entonces deberá ser $P(P) =$ `'no'`. También descartamos esta posibilidad.
- La única posibilidad es que $P(P)$ cicle, pero como hemos visto, esto no es posible.

□

```
1  import utilidades
2  from adivina_consistente import adivina_consistente # NO es un oráculo
3  from ignora_entrada import ignora_entrada

5  def modifica_adivina_consistente(programa):
6      utilidades.escribir('disco/programa.txt', programa)
7      utilidades.escribir('disco/entrada.txt', programa)

9      salida = adivina_consistente(utilidades.leer('ignora_entrada.py'))

11     if salida == 'sí':
12         return 'no'
13     elif salida == 'no':
14         return 'sí'
```

Programa 4.13: `modifica_adivina_consistente.py`

Los problemas ParadaEnVacío y AdivinaConsistente serán centrales en la demostración de dos variantes del resultado principal de este trabajo, en el capítulo 6. A modo de resumen, se proporciona la tabla 4.6 donde se resumen todos los problemas presentados en este capítulo.

	Entrada	Salida	Decidible
MásAQueB (problema 4.4)	$u \in \{\texttt{'a'}, \texttt{'b'}\}^*$	– `'sí'`, si $\#(u, a) > \#(u, b)$ – `'no'`, en caso contrario	sí (prop. 4.1)
Sí (problema 4.5)	cualquiera	– `'sí'`	sí (pág. 36)
Universal (problema 4.6)	P, I	– `'sí'`, si $P(I) = \texttt{'sí'}$ – `'no'`, en caso contrario	no (prop. 4.3)
Diagonal (problema 4.7)	P	– `'sí'`, si $P(P) = \texttt{'sí'}$ – `'no'`, en caso contrario	no (pág. 41)
C-Diagonal (problema 4.8)	P	– `'no'`, si $P(P) = \texttt{'sí'}$ – `'sí'`, en caso contrario	no (prop. 4.2)
Parada (problema 4.9)	P, I	– `'sí'`, si P para con entrada I – `'no'`, en caso contrario	no (prop. 4.6)
ParadaEnVacío (problema 4.10)	P	– `'sí'`, si P para con entrada `''` – `'no'`, en caso contrario	no (prop. 4.7)
AdivinaConsistente (problema 4.11)	P	– `'sí'`, si $P(\texttt{''})$ para y $P(\texttt{''}) = \texttt{'sí'}$ – `'no'`, si $P(\texttt{''})$ para y $P(\texttt{''}) \neq \texttt{'sí'}$ – `'no'`, si $P(\texttt{''})$ cicla	no (prop. 4.8)

Tabla 4.6: Resumen de los problemas expuestos en este capítulo

Parte III.

Incompletitud

A partir de la decidibilidad del problema de parada, probaremos que la aritmética de Peano es incompleta, bajo unas ciertas condiciones. Veremos que la elección de la aritmética de Peano no restringe sacar conclusiones de otros sistemas lógicos. A continuación, compraremos la demostración de este trabajo con la original de Gödel. Finalmente, veremos las limitaciones de nuestro teorema, y aprenderemos un truco que nos permitirá relajar las condiciones del primer teorema.

5. Sistemas lógicos

> *No debemos creer a quienes hoy, con porte filosófico y tono deliberativo, profetizan la caída de la cultura y aceptan el* ignorabimus*. Para nosotros no hay* ignorabimus*, y en mi opinión tampoco lo hay en la ciencia natural. En oposición al necio* ignorabimus *nuestro lema será: "¡debemos saber — sabremos!"*
>
> — David Hilbert, [45]

En este capítulo, nos preguntamos por si podemos expresar de forma algorítmica los conceptos de *demostración* y *verdad*. Esto es, nos preguntaremos por si existe una forma automática de resolver teoremas y comprobar afirmaciones. Como explicamos en el capítulo 2, este fue uno de los ejes centrales de las matemáticas de principios del siglo XX.

Las matemáticas consisten en deducir teoremas a partir de una cierta lista de axiomas. Para poder definir precisamente este concepto, recurrimos a los *sistemas formales* (sección 5.1).

Sin embargo, nos interesa especialmente saber si las afirmaciones (o *fórmulas*) de un sistema son *verdaderas*. Para esto, necesitaremos definir los *sistemas lógicos* (sección 5.2).

Una vez realizada la distinción entre *demostrable* y *verdadero*, nos preguntamos por la relación que estos dos conceptos tienen en un determinado sistema lógico: a partir de esto, comprenderemos en la sección 5.3 los conceptos de *solidez* y *completitud*, y finalmente preguntaremos si podemos comprobar si una fórmula es verdadera de forma algorítmica, pudiendo decir si un sistema lógico es *decidible* o no (véase el capítulo 4). A lo largo de esta sección introduciremos múltiples ejemplos de sistemas lógicos que reforzarán los conceptos ya mencionados, todos girando en torno a una operación aritmética sencilla.

Finalmente, introduciremos un sistema lógico de especial importancia: la aritmética de Peano (sección 5.4). Este sistema es capaz de demostrar una gran cantidad de teoremas de la teoría de números, por lo que nos resultará especialmente interesante saber si todas sus afirmaciones verdaderas son demostrables (este es, justamente, el Primer Teorema de Incompletitud, que demostramos en la sección 6.1). Para ello, encontraremos una forma de traducir el problema de parada (sección 4.6) en una fórmula de este sistema.

5.1. Sistemas formales

A lo largo de este trabajo, hemos presentado múltiples demostraciones, pero ¿qué es exactamente una demostración? Cuando hablamos de *demostrar* algo, nos referimos a hacerlo de forma matemática. Al contrario que las pruebas científicas, las pruebas matemáticas son exactas y comprobables. De este modo, podemos preguntarnos si un ordenador puede realizar demostraciones de forma automática.

Para responder a todas estas preguntas, necesitaremos crear una estructura matemática que nos permita formalizar el concepto de *demostración*.

Definición 5.1 (Sistema formal)**.** Un *sistema formal*, $\mathcal{S}$, es una tripleta $\mathcal{S} = (\mathcal{L}, \mathcal{A}, \mathcal{R})$, donde:

- $\mathcal{L}$ es el *lenguaje formal* sobre un cierto alfabeto A. Cada uno de los elementos del lenguaje se denomina *fórmula bien formada* (o *fórmula*).[1]
- $\mathcal{A}$ es el conjunto de *axiomas*, que es un conjunto de fórmulas de $\mathcal{L}$.
- $\mathcal{R}$ es el conjunto de *reglas de inferencia*, que es un conjunto de funciones que mapean fórmulas de $\mathcal{L}$, llamadas *premisas*, en fórmulas de $\mathcal{L}$, llamadas *derivaciones*. Todas las funciones de $\mathcal{R}$ deben ser computables (ver definición 4.5).

  Si dadas dos fórmulas $\phi_1, \phi_2 \in \mathcal{L}$, ϕ_2 se deriva a partir de ϕ_1, usaremos la notación $\phi_1 \mapsto \phi_2$. Si ϕ_2 se deriva a partir de ϕ_1 en un número finito de derivaciones, usaremos la notación $\phi_1 \overset{*}{\mapsto} \phi_2$.

Veamos un ejemplo de sistema formal que nos será extremadamente útil en este capítulo.

Sistema formal 5.1. Sea **SumaBinaria** $= (\mathcal{L}, \mathcal{A}, \mathcal{R})$ el sistema formal compuesto por:

- Lenguaje $\mathcal{L}$: está formado sobre el alfabeto $\{\texttt{0}, \texttt{1}, \texttt{+}, \texttt{=}\}$, y es el reconocido por la expresión regular[2]
  $$N(\texttt{+}N)^* = N(\texttt{+}N)^*$$
  donde N es una abreviatura de la expresión regular $\texttt{1}(\texttt{0}|\texttt{1})^*$.
- Conjunto de axiomas $\mathcal{A}$: está formado por un único axioma,
  $$\mathcal{A} = \{\texttt{1=1}\}$$
- Conjunto de reglas de inferencia $\mathcal{R}$: son las siguientes:
  $$\begin{array}{lccc} \text{(R1)} & A\texttt{=}B & \mapsto & \texttt{1+}A\texttt{=1+}B \\ \text{(R2)} & A\underline{N}\texttt{+}\underline{N}B & \mapsto & AN\texttt{0}B \\ \text{(R3)} & A\underline{\texttt{1}}\texttt{+}\underline{N\texttt{0}}B & \mapsto & AN\texttt{1}B \end{array}$$
  donde usamos la siguiente notación: A y B son palabras del alfabeto $\{\texttt{0}, \texttt{1}, \texttt{+}, \texttt{=}\}$ y N es una palabra que obedece la expresión regular $\texttt{1}(\texttt{0}|\texttt{1})^*$. Las partes subrayadas son compilaciones *maximales* de la expresión regular de N.

Es importante recalcar la notación usada para describir las reglas de inferencia de **SumaBinaria**, en concreto, sobre el significado de *compilación maximal*. Una *compilación maximal*, en este caso, es una subcadena que verifica la expresión regular de N y no puede extenderse en más direcciones. En el caso de la fórmula `100+11+1001=101`, tenemos cuatro compilaciones maximales de este tipo: `100`, `11`, `1001` y `101`, en ese orden.

[1] Siempre que hablamos de *fórmulas* nos referimos a fórmulas bien formadas.
[2] Aquí el símbolo de suma + es un literal, no el operador $+$.

Sería posible describir estas reglas mediante expresiones regulares, pero esto haría que la notación fuese menos sencilla e intuitiva.

Una vez definido un sistema formal, será posible probar (o *demostrar*) algunas de sus fórmulas, a las que llamaremos *teoremas*.

Definición 5.2 (Teorema y demostración)**.** Dado un sistema formal $\mathcal{S} = (\mathcal{L}, \mathcal{A}, \mathcal{R})$, una *demostración* de una fórmula de $\mathcal{S}$ es una secuencia finita de fórmulas de $\mathcal{S}$, donde cada una de las fórmulas es un axioma, o se sigue de aplicar una regla de inferencia a las fórmulas que aparecen anteriormente en la secuencia.

Una fórmula $\phi \in \mathcal{L}$ se dice *teorema* (o *fórmula demostrable*, o *fórmula probable*) si existe una demostración en $\mathcal{S}$.

Veamos un ejemplo de teorema en **SumaBinaria**.

Ejemplo 5.1. Sea `1+1+1=11` una fórmula de **SumaBinaria**. Veamos que es un teorema:

	(F1)	`1=1`	(axioma)
$\mapsto$	(F2)	`1+1=1+1`	(aplicar R1 a F1)
$\mapsto$	(F3)	`1+1=10`	(aplicar R2 a F2)
$\mapsto$	(F4)	`1+1+1=1+10`	(aplicar R1 a F3)
$\mapsto$	(F5)	`1+1+1=11`	(aplicar R3 a F4)

Observa que, aunque no esté en la definición, las demostraciones de los teoremas suelen ir acompañadas de notas que ayudan a comprobar la prueba. En el código de este trabajo se proporciona una función `suma_binaria.py` que puede generar demostraciones: `es_teorema`. Para probar si una fórmula es un teorema en alguno de los sistemas de este capítulo, especifica el nombre del sistema seguido de la fórmula.

A continuación, algunos ejemplos de uso de esta función:

```
>>> from suma_binaria import es_teorema
>>> es_teorema('SumaBinaria', '1+1+1=111')
    (F1) 1=1          (Axioma)
 -> (F2) 1+1=1+1      (R1)
 -> (F3) 1+1+1=1+1+1 (R1)
 -> (F4) 1+1+1=1+10   (R2)
 -> (F5) 1+1+1=11     (R3)
'teorema'
>>> es_teorema('SumaBinaria', '11+1=10')
'no demostrado en las primeras 109322 fórmulas generadas'
>>> es_teorema('SumaBinaria', 'mal-formada')
'fórmula mal formada'
>>> es_teorema('SistemaQueNoExiste', 'lo-que-sea')
'el sistema no existe'
```

La prueba de `1+1+1=111` que realiza `es_teorema` es diferente de la del ejemplo 5.1. Esto es perfectamente posible: un teorema puede tener diversas demostraciones. De hecho, este hecho se evidencia de la propia definición 5.2: una fórmula es un teorema si "*existe* una demostración".

5.2. Sistemas lógicos

Como dice el propio nombre, el sistema **SumaBinaria** pretende comprobar fórmulas de la suma binaria. De este modo, una palabra como `11` representa el número 3, y una fórmula como `1+1+1=11` representa que $1+1+1=3$. Observa que esto nos permite reconocer si una fórmula de **SumaBinaria** es verdadera: simplemente la interpretamos como si se tratase de una fórmula aritmética, y la declaramos como cierta si es una fórmula aritmética correcta. De este modo, fórmulas como `10+1=11` son verdaderas, mientras que otras como `11+1=10` no lo son.

En el apartado anterior, no mencionamos el concepto de *verdad* de forma intencionada: un sistema formal no tiene ninguna noción de certeza. Esto es porque las definiciones de *teorema* y *fórmula verdadera* son completamente diferentes.

Los teoremas solo dependen de la sintaxis: "¿podemos demostrar un teorema a partir de unos ciertos axiomas y unas reglas de inferencia?" La verdad, sin embargo, depende de la semántica: "¿es esta fórmula cierta cuando es interpretada dándole cierto significado a sus símbolos?".

Por definición, los sistemas formales solo dependen de la sintaxis. En esta sección, introduciremos los *sistemas lógicos*, que no son nada más que sistemas formales a los que le añadimos una *semántica*. Cuando añadimos la semántica de la aritmética que hemos comentado al sistema formal **SumaBinaria** obtenemos el sistema lógico **SumaBinariaLógico**.

