Una alternativa docente a la Máquina de Turing

Morales, Rafael Ramos, Gonzalo Vico, Francisco J. Triguero, Francisco
Dpto. de Lenguajes y Ciencias de la Computación
E.T.S. Ingeniería Informática
Universidad de Málaga
Bulevar Louis Pasteur 35, 29071-Málaga
e-mail: morales@lcc.uma.es

Resumen

En la docencia de la Informática Teórica se necesita definir un modelo de cálculo para introducir formalmente el concepto de calculabilidad, y los que se derivan de éste. Esta introducción se suele hacer en la asignatura Teoría de Autómatas y Lenguajes Formales (TALF), generalmente en segundo año del primer ciclo de Informática, y habitualmente escogiendo la Máquina de Turing (MT) como modelo de cálculo. Presentamos aquí una alternativa docente a la MT basada en utilizar como modelo de cálculo un lenguaje estructurado simple, el lenguaje WHILE. Esta alternativa tiene dos ventajas docentes importantes. La primera es que dicho modelo es más cercano a los conocimientos que ya posee el alumno, por lo que es más fácil su asimilación. La segunda es que las demostraciones de las proposiciones y teoremas relacionados son mucho más cortas y claras.

1. Introducción

Dentro de la Informática Teórica, la Teoría de la Calculabilidad (TC) es una de sus partes más importantes. En la docencia de la TC se suele presentar una dificultad para su asimilación por parte del discente de Informática debido, entre otros factores, al alto grado de abstracción de sus contenidos. Por ello es importante cualquier mejora que podamos introducir en la forma de impartirla.

Un concepto básico, quizás el más importante de toda la TC, es el de modelo de cálculo, que formaliza el concepto intuitivo de algoritmo. Su presentación se suele hacer en la asignatura Teoría de Autómatas y Lenguajes Formales (TALF), generalmente en segundo año del primer ciclo de Informática.

Tradicionalmente se utiliza la MT como modelo de cálculo. Sin embargo, esta elección no se apoya en sus bondades pedagógica. Los principales motivos de su uso extendido son: haber sido uno de los cuatro que se definieron casi simultáneamente en 1936 (λ-cálculo, funciones μ-recursivas, MT y máquina de Post), el enlazar adecuadamente con los autómatas descritos en la primera parte de TALF, gozar de una gran popularidad, incluso en ambientes no informáticos y en la mayor parte de la bibliografía habitual, aunque la razón de mayor peso es el imperativo legal.

Todo el que haya impartido TALF con la MT como modelo de cálculo sabe de sus inconvenientes docentes. Por una parte la MT queda muy alejada del conocimiento que posee el alumno de Informática en segundo año. Incluso cuando, con esfuerzo, comprende el modelo, lo ubica alejado del resto de los contenidos de la carrera, sin mucha conexión con la misma. Por otra parte las demostraciones con MT, debido a su "bajo nivel", suelen ser largas y engorrosas, lo cual dificulta aún más una buena asimilación por parte del alumno.

Para intentar solucionar estos inconvenientes nosotros hemos elegido como modelo de cálculo para impartir TALF un lenguaje estructurado simple, el lenguaje WHILE. Este modelo es, por una parte, más cercano a los conocimientos que ya posee el alumno, por lo que es más fácil su asimilación, por otra, las demostraciones de las proposiciones y teoremas de la TC son mucho más cortas y claras con él. Este modelo se introdujo en una asignatura optativa de segundo ciclo hace unos diez cursos y se utiliza en TALF

desde hace cinco. Nuestra experiencia confirma totalmente lo adecuado de dicha elección.

En este trabajo presentamos este modelo y justificamos su uso. Para ello a continuación mostramos la evolución de los modelos formales de cálculo. Después dedicamos un apartado a presentar el modelo "lenguaje WHILE" que proponemos. En el apartado cuatro mostramos un ejemplo de demostración con dicho modelo, para terminar con un quinto de discusión.

