LA MÁQUINA DE TURING EN EL ÁMBITO DE LOS LENGUAJES DE PROGRAMACIÓN

FAVIO EZEQUIEL MIRANDA PEREA, ARACELI LILIANA REYES CABELLO, RAFAEL REYES SÁNCHEZ, AND LOURDES DEL CARMEN GONZÁLEZ HUESCA

Resumen. En este artículo presentamos algunos lenguajes de programación que resultan equivalentes entre sí y con la Máquina de Turing con el objetivo de acercar a este protagonista de las teorías de computabilidad y complejidad computacional a un ámbito más familiar para el estudiante de computación. Para esto nos servimos de técnicas e instrucciones de programación usuales, como el enunciado de asignación o la iteración, así como de conceptos de la teoría de lenguajes de programación. Se hará enfasis en la equivalencia entre la Máquina de Turing, los lenguajes estructurados y los no estructurados. De esta manera obtenemos como resultado no sólo un acercamiento de la Máquina de Turing a los lenguajes de programación actuales sino un ejemplo avanzado de la interacción entre las teorías de la computación y de lenguajes de programación que resulta ser una evidencia matemática formal de la validez de la tesis de Church-Turing.

1. Introducción

La Máquina de Turing, originalmente presentada en [5], es el modelo de cómputo más prominente, entre otras razones por ser el primer modelo matemático de cómputo, claro, intuitivo y bien definido antes de la existencia de las computadoras; además, es la base de las teorías de computabilidad y complejidad computacional, proporcionando una herramienta de razonamiento conceptualmente simple y elegante. Sin embargo, este modelo resulta difícil de asimilar como un sistema de programación ya que, en nuestra opinión, se apega más al hardware o al lenguaje de máquina que a un lenguaje de alto nivel. El propósito principal de este artículo es acercar a la Máquina de Turing al ámbito de la teoría y práctica de la programación, para lo cual nos planteamos dos objetivos: el primero es presentarla como un lenguaje de programación; el segundo es responder a la cuestión de si un lenguaje particular es o no Turing-completo dentro del ámbito de los lenguajes de programación, es decir, sin apelar a la implementación directa de un simulador de la Máquina de Turing en el lenguaje en consideración.

Para alcanzar nuestro primer objetivo definimos un lenguaje de programación, al que llamamos LTuring, mostrando que es equivalente a la máquina

Date: 2 de agosto de 2012.

de Turing. Si bien esto acerca el maravilloso invento de Alan Turing al ámbito de la programación, este lenguaje aun es muy rudimentario, por lo que no resulta similar a los lenguajes de alto nivel a los que estamos acostumbrados y por lo tanto tampoco resuelve la cuestión de verificar directa y fácilmente la Turing-completud de un lenguaje. Para remediar esta situación presentamos un lenguaje imperativo estructurado simple, llamado WHILE, que también resulta equivalente a la máquina de Turing y que, salvo detalles de sintaxis, forma parte de cualquier lenguaje de programación moderno. De esta manera logramos el segundo objetivo planteado, pues para verificar la Turing-completud de un lenguaje de programación, basta verificar que éste incluya al lenguaje WHILE, lo cual en la mayoría de los casos resulta muy sencillo.

Consideramos conveniente aclarar que, a lo largo de este trabajo, con lenguaje de programación no nos referimos a los lenguajes de alto nivel como JAVA,C o HASKELL, sino a lenguajes formales considerados como micromodelos de lenguajes de programación reales y que son el principal motivo de estudio de un curso de fundamentos de lenguajes de programación. Cursos, cuyos principales métodos de definición y demostración se sirven de diversos conceptos que le resultarán familiares a cualquier persona que haya estudiado lógica computacional o matemática y teoría de la computación.

Es pues nuestra exposición un capítulo en la intersección de dos grandes teorías en las ciencias de la computación: la de la computación y la de los lenguajes de programación.

2. La Máquina de Turing clásica

Comenzamos nuestra discusión fijando la definición de Máquina de Turing que adoptaremos en el resto del artículo. Cabe señalar que cualquier concepto utilizado y no definido se supone conocido, por lo cual recomendamos como bibliografía de apoyo cualquier libro de teoría de la computación, por ejemplo [4, 2].

