Autómatas de Pila y Máquinas de Turing Estructurados

Jairo Rocha
Dpto. Ciencias Matemáticas e Informática
Universidad de las Islas Baleares
07122 Palma de Mallorca
e-mail: jairo@uib.es

Resumen

Presentamos una sintaxis simple para definir máquinas de Turing y autómatas de pila indeterministas que evita el uso de estados y, en su lugar, usa listas de instrucciones y bucles. De esta manera, una máquina de Turing es un programa con instrucciones y bucles estructurados, y un autómata de pila es un programa con una pila. Se persiguen dos objetivos: el primero, acercar los resultados de autómatas a la práctica diaria de programación estructurada y, el segundo, simplificar las demostraciones apelando a la intuición del estudiante conocedor del potencial de la programación moderna.

El resultado es un mayor interés y confianza de los estudiantes en el estudio de estos temas.

1. Introducción

Aunque el concepto de máquina que ejecuta transiciones de estados y datos es la esencia de la informática, la sintaxis del lenguaje de programación que definió Turing no se ha utilizado nunca en los programas reales. El motivo por el cual hay que esforzarse para encontrar libros de texto de autómatas formales que no usen máquinas de Turing es por su formalismo simple que hace las demostraciones más cortas que si se hicieran sobre lenguajes más complejos.

El hecho de que los programas de ordenador sean el pan de cada día de los informáticos tiene diversas consecuencias sobre la forma como se deberían presentar los contenidos sobre programas formales. Primero, el objetivo no es desarrollar estrategias de programación en el lenguaje de las máquinas de Turing, o, en cualquier otro lenguaje, porque el interés no es la programación. Segundo, con el fin de no tener que escribir programas largos, complicados y poco interesantes para resolver ciertos problemas de codificación, traducción o ejecución de programas, simplemente se dice que es posible construir programas formales que los resuelvan y se le pide al lector que se lo crea, todo basándose en sus conocimientos de programación. En tercer lugar, una vez introducida una sintaxis de los programas formales, se amplía, con el fin de que el estudiante pueda usar un formalismo de más alto nivel (justificándolo pero sin demostrar en detalle su equivalencia). Y, finalmente, la presentación será guiada por el objetivo de establecer límites de la computación y no por la programación en sí misma.

Por lo tanto, evitando demostraciones tediosas que se saben ciertas también empíricamente porque hacen que los ordenadores funcionen cada día, y usando una sintaxis moderna, es posible dar unos contenidos teóricos fácilmente asimilables a los programas de hoy en día en el área de indecidibilidad.

De igual forma, el uso de estados en los autómatas de pila recuerda el uso de los `go-to`s y se hace necesario un esfuerzo para traducir un autómata de pila a un programa moderno. Por lo tanto, se sugiere definir autómatas de pila como programas restringidos y con una pila, heredando así, el estructuralismo de los programas, y acercando los teoremas y los resultados a su uso concreto. Este enfoque junto con otros sobre la enseñanza de la verificación de autómatas se desarrolla en nuestro libro [4].

2. Programas formales

Comenzamos definiendo un programa simple como una secuencia de instrucciones que actúan sobre un puntero de un vector de símbolos que tiene tantas posiciones como sea necesario (que corresponde al cabezal de la cinta semi-infinita de una máquina de Turing).

2.1. Definiciones

Las instrucciones fundamentales son leer el contenido del vector en la posición del puntero, escribir en esta posición y cambiar la posición del puntero a izquierda o a derecha. Las instrucciones compues-

tas son el condicional y la repetición de instrucciones que dependen de una condición. Las condiciones son la igualdad de símbolos o su negación. En una ejecución, el vector contiene al principio la entrada compuesta por símbolos y termina con la marca del fin de entrada que es el símbolo distinguido $\#$ que nunca forma parte de la entrada y al que llamaremos *blanco*. Estos símbolos más otros símbolos predefinidos pueden aparecer en el programa. Además del vector de símbolos, hay una variable que contiene un símbolo y que se utiliza para dar el resultado del reconocimiento: un 1 significa que la palabra de entrada es aceptada.

