Docencia de la asignatura Modelos Abstractos de Cálculo.

Pilar Arques Corrales
Domingo Gallardo López
Miguel Ángel Cazorla Quevedo

Departamento de Ciencia de la Computación e Inteligencia Artificial
Universidad de Alicante
(arques,domingo,miguel@dccia.ua.es)

Resumen

En este trabajo presentamos nuestro enfoque de la asignatura Modelos Abstractos de Cálculo, que introduce a los estudiantes de las titulaciones de Informática en la Teoría de la Computabilidad. Hacemos especial énfasis en la adaptación de los contenidos teóricos a los conocimientos y habilidades de dichos estudiantes.

1 Introducción

La asignatura Modelo Abstractos de Cálculo plantea un estudio en amplitud sobre los aspectos fundamentales de la Teoría de la Computabilidad: definición de lenguajes y conjuntos, modelos de computación e indecidibilidad.

La asignatura es obligatoria en segundo curso de las titulaciones de Ingeniería Informática e Ingeniería Técnica en Informática de Sistemas, y es optativa en la Ingeniería Técnica en Informática de Gestión. En todos los casos tiene 3 créditos teóricos y 1,5 prácticos.

El objetivo general de la asignatura es *dotar a los alumnos de conocimientos y habilidades relativas a los conceptos y problemas básicos de la computabiliad.* Además, como en todas las asignaturas fundamentalmente teóricas de las titulaciones de informática, se pretende mejorar las capacidades de formalización, abstracción y rigor de los estudiantes.

Como objetivos más concretos destacamos:

1. Conocer los límites de la computación, sabiendo reconocer algunos problemas como no computables.
2. Conocer distintas clases de conjuntos, en función de su recursividad y decidibilidad.
3. Dominar técnicas formales para la demostración de propiedades de conjuntos relacionadas con la computabilidad de los mismos.
4. Conocer diferentes modelos de computación – lenguaje L, Máquinas de Turing, funciones recursivas –, utilizándolos para definir lenguajes y conjuntos, relacionándolos con lenguajes de programación usuales y demostrando su equivalencia.
5. Utilizar las técnicas de reducción de problemas para demostrar la indecidibilidad de conjuntos.
6. Comprender la importancia de la materia de la asignatura como base central de los aspectos teóricos de la informática.

Como bibliografía básica de la asignatura veníamos recomendando los manuales Davis [1] y Hopcroft [3]. En el libro [2] se recopila el material y la experiencia acumulada a lo largo de los años de impartir la asignatura, siendo este el manual que actualmente se sigue.

2 Problemática de la asignatura

Esta asignatura, eminentemente teórica, no es acogida de buen grado por lo alumnos de informática, más habituados a asignaturas con perfil práctico y aplicado.

En nuestro enfoque de la asignatura pretendemos motivar al alumno utilizando ejemplos y técnicas sencillas y cercanas a ellos como los lenguajes de programación, los programas, los compiladores e intérpretes, etc.

Sin embargo, no pretendemos dejar de lado ningún formalismo ni demostración matemática, pero sí queremos darle un enfoque con el que el alumno de informática se sienta más identificado, es decir, hacemos especial énfasis en relacionar la mayoría de conceptos de la teoría de la Computabilidad con la experiencia en lenguajes de programación que tienen estos alumnos.

3 Metodología docente

Los 4,5 créditos de la asignatura se dividen en 2 horas semanales de teoría y 1 de prácticas. Tanto los créditos prácticos como teóricos, se imparten en clases típicas "de pizarra". Además de las clases presenciales, se propone a los alumnos un trabajo para casa, que consiste en la resolución de un número de *hojas de problemas* cuya evaluación se añadirá a la nota obtenida por el alumno en el típico examen final.

- **Clases teóricas**. En las clases de teoría se explican los contenidos de la asignatura, intentando en todo momento basarnos en ejemplos relacionados con conceptos informáticos.

