El método de reducción en Teoría de la Computabilidad

Ana Sánchez; Jesús Ibáñez; Arantza Irastorza

UPV/EHU/ LSI / TR ?-2010

Índice

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 que no pueden ser superadas por ninguna tecnología digital de procesamiento de la información, a modo de Leyes fundamentales que gobiernan las propias condiciones de existencia de la Informática.

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 se utilizan para demostrar tanto la computabilidad como la incomputabilidad de muchas funciones relevantes.

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 exhibirían quince años más tarde. El problema de parada es el primer y más conocido ejemplo de problema no resoluble mediante técnicas algorítmicas: ningún ordenador, por muy potente que sea, puede anticipar el comportamiento de los programas en ejecución, y decidir de antemano si terminarán o no. Este problema nos proporciona un soporte intuitivo para anticipar la incomputabilidad de otros problemas relacionados y un procedimiento para resolverlos, ya que el método de diagonalización que se sigue para demostrar la indecidibilidad del problema de parada es fácilmente generalizable para poder aplicarse a otras demostraciones de incomputabilidad[1]. Sin embargo para determinados problemas también incomputables hay que recurrir a otros métodos.

Este informe incluye una descripción de otra técnica básica de Teoría de la Computabilidad: la Reducción. La base del método estriba en demostrar que ciertos pares de problemas están fuertemente relacionados de modo que si el segundo tiene solución algorítmica entonces el primero debe tenerla necesariamente también. Esta relación se

[1] Puede leerse una explicación muy detallada de la técnica en [ISI 03], y en [IIS 00] figura una colección de ejemplos de problemas incomputables y su demostración mediante la misma.

establece por medio de funciones transformadoras computables, que permiten convertir de manera automática las instancias positivas del primer problema en instancias positivas del segundo. La estrategia del método consiste en utilizar la reducción en forma negativa, de manera que si se consigue relacionar un problema **P** cuya incomputabilidad está claramente establecida (como el problema de parada) con otro cualquiera **X**, entonces este segundo problema tampoco podrá tener solución algorítmica y su incomputabilidad quedará demostrada. La razón es muy simple: si **X** fuese computable y por tanto hubiese un algoritmo que lo resolviese, podríamos conseguir que dicho algoritmo resolviese también **P.** Para ello bastaría con transformar las instancias de **P** en instancias de **X** mediante la función que relaciona ambos problemas y aplicar el algoritmo-solución de **X** a dichas instancias transformadas.

Esta técnica se utiliza muy a menudo porque resulta comparativamente más sencilla que la diagonalización, ya que en general requiere menos esfuerzo para demostrar la incomputabilidad de un mismo problema. Una explicación intuitiva de esta ventaja es que, mientras que la diagonalización debe atacar el centro de la incomputabilidad de un problema en términos absolutos, la Reducción se conforma con establecer que dicho problema es "al menos tan incomputable" como otro que se utiliza como rasero, y si se elige bien este último es posible explotar las similitudes que ambos problemas puedan compartir.

Por otro lado, esta misma técnica resulta aún más crucial en la disciplina hermana denominada Teoría de la Complejidad algorítmica, en la que también se intentan clasificar los problemas, pero en este caso de acuerdo con su coste inherente, entendiendo por tal el mínimo coste entre todas sus posibles soluciones. Así, se establecen relaciones jerárquicas entre problemas más y menos costosos de resolver, y cada vez que se fija una cota superior es posible definir una clase de complejidad que agrupe a todos aquellos problemas cuyo coste inherente no supere dicha cota.

Análogamente a lo que sucede en Teoría de la Computabilidad, la hipótesis "sencilla" de verificar es la de pertenencia a una determinada clase: bastará con encontrar un algoritmo que resuelva el problema cuyo coste no supere la cota para dicha clase. Por el contrario, la demostración de que un problema **X** tiene un coste inherente superior al tolerado para cierta clase **C** requiere demostrar que ninguna de sus infinitas soluciones puede ejecutarse con los recursos requeridos para los miembros de la misma, y la prueba directa de este hecho puede ser enormemente costosa o directamente inasequible. Ahora bien, imaginemos que se encuentra otro problema **P** cuya no pertenencia a **C** está

claramente establecida[2] porque se sabe que tiene un coste inherente superior. Imaginemos asimismo que se pueden relacionar las instancias de **P** y **X** mediante una función transformadora como la descrita más arriba (que además de computable tenga un coste irrelevante para la correspondiente clase de complejidad). Entonces se podrá aplicar un razonamiento similar: no puede existir un algoritmo que resuelva **X** con un coste suficientemente bajo, porque de existir podría combinarse con la función transformadora para resolver **P** con un coste también lo suficientemente bajo como para permitir su entrada en la clase **C**.

No obstante en este informe nos limitaremos a exponer el método de Reducción entre problemas de acuerdo con su computabilidad relativa, tratando de presentar la misma como una herramienta básica y eficaz de demostración de incomputabilidad de muchos problemas relacionados con la propia computación. Su elaboración obedece a la necesidad de completar el material de apoyo docente a 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 que se publicó en informes anteriores. Los resultados se presentarán, allí donde sea necesario, utilizando como estándar de programación los programas-while tal como los describimos en [IIS 96]. A pesar de que este trabajo es la continuación natural de [IIS 96, ISI 03] junto a los que constituye el grueso de la presentación teórica de la asignatura arriba mencionada, quien quiera prescindir de la lectura de los informes previos encontrará las definiciones y conceptos fundamentales en el capítulo segundo, así como una descripción más detallada del lenguaje de los programas-while junto con otros conceptos e informaciones contextualizadoras en el apéndice. Por la misma razón, quien haya desembocado en el presente trabajo con el conocimiento de los informes anteriores encontrará muchas definiciones conocidas en el capítulo 2, y es probable que juzgue superfluo el contenido del apéndice.

Nuestra exposición pretende situar la discusión en su apropiado punto de partida. Cuando nos enfrentamos al reto de probar que ciertos problemas no son resolubles por ningún sistema de programación no se puede decir que los métodos abunden. La diagonalización es el mecanismo que abrió brecha en este campo obteniendo sus primeros éxitos, pero ciertas limitaciones (sobre todo de índole práctica) hacen desear de forma natural que exista una alternativa. Su aplicabilidad en las circunstancias en las que la

[2] Lamentablemente en Teoría de la Complejidad, y a diferencia de la Teoría de la Computabilidad, la norma es que la no pertenencia a clases de complejidad no llegue a estar "claramente establecida", sino sólo "razonablemente soportada". Un gran número de sus resultados están condicionados a ciertas hipótesis fundamentales que no han podido ser probadas, como puede ser la conjetura P≠NP.

diagonalización fracasa total o parcialmente es precisamente la virtud principal del método de Reducción, por lo que primero daremos un repaso a dichas limitaciones en el Capítulo 3.

En el Capítulo 4 pasamos a exponer la técnica de Reducción computacional en su versión menos artificiosa, con el objeto de proporcionar una descripción lo más comprensible posible de la misma. Dado que la reducción habitualmente se orienta a la rápida presentación de métodos "fuertes" como los Teoremas s-m-n y de Rice, esta es la parte que suele estar más desatendida en la literatura. Sin embargo creemos que la cabal comprensión del Método de Reducción pasa por la experimentación con algunos de los problemas que se presentan cuando se intenta aplicar de manera directa, sin la ayuda de ningún otro resultado ([IS 09]). Además hay muchos problemas incomputables de tipo "práctico" (es decir, que conciernen al ejercicio real de la Informática) que sólo se pueden abordar en este estadio del método.

Una vez establecidas las pautas de la técnica pasamos a centrarnos en ciertos tipos de problemas que permiten una visión de la Reducción mucho más orientada a resultados. Estamos en el terreno de la Reducibilidad, que se explica en el Capítulo 5 para permitir establecer relaciones jerárquicas entre las propiedades de incomputabilidad.

El Capítulo 6 presenta el resultado más importante relacionado con la Reducción. El Teorema s-m-n establece las condiciones para aplicar el método con seguridad y eficacia a una enorme batería de problemas. Este potencial que proporciona el Teorema hace que en muchos cursos se tenga una versión muy utilitaria del mismo, hasta el punto de que se puede adquirir la sensación de que se trata de un truco sobre cuyos fundamentos no cabe plantearse preguntas. Frente a esta visión presentamos este resultado como colofón o consecuencia lógica del afán por eliminar las principales aristas del Método de Reducción.

No es objetivo de este informe seguir por ese camino e introducir técnicas adicionales que completan el Método de Reducción, como son los Teoremas de Rice o los de Recursión. Sin embargo, los Teoremas de Rice están tan íntimamente ligados a las técnicas repasadas en este informe que se ha introducido un séptimo Capítulo a modo de epílogo para explicar de manera muy sucinta cómo se continuaría por el camino de buscar grandes grupos de problemas sobre los que el Teorema s-m-n se aplica de forma homogénea.

2. Nociones básicas y notación

El estudio de las propiedades teóricas de los sistemas computacionales requiere utilizar un modelo formal que defina lo que entendemos por computador. Existe un gran número de trabajos que han abordado esta tarea, y se han utilizado para ello modelos muy diferentes entre sí. ¿A qué se debe que el mismo concepto de computador abstracto pueda ser representado por formalismos tan variopintos? La razón es que un ordenador ideal necesita sólo un pequeño conjunto de operaciones muy elementales para tener plena funcionalidad[3].

En nuestro caso utilizaremos el modelo de los programas-while, que define dicho computador en términos del lenguaje de programación que es capaz de entender. Pero debe quedar claro desde el principio que si hubiéramos partido de otro mecanismo de programación o de otros modelos alternativos de cómputo habríamos llegado igualmente a un ***sistema de programación universal y aceptable***, que aunque diferente, nos habría permitido establecer los mismos resultados aquí expuestos.

Revisamos a continuación algunos de los resultados básicos de Teoría de la Computabilidad a los que, insistimos, se llega de manera independiente con cualquier modelo abstracto de cómputo, pero para ello establecemos previamente algunos conceptos relacionados con el sistema computacional elegido en nuestro caso. Nos detendremos más en aquellos detalles que necesitaremos utilizar para establecer los resultados necesarios relacionados con la técnica de reducción.

2.1. Procesamiento de la información simbólica

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 un alfabeto $\sum$, definimos $\sum^*$ como el conjunto de las ***palabras*** o ***cadenas*** sobre el mismo. Utilizaremos

[3] Esta plena funcionalidad se resume en algo tan prosaico, como "la capacidad de programar todas las funciones computables". En esencia las funciones computables se definen en relación a un sistema de programación de referencia, papel que históricamente corresponde a las Máquinas de Turing. Alan Turing fue el primero en demostrar que las funciones computables podían enumerarse de manera efectiva en forma de lista.

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.

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 $\mathbf{x}$ 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 $\psi : \sum^{*k} \longrightarrow \sum^*$. En este caso indicaremos que $\psi(\mathbf{x}_1, ..., \mathbf{x}_k)\downarrow$ o 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 cadena como 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, aunque también puede consultarse una descripción más detallada en [IIS 96]. Dadas la equivalencia entre los distintos sistemas de programación y el hecho de que en este texto utilizaremos los programas-while en exclusiva, nos referiremos a estos últimos como "programas" de forma genérica.

El comportamiento de un programa **P** estará descrito por una función parcial que denotaremos $\varphi_P : \sum^* \longrightarrow \sum^*$, y que describe la relación que existe entre la cadena de entrada que recibe **P** y la que produce como resultado. Como para devolver dicho resultado el programa debe terminar su ejecución, cuando el comportamiento de **P** ante una entrada $\mathbf{x}$ consista en ciclar indefinidamente consideraremos que el resultado del programa es indefinido y lo notaremos de manera consistente como $\varphi_P(\mathbf{x})\uparrow$. Si φ_P es ***total*** (está definida para todos los posibles argumentos) entonces el programa **P** es capaz de terminar su ejecución sean cuales fueren las circunstancias del inicio de la misma. En el otro extremo 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 la notaremos como $\phi_{\mathbf{P}}^{\mathbf{k}} : \sum^{*\mathbf{k}} \longrightarrow \sum^{*}$.

COMPUTABILIDAD: Una función $\psi : \sum^{*\mathbf{k}} \longrightarrow \sum^{*}$ es ***computable*** si existe un programa **P** que la computa ($\phi_{\mathbf{P}}^{\mathbf{k}} \cong \psi$), es decir, que obtiene sistemáticamente los resultados de la función para cualesquiera valores de sus argumentos. En el mismo sentido que lo señalado anteriormente para los programas, hablamos de ***computabilidad*** en general sin ligarla a ningún sistema de programación concreto dando por supuestas la equivalencia de todos ellos y la aceptación de la tesis de Church-Turing (véanse por ejemplo [Mor 98], [Har 87], [SW 88] ó [ISI 03]).

PREDICADOS: Hay una clase especial de funciones que tienen una importancia especial en Teoría de la Computabilidad: los predicados o funciones booleanas totales. De la misma manera que podemos clasificar las funciones en computables o incomputables de acuerdo con la posibilidad de que un programa 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 ***decidible*** si existe un programa **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 decidible si es posible decidir su veracidad mediante un programa.

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

2.2. Extensión a información no simbólica

La computación digital está basada en el reconocimiento y manipulación de símbolos discretos distinguibles, y ésta es la razón de que los sistemas de programación estén en general definidos para operar sobre cadenas de caracteres (palabras)[4]. Sin embargo las palabras no son sino instrumentos para representar otras cosas, por lo que si deseamos procesar información en términos de otro conjunto de datos bastará con definir alguna regla de representación que establezca una correspondencia entre los significantes

[4] Existen modelos de computación generalizados más abstractos que utilizan conjuntos de datos arbitrariamente complejos. En ellos se asume como axioma la computabilidad de las operaciones algebraicas básicas definidas para los datos en cuestión. Lógicamente, en estos modelos no se cuestiona la plausibilidad física de tal axioma.

(palabras sobre un alfabeto escogido) y los posibles significados (elementos de dicho conjunto de datos), de forma que cada cadena represente de manera adecuada un elemento del nuevo conjunto y que la computación sobre palabras sea congruente con la representación de los elementos del nuevo tipo. Si esto se hace de manera cuidadosa ello nos permitirá trabajar con funciones computables en dominios distintos de $\sum^*$, así como utilizar los elementos y las operaciones de dichos tipos en nuestros programas como si fueran primitivas del lenguaje de programación. Así 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. Este último hecho es esencial para la Teoría de la Computabilidad, ya que los problemas más interesantes por su dificultad intrínseca se localizan precisamente en el dominio de los programas y su comportamiento, por lo que el estudio de la computabilidad en el tipo de datos $\mathbb{W}$ proporcionará los resultados más relevantes.

En el apéndice se listan los predicados y las funciones definidas sobre estos tipos de datos cuya computabilidad fue demostrada en [IIS 96] y que podrán ser utilizadas en los programas que construyamos de aquí en adelante. 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.

Salvo en el caso de los tipos finitos, como los booleanos, las implementaciones pueden definirse siempre biyectivas. Esto quiere decir que, por ejemplo, existe una correspondencia biunívoca entre los números naturales y las palabras de $\sum^*$ que sirven para representarlos, y este hecho se manifiesta independientemente del alfabeto $\sum$ elegido. Por ello, podemos ***enumerar*** el conjunto $\sum^*$ de palabras sobre un alfabeto cualquiera de la forma $\sum^* = \{\mathbf{w}_0, \mathbf{w}_1, \mathbf{w}_2, \mathbf{w}_3, ...\}$, donde $\mathbf{w}_i$ es la palabra que sirve para representar el número natural **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**).

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 a su vez por la palabra $\mathbf{w}_i$. Diremos que $\mathbf{w}_i$ (y, por extensión, también el número asociado $\mathbf{i}$) 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. Al conjunto de todos los índices de una función ψ lo denotamos por $\mathrm{Ind}(\psi)$.

Por último destacaremos un convenio notacional más: nos referiremos al ***dominio*** de la función computable φ_i como $\mathbf{W}_i$ y a su ***rango*** como $\mathbf{R}_i$ [5]. 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.

2.3. Computación práctica en el tipo W

Pueden definirse infinidad de funciones que utilizan los programas como datos corrientes, siendo el sintáctico el nivel de manipulación más sencillo que podemos definir: podemos estudiar el texto de un programa, examinar sus instrucciones o su estructura, y resolver cuestiones en cuanto a la misma. A este nivel nos encontraremos típicamente con funciones totales que toman como argumento un programa 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

[5] 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 $\{\varphi^{\mathbf{k}}_0, \varphi^{\mathbf{k}}_1, \varphi^{\mathbf{k}}_2, \varphi^{\mathbf{k}}_3 \ldots\}$, donde $\varphi^{\mathbf{k}}_{\mathbf{i}}$ es la función computada por el programa $\mathbf{P}_i$ cuando se le suministran **k** datos de entrada. Lo dicho para un argumento se extiende de manera natural para **k** argumentos, y en particular la notación $\mathbf{W}^{\mathbf{k}}_{\mathbf{i}}$ y $\mathbf{R}^{\mathbf{k}}_{\mathbf{i}}$ para el dominio y el rango de $\varphi^{\mathbf{k}}_{\mathbf{i}}$ respectivamente.

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. Algunas de ellas vienen listadas en el apéndice.

Otro tipo de funciones cuya computabilidad es más difícil de demostrar son las relacionadas con las ***propiedades dinámicas*** de los programas, aquellas que dependen de la ejecución de los mismos. Por ejemplo, podemos definir funciones que nos indiquen cuántos pasos conlleva la ejecución de un programa (si es que termina), si la ejecución entra en un bucle determinado o no, o si todos los resultados del programa son números menores que 1000. Entre ellas se encuentra una de importancia capital para la Teoría de la Computación: la ***función universal***, que dado cualquier par programa-dato indica el resultado de la ejecución del primero sobre el segundo. La descripción formal de esta función puede encontrarse en el apéndice. Su computabilidad fue probada por Turing, y puede consultarse en [ISI 03] una demostración de la misma basada en el formalismo de los programas-while.

2.4. Decidibilidad y semidecidibilidad

La Teoría de la Computabilidad tiene la complicación intrínseca de tratar de describir precisamente las propiedades de los problemas que son lo suficientemente complejos como para no permitir una aproximación de tipo algorítmico. Pero al menos podemos reducir la complicación en el método de aproximación a dichos problemas si procuramos utilizar la formulación más sencilla posible de los mismos. Es la aplicación de esta idea lo que se encuentra en las sucesivas decisiones de ceñirnos, primero a problemas sobre cadenas de caracteres (frente al posible tratamiento de "otras formas de información") luego a aquellos que pueden formularse en términos de un sólo resultado (expresables mediante funciones), y a menudo a los que se basan en un único dato de entrada (funciones unarias). Estas sucesivas restricciones se han hecho porque no suponen ninguna pérdida de generalidad: la computabilidad en dominios diferentes a los de las cadenas de símbolos puede reformularse vía implementaciones, los problemas con resultados múltiples pueden descomponerse en múltiples subproblemas con resultado único, y una función con varias entradas puede ser fácilmente traducida a su versión unaria equivalente mediante la oportuna codificación.

Aún es posible una simplificación adicional dada la evidencia de que la mayor parte de los problemas con propiedades de incomputabilidad interesantes tienen asociada

una formulación con propiedades similares en forma de problema de decisión, es decir, que puede expresarse en forma de predicado o función total de resultado booleano. Dado que los predicados son más fáciles de manejar y estudiar que las funciones generalizadas, es habitual concentrarse en los mismos una vez sentadas las bases de la teoría.

Una de las decisiones que más ayudan al trabajar con predicados es tener en cuenta que todo predicado tiene asociado un conjunto y obrar en consecuencia. Dado un predicado $\mathbf{S}: \sum^* \rightarrow \mathbb{B}$ se define de manera natural el conjunto $\{ \mathbf{x}: \mathbf{S}(\mathbf{x}) \}$ de las palabras que verifican el mismo. De la misma manera, dado un conjunto cualquiera $\mathbf{A} \subseteq \sum^*$ se define su función característica $\mathbf{C}_A$ de la siguiente manera:

$$\mathbf{C}_A(\mathbf{x}) = \begin{cases} \text{true} & \mathbf{x} \in \mathbf{A} \\ \text{false} & \text{c.c.} \end{cases}$$

Naturalmente, esta función es el predicado que elucida la pertenencia de una palabra cualquiera al conjunto **A**. Dada esta correspondencia es habitual tratar más con conjuntos que con funciones y estudiarlos de acuerdo a la computabilidad de su función característica. Aquellos conjuntos o predicados cuya función característica es computable constituyen la clase de los conjuntos ***decidibles***, que se denota como Σ_0.

Esta primera clasificación entre conjuntos decidibles o indecidibles[6] puede refinarse. Siendo indecidible todo conjunto para el que no se puede construir un algoritmo que diferencie claramente entre sus elementos y los de su complementario, a veces es posible encontrar una solución parcial: un programa que al menos sea capaz de dar respuesta positiva ante las entradas que pertenecen al conjunto, aunque cicle ante las que no lo son. Para formalizar esto definimos la función semicaracterística de un conjunto **A** de la siguiente manera:

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

Si χ_A es computable diremos que **A** es ***semidecidible***, y denominaremos Σ_1 a la clase de todos los conjuntos semidecidibles. Aunque la noción de semidecidible se introduce para hacer distinciones adicionales entre los conjuntos indecidibles, claramente todo conjunto decidible es semidecidible, es decir $\Sigma_0 \subset \Sigma_1$. Sin embargo la inclusión inversa no es cierta, ya que es posible encontrar muchos conjuntos que, siendo semidecidibles, no son decidibles.

[6] En buena parte de la literatura clásica sobre Teoría de la Computabilidad encontramos que a los conjuntos decidibles se les denomina recursivos y a los semidecidibles recursivamente enumerables.

Algunos de esos conjuntos son muy conocidos y útiles en Teoría de la Computabilidad y por ello reciben nombres específicos. Por ejemplo el conjunto $\mathbf{K} = \{ \mathbf{x} \in \mathbb{W} : \varphi_x(\mathbf{x}) \downarrow \}$, que 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 ello el conjunto **K** se utiliza en muchas demostraciones que prueban por reducción al absurdo la indecidibilidad de otros conjuntos menos accesibles. Otro ejemplo de conjunto semidecidible pero no decidible es $\overline{\mathbf{VAC}} = \{ \mathbf{x} \in \mathbb{W} : \exists \mathbf{y}\ \varphi_x(\mathbf{y}) \downarrow \}$, el conjunto de los programas que son capaces de devolver resultado al menos en un caso.