2. Evolución de los modelos formales de cálculo

Tras los modelos iniciales de Church, Kleene, Turing y Post que se definieron casi simultáneamente en 1936, han aparecido otros con diferentes orientaciones. Veamos cómo la aparición de estos modelos ha ido paralela al desarrollo de los ordenadores y los lenguajes de programación. En primer lugar citamos el modelo URM definido por Shepherdson y Sturgis [12] y usado como modelo en [7], que dió origen al modelo RAM de amplio uso [1]. En este modelo se evitan los tediosos recorridos por la cinta de la máquina de Turing, mediante la introducción de registros en los que se almacenan datos, sin importar su tamaño. Junto con esta forma de almacenamiento está un adecuado juego de instrucciones que incluye sumar uno, restar uno, copiar de un registro a otro, salto incondicional y salto condicional. Los modelos derivados de URM tienen un juego de instrucciones que se parece mucho a un ensamblador.

El siguiente paso en los modelos fue pasar de registros a variables, de forma que una "máquina" pasa a tener aspecto de programa. Uno de estos lenguajes se puede ver en libros mas recientes como [3,4]. Es un lenguaje no estructurado con variables de entrada, variables de uso interno y variables de salida.

Volviendo a la evolución de los lenguajes, en 1966 se publica el trabajo [2] sobre la posibilidad de escribir programas con solo dos reglas de formación: secuencia y repetición (aunque habitualmente se habla de tres reglas, la selección es deducible de la repetición). Señalar que en este trabajo se usan diagramas de flujo, que se traducen de forma directa a sentencias de un lenguaje de programación.

Este trabajo junto con otros desarrollos prácticos dio impulso a la programación estructurada y, en el ámbito de la Informática Teórica, surgieron modelos de cálculo basados en programación estructurada en los trabajos de Meyer y Ritchie [8,11]. El lenguaje derivado de los trabajos anteriores tiene sentencias de asignación y una única estructura de control: el bucle indefinido con control al principio, demostrándose su equivalencia con los demás modelos. Si bien esta propuesta inicial fue realizada hace unos 35 años con una sentencia de bucle definido, no queda claro en la bibliografía cuando se produce la ampliación del lenguaje con un bucle indefinido, que solo queda recogida en libros.

Otra versión moderna en una dirección similar se puede encontrar en [5]. No obstante el lenguaje utilizado se basa en listas y las sentencias básicas son de tipo LISP (cabeza y cola), lo cual se aleja de la formación de los alumnos de segundo curso.

Si tomamos la perspectiva de las funciones calculadas por cada modelo, podemos observar como una adecuada secuencia de modelos van refinando la clasificación de las funciones.

Las máquinas de Turing dan origen a las funciones Turing-calculables y el modelo no permite refinar esa clase. Esas funciones, mediante el modelo de Kleene [6], se refinan en recursivas primitivas y μ-recursivas. Consideremos ahora un lenguaje estructurado. A partir de una sentencia WHILE (que corresponde a la minimización o a la i-iteración [13]) se puede definir una sentencia FOR (que corresponde a la recursión primitiva o a la exponenciación [12]). Se podría pensar en el nivel de anidamiento de sentencias WHILE, pero esto no tiene interés pues ya sabemos, por el teorema de la forma normal de Kleene, que cada función calculable se puede expresar mediante un programa que se escribe usando asignaciones, estructuras FOR y una sola estructura WHILE. Pero sí tiene interés considerar el nivel de anidamiento de estructuras FOR, y esto supone un refinamiento de la clase de las funciones recursivas primitivas mediante una jerarquía infinita LOOP0, LOOP1, ..., LOOP*n*, ...

Veamos en el siguiente apartado una descripción del lenguaje WHILE tal y como aparece en [10].