- **Clases prácticas**. Consideramos más interesante ocupar las 15 horas de laboratorio en clases prácticas de actividades y de resolución de problemas, en las que se aplican y refuerzan los conceptos aprendidos en las clases de teoría, que utilizarlas como laboratorios de programación. En una asignatura principalmente abstracta como MAC, las prácticas de programación (como la implementación de una máquina de Turing o la utilización de algún lenguaje de programación para implementar funciones recursivas) son de un interés lateral, pues no las consideramos imprescindibles para conseguir los objetivos que nos hemos marcado. La componente de aplicación de la asignatura se orienta más hacia la resolución de problemas *sobre el papel*, intentando aumentar las capacidades de abstracción de los estudiantes.

  Se cuenta con una colección de alrededor de 100 problemas que cubren todos los temas de la asignatura. Conforme avanzan las exposiciones teóricas, se pide que los estudiantes intenten resolver algunos de ellos. Las clases de problemas deberán, entonces, servir para resolver dudas planteadas al intentar solucionar los problemas y para comparar diferentes enfoques. El hecho de ser clases con un número reducido de alumnos (alrededor de 30) facilitará la posibilidad de que la mayoría de ellos participen de forma activa en la discusión.

  Sin embargo, la realidad que se plantea de forma habitual está bastante alejada de estos planteamientos. En la realidad, la mayoría de alumnos que asisten a estas clases no ha intentado resolver ninguno de los ejercicios, o, si lo ha hecho, no lo suele demostrar con intervenciones activas. Para evitar esto, el profesor debe tomar un papel activo en el que se animen las intervenciones y la participación dentro de los grupos.

- **Hojas de problemas**. A lo largo del curso se plantean a los alumnos selecciones de problemas de un nivel de dificultad algo superior al de los resueltos en las clases de prácticas. La resolución de estas hojas se hace de forma individual y deben ser entregadas en un plazo breve de tiempo previamente fijado. Estas hojas se puntúan y su evaluación se añade a la del examen final.

  Estas hojas de ejercicios cumplen el objetivo de motivar a los alumnos a realizar un estudio continuo de la materia.

4 Temario

Dividimos la asignatura en 7 capítulos

- **Preliminares**. En él tratamos de introducir brevemente conceptos matemáticos muy sencillos que serán necesarios para el seguimiento de la asignatura: teoría de conjuntos, funciones, alfabetos, métodos de demostración,...

- **Máquinas de Turing**. Se estudia el concepto de Máquina de Turing como primera formalización del concepto de algoritmo. Estudiamos el modelo de Máquina de Turing tanto para la definición de lenguajes de cadenas para computar funciones. Definimos las funciones Turing-computables como aquellas que pueden ser calculadas por Máquinas de Turing. Por último se demuestra, de forma intuitiva, la equivalencia entre diferentes modelos de Máquinas de Turing.

  La introducción del modelo computacional de Máquina de Turing al comienzo de la asignatura se justifica tanto por motivos históricos como

por motivos metodológicos. Se trata de un modelo muy sencillo y muy cercano a los esquemas operativos de un estudiante acostumbrado a diseñar programas. También sirve para introducir al estudiante en la importante idea (reiterada en múltiples ocasiones a lo largo del curso) de que la simplicidad semántica no implica una merma de la capacidad computacional del modelo.

- **Funciones L-computables.** Un sencillo lenguaje de programación de funciones, el lenguaje L, nos servirá para definir los conceptos de función L-computable y función L-computable parcial. No se pretende que este lenguaje sea fácil de utilizar ni eficiente, sino que sea *mínimo*. Entendemos por ello que tenga la misma capacidad computacional que cualquier lenguaje de programación al uso y que posea la menor complejidad semántica y sintáctica posible.

- **Funciones recursivas-primitivas.** Introducimos una nueva notación para definir funciones matemáticas y demostramos que la recursión primitiva no es una notación completa, ya que existen funciones L-computables totales que no se pueden definir con ella. Por último se introducen, de manera intuitiva, las funciones recursivas-μ, proporcionando esta definición el elemento formal que nos faltaba para ampliar la notación de funciones recursivas primitivas, y hacerla equivalente a la de funciones L-computables.