Otros conjuntos son aún más inasequibles desde el punto de vista de la Teoría de la Computabilidad, puesto que ni siquiera son semidecidibles. Un ejemplo de los más interesantes es el conjunto $\mathbf{TOT} = \{ \mathbf{x} \in \mathbb{W} : \forall \mathbf{y}\ \varphi_x(\mathbf{y}) \downarrow \}$ de los índices de funciones totales, es decir, de los programas lo suficientemente bien construidos como para no ciclar en ningún caso. Si **TOT** fuera decidible existiría alguna forma de distinguir (y en su caso apartar) los programas que pudieran tener cómputos infinitos, pero lamentablemente no es posible siquiera disponer de un mecanismo de detección de los programas que cumplen unos requisitos mínimos de usabilidad.

Las demostraciones de indecidibilidad y no semidecidibilidad citadas se pueden consultar en [IIS 00] y en [ISI 03]. En estas fuentes se utiliza la técnica de diagonalización, tal y como se esboza en el siguiente capítulo, pero en todos los casos sería más sencillo el uso de mecanismos reducción, objeto del presente informe, y que serán explicados y utilizados a partir del Capítulo 4.

3. Incomputabilidad y diagonalización

Cuando pasamos del estudio del nivel sintáctico de los programas al nivel semántico, las preguntas sobre la forma de los programas son desplazadas por otras sobre su comportamiento, evolución y resultados. Como consecuencia de ello los nuevos problemas que surgen son más difíciles de computar y además suelen quedar asociados a funciones no totales, al reflejar los casos en que el programa estudiado no converge. Pero esa no es la dificultad más grave: cuando no nos contentamos con describir el comportamiento de los programas, sino que tratamos de predecirlo, empiezan a aparecer funciones para las cuales no encontramos un programa que las compute por la sencilla razón de que tal programa no existe: entramos en el reino de la incomputabilidad.

3.1. Problemas incomputables

Los problemas para los que no existe ningún algoritmo posible, los incomputables, no tienen por qué ser especialmente rebuscados ni difíciles de enunciar. Por ejemplo, nos podemos plantear el problema de decidir si dos programas cualesquiera son equivalentes o no. En rigor esto significa encontrar un programa que compute la función:

$$\textbf{EQ}: \mathbb{W} \times \mathbb{W} \rightarrow \mathbb{B}$$

que produce un resultado verdadero si los dos programas que se le suministran como argumento producen siempre los mismos resultados, y falso en caso contrario. Lamentablemente es posible demostrar que esta función no es computable.

EQ ilustra bien el hecho de que la incomputabilidad no es un fenómeno de exclusivo interés teórico. La no computabilidad de **EQ** implica la imposibilidad de demostrar, en general, que un programa cumple con corrección los fines para los que ha sido diseñado, ni siquiera cuando disponemos de otro programa correcto como modelo.

Otro ejemplo que ilustra cómo algunos problemas que surgen en la informática práctica no pueden ser resueltos por la propia informática por motivos teóricos, nos lo proporciona la siguiente función:

$$\textbf{traza}: \mathbb{W} \times \Sigma^* \times \mathbb{N} \rightarrow \mathbb{B}$$

definida de forma que **traza**(**x**,**y**,**n**) es cierto si y solo sí la ejecución de $\mathbf{P}_x$ sobre el dato **y** llega alguna vez a la **n**-ésima instrucción del código de $\mathbf{P}_x$. Por tanto, tampoco es posible encontrar un algoritmo para anticipar si la ejecución de un programa va a pasar por una instrucción concreta o no.

3.2. Diagonalización

El problema de parada, es sin duda el miembro más conocido de la familia de los incomputables, y es también el "responsable" primario de la incomputabilidad de los ejemplos antes citados y de la de muchos otros. La formulación de la incomputabilidad del problema de parada afirma que no existe un algoritmo para decidir con carácter general si un programa ciclará o no al recibir unos datos de entrada concretos. Más formalmente el problema de parada es un predicado que plantea una propiedad dinámica sobre los programas, 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. Nos referiremos a la función del problema de parada como **H** (de Halt).

TEOREMA (INDECIDIBILIDAD DEL PROBLEMA DE PARADA): La siguiente función no es computable:

$$\mathbf{H}: \mathbb{W} \times \Sigma^* \rightarrow \mathbb{B}$$

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

Asumimos aquí el resultado sin entrar en su demostración, que puede consultarse en cualquier libro de computabilidad ([MAK 88], [Mor 98], [SW 88]) o en informes anteriores ([ISI 03]). 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 ***posibles*** 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.

La técnica de diagonalización [IIS 00, ISI 03] es la más básica de ellas, y resulta bastante conocida al no tratarse de una herramienta específica de la Informática Teórica. Su invención se remonta a 1873 y se debe a Cantor, que acudió a un argumento diagonal para probar que el cardinal de los números reales es mayor que el de los naturales, estableciendo la distinción entre infinito numerable y no numerable. Para su demostración Cantor probó que ninguna función entre $\mathbb{N}$ y $\mathbb{R}$ puede ser biyectiva, ya que siempre es posible construir un número real que no esté en el rango de dicha función.

Las demostraciones por diagonalización se fundamentan en un argumento de contradicción o reducción al absurdo, pero con un proceso de construcción intermedio. Para demostrar la incomputabilidad de una cierta función **f** empezamos suponiendo justamente lo contrario, es decir que **f** es computable, y basándonos en esa hipótesis construimos una función diagonal δ que debería ser también computable. Sin embargo δ es definida cuidadosamente para ser necesariamente diferente de todas las funciones computables existentes, con lo que el argumento se cierra con la correspondiente contradicción.

En el método de diagonalización (descrito con detalle en [ISI 00]) se utiliza una tabla con todas las funciones computables $\{\varphi_0, \varphi_1, \varphi_2, ...\}$ que sirva de apoyo para construir la función diagonal. En dicha tabla cada columna representa una función computable y cada fila un posible argumento, de forma que la casilla de coordenadas (**i**,**j**) se utiliza para albergar información que **f** nos aporta sobre $\varphi_i(\mathbf{j})$, es decir sobre el resultado de utilizar el programa $\mathbf{P}_i$ sobre el dato **j**. Esta información depende fuertemente de la naturaleza de **f**, puede ser más o menos concreta y puede ser extensiva a todos los puntos de la función φ_i o solo a algunos. Apoyándonos en lo que sabemos sobre cada una de las funciones φ_i tratamos de construir una función diagonal apropiada δ en una columna adicional, cuya computabilidad se podrá deducir de la de **f**.

Lo más sencillo suele ser concentrarse en la información que **f** proporciona sobre cada φ_i precisamente en el punto **i**. Si gracias a **f** se puede averiguar que $\varphi_i(\mathbf{i})$ cumple una cierta propiedad $\mathbf{Q}_i$ (por ejemplo que converge, o que devuelve un valor par, o que en caso de converger el resultado nunca es mayor que 5), y podemos utilizar esta información para, sin violar las leyes de la computabilidad, conseguir que $\delta(\mathbf{i})$ incumpla esa misma propiedad (haciendo que diverja, o que produzca un resultado impar, o que devuelva 6) nos aseguraremos de que δ sea distinta de cualquier función computable φ_i al menos en el punto **i**. En la figura 3.1 se ilustra este procedimiento de forma somera.

δ	**x**	φ_0	φ_1	φ_2	…	φ_i
$\neg\mathbf{Q}_0(\delta(0))$	0	$\mathbf{Q}_0(\varphi_0(0))$				
$\neg\mathbf{Q}_1(\delta(1))$	1		$\mathbf{Q}_1(\varphi_1(1))$			
$\neg\mathbf{Q}_2(\delta(2))$	2			$\mathbf{Q}_2(\varphi_2(2))$		
…	…				…	
$\neg\mathbf{Q}_i(\delta(\mathbf{i}))$	**i**					$\mathbf{Q}_i(\varphi_i(\mathbf{i}))$

Figura 3.1: 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. De este modo δ ha de ser necesariamente incomputable, ya que en la lista infinita $\{\varphi_0, \varphi_1, \varphi_2, ...\}$ están todas las funciones computables. Es esencial que la propiedad $\mathbf{Q}_i$ utilizada para caracterizar el comportamiento de $\varphi_i(\mathbf{i})$ pueda ser deducida de la presunta computabilidad de **f**. Por ejemplo, cuando se prueba la incomputabilidad de **H** la información que la función de parada nos proporciona sobre $\varphi_i(\mathbf{i})$ es en unos casos que converge y en otros que no.

De este modo la hipótesis de computabilidad de **f** nos lleva a la existencia de una δ, que es a la vez computable (al serlo **f**) e incomputable (por ser distinta de todas ellas), con lo que tendremos una contradicción.

La utilización estricta de la diagonal (diferenciar δ de cada φ_i precisamente en el punto **i**) no es imprescindible, aunque sí conveniente. Lo sustancial del método es encontrar para cada función φ_i al menos un punto vulnerable sobre el que la información proporcionada por **f** nos permita establecer la diferenciación. La expuesta es la versión más sencilla del método, pudiendo encontrarse descritas otras variantes en [ISI 00].

3.3. Limitaciones de la diagonalización

Siendo poderosa y elegante, la técnica de diagonalización resulta desproporcionadamente complicada para la demostración de incomputabilidad de algunos problemas, y frustrantemente insuficiente para probar la de otros.

Un ejemplo del primer caso lo proporciona la demostración por diagonalización de la indecidibilidad del conjunto $\mathbf{PERM} = \{ \mathbf{x} : \varphi_x \text{ es biyectiva} \}$. La demostración implica la construcción de una función diagonal, que resulte distinta de todas las biyecciones computables y que a su vez sea una biyección. La construcción de este engendro (cuyos detalles pueden consultarse en [IIS 00]) se asemeja bastante a un puzzle difícil de encajar, ya que la asignación de cualquier valor afecta a la posible biyectividad de toda la función.

Esto contrasta fuertemente con la sencillez de la aplicación de la técnica de reducción para este mismo problema, que veremos más adelante.

Dentro del segundo caso podemos encontrar fácilmente ejemplos. Consideremos el conjunto **K0** = { **x** : $\varphi_x(0)\downarrow$ } de los programas que convergen sobre el dato 0. Intuitivamente parece indecidible por lo que sería deseable probarlo (de hecho lo haremos más adelante utilizando la reducción). Sin embargo este es uno de los casos que resultan inmunes a la diagonalización.

No hay como intentarlo para convencerse de ello. Para aplicar la técnica de diagonalización suponemos que el conjunto **K0** es decidible, y por tanto que podemos saber, dado un programa de código **x**, si éste pertenece o no al conjunto **K0**, al ser su función característica $\mathbf{C}_{K0}$ computable. Basándonos en esta hipótesis intentamos construir una función δ computable pero diferente de todas las funciones computables en por lo menos un punto (si podemos, el de la diagonal). La información que $\mathbf{C}_{K0}$ nos proporciona se describe en la figura 3.2. Para cada función φ_i puede suceder:

- que $\mathbf{i} \in \mathbf{K0}$ (cosa que averiguamos gracias a que $\mathbf{C}_{K0}(\mathbf{i})$ = true), con lo que $\varphi_i(0)\downarrow$.
- que $\mathbf{i} \notin \mathbf{K0}$ (cosa que averiguamos gracias a que $\mathbf{C}_{K0}(\mathbf{i})$ = false), con lo que $\varphi_i(0)\uparrow$

		$0 \in \mathbf{K0}$	$1 \notin \mathbf{K0}$	$2 \notin \mathbf{K0}$	$3 \in \mathbf{K0}$	$4 \notin \mathbf{K0}$	$5 \in \mathbf{K0}$
δ	**x**	φ_0	φ_1	φ_2	φ_3	φ_4	φ_5
???	0	$\varphi_0(0)\downarrow$	$\varphi_1(0)\uparrow$	$\varphi_2(0)\uparrow$	$\varphi_3(0)\downarrow$	$\varphi_4(0)\uparrow$	$\varphi_5(0)\downarrow$
	1						
	2						
	3						
	4						
	5						

Figura 3.2: Tabla-ejemplo en la que se observa la información de que disponemos sobre las distintas funciones computables gracias a $\mathbf{C}_{K0}$ (fila superior). Dado que dicha información corresponde siempre a los valores en un único punto, idéntico para todas las funciones, no es posible la construcción de la función δ.

En ambos casos es sencillo conseguir que $\delta(0)$ sea diferente de $\varphi_i(0)$ ***para un valor concreto de i***. Pero sea cual sea la opción que se elija para $\delta(0)$ coincidirá necesariamente con $\varphi_j(0)$ para otra función computable indexada por **j**. No existe ningún valor aceptable para $\delta(0)$ que garantice que δ va a ser diferente de todas las funciones computables en el

punto 0 ***al mismo tiempo***. Y $\mathbf{C}_{K_0}$ no nos da información sobre ningún otro punto de todas estas funciones.

Otro ejemplo de conjunto cuya indecidibilidad no es abordable mediante la técnica de diagonalización es **CONS** = { $\mathbf{x} \in \mathbb{W}$: φ_x es total y constante }, ya que necesitaríamos definir una función diagonal que fuera constante, pero diferente a todas las funciones constantes existentes, lo cual es imposible.

Pero el método de diagonalización no sólo es impracticable en ocasiones. Incluso cuando su uso es apropiado y de complejidad razonable puede resultar bastante ineficiente, ya que cuando demostramos que una función es incomputable, este hecho no nos sirve para facilitar ninguna otra prueba de incomputabilidad, salvo la explotación de analogías entre su demostración y la de otros ejemplos. Es posible demostrar por diagonalización la indecidibilidad de los conjuntos **A** = { **x**: $\mathbf{W}_x$ = { **y**: **y** mod 2 = 0 } } (de los programas que convergen exactamente sobre los argumentos pares) y **B** = { **x**: $\mathbf{W}_x$ = { **y**: **y** mod 2 = 1 } } (de los programas que convergen exactamente sobre los argumentos impares). Pero la demostración de uno de ellos no nos sitúa más cerca de la demostración del otro, ya que es preciso reconstruir todos sus pasos desde cero.

3.4. Un cambio de perspectiva

La demostración por diagonalización de que una cierta función ψ no es computable se centra en probar la existencia de una relación lógica como la siguiente

$$\psi \text{ computable} \Rightarrow \delta \text{ computable}$$

(siendo δ un función diagonal específica y trabajosamente construida)

Si intentamos aplicar el método a otra función χ deberemos construir otra función diagonal ad hoc. Aunque a veces la analogía con demostraciones anteriores resulta útil, cuando empezamos cualquier demostración de incomputabilidad por diagonalización siempre estamos básicamente a la misma distancia del objetivo, ya que nuestro marco de referencia es la misma tabla de las funciones computables.

Si bien esta situación es relativamente habitual en matemáticas, se antoja algo más insólita en la formación en Computación, donde las técnicas suelen ser más inductivas e incrementales. Piénsese en el contraste con las técnicas de ***demostración de computabilidad***, en las que toda nueva función computable constituye una plataforma desde la que se observan con más claridad nuevos objetivos, porque dicha función puede

ser utilizada como subrutina en el diseño de nuevos programas con un grado de abstracción más alto, que resulta muy beneficioso para nuestro trabajo.

Un cambio sutil pero relevante se produciría si para demostrar la incomputabilidad de ψ nos pudiéramos centrar en probar una relación de la forma

$$\psi \text{ computable} \Rightarrow \xi \text{ computable}$$
(siendo ξ una función incomputable ***ya conocida***)

Aparte de ahorrarnos la construcción de una función diagonal, un enfoque de esta naturaleza nos permitiría cierta libertad a la hora de elegir la función ξ. Es razonable esperar que cuanto más se parezcan la función elegida y la función original ψ más sencilla será la demostración de la implicación de arriba. Adicionalmente la aplicación del propio método irá enriqueciendo nuestro catálogo de funciones incomputables que podrían ser incorporadas en el futuro para demostrar la incomputabilidad de nuevas funciones.

La posibilidad de relacionar la indecidibilidad de unos problemas con la de otros tendrá otra consecuencia: el establecimiento de una jerarquía de "dificultad computacional" cuyos elementos inferiores ya se conocen. Si probamos que

$$\psi \text{ computable} \Rightarrow \chi \text{ computable} \Rightarrow \xi \text{ computable}$$

y por tanto que

$$\xi \text{ incomputable} \Rightarrow \chi \text{ incomputable} \Rightarrow \psi \text{ incomputable}$$

sabiendo de antemano que ξ no es computable, no sólo demostraremos que ψ y χ tampoco pueden serlo: el esquema implica asimismo que ψ es "tanto o más incomputable" que χ, que a su vez lo es con respecto a ξ. Si se consigue probar que las implicaciones son estrictas (es decir, que sus contrarias no son ciertas) podremos deducir que ψ es "más incomputable" que χ, que a su vez lo será más que ξ.

Esta jerarquía, llevada al terreno de los conjuntos, nos permitirá comprobar que los decidibles son los problemas de decisión más "sencillos" y que los semidecidibles que no son decidibles quedan por encima de ellos en grado de dificultad. 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. Por extraño que parezca no sólo existen conjuntos "más indecidibles" que otros, sino además se pueden encontrar con un "grado de indecidibilidad" tan alto como se desee.

Un ejemplo intuitivo de esta jerarquía podría ser el siguiente. Considérense los siguientes conjuntos:

$$\mathbf{A1000} = \{ \mathbf{x}: \mathbf{T}(\mathbf{x}, 0, 999) \}$$

$$\mathbf{K0} = \{ \mathbf{x} : \varphi_x(0)\downarrow \}$$

$$\mathbf{TOT} = \{ \mathbf{x} : \forall \mathbf{y} \varphi_x(\mathbf{y})\downarrow \}$$

que agrupan, respectivamente, los programas que convergen sobre el dato 0 en menos de 1000 pasos[7], los que simplemente convergen sobre dicho dato y los que convergen sobre todos los datos. Se puede demostrar la veracidad de la siguiente implicación

$$\mathbf{TOT} \text{ decidible} \Rightarrow \mathbf{K0} \text{ decidible} \Rightarrow \mathbf{A1000} \text{ decidible}$$

que, atención, no establece nada sobre la decidibilidad de ninguno de los tres conjuntos, sino de cómo el hipotético estatuto de decidibilidad de unos afectaría al de los otros. Al mismo tiempo, aunque no nos diga cuáles de estos conjuntos son decidibles (podrían serlo todos o ninguno), lo que sí indica esta implicación es que **TOT** es un conjunto más (o igual de) "difícil" que **K0**, que a su vez lo es con respecto a **A1000**.

Vamos a demostrar que la propiedad es cierta, empezando por la segunda implicación. Supongamos que **K0** es decidible, y por tanto que existe un procedimiento general para saber si un programa converge o no sobre el dato 0. Se puede entonces diseñar el siguiente procedimiento para saber si un programa $\mathbf{P}_x$ está en **A1000** o no:

1. Averiguar si $\mathbf{P}_x$ está o no en **K0** (que por hipótesis es posible)
2. Si $\mathbf{x} \notin \mathbf{K0}$ deducimos que **x** tampoco está en **A1000** y terminamos
3. Si $\mathbf{x} \in \mathbf{K0}$ ejecutamos $\mathbf{P}_x(0)$, sabiendo que es un proceso finito, contando los pasos de ejecución
4. Si al finalizar se han ejecutado menos de 1000 pasos deducimos que **x** está en **A1000** y terminamos
5. Si se han ejecutado 1000 o más pasos deducimos que **x** no está en **A1000** y terminamos

[7] El predicado $\mathbf{T}(\mathbf{x},\mathbf{y},\mathbf{z})$ se verifica cuando el programa $\mathbf{P}_x$ converge sobre el dato **y** en un número de pasos no superior a **z** (véase el apéndice).

Por tanto, $\mathbf{K0} \in \Sigma_0 \Rightarrow \mathbf{A1000} \in \Sigma_0$, y lo que es más importante, $\mathbf{A1000} \notin \Sigma_0 \Rightarrow \mathbf{K0} \notin \Sigma_0$. En este caso la implicación demostrada no sería demasiado fructífera, ya que no se verifica la premisa $\mathbf{A1000} \notin \Sigma_0$. También se puede probar que la decidibilidad de **A1000** ***no implica*** la de **K0**, pero no lo haremos aquí[8].

La primera implicación es algo más enrevesada de demostrar, pero vayamos a ello. Supongamos que **TOT** es decidible, y por tanto que existe un procedimiento general para saber si un programa converge sobre todos los posibles datos o, por el contrario, existe algún posible cómputo divergente para el mismo. Se puede entonces diseñar el siguiente procedimiento para saber si un programa $\mathbf{P}_x$ está en **K0** o no:

1. Añadir al principio de $\mathbf{P}_x$ instrucciones para que el programa ignore el dato de entrada y lo sustituya sistemáticamente por 0. Obtener como resultado el programa $\mathbf{P}_y$ (que sería una especie de versión degenerada de $\mathbf{P}_x$, que actúa sobre todos sus datos como $\mathbf{P}_x$ lo hace sobre 0)

2. Averiguar si $\mathbf{P}_y$ está o no en **TOT** (que por hipótesis es posible)

3. Si $\mathbf{y} \notin$ **TOT** es porque diverge sobre cualquier dato (ya que el comportamiento de $\mathbf{P}_y$ no depende de la entrada suministrada). Pero si esto es así es porque $\mathbf{P}_x$ no converge sobre 0, y por ello deducimos que **x** tampoco está en **K0** y terminamos

4. Si $\mathbf{y} \in$ **TOT** deducimos análogamente que **x** también está en **K0** y terminamos

Por tanto, $\mathbf{TOT} \in \Sigma_0 \Rightarrow \mathbf{K0} \in \Sigma_0$, y lo que es más importante, $\mathbf{K0} \notin \Sigma_0 \Rightarrow \mathbf{TOT} \notin \Sigma_0$. Como se puede probar la primera indecidibilidad (lo haremos en una sección posterior), esta nos conducirá automáticamente a la segunda. Al igual que antes, también se puede probar que la decidibilidad de **K0** ***no implica*** la de **TOT**, pero tampoco lo haremos aquí[9].