Definición 1 (Programa). Sea Σ un alfabeto (de entrada) finito que no contiene $\#$ y Γ un alfabeto (del vector) finito de símbolos que contiene $\Sigma \cup \{0, 1, \#\}$. Un *programa* P es $P = (\Sigma, \Gamma, p)$ donde p es una cadena producida por la gramática $(V, T, R, Prog)$, donde

$$\begin{aligned} V = \ & \{Prog, Bloq, Ins, Cd, Cdno, Rept, \\ & Term, Exp\} \\ T = \ & \{\texttt{acepta}, \texttt{der}, \texttt{ent}, \texttt{escr}, \texttt{izq}, \texttt{haga}, \texttt{fin}, \\ & \texttt{fmientras}, \texttt{fsi}, \texttt{id}, \texttt{leer}, \texttt{mientras}, \\ & \texttt{programa}, \texttt{si}, \texttt{si\_no}, \sigma, `, `, (,), =, /=\} \end{aligned}$$

y el conjunto de producciones R es

$$\begin{aligned} Prog &\rightarrow \texttt{programa}\ id\ Bloc\ \texttt{fin} \\ Bloq &\rightarrow Ins\ Bloq \mid \lambda \\ \\ Ins &\rightarrow \texttt{der(ent)} \mid \texttt{izq(ent)} \mid Cd \mid Rept \\ &\quad \mid \texttt{escr(ent,}Term\texttt{)} \\ &\quad \mid \texttt{escr(acepta,}Term\texttt{)} \\ Cd &\rightarrow \texttt{si}\ Exp\ \texttt{ent}\ Bloq\ Cdno\ \texttt{fsi} \\ Cdno &\rightarrow \texttt{si\_no}\ Bloq \mid \lambda \\ Rept &\rightarrow \texttt{mientras}\ Exp\ \texttt{haga} \\ &\quad Bloq\ \texttt{fmientras} \\ Exp &\rightarrow Term = Term \mid Term /= Term \\ Term &\rightarrow \sigma \mid \texttt{leer(ent)} \mid \texttt{leer(acepta)} \end{aligned}$$

donde id puede ser reemplazada por una palabra cualquiera que identifica el programa y σ por un elemento del alfabeto del vector, Γ.

La anterior definición hace uso de las gramáticas, lo que da un factor de motivación adicional a otro tema de la asignatura.

A cada instrucción básica, a cada condición de una instrucción condicional o de repetición, a cada `si_no, fsi, fmientras` y `fin` se le puede asignar un único número de orden en la lista de instrucciones del programa. Este número determina la *posición* de una única instrucción en el programa.

Ejemplo 2. Consideremos el programa con alfabetos $\Sigma = \{0, 1\}$ y $\Gamma = \{0, 1, \#\}$ y las siguientes instrucciones con las correspondientes posiciones:

```
    programa  pordos(ent)
1   mientras leer(ent) /= # haga
2       der(ent)
3   fmientras
4   izq(ent)
5   mientras leer(ent) /= # haga
6       si leer(ent) = 0 ent
7          der(ent)
8          escr(ent,0)
9       si_no
10         der(ent)
11         escr(ent,1)
12      fsi
13      izq(ent)
14      izq(ent)
15  fmientras
16  der(ent)
17  escr(ent,0)
18  escr(acepta,1)
19  fin
```

El programa anterior multiplica un número binario por 2 si los bits menos significativos están a la izquierda.

En un momento determinado de la ejecución de un programa, la posición de la próxima instrucción a ejecutar es conocida. La instrucción cambia la posición del puntero, el contenido del vector, el contenido de la variable o una expresión es evaluada. Después, la posición de la próxima instrucción queda conocida y el proceso se repite. En resumen, la estructura de datos y las instrucciones básicas de los programas son las mismas que las de una máquina de Turing.