3. El lenguaje WHILE

El lenguaje WHILE contiene sentencias para calcular la constante cero, el incremento, el decremento y la asignación, además de una estructura para el bucle indefinido. Dicho bucle funciona como un bucle de tipo *while*, por lo que los programas definidos en este lenguaje se denominan programas WHILE. Sólo se dispone de estas sentencias simples como primitivas, por lo que otras operaciones como la multiplicación, división, etc. serán implementadas como programas WHILE. Señalar también que este lenguaje no contiene instrucciones de entrada/salida, y que el único tipo de datos utilizado por un programa WHILE es el número natural, que se almacena en variables. Cada programa representa una función matemática.

Definimos la sintaxis del lenguaje WHILE en notación *BNF* de la siguiente forma:

Sea $G = (N, T, R, S)$ una gramática donde

T = { DO, OD, :, =, WHILE, ≠, +, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, X, ; }

N = { *programa*, *asignacion*, *sent*, *sentencias*, *identificador*, *numero* }

El conjunto de reglas R es:

1 S=<*programa*> ::= {(*n*,*p*,<*sentencias*>)|*n*,1≤*p*}
2 <*sentencias*> ::= <*sent*>
3 | <*sentencias*>;<*sent*>
4 <*sent*> ::= <*asignacion*>
5 | WHILE<*identificador*>≠0DO<*sentencias*>OD
6 <*asignacion*> ::= <*identificador*>:=0
7 | <*identificador*>:=<*identificador*>
8 | <*identificador*>:=<*identificador*>+1
9 | <*identificador*>:=<*identificador*>–1
10 <*identificador*> ::= X<*numero*>
11 <*numero*> ::= (1|2|3|4|5|6|7|8|9) {0|1|2|3|4|5|6|7|8|9}

Con esta sintaxis: si X1 y X2 son nombres de variables, entonces las expresiones X1:=0, X1:=X1+1, X1:=X1-1 y X1:=X2 son <sentencias> WHILE; si P y Q son <sentencias> WHILE, entonces la secuencia P;Q es válida; siéndolo también la expresión WHILE X1≠0 DO P OD. En este bucle se diferencian la cabecera (WHILE X1≠0 DO), el cuerpo P y la cola (OD) del bucle.

Un programa WHILE es una terna (*n*,*p*,P), consistente en dos números naturales y una secuencia de sentencias P.

Comencemos con la semántica del lenguaje diciendo que las variables que participan en la secuencia P son las variables del programa. El número p especifica que {X1,...,Xp} son los nombres de dichas variables del programa, y el número n especifica cuántas de estas variables son de entrada, es decir X1,...,Xn. La única variable de salida es X1. Cada línea del programa (*n*,*p*,P) contiene: una asignación, una cabecera de bucle *while* o bien una cola de bucle *while*.

Vamos a definir la función $f : \mathrm{N}^n \to \mathrm{N}$ calculada por un programa WHILE (con n variables de entrada). Partiendo de $(a_1, ..., a_n) \in \mathrm{N}^n$ llegaremos a determinar $f(a_1, ..., a_n)$:

Sea $(a_1, ..., a_n) \in \mathrm{N}^n$. A partir de estos valores de entrada, el vector de estado inicial es el vector $(a_1, ..., a_n, 0, ...,0) \in \mathrm{N}^p$ formado por esos valores y ceros (correspondientes a las variables que no son de entrada).

El cálculo comienza sobre este vector de estado y en la primera línea del programa, es decir, la iniciación de variables no se considera parte del cálculo. El cálculo procede secuencialmente de acuerdo con la estructura del programa y el significado de cada sentencia, así, un programa WHILE de la forma "P;Q" supone la ejecución primero de P y después (manteniendo los valores de las variables como resultan de ejecutar P) de Q. Como hemos comentado en la sintaxis hay dos tipos de sentencias: asignación y bucle.

La ejecución de una sentencia de asignación (X3:=0, X4:=X5, X2:=X3+1, X1:=X4-1) transforma el valor actual del vector de estado reemplazando una componente por cero, por el valor de otra componente, o bien por el sucesor o predecesor de otro componente, respectivamente. Esta transformación se considera un paso de cálculo. Por otro lado, la secuencia P en un bucle *while* de la forma "WHILE X1≠0 DO P OD" se ejecutará mientras la variable X1 no almacene el valor cero, si tal condición tiene lugar.

