TÉCNICAS BÁSICAS DE COMPUTABILIDAD

Arantza Irastorza, Ana Sánchez, Jesús Ibáñez

UPV/EHU/LSI/TR 3-2003

Indice

1. Introducción

La Teoría de la Computabilidad es una disciplina encuadrada en la Informática Teórica que tiene como objetivo establecer los límites lógicos que presentan los sistemas informáticos a la hora de resolver problemas mediante el diseño de algoritmos. Frente a las disciplinas y técnicas que día a día amplían el campo de aplicabilidad práctica de los computadores, esta teoría establece una serie de barreras insalvables por ninguna tecnología digital de procesamiento de la información, a modo de Leyes fundamentales que gobiernan las propias condiciones de existencia de la Informática.

Los métodos propios de la Teoría de la Computabilidad pueden ser extraordinariamente complejos, y sobre todo resultar relativamente extraños para una formación eminentemente orientada a la vertiente tecnológica. Sus resultados más avanzados son de difícil comprensión incluso para informáticos experimentados con una buena formación teórica. Sin embargo, existe un núcleo de resultados fundamentales que son abordables mediante técnicas más asequibles, y que tienen la virtud de reflejar razonablemente el concepto central de indecidibilidad computacional, y son deducibles con poco esfuerzo gracias a algunas técnicas que, si bien no son privativas de la Teoría de la Computabilidad, han resultado enormemente fructíferas en esta área.

Este informe incluye una descripción de los conceptos y técnicas que configuran ese núcleo básico de la Teoría. Su propósito es dar cuenta de la primera batería de resultados relacionados con la incomputabilidad de algunos problemas conocidos y relevantes en Informática. Los resultados se presentarán utilizando como estándar de programación los programas-while tal como los describimos en otro informe previo [IIS 96], incluyéndose una explicación detallada y sistemática de la técnica de Diagonalización.

Aunque ha sido escrito como material de apoyo docente para las y los estudiantes de la asignatura Modelos Abstractos de Cómputo II del Plan de Estudios de Ingeniería en Informática de la Facultad de Informática de la UPV/EHU, la pretensión de las autoras es proporcionar una descripción comprensible de los resultados y técnicas arriba apuntados. Aunque este informe es la continuación del citado en el párrafo anterior, quien quiera prescindir de la lectura de este último encontrará las definiciones y conceptos fundamentales en el capítulo segundo, así como una descripción más detallada del lenguaje de los programas-while junto con otras informaciones contextualizadoras en los apéndices.

Sin embargo, quien haya desembocado en el presente trabajo a partir del informe [IIS 96] encontrará muchas definiciones conocidas en el capítulo 2, aunque figuran algunas que no se incluyeron en él, y es probable que juzgue superfluo el contenido de los apéndices. Aquí asumiremos los resultados ya expuestos en aquel documento y mantendremos todas las convenciones allí establecidas excepto en lo que concierne al encabezamiento de los macroprogramas. Si en la sección 3.5 del citado informe introducíamos la noción de importación de funciones previamente implementadas (mediante la notación **package** COMPUTABLES) como medio para imbuir del necesario sentido jerárquico al proceso incremental de demostraciones de computabilidad, hemos llegado a un punto en el que ya no resulta necesario y sí engorroso realizar esta precisión de manera sistemática.

En el capítulo 3 abordamos el tema central del presente trabajo desde una perspectiva aún informal. Es importante que el concepto de incomputabilidad se desligue de interpretaciones contingentes basadas en el estado de conocimiento sobre los sistemas informáticos. Una forma adecuada de hacerlo es distinguir entre funciones no computables y computables de manera no efectiva. Al mismo tiempo ello nos conduce al establecimiento de la premisa fundamental sobre la que se basa todo el análisis de la Teoría de la Computabilidad: que las propiedades de incomputabilidad son independientes del sistema de programación elegido.

El capítulo 5, en el que se expone con cierto detalle la técnica de diagonalización, fundamental para demostrar la incomputabilidad de funciones e indecidibilidad de predicados, viene precedido por otro que recoge una serie de resultados y técnicas extraordinariamente útiles para nuestros propósitos. Estos resultados acreditan hechos tan relevantes como la necesaria existencia de ordenadores de propósito general o la equivalencia entre los modelos de procesado en paralelo y los secuenciales, pero sobre todo proporcionan importantes herramientas que luego se utilizarán para demostrar tanto la computabilidad como la incomputabilidad de muchas funciones relevantes.

Una vez estudiada la diagonalización y sus aplicaciones se ha incluido un último capítulo centrado en problemas de decisión y que aporta un concepto, la semidecidibilidad, que resulta esclarecedor para tener una visión un poco más posibilista sobre la incomputabilidad de algunos problemas.

2. Nociones básicas y notación

ALFABETO Y PALABRA: La noción de procesamiento de la información va ligada a la idea de transformación controlada de objetos que resultan de la combinación de símbolos. Por ello llamaremos ***alfabeto*** a cualquier conjunto finito $\sum$ de símbolos. Dado cualquier alfabeto $\sum$, definimos $\sum^*$ como el conjunto de las ***palabras*** o ***cadenas*** sobre el mismo. Utilizaremos las nociones clásicas de palabra vacía (ε), concatenación e inversión de palabras. Estas y otras funciones de manipulación de símbolos vienen listadas en el apéndice C, sección ***a***.

Las palabras son los elementos que nos permiten representar objetos del dominio, y por tanto son la base de la información. Así, la misión de los programas consistirá en manipular unas palabras (datos) para producir otras (resultados).

FUNCIONES: Para describir el comportamiento de los programas en términos de entrada/salida utilizaremos ***funciones parciales*** entre palabras. Estas pueden estar indefinidas para algunas de sus posibles entradas. Si la función $\psi : \sum^* \longrightarrow \sum^*$ aplicada sobre una palabra **x** de entrada ***converge*** (tiene imagen) lo indicamos mediante $\psi(\mathbf{x})\downarrow$. Si, por el contrario, la función está indefinida en ese punto escribimos $\psi(\mathbf{x})\uparrow$ y decimos que ***diverge***.

También utilizaremos funciones con más de un argumento (aunque siempre con un único resultado), de la forma $\psi : \sum^{*\mathbf{k}} \longrightarrow \sum^*$. En este caso indicaremos que $\psi(\mathbf{x_1},...,\mathbf{x_k})\downarrow$ ó que $\psi(\mathbf{x_1},...,\mathbf{x_k})\uparrow$.

PROGRAMAS: Un programa es la especificación no ambigua de un proceso de manipulación de símbolos que permite transformar una o varias cadenas de entrada (datos) para obtener otra de salida (resultado). Un programa debe estar construido de acuerdo a unas normas sintácticas que definen el lenguaje de programación. En nuestro caso el lenguaje utilizado es el de los ***programas-while***, cuya sintaxis precisa se describe en el Apéndice A, aunque también puede consultarse una descripción más detallada [IIS 96].

El comportamiento de un programa **P** estará descrito por una función parcial que denotaremos $\varphi_{\mathbf{P}} : \sum^* \longrightarrow \sum^*$, y que describe la relación que existe entre la cadena de entrada que recibe **P** y la que produce como resultado. Como para devolver un resultado el programa debe terminar su ejecución, cuando el comportamiento de **P** ante una entrada **x** consista en ciclar indefinidamente, ello quedará reflejado en el hecho de que $\varphi_{\mathbf{P}}(\mathbf{x})\uparrow$, puesto que el resultado del programa es indefinido. Si $\varphi_{\mathbf{P}}$ es ***total*** (que está definida para todos los posibles argumentos) entonces el programa **P** termina en cualquier circunstancia. En el otro extremo está

está el caso en el que φ_P es la función vacía ($\perp\!\!\!\perp$), que está siempre indefinida e indica por tanto que **P** cicla ante cualquier entrada.

El concepto se extiende de forma natural cuando el programa **P** recibe **k** datos de entrada en lugar de uno solo. Entonces la función que describe su comportamiento es $\varphi_P^k : \sum^{*k} \longrightarrow \sum^*$.

COMPUTABILIDAD: Una función $\psi : \sum^{*k} \longrightarrow \sum^*$ es ***while-computable*** si existe un programa-while **P** que la computa ($\varphi_P^k \cong \psi$), es decir, que obtiene sistemáticamente los resultados de la función para cualesquiera valores de sus argumentos. Si hablamos de ***computabilidad*** a secas nos referimos a la posibilidad de computar ψ utilizando cualquier lenguaje o sistema de programación imaginable, y no necesariamente el de los programas-while. En la sección 3.2 estudiaremos la relación que existe entre while-computabilidad y computabilidad.

El objetivo de este informe es el establecimiento de mecanismos de distinción entre funciones computables e incomputables.

PREDICADOS: Hay una clase especial de funciones que tienen una importancia especial en Teoría de la Computabilidad: los predicados o funciones booleanas totales (véase el capítulo 6). De la misma manera que podemos clasificar las funciones en while-computables o incomputables de acuerdo con la posibilidad de que un programa-while calcule sus valores, también podemos clasificar los predicados de acuerdo con la posibilidad de que un programa distinga sus casos ciertos de los falsos. Decimos que un predicado $\mathbf{S} : \sum^* \rightarrow \mathbb{B}$ (donde $\mathbb{B}$ es el conjunto de los valores booleanos {*true,false*}) es ***while-decidible*** si existe un programa-while **P** que produce un resultado $\mathbf{r_1}$ para todos los valores que cumplen $\mathbf{S}(\mathbf{x})$ y otro resultado distinto $\mathbf{r_2}$ para todos los que se verifica $\neg\mathbf{S}(\mathbf{x})$. Como su propio nombre indica, un predicado es while-decidible si es posible decidir su veracidad mediante un programa-while.

Esta noción se extiende de manera natural a los predicados **k**-arios de la forma $\mathbf{S} : \sum^{*k} \rightarrow \mathbb{B}$.

MACROPROGRAMAS: Para facilitar la tarea de demostrar la while-computabilidad de funciones complejas se define un lenguaje de ***macros*** sobre los programas-while que permite abreviar la escritura de los programas y entender mejor su significado. Llamamos macroprogramas a las abreviaturas resultantes, que permiten utilizar funciones cuya computabilidad ya ha sido demostrada como si fueran parte del lenguaje. Las principales clases de macros cuyo uso se ha justificado demostrando que realmente equivalen (pueden ***expandirse***) a estructuras propias del lenguaje de los programas-while se describen en el Apéndice B.

Dado que en los macroprogramas se permite hacer llamadas a funciones while-computables y predicados while-decidibles cualesquiera como si pasaran a incorporarse al lenguaje de programación, en el Apéndice C se da una lista de las funciones y predicados cuya while-computabilidad fue demostrada en [IIS 96] y que serán utilizadas en los macroprogramas que construyamos de aquí en adelante. Gracias a todas estas convenciones dispondremos de un lenguaje de partida mucho más rico y cómodo de manipular que el de los programas-while.

TIPOS DE DATOS: Aunque los programas-while sólo están definidos para operar sobre cadenas de caracteres (palabras), es posible definir sencillas correspondencias entre otros conjuntos de datos y los elementos de Σ^*, de forma que cada cadena represente un elemento del nuevo conjunto y que la computación sobre palabras sea congruente con la representación de los elementos del nuevo tipo. De esta forma conseguimos la ***implementación*** de algunos tipos de datos sencillos y útiles, como son los ***booleanos*** $\mathbb{B}$ o los números ***naturales*** $\mathbb{N}$. De manera análoga se implementan tipos estructurados como las ***pilas*** $\mathbb{P}$ y los ***vectores dinámicos*** $\mathbb{V}$. Finalmente se demuestra que se pueden implementar los propios ***programas-while*** $\mathbb{W}$ como tipo de datos. Todo esto nos permite trabajar con funciones computables en dominios distintos de Σ^*, así como utilizar los elementos y las operaciones de dichos tipos en nuestros macroprogramas.

En las secciones ***c***, ***d*** y ***e*** del Apéndice B también se listan las funciones y predicados definidas sobre estos tipos de datos cuya while-computabilidad fue demostrada en [IIS 96] y que serán utilizadas en los macroprogramas que construyamos de aquí en adelante. De nuevo obtenemos como ventaja un enriquecimiento del lenguaje de programación con objetos propios (constantes, funciones y predicados) de tipos de datos muy útiles para nuestros propósitos.

ENUMERACIÓN DE PALABRAS: Salvo en el caso de los booleanos (por razones obvias) las implementaciones pueden definirse siempre biyectivas. Esto quiere decir que, por ejemplo, existe una correspondencia biunívoca entre los números naturales y las palabras de Σ^* que sirven para representarlos, y este hecho se manifiesta independientemente del alfabeto Σ elegido. Por ello, podemos ***enumerar*** cualquier conjunto Σ^* de palabras de la forma $\Sigma^* = \{\mathbf{w_0}, \mathbf{w_1}, \mathbf{w_2}, \mathbf{w_3}, ...\}$, donde $\mathbf{w_i}$ es la palabra que sirve para representar el número **i**. Establecido un orden podemos programar la función que obtiene la palabra siguiente a la recibida como dato de entrada (función *sig*) o la anterior (función *ant*).

En nuestro caso el orden elegido es el que ordena las palabras por longitud, y a igual longitud por orden lexicográfico. Así, para el alfabeto {**a**, **b**, **c**} el orden inducido en las palabras sería: ε - **a** - **b** - **c** - **aa** - **ab** - **ac** - **ba** - **bb** - **bc** - **ca** - **cb** - **cc** - **aaa** - **aab** - **aac** ..., de forma que $\mathbf{w_0} = \varepsilon$ y $\mathbf{w_9}$ = **bc**, sig(**abc**) = **aca** y ant(**aaaa**) = **ccc**.

FUNCIONES DE CÓDIGO: Podemos destacar también otra colección de funciones computables llamadas funciones de código (funciones cod^k), que permiten comprimir o agrupar series de palabras en una sola, así como restituir la serie original a partir de su codificación, también de manera computable (funciones *decod*). Para el caso **k**=2 la forma de asociar códigos a los pares de palabras viene ilustrada en la tabla de la figura 2.1.

	$\mathbf{w_0}$	$\mathbf{w_1}$	$\mathbf{w_2}$	$\mathbf{w_3}$	$\mathbf{w_4}$	**...**
$\mathbf{w_0}$	w_0	w_2	w_5	w_9	w_{14}	
$\mathbf{w_1}$	w_1	w_4	w_8	w_{13}	w_{19}	
$\mathbf{w_2}$	w_3	w_7	w_{12}	w_{18}	w_{25}	
$\mathbf{w_3}$	w_6	w_{11}	w_{17}	w_{24}	w_{32}	
$\mathbf{w_4}$	w_{10}	w_{16}	w_{23}	w_{31}	w_{40}	
...						...

Figura 2.1: Tabla de asociación que permite determinar el código de un par de palabras. Así, $cod^2(\mathbf{w_0}, \mathbf{w_3}) = \mathbf{w_9}$ (para el ejemplo concreto de $\sum = \{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$ podríamos precisar que $cod^2(\varepsilon, \mathbf{c}) = \mathbf{bc}$). Las funciones que recuperan los elementos de un par a partir de su código son $decod_{2,1}(\mathbf{w_9}) = \mathbf{w_0}$ y $decod_{2,2}(\mathbf{w_9}) = \mathbf{w_3}$.

En la sección ***b*** del Apéndice C se da una descripción más precisa y detallada de estas funciones.

ENUMERACIÓN DE PROGRAMAS Y FUNCIONES: La enumeración de las palabras puede hacerse transitiva a todo tipo de datos implementado, siendo muy destacable el caso de los programas-while, que pueden enumerarse de la forma $\mathbb{W} = \{\mathbf{P_0}, \mathbf{P_1}, \mathbf{P_2}, \mathbf{P_3}, ...\}$, donde $\mathbf{P_i}$ es el programa representado por la palabra $\mathbf{w_i}$. Se dice entonces de $\mathbf{w_i}$ (y, por extensión, también de **i**) que es el ***código*** del programa $\mathbf{P_i}$.

Dado que en esta enumeración están todos los programas posibles, también podemos enumerar en una lista la clase de todas las funciones computables de la forma $\{\varphi_0, \varphi_1, \varphi_2, \varphi_3, \ldots\}$, donde φ_i es la función calculada por el programa $\mathbf{P_i}$ representado por la palabra $\mathbf{w_i}$. Diremos que $\mathbf{w_i}$ (y, por extensión, también el número asociado **i**) es un ***índice*** de la función φ_i.

Por todo ello, y como afirmar que una función ψ es computable equivale a demostrar que existe un programa **P** que la computa, también equivale a aseverar que existe un índice para ella, es decir un valor **e** tal que $\psi \cong \varphi_e$. Pero debemos añadir que, en contraste con los programas, que tienen un código único, toda función computable tiene infinitos índices porque podemos encontrar infinitos

programas equivalentes. Esto se debe a que existen infinitas posibilidades para modificar el texto de un programa sin alterar su semántica.

Por último destacaremos un convenio notacional más: nos referiremos al ***dominio*** de la función φ_i como $\mathbf{W_i}$ y a su ***rango*** como $\mathbf{R_i}$[1]. Estos conjuntos tienen cierta relevancia a la hora de describir el comportamiento de los programas, ya que $\mathbf{W_i}$ representa el conjunto de datos que $\mathbf{P_i}$ acepta como válidos, mientras que $\mathbf{R_i}$ se refiere a los resultados concebibles para el mismo programa.

[1] En rigor, un mismo programa $\mathbf{P_i}$ puede ser utilizado de varias maneras dependiendo del número de datos que se le suministren. La enumeración que hemos descrito más arriba contiene solamente las funciones computables unarias. Si tenemos interés en funciones con un número arbitrario de argumentos, podemos definir para cada $\mathbf{k}>0$ la lista $\{\varphi_0^k, \varphi_1^k, \varphi_2^k, \varphi_3^k \ldots\}$, donde φ_i^k es la función computada por el programa $\mathbf{P_i}$ cuando se le suministran $\mathbf{k}$ datos de entrada. Lo dicho para un argumento se extiende de manera natural para $\mathbf{k}$ argumentos, y en particular la notación $\mathbf{W_i^k}$ y $\mathbf{R_i^k}$ para el dominio y el rango de φ_i^k respectivamente.

3. Computabilidad versus incomputabilidad

Hemos descrito lo que entendemos por función while-computable: aquélla que admite un programa-while que la compute. Esta noción se extiende de manera natural a cualquier problema que nos podamos plantear. Un problema será computable si se puede expresar en términos de función parcial en la que los datos y los resultados sean cadenas de caracteres (o pertenezcan a un tipo de datos cuya implementación haya sido probada), de forma que esta función se pueda computar. Por ejemplo, el problema de determinar el reparto de escaños en unas elecciones a partir del número de votos obtenido por cada candidatura según la ley D'Hont es while-computable. El problema se puede expresar mediante una función de la forma:

$$\text{reparto}: \mathbb{N} \times \mathbb{V} \longrightarrow \mathbb{V}$$

donde, si **n** es el número de escaños a repartir y **V** es el vector que contiene los votos de las distintas candidaturas, reparto(**n**,**V**) es otro vector con la misma dimensión que **V** y que contiene los escaños de las distintas candidaturas. Esta función está correctamente definida porque para un par (**n**,**V**) determinado sólo existe un reparto posible de escaños[2]. Sabemos además que es computable, pues hace tiempo que los informáticos lo han resuelto en otros entornos de programación (si nos quedara alguna duda sería muy sencillo comprobarlo directamente haciendo un pequeño macroprograma).

La cuestión ahora es, ¿qué querría decir que un problema no es while-computable? Y, una vez lo tengamos claro, ¿existen problemas incomputables? Y si los hay ¿cómo podemos encontrarlos y reconocerlos?

Decimos que una función es while-incomputable cuando no existe ningún programa-while que la computa. Y entendemos que un problema es incomputable cuando puede expresarse en forma de función entre cadenas de caracteres, pero dicha función resulta incomputable. Es decir, que si no conseguimos expresar un problema en forma de función (véase la nota al pie), o tenemos dificultades para expresar sus datos o resultados en forma de cadenas de símbolos, entonces es que el problema es difícil (o imposible) de especificar, pero no tiene sentido entrar a discutir si es computable o no. La while-computabilidad, por tanto, se plantea una vez

[2] En realidad, esto sólo será cierto si existe una norma precisa y determinista para deshacer los empates entre candidaturas que obtengan exactamente el mismo número de votos. Si el reglamento prevé algún tipo de sorteo, entonces *reparto* no será una función bien definida, ya que el resultado no dependerá exclusivamente de los datos. En ese caso el problema no puede expresarse funcionalmente y no tiene sentido plantear si es while-computable o no.

realizada la especificación funcional del problema, y es por eso que nos limitaremos a hablar en general de funciones (y no de problemas) computables e incomputables.

> *Existen numerosos problemas en la vida real cuya especificación es muy complicada. En Inteligencia Artificial son moneda corriente. Por ejemplo, si planteamos como problema "jugar bien al ajedrez", ¿cómo podemos poner esto en términos de entrada/salida para construir un programa que resuelva dicho problema? Por mucho que precisemos (por ejemplo, indicando que jugar bien al ajedrez significa obtener una puntuación ELO superior a 7.500) el problema seguirá sin admitir un planteamiento en forma de función. En este terreno no son aplicables ni el concepto de computabilidad ni el de incomputabilidad.*
> *Naturalmente, ello no significa que la Informática se desentienda de estos problemas, pero en lugar de abordar su resolución (cosa que no tiene sentido al no poderse hablar de "la solución" del problema), utiliza técnicas para su aproximación (heurísticas, probabilísticas, etc).*

Una vez aclarado el terreno sobre el que nos vamos a mover cabe preguntarse por la existencia o no de funciones while-incomputables. La cuestión es pertinente: la primera vez que un informático se encuentra con la definición de computabilidad es muy raro que se haya planteado previamente que pueda haber problemas imposibles para su programación. Su experiencia le indicará que hay algunos más difíciles que otros, pero normalmente tendrá la sensación de que, con herramientas lo suficientemente potentes, todo acaba siendo posible en el mundo de la computación. Sin embargo esto es falso: no solamente existen problemas incomputables, como veremos, sino que su número es abrumadoramente mayor que el de los computables.

Supongamos que un informático se enfrenta con el siguiente problema: dado un programa cualquiera, comprobar si se ajusta a la especificación bajo la cual ha sido construido, es decir, si hace lo que se espera de él para cualquier posible entrada. Este problema se puede definir funcionalmente si previamente definimos el tipo de datos $\mathbb{E}$ de las especificaciones y lo implementamos mediante cadenas de símbolos (por ejemplo, podemos usar especificaciones pre-post como las empleadas en el cálculo de Hoare para la verificación de programas). Entonces, el problema quedaría definido mediante la función (predicado en realidad):

$$\text{cumple\_especificación?}: \mathbb{W} \times \mathbb{E} \longrightarrow \mathbb{B}$$

Ahora bien, si intentamos construir un programa que compute esta función nos daremos cuenta en seguida de que no es una tarea sencilla. De hecho, si consultamos bibliografía veremos que nadie ha conseguido jamás dicho programa en ningún entorno de programación, por lo que podemos empezar a sospechar que quizá

tampoco sea while-computable. ¿Qué haremos entonces? Podemos utilizar argumentos del tipo "yo no sé hacerlo", "no he logrado contratar a nadie que lo haga", "es que en la Facultad no me han enseñado eso", o "no he encontrado en la *web* ninguna empresa de software que disponga de utilidades para resolverlo". Quizá nos hace falta media docena de ideas felices para encontrar el programa, quizá su solución está reservada a la tecnología de dentro de treinta años. Pero la incomputabilidad es algo mucho más serio: si la función *cumple_especificación?* no es while-computable, ello quiere decir que no merece la pena que nadie dedique jamás ni cinco minutos a intentar desarrollar el programa correspondiente, porque ello es sencillamente imposible. ¿Cómo podemos encontrar un argumento convincente para este hecho?, es decir, que ponga de acuerdo a todo el mundo sobre dicha imposibilidad. ¿Cómo podemos pasar de la impotencia del "no se ha conseguido todavía" a la fatalidad del "es imposible"? Solo en la lógica matemática se encuentra solución a este problema: demostrando que la existencia de dicho programa sería contradictoria con hechos que ya sabemos que sí son ciertos.

3.1 Computabilidad y computabilidad no efectiva

Antes de abordar el problema de demostrar la incomputabilidad de algunas funciones, conviene reflexionar un poco más en detalle sobre esa diferencia fundamental entre decir "no podemos" y decir "no se puede" que hemos apuntado al principio del capítulo. En principio, cuando tratamos de demostrar la computabilidad de una función hemos de dar con un programa que la calcula. Pero hay ocasiones en que es imposible encontrar el programa-while que demuestra que una función es while-computable, ¡y la función, a pesar de todo, es while-computable! ¿Es posible que hayamos dedicado numerosas páginas a describir los programas-while, para descubrir que el modelo que representan no es completo?, es decir, que no sirve para abarcar todas las funciones computables.

Antes de sucumbir en el nerviosismo y empezar a buscar otro modelo alternativo, conviene que insistamos en que la definición de función while-computable exige que "exista un programa que la compute". Y no es lo mismo *que exista* un programa que *que podamos encontrarlo*. Obviamente lo segundo implica lo primero, pero veremos que esa relación no es recíproca: hay ocasiones en que el programa ciertamente existe, y sin embargo no resulta factible saber cuál es. La cuestión es entonces: ¿cómo demonios sabemos que existe? Tomemos como ejemplo el siguiente predicado:

$$\mathbf{f(x)} = \begin{cases} \text{true} & \text{la barba de Fidel tiene exactamente 10.253 pelos} \\ \text{false} & \text{c.c.} \end{cases}$$

Puede parecer que el predicado a evaluar es absurdo, porque existen enormes dificultades para determinar su valor. Depende de si conocemos a Fidel, de si nos deja tocarle la barba, o de si tiene guardaespaldas. Suponiendo que tengamos acceso irrestricto a su barba, hay dificultades relacionadas con el momento exacto en el que le contemos los pelos: puede crecerle alguno nuevo, o caérsele, puede decidir afeitarse, puede morirse de repente y ser incinerado. Es razonable pensar que jamás sabremos cuántos pelos tiene la barba de Fidel (a no ser que la CIA tenga un informe secreto al respecto), y por tanto que no sabremos si son o no 10.253, y que, por tanto, nunca podremos computar el predicado **f**.

Pero, ¿necesitamos realmente saber el número? Pensemos en la estructura del programa que habremos de construir. ¿De qué depende el valor que ha de devolver dicho programa? Si prestamos más atención a la definición de la función vemos que la salida no guarda relación alguna con la entrada **x**, porque solo hay dos posibilidades en la evaluación del predicado: o la barba de Fidel tiene 10.253 pelos (y **f** produce el resultado *true* para todo **x**) o no los tiene (y **f** produce el resultado *false* para todo **x**). Es decir, que el predicado **f** es o siempre cierto o siempre falso (no hay más alternativas). Por lo tanto, hay dos programas candidatos a calcular la función **f**. O bien

```
X0 := true;
```

o bien

```
X0 := false;
```

Lo que sucede es que no podemos decidir cuál de los dos es exactamente el que computa el predicado, pero tenemos la absoluta seguridad de que uno de ellos lo hace. Ello demuestra que existe un programa que computa **f**, y que por tanto este es decidible. El argumento utilizado es parecido al que se exhibe en algunas novelas policíacas: se sabe con seguridad que el asesino es uno de los ocupantes de la casa, aunque no exactamente cuál; sin embargo, como se sabe quiénes son dichos ocupantes se puede deducir que el asesino es varón, o que no tiene como móvil el robo.

Análogamente, en nuestro caso la definición del predicado **f** no es lo suficientemente precisa como para que sepamos de qué función se trata exactamente, pero sí lo bastante como para determinar que tiene que ser computable.

El deducir que una función es constante no es el único método para inferir su computabilidad sin necesidad de escribir el programa. Sea por ejemplo el predicado:

$$\mathbf{g}(\mathbf{x}) = \begin{cases} \text{true} & \text{la barba de Fidel tiene exactamente } \mathbf{x} \text{ pelos} \\ \text{false} & \text{c.c.} \end{cases}$$

Obviamente no podemos decir ahora que **g** sea constante, ya que es verdadero en un caso (depende de **x**) y falso en los demás. ¿Estamos obligados ahora a saber cuántos pelos tiene Fidel en la barba? Si quisiéramos escribir el programa concreto que computa **g** ello sería obligatorio. Sin embargo, ese no es nuestro propósito. No olvidemos que no necesitamos poner el programa encima de la mesa: bastará con demostrar su existencia.

¿Cómo es ahora **g** y qué debería hacer un programa para computarla? Como hemos indicado, el programa ha de devolver *true* en un solo caso (cuando la entrada coincida con el número de pelos de la barba de Fidel) y *false* en todos los demás. Si supiéramos cuál es el número M de pelos en la barba de Fidel podríamos construir el macroprograma:

```
X0 := X1 = M;
```

Pero, de nuevo, el valor de M es irrelevante para probar la existencia del programa. Podemos hacer una lista con todos los programas que tienen esta forma, uno para cada posible valor de M:

```
X0 := X1 = 0;
```

o bien

```
X0 := X1 = 1;
```

o bien

```
X0 := X1 = 2;
```

…

o bien

```
X0 := X1 = 10.253;
```

…

Así como decíamos que para **f** teníamos dos programas (o sus equivalentes) sin poder decidir cuál era el bueno, para **g** tenemos infinitas elecciones, una por cada valor de **M**, y nos resulta imposible seleccionar una concreta. Pero tenemos la seguridad de que uno de los programas computa **g**, y que por lo tanto esta es computable.

Veamos un último ejemplo algo más rebuscado. Podemos definir el conjunto **A** = {**z**: puede existir un ser vivo cuyo genoma tiene exactamente longitud **z**}. Con un examen superficial ya se puede apreciar que esta definición encierra aspectos complicados.

Por una parte sólo conocemos las longitudes exactas de los genomas de unos pocos seres vivos. Por otra, existen numerosas especies desconocidas, y todos los días se extinguen otras, por no hablar de las que aún no ha producido la evolución, de las que aún no ha creado la ingeniería genética, o de las que simplemente "pueden existir" pero no existirán jamás. Teniendo todo esto en cuenta podemos definir el predicado:

$$\mathbf{h}(\mathbf{x}) = \begin{cases} \text{true} & \exists \mathbf{y}\ (\mathbf{x} + \mathbf{y}) \in \mathbf{A} \\ \text{false} & \text{c.c.} \end{cases}$$

Y preguntarnos ¿es o no computable? Veremos que no necesitamos tener conocimiento alguno de genética para demostrar que sí. Es evidente que **h**(**x**) será *true* cuando pueda existir un organismo cuyo genoma tenga longitud mayor o igual que **x**, por lo que cuando se verifica **h**(**x**) también son ciertos los valores de **h** para entradas menores que **x**.

En principio existe la posibilidad de que **A** sea infinito (es decir, que no exista un límite superior a la longitud de los genomas de organismos viables). En ese caso tendríamos que **h** sería la función constante *true*, porque para cualquier **x** habrá valores **z** mayores que **x** en el conjunto **A**.

Pero **A** podría ser finito: quizá existe una longitud máxima de genoma **L** que no puede ser sobrepasada por ningún ser vivo. En ese caso el predicado **h** sería cierto para los valores menores o iguales que **L**, y falso para el resto. Y no hay ninguna otra posibilidad (**A** sólo puede ser finito o infinito), por lo que uno de los siguientes programas debe computar necesariamente **h**:

```
X0 := true;
```

o bien

```
X0 := X1 <= 0;
```

o bien

```
X0 := X1 <= 1;
```

…

o bien

```
X0 := X1 <= 100.893.873.343.157.2245.234.388.253;
```

…

Aunque, como en los casos anteriores, no tengamos forma de decidir cuál (salvo que, quizá, las investigaciones en Biología Teórica nos lo aclaren en el futuro).

Hemos visto que dentro de las funciones while-computables existen algunas para las cuáles no es factible encontrar el programa correspondiente. Decimos entonces que no son ***computables de manera efectiva***, por contraposición a las que sí admiten la construcción de un programa concreto. Aunque las primeras tienen un interés práctico limitado, su existencia nos permite reflexionar más en profundidad sobre el concepto de incomputabilidad y sobre su carácter absoluto, que no depende de factores como la exactitud en la especificación del problema. Por el contrario, veremos que las funciones incomputables se especifican con claridad meridiana, no existiendo ninguna duda sobre su interpretación o significado.

3.2 La Tesis de Church-Turing

Otra cuestión importante es la de la adecuación del lenguaje de programación elegido: ¿Hasta qué punto forman los programas-while un lenguaje representativo? Hemos visto que ciertos problemas son while-computables, y pronto empezaremos a encontrar otros while-incomputables. Ahora bien, ¿qué sucedería si pasáramos a utilizar un lenguaje de programación diferente, por ejemplo uno más potente? Si aumentamos nuestro lenguaje con nuevas primitivas, ¿no podríamos quizá descubrir que se convierten en computables funciones que no lo eran para la versión simple del lenguaje?

Esta cuestión es absolutamente fundamental, ya que para nosotros los programas-while no tienen ningún interés intrínseco. Si descubriésemos que el concepto de while-computabilidad sólo es aplicable para este lenguaje, todos nuestros resultados carecerían de interés, ya que en la práctica no vamos a utilizar los programas-while. Por el contrario, aunque saber para cada problema en qué lenguajes concretos resulta programable puede tener alguna relevancia, la incomputabilidad tiene interés como noción absoluta: una función sólo debería llamarse incomputable si fuera imposible, por ningún método, lenguaje o sistema computacional, establecer un procedimiento para calcular sus valores. Lo que no tiene mucho sentido es limitarse a demostrar que un problema determinado no es computable en el sistema **S** y sí lo es en el **T**. Ello lo único que nos indicará es que el sistema **S** es de peor calidad (porque tiene menor capacidad expresiva) y que tal vez deberíamos abandonarlo.

Todo sistema de programación tiene asociada una clase de funciones computables mediante el mismo. Si llamamos $\mathcal{F}$ a la clase de todas las funciones entre cadenas de símbolos, cada sistema de programación **S** establece una línea que separa las funciones **S**-computables de las **S**-incomputables. Como la noción genérica de computabilidad no puede depender del sistema de programación

concreto, definiremos que una función es simplemente ***computable*** si existe algún sistema de programación **S** que la convierte en **S**-computable. Llamaremos $\mathcal{C}$ a la clase de todas las funciones computables, y su situación sería como la reflejada en el diagrama de la figura 3.1.

Si esta fuera la situación real tendríamos un grave problema: para demostrar que una función no es computable deberíamos probar, uno por uno, todos los sistemas de programación existentes para verificar que no pueden con ella. Afortunadamente el panorama esbozado en la figura 3.1 no se ajusta a la realidad, y los lenguajes de programación se acaban pareciendo entre sí mucho más de lo que se podría sospechar. Por ejemplo, toda la utilización que hemos hecho de las macros en [IIS 96] nos ha demostrado que, cada vez que hemos echado de menos alguna facilidad en el lenguaje de los programas-while (anidamiento de expresiones, más rutinas preprogramadas, más tipos de datos, estructuras más complejas, objetos de usuario,...) hemos podido probar que no era preciso añadirla al lenguaje, porque se podía simular con los elementos ya existentes en el mismo. Es decir, que todavía no hemos topado con ninguna característica de otro lenguaje que los programas-while tengan que envidiar.