Sistema lógico 5.1. Sea **SumaBinariaLógico** el sistema lógico que se obtiene al añadir la semántica de la aritmética al sistema formal **SumaBinaria**.

Una fórmula en **SumaBinariaLógico** es verdadera si es cierta vista como una fórmula aritmética de la suma de números binarios.

Formalmente, podemos definir estos sistemas como:

Definición 5.3 (Sistema lógico)**.** Un sistema lógico $\mathcal{G} = (\mathcal{S}, \mathcal{I})$ consiste en un sistema formal $\mathcal{S} = (\mathcal{L}, \mathcal{A}, \mathcal{R})$ y una semántica o *asignación de verdad* $\mathcal{I}$, donde $\mathcal{I}$ es una función que asigna a fórmulas de $\mathcal{L}$ una *interpretación*, que es un valor del conjunto $\{\text{verdadero}, \text{falso}\}$.

La asignación de verdad no debe ser necesariamente computable (definición 4.5) y puede estar definida para todas las fórmulas o para un subconjunto de ellas.[3]

Una vez que incorporamos a un sistema formal una asignación de verdad, podemos decir cuándo una fórmula es verdadera. Precisamos esto en la definición siguiente:

Definición 5.4 (Fórmula verdadera)**.** Dado un sistema lógico $\mathcal{G} = (\mathcal{S}, \mathcal{I})$ con $\mathcal{S} = (\mathcal{L}, \mathcal{A}, \mathcal{R})$, decimos que la fórmula $\phi \in \mathcal{S}$ es *verdadera* si $\mathcal{I}(\phi)$ está bien definido y es $\mathcal{I}(\phi) = \text{verdadero}$.

Veamos ejemplos de fórmulas verdaderas y no verdaderas para **SumaBinariaLógico**. A las fórmulas no verdaderas las llamamos *falsas*.

[3] Cuando una fórmula no está bien definida o no tiene una interpretación definida, podemos asumir que es falsa.

Ejemplo 5.2. Sea `1+1+1=11` una fórmula de **SumaBinariaLógico**. Claramente, esta fórmula es verdadera dado que la siguiente operación aritmética es cierta para el sistema de numeración binario:

$$1 + 1 + 1 = 11$$

En `suma_binaria.py` también se proporciona una función para comprobar la veracidad de las fórmulas en sistemas lógicos, llamada `es_verdadero`. Un ejemplo:

```
>>> from suma_binaria import es_verdadero
>>> es_verdadero('SumaBinariaLógica', '1+1+1=111')
True
>>> es_verdadero('SumaBinariaLógica', '11+1=10')
False
>>> es_verdadero('SumaBinariaLógica', 'mal-formada')
False
>>> es_verdadero('SumaBinaria', '1+1+1=111')
'el sistema no existe o no es un sistema lógico'
```

Introduciremos más sistemas lógicos que resultarán de añadir (sistema lógico 5.2) y eliminar (sistema lógico 5.3) reglas de **SumaBinariaLógico**.

Sistema lógico 5.2. Sea **SumaBinariaLógicoRoto** el sistema lógico que se obtiene al añadir la regla de inferencia siguiente a **SumaBinariaLógico**:

$$\text{(R1b)} \quad A{=}B \quad \mapsto \quad 1{+}A{=}B$$

Sistema lógico 5.3. Sea **SumaBinariaLógicoRestringido** el sistema lógico que se obtiene al eliminar la regla de inferencia (R3) a **SumaBinariaLógico**.

Estos sistemas son especialmente interesantes, pues en ellos podemos encontrar tanto fórmulas demostrables y no verdaderas (ejemplo 5.3) como fórmulas verdaderas y no demostrables (ejemplo 5.4).

Ejemplo 5.3. La fórmula `1+1=1` es un teorema y no es verdadera en **SumaBinariaLógicoRoto**.

Claramente, $1 + 1 = 1$ no una operación cierta de la suma binaria, y por tanto no es una fórmula verdadera. Sin embargo, se trata de un teorema porque podemos demostrarlo directamente a partir del axioma mediante la nueva regla (R1b) introducida.

```
>>> from suma_binaria import es_teorema, es_verdadero
>>> es_teorema('SumaBinariaLógicoRoto', '1+1=1')
    (F1) 1=1   (Axioma)
 -> (F2) 1+1=1 (R1b)
'teorema'
>>> es_verdadero('SumaBinariaLógicoRoto', '1+1=1')
False
```

Ejemplo 5.4. La fórmula `1+10=11` no es un teorema y es verdadera en **SumaBinariaLógicoRestringido**.

La fórmula es verdadera dado que $1 + 10 = 11$ es una operación correcta en la suma binaria. Sin embargo, no es un teorema porque, al eliminar (R3), no podemos reemplazar los `0` con `1`.

```
>>> from suma_binaria import es_teorema, es_verdadero
>>> es_teorema('SumaBinariaLógicoRestringido', '1+10=11')
'no demostrado en las primeras 154639 fórmulas generadas'
>>> es_verdadero('SumaBinariaLógicoRestringido', '1+10=11')
True
```

Es importante notar que la función `es_teorema` no puede demostrar que una fórmula no es demostrable debido a la forma en la que lo hemos diseñado. El código anterior no prueba que `1+10=11` no sea un teorema – la afirmación de este ejemplo sí.

5.3. Solidez, completitud y decidibilidad

Hemos encontrado sistemas lógicos en los que hay fórmulas verdaderas y no demostrables, y viceversa. Esto nos conduce a la definición de dos importantes propiedades de los sistemas lógicos.

Definición 5.5 (Solidez)**.** Un sistema lógico es *sólido* si todos sus teoremas son verdaderos. En caso contrario, decimos que es *no sólido*.

Definición 5.6 (Completitud)**.** Un sistema lógico es *(semánticamente) completo*[4] si todas las fórmulas verdaderas son teoremas. En caso contrario, decimos que es *incompleto*.

Veamos si los sistemas lógicos que hemos presentado hasta ahora son sólidos y completos. Un resumen de todos los resultados se puede ver en la tabla 5.1.

Probar que **SumaBinariaLógico** es sólido y completo es tedioso: por eso dejamos las demostraciones en el apéndice B.

Proposición 5.1. *El sistema lógico* **SumaBinariaLógico** es sólido y completo.

Demostración. La demostración de solidez se prueba en la proposición B.3, mientras que la de completitud se muestra en la proposición B.4. □

Dado que los otros sistemas lógicos se "obtienen" a partir de **SumaBinariaLógico**, las proposiciones siguientes son mucho más sencillas de demostrar. Como es evidente, los ejemplos 5.3 y 5.4 prueban que **SumaBinariaLógicoRoto** es no sólido y que **SumaBinariaLógicoRestringido** es incompleto, respectivamente.

Veamos a continuación tales proposiciones.

[4] Más adelante introducimos la noción de *completitud sintáctica* (definición 5.9). Siempre que llamemos a un sistema *completo*, sin especificar, nos referiremos a su completitud semántica.

Proposición 5.2. *El sistema lógico* **SumaBinariaLógicoRoto** es completo y no sólido.

Demostración. Es fácil ver que este sistema es completo a partir de la completitud de **SumaBinariaLógico**. Observamos que ambos sistemas lógicos tienen la misma sintaxis, es decir, tienen las mismas fórmulas verdaderas. De este modo, una fórmula ϕ verdadera en **SumaBinariaLógico** también lo es en **SumaBinariaLógicoRoto**.

Veamos si ϕ es un teorema en **SumaBinariaLógicoRoto**. Por la completitud de **SumaBinariaLógico**, ϕ es un teorema en este sistema. Pero como **SumaBinariaLógicoRoto** tiene todas las reglas de **SumaBinariaLógico** (y una más), ϕ también es un teorema en **SumaBinariaLógicoRoto**.

Sin embargo, el sistema lógico es no sólido. En el ejemplo 5.3 vimos una fórmula que es un teorema y no es verdadera. □

Proposición 5.3. *El sistema lógico* **SumaBinariaLógicoRestringido** es incompleto y sólido.

Demostración. La incompletitud de este sistema se sigue del ejemplo 5.4. La solidez se sigue de la solidez de **SumaBinariaLógico**.

Sea ϕ un teorema de **SumaBinariaLógicoRestringido**. Como **SumaBinariaLógico** tiene todas las reglas de **SumaBinariaLógicoRestringido** (y una más), ϕ es un teorema en **SumaBinariaLógico**. De la completitud de **SumaBinariaLógico**, ϕ es verdadero. Pero como **SumaBinariaLógicoRestringido** tiene la misma sintaxis, ϕ también será verdadero en este sistema. □

Los sistemas lógicos **SumaBinariaLógicoRoto** y **SumaBinariaLógicoRestringido** los obtuvimos modificando la sintaxis de **SumaBinariaLógico**. Introducimos ahora el último sistema de este tipo, que surge de modificar la semántica de **SumaBinariaLógicoRoto**.

Sistema lógico 5.4. Sea **SumaBinariaLógicoArreglado** el sistema lógico que se obtiene a partir de **SumaBinariaLógicoRoto** modificando su semántica. Interpretaremos las fórmulas como fórmulas aritméticas de la suma de números binarios, pero el símbolo = será interpretado como "mayor o igual que" ($\geqslant$).

En este último sistema lógico podríamos haber modificado el símbolo = por >=. Sin embargo, no lo hacemos de forma intencionada, y para recordar que la sintaxis y la semántica son independientes – podemos asignar el valor de verdad que queramos.

Veamos un ejemplo sencillo e ilustrativo de fórmula verdadera y de fórmula falsa.

Ejemplo 5.5. La fórmula `1+1=1` es verdadera en **SumaBinariaLógicoArreglado**, mientras que `1=1+1` es falsa.

```
>>> from suma_binaria import es_verdadero
>>> es_verdadero('SumaBinariaLógicoArreglado', '1+1=1')
True
>>> es_verdadero('SumaBinariaLógicoArreglado', '1=1+1')
False
```

Al modificar la sintaxis, hemos logrado "arreglar" el sistema **SumaBinariaLógicoRoto**, como bien dice su nombre. En efecto, **SumaBinarioLógicoArreglado** es completo y sólido.

Proposición 5.4. *El sistema lógico* **SumaBinariaLógicoArreglado** es completo y sólido.

Demostración. Es sencillo ver que este sistema lógico es sólido. Basta ver que la regla (R1b) nos permite probar afirmaciones como 1+1=1. Esta regla siempre añade más 1 a la izquierda de = que a la derecha. De este modo, en todo teorema del sistema la parte izquierda del igual será mayor o igual que la parte derecha, coincidiendo con la definición de la asignación de verdad (semántica).

La completitud de este sistema se prueba en la proposición B.5. □

A modo de resumen, vemos en la tabla siguiente la completitud y solidez de los sistemas presentados (junto con otra información que veremos más adelante).

	Sólido	Completo	Decidible
SumaBinariaLógico (reglas R1, R2, R3)	sí	sí	sí
SumaBinariaLógicoRoto (reglas R1, R1b, R2, R3)	sí	no	sí
SumaBinariaLógicoRestringido (reglas R1, R2)	no	sí	sí
SumaBinariaLógicoArreglado (reglas R1, R1b, R2, R3 y una semántica diferente)	sí	sí	sí
AritméticaPeano	asumido	no	no

Tabla 5.1: Solidez, completitud y decidibilidad de los sistemas lógicos de este capítulo

Adicionalmente a las propiedades de solidez y consistencia, podemos introducir una propiedad computacional: la de *decidibilidad*. Ya sabemos qué significa *decidible* para un problema de decisión: que podemos resolverlo con un programa que siempre para y tiene salida correcta. Para un sistema lógico $\mathcal{G}$ podemos definir un problema de decisión, EsVerdadero$_{\mathcal{G}}$.

EsVerdadero$_{\mathcal{G}}$

- **Entrada:** una fórmula ϕ de $\mathcal{G}$.
- **Salida:** `'sí'` si $\mathcal{I}(\phi)$ está bien definido y $\mathcal{I}(\phi) =$ verdadero, y `'no'` en caso contrario.

Problema 5.1: EsVerdadero$_{\mathcal{G}}$

Para cada sistema lógico $\mathcal{G}$ obtenemos un problema de decisión distinto. Una vez definido este problema, es natural definir la decidibilidad de un sistema lógico mediante la decidibilidad del problema.

Definición 5.7 (Decibilidad de un sistema lógico)**.** Decimos que un sistema lógico $\mathcal{G}$ es *decidible* si el problema de decisión EsVerdadero$_{\mathcal{G}}$ es decidible.

De este modo, un sistema lógico es decidible si existe un programa que decide si una fórmula es verdadera o no. Es importante notar que algunos autores definen como sistema decidible aquel en el que el problema EsTeorema$_{\mathcal{S}}$ (que definimos a continuación) es decidible. Como hemos visto, la verdad y la demostración son conceptos diferentes, y nosotros nos referiremos siempre a la verdad al preguntarnos por si un sistema lógico es decidible.

EsTeorema$_{\mathcal{S}}$

- **Entrada:** una fórmula ϕ de $\mathcal{S}$.
- **Salida:** `'sí'` si ϕ es un teorema de $\mathcal{S}$, y `'no'` en caso contrario.