Definición 1. *Una máquina de Turing clásica de una cinta es una tupla de la forma*

$$T = \langle \Sigma, Q, q_0, q_f, \delta \rangle,$$

donde

- $\Sigma \neq \varnothing$ *es un alfabeto finito que contiene un símbolo distinguido* $\sqcup$*, llamado símbolo blanco,*
- $Q \neq \varnothing$ *es el conjunto finito de estados, el cual incluye* q_0 *y* q_f*,*
- q_0 *es el estado inicial,*
- q_f *es el estado final de aceptación,*
- $\delta : Q \times \Sigma \to Q \times \Sigma \times M$*, es la función de transición tal que si* $\delta(\ell, s) = (\ell', s', m)$ *entonces*
  - ℓ *es el estado actual,*
  - s *el símbolo que está leyendo la cabeza,*
  - ℓ' *el estado al cual nos llevará la transición,*

- *s' el símbolo que se escribirá,*
- *m el movimiento que realizará la cabeza.*

En nuestra definición consideramos que el alfabeto usado es $\{0, 1, \sqcup\}$ y que $M = \{\leftarrow, \rightarrow, -\}$ es el conjunto de movimientos realizados por la cabeza lectora ya se a la izquierda, a la derecha o permanecer en la misma posición. En adelante denotamos con $\mathcal{MT}$ al conjunto de máquinas de Turing.

El proceso de ejecución de una máquina de Turing se formaliza mediante la relación de transición entre configuraciones instantaneas de acuerdo a la siguiente

Definición 2. *Sea $T = \langle \Sigma, Q, q_0, q_f, \delta \rangle \in \mathcal{MT}$. Una configuración instantanea es un par $\langle q, l\underline{s}r \rangle$ donde $q \in Q$ y $l\underline{s}r \in \Sigma^\star$ y s es el símbolo actual, es decir, el símbolo que está leyendo la cabeza de T. La relación de transición $\vdash$ entre configuraciones se define a partir de la función de transición δ como sigue: si $\delta(q, s) = (p, s', \leftarrow)$ entonces*

$$(q, ls'\underline{s}r) \vdash (p, l\underline{s'}sr)$$

y análogamente para los movimientos $\rightarrow, - \in M$, agregando un blanco $\sqcup$ al extremo de la cadena en caso de ser necesario. Como es costumbre denotamos con $\vdash^\star$ a la cerradura reflexiva y transitiva de $\vdash$.

El lenguaje aceptado por una máquina de Turing se define mediante la noción de configuración de la siguiente manera

Definición 3. *Sea $T = \langle \Sigma, Q, q_0, q_f, \delta \rangle \in \mathcal{MT}$. El lenguaje aceptado por T, denotado $L(T)$ se define como*

$$L(T) = \{x \in \Sigma^\star \mid (q_0, \underline{\sqcup}x) \vdash^\star (q_f, l\underline{s}r)\}$$

Ejemplo 1. *La siguiente máquina de Turing $T = \langle \{0, 1, \sqcup\}, \{0, 1, 2, 3\}, 1, 3, \delta \rangle$ verifica que exista un número par de ceros en la cadena de entrada.*

δ	0	1	$\sqcup$
0	$-$	$-$	$(1, \sqcup, \rightarrow)$
1	$(2, 0, \rightarrow)$	$(1, 1, \rightarrow)$	$(3, \sqcup, -)$
2	$(1, 0, \rightarrow)$	$(2, 1, \rightarrow)$	$-$
3	$-$	$-$	$-$

donde 0 es el estado inicial y 3 el estado final. El lector puede verificar, por ejemplo que $(1, \underline{\sqcup}01011) \vdash^\star (3, \sqcup 01011\underline{\sqcup})$ y por lo tanto $01011 \in L(T)$.

A continuación damos la definición de lenguaje de programación. Los tres formalismos presentados en este artículo son ejemplos particulares de esta clase de lenguajes de programación.

3. Lenguajes de programación

Un lenguaje de programación puede entenderse como un formalismo que permite definir programas que transforman datos de entrada en resultados o datos de salida. Con esta idea en mente enunciamos la siguiente

Definición 4. *Un lenguaje de programación es una terna* $\mathcal{L} = \langle \mathcal{P}_\mathcal{L}, \mathcal{D}_\mathcal{L}, [\![\cdot]\!]_\mathcal{L} \rangle$ *donde*