Con este ejemplo intuitivo hemos querido vislumbrar el efecto que la técnica de la reducción, que desarrollaremos en los siguientes capítulos, tiene en la práctica de la Teoría de la Computabilidad: la posibilidad de entretejer una red de relaciones jerárquicas entre problemas con algún grado de similaridad de manera que la incomputabilidad de uno de ellos se propague con efecto dominó a lo largo de la jerarquía.

[8] La forma más sencilla de hacerlo es comprobar que **A1000** sí es decidible pero **K0** no. De hecho el procedimiento descrito más arriba es un poco tramposo, ya que utiliza la decidibilidad de **K0** como un paso presuntamente necesario para probar la de **A1000**, cuando es relativamente sencillo escribir un procedimiento más abreviado que decide computablemente sobre la pertenencia a **A1000** sin precondiciones.

[9] En este caso demostraríamos que **K0** es semidecidible pero **TOT** no.

4. Reducción en Computabilidad

Para ilustrar la noción de reducción de una manera informal recurrimos al ejemplo de B. Moret [Mor 98]. Un matemático está en una habitación que contiene una mesa, un grifo y un recipiente vacío en el suelo. Se le pide que ponga el recipiente lleno de agua en la mesa. Rápidamente coge el recipiente, lo llena de agua y lo coloca en la mesa. Un rato más tarde está en la habitación equipada como antes, con la diferencia de que ahora el recipiente está lleno en el suelo. Cuando se le pide que resuelva la misma tarea, el matemático vacía el recipiente, lo deja en el suelo y responde: "he reducido la tarea al primer problema".

4.1. Idea intuitiva de reducción

Decimos coloquialmente que el problema **A** se reduce al problema **B** cuando basta con solucionar **B** para tener automáticamente solucionado el problema **A**. Lo que se entiende por "automáticamente" puede variar según el contexto, pero en general queremos decir que la solución del problema **B** se puede convertir en solución de **A** con poco esfuerzo adicional. En el ejemplo de Moret ese esfuerzo consistiría en vaciar el recipiente y devolverlo al suelo. Nótese que aunque la palabra reducción puede sugerir simplificación, no debe entenderse así en el contexto en el que la estamos utilizando: la solución obtenida por el matemático para el segundo problema no es desde luego simple ni "reducida" en términos absolutos, pues implicaría la absurda operación de vaciar el recipiente para volverlo a llenar. Lo que hace dicha solución es poner de manifiesto que existe una relación entre los dos problemas que se le han planteado.

En términos computacionales la reducción se utiliza muchas veces de forma más o menos explícita. Es sabido que la operación de multiplicación de números naturales puede realizarse mediante sumas iteradas, de la forma:

```
X0:= 0; CONT:= 0;
while CONT≠X2 loop X0:= X0+X1; CONT:= CONT+1; end loop;
```

El anterior programa muestra que el producto se reduce a la suma, ya que si se tiene un algoritmo para realizar sumas se puede construir otro que realice productos. Este hecho evoca la idea de que la multiplicación es una operación más complicada que la

adición, pues la primera se construye sobre la segunda con una cierta cantidad de trabajo. Sin embargo considérese el siguiente programa:

```
A:= 10**X1;  B:= 10**X2;  X0:= log(A*B);
```

que permite computar la suma de dos números a partir del producto. Lo que sucede en este caso es que la utilización de operaciones como el logaritmo o la potencia sugiere que el esfuerzo necesario para reducir la suma al producto es excesiva. Sin embargo no lo consideraremos así en nuestro contexto, y a la hora de reducir un problema a otro tomaremos como razonable cualquier operación adicional que sea computable.

Otra cuestión importante es que la reducción de un problema a otro es siempre un hecho condicional. Si **A** se reduce a **B** existirá un procedimiento que nos resuelve **A** a partir de una solución de **B**. Pero ello no implica que exista dicha solución de **B**, solo que si existiese entonces podría convertirse en una de **A**. Tampoco implica que la solución de **B** sea imprescindible para resolver **A**, ya que podría haber otras maneras diferentes de resolverlo. Son posibles las siguientes situaciones:

- **B** tiene solución (y **A** también, necesariamente)
- **B** no tiene solución pero **A** sí
- Ni **A** ni **B** tienen solución

Veámoslo con un ejemplo de la vida ordinaria. Imaginemos que un policía tiene que resolver un caso y averiguar quién es el asesino. Llamaremos a esto el problema **A**. No hay testigos, pero el criminal dejó en la escena un zapato hecho a mano por un zapatero de la ciudad, aparentemente fácil de identificar. Naturalmente a la policía le interesa que el zapatero indique para quién fabricó ese zapato. Conseguir ese testimonio será el problema **B**, y podemos asegurar que **A** se reduce a **B**, ya que si se resuelve el segundo problema automáticamente se resuelve también el primero. Esta reducción es lo único seguro que podemos afirmar, la certeza de que si **B** tiene solución entonces **A** también la tiene. Pero, ¿conseguirá la policía resolver el asesinato? Como hemos indicado antes, se pueden dar diferentes situaciones que respetan los hechos conocidos:

- Podemos conseguir que el zapatero testifique y por tanto podemos resolver el caso (hay solución para **B** y **A**).
- Es imposible obtener el testimonio del zapatero (por ejemplo, porque no le gusta colaborar con la policía, porque tiene un miedo insuperable, porque fabrica muchos zapatos iguales o porque ha sido asesinado a su vez). Entonces o bien existen otras formas de resolver el caso (otras pruebas materiales,

testigos inesperados o confesión espontánea del culpable), o no existen. Cuando **B** no tiene solución ello no afecta al estatuto de **A**, puede que la tenga o que no.

Lo que también nos indica este ejemplo es que la reducción a veces implica la búsqueda de una solución indirecta de los problemas. En este caso para obtener respuesta a la pregunta ¿quién lo mató?, que no es susceptible de ser respondida, tratamos de responder a la cuestión ¿de quién es este zapato?. Veremos que en los esquemas de reducción es muy habitual tratar de encontrar escenarios y preguntas diferentes que nos permitan resolver de manera indirecta problemas aparentemente sin relación.

4.2. El esquema de la reducción

Antes de trasladar la idea de reducción al contexto de la Teoría de la Computabilidad conviene que nos pongamos en guardia sobre algunos peligros argumentales en los que se puede incurrir si no se tiene una actitud cuidadosa.

En el capítulo anterior comentamos que las demostraciones de computabilidad de unas funciones facilitaban las de otras porque las funciones computables conocidas pueden usarse como subrutinas para hacer algoritmos más complejos. Por contra, la extensión de esta idea a las demostraciones de incomputabilidad produce argumentos falaces: aunque usemos una función incomputable ψ en el programa para computar otra función χ, esto no convierte a la segunda en incomputable.

El riesgo de hacer este tipo de razonamiento es mayor de lo que puede parecer a primera vista. Imaginemos que queremos demostrar que el conjunto $\mathbf{K0} = \{ \mathbf{x} : \varphi_x(0)\downarrow \}$ es indecidible. Un argumento (falso) en el que a veces se cae es el siguiente:

1. Para decidir si un programa $\mathbf{P}_x$ pertenece a **K0** hay que determinar si $\mathbf{P}_x(0)\downarrow$
2. Para decidir si $\mathbf{P}_x(0)\downarrow$ habría que resolver el problema de parada
3. Como esto último no es posible, no hay forma de decidir si $\mathbf{P}_x(0)\downarrow$ o no.

El razonamiento subyacente es falaz por el uso de la expresión "hay que". Efectivamente, si pudiéramos resolver el problema de parada, la función característica de $\mathbf{C}_{K0}$ sería automáticamente computable mediante el siguiente programa, que demuestra que **K0** se puede reducir a **H**:

```
X0:= H(X1, 0);
```

Esto indica una posible vía para computar $\mathbf{C}_{K0}$, pero en ningún caso que sea la única posible. El fracaso de una aproximación inadecuada al problema no implica que este no tenga solución. El argumento anterior simplemente no aporta nada al problema de la decidibilidad de **K0**. Nos dice que si **H** fuera decidible **K0** también lo sería, pero como **H** no lo es, no podemos deducir nada. También nos dice que si **K0** fuera indecidible **H** también lo sería. Como la indecidibilidad de **H** ya está probada, la implicación es irrelevante.

En general, cuando tengamos una función ψ cuya incomputabilidad es conocida, y otra función χ cuya incomputabilidad sospechamos y queremos probar, hay que tener cuidado con las líneas argumentales del estilo *para computar χ es necesario utilizar ψ, y como ψ es incomputable, entonces χ también lo es*. Lo que estamos haciendo es indicar la existencia de un algoritmo para computar χ cuyo texto invoca ψ (figura 4.1), lo que no prueba nada. Simplemente tendríamos una forma "ilegal" de computar χ, pero ello no implicaría la inexistencia de alguna otra forma "legal" de hacerlo.

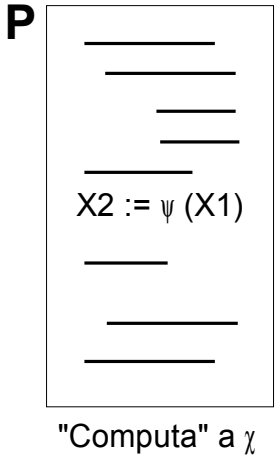


Fig 4.1: Si ψ es computable su inclusión en un programa que compute a χ permite demostrar que esta última también es computable. Pero si ψ es incomputable, el haberla utilizado para computar χ no nos dice nada de esta última.

Pero, ¿qué sucedería si hiciéramos a la inversa? Si utilizáramos χ en un programa **Q** para computar ψ (figura 4.2) podríamos tener una interesante situación, ya que sabemos de antemano que ningún programa puede computar ψ.

Si quitando la invocación a χ todos los demás elementos del programa fueran "legales" (de computabilidad contrastada), entonces podríamos hacer la siguiente deducción: con la sola hipótesis de que χ fuera computable se podría construir el macroprograma **Q** que demostraría la computabilidad de ψ , y como sabemos que esto es

imposible concluimos que la hipótesis de computabilidad de χ ha de ser necesariamente falsa.

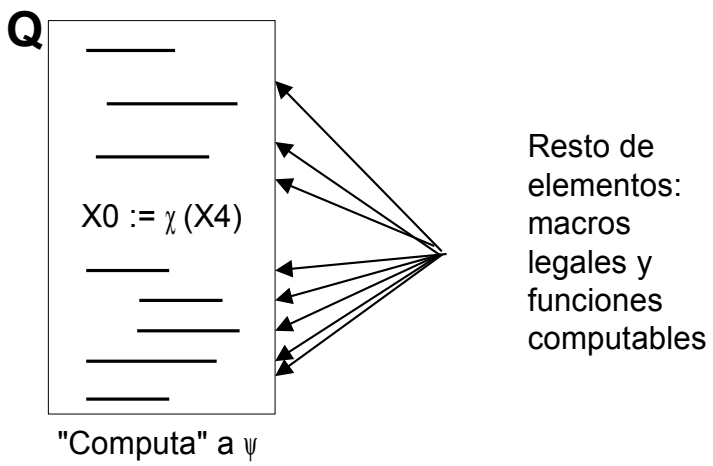


Fig 4.2: Si Ψ es incomputable, el que χ nos proporcionara un medio para computarla sería una prueba de que χ tampoco puede ser computable.

La moraleja podría ser que para ver que Ψ es incomputable no intentemos construirla a base de absurdos; probemos más bien a construir un absurdo con ella. Dicho de otra manera, la reducción debe ser usada en el sentido apropiado. Si queremos demostrar que Ψ es incomputable no intentemos reducirla a otros problemas incomputables, sino más bien tratemos de reducir otros problemas incomputables a Ψ.

Esta es la esencia del método de reducción, técnica que, como veremos, hará más grata la edificación de la Teoría de la Computabilidad al exhibir un patrón incremental que permitirá utilizar viejas incomputabilidades para obtener otras nuevas.

4.3. Ejemplos de aplicación de la reducción

Realizaremos una serie de ejemplos de aplicación del esquema y observaremos que, efectivamente, éste tiene éxito allí donde la diagonalización falla. Para simplificar utilizaremos el esquema de reducción normalmente sobre el mismo punto de partida conocido, que será el problema de parada **H**. Así veremos que este problema se puede reducir a una serie de funciones que involucran propiedades dinámicas de los programas, y por tanto tienen al menos un argumento en $\mathbb{W}$.

Como veremos, en cada caso reduciremos el problema de parada al que tengamos entre manos para demostrar su incomputabilidad. Construiremos por tanto un

procedimiento para decidir si un programa cualquiera $\mathbf{P}_x$ converge sobre un dato cualquiera **y**.

4.3.1. Un problema de depuración de programas

Consideremos el siguiente problema de programación: cuando construimos un programa es razonable esperar que toda variable reciba explícitamente algún valor antes de ser usada por primera vez en una expresión, pues de lo contrario pueden producirse errores inesperados. Por tanto sería deseable contar con una rutina automática que nos avisara en tiempo de compilación cuándo esto no se produce. Desafortunadamente tal rutina es imposible de ser programada.

Naturalmente el que una variable sea inicializada adecuadamente depende de la computación concreta. Además para simplificar nos vamos a centrar en la versión más sencilla posible y nos interesaremos por lo que sucede con una única variable. El problema podría establecerse de la siguiente manera. Dados un programa $\mathbf{P}_x$, un dato **y**, y una variable **XN**, ¿la ejecución de $\mathbf{P}_x(\mathbf{y})$ se realiza de forma que, cuando **XN** es utilizada, se hace tras la preceptiva inicialización? La respuesta es negativa sólo si se produce una violación del principio *inicializar antes de usar*, por lo que si **XN** no es utilizada en esa computación concreta (o si ni siquiera aparece en el texto del programa) diremos que el uso de **XN** es correcto. Adviértase que no nos metemos en el delicado asunto de si $\mathbf{P}_x(\mathbf{y})\downarrow$. Cuando la computación es divergente solamente nos planteamos qué uso se da a la variable **XN** en ese cómputo infinito (que puede ser correcto o no).

Formalmente consideremos el predicado **INIT**(**x**,**y**,**n**), que se cumple si y solo si al ejecutar el programa $\mathbf{P}_x$ sobre el dato **y** la variable **XN** nunca es utilizada en algún momento anterior a su inicialización explícita (es decir, su primera aparición, de haberla, en la ejecución del proceso corresponde a la parte izquierda de una asignación, en lugar de a una parte derecha o a una condición).

Al examinar este predicado para estudiar su posible decidibilidad, se puede caer en un argumento falaz del tipo indicado en la Sección 4.2. Si un programa tiene la siguiente estructura:

```
P
X0:= cdr(XN);
Q
```

donde la asignación entre **P** y **Q** es la primera en la que aparece **XN**, esta variable se usará incorrectamente sólo cuando **P** termine su ejecución, porque en caso contrario no llegará a ser utilizada. Por tanto, para determinar si **XN** no es usada sin inicializar hemos de averiguar si **P** termina, y como esto es imposible deducimos que **INIT** no puede ser decidible.

Como se ha indicado, este argumento falla porque se basa en proponer una solución para computar **INIT** y ver que no funciona. No se trata de encontrar que **H** puede ser usado para computar **INIT** (cosa que, por otro lado sólo se hace de forma parcial, ya que hay programas que no se ajustan a la estructura indicada), sino justamente de lo contrario. ¿Podría ser usado **INIT** para resolver el problema de parada?. O, dicho de otra manera, ¿se puede reducir **H** a **INIT**?. Veremos que así es, y que por tanto **INIT** no es decidible.

Para reducir **H** a **INIT** construimos un algoritmo que decida, gracias a **INIT**, si un programa $\mathbf{P}_x$ converge o no sobre un dato **y**. Dado que no podemos preguntar directamente (recuérdese el ejemplo del asesinato) lo que haremos es movernos a otro terreno en el que podamos hacer una pregunta diferente (relacionada con **INIT**), pero cuya respuesta nos permita resolver el problema original.

1. Basándonos en $\mathbf{P}_x$ construimos otro programa $\mathbf{P}_z$, que resulta de añadir a $\mathbf{P}_x$ una instrucción adicional

   $$P_z \equiv P_x \; X0 := cdr(XN);$$

   Se trata de una asignación que tiene en su parte derecha una variable especial **XN**, elegida con cuidado para que no aparezca en ningún otro lugar de $\mathbf{P}_z$ (por lo que nunca podrá ser inicializada).

2. Preguntamos si **INIT**(**z**,**y**,**n**)

3. Devolvemos la negación del resultado anterior

Es evidente que $\mathbf{P}_z$ sólo puede utilizar **XN** antes de inicializarla si consigue llegar a la instrucción fatídica, y esto solo sucede si $\mathbf{P}_x$ termina, por lo que se verifica que **INIT**(**z**,**y**,**n**) $\leftrightarrow \neg$**H**(**x**,**y**). Concluimos que el procedimiento anterior sirve para calcular los valores del predicado **H**.

Lo haremos de modo más formal construyendo un programa **Q** que compute dicho procedimiento. La variable especial debe elegirse de forma que no esté en $\mathbf{P}_x$, para lo que bastará tomar un índice de variable mayor que cualquiera de los que aparecen en él:

```
-- Primero calculamos cuál va a ser la variable especial[10]. Recuérdese que en X1 está el programa Px
 N:= ult_variable(X1)+1;
-- Añadimos a Px la asignación que utiliza la variable elegida en el lado derecho
 Z:= haz_composición(X1, haz_asig_cdr(0, N));
-- Recuérdese que en X2 está el dato y
 X0:= not INIT(Z, X2, N);
```

Si **INIT** fuera decidible entonces **Q** sería un programa legal que computaría **H**. Como esto es imposible deduciremos que **INIT** no puede ser decidible.

El resultado que hemos visto es coherente con lo que sucede en la informática real. Algunos lenguajes de programación tienen valores por defecto para inicializar las variables de los programas. La mayoría simplemente ignora el problema y en el caso de variables no inicializadas, toma el valor que circunstancialmente se encuentre en la memoria asignada a la variable en caso de necesidad. Sería concebible una aproximación en la que el compilador añada una tabla de inicializaciones al código del programa objeto para que produzca un error en tiempo de ejecución si una variable es usada sin inicializar. Pero en ningún caso el compilador avisa de la eventualidad, por la sencilla razón de que no hay modo de hacerlo en tiempo de compilación.

Naturalmente la dificultad de hacer este tipo de predicciones reside en el hecho de que un programa puede ciclar. Si todos los programas terminasen sería posible simular su ejecución hasta el punto en que utilizan la variable que se quiere monitorizar, o hasta al final en caso de que no la utilicen, e indicar si ha habido o no un uso incorrecto. Por ejemplo, se puede demostrar que la siguiente función es computable:

$$\psi(\mathbf{x},\mathbf{y},\mathbf{n}) \cong \begin{cases} \text{true} & \text{si en el cómputo de } \mathbf{P}_x(\mathbf{y}) \text{ la variable } \mathbf{XN} \\ & \quad \text{es inicializada antes de cualquier posible uso} \\ \text{false} & \text{si en el cómputo de } \mathbf{P}_x(\mathbf{y}) \text{ la variable } \mathbf{XN} \\ & \quad \text{es utilizada sin previa inicialización} \\ \text{true} & \text{si en el cómputo de } \mathbf{P}_x(\mathbf{y}) \text{ la variable } \mathbf{XN} \\ & \quad \text{no es ni inicializada ni utilizada y } \mathbf{P}_x(\mathbf{y})\downarrow \\ \perp & \text{si en el cómputo de } \mathbf{P}_x(\mathbf{y}) \text{ la variable } \mathbf{XN} \\ & \quad \text{no es ni inicializada ni utilizada pero } \mathbf{P}_x(\mathbf{y})\uparrow \end{cases}$$

[10] La demostración de la computabilidad de la función **ult_variable** puede encontrarse en el apéndice.

para lo cuál es preciso utilizar las técnicas propias de la implementación de la función universal, que pueden consultarse en [ISI 03].

4.3.2. Otro problema de traza

Se podrían definir muchos otros ejemplos para ilustrar la imposibilidad de predecir en general las vicisitudes del cómputo de un programa. Veremos otro ejemplo que nos permite profundizar en la técnica. Basándonos en el concepto de anidamiento de instrucciones definimos el siguiente predicado:

NEST = {(**x**,**y**,**n**): la ejecución de $\mathbf{P}_x$ sobre **y** entra en alguna instrucción con nivel **n** de anidamiento}

Por razones análogas a las del apartado anterior, el conjunto **NEST** no resulta decidible, y lo probaremos por reducción. Para ello veremos que **H** se puede reducir a este problema, con lo que si $\mathbf{C}_{NEST}$ fuera computable **H** también lo sería.