Definición 3 (Configuración). Una *configuración* de un programa P sobre un alfabeto Γ es una cuádrupla

$$(n, \alpha, i, r) \in \mathbb{N} \times \Gamma^* \times \mathbb{N} \times \Gamma$$

formada por la posición n en el programa, el contenido α del vector, la posición i del puntero y el valor r de la variable `acepta`.

En general, identificamos todas las configuraciones que sólo difieren en el número de blancos a la derecha de la segunda componente. La posición cero del vector siempre tiene un # que no se puede cambiar.

Dado el programa P y la configuración (n, α, i, r) es posible saber la próxima instrucción a ejecutar, el contenido no blanco del vector, la posición del puntero en el vector y el valor de la variable `acepta`. La descripción intuitiva de la *semántica* de un programa dada antes se traduce formalmente en una relación $\vdash$ de derivación directa sobre el conjunto de sus configuraciones, que no daremos aquí por falta de espacio. Indicamos con $\vdash^*$ la clausura reflexiva y transitiva de $\vdash$ y, cuando $C \vdash^* C'$, decimos que C' *deriva* de C.

Ejemplo 4. Consideremos el programa con alfabeto $\Gamma = \{0, 1, \#\}$ y las siguientes instrucciones con las correspondientes posiciones:

```
    programa ceroporuno(ent)
1   mientras leer(ent)=1 haga
2       der(ent)
3   fmientras
4   mientras leer(ent)=# haga
5       der(ent)
6   fmientras
7   si leer(ent)=0 ent
8       escr(ent,1)
9       der(ent)
10  fsi
11  mientras leer(ent)=1 haga
12      der(ent)
13  fmientras
14  si leer(ent)=# ent
15      izq(ent)
16  fsi
17  mientras leer(ent)=1 haga
18      izq(ent)
19  fmientras
20  si leer(ent)=# ent
21      der(ent)
22      escr(acepta,1)
23  fsi
24  fin
```

Entonces, por ejemplo, tenemos la derivación siguiente:

$$\begin{aligned}(1,11,1,0) &\vdash (2,11,1,0) \vdash (3,11,2,0)\\ &\vdash (1,11,2,0) \vdash (2,11,2,0)\\ &\vdash (3,11,3,0) \vdash (1,11,3,0)\\ &\vdash (4,111,3,0) \vdash (5,111,3,0)\\ &\vdash (6,111,4,0) \vdash (4,111,4,0)\\ &\vdash (5,111,4,0) \vdash \cdots\end{aligned}$$

que nunca se para.

Con esta definición de derivación podemos definir un programa que siempre se para, un lenguaje aceptado por un programa y lenguajes decidibles y semidecidibles.

2.2. Programas multivector

Con el fin de facilitar el uso de los programas, en esta sección se introduce un formalismo más amplio para los programas que permite programar con más facilidad: los *programas multivector*.

Como su nombre lo indica, los programas multivector disponen de diversos vectores con un rango tan amplio como se necesite, cada uno de los cuales tiene su propio puntero de lectura-escritura. El programa puede leer cada uno de los vectores, escribir un símbolo en cada uno de los vectores y desplazar el puntero de cada vector de forma independiente. Por esto, el primer parámetro de las operaciones sobre vectores es variable e identifica al vector.

Además, el programa tendrá un encabezado de la forma

```
programa nomprog(ent1,ent2,...,entm)
v1,v2,...,vn:vector
```

en el que `ent1` es el nombre del vector donde se encuentra la primera palabra de entrada, `ent2`, donde se encuentra la segunda, etc., y `v1, v2, ...` son nombres de vectores internos. El número de vectores del programa es el número de vectores de entrada más el número de vectores internos.