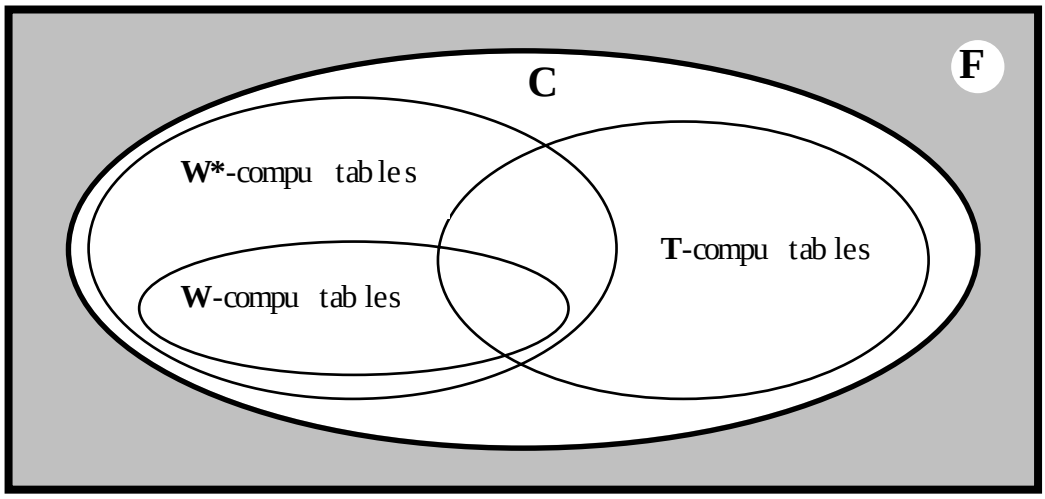

Figura 3.1: Si tuviéramos un sistema de programación **W** mediante el cual se alcanzara a computar la clase de las funciones **W**-computables, podría darse el caso de que consiguiéramos una versión mejorada del sistema **W*** gracias a la cual fuera posible ampliar dicha clase con funciones que para **W** resultaban incomputables. También podrían desarrollarse sistemas independientes, como el **T** de la figura, que permitiera computar funciones nuevas pero en el que no fuera posible programar algunas funciones **W**-computables. La clase de las funciones computables $\mathcal{C}$ (línea más gruesa) estaría constituida por la unión de todas las computadas por los distintos sistemas particulares, y las incomputables serían las que no admitieran solución por ***ninguno*** de los métodos posibles (zona rayada).

Sin embargo, por muy reveladora que sea nuestra propia experiencia a este respecto, mucho más lo es la contundente unanimidad producida en los casi 70 años de existencia de las Ciencias de la Computación: a pesar de que se han diseñado

modelos realmente muy alejados entre sí (algunos rayando en el esoterismo), jamás se ha conseguido un sistema de programación más perfecto que los programas-while. Lo que es aún más significativo, tampoco se han dado modelos menos perfectos: todos los sistemas **S** inventados hasta el momento han producido exactamente la misma clase de funciones **S**-computables[3]. La situación sería la reflejada en la figura 3.2.

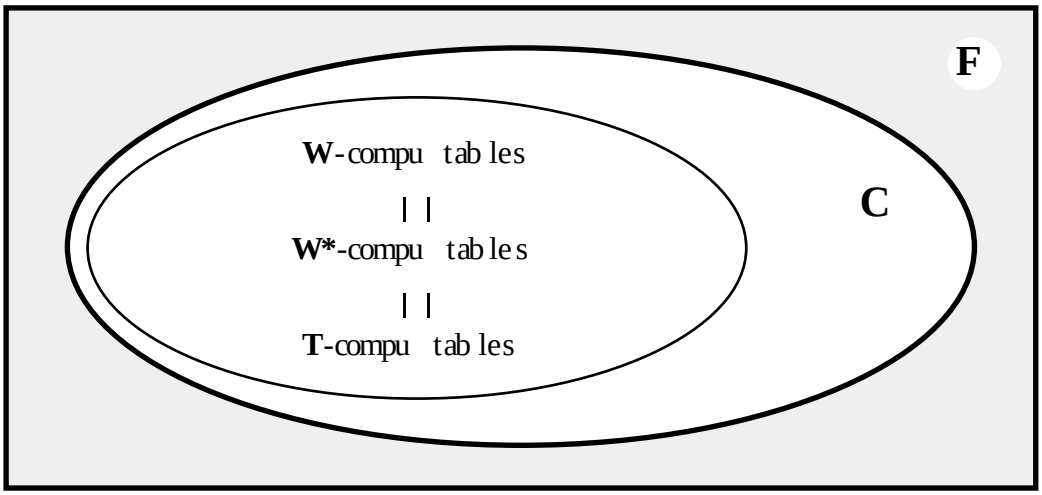

Figura 3.2: Todos los sistemas de programación descubiertos hasta la fecha con todas sus variantes computan las mismas funciones. Sin embargo, todavía podría quedar abierta la puerta a la aparición de nuevos sistemas de programación que permitieran computar algunas de las funciones que, hoy por hoy, nos resultan incomputables. Si este fuera el caso, habría una zona de funciones computables no alcanzada aún por los sistemas actuales.

En este escenario algunas cosas han mejorado y otras no. Por un lado, si demostramos que una función no es while-computable estamos probando al mismo tiempo que no se puede computar con ningún otro lenguaje conocido, lo que no es poco. Sin embargo no tendríamos forma alguna de comprobar que es incomputable "de verdad", ya que para ello deberíamos estudiar su relación con sistemas de programación que aún no se han inventado y quizá no lo sean nunca.

Sin embargo, la persistente coincidencia de todos los sistemas de programación en producir la misma noción de computabilidad llevó desde muy temprano a los informáticos teóricos a una conclusión tajante: la noción absoluta de computabilidad no va a ir más allá de los sistemas computacionales existentes, es decir, que no existe ningún sistema de programación mejor que los conocidos. Esa sería la razón de que ninguna innovación en Informática haya conseguido computar funciones que no

[3] Esta aseveración debe ser matizada. Si se mutila suficientemente un lenguaje de programación se puede perder la capacidad de computar ciertas funciones. Por ejemplo, si definimos los programas-for de manera idéntica a los programas-while pero sustituyendo los bucles generalizados por bucles controlados tipo for, el lenguaje pierde capacidad expresiva y se computan menos funciones que con el lenguaje original.

fueran ya computables por los primeros modelos abstractos de cómputo. Esta teoría es conocida como la ***Tesis de Church-Turing***, y se considera universalmente aceptada. Hace ya muchos años que en Ciencias de la Computación se trabaja asumiéndola sin reservas. Sus consecuencias se reflejan en la figura 3.3.

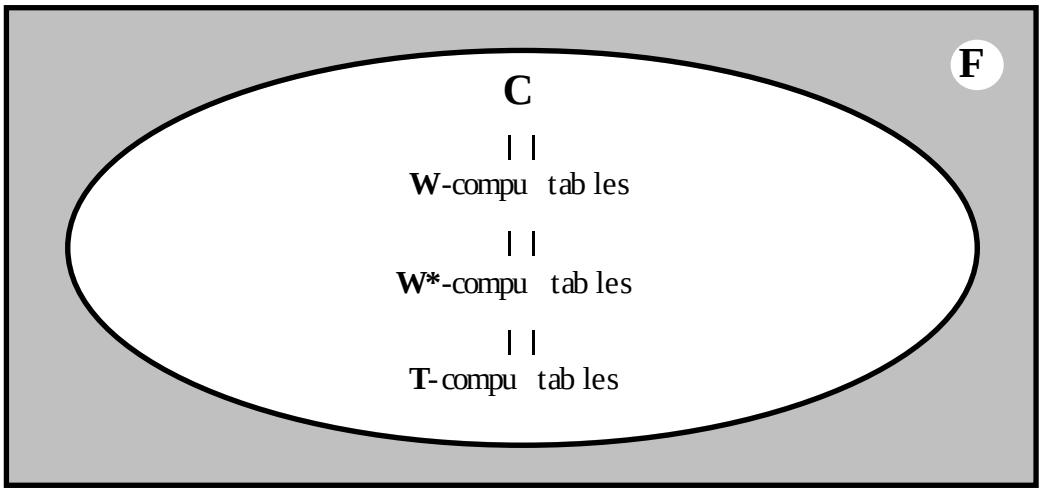

Figura 3.3: Consecuencia de la Tesis de Church-Turing: la noción de computabilidad obtenida para los distintos sistemas de programación es absoluta, y ningún sistema futuro la cambiará.

La consecuencia inmediata de la Tesis de Church-Turing es que las nociones de while-computabilidad y while-incomputabilidad son también válidas para todos los sistemas de programación habidos y por haber. Por tanto, cuando consigamos demostrar que no existe ningún programa-while que compute una cierta función ψ, estaremos probando algo de muchísimo más calado: que la Informática no tiene ni tendrá jamás medios para computar dicha función. Habremos encontrado una limitación con rango de Ley Universal.

Esta es la razón por la que a partir de ahora omitiremos el prefijo *while* ante las palabras computabilidad, incomputabilidad, decidibilidad o indecidibilidad. Aunque seguiremos demostrando que una función es computable encontrando el programa-while que la computa, o que es incomputable verificando que ningún programa-while lo hace, sabemos que las consecuencias de esas demostraciones van más allá del lenguaje utilizado (que no es más que un soporte) y afectan al estatuto de dicha función dentro de las Leyes de la Computación.

4. La Función Universal

Una vez establecidos algunos conceptos esenciales que aclaran el alcance de las nociones de computabilidad e incomputabilidad, estamos preparados para acometer la búsqueda de las primeras funciones incomputables. Sin embargo, como veremos en el siguiente capítulo, esa tarea no resulta trivial y para ponerla en práctica con éxito en algunos casos necesitaremos herramientas adicionales. Una de las características más curiosas de las demostraciones de incomputabilidad es que llevan aparejada la construcción de programas, muchas veces bastante complejos.

Otra razón para estudiar aún más funciones computables reside en la posibilidad de explorar la última frontera de lo computable. De alguna forma, las funciones y predicados que estudiaremos en este capítulo son las más complicadas que nos podemos plantear dentro de la computabilidad. Si intentamos ir un poco más allá nos encontraremos con las primeras incomputables (véase el capítulo 5).

Hay un tipo de datos cuyo interés va mucho más allá del de cualquier otro que hayamos conocido: el tipo $\mathbb{W}$. Mientras que en el caso de cadenas de caracteres, de valores lógicos o numéricos, o de estructuras agregadas como pilas y árboles, la inmensa mayoría de los problemas que se nos plantean de manera práctica son computables y están resueltos desde hace mucho tiempo (en algunos casos desde la civilización egipcia), los programas (y en particular, los programas-while) son objetos mucho más complejos en los que la incomputabilidad aparece casi a ras de superficie.

Tal como nos ha sucedido con otras implementaciones, podemos plantearnos la computabilidad de operaciones específicas sobre programas, y después utilizar dichas operaciones (mejor dicho, las que resulten computables entre ellas) como herramientas auxiliares para resolver otros problemas. Así mismo podemos definir conjuntos de programas y utilizarlos en nuestras definiciones:

$$VAC = \{ \mathbf{x}\in\mathbb{W}: \forall \mathbf{y}\in\sum^* \varphi_{\mathbf{x}}(\mathbf{y})\uparrow \} = \{ \mathbf{x}\in\mathbb{W}: \mathbf{W_x} = \varnothing \}$$

$$COF = \{ \mathbf{x}\in\mathbb{W}: \mathbf{W_x} \text{ es cofinito} \} = \{ \mathbf{x}\in\mathbb{W}: \overline{\mathbf{W_x}} \text{ es finito} \}$$

$$SUP = \{ \mathbf{x}\in\mathbb{W}: \varphi_{\mathbf{x}} \text{ es sobreyectiva} \} = \{ \mathbf{x}\in\mathbb{W}: \mathbf{R_x} = \sum^* \}$$

$$INJ = \{ \mathbf{x}\in\mathbb{W}: \varphi_{\mathbf{x}} \text{ es inyectiva} \} = \{ \mathbf{x}\in\mathbb{W}: \neg\exists \mathbf{y}\exists \mathbf{z}\ (\mathbf{y}\neq\mathbf{z} \wedge \varphi_{\mathbf{x}}(\mathbf{y})=\varphi_{\mathbf{x}}(\mathbf{z})) \}$$

$$K = \{ \mathbf{x}\in\mathbb{W}: \varphi_{\mathbf{x}}(\mathbf{x})\downarrow \}$$

SIC = { $\mathbf{x} \in \mathbb{W}$: en **x** no aparecen subprogramas de la forma if_then }

CER = { $\mathbf{x} \in \mathbb{W}$: ninguna ejecución de **x** utiliza el contenido de X0}

Tenemos que, por ejemplo, VAC es el conjunto de programas que ciclan ante cualquier entrada, COF es el de los que convergen casi para cualquier dato (es decir, exceptuando un número finito de comportamientos divergentes), SUP el de aquellos para los que cualquier resultado es admisible, e INJ el de los que no repiten resultados para datos distintos. Análogamente K es el conjunto de los programas que convergen sobre su propio código, SIC el de aquellos cuyo texto no incluye instrucciones condicionales, y CER contiene a los que no utilizan ni modifican la variable X0 (nunca pasan por una asignación que contenga X0 en su parte derecha). Como veremos, todos estos conjuntos de programas (menos uno) son indecidibles.

Por las razones aquí expuestas vamos a dedicar algún esfuerzo a desarrollar habilidades en torno a la programación con programas-while como datos, de lo que obtendremos una serie de funciones computables y predicados decidibles, pero que paradójicamente serán de gran ayuda para demostrar que otras funciones y predicados no lo son.

4.1 Operaciones sintácticas sobre los programas-while

Si nos planteamos utilizar los programas-while como datos corrientes, el nivel de manipulación más sencillo que podemos definir es el sintáctico: podemos estudiar el texto de un programa, examinar sus instrucciones o su estructura, y resolver cuestiones sencillas en cuanto a la misma. A este nivel nos encontraremos típicamente con funciones que toman como argumento un programa-while y devuelven un resultado simple (numérico o booleano).

Por ejemplo, podemos definir funciones que nos indiquen el número de variables o de instrucciones que tiene un programa, el número máximo de bucles anidados que contiene, si aparece alguna estructura con especiales características (por ejemplo, un bucle sospechoso en cuyo interior no se modifique la variable de control del while), etc. No es difícil ir demostrando la computabilidad de este tipo de funciones que únicamente trabajan sobre ***propiedades estáticas*** de los programas, es decir, aquellas que pueden ser determinadas en tiempo de compilación. Para verlo utilizaremos como ejemplo la función:

$$\text{ult\_variable}: \mathbb{W} \to \mathbb{N}$$

que determina cuál es el máximo índice de variable que aparece en un programa cualquiera. Para demostrar su computabilidad escribiremos un macroprograma

que, para calcular *ult_variable*(**x**), utiliza las operaciones propias de $\mathbb{W}$ para analizar el programa $\mathbf{P_x}$ en busca de sus instrucciones elementales (asignaciones y condiciones), que es donde aparecen referenciadas las variables. Este proceso de análisis es lineal en la mayor parte de los casos. Por ejemplo, si $\mathbf{P_x}$ es una asignación bastará con extraer los índices de sus variables y tomar el máximo. Si se trata de una instrucción condicional o iterativa examinaremos la variable afectada por la condición, extraeremos la instrucción interna y continuaremos el proceso con esta última. Sin embargo, si $\mathbf{P_x}$ es una composición secuencial del tipo **P Q** nos encontramos con que debemos inspeccionar **P** y después **Q** (o al revés). Pero como **P** puede a su vez ser una composición secuencial de la forma **R S**, su análisis puede exigir que dejemos aparcado **S** mientras analizamos **R**. Naturalmente, esta situación se puede complicar indefinidamente, con lo que hemos de prever la posibilidad de tener un número indeterminado de subprogramas de $\mathbf{P_x}$ en espera de ser analizados (ver figura 4.1).

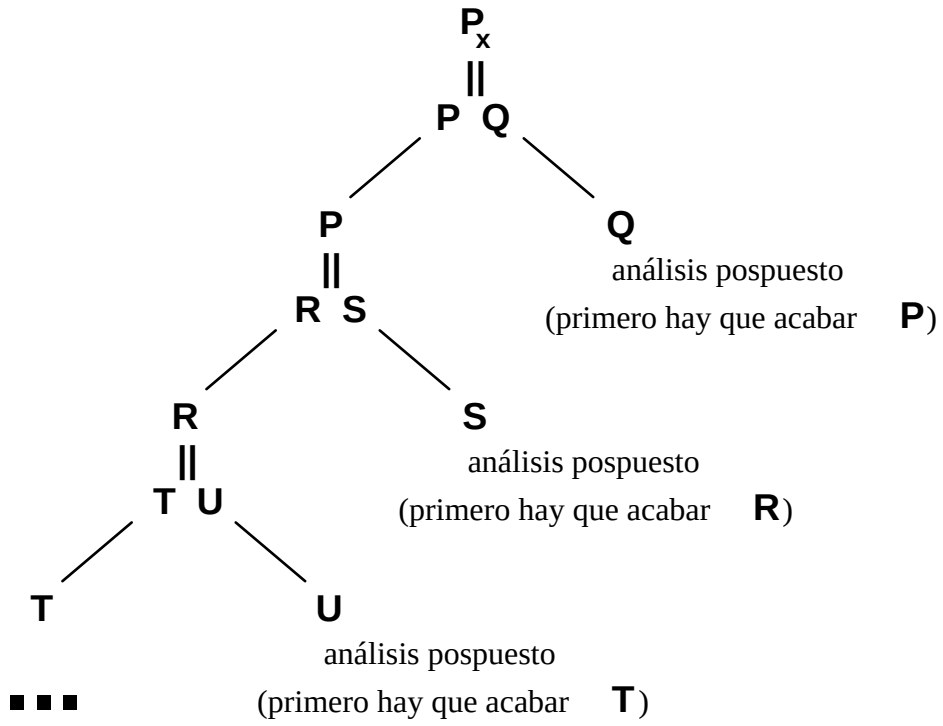

Figura 4.1: Al analizar un programa-while de estructura arborescente puede ser necesario mantener una lista arbitrariamente larga de subprogramas esperando análisis.

Resolveremos esto manteniendo en todo momento una pila con los programas-while que aún no han sido analizados. Esta pila contendrá inicialmente el programa $\mathbf{P_x}$ y será gestionada por un bucle que, en cada iteración, procesará el programa-while que se encuentre en su cima, reintroduciendo subprogramas del mismo en la pila para su posterior procesamiento cuando convenga. El vaciado de la pila indicará el fin de la tarea.

Para manipular los programas-while utilizaremos las operaciones del tipo que están descritas en la sección ***e*** del Apéndice 3.

```
PILA :=  <];
PILA :=  empilar (X1, PILA);
X0 :=  0;
-- En X0 mantendremos el máximo índice hallado hasta el momento
while not  P_vacía? (PILA)  loop
-- Se extrae de la pila el subprograma que toca analizar
      PROG :=  cima (PILA);
      PILA :=  desempilar (PILA);
-- Se procesa dicho subprograma según su clase
      if  asig_vacía? (PROG)  then
            if ind_var (PROG) > X0  then
                  X0 := ind_var (PROG);
            end if;
      elsif asig_cons? (PROG) or asig_cdr? (PROG)  then
            if ind_var (PROG) > X0  then
                  X0 := ind_var (PROG);
            end if;
            if ind_var_2 (PROG) > X0  then
                  X0 := ind_var_2 (PROG);
            end if;
      elsif condición? (PROG) or iteración? (PROG)  then
            if ind_var (PROG) > X0  then
                  X0 := ind_var (PROG);
            end if;
            PILA :=  empilar (inst_int (PROG), PILA);
      else  PILA := empilar  (inst_int_2 (PROG), PILA);
            PILA :=  empilar (inst_int (PROG), PILA);
      end if;
end loop;
```

4.2 El Teorema de Enumeración

De manera similar se tratan otras propiedades estáticas de los programas-while. Sin embargo, se obtienen resultados más interesantes cuando nos adentramos en otras que dependen del ***comportamiento dinámico*** de los programas, es decir, de situaciones verificables en tiempo de ejecución.

Así fijaremos nuestra atención en una función bastante especial que, como veremos, resultará computable: ***la función universal.*** Ésta toma como argumentos un programa-while $\mathbf{P_x}$ y un dato (palabra) **y**, relacionando este par con la salida que produciría el programa al suministrársele el dato indicado. Así, la función universal

será capaz de describir el resultado del comportamiento de cualquier cómputo, y de ahí su pretencioso nombre. Se denota con la letra Φ y se define como:

$$\Phi(\mathbf{x},\mathbf{y}) \cong \varphi_{\mathbf{x}}(\mathbf{y})$$

Debemos resaltar la importancia de esta función porque simula el comportamiento de *cualquier* función computable de un argumento, sin más que administrarle como datos el índice de la función a calcular y el dato de entrada para esta función. De alguna forma, el programa que computa la función universal es capaz de representar dignamente a todos los demás programas, encerrando dentro de sí todas las esencias de la computación.

TEOREMA (DE ENUMERACIÓN): La siguiente función $\Phi : \mathbb{W} \times \sum^* \longrightarrow \sum^*$ es computable:

$$\Phi(\mathbf{x},\mathbf{y}) \cong \begin{cases} \varphi_{\mathbf{x}}(\mathbf{y}) & \varphi_{\mathbf{x}}(\mathbf{y}) \downarrow \\ \\ \perp & \text{c.c.} \end{cases}$$

⌦ El programa **U** (***programa universal***) que hemos de escribir para probar que Φ es computable recibirá como entradas un código de programa cualquiera **x** y un dato **y**, y deberá obtener como resultado la salida (si la hay) que $\mathbf{P_x}$ produciría sobre **y** (ver fig. 4.2).

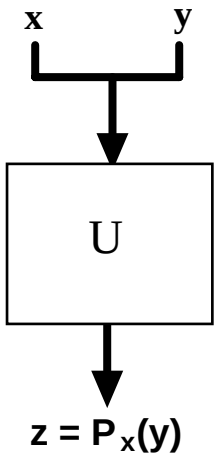

Figura 4.2: Esquema del funcionamiento del programa **U** que hemos de diseñar.

La forma natural de realizar esto es que **U** simule el comportamiento de $\mathbf{P_x}$ sobre **y** (es decir, su cómputo). Por tanto **U** será un *intérprete* de programas-while: deberá inicializar y mantener el vector de estado apropiado, siguiendo cuidadosamente las transformaciones que en el mismo operan las instrucciones del programa $\mathbf{P_x}$. Ahora bien, en el momento de escribir **U** no sabemos qué tamaño tendrá dicho vector de estado, puesto que no sabemos qué programa concreto $\mathbf{P_x}$

habremos de simular. De hecho, no podemos siquiera fijar un tamaño máximo **k** para dicho vector de estado, ya que siempre existirá un programa que desborde esas previsiones por necesitar más espacio. La solución es utilizar un vector dinámico VARS que contenga todo el vector de estado, de forma que VARS(**i**) sea en cada momento el contenido de la variable XI del programa $\mathbf{P_x}$. La inicialización de VARS se hará con el valor ε en todas sus posiciones, salvo VARS(1) que habrá de contener el dato **y**. Del mismo modo, una vez finalizada la simulación de la ejecución de $\mathbf{P_x}$ iremos a buscar su resultado a la posición 0-ésima del vector.

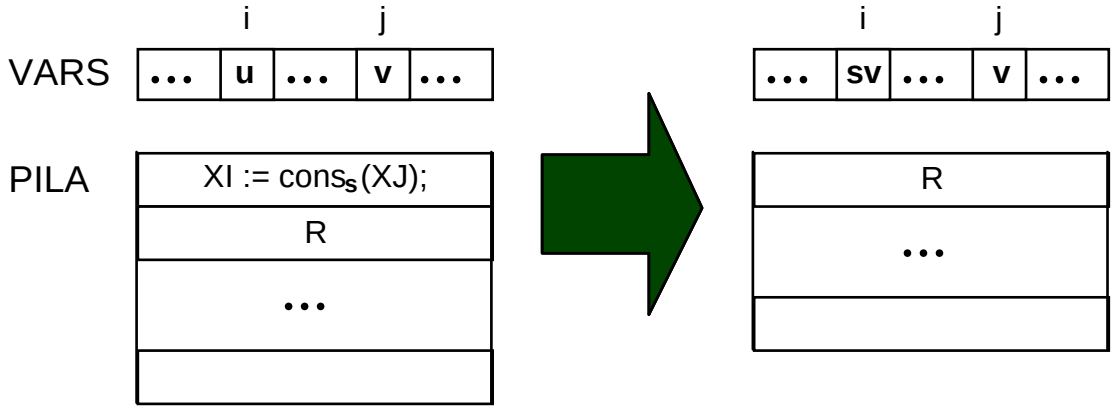

Figura 4.3: Ciclo de ejecución de **U** cuando el siguiente programa a ejecutar es una asignación.

Por otro lado, la propia ejecución de $\mathbf{P_x}$ nos plantea un problema similar al que teníamos con *ult_variable* y que describíamos en la figura 4.1. Lo solucionaremos de manera idéntica, pero en esta ocasión la pila contendrá los programas que esperan ser ejecutados en lugar de los que deben aún ser analizados. Únicamente habrá de cuidarse que los programas en espera se introduzcan en la pila de forma que luego se produzca el orden de ejecución correcto. Inicialmente PILA contendrá sólo el programa $\mathbf{P_x}$, y el ciclo básico de ejecución consistirá en sacar un programa de PILA y tratar de ejecutarlo. Si se trata de una asignación, ello se reflejará en un cambio en el estado de cómputo VARS (figura 4.3).

Si se trata de una composición secuencial nos contentaremos con dividir el programa en sus dos componentes (más sencillos) y meterlos en la pila (figura 4.4). Si es una composición condicional, evaluaremos la condición en el estado de cómputo VARS y, si procede ejecutar la instrucción interna al **if**, meteremos esta en la pila (figura 4.5) para que se ejecute en la siguiente iteración. Por último, si se trata de un programa iterativo también evaluaremos la condición, pero en este caso, si procede entrar en el bucle meteremos en la pila tanto la instrucción interna a este como una copia del bucle completo, para que vuelva a ser reconsiderado una vez terminada la iteración en curso (figura 4.6).

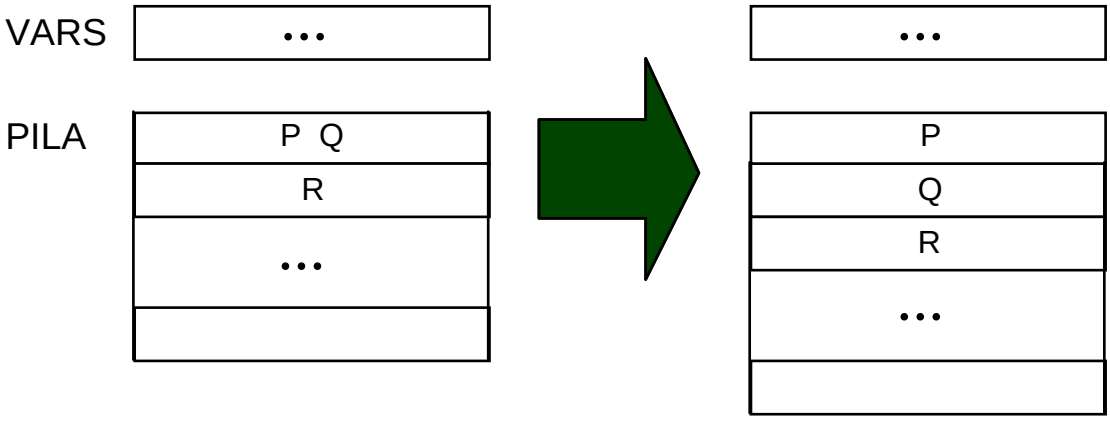

Figura 4.4: Ciclo de ejecución de **U** cuando el siguiente programa a ejecutar es una composición secuencial.

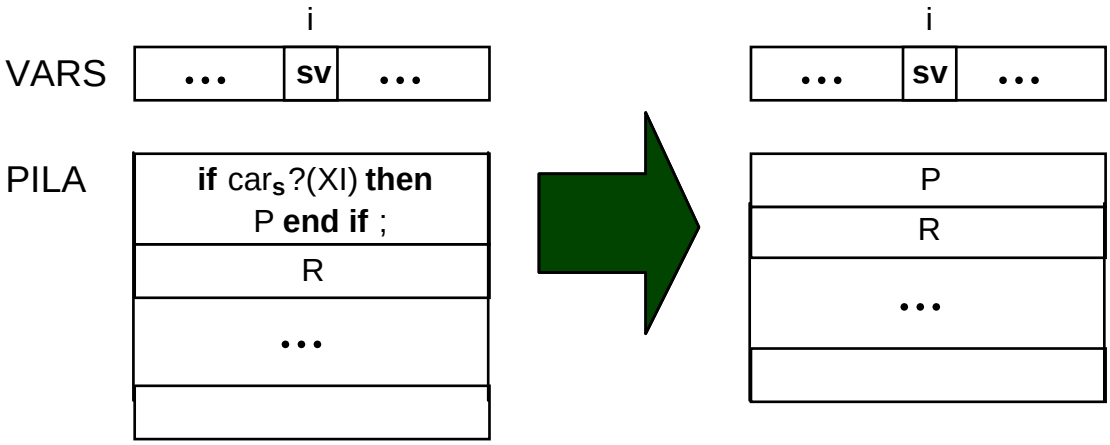

Figura 4.5: Ciclo de ejecución de **U** cuando el siguiente programa a ejecutar es una composición condicional cuya condición se cumple.

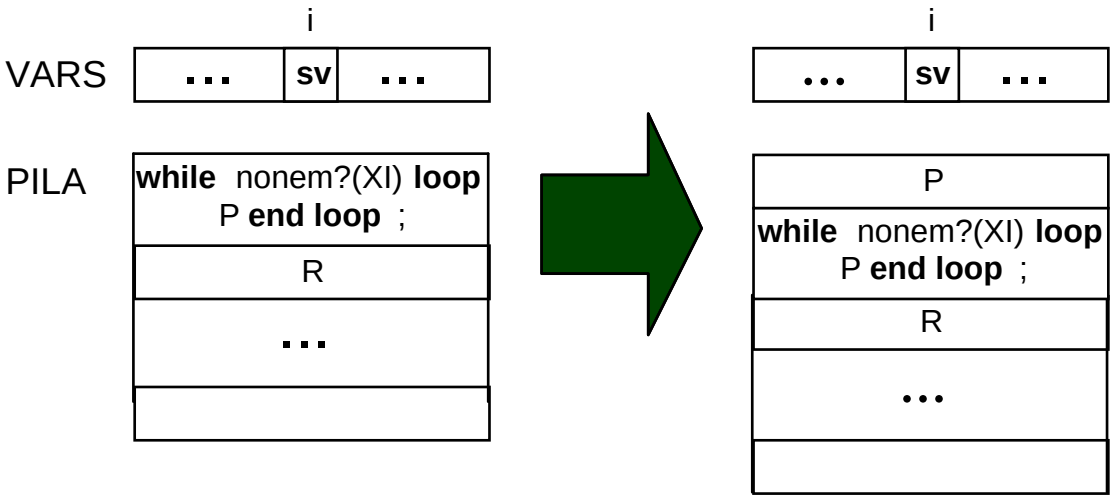

Figura 4.6: Ciclo de ejecución de **U** cuando el siguiente programa a ejecutar es iterativo y además su condición se cumple: entonces se efectuará al menos su primera iteración.

Puede parecer a simple vista que los programas de *ult_variable* y de Φ siguen los mismos principios y tienen comportamientos semejantes, pero existe entre ellos una diferencia fundamental que afecta al uso de la pila. El primero corresponde a un proceso de análisis, y un programa que sale de la pila no vuelve a entrar en ella (aunque probablemente sí sus fragmentos). Sin embargo **U** corresponde a un proceso de simulación en el que la ejecución de los bucles puede exigir reintroducir una y otra vez el mismo programa.

La estrategia detallada en los párrafos y figuras que anteceden dan lugar al siguiente programa **U**:

```
VARS(ult_variable (X1)) := ε;
VARS(1) := X2;
-- inicializamos a ε todas las variables del vector de estado de Px antes de cargar el dato y
PILA := empilar (X1, <]);
while not P_vacía? (PILA) loop
      PROG := cima (PILA); PILA := desempilar (PILA);
      if asig_vacía? (PROG) then
            VARS(ind_var(PROG)) := ε;
      elsif asig_cons? (PROG) then
            VARS(ind_var(PROG)) :=
                  ind_simb (PROG) & VARS(ind_var2 (PROG));
      elsif asig_cdr? (PROG) then
            VARS(ind_var(PROG)) := cdr (VARS(ind_var2 (PROG)));
      elsif composición?(PROG) then
            PILA := empilar (inst_int (PROG),
                              empilar (inst_int2 (PROG), PILA));
      elsif condición? (PROG) then
            if ind_simb(PROG) = primero (VARS(ind_var (PROG))) then
                  PILA := empilar (inst_int (PROG), PILA);
            end if;
      else
            -- se trata de una iteración
            if nonem? (VARS(ind_var (PROG))) then
                  PILA := empilar (inst_int (PROG), empilar (PROG, PILA));
            end if;
      end if;
end loop;
-- si la ejecución de Px termina descargamos el resultado
X0 := VARS (0); ⌫
```

De nuevo nos remitimos al Apéndice C para cualquier aclaración sobre las operaciones realizadas con los tipos implementados que se usan en el macroprograma.