Problema 5.2: EsTeorema$_{\mathcal{S}}$

Veamos si los sistemas lógicos que hemos introducido en este capítulo son decidibles. En el caso de **SumaBinariaLógico**, es sencillo crear un programa que sea capaz de comprobar si las fórmulas que se insertan están bien formadas, y que comprueba si la operación aritmética es correcta. De hecho, este programa está implementado: es la función `es_verdadero` de `suma_binaria.py`:

```
>>> from suma_binaria import es_verdadero
>>> es_verdadero('SumaBinariaLógico', '1+1+1=11')
True
>>> es_verdadero('SumaBinariaLógico', '11+1=10')
False
```

En el caso del resto de sistemas lógicos, crear programas que los decidan resulta igual de sencillo. Las implementaciones a estos programas se encuentran igualmente en `es_verdadero`. Vemos a continuación más ejemplos de la ejecución de esta función en el resto de sistemas lógicos.

```
>>> from suma_binaria import es_verdadero
>>> es_verdadero('SumaBinariaLógico', '11=1+1')
False
>>> es_verdadero('SumaBinariaLógicoRoto', '11=1+1')
False
>>> es_verdadero('SumaBinariaLógicoRestringido', '11=1+1')
False
>>> es_verdadero('SumaBinariaLógicoArreglado', '11=1+1')
True
```

5.4. La aritmética de Peano

Los sistemas lógicos que hemos explorado son de poca utilidad. Lo que queremos es trabajar con sistemas lógicos que permitan probar resultados interesantes en matemáticas. Los matemáticos, lógicos y filósofos han estudiado multitud de sistemas lógicos desde finales del siglo xix, pero dos destacan por su importancia: los *axiomas de Zermelo-Fraenkel*, de teoría de conjuntos, y la *aritmética de Peano*, de teoría de números. En este apartado nos centraremos en el último, pues la aritmética de Peano contiene fórmulas sobre enteros, adición, multiplicación, etc.

Llamaremos **Peano** al sistema lógico de la aritmética de Peano. Puedes encontrar la descripción completa de este sistema en el apéndice C. En la figura 5.1 puedes encontrar algunos detalles de tal descripción.

> La aritmética de Peano se compone de símbolos lógicos como $\forall$, $\exists$, $\vee$, $\wedge$, $\neg$, $=$, $\Rightarrow$ y $\iff$. Los axiomas y reglas de inferencia de **Peano** permiten deducir propiedades de los números naturales. Éstos se representan a partir del 0 mediante la función sucesora S, de modo que 1 es $S(0)$, 2 es $S(S(0))$, etc. De forma general, $S(n)$ representa $n+1$ para un natural $n \in \mathbb{N}$. Las reglas de inferencia incluyen fórmulas lógicas clásicas, tales como el *modus ponens*: dado a y $a \Rightarrow b$, podemos deducir b.

Figura 5.1.: Algunos detalles de **Peano**

En este sistema podemos producir fórmulas sobre aritmética muy interesantes. Vemos algunas de ellas a continuación.

Ejemplo 5.6. Algunas fórmulas de **Peano** (expresadas en lenguaje natural):[5]

(PF1) $1 + 1 = 2$.

(PF2) Para todo entero x, es $x^2 + 1 > 0$.

(PF3) 2 y 5 son divisores de 30.

(PF4) 13 es primo.

(PF5) Sea $f(x) = x^2 + 1$. Entonces, $f^3(0) = 5$.[6]

(PF6) Si $n > 3$, no existen x, y, z enteros tal que $x^n + y^n = z^n$ (el *último teorema de Fermat*).

(PF7) Sea $f(x) = x + 2$. Entonces, para todo entero positivo y existe un entero positivo x de modo que $f(x) = y$.

[5] Se pueden traducir sencillamente a aritmética de Peano. Por ejemplo, (F2) es $\forall x : x^2 + 1 > 0$, y (F6) es $n > 3 \wedge \neg\exists x, y, z : x^n + y^n = z^n$. Aunque las operaciones de comparación y exponenciación no estén inicialmente definidas en **Peano**, podemos definirlas mediante fórmulas del sistema.

[6] f^3 significa iterar f tres veces: $f^3(x) = f(f(f(x)))$.

Para poder trabajar con **Peano** en programas en Python, deberemos encontrar una forma de codificar los diversos símbolos, que especificamos en la tabla 5.2.

Símbolo	Codificación	Descripción
$\neg(p)$	`NOT (p)`	Negación
$(p) \vee (q)$	`(p) OR (q)`	Disyunción
$\forall x : p$	`FORALL (x) : (p)`	Cuantificador universal
$S(a)$	`S(a)`	Sucesor
$\exists x : p$	`EXISTS (x) : (p)`	Cuantificador existencial
$(p) \Rightarrow (q)$	`(p) => (q)`	Implicación
$(p) \iff (q)$	`(p) <=> (q)`	Coimplicación
$(p) = (q)$	`(p) = (q)`	Igualdad

Tabla 5.2: Una forma de traducir los símbolos de la aritmética de Peano a *strings*

Por ejemplo, consideremos la afirmación "no existen enteros x e y de modo que $2x = y$ e $y = 3$", que puede representarse en **Peano** como $\neg\exists x, y : S(S(0)) * x = y \wedge y = S(S(S(0)))$ y que admite la siguiente codificación en *string*:

```
NOT EXISTS x,y : S(S(0))*x = y AND y = S(S(S(0)))
```

Las fórmulas de **Peano**, por ser de un sistema lógico, pueden ser verdaderas o falsas. En el ejemplo 5.6, las primeras seis fórmulas (de (PF1) a (PF6)) son verdaderas,[7] mientras que la última (la fórmula (PF7)) es falsa.

Para poder probar si una fórmula de **Peano** es verdadera, necesitamos que sea *cerrada*: esto ocurre si todas sus variables son cuantificadas por un "para todo" ($\forall$) o un "existe" ($\exists$).[8] Un ejemplo de esto es $\forall x, y : x = y \wedge x + S(0) = y + S(0)$, que es una fórmula cerrada pues x e y son cuantificadas, mientras que $x = y \wedge x + S(0) = y + S(0)$ no es una fórmula cerrada. Esta distinción es importante porque, si una fórmula es cerrada, no podemos saber si es verdadera. Solo consideraremos fórmulas cerradas en **Peano** de aquí en adelante.

Una vez que aceptamos que la aritmética de Peano puede expresar afirmaciones matemáticas como las del ejemplo 5.6, podemos continuar para expresar las salidas de un programa en **Peano**. La siguiente proposición muestra cómo hacerlo para un conjunto concreto de fórmulas sobre los programas que necesitaremos más adelante.

Proposición 5.5. *Sea P un programa de ordenador (o una máquina de Turing).*[9] *La afirmación S dada por*

> *"P para con entrada vacía"*

[7] El último teorema de Fermat tiene especial relevancia histórica: fue formulado en 1637 y no fue resuelto hasta el año 1995 por Andrew Wiles. [47]

[8] Recuerda que en la definición 5.3 dijimos que la asignación de verdad puede estar definida para un subconjunto de las fórmulas.

[9] Probamos la equivalencia en el teorema 3.1.

puede ser traducida en una fórmula ϕ_S equivalente en **Peano**. (En este caso, con "equivalente" nos referimos a que S es verdadera si y solo si ϕ_S es verdadera.)

Demostración. Haremos un boceto de la prueba, dado que realizarla con un rigor estricto sería largo y tedioso.

En este caso, es más sencillo probar esto mediante máquinas de Turing. Recuerda que podemos definir cada momento de la ejecución de la máquina mediante una *configuración*. En el caso de una máquina de Turing de una sola cinta, podemos codificar de forma sencilla una configuración cualquiera: podemos usar una codificación similar a la especificada en la proposición 3.5, añadiendo el carácter ˆ para indicar la posición del cabezal. A modo de ejemplo, se especifica a continuación un ejemplo de codificación de una configuración del ejemplo 3.1:

$$q_R \ : \ X\,X\,\boxed{X}\,X\,a \quad \longrightarrow \quad \texttt{q\_R : X X ˆX X a}$$

Del mismo modo, podemos codificar los *strings* mediante números en un *string* binario (una palabra del alfabeto $\{0, 1\}$. Así, es razonable asumir que la configuración de una máquina de Turing puede ser representada por un número en *string* binario. La función de transición de la máquina de Turing toma una de estas configuraciones en binario como *string* de entrada, y tiene como salida la nueva configuración, codificada de la misma forma. Desde este punto de vista, la función de transición es una función de enteros en enteros. Llamaremos a esta función *Paso*, dado que evaluar esta función una sola vez es equivalente a un paso de cálculo en la máquina de Turing. Observa que la función *Paso* puede escribirse mediante aritmética de Peano: no es más que una función simple que depende en unos pocos bits de entrada.

Supongamos que c_0 es el entero que representa la configuración inicial de la máquina. Tras un paso de cálculo, la configuración de la máquina será $Paso(c_0)$, y tras dos pasos, será $Paso(Paso(c_0))$. Podemos abreviar esto como $Paso^2(c_0)$. Tras n pasos, la configuración de la máquina será $Paso^n(c_0)$. Es importante destacar que cada una de las configuraciones puede ser expresada mediante aritmética de Peano.

Por otra parte, es evidente que la afirmación

> "P para con entrada vacía"

es equivalente a

> "Existe un natural n de modo que $Paso^n(c_0)$ es una configuración de parada"

Por "configuración de parada" nos referimos a que la máquina de Turing no puede seguir ejecutando transiciones: ha parado. Comprobar esto también es sencillo. Vamos a escribir una función *Parado* que tiene como entrada un entero m y como salida 0 ó 1. $Parado(m) = 1$ si la configuración m es de parada, y $Parado(m) = 0$ en caso contrario.

Escribir esta función en **Peano** tampoco es difícil: ya en la definición 3.7 dijimos que una máquina para si no hay ninguna transición definida para la configuración en la que se encuentre. Es evidente que podemos traducir esta afirmación a una fórmula de la aritmética de Peano.

De este modo, hemos probado que $Parado(Paso^n(c_0))$ es una fórmula de **Peano** para un natural $n \in \mathbb{N}$ y un binario c_0. Por tanto, la afirmación:

> "Existe un natural n de modo que $Parado(Paso^n(c_0)) = 1$"

también puede traducirse a una fórmula de **Peano**. Esta afirmación es equivalente a las anteriores, y es esta fórmula justamente la ϕ_S que buscamos. □

De la prueba anterior, es evidente que la construcción de ϕ_S a partir de S se realiza de una forma completamente algorítmica, que podría en principio ser implementada por un programa en Python. Sea tal programa `parada_a_peano.py`. En el código de este trabajo no se proporciona un programa que implemente esto, ya que hacerlo es una tarea compleja.[10] Sin embargo, en el resto de este capítulo, importaremos la función *main* `parada_a_peano` cuando sea necesario.

Definiremos a continuación dos problemas de decisión importantes: EsTeoremaPeano y EsVerdaderoPeano.[11]

EsTeoremaPeano

- **Entrada:** una fórmula ϕ de **Peano**.
- **Salida:** `'sí'` si ϕ es una fórmula bien formada y cerrada en **Peano**, y se trata de un teorema de **Peano**; `'no'` en caso contrario.

Problema 5.3: EsTeoremaPeano

EsVerdaderoPeano

- **Entrada:** una fórmula ϕ de **Peano**.
- **Salida:** `'sí'` si ϕ es una fórmula bien formada, cerrada y verdadera en **Peano**; `'no'` en caso contrario.

Problema 5.4: EsVerdaderoPeano

Dada una afirmación ϕ en **Peano**, los problemas anteriores preguntan si ϕ es demostrable y/o verdadera. Resulta que ambos problemas son no decidibles. La no decidibilidad de EsVerdaderoPeano nos indica, mediante la definición 5.7, que **Peano** no es decidible (como ya adelantamos en la tabla 5.1).

Proposición 5.6. **Peano** es no decidible.

Demostración. Procederemos mediante la reducción

$$\text{ParadaEnVacío (problema 4.10)} \leqslant_T \text{EsVerdaderoPeano}.$$

[10] Existen repositorios de código que realizan esta tarea. Un ejemplo podemos encontrarlo en [19].
[11] Observa que $\text{EsTeoremaPeano} = \text{EsTeorema}_{\textbf{Peano}}$ y $\text{EsVerdaderoPeano} = \text{EsVerdadero}_{\textbf{Peano}}$ (véanse los problemas 5.1 y 5.2).

Probaremos que esta reducción se implementa en el programa 5.1. La función *main* (línea 4) tiene como entrada un programa P. Usando la proposición 5.5, convertimos la afirmación $S =$ "P para con entrada vacía" en una fórmula ϕ_S equivalente en la aritmética de Peano (línea 5). Es importante notar que `parada_a_peano` es una función computable, como comentamos anteriormente. Claramente, P para si y solo si ϕ_S es verdadero, lo que demuestra la reducción. El resultado se sigue de la proposición 4.5. □

```
1 from es_verdadero_peano import es_verdadero_peano # oráculo
2 from parada_a_peano import parada_a_peano # NO es un oráculo

4 def parada_en_vacio_a_es_verdadero_peano(programa):
5     para_en_peano = parada_a_peano(programa)
6     return es_verdadero_peano(para_en_peano)
```

Programa 5.1: `parada_en_vacio_a_es_verdadero_peano.py`

Proposición 5.7. EsTeoremaPeano *es no decidible.*

Demostración. De forma similar a la proposición 5.6, usaremos la reducción ParadaEnVacío $\leqslant_T$ EsVerdaderoPeano.

Antes de esto, debemos apreciar una propiedad de las afirmaciones de la forma "P para con entrada vacía". Llamemos a tal afirmación $H(P)$,[12] y mostraremos que para cualquier programa P, si $\phi_{H(P)}$ es verdadera en **Peano**, entonces es un teorema en **Peano**. Es importante notar que este no es el caso para todas las fórmulas (si no, **Peano** sería completo).