Ejemplo 5. Queremos dar un programa que, recibiendo una palabra $w \in \{0, 1\}^*$ como entrada, escriba la palabra ww en el mismo vector de entrada.

Con este fin, usaremos un programa de dos vectores que primero copiará el contenido del vector de entrada al segundo vector, y a continuación copiará el contenido del segundo vector a la derecha del contenido del primer vector. El programa es

```
programa doblar(w)
v: vector
mientras leer(w) /= # haga
    escr(v,leer(w))
    der(w)
    der(v)
fmientras
izq(v)
mientras leer(v) /= #  haga
    izq(v)
fmientras
der(v)
mientras leer(v) /= # haga
    escr(w,leer(v))
    der(w)
    der(v)
fmientras
izq(w)
mientras leer(w) /= # haga
    izq(w)
fmientras
der(w)
escr(acepta,1)
fin
```

Notemos que esta solución es conceptualmente más simple que la que podríamos dar si únicamente usáramos un vector.

Ejemplo 6. Más adelante usaremos el siguiente programa que copia el contenido del primer vector al segundo y aparca los punteros (los deja en la primera casilla):

```
programa copiar(ent,copia)
  mientras leer(ent) \= # haga
    escr(copia,leer(ent))
    der(ent)
    der(copia)
  fmientras
  izq(ent)
  izq(copia)
  mientras leer(ent) \= # haga
    izq(ent)
    izq(copia)
  fmientras
  der(ent)
  der(copia)
fin
```

Como antes, se puede definir una configuración que en este caso contendrá información sobre cada uno de los vectores.

Para demostrar que los lenguajes que pueden ser reconocidos con programas multivector son exactamente los semidecidibles, es necesario traducir un programa en uno de sólo un vector. La traducción consiste en definir una estructura de datos sobre un vector que pueda simular el conjunto finito de vectores de un programa multivector dado. Aquí preferimos apelar a la intuición y la experiencia en programación de los estudiantes, y pedirles que se convenzan que es posible definir instrucciones para un vector que traduzca cada instrucción del programa multivector en instrucciones sobre el vector único (esta traducción es automática). Además, el nuevo programa de un solo vector debería dejar, al detenerse, un contenido sobre el vector análogo al que deja el programa original sobre el primer vector y los dos deberían aceptar exactamente las mismas entradas.

La demostración formal de la equivalencia entre los programas multivector y los de un único vector es muy larga, y además creemos que poco interesante, así que se omite. De hecho, se trata de un ejercicio de representación de una estructura en otra más simple, y con seguridad el estudiante tendrá la oportunidad de hacer ejercicios de este tipo más útiles en un curso de estructuras de datos. El estudiante interesado que no quiera, o no pueda, hacer la demostración rigurosa solo, la encontrará en muchos libros de autómatas y lenguajes formales como, por ejemplo, [1].

2.3. Programas que llaman otros programas

Para simplificar los programas, se utiliza de manera sistemática un recurso que nos permite el uso de un programa dentro de otro programa. Este recurso se corresponde, en el mundo de los lenguajes de programación, a cuando una rutina llama otra rutina; la definición dada aquí es más simple.

Suponemos que el programa llamado `rutina2` en la posición n tiene la llamada

```
rutina1(v1,...,ve)
```

donde `v1,...,ve` son vectores diferentes del programa `rutina2`. Si el programa `rutina1` tiene el encabezado

```
programa rutina1(x1,...,xe)
  y1,...,ym:vector
```

entonces la *llamada* es una abreviación de insertar en la posición n todas las instrucciones (excepto el `fin`) del programa `rutina1` donde previamente se han reemplazado todas las ocurrencias de cada una de las variables `x1,...,xe` por `v1,...,ve`, respectivamente; además, todas las ocurrencias de `y1,...,ym` han de ser reemplazadas por otro nombre si hay un vector de `rutina2` que tenga el mismo nombre; finalmente, se supone que los vectores `y1,...,ym` (con su nuevo nombre si es necesario) son declarados al programa `rutina2`.