GENERALIZACIÓN: Este resultado se generaliza para demostrar que también es computable la función universal para funciones computables de **k** argumentos. Así, tenemos que para todo $\mathbf{k}\geq 0$ la siguiente función $\Phi^{k+1} : \mathbb{W} \times \sum^{*k} \longrightarrow \sum^{*}$ es computable:

$$\Phi^{k+1}(\mathbf{x},\mathbf{y_1},\ldots,\mathbf{y_k}) = \begin{cases} \varphi_{\mathbf{x}}(\mathbf{y}_1,\mathbf{y}_2,\ldots,\mathbf{y_k}) & \varphi_{\mathbf{x}}(\mathbf{y}_1,\mathbf{y}_2,\ldots,\mathbf{y_k})\downarrow \\ \perp & \text{c.c.} \end{cases}$$

⌦ Bastará un programa idéntico al descrito más arriba, pero sustituyendo su segunda línea por:

```
VARS(1):= X2;
VARS(2):= X3;
VARS(3):= X4;
 ...
VARS(K):= X[K+1]; ⌫
```

> *Históricamente, la demostración del Teorema de Enumeración supuso un avance conceptual mucho mayor que el que deja entender su propia demostración. Hay que tener en cuenta que, en los albores de la Teoría de la Computabilidad (cuando, recordemos, aún no existían los computadores) cada programa era considerado la descripción del comportamiento operativo de un sistema computacional concreto. Es decir que la existencia de un algoritmo demostraba que podía construirse una máquina que realizase esa tarea concreta. La computabilidad de la Función Universal supuso la demostración de la existencia de computadores de propósito general, ya que un dispositivo único (aquel que es capaz de computar el programa* ***U****) resulta capaz de modular su comportamiento para simular eficientemente el de cualquier otro dispositivo físico existente. Esto supuso el establecimiento definitivo de la dualidad hardware/software y la formalización del concepto de programabilidad, o capacidad que tienen los dispositivos computacionales de cambiar radicalmente su funcionalidad sin necesitar alterar su estructura física.*

Dado que, como hemos demostrado, la función universal es computable podremos utilizarla en macroexpresiones de cualquier otro programa, y así demostrar la computabilidad de funciones como, por ejemplo, $\chi : \mathbb{W} \times \sum^{*} \longrightarrow \sum^{*}$ definida como sigue:

$$\chi(\mathbf{x},\mathbf{y}) \cong \begin{cases} \varphi_{\mathbf{x}}(\mathbf{y}) & \text{car}_{\mathbf{a}}?(\mathbf{y}) \\ \mathbf{y}^{R} & \text{car}_{\mathbf{b}}?(\mathbf{y}) \wedge \varphi_{\mathbf{x}}(\mathbf{x}\bullet\mathbf{x})\downarrow \\ \perp & \text{c.c.} \end{cases}$$

El macroprograma que la computará será algo tan sencillo como:

```
if car_a?(X2) then
      X0 := Φ(X1, X2);
elsif car_b?(X2) then
      R := Φ(X1, X1&X1);
      X0 := X2^R;
else X0 := ⇊ ;
end if;
```

Adviértase que la macroinstrucción R := Φ(X1, X1&X1); sólo tiene por objeto actuar de filtro. En caso de que $\varphi_x(\mathbf{x}\bullet\mathbf{x})$ diverja obtenemos el comportamiento esperado ya que $\chi(\mathbf{x},\mathbf{y})$ quedará indefinida. En caso contrario el programa pasará a la siguiente instrucción pudiéndose calcular y devolver $\mathbf{y}^R$.

4.3 Ejecución suspendida

Con el teorema de enumeración comprobamos que podemos simular el funcionamiento de cualquier programa-while hasta el final, pero en ocasiones esto no resulta lo adecuado, dado que si dicho programa cicla nos arrastrará en su computación infinita.

Ocasionalmente resultará más conveniente suspender la ejecución de un programa tras un tiempo prudencial, aunque haciéndolo no sepamos con certeza si el cómputo estaba destinado a finalizar o no. Para ello resultará muy útil el predicado **T**, que explora la convergencia de un programa sobre un dato con un número de pasos como tope máximo, donde entendemos que un paso de computación corresponde a la realización de una acción del programa $\mathbf{P_x}$, es decir, ejecución de una asignación o evaluación de una condición.

PROPOSICIÓN: El predicado $\mathbf{T}: \mathbb{W} \times \sum^* \times \mathbb{N} \to \mathbb{B}$, definido como sigue, es decidible:

$$\mathbf{T}(\mathbf{x},\mathbf{y},\mathbf{p}) \Leftrightarrow \mathbf{P_x} \text{ converge sobre } \mathbf{y} \text{ en } \mathbf{p} \text{ ó menos pasos}$$

⌦ El programa que computa el predicado **T** será muy similar al de Φ, pero usando un contador de pasos para suspender la ejecución:

```
VARS(ult_variable (X1)) := ε;   VARS(1) := X2;
PASOS := 0;  PILA := empilar (X1, <]);
while  not P_vacía? (PILA) and PASOS<X3  loop
      PASOS := PASOS+1;
      PROG := cima (PILA);  PILA := desempilar (PILA);
      if  asig_vacía? (PROG)  then
            VARS(ind_var(PROG)) := ε;
```

```
        elsif  asig_cons? (PROG)  then
               VARS (ind_var(PROG)) :=
                     ind_simb (PROG) • VARS (ind_var2 (PROG));
        elsif  asig_cdr? (PROG)  then
               VARS (ind_var(PROG)) := cdr (VARS (ind_var2 (PROG)));
        elsif  composición?(PROG)  then
               PILA:= empilar(inst_int(PROG),
                              empilar(inst_int2 (PROG), PILA));
               PASOS := PASOS–1;
               -- único caso en el que no se realiza acción alguna
        elsif  condición? (PROG)  then
               if  ind_simb(PROG) = primero(VARS(ind_var (PROG)))  then
                     PILA := empilar (inst_int (PROG), PILA);
               end if;
        else
               if  nonem? (VARS (ind_var (PROG)))  then
                     PILA := empilar (inst_int (PROG), empilar (PROG, PILA));
               end if;
        end if;
  end loop;
-- el predicado será cierto si la ejecución de Px se ha completado
  X0 := P_vacía? (PILA);  ⌫
```

Nótese que la ejecución de un programa no puede realizarse nunca en menos de un paso, por lo que T(**x**,**y**,0) siempre será falso.

Naturalmente, cuando el resultado de **T**(**x**,**y**,**p**) es falso no recibimos información de la causa. Puede deberse a que $\mathbf{P_x}$ no converge sobre **y** o a que el número de pasos **p** es insuficiente para completar el cómputo.

Algunas veces resultará útil no solamente preguntar si el cómputo finaliza, sino también el resultado con el que lo hace. Por ello definimos otro predicado **E**, que toma como argumento adicional el resultado por el que se desea indagar.

PROPOSICIÓN: El siguiente predicado $\mathbf{E} : \mathbb{W} \times \sum^* \times \mathbb{N} \times \sum^* \rightarrow \mathbb{B}$ es decidible:

E(**x**,**y**,**p**,**z**) $\Leftrightarrow$ $\mathbf{P_x}$ converge sobre **y** en **p** o menos pasos produciendo **z** como salida

⌦ Solo es necesario un pequeño retoque sobre el programa que computa **T** para obtener el que computa **E**. Bastará con cambiar su última instrucción por:

```
X0 := P_vacía? (PILA)  and  VARS(0) = X4;  ⌫
```

De nuevo nos encontramos con que un resultado negativo para **E**(**x**,**y**,**p**,**z**) puede ser por motivos variados. Puede deberse a que $\mathbf{P_x}$ no converge sobre **y**, a que el número de pasos **p** es insuficiente para completar el cómputo, o a que éste termina produciendo un resultado distinto de **z**.

GENERALIZACIÓN: Los resultados anteriores se generalizan para demostrar que también son decidibles los predicados **T** y **E** para programas-while usados con **k** entradas. Así, tenemos que para todo $\mathbf{k}\geq 0$ los siguientes predicados $\mathbf{T}^{k+2}: \mathbb{W} \times \sum^{*k} \times \mathbb{N} \rightarrow \mathbb{B}$ y $\mathbf{E}^{k+3}: \mathbb{W} \times \sum^{*k} \times \mathbb{N} \times \sum^{*} \rightarrow \mathbb{B}$, definidos como sigue, son decidibles:

$$\mathbf{T}^{k+2}(\mathbf{x},\mathbf{y}_1, \ldots,\mathbf{y}_k,\mathbf{p}) \Leftrightarrow \mathbf{P}_\mathbf{x} \text{ converge sobre } (\mathbf{y}_1, \ldots,\mathbf{y}_k) \text{ en } \mathbf{p} \text{ o menos pasos}$$

$$\mathbf{E}^{k+3}(\mathbf{x},\mathbf{y}_1, \ldots,\mathbf{y}_k,\mathbf{p},\mathbf{z}) \Leftrightarrow \mathbf{P}_\mathbf{x} \text{ converge sobre } (\mathbf{y}_1, \ldots,\mathbf{y}_k) \text{ en } \mathbf{p} \text{ o menos pasos produciendo } \mathbf{z} \text{ como resultado}$$

⌦ En el caso de $\mathbf{T}^{k+2}$ habrá que modificar el programa de **T** sustituyendo su primera línea por:

```
VARS (1) := X2;
VARS (2) := X3;
VARS (3) := X4;
 ...
VARS (K) := X[K+1];
```

y la condición del bucle por:

```
while not P_vacía? (PILA) and PASOS < X[K+2] loop
```

Para $\mathbf{E}^{k+3}$ será necesario, además de lo anterior, reemplazar la asignación final por lo siguiente:

```
X0 := P_vacía? (PILA) and VARS(0) = X[K+3];
```
⌫

Naturalmente, los predicados **T** y **E** definidos inicialmente no son sino los casos particulares $\mathbf{T}^3$ y $\mathbf{E}^4$.

4.4 Intercalado de procesos

Cuando programamos un algoritmo para demostrar la computabilidad de una cierta función, podemos encontrarnos con la necesidad de ejecutar, dentro de nuestro proceso, uno o varios programas con uno o varios datos para obtener nuestro objetivo. Por ejemplo, si queremos probar que la siguiente función $\psi: \mathbb{W} \times \mathbb{N} \dashrightarrow \mathbb{N}$ es computable:

$$\psi(\mathbf{x},\mathbf{y}) \cong \begin{cases} 12 & \varphi_\mathbf{x}(\mathbf{y})\downarrow \ \wedge \ \varphi_\mathbf{x}(\mathbf{y}+1)\downarrow \\ \\ \perp & \text{c.c.} \end{cases}$$

hemos de ejecutar dos procesos: el correspondiente al programa **x** con el dato **y** ($\varphi_\mathbf{x}(\mathbf{y})$) y el correspondiente al programa **x** con el dato **y**+1 ($\varphi_\mathbf{x}(\mathbf{y}+1)$). Como en este

caso queremos comprobar si ambos son convergentes, los podemos ejecutar de manera secuencial. Por ejemplo, el siguiente programa computaría ψ:

```
R := Φ(X1, X2);
R := Φ(X1, X2+1);
X0 := 12;
```

Se pueden producir tres situaciones:

a) $\varphi_x(y)\uparrow$, con lo que la primera asignación de nuestro macroprograma no llega a término; el comportamiento es correcto, ya que en ese caso no se cumple la condición $\varphi_x(y)\downarrow \wedge \varphi_x(y+1)\downarrow$, con lo que $\psi(x,y)$ es indefinido.

b) $\varphi_x(y)\downarrow$ pero $\varphi_x(y+1)\uparrow$, con lo que la primera asignación de nuestro macroprograma termina pero la segunda no; el comportamiento es correcto, ya que en ese caso tampoco se cumple la condición $\varphi_x(y)\downarrow \wedge \varphi_x(y+1)\downarrow$, con lo que $\psi(x,y)$ es indefinido.

c) $\varphi_x(y)\downarrow$ y $\varphi_x(y+1)\downarrow$, con lo que las dos primeras asignaciones de nuestro macroprograma terminan y producimos el resultado 12; el comportamiento es correcto, ya que en ese caso sí se cumple la condición $\varphi_x(y)\downarrow \wedge \varphi_x(y+1)\downarrow$, con lo que $\psi(x,y)$ es precisamente ese valor.

Sin embargo, si nos encontráramos ante la necesidad de demostrar la computabilidad de la siguiente función $\chi : \mathbb{W}^2 \times \mathbb{N} \longrightarrow \mathbb{N}$:

$$\chi(x,y,z) \cong \begin{cases} z^2 & \varphi_x(z)\downarrow \ \vee \ \varphi_y(2*z)\downarrow \\ \\ \perp & \text{c.c.} \end{cases}$$

estaríamos en una situación muy diferente. Al igual que antes, tenemos dos procesos sobre cuya convergencia tenemos interés: el correspondiente al programa **x** con el dato **z** ($\varphi_x(z)$) y el correspondiente al programa **y** con el dato $2*z$ ($\varphi_y(2*z)$). Pero ahora no nos interesa la convergencia de ambos, ya que nos basta la de uno solo de ellos. El problema es que si solamente uno de ellos converge no podemos saber *a priori* de cuál se trata, por lo que si nuestro programa ejecuta $\varphi_x(z)$ antes que $\varphi_y(2*z)$ no computará correctamente la función χ para todos los casos en los que $\varphi_x(z)\uparrow$ pero $\varphi_y(2*z)\downarrow$ (el programa ciclará cuando debería devolver z^2). Pero tampoco resolvemos nada ejecutando primero $\varphi_y(2*z)$, ya que habrá otros casos en los que $\varphi_x(z)\downarrow$ pero $\varphi_y(2*z)\uparrow$.

La solución ideal sería ejecutar ambos procesos en paralelo, de forma que en cuanto uno de ellos terminase pudiéramos abortar el otro y producir el resultado deseado, pero no disponemos de procesamiento paralelo en los programas-while. Lo que sí podemos hacer es simular el paralelismo cediendo alternativamente

tiempo de cómputo a ambos procesos: primero ejecutamos un paso del proceso $\varphi_x(\mathbf{z})$, luego un paso de $\varphi_y(2*\mathbf{z})$, a continuación un segundo paso de $\varphi_x(\mathbf{z})$, otro de $\varphi_y(2*\mathbf{z})$, y así sucesivamente. De este modo seremos capaces de detectar si uno de los dos procesos tiene fin independientemente de que el otro termine o no. La situación se ilustra con un ejemplo hipotético en la figura 4.7.

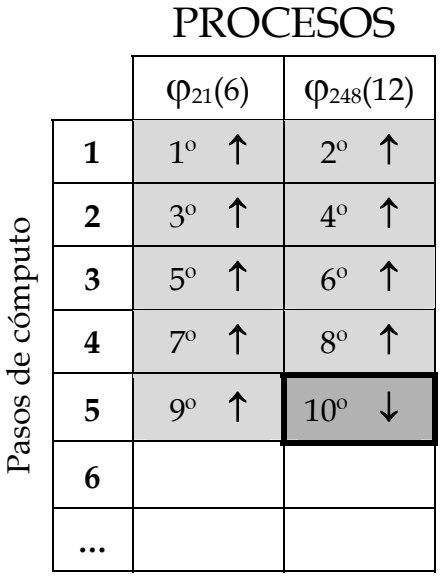

Figura 4.7: Ejemplo de cálculo de $\chi(21, 248, 6)$, suponiendo que $\varphi_{21}(6)$ diverge y que $\varphi_{248}(12)$ converge en 5 pasos. Se indica en la tabla el orden de ejecución de los pasos de ambos procesos hasta dar con la condición buscada, así como el resultado obtenido en cada caso, que puede ser la terminación del proceso en esos pasos (↓) o la no terminación del mismo (↑), de momento.

Para realizar esto será crucial el concurso del predicado decidible **T** definido en la sección anterior. Dado que este nos informa de la convergencia o no de un cómputo en un número de pasos concreto, utilizaremos **T**(**x**, **z**, **p**) para "ejecutar **p** pasos de $\varphi_x(\mathbf{z})$". Veamos cómo sería el programa correspondiente a la función χ:

```
PASOS := 1;
-- la variable PASOS indica hasta dónde hemos llegado a computar ambos procesos
while not T(X1, X3, PASOS) and not T(X2, 2*X3, PASOS) loop
      PASOS := PASOS+1;
end loop;
X0 := X3*X3;
```

De nuevo se pueden producir tres situaciones:

a) $\varphi_x(\mathbf{z})\downarrow$ pero $\varphi_y(2*\mathbf{z})\uparrow$ (o viceversa); la variable PASOS alcanzará el valor necesario para completar el proceso convergente, por lo que el bucle terminará y produciremos el valor esperado $\mathbf{z}^2$.

b) $\varphi_x(\mathbf{z})\downarrow$ y $\varphi_y(2*\mathbf{z})\downarrow$; la variable PASOS alcanzará el valor necesario para completar el proceso que converja primero, por lo que el bucle terminará y produciremos el valor esperado $\mathbf{z}^2$.

c) $\varphi_x(\mathbf{z})\uparrow$ y $\varphi_y(2*\mathbf{z})\uparrow$; la variable PASOS crecerá indefinidamente y el bucle no terminará, con lo que volveremos a tener el comportamiento correcto, ya que en este caso $\chi(\mathbf{x},\mathbf{y},\mathbf{z})$ es indefinido.

> *Hay alguna diferencia notable entre la mecánica de la simulación paralela de cómputos que hemos descrito y la que sería razonable en un entorno real, como puede ser un sistema operativo. En el ejemplo de la figura 4.7, cuando comprobamos que $\varphi_{248}(12)$ no converge en 4 pasos y procedemos a calcular el quinto paso de $\varphi_{21}(6)$, no seguimos este último proceso desde el punto donde lo dejamos, sino que recalculamos los cuatro primeros pasos ya realizados en la iteración anterior. Aunque esto sería prohibitivo en un sistema real, nos evita tener que almacenar de forma explícita en nuestro programa el vector de estado de todos los procesos intercalados. Lo mismo se puede decir en cuanto al tiempo asignado a cada proceso: en nuestro caso sólo una nueva instrucción en cada turno, mientras que en la vida real no es raro permitir algunas decenas de miles, buscando optimizar tanto los tiempos de espera como el trasiego en los registros del procesador.*

Podríamos resumir diciendo que hay ocasiones en las que la resolución de un problema nos exige analizar un conjunto de procesos que eventualmente podrían ser divergentes, aunque el análisis no requiere necesariamente completar toda la serie de cómputos, sino únicamente encontrar entre ellos alguno que no diverja y verifique una condición determinada. En estos casos tendremos que aguzar el ingenio para buscar la solución sorteando los cómputos que sí son infinitos, para así poder dar la respuesta requerida. Nos será de gran utilidad la técnica de ***intercalado de procesos*** que acabamos de describir, que se caracteriza por permitirnos encajar o ensamblar ('*dovetail*', en inglés) los cómputos asociados a los mismos, aunque sean potencialmente infinitos. Es por ello que se le suele denominar también ***técnica de 'dovetailing'***.

4.4.1 Intercalado de un número acotado de procesos

El número de procesos que es necesario ejecutar en paralelo puede ser mucho mayor que dos, e incluso no poder determinarse hasta el tiempo de ejecución.

Por ejemplo, si queremos demostrar la computabilidad de la función $\xi: \mathbb{W} \longrightarrow \mathbb{B}$ que busca para cada programa $\mathbf{P_x}$ una entrada que produzca como resultado la palabra **ab**. Esta entrada **z** debe encontrarse dentro de un rango cuyo tamaño depende a su vez de **x**:

$$\xi(\mathbf{x}) \cong \begin{cases} \text{true} & \exists \mathbf{z}\ (\mathbf{z} < \mathbf{x} \ \wedge\ \varphi_{\mathbf{x}}(\mathbf{z}) = \mathbf{ab}) \\ \perp & \text{c.c.} \end{cases}$$

En este caso, dado un programa $\mathbf{P_x}$ se trata de comprobar si alguno de los procesos $\varphi_x(0)$, $\varphi_x(1)$, …, $\varphi_x(\mathbf{x}\text{-}1)$ produce **ab** como resultado, para lo cual será necesario que converja.

De nuevo, la ejecución secuencial de dichos procesos resulta totalmente inadecuada, por lo que procederemos a hacer un análisis paralelo de los mismos, mediante asignación ordenada de pasos de ejecución a cada uno de ellos. En la figura 4.8 se describe un ejemplo hipotético de dicho análisis.

PROCESOS

Pasos de cómputo	$\varphi_6(0)$	$\varphi_6(1)$	$\varphi_6(2)$	$\varphi_6(3)$	$\varphi_6(4)$	$\varphi_6(5)$
1	1^o ↑	2^o ↑	3^o ↑	4^o ↑	5^o ↑	6^o ↑
2	7^o ↑	8^o ↑	9^o ↑	10^o ↑	11^o ↑	12^o ↑
3	13^o ↑	14^o **=a**	15^o ↑	16^o ↑	17^o ↑	18^o ↑
4	19^o ↑	20^o **=a**	21^o ↑	22^o ↑	23^o **=ε**	24^o ↑
5	25^o ↑	26^o **=a**	27^o **=ab**			
6						**=ab**
…						

Figura 4.8: Ejemplo de cálculo de $\xi(6)$. Se indica en la tabla el orden de ejecución de los pasos de los procesos hasta dar con la condición buscada, así como el resultado obtenido en cada caso, que puede ser la no terminación del proceso en esos pasos (↑) o su terminación produciendo como resultado una palabra **w** (=**w**). El análisis descubre que $\varphi_6(2)$ produce efectivamente **ab** en 5 pasos, sin que suponga obstáculo para descubrirlo el hecho de que diverjan $\varphi_6(0)$ y $\varphi_6(3)$. Se ha supuesto también que $\varphi_6(5)$ produce asimismo como resultado **ab**, pero como necesita 6 pasos para ello nuestro análisis no llega a comprobarlo.

El programa que implementará el algoritmo explicado en dicha figura tendrá un bucle principal gobernado por una variable PASOS análoga a la del ejemplo anterior. Dentro del mismo habrá otro bucle que explorará los procesos correspondientes hasta el número de pasos indicado, utilizando para ello el también útil predicado **E** (ya que en este caso nos interesa el resultado de la computación):

```
PASOS := 1;
ENCONTRADO := false;
while not ENCONTRADO loop
    for Z in ε .. ant(X1) loop
        ENCONTRADO := ENCONTRADO or E(X1, Z, PASOS, ab);
    end loop;
    PASOS := PASOS+1;
end loop;
X0 := true;
```

Como en los casos anteriores, puede suceder que el bucle principal incremente indefinidamente la variable PASOS sin encontrar un valor de **z** que haga que $\mathbf{P_x}$ devuelva el resultado **ab**. pero en ese caso el comportamiento divergente de nuestro programa también se corresponde con el hecho de que $\xi(\mathbf{x})\uparrow$.

> *Otra característica poco realista de nuestro método de simulación paralela es que cuando un proceso termina con resultado insatisfactorio el proceso de análisis no termina para dicho proceso. En el ejemplo de la figura 4.8, tras la ejecución de su tercer paso ya sabemos que $\varphi_6(1) = \boldsymbol{a}$, y que por tanto el dato 1 no nos interesa. Sin embargo seguimos probando con 4 y 5 pasos. Podríamos evitar esto, pero ello nos obligaría a llevar una tabla de procesos terminados sin éxito. De nuevo nos encontramos con el hecho de que, para nuestros propósitos, es preferible un programa corto y sencillo aunque sea ineficiente. La razón es que lo único relevante de los programas-while y macroprogramas es su mera existencia. Si no están destinados a ser ejecutados, ahorremos al menos esfuerzos a la hora de diseñarlos.*

4.4.2 Intercalado de un número no acotado de procesos

Con todo, aún se puede realizar un refinamiento ulterior de esta idea de simulación del paralelismo de manera que nos sea útil para resolver problemas aún más complicados. En la función del último ejemplo, el número de procesos a analizar, aunque se determinaba en tiempo de ejecución, siempre estaba acotado (para la entrada $\mathbf{P_x}$ era preciso estudiar **x** procesos), y ello nos permitía definir un algoritmo justo de reparto de los tiempos de ejecución basado en turnos. Sin embargo, en otros casos el número de procesos implicado puede ser infinito, lo que representa una complicación notable para la aplicación de un algoritmo similar.

Ejemplo de ello puede ser la función $\theta : \mathbb{W} \times \sum^* \longrightarrow \mathbb{B}$:

$$\theta(\mathbf{x},\mathbf{y}) \cong \begin{cases} \text{true} & \mathbf{y} \in \mathbf{R_x} \\ \perp & \text{c.c.} \end{cases}$$

Supongamos que queremos demostrar su computabilidad. Un posible método para comprobar si $\mathbf{y} \in \mathbf{R_x}$ sería estudiar los posibles resultados del programa $\mathbf{P_x}$ para cada una de las posibles entradas, ejecutando los procesos $\varphi_x(0)$, $\varphi_x(1)$, $\varphi_x(2)$… . El problema es que, a diferencia de antes, esta lista no tiene fin, lo que tiene un efecto desagradable sobre nuestra técnica: ya no tiene sentido que calculemos primero un paso de cómputo de todos los procesos, luego dos, luego tres, …, por la sencilla razón de que nunca terminaríamos de adjudicar a todos ellos su primer paso.

Una solución razonable podría ser empezar con pocos pasos y con pocos procesos, para ir incrementando ambas cosas a medida que el análisis avanza. Si

hacemos esto de manera cuidadosa, todos los procesos tendrán su oportunidad de ser analizados, y además todos ellos podrán recibir una asignación de pasos tan generosa como sea necesario. Por ejemplo, podemos calcular 4 pasos de los procesos $\varphi_x(0)$, $\varphi_x(1)$, $\varphi_x(2)$, $\varphi_x(3)$ y $\varphi_x(4)$. Si no obtenemos el resultado esperado (porque no terminan, o porque no producen la salida buscada), pasaremos a ejecutar 5 pasos de los procesos $\varphi_x(0)$, $\varphi_x(1)$, $\varphi_x(2)$, $\varphi_x(3)$, $\varphi_x(4)$ y $\varphi_x(5)$, y así sucesivamente. Parece obvio que de esta manera el que unos procesos diverjan so supone obstáculo para que los convergentes sean llevados a término. En la figura 4.9 se observa la aplicación de esta idea al cálculo de un valor de la función θ.

		PROCESOS						
		$\varphi_{207}(0)$	$\varphi_{207}(1)$	$\varphi_{207}(2)$	$\varphi_{207}(3)$	$\varphi_{207}(4)$	$\varphi_{207}(5)$	•••
Pasos de cómputo	**1**	1º ↑	2º ↑					
	2	3º ↑	4º ↑	5º ↑				
	3	6º ↑	7º =ε	8º ↑	9º ↑			
	4	10º ↑	11º =ε	12º ↑	13º ↑	14º ↑		
	5	15º ↑	16º =ε	17º **=ab**	18º ↑	19º ↑	20º **=ab**	
	6	21º ↑	22º =ε	23º **=ab**	24º **=bb**			
	7							
	…							

Figura 4.9: Ejemplo de cálculo de $\theta(207,\mathbf{bb})$. Se indica en la tabla el orden de adjudicación de los pasos a los procesos hasta dar con la condición buscada, así como el resultado obtenido en cada caso, que puede ser la no terminación del proceso en ese número de pasos (↑) o su finalización produciendo como resultado una palabra **w** (=**w**). El análisis descubre que $\mathbf{bb} \in \mathbf{R_{207}}$ ya que $\varphi_{207}(3)$ produce **bb** en 6 pasos. Al final sólo se han analizado 6 entradas diferentes, pero si, por ejemplo, $\varphi_{207}(3)$ hubiera necesitado 1000 pasos para calcularse y **bb** no se produjera como resultado en ningún otro cómputo, se habría explorado hasta $\varphi_{207}(1000)$.

El programa que computaría la función θ sería:

```
PASOS := 1;
ENCONTRADO := false;
while not ENCONTRADO loop
      for DATO in ε .. PASOS loop
            ENCONTRADO := ENCONTRADO or E(X1, DATO, PASOS, X2);
      end loop;
      PASOS:= PASOS+1;
end loop;
X0:= true;
```

Concluimos que la técnica de intercalado es extensible a los casos en los que la lista de procesos a analizar es infinita. Este aparente contrasentido de explorar exhaustivamente un conjunto infinito se resuelve porque la solución, de hallarse, se encontrará en algún proceso convergente concreto. El único problema es que no sabemos de antemano de cuál se trata, y de ahí la necesidad de ampliar indefinidamente y de forma sistemática nuestro campo de búsqueda para tener asegurado el éxito por muy lejos que se encuentre en la lista el cómputo buscado, y por muchos pasos que se necesiten para verificarlo. Naturalmente, como en todos los demás casos de *dovetailing*, este tipo de búsqueda no tiene fin en caso de que no se verifique nunca la condición de la solución, por lo que solo será aplicable cuando no tengamos como exigencia producir algún tipo de respuesta ante las búsquedas fallidas. Por ejemplo, si quisiéramos computar la siguiente función $\mathbf{g}: \mathbb{W} \times \sum^* \to \mathbb{B}$:

$$\mathbf{g}(\mathbf{x},\mathbf{y}) \cong \begin{cases} \text{true} & \mathbf{y} \in \mathbf{R}_\mathbf{x} \\ \text{false} & \text{c.c.} \end{cases}$$

entonces la técnica de intercalado sería absolutamente inútil[4].

4.4.3 Uso de las funciones de codificación

En algunos casos la técnica de intercalado de procesos puede complicarse por el hecho de que nuestro espacio de búsqueda tenga muchas dimensiones. En los ejemplos vistos anteriormente (caso de las funciones ξ y θ) tal cosa no sucedía: el problema original nos pedía buscar un dato que provocara un cierto comportamiento sobre un programa dado, y el método de intercalado nos pedía además buscar un número de pasos suficiente para manifestar dicho comportamiento mediante la convergencia del programa. Esas dos dimensiones de búsqueda (dato, pasos) nos permitían reflejar la idea de intercalado en una tabla bidimensional. Pero si tratamos con la siguiente función $\eta: \mathbb{W} \longrightarrow \mathbb{B}$:

$$\eta(\mathbf{x}) \cong \begin{cases} \text{true} & \varphi_\mathbf{x} \text{ no es inyectiva} \\ \perp & \text{c.c.} \end{cases}$$

la situación es un poco más complicada. ¿Cómo podemos comprobar, dado un programa $\mathbf{P_x}$, que computa una función no inyectiva? Habremos de encontrar dos datos distintos para los que el programa produzca el mismo resultado **r**. La búsqueda de esos dos valores está condicionada por el hecho de que, si intentamos

[4] Cosa, por otro lado, bastante esperable, ya que se trata de una función incomputable. De todas formas no estaremos en condiciones de probarlo hasta el siguiente capítulo.

verificar lo que buscamos con dos valores concretos **y** y **z**, tanto $\varphi_x(\mathbf{y})$ como $\varphi_x(\mathbf{z})$ pueden divergir. Por ello utilizaremos la técnica de intercalado de procesos: empezaremos probando con un paso de computación, luego con dos, y así sucesivamente, y para cada número de pasos **p** probaremos todos los pares de datos distintos (**y**,**z**) cuyos valores estén en el rango 0..**p**. De este modo, tendremos la seguridad de que, en caso necesario, todos los posibles pares de datos distintos que se le pueden suministrar como entrada a $\mathbf{P_x}$ tendrán su oportunidad de ser comprobados, y al mismo tiempo, que cada hipotético par de procesos ($\varphi_x(\mathbf{y})$, $\varphi_x(\mathbf{z})$) será ejecutado tantos pasos de cómputo como sea necesario. El programa resultante sería:

```
PASOS := 1;
ENCONTRADO := false;
while not ENCONTRADO loop
      for Y in Ɛ .. PASOS loop
            for Z in Ɛ .. PASOS loop
                  if Y<>Z and T(X1, Y, PASOS) and T(X1, Z, PASOS) then
                        ENCONTRADO := ENCONTRADO ∨ Φ(X1,Y) =
Φ(X1,Z);
                  end if;
            end loop;
      end loop;
      PASOS := PASOS+1;
end loop;
X0 := true;
```

En este tipo de problemas puede ser muy útil la siguiente consideración: en realidad la técnica de intercalado es una manera de buscar exhaustivamente en un espacio multidimensional. Por ejemplo, para computar la función η necesitamos encontrar varias cosas: un dato **y**, otro dato **z** y un número de pasos **p** que asegure la convergencia tanto de $\varphi_x(\mathbf{y})$ como $\varphi_x(\mathbf{z})$.

> *En realidad podríamos tener en cuenta que los procesos de computación de $\varphi_x(\mathbf{y})$ y $\varphi_x(\mathbf{z})$ no tienen por qué necesitar el mismo número de pasos para su terminación, y acorde con ello buscar dos valores $\mathbf{p_1}$ y $\mathbf{p_2}$ específicos para cada uno de ellos. Sin embargo, aunque un criterio de eficiencia podría aconsejarnos no simular la ejecución de los programas más allá de lo necesario, para nuestros objetivos es más importante abreviar el texto del programa a diseñar. En este caso, aprovechando que los predicados **T** y **E** se siguen cumpliendo para números de pasos más altos de los estrictamente necesarios, nos resulta más sencilla la búsqueda de un **p** único.*

Dado que tanto **y**, como **z**, como **p** pueden tomar cualquier valor (siempre que **y≠z**), lo que en realidad hacemos es recorrer de manera sistemática posibles valores

para la tripleta (**y**,**z**,**p**) hasta encontrar los adecuados, si es que existen. Pero sabemos que existe una codificación biunívoca entre palabras y ternas de palabras (véase la sección ***b*** del Apéndice C), por lo que recorrer todos los posibles tríos de la forma (**y**,**z**,**p**) equivale a recorrer todos los posibles códigos de tríos de la forma **u** = cod^3(**y**,**z**,**p**), con la ventaja de que estos son simplemente las palabras de $\sum^*$ y pueden ordenarse de manera secuencial. Para acceder a los componentes individuales de **u** acudiremos a las funciones de decodificación, ya que **y** = $decod_{3,1}$(**u**), **z** = $decod_{3,2}$(**u**) y **p** = $decod_{3,3}$(**u**). El programa asociado a la técnica descrita de recorrido y búsqueda podría ser el siguiente:

```
X0 := true;
U := ℰ;
ENCONTRADO := false;
while  not ENCONTRADO loop
-- Comprobamos si U constituye una terna de valores (y,z,p) adecuada
      Y := decod_3_1(U);
      Z := decod_3_2(U);
      P := decod_3_3(U);
      if  Y<>Z and T(X1, Y, P) and T(X1, Z, P)  then
            ENCONTRADO := ENCONTRADO ∨ Φ(X1, Y) = Φ(X1, Z);;
      end if;
      U := sig (U);
end loop;
```

Aún podemos hacer un programa más compacto si, en vez de calcular el resultado común deseado **r** = φ_x(**y**) = φ_x(**z**), lo "buscamos" entre todos los posibles, tal como se indica en la figura 4.10.

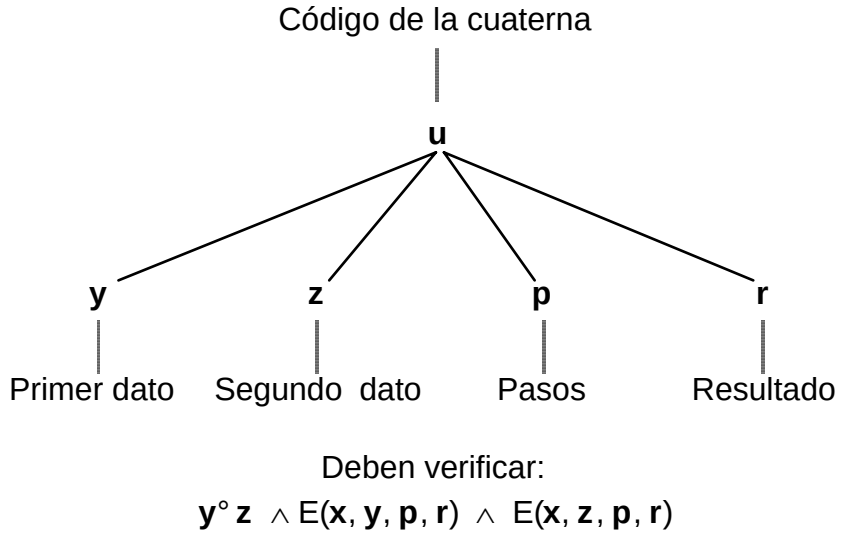

Figura 4.10: Para determinar que φ_x no es inyectiva buscamos dos datos distintos **y** y **z**, un número de pasos **p** que haga convergir a los procesos φ_x(**y**) y φ_x(**z**) y un resultado común **r** para ambos. Una forma sencilla de hacerlo es recorriendo todas las posibles cuaternas de valores (**y**,**z**,**p**,**r**) mediante sus códigos cod^4(**y**,**z**,**p**,**r**).