¿Cómo podríamos utilizar $\mathbf{C}_{NEST}$ para computar **H**?. Supongamos que nos dan un programa $\mathbf{P}_x$ y un dato **y**, y que queremos averiguar con ayuda de $\mathbf{C}_{NEST}$ si $P_x(\mathbf{y})\downarrow$. Podemos obtener el programa $\mathbf{P}_z$ añadiendo al final de $\mathbf{P}_x$ un bucle con un número **n** de anidamientos mayor que el de cualquier instrucción de $\mathbf{P}_x$. Esta sería la estructura de $\mathbf{P}_z$:

```
Px
X1:= consa(X1);
while nonem?(X1) loop
        while nonem?(X1) loop
           . . .
              while nonem?(X1) loop
                 X1:= ε;
              end loop;
           . . .
        end loop;
end loop;
```

De este modo sabremos que la única forma de que $\mathbf{P}_z$ entre en una instrucción anidada a nivel **n** es llegando a este bucle que hemos añadido, para lo cual $\mathbf{P}_x$ debe terminar primero. Alternativamente, si $\mathbf{P}_z$ no entra nunca en una instrucción anidada a nivel **n** es porque no ha conseguido llegar a completar la ejecución de $\mathbf{P}_x$, por lo que este

debe ciclar (el bucle final se ha construido de tal modo que si se llega a él el programa entra necesariamente hasta su instrucción más interna)

Por ello, si preguntamos $\mathbf{C}_{\text{NEST}}(\mathbf{z},\mathbf{y},\mathbf{n})$ la respuesta que obtengamos nos indica con certeza si $\mathbf{H}(\mathbf{x},\mathbf{y})$ o no. Siguiendo estas ideas, vamos a escribir el programa que computaría **H** (y demostraría la indecidibilidad de **NEST**):

```
-- Primero calculamos el nivel de anidamiento[11] al que debemos llegar
 N:= nivel_anidamiento(X1) + 1;
-- Inicializamos el programa Pz con el cuerpo del while más interno
 Z:= haz_asig_vacía(1);
-- Construimos todos los bucles necesarios
 for I in 1..N loop
          Z:= haz_iteración(1, Z);
 end loop;
-- Añadimos al principio la instrucción que nos asegura la entrada en los bucles anidados
 Z:= haz_composición(haz_asig_cons_a(1, 1), Z);
-- Añadimos todo ello detrás del programa Px
 Z:= haz_composición(X1, Z);
 X0:= CNEST(Z, X2, N);
```

4.3.3. Superando la diagonalización: K0 tampoco es decidible

Para terminar veamos cómo podemos demostrar por reducción la indecidibilidad de **K0**, conjunto interesante porque vimos en la Sección 3.3 que no podíamos aplicarle el método de diagonalización. Recordemos que $\mathbf{K0} = \{ \mathbf{x} : \varphi_x(0)\downarrow \}$. Al igual que en los casos anteriores, trataremos de utilizar la función $\mathbf{C}_{K0}$ sobre un programa $\mathbf{P}_z$ relacionado con $\mathbf{P}_x$, de forma que en función de que **z** pertenezca o no a **K0** podamos averiguar algo sobre la convergencia de $\mathbf{P}_x(\mathbf{y})$.

El siguiente procedimiento permitirá computar **H**:

1. Tomemos un programa $\mathbf{P}_z$ y un dato **y** sobre el que queremos determinar si converge o no.

2. Construimos otro programa $\mathbf{P}_z$ que resulta de añadir a $\mathbf{P}_x$ instrucciones adicionales al principio. En este caso una serie de asignaciones que obligan a que el dato de entrada (sobre el que se ejecutará el código de $\mathbf{P}_x$) sea justo **y**.

3. Preguntamos por $\mathbf{C}_{K0}(\mathbf{z})$ y devolvemos la respuesta como resultado.

[11] La demostración de la computabilidad de la función **nivel_anidamiento** puede encontrarse en el apéndice.

La estructura del programa $\mathbf{P}_z$ será la siguiente:

X1 := y;
P_x

Es evidente que $\mathbf{P}_z$ termina (además sobre cualquier dato) si y sólo si $\mathbf{P}_x$ lo hace sobre el dato **y**, por lo que se verifica que $\mathbf{P}_z(0)\downarrow \leftrightarrow \mathbf{P}_x(\mathbf{y})\downarrow$, y el procedimiento habrá servido para computar **H**.

Ahora bien, la construcción del programa **Q** que implementa el procedimiento anterior es un poco más compleja, sobre todo por el riesgo de confundir las variables de los tres programas que intervienen: el propio programa **Q**, $\mathbf{P}_x$ y $\mathbf{P}_z$. **Q** tiene como entradas $\mathbf{P}_x$ (variable **X1**) e **y** (variable **X2**), y utiliza como dato intermedio $\mathbf{P}_z$ (macrovariable **Z**). Por tanto $\mathbf{P}_x$ y $\mathbf{P}_z$ no serán en realidad ejecutados, sino manipulados en sus instrucciones. Pero como esa manipulación se hace con vistas a que tengan un cierto comportamiento, tendremos presente cuáles serán las entradas que esos programas tomarían si se ejecutaran (**y** en el caso de $\mathbf{P}_x$, y 0 en el de $\mathbf{P}_z$).

Al añadir instrucciones equivalentes al macroprograma X1:= y; al principio de $\mathbf{P}_z$, hay que tener en cuenta que ese valor **y** se refiere al contenido de la variable **X2** del programa **Q**, y por tanto el macroprograma no es equivalente a la asignación X1:= X2; (que asignaría a **X1** la palabra vacía, ya que $\mathbf{P}_z$ solo tiene una variable de entrada).

Al escribir **Q** podemos usar macros como en cualquier otro programa. Sin embargo $\mathbf{P}_z$ es construido por **Q**. Como los macroprogramas no forman un tipo de datos implementado (no pueden hacerlo al constituir una notación abierta), **Q** solo puede utilizar programas-while puros, y $\mathbf{P}_z$ deberá ser diseñado sin macros. Así, la instrucción X1:= y; será una secuencia de asignaciones que servirá para construir el valor **y**, símbolo a símbolo, de derecha a izquierda, a partir de la palabra vacía. Por todo ello el programa $\mathbf{P}_z$ que ha de construir **Q** a partir de $\mathbf{P}_x$ será como sigue (hemos supuesto que $\mathbf{y} = \mathbf{s}_1\mathbf{s}_2...\mathbf{s}_k$):

X1:= ε;
X1:= $cons_{sk}$(X1);
X1:= $cons_{s(k-1)}$(X1);
....
X1:= $cons_{s2}$(X1);
X1:= $cons_{s1}$(X1);
P_x

Tal como previmos, $\mathbf{P}_z$ es un programa que siempre se comporta de la misma manera, independientemente de la entrada.

Ahora podemos construir el programa **Q** que se encarga de construir $\mathbf{P}_z$ con las operaciones disponibles para el tipo de datos de los programas-while. Aunque podríamos hacerlo para cualquier alfabeto, vamos a suponer por simplicidad que $\Sigma = \{\mathbf{a}, \mathbf{b}\}$

```
Z:= haz_asig_vacía(1);
-- Hay que averiguar cuáles son los símbolos de y y construir la secuencia de asignaciones apropiada
AUX:= X2^R;
while nonem?(AUX) loop
        if car_a?(AUX) then
            Z:= haz_composición(Z, haz_asig_cons_a(1, 1));
        end if;
        if car_b?(AUX) then
            Z:= haz_composición(Z, haz_asig_cons_b(1, 1));
        end if;
        AUX := cdr(AUX);
end loop;
Z:= haz_composición(Z, X1);
X0:= C_K0(Z);
```

De este modo concluimos que **K0** no puede ser decidible, pues de serlo el programa **Q** computaría **H**, que ya sabemos que no es decidible.

5. La reducción como jerarquía entre conjuntos

El esquema de reducción seguido en el capítulo anterior es bastante rígido y sigue una estructura bien definida, con algunos pasos que se realizan de manera mecánica y otros no tanto. Partiendo de un problema **A** cuya indecidibilidad queremos demostrar:

1. Tomamos un problema que sabemos indecidible (hasta ahora ha sido **H** en todos los casos) y sospechamos puede ser reducible a **A**.

2. Planificamos la construcción de un programa **Q** que resuelva el problema **H**, utilizando la supuesta computabilidad de **A**. Para ello **Q** debe determinar si un programa arbitrario $\mathbf{P}_x$ converge sobre un dato cualquiera **y**.

    - La primera parte de **Q** consiste en construir a partir de $\mathbf{P}_x$ otro programa $\mathbf{P}_z$, estrechamente relacionado.
    - La segunda parte de **Q** consiste en preguntar si $\mathbf{P}_z$ satisface o no **A**, y si la elección de $\mathbf{P}_z$ es correcta ello nos indicará si $\mathbf{P}_x(\mathbf{y})\downarrow$ o no.

La experiencia nos indica que el nudo gordiano de las pruebas de reducción se encuentra en la construcción del programa $\mathbf{P}_z$ a partir de $\mathbf{P}_x$, siendo los demás pasos razonablemente previsibles. La esencia de la reducción se concentra en una transformación de programas, y esta tiene tal relevancia que la sistematizaremos señalando el proceso constructivo en términos funcionales: el programa **Q** utiliza un mecanismo de transformación **M** que a partir de $\mathbf{P}_x$ obtiene otro programa $\mathbf{P}_{M(x)}$. Dado que **Q** sólo puede utilizar instrucciones válidas en el lenguaje de los programas-while, y que siempre produce un resultado (si no fuera así **Q** ciclaría y no podría resolver el problema de parada en todos los casos), el mecanismo utilizado por **Q** puede ser formalizado como una función computable y total **h**: $\mathbb{W} \times \sum^* \rightarrow \mathbb{W}$ que transforme cada programa original **x** en el nuevo programa **h**(**x**,**y**) (lo que anteriormente llamábamos **z**). La verdadera operación de reducción es realizada por esta función. La función **h** parte del par (**x**,**y**) sobre el que se quiere determinar si **H**(**x**,**y**), aunque en ocasiones no necesita utilizar el dato **y** para producir el programa transformado. En los ejemplos del capítulo anterior el uso de la función **h** ha sido el siguiente:

- En el caso de **INIT** el programa transformado ha sido utilizado para obtener la reducción **H**(**x**,**y**) ⇔ **INIT**(**h**(**x**), **y**, **ult_variable**(**x**)+1), de forma que $\mathbf{P}_{h(x)}$ utiliza

cierta variable no inicializada cuando opera sobre el dato **y** en función de si $\mathbf{P}_x$ converge o no sobre ese mismo dato.

- En el caso de **NEST** el programa transformado ha sido utilizado para obtener la reducción $\mathbf{H}(\mathbf{x},\mathbf{y}) \Leftrightarrow \mathbf{NEST}(\mathbf{h}(\mathbf{x}), \mathbf{y}, \mathbf{nivel\_anidamiento}(\mathbf{x})+1)$, de forma que $\mathbf{P}_{\mathbf{h}(\mathbf{x})}$ alcanza un cierto nivel de anidamiento en su ejecución sobre el dato **y** en función de si $\mathbf{P}_x$ converge o no sobre ese mismo dato.
- En el caso de **K0** el programa transformado ha sido utilizado para obtener la reducción $\mathbf{H}(\mathbf{x},\mathbf{y}) \Leftrightarrow \mathbf{C}_{K0}(\mathbf{h}(\mathbf{x},\mathbf{y}))$, de forma que $\mathbf{P}_{\mathbf{h}(\mathbf{x},\mathbf{y})}$ se comporta como lo haría $\mathbf{P}_x$ sobre **y** independientemente del dato **z** que se le suministre. Este es el único caso en el que **y** se ha utilizado en la construcción del programa transformado.

Veremos que se obtienen importantes ventajas si se sistematiza el esquema visto haciendo abstracción de todos los pasos de reducción que no sean la construcción de la función **h**.

5.1. Reducibilidad

Aunque los pasos que vamos a dar restringen algo el campo de aplicación del método, vamos a comprobar que si las funciones implicadas (tanto la incomputable de partida como aquella cuyo estatuto desea ser probado) son predicados de un solo argumento[12], podemos introducir algo de maquinaria teórica que nos permita enfocar el problema de encontrar una reducción en la consecución de la función **h**.

DEFINICIÓN: Dados dos conjuntos $\mathbf{A}, \mathbf{B} \subseteq \sum^*$, decimos que **A** ***es reducible a*** **B**, y lo notamos como $\mathbf{A} \leq \mathbf{B}$, si existe una función **h** computable y total que verifica:

$$\forall \mathbf{x} \in \sum^* (\mathbf{x} \in \mathbf{A} \leftrightarrow \mathbf{h}(\mathbf{x}) \in \mathbf{B})$$

La reducibilidad es un concepto que se extiende de manera natural a otros dominios diferentes de $\sum$. La única condición para poder hablar de reducibilidad de un conjunto $\mathbf{A} \subseteq \mathbb{T}$ es que el tipo de datos $\mathbb{T}$ esté implementado y tenga por tanto sentido hablar de funciones computables con origen y destino en $\mathbb{T}$. De hecho será bastante habitual que hablemos de relaciones de reducibilidad entre conjuntos de programas de la forma $\mathbf{A} \subseteq \mathbb{W}$.

[12] En realidad el esquema podría generalizarse fácilmente para predicados de más argumentos, utilizando por ejemplo funciones de codificación como las descritas en el apéndice.

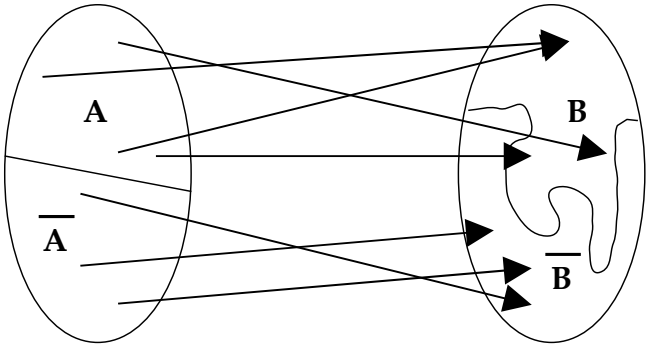


Fig. 5.1: Esquema de la actuación de una función de reducibilidad entre dos conjuntos **A** y **B**.

Desde el punto de vista conjuntista la transformación operada por **h** es muy sencilla de visualizar: los elementos de **A** son trasladados a **B**, y los de $\overline{\mathbf{A}}$ a $\overline{\mathbf{B}}$. Esto hace que cualquier herramienta de distinción que exista para los elementos de **B** sea utilizable para **A**, y en particular el problema de pertenencia al conjunto **A** se pueda reducir al problema de pertenencia al conjunto **B**. En efecto, para saber si $\mathbf{x}\in\mathbf{A}$ bastará con calcular la palabra transformada $\mathbf{h}(\mathbf{x})$ y averiguar si $\mathbf{h}(\mathbf{x})\in\mathbf{B}$ o no. Como siempre nos sucede con las variantes del concepto de reducción esto no es sino una afirmación condicional: no sabemos si es posible o no resolver los problemas de pertenencia a **A** o **B**, lo único que se afirma es que si se puede resolver el de **B** entonces automáticamente queda resuelto el de **A**. Esta dependencia entre las propiedades de decidibilidad de ambos conjuntos queda bien reflejada en las siguientes propiedades.

PROPIEDAD 1 (RELACIÓN ENTRE REDUCIBILIDAD Y DECIDIBILIDAD): Si **A** es reducible a **B** y **B** es decidible, entonces **A** también es decidible. Expresado formalmente:

$$\mathbf{A}\leq\mathbf{B}\wedge\mathbf{B}\in\Sigma_0\ \Rightarrow\ \mathbf{A}\in\Sigma_0$$

⌦ Por definición de reducibilidad, si $\mathbf{A}\leq\mathbf{B}$ existe **h** función computable y total tal que $\forall\mathbf{x}\in\sum^*\ (\mathbf{x}\in\mathbf{A}\leftrightarrow\mathbf{h}(\mathbf{x})\in\mathbf{B})$. Como $\mathbf{B}\in\Sigma_0$, su función característica $\mathbf{C_B}$ es computable. La composición de dos funciones computables es computable, luego $\mathbf{C_B}\circ\mathbf{h}$ es computable. Pero se tiene que, para cualquier $\mathbf{x}\in\sum^*$:

$$\mathbf{x}\in\mathbf{A}\Rightarrow\mathbf{h}(\mathbf{x})\in\mathbf{B}\Rightarrow\mathbf{C_B}\circ\mathbf{h}(\mathbf{x})=\text{true}$$

$$\mathbf{x}\notin\mathbf{A}\Rightarrow\mathbf{h}(\mathbf{x})\notin\mathbf{B}\Rightarrow\mathbf{C_B}\circ\mathbf{h}(\mathbf{x})=\text{false}$$

De donde se deduce que $\mathbf{C_B}\circ\mathbf{h}$ es precisamente $\mathbf{C_A}$, la función característica de **A**. Así pues tenemos que $\mathbf{C_A}$ es computable, por lo que **A** tiene que ser decidible. ⌫

PROPIEDAD 2 (RELACIÓN ENTRE REDUCIBILIDAD Y SEMIDECIDIBILIDAD): Si **A** es reducible a **B** y **B** es semidecidible, entonces **A** también es semidecidible. Expresado formalmente:

$$\mathbf{A} \leq \mathbf{B} \wedge \mathbf{B} \in \Sigma_1 \Rightarrow \mathbf{A} \in \Sigma_1$$

⌦ Por definición de reducibilidad, si $\mathbf{A} \leq \mathbf{B}$ existe **h** función computable y total tal que $\forall \mathbf{x} \in \sum^* (\mathbf{x} \in \mathbf{A} \leftrightarrow \mathbf{h}(\mathbf{x}) \in \mathbf{B})$. Como $\mathbf{B} \in \Sigma_1$, su función semicaracterística χ_B es computable. La composición de dos funciones computables es computable, luego $\chi_B \circ \mathbf{h}$ es computable. Pero se tiene que, para cualquier $\mathbf{x} \in \sum^*$:

$$\mathbf{x} \in \mathbf{A} \Rightarrow \mathbf{h}(\mathbf{x}) \in \mathbf{B} \Rightarrow \chi_B \circ \mathbf{h}(\mathbf{x}) = \text{true}$$

$$\mathbf{x} \notin \mathbf{A} \Rightarrow \mathbf{h}(\mathbf{x}) \notin \mathbf{B} \Rightarrow \chi_B \circ \mathbf{h}(\mathbf{x}) \uparrow$$

De donde se deduce que $\chi_B \circ \mathbf{h}$ es precisamente χ_A, la función semicaracterística de **A**. Por tanto tenemos que χ_A es computable, por lo que **A** tiene que ser semidecidible. ⌫

Normalmente las propiedades anteriores se aplican de manera negativa para demostrar la indecidibilidad y no semidecidibilidad de conjuntos, que es donde necesitamos más herramientas:

$$\mathbf{A} \leq \mathbf{B} \wedge \mathbf{A} \notin \Sigma_0 \Rightarrow \mathbf{B} \notin \Sigma_0$$

$$\mathbf{A} \leq \mathbf{B} \wedge \mathbf{A} \notin \Sigma_1 \Rightarrow \mathbf{B} \notin \Sigma_1$$

A partir de este momento tenemos una vía rápida alternativa para mostrar la indecidibilidad (respectivamente, la no semidecidibilidad) de un conjunto cualquiera **B**: basta con encontrar otro conjunto indecidible (respectivamente, no semidecidible) que sea reducible a él. Para la inmensa mayoría de los casos, además, no es necesario romperse la cabeza, pues suele ser sencillo y eficaz utilizar el conjunto **K** (respectivamente, $\overline{\mathbf{K}}$). Aunque también se pueden utilizar, y en algunos casos será conveniente, otros conjuntos no semidecidibles como **TOT** o $\overline{\mathbf{TOT}}$.

5.2. Reducibilidad y dificultad relativa

Las propiedades que hemos demostrado en la sección anterior nos dan una pista acerca de cómo entender la relación de reducibilidad. Existe una interpretación intuitiva bastante natural: $\mathbf{A} \leq \mathbf{B}$ viene a indicar que el conjunto **A** es "más sencillo" que el conjunto **B** desde el punto de vista de la computabilidad, o lo que es lo mismo, que la dificultad de

determinar si un **x** cualquiera está en **A** es tanto o más fácil que el de averiguar si está en **B**. Como hemos visto, cuando $\mathbf{A} \leq \mathbf{B}$ las principales herramientas de decisión sobre **B** son automáticamente transmisibles a **A**, y con ellas sus propiedades de decidibilidad[13]. Por la misma razón, las propiedades de indecidibilidad de **A** son heredadas hacia arriba por **B**: si se puede probar que **A** es un conjunto lo suficientemente "complicado" (como para no resultar decidible o semidecidible), entonces **B**, que está "por encima" de **A**, con menos razón aún (todo lo que se consiguiera para reducir el nivel de "complicación" de **B** se trasladaría automáticamente a **A**, en contradicción con lo que sabemos de este último).

Esta idea intuitiva de mayor o menor dificultad y la propia utilización del símbolo ≤ para denotar la reducibilidad vienen justificadas por el siguiente resultado:

PROPIEDAD 3 (ORDEN PARCIAL INDUCIDO POR LA REDUCIBILIDAD): La relación de reducibilidad establece un ***orden parcial*** en $\wp(\sum^*)$.

⌦ Probar la reflexividad es sencillo. Sea $\mathbf{A} \subseteq \sum^*$ un conjunto cualquiera. La función identidad **id**: $\sum^* \rightarrow \sum^*$ es claramente computable y total, y es evidente que verifica:

$$\forall \mathbf{x} \in \sum^* (\mathbf{x} \in \mathbf{A} \leftrightarrow \mathbf{id}(\mathbf{x}) \in \mathbf{A})$$

por lo que $\mathbf{A} \leq \mathbf{A}$.

Supongamos ahora que tenemos tres conjuntos $\mathbf{A},\mathbf{B},\mathbf{C} \subseteq \sum^*$ que verifican $\mathbf{A} \leq \mathbf{B}$ y $\mathbf{B} \leq \mathbf{C}$. Existirán por tanto dos funciones computables y totales **f** y **g** que cumplirán:

$$\forall \mathbf{x} \in \sum^* (\mathbf{x} \in \mathbf{A} \leftrightarrow \mathbf{f}(\mathbf{x}) \in \mathbf{B})$$

$$\forall \mathbf{x} \in \sum^* (\mathbf{x} \in \mathbf{B} \leftrightarrow \mathbf{g}(\mathbf{x}) \in \mathbf{C})$$

La función $\mathbf{h} = \mathbf{g} \circ \mathbf{f}$ también es computable y total. Sea un $\mathbf{x} \in \sum^*$ cualquiera:

$$\mathbf{x} \in \mathbf{A} \Leftrightarrow \mathbf{f}(\mathbf{x}) \in \mathbf{B} \Leftrightarrow \mathbf{g}(\mathbf{f}(\mathbf{x})) \in \mathbf{C} \Leftrightarrow \mathbf{g} \circ \mathbf{f}(\mathbf{x}) \in \mathbf{C} \Leftrightarrow \mathbf{h}(\mathbf{x}) \in \mathbf{C}$$

por lo que necesariamente $\mathbf{A} \leq \mathbf{C}$**.** La demostración de transitividad completa nuestra prueba. ⌫

[13] Existen otras propiedades de computabilidad más rebuscadas que las que son objeto del presente curso, que también son "transmitidas" mediante la relación de reducibilidad. Asimismo también existen formulaciones alternativas de la noción de reducibilidad, que si bien varían bastante en sus propiedades, siempre mantienen este aspecto de establecer una relación de mayor o menor dificultad en resolver las propiedades de computabilidad de un conjunto.