En realidad, con la definición anterior, las llamadas no se hacen en el momento de la ejecución sino que representan un cambio en el programa que hace la llamada. Además, todos los vectores de entrada del programa llamado son compartidos por los dos programas; la variable `acepta` también es compartida.

2.4. Lenguajes indecidibles

En esta sección mostraremos cómo se definen problemas que no se pueden resolver mediante programas.

Como escribiremos programas que tienen por entrada otros programas y ésta tiene que ser una palabra binaria, tenemos que codificar los programas mediante palabras sobre $\{0, 1\}$; notemos que los programas de ordenador reales ya están codificados en binario usando el código ASCII. Se usa aquí la *misma* codificación y $<P>$ denota la palabra binaria que representa el programa P.

La codificación de expresiones por números fue un concepto muy nuevo hace setenta años, y tiene el origen en la codificación de Gödel de fórmulas aritméticas por medio de números. Hoy en día, no sólo es que los programas son códigos, sino que estamos acostumbrados a que toda la información sobre personas, aviones, relaciones, imágenes, sonido, etc., esté codificada por números, porque de otra manera no se podría introducir en el ordenador.

Tenemos el resultado siguiente que muestra que no es posible decidir si la ejecución de un programa dado sobre una palabra dada se parará o no, es decir, el problema de la parada no es decidible.

Teorema 7. *El* lenguaje de la parada

$$L_p = \{<P> \,\&w \mid P \text{ se para con entrada } w\}$$

no es decidible.

Demostración. La idea de la demostración es suponer que existe un programa llamado `sePara(p,w)` (el primer parámetro son los bits antes del &, y el segundo, los de después) que se para siempre y reconoce el lenguaje de la parada, y emplearlo para construir un programa `prueba(q)` con el que `sePara` no funciona bien, lo que nos lleva a una contradicción.

Sea, entonces, el siguiente programa:

```
programa prueba(q)
     q1:vector
  copiar(q,q1)
  sePara(q,q1)
  si leer(acepta) \= 1 llavors
     escr(acepta,1)
  si_no
     mientras 0 = 0 haga
     fmientras
  fsi
fi
```

Suponemos que llamamos `prueba(<prueba>)` (es decir, `q` contiene $<$`prueba`$>$ como entrada).

Si `sePara(<prueba>,<prueba>)` acepta la entrada, entonces `prueba(<prueba>)` entra en un bucle infinito. Esto quiere decir que `prueba(<prueba>)` no se para y, por tanto, `sePara(<prueba>,<prueba>)` no debería haber aceptado.

Entonces, `sePara(<prueba>,<prueba>)` no acepta la entrada pero se para ya que siempre se para; entonces, `prueba(<prueba>)` acepta la entrada y se para, y `sePara(<prueba>,<prueba>)` debería haber aceptado.

Como ambos casos son imposibles, el programa `sePara(p,w)` no puede existir. □

3. Autómatas de pila

Informalmente, un autómata de pila es un programa con un vector, `ent`, para la entrada y un vector, `pila`, para hacer operaciones con una pila. Los autómatas de pila son programas restringidos: primero, el vector de entrada no puede ser cambiado ni el puntero se puede mover a la izquierda; segundo, el vector de la pila puede ser manipulado únicamente con las operaciones definidas más adelante que impiden leer las casillas a la derecha del puntero.

Los tres objetivos para enseñar autómatas de pila son: aprender a usar autómatas de pila, hacer un reconocedor de programas que es útil a la hora de escribir programas que *interpretan* programas (programas universales), y ver más aplicaciones prácticas de programas.