- $\mathcal{P}_\mathcal{L} \neq \varnothing$ *es el conjunto de programas de* $\mathcal{L}$
- $\mathcal{D}_\mathcal{L} \neq \varnothing$ *es el conjunto de datos (de entrada y salida) de* $\mathcal{L}$
- $[\![\cdot]\!]_\mathcal{L}$ *es la función semántica de* $\mathcal{L}$ *tal que*

$$[\![\cdot]\!]_\mathcal{L} : \mathcal{P}_\mathcal{L} \to \mathcal{D}_\mathcal{L} \rightharpoonup \mathcal{D}_\mathcal{L}$$

En el resto de este artículo entendemos por lenguaje de programación, única y exclusivamente a aquellos formalismos que cumplen con esta definición. Se observa que dado un programa p, su significado es la función $[\![p]\!]_\mathcal{L} : \mathcal{D}_\mathcal{L} \rightharpoonup \mathcal{D}_\mathcal{L}$ que es una función parcial puesto que $[\![p]\!]_\mathcal{L}(x)$ devuelve el resultado de la ejecución de p al recibir a $x \in \mathcal{D}_\mathcal{L}$ como dato de entrada, el cual podría no existir, por ejemplo si p se cicla infinitamente. En adelante cuando nos encontremos con afirmaciones de la forma $[\![p]\!]_\mathcal{L}(x) = y$ estamos suponiendo que el valor $[\![p]\!]_\mathcal{L}(x)$ existe y que es igual a y.

La función semántica de un lenguaje de programación puede definirse de manera directa y es esencialmente una función de interpretación en el estilo de la lógica de primer orden, donde los programas se interpretan como funciones que transforman datos pertenecientes a cierto dominio fijo. Esto se conoce como una semántica denotativa. Otra manera de definir a esta función, que es la que utilizaremos en adelante en nuestra exposición, es mediante un sistema de transición similar al de un autómata o máquina de Turing. En este caso se trata de una semántica operacional definida como sigue.

Definición 5. *Sea* $\mathcal{L}$ *un lenguaje de programación. Una semántica operacional para* $\mathcal{L}$ *es una tupla* $\mathcal{O}_\mathcal{L} = \langle \mathcal{S}, \mathcal{E}, \cdot \rhd \cdot \to \cdot, \mathsf{final}, \mathsf{init}, \mathsf{output} \rangle$ *tal que*

- $\mathcal{S} \neq \varnothing$ *es el conjunto de memorias*[1] *para* $\mathcal{O}_\mathcal{L}$.
- $\mathcal{E} \neq \varnothing$ *es el conjunto de estados cuya definición involucra usualmente a* $\mathcal{S}$.
- $\cdot \rhd \cdot \to \cdot \subseteq \mathcal{P}_\mathcal{L} \times \mathcal{E} \times \mathcal{E}$ *es una relación ternaria de transición entre programas, estados y estados cuyo significado intensional es*

  $p \rhd s \to s'$ *si y sólo si la ejecución del programa* $p \in \mathcal{P}_\mathcal{L}$ *causa la transición del estado* s *al estado* s'
- $\mathsf{final} : \mathcal{P}_\mathcal{L} \to \mathcal{P}(\mathcal{E})$ *es la función que define estados finales. Es decir, un estado* s *es final para el programa* p *si y sólo si* $s \in \mathsf{final}(p)$.
- $\mathsf{init} : \mathcal{P}_\mathcal{L} \times \mathcal{D}_\mathcal{L} \to \mathcal{E}$ *es la función de inicialización de la ejecución.*
- $\mathsf{output} : \mathcal{P}_\mathcal{L} \times \mathcal{E} \to \mathcal{D}_\mathcal{L}$ *es la función de salida.*

La función semántica de un lenguaje de programación puede definirse a partir de una semántica operacional de acuerdo a la siguiente

Definición 6. *Sea* $\mathcal{O}_\mathcal{L}$ *una semántica operacional para* $\mathcal{L}$. *Definimos la función semántica de* $\mathcal{L}$ *a partir de* $\mathcal{O}_\mathcal{L}$ *como sigue:*

[1] En inglés *stores*

$$\forall p \in \mathcal{P}_{\mathcal{L}} \forall x, y \in \mathcal{D}_{\mathcal{L}} \big([\![p]\!](x) = y \text{ si y sólo si } p \rhd s_0 \to^\star s_f \big)$$

donde $s_0 = \mathsf{init}(p, x)$, $y = \mathsf{output}(p, s_f)$ *y* $s_f \in \mathsf{final}(p)$. *Aquí* $s \to^\star s'$ *denota, como es usual, a la cerradura reflexiva y transitiva de la relación* $\to$.