El siguiente programa implementa la técnica de búsqueda citada:

```
X0 := true;   U := ε;
while  not (decod_4_1(U) <> decod_4_2(U) and
      E (X1, decod_4_1(U), decod_4_3(U), decod_4_4(U)) and
      E (X1, decod_4_2(U), decod_4_3(U), decod_4_4(U)))  loop
            U := sig (U);
end loop;
```

que resulta de estructura mucho más simple que las dos versiones anteriores, aunque contenga una condición de terminación relativamente compleja. La ventaja fundamental de este método es su escalabilidad: al no tener que explicitar el orden de exploración de los distintos valores en el esquema de control del programa, la técnica se aplica con poca dificultad adicional a casos en los que sea precisa una búsqueda en un espacio de varias dimensiones.

4.4.4 Inversión de funciones

Un asunto interesante que se puede abordar mediante la técnica de intercalado de procesos es el de la reversibilidad de las operaciones realizadas por un computador. Si un programa transforma el dato **x** en el resultado **y**, ¿qué posibilidades hay de recuperar **x** a partir de **y**?

En algunos casos no tiene sentido plantearse esta cuestión. Si un programa **P** transforma cualquier dato **x** en la palabra **abbca** no nos podemos plantear invertir el proceso, ya que a partir del resultado es imposible deducir de qué dato particular **x** se ha originado (se dice que la actuación del programa **P** hace perder información). Esta situación se va a repetir siempre que el programa **P** compute una función no inyectiva, y no tendrá sentido invertir el comportamiento de **P** en estos casos.

Centremos por tanto la cuestión de otra manera. Supongamos que tenemos un programa **P** que computa una función inyectiva (es decir, que nunca produce resultados iguales para datos distintos). Entonces, ¿cuándo es posible encontrar otro programa **Q** que invierta el proceso de **P**, es decir, que recupere el dato de **P** a partir de su resultado? La respuesta es que siempre: en estos casos, todo lo que un programa hace, otro lo puede deshacer.

TEOREMA (LEY DE LA FUNCIÓN INVERSA): Sea ψ: $\sum^* \longrightarrow \sum^*$ una función computable e inyectiva. Entonces ψ^{-1}: $\sum^* \longrightarrow \sum^*$ es también computable.

⌦ Si ψ es computable, existe un código e que cumple $\psi \cong \varphi_a$. Para computar $\psi^{-1}(\mathbf{y})$ calcularemos $\psi(0)$, $\psi(1)$, $\psi(2)$, $\psi(3)$, …, en busca de un valor **x** que satisfaga $\psi(\mathbf{x}) = \mathbf{y}$. Naturalmente haremos los cálculos en paralelo para evitar los riesgos asociados a los cómputos de la forma $\psi(\mathbf{x})$ que puedan resultar infinitos. Si

utilizamos las funciones de codificación para recorrer los posibles pares (**x**,**p**) de datos y números de pasos, el programa podría ser el siguiente:

```
PAR  := ε;
while  not  E (A, decod_2_1 (PAR), decod_2_2 (PAR), X1)  loop
       PAR :=  sig (PAR);
end loop;
X0:=  decod_2_1 (PAR);  ⌫
```

5. Diagonalización

Hemos obtenido importantes resultados en nuestro afán por demostrar la computabilidad de algunas funciones, como son el desarrollo de las macros y los tipos de datos que nos ayudan a simplificar la escritura de programas-while, la capacidad de realizar programas complejos como el de la función universal, la disponibilidad de técnicas avanzadas como la de intercalado, o el concepto de computabilidad no efectiva para las funciones cuyo programa exacto sea difícil de construir. Es decir, hemos imitado el comportamiento clásico de cualquier disciplina informática: hemos definido la herramienta de los programas-while y nos hemos dedicado a resolver problemas cada vez más complejos con ella. Cuanta más habilidad hemos desarrollado en su manejo, mayor utilidad hemos encontrado a la herramienta, hasta el punto que, consciente o inconscientemente, hemos sucumbido a una cierta ilusión de que, con un poco de imaginación, ningún problema de naturaleza práctica podrá resistirse a nuestros programas.

Sin embargo, hace muchos años que se demostró que existen problemas que no pueden ser resueltos mediante ningún método informático, por sofisticado que sea: son incomputables y ningún lenguaje de programación puede alterar ese hecho. Es más, por extraño que parezca lo normal en un problema (en el sentido de que es lo más frecuente) es que sea incomputable.

Ha llegado el momento de que nos enfrentemos a las funciones que no son computables. Lo primero que haremos será demostrar su existencia de forma genérica, para a continuación estudiar ejemplos concretos. Finalmente detallaremos el aspecto crucial de este informe: cómo se puede demostrar, utilizando un procedimiento razonablemente sistemático, que una función no es computable. A pesar de que las técnicas y demostraciones que veremos en este capítulo son esencialmente no constructivas, al final del mismo habremos adquirido la capacidad de determinar la incomputabilidad de una clase muy amplia de funciones.

5.1 El argumento de cardinalidad

Sabemos que existe una lista $\{\varphi_0, \varphi_1, \varphi_2, \varphi_3, \ldots\}$ de todas las funciones computables de un argumento. Ello tiene consecuencias sobre el número (transfinito) de funciones computables, que no resulta suficiente para abarcar la clase de todas las funciones existentes.

PROPOSICIÓN 1: Existen infinitas funciones incomputables.

⌦ Recordemos que hemos llamado $\mathcal{F}$ a la clase de todas las funciones que reciben y devuelven cadenas de símbolos, y $\boldsymbol{C}$ a la subclase formada por las que son computables (véase la sección 3.2). Nos vamos a concentrar en las clases $\mathcal{F}_1 \subset \mathcal{F}$ de las funciones con un solo argumento, y $\boldsymbol{C}_1 \subset \boldsymbol{C}$ de las que son computables unarias. Vamos a demostrar que $\boldsymbol{C}_1$ es una clase más pequeña que $\mathcal{F}_1$, comprobando que su cardinal es menor.

Primeramente veamos que $\boldsymbol{C}_1$ es una clase numerable. Hemos visto que, en virtud de la implementación del tipo $\mathbb{W}$, tenemos que $\boldsymbol{C}_1 = \{\varphi_0, \varphi_1, \varphi_2, \varphi_3, \ldots\}$. Por tanto existe una función total y sobreyectiva:

$$\mathbf{f}: \mathbb{N} \to \boldsymbol{C}_1$$

que relaciona cualquier número **i** con la función computable φ_i. Esta función **f** no es una biyección, ya que las funciones computables están repetidas en la lista (por ejemplo, **f**(0) = **f**(2), ya que los programas $\mathbf{P_0}$ (X0:= ε;) y $\mathbf{P_2}$ (X1:= ε;) computan ambos la función constante ε). Por ello tenemos que no hay más funciones computables que números naturales, es decir que $|\boldsymbol{C}_1| \leq |\mathbb{N}|$ (podría verse la igualdad, pero para nuestro razonamiento es irrelevante).

Ahora comprobemos que $\mathcal{F}_1$ es una clase no numerable. Para ello construiremos una función total y sobreyectiva:

$$\mathbf{g}: \mathcal{F}_1 \to [0,1]$$

en el intervalo real [0,1]. A cada función unaria ψ le corresponderá el número real $\mathbf{g}(\psi)$, que definido como el dígito en base 2 de la forma de $0,\mathbf{d_0d_1d_2d_3d_4}\ldots$, donde cada decimal se calcula de la siguiente manera:

$$\mathbf{d_i} = \begin{cases} 0 & \psi(\mathbf{w}_\mathbf{i})\uparrow \\ 1 & \text{c.c.} \end{cases}$$

Por ejemplo, si ψ es una función total se tiene que $\mathbf{g}(\psi)$=0,11111111…, es decir, el valor 1. Análogamente, si dom(χ) = {0} y dom(ξ) = {**x**: **x** mod 2 = 0} entonces $\mathbf{g}(\chi)$=0,1 y $\mathbf{g}(\xi)$=0,1010101…. En el primer caso tenemos la representación binaria de $^1/_2$, y en la segunda de $^2/_3$, ya que se verifica:

$$\frac{1}{2} + \frac{1}{8} + \frac{1}{32} + \frac{1}{128} + \ \ldots \ = \ \sum_{\mathbf{i}=0}^{\infty} \frac{1}{2 * 4^{\mathbf{i}}} \ = \ \frac{2}{3}$$

Es claro que cualquier número real **r** del intervalo [0,1] se puede poner en forma de secuencia de decimales binarios $0,\mathbf{d_0d_1d_2d_3d_4}\ldots$, y que dada tal secuencia existe una función ψ que satisface $\mathbf{g}(\psi) = 0$, $\mathbf{d_0d_1d_2d_3d_4}\ldots = \mathbf{r}$, por lo que la función **g** es sobreyectiva, con lo que $|[0,1]| \leq |\mathcal{F}_1|$ (de nuevo podríamos ver la igualdad,

pero para nuestro razonamiento es irrelevante). Pero es sabido que en el intervalo [0,1] hay tantos números reales como en $\mathbb{R}$, por lo que $|\mathbb{R}| \leq |\mathcal{F}_1|$.

Sin embargo, sabemos desde Cantor que el cardinal transfinito $|\mathbb{N}|$ es estrictamente menor que $|\mathbb{R}|$, por lo que:

$$|\mathcal{C}_1| \leq |\mathbb{N}| < |\mathbb{R}| \leq |\mathcal{F}_1|$$

Es decir, que dado que el cardinal de programas-while es numerable, el número de funciones computables también lo es. Sin embargo, el conjunto de funciones existentes no es numerable, por lo que es imposible computarlas todas: no hay programas suficientes. ⌫

De hecho, como $|\mathbb{R}| \geq 2^{|\mathbb{N}|}$, tenemos que la inmensa mayoría de las funciones existentes no es computable, siendo su número infinitamente mayor que el de las computables. Esto es válido además para cualquier lenguaje de programación que imaginemos: mientras los programas se expresen mediante caracteres de un alfabeto finito (o incluso infinito numerable), el conjunto total de programas disponible siempre será numerable, y por tanto insuficiente para abarcar la clase de todas las funciones posibles.

El problema es que la argumentación anterior, por irrefutable que resulte, no nos aporta nada sobre la naturaleza de la incomputabilidad: nos dice que hay muchas funciones incomputables, pero no dónde están ni qué pinta tienen. Sin embargo, desde el punto de vista práctico no nos interesa saber cuántas hay, ni limitarnos a conocer algunos ejemplos, sino tener algún método para determinar, dada una función, si es incomputable o no, porque si lo es debemos saberlo para evitar perder el tiempo intentando programarla.

> *Los primeros problemas incomputables que se encontraron lo fueron allá por la década de los años 30, es decir, cuando los ordenadores no eran siquiera una promesa de futuro. Aunque no se disponía aún de la tecnología para construirlos, se conocían muchas de las propiedades que los computadores digitales habrían de exhibir quince años más tarde. Así, se sabía que ningún ordenador, por muy potente que fuese, podría anticipar el comportamiento de los programas en ejecución, o decidir si dos programas dados son equivalentes (es decir, si producen los mismos resultados cuando operan bajo las mismas condiciones iniciales).*

A diferencia de lo que sucede cuando se pretende probar la computabilidad de un problema, para lo cual es suficiente con encontrar un algoritmo concreto que lo resuelva, al enfrentarnos a la tarea de demostrar su incomputabilidad necesitamos un argumento lógico que certifique la inexistencia de tal algoritmo, o lo que es lo mismo, que pruebe que ninguno de los algoritmos existentes es capaz de resolver

dicho problema. Tal argumento de carácter universal no suele ser sencillo de establecer, y normalmente está relacionado con una demostración por reducción al absurdo. Existen distintas técnicas para lograr este objetivo, siendo la de diagonalización, la de reducción y la del punto fijo las más importantes. Es la primera la única que trataremos en este capítulo.

De nuevo, la existencia de una enumeración $\mathcal{C}_1 = \{\varphi_0, \varphi_1, \varphi_2, \varphi_3, \ldots\}$ de todas las funciones computables será la piedra de toque sobre la que se asentarán las demostraciones de incomputabilidad, pero esta vez de manera más constructiva. Podremos probar que una función no es computable verificando que no se encuentra en dicha lista. Y también podremos definir algunas funciones incomputables construyéndolas de forma que sean distintas de todas las existentes en la lista.

5.2 El problema de parada

El **problema de parada** es el primer y más conocido ejemplo de problema no resoluble mediante técnicas algorítmicas, y como tal, capítulo obligado en cualquier libro de Teoría de la Computabilidad [Har 87, MAK 88, SW 88]. También nos proporciona un soporte intuitivo para anticipar la incomputabilidad de otros problemas relacionados.

En secciones anteriores vimos ejemplos de funciones que trabajaban sobre propiedades de los programas-while. Éstas podían ser propiedades estáticas, relativas a la sintaxis del programa-while (como *ult_variable*.), pero también podían ser propiedades dinámicas relacionadas con la semántica, la ejecución del programa-while (como la función universal). El predicado que expresa el problema de parada es una de esas últimas funciones, pues a partir de los valores de entrada **x** e **y** trata de saber si la ejecución del programa $\mathbf{P_x}$ sobre el dato **y** es convergente o no.

TEOREMA (INDECIDIBILIDAD DEL PROBLEMA DE PARADA): La función *halt*: $\sum^* \times \sum^* \rightarrow \mathbb{B}$, definida como sigue, no es computable

$$\text{halt}(\mathbf{x}, \mathbf{y}) = \begin{cases} \text{true} & \varphi_{\mathbf{x}}(\mathbf{y})\downarrow \\ \text{false} & \varphi_{\mathbf{x}}(\mathbf{y})\uparrow \end{cases}$$

La consecuencia de este teorema será la inexistencia de un *método general* (y por tanto ningún formalismo de programación) para decidir sobre la parada de los programas-while.

⌦ Haremos la demostración por reducción al absurdo. Supongamos que la función *halt* es computable. Entonces nos basamos en ella para construir la función $\delta : \sum^* \longrightarrow \sum^*$

$$\delta(\mathbf{x}) \cong \begin{cases} \varepsilon & \neg\text{halt}(\mathbf{x},\mathbf{x}) \\ \perp & \text{halt}(\mathbf{x},\mathbf{x}) \end{cases}$$

que resultará también computable, como lo demuestra el siguiente programa:

if halt(X1, X1) **then** X0 := ⇊ ; **else** X0 := ε; **end if**;

Por ser δ computable existe un índice[5] **e** tal que $\delta \cong \varphi_{\mathbf{e}}$. Podemos plantearnos calcular el valor de $\delta(\mathbf{e})$, que dada la naturaleza de δ puede ser convergente o divergente. Veamos ambas posibilidades:

Caso 1: $\delta(\mathbf{e})\downarrow \overset{\delta\cong\varphi_e}{\Rightarrow} \varphi_{\mathbf{e}}(\mathbf{e})\downarrow \overset{\text{def. halt}}{\Rightarrow} \text{halt}(\mathbf{e},\mathbf{e}) \overset{\text{def . }\delta}{\Rightarrow} \delta(\mathbf{e})\uparrow$ ↯

que es contradictorio, y por tanto imposible. Por otro lado

Caso 2: $\delta(\mathbf{e})\uparrow \overset{\delta\cong\varphi_e}{\Rightarrow} \varphi_{\mathbf{e}}(\mathbf{e})\uparrow \overset{\text{def. halt}}{\Rightarrow} \neg\text{halt}(\mathbf{e},\mathbf{e}) \overset{\text{def . }\delta}{\Rightarrow} \delta(\mathbf{e}) = \varepsilon \Rightarrow \delta(\mathbf{e})\downarrow$ ↯

que también es contradictorio. En definitiva, hemos llegado a la equivalencia $\delta(\mathbf{e})\downarrow \Leftrightarrow \delta(\mathbf{e})\uparrow$, lo que quiere decir que la función δ no puede ni convergir ni divergir cuando es aplicada sobre la entrada **e**. Esta contradicción nos hace concluir que la hipótesis ha de ser necesariamente falsa, por lo que la función *halt* no puede ser computable. ⌫

Como ya hemos indicado, este teorema (que es uno de los más importantes de toda la Informática Teórica) establece categóricamente que no existe ningún método general (es decir, que valga para todos los casos) para decidir cuándo las ejecuciones de los programas están destinadas a convergir y cuándo a divergir.

> *Este resultado no contradice el hecho de que en numerosas ocasiones seamos capaces de predecir si un programa va a terminar o no. Por ejemplo, no hace falta ser ningún lince para determinar que un programa sin bucles va a detenerse para cualquier entrada que se le suministre. Por otro lado, un somero análisis del programa:*
>
> **while** $\text{car}_{\mathbf{a}}$?(X1) **loop** X1:= $\text{cons}_{\mathbf{a}}$(X1); **end loop**;
>
> *nos permite descubrir que converge exactamente sobre las palabras que no empiezan por **a**. Lo que sucede es que el método que nos ha permitido averiguarlo no es generalizable para cualquier programa. De hecho*

[5] Un índice posible es la codificación del programa-while que obtendríamos al expandir el macro programa con el que hemos demostrado la computabilidad.

existen algunos programas relativamente sencillos para los cuáles no se sabe con certeza si convergen o no. Un ejemplo de ellos es el programa:

```
while X1<>1 loop
    if X1 mod 2 = 0 then  X1:= X1 / 2;
    else  X1:= 3*X1 + 1;
    end if;
end loop;
```

cuya secuencia de operaciones y resultado están relativamente claros (cuando converge siempre devuelve 1) pero que parece tener un comportamiento errático e imprevisible. Mientras para unas entradas (por ejemplo, las potencias de 2) converge muy rápidamente, para otras le cuesta dar unas cuantas vueltas (43 iteraciones para el 39, por ejemplo). Aunque se ha comprobado que termina para numerosos datos numéricos, no se ha podido demostrar que lo haga para todos, a pesar de que bastantes matemáticos competentes lo han intentado. El mejor antídoto para quienes no están muy convencidos de que el problema de parada sea realmente indecidible es que intenten predecir el comportamiento de este programa, averiguando para qué casos converge y para cuáles no.

5.3 Paradojas y diagonales

Reflexionemos ahora sobre el razonamiento de la demostración de indecidibilidad del problema de parada. El esquema de la misma parece relativamente directo. Primero suponemos que *halt* es computable. Después definimos δ, cuya computabilidad se deduce de la de *halt*. Por último, encontramos un comportamiento contradictorio en δ al aplicarla sobre su propio índice **e**, y concluimos que nuestra suposición inicial era falsa.

Parece claro que el elemento central de la demostración es la misteriosa función δ cuya construcción nos conduce directamente a una contradicción. ¿Qué tiene de contradictorio dicha función? Para verlo conviene que hagamos algunas consideraciones. Sabemos que los programas-while pueden tener comportamiento convergente o divergente, y que este comportamiento puede variar según el dato que se le suministre como entrada. Ahora bien, cuando este dato es a su vez interpretado como un programa-while, tiene sentido decir que los programas convergen o divergen sobre otros programas. Por ejemplo, el programa **P**:

```
X0 := ⇊;
```

diverge sobre cualquier programa que se le suministre como entrada, mientras que el programa **Q**:

```
X0 := haz_composición (X1, X1);
```

converge sobre cualquier programa que se le suministre como entrada. Un comportamiento algo más complejo es el presentado por el programa **R**, que analiza el programa-while que se le suministra como dato intentando encontrar algun subprograma iterativo. Si lo encuentra se detiene, devolviendo dicho subprograma, y en caso contrario cicla:

```
PILA := <];
PILA := empilar (X1, PILA);
ENCONTRADO := false;
-- ENCONTRADO indica la detección de un while_loop en la estructura del dato
while not P_vacía? (PILA) and not ENCONTRADO loop
      PROG := cima (PILA);
      PILA := desempilar (PILA);
      if composición? (PROG) then
            PILA := empilar (inst_int_2 (PROG), PILA);
            PILA := empilar (inst_int (PROG), PILA);
      elsif condición? (PROG) then
            PILA := empilar (inst_int (PROG), PILA);
      elsif iteración? (PROG) then
            ENCONTRADO := true;
            X0 := PROG;
      end if;
end loop;
if not ENCONTRADO then X0 := ⊥;
end if;
```

Un último ejemplo sería el representado por el programa **S**, que estudia el comportamiento del programa-while que se le suministre como dato sobre la entrada 12. Sólo se detiene si el resultado es un 5, divergiendo en cualquier otro caso:

```
R := Φ(X1, 12);
if R=5 then
      X0 := 0;
else X0 := ⊥;
end if;
```

Dado que estos programas se detienen o no según el dato (a su vez un programa-while) que se les suministre, cabe preguntarse ¿qué sucede cuando el dato suministrado es *el mismo programa*? O, dicho de otra manera, ¿convergen estos programas sobre sí mismos? En el caso de **P** y **Q** la respuesta es sencilla: **P** no converge sobre nadie (y tampoco sobre su propio código, lógicamente) mientras que **Q** implementa una función total, por lo que también converge sobre sí mismo.

En este punto conviene hacer una precisión. Los datos que reciben **P** y **Q** son programas-while pertenecientes al tipo $\mathbb{W}$. Los macroprogramas no han sido

implementados y no pueden, por tanto, ser representados mediante palabras ni funcionar como datos, así que en rigor **P** no puede ni convergir ni divergir sobre sí mismo. Por tanto, cuando decimos "**P** diverge sobre sí mismo" queremos decir en realidad que "el programa-while que resulta de la expansión de **P** diverge sobre sí mismo".

En el caso de **R** tenemos que también converge sobre sí mismo, ya **R** que contiene instrucciones de tipo while_loop que su expansión conservará. El caso de **S** es sólo un poco más rebuscado. Dado que **S** sólo puede convergir o devolver 0, está claro que nunca podrá producir el resultado 5, ni sobre la entrada 12 ni sobre ninguna otra. Por tanto **S** diverge sobre sí mismo.

La enseñanza que extraemos es que podemos clasificar los programas en dos tipos: los que convergen sobre sí mismos y los que no. Ahora bien, volviendo a la función δ utilizada en la demostración de la sección anterior, si la examinamos con detenimiento observamos que parece basar su funcionamiento en esa curiosa clasificación, ya que converge exclusivamente cuando se aplica sobre programas que no convergen sobre sí mismos:

$$\delta(\mathbf{x}) \cong \begin{cases} \varepsilon & \varphi_{\mathbf{x}}(\mathbf{x})\uparrow \\ \\ \perp & \varphi_{\mathbf{x}}(\mathbf{x})\downarrow \end{cases}$$

Es decir, que δ convergirá sobre programas como **Q** o **S**, y divergirá sobre otros como **P** o **R**. Pero como el razonamiento que seguimos nos lleva a demostrar la computabilidad de δ, se produce un importante salto cualitativo: podemos ver a δ no sólo como una función más o menos excéntrica, sino bajo el aspecto de un programa $\mathbf{P_e}$, un programa-while que, como hemos dicho, "converge exclusivamente cuando se aplica sobre programas que no convergen sobre sí mismos". Parece legítima la curiosidad que nos impulsa a plantearnos ¿qué sucede cuando $\mathbf{P_e}$ se aplica sobre sí mismo, es decir, sobre su propio código **e**?

Por un lado podría suceder que $\mathbf{P_e}$ fuera de los programas que convergen sobre sí mismos (como **P** o **R**). Pero entonces es uno de las entradas sobre las que $\mathbf{P_e}$ debe divergir. Así que si $\mathbf{P_e}$ se detiene frente a la entrada **e**, entonces está obligado a ciclar sobre **e**.

Pero la otra alternativa tampoco es posible. Si $\mathbf{P_e}$ fuera de los programas que divergen sobre sí mismos (como **Q** ó **S**), entonces sería precisamente una de las entradas sobre las que $\mathbf{P_e}$ debería convergir. Así que si $\mathbf{P_e}$ cicla frente a la entrada **e**, entonces está obligado a detenerse sobre **e**.

Por tanto $\mathbf{P_e}$ no puede ni convergir ni divergir sobre sí mismo, porque haga lo que haga está obligado a hacer justamente lo contrario. Estamos ante una paradoja

sin salida. ¿Cómo nos hemos metido en ella? Porque nos ha surgido un programa demente $\mathbf{P_e}$ que no puede existir por su naturaleza contradictoria. ¿Y de dónde ha surgido semejante monstruo? Como consecuencia directa de la computabilidad de δ ¿Y cómo hemos llegado a deducir tal computabilidad? Por haber supuesto que *halt* era computable. La única forma de salir de la paradoja es abandonar la suposición que nos ha llevado a ella: la hipótesis de computabilidad de *halt*, que es la única aserción no fundamentada de nuestro razonamiento.

> *Esta forma de construir paradojas no es privativa de la Teoría de la Computabilidad. En [Gar 83] podemos encontrar otras muy similares a ésta, como la del barbero, formulada por Bertrand Russell. Esta paradoja examina las consecuencias de plantear la existencia de un barbero que afeite a todas las personas que no se afeitan a sí mismas, y sólo a ésas. El barbero se contradice cuando debe decidir qué hacer consigo mismo. Si decide afeitarse no estaría cumpliendo la condición "y sólo a ésas", y si no se afeita, no cumpliría "afeita a todas las personas que no se afeitan a sí mismas". Esta paradoja tiene una importancia capital en la formulación de la Teoría de Conjuntos, realizada a principios del siglo pasado.*
> *La construcción de todas estas paradojas consiste en, dado un conjunto, definir un elemento del mismo que sea capaz de discriminar al resto de los elementos en dos subconjuntos dependiendo de cierta característica, así como de decidir su propio comportamiento en función de ello. Dicho comportamiento deberá estar en contraposición a la característica que se cumpla en cada caso. La paradoja, surgirá cuando deba decidir en cuál de los subconjuntos debe colocarse él mismo. Cualquiera que sea el subconjunto elegido habrá una contradicción con el comportamiento que debe mostrar.*

La función δ que hemos definido es bastante particular. Su construcción se caracteriza por diferenciarse de todas las funciones computables φ_0, φ_1, φ_2, φ_3, … En efecto, dada cualquier función computable $\varphi_{\mathbf{x}}$ tenemos dos posibilidades:

a) Puede suceder que $\varphi_{\mathbf{x}}(\mathbf{x})\downarrow$. En ese caso, δ está construida de forma que $\delta(\mathbf{x})\uparrow$

b) Puede suceder que $\varphi_{\mathbf{x}}(\mathbf{x})\uparrow$. En ese caso, δ está construida de forma que $\delta(\mathbf{x})\downarrow$

Es decir, que δ es necesariamente diferente de cualquier función computable $\varphi_{\mathbf{x}}$ precisamente en el punto **x** (véase la figura 5.1). Por tanto, δ no puede estar en la lista $\boldsymbol{C}_1 = \{\varphi_0, \varphi_1, \varphi_2, \varphi_3, \ldots\}$, y consiguientemente debe ser incomputable.

El mecanismo utilizado para construir δ se denomina ***diagonalización***, y fue utilizado por primera vez por Georg Cantor (entre otras cosas, para demostrar que $|\mathbb{N}| < |\mathbb{R}|$, resultado que casualmente hemos utilizado en la sección 5.1). Este método es aplicable cuando tengamos una lista infinita de objetos (en nuestro caso, las funciones computables) cada uno de los cuales se define a su vez en forma de

lista infinita de subobjetos (en nuestro caso, los comportamientos que cada función tiene sobre cada uno de los posibles datos). En esas circunstancias es posible construir un nuevo objeto diferente de todos los de la lista.

δ	**x**	φ_0	φ_1	φ_2	**...**	φ_x
↑	**0**	↓				
↓	**1**		↑			
↑	**2**			↓		
...	**...**				**...**	
↑	**x**					↓

Figura 5.1: La función δ es diferente a todas y cada una de las funciones computables porque el comportamiento de δ sobre cada punto **x** es precisamente el contrario al de φ_x sobre ese mismo valor. En esta tabla-ejemplo[6] se observa esa situación.

De todas formas, lo importante de la función δ que hemos construido no es que sea incomputable. Lo esencial es que su naturaleza es contradictoria, y que su contradicción arrastra consigo la hipótesis de computabilidad de *halt*. ¿A qué se debe esta naturaleza contradictoria? Hemos visto que δ es cuidadosamente construida para que sea incomputable. Pero, por otro lado, hemos construido un programa para esta función, programa en el que *halt* tiene un papel muy destacado. Por tanto, δ participa de una doble naturaleza:

- Por un lado es incomputable (distinta de todas las funciones computables)
- Por otro lado es computable (su computabilidad se deduce de la presunta computabilidad de *halt*).

La conjunción de estos dos factores es la que provoca la contradicción que nos lleva a donde queríamos. Vamos a recapitular los distintos hitos que hemos recorrido en este proceso:

[6] En las tablas-ejemplo que utilizaremos en esta sección no nos preocuparemos de que los datos proporcionados sean fidedignos, sino de que ilustren las distintas posibilidades a examinar. Por ejemplo, sabemos que no es cierto que $\varphi_1(1)\uparrow$ (ya que P1 ≡ X0:=cons$_a$(X0);). Sin embargo nos interesa examinar un caso en el que $\varphi_x(x)\uparrow$, y no es razonable extender la tabla hasta el primer programa real en que esto ocurre (si trabajáramos con un alfabeto de dos elementos sería P36 ≡ **while** nonem?(X1) **loop** X0:=ε; **end loop**;). La solución es "olvidar" el comportamiento real de los programas y atribuir a estos de manera arbitraria los diferentes casos para que sirvan de ejemplo.

1. Hemos supuesto que *halt* es computable.
2. Según lo anterior, siempre que lo necesitemos podremos preguntar si es cierto *halt*(**x**,**y**). Basándonos en la información que *halt* proporciona sobre cualquier función computable, hemos construido una nueva función δ.
3. Esta construcción se basa en definir δ de forma que se diferencie de cada función computable φ_x en un punto concreto (en nuestro caso el propio **x**, pero no necesariamente).
4. Demostramos que δ es computable, y además su computabilidad depende de la de *halt*.
5. Dado que δ es computable, le asignamos un índice **e**.
6. Buscamos el punto concreto en que δ se tiene que diferenciar de φ_e.
7. Aplicamos δ sobre dicho punto y concluimos que δ tiene que ser distinta de sí misma, lo cual constituye una contradicción.

5.4 La técnica de diagonalización

El método de diagonalización que hemos seguido para demostrar la indecidibilidad del problema de parada es fácilmente generalizable para poder aplicarse a otras demostraciones de incomputabilidad. Dada una función **f** cuya incomputabilidad se desea demostrar, el método consiste en construir otra función δ, llamada ***función diagonal***, de naturaleza contradictoria: por un lado será computable bajo la suposición de que **f** también lo sea, pero por otro será distinta de todas las funciones computables, porque así la construiremos cuidadosamente. Ahora bien, antes de lanzarnos a la aplicación de los aspectos más técnicos del método, es necesario estudiar cuidadosamente el problema y las fases a seguir en su resolución. Lo mejor es hacerlo de la forma más metódica posible para identificar los pasos más difíciles y evitar tomar caminos sin salida.

A continuación detallaremos las cuatro fases de esta técnica, y las ilustraremos realizando al mismo tiempo un ejemplo de la utilización del método para demostrar la incomputabilidad de la siguiente función:

$$\mathbf{f}(\mathbf{x},\mathbf{y}) = \begin{cases} \text{true} & \varphi_x(\mathbf{y}) = '\,\mathbf{aba}' \\ \text{false} & \text{c.c.} \end{cases}$$

Es la función que determina si un programa $\mathbf{P_x}$ cualquiera sobre un dato **y** cualquiera produce como resultado la palabra **aba**.

FASE 1 – JUSTIFICACIÓN. Tenemos que plantearnos de forma intuitiva si hay razones para fundamentar la sospecha de que **f** no es computable.

Es importante que tengamos alguna pista sobre la naturaleza de **f** antes de intentar nada. Téngase en cuenta que las demostraciones de computabilidad (por ejemplo, mediante el diseño de un programa-while) y las de incomputabilidad no guardan ninguna conexión entre sí. Por ello conviene acertar en el camino a tomar, porque en caso contrario todo el trabajo realizado será inútil. Si **f** al final resultara computable, todos los esfuerzos que hayamos hecho para aplicar la técnica de diagonalización fracasarán, y además no nos darán ninguna pista[7] de por dónde puede ir el programa que compute **f**, con lo que estaremos como al principio, pero más cansados.

Será la experiencia con anteriores funciones incomputables la que nos pueda ayudar, por analogía, a intuir por dónde pueden venir los problemas de incomputabilidad de **f**, y muy especialmente su posible relación con el problema de parada.

EJEMPLO: Aunque explicite dos casos, en realidad **f** oculta un tercero

- Si $\varphi_x(\mathbf{y})=\mathbf{aba}$, podríamos comprobarlo ejecutando el programa $\mathbf{P_x}$ sobre **y**, y a su terminación sabríamos que $\mathbf{f}(\mathbf{x},\mathbf{y})$=true.
- Si $\varphi_x(\mathbf{y})\neq\mathbf{aba}$ (es decir, si $\mathbf{P_x}$ produce sobre **y** un resultado que no es **aba**), también podríamos comprobarlo ejecutando el programa, y a su terminación sabríamos que $\mathbf{f}(\mathbf{x},\mathbf{y})$=false.
- El problema sobreviene cuando $\varphi_x(\mathbf{y})\uparrow$. En este caso la ejecución del programa $\mathbf{P_x}$ sobre **y** no nos indica nada, ya que no podemos anticipar (puesto que *halt* es indecidible) si el programa no termina porque no se le ha dado suficiente tiempo o porque cicla indefinidamente. En este caso $\mathbf{f}(\mathbf{x},\mathbf{y})$=false puesto que no es cierto $\varphi_x(\mathbf{y})=\mathbf{aba}$, pero no parece haber un método para determinarlo.