El motivo por el que esto ocurre justamente para las fórmulas $\phi_{H(P)}$ es porque, simplemente, debemos simular P hasta que pare, algo que podemos definir mediante aritmética de Peano: simplemente calculamos $Parado(Paso^n(c_0))$ para cada $n \in \mathbb{N}$ de forma incremental, hasta que el resultado sea 1.

Ahora podemos realizar la reducción usando el programa 5.1 sustituyendo en la línea 1 a `es_verdadero_peano` por `es_teorema_peano`. Como las fórmulas verdaderas del tipo $\phi_{H(P)}$ son teoremas, la reducción es correcta. □

El caso de EsTeoremaPeano es menos grave, pues el problema es, de hecho, semidecidible (ver definición 4.6). Esto quiere decir que podemos escribir un programa que compruebe si una fórmula es un teorema (en caso de que la fórmula no sea un teorema, el programa no tiene por qué parar).

Proposición 5.8. EsTeoremaPeano *es semidecidible.*

Demostración. Es fácil ver que el programa 5.2 hace a EsTeoremaPeano semidecidible. La idea es iterar sobre todos los posibles *strings*, hasta ver si uno de ellos es una prueba de la fórmula de entrada.

En la línea 2 importamos la función `es_prueba_peano`, que dada una fórmula y una demostración en **Peano** comprueba si la demostración prueba que la fórmula es un teorema. Claramente,

[12] Usamos H del inglés *halting*.

esta función no es un oráculo. De hecho, su implementación es sencilla: basta probar si cada una de las fórmulas de la demostración es un axioma o se sigue de las fórmulas anteriores a través de las reglas de inferencia del sistema, comprobando que la última fórmula coincida con la fórmula que queremos probar.

En el bucle de la línea 7 vamos iterando por todos los *strings* para ver si son prueba de la fórmula `fórmula`. Estos *strings* se enumeran en orden *shortlex*: primero se enumeran en orden lexicográfico todos los *strings* de longitud 1, luego de longitud 2, etc. Esta ordenación la proporciona la función `siguiente_string` de `utilidades` (línea 10).

Si `fórmula` es un teorema, su demostración se encontrará eventualmente y el programa devolverá `sí`. En caso contrario, el programa no parará, pero esto no es un problema, ya que pretendemos mostrar la semidecibilidad del problema, la cual no requiere que el programa pare para instancias negativas. □

```
1  import utilidades
2  from es_prueba_peano import es_prueba_peano # NO es un oráculo

4  def es_teorema_peano(fórmula):
5      demostración = ''

7      while True:
8          if es_prueba_peano(demostración, fórmula) == 'sí':
9              return 'sí'
10         demostración = utilidades.siguiente_string(demostración)
```

Programa 5.2: `es_teorema_peano.py`

El hecho de que EsTeoremaPeano sea semidecidible es extremadamente importante, y se debe a una propiedad sintáctica de **Peano**: es recursivamente axiomatizable.

Definición 5.8 (Recursivamente axiomatizable)**.** Un sistema formal $\mathcal{S}$ es *recursivamente axiomatizable* si sus axiomas son recursivamente enumerables, esto es, si existe un programa que enumera sus axiomas.[13]

Decir que un sistema lógico $\mathcal{G}$ es recursivamente axiomatizable no es más que decir que el problema $\text{EsTeorema}_{\mathcal{G}}$ es semidecidible. Esta propiedad es fundamental para la prueba que realizaremos del Primer Teorema de Incompletitud (teorema 6.1), pues si **Peano** no fuerse recursivamente axiomatizable, no podríamos usar la función `es_teorema_peano`.

El problema EsTeorema guarda una fuerte relación con el *Entscheidungsproblem* (ver capítulo 2). Este problema pretende encontrar un algoritmo general que, dada una afirmación matemática, determine si es un teorema. Por la proposición 5.7, EsTeoremaPeano es no decidible, lo cual prueba el *Entscheidungsproblem* en el caso particular de la aritmética de Peano. El argumento que hemos realizado es similar al que hizo Turing en [44].

Finalmente, dado que **Peano** es un sistema lógico, nos podemos preguntar por su solidez y completitud. La pregunta sobre si **Peano** es completo es el objeto central de este trabajo, y se tratará en el capítulo 6.

[13] Si $\mathcal{S}$ es infinito, el programa correrá infinitamente.

Preguntarnos por la solidez de **Peano** es una pregunta compleja. En este trabajo, asumiremos su solidez, dado que es razonable que los teoremas de **Peano** sean verdaderos.[14]

5.5. Completitud sintáctica y consistencia

Los libros de lógica son extremadamente inconsistentes[15] al definir propiedades de sistemas lógicos (incluso al definir qué es un sistema formal y un sistema lógico). Por ahora, hemos definido las propiedades de solidez, completitud (semántica) y decidibilidad. Existe otra definición de completitud que depende únicamente de la sintaxis, y que por tanto es aplicable para sistemas formales. [27]

Definición 5.9 (Completitud sintáctica)**.** Sea $\mathcal{S}$ un sistema formal. Siempre que sea posible,[16] decimos que $\mathcal{S}$ es sintácticamente completo si, para toda fórmula ϕ de $\mathcal{S}$, es ϕ o $\neg\phi$ un teorema.[17]

Observa que, para que esta definición sea posible, es necesario que en el sistema formal esté bien definida la noción de *negación*. Esto se traduce a que el sistema sea una *lógica de primer orden*. Definir este concepto de forma exacta queda fuera del alcance de este trabajo. Simplemente notaremos que los sistemas lógicos suficientemente expresivos, como la aritmética de Peano o los axiomas de Zermelo-Fraenkl, son de este tipo.

Para esta clase de sistemas también es posible definir una nueva propiedad: la *consistencia*. Nuevamente, podemos definir dos tipos de consistencia, una sintáctica y una semántica.

Definición 5.10 (Consistencia sintáctica)**.** Sea $\mathcal{S}$ un sistema formal. Siempre que sea posible, decimos que $\mathcal{S}$ es *sintácticamente completo* si, para toda fórmula ϕ de $\mathcal{S}$, no es posible que tanto ϕ como $\neg\phi$ sean teoremas.

Definición 5.11 (Consistencia semántica)**.** Sea $\mathcal{G}$ un sistema lógico. Siempre que sea posible, decimos que $\mathcal{G}$ es *semánticamente completo* si, para toda fórmula ϕ de $\mathcal{G}$, no es posible que tanto ϕ como $\neg\phi$ sean verdaderas.

En otras palabras, un sistema es consistente si no da lugar a demostraciones o a verdades contradictorias. En un sistema sólido y completo, una fórmula es consistente sintácticamente si y solo si lo es semánticamente.

Es importante hacer una apreciación respecto a las definiciones de completitud y consistencia sintácticas: a pesar de que las definimos para sistemas formales, es sencillo extender esta definición a sistemas lógicos, teniendo en cuenta su sintaxis.

De este modo, decimos que un sistema lógico $\mathcal{G}$ es sintácticamente completo si, dada una fórmula ϕ de $\mathcal{G}$, ϕ o $\neg\phi$ son teoremas. Análogamente, $\mathcal{G}$ es sintácticamente consistente si no es posible que tanto ϕ como $\neg\phi$ sean teoremas.

[14] Puedes ver tales axiomas en el apéndice C.
[15] Nótese la ironía.
[16] Es decir, que en el sistema formal esté bien definida la noción de *negación* ($\neg$).
[17] Es posible que tanto ϕ como $\neg\phi$ sean teoremas.

Las definiciones que hemos hecho hasta este punto (sin incluir la de decidibilidad) se encuentran resumidas en la figura 5.2.

sean $\mathcal{G}$ sistema lógico, ϕ fórmula de $\mathcal{G}$

$\mathcal{G}$ sólido (definición 5.5)	:	ϕ teorema $\Rightarrow$ ϕ verdadero
$\mathcal{G}$ (semánticamente) completo (definición 5.6)	:	ϕ verdadero $\Rightarrow$ ϕ teorema
$\mathcal{G}$ sintácticamente completo* (definición 5.9)	:	ϕ o $\neg\phi$ son teoremas
$\mathcal{G}$ sintácticamente consistente* (definición 5.10)	:	no son ϕ y $\neg\phi$ teoremas
$\mathcal{G}$ semánticamente consistente (definición 5.11)	:	no son ϕ y $\neg\phi$ verdaderos

*definición también aplicable a sistemas formales

Figura 5.2.: Algunas de las propiedades de los sistemas lógicos definidas en este capítulo

6. Los Teoremas de Incompletitud

> *Si una "religión" se define como un sistema de ideas que contiene afirmaciones indemostrables, entonces Gödel nos enseñó que las matemáticas no sólo son una religión, sino que son la única religión que puede demostrar que lo es.*
>
> — John D. Barrow, [4]

Los matemáticos de principios del siglo XX, tal y como Leibniz, Boole y Hilbert, imaginaron que sería posible determinar la veracidad de fórmulas matemáticas de forma automática. Nosotros ahora nos preguntamos lo mismo: hemos trabajado con sistemas lógicos sencillos que son decidibles, es decir, cuyas fórmulas pueden ser comprobadas verdaderas mediante programas. En el caso de sistemas lógicos de más complejidad, como la aritmética de Peano, esto no es posible. Esto se debe a que la existencia de problemas computacionales indecidibles conduce directamente a la existencia de afirmaciones matemáticas no decidibles.

6.1. El Primer Teorema de Incompletitud

Antes de proceder la demostración de una primera versión del Primer Teorema de Incompletitud, discutiremos de forma detallada el programa 6.1. En la línea 3 importamos la función `es_teorema_peano`, cuya existencia probamos en la proposición 5.8. Recuerda que, como los axiomas de **Peano** son recursivamente enumerables (es decir, **Peano** es recursivamente axiomatizable), podemos escribir el programa `es_teorema_peano(fórmula)` que devuelve `'sí'` si `fórmula` es un teorema en **Peano**, y que no para en caso contrario (ver proposición 5.8).

```
1  import utilidades
2
3  from es_teorema_peano import es_teorema_peano # NO es un oráculo
4  from parada_a_peano import parada_a_peano # NO es un oráculo
5
6  def godel_peano(entrada):
7      programa_godel = utilidades.leer('godel\_peano.py')
8      para_en_peano = parada_a_peano(programa_godel)
9      no_para_en_peano = 'NOT (' + para_en_peano + ')'
10
11     if es_teorema_peano(no_para_en_peano) == 'sí':
12         return 'para'
13     else:
14         utilidades.ciclar()
```

Programa 6.1: `godel_peano.py`

La línea 4 importa la función `parada_a_peano`, cuya existencia ya discutimos tras la proposición 5.5. Esta función traduce, dado un programa P, la afirmación "P para con entrada vacía" en una fórmula de **Peano**.

En la línea 7 guardamos el propio programa `godel_peano.py` en la variable `programa_godel`. En la línea 8 usamos la función `parada_a_peano` recién comentada, y en la línea 9 negamos la fórmula `para_en_peano`. De este modo:

`no_para_en_peano` significa que "`godel.py` no para con entrada vacía" (⋆1)

Ahora es cuando las cosas se ponen interesantes. Si `no_para_en_peano` es un teorema, la función `es_teorema_peano` devolverá `'sí'`, y de este modo el programa verificará la condición de la línea 11 y devolverá `'para'`, haciendo que el programa pare. Si `no_para_en_peano` no es un teorema, se entrará en un bucle infinito (`es_teorema_peano` no parará, y por lo tanto tampoco `godel_peano.py`). Es claro que la línea 14 nunca se ejecutará, pero la incluimos para enfatizar este hecho.

Ahora, nos preguntamos: ¿cuál es el comportamiento de `godel_peano.py` con entrada vacía?

Proposición 6.1. *El programa 6.1 no para con entrada vacía.*

Demostración. Asumimos que `godel_peano.py` para con entrada vacía, y entraremos en contradicción. La única forma de que pare es en la línea 12, algo que ocurre solamente si:

```
es_teorema_peano(no_para_en_peano) == 'sí'
```
(⋆2)

(⋆2) es equivalente a decir que `no_para_en_peano` es un teorema. Por la solidez de **Peano**, `no_para_en_peano` es verdadero.

Pero el hecho de que `no_para_en_peano` sea verdadero, junto con (⋆1), nos dice que `godel_peano.py` no para con entrada vacía, encontrando la contradicción esperada. □

Ahora estamos en condiciones de probar el resultado principal de este trabajo.

Teorema 6.1 (Incompletitud de la aritmética de Peano)**.** *La aritmética de Peano es incompleta: existen fórmulas verdaderas de* **Peano** que no son teoremas. Concretamente, la fórmula guardada en la variable `no_para_en_peano` en la línea 9 del programa 6.1 es un ejemplo de una fórmula verdadera pero no demostrable (no es un teorema) de **Peano**.

Demostración. Debemos mostrar que `no_para_en_peano` es verdadero y no es un teorema.

Para probar que es verdadero, por la proposición 6.1, sabemos que `godel_peano.py` no para con entrada vacía. Recordando (⋆1), inmediatamente obtenemos que `no_para_en_peano` es verdadero.

Para probar que no es un teorema, de nuevo usamos el hecho de que `godel_peano.py` no para con entrada vacía. Esto nos indica que, en la línea 11, la llamada a la función

```
es_teorema_peano(no_para_en_peano)
```

no puede devolver `'sí'`. Pero esto inmediatamente implica que `no_para_en_peano` no es demostrable, lo que completa la prueba. □

Acabamos de demostrar que **Peano** es incompleto, como ya anticipamos en la tabla 5.1. Las consecuencias de este hecho las exploramos en detalle en el capítulo 8. Sin embargo, paremos un momento a analizar qué es lo que acabamos de probar.