Si el programa nunca alcanza la última línea (condición de terminación del cálculo) entonces

calcula indefinidamente, en cuyo caso decimos que, para ese vector de entrada, la función diverge.

Por el contrario, si ejecuta la última línea, entonces el cálculo acaba. El valor calculado (la salida del programa) es el valor almacenado en el primer componente del vector de estado (`X1`). Ese es el valor $f(a_1, ..., a_n)$ de la función calculada por el programa.

Diremos que dos programa son equivalentes se calculan la misma función.

Ilustramos el modelo de programas WHILE con un programa equivalente al que calcula la sentencia IF (con su significado normal), de la forma `IF X≠0 THEN` P `FI`:

```
WHILE X1≠0 DO
     P;
     X1:=0;
OD
```

4. Una demostración

Usando el lenguaje WHILE, desarrollamos en este apartado, una función no computable. Para ello seguiremos el trabajo [9], completado con una demostración sencilla de la insolubilidad del problema de la parada.

Vamos a considerar aquí la clase de programas con una sola entrada (`X1`).

Definición: Sea (*1*,*p*,P) un programa WHILE. Se define la longitud de P como su número de líneas, contando cada asignación y cada cabeza de bucle como una línea.

Por ejemplo, el siguiente programa (*1*,*2*,P) tiene longitud 4.

```
X2:=X1;
WHILE X2≠0 DO
     X1:=X1+1;
     X2:=X2-1;
OD;
```

Definición: La función castor afanoso (*busy-beaver*), Σ, se define como sigue: Para cada *n*, Σ(*n*) es el mayor valor que puede almacenarse en la variable `X1` usando un programa de longitud exactamente *n* que acaba, cuando comienza con `X1=0`.

Explicación intuitiva: dado un valor para *n*, p.e. 15,

- escribir todos los programas de longitud 15,
- seleccionar los que acaban y poner como entrada `X1=0`,
- ver su salida y tomar el valor mayor.

Teorema: La función castor afanoso (Σ) es no calculable, es decir, no existe un programa WHILE que la calcule.

La demostración de este teorema precisa dos resultados previos:

Proposición 1: Para cada *n*, $\Sigma(n+1) > \Sigma(n)$.

Demostración: Sea P el programa que calcula Σ(*n*), esto es, P es de longitud *n* y con entrada `X1=0`, acaba y produce la mayor salida.

Sea P' el programa siguiente:

```
P
X1:=X1+1
```

Este programa tiene longitud *n*+1 y produce como salida Σ(*n*)+1, y puede haber algún otro programa de longitud *n*+1 que produzca una salida mayor, luego $\Sigma(n+1) \geq \Sigma(n)+1 > \Sigma(n)$.

Proposición 2: Si *f* es calculable mediante un programa de longitud *k*, entonces para cada *n* $\Sigma(n+k) \geq f(n)$.

Demostración: Sea P el programa de longitud *k* que calcula *f*. Sea P'el siguiente programa:

```
X1:=X1 + 1;
… (n veces)
X1:=X1 + 1;
P
```

Este programa tiene longitud *n*+*k*; con entrada `X1=0`, tras las *n* primeras líneas `X1` vale *n*, y tras la ejecución de P, que calcula *f*, produce como salida *f*(*n*). Luego tenemos un programa de longitud *n*+*k* cuya salida es *f*(*n*) y puede haber algun otro de esa longitud que produzca una salida mayor, luego $\Sigma(n+k) \geq f(n)$.

Ya estamos en condiciones de demostrar del teorema:

Supongamos que Σ es calculable. $f(x) = 2x$ es calculable. La composición de funciones calculables es calculable. Entonces $ß(n) = Σ(2n)$ será calculable mediante un programa P' de longitud k. Aplicando la proposición 2 a $ß(n)$, tenemos que para cada n: $Σ(n+k) \geq ß(n) = Σ(2n)$.