5.3. Reducibilidad e incomputabilidad: algunos ejemplos

Ilustraremos a continuación el uso de la noción de reducibilidad. Si bien esta no nos permite obviar el paso más arduo de la técnica de reducción (el diseño de la función de reducibilidad **h**), al menos nos permitirá concentrarnos en el mismo sin distracciones.

5.3.1. El caso de CONS (indecidibilidad)

Hemos hablado del conjunto **CONS** como uno de los ejemplos que muestran las limitaciones de la diagonalización. Recordemos que:

$$\mathbf{CONS} = \{ \mathbf{x} \in \mathbb{W} : \varphi_x \text{ es total y constante} \}$$

Intuitivamente parece que **CONS** no debería ser un conjunto decidible. ¿Cómo podríamos asegurar, por mera inspección de un programa, que para ante cualquier entrada y que siempre devuelve el mismo resultado? Para ver que no es posible demostraremos que se encuentra por encima de otro conjunto de conocida indecidibilidad, y como se ha indicado empezaremos con la elección más evidente: **K**.

⌦ Trataremos pues de probar que $\mathbf{K} \leq \mathbf{CONS}$.

Según la definición de reducibilidad se trata de encontrar una función **h** computable y total tal que

$$\forall \mathbf{x} \in \mathbb{W}\ (\mathbf{x} \in \mathbf{K} \leftrightarrow \mathbf{h}(\mathbf{x}) \in \mathbf{CONS})$$

es decir, que $\forall \mathbf{x} \in \mathbb{W}\ (\varphi_x(\mathbf{x})\downarrow \leftrightarrow \varphi_{h(x)}$ es una función constante), o en términos de programas, hemos de encontrar una función que transforme cada programa **x** en otro **h**(**x**) de manera que

- si $\mathbf{P}_x$ converge sobre el dato **x** entonces el programa $\mathbf{P}_{h(x)}$ siempre termina, devolviendo un resultado constante
- si $\mathbf{P}_x$ diverge sobre el dato **x** entonces el programa $\mathbf{P}_{h(x)}$ no computa una función constante (por ejemplo, porque no termina para alguna entrada o porque produce resultados variados)

Naturalmente, dado un programa $\mathbf{P}_x$ no podemos preguntar alegremente si converge o no sobre **x**. En todo caso podremos poner en marcha la ejecución de $\mathbf{P}_x(\mathbf{x})$ en un entorno controlado, para que suceda lo que nos interesa en cada caso en función de su

comportamiento. Así, nuestro programa transformado $\mathbf{P}_{h(x)}$ incluirá instrucciones de la forma X1:= x; P_x dentro de su código.

Si la ejecución de $\mathbf{P}_x(\mathbf{x})$ cicla no podremos predecirlo ni evitarlo. Pero en nuestro esquema esa divergencia no es un problema, ya que $\mathbf{P}_{h(x)}$ también ciclará y obtendremos el resultado deseado porque de este modo no podrá computar una función constante.

Vayamos ahora al caso en el que $\mathbf{P}_x(\mathbf{x})$ converge. En esta situación necesitamos que $\mathbf{P}_{h(x)}$ devuelva siempre el mismo resultado (por ejemplo ε), pero para conseguir esto bastará con añadir otra instrucción X0:=ε; al final del código. Así, la **h** que buscamos será una función que transforma cada programa $\mathbf{P}_x$ en el programa:

$$\mathbf{h:}\ \mathbf{P}_x \longrightarrow \mathbf{P}_{h(x)} \quad \begin{array}{l} X1:= x; \\ P_x \\ X0:= \varepsilon; \end{array}$$

De esta manera, para aquellos programas que convergen sobre sí mismos el programa construido por **h** siempre devuelve ε. Y para aquellos que divergen sobre sí mismos el programa que obtenemos mediante **h** diverge para cualquier entrada.

Ahora que tenemos claro su resultado hemos de demostrar la computabilidad de la función **h**, para lo cuál utilizaremos técnicas similares a las del apartado **4.3.3**. Sin pérdida de generalidad supondremos que el alfabeto utilizado es $\Sigma = \{\mathbf{a}, \mathbf{b}\}$.

```
X0:= haz_asig_vacía(1);
AUX:= X1^R;
  – Aquí el dato x es usado como texto y desmenuzado carácter a carácter
  – para que dicho texto sea cargado en la variable X1 del programa construido
while nonem?(AUX) loop
        if car_a?(AUX) then
            X0 := haz_composición(X0, haz_asig_cons_a(1, 1));
        else
            X0 := haz_composición(X0, haz_asig_cons_b(1, 1));
        end if;
        AUX := cdr(AUX);
end loop;
  – Aquí el dato x es usado como instrucción y añadido como tal al programa construido
X0 := haz_composición(X0, X1);
X0 := haz_composición(X0, haz_asig_vacía(0));
```

El programa resultante tiene un elemento de dificultad adicional. El argumento **x** de **h** es utilizado de dos maneras: por un lado como programa, siendo sus instrucciones añadidas al lugar apropiado de $\mathbf{P}_{h(x)}$; por otro como dato, ya que es preciso que la copia de $\mathbf{P}_x$ incluida tome como argumento precisamente el texto de $\mathbf{P}_x$.

Tenemos entonces una función computable y total **h** que verifica para cualquier **x**:

$$\mathbf{x}\in\mathbf{K} \Rightarrow \varphi_x(\mathbf{x})\downarrow \Rightarrow \forall \mathbf{y}\ \varphi_{h(x)}(\mathbf{y})=\varepsilon \Rightarrow \varphi_{h(x)} \text{ es constante} \Rightarrow \mathbf{h}(\mathbf{x})\in\mathbf{CONS}$$

$$\mathbf{x}\notin\mathbf{K} \Rightarrow \varphi_x(\mathbf{x})\uparrow \Rightarrow \forall \mathbf{y}\ \varphi_{h(x)}(\mathbf{y})\uparrow \Rightarrow \varphi_{h(x)} \text{ no es constante} \Rightarrow \mathbf{h}(\mathbf{x})\notin\mathbf{CONS}$$

Por tanto $\mathbf{K}\leq\mathbf{CONS}$ y como $\mathbf{K}\notin\Sigma_0$ deducimos que $\mathbf{CONS}\notin\Sigma_0$. ⌫

5.3.2. El caso de CONS (no semidecidibilidad)

Una inspección más atenta al conjunto **CONS** daría como resultado serias dudas no solo sobre su decidibilidad, sino incluso sobre su semidecidibilidad. Si el hecho de que $\mathbf{P}_x$ compute una función constante depende del resultado que produzca sobre todas y cada una de las entradas posibles, ¿qué prueba (computable) sobre él nos permitiría detectar que tal cosa sucede? Parece razonable ser un poco más ambiciosos y tratar de probar que **CONS** es aún más complicado de lo que se ha visto en la sección anterior. Para ello probaremos que podemos situar por debajo de él otro conjunto que ya sabemos que no es semidecidible.

⌦ En concreto probaremos $\mathbf{CONS}\notin\Sigma_1$ mediante la reducción $\mathbf{TOT}\leq\mathbf{CONS}$. Para ello necesitamos una función computable **h** que cumpla que, para cada **x**

- si $\mathbf{P}_x$ es un programa que converge siempre, entonces el programa $\mathbf{P}_{h(x)}$ también debe terminar siempre y además devolviendo en todos los casos el mismo resultado
- si existe algún dato **y** para el cual $\mathbf{P}_x$ diverge, entonces el programa $\mathbf{P}_{h(x)}$ debe esforzarse en no computar una función constante (como en la sección anterior)

De nuevo nos encontramos con la dificultad de que no es posible preguntar directamente por ninguna propiedad semántica del programa $\mathbf{P}_x$ (mucho menos aún por una tan compleja como la totalidad). Sólo podemos, en todo caso, simular su ejecución para alguna entrada concreta, así que haremos lo siguiente: $\mathbf{P}_{h(x)}$ actuará sobre cualquier entrada **y** exactamente de la misma forma que $\mathbf{P}_x$, y si esta actuación termina producirá un resultado fijo, por ejemplo ε. Así, la transformación realizada por **h** será como sigue:

$$\mathbf{h}: P_x \longrightarrow P_{h(x)} \quad \begin{array}{l} P_x \\ X0:=\varepsilon; \end{array}$$

De esta manera, si el programa $\mathbf{P}_x$ converge para todas las entradas entonces $\mathbf{P}_{h(x)}$ también, y además devolverá siempre ε. Sin embargo, bastará con que $\mathbf{P}_x$ diverja para un solo dato para que $\mathbf{P}_{h(x)}$ también lo haga y su comportamiento no pueda ser asociado al de una función constante.

Es claro que **h** es una función total, veamos que además es computable mediante el siguiente programa:

```
X0:= X1;
X0 := haz_composición(X0, haz_asig_vacía(0));
```

Además **h** es la función de reducibilidad buscada:

$$\mathbf{x} \in \mathbf{TOT} \Rightarrow \forall \mathbf{y}\ \varphi_x(\mathbf{y})\downarrow \Rightarrow \forall \mathbf{y}\ \varphi_{h(x)}(\mathbf{y})=\varepsilon \Rightarrow \varphi_{h(x)} \text{ es constante} \Rightarrow \mathbf{h}(\mathbf{x}) \in \mathbf{CONS}$$

$$\mathbf{x} \notin \mathbf{TOT} \Rightarrow \exists \mathbf{y}\ \varphi_x(\mathbf{y})\uparrow \Rightarrow \exists \mathbf{y}\ \varphi_{h(x)}(\mathbf{y})\uparrow \Rightarrow \varphi_{h(x)} \text{ no es constante} \Rightarrow \mathbf{h}(\mathbf{x}) \notin \mathbf{CONS}$$

Por tanto $\mathbf{TOT} \leq \mathbf{CONS}$ y como $\mathbf{TOT} \notin \Sigma_1$ deducimos que $\mathbf{CONS} \notin \Sigma_1$. ⌫

5.4. Jerarquía e incrementalidad en las pruebas de reducción

El uso de la técnica de reducción mediante funciones de reducibilidad confirma lo que habíamos indicado en la introducción. Por un lado se supera claramente el rendimiento de la técnica de diagonalización, al menos en tres aspectos:

- La reducción es efectiva en muchos casos en los que la diagonalización no era aplicable.
- La reducción es menos costosa y compleja de aplicar en los casos en que ambas técnicas pueden ser empleadas.
- La diagonalización incrementa su dificultad cuando se quiere probar la incomputabilidad de funciones no totales. Esto no sucede en el caso de la reducción, donde puede ser más sencilla la prueba de no semidecidibilidad de un conjunto que la de indecidibilidad (véase el ejemplo de **CONS** en la sección anterior).

En realidad la dificultad de una prueba de reducción depende grandemente del conjunto elegido para comparar. En este sentido se confirma también la idea de incrementalidad en este tipo de demostraciones. A medida que vamos incorporando conjuntos a nuestro catálogo de indecidibles y no semidecidibles, va aumentando nuestra capacidad de encontrar otros nuevos, puesto que se convierten en herramientas de comparación disponibles.

Finalmente es muy destacable la idea de jerarquía introducida por el concepto de reducción. No se trata únicamente de establecer una barrera entre conjuntos decidibles e indecidibles, ya que la reducibilidad muestra que hay unos "más indecidibles" que otros. Sin embargo no debemos dejarnos llevar por el entusiasmo, ya que no siempre se puede establecer una comparación directa entre dos conjuntos. Esto lo vamos a ver en la Propiedad 5, pero previamente probaremos otro resultado:

PROPIEDAD 4 (REDUCIBILIDAD Y COMPLEMENTARIEDAD): Sean $\mathbf{A},\mathbf{B} \subseteq \sum^*$ cualesquiera. Se verifica que $\mathbf{A} \leq \mathbf{B} \Leftrightarrow \overline{\mathbf{A}} \leq \overline{\mathbf{B}}$.

⌦ La prueba es inmediata. Si $\mathbf{A} \leq \mathbf{B}$ existe **h** computable y total que cumple:

$$\forall \mathbf{x} \in \sum^* (\mathbf{x} \in \mathbf{A} \leftrightarrow \mathbf{h}(\mathbf{x}) \in \mathbf{B})$$

pero esta misma función sirve para demostrar la reducibilidad de $\overline{\mathbf{A}}$ a $\overline{\mathbf{B}}$:

$$\mathbf{x} \in \overline{\mathbf{A}} \Leftrightarrow \mathbf{x} \notin \mathbf{A} \Leftrightarrow \mathbf{h}(\mathbf{x}) \notin \mathbf{B} \Leftrightarrow \mathbf{h}(\mathbf{x}) \in \overline{\mathbf{B}}$$

de donde se deduce que $\overline{\mathbf{A}} \leq \overline{\mathbf{B}}$. ⌫

PROPIEDAD 5 (LA REDUCIBILIDAD NO ES UN ORDEN TOTAL): Existen conjuntos $\mathbf{A},\mathbf{B} \subseteq \sum^*$ para los que no se verifica ni $\mathbf{A} \leq \mathbf{B}$ ni $\mathbf{A} \leq \mathbf{B}$.

⌦ Basta con encontrar una pareja de tales conjuntos. Utilizaremos **K** y $\overline{\mathbf{K}}$, y aunque en general demostrar que no se da una cierta reducibilidad puede ser una tarea muy respetable, en este caso se deduce de manera sorprendentemente sencilla.

Sabemos que $\mathbf{K} \in \Sigma_1$ pero $\overline{\mathbf{K}} \notin \Sigma_1$. Por tanto no puede darse $\overline{\mathbf{K}} \leq \mathbf{K}$, porque ello implicaría $\overline{\mathbf{K}} \in \Sigma_1$. Pero por la propiedad 4 tampoco puede darse $\mathbf{K} \leq \overline{\mathbf{K}}$, ya que esto implicaría automáticamente $\overline{\mathbf{K}} \leq \mathbf{K}$, así que queda demostrado que no puede darse ninguna de las dos posibles reducibilidades. ⌫

DEFINICIÓN: Como tal orden parcial la reducibilidad induce una relación de equivalencia entre conjuntos: cuando al mismo tiempo se verifican $\mathbf{A} \leq \mathbf{B}$ y $\mathbf{B} \leq \mathbf{A}$ decimos

que **A** y **B** son ***equivalentes*** y lo notamos como $\mathbf{A} \equiv \mathbf{B}$. Los conjuntos equivalentes entre sí comparten exactamente las mismas propiedades de incomputabilidad. Por ejemplo se dan las dos siguientes propiedades:

PROPIEDAD 6 (EQUIVALENCIA DE LOS CONJUNTOS DECIDIBLES): Si **A** y **B** son conjuntos decidibles no triviales, entonces son equivalentes. Formalmente:

$$\left.\begin{array}{l} \mathbf{A},\mathbf{B} \subseteq \Sigma^* \\ \mathbf{A},\mathbf{B} \in \Sigma_0 \\ \mathbf{A},\mathbf{B} \neq \emptyset \\ \mathbf{A},\mathbf{B} \neq \Sigma^* \end{array}\right\} \Rightarrow \mathbf{A} \equiv \mathbf{B}$$

⌦ Probaremos que $\mathbf{A} \leq \mathbf{B}$. Como $\mathbf{A} \in \Sigma_0$, la función característica $\mathbf{C_A}$ es computable. Como $\mathbf{B} \neq \emptyset$ y $\mathbf{B} \neq \Sigma^*$ existen al menos un elemento $\mathbf{m} \in \mathbf{B}$ y otro $\mathbf{n} \notin \mathbf{B}$. La siguiente función **h**:

$$\mathbf{h}(\mathbf{x}) = \begin{cases} \mathbf{m} & \mathbf{x} \in \mathbf{A} \\ \mathbf{n} & \mathbf{x} \notin \mathbf{A} \end{cases}$$

es obviamente computada por el programa:

if C_A(X1) **then** X0:= m; **else** X0:= n; **end if**;

y como además se verifica:

$$\mathbf{x} \in \mathbf{A} \Rightarrow \mathbf{h}(\mathbf{x}) = \mathbf{m} \in \mathbf{B}$$

$$\mathbf{x} \notin \mathbf{A} \Rightarrow \mathbf{h}(\mathbf{x}) = \mathbf{n} \notin \mathbf{B}$$

tenemos que $\mathbf{A} \leq \mathbf{B}$ vía **h**. Por simetría se sigue que $\mathbf{B} \leq \mathbf{A}$, por lo que $\mathbf{A} \equiv \mathbf{B}$. ⌫

De modo análogo podemos probar que los conjuntos decidibles constituyen el primer escalón en la escala de la incomputabilidad.

PROPIEDAD 7 (LA DECIDIBILIDAD COMO COTA INFERIOR): Si **A** es un conjunto decidible no trivial, y **B** es otro conjunto cualquiera asimismo no trivial, entonces $\mathbf{A} \leq \mathbf{B}$. Formalmente:

$$\left.\begin{array}{l} \mathbf{A},\mathbf{B} \subseteq \Sigma^* \\ \mathbf{A} \in \Sigma_0 \\ \mathbf{A},\mathbf{B} \neq \emptyset \\ \mathbf{A},\mathbf{B} \neq \Sigma^* \end{array}\right\} \Rightarrow \mathbf{A} \leq \mathbf{B}$$

⌦ Se puede seguir punto por punto la demostración de la propiedad 3 (nótese que no se hizo uso de la hipótesis de reducibilidad de **B**). ⌫

Naturalmente, la exclusión de los conjuntos triviales ∅ y ∑* de las anteriores propiedades no obedece a que sean conjuntos decidibles más o menos complicados que los demás, sino a la imposibilidad de realizar con ellos reducciones normales (∅ no tiene elementos a los que **h** pueda enviar los de otro conjunto **A**, ni ∑* deja sitio donde enviar los de $\overline{\mathbf{A}}$).

El panorama se complica bastante por encima de $\mathbf{\Sigma}_0$. Si tomamos los conjuntos indecidibles pero semidecidibles de la clase $\mathbf{\Sigma}_1 - \mathbf{\Sigma}_0$ nos encontramos que, lejos de ser equivalentes entre sí, se subdividen en un gran número (de hecho infinito) de pequeñas clases de equivalencia. Y la situación es aún más extraña por encima de la semidecidibilidad, con clases de incomputabilidad creciente y no acotada. De todas formas la identificación y estudio de esas clases queda muy por encima de los objetivos del presente texto. Baste decir que esta relación jerárquica induce un retículo que sirve de soporte a una rica teoría que permite medir de alguna manera lo lejano que se encuentra un problema de poder ser resuelto mediante un proceso algorítmico.

En todo caso es justo reconocer que, si bien la reducibilidad introduce un interesante orden teórico en el esquema de la reducción, desde el punto de vista técnico no hemos avanzado tanto, ya que no hemos conseguido erosionar la dificultad de la parte más dura de las demostraciones: la de construir programas que transforman otros programas, en los que se cruzan los datos e instrucciones del constructor con los del construido. Y eso teniendo en cuenta que las transformaciones que hemos utilizado en los ejemplos vistos son relativamente sencillas, como añadir instrucciones al principio y al final o cargar valores de constantes.

Afortunadamente, en el siguiente capítulo estudiaremos un método que supondrá un salto cualitativo importantísimo al automatizar gran parte de dicho proceso constructivo.

6. Teorema s-m-n

El siguiente paso que abordaremos en nuestro esfuerzo por simplificar la aplicación de la técnica de reducción será el más significativo de todos y justificará con creces nuestros esfuerzos precedentes. Veremos que en un gran número de ocasiones la función de reducción **h** puede ser diseñada de forma que la prueba de su computabilidad resulte trivial. Más específicamente, veremos que si **h** es especificada con cierto cuidado su computabilidad estará garantizada y la prueba de reducción asociada se simplificará enormemente. Naturalmente, para que esto sea posible necesitaremos un resultado previo que lo garantice de manera genérica.

El terreno en el que nos moveremos mantendrá una de las características peculiares de la técnica de reducción: demostrar la ***computabilidad*** de ciertas funciones será la forma más eficaz de probar la ***incomputabilidad*** de otras. Sólo que en este caso la naturaleza de las primeras hace que sea realmente complicado construir el programa que las computa, por lo que necesitaremos una ayudita que es, en sí mismo, uno de los pilares de la Teoría de la Computabilidad: el Teorema s-m-n o Teorema de Parametrización.

6.1. Parametrizaciones

Ya nos hemos familiarizado con la idea de que cuando dos funciones están relacionadas de cierta manera ello tiene influencia en sus propiedades de computabilidad. Por ejemplo, si **k** es la composición de las funciones computables **f** y **g**, entonces también **k** es computable. Un caso muy sencillo de este mismo principio lo tenemos cuando una función es la ***particularización*** de otra. A partir de una función computable como el producto $\mathbf{f}(\mathbf{x},\mathbf{y}) = \mathbf{x}*\mathbf{y}$ podemos definir el siguiente caso particular: la función de un argumento que calcula el doble de la entrada $\mathbf{g}(\mathbf{y}) = \mathbf{f}(2,\mathbf{y}) = 2*\mathbf{y}$. La computabilidad de **g** resulta evidente a partir de la de **f** pues queda demostrada con el programa `X0:= f(2, X1);`. De este modo, tomando diferentes constantes como valor fijo del argumento **x** podemos obtener infinitas funciones computables, todas ellas casos particulares de **f**.

El interés de las particularizaciones de funciones computables no está en su computabilidad, ya que no resulta especialmente útil pasar de funciones más complejas a más sencillas. La enorme importancia que tienen se fundamenta en la posibilidad de

encontrar un método para construir cada una de ellas de manera sistemática a partir del programa de la función más general. Por tanto no nos interesa la computabilidad de las particularizaciones, sino la del mecanismo de generarlas.