Para encontrar una fórmula verdadera y no demostrable, hemos necesitado un sistema lógico que sea, en primer lugar, recursivamente axiomatizable. Si no, no hubiesemos podido crear un programa capaz de saber cuándo una fórmula del sistema es un teorema. Por otra parte, hemos asumido que el sistema es sólido, lo cual es algo esperable. Finalmente, hemos exigido que sea capaz de expresar una cierta afirmación de las máquinas de Turing (véase proposición 5.5). Sin embargo, en ningún momento hemos utilizado el sistema **Peano** en sí. Esto nos indica que podríamos realizar esta demostración con otro sistema que no sea **Peano**, siempre y cuando verifique tales condiciones. Indicamos este resultado a continuación.

Teorema 6.2 (Versión semántica del Primer Teorema de Incompletitud)**.** *Sea $\mathcal{G}$ un sistema lógico que verifique las siguientes condiciones:*

(A1) *$\mathcal{G}$ es sólido.*

(A2) *$\mathcal{G}$ es recursivamente axiomatizable.*

(A3) *$\mathcal{G}$ puede expresar afirmaciones sobre máquinas de Turing o, equivalentemente, sobre programas;*[1] *en concreto, dada una máquina de Turing M, puede expresar que "M para con entrada vacía" o, equivalentemente, dado un programa P, puede expresar que "P para con entrada vacía".*

Entonces, $\mathcal{G}$ es incompleto, es decir, existen fórmulas verdaderas de $\mathcal{G}$ que no son teoremas.

Para reforzar este importante resultado, incluimos un resumen del proceso que hemos seguido para **Peano**, de forma general, en la demostración de este teorema.

Demostración. En el programa 6.2 importamos dos funciones. Por una parte, en la línea 3 importamos la función `es_teorema_en_G`, que comprueba si una fórmula es un teorema en $\mathcal{G}$. Esta función parará si la fórmula es un teorema, y se ejecutará indefinidamente en caso de que no lo sea. La existencia de esta función la garantiza (A2).

Por otra, en la línea 4 importamos la función `parada_a_G` que, dado un programa P, convierte la afirmación "P para con entrada vacía" en una fórmula de $\mathcal{G}$, algo que es posible por (A3).

En la línea línea 9 tenemos que:

`no_para_en_G` significa que "`godel.py` *no para con entrada vacía*" (en $\mathcal{G}$) ($\star$3)

Si `no_para_en_G` es un teorema, la función `es_teorema_en_G(no_para_en_G)` (línea 11) parará y devolverá `'sí'`, mientras que la misma función ciclará en caso de que no lo sea.

La pregunta que debemos hacernos es: ¿qué le pasa a `godel.py` cuando `entrada` es vacía?

Si asumimos que `godel.py` para con entrada vacía, la única forma de que lo haga es porque en la línea 11 es `es_teorema_en_G(no_para_en_G) == 'sí'`, lo cual solo puede pasar si `no_para_en_G` es un teorema. Pero, por (A1), y dado que $\mathcal{G}$ es sólido, tenemos que `no_para_en_G` es verdadero.

Pero si `no_para_en_G` es verdadero, por ($\star$3), esto querría decir que `godel.py` no para con entrada vacía, entrando en contradicción, por lo que debe ser:

[1] La equivalencia la probamos en el teorema 3.1.

`godel.py` no para con entrada vacía (⋆4)

Comprobamos que `no_para_en_G` es verdadero pero no es un teorema. En efecto, es verdadero por (⋆4). Sin embargo, no es un teorema. Para ello, usamos de nuevo (⋆4). El hecho de que `godel.py` no pare indica que es imposible que la condición de la línea 11, `es_teorema_en_G(no_para_en_G) == 'sí'`, se verifique – en otras palabras, `no_para_en_G` no es un teorema. □

```
1  import utilidades

3  from es_teorema_en_G import es_teorema_en_G # NO es un oráculo
4  from parada_a_G import parada_a_G # NO es un oráculo

6  def incompleto_semantico(entrada):
7      programa = utilidades.leer('incompleto_semantico.py')
8      para_en_G = parada_a_G(programa)
9      no_para_en_G = 'NOT (' + para_en_G + ')'

11     if es_teorema_en_G(no_para_en_G) == 'sí':
12         return 'para'
13     else:
14         utilidades.ciclar()
```

Programa 6.2: `godel.py`

Nuestro resultado nos dice que, si tenemos un sistema lógico que sea recursivamente axiomatizable y que verifique (A3), no podemos esperar que pueda ser sólido y completo al mismo tiempo. Es decir, si todos los teoremas del sistema son verdaderos, siempre podremos encontrar fórmulas del sistema que sean verdaderas pero no sean teoremas.

Podemos aplicar el teorema 6.2 a otros sistemas lógicos, siempre que verifiquen las hipótesis. Uno de ellos es, por ejemplo, el sistema de axiomas de Zermelo-Fraenkl.

El resultado que acabamos de probar es semántico. Sin embargo, es posible probar otro resultado, esta vez sintáctico, y más parecido al demostrado por Gödel.

Para poder probar esta variación, debemos de tener en cuenta una sutileza del problema EsTeorema en el caso de sistemas formales sintácticamente consistentes.

Proposición 6.2. *Sea $\mathcal{S}$ un sistema lógico recursivamente axiomatizable y sintácticamente consistente. Entonces,* EsTeorema$_{\mathcal{S}}$ *es decidible.*

Demostración. Veamos que el programa 6.3 decide EsTeorema$_{\mathcal{S}}$. En efecto, vemos en primer lugar que en la línea 2 importamos la función `es_prueba_S`, la cual es computable por ser $\mathcal{S}$ recursivamente axiomatizable.

A continuación, en la línea 5 almacenamos la negación de `formula` en `neg_formula`. En la línea 6 declaramos la demostración.

Por ser $\mathcal{S}$ consistente, bien `formula` bien `neg_formula` tiene una demostración. En el bucle de la línea 8, vamos comprobando si `demostracion` es una demostración de `formula` o de `neg_formula`.

En el primer caso, devolvemos `'sí'`, mientras que en el segundo, devolvemos `'no'`. En caso de que `demostracion` no sea una demostración de ninguna de ambas fórmulas, procedemos en la línea 13 a enumerar la siguiente demostración en orden *shortlex* (ver proposición 5.8).

Por la consistencia de $\mathcal{S}$, debe existir una demostración bien de `formula` o de `neg_formula`, por lo que el programa debe parar al llegar a ser `demostracion` tal demostración. □

```
1  import utilidades
2  from es_prueba_S import es_prueba_S # NO es un oráculo

4  def es_teorema_S(formula):
5      negacion_formula = 'NOT (' + formula + ')'
6      demostracion = ''

8      while True:
9          if es_prueba_S(demostracion, formula) == 'sí':
10             return 'sí'
11         if es_prueba_S(demostracion, negacion_formula) == 'sí':
12             return 'no'
13         demostracion = utilidades.siguiente_string(demostracion)
```

Programa 6.3: `es_teorema_S.py`

Ahora estamos en condiciones de probar el resultado sintáctico, que dependerá de la no decidibilidad de AdivinaConsistente (problema 4.11).

Teorema 6.3 (Versión sintáctica del Primer Teorema de Incompletitud)**.** *Sea $\mathcal{S}$ un sistema formal que verifique las siguientes condiciones:*

(B1) *$\mathcal{S}$ es sintácticamente consistente.*

(B2) *$\mathcal{S}$ es recursivamente axiomatizable.*

(B3) *$\mathcal{S}$ puede expresar afirmaciones sobre máquinas de Turing o, equivalentemente, sobre programas; en concreto, dada una máquina de Turing M, puede expresar que "M para con entrada vacía" o, equivalentemente, dado un programa P, puede expresar que "P para con entrada vacía".*

Entonces, $\mathcal{S}$ es sintácticamente incompleto, es decir, existen fórmulas ϕ de $\mathcal{S}$ tal que ni ϕ ni $\neg\phi$ son teoremas.

Demostración. Vamos a suponer que $\mathcal{S}$ es sintácticamente completo y veremos que, en tal caso, podríamos decidir AdivinaConsistente (problema 4.11), lo cual entraría en contradicción con la proposición 4.8.

Comencemos comentando las funciones importadas. En la línea 2 importamos la función `oarada_en_S`, que traduce para un programa P la afirmación "P para con entrada vacía" en una fórmula de S, y cuya existencia garantiza la hipótesis (B3). En la línea 3 importamos la función `es_teorema_S`, que decide el problema $\text{EsTeorema}_{\mathcal{S}}$, y que, por las hipótesis (B1) y (B2), siempre para (ver proposición 6.2) y devuelve `'sí'` si la fórmula de entrada es un teorema en $\mathcal{S}$ y `'no'` en caso contrario.

En la línea 6 usamos la función `parada_a_S` para almacenar en `programa_para` la fórmula de $\mathcal{S}$ equivalente a "`programa` para con entrada vacía". En la línea 8, asignamos a la variable `es_teorema` uno de dos valores posibles: `'sí'` o `'no'`, como ya comentamos anteriormente.

Llegado a la línea 10, implementamos la siguiente lógica:

- Si `programa` acepta con entrada vacía (para y devuelve `'sí'`), la condición de la línea 10 se verificará, así como la de la línea 12, y `adivina_consistente.py` devolverá `'sí'`.
- Si `programa` rechaza con entrada vacía (para y devuelve `'no'`, la condición de la línea 10 se verificará, mientras que no la de la línea 12, y `adivina_consistente.py` devolverá `'no'`.
- Si `programa` cicla con entrada vacía, la condición de la línea 10 no se verificará, y `adivina_consistente.py` devolverá `'no'`.

Dicha lógica hace evidente que el programa decide AdivinaConsistente. □

```
1  import utilidades
2  from parada_a_S import parada_a_S # NO es un oráculo
3  from es_teorema_S import es_teorema_S # NO es un oráculo

5  def adivina_consistente(programa):
6      programa_para = parada_a_S(programa)

8      es_teorema = es_teorema_S(programa_para)

10     if es_teorema == 'sí':
11         salida = maquina_universal(programa, '')
12         if salida == 'sí':
13             return 'sí'
14         else:
15             return 'no'
16     else:
17         return 'no'
```

Programa 6.4: `adivina_consistente.py`

Hemos probado dos versiones del Primer Teorema de Incompletitud. Resumimos las diferencias entre ambas a continuación.

versión semántica (teorema 6.2)	**versión sintáctica** (teorema 6.3)
$\mathcal{G}$ sistema lógico sólido (y A2, A3)	$\mathcal{S}$ sistema formal sintácticamente consistente (y B2, B3)
ϕ teorema $\Rightarrow$ ϕ verdadero	$\nexists\phi : \phi$ y $\neg\phi$ son teoremas
$\Downarrow$	$\Downarrow$
$\mathcal{G}$ no (semánticamente) completo	$\mathcal{S}$ no sintácticamente completo
$\exists\phi : \phi$ es verdadero y no es teorema	$\exists\phi :$ ni ϕ ni $\neg\phi$ son teoremas

Figura 6.1.: Comparativa entre las dos versiones del Primer Teorema de Incompletitud

La demostración original de Gödel es puramente sintáctica, y muy parecida a la probada en el teorema 6.3. La enunciamos a continuación.

Teorema 6.4 (Primer Teorema de Incompletitud de Gödel-Rosser)**.** *Sea $\mathcal{S}$ un sistema formal que verifique las siguientes condiciones:*

(C1) *$\mathcal{S}$ es sintácticamente consistente.*

(C2) *$\mathcal{S}$ es recursivamente axiomatizable.*

(C3) *$\mathcal{S}$ contiene una cierta cantidad de aritmética elemental.*

Entonces, $\mathcal{S}$ es sintácticamente incompleto, es decir, hay fórmulas ϕ de $\mathcal{S}$ de modo que ni ϕ ni $\neg\phi$ son teoremas.

Tal y como en nuestra versión sintáctica, requerimos que el sistema sea sintácticamente consistente y recursivamente axiomatizable. El teorema original de Gödel exigía como hipótesis una forma más fuerte de consistencia sintáctica llamada *ω-consistencia*. Esta forma más general del teorema se debe al trabajo de John B. Rosser, autor del hoy conocido como *truco de Rosser*.[2] [38]

La tercera hipótesis exige que el sistema sea capaz de probar ciertas afirmaciones de aritmética elemental. Esto se debe a que Gödel pretendía probar que la teoría de números no era axiomatizable (ver capítulo 2), por lo que usó en su demostración una forma de numerar los teoremas que hoy en día se conoce como *numeración de Gödel*.

6.2. El Segundo Teorema de Incompletitud

El Segundo Teorema de Incompletitud, en su versión sintáctica, puede obtenerse inmediatamente a partir del Primer Teorema de Incompletitud.

[2] Es por ello por lo que lo llamamos teorema de "Gödel-Rosser", para distinguirlo del resultado original de Gödel.

Lema 6.1. *Sea la siguiente fórmula un teorema:*

$$\phi_1 \Rightarrow \phi_2$$

Entonces, si ϕ_2 no es un teorema, ϕ_1 tampoco lo es.

Demostración. Este resultado se obtiene de la implicación inmediata siguiente: si ϕ_1 es un teorema, entonces ϕ_2 ha de serlo también, pues se deduce inmediatamente de $\phi_1 \Rightarrow \phi_2$ mediante *modus ponens*. Concluimos la prueba por el contrarrecíproco. □

Teorema 6.5 (Versión semántica del Segundo Teorema de Incompletitud)**.** *Sea $\mathcal{G}$ un sistema lógico que verifique las siguientes condiciones:*

(A2) *$\mathcal{G}$ es recursivamente axiomatizable.*

(A3) *$\mathcal{G}$ puede expresar afirmaciones sobre máquinas de Turing o, equivalentemente, sobre programas; en concreto, dada una máquina de Turing M, puede expresar que "M para con entrada vacía" o, equivalentemente, dado un programa P, puede expresar que "P para con entrada vacía".*

Entonces, la fórmula correspondiente a "$\mathcal{G}$ es sólido" no es un teorema en $\mathcal{G}$.