A continuación presentamos nuestro primer lenguaje de programación, así como su equivalencia con la máquina de Turing.

4. LTuring: Un lenguaje para máquinas de Turing

El propósito de esta sección es acercar a la máquina de Turing al ámbito de la programación mediante la definición de un lenguaje que esencialmente permite eliminar la definición directa de una función de transición a favor de un programa.

Definición 7. *El lenguaje* LTuring *se define como sigue:*

- *Datos:* $\mathcal{D}_{\mathcal{L}} = \{0, 1, \sqcup\}^\star$
- *Programas:* $\mathcal{P}_{\mathcal{L}}$ *se define mediante la siguiente gramática*
  $\mathcal{P}_{\mathcal{L}} \ni p ::= 1 : \mathcal{I}_1; \ldots; m : \mathcal{I}_m : m+1 : \texttt{halt}$
  $\mathcal{I} ::= \texttt{right} \mid \texttt{left} \mid \texttt{write}\, s \mid \texttt{if}\, s\, \texttt{then goto}\, \ell\, \texttt{else goto}\, \ell' \mid \texttt{goto}\, \ell$
  donde $s \in \{0, 1, \sqcup\}$ *y* $\ell, \ell' \in \mathbb{N}$.
- *La función semántica* $[\![\cdot]\!]$ *se define mediante la siguiente semántica operacional:*
  - *Memorias:* $\mathcal{S} =_{def} \mathcal{D}_{\mathcal{L}} \times \{0, 1, \sqcup\} \times \mathcal{D}_{\mathcal{L}}$, *donde usualmente escribiremos* $L\underline{s}R \in \mathcal{S}$ *en vez de* $(L, s, R) \in \mathcal{S}$. *Obsérvese que una memoria puede verse como la cinta de una máquina de Turing.*
  - *Estados:* $\mathcal{E} = \mathbb{N} \times \mathcal{S}$
  - *Relación de transición: sea* $p = 1 : \mathcal{I}_1; \ldots; m : \mathcal{I}_m; m+1 : \texttt{halt}$.
    - *Si* $\mathcal{I}_\ell = \texttt{right}$ *entonces* $p \rhd \langle \ell, L\underline{s}s'R \rangle \to \langle \ell + 1, Ls\underline{s'}R \rangle$
    - *Si* $\mathcal{I}_\ell = \texttt{right}$ *entonces* $p \rhd \langle \ell, L\underline{s} \rangle \to \langle \ell + 1, Ls\underline{\sqcup} \rangle$
    - *Si* $\mathcal{I}_\ell = \texttt{left}$ *entonces* $p \rhd \langle \ell, Ls\underline{s'}R \rangle \to \langle \ell + 1, L\underline{s}s'R \rangle$
    - *Si* $\mathcal{I}_\ell = \texttt{left}$ *entonces* $p \rhd \langle \ell, \underline{s}R \rangle \to \langle \ell + 1, \underline{\sqcup}sR \rangle$
    - *Si* $\mathcal{I}_\ell = \texttt{write}\, s$ *entonces* $p \rhd \langle \ell, L\underline{s'}R \rangle \to \langle \ell + 1, L\underline{s}R \rangle$
    - *Si* $\mathcal{I}_\ell = \texttt{goto}\, \ell'$ *entonces* $p \rhd \langle \ell, L\underline{s}R \rangle \to \langle \ell', L\underline{s}R \rangle$
    - *Si* $\mathcal{I}_\ell = \texttt{if}\, s\, \texttt{then goto}\, \ell'\, \texttt{else goto}\, \ell''$ *entonces* $p \rhd \langle \ell, L\underline{s}R \rangle \to \langle \ell', L\underline{s}R \rangle$
    - *Si* $\mathcal{I}_\ell = \texttt{if}\, s\, \texttt{then goto}\, \ell'\, \texttt{else goto}\, \ell''$ *entonces* $p \rhd \langle \ell, L\underline{s'}R \rangle \to \langle \ell'', L\underline{s'}R \rangle$
  - *Estados finales:* $\mathsf{final}(p) = \{\langle m+1, \sigma \rangle | \sigma \in \mathcal{S}\}$.
  - *Función de inicialización:* $\mathsf{init} : \mathcal{P}_{\mathcal{L}} \times \mathcal{D}_{\mathcal{L}} \to \mathcal{E}$, $\mathsf{init}(x) = \langle 1, \underline{\sqcup}x \rangle$
  - *Función de salida:* $\mathsf{output} : \mathcal{P}_{\mathcal{L}} \times \mathcal{E} \to \mathcal{D}_{\mathcal{L}}$, $\mathsf{output}(\langle \ell, L\underline{s}R \rangle) = R$

Veamos ahora un ejemplo de programación en LTuring.