Definición 8 (Autómata de pila). Un *autómata de pila* se define como un programa de dos vectores, el de entrada y el de la pila. Se permiten solamente dos operaciones en el vector de entrada: `leer(ent)` y `der(ent)`; en el vector `pila` se permiten las operaciones básicas siguientes:

vacía() que tiene el valor 1 si la pila está vacía,

cima() que tiene el valor de la cima de la pila y no la modifica,

desapila() que desapila el símbolo de la cima de la pila, y

apila_pal(w**)** que añade a la pila cada uno de los símbolos de una palabra w de manera ordenada, dejando el primer símbolo de la palabra a la cima de la pila.

Además, permitimos dos extensiones de la sintaxis:

1. las condiciones de las instrucciones `si` y `mientras` pueden contener operadores de conjunción y disyunción, y

2. pueden haber otras variables de una sola casilla que se pueden leer y modificar de la misma forma que la variable `acepta`.

Las últimas extensiones nos permiten escribir programas más claros aunque se podrían simular usando la variable `acepta` con muchos símbolos que representen otras variables y resultados de combinaciones de expresiones.

Si la pila es vacía, el valor de `cima` es $\#$ y `desapila` no tiene ningún efecto.

Las configuraciones de programas nos permiten representar los contenidos de la entrada y de la pila. Por ejemplo, con la instrucción `apila_pal(`abc`)` tenemos el paso

$$(n, abcde, 3, Zfg, 3, 0) \vdash (n+1, abcde, 3, Zfgcba, 6, 0)$$

ya que la cima de la pila la representamos a la posición más a la derecha. Es decir, `cima()` tiene el valor a en la última configuración, y `desapila()` deriva a la configuración $(n+2, abcde, 3, Zfgcb, 5, 0)$.

Ejemplo 9. Consideremos el lenguaje

$$L_2 = \{w2w^t \mid w \in \{0,1\}^*\}.$$

Un programa con pila para este lenguaje es:

```
programa rec_w2wt(ent)
   pila:vector
  escr(acepta,1)
  mientras leer(ent) /= 2
           y leer(ent) /= # haga
    apila_pal(leer(ent))
    der(ent)
  fmientras
  si leer(ent) = 2 ent
     der(ent)
  si_no
     escr(acepta,0)
  fsi
  mientras vacía() = 0
           y cima()=leer(ent) haga
    desapila()
    der(ent)
  fmientras
  si vacía() = 0 o leer(ent) \= # ent
      escr(acepta,0)
  fsi
fin
```

Con el fin de definir automatas de pila no deterministas definimos instrucciones no deterministas, una introducción sencilla al paralelismo.

Definición 10 (Programa indeterminista). Un *programa no determinista*, o *indeterminista*, es un programa donde se permiten instrucciones de la forma

```
paralelo
   S1 con S2 con ... con Sm
fparalelo
```

donde `S1, S2, ..., Sm` son instrucciones.

La semántica intuitiva es que una configuración que ejecuta un bloque de paralelismo se convierte en m configuraciones independientes, todas iguales en datos excepto por la posición de la siguiente instrucción a ejecutar: cada configuración i-ésima comienza

en la posición de S_i. Estas configuraciones generan m hilos de ejecución independientes y, como es costumbre en los autómatas indeterministas, si uno de ellos acepta la entrada se dice que el programa acepta la entrada.

Ejemplo 11. Queremos construir un programa que implemente un autómata de pila para el lenguaje de la gramática

$$P \to pBf$$
$$B \to IB \mid \lambda$$
$$I \to i \mid sBg \mid mBh.$$

Esta gramática es una simplificación de la de los bloques de un programa y por eso tiene gran importancia.