- **Un programa Universal.** Demostramos que es posible, con el lenguaje L, construir un programa capaz de tomar por entrada un programa P cualquiera y que compute la misma función computada por P. Establecemos paralelismos entre este comportamiento y el funcionamiento de un intérprete de lenguaje L, acercando así a los alumnos el concepto planteado por Turing de Computación Universal.

  Para ello utilizamos la numeración de Gödel para codificar mediante números naturales tanto programas L como instantáneas de las ejecuciones de los mismos. Definimos entonces funciones recursivas primitivas que permiten simular la ejecución de un programa cualquiera operando sobre estas codificaciones.

- **Equivalencia de modelos computacionales.** En este tema se demuestra formalmente la equivalencia de los conceptos de función Turing-computable, L-computable y recursiva-μ-computable, presentando construcciones que permiten simular un modelo computacional con otro.

- **Decidibilidad e indecidibilidad.** Vemos resultados sobre las limitaciones del lenguaje de programación L y, como resultado de la tesis de Church, de cualquier proceso algorítmico.

  Estudiamos dos problemas indecidibles, clásicos en la teoría de la Computabilidad como son el problema de la parada y el castor afanoso. Definimos los conjuntos recursivos, recursivamente enumerables y no recursivamente enumerables, que nos van a formalizar los conceptos de decidibilidad e indecidibilidad. Para finalmente concluir enunciando el teorema de Rice. Todo ello lo hacemos utilizando el lenguaje de programación L, como veremos con más detalle en el apartado siguiente.

5 Ejemplo: conjuntos recursivamente enumerables

En este apartado presentamos un ejemplo concreto de cómo un concepto de un nivel de abstracción matemática elevado, como es el de *conjunto recursivamente enumerable*, se adapta a esquemas conceptuales comunes en los alumnos de Informática, como son los lenguajes de programación y el hecho de que un programa termine o se *cuelgue*.

Definición 1 *Un conjunto $B \subseteq N$ se denomina* recursivamente enumerable *(r.e.) si existe una función L-computable $g(x)$ que esté definida únicamente para aquellos números naturales que pertenecen a B:*

$$B = \{x \in N \mid g(x) = \downarrow\}$$

Esto es, un conjunto es recursivamente enumerable cuando es el dominio de una función L-computable.

El programa P que computa la función g será un programa que devolverá un resultado para algunos valores de entrada, y que se colgará para otros valores. Entonces, el conjunto B será aquel conjunto de valores para los que el programa no se queda colgado. Por ello, podemos pensar en P como un algoritmo con el que determinar la pertenencia de un número a B, de forma que, para aquellos números que

pertenezcan a B el programa, al terminar, dará una respuesta "SÍ", pero para aquellos números que no pertenezcan a B el programa no termina nunca. A este tipo de algoritmos se les denomina *procedimientos de semidecisión*, ya que permiten definir de forma parcial el conjunto B (podemos saber cuándo un elemento pertenece a B pero no cuando *no* pertenece). Veremos más adelante que cualquier conjunto recursivo es también r.e. pero (y esto es lo más importante) que existen conjuntos r.e. para los que no es posible definir un procedimiento de decisión (no son recursivos).

Los conjuntos r.e. no recursivos son conjuntos, en cierta manera, semidefinidos. Mediante los procedimientos de semidecisión que los caracterizan se puede saber cuáles son sus elementos, pero no qué elementos se quedan fuera de ellos. Veremos que una de las características fundamentales de un conjunto r.e. no recursivo es que no es posible conocer su conjunto complementario.

Presentamos a continuación dos ejemplos de conjuntos r.e., el conjunto **K** y el conjunto **NO_VACÍO**, así como la demostración de que cada uno de ellos es r.e.