Tomando $n=k+1$, resulta $Σ(2k+1) \geq Σ(2k+2)$, que entra en contradicción con la proposición 1 (crecimiento estricto de la función Σ).

Veamos ahora la irresolubilidad del problema de la parada a partir de la no calculabilidad de la función castor afanoso.

Supongamos que el problema de la parada es resoluble. Para cada longitud n dada, solo hay un número finito de programas distintos. Como suponemos que el problema de la parada es resoluble, seleccionar los que acaban y obtener su salida. La mayor salida es Σ(n). Con lo cual la función castor afanoso sería calculable, en contra de lo que acabamos de demostrar; por tanto, el problema de la parada es recursivamente irresoluble. Nótese que una demostración directa de la irresolubilidad del problema de la parada (es decir, su predicado asociado es indecidible) requiere una gödelización de los programas y la correspondiente indexación de funciones.

5. Discusión

En este trabajo hemos analizamos la fuerte dependencia del modelo de cálculo en la docencia de la TC. La MT es el modelo más extendido en la programación de TALF. Opinamos que principalmente por razones históricas, dadas las dificultades que plantea su uso en el aula. Basándonos en nuestra experiencia docente destacamos dos puntos débiles: (1) el bajo nivel del modelo, y (2) su complejidad inherente.

Respecto a (1), el bajo nivel de la MT tiene dos consecuencias principales: (a) se separa excesivamente de los lenguajes de programación de alto nivel, lo que a menudo impide al alumno relacionar las propiedades de los modelos de cálculo con otras áreas de conocimiento. Conceptos tan cercanos como por ejemplo la MT universal y un compilador no suelen asociarse espontáneamente; (b) para asimilar el funcionamiento de la MT, el alumno frecuentemente se desvía en exceso de su naturaleza matemática y prefiere verlo como un sistema mecánico, situándolo más cerca de un autómata industrial que de un modelo formal. En relación a (2), la complejidad de la MT se refleja principalmente en las demostraciones de los principios incluidos en la TC. Demostraciones de teoremas como la equivalencia con otros modelos, la no calculabilidad del castor afanoso o el problema de la parada, o aún la definición de la MT universal, resultan largas y elaboradas en exceso, lo que perjudica tanto el seguimiento como la asimilación del teorema.

Los modelos basados en lenguajes estructurados solucionan en parte los problemas de la MT: la asimilación del modelo resulta fácil, ya que presenta muchos de los rasgos de los lenguajes de programación que el alumno conoce, lo que garantiza la conexión con muchos conceptos afines de otras áreas; por otra parte, la demostración de los teoremas se simplifica bastante, quedando mucho más legible y en una forma muy intuitiva, lo que reduce los tiempos de explicación y favorece la asimilación, ya que hacemos uso de un buen número de nociones de las que ya dispone el alumno, como podrían ser la invocación de subrutinas o el funcionamiento de un condicional múltiple.

La introducción del modelo WHILE presenta algunas características adicionales. Por un lado, el modelo es entendido inicialmente por el alumno como un lenguaje de programación de ordenadores, lo que constituye un inconveniente, ya que nos interesa resaltar su naturaleza matemática y su papel como modelo formal. No obstante, esta faceta puede ser utilizada para alcanzar tres objetivos adicionales: (1) la comprensión de que, en realidad, cuando se programa se está demostrando formalmente la calculabilidad de la función definida en las especificaciones; (2) la noción de que los lenguajes de programación actuales descansan en unas primitivas de cálculo muy elementales, y que cada elemento del lenguaje tiene una función de apoyo a la programación, sin alterar su capacidad de cálculo; concretamente, la demostración de que una versión extendida de WHILE, con muchas de las características de un lenguaje evolucionado, se obtiene a partir de la forma básica, enseña al alumno qué es en realidad esencial en la

definición del lenguaje; simultáneamente (3) se justifica el uso de algoritmos en pseudocódigo para demostrar teoremas, práctica muy extendida en algunas materias, especialmente en la primera mitad de la asignatura TALF. Por otro lado, WHILE permite una mejor comprensión de la godelización de programas y por tanto de la indexación de funciones, pudiéndose además ilustrar con ejemplos de desarrollo completo la obtención de índices para funciones familiares.