Demostración. El teorema 6.2 nos dice que, suponiendo (A2) y (A3), en caso de que $\mathcal{G}$ fuese sólido, entonces tendríamos una fórmula ϕ que (1) es verdadera pero (2) no es un teorema.

Como hemos logrado probar (1) en $\mathcal{G}$, tenemos el siguiente teorema:

$$\text{“}\mathcal{G} \text{ es sólido”} \Rightarrow \phi$$

Ahora bien, aplicando (2) y el lema 6.1, tenemos que "$\mathcal{G}$ es sólido" no es un teorema. □

Nota: 6.1 *para probar la versión sintáctica (que es el verdadero segundo teorema de incompletitud) sería necesario encontrar una versión constructiva del teorema 6.3, que nos diese una fórmula ϕ que fuese verdadera de modo que ni ϕ ni $\neg\phi$ sean teoremas. Pero esto es un tanto extraño porque si el teorema es sintáctico, ¿qué quiere decir que una fórmula sea verdadera?*

6.3. Consecuencias

Parte IV.

Conclusión

...

7. Resumen final

8. Conclusión

> *Sabemos que cada época tiene sus propios problemas, que la siguiente resuelve o desecha por inútiles y sustituye por otros nuevos.*
>
> — David Hilbert, [20]

WIP

8.1. Implicaciones matemáticas

8.2. Implicaciones filosóficas

Parte V.

Apéndices

A. Cómo usar el código de este trabajo

Junto a esta memoria, el trabajo incluye un repositorio de código que puedes acceder en:

https://github.com/mianfg-DGIIM/TFG

En el repositorio podrás ver la siguiente estructura de archivos:

```
/TFG
├── memoria ..................................................... Memoria del trabajo
│   ├── memoria.pdf ..................................... Archivo PDF de esta memoria
│   └── tex ..................................... Contiene el código en LaTeX de la memoria
│       ├── memoria.tex
│       └── ...
└── codigo ..................................................... Todo el código del trabajo
    ├── maquinas_turing ..................... Contiene descripciones de máquinas de Turing
    │   ├── mas_a_que_b.mt
    │   └── ...
    ├── c_d.py
    └── ...
```

En la carpeta `codigo` se encuentran:

- Por una parte, las descripciones de máquinas de Turing comentadas en la proposición 3.5, como archivos `.mt` dentro de la carpeta `maquinas_turing`.
- Los programas desarrollados en este trabajo, junto con algunos otros.

Puedes ejecutar los programas de la carpeta `codigo` importando las funciones que quieras en el intérprete de Python. En la librería `utilidades.py` aparecen múltiples funciones que se usan en varios programas, y que son útiles para poder ejecutar otros. A modo de ejemplo, si queremos ejecutar la función `simula_turing_mult` de `simula_turing.py`, haremos:

```
TFG/codigo$ python
>>> from utilidades import leer
>>> from simula_turing import simula_turing_mult
>>> codificacion = leer('./maquinas_turing/mas_a_que_b.mt')
>>> entrada = 'abaaabbb'
>>> simula_turing_mult(codificacion, entrada)
'q_0 : X X X X X X X X [_] (rechaza)'
```

Tienes una referencia completa de los programas desarrollados en el repositorio. Algunos de ellos no pueden ejecutarse, y aparecen como referencia. Los motivos por los que esto es así también se explican en el repositorio.

B. Demostraciones adicionales

Proposición B.1. *Toda máquina de Turing multicinta puede ser simulada por una máquina de Turing de una sola cinta.*

Demostración. Sea $M = (Q, A, B, \delta, q_0, \sqcup, F)$ una máquina de Turing con k cintas. Definiremos la máquina $\widetilde{M} = (\widetilde{Q}, A, \widetilde{B}, \widetilde{\delta}, q_0, [\sqcup^k], F)$. Para ello, definimos en primer lugar $\breve{B}$ como:

$$\breve{B} = \{\breve{b} : b \in B\}$$

Para definir $\widetilde{B}$, observemos que un símbolo de un alfabeto no tiene por qué ser únicamente un carácter. Para representar un único símbolo, lo rodearemos de corchetes ([ ]). Por tanto, el siguiente alfabeto tiene sentido:

$$\widetilde{B} = \{[u_1\, u_2 \ldots u_k] : u_i \in B \cup \breve{B}\} \cup \{\#\}$$

En $\widetilde{B}$ insertamos, además de los símbolos que hemos comentado, un símbolo delimitador #.

El conjunto de estados $\widetilde{Q}$ lo hacemos más amplio que Q. Para ello, numeraremos todos los estados de Q:

$$Q = \{q_0, q_1, \ldots, q_m\}$$

Y definimos el nuevo conjunto de estados como:

$$\widetilde{Q} = Q \cup \left(\bigcup_{i=0}^{m} \bigcup_{j=1}^{k} \left\{ \widetilde{q}_{i,j},\ \widetilde{q}_{i,j}^{(R)},\ \widetilde{q}_{i,j}^{(I)},\ \widetilde{q}_{i,j}^{(Is)},\ \widetilde{q}_{i,j}^{(D)},\ \widetilde{q}_{i,j}^{(Ds)} \right\} \right)$$

Definimos como configuración inicial de la máquina con entrada $u = u_1\, u_2 \ldots u_n \in A^*$ a:[1]

$$q_0 \ :\ \boxed{\#}\, [\widetilde{u_1}\ \underbrace{\breve{\sqcup} \ldots \breve{\sqcup}}_{k-1)}]\, [u_2\ \underbrace{\sqcup \ldots \sqcup}_{k-1)}] \ldots [u_n\ \underbrace{\sqcup \ldots \sqcup}_{k-1)}]\, \#$$

Describiremos $\widetilde{\delta}$ poco a poco. En primer lugar, queremos partir del mismo estado que la máquina multicinta (esto nos permitirá que el lenguaje aceptado sea el mismo, como veremos más adelante). En el momento en el que la máquina se encuentre en este estado, nos movemos a la derecha y comenzaremos a buscar la posición en la cinta del primer cabezal. Usaremos el estado $\widetilde{q}_{i,j}$ para buscar el cabezal j-ésimo, estando la máquina multicinta en el estado q_i. Añadimos para hacer esto las siguientes transiciones:

$$\widetilde{\delta}(q_i, \#) = (\widetilde{q}_{i,1}, \#, D) \quad \text{para cada} \quad i = 0, 1, \ldots, m$$

[1] Siempre es posible cambiar la posición de la configuración inicial de una máquina de Turing, ejecutando las transiciones necesarias.

Para buscar el cabezal j-ésimo, estando la máquina multicinta en el estado q_i, nos iremos moviendo a la derecha mientras que no hayamos encontrado un carácter de $\breve{B}$ en la posición j-ésima. Para ello, añadimos para cada $i = 0, 1, ..., m$ y $j = 1, 2, ..., k$:

$$\widetilde{\delta}(\widetilde{q}_{i,j}, [w_1\, w_2\, ...\, w_l\, ...\, w_k]) = (\widetilde{q}_{i,j}, [w_1\, w_2\, ...\, w_l\, ...\, w_k], D) \quad \text{con} \quad w_j \in B, w_l \in B \cup \breve{B} \text{ para } l \neq j$$

Nótese que, de forma explícita, hacemos que el símbolo w_j no pueda estar en $\breve{B}$ (de aquí en adelante, si no explicitamos, será $w_l \in B \cup \breve{B}$ para cada $l = 0, 1, ..., k$). Esto es para continuar moviéndonos a la derecha hasta encontrar un carácter con cabezal en la posición j-ésima de cada símbolo.

Ahora estamos en condiciones de incluir las transiciones de δ en $\widetilde{\delta}$. Sea una transición de δ:

$$\delta(q_i, b_1, b_2, ..., b_k) = (q_j, c_1, M_1, c_2, M_2, ..., c_k, M_k)$$

Para cada una de estas transiciones, deberemos insertar reglas en $\widetilde{\delta}$. Estas transiciones dependerán de cómo se mueva el cabezal, ya que si lo hace a la derecha o a la izquierda hasta toparse con el delimitador #, deberemos extender la cinta de $\widetilde{M}$ con un símbolo blanco de $\widetilde{M}$ (que es $[\sqcup^k]$), y desplazar el delimitador # una posición. Dada la transición anterior, y para cada $l = 1, 2, ..., k$:

- Si $M_l = D$, insertamos varias transiciones. En primer lugar, la transición correspondiente a sustituir el símbolo en el que se encuentra el cabezal, b_l, por el correspondiente, c_l:

  $$\widetilde{\delta}(\widetilde{q}_{i,l}, [w_1\, w_2\, ...\, w_{l-1}\, \breve{b_l}\, w_{l+1}\, ...\, w_k]) = (\widetilde{q}^{(D)}_{i,l}, [w_1\, w_2\, ...\, w_{l-1}\, c_l\, w_{l+1}\, ...\, w_k], D)$$

  A continuación simulamos que el cabezal l-ésimo se mueve a la derecha. En caso de que no hayamos alcanzado el separador #, la transición es directa, simplemente indicamos que el cabezal está en la posición determinada, y volvemos al primer delimitador # mediante el estado $^{(R)}$ (que veremos más adelante):

  $$\widetilde{\delta}(\widetilde{q}^{(D)}_{i,l}, [w_1\, w_2\, ...\, w_{l-1}\, w_l\, w_{l+1}\, ...\, w_k]) = (\widetilde{q}^{(R)}_{i,l}, [w_1\, w_2\, ...\, w_{l-1}\, \widetilde{w_l}\, w_{l+1}\, ...\, w_k], I)$$

  En caso de que nos topemos con el delimitador #, iniciaremos una "subrutina" indicada con los estados $^{(Ds)}$, en el que añadiremos un símbolo blanco y moveremos el delimitador a la derecha, siguiendo al estado $^{(R)}$:

  $$\widetilde{\delta}(\widetilde{q}^{(D)}_{i,l}, \#) = (\widetilde{q}^{(Ds)}_{i,l}, [\underbrace{\sqcup ... \sqcup}_{l-1)}\ \breve{\sqcup}\ \underbrace{\sqcup ... \sqcup}_{k-l)}], D)$$

  $$\widetilde{\delta}(\widetilde{q}^{(Ds)}_{i,l}, [\sqcup^k]) = (\widetilde{q}^{(R)}_{i,l}, \#, I)$$

- Si $M_l = I$, añadimos de forma análoga las transiciones:

$$\widetilde{\delta}(\widetilde{q}_{i,l}, [w_1\, w_2\, ...\, w_{l-1}\, \breve{b_l}\, w_{l+1}\, ...\, w_k]) = (\widetilde{q}_{i,l}^{(I)}, [w_1\, w_2\, ...\, w_{l-1}\, c_l\, w_{l+1}\, ...\, w_k], I)$$
$$\widetilde{\delta}(\widetilde{q}_{i,l}^{(I)}, [w_1\, w_2\, ...\, w_{l-1}\, w_l\, w_{l+1}\, ...\, w_k]) = (\widetilde{q}_{i,l}^{(R)}, [w_1\, w_2\, ...\, w_{l-1}\, \breve{w_l}\, w_{l+1}\, ...\, w_k], I)$$
$$\widetilde{\delta}(\widetilde{q}_{i,l}^{(I)}, \#) = (\widetilde{q}_{i,l}^{(Is)}, [\underbrace{\sqcup ... \sqcup}_{l-1)}\ \breve{\sqcup}\ \underbrace{\sqcup ... \sqcup}_{k-l)}], I)$$
$$\widetilde{\delta}(\widetilde{q}_{i,l}^{(Is)}, [\breve{\sqcup}^k]) = (\widetilde{q}_{i,l}^{(R)}, \#, D)$$

- Si $M_l = S$, la transición es más inmediata, dado que no debemos mover el cabezal, simplemente reemplazar el símbolo y cambiar a $^{(R)}$:

$$\widetilde{\delta}(\widetilde{q}_{i,l}, [w_1\, w_2\, ...\, w_{l-1}\, \breve{b_l}\, w_{l+1}\, ...\, w_k]) = (\widetilde{q}_{i,l}^{(R)}, [w_1\, w_2\, ...\, w_{l-1}\, \breve{c_l}\, w_{l+1}\, ...\, w_k], I)$$

Ahora, nos queda una forma de cambiar la cinta que estamos analizando. Comenzaremos añadiendo para ello un estado de retorno, que posiciona el cabezal en el primer delimitador #. Para cada $i = 0, 1, ..., m$ y $j = 1, 2, ..., k$ añadimos las transiciones:

$$\widetilde{\delta}(\widetilde{q}_{i,j}^{(R)}, [w_1\, w_2\, ...\, w_l\, ...\, w_k]) = (\widetilde{q}_{i,j}^{(R)}, [w_1\, w_2\, ...\, w_l\, ...\, w_k], I)$$

Para leer la cinta siguiente, simplemente añadimos estas transiciones para cada $i = 0, 1, ..., m$ y $j = 1, 2, ..., k-1$:

$$\widetilde{\delta}(\widetilde{q}_{i,j}^{(R)}, \#) = (\widetilde{q}_{i,j+1}^{(R)}, \#, D)$$

Si ya hemos leído la última cinta, cambiamos el estado a uno de los de Q de acuerdo a la función de transición. De nuevo revisitamos δ. Para cada transición:

$$\delta(q_i, b_1, b_2, ..., b_k) = (q_j, c_1, M_1, c_2, M_2, ..., c_k, M_k)$$

añadimos la transición de cambio de estado:

$$\widetilde{\delta}(\widetilde{q}_{i,k}^{(R)}, \#) = (q_j, \#, S)$$

Ya hemos definido todas las transiciones necesarias. Ahora, comprobaremos que la simulación es correcta, es decir, que $L(M) = L(\widetilde{M})$. En efecto, hemos construido una máquina $\widetilde{M}$ que ejecuta el mismo proceso de cálculo que M, y que además pasa por los mismos estados Q de M tras modificar la cinta. En otras palabras, la máquina M Es precisamente por esto por lo que, en caso de la máquina parar, se encontrará en un estado $q \in Q$ (la hemos construido para que ocurra esto). De este modo, el lenguaje aceptado por ambas máquinas coincide, dado que usamos el mismo conjunto de estados finales F. □

Proposición B.2. *Para cada problema $F : X \longrightarrow Y$, existe un problema equivalente $\widehat{F} : S \longrightarrow S$.*

Demostración. Si queremos resolver un problema de ordenador, deberemos codificar las entradas y las salidas mediante un formato que pueda entender. La idea es poder traducir un problema F a otro problema $\widehat{F}$ de *strings*, de modo que procese la misma función F para entradas correctamente codificadas, y se devuelva `'no'` en caso de que la codificación de entrada sea incorrecta.