Es bastante intuitivo y podemos verlo con el primer ejemplo que hemos planteado. Supongamos que queremos obtener, a partir del programa que computa el producto, los de las distintas particularizaciones que consisten en hacer productos por constantes específicas (duplicar, triplicar, etc.). Si disponemos de un programa $\mathbf{P}_w$ para $\mathbf{f}(\mathbf{x},\mathbf{y}) = \mathbf{x}*\mathbf{y}$ y queremos obtener un programa $\mathbf{P}_z$ para $\mathbf{g}(\mathbf{y}) = 2*\mathbf{y}$ basta con modificar el primero, preparando cuidadosamente el contenido de sus variables de entrada. Más concretamente pasaremos el contenido de la única variable de entrada de $\mathbf{P}_z$ a una segunda variable, y una vez libre la primera variable le asignamos la constante 2. Por tanto $\mathbf{P}_z$ es el programa equivalente al siguiente macroprograma:

```
X2:= X1;
X1:= 2;
Pw
```

Con cambios similares podríamos obtener programas para cualquier particularización de **f** que se nos ocurra. Pero aún podemos ir más lejos: en lugar de construir a mano los programas que computan $2*\mathbf{y}$, $3*\mathbf{y}$, $4*\mathbf{y}$, etc, podríamos aspirar a que esta tarea fuera realizada a su vez por un programa, dado que el mecanismo de construcción es siempre el mismo, una pequeña adición de instrucciones al principio de $\mathbf{P}_w$. Este programa tomaría como entrada un ***parámetro*** **t**=2, 3, 4, ... y obtendría como salida un programa que computaría el producto particular $\mathbf{t}*\mathbf{y}$.

La realización de esta tarea de manera sistemática nos permitiría hablar de una función total y computable **h** que a partir de un parámetro nos proporciona un programa cuyo comportamiento depende de ese parámetro. Es decir, **h**(2) nos proporcionaría el programa descrito un poco más arriba (que calcula el doble de un número) y **h**(4) produciría otro similar cambiando la aparición de la constante 2 por la constante 4 (y por tanto calculará el cuádruple de un número). Esta función **h** es una constructora de programas, variantes de $\mathbf{P}_w$ mediante reubicación y reasignación del valor de sus variables de entrada. No vamos a demostrar la computabilidad de esta **h** porque los resultados de la siguiente sección lo harán superfluo. Lo que nos interesa es subrayar que el uso de las funciones constructoras de programas nos permitiría de hecho diseñar el algoritmo que se encarga de esta construcción. Lo que es más importante, esto podríamos hacerlo

independientemente de la función computable de partida **f**. De hecho puede tratarse de una función genérica ψ, ya que para nuestro argumento es irrelevante que sea o no total.

Antes de centrarnos en la utilización práctica de este principio para demostrar la indecidibilidad de algunos conjuntos nos vamos a detener en algunos ejemplos. Consideremos la siguiente función no total:

$$\psi(\mathbf{x},\mathbf{y}) \cong \begin{cases} \dfrac{\mathbf{x} * \mathbf{y}}{2} & \mathbf{x} \bmod 2 = 0 \\ \perp & \text{c.c} \end{cases}$$

Si procedemos a estudiar las particularizaciones que resultan de fijar valores para el primer argumento veremos que se obtienen resultados no tan homogéneos como en el caso del producto:

- para **x**= 2 obtendríamos la función $\xi(\mathbf{y}) \cong \psi(2,\mathbf{y}) \cong \mathbf{y}$
- para **x**= 4 obtendríamos la función $\chi(\mathbf{y}) \cong \psi(4,\mathbf{y}) \cong 2*\mathbf{y}$
- para **x**= 5 obtendríamos la función $\theta(\mathbf{y}) \cong \psi(5,\mathbf{y}) \cong \perp$

Al igual que antes, no nos impresiona la computabilidad de estas funciones (que pueden ser totales o no, como se ve), sino la de otra función **h** que produzca sistemáticamente los programas que las computan, de forma que **h**(2) sea un programa para la función identidad $\xi(\mathbf{y})$, y **h**(5) otro para la función vacía $\theta(\mathbf{y})$. O dicho de otro modo, tal que:

$$\varphi_{h(2)}(\mathbf{y}) \cong \xi(\mathbf{y}) \cong \psi(2,\mathbf{y}) \cong \mathbf{y}$$

$$\varphi_{h(4)}(\mathbf{y}) \cong \chi(\mathbf{y}) \cong \psi(4,\mathbf{y}) \cong 2*\mathbf{y}$$

$$\varphi_{h(5)}(\mathbf{y}) \cong \theta(\mathbf{y}) \cong \psi(5,\mathbf{y}) \cong \perp$$

A diferencia de las funciones ξ, χ o θ, esta función **h** ***sí será siempre total***, ya que sea cual sea el valor que le suministremos nos dará como resultado el código de un programa.

Este esquema se revela aún más poderoso cuando el parámetro que utilizamos no es un simple número, sino un programa cuyas propiedades son relevantes para la definición de la función de partida ψ. Consideremos el siguiente ejemplo:

$$\psi(\mathbf{x},\mathbf{y}) \cong \begin{cases} \dfrac{2}{\mathbf{y}} & \mathbf{x} \in K \wedge \mathbf{y} \neq 0 \\ 0 & \mathbf{x} \in K \wedge \mathbf{y} = 0 \\ \perp & \text{c.c.} \end{cases}$$

En este caso los argumentos de ψ son el programa $\mathbf{P}_x$ y el número **y**, y su comportamiento depende de la facultad que tenga $\mathbf{P}_x$ de converger sobre su propio código y del hecho de que **y** sea o no nulo. Si tomamos como parámetro el primer argumento observamos que solo se pueden producir dos casos particulares, una función total y otra que no lo es:

- si $\mathbf{x} \in \mathbf{K}$ (independientemente de su valor concreto) obtenemos la función

$$\xi(\mathbf{y}) \cong \varphi_{h(x)}(\mathbf{y}) \cong \begin{cases} \dfrac{2}{\mathbf{y}} & \mathbf{y} \neq 0 \\ 0 & \mathbf{y} = 0 \end{cases}$$

- si $\mathbf{x} \notin \mathbf{K}$ (independientemente de su valor concreto) resulta $\theta(\mathbf{y}) \cong \varphi_{h(x)}(\mathbf{y}) \cong \perp$

Dada la naturaleza de **x** nos encontramos que, en este caso, la función **h** resulta ser una ***transformadora de programas*** que toma como argumento un programa $\mathbf{P}_x$ y, según su comportamiento sobre su propio código, lo transforma en una de las dos posibilidades que hemos visto. Es esta faceta de transformación la que nos resultará realmente práctica para obtener funciones de reducción que se puedan utilizar para demostrar la incomputabilidad de algunos problemas. Pero para ello será esencial que estas funciones **h** que hemos descrito en este apartado sean computables, y la verificación de este hecho es la tarea que vamos a acometer en la siguiente sección.

6.2. El teorema s-m-n

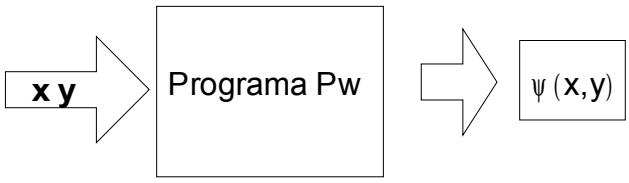
La generalización que estamos planteando es la que se recoge en el siguiente teorema, llamado ***de parametrización*** o ***teorema s-m-n***, que dice que para cualquier función computable, existe otra función computable y total que nos proporciona índices de funciones particulares de la primera.

TEOREMA S-M-N: Para toda función computable $\psi^2: \sum^* \times \sum^* \dashrightarrow \sum^*$ existe una función **h**: $\sum^* \rightarrow \sum^*$ total y computable que verifica:

$$\forall \mathbf{x} \forall \mathbf{y}\ \psi(\mathbf{x},\mathbf{y}) \cong \varphi_{h(x)}(\mathbf{y})$$

DEMOSTRACIÓN:

⌦ Partimos de una función de dos argumentos ψ computable. Sea $\mathbf{P_w}$ el programa que la computa.



Tenemos que demostrar la computabilidad de una cierta función **h** total que toma como argumento un valor cualquiera **x** y devuelve un programa **h**(**x**) que computa una función íntimamente relacionada con ψ. Por tanto **h** es una constructora de programas que a partir de cualquier valor **x** obtiene otro programa que computa una versión particular de ψ, precisamente la que resulta de fijar ese valor concreto de **x** como primer argumento de ψ.

Para conseguirlo hemos de construir un programa **Q** que compute **h**. **Q** tomará como dato **x** (que quedará cargado en ***su*** variable de entrada **X1**), y deberá construir un programa **P** y situarlo en ***su*** variable de salida **X0**.

Para entender cómo debe ser la estructura de este programa **P** (que debe ser escrito por **Q**) hay que considerar que su dato de entrada será **y** (cargado inicialmente en ***su propia*** variable de entrada **X1**) y su resultado será $\psi(\mathbf{x},\mathbf{y})$, que ha de devolver en su propia variable de salida **X0**.

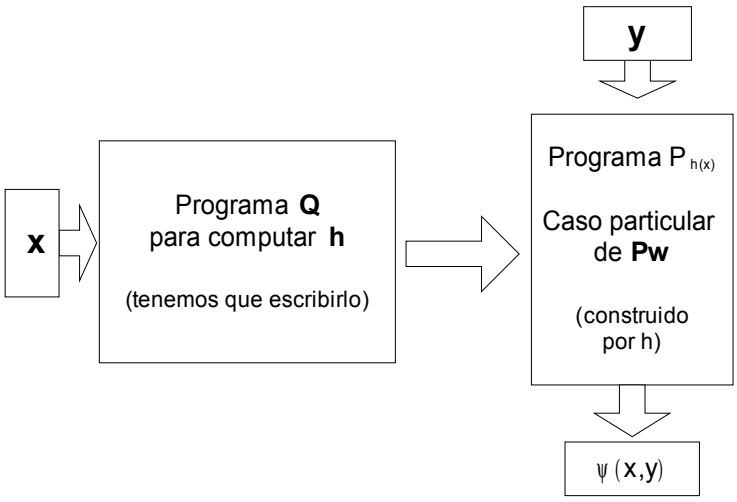


Fig. 6.1: Esquema de los programas que intervienen en el Teorema s-m-n, sus datos y resultados

Es importante no confundir los distintos programas que intervienen en esta demostración, que actúan en momentos y niveles muy diferentes, y sobre todo tampoco sus respectivas variables. Esta distinción se complica, ya que por convención las variables de todos los programas están condenadas a denominarse de igual manera. El diagrama de la figura 6.1 describe esta situación.

El programa **Q** que computa la función **h** es el que debemos construir nosotros para demostrar el teorema. Tiene una única variable de entrada **X1** que recibe el parámetro **x**, y produce como resultado el programa $\mathbf{P}_{h(x)}$, al que para abreviar hemos llamado **P**. Podemos utilizar en la construcción de **Q** todas las abreviaturas usuales en la definición de macroprogramas.

El programa **P** es construido mediante la ejecución de **Q**, pero no es ejecutado en ningún momento. Sin embargo su comportamiento es muy importante, ya que debe computar una versión particular de ψ. Tiene una variable **X1** que recibe el dato de entrada **y** y su resultado es $\psi(\mathbf{x},\mathbf{y})$, que debe ser colocado en su variable **X0**. Nótese que el comportamiento de **P** depende necesariamente del valor de **x**, pero este no es una de sus entradas. Esto se debe a que dicho valor ya ha quedado incorporado al código de **P** en forma de constante cuando este es construido.

Empecemos imaginando cómo será ese programa **P**, que luego ya nos encargaremos de que sea construido por **Q**. El resultado de **P** sobre **y** ha de ser el mismo que $\psi(\mathbf{x},\mathbf{y})$, pero para un valor **x** particular que conocemos. El programa $\mathbf{P}_w$ que calcula $\psi(\mathbf{x},\mathbf{y})$ también es conocido, así que lo más directo será usar dicho programa como núcleo de **P**. Para que **P** opere sobre los datos que nos interesa es necesario situar los datos **x** (que conocemos al construir **P**) e **y** (que se conocerá sólo al ejecutar **P**) como valores de las variables **X1** y **X2**. Entonces el dato de entrada **y** que recibe **P** en **X1** debe trasladarse a **X2**, para poder asignar a **X1** el valor **x**. Es importante hacerlo en este orden para no perder los datos. Después basta con ejecutar $\mathbf{P}_w$. El programa **P** buscado será:

```
X2:= X1;
X1:= x;
Pw
```

Si nuestra misión fuera demostrar la existencia de **P** ya habríamos terminado. Pero ello sólo demostraría que las particularizaciones de ψ son computables, cosa que ya sabíamos. El reto al que nos enfrentamos es escribir un programa **Q** que sea quien construya **P** (es decir $\mathbf{P}_{h(x)}$) a partir del valor de **x**. Y no disponemos de instrucciones para manipular macroprogramas como el escrito más arriba porque sólo se puede operar con

los tipos de datos ya implementados, entre los cuáles no están (ni tiene lógica que estén) los macroprogramas. Esto nos obliga a especificar **P** como un estricto programa-while.

Para conocer la expansión correspondiente al programa anterior tenemos que hacer dos cosas. La primera es sencilla porque sólo hay que reasignar el contenido de la entrada a otra variable. La segunda exige que introduzcamos el valor de **x** en la variable **X1**, para lo que tenemos que referirnos a sus símbolos constituyentes para añadirlos uno a uno. Si $\mathbf{x} = s_{k1}s_{k2}\ldots s_{kn}$:

```
 X0:= ε;
 X2:= cons_s1(X1);  X2:= cdr(X2);
-- Reubicación de y en X2
 X1:= ε;
 X1:= cons_skn(X1);
 ...
 X1:= cons_sk2(X1);
 X1:= cons_sk1(X1);
-- Copiado de x en X1
 P_w
```

Una vez aclarado qué programa queremos construir es fácil demostrar la computabilidad de **h**. Solo hay que tener en cuenta que su dato de entrada **x** es el que el programa **P** utiliza como constante.

```
 X0 := haz_asig_vacía (0);
 X0 := haz_composición (X0,
                haz_composición (haz_asig_cons_s1 (2, 1), haz_asig_cdr (2, 2)));
 X0 := haz_composición (X0, haz_asig_vacía (1));
-- Hay que invertir el dato x para reconstruirlo correctamente
 AUX := X1^R;
 while nonem?(AUX) loop
          Z := primero(AUX);
          case Z is
-- Cada símbolo de x que leemos será añadido a X1
             when s1 => X0 := haz_composición (X0, haz_asig_cons_s1 (1,1));
             ...
             when sk => X0 := haz_composición (X0, haz_asig_cons_sk (1,1));
          end case;
          AUX:= cdr(AUX);
 end loop;
 X0:= haz_composición (X0, w);
```

El anterior programa sería el **Q** buscado para computar la función total **h** cuya computabilidad queríamos probar. De la construcción de **Q** se concluye la demostración del teorema ⌫

Extensión: El teorema s-m-n se cumple para funciones de cualquier número de argumentos mayor o igual que 2, pudiendo elegirse un número arbitrario de parámetros. Su enunciado más general es:

Para toda función computable $\psi^{m+n}: \sum^{*m} \times \sum^{*n} \dashrightarrow \sum^{*}$ existe una función **h**: $\sum^{*m} \to \sum^{*}$ total y computable que verifica

$$\psi^{m+n}(x_1,\ldots,x_m,y_1,\ldots,y_n) \cong \varphi_{h(x1,\ldots,xm)}(y_1,\ldots,y_n)$$

De hecho el nombre s-m-n le viene al teorema de esta formulación general, pues se solía nombrar la función **h** con la letra **s**.

6.3. Ejemplos de aplicación del teorema s-m-n

La mejor forma de entender el alcance del teorema s-m-n es aplicarlo. En los ejemplos que se detallan en las siguientes secciones se puede apreciar hasta qué punto simplifica este resultado las demostraciones por reducción. Veremos así que, a la mejora que supone el método sobre la diagonalización, se añade la facilidad de no tener que demostrar la computabilidad de la función de reducción **h**.

6.3.1. PERM no es decidible

El conjunto **PERM** incluye todos los programas que computan funciones biyectivas, es decir, totales, inyectivas y sobreyectivas. $\mathbf{PERM} = \{ x \in \mathbb{W}: W_x=\sum^* \wedge R_x=\sum^* \wedge \forall y \forall z\ \varphi_x(y)=\varphi_x(z) \to y=z \}$ Demostraremos que no es decidible probando que $\mathbf{K} \leq \mathbf{PERM}$. Como sabemos que **K** no es decidible, **PERM** tampoco lo podrá ser.

Necesitamos encontrar una función computable y total **h**: $\sum^* \to \sum^*$ que verifique:

$$\forall x\ (x \in K \leftrightarrow h(x) \in PERM)$$

o, lo que es lo mismo:

$$\forall x\ (\varphi_x(x)\downarrow \leftrightarrow \varphi_{h(x)} \text{ es biyectiva})$$

Por tanto, la función **h** que necesitamos deberá tomar un programa **x** y transformarlo en otro programa **h**(**x**), relacionando el comportamiento del segundo con el del primero:

- cuando el programa original **x** converge sobre sí mismo, el programa transformado **h**(**x**) converge siempre, nunca devuelve el mismo valor para datos diferentes, y devuelve todos los valores posibles
- cuando el programa original **x** diverge sobre sí mismo, el programa transformado **h**(**x**) o bien diverge para algún dato, o bien existen dos entradas para las que devuelve el mismo valor, o bien algún valor no aparece entre sus posibles salidas

Esto nos sugiere que el teorema s-m-n puede ser aplicable para justificar la computabilidad de **h** sin necesidad de escribir el programa que la computa. Para ello buscaremos una función $\psi(\mathbf{x},\mathbf{y})$ sobre la que aplicar el teorema, a resultas de lo cuál obtendremos la h buscada de la forma:

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

Cuando $\mathbf{x} \in \mathbf{K}$ queremos que $\varphi_{h(x)}$ sea una biyección, por ejemplo la identidad $\varphi_{h(x)}(\mathbf{y}) = \mathbf{y}$. Esto nos sugiere acometer la definición de ψ de la siguiente forma:

$$\psi(\mathbf{x},\mathbf{y}) \cong \varphi_{h(x)}(\mathbf{y}) \cong \begin{cases} \mathbf{y} & \varphi_x(x)\downarrow \\ ?? & \text{c.c.} \end{cases}$$

Pero si ψ ha de ser computable y preguntamos si $\varphi_x(\mathbf{x})\downarrow$, la única forma de hacerlo será simulando la ejecución de $\mathbf{P}_x(\mathbf{x})$, y cuando no haya convergencia no tendremos una respuesta explícita, sólo que la ejecución ciclará. Por tanto no tenemos libertad para devolver lo que queramos en caso de que $\varphi_x(\mathbf{x})\uparrow$. Afortunadamente tampoco la necesitamos, ya que si $\varphi_{h(x)}$ diverge siempre que $\mathbf{x} \notin \mathbf{K}$, tendremos para ese caso que la función $\varphi_{h(x)}$ no es una biyección. Así pues la función

$$\psi(\mathbf{x},\mathbf{y}) \cong \begin{cases} \mathbf{y} & \varphi_x(\mathbf{x})\downarrow \\ \perp & \mathbf{c.c.} \end{cases}$$

es computada por el siguiente programa, que usa la función universal Φ (ver apéndice):

```
R:= Φ(X1, X1);   X0:= X2;
```

Dado que ψ es computable, el teorema s-m-n nos asegura que existe una función $\mathbf{h}: \sum^* \rightarrow \sum^*$ computable y total que cumple $\forall \mathbf{x} \forall \mathbf{y}\ \psi(\mathbf{x},\mathbf{y}) \cong \varphi_{h(x)}(\mathbf{y})$. Comprobaremos que esta función es justo la transformadora de programas que deseamos, y por tanto permite reducir **K** a **PERM**:

- Cuando $\mathbf{x} \in \mathbf{K}$, $\varphi_{h(x)}$ resulta ser la función identidad, que sí es biyectiva.
- Cuando $\mathbf{x} \notin \mathbf{K}$, $\varphi_{h(x)}$ resulta ser la función vacía, que no es biyectiva.

Si lo describimos formalmente, para una **x** cualquiera:

$$\mathbf{x} \in \mathbf{K} \Rightarrow \varphi_x(\mathbf{x})\downarrow \Rightarrow \forall \mathbf{y}\ \psi(\mathbf{x},\mathbf{y})=\mathbf{y} \Rightarrow \forall \mathbf{y}\ \varphi_{h(x)}(\mathbf{y}) = \mathbf{y} \Rightarrow \varphi_{h(x)}=\mathbf{id} \Rightarrow \mathbf{h}(\mathbf{x}) \in \mathbf{PERM}$$

$$\mathbf{x} \notin \mathbf{K} \Rightarrow \varphi_x(\mathbf{x})\uparrow \Rightarrow \forall \mathbf{y}\ \psi(\mathbf{x},\mathbf{y})\uparrow \Rightarrow \forall \mathbf{y}\ \varphi_{h(x)}(\mathbf{y})\uparrow \Rightarrow \varphi_{h(x)} \cong \perp\!\!\!\perp \Rightarrow \mathbf{h}(\mathbf{x}) \notin \mathbf{PERM}$$

Por tanto, hemos demostrado que $\mathbf{K} \leq \mathbf{PERM}$ por medio de **h**, y como $\mathbf{K} \notin \Sigma_0$, podemos concluir que $\mathbf{PERM} \notin \Sigma_0$.

Lo más destacable de este procedimiento es que, en vez de escribir el programa que computa a **h** y de ir construyendo las instrucciones de $\mathbf{P}_{h(x)}$ para que tenga el comportamiento deseado, ha sido suficiente con hacer un programa (el de ψ) que ***se comporta como*** las $\varphi_{h(x)}$. En nuestro caso hemos programado la identidad en lugar de programar la escritura de un programa que compute la identidad.