La conclusión es que la función no parece computable, pero ¡cuidado! la argumentación que hemos presentado no constituye una prueba (si lo fuera nos sobraría el resto del método). Un razonamiento como el que hemos hecho podría ser fraudulento: decimos que $\mathbf{f}(\mathbf{x},\mathbf{y})$ no parece computable porque "tendríamos que

[7] Esta situación no tiene por qué ser la más habitual en Lógica o Matemáticas. Muchas veces, al intentar demostrar que es cierta una propiedad que no lo es, nuestros intentos fallidos nos proporcionan alguna pista de por qué no se cumple la propiedad, e incluso de cómo podremos demostrar que es falsa.

decidir los casos en los que $\varphi_x(\mathbf{y})\uparrow$, y ello no es posible". Sin embargo, ¿podemos asegurar que no existe ningún otro camino, ninguna forma alternativa de computar **f** sin tener que resolver dicho problema insoluble? La única forma de obtener esa certeza es completar la demostración formalmente.

> **FASE 2 – PLANIFICACIÓN.** Hemos de estudiar en detalle la información que nos proporciona la función **f**. Después, apoyándonos en esta información planificaremos la construcción de la función diagonal δ, que procuraremos sea diferente de todas las funciones computables φ_x, y cuya computabilidad se deducirá de la de **f**.

La esencia del argumento de diagonalización es suponer que **f** es computable y demostrar que ello nos lleva a la construcción de un objeto imposible: la función δ que será a la vez incomputable y computable. Construir una función diagonal que sea distinta de todas las computables es muy fácil. Lo complicado es construir una δ íntimamente relacionada con **f**, de forma que la computabilidad de δ se pueda deducir de la de **f**, ya que así la naturaleza contradictoria de δ arrastrará a **f** consigo.

Por ello es imprescindible plantearse que **f** ha de ser la herramienta o subrutina principal que hemos de usar para construir δ: debemos entender en profundidad qué posibilidades nos ofrece esa herramienta antes de abordar la construcción de la función diagonal. Debemos evitar incluir en la definición de δ ningún otro elemento sospechoso de incomputabilidad, porque en caso contrario, cuando comprobemos que la computabilidad de δ no se sostiene, no podremos echarle la culpa a **f**.

Para analizar la información que **f** nos proporciona sobre las distintas funciones computables φ_x y poder usarla después para conseguir que δ se diferencie de todas ellas utilizaremos esquemas como el sugerido en la figura 5.1. Dado que vamos a tener información de muchas funciones será útil organizarla en una ***tabla-ejemplo***. Las cabeceras de las columnas de esta tabla infinita están ocupadas por las funciones computables φ_0, φ_1, φ_2, φ_3, φ_4, … . Las cabeceras de las filas indican los posibles valores de entrada[8] para dichas funciones 0**,** 1**,** 2**,** 3**, ...**. Por tanto, cada celda de la tabla corresponderá al resultado de la función φ_i sobre el dato **j**, y en la misma trataremos de expresar qué es lo que podemos averiguar sobre $\varphi_i(\mathbf{j})$ gracias a la ayuda de la función **f**. Cuando tengamos una idea clara de qué es lo que **f** nos proporciona estaremos en condiciones de usar dicha función en la construcción de la función diagonal δ.

[8] Por razones de espacio y facilidad de uso resulta mucho más cómodo indexar las posibles palabras de entrada mediante los números naturales representados por dichas palabras, y a esa convención nos remitiremos.

Para que δ sea distinta de todas las que aparecen en la tabla debe diferenciarse de cada una de ellas, es decir, de cada columna, en al menos un punto. Este punto podría seleccionarse de distintas maneras, pero por simplicidad tomamos el de la ***diagonal*** siempre que sea posible. Trataremos pues de que δ se diferencie de φ_i en el punto **i**. Para ello miraremos en la tabla qué es lo que sabemos sobre $\varphi_i(\mathbf{i})$ (gracias a f) y definiremos $\delta(\mathbf{i})$ de forma que sea incompatible con ello.

EJEMPLO: Llevándolo a la práctica en nuestro ejemplo ¿Qué información nos proporciona **f**? Para cada función φ_i y para cada dato **j**, si $\varphi_i(\mathbf{j})$ es **aba** o no. En este último caso, no nos dice si es porque $\varphi_i(\mathbf{j})$ es otro valor, o porque está indefinida. La figura 5.2 muestra la tabla-ejemplo para este caso.

	φ_0	φ_1	φ_2	…	φ_i
0	**=aba**	**≠aba** | ⊥	**=aba**		**=aba**
1	**=aba**	**=aba**	**=aba**		**≠aba** | ⊥
2	**=aba**	**≠aba** | ⊥	**≠aba** | ⊥		**=aba**
…				**…**	
j	**=aba**	**≠aba** | ⊥	**≠aba** | ⊥		**≠aba** | ⊥

Figura 5.2: Tabla-ejemplo para reflejar la información que **f** nos proporciona sobre las distintas funciones computables. En los casos en los que $\mathbf{f(i,j)}$, descubrimos que $\varphi_i(\mathbf{j})$ es **aba**. Por el contrario, cuando $\neg\mathbf{f(i,j)}$ averiguamos que, o bien $\varphi_i(\mathbf{j})$ es un valor distinto de **aba**, o bien es indefinido.

En el presente ejemplo, al igual que en el del problema de parada, tenemos información de lo que ocurre en cada punto de cada función. En otros casos que veremos en ejemplos posteriores, esta se limitará a algunos puntos, pero la tabla nos ayudará a ver en cuáles.

Por tanto, la información proporcionada en todos los puntos es la presencia o no de la cadena **aba** como resultado. Podemos elegir los valores de la diagonal para que la función δ haga lo contrario que cada una de las φ_i en el punto **i**. Así, $\delta(\mathbf{i})$ será **aba** cuando $\varphi_i(\mathbf{i})$ no lo sea, y será cualquier cosa diferente de **aba** (por ejemplo, **bb**) cuando averigüemos, siempre gracias a **f**, que $\varphi_i(\mathbf{i})$ es precisamente **aba**.

Para comprender mejor la definición y como resumen del proceso, añadimos a la tabla anterior los valores previstos para la nueva función δ en forma de columna, tal como se muestra en la figura 5.3.

Es obvio que la función δ es diferente de cada una de las que hay en la tabla al menos en el punto correspondiente a la diagonal, tal como nos proponíamos:

- δ es distinta de la función φ_0 en el punto 0, ya que $\varphi_0(0)$=**aba** y $\delta(0)$=**bb**
- δ es distinta de la función φ_1 en el punto 1, ya que $\varphi_1(1)$=**aba** y $\delta(1)$=**bb**
- δ es distinta de la función φ_2 en el punto 2, ya que $\neg(\varphi_2(2)$=**aba**$)$ y $\delta(2)$=**aba**
- …
- En general, δ es distinta de la función φ_i en el punto **i**, ya que o bien φ_i(**i**)=**aba** y δ(**i**)=**bb**, o bien $\neg(\varphi_i$(**i**)=**aba**$)$ y δ(**i**)=**aba**

δ	**x**	φ_0	φ_1	φ_2	**…**	φ_i
bb	**0**	**=aba**	**≠aba** | ⊥	**=aba**		**=aba**
bb	**1**	**=aba**	**=aba**	**=aba**		**≠aba** | ⊥
aba	**2**	**=aba**	**≠aba** | ⊥	**≠aba** | ⊥		**=aba**
…	**…**				**…**	
aba	**x**	**=aba**	**≠aba** | ⊥	**≠aba** | ⊥		**≠aba** | ⊥

Figura 5.3: Plan de construcción de la función diagonal δ, cuyo comportamiento sobre cada punto **i** es precisamente el contrario al de φ_i sobre ese mismo valor. Se han sombreado los puntos que se han tenido en cuenta para definir δ, en este caso los de la diagonal.

En definitiva, hemos conseguido aprovechar las características específicas de la función **f** para diseñar otra función δ diferente de todas las computables. Hemos tenido especial cuidado de no utilizar ninguna operación sospechosa (aparte, claro está, de la propia **f**) en el diseño de δ. Ahora deberemos implementarla.

> **FASE 3 – CONSTRUCCIÓN.** Es el momento de establecer los elementos de la demostración propiamente dicha. Tras suponer la computabilidad de **f** definiremos formalmente la función δ, y demostraremos su computabilidad. Siempre deberemos asegurarnos de que la computabilidad de δ se basa en la de **f**. Concluiremos que $\delta \equiv \varphi_e$ para algún índice **e**.

En la mayor parte de los casos la definición de δ se sigue directamente de la tabla, y el programa que la computa no plantea grandes desafíos. En algunos ejercicios más rebuscados tanto la definición como el diseño del programa pueden requerir una atención especial, por ejemplo cuando δ es definida recursivamente.

EJEMPLO: De acuerdo con el plan, para nuestro caso definimos la función δ como sigue:

$$\delta(\mathbf{x}) \cong \begin{cases} \mathbf{bb} & \mathbf{f(x,x)} \\ \mathbf{aba} & \neg\mathbf{f(x,x)} \end{cases}$$

Suponiendo que **f** es computable, podemos demostrar la computabilidad de δ mediante el programa

if f(X1, X1) **then** X0 := '**bb**'; **else** X0 := '**aba**'; **end if**;

Dado que δ es una función computable, tenemos que $\delta \cong \varphi_{\mathbf{e}}$. Por tanto, la función δ debería corresponderse exactamente con la **e**-ésima columna de la tabla, al ser un elemento de la lista de funciones computables.

> **FASE 4 – CONTRADICCIÓN.** Debemos encontrar la contradicción en el comportamiento de δ. Dado que es diferente de todas las funciones computables al menos en un punto, la aplicaremos sobre el valor concreto que la diferencia de $\varphi_{\mathbf{e}}$, es decir, de sí misma (normalmente el propio **e** de la diagonal u otro relacionado con **e**.).

Si δ ha sido construida correctamente no debe ser difícil llegar a una contradicción. Hemos demostrado que es computable y, por tanto, $\delta \cong \varphi_{\mathbf{e}}$, pero hemos construido específicamente δ para conseguir que se diferencie de $\varphi_{\mathbf{e}}$ en un punto concreto (pongamos que es **e**). Es decir, que δ debe ser distinta de sí misma en el punto **e**, lo cual es absurdo. Estudiaremos las posibles alternativas para $\delta(\mathbf{e})$ y observaremos que en todos los casos posibles llegamos a una contradicción.

Si no se ha definido con cuidado puede darse el caso de que sea complicado examinar todos sus casos (por ejemplo, porque se solapan y dan lugar a condiciones lógicas complejas). En ese caso suele ser una buena alternativa utilizar los casos de la propia **f** para buscar la contradicción.

EJEMPLO: Volviendo a nuestra función δ, tenemos que ha sido diseñada para diferenciarse de cada $\varphi_{\mathbf{x}}$ en el punto **x**. Estudiamos, por tanto, que sucede en el punto **e** y vemos que tenemos dos posibilidades para $\delta(\mathbf{e})$ de acuerdo con la definición de δ: o bien vale **bb**, o bien vale **aba** (no hay otra posibilidad).

Caso 1: $\delta(\mathbf{e})=\mathbf{bb} \overset{\text{def. }\delta}{\Rightarrow} \mathbf{f(e,e)} \overset{\text{def. }\mathbf{f}}{\Rightarrow} \varphi_{\mathbf{e}}(\mathbf{e})=\mathbf{aba} \overset{\delta \cong \varphi_{\mathbf{e}}}{\Rightarrow} \delta(\mathbf{e})=\mathbf{aba}$ ↯

Caso 2: $\delta(\mathbf{e})=\mathbf{aba} \overset{\text{def. }\delta}{\Rightarrow} \neg\mathbf{f(e,e)} \overset{\text{def. }\mathbf{f}}{\Rightarrow} \neg(\varphi_{\mathbf{e}}(\mathbf{e})=\mathbf{aba}) \overset{\delta \cong \varphi_{\mathbf{e}}}{\Rightarrow} \neg(\delta(\mathbf{e})=\mathbf{aba})$ ↯

Ambos casos conducen a una contradicción, por lo que $\delta(\mathbf{e})$ no puede ni ser **aba** ni dejar de serlo. La contradicción tiene que proceder de la única hipótesis no probada que hemos hecho, que es suponer que **f** era computable.

En este caso ha sido sencillo analizar los casos de $\delta(\mathbf{e})$. Como alternativa podríamos haber contemplado los inducidos por la propia función **f**, que corresponden a los valores *true* y *false*:

$$\underline{\text{Caso 1:}}\ \mathbf{f}(\mathbf{e},\mathbf{e}) \overset{\text{def. } \mathbf{f}}{\Rightarrow} \varphi_{\mathbf{e}}(\mathbf{e})=\mathbf{aba} \overset{\delta\cong\varphi_e}{\Rightarrow} \delta(\mathbf{e})=\mathbf{aba} \overset{\text{def. } \delta}{\Rightarrow} \neg\mathbf{f}(\mathbf{e},\mathbf{e}) \ \lightning$$

$$\underline{\text{Caso 2:}}\ \neg\mathbf{f}(\mathbf{e},\mathbf{e}) \overset{\text{def. } \mathbf{f}}{\Rightarrow} \neg(\varphi_{\mathbf{e}}(\mathbf{e})=\mathbf{aba}) \overset{\delta\cong\varphi_e}{\Rightarrow} \neg(\delta(\mathbf{e})=\mathbf{aba}) \overset{\text{def. } \delta}{\Rightarrow} \delta(\mathbf{e})=\mathbf{bb} \overset{\text{def. } \delta}{\Rightarrow} \mathbf{f}(\mathbf{e},\mathbf{e}) \ \lightning$$

que también nos hubieran llevado a una contradicción.

5.5 Variantes de la técnica de diagonalización

El método de la diagonalización para la demostración de la no computabilidad de funciones puede aplicarse en muchos otros casos de forma análoga a como se ha explicado en la sección anterior. Sin embargo, en otros no es tan sencillo aplicar una interpretación tan directa del método, fundamentalmente porque podemos tener dificultades a la hora de aprovechar la información que nos proporciona **f** para construir δ. A continuación veremos algunos ejemplos en los que surgen obstáculos en el sentido apuntado, y comprobaremos cómo podemos adaptar el método para que siga siendo aplicable en los mismos. En [IIS 00] se presenta una colección con más ejemplos resueltos.

5.5.1 Desplazamiento de la Diagonal

Puede suceder que **f** no nos proporcione información sobre los valores $\varphi_{\mathbf{i}}(\mathbf{i})$ de la diagonal. Si no tenemos datos sobre lo que sucede en cada $\varphi_{\mathbf{i}}(\mathbf{i})$ no podremos diseñar δ para que se diferencie de $\varphi_{\mathbf{i}}$ en el punto **i**. Sin embargo, podemos mantener el espíritu del método si encontramos otro posible punto de diferenciación entre δ y cada función computable, aunque no sea el de la diagonal. Para ilustrar esta idea probaremos que la siguiente función no es computable:

$$\mathbf{f}(\mathbf{x}) = \begin{cases} 0 & \varphi_{\mathbf{x}}(\mathbf{x}+10)=\mathbf{x}^2 \\ \mathbf{x}+1 & \text{c.c.} \end{cases}$$

JUSTIFICACIÓN: Intuitivamente podemos pensar que su posible incomputabilidad reside en que para el caso en que $\varphi_{\mathbf{x}}(\mathbf{x}+10)\!\uparrow$ no podremos dar la respuesta requerida $\mathbf{x}+1$, en razón de la indecidibilidad del problema de parada. Los otros dos casos ($\varphi_{\mathbf{x}}(\mathbf{x}+10)=\mathbf{x}^2$ y $\varphi_{\mathbf{x}}(\mathbf{x}+10)\neq\mathbf{x}^2$) no parecen problemáticos *per se*.

PLANIFICACIÓN: La función **f** nos proporciona información sobre lo que ocurre en determinado punto de cada función. Concretamente, para cada $\varphi_{\mathbf{i}}$ nos dice si el

valor que devuelve en el punto **i**+10 es **i**^{2} ó no. Esta distinción no la hace mediante valores lógicos como en los ejemplos precedentes, sino utilizando valores numéricos distinguibles: estaremos en el primer caso cuando **f**(**i**)=0 y en el segundo cuando **f**(**i**)>0. En la tabla-ejemplo de la figura 5.4 incorporamos los hipotéticos valores de **f**(**i**) para visualizar los datos que esa función nos aporta.

Obsérvese que no tenemos información de todos los puntos de la tabla, sino solamente de uno por columna. La función δ que debemos construir debe diferenciarse de cada una de las que aparecen en la tabla, pero como debemos obligatoriamente apoyarnos en **f** para producir esta distinción, los puntos en los que nos basaremos habrán de ser aquellos sobre los que **f** nos suministra alguna información. Como se aprecia en la tabla obtenemos una *diagonal desplazada*.

		f(0)>0	**f**(1)=0	**f**(2)=0	**f**(3)>0	**f**(4)=0	**f**(5)=0
δ	**x**	φ_0	φ_1	φ_2	φ_3	φ_4	φ_5
	0						
	1						
	…						
0	**10**	$\perp \mid \neq 0$					
$\perp$	**11**		1				
$\perp$	**12**			4			
9	**13**				$\perp \mid \neq 9$		
$\perp$	**14**					16	
$\perp$	**15**						25

Figura 5.4: Tabla-ejemplo en la que se observa que no disponemos de información sobre los valores de la diagonal. Sin embargo los valores sombreados permiten la construcción de la función δ a partir de la información obtenida en una diagonal desplazada.

Teniendo en cuenta ese condicionamiento, procuraremos que δ sea diferente de φ_0 en el punto 10, de φ_1 en el punto 11, de φ_2 en el punto 12, y así sucesivamente. Por ejemplo, sabemos que $\varphi_0(10)$ no vale 0 (bien porque diverge o bien porque toma un valor distinto), por lo que nos viene al pelo definir $\delta(10)=0$. También sabemos que $\varphi_1(11)=1$, así que nos bastará con que $\delta(11)$ diverja (también podríamos haber decidido poner cualquier valor distinto de 1). En general, si la función de índice **i** en el punto **i**+10 devuelve **i**^{2}, la nueva función δ divergirá en ese punto, y en caso contrario δ(**i**+10) valdrá precisamente **i**^{2}. Naturalmente, para saber

qué es lo que ocurre en $\varphi_i(\mathbf{i}+10)$ bastará con ver si $\mathbf{f}(\mathbf{i})=0$ ó $\mathbf{f}(\mathbf{i})>0$. Nuestro plan queda reflejado en la tabla-ejemplo de la figura 5.4.

Sucede que los valores $\delta(0)$ ... $\delta(9)$ son indiferentes para el resultado del nuestro razonamiento, ya que con el resto nos apañamos para obtener la función diagonal. Para facilitar la definición podemos hacer que diverja en esos puntos.

Por tanto vamos a definir una función δ que utilizará cada punto $\mathbf{x}\geq 10$ para diferenciarse de la función computable $\varphi_{\mathbf{x}-10}$. Los puntos $\mathbf{x}<10$ los definimos de manera arbitraria:

- $\delta(0)$ ... $\delta(9)$ no tienen utilidad ninguna
- $\delta(10)$ permite diferenciar a δ de φ_0, ya que $\delta(10)=0^2$ y $\varphi_0(10)$ no
- $\delta(11)$ permite diferenciar a δ de φ_1, ya que $\varphi_1(11)=1^1$ y $\delta(11)$ no
- $\delta(12)$ permite diferenciar a δ de φ_2, ya que $\varphi_2(12)=2^2$ y $\delta(12)$ no
- …
- En general, $\delta(\mathbf{x})$ permite diferenciar a δ de $\varphi_{\mathbf{x}-10}$, ya que cuando $\varphi_{\mathbf{x}-10}(\mathbf{x})=(\mathbf{x}-10)^2$ entonces $\delta(\mathbf{x})$ no toma ese valor, y viceversa

CONSTRUCCIÓN: Supondremos entonces que **f** es computable. De acuerdo con el plan, definiremos formalmente la función δ:

$$\delta(\mathbf{x}) \cong \begin{cases} (\mathbf{x}-10)^2 & \mathbf{x}\geq 10 \ \wedge \ \mathbf{f}(\mathbf{x}-10)>0 \\ \\ \perp & \text{c.c.} \end{cases}$$

Cuya computabilidad resulta clara mediante el programa

if X1≥10 **and** f(X1-10)>0 **then** X0 := (X1-10)^2; **else** X0 := ⇊ ; **end if**;

Por lo tanto, al ser δ computable existe un índice **e**, tal que $\delta\cong\varphi_{\mathbf{e}}$.

CONTRADICCIÓN: Veamos que, tal y como hemos razonado en la construcción de δ, la existencia de ese índice es imposible porque δ se diferencia de cada función computable de índice **i** en el punto **i**+10. Comprobaremos que obtenemos una contradicción si aplicamos la función δ en el punto **e**+10. Tenemos dos posibles casos para $\delta(\mathbf{e}+10)$:

Caso 1: $\delta(\mathbf{e}+10)=\mathbf{e}^2 \overset{\text{def. }\delta}{\Rightarrow} \mathbf{e}+10\geq 10 \wedge \mathbf{f}(\mathbf{e})>0 \overset{\text{def. }\mathbf{f}}{\Rightarrow} \mathbf{f}(\mathbf{e})=\mathbf{e}+1 \overset{\delta\cong\varphi_{\mathbf{e}}}{\Rightarrow}$

$\neg(\varphi_{\mathbf{e}}(\mathbf{e}+10)=\mathbf{e}^2) \Rightarrow \neg(\delta(\mathbf{e}+10))=\mathbf{e}^2$ ↯

Caso 2: $\delta(\mathbf{e}+10)\uparrow \overset{\text{def. }\delta}{\Rightarrow} \mathbf{e}+10<10 \vee \mathbf{f}(\mathbf{e})=0 \Rightarrow \mathbf{f}(\mathbf{e})=0 \overset{\text{def. }\mathbf{f}}{\Rightarrow} \varphi_{\mathbf{e}}(\mathbf{e}+10)=\mathbf{e}^2 \overset{\delta\cong\varphi_{\mathbf{e}}}{\Rightarrow}$

$\Rightarrow \delta(\mathbf{e}+10)=\mathbf{e}^2$ ↯

Ninguno de ellos es en realidad posible porque nos llevan a una contradicción, que surge de la única hipótesis que hemos hecho: la de suponer que **f** es computable.

5.5.2 Deformación de la diagonal

Siguiendo con la misma idea de tener que lidiar con una **f** cuya incomputabilidad queremos demostrar, pero que no proporciona información sobre los valores $\varphi_i(\mathbf{i})$ de la diagonal, podemos encontrar otras variantes en las que estemos obligados a utilizar sustitutos más o menos sofisticados de la misma. Un ejemplo puede ser la demostración de que la siguiente función no es computable:

$$\mathbf{f}(\mathbf{x}) = \begin{cases} 8 & \varphi_x(2*\mathbf{x}) \bmod 3 = 0 \\ 5 & \text{c.c.} \end{cases}$$

JUSTIFICACIÓN: Podemos intuir que no es computable. Cuando la función de índice **x** converge en el punto $2*\mathbf{x}$ no hay ningún problema en determinar si el resultado es o no múltiplo de 3. Sin embargo, cuando $\varphi_e(2*\mathbf{x})\uparrow$ no podemos dar una respuesta satisfactoria, ya que la indecidibilidad del problema de parada nos impide anticipar este hecho.

PLANIFICACIÓN: La función **f** nos proporciona información sobre lo que ocurre en determinado punto de cada función. Según sea el resultado de $\mathbf{f}(\mathbf{x})$ sabremos si φ_e en el punto $2*\mathbf{x}$ produce un múltiplo de 3 ó no. En la tabla-ejemplo de la figura 5.5 se muestran sombreados estos puntos, y vemos que en este caso tenemos información de un valor por columna, pero que no coincide con el de la diagonal, ni siquiera desplazada. Sin embargo, esta diagonal deformada representa también una alternativa válida para la construcción de la función diagonal δ.

Para diferenciar la función δ de todas las funciones φ_i utilizaremos el punto del que tenemos información. Nos aseguraremos de que δ sea distinta de φ_0 en el punto 0, de φ_1 en el punto 2, φ_2 en el punto 4, y así sucesivamente. Como además sabemos que la función φ_0 en el punto 0 tiene un valor múltiplo de 3, haremos que la nueva función tome el valor 100 (o cualquier otro que no sea múltiplo de 3 ó incluso podría divergir). Como sabemos que la función φ_1 en el punto 2, no es múltiplo de 3 (bien porque diverge o bien porque toma un valor que no es de la forma $3*\mathbf{n}$) podemos hacer que la nueva función δ tome el valor 3 (o cualquier otro múltiplo de 3). Podemos generalizar el proceso y obtendríamos los puntos en los que δ se va a diferenciar de cada función φ_i. Pero conforme a la información que nos da **f** eso nos lleva a saber cómo definir δ únicamente para las entradas pares.

		f(0)=8	f(1)=5	f(2)=8	f(3)=8	**f**(4)=5
δ	**x**	φ_0	φ_1	φ_2	φ_3	φ_4
100	**0**	mod 3 = 0				
$\perp$	**1**					
3	**2**		$\perp$ | mod 3 ≠ 0			
$\perp$	**3**					
100	**4**			mod 3 = 0		
$\perp$	**5**					
100	**6**				mod 3 = 0	
$\perp$	**7**					
3	**8**					$\perp$ | mod 3 ≠ 0

Figura 5.5: Tabla-ejemplo en la que se observa que no disponemos de información sobre los puntos de la diagonal. Sin embargo los valores sombreados permiten la construcción de la función δ a partir de la información obtenida en una diagonal deformada.

Para que la definición de δ sea correcta también debemos especificar cómo se define para los puntos impares, aunque el valor concreto sea indiferente para el resultado del razonamiento. Parece más conveniente no utilizar ni 100 ni 3 para evitar solapamientos en la definición que puedan complicar el establecimiento de la contradicción. Podemos hacer que, por ejemplo, diverja en esos puntos.

- $\delta(1)$, $\delta(3)$, $\delta(5)$, $\delta(7)$, $\delta(9)$, ... no tienen utilidad ninguna
- $\delta(0)$ permite diferenciar a δ de φ_0, ya que $\delta(0)=100$ y $\varphi_0(0)$ es múltiplo de 3
- $\delta(2)$ permite diferenciar a δ de φ_1, ya que $\delta(2)=3$ y $\varphi_1(2)$ no
- $\delta(4)$ permite diferenciar a δ de φ_2, ya que $\delta(4)=100$ y $\varphi_2(4)$ es múltiplo de 3
- …
- En general, $\delta(2*\mathbf{x})$ permite diferenciar a δ de $\varphi_{\mathbf{x}}$, ya que cuando $\varphi_{\mathbf{x}}(2*\mathbf{x})$ es múltiplo de 3 entonces $\delta(\mathbf{x})$ es 100, y cuando $\varphi_{\mathbf{x}}(2*\mathbf{x})$ no es múltiplo de 3 entonces $\delta(\mathbf{x})$ es 3

CONSTRUCCIÓN: Definimos formalmente la función δ, que como hemos visto distingue las entradas pares de las impares y solamente para las entradas pares utilizamos la información de **f**:

$$\delta(\mathbf{x}) \cong \begin{cases} 100 & \mathbf{x} \bmod 2 = 0 \ \wedge\ f(\mathbf{x}/2) = 8 \\ 3 & \mathbf{x} \bmod 2 = 0 \ \wedge\ f(\mathbf{x}/2) = 5 \\ \perp & \mathbf{x} \bmod 2 \neq 0 \end{cases}$$

Una vez definida la función y suponiendo que **f** es computable, tenemos que la función δ es computable como lo demuestra el siguiente programa

```
X0 := 3;
if  X1 mod 2 = 0  then
        if  f(X1 div 2) = 8  then X0 := 100; end if;
else   X0 := ⇊ ;
end if;
```

Por tanto existe un índice **e** tal que $\delta \cong \varphi_{\mathbf{e}}$.

CONTRADICCIÓN: Veamos que, tal y como hemos razonado la construcción de δ, ese índice es imposible, porque obtenemos una contradicción si aplicamos la función δ en el punto $2*\mathbf{e}$.

Caso 1: $\delta(2*\mathbf{e})=100 \overset{\text{def. }\delta}{\Rightarrow} (2*\mathbf{e}) \bmod 2=0 \wedge \mathbf{f}(\mathbf{e})=8 \overset{\text{def. }\mathbf{f}}{\Rightarrow} \varphi_{\mathbf{e}}(2*\mathbf{e}) \bmod 3 = 0 \overset{\delta\cong\varphi_e}{\Rightarrow} \delta(2*\mathbf{e}) \bmod 3 = 0$ ↯

Caso 2: $\delta(2*\mathbf{e})=3 \overset{\text{def. }\delta}{\Rightarrow} (2*\mathbf{e}) \bmod 2=0 \wedge \mathbf{f}(\mathbf{e})=5 \overset{\text{def. }\mathbf{f}}{\Rightarrow} \neg(\varphi_{\mathbf{e}}(2*\mathbf{e}) \bmod 3 = 0) \overset{\delta\cong\varphi_e}{\Rightarrow} \neg(\delta(2*\mathbf{e}) \bmod 3 = 0)$ ↯

Caso 3: $\delta(2*\mathbf{e})\uparrow \overset{\text{def. }\delta}{\Rightarrow} (2*\mathbf{e}) \bmod 2\neq 0$ ↯

Por tanto $\delta(2*\mathbf{e})$ no puede ser ni 3 ni 100 (en los dos casos tenemos contradicción), pero tampoco puede divergir ya que, por definición de la función, esto ocurre solo para los impares. Por tanto la contradicción en el punto $2*\mathbf{e}$ surge de la única hipótesis que hemos hecho, la de suponer que **f** era computable.

5.5.3 Diagonalización asimétrica

Por último veremos un caso algo más peculiar. En ocasiones, la función **f** cuya incomputabilidad pretendemos demostrar nos da una información sesgada. Sobre algunas de las funciones computables nos proporciona abundantes datos que podemos aprovechar sin esfuerzo para construir la función diagonal. Sobre otras, en cambio, prácticamente no nos aporta nada, dificultando mucho la tarea de diferenciar ψ de estas funciones.

Cuando se dé este fenómeno habremos de realizar la diagonalización con dos estrategias diferentes y simultáneas, para abordar eficazmente cada una de las dos situaciones descritas. Un ejemplo del mismo se da al intentar demostrar que la siguiente función no es computable:

$$\mathbf{f}(\mathbf{x}) = \begin{cases} \text{true} & \mathbf{x} \in \text{TOT} \Leftrightarrow \mathbf{W}_x = \Sigma^* \Leftrightarrow \varphi_x \text{ es total} \\ \text{false} & \text{c.c.} \end{cases}$$

JUSTIFICACIÓN: Intuitivamente podemos sospechar que no es computable, ya que averiguar si $\mathbf{x} \in$TOT equivale a probar que $\forall \mathbf{y}\ \varphi_x(\mathbf{y})\downarrow$, lo cual parece exigir que resolvamos el problema de parada para un número infinito de entradas.

PLANIFICACIÓN: La función **f** nos proporciona información sobre lo que ocurre con cada función computable de la tabla. Concretamente, para cada φ_x nos indica si converge para todos los puntos o diverge en alguno.

- Si **x** está en TOT (cosa que averiguaremos porque se cumple $\mathbf{f}(\mathbf{x})$), sabremos que para toda palabra **y** la función φ_x está definida. Esta información sobre φ_x es muy valiosa, ya que podemos preguntar por cualquier valor $\varphi_x(\mathbf{y})$ sin miedo a que cicle, y utilizar la respuesta obtenida para conseguir que $\delta(\mathbf{y})$ sea diferente. Concretamente, si queremos diferenciarnos en la diagonal $\varphi_x(\mathbf{x})$ podemos hacer que $\delta(\mathbf{x})\uparrow$, o preguntar si $\varphi_x(\mathbf{x})=0$ y utilizar 1 ó 0 según la respuesta, o bien calcular valores diferentes de $\varphi_x(\mathbf{x})$ como son $\varphi_x(\mathbf{x})+100$, $\varphi_x(\mathbf{x})+1$ o $2*\varphi_x(\mathbf{x})+7$.
- Si **x** no está en TOT (cosa que averiguamos porque no se cumple $\mathbf{f}(\mathbf{x})$), sabremos que al menos para alguna palabra **y** la función φ_x diverge. Esta información sobre φ_x es muy pobre, ya que no se nos indica sobre qué palabra o palabras se producirá en concreto la indefinición. Por tanto no tenemos información concreta sobre el comportamiento de φ_x sobre la palabra **x** de la diagonal, y lo que es más grave, sobre ninguna otra palabra **y**. Tampoco podemos preguntar nada más sobre ningún valor, ya que al ser φ_x no total la curiosidad puede resultar fatídica. La conclusión es que no tenemos ningún apoyo para conseguir que se diferencie de φ_x para una entrada concreta.

Dado que la calidad de la información que recibimos sobre las funciones de distintas clases es notoriamente despareja, llamamos a este caso el de la *diagonalización asimétrica*. La situación es tal como se presenta en la tabla representada en la figura 5.6.

Como vemos, la dificultad reside en el caso $\mathbf{x} \notin$TOT porque no podemos conseguir que las funciones δ y φ_x se diferencien localmente en ningún punto. Sin

embargo, la información que poseemos sobre φ_x no es del todo inútil: al menos sabemos que es una función no total. Visto así podríamos conseguir que δ y φ_x sean diferentes si definimos δ de forma que sí sea total. Aunque no sepamos en qué punto concreto se produce esta diferencia local, si cumplimos este objetivo tendremos la seguridad de que δ (total) y φ_x (no total) no pueden ser la misma función, que es lo que en realidad nos interesa. Lo que hacemos en este caso es definir δ de forma que *se diferencie globalmente* de φ_x.

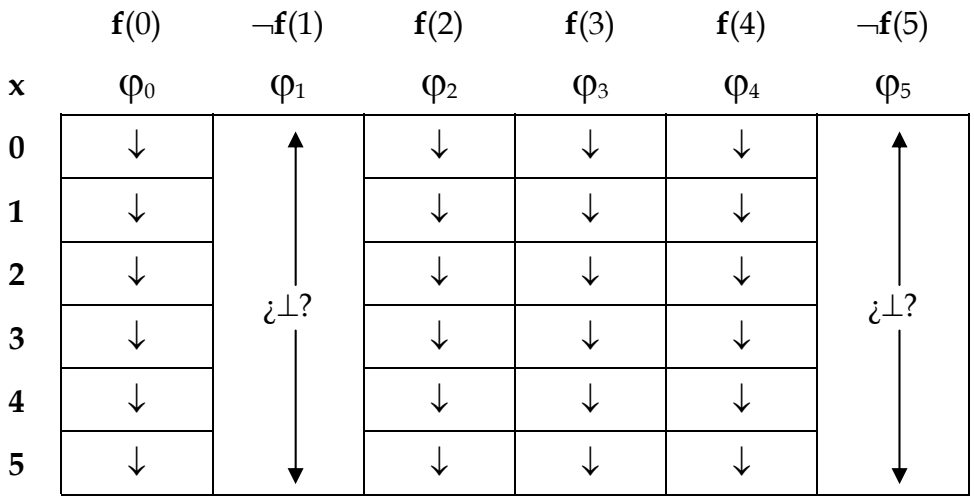

Figura 5.6: Los casos de diagonalización asimétrica vienen determinados por el hecho de que la función presuntamente incomputable **f** ofrece información muy pobre en algunos casos. Aquí cuando **f** nos indica que φ_x es total no hay problema, pero si se da $\neg\mathbf{f}(\mathbf{x})$ únicamente sabemos que la función φ_x está indefinida en algún lugar, pero no sabemos cuál.