En definitiva, hemos presentado una alternativa a la MT como modelo de cálculo dentro de la docencia de TALF. Dicho modelo está al final de la evolución teórica de los modelos de cálculo, basándose en un lenguaje estructurado simple, el lenguaje WHILE. Este modelo presenta ventajas para la docencia. Por una parte es más cercano a los conocimientos que posee el alumno en el momento curricular en que se imparte, percibiéndose además como más cercano al resto de los contenidos de la carrera de Informática. Por otra parte hace posible, como hemos mostrado, realizar demostraciones de la TC mucho más cortas y legibles. Además, permite una inclusión fácil del no determinismo, a partir de una sentencia "either" de ramificación binaria.

Por todo ello, basándonos en nuestra experiencia positiva de varios años de utilización, aconsejamos, que si bien se explique a los alumnos en qué consiste la MT por ser el modelo más conocido, para el desarrollo y las demostraciones de la TC se utilice el lenguaje WHILE como modelo teórico de cálculo.

Enlazando con la Teoría de la Complejidad señalamos que el modelo MT es adecuado para el estudio de los recursos espacio y tiempo. El lenguaje WHILE no es adecuado para el estudio del recurso espacial de forma directa (a una variable se le puede asignar un valor arbitrario) y habría que considerar el logaritmo del mayor valor almacenado en una cualquiera de las variables del programa. Sin embargo, sí es adecuado de forma directa para el estudio del recurso tiempo, ya que el aumento de complejidad en tiempo al pasar de un programa WHILE a su correspondiente MT es solo polinómico, operación para la que son estables la mayoría de las clases de complejidad, y en particular las que usualmente se introducen en ese momento de la formación del alumnos, como son P y NP.

Referencias

[1] Aho, A.V., Hopcroft J.E. & Ullman, J.D. *The Design and Analysis of Computer Algorithms*. Addison-Wesley. 1974.

[2] Böhm C. & Jacopini G. Flow Diagrams, Turing Machines and Languages with only two Formation Rules. *Communications of the ACM*, **9**, 5, pp. 366-371, 1966.

[3] Davis M. & Weyuker E. *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science*. Academic Press, 1983.

[4] Davis, M.D., Sigal, R. & Weyuker, E.J. *Computability, complexity and languages*. Academic Press. 1994.

[5] Jones, N. *Computability and Complexity. From a Programming Perspective*. MIT Press, 1997.

[6] Kleene S.C. General Recursive Functions of Natural Numbers. *Mathematicsche Annalen*, **112**, 5, pp. 727-742, 1936.

[7] Machtey, M. & Young, P. *An introduction to the General Theory of Algorithms*. Elsevier North-Holland Inc. 1978.

[8] Meyer A.R. & Ritchie D.M. The complexity of loop programs. *Proc. ACM Nat. Meeting*, pp. 465-469, 1976.

[9] Morales-Bueno, R. *Noncomputability is easy to understand.* Comm. ACM, **38**, pp. 116-117, 1995.

[10] Morales-Bueno, R.; Fortes, I.; Mora L. & Triguero, F. *Two classical theorems revisited*. Bull. EATCS, **71**, pp. 204-215, 2000.

[11] Ritchie, D.M. *Program structure and computational complexity*. Doctoral Dissertation. Harvard University. 1968.

[12] Shepherdson, J. & Sturgis, H. Computability of recursive functions. *Journal of the ACM*, **10**, 2, pp. 217-255. 1963.

[13] Sommerhalder R. & van Westrhenen S.C. *The Theory of Computability: Programs, Machines, Effectiveness and Feasibility*. Addison-Wesley. 1988.