6.3.2. LIM no es semidecidible

Vamos a ver un ejemplo algo más complejo. Estudiaremos el caso del conjunto **LIM**, formado por los programas que computan funciones totales y acotadas, es decir cuyos valores del rango están limitados por una constante. $\mathbf{LIM} = \{ \mathbf{x} \in \mathbb{W}: \mathbf{W}_x = \sum^* \wedge \exists \mathbf{c}\ (\forall \mathbf{y}\ \varphi_x(\mathbf{y}) \leq \mathbf{c}) \}$. Demostraremos que no es semidecidible probando $\overline{\mathbf{K}} \leq \mathbf{LIM}$. Para conseguirlo necesitamos encontrar una función computable y total $\mathbf{h}: \sum^* \rightarrow \sum^*$ que verifique:

$$\forall \mathbf{x}\ (\mathbf{x} \in \overline{\mathbf{K}} \leftrightarrow \mathbf{h}(\mathbf{x}) \in \mathbf{LIM})$$

o, lo que es lo mismo:

$$\forall \mathbf{x}\ (\varphi_x(\mathbf{x})\uparrow \leftrightarrow \varphi_{h(x)} \text{ es una función total y acotada})$$

La relación que debe existir entre el programa original **x** y el transformado **h**(**x**) será la siguiente:

- si $\mathbf{P}_x$ diverge sobre sí mismo, $\mathbf{P}_{h(x)}$ debe converger para cualquier entrada y sus resultados deben estar acotados por una constante
- si $\mathbf{P}_x$ converge sobre sí mismo, $\mathbf{P}_{h(x)}$ debe o no ser total o no ser acotada (sus resultados pueden ser arbitrariamente grandes)

El hecho de que **h** sea una función transformadora de programas relacionados por su comportamiento nos sugiere que el teorema s-m-n puede ser aplicable para justificar su computabilidad sin necesidad de escribir el programa correspondiente. Ahora bien, en este caso no podemos utilizar la misma estrategia que en el precedente, de preguntar si $\mathbf{x} \in \mathbf{K}$, ya que entonces un esquema del tipo:

$$\psi(\mathbf{x},\mathbf{y}) \cong \varphi_{h(x)}(\mathbf{y}) \cong \begin{cases} \mathbf{y}^2 & \varphi_x(x)\downarrow \\ ?? & \text{c.c.} \end{cases}$$

nos funcionaría sólo a medias. En efecto, de este modo cuando $\mathbf{x} \in \mathbf{K}$ tenemos que $\varphi_{h(x)}(\mathbf{y}) = \mathbf{y}^2$ es una función no acotada (podríamos haber elegido cualquier otra, o simplemente una no total). Pero en caso contrario estamos condenados a mantener la función indefinida, y en esta ocasión no nos vale, ya que cuando $\mathbf{x} \in \overline{\mathbf{K}}$ precisamos que $\varphi_{h(x)}$ sea total y acotada.

Nos vendría muy bien cruzar los casos construyendo una función del tipo:

$$\psi(\mathbf{x},\mathbf{y}) \cong \varphi_{h(x)}(\mathbf{y}) \cong \begin{cases} 8 & \varphi_x(x)\uparrow \\ \perp & \text{c.c.} \end{cases}$$

pero al no ser $\overline{\mathbf{K}}$ semidecidible no resulta computable y no se le puede aplicar s-m-n.

Una estrategia más sutil se basa en lo siguiente: no podemos preguntar si $\mathbf{x} \notin \mathbf{K}$, pero recordemos que $\mathbf{x} \notin \mathbf{K} \leftrightarrow \forall \mathbf{y}\, \neg \mathbf{T}(\mathbf{x},\mathbf{x},\mathbf{y})$, es decir, si $\mathbf{P}_x(\mathbf{x})$ no converge para ningún número de pasos. En lugar de pretender definir $\varphi_{h(x)}$ de golpe, basándonos en una pregunta directa sobre la pertenencia de **x** a $\overline{\mathbf{K}}$, para cada valor de **y** preguntaremos si $\mathbf{T}(\mathbf{x},\mathbf{x},\mathbf{y})$ (cosa factible, ya que **T** es decidible). Si resulta falso sabremos que $\mathbf{P}_x(\mathbf{x})$ no converge el menos para ese número de pasos, por lo que ante la sospecha de que $\varphi_x(\mathbf{x})\uparrow$ actuaremos en consecuencia devolviendo un valor orientado a que $\varphi_{h(x)}$ sea total y acotada. Por el

contrario, si resulta cierto tendremos la certeza de que $\mathbf{x} \notin \overline{\mathbf{K}}$ y abortaremos la construcción de dicha función total y acotada dejándola indefinida.

Con ese objetivo definimos la función:

$$\psi(\mathbf{x}, \mathbf{y}) \cong \begin{cases} 8 & \neg\, \mathbf{T}(\mathbf{x}, \mathbf{x}, \mathbf{y}) \\ \perp & \text{c.c.} \end{cases}$$

computada por el programa:

```
if not T(X1, X1, X2) then
        X0:= 8;
else
        X0:= ⊥⊥ ;
end if;
```

Dado que ψ es computable, el teorema s-m-n nos asegura que existe una función $\mathbf{h}$: $\sum^* \rightarrow \sum^*$ computable y total que cumple $\forall \mathbf{x} \forall \mathbf{y}\ \psi(\mathbf{x},\mathbf{y}) \cong \varphi_{\mathbf{h}(\mathbf{x})}(\mathbf{y})$. Esta función es justo la transformadora de programas que deseamos, y por tanto permite reducir $\overline{\mathbf{K}}$ a **LIM**:

- Cuando $\mathbf{x} \in \overline{\mathbf{K}}$, el predicado $\mathbf{T}(\mathbf{x},\mathbf{x},\mathbf{y})$ no se verifica para ningún número de pasos, con lo que la función $\varphi_{\mathbf{h}(\mathbf{x})}$ resulta ser la constante 8, que es total y acotada.
- Cuando $\mathbf{x} \notin \overline{\mathbf{K}}$, a partir de un cierto número de pasos se verifica el predicado $\mathbf{T}(\mathbf{x},\mathbf{x},\mathbf{y})$. Para ese valor $\varphi_{\mathbf{h}(\mathbf{x})}(\mathbf{y})\uparrow$ y $\varphi_{\mathbf{h}(\mathbf{x})}$ no es una función total.

Si lo describimos formalmente, sea un $\mathbf{x}$ cualquiera:

$$\mathbf{x} \in \overline{\mathbf{K}} \Rightarrow \varphi_{\mathbf{x}}(\mathbf{x})\uparrow \Rightarrow \forall \mathbf{y}\ \neg \mathrm{T}(\mathbf{x},\mathbf{x},\mathbf{y}) \Rightarrow \forall \mathbf{y}\ \ \psi(\mathbf{x},\mathbf{y}) = 8 \Rightarrow \forall \mathrm{y}\ \varphi_{\mathbf{h}(\mathbf{x})}(\mathbf{y}) = 8 \Rightarrow \mathbf{h}(\mathbf{x}) \in \mathbf{LIM}$$

$$\mathbf{x} \notin \overline{\mathbf{K}} \Rightarrow \varphi_{\mathbf{x}}(\mathbf{x})\downarrow \Rightarrow \exists \mathbf{y}\ \mathrm{T}(\mathbf{x},\mathbf{x},\mathbf{y}) \Rightarrow \exists \mathbf{y}\ \psi(\mathbf{x},\mathbf{y})\uparrow \Rightarrow \exists \mathbf{y}\ \varphi_{\mathbf{h}(\mathbf{x})}(\mathbf{y})\uparrow \Rightarrow \mathbf{h}(\mathbf{x}) \notin \mathbf{LIM}$$

Puesto que $\overline{\mathbf{K}} \leq \mathbf{LIM}$ por medio de $\mathbf{h}$, y como $\overline{\mathbf{K}} \notin \Sigma_1$, podemos concluir que $\mathbf{LIM} \notin \Sigma_1$.

La anterior demostración no adopta la única vía posible para resolver el problema planteado. Cuando elegimos la función para aplicar el teorema s-m-n solemos tener varias elecciones abiertas. En el caso anterior hemos optado por que $\mathbf{h}(\mathbf{x})$ quede fuera de **LIM** obligando a $\varphi_{\mathbf{h}(\mathbf{x})}$ a no ser total. Pero otra opción adecuada hubiera sido obligarla a no ser acotada, mediante una construcción del tipo:

$$\xi(\mathbf{x},\mathbf{y}) \cong \varphi_{h(x)}(\mathbf{y}) \cong \begin{cases} 8 & \neg\, \mathbf{T}(\mathbf{x},\mathbf{x},\mathbf{y}) \\ \mathbf{y} & \text{c.c.} \end{cases}$$

que también sería computable. Nos basaríamos en el hecho de que una vez **T**(**x**,**x**,**y**) se hace cierto para un valor de **y** lo continua siendo para todos los valores mayores, ya que su significado es que $\mathbf{P}_x(\mathbf{x})$ converge en **y** ***o menos*** pasos.

Adviértase también que la complejidad de la función ψ que se ha utilizado nos indica que, en este caso, la utilización de la reducción sin el auxilio del teorema s-m-n resultaría prohibitiva. En efecto, la función **h** que estaríamos obligados a programar a mano transformaría el programa $\mathbf{P}_x$ en $\mathbf{P}_{h(x)}$, que sería algo similar a:

```
PROG:= x;
if not T(PROG, PROG, X1) then
        X0:= 8;
else
        X0:= ⊥⊥ ;
end if;
```

con el agravante de que debería ser construido como un programa-while. Esto nos obligaría a meter las miles de instrucciones asociadas a la ejecución de variantes de la función universal, lo que es totalmente irrazonable.

6.3.3. El caso de SEI y XEI

Como colofón desarrollaremos a fondo el ejemplo del conjunto de los programas que convergen al menos sobre seis entadas diferentes $\mathbf{SEI} = \{ \mathbf{x} \in \mathbb{W} : |\mathbf{W}_x| \geq 6 \}$. Naturalmente el valor seis es puramente arbitrario y el caso sirve para ilustrar la dificultad de determinar que un programa tiene un dominio de tamaño mínimo fijado cualquiera.

Demostraremos que no es decidible probando que existe algún conjunto reducible a **SEI** que tampoco lo es. Como ya es habitual probaremos suerte con **K** y buscaremos una función computable y total $\mathbf{h}: \sum^* \rightarrow \sum^*$ que verifique:

$$\forall \mathbf{x}\, (\mathbf{x} \in \mathbf{K} \leftrightarrow \mathbf{h}(\mathbf{x}) \in \mathbf{SEI})$$

o, lo que es lo mismo:

$$\forall \mathbf{x}\, (\varphi_x(\mathbf{x})\downarrow \leftrightarrow |\mathbf{W}_{h(x)}| \geq 6)$$

La función **h** que necesitamos toma un programa **x** y lo transforma en otro programa **h**(**x**) manteniendo la siguiente relación entre sus comportamientos:

- cuando el programa original $\mathbf{P}_x$ converge sobre sí mismo, el programa transformado $\mathbf{P}_{h(x)}$ converge sobre seis datos distintos o más
- cuando el programa original $\mathbf{P}_x$ diverge sobre sí mismo, el programa transformado $\mathbf{P}_{h(x)}$ converge sobre a lo sumo cinco datos distintos

Esto nos sugiere que el teorema s-m-n puede ser aplicable para justificar la computabilidad de **h** sin necesidad de escribir el programa que la computa. Además en este caso la construcción no será muy difícil, ya que con la función:

$$\psi(\mathbf{x},\mathbf{y}) \cong \begin{cases} 0 & \varphi_{\mathbf{x}}(\mathbf{x})\downarrow \\ \\ \perp & \text{c.c.} \end{cases}$$

que es computable, como lo demuestra el programa:

```
R:= Φ(X1, X1);
X0:= 0;
```

cumplimos nuestros objetivos. El teorema s-m-n nos asegura la existencia de una **h**: $\sum^* \rightarrow \sum^*$ computable y total que cumple $\forall \mathbf{x} \forall \mathbf{y}\ \psi(\mathbf{x},\mathbf{y}) \cong \varphi_{h(x)}(\mathbf{y})$. Pero se tiene, para un **x** cualquiera:

$$\mathbf{x} \in \mathbf{K} \Rightarrow \varphi_x(\mathbf{x})\downarrow \Rightarrow \forall \mathbf{y}\ \psi(\mathbf{x},\mathbf{y})=0 \Rightarrow \forall \mathbf{y}\ \varphi_{h(x)}(\mathbf{y})=0 \Rightarrow \mathbf{W}_{h(x)}=\sum^* \Rightarrow |\mathbf{W}_{h(x)}| \geq 6 \Rightarrow \mathbf{h}(\mathbf{x}) \in \mathbf{SEI}$$

$$\mathbf{x} \notin \mathbf{K} \Rightarrow \varphi_x(\mathbf{x})\uparrow \Rightarrow \forall \mathbf{y}\ \psi(\mathbf{x},\mathbf{y})\uparrow \Rightarrow \forall \mathbf{y}\ \varphi_{h(x)}(\mathbf{y})\uparrow \Rightarrow \mathbf{W}_{h(x)}=\varnothing \Rightarrow |\mathbf{W}_{h(x)}|<6 \Rightarrow \mathbf{h}(\mathbf{x}) \notin \mathbf{SEI}$$

Hemos demostrado que $\mathbf{K} \leq \mathbf{SEI}$ por medio de **h**, y como $\mathbf{K} \notin \mathbf{\Sigma}_0$, podemos concluir que $\mathbf{SEI} \notin \mathbf{\Sigma}_0$. Adviértase que al mismo tiempo estamos probando que $\overline{\mathbf{K}} \leq \overline{\mathbf{SEI}}$, por lo que $\overline{\mathbf{SEI}} \notin \mathbf{\Sigma}_1$.

Ahora nos podemos preguntar también qué sucede con **SEI**. ¿Podremos demostrar que tampoco es semidecidible? Ciertamente no, por la sencilla razón de que no es cierto. En efecto, dado un programa cualquiera $\mathbf{P}_x$ es posible detectar si su dominio contiene al menos seis elementos. Usando la técnica de intercalado descrita en [ISI 03], es posible explorar en paralelo los cómputos $\mathbf{P}_x(0)$, $\mathbf{P}_x(1)$, $\mathbf{P}_x(2)$, … e ir anotando los convergentes hasta obtener seis, si es que los hay, y devolver el resultado **true** cuando aparezcan.

Otra cosa diferente sería si estudiáramos el conjunto $\mathbf{XEI} = \{ x \in \mathbb{W} : |W_x| = 6 \}$, que tiene propiedades de decidibilidad no coincidentes. Por ejemplo, no es posible aplicar la técnica de intercalado para demostrar la semidecidibilidad de **XEI**, ya que aunque podamos detectar que hay seis elementos en W_x no tenemos forma de asegurar que no se va a producir un séptimo.

Veamos cómo aplicar las técnicas estudiadas para demostrar que $\mathbf{K} \leq \mathbf{XEI}$. Necesitamos una función computable y total $h: \Sigma^* \to \Sigma^*$ que verifique:

$$\forall x \, (\varphi_x(x)\downarrow \leftrightarrow |W_{h(x)}| = 6)$$

es decir, que:

- cuando el programa original P_x converge sobre sí mismo, el programa transformado $P_{h(x)}$ converge sobre exactamente seis datos distintos
- cuando el programa original P_x diverge sobre sí mismo, el programa transformado $P_{h(x)}$ converge sobre más o menos de seis datos distintos, pero jamás para exactamente seis

Para intentar aplicar el teorema s-m-n construimos la siguiente función:

$$\psi(x,y) \cong \begin{cases} 0 & \varphi_x(x)\downarrow \wedge \, y \leq 5 \\ \perp & \text{c.c.} \end{cases}$$

que es computable, como lo demuestra el programa:

```
R:= Φ(X1, X1);
if X2≥6 then X0:= ⊥⊥ ; end if;
X0:= 0;
```

El teorema s-m-n nos asegura la existencia de una $h: \Sigma^* \to \Sigma^*$ computable y total que cumple $\forall x \forall y \, \psi(x,y) \cong \varphi_{h(x)}(y)$. Pero se tiene, para un x cualquiera:

$$x \in \mathbf{K} \Rightarrow \varphi_x(x)\downarrow \Rightarrow \forall y \, (\psi(x,y)\downarrow \leftrightarrow y \leq 5) \Rightarrow \forall y \, (\varphi_{h(x)}(y)\downarrow \leftrightarrow y \leq 5) \Rightarrow |W_{h(x)}| = 6 \Rightarrow h(x) \in \mathbf{XEI}$$

$$x \notin \mathbf{K} \Rightarrow \varphi_x(x)\uparrow \Rightarrow \forall y \, \psi(x,y)\uparrow \Rightarrow \forall y \, \varphi_{h(x)}(y)\uparrow \Rightarrow W_{h(x)} = \varnothing \Rightarrow |W_{h(x)}| = 0 \Rightarrow h(x) \notin \mathbf{XEI}$$

Hemos demostrado que $\mathbf{K} \leq \mathbf{XEI}$ por medio de h, y como $\mathbf{K} \notin \Sigma_0$, podemos concluir que $\mathbf{XEI} \notin \Sigma_0$. Adviértase que al mismo tiempo estamos probando que $\overline{\mathbf{K}} \leq \overline{\mathbf{XEI}}$, por lo que $\overline{\mathbf{XEI}} \notin \Sigma_1$.

Ahora bien, en este caso es posible probar que **XEI** tampoco es semidecidible, lo que haremos verificando que $\overline{\mathbf{K}} \leq \mathbf{XEI}$. Para conseguirlo necesitamos encontrar una función computable y total **h**: $\sum^* \rightarrow \sum^*$ que verifique:

$$\forall \mathbf{x}\ (\varphi_{\mathbf{x}}(\mathbf{x})\uparrow \leftrightarrow |\mathbf{W}_{\mathbf{h}(\mathbf{x})}|=6)$$

La relación que debe existir entre el programa original **x** el transformado **h**(**x**) será la siguiente:

- si $\mathbf{P}_x$ diverge sobre sí mismo, $\mathbf{P}_{h(x)}$ debe converger sobre exactamente 6 elementos
- si $\mathbf{P}_x$ converge sobre sí mismo, $\mathbf{P}_{h(x)}$ debe converger sobre más de seis o menos de seis elementos

Para aplicar el teorema s-m-n elegimos la siguiente función:

$$\psi(\mathbf{x},\mathbf{y}) \cong \begin{cases} 0 & \varphi_x(x)\downarrow \vee\ \mathbf{y} \leq 5 \\ \perp & \text{c.c.} \end{cases}$$

que es computada por el programa:

```
X0:= 0;
if X2>5 then
        R:= Φ(X1, X1);
end if;
```

Dado que ψ es computable, el teorema s-m-n nos asegura que existe una función **h**: $\sum^* \rightarrow \sum^*$ computable y total que cumple $\forall \mathbf{x}\forall \mathbf{y}\ \psi(\mathbf{x},\mathbf{y}) \cong \varphi_{h(x)}(\mathbf{y})$. Veamos cómo opera esta función transformadora de programas:

- Cuando $\mathbf{x} \in \overline{\mathbf{K}}$ $\varphi_{h(x)}$ sólo puede converger para los valores $\mathbf{y} \leq 5$.
- Cuando $\mathbf{x} \notin \overline{\mathbf{K}}$ $\varphi_{h(x)}$ converge siempre.

Si lo describimos formalmente, sea un **x** cualquiera:

$$\mathbf{x} \in \overline{\mathbf{K}} \Rightarrow \varphi_x(\mathbf{x})\uparrow \Rightarrow \forall \mathbf{y}\ (\psi(\mathbf{x},\mathbf{y})\downarrow \leftrightarrow \mathbf{y} \leq 5) \Rightarrow \forall \mathbf{y}\ (\varphi_{h(x)}(\mathbf{y})\downarrow \leftrightarrow \mathbf{y} \leq 5) \Rightarrow |\mathbf{W}_{h(x)}|=6 \Rightarrow \mathbf{h}(\mathbf{x}) \in \mathbf{XEI}$$

$$\mathbf{x} \notin \overline{\mathbf{K}} \Rightarrow \varphi_x(\mathbf{x})\downarrow \Rightarrow \forall \mathbf{y}\ \psi(\mathbf{x},\mathbf{y})\downarrow \Rightarrow \forall \mathbf{y}\ \varphi_{h(x)}(\mathbf{y})\downarrow \Rightarrow \mathbf{W}_{h(x)}=\sum^* \Rightarrow \mathbf{h}(\mathbf{x}) \notin \mathbf{XEI}$$

Puesto que $\overline{\mathbf{K}} \leq \mathbf{XEI}$ por medio de **h**, y como $\overline{\mathbf{K}} \notin \Sigma_1$, podemos concluir que $\mathbf{XEI} \notin \Sigma_1$.

7. Epílogo

Hemos visto cómo la técnica de reducción puede ser aplicada para demostrar propiedades de incomputabilidad de una gran variedad de problemas. La aplicación de dicha técnica se ha realizado de manera progresiva, empezando por sus versiones más artesanales y evolucionando hacia métodos de aplicación que mecanizan y sistematizan buena parte de sus pasos. La culminación de esta evolución ha sido la utilización del Teorema s-m-n, que supone un gran avance sobre versiones previas de la técnica con un rango de aplicabilidad práctica mucho mayor.

Sin embargo, esta evolución podría continuarse si se observan algunas regularidades en la aplicación del teorema que pueden a su vez sistematizarse.

Así, podemos observar que para un gran número de problemas seguimos un esquema muy similar. Al tratar de probar una reducción del tipo $\mathbf{K} \leq \mathbf{A}$ definimos una función de la forma:

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

que es computable siempre que lo sea χ, ya que entonces podemos escribir el programa:

X0:= Φ(X1,X1); X0:= χ(X2);

La idea consiste en elegir la función $\chi(\mathbf{y})$ de manera que sus índices estén precisamente en el conjunto objetivo **A**., de manera que los $\mathbf{x} \in \mathbf{K}$ queden transformados en $\mathbf{h}(\mathbf{x}) \in \mathbf{A}$ cuando apliquemos el teorema s-m-n. Ejemplos de este proceder lo constituyen $\chi(\mathbf{y})=0$ (para convertir los programas de **K** en constantes) ó $\chi(\mathbf{y})=\mathbf{y}$ (para convertir los programas de **K** en biyecciones).

Pero para que el razonamiento sea correcto son necesarias dos condiciones: elegir la función χ de forma que todos sus índices estén en el conjunto **A**, pero también que todos los índices de la función vacía pertenezcan al complementario de **A**. De otro modo los $\mathbf{x} \notin \mathbf{K}$ también se transformarían en $\mathbf{h}(\mathbf{x}) \in \mathbf{A}$ y no se produciría la reducibilidad.