Ahora bien, si decidimos convertir a δ en total, la libertad de opciones que teníamos para el caso $\mathbf{x}\in$TOT se nos verá restringida, ya que la posibilidad de divergir quedará eliminada. Dado que δ va a ser total será más adecuado llamar **d** a la función diagonal que nos proponemos describir.

- Si $\mathbf{x}\in$TOT, definiremos $\mathbf{d}(\mathbf{x})$ como $\varphi_x(\mathbf{x})+3$, con lo que **d** será diferente de φ_x de manera local en el punto **x**. Podemos hacer esto porque sabemos que $\varphi_x(\mathbf{x})\downarrow$, y mantenemos nuestro propósito de que **d** sea total.
- Si $\mathbf{x}\notin$TOT no necesitamos el punto **x** para diferenciarnos localmente de φ_x. Eso sí, tendremos que dar a $\mathbf{d}(\mathbf{x})$ algún valor, por ejemplo el 10, para conseguir que **d** sea total y se diferencie así globalmente de φ_x.

El esquema quedaría ilustrado en la tabla-ejemplo de la figura 5.7, donde podemos ver que la nueva función **d** se diferencia de las funciones computables totales en el punto de la diagonal, y de las computables no totales globalmente: no podemos determinar en qué punto exactamente, pero lo seguro es que los índices de **d** están en TOT y los de la correspondiente φ_x no.

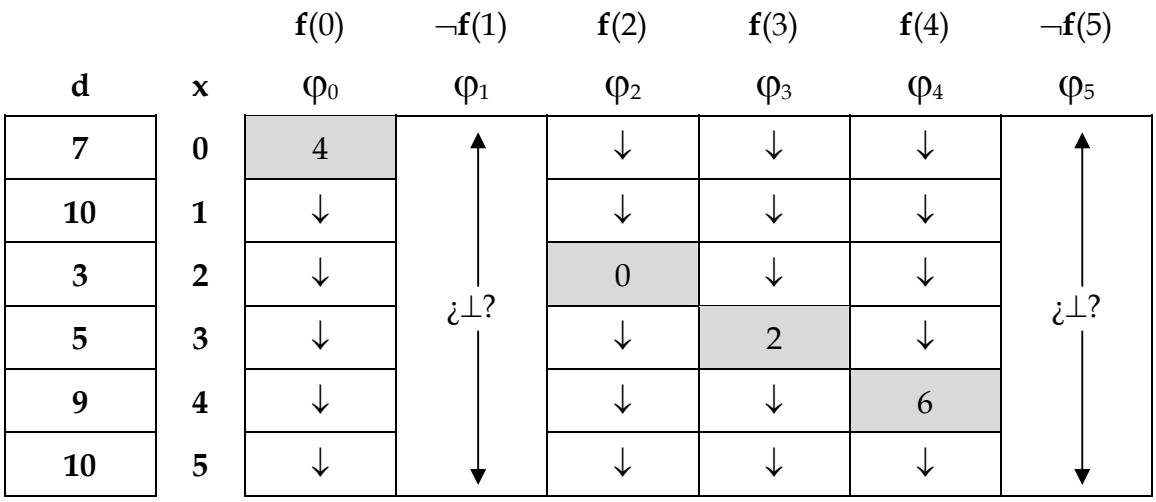

Figura 5.7: Construcción de una función diagonal asimétrica **d**. Por un lado conseguimos que sea computable y total, y por otro que se diferencie en los valores de la diagonal de todas las funciones computables y totales. Para esto último averiguamos dichos valores aprovechando que sabemos que la función es total.

Quizá convenga resaltar el mecanismo de diferenciación que hemos descrito, para ver por qué la función **d** es diferente de cada una de las que hay en la tabla:

- **d** es distinta de la función total φ_0 en el punto 0, pues $\varphi_0(0)=2$ y $\mathbf{d}(0)=5$.
- **d** es distinta de la función no total φ_1 aunque no sabemos dónde. Pero estamos seguros de que existe un punto **y** tal que $\varphi_1(\mathbf{y})\uparrow$, mientras que **d** va a convergir en dicho punto.
- ...
- **d** es distinta de cualquier función total $\varphi_{\mathbf{x}}$ en el punto **x**, pues $\varphi_{\mathbf{x}}(\mathbf{x})$ está definido y $\mathbf{d}(\mathbf{x})=\varphi_{\mathbf{x}}(\mathbf{x})+3$.
- **d** es distinta de cualquier función no total $\varphi_{\mathbf{x}}$ aunque no sabemos dónde. Pero existe un punto **y** tal que $\varphi_{\mathbf{x}}(\mathbf{y})\uparrow$, mientras que $\mathbf{d}(\mathbf{y})$ va a convergir. Es importante recalcar que puede darse el caso de que $\varphi_{\mathbf{x}}(\mathbf{x})=10=\mathbf{d}(\mathbf{x})$, y que ello no afecta para nada al argumento, ya que el hecho de que **d** y $\varphi_{\mathbf{x}}$ son funciones diferentes no reside en el valor que otorgan al punto **y**.

CONSTRUCCIÓN: Supongamos que **f** es computable. Definimos formalmente la función **d** tal como lo hemos planeado:

$$\mathbf{d}(\mathbf{x}) = \begin{cases} 10 & \neg\mathbf{f}(\mathbf{x}) \\ \varphi_{\mathbf{x}}(\mathbf{x})+3 & \mathbf{f}(\mathbf{x}) \end{cases}$$

Como **f** es computable (decidible, pues se trata de un predicado) tenemos que la función **d** es también computable, como lo demuestra el siguiente programa:

if not f(X1) **then** X0 := 10; **else** X0 := Φ(X1,X1)+3; **end if**;

Por tanto existe un índice **e**, tal que $\mathbf{d}=\varphi_{\mathbf{e}}$. Es importante notar que la función **d** es total, pues cuando invocamos la función universal en la parte **else** no lo hacemos sin comprobar mediante **f** que el programa cuya ejecución vamos a simular corresponde a una función total. Así que *además se verifica* ***f(e)***.

CONTRADICCIÓN: Veamos que, tal y como hemos razonado la construcción de **d**, ese índice es imposible. Como $\mathbf{d}=\varphi_{\mathbf{e}}$ es total, **d** se diferenciará de $\varphi_{\mathbf{e}}$ en el punto **e**, por lo que obtendremos una contradicción al calcular **d**(**e**). En principio hay dos posibilidades, pero con la segunda no llegamos muy lejos por el hecho de ser $\varphi_{\mathbf{e}}$ total:

Caso 1: $\mathbf{d}(\mathbf{e})\cong\varphi_{\mathbf{e}}(\mathbf{e})+3 \overset{\mathbf{d}\cong\varphi_e}{\Rightarrow} \mathbf{d}(\mathbf{e})\cong\mathbf{d}(\mathbf{e})+3 \Rightarrow \mathbf{d}(\mathbf{e})\uparrow \Rightarrow$ **d** no es total ↯

Caso 2: $\mathbf{d}(\mathbf{e})=10 \overset{\text{def. } \mathbf{d}}{\Rightarrow} \neg\,\mathbf{f}(\mathbf{e}) \overset{\text{def. } \mathbf{f}}{\Rightarrow} \varphi_{\mathbf{e}}$ no es total $\overset{\mathbf{d}\cong\varphi_e}{\Rightarrow}$ **d** no es total ↯

Vemos que esta contradicción es algo diferente a las que nos hemos encontrado hasta el presente. La razón está en la asimetría de la diagonalización. Como era de esperar, la totalidad de **d** se convierte en un elemento esencial de la prueba, y la contradicción nos demuestra que la suposición inicial de que **f** era computable es falsa.

> *El que la función f que acabamos de ver sea incomputable es una severa limitación para la actividad de la programación. Es evidente que en la mayor parte de las aplicaciones de la Informática práctica se busca construir programas y rutinas que terminen la tarea que se les ha encomendado. Un primer criterio de depuración del software producido debería ser la eliminación o sustitución de todo programa que pueda ciclar indefinidamente en alguna circunstancia (es decir, que compute una función no total). Desgraciadamente, acabamos de comprobar que no existe modo alguno de realizar esta tarea de forma algorítmica. Esto nos puede hacer reflexionar sobre las razones de que exista tanto software con errores. El problema es que no existen métodos para detectar sistemáticamente dichos errores, ni siquiera los más garrafales, como puede ser la introducción accidental de bucles infinitos en un programa. Como en el caso del problema de parada, esto no quiere decir que en ningún caso sea posible comprobar si un programa es seguro y para siempre, o puede ciclar para algunas ejecuciones: hay programas tan sencillos que no resisten un mínimo análisis.*

Hay una última cuestión relevante. Podría pensarse que la última demostración que hemos hecho es superflua, ya que la incomputabilidad de *halt* parece implicar la de la función **f** que distingue los programas totales de los que no lo son. El razonamiento sería: si no podemos decidir si un programa para sobre un

dato concreto, ¿cómo vamos a poder decidir si para sobre todos los datos? Sin embargo esta impresión es falsa, ya que la incomputabilidad de *halt* y **f** son en principio independientes. Aunque existiera un método para comprobar la totalidad de un programa, *halt* podría seguir siendo incomputable. La razón es que cuando $\mathbf{f}(\mathbf{x})$ se cumpliera sabríamos que $\mathbf{P}_\mathbf{x}$ para ante cualquier dato, y por tanto "su" problema de parada "particular" estaría resuelto. Sin embargo, cuando $\neg\mathbf{f}(\mathbf{x})$ sabríamos que $\varphi_\mathbf{x}$ no es total, pero el método no tendría por qué permitirnos saber dónde converge y dónde no, y no resolveríamos la indecidibilidad de *halt*.

6. Decidibilidad y semidecidibilidad

En el apartado anterior hemos demostrado la existencia de problemas no computables y presentado la técnica de diagonalización para establecer la no computabilidad de un problema. A través de algunos ejemplos hemos podido intuir dónde reside la clave de esa incomputabilidad. En muchos casos no existe un algoritmo porque es imposible dar una respuesta cuando esta depende de comprobar si un programa cicla (y no hay otra manera de solventarlo).

El hecho de que un programa pueda ciclar nos ha forzado a acordar un valor ficticio $\perp$ para representar ese comportamiento (resultado indefinido), así como a trabajar de manera sistemática con funciones parciales. Por un lado, la evidencia de que los programas pueden ciclar nos indica que debemos contar con ese fenómeno para describir con precisión el fenómeno de la computabilidad. Por otro lado, la idea de un programa que no termina choca con el concepto de algoritmo como método finito. Parece un contrasentido que tengamos que considerar como algoritmo un programa como el que computa la función vacía, y que por tanto es incapaz de producir ningún resultado bajo ninguna circunstancia.

> *Parece que hubiera sido mucho más razonable inventar una informática sin el molesto problema de los programas que no paran. Una posibilidad hubiera sido definir lenguajes en los que esta posibilidad no se dé, pero con esta aproximación (como en el caso de los programas-for mencionados en la nota de la página 21) lo único que se consigue son sistemas defectuosos que no alcanzan a cubrir muchas funciones computables interesantes. Otra posible vía sería tomar un lenguaje completo, como el de los programas-while, y expurgar los que computen funciones no totales. También hemos visto en el capítulo anterior que esto no es posible, ya que ese conjunto de programas es indecidible: si no podemos distinguir los programas "buenos" de los "malos", malamente nos podremos librar de estos últimos. De una u otra forma estamos condenados a convivir con el comportamiento divergente.*

Sin embargo, algunos algoritmos que a veces no terminan son esenciales para la computación, como lo demuestra el programa universal **U** descrito en la sección 4.2. La razón es que esos programas no totales son lo mejor que tenemos para resolver, siquiera parcialmente, algunos problemas extremadamente difíciles. Vamos entonces a retomar los problemas computables para hacer una clasificación atendiendo a este criterio: si los podemos resolver por entero o si nos tenemos que contentar con una aproximación incompleta.

6.1 Problemas de decisión y conjuntos

En primer lugar vamos a reducir el término general problema a algo más concreto. Desde el principio de este documento hemos presentado los problemas a resolver como funciones, entendiendo que lo que se pretende es encontrar un programa (si existe) que obtenga a partir de unos datos de entrada unos resultados (salida). Sin embargo, a lo largo de los ejemplos que hemos ido desarrollando a lo largo del capítulo 5 hemos podido observar que en ningún caso ha resultado relevante el mero resultado de la función para su incomputabilidad. Cuando ha habido un obstáculo para que una función sea computable este siempre se ha sustentado en la dificultad de decidir una condición o decisión de tipo *true*/*false*, y las acciones anteriores o posteriores a la resolución de dicho dilema han resultado meros adornos que no afectaban a la naturaleza incomputable de la función. Podemos sospechar que, desde el punto de vista de la dificultad (o imposibilidad) de computar algo, las funciones más interesantes son los predicados

Basándonos en esta experiencia vamos a simplificar un poco nuestro escenario para fijarnos únicamente en las funciones con resultado booleano, es decir, en los predicados. Los problemas que pueden expresarse mediante predicados se denominan ***problemas de decisión***. Y, aunque no todos los problemas interesantes son de este tipo, lo que sí podemos decir es que todos los problemas relevantes para la Teoría de la Computabilidad tienen asociado un problema de decisión.

Para ilustrar el fundamento de este argumento, tomemos como ejemplo un programa que resuelva un problema difícil, como puede ser un compilador de C: su tarea consiste en traducir un programa del lenguaje C a un lenguaje máquina concreto. Naturalmente su funcionamiento y resultado quedan muy lejos de los de los programas que devuelven únicamente los valores *true* o *false*. Pero el compilador, entre otras cosas, ha de decidir sobre la validez del programa fuente. De hecho esta es su acción fundamental, pues solo tras este análisis puede construir otro programa equivalente en lenguaje objeto. Así pues, la computabilidad del problema que resuelve el compilador se basa a su vez en la computabilidad de un problema aparentemente más sencillo: el que responde a la pregunta ¿forma la secuencia de caracteres de entrada un programa correcto de acuerdo a la especificación del lenguaje C? y cuya respuesta es un valor booleano. Si este nuevo problema no fuera computable no existiría el compilador en C.

Además de ocuparnos especialmente de los problemas de decisión lo haremos desde una perspectiva ligeramente diferente de la funcional: en lugar de tomar como objeto de nuestro discurso los predicados hablaremos más bien de *conjuntos*. Aunque nos resultará muy cómoda, la distinción no es tan importante, ya que todo

predicado tiene asociado un conjunto y viceversa. Por ejemplo, el predicado que responde con *true* o *false* a la pregunta "¿forma la secuencia de caracteres de entrada un programa correcto de acuerdo a la especificación del lenguaje C?" define el conjunto de las secuencias de caracteres que sí forman un programa en C, es decir, que satisfacen el predicado. Análogamente, un conjunto como el formado por los números naturales que tienen exactamente 1.347 divisores distintos tiene asociado el predicado "¿tiene el número de entrada exactamente 1.347 divisores distintos?".

Los conjuntos que manejaremos estarán formados por objetos que pertenezcan a tipos de datos implementados, ya que lo que nos interesa de estos conjuntos son sus propiedades de computabilidad, y por tanto la posibilidad de manipular sus elementos mediante programas-while. Los siguientes son ejemplos de tales conjuntos:

$\mathbf{A} = \{ \mathbf{x} \in \Sigma^*: |\mathbf{x}|_a > 3 \}$

$\mathbf{B} = \{ \mathbf{x} \in \mathbb{N}: \exists \mathbf{y}\ \mathbf{y}^3 = \mathbf{x} \}$

$\mathbf{C} = \{ (\mathbf{x},\mathbf{y}) \in \Sigma^* \times \mathbb{P}: \mathbf{x}$ es prefijo de cima($\mathbf{y}$) $\}$

aunque, de acuerdo con lo que hemos visto hasta ahora, gozarán de nuestra preferencia los conjuntos que involucran programas, ya que estos son los objetos que tienen propiedades más interesantes:

$\mathbf{D} = \{ \mathbf{x} \in \mathbb{W}: \mathbf{P}_\mathbf{x}$ no tiene bucles $\}$

$\mathbf{E} = \{ (\mathbf{x},\mathbf{y}) \in \mathbb{W} \times \Sigma^*: \varphi_\mathbf{x}(\mathbf{y})\downarrow \}$

$\mathbf{F} = \{ (\mathbf{x},\mathbf{y},\mathbf{z}) \in \mathbb{W} \times \Sigma^* \times \mathbb{N}: \mathbf{P}_\mathbf{x}(\mathbf{y})$ converge en exactamente $\mathbf{z}$ pasos $\}$

Respecto de la forma sistemática de relacionar conjuntos y predicados introducimos la siguiente definición:

DEFINICIÓN: Sea $\mathbf{A} \subseteq \Sigma^*$ un conjunto de palabras. Llamamos ***función característica*** de **A** al predicado:

$$\mathbf{C}_\mathbf{A}(\mathbf{x}) = \begin{cases} \text{true} & \mathbf{x} \in \mathbf{A} \\ \text{false} & \mathbf{x} \notin \mathbf{A} \end{cases}$$

Por tanto, la función característica de un conjunto **A** es el predicado que sirve para *distinguir* los elementos de **A** de los que no lo son. En definitiva, nuestros problemas de decisión van a estar relacionados con la pertenencia o no de un dato a un conjunto determinado.

Esta noción se generaliza de forma natural a conjuntos de k-tuplas de palabras. En efecto, si $\mathbf{B}\subseteq\sum^{*k}$, entonces su función característica será:

$$\mathbf{C_B}(\mathbf{x}_1,\mathbf{x}_2,...\mathbf{x}_k) = \begin{cases} \text{true} & (\mathbf{x}_1,\mathbf{x}_2,...,\mathbf{x}_k)\in\mathbf{B} \\ \text{false} & (\mathbf{x}_1,\mathbf{x}_2,...,\mathbf{x}_k)\notin\mathbf{B} \end{cases}$$

Idénticamente se definen las funciones características de conjuntos de objetos o tuplas de objetos diferentes de las palabras, es decir, de la forma $\mathbf{A}\subseteq\mathbb{T}$ o $\mathbf{B}\subseteq\mathbb{T}_1\times\mathbb{T}_2\times...\times\mathbb{T}_k$, donde $\mathbb{T}$, $\mathbb{T}_1$, $\mathbb{T}_2$,..., $\mathbb{T}_k$ son tipos de datos implementados).

Con respecto a los ejemplos de más arriba, tenemos por ejemplo que la función característica $\mathbf{C_F}$: $\mathbb{W}\times\sum^*\times\mathbb{N}\rightarrow\mathbb{B}$ será:

$$\mathbf{C_F}(\mathbf{x},\mathbf{y},\mathbf{z}) = \begin{cases} \text{true} & \mathbf{P_x}(\mathbf{y}) \text{ converge en exactamente } \mathbf{z} \text{ pasos} \\ \text{false} & \text{c.c.} \end{cases}$$

6.2 Conjuntos decidibles y sus propiedades

Teniendo en cuenta la perspectiva con la que hemos decidido aproximarnos a los conjuntos y el hecho declarado de que lo que nos interesa son sus propiedades de computabilidad, lo primero que cabe plantearse es qué clase de manipulación podemos ejercer sobre un conjunto mediante un programa.

DEFINICIÓN: Sea $\mathbf{A}\subseteq\sum^*$ un conjunto. Decimos que **A** es ***decidible*** si su función característica $\mathbf{C_A}$ es computable.

Esta noción se generaliza de forma natural para conjuntos de varias dimensiones o cuyos elementos pertenezcan a otros tipos de datos.

La idea de *conjunto decidible* es coherente con la que hemos dado de *predicado decidible*, ya que un conjunto es decidible si y sólo si su predicado asociado (es decir, su función característica) lo es también. Por tanto nos podremos permitir usar el término decidible de forma indistinta para ambos tipos de objetos.

Cuando queramos demostrar que un conjunto es decidible deberemos, por tanto, encontrar un programa que compute su función característica. En el caso del ejemplo **B** podría servir el programa:

```
X0 := false;
for IND in 0..X1 loop
      X0 := X0 ∨ (X1=IND∗IND∗IND);
end loop;
```

que devuelve *true* cuando el dato de entrada es un cubo perfecto y *false* en caso contrario.

Por tanto, intuitivamente un conjunto es decidible cuando existe un algoritmo que permite *distinguir* los elementos del conjunto de los que no lo son. La mayoría (si no la totalidad) de los conjuntos con los que trabajamos ordinariamente son decidibles (los números primos, los vectores con más posiciones nulas que no nulas, las cadenas de caracteres que representan un comando válido en UNIX BSD, etc...).

En contrapartida, un conjunto será indecidible cuando sea imposible encontrar un algoritmo que permita distinguir entre elementos pertenecientes y no pertenecientes al conjunto. Demostrar la no decidibilidad de un conjunto implica la demostración de incomputabilidad de una función característica, por lo que en general habremos de recurrir a la diagonalización. Un conjunto del que tenemos constancia de su indecidibilidad es el ejemplo **E** de más arriba, ya que su función característica $\mathbf{C_E}$ es precisamente *halt*. Relacionado con E está el conjunto **K** de los programas que convergen sobre sí mismos:

$$\mathbf{K} = \{ \mathbf{x} \in \mathbb{W} : \varphi_{\mathbf{x}}(\mathbf{x})\downarrow \}$$

Aunque no hemos demostrado formalmente que $\mathbf{C_K}$ no es computable, ello no resulta en modo alguno difícil. Sólo tenemos que adaptar muy ligeramente la prueba presentada en la sección 5.2 para la incomputabilidad de *halt*. Otro ejemplo es el conjunto:

$$\mathbf{TOT} = \{ \mathbf{x} \in \mathbb{W} : \forall \mathbf{y}\ \varphi_{\mathbf{x}}(\mathbf{y})\downarrow \}$$

ya que la indecidibilidad de $\mathbf{C_{TOT}}$ fue establecida en el apartado 5.5.3 (aunque entonces aún no la llamábamos con ese nombre).

NOTACIÓN: Denominamos Σ_0 a la clase de todos los conjuntos decidibles. Así pues, es equivalente decir que **A** es decidible o que $\mathbf{A} \in \Sigma_0$.

> *En buena parte de la literatura clásica sobre Teoría de la Computabilidad encontramos que a los conjuntos decidibles se les denomina* ***recursivos*** *por razones históricas relacionadas con los primeros modelos utilizados para investigar las propiedades de computabilidad. Sin embargo consideramos que ese término es confuso y preferimos el equivalente decidibles.*

6.2.1 Propiedades de cierre de los conjuntos decidibles

En general las propiedades de los conjuntos decidibles son sencillas de entender y demostrar.

PROPOSICIÓN 1: Los conjuntos *finitos* y *cofinitos* son decidibles.

⌦ Sea $\mathbf{A}\subseteq\sum^*$ un conjunto finito sean $\{\mathbf{y}_1, \ldots, \mathbf{y}_n\}$ sus **n** elementos. El programa que computa la función característica $\mathbf{C_A}$ es:

```
X0 := false;
if  X1=Y1  or  …  or  X1= YN   then    X0 := true;    end if;
```

Sea ahora $\mathbf{B}\subseteq\sum^*$ un conjunto cofinito y sean $\{\mathbf{z}_1, \ldots, \mathbf{z}_m\}$ los **m** elementos del conjunto finito $\overline{\mathbf{B}}$. El programa que computa la función característica $\mathbf{C_B}$ es:

```
X0 := true;
if  X1=Z1  or  …  or  X1=ZM   then    X0 := false;    end if;
```
⌫

Este resultado tiene, por una parte, una lógica sencilla: si un conjunto sólo contiene un número finito **n** de elementos siempre existirá un programa (que al expandirse consistirá en una serie de **n** casos if, uno para comparar el dato de entrada con cada uno de los posibles elementos del conjunto). Pero por otro lado tiene consecuencias complejas, porque hay conjuntos finitos que nos resultan extraordinariamente difíciles de entender o abarcar. Por ejemplo, el conjunto de todos los días de febrero del año 1325 en los que llovió en alguna parte de la costa Malabar, o el conjunto de todos los programas totales cuyo código es menor que un millón. Son conjuntos finitos y por tanto decidibles, pero no seríamos capaces de construir un programa para su función característica. Para conciliar esto acudimos de nuevo al concepto de *computabilidad no efectiva* que vimos en la sección 3.1. Estos conjuntos son decidibles y existe un programa que computa su función característica. Lo que sucede es que no conocemos lo suficiente la naturaleza de dichos conjuntos (porque su especificación no nos ayuda a ello) como para encontrar ese programa concreto.

PROPOSICIÓN 2: La unión e intersección de conjuntos decidibles son también decidibles.

⌦ Sean $\mathbf{A}, \mathbf{B}\subseteq\sum^*$ conjuntos decidibles. Por tanto, sus funciones características $\mathbf{C_A}$ y $\mathbf{C_B}$ son computables. Para ver que $\mathbf{A}\cap\mathbf{B}\in\sum_0$ utilizaremos el siguiente programa que computa su función característica $\mathbf{C_{A\cap B}}$:

```
X0 := C_A(X1) ∧ C_B(X1);
```

Para ver que $\mathbf{A}\cup\mathbf{B}\in\sum_0$ utilizaremos el siguiente programa que computa su función característica $\mathbf{C_{A\cup B}}$:

```
X0 := C_A(X1) ∨ C_B(X1);
```
⌫

PROPOSICIÓN 3: El complementario de un conjunto decidible es decidible.

⌦ Sea $\mathbf{A}\subseteq\sum^*$ un conjunto decidible. Por tanto, su función característica $\mathbf{C_A}$ es computable. El programa que computa la función característica $\mathbf{C_{\overline{A}}}$ es:

X0 := $\neg C_A$ (X1); ⌫

Como vemos la manipulación de los conjuntos decidibles mediante operaciones clásicas de Teoría de Conjuntos rinde buenos resultados. Ello se debe a que se corresponde con una manipulación lógica de los predicados asociados, que a su vez se refleja en una combinación sencilla de programas totales mediante distinción de casos.

Estas propiedades se generalizan de forma natural a conjuntos de k-tuplas de palabras ($\mathbf{B}\subseteq\sum^{*k}$) o de objetos diferentes de las palabras ($\mathbf{B}\subseteq\mathbb{T}_1\times\mathbb{T}_2\times...\times\mathbb{T}_k$, donde $\mathbb{T}$, $\mathbb{T}_1$, $\mathbb{T}_2$,..., $\mathbb{T}_k$ son tipos de datos implementados). Las demostraciones son análogas pero teniendo en cuenta que los datos son multidimensionales y que por tanto utilizan **k** variables de entrada.

6.3 Conjuntos semidecidibles y sus propiedades

Hemos señalado que los conjuntos **K** y **TOT** no son decidibles porque sus funciones características no son computables. Sin embargo entre ambos conjuntos podemos intuir alguna diferencia. Si intentamos lo imposible, es decir computar la función característica $\mathbf{C_K}$ nos encontramos con que la dificultad estriba en que resulta imposible devolver *false* si $\varphi_{\mathbf{x}}(\mathbf{x})\uparrow$, pero cuando $\varphi_{\mathbf{x}}(\mathbf{x})\downarrow$ no tendríamos problema en devolver *true*. En este sentido podemos decir que es "casi computable" porque bastaría cambiar *false* por indefinido para obtener una función computable.

Por el contrario, para el conjunto **TOT** no parece fácil encontrar una solución similar: ni en el caso en el que $\forall\mathbf{y}\ \varphi_{\mathbf{x}}(\mathbf{y})\downarrow$ ($\mathbf{x}\in\mathbf{TOT}$) ni cuando $\exists\mathbf{y}\ \varphi_{\mathbf{x}}(\mathbf{y})\uparrow$ ($\mathbf{x}\notin\mathbf{TOT}$) tenemos posibilidad alguna de obtener una respuesta satisfactoria. De alguna manera podemos decir que $\mathbf{C_{TOT}}$ está más lejos de ser computable que $\mathbf{C_K}$.

El objetivo que nos proponemos es refinar la división actual de los conjuntos (decidibles y no decidibles), distinguiendo algunos de entre los indecidibles que están más cercanos a la computabilidad. Estos conjuntos, a pesar de no ser decidibles, admiten una solución parcial: disponen de un programa que, siendo capaz de dar respuesta positiva ante las entradas que pertenecen al conjunto, cicla ante las que no lo son.

Recobremos el primer ejemplo de función incomputable que hemos visto: la función *halt* del problema de parada. Vemos que relajando la exigencia de que termine siempre encontramos la función computable:

$$\xi(\mathbf{x},\mathbf{y}) \cong \begin{cases} \text{true} & \varphi_{\mathbf{x}}(\mathbf{y})\downarrow \\ \\ \perp & \text{c.c.} \end{cases}$$

Demostrar la computabilidad de esta función es sencillo pues basta utilizar la función universal.

```
R := Φ(X1, X2);
X0 := true;
```

Como hemos adelantado en la introducción de esta sección, para algunos conjuntos indecidibles podemos encontrar una función computable relajando la función característica de modo similar a como acabamos de hacer con el problema de parada. Sustituyendo en la función característica la salida *false* por indefinido obtenemos la función semicaracterística.

DEFINICIÓN: Dado un conjunto **A**, llamamos ***función semicaracterística*** de **A** a la siguiente función parcial:

$$\chi_{\mathbf{A}}(\mathbf{x}) \cong \begin{cases} \text{true} & \mathbf{x} \in \mathbf{A} \\ \\ \perp & \text{c.c.} \end{cases}$$

DEFINICIÓN: Un conjunto $\mathbf{A} \subseteq \sum^*$ es ***semidecidible*** si su función semicaracterística, $\chi_{\mathbf{A}}$, es computable:

Estas dos nociones se generalizan de forma natural para conjuntos de varias dimensiones o cuyos elementos pertenezcan a otros tipos de datos.

Cuando queramos demostrar que un conjunto es semidecidible deberemos, por tanto, encontrar un programa que compute su función semicaracterística. Por ejemplo el conjunto $\mathbf{K} = \{ \mathbf{x} \in \mathbb{W} : \varphi_{\mathbf{x}}(\mathbf{x})\downarrow \}$ (que hemos visto que no es decidible) sí resulta semidecidible. Su función semicaracterística $\chi_{\mathbf{K}}$ es calculada por el siguiente programa:

```
R := Φ(X1, X1);
X0 := true;
```

Este conjunto tiene gran importancia técnica porque resulta muy manejable para recurrir a él como arquetipo de conjunto indecidible pero semidecidible (es decir, de los no computables pero "casi"). Por ejemplo se utiliza en muchas demostraciones que prueban por reducción al absurdo la indecidibilidad de ciertos conjuntos.

Otro ejemplo de conjunto semidecidible sería

$\overline{\mathbf{VAC}} = \{ \mathbf{x} \in \mathbb{W} : \exists \mathbf{y}\ \varphi_{\mathbf{x}}(\mathbf{y})\downarrow \}$

de los programas que son capaces de devolver resultado al menos en un caso. Su función semicaracterística es calculada por un programa que utiliza la técnica del intercalado de procesos:

```
PAR := 0;
while not T(X1,decod_2_1(PAR),decod_2_2(PAR)) loop
      PAR := suc(PAR);
end loop;
X0 := true;
```

Por tanto, intuitivamente un conjunto es decidible cuando existe un algoritmo que permite *detectar* los elementos del conjunto, aunque dicho algoritmo no tiene por qué encontrar una respuesta para los que no lo son.

En contrapartida, un conjunto no será semidecidible cuando sea imposible siquiera encontrar un algoritmo de detección para sus elementos. Demostrar la no semidecidibilidad de un conjunto implica la demostración de incomputabilidad de una función semicaracterística, pero se puede comprobar que, en general, las variantes de la diagonalización que hemos visto en el capítulo 5 no se acomodan bien para probar la incomputabilidad de funciones no totales. En el apartado 6.3.4 enriqueceremos la diagonalización para hacer frente a este problema.

NOTACIÓN: Denominamos Σ_1 a la clase de todos los conjuntos semidecidibles. Así pues, es equivalente decir que $\mathbf{A}$ es semidecidible o que $\mathbf{A}\in\Sigma_1$.

> *Al igual que pasa con los conjuntos decidibles (que son llamados también recursivos) encontramos muy extendida la denominación de* ***recursivamente enumerables*** *para los conjuntos semidecidibles. Por razones análogas no utilizaremos esa nomenclatura.*

6.3.1 Relación entre semidecidibilidad y decidibilidad

Aunque hayamos podido dar la impresión de que los conjuntos semidecidibes son una clase aparte de problemas más difíciles esto no es así, ya que la relación es inclusiva.

PROPOSICIÓN 4: Todo conjunto decidible es semidecidible. Es decir $\Sigma_0\subset\Sigma_1$

⌦ Sea $\mathbf{A}\subseteq\sum^*$ un conjunto decidible. Por tanto, su función característica $\mathbf{C_A}$ es computable, y nos basaremos en ella para escribir el programa que computa la función semicaracterística $\chi_\mathbf{A}$:

if $C_A(X1)$ **then** X0 := true; **else** X0 := ⇊ ; **end if**;

Con esto hemos demostrado que $\Sigma_0\subseteq\Sigma_1$. Para ver que el contenido es estricto basta recordar que $\mathbf{K}\in\Sigma_1$ pero $\mathbf{K}\notin\Sigma_0$. ⌫

Tenemos entonces que entender la clase Σ_1 en un sentido no restrictivo. Si **B** es semidecidible sabemos que existe un programa **P** para detectar los elementos de **B**. Este programa es defectuoso al tratar con los elementos de $\overline{\mathbf{B}}$, pero no podemos deducir que **P** sea *lo mejor a que podemos aspirar* para trabajar con **B**. Quizá exista, además de **P**, otro programa **Q** más eficaz, en el sentido de que trate correctamente las entradas de $\overline{\mathbf{B}}$ distinguiéndolas de las de **B**. En ese caso el conjunto **B** será, además de semidecidible, decidible. Es fundamental no confundir la clase Σ_1 (que contiene conjuntos tanto decidibles como indecidibles) con la clase $\Sigma_1-\Sigma_0$ (que contiene sólo conjuntos indecidibles).

Existe no obstante una relación mucho más estrecha entre ambas clases de conjuntos, y esta nos viene dada por la forma en que se comportan sus complementarios.

PROPOSICIÓN 5 (TEOREMA DEL COMPLEMENTARIO): Un conjunto $\mathbf{A}\subseteq\sum^*$ es decidible si y sólo tanto **A** como su complementario $\overline{\mathbf{A}}$ son semidecidibles.