Para formalizar este concepto convenientemente, definiremos los conjuntos $X^{\oslash} = X \cup \{\oslash\}$ y $Y^{\oslash} = Y \cup \{\oslash\}$. A $\oslash$ lo llamaremos el *elemento error*.

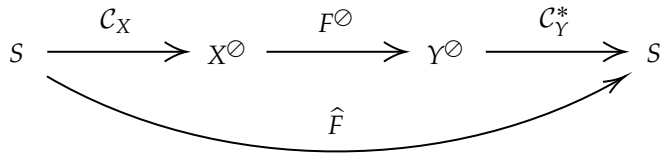
Expandiremos el problema F para aceptar el elemento error como entrada, de modo que su solución sea el elemento error. Lo llamaremos $F^{\oslash} : X^{\oslash} \longrightarrow Y^{\oslash}$, definido como:

$$F^{\oslash}(x) = \begin{cases} F(x), & \text{si} \quad x \in X \\ \oslash, & \text{si} \quad x = \oslash \end{cases}$$

Sea $\mathcal{C}_X : S \longrightarrow X^{\oslash}$ la función que, para cada codificación en S, asigna la entrada de X correspondiente. Si la codificación es inválida, asigna $\oslash$.

Sea $\mathcal{C}_Y^* : Y^{\oslash} \longrightarrow S$ la función que, para cada solución de Y, asigna la codificación en S correspondiente. Para la solución error $\oslash$, asigna la codificación `'no'`. Del mismo modo que para las entradas, también podemos definir $\mathcal{C}_Y : S \longrightarrow Y^{\oslash}$, que para cada codificación de una solución en S, obtiene su solución en Y. Para codificaciones inválidas, asigna $\oslash$.

Ahora estamos en condiciones de definir $\widehat{F} = \mathcal{C}_Y^* \circ F^{\oslash} \circ \mathcal{C}_X$.



El problema $\widehat{F}$ resuelve el mismo problema que F, y es capaz de tratar con entradas erróneas. Basta ver que, si $I \in S$ es una codificación correcta de una entrada $x \in X$ (es decir, $\mathcal{C}_X(I) = x$), entonces es $\widehat{F}(I) = \mathcal{C}_Y^*(F^{\oslash}(\mathcal{C}_X(I))) = \mathcal{C}_Y^*(F^{\oslash}(x)) = \mathcal{C}_Y^*(F(x)) = O$, siendo O la codificación de $F(x)$, $O = \mathcal{C}_Y^*(F(x))$. Dada la codificación de la solución O, es fácil obtener la solución $F(x) = \mathcal{C}_Y(O)$.

En el caso de que $I \in S$ sea una codificación de entrada incorrecta, será $\widehat{F}(I) = \mathcal{C}_Y^*(F^{\oslash}(\mathcal{C}_X(I))) = \mathcal{C}_Y^*(F^{\oslash}(\oslash)) = \mathcal{C}_Y^*(\oslash) =$ `'no'`. □

Proposición B.3. *El sistema lógico* **SumaBinariaLógico** es consistente.

Demostración. Para probar que el sistema es consistente, veremos que todas las reglas de inferencia mapean fórmulas verdaderas a fórmulas verdaderas. En primer lugar, vemos que el axioma 1=1 es verdadero, pues es cierto que $1 = 1$.

Ahora, comprobemos cada regla:

$$\text{(R1)} \quad A\texttt{=}B \quad \mapsto \quad \texttt{1+}A\texttt{=1+}B$$

En este caso, si A y B son números binarios y asumimos que la fórmula A=B es verdadera, entonces tenemos que $A = B$. Por tanto, la fórmula 1+A=1+B es verdadera, ya que es cierto que $1 + A = 1 + B$.

$$\text{(R2)} \quad A\underline{N}\texttt{+}\underline{N}B \quad \mapsto \quad AN\texttt{0}B$$

El hecho de que las N tengan coincidencias maximales nos lleva a comprobar si dado N un número binario, asumiendo que N+N es verdadera tenemos que N0 es verdadera, o lo que es lo mismo, debemos comprobar que $N + N = N0$.

Un número binario N se representa mediante m dígitos, ya sean 0 o 1, siendo $N = n_{m-1}\, n_{m-2} \ldots n_1\, n_0$, y se verifica que:

$$N = n_{m-1} \cdot 2^{m-1} + n_{m-2} \cdot 2^{m-2} + \ldots + n_1 \cdot 2 + n_0$$

Basta ver que:

$$N + N = 2N = n_{m-1} \cdot 2^{m} + n_{m-2} \cdot 2^{m-1} + \ldots + n_1 \cdot 2^2 + n_0 \cdot 2$$

y que el número $N + N$ corresponde exactamente a $n_{m-1}\, n_{m-2} \ldots n_1\, n_0\, 0 = N\, 0$.

$$\text{(R3)} \quad A\underline{1}\texttt{+}\underline{N\texttt{0}}B \quad \mapsto \quad AN\texttt{1}B$$

De forma análoga a la regla anterior, comprobamos que:

$$1 + N\, 0 = N\, 0 + 1 = n_{m-1} \cdot 2^{m} + n_{m-2} \cdot 2^{m-1} + \ldots + n_1 \cdot 2^2 + n_0 \cdot 2 + 1 = N\, 1$$

□

Proposición B.4. *El sistema lógico* **SumaBinariaLógico** es completo.

Demostración. Veremos que toda fórmula verdadera es un teorema. Vemos en primer lugar que toda fórmula verdadera ϕ es del tipo

$$\phi = N_1\texttt{+}N_2\texttt{+}\ldots\texttt{+}N_k\texttt{=}M_1\texttt{+}M_2\texttt{+}\ldots\texttt{+}M_l$$

donde cada $N_i, M_j\ \forall i \in \{1, 2, \ldots, k\}, j \in \{1, 2, \ldots, l\}$ es un número binario (que verifica la expresión

regular `1(0|1)*`, y de modo que la siguiente fórmula aritmética es cierta:

$$N_1 + N_2 + ... + N_k = M_1 + M_2 + ... + M_l$$

Nuestro objetivo es probar que la fórmula ϕ es un teorema. Para ello, vamos a probar que, dado un número binario n que sea compilación maximal de la expresión regular `1(0|1)*`, éste puede derivarse a partir de la suma de n unos, es decir, debemos probar que:

$$A\underline{n}B \overset{*}{\mapsto} A\ \underbrace{1{+}1{+}...{+}1}_{n)}\ B$$

Dado que, por ser esto cierto, tenemos que:

$$\psi = \underbrace{1{+}1{+}...{+}1}_{N_1)} + \underbrace{1{+}1{+}...{+}1}_{N_2)} + ... + \underbrace{1{+}1{+}...{+}1}_{N_k)} = \underbrace{1{+}1{+}...{+}1}_{M_1)} + \underbrace{1{+}1{+}...{+}1}_{M_2)} + ... + \underbrace{1{+}1{+}...{+}1}_{M_l)} \overset{*}{\mapsto} \phi$$

Y como tenemos $N_1 + N_2 + ... + N_k$ unos a la izquierda de = y $M_1 + M_2 + ... + M_l$ a la derecha, y es

$$N_1 + N_2 + ... + N_k = M_1 + M_2 + ... + M_l = n$$

tenemos que

$$\psi = \underbrace{1{+}1{+}...{+}1}_{n)} = \underbrace{1{+}1{+}...{+}1}_{n)}$$

pero es claro que podemos obtener ψ aplicando la regla (R1) n veces a partir del axioma $1 = 1$, luego es

$$1{=}1 \overset{*}{\mapsto} \psi = \underbrace{1{+}1{+}...{+}1}_{n)} = \underbrace{1{+}1{+}...{+}1}_{n)} \overset{*}{\mapsto} \phi$$

y como $1{=}1 \overset{*}{\mapsto} \phi$ y es $1{=}1$ un axioma, ϕ es un teorema.

Probemos que, en efecto, es

$$A\underline{n}B \overset{*}{\mapsto} A\ \underbrace{1{+}1{+}...{+}1}_{n)}\ B$$

Para ello, primero observemos que, si m es un número binario positivo, entonces puede que sea par o impar.

Si m es impar, entonces puede ser escrito con k dígitos: $m = b_{k-1}\, b_{k-2} ... b_1\, 1$. Sea φ una fórmula de **SumaBinariaLógico**, y supongamos que φ contiene un *substring* m como compilación maximal de `1(0|1)*`, es decir, $\phi = A\ \underline{m}\ B$. Entonces, φ puede ser derivado de un cierto φ', $\varphi' \mapsto \varphi$ mediante la regla (R3), siendo

$$\varphi' = A\ \underline{b_{k-1}\, b_{k-2} ... b_1\, 0}\ B$$

En caso de que m sea par, es claro que puede expresarse como la suma $p + p$, cierto p un cierto binario positivo. Entonces, si φ contiene un *substring* m como compilación maximal de `1(0|1)*`, $\varphi = A\ \underline{m}\ B$, es $\varphi' \mapsto \varphi$ con

$$\varphi' = A\ \underline{p}{+}\underline{p}\ B$$

Sabiendo esto, cualquier ocurrencia de un entero positivo m en una fórmula de **SumaBinaria-**

Lógico puede ser reducida a la suma de m unos, es decir, dado $\varphi = A \underline{m} B$, se tiene que:

$$A \underbrace{1{+}1{+}...{+}1}_{m)} B \overset{*}{\mapsto} \varphi$$

Esto es debido a que todo número natural m puede ser reducido a la suma de m unos aplicando recursivamente las operaciones anteriormente descritas (dividir por 2 en caso de ser par y restar 1 en caso de ser impar). □

Proposición B.5. *El sistema lógico* **SumaBinariaLógicoArreglado** es completo.

Demostración. Para demostrar esto, usaremos el resultado obtenido en la proposición anterior. Toda fórmula verdadera ϕ es del tipo

$$\phi = N_1{+}N_2{+}...{+}N_k{=}M_1{+}M_2{+}...{+}M_l$$

donde cada N_i, M_j $\forall i \in \{1, 2, ..., k\}, j \in \{1, 2, ..., l\}$ es un número binario (que verifica la expresión regular `1(0|1)*`, y de modo que la siguiente fórmula aritmética es cierta:

$$N_1 + N_2 + ... + N_k \geqslant M_1 + M_2 + ... + M_l$$

Vamos a llamar a ambas partes con n y m:

$$n = N_1 + N_2 + ... + N_k, \quad m = M_1 + M_2 + ... + M_l$$

Como $n \geqslant m$, existe un $p \geqslant 0$ de modo que $n = m + p$. Ya probamos en la proposición anterior que:

$$\psi = \underbrace{1{+}1{+}...{+}1}_{p)} + \underbrace{1{+}1{+}...{+}1}_{m)} = \underbrace{1{+}1{+}...{+}1}_{m)} \overset{*}{\mapsto} \phi$$

Ahora basta ver que:

$$1{=}1 \underset{(1)}{\overset{*}{\longmapsto}} \underbrace{1{+}1{+}...{+}1}_{m)} = \underbrace{1{+}1{+}...{+}1}_{m)} \underset{(2)}{\overset{*}{\longmapsto}} \psi \overset{*}{\mapsto} \phi$$

Donde en (1) aplicamos m veces la regla (R1) y en (2) aplicamos p veces la regla (R1b). □

C. Descripción de la aritmética de Peano

A continuación se especifica con detalle el sistema lógico **Peano**, correspondiente a la aritmética de Peano de *Principia Mathematica* [46] y descrito por Gödel en su publicación en la que demostró el Primer Teorema de Incompletitud [15]. Los contenidos del sistema lógico C.1 se corresponden a una traducción de tal artículo en [14], con la ayuda de [41].

En el artículo original de Gödel, los símbolos usados son diferentes. En la tabla C.1 se especifican tales diferencias.

Sistema lógico C.1 (Aritmética de Peano)**.** Describimos las fórmulas del *lenguaje* de **Peano** como sigue:

I. Las *constantes*: $\neg$, $\vee$, $\forall$, 0, S, (y) (véase la tabla C.1).

II. Las *variables de tipo uno* (individuales, son números naturales incluyendo el 0): $x_1, y_1, z_1, ...$

Las *variables de tipo dos* (clases de individuos, subconjuntos de $\mathbb{N}$): $x_2, y_2, z_2, ...$

Las *variables de tipo tres* (clases de clases de individuos, subconjuntos de subconjuntos de $\mathbb{N}$): $x_3, y_3, z_3, ...$

Y sigue con un tipo para cada número natural.