Este esquema es tan ampliamente aplicable que se generaliza mediante la siguiente propiedad:

PROPIEDAD: Sea $\mathbf{A} \subseteq \Sigma^*$. Si **VAC** $\subseteq \overline{\mathbf{A}}$ y existe alguna función computable χ que cumple Ind(χ)$\subseteq$**A**, entonces **K**$\leq$**A** y por tanto $\mathbf{A} \notin \Sigma_0$.

Esta generalización nos permite identificar "casos tipo" a partir de los cuales la construcción s-m-n queda garantizada para demostrar la indecidibilidad de un conjunto. La abstracción conseguida es tal que ya no es preciso siquiera pensar en términos de reducción: basta con realizar tareas triviales como verificar, por un lado, que ciertos programas están en **A** (los que computan a χ) que ciertos otros están fuera (los programas que computan la función vacía).

Dado un conjunto $\mathbf{A} \subseteq \mathbb{W}$, cuando todos los índices de una función computable ψ están o bien completamente contenidos en **A** o completamente excluidos de él se dice que **A** ***respeta*** la función ψ. En el caso anterior tenemos que, para aplicar la propiedad a un conjunto **A** este debe respetar dos funciones al menos: χ y $\perp\!\!\!\perp$. Unos conjuntos que resultan muy interesantes desde el punto de vista de la computabilidad son los llamados conjuntos index o conjuntos de índices: aquellos que respetan ***todas*** las funciones computables.

DEFINICIÓN: Dado un conjunto $\mathbf{A} \subseteq \mathbb{W}$, decimos que **A** es un ***índex*** si para cualquier función computable ψ se verifica o bien Ind(ψ)$\subseteq$**A** o bien Ind(ψ)$\subseteq \overline{\mathbf{A}}$.

Un índex puede ser visto como un conjunto que describe propiedades semánticas de los programas. Todos los programas equivalentes entre sí (es decir, que son índices de una misma función) tienen que cumplir las mismas propiedades semánticas, y si un conjunto **A** de programas-while está definido por una propiedad de ese tipo, entonces todos están obligados a cumplirla (todos pertenecen al conjunto) o a incumplirla (todos pertenecen al complementario). No puede darse el caso de dos programas que hagan lo mismo y estén en bandos diferentes.

Muchos de los conjuntos con los que hemos trabajado en los capítulos anteriores, como **TOT**, **VAC**, **CONS** o **PERM**, son índex. Una notable excepción es **K**., ya que el que un programa converja sobre sí mismo no implica que lo haga sobre todos los programas equivalentes a él.

El resultado indicado en la propiedad enunciada más arriba y otros similares dan lugar a los ***Teoremas de Rice***, que precisamente estudian y clasifican los index, encontrando muchas propiedades interesantes. La primera y más llamativa es que ***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. Otro resultado establece que para que una propiedad semántica de los programas-while no sea siquiera semidecidible basta con que dicha propiedad sea satisfecha por un programa trivial (que siempre devuelve indefinido).

Aunque queda fuera del objeto de este trabajo enunciar en extensión o demostrar estas propiedades, valga como colofón este paradójico resultado, que por otro lado se demuestra de forma muy elegante gracias a las técnicas de reducción que hemos intentado desbrozar. La Ingeniería Informática, que ha demostrado ser una herramienta tan extraordinariamente útil para resolver problemas en tantos campos, incluso hasta el punto de resultar instrumental en la demostración de algunos teoremas matemáticos complejos, es incapaz de resolver la pregunta más nimia que se haga dentro de su propio campo cuando se toca el tema de predecir el comportamiento de los programas.

Apéndice : Programas-while y funciones computables

En este apéndice se repasan todos los fundamentos que nos han permitido programar como lo hemos hecho a lo largo del presente informe. En primer lugar describiremos el lenguaje de los programas-while, que son el soporte último de la teoría de la computabilidad que hemos desarrollado. A pesar de su simplicidad, este lenguaje contiene los elementos necesarios para programar cualquier función computable.

El hecho de definir un lenguaje minimalista no sólo obedece al deseo de explorar cuáles son las primitivas esenciales para tener un sistema de programación completo. Como hemos visto en el presente informe, los programas-while también son utilizados como objetos sobre los que se construyen otros programas que los editan, manipulan o ejecutan. En ese momento resulta esencial que los programas-while constituyan un conjunto manejable de objetos.

La contrapartida es que su uso como formalismo para la descripción de algoritmos resulta farragoso, pues hasta las acciones más sencillas requieren algunas líneas de código. Por ello veremos distintos métodos para describir programas-while complejos mediante abreviaturas accesibles llamadas macroprogramas. Y dado que uno de los principios de los macroprogramas es incorporar a la notación de descripción algorítmic las funciones cuya computabilidad ya ha sido demostrada, iremos dando una lista de funciones computables que han sido utilizadas en el texto pero cuya computabilidad no se ha reproducido en el texto.

A.1. El lenguaje de los programas-while

Los programas-while están constituidos por instrucciones que operan sobre palabras construidas a partir de un alfabeto $\sum = \{\mathbf{s}_1, \ldots, \mathbf{s}_n\}$. Para referenciar las palabras que el programa manipula se utilizan 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 := $cons_s$(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 car_s?(XI) then Q end if;
$\mathbf{i} \in \mathbb{N}$ $Q \in \mathbb{W}$	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 éstos ya se encuentran almacenados en las variables **X1**, **X2**, …, **XN**, por ese orden (variables de entrada).
2. Las variables del programa que no son ocupadas por datos de entrada se inicializan con el valor ε (palabra vacía).
3. Se ejecutan las instrucciones del programa sobre las variables.
4. Cuando el programa termina (si lo hace), se considera como único resultado de la computación el contenido final de la variable **X0** (variable de salida).
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 tipos de programas de la definición es:

XI := ε; — Sustituir el contenido de la variable **XI** por la palabra vacía

XI := $cons_s$(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 $car_s?$(XI) **then** Q **end if**;	Ejecutar el programa **Q** si el primer símbolo por la izquierda de la palabra contenida en **XI** es **s**
while nonem?(XI) **loop** Q **end loop**;	Mientras el contenido de **XI** sea una palabra no vacía, ejecutar Q

A.2. Macros

Una macro es un mecanismo de abreviatura que permite describir un programa-while de manera compacta y evita la repetición de código utilizado frecuentemente. Pueden definirse tantos tipos de macros como se desee, pero cada vez que se introduce un nuevo tipo es necesario proporcionar su ***expansión***, es decir, la forma en que esa macro puede ser convertida en un programa-while equivalente [IIS 96]. Por tanto la macro no es más que una abreviatura de dicho programa-while. Nos limitaremos a describir las macros que resultan útiles para el desarrollo del texto, pero para consultar su expansión el lector deberá remitirse al informe mencionado [IIS 96].

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 cuya while-computabilidad ya ha sido probada. 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, en lugar de repetir de manera tediosa el código del programa que computa ψ[14]. En secciones posteriores de este apéndice recopilamos una lista de la

[14] 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

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;
  while R(V_1, ..., V_K) **loop** Q **end loop**;

donde **R** es un predicado cuya while-decidibilidad ya ha sido probada, $\mathbf{V_1}$, ..., $\mathbf{V_K}$ pueden ser variables o macrovariables y **Q** es un macroprograma cualquiera. 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. En secciones posteriores de este apéndice 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.

- **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**.

- **case** V **is when** A_1 => P_1 ... **when** A_K => P_K **otherwise** P_{K+1} **end case**;

donde **V** es una variable o macrovariable, $\mathbf{A_1}$ ... $\mathbf{A_K}$ son macroexpresiones y $\mathbf{P_1}$ **...** $\mathbf{P_{K+1}}$ son macroprogramas.

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. Los macroprogramas son útiles y cómodos, pero en ningún caso imprescindibles, ya que el lenguaje de los programas-while es suficiente para demostrar la computabilidad de cualquier función que lo permita.

A.3. Funciones con palabras

Hay un gran número de funciones computables que son utilizadas en nuestros macroprogramas, empezando por algunas que resultan tremendamente útiles para

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';.

manipular las palabras. Incluimos las operaciones que utilizamos para trabajar con palabras sobre un alfabeto genérico $\sum = \{\mathbf{s}_1, \ldots, \mathbf{s}_n\}$.

- La función $\mathbf{cons}_s : \sum^* \to \sum^*$

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

- La función $\mathbf{cdr} : \sum^* \to \sum^*$

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

- El predicado $\mathbf{car}_s\mathbf{?} : \sum^* \to \mathbb{B}$

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

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

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

- 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

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

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

Esta función es usada en el texto con su notación algebraica ($\mathbf{x} \bullet \mathbf{y}$ en vez de $\mathbf{x} \& \mathbf{y}$).

- 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*** $K_w : \sum^* \to \sum^*$ se define como

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

- 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:

$$\mathbf{sig}(\varepsilon) = \mathbf{a}_1$$

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

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

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

- La función $\mathbf{cod}^2 : \sum^* \times \sum^* \to \sum^*$, biyección que codifica cada par de palabras en una sola de manera biunívoca:

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

$$\mathbf{cod}^2(\mathrm{sig}(\mathbf{x}), \varepsilon) = \mathrm{sig}(\mathbf{cod}^2(\varepsilon, \mathbf{x}))$$

$$\mathbf{cod}^2(\mathbf{x}, \mathrm{sig}(\mathbf{y})) = \mathrm{sig}(\mathbf{cod}^2(\mathrm{sig}(\mathbf{x}), \mathbf{y}))$$

- Las funciones $\mathbf{decod}_{2,1} : \sum^* \to \sum^*$ y $\mathbf{decod}_{2,2} : \sum^* \to \sum^*$ que recuperan las componentes de un par de palabras codificadas mediante $\mathbf{cod}^2$:

$$\mathbf{cod}^2(\mathbf{decod}_{2,1}(\mathbf{x}), \mathbf{decod}_{2,2}(\mathbf{x})) = \mathbf{x}$$

A.4. Tipos de datos simples y funciones asociadas

Tenemos otro convenio para poder utilizar en nuestros macroprogramas operaciones correspondientes a otros tipos de datos distintos de las palabras. En realidad estos son representados adecuadamente mediante palabras, pero su uso directo en los algoritmos resultan más transparentes para el programador. Para poder usar un tipo de datos es necesario proceder a su ***implementación***, es decir, a describir cuál es el mecanismo de representación subyacente, como se describe en [IIS 96], donde se implementan algunos tipos de datos que resultan útiles en nuestro campo. Los tipos soportados son los

siguientes: los números 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. Como ya hemos indicado éstos 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 elementos constitutivos entre los símbolos **(** y **)**, por ejemplo **(aba, bb, ε, a)**, en el que **aba** es la 0-esima componente del vector. Decimos que son vectores dinámicos porque su tamaño no está limitado de antemano, pudiéndose añadir un número arbitrario de componentes en tiempo de ejecución.

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:

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

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

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

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

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

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

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

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

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

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):

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

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

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

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

Por respeto a la notación más utilizada, para indicar **acceso**(**v**,**i**) en general utilizaremos 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:

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

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

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 ***macroasignación a componente de vecto***r `V(I):= W;`.

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

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 asignaciones como:

```
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.

A.5. El tipo de datos programas-while y sus funciones asociadas

El tipo de datos $\mathbb{W}$ está constituido por el conjunto de los programas-while. 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.

La función **haz_asig_vacía** : $\mathbb{N} \to \mathbb{W}$ construye una asignación vacía:

$$\textbf{haz\_asig\_vacía}(\mathbf{n}) = \mathrm{XN{:=}}\ \varepsilon;$$

Las funciones **haz_asig_cons_s** : $\mathbb{N} \times \mathbb{N} \to \mathbb{W}$ (hay una para cada símbolo del alfabeto utilizado) construye una asignación tipo **cons**:

$$\textbf{haz\_asig\_cons\_s}(\mathbf{n},\mathbf{m}) = \mathrm{XN{:=}}\ cons_s(\mathrm{XM});$$

La función **haz_asig_cdr** : $\mathbb{N} \times \mathbb{N} \to \mathbb{W}$ construye una asignación tipo **cdr**:

haz_asig_cdr(**n**,**m**) = XN:= cdr(XM);

La función **haz_composición** : $\mathbb{W} \times \mathbb{W} \to \mathbb{W}$ concatena secuencialmente programas previamente definidos:

haz_composición($\mathbf{P_1}$,$\mathbf{P_2}$) = P_1 P_2

Las funciones **haz_condición_s** : $\mathbb{N} \times \mathbb{W} \to \mathbb{W}$ (hay una para cada símbolo del alfabeto utilizado) construyen un programa condicional:

haz_condición_s(**n**,**Q**) = **if** car_s?(XN) **then** Q **end if**;

La función **haz_iteración** : $\mathbb{N} \times \mathbb{W} \to \mathbb{W}$

haz_iteración(**n**,**Q**) = **while** nonem?(XN) **loop** Q **end loop**;

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

asig_vacía? : $\mathbb{W} \to \mathbb{B}$

asig_cons? : $\mathbb{W} \to \mathbb{B}$

asig_cdr? : $\mathbb{W} \to \mathbb{B}$

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

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

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

- Las ***operaciones de acceso***, que nos indican qué elementos intervienen como componentes de un programa-while.

Las funciones **ind_var** : $\mathbb{W} \to \mathbb{B}$ e **ind_var_2** : $\mathbb{W} \to \mathbb{B}$ nos devuelven el índice de las variables implicadas en el más alto nivel de construcción de un programa:

ind_var(XI:=ε;) = **i**

ind_var(XI:=$cons_s$(XJ);) = **i**

ind_var(XI:=cdr(XJ);) = **i**

ind_var(**if** car_s?(XI) **then** Q **end if**;) = **i**

ind_var(**while** nonem?(XI) **loop** Q **end loop**;) = **i**

ind_var(P_1 P_2) ≅ ⊥

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.

$$\textbf{ind\_var\_2}(\texttt{XI:=cons}_s\texttt{(XJ);}) = \textbf{j}$$

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

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

Disponemos también de la función **ind_simb** : $\mathbb{W} \to \sum^*$ que nos permite extraer el símbolo implicado en las funciones **cons** y **car** cuando estas intervienen en el nivel más alto de construcción del programa (es decir, si se trata de una asignación **cons** o de una condición)

$$\textbf{ind\_simb}(\texttt{XI:=cons}_s\texttt{(XJ);}) = \textbf{s}$$

$$\textbf{ind\_simb}(\textbf{if } \texttt{car}_s\texttt{?(XI)} \textbf{ then } \texttt{P } \textbf{end if;}) = \textbf{s}$$

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

Por último, las funciones **inst_int** : $\mathbb{W} \to \mathbb{W}$ e **inst_int_2** : $\mathbb{W} \to \mathbb{W}$ permiten acceder a las instrucciones internas o subprogramas-while constitutivos de un programa, cuando procede.

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

$$\textbf{inst\_int}(\textbf{if } \texttt{car}_s\texttt{?(XI)} \textbf{ then } \texttt{Q } \textbf{end if;}) = \mathbf{Q}$$

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

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

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

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

A.5.1. Funciones estáticas sobre los programas while

Una vez definidas las funciones que realizan operaciones elementales sobre los programas-while podemos pasar a definir y programar otras más complejas que

manipulan el texto de los programas en busca de propiedades que pueden detectarse en tiempo de compilación, es decir, que se derivan exclusivamente del texto del programa. Algunos ejemplos que nos resultan de utilidad para el presente texto son:

ult_variable: $\mathbb{W} \to \mathbb{N}$

nivel_anidamiento: $\mathbb{W} \to \mathbb{N}$

La primera determina cuál es el máximo índice de variable que aparece en un programa cualquiera, y la segunda calcula el máximo nivel de anidamiento que contiene un programa. Veremos cómo se puede demostrar la computabilidad de ambas. El esquema es el mismo en ambas funciones: ir descomponiendo el programa-while que nos suministran como dato y procesar cada "trozo" de programa para resolver el problema: en un caso analizar las variables que aparecen en cada instrucción y en el otro estudiar el nivel de anidamiento alcanzado. Pero la descomposición del programa puede producir una lista de subprogramas a la espera de ser procesados, y además esta lista puede ser arbitrariamente larga. Para ocuparnos de ellos de manera ordenada los iremos guardando en una pila, que inicialmente contendrá como único elemento el programa-while que se toma como dato, y que marcará el fin del proceso en el momento de vaciarse

En el caso de **ult_variable** iremos almacenando en **X0** el mayor índice de variable que llevemos encontrado hasta el momento.

```
 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 tipo
         if asig_vacía? (PROG) then
            X0 := max(X0, ind_var (PROG));
         elsif asig_cons? (PROG) or asig_cdr? (PROG) then
            X0 := max(X0, ind_var (PROG), ind_var_2 (PROG));
         elsif condición? (PROG) or iteración? (PROG) then
            X0 := max(X0, ind_var (PROG));
            PILA := empilar (inst_int (PROG), PILA);
         else PILA := empilar (inst_int_2 (PROG), PILA);
            PILA := empilar (inst_int (PROG), PILA);
         end if;
 end loop;
```

Para calcular el máximo nivel de anidamiento hay que considerar que este se incrementa cada vez que entramos en un **if** o un **while** y que se reduce al salir. En este caso en la pila guardaremos pares (**P**,**n**), donde **P** es un subprograma del que teníamos originalmente, y **n** es el nivel de anidamiento a que se ha encontrado dicho subprograma.

```
PILA := empilar(cod_2(X1, 0), <]);
X0 := 0;
while not P_vacía?(PILA) loop
        PROG := decod_2_1 (cima(PILA));
        NIVEL := decod_2_2 (cima(PILA));
        PILA := desempilar(PILA);
        if asig_vacía?(PROG) or asig_cons?(PROG) or asig_cdr?(PROG) then
            X0 := max(NIVEL,X0);
        elsif composicion?(PROG) then
            PILA := empilar(cod_2(inst_int_2(PROG), NIVEL), PILA);
            PILA := empilar(cod_2(inst_int(PROG), NIVEL), PILA);
        else PILA := empilar(cod_2(inst_int(PROG), NIVEL+1), PILA);
        end if;
end loop;
```

A.5.2 Funciones dinámicas sobre los programas-while

Desde el punto de vista de la Teoría de la Computabilidad son mucho más interesantes las funciones que, actuando sobre programas, su resultado depende del comportamiento o ejecución de éstos. Y entre ellas la más importante es la ***función universal***, descrita en detalle en [ISI 03].

La función universal toma como argumentos un programa $\mathbf{P}_x$ y un dato (palabra) **y**, y relaciona este par con la salida (palabra) que produciría el programa al suministrarle el dato indicado. Así, la función universal describe el resultado de cualquier cómputo imaginable, y de ahí su pretencioso nombre. Se denota y define de la siguiente manera:

$$\Phi: \mathbb{W} \times \Sigma^*: — \to \Sigma^*, \text{ donde } \Phi(\mathbf{x}, \mathbf{y}) \cong \varphi_x(\mathbf{y})$$

La importancia de esta función reside en el hecho de que contiene en su seno a todas las funciones computables, independientemente de que los programas que las computen hayan sido (o vayan a ser) formulados alguna vez. El truco está en que Φ en realidad asocia cada algoritmo con sus resultados (ya que hay que administrarle como dato el índice de la función a calcular, o los que es lo mismo, el código del programa que la

calcula). De esta manera el programa que computa la función universal es capaz de representar a todos los demás programas, encerrando dentro de sí todas las esencias de la computación. Para demostrar su computabilidad (cosa que no haremos aquí pero puede consultarse en [ISI 03]) se utiliza una técnica parecida a la estudiada en la sección anterior: mediante una pila se va descomponiendo el programa en sus componentes elementales, y cuando se accede a cada uno de ellos se ejecuta. Para reflejar la acción del programa en su medio se utiliza un vector de estado que conserva el contenido de las variables a medida que la ejecución del programa es emulada.

Puede haber casos en los que, en lugar de averiguar el comportamiento de un programa hasta su finalización interese simular solo un cierto número de pasos del mismo. La razón de realizar una prospección limitada estriba en la necesidad de evitar ser arrastrados por una eventual divergencia del programa: si $\varphi_x(\mathbf{y})\uparrow$ entonces también $\Phi(\mathbf{x}, \mathbf{y})\uparrow$, y preguntar por el destino de un programa que cicla nos obliga a compartirlo. El predicado decidible **T** explora la convergencia de un programa sobre un dato con un número de pasos como tope máximo, evitando ese riesgo (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). Formalmente:

$\mathbf{T}(\mathbf{x},\mathbf{y},\mathbf{p}) \Leftrightarrow \mathbf{P}_x$ converge sobre la entrada **y** en **p** o menos pasos

A veces es conveniente averiguar no solamente si el programa converge o no, sino cuál es su resultado. Para ello se dispone de un predicado decidible **E** que toma un argumento más que **T** para poder hacer esta pregunta:

$\mathbf{E}(\mathbf{x},\mathbf{y},\mathbf{p},\mathbf{z}) \Leftrightarrow \mathbf{T}(\mathbf{x},\mathbf{y},\mathbf{p})$ y además $\varphi_x(\mathbf{y})=\mathbf{z}$

Estos dos mecanismos de ejecución limitada pueden ser muy útiles cuando se quieren explorar a la vez los comportamientos de varios programas. En [ISI 03] se estudian algunas aplicaciones de esta técnica.

Tanto la función Φ como los predicados **T** y **E** admiten extensiones con un número mayor de argumentos. Esto sucede cuando el programa $\mathbf{P}_x$ que opera como primer argumento es utilizado con más de un dato de entrada.

Referencias

[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.

[IS 09] J. IBÁÑEZ y A. SÁNCHEZ: Constructive Reduction: Understanding uncomputability through programming. ACM SIGCSE Bulletin, vol. 41, nº 2, pp.: 90-94, junio 2009.

[ISI 03] A. IRASTORZA; A. SANCHEZ; J. IBAÑEZ. Técnicas Básicas de Computabilidad. Informe interno UPV/EHU/LSI/TR 3-2003.

[Mor 98] B. MORET;. The theory of Computation. Addison-Wesley, 1998

[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 1988