⌦ Primero comprobaremos que $\mathbf{A}\in\Sigma_0 \Rightarrow \mathbf{A}\in\Sigma_1 \wedge \overline{\mathbf{A}}\in\Sigma_1$, que es la parte más evidente, pues se deduce de las propiedades que acabamos de enunciar y demostrar.

$$\left.\begin{array}{l} \mathbf{A}\in\Sigma_0 \overset{\text{prop. 4}}{\Rightarrow} \mathbf{A}\in\Sigma_1 \\ \mathbf{A}\in\Sigma_0 \overset{\text{prop. 3}}{\Rightarrow} \overline{\mathbf{A}}\in\Sigma_0 \overset{\text{prop. 4}}{\Rightarrow} \overline{\mathbf{A}}\in\Sigma_1 \end{array}\right| \Rightarrow \mathbf{A}\in\Sigma_1 \wedge \overline{\mathbf{A}}\in\Sigma_1$$

Veamos ahora que $\mathbf{A}\in\Sigma_1 \wedge \overline{\mathbf{A}}\in\Sigma_1 \Rightarrow \mathbf{A}\in\Sigma_0$. De entrada, si **A** y $\overline{\mathbf{A}}$ son semidecidibles sus funciones semicaracterísticas $\chi_{\mathbf{A}}$ y $\chi_{\overline{\mathbf{A}}}$ serán computables. Que $\chi_{\mathbf{A}}$ sea computable quiere decir que tenemos un algoritmo para detectar cuándo un elemento es de **A**. Pero que $\chi_{\overline{\mathbf{A}}}$ sea también computable quiere decir que tenemos además otro algoritmo para detectar cuándo un elemento es de $\overline{\mathbf{A}}$. Parece lógico inferir que combinando ambos algoritmos en paralelo tendremos un método infalible para determinar en cuál de ambos conjuntos reside un elemento cualquiera.

Sean entonces $\mathbf{e_1}$ y $\mathbf{e_2}$ dos índices de las funciones $\chi_{\mathbf{A}}$ y $\chi_{\overline{\mathbf{A}}}$ respectivamente. Demostraremos que $\mathbf{C_A}$ es computable (y por tanto que **A** es decidible) mediante el siguiente programa:

```
PASOS := 1;
while not T(E1, X1, PASOS) and not T(E2, X1, PASOS) loop
      PASOS:= suc(PASOS);
end loop;
X0 := T(E1, X1, PASOS);
```

Nótese que, a pesar de su aspecto, este programa computa efectivamente una función total, ya que para cualquier dato **x** necesariamente uno y sólo uno de los cómputos de $\chi_A(\mathbf{x})$ y $\chi_{\overline{A}}(\mathbf{x})$ será convergente ⌫

Este resultado podemos aplicarlo para demostrar que algunos conjuntos no son semidecidibles sin necesidad de hacer ninguna demostración de incomputabilidad. Tomemos por ejemplo el conjunto $\overline{\mathbf{K}}$ y veamos que no puede estar en Σ_1 por reducción al absurdo. Si $\overline{\mathbf{K}}$ fuese semidecidible, como ya sabemos que **K** lo es, podríamos aplicar la proposición 5 y deducir que **K** es decidible, lo cual sabemos que no es cierto. Por tanto $\overline{\mathbf{K}}$ constituye nuestro primer ejemplo de conjunto que no es semidecidible.

6.3.2 Propiedades de cierre de los conjuntos semidecidibles

Algunas de las propiedades de los conjuntos decidibles se verifican también para los semidecidibles, pero no podemos completar la demostración repitiendo exactamente los mismos pasos debido a que los programas no totales son mucho más difíciles de combinar que los totales. Además tenemos restricciones sintácticas, ya que las funciones características, al ser totales, pueden ser usadas en las macrocondiciones, cosa que nos está vedado para las funciones semicaracterísticas por no ser, en general, totales.

PROPOSICIÓN 6: La unión e intersección de conjuntos semidecidibles son también semidecidibles.

⌦ Sean **A** y **B** conjuntos semidecidibles. Por tanto, sus funciones semicaracterísticas χ_A y χ_B son computables Para ver que $\mathbf{A}\cap\mathbf{B}\in\Sigma_1$ utilizaremos el siguiente programa que computa la función $\chi_{A\cap B}$:

```
R := χA (X1);
R := χB (X1);
X0 := true;
```

Vemos que las dos primeras asignaciones actúan como filtros que sólo pueden superar con éxito los elementos de la intersección.

Para ver que $\mathbf{A}\cup\mathbf{B}\in\Sigma_1$ no podemos utilizar un método tan directo. Dada una entrada **x** cualquiera, tenemos el problema de decidir qué función semicaracterística le aplicamos primero. Si empezamos por calcular $\chi_A(\mathbf{x})$ corremos un grave riesgo, ya que **x** podría estar en **B**−**A**, y entonces divergiríamos sin tener oportunidad de comprobar que estaba en **B** y por tanto en la unión. Pero lo mismo sucedería si empezáramos con $\chi_B(\mathbf{x})$, ya que no podemos saber a priori si **x** está o no en **A**−**B**. La solución es aplicar ambas funciones en paralelo en lugar de en serie, para dar a ambas la oportunidad de detectar la presencia de **x** en **A** o **B**.

Como χ_A y χ_B son computables, sean $\mathbf{e_1}$ y $\mathbf{e_2}$ dos índices de las mismas. El programa que computa la función semicaracterística $\chi_{A\cup B}$ es:

```
PASOS := 1;
while  not  (T(E1, X1, PASOS) or T(E2, X1, PASOS))  loop
      PASOS := succ(PASOS);
end loop;
X0 := true;  ⌫
```

PROPOSICIÓN 7: El complementario de un conjunto semidecidible no tiene por qué ser decidible.

⌦ Podemos poner como contraejemplo el conjunto **K**. Hemos visto que es semidecidible pero que su complementario no puede serlo como consecuencia de la proposición 5 ⌫

En realidad la demostración aplicada a **K** para demostrar que $\overline{\mathbf{K}} \notin \Sigma_1$ puede extenderse a cualquier otro conjunto **A** que esté en la clase $\Sigma_1 - \Sigma_0$: si tenemos un procedimiento para detectar los elementos de **A** no podemos tenerlo también para los de $\overline{\mathbf{A}}$, porque entonces podríamos combinar ambos para construir un procedimiento de decisión para **A**. Por tanto $\overline{\mathbf{A}} \notin \Sigma_1$.

En general, recurrir a estudiar el complementario de un conjunto **A** es una buena táctica para ayudar a clasificarlo en una de las tres clases ilustradas en la figura 6.1, porque las combinaciones posibles son únicamente tres:

1. $\mathbf{A} \in \Sigma_0$ (y entonces también $\overline{\mathbf{A}} \in \Sigma_0$) Es posible un procedimiento de detección para los elementos de **A** y otro para los de $\overline{\mathbf{A}}$ (ambos son semidecidibles y por tanto decidibles)
2. $\mathbf{A} \in \Sigma_1 - \Sigma_0$ (y entonces $\overline{\mathbf{A}} \in \overline{\Sigma_1}$). Es posible un procedimiento de detección para los elementos de **A**, pero no para los de $\overline{\mathbf{A}}$ (el primero es semidecidible y el segundo no). También puede suceder el caso recíproco.
3. $\mathbf{A} \in \overline{\Sigma_1}$ (y entonces también $\overline{\mathbf{A}} \in \overline{\Sigma_1}$)[9]. No existen procedimientos algorítmicos de detección para ninguno de los dos conjuntos

A decir verdad con nuestras herramientas actuales aún no podemos probar que existen conjuntos de la clase 3, pero podemos adelantar que **TOT** será uno de ellos (no son semidecidibles ni él ni su complementario), tal como se indica en la figura 6.1.

[9] En realidad el complementario de **A** podría ser semidecidible, pero entonces estaríamos en el caso 2 en su versión recíproca.

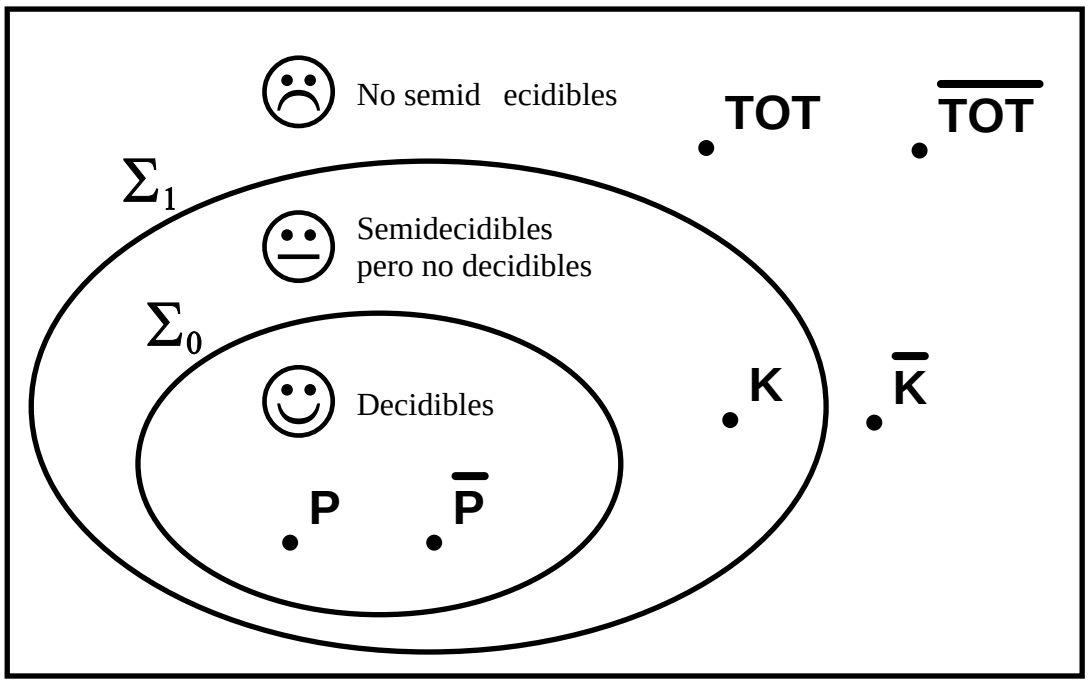

Figura 6.1: Clasificación de los conjuntos de acuerdo a su decidibilidad y/o semidecidibilidad Se aprecian tres ejemplos de las tres situaciones posibles para un conjunto y su complementario: ambos en Σ_0 (caso de **P**, conjunto de los números pares), uno en Σ_1-Σ_0 y el otro fuera de Σ_1 (caso de **K**, conjunto de los programas que convergen sobre sí mismos) o ambos fuera de Σ_1 (caso de **TOT**, conjunto de los programas totales)

Como en el caso de las enunciadas en los apartados precedente, estas propiedades se generalizan de forma natural a conjuntos de k-tuplas de palabras ($\mathbf{B}\subseteq\sum^{*k}$) o de objetos diferentes de las palabras ($\mathbf{B}\subseteq\mathbb{T}_1\times\mathbb{T}_2\times...\times\mathbb{T}_k$, donde $\mathbb{T}$, $\mathbb{T}_1$, $\mathbb{T}_2$,..., $\mathbb{T}_k$ son tipos de datos implementados). Las demostraciones son análogas y si no las incluimos es porque no aportan nada significativo.

6.3.3 Caracterización de los conjuntos semidecidibles

A diferencia de lo que sucede con los conjuntos decidibles, existen otras formas de caracterizar los conjuntos semidecidibles sin recurrir a su función semicaracterística que resultan extraordinariamente útiles cuando manipulamos dichos conjuntos. Procederemos a enunciarlas y a demostrar su validez en forma de teorema.

TEOREMA (DE CARACTERIZACIÓN DE LA CLASE Σ_1): Sea $\mathbf{A}\subseteq\sum^*$ un conjunto cualquiera. Las siguientes propiedades de **A** son equivalentes:

a) $\mathbf{A}\in\Sigma_1$.

b) **A** es el dominio de una función computable. Es decir, existe $\psi: \sum^* \longrightarrow \sum^*$ computable que verifica $dom(\psi)=\mathbf{A}$.

c) **A** es la cuantificación existencial (o proyección) de un predicado decidible de dos argumentos. Es decir, existe $\mathbf{P}: \sum^* \times \sum^* \to \mathbb{B}$ decidible que verifica $\mathbf{A} = \{ \mathbf{x} \in \sum^*: \exists \mathbf{y}\ \mathbf{P}(\mathbf{x},\mathbf{y}) \}$.

d) **A** es, o bien el conjunto vacío, o bien el rango de una función computable y total. Es decir, si $\mathbf{A} \neq \varnothing$ entonces existe una función $\mathbf{f}: \sum^* \to \sum^*$ computable y total que verifica $\text{ran}(\mathbf{f})=\mathbf{A}$.

e) **A** es el rango de una función computable cualquiera. Es decir, existe $\xi: \sum^* \dashrightarrow \sum^*$ computable que verifica $\text{ran}(\xi)=\mathbf{A}$.

⌦ Estas condiciones son en general muy diferentes entre sí. Más provechoso que demostrar su equivalencia dos a dos será realizar algún tipo de prueba circular. Si, por ejemplo, demostramos que **a)** $\Rightarrow$ **b)** $\Rightarrow$ **c)** $\Rightarrow$ **d)** $\Rightarrow$ **e)** $\Rightarrow$ **a)** habremos probado la equivalencia entre todas minimizando el esfuerzo.

a) $\Rightarrow$ **b)**

Partimos de que **A** es semidecidible, y debemos encontrar una función computable cualquiera cuyo dominio sea **A**.

Pero esto es trivial, ya que si $\mathbf{A} \in \Sigma_1$ entonces su función semicaracterística:

$$\chi_{\mathbf{A}}(\mathbf{x}) \cong \begin{cases} \text{true} & \mathbf{x} \in \mathbf{A} \\ \\ \perp & \text{c.c.} \end{cases}$$

es computable, y el dominio de esta funcion es precisamente **A**. Por tanto, si elegimos $\psi \cong \chi_{\mathbf{A}}$ ya tenemos la función computable buscada con $\text{dom}(\psi)=\mathbf{A}$.

b) $\Rightarrow$ **c)**

Partimos de que $\mathbf{A}=\text{dom}(\psi)$ con ψ computable, y debemos encontrar un predicado binario y decidible **P** que verifique $\mathbf{A} = \{ \mathbf{x} \in \sum^*: \exists \mathbf{y}\ \mathbf{P}(\mathbf{x},\mathbf{y}) \}$.

Vamos a desarrollar el significado de la expresión $\mathbf{x} \in \mathbf{A}$ por si conseguimos expresarlo en los términos indicados de cuantificación existencial. Lo único que sabemos es que:

$$\mathbf{x} \in \mathbf{A} \Leftrightarrow \mathbf{x} \in \text{dom}(\psi) \Leftrightarrow \psi(\mathbf{x})\downarrow$$

pero como ψ es computable podemos asumir que tiene un índice **e**, y así introducir la cuantificación existencial gracias al predicado **T** (que por cierto, es decidible):

$$\mathbf{x} \in \mathbf{A} \Leftrightarrow \varphi_{\mathbf{e}}(\mathbf{x})\downarrow \Leftrightarrow \exists \mathbf{y}\ \mathbf{T}(\mathbf{e},\mathbf{x},\mathbf{y})$$

y teniendo en cuenta que **e** es una constante, podemos definir $\mathbf{P}(\mathbf{x},\mathbf{y}) \equiv \mathbf{T}(\mathbf{e},\mathbf{x},\mathbf{y})$, que sin duda es decidible por tratarse de una particularización de **T**. De esta forma conseguimos $\mathbf{A} = \{ \mathbf{x} \in \sum^* : \exists \mathbf{y}\ \mathbf{P}(\mathbf{x},\mathbf{y}) \}$ con **P** decidible.

c) ⇒ d)

Partimos de que $\mathbf{A} = \{ \mathbf{x} \in \sum^* : \exists \mathbf{y}\ \mathbf{P}(\mathbf{x},\mathbf{y}) \}$ con **P** decidible, y debemos encontrar, bajo la suposición de adicional de que $\mathbf{A} \neq \varnothing$, una función $\mathbf{f}: \sum^* \to \sum^*$ computable y total que verifique $\text{ran}(\mathbf{f})=\mathbf{A}$.

Si **A** no es vacío ha de tener al menos un elemento $\mathbf{c} \in \mathbf{A}$. Definimos entonces la función:

$$\mathbf{g}(\mathbf{x},\mathbf{y}) = \begin{cases} \mathbf{x} & \mathbf{P}(\mathbf{x},\mathbf{y}) \\ \mathbf{c} & \neg\mathbf{P}(\mathbf{x},\mathbf{y}) \end{cases}$$

que resulta claramente total y computable, mediante el programa:

if P(X1, X2) **then** X0:=X1; **else** X0 := C; **end if**;

y cuyo su rango es

$$\text{ran}(\mathbf{g}) = \{ \mathbf{x} \in \sum^* : \exists \mathbf{y}\ \mathbf{P}(\mathbf{x},\mathbf{y}) \} \cup \{\mathbf{c}\} = \mathbf{A} \cup \{\mathbf{c}\} = \mathbf{A}$$

Basta ahora con definir la función $\mathbf{f}(\mathbf{x}) = \mathbf{g}(\text{decod}_{2,1}(\mathbf{x}), \text{decod}_{2,2}(\mathbf{x}))$. Mantendrá su computabilidad y totalidad por ser composición de funciones computables y totales, y su rango seguirá siendo el mismo. Por tanto **f** es la función que estábamos buscando.

d) ⇒ e)

En este caso partimos de que o bien $\mathbf{A}=\varnothing$ o bien $\mathbf{A}=\text{ran}(\mathbf{f})$ con **f** computable y total, y hemos de encontrar una función computable $\xi: \sum^* \dashrightarrow \sum^*$ que verifique $\text{ran}(\xi)=\mathbf{A}$.

Si estamos en el primer caso y $\mathbf{A}=\varnothing$ la función computable que buscamos sería $\xi \cong \perp\!\!\!\perp$, ya que es claro que en ese caso $\text{ran}(\xi) = \varnothing = \mathbf{A}$.

Si estamos en el segundo caso y $\mathbf{A}=\text{ran}(\mathbf{f})$ podemos tomar directamente $\xi \cong \mathbf{f}$, que es computable.

e) ⇒ a)

Partimos de que $\mathbf{A}=\text{ran}(\xi)$ con $\xi: \sum^* \dashrightarrow \sum^*$ computable y hemos de probar que $\mathbf{A} \in \Sigma_1$. Para demostrar que χ_A es computable observamos que:

$$\chi_A(\mathbf{x}) \cong \begin{cases} \text{true} & \mathbf{x} \in \mathbf{A} \Leftrightarrow \mathbf{x} \in \text{ran}(\xi) \Leftrightarrow \exists \mathbf{z}\ \xi(\mathbf{z}) = \mathbf{x} \\ \bot & \text{c.c.} \end{cases}$$

Para computar esta función podemos calcular en paralelo la serie de valores $\xi(0)$, $\xi(1)$, $\xi(2)$, $\xi(3)$, ... hasta dar con uno que coincida con **x**. Si no aparece nuestra búsqueda no terminará, pero ello carecerá de importancia, ya que eso querrá decir que **x**=ran(ξ). Sea entonces **b** un índice de la función computable ξ. El programa que computa χ_A será:

```
PAR := 0;
while not E(B, decod_2_1(PAR), decod_2_2(PAR), X1) loop
      PAR := succ(PAR);
end loop;
X0 := true;
```

por lo que $\mathbf{A} \in \Sigma_1$. Con esto completamos la demostración circular y probamos la equivalencia de las cinco condiciones, por lo que cualquiera de ellas constituye un caracterización de los conjuntos semidecidibles. ⌫

Las caracterizaciones **b)**-**e)** del teorema que acabamos de probar no son sino posibles definiciones alternativas de los conjuntos semidecidibles, y todas ellas nos indican cosas interesantes sobre tales conjuntos.

Por ejemplo, cuando **c)** nos dice que todo conjunto semidecidible **A** se puede poner como cuantificación existencial de un predicado decidible **P** nos está dando una idea muy intuitiva de la naturaleza de **A**. Nos dice que la condición de pertenencia $\mathbf{x} \in \mathbf{A}$ es siempre expresable en términos de una fórmula de tipo $\exists \mathbf{y}\ \mathbf{P}(\mathbf{x},\mathbf{y})$, lo que nos sugiere que para averiguar si **x** está en **A** hemos de realizar una *búsqueda* sobre posibles valores de **y** hasta dar con uno que satisfaga $\mathbf{P}(\mathbf{x},\mathbf{y})$. Esta noción de búsqueda traduce bastante bien la idea de que los elementos de un conjunto semidecidible son detectables, si la búsqueda tiene éxito, pero los ajenos al mismo quizá no lo sean, porque el método de búsqueda no termina para ellos. Quizá exista otro método alternativo para detectar esos otros elementos (si el conjunto además es decidible), o quizá no.

La caracterización **b)** nos asegura que los conjuntos semidecidibles y los dominios de funciones computables son la misma cosa. Dado que hemos asumido la notación $\mathbf{W_e} = \text{dom}(\varphi_e)$ nos encontramos con que podemos enumerar en forma de lista infinita la clase de todos los conjuntos semidecidibles $\Sigma_1 = \{ \mathbf{W_0}, \mathbf{W_1}, \mathbf{W_2}, \mathbf{W_3}, \mathbf{W_4}, \ldots \}$. Naturalmente en esta lista cada conjunto específico aparece repetido muchas veces: por ejemplo, el conjunto semidecidible $\sum^*$ figura en todas las posiciones asociadas a funciones totales (y sabemos que hay infinitas de ellas y que cada una aparece representada infinitas veces, porque hay infinitos programas

distintos para computarla). En virtud de esta enumeración podemos asociar *índices* a los conjuntos semidecidibles igual que lo hacíamos con las funciones computables, dado que $\mathbf{A} \in \Sigma_1 \Leftrightarrow \exists \mathbf{e}\ \mathbf{A} = \mathbf{W_e}$.

Algo muy similar sucede con la caracterización **e)**, pero en este caso nos basamos en la notación estándar $\mathbf{R_e} = \text{ran}(\varphi_e)$ para obtener una enumeración alternativa de la clase $\Sigma_1 = \{ \mathbf{R_0}, \mathbf{R_1}, \mathbf{R_2}, \mathbf{R_3}, \mathbf{R_4}, \ldots \}$.

Sin embargo, si alguna de las formulaciones de la semidecidibilidad merece la pena ser resaltada es la **d)**, que caracteriza un conjunto semidecidible como el rango de una función computable y total **f** salvo en el caso trivial en que el conjunto es vacío. Dado un conjunto semidecidible **A** el que podamos expresarlo como el rango de **f** nos proporciona una herramienta valiosísima para trabajar con el conjunto, ya que **f** nos permite enumerar todos los elementos del conjunto $\mathbf{A} = \text{ran}(\mathbf{f}) = \{ \mathbf{f}(0), \mathbf{f}(1), \mathbf{f}(2), \mathbf{f}(3), \mathbf{f}(4), \ldots \}$. Intuitivamente podemos extraer la enseñanza de que los conjuntos de Σ_1 se caracterizan por tener asociado un método computable para *listar* todos sus elementos. Esto quiere decir que si $\mathbf{x} \in \mathbf{A}$ entonces acabará apareciendo en esta lista (posiblemente en más de una posición), por lo que se denomina a **f** *función de enumeración* del conjunto **A**.

Uno de los aspectos más interesantes de esta propiedad es el uso que vamos a hacer de ella para, combinándola con el método de diagonalización, demostrar que un conjunto no es semidecidible.

6.3.4 Diagonalización y no semidecidibilidad

Hasta ahora nos hemos centrado en enunciar y propiedades que pueden ser útiles para demostrar que ciertos conjuntos son decidibles o semidecidibles, pero es de esperar que el desafío que más nos interesa desde el punto de vista de la Teoría de la Computabilidad sea justamente el contrario: probar que un conjunto **A** dado es indecidible o incluso que no es semidecidible. Para ello tendremos que demostrar la incomputabilidad de su función característica $\mathbf{C_A}$ o de su función semicaracterística χ_A, y dado que el único método que tenemos para demostrar incomputabilidades es el de diagonalización, a ese mecanismo nos tendremos que remitir.

Sin embargo, si nos fijamos con cierto detalle observaremos que el método de diagonalización ha sido aplicado hasta ahora para demostrar la incomputabilidad de funciones totales (típicamente predicados). Si seguimos haciéndolo de la misma manera parece razonable esperar que el método funcione bien para las funciones características, que son siempre totales, pero, ¿podemos esperar el mismo comportamiento para las funciones semicaracterísticas, que no lo son? La respuesta

es no, y la razón es muy simple: si partimos de una función no total ψ cuya incomputabilidad queremos probar, hemos de construir una función diagonal δ, y para ello hemos de utilizar la información que ψ nos proporciona en cada caso. ¿Y qué sucede en los casos en los que $\psi(\mathbf{x})\uparrow$? Pues que no existe dicha información y por tanto no puede ser utilizada.

Veamos un ejemplo: sea el conjunto $\mathbf{A} = \{ \mathbf{x}\in\sum^*: \varphi_{\mathbf{x}}$ es una función total y acotada $\} = \{ \mathbf{x}\in\sum^*: \exists\mathbf{k}\forall\mathbf{y}\ \varphi_{\mathbf{x}}(\mathbf{y})<\mathbf{k} \}$, y supongamos que queremos demostrar que no es semidecidible. Para ello hemos de conseguir probar que su función semicaracterística:

$$\chi_{\mathbf{A}}(\mathbf{x}) \cong \begin{cases} \text{true} & \mathbf{x} \in \mathbf{A} \ \Leftrightarrow\ \varphi_{\mathbf{x}} \text{ es total y acotada} \\ \perp & \text{c.c.} \end{cases}$$

no es computable

Supongamos que es computable y tratemos de estudiar la información que $\chi_{\mathbf{A}}(\mathbf{x})$ nos proporciona sobre lo que ocurre con cada función $\varphi_{\mathbf{x}}$:

- Si $\chi_{\mathbf{A}}(\mathbf{x})$ = true, sabremos que para toda entrada $\mathbf{y}$ se verifica que $\varphi_{\mathbf{x}}(\mathbf{y})$ converge, y además devuelve un valor menor que cierta cota propia de la función $\varphi_{\mathbf{x}}$. Entonces tendríamos numerosas alternativas para construir la función diagonal δ en el punto $\mathbf{x}$ para que sea diferente: podemos hacer $\delta(\mathbf{x}) = \varphi_{\mathbf{x}}(\mathbf{x})+1$ ó $\delta(\mathbf{x}) \cong \perp$, ya que sabemos que $\varphi_{\mathbf{x}}(\mathbf{x})$ converge.
- Si $\chi_{\mathbf{A}}(\mathbf{x})\uparrow$, no tendremos información ninguna sobre lo que sucede en $\varphi_{\mathbf{x}}(\mathbf{x})$. Como no obtenemos respuesta no sabemos siquiera si $\varphi_{\mathbf{x}}$ es total y acotada o no.

Entiéndase bien, la situación no es comparable a lo que nos sucedía en los casos de diagonalización asimétrica del apartado 5.5.3. *Entonces recibíamos una información escasa o insuficiente*, por lo que no podíamos aprovechar el resultado de la respuesta para decidir un valor adecuado de $\delta(\mathbf{x})$ que fuera diferente de $\varphi_{\mathbf{x}}(\mathbf{x})$. *Ahora no recibimos respuesta alguna*, con lo que el mero hecho de preguntar qué sucede con $\chi_{\mathbf{A}}(\mathbf{x})$ hace que nuestro proceso de determinación del valor de $\delta(\mathbf{x})$ tampoco tenga fin. Así, la función $\chi_{\mathbf{A}}$ se convierte en una trampa de la que no podemos salir en los casos en los que diverge. Lo malo es que $\delta(\mathbf{x})$ resulta indefinido en el momento más inoportuno, pues esto se da precisamente cuando $\mathbf{x}\notin\mathbf{A}$ y, por tanto, $\varphi_{\mathbf{x}}$ puede tranquilamente ser una función no total que diverja en el punto $\mathbf{x}$, con lo que no tenemos garantía alguna de que δ se diferencie de ella. La situación indicada se refleja en la tabla-ejemplo de la figura 6.2.

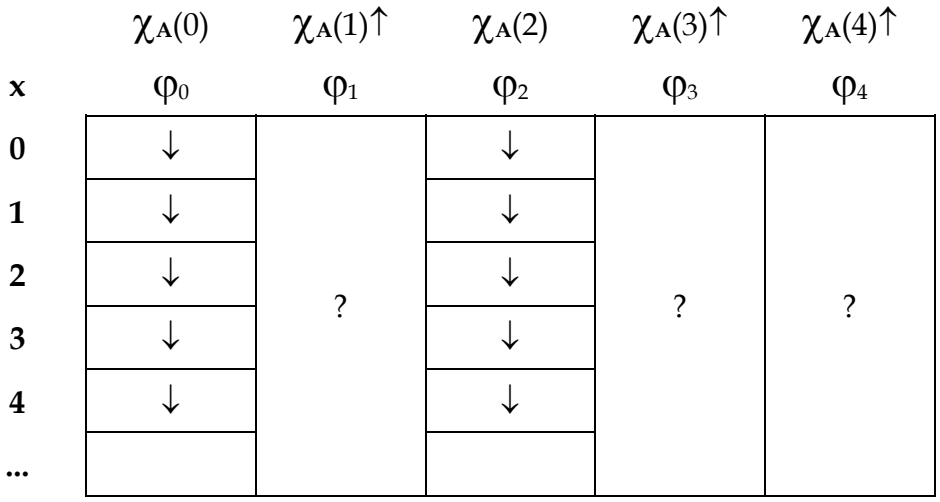

Figura 6.2: Tabla ejemplo que ilustra la inaplicabilidad del método de diagonalización tal como lo conocemos a una función no total.

Sin embargo, es posible modificar el método de diagonalización para aplicarlo en este y otros casos similares. La idea consiste en buscar la contradicción por otro camino. Cuando suponemos que **A** es semidecidible, en lugar de tomar la función χ_A para buscar la contradicción, utilizaremos el apartado **d)** del teorema de caracterización de la clase Σ_1, que nos dice (siempre que $A \neq \emptyset$) que existe una función computable y total **f** cuyo rango es precisamente **A**. La función **f**, al ser total, será mucho más segura de manipular.

Una vez establecida esta línea de razonamiento podemos argumentar que $A = ran(f) = \{ f(0), f(1), f(2), f(3), f(4), \dots \}$ y que, por tanto, la lista $\{ \varphi_{f(0)}, \varphi_{f(1)}, \varphi_{f(2)}, \varphi_{f(3)}, \varphi_{f(4)}, \dots \}$ contiene una enumeración completa de todas las funciones computables totales y acotadas (ya que todos los programas que computan dichas funciones están en **A**). Realizada esta finta nos planteamos la aplicación del método de diagonalización, pero en vez de tomar toda la lista de funciones computables e intentar definir una función δ diferente de todas ellas, utilizaremos únicamente la nueva lista restringida de funciones computables, totales y acotadas que nos proporciona la función **f**, y luego seguiremos el mismo proceso que en casos anteriores.

Vamos a ilustrar esta variante del método para probar que el mencionado conjunto $A = \{ x \in \sum^* : \varphi_x \text{ es una función total y acotada} \}$ no es semidecidible.

JUSTIFICACIÓN: Intuitivamente parece razonable sospechar que $A \notin \Sigma_1$, ya que para detectar que $x \in A$ necesitaríamos probar que $\exists k \forall y \ \varphi_x(y) < k$, lo cual conllevaría infinitas comprobaciones que nos impiden dar una respuesta en un tiempo finito.

PLANIFICACIÓN: Suponemos, como hipótesis, que el conjunto **A** es semidecidible. Sabemos que **A** no es vacío, porque todos los programas que

computan funciones constantes están en él. Entonces sabemos también, por el teorema de caracterización, que **A** coincide con el rango de una función computable y total **f**. Esta función nos proporciona una enumeración computable de todas las funciones computables, totales y acotadas de la forma $\mathcal{L} = \{ \varphi_{\mathbf{f}(0)}, \varphi_{\mathbf{f}(1)}, \varphi_{\mathbf{f}(2)}, \varphi_{\mathbf{f}(3)}, \varphi_{\mathbf{f}(4)}, \ldots \}$.

Ahora definiremos la función diagonal **d** de manera que, por un lado, sea *computable , total y acotada*, pero por otro lado sea diferente de todas la funciones de la lista $\mathcal{L}$. Como las funciones de $\mathcal{L}$ son muy fáciles de manejar al ser todas totales, podremos utilizar el valor de la diagonal en el convencimiento de que siempre se va a verificar $\varphi_{\mathbf{f}(\mathbf{x})}(\mathbf{x})\downarrow$. Así **d** será diferente de $\varphi_{\mathbf{f}(0)}$ en el punto 0, de $\varphi_{\mathbf{f}(1)}$ en el punto 1, etc. Como ya hemos comentado antes, la manera concreta de diferenciarse puede establecerse de muchas formas, pero con cuidado. No podemos hacer $\mathbf{d}(\mathbf{x})\uparrow$, porque entonces **d** no sería total y se nos arruinaría la argumentación. Tampoco podemos introducir uno de los mecanismos clásicos de variación como $\mathbf{d}(\mathbf{x}) = \varphi_{\mathbf{f}(\mathbf{x})}(\mathbf{x})+1$, ya que entonces **d** no será acotada[10].

Para conseguir que sí lo sea haremos que $\mathbf{d}(\mathbf{x})$ sólo pueda tomar dos valores. En general definiremos $\mathbf{d}(\mathbf{x}) = 3$, siempre y cuando $\varphi_{\mathbf{f}(\mathbf{x})}(\mathbf{x})$ no sea precisamente 3, porque en ese caso le asignaremos el valor 0 para que sea diferente. De esta forma **d** será total y acotada, porque siempre convergirá devolviendo valores menores o iguales que 3. La idea se ilustra en la tabla-ejemplo de la figura 6.3.

d	**x**	$\varphi_{\mathbf{f}(0)}$	$\varphi_{\mathbf{f}(1)}$	$\varphi_{\mathbf{f}(2)}$	$\varphi_{\mathbf{f}(3)}$	$\varphi_{\mathbf{f}(4)}$	$\varphi_{\mathbf{f}(5)}$
3	**0**	0	≤ 25	≤ 3	≤ 31	≤ 6	≤ 1917
3	**1**	≤ 0	20	≤ 3	≤ 31	≤ 6	≤ 1917
0	**2**	≤ 0	≤ 25	3	≤ 31	≤ 6	≤ 1917
3	**3**	≤ 0	≤ 25	≤ 3	2	≤ 6	≤ 1917
0	**4**	≤ 0	≤ 25	≤ 3	≤ 31	3	≤ 1917
3	**5**	≤ 0	≤ 25	≤ 3	≤ 31	≤ 6	733

Figura 6.3: Tabla-ejemplo para la construcción de una función diagonal **d** sobre la lista restringida de funciones computables proporcionada por el rango de **f**. Al ser **f** computable y total podemos calcular siempre $\varphi_{\mathbf{f}(\mathbf{x})}(\mathbf{x})$, que a su vez siempre estará

[10] En efecto, aunque todas las funciones de la lista estén acotadas eso no implica que exista una cota común para todas ellas. Si hiciéramos $\mathbf{d}(\mathbf{x}) = \varphi_{\mathbf{f}(\mathbf{x})}(\mathbf{x})+1$ no sería acotada porque, para cualquier posible cota **k** tendríamos que la función constante $\mathbf{g}(\mathbf{x}) = \mathbf{k}$ sería computable, total y acotada, y estaría en la lista $\mathcal{L}$. Digamos que entonces **g** podría ponerse de la forma $\varphi_{\mathbf{f}(\mathbf{a})}$ para algún **a**, y entonces $\mathbf{d}(\mathbf{a}) = \varphi_{\mathbf{f}(\mathbf{a})}(\mathbf{a})+1 = \mathbf{k}+1$, valor que superaría la cota **k**.

definida por ser $\varphi_{f(x)}$ una función total y acotada. Ello nos permitirá dar un valor diferente para $\mathbf{d}(\mathbf{x})$.