Definición 2 *Definimos el conjunto* **K** *como aquellos códigos de programa n que representan programas que se detienen cuando se les pasa el propio número n como entrada:*

$$\mathbf{K} = \{x \in N \mid \Omega(x, x) = \downarrow\}.$$

Por ejemplo, el número $2^{10} - 1$ pertenece a **K**, ya que codifica el programa

$$X{+}{+}$$

que es un programa que se detiene para cualquier entrada (y, por lo tanto, también se detiene para $X = 2^{10} - 1$).

Sin embargo, el número $2^{22} \cdot 3^{93} - 1$ no pertenece a **K**, ya que codifica el programa

$$\begin{array}{ll} & X{-}{-} \\ (A) & \text{IF } X \neq 0 \text{ GOTO } A \end{array}$$

que sólo se detiene con los números 0 e 1, por lo que no se detiene con su propio código $(2^{22} \cdot 3^{93} - 1)$.

Teorema 1 *El conjunto* **K** *es r.e.*

Demostración

Demostramos que **K** es r.e. mediante el siguiente procedimiento de semidecisión

$$\begin{array}{ll} (A) & \text{IF PASOS}(X, X, Z) \text{ GOTO } S \\ & Z{+}{+} \\ & \text{GOTO } A \end{array}$$

El predicado PASOS(x, y, z) toma como entrada un número natural x, un código de programa y y un número de pasos de ejecución z, devolviendo TRUE si y sólo si el programa y se detiene con la entrada x en z pasos de ejecución (o en menos).

De esta forma, el programa anterior comprueba si existe un cierto número de pasos Z tal que el programa codificado con X se detiene con la entrada X. Este programa se detendrá con aquellos $x \in \mathbf{K}$, pero entrará en un bucle infinito con el resto, por lo que constituye un procedimiento de semidecisión de **K**, demostrando que este conjunto es r.e.

Definición 3 *Sea el conjunto* ***NO_VACÍO*** *el conjunto de codificaciones de programas que se detienen para algún número de entrada,*

$$\textit{NO\_VACÍO} = \{x \in N \mid \exists\, y \in N\ \Omega(y, x) = \downarrow\}.$$

Por ejemplo, el número $2^2 - 1 = 3$ codifica al programa

$$Y{+}{+}$$

que se detiene se cual sea el valor de la variable de entrada que se le pase. Por ello, $3 \in$ NO_VACÍO.

Sin embargo, el número $2^{21} \cdot 3^{46} - 1 \notin$ NO_VACÍO, ya que codifica al programa

$$\begin{array}{ll} (A) & X{+}{+} \\ & \text{IF } X \neq 0 \text{ GOTO } A \end{array}$$

que se cuelga para cualquier número de entrada.

Teorema 2 *El conjunto* ***NO_VACÍO*** *es r.e.*

Demostración

Para demostrar que **NO_VACÍO** es r.e. construiremos, utilizando el predicado PASOS, un programa de semidecisión que, para un número de entrada x decida si el programa codificado por este número se detiene para *algún* valor de entrada:

$$\begin{array}{ll} (A) & \text{IF PASOS}(Z, X, Z_2) \text{ GOTO } S \\ & Z{+}{+} \\ & \text{IF } Z \leq Z_2 \text{ GOTO } A \\ & Z_2{+}{+} \\ & Z \leftarrow 0 \\ & \text{GOTO } A \end{array}$$

Como se puede comprobar, el programa va comprobando todas las posibles combinaciones de número de pasos y números de entrada sobre el programa codificado X. Así, si existe algún número k con el que X se detiene después de l pasos, el programa anterior también se detendrá en el momento en que $Z = k$ y $Z_2 = l$, por lo que el programa se detendrá únicamente con aquellos $x \in$ **NO_VACÍO**.

Referencias

[1] Martin D. Davis, Ron Sigal y Elaine J. Weyuker. *Computability, Complexity, and Languages.* Academic Press, 1994.

[2] Domingo Gallardo, Pilar Arques e Ignacio Lesta. *Introducción a la Teoría de la Computabilidad.* Publicaciones de la Universidad de Alicante, 1998.

[3] J. E. Hopcroft y J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979. Traducción española en Ed. CECSA, 1993.