Una nota: las variables para las funciones de dos o más términos son superfluas como signos básicos, ya que las relaciones pueden ser definidas como clases de pares ordenados y los pares ordenados pueden ser definidos como clases de clases, por ejemplo, el par ordenado a, b por $((a), (a, b))$ donde (x, y) significa la clase cuyos elementos son x e y, y x la clase cuyo único elemento es x.

Por un *signo de primer tipo* entendemos una combinación de signos de la forma:

$$a, S(a), S(S(a)), S(S(S(a))), ...$$

donde a es bien 0 o una variable de primer tipo. En el primer caso llamamos a tal símbolo un *símbolo numérico*. Para $n > 1$ entendemos un *símbolo de tipo n-ésimo* lo mismo que una *variable de tipo n-ésimo*. Las combinaciones de los símbolos de la forma $a(b)$, donde b es un símbolo de tipo n y a es un símbolo de tipo $n + 1$, se llaman *fórmulas elementales*.

Las *fórmulas* [del sistema lógico] las definimos como la menor clase que contiene todas las fórmulas elementales y, además, con cualquier a y b las siguientes: $\neg(a)$, $(a) \vee (b)$, $\forall x : (a)$ (donde x es cualquier variable). Llamamos a $(a) \vee (b)$ la *disyunción* de a y b, a $\neg(a)$ la *negación* y a $\forall x : (a)$ la *generalización* de a. Una fórmula en la que no hay ninguna variable libre[1] se llama *fórmula proposicional*. [...]

Denotamos como Subst $a\binom{v}{b}$ (donde a es una fórmula, v es una variable y b es un símbolo del

[1] Las variables libres son aquellas que no son cuantificadas por un $\forall$ o un $\exists$.

mismo tipo que v) a la fórmula que se deriva a partir de a reemplazando todas las ocurrencias de v, donde sean libres, por b.

Decimos que una fórmula a es una *elevación de tipo* de otra fórmula b si podemos obtener b incrementando el tipo de todas las variables de a por el mismo número.

Las fórmulas siguientes (I-V) son los *axiomas* [de nuestro sistema lógico]. Los indicamos con las abreviaturas $\wedge$, $\Rightarrow$, $\iff$, $\exists$ y $=$[2] y sujetos a las convenciones usuales para eliminar paréntesis:[3]

I.

$$\begin{aligned}
&(\text{P1}) \quad \neg(S(x_1) = 0)\\
&(\text{P2}) \quad S(x_1) = S(y_1) \Rightarrow x_1 = y_1\\
&(\text{P3}) \quad (x_2(0) \wedge \forall x_1 : x_2(x_1) \Rightarrow x_2(S(x_1))) \Rightarrow \forall x_1 : x_2(x_1)
\end{aligned}$$

II. Toda fórmula derivada de insertar cualquier fórmula para p, q y r en los esquemas siguientes:

$$\begin{aligned}
&(\text{P4}) \quad p \vee p \Rightarrow p\\
&(\text{P5}) \quad p \Rightarrow p \vee q\\
&(\text{P6}) \quad p \vee q \Rightarrow q \vee p\\
&(\text{P7}) \quad (p \Rightarrow q) \Rightarrow (r \vee p \Rightarrow r \vee q)
\end{aligned}$$

III. Cada fórmula obtenida de los dos esquemas:

$$\begin{aligned}
&(\text{P8}) \quad (\forall v : a) \Rightarrow \text{Subst}\, a \begin{pmatrix} v \\ c \end{pmatrix}\\
&(\text{P9}) \quad (\forall v : b \vee a) \Rightarrow (b \vee \forall v : a)
\end{aligned}$$

haciendo las siguientes sustituciones para a, v, b y c (y realizando en (P8) la operación denotada por Subst): para cada cualquier fórmula a, cualquier variable v, para una fórmula cualquiera b en la que v no aparece de forma libre, y para c un signo del mismo tipo que v, de modo que c no contenga una variable que esté enlazada a a en un lugar donde v sea libre.[4]

IV. Cada fórmula obtenida del esquema:

$$(\text{P10}) \quad \exists u : \forall v : (u(v) \iff a)$$

insertando en el lugar de v y u cualquier variable de tipo n y $n+1$, respectivamente, y en a una fórmula sin ocurrencias libres de u. Este axioma es el de *reducibilidad*.[5]

V. Cualquier fórmula obtenida de la siguiente mediante elevación de tipo (y la fórmula en

[2] Observa que: $p \wedge q$ se define como $\neg(\neg p \vee \neg q)$, $p \Rightarrow q$ como $\neg p \vee q$, $p \iff q$ como $(\neg p \vee q) \wedge (\neg q \vee p)$, $\exists x : p$ como $\neg\forall x : \neg p$, y $=$ como $\forall x_2 : (x_2(x_1) \Rightarrow x_2(y_1))$.

[3] Eliminamos los paréntesis cuando no dé lugar a confusión.

[4] c es bien 0, bien un signo de la forma $S(S(S...(u)..)$ donde u es 0 o una variable de tipo 1.

[5] Un axioma de la teoría de conjuntos.

sí misma):

$$\text{(P11)} \quad (\forall x_1 : (x_2(x_1) \iff y_2(x_1))) \Rightarrow x_2 = y_2$$

Este axioma nos indica que una clase está completamente determinada por sus elementos.

Una fórmula c se llama *consecuencia inmediata* de a y b si a es la fórmula $(\neg(b)) \vee (c)$, y una *consecuencia inmediata* de [únicamente] a si c es la fórmula $\forall v : a$, donde v es cualquier variable. La clase de [*teoremas*] es definida como la menor clase de fórmulas que contiene a los axiomas y que es cerrada bajo la operación "consecuencia inmediata".

Una fórmula de **Peano** es *verdadera* si es cierta interpretándola como una afirmación lógica sobre los números naturales.

Símbolo	Notación original (Gödel)	Descripción
$\neg$	$\sim$	Negación
$\vee$	$\vee$	Disyunción
$\forall$	Π	Cuantificador universal
S	f	Sucesor
$($	$($	Paréntesis de apertura
$)$	$)$	Paréntesis de cierre
$\exists$	(Ex)	Cuantificador existencial
$:$	$.$	Tal que
$\Rightarrow$	$\supset$	Implicación
$\iff$	$\equiv$	Coimplicación
$=$	$=$	Igualdad

Tabla C.1: Correspondencia entre los símbolos usados en este trabajo y los símbolos usados originalmente por Gödel

Índice alfabético

Bibliografía

[1] Aaronson, S. (2017). Rosser's Theorem via Turing machines. https://scottaaronson.blog/?p=710

[2] Aiken, H. (1956). Elektronische Rechenmaschinen und Informationsverarbeitung [Máquinas electrónicas de cálculo y procesamiento de información] [Traducido en (31)]. *Nachrichtentechnische Fachberichte*, 2, 32-34.

[3] Arora, S. & Barak, B. (2009). *Computational Complexity: A Modern Approach* (Illustrated). Cambridge University Press.

[4] Barrow, J. (2011). *The Artful Universe Expanded*. OUP Oxford.

[5] Bonola, R. (1955). *Non-Euclidean Geometry*. Courier Corporation.

[6] Carnielli, W. & Marcos, J. (2001). Ex Contradictione Non Sequitur Quodlibet. http://sqig.math.ist.utl.pt/pub/MarcosJ/01-CM-ECNSQL.pdf

[7] Church, A. (1937). A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. Proceedings of the London Mathematical Society, 2 s. vol. 42 (1936–1937), pp. 230–265. *Journal of Symbolic Logic*, *2*(1), 42-43. https://doi.org/10.1017/s002248120003958x

[8] Church, A. (1936). An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics*, *58*(2), 345. https://doi.org/10.2307/2371045

[9] Cook, S. A. (1971). The complexity of theorem-proving procedures. *Proceedings of the third annual ACM symposium on Theory of computing - STOC '71*. https://doi.org/10.1145/800157.805047

[10] De Mol, L. (2021). Turing Machines. En E. N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy* (Winter 2021). Metaphysics Research Lab, Stanford University.

[11] Evans, B. & Flanagan, D. (2018). *Java in a Nutshell*. O'Reilly Media.

[12] Evans, C. & Robertson, A. (1968). *Cybernetics*. University Park Press. https://books.google.ie/books?id=zpIhAQAAIAAJ

[13] Fuegi, J. & Francis, J. (2003). Lovelace Babbage and the creation of the 1843 'notes'. *IEEE Annals of the History of Computing*, *25*(4), 16-26. https://doi.org/10.1109/MAHC.2003.1253887

[14] Gödel, K. (1992). *On Formally Undecidable Propositions of Principia Mathematica and Related Systems*. Dover Publications.

[15] Gödel, K. (1931). *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme* [Translated in [Gö92]].

[16] Haines, M. D. (1993). Distributed runtime support for task and data management. *Technical Report CS-93-110, Department of Computer Science of Colorado State University*.

[17] Hamming, R. W. (1986). *Numerical Methods for Scientists and Engineers*. Courier Corporation.

[18] Hartmanis, J. & Stearns, R. E. (1965). On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, *117*(0), 285-306. https://doi.org/10.1090/s0002-9947-1965-0170805-7

[19] Henderson, T. (2013). Peano Arithmetic.

[20] Hilbert, D. (1902). Mathematical problems. *Bulletin of the American Mathematical Society*, *8*(10), 437-479. https://doi.org/10.1090/s0002-9904-1902-00923-3

[21] Hofstadter, D. (1999). *Gödel, Escher, Bach: An Eternal Golden Braid* (20th Anniversary ed.). Basic Books.

[22] Hopper, G. M. (1952). The education of a computer. *Proceedings of the 1952 ACM national meeting (Pittsburgh) on - ACM '52*. https://doi.org/10.1145/609784.609818

[23] Howden, W. (1978). Theoretical and Empirical Studies of Program Testing. *IEEE Transactions on Software Engineering*, *SE-4*(4), 293-298. https://doi.org/10.1109/TSE.1978.231514

[24] Katz, V. J. (2009). *A History of Mathematics*. Addison-Wesley Longman.

[25] Kaye, P., Laflamme, R. & Mosca, M. (2007). *An Introduction to Quantum Computing*. Oxford University Press.

[26] Kleene, S. C. (1936). General recursive functions of natural numbers. *Mathematische Annalen*, *112*(1), 727-742. https://doi.org/10.1007/bf01565439

[27] Kleene, S. C. (1971). *Introduction to Metamathematics*.

[28] Linz, P. (2011). *An Introduction to Formal Languages and Automata, 5th Edition* (5.ª ed.). Jones Bartlett Learning.

[29] Lovelave, A. (1843). *Sketch of the analytical engine invented by Charles Babbage*.

[30] Lutz, M. (2013). *Learning Python*. .ºReilly Media, Inc.".

[31] MacCormick, J. (2018). *What Can Be Computed?: A Practical Guide to the Theory of Computation*. Princeton University Press.

[32] Martelli, A. (2006). *Python in a Nutshell*. .ºReilly Media, Inc.".

[33] Moore, C. & Mertens, S. (2011). *The Nature of Computation* (1.ª ed.). Oxford University Press.

[34] Nisan, N. & Schocken, S. (2008). *The Elements of Computing Systems*. MIT Press.

[35] Nisan, N. & Schocken, S. (2005). *The Elements of Computing Systems: Building a Modern Computer from First Principles*. The MIT Press.

[36] Petzold, C. (2008). *The Annotated Turing: A Guided Tour Through Alan Turing's Historic Paper on Computability and the Turing Machine* (1.ª ed.). Wiley.

[37] Rogers, H. (1957). *Theory of Recursive Functions and Effective Computability*. Massachusetts Institute of Technology. https://books.google.es/books?id=v-vuAAAAMAAJ

[38] Rosser, B. (1936). Extensions of some theorems of Gödel and Church. *The Journal of Symbolic Logic*, *1*(3), 87-91. https://doi.org/10.2307/2269028

[39] Searle, J. R. (1980). Minds, brains, and programs. *Behavioral and Brain Sciences*, *3*(3), 417-424. https://doi.org/10.1017/s0140525x00005756

[40] Sipser, M. (2012). *Introduction to the Theory of Computation* (3.ª ed.). Cengage Learning.

[41] Translation of "On formally undecidable propositions of Principia Mathematica and related systems I". (2000). https://hirzels.com/martin/papers/canonoo-goedel.pdf

[42] Tseitin, G. S. (1983). On the Complexity of Derivation in Propositional Calculus. *Automation of Reasoning*, 466-483. https://doi.org/10.1007/978-3-642-81955-1{_}28

[43] Turing, A. M. (1950). Computing Machinery and Intelligence. *Mind*, *59*(236), 433-460. https://doi.org/10.1093/mind/lix.236.433

[44] Turing, A. M. (1937). On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, *s2*(1), 230-265. https://doi.org/10.1112/plms/s2-42.1.230

[45] Vinnikov, V. (1999). We shall know: Hilbert's apology. *The Mathematical Intelligencer*, *21*(1), 42-46. https://doi.org/10.1007/bf03024831

[46] Whitehead, A. N. & Russell, B. (1927). *Principia Mathematica*. Cambridge University Press.

[47] Wiles, A. D. (1995). Modular Elliptic Curves and Fermat's Last Theorem. *Annals of Mathematics*, *141*(3), 443. https://doi.org/10.2307/2118559

[48] Wolfram, S. (2002). *A New Kind of Science* (1.ª ed.). Wolfram Media.