CONSTRUCCIÓN: Definimos formalmente la función **d**:

$$\mathbf{d}(\mathbf{x}) = \begin{cases} 0 & \varphi_{\mathbf{f(x)}}(\mathbf{x}) = 3 \\ 3 & \text{c.c.} \end{cases}$$

Dado que **f** es computable, **d** también lo será, como lo demuestra el programa:

```
R := Φ(f(X1), X1);
X0 := 3;
if R = 3 then X0 := 0; end if;
```

Por ser una función computable, existirá un índice $\mathbf{e}\in\sum^*$, tal que $\mathbf{d} \cong \varphi_{\mathbf{e}}$. Pero lo que en realidad nos interesa no es acreditar que está en la lista $\mathcal{C}$ de las funciones computables, sino el la lista $\mathcal{L}$ de las computables, totales y acotadas. Ya hemos argumentado al definirla que $\forall \mathbf{y}\ (\mathbf{d}(\mathbf{y})=3 \vee \mathbf{d}(\mathbf{y})=0)$. Por tanto su índice **e** corresponde a una función total y acotada, lo que implica que $\mathbf{e}\in\mathbf{A}$. Como **A** es a su vez el rango de **f**, tendremos que existe una palabra $\mathbf{m}\in\sum^*$ que cumple $\mathbf{f}(\mathbf{m})=\mathbf{e}$. Hemos, por tanto, construido una función cuyo programa está en **A**, pero por otro lado la técnica que hemos utilizado para su construcción nos asegura que el comportamiento de dicho programa es diferente al de todos los de la lista en la diagonal. Esto nos permitirá establecer una contradicción.

CONTRADICCIÓN: Veamos que, a pesar de nuestro razonamiento, es imposible que exista ese índice **e**. Hemos construido **d** de forma que sea computable ($\mathbf{d} \cong \varphi_{\mathbf{e}}$), y además total y acotada ($\mathbf{f}(\mathbf{m})=\mathbf{e}$). Al mismo tiempo la hemos hecho diferente de todas las totales y acotadas $\varphi_{\mathbf{f(x)}}$ en el punto **x**. En consecuencia también se diferenciará de $\varphi_{\mathbf{f(m)}}$ en el punto **m**. Veamos que obtenemos una contradicción si aplicamos la función **d** en el punto **m**.

Caso 1: $\mathbf{d}(\mathbf{m})=3 \overset{\text{def. } \mathbf{d}}{\Rightarrow} \neg\varphi_{\mathbf{f(m)}}(\mathbf{m})=3 \overset{\mathbf{f}(\mathbf{m})=\mathbf{e}}{\Rightarrow} \neg\varphi_{\mathbf{e}}(\mathbf{m})=3 \overset{\mathbf{d}\cong\varphi_{\mathbf{e}}}{\Rightarrow} \neg\mathbf{d}(\mathbf{m})=3$ ↯

Caso 2: $\mathbf{d}(\mathbf{m})=0 \overset{\text{def. } \mathbf{d}}{\Rightarrow} \varphi_{\mathbf{f(m)}}(\mathbf{m})=3 \overset{\mathbf{f}(\mathbf{m})=\mathbf{e}}{\Rightarrow} \varphi_{\mathbf{e}}(\mathbf{m})=3 \overset{\mathbf{d}\cong\varphi_{\mathbf{e}}}{\Rightarrow} \mathbf{d}(\mathbf{m})=3$ ↯

Podemos concluir que la contradicción surge de la hipótesis de partida, es decir, de suponer que **A** es semidecidible, con lo cual queda demostrado que $\mathbf{A}\notin\Sigma_1$.

Esta extensión del método de diagonalización nos permitirá clasificar como no semidecidibles un buen número de conjuntos.

7. Conclusiones

A lo largo de estas páginas hemos respondido a una preguntas esencial para entender los fundamentos teóricos de la Informática: ¿existen problemas que no podamos resolver algorítmicamente? No nos hemos conformado con responder afirmativamente demostrando la existencia de tales problemas, sino que también hemos estudiado algunos ejemplos concretos, destacando como fundamental el problema de parada, es decir, el problema de determinar *a priori* cuándo un programa va a terminar su ejecución. En definitiva, hemos tenido ocasión de validar una ley fundamental de la computación: la de que no existen atajos para anticipar el comportamiento de los programas, el cuál es intrínsecamente impredecible. La forma más directa que tenemos, *en general*, de saber el resultado de una computación es seguirla paso por paso. De acuerdo con este resultado podemos verificar que un programa termina, pero no hay método finito de comprobar que no lo hace.

Esta impredecibilidad tiene algunas consecuencias prácticas de primer orden. Por ejemplo, sabemos que los compiladores son muy eficaces en la detección de cierto tipo de fallos en los programas (los llamados errores sintácticos), mientras que se muestran incompetentes para predecir otras contingencias que a simple vista no parecen implicar mucha complejidad (como, por ejemplo, el que el índice de una tabla se salga de rango). Esto se debe a que los errores en ejecución tienen en su naturaleza la misma indecidibilidad que el problema de parada: el único método general para saber si se producen o no es realizar la ejecución del programa ¡que es, en el fondo, lo que hace el compilador: como es imposible determinar de antemano dichos errores nos larga el programa para que nos encontremos con ellos en ejecución, si es que se dan!

Existen otros problemas relacionados con la predicción de los comportamientos de los programas que afectan muy directamente al proceso de producción de *software*. No es posible determinar en general si un programa se va a atener a unas especificaciones dadas (es decir, si va a hacer lo que se ha previsto al diseñarlo). Ni siquiera es posible determinar si dos programas sirven para lo mismo, por lo que no podemos construir herramientas que comprueben si con los últimos refinamientos o cambios hechos a un programa, la nueva versión sigue teniendo el mismo comportamiento entrada/salida que antes.

Otro problema relacionado con la semántica de los programas que también ha resultado incomputable es el de, dado un programa, encontrar la versión más corta que resuelva el mismo problema. Y así podríamos seguir enumerando ejemplos (no olvidemos que existen infinitos conjuntos indecidibles).

De hecho, hay un resultado muy llamativo (y no muy difícil de probar) según el cuál ***todas*** las propiedades relacionadas con el comportamiento funcional de los programas son indecidibles. Es decir, que cualquier pregunta que se refiera a la relación entre las entradas y las salidas de un algoritmo está condenada a no poder ser resuelta algorítmicamente. Resulta hasta cierto punto paradójico para la Ingeniería Informática, que ha demostrado ser una herramienta tan útil para resolver problemas en tantos campos, que sea en su propia área donde más se dejan sentir sus límites.

Para abordar la incomputabilidad de todos estos problemas la técnica de diagonalización resulta, empero, insuficiente. En muchos casos es necesaria otra llamada de ***reducción***, que permite relacionar la indecidibilidad de unos problemas con la de otros estableciendo una jerarquía de "dificultad computacional" cuyos elementos inferiores hemos visto en este trabajo. Así, los conjuntos decidibles son los más "sencillos" posibles, quedando por encima de ellos los semidecidibles que no son decidibles. Pero es posible ir más allá, dado que, dentro de los conjuntos no semidecidibles también los hay más o menos "difíciles", estableciéndose de hecho una jerarquía infinita. Es decir, que por extraño que parezca existen conjuntos "más indecidibles" que otros, y además se pueden encontrar con un "grado de indecidibilidad" tan alto como se desee.

Sin embargo, aunque resulta sorprendente sumergirse en el mundo de la incomputabilidad y descubrir la enorme cantidad de problemas que podemos encontrar en él, no resultan menos destacables algunas ideas que han surgido en torno a los problemas que sí son computables. No debemos olvidar que la computación sigue siendo la poderosísima herramienta que estamos acostumbrados a utilizar para resolver difíciles problemas. Lo que sucede es que ahora podemos verla con una óptica algo diferente, porque hemos comprobado de cerca su extremada sencillez: un lenguaje de lo más humilde es capaz de expresar los mismos algoritmos que el más barroco entorno de programación. La experiencia en Informática ha demostrado que un conjunto de operaciones muy elemental es suficiente para programar todas las funciones computables, que sabemos que se pueden enumerar en forma de lista. Si hubiéramos partido de otro mecanismo de programación o de otros modelos alternativos de cómputo habríamos llegado igualmente a un de ***sistema de programación universal y aceptable***, que aunque diferente, nos habría permitido establecer los mismos resultados aquí expuestos.

Apéndice A: El lenguaje de los programas-while

Los programas-while están constituidos por instrucciones que operan sobre palabras construidas a partir de un alfabeto $\sum = \{\mathbf{a_1}, \ldots, \mathbf{a_n}\}$. Para referenciar dichas palabras el programa utiliza variables de la forma **X0**, **X1**, **X2**, ... No existe límite en el número de variables que un programa puede utilizar, ni tampoco en el tamaño que dichas palabras pueden alcanzar durante la ejecución del programa.

Denotamos por $\mathbb{W}$ el conjunto de los programas-while, que se definen inductivamente de la siguiente manera:

Si ...	**entonces...**
$\mathbf{i} \in \mathbb{N}$	XI := ε;
$\mathbf{i}, \mathbf{j} \in \mathbb{N}$ $\mathbf{s} \in \sum$	XI := $\text{cons}_{\mathbf{s}}$(XJ);
$\mathbf{i}, \mathbf{j} \in \mathbb{N}$	XI := cdr (XJ);
$\mathbf{P_1}, \mathbf{P_2} \in \mathbb{W}$	P_1 P_2
$\mathbf{i} \in \mathbb{N}$ $\mathbf{s} \in \sum$ $Q \in \mathbb{W}$	**if** $\text{car}_{\mathbf{s}}$?(XI) **then** Q **end if**;
$\mathbf{i} \in \mathbb{V}$ $Q \in \mathbb{W}$	**while** nonem?(XI) **loop** Q **end loop**;
	... es un programa-while

La semántica de un programa-while es la siguiente:

1. Si el programa tiene **n** datos, se supone que al comenzar a operar estos ya se encuentran almacenados en las variables **X1**, **X2**, …, **XN**
2. El resto de las variables del programa se inicializan con el valor ε (palabra vacía)
3. Se ejecutan las instrucciones del programa sobre las variables
4. Cuando el programa termina (si es que lo hace), se considera como único resultado de la computación el contenido final de la variable **X0**
5. Si por el contrario el programa no termina, se considera indefinido el resultado de la computación

El significado informal de cada uno de los programas definidos es:

XI := ε;	Introducir la palabra vacía en la variable **XI**
XI := $cons_{\mathbf{s}}$(XJ);	Añadir una **s** por la izquierda a la palabra contenida en **XJ** y asignar el valor resultante a **XI**
XI := cdr (XJ);	Eliminar el primer símbolo por la izquierda a la palabra contenida en **XJ** y asignar el valor resultante a **XI**
P_1 P_2	Ejecutar secuencialmente los programas $\mathbf{P_1}$ y $\mathbf{P_2}$
if $car_{\mathbf{s}}$?(XI) **then** Q **end if**;	Ejecutar el programa **Q** si el primer símbolo por la izquierda de **XI** es **s**
while nonem?(XI) **loop** Q **end loop**;	Mientras el contenido de **XI** sea una palabra no vacía, ejecutar Q

Apéndice B: Macros

Una macro es un mecanismo de abreviatura para programas-while que evita la repetición de código. Para completar la definición de cada tipo de macro es necesario proporcionar su ***expansión***, es decir, la forma en que esa macro puede ser convertida en un programa-while equivalente [IIS 96]. Se considera entonces que la macro no es más que una abreviatura de dicho programa-while.

Utilizando macros podemos escribir instrucciones con el siguiente formato:

- $U := \psi(V_1, ..., V_K)$;

  donde **U**, $\mathbf{V_1}$, ..., $\mathbf{V_K}$ pueden ser variables o identificadores cualesquiera (llamadas ***macrovariables*** y usadas a efectos nemotécnicos), y ψ es una función obligatoriamente while-computable. La utilidad de estas macros (cuya parte derecha se denomina ***macroexpresión***) es que permiten ir reutilizando las funciones que vamos programando para ser utilizadas como si ya estuvieran incorporadas al lenguaje[11]. En el apéndice C recopilamos una lista de la mayoría de las funciones utilizadas en los programas de este documento y cuya while-computabilidad ya fue demostrada [IIS 96].

- **if** $R(V_1, ..., V_K)$ **then** Q **end if**;
  ó
  while $R(V_1, ..., V_K)$ **loop** Q **end loop**;

  donde **R** es un predicado obligatoriamente while-decidible y como antes, $\mathbf{V_1}$, ..., $\mathbf{V_K}$ pueden ser variables o macrovariables, y **Q** es un macroprograma. La utilidad de las ***macrocondiciones*** incluidas en estas macros es que permiten ir reutilizando los predicados que vamos programando para ser utilizados como si ya estuvieran incorporados al lenguaje[11]. En el apéndice C recopilamos una lista con la mayoría de predicados utilizados en los programas de este documento y cuya while-decidibilidad ya fue demostrada [IIS 96].

- **if** B_1 **then** Q_1 **elsif** B_2 **then** Q_2 **... elsif** B_n **then** Q_n **else** Q_{n+1} **end if**;

  donde $\mathbf{B_1}$, ..., $\mathbf{B_n}$ son macroexpresiones y $\mathbf{Q_1}$, ..., $\mathbf{Q_{n+1}}$ son macroprogramas.

---

[11] Al utilizar funciones while-computables en las macroexpresiones y predicados while-decidibles en las macrocondiciones procuramos respetar al máximo las notaciones más usuales en los lenguajes de programación, heredadas a su vez de las convenciones matemáticas. Así, las operaciones como la función concatenación & o el predicado igualdad = se utilizan en su notación infija habitual: así X7:=AUX & X2; en vez de X7:=&(AUX, X2);. Análogamente la invocación a las funciones while-computables constantes e identidad se realizan del modo tradicional: así escribiremos X4:=IND; y P:='abbac';.

- **for** V **in** A..B **loop** Q **end loop**;

  donde **V** es una variable o macrovariable, **A** y **B** son macroexpresiones y **Q** es un macroprograma que no incluye ninguna asignación que pueda modificar **V**.

Con estas instrucciones los programas que escribimos adquieren un aspecto más cercano a los formulados mediante lenguajes de programación más habituales para estudiantes de Ingeniería Informática, pero es importante recordar que para cada macroprograma existe un programa-while equivalente.

Apéndice C: Lista de funciones computables y predicados decidibles

A continuación relacionamos las funciones y predicados que se utilizan en el texto asumiendo su while-computabilidad y while-decidibilidad como demostradas previamente [IIS 96]. En primer lugar incluimos las operaciones que utilizamos para trabajar con palabras sobre un alfabeto genérico $\sum = \{\mathbf{a_1}, \ldots, \mathbf{a_n}\}$.

a) Operaciones más usuales con palabras

- La función $cons_s : \sum^* \to \sum^*$ se define como

$$\mathrm{cons}_{\mathbf{s}}(\mathbf{w}) = \mathbf{s} \bullet \mathbf{w}$$

- La función $cdr : \sum^* \to \sum^*$ se define como

$$\mathrm{cdr}(\mathbf{w}) = \begin{cases} \varepsilon & \mathbf{w} = \varepsilon \\ \mathbf{v} & \exists \mathbf{s}\ \mathrm{w} = \mathbf{s} \bullet \mathbf{v} \end{cases}$$

- El predicado $car_s? : \sum^* \to \mathbb{B}$ se define como

$$\mathrm{car}_{\mathbf{s}}?(\mathbf{w}) = \begin{cases} \mathrm{true} & \mathbf{w} = \mathbf{s} \bullet \mathbf{v} \\ \mathrm{false} & \text{c.c.} \end{cases}$$

- El predicado $nonem? : \sum^* \to \mathbb{B}$ se define como

$$\mathrm{nonem}?(\mathbf{w}) = \begin{cases} \mathrm{false} & \mathbf{w} = \varepsilon \\ \mathrm{true} & \mathbf{w} \neq \varepsilon \end{cases}$$

- La función *vacía* $\perp\!\!\!\perp : \sum^* \to \sum^*$ se define como

$$\perp\!\!\!\perp(\mathbf{w}) \cong \perp$$

- La función *inversa* $\cdot^R : \sum^* \to \sum^*$ se define como

$$\mathbf{w}^R = \begin{cases} \varepsilon & \mathbf{w} = \varepsilon \\ \mathbf{u}^R \bullet \mathbf{s} & \mathbf{w} = \mathbf{s} \bullet \mathbf{u} \end{cases}$$

- La función *concatenación* $\cdot \& \cdot : \sum^* \times \sum^* \to \sum^*$ se define como

$$\varepsilon \,\&\, \mathbf{y} = \mathbf{y}$$

$$\mathbf{s} \,\&\, \mathbf{y} = \text{cons}_{\mathbf{s}}(\mathbf{y})$$

$$\text{cons}_{\mathbf{s}}(\mathbf{x}) \,\&\, \mathbf{y} = \text{cons}_{\mathbf{s}}(\mathbf{x} \,\&\, \mathbf{y})$$

Esta función también es usada en el texto (fuera de los programas) con su notación algebraica, es decir como **x•y** en lugar de **x**&**y**.

- La función *primero* : $\sum^* \to \sum^*$ se define como

$$\text{primero}(\mathbf{w}) = \begin{cases} \varepsilon & \mathbf{w} = \varepsilon \\ \mathbf{s} & \mathbf{w} = \mathbf{s} \bullet \mathbf{u} \end{cases}$$

- Para cada palabra **w**, la función *constante* $K_{\mathbf{w}}$: $\sum^* \to \sum^*$ se define como

$$K_{\mathbf{w}}(\mathbf{x}) = \mathbf{w}$$

- La función *sig* : $\sum^* \to \sum^*$ que dada una palabra devuelve la siguiente según el orden que a) respeta la longitud, y b) entre las de igual longitud sigue el orden lexicográfico. Es decir, se define como:

$$\text{sig}(\varepsilon) = '\mathbf{a_1}'$$

$$\text{sig}(\mathbf{w} \bullet \mathbf{a_i}) = \begin{cases} \mathbf{w} \bullet \mathbf{a}_{\mathbf{i}+1} & \mathbf{i} < \mathbf{n} \\ \text{sig}(\mathbf{w}) \bullet \mathbf{a}_1 & \mathbf{i} = \mathbf{n} \end{cases}$$

- La función *ant* : $\sum^* \to \sum^*$ que devuelve la palabra anterior según el orden definido en $\sum^*$, se define como

$$\text{ant}(\mathbf{x}) = \begin{cases} \varepsilon & \mathbf{x} = \varepsilon \\ \text{sig}^{-1}(\mathbf{x}) & \mathbf{x} \neq \varepsilon \end{cases}$$

b) Funciones de código

Definimos la función cod^2 : $\sum^* \times \sum^* \to \sum^*$, de ***codificación*** que permite asociar a cada par de palabras una única palabra que lo representa de forma unívoca. Se define inductivamente como sigue:

$$\text{cod}^2(\varepsilon,\varepsilon) = \varepsilon$$

$$\text{cod}^2(\text{sig}(\mathbf{w}), \varepsilon) = \text{sig}(\text{cod}^2(\varepsilon, \mathbf{w}))$$

$$\text{cod}^2(\mathbf{v}, \text{sig}(\mathbf{w})) = \text{sig}(\text{cod}^2(\text{sig}(\mathbf{v}), \mathbf{w})$$

La función de codificación es extraordinariamente útil para representar estructuras de datos en nuestro lenguaje.

Para recuperar los componentes de un par a partir de su código se definen dos funciones inversas de **decodificación** $decod_{2,1}$, $decod_{2,2}$: $\sum^* \to \sum^*$ que cumplen:

$$cod^2(decod_{2,1}(\mathbf{w}), decod_{2,2}(\mathbf{w})) = \mathbf{w}$$

Podemos extender la idea para definir codificaciones de tuplas de **k** palabras. Para cada $\mathbf{k}\geq 1$ las funciones de codificación cod^k : $\sum^{*\mathbf{k}} \to \sum^*$ se definen de manera inductiva sobre el número de argumentos:

$$cod^1(\mathbf{z}) = \mathbf{z}$$

$$cod^{k+1}(\mathbf{z_1}, \dots, \mathbf{z_{k+1}}) = cod^2(\mathbf{z_1}, cod^k(\mathbf{z_2}, \dots, \mathbf{z_{k+1}})) = cod^2(\mathbf{z_1}, cod^2(\mathbf{z_2}, \dots cod^2(\mathbf{z_{k-1}}, \mathbf{z_k})\dots)))$$

Todas las funciones de codificación son biyectivas (es decir, totales, inyectivas y sobreyectivas) [IIS 96], y disponen de sus propias funciones de decodificación $decod_{k,i}$: $\sum^* \to \sum^*$ que nos devuelven el **i**-ésimo elemento de la **k**-tupla codificada por una cierta palabra ($1\leq \mathbf{i}\leq \mathbf{k}$). Es decir, se definen como:

$$decod_{\mathbf{k},\mathbf{i}}(\mathbf{w}) = \begin{cases} \mathbf{w} & \mathbf{i}=1 \ \wedge \ \mathbf{k}=1 \\ decod_{2,1}(\mathbf{w}) & \mathbf{i}=1 \ \wedge \ \mathbf{k}>1 \\ decod_{\mathbf{k}-1,\mathbf{i}-1}(decod_{2,2}(\mathbf{w})) & \mathbf{i}>1 \ \wedge \ \mathbf{k}>1 \end{cases}$$

c) Funciones asociadas a otros tipos de datos

Tenemos otro convenio para poder utilizar en nuestros macroprogramas operaciones correspondientes a otros tipos de datos distintos de las palabras: los naturales $\mathbb{N}$, los booleanos $\mathbb{B}$, las pilas $\mathbb{P}$, los vectores dinámicos $\mathbb{V}$ y los propios programas-while $\mathbb{W}$. Gracias al mismo podremos utilizar valores de dichos tipos en nuestros programas y manipularlos. Naturalmente estos no son sino abreviaturas de palabras que representan dichos valores.

Para trabajar con los números naturales además de la operación constructora *succ*, (que suma 1) utilizamos algunas de las operaciones básicas: *pred* (que resta 1, con la particularidad de que *pred*(0)=0), suma (+), producto (*), divisón (/) y potencia (**). También usaremos los comparadores de orden (<, ≤, >, ≥).

Los booleanos se definen para implementar operaciones lógicas entre las que utilizamos la negación (¬), la conjunción (∧), la disyunción (∨) y la equivalencia o igualdad lógica (↔).

Las pilas y los vectores dinámicos son dispositivos de almacenamiento de datos, palabras en este caso. Las primeras se forman por la agregación ordenada de cero o más cadenas de caracteres y las representamos listando sus componentes entre los símbolos "<" y "]", por ejemplo < **aba**, **bb**, ε, **a**] ó <] (denominada ésta *pila vacía*). En los vectores dinámicos, a diferencia de las pilas, cualquiera de sus componentes es accesible en cualquier momento mediante su índice. Los vectores están indexados partir de 0, y todo vector tiene al menos un componente. Representamos un vector dinámico listando sus componentes entre los símbolos "(" y ")", por ejemplo (**aba**, **bb**, ε, **a**).

Para el tipo de datos pila utilizamos sus operaciones básicas habituales:

- La función *empilar* : $\sum^* \times \mathbb{P} \to \mathbb{P}$, que añade una nueva palabra a la cima de una pila:

$$\text{empilar}(\mathbf{v}, <\mathbf{z_1}, \ldots, \mathbf{z_n}]) = <\mathbf{v}, \mathbf{z_1}, \ldots, \mathbf{z_n}]$$

- El predicado *P_vacía?* : $\mathbb{P} \to \mathbb{B}$, que indica cuándo una pila no contiene elementos:

$$\text{P\_vacía?}(<\mathbf{z_1}, \ldots, \mathbf{z_n}]) = \text{true} \Leftrightarrow \mathbf{n}=0$$

- La función *cima* : $\mathbb{P} \to \sum^*$, que extrae el último elemento introducido en una pila:

$$\text{cima}(<]) \cong \perp$$

$$\text{cima}(\text{empilar}(\mathbf{v}, <\mathbf{z_1}, \ldots, \mathbf{z_n}])) = \mathbf{v}$$

- La función *desempilar* : $\mathbb{P} \to \mathbb{P}$, que elimina la cima de una pila:

$$\text{desempilar}(<]) \cong \perp$$

$$\text{desempilar}(\text{empilar}(\mathbf{v}, <\mathbf{z_1}, \ldots, \mathbf{z_n}])) = <\mathbf{z_1}, \ldots, \mathbf{z_n}]$$

En cuanto a los vectores dinámicos consideraremos las siguientes operaciones:

- La función *ult_índice* : $\mathbb{V} \to \mathbb{N}$, que indica la posición del último elemento indexable de un vector (es decir, uno menos que su número de componentes):

$$\text{ult\_índice}((\mathbf{z_0}, \mathbf{z_1}, \ldots, \mathbf{z_n})) = \mathbf{n}$$

- La función *acceso* : $\mathbb{V} \times \mathbb{N} \to \sum^*$, que permite recuperar un componente de un vector a partir de su índice:

$$\text{acceso}((\mathbf{z_0}, \mathbf{z_1}, \ldots, \mathbf{z_i}, \ldots, \mathbf{z_n}), \mathbf{i}) = \mathbf{z_I} \quad \text{si } 0 \leq \mathbf{i} \leq \mathbf{n}$$

$$\text{acceso}((\mathbf{z_0}, \mathbf{z_1}, ..., \mathbf{z_n}), \mathbf{i}) \cong \perp, \quad \text{si } \mathbf{i>n}$$

En los macroprogramas, para indicar acceso(**v**, **i**) en general se utilizará la expresión **v**(**i**).

- La función *modifica* : $\mathbb{V} \times \mathbb{N} \times \sum^* \to \mathbb{V}$, que introduce un nuevo valor en una posición concreta de un vector devolviendo el vector resultante:

$$\text{modifica}((\mathbf{z_0}, \mathbf{z_1}, ..., \mathbf{z_i}, \ldots, \mathbf{z_n}), \mathbf{i}, \mathbf{w}) = ((\mathbf{z_0}, \mathbf{z_1}, ..., \mathbf{z_{i-1}}, \mathbf{z_i}, \mathbf{z_{i+1}}, \ldots, \mathbf{z_n}) \quad \text{si } \mathbf{i} \leq \mathbf{n}$$

$$\text{modifica}((\mathbf{z_0}, \mathbf{z_1}, ..., \mathbf{z_n}), \mathbf{i}, \mathbf{w}) \cong (\mathbf{z_0}, \mathbf{z_1}, ..., \mathbf{z_n}, \varepsilon, \ldots, \varepsilon, \mathbf{w}) \quad \text{si } \mathbf{i>n}$$

Nótese que, mientras que la operación *acceso* produce error cuando el índice es mayor de lo permitido, en el caso de *modifica* provoca la ampliación del vector dinámico para dar cabida al nuevo valor en la posición indicada.

Las frecuentes asignaciones de la forma V:= modifica(V, I, W); serán reemplazadas por la macro V(I):= W;.

d) Otras formas de utilizar operaciones en los macroprogramas

Algunas funciones y predicados pueden definirse para cualquier tipo de datos como la función vacía ($\perp\!\!\!\perp$) y los predicados de igualdad (=) y desigualdad (≠).

Podemos obtener nuevas funciones while-computables mediante la composición de otras ya conocidas. De forma análoga las operaciones while-computables y totales también pueden componerse con predicados while-decidibles para utilizarse en las macrocondiciones. Por último podemos utilizar los operadores lógicos **not**, **and**, **or** para construir macrocondiciones aún más complejas. En definitiva es posible construir expresiones anidadas en programas como el siguiente:

```
while  not P_vacía?(PILA) or AUX<K**2  loop
      PILA := desempilar (PILA);
      AUX := AUX+1;
end loop;
```

Teniendo en cuenta que el contenido de una macrovariable puede ser cualquier elemento de un tipo de datos, y que podemos asignarle el resultado de aplicar cualquier función while-computable es correcto también escribir la asignación

```
AUX:= X5=28 ;
```

ya que estaríamos asignando a AUX un valor de tipo booleano. Tenemos entonces que las expresiones booleanas son válidas tanto en las condiciones como en las partes derechas de las asignaciones.

e) Funciones asociadas a los programas-while

El tipo de datos $\mathbb{W}$ está constituido por el conjunto de los programas-while, cuya definición puede verse en el apéndice A. Entre las operaciones que manejan este tipo de datos podemos distinguir las siguientes:

- Las funciones constructoras que nos permiten formar objetos del tipo $\mathbb{W}$ a partir de los datos de las variables, símbolos y/o subprogramas que intervienen en su construcción.

$$\text{haz\_asig\_vacía} : \mathbb{N} \to \mathbb{W}$$

$$\text{haz\_asig\_cons\_}\mathbf{s} : \mathbb{N} \times \mathbb{N} \to \mathbb{W}$$

$$\text{haz\_asig\_cdr} : \mathbb{N} \times \mathbb{N} \to \mathbb{W}$$

$$\text{haz\_composición} : \mathbb{W} \times \mathbb{W} \to \mathbb{W}$$

$$\text{haz\_condición\_}\mathbf{s} : \mathbb{N} \times \mathbb{W} \to \mathbb{W}$$

$$\text{haz\_iteración} : \mathbb{N} \times \mathbb{W} \to \mathbb{W}$$

  Por ejemplo, con la operación haz_condición_**b**(4, X0:=ε;) resultará el programa **if** $\text{car}_{\mathbf{b}}$?(X4) **then** X0:=ε; **end if**;

- Los predicados de inspección, que se aplican a programas y dan resultado booleano indicándonos si el programa es de un tipo concreto o no.

$$\text{asig\_vacía?} : \mathbb{W} \to \mathbb{B}$$

$$\text{asig\_cons?} : \mathbb{W} \to \mathbb{B}$$

$$\text{asig\_cdr?} : \mathbb{W} \to \mathbb{B}$$

$$\text{composición?} : \mathbb{W} \to \mathbb{B}$$

$$\text{condición?} : \mathbb{W} \to \mathbb{B}$$

$$\text{iteración?} : \mathbb{W} \to \mathbb{B}$$

- Las operaciones de acceso, que nos indican qué elementos intervienen como componentes de un programa-while. Las funciones $\mathit{ind\_var} : \mathbb{W} \to \mathbb{N}$ e $\mathit{ind\_var\_2} : \mathbb{W} \to \mathbb{N}$ nos devuelven el índice de las variables implicadas en el más alto nivel de construcción de un programa. Análogamente disponemos de la

función *ind_simb* : $\mathbb{W} \to \sum^*$ nos permite extraer el símbolo e *inst_int* : $\mathbb{W} \to \mathbb{W}$ e *inst_int_2* : $\mathbb{W} \to \mathbb{W}$ las instrucciones internas.

$$\text{ind\_var}(XI:=\varepsilon;) = \mathbf{i}$$

$$\text{ind\_var}(XI:=\text{cons}_{a_k}(XJ);) = \mathbf{i}$$

$$\text{ind\_var}(XI:=\text{cdr}(XJ);) = \mathbf{i}$$

$$\text{ind\_var}(\textbf{if } \text{car}_{a_k}?(XI) \textbf{ then } P \textbf{ end if};) = \mathbf{i}$$

$$\text{ind\_var}(\textbf{while } \text{nonem}?(XI) \textbf{ loop } P \textbf{ end loop};) = \mathbf{i}$$

$$\text{ind\_var}(P_1\ P_2) \cong \perp$$

Nótese que en este último caso la función *ind_var* está indefinida, pues aunque el programa ha de contener alguna variable esta no se ha utilizado en el máximo nivel, correspondiente a la composición.

$$\text{ind\_var\_2}(XI:=\text{cons}_{a_k}(XJ);) = \mathbf{j}$$

$$\text{ind\_var\_2}(XI:=\text{cdr}(XJ);) = \mathbf{j}$$

Para el resto de los casos la función *ind_var_2* queda indefinida

$$\text{ind\_simb}(XI :=\text{cons}_{a_k}(XJ);) = \mathbf{a_k}$$

$$\text{ind\_simb}(\textbf{if } \text{car}_{a_k}?(XI) \textbf{ then } P \textbf{ end if};) = \mathbf{a_k}$$

Para el resto de los casos la función *ind_simb* queda indefinida

$$\text{inst\_int}(P_1\ P_2) = \mathbf{P_1}$$

$$\text{inst\_int}(\textbf{if } \text{car}_{\mathbf{s}}?(XI) \textbf{ then } P \textbf{ end if};) = \mathbf{P}$$

$$\text{inst\_int}(\textbf{while } \text{nonem}?(XI) \textbf{ loop } P \textbf{ end loop};) = \mathbf{P}$$

Para el resto de los casos la función *inst_int* queda indefinida

$$\text{inst\_int\_2}(P_1\ P_2) = \mathbf{P_2}$$

Para el resto de los casos la función *inst_int_2* queda indefinida

Referencias

[Gar 84] M. GARDNER. ¡Ajá! Paradojas. 2ª edición. Ed. Labor S.A. 1984.

[Har 87] D. HAREL. Algorithmics. The spirit of Computing. Addison-Wesley, 1987

[IIS 96] J. IBAÑEZ; A. IRASTORZA; A. SANCHEZ. Los Programas while. Bases para una teoría de la computabilidad. Informe interno UPV/EHU/LSI/TR 5-96.

[IIS 00] J. IBAÑEZ; A. IRASTORZA; A. SANCHEZ. Algunas demostraciones de incomputabilidad usando la técnica de diagonalización. Informe interno UPV/EHU/LSI/TR 08-2000.

[MAK 88] R. N. MOLL; M. A. ARBIB; A. J. KFOURY. A programming approach to computablity. Springer-Verlag, 1988

[SW 88] R. SOMMERHALDER; S. C. van WESTRHENEN The theory of computability. Programs, Machines, Effectiveness and Feasibility. Addison-Wesley 1.988

[Sta 81] G. STAHL. *El método diagonal en teoría de conjuntos.* Teorema. Vol 11, nº1, pp 27-35, 1981