Usando el teorema que permite construir un autómata de pila no determinista para una gramática [4, 1], tenemos el siguiente programa indeterminista. Una variable de la cima de la pila se reemplaza por cada una de sus posibles partes derechas. Hay un hilo de ejecución diferente e independiente para cada posible reemplazo.

```
   programa recProg(ent)
     pila:vector
1  escr(acepta,1)
2  apilar(P)
3  mientras vacía()=0 y
                leer(acepta)=1 haga
4   si cima() = P ent
5      desapila()
6      apila_pal(pBf)
7   si_no
8   si cim() = B ent
9      desapila()
10     paralelo
11          apila_pal(IB)
12       con
13     fparalelo
14  si_no
15  si cim() = I ent
16     desapila()
17     paralelo
18          apila_pal(i)
19       con
20          apila_pal(sBg)
21       con
22          apila_pal(mBh)
23     fparalelo
24  si_no
25  si cim() en {p,f,i,s,g,m,h}  ent
26     si cim() = leer(ent) ent
27        desapila()
28        der(ent)
29     si_no
30        escr(acepta,0)
31     fsi
32  fsi fsi fsi fsi
36 fmientras
37 si leer(ent) /= # ent
38   escr(acepta,0)
39 fsi
40 fin
```

donde el `con` de la instrucción 12 no hace nada porque no se tiene que apilar nada en el caso $B \to \lambda$.

Aunque es una solución simple y directa basada en la gramática es muy ineficiente por el no determinismo. En este caso, es posible dar un programa determinista que reconozca el lenguaje haciendo una observación importante: si se puede determinar cuál de las producciones se tiene que usar, en función de la entrada, entonces el bloque de paralelismo se puede evitar.

Nos detenemos aquí esperando haber convencido al lector de la facilidad de entendimiento que programas como los anteriores tienen, en contraste con las transiciones con estados las cuales requieren un doble esfuerzo.

4. Discusión

Hemos definido equivalentes estructurados de máquinas de Turing, máquinas de Turing indeterministas y autómatas de pila. Las definiciones permiten dar los mismos resultados teóricos tradicionales pero con la motivación adicional de ser más accesibles para los estudiantes debido a su carácter estructurado.

Estos conceptos se han usado por primera vez en una asignatura de autómatas y lenguajes formales de este año. El resultado ha sido interesante: los estudiantes están activos y atentos a hacer comentarios sobre mejoras al programa o dudas sobre las operaciones; los estudiantes se sienten seguros al hacer preguntas y comentarios. Además, durante la demostración de la indecidibilidad del problema de la parada, la contradicción les parecía que se debía a una especie de error de programación superable porque no

veían ningún problema en los programas. Es decir, la demostración es tan directa que salta a la vista la contradicción y choca con la intuición. Todo esto en contraste con su pasividad tradicional cuando se hace la demostración numerando máquinas de Turing.

Existen varios libros y artículos que hacen intentos de dar versiones para programadores de la teoría de la computación. Por ejemplo, Morales y otros [3] definen un máquina contadora estructurada; el problema de usar enteros positivos como estructura básica es que el manejo de programas codificados no es natural; además, no es un buen modelo en teoría de la complejidad porque no es natural suponer que en un paso se puede incrementar un entero. El mejor libro con un enfoque moderno es, sin duda, el de Jones [2]; no es, sin embargo, un libro de autómatas sino de teoría de la computación en cuyo prefacio se apuesta por un enfoque a la computabilidad que evite las máquinas de Turing y las funciones primitivas recursivas y use las estructuras de datos modernas de Lisp y la semántica denotacional de programas.

No conocemos ningún libro que presente los resultados teóricos de autómatas de pila con un enfoque estructurado como el dado aquí.

Referencias

[1] J. Hopcroft, R. Motwani, J. Ullman. *Introducción a la teoría de autómatas, lenguajes y computación*. Addison-Wesley, 2002.

[2] N. Jones. *Computability and Complexity: From a Programming Perspective*, MIT press, 1997.

[3] R. Morales et al. *Una alternativa docente a la máquina de Turing*, Memorias de JENUI 2003, pp 249-258, Ed. Thomson.

[4] J. Rocha, F. Roselló *Autòmatas, Gramàtiques i Programes: Verificació i Concurrència* , Materials Didàctics, UIB, 2a ed. en imprenta, 2005.