Ejemplo 2. *El siguiente programa* `swap` *invierte una cadena de ceros y unos.*

```
1: if 1 then goto 2 else goto 5
2: write 0
```

```
 3: right
 4: goto 1
 5: if 0 then goto 6 else goto 9
 6: write 1
 7: right
 8: goto 1
 9: left
10: if ⊔ then 11 else 9
11: halt
```

El lector puede verificar por ejemplo que `swap` $\triangleright (1, \sqcup 1010) \to^\star (11, \sqcup 0101)$ *y por lo tanto* $[\![\texttt{swap}]\!](1010) = 0101$.

De la definición misma del lenguaje y del ejemplo anterior se observa que la semántica operacional de LTuring simula al proceso de ejecución de una Máquina de Turing de una manera muy directa, puesto que un programa es esencialmente la definición de la función de transición δ. Se intuye entonces que todo programa p puede transformarse en una máquina de Turing. A continuación demostramos esta afirmación

4.1. Equivalencia de LTuring con la máquina de Turing clásica. En esta sección mostramos que todo programa en LTuring puede convertirse en una máquina de Turing que se comporte de igual forma y viceversa.

Proposición 1 (De $\mathcal{MT}$ a LTuring)**.** *Sea* $T = \langle \Sigma, Q, q_0, q_f, \delta \rangle$ *con* $\Sigma = \{0, 1, \sqcup\}$ *una máquina de Turing clásica. Existe un programa* p_T *del lenguaje* LTuring *tal que*

$$L(T) \subseteq \{x \in \Sigma^\star \mid [\![p_T]\!](x) \textit{existe}\}$$

Más aún, si $(q_0, \sqcup x) \to^\star (q_f, w \underline{s}\, y)$ *entonces* $[\![p_T]\!](x) = y$.

Demostración. Dada la máquina de Turing T, definiremos un programa p_T en Goto. La idea para construir p_T es simular cada transición de la máquina de Turing mediante una secuencia de instrucciones del lenguaje Goto. Para esto usamos la siguiente notación: si $m \in \{\to, \leftarrow\}$ entonces definimos m^g como $(\to)^g =$ `right` y $(\leftarrow)^g =$ `left`. Por otra parte si $\mathcal{J}$ es una instrucción o secuencia de instrucciones en Goto, denotamos con $\widehat{\mathcal{J}}$ a la etiqueta correspondiente a (la primera instrucción de) $\mathcal{J}$ dentro del programa p_T, la cual se determinará posteriormente. Las instrucciones involucradas en p_T se definen como sigue:

- Para cada estado $q \in Q$ definimos la siguiente instrucción $\mathcal{I}_q$, la cual se encarga de transferir el control del programa a la parte que simulará la transición adecuada.

$$\begin{aligned} \mathcal{I}_q \quad =_{def} \quad \texttt{case}\ s\ \texttt{of} \quad & 0 \Rightarrow \texttt{goto}\ \widehat{\mathcal{S}_{q,0}}; \\ & 1 \Rightarrow \texttt{goto}\ \widehat{\mathcal{S}_{q,1}}; \\ & \sqcup \Rightarrow \texttt{goto}\ \widehat{\mathcal{S}_{q,\sqcup}} \end{aligned}$$

- Cada transición $\delta(q, s) = (p, s', m)$ se simulará mediante la secuencia de instrucciones $\mathcal{S}_{q,s}$ definida como sigue:
  - Si $m \in \{\rightarrow, \leftarrow\}$ y $s \neq s'$ entonces
$$\mathcal{S}_{q,s} =_{def} \texttt{write}\, s'; m^g; \texttt{goto}\, \widehat{\mathcal{I}_p}$$
  - Si $m = -$ y $s \neq s'$ entonces
$$\mathcal{S}_{q,s} =_{def} \texttt{write}\, s'; \texttt{goto}\, \widehat{\mathcal{I}_p}$$
  - Si $m \in \{\rightarrow, \leftarrow\}$ y $s = s'$ entonces
$$\mathcal{S}_{q,s} =_{def} m^g; \texttt{goto}\, \widehat{\mathcal{I}_p}$$
  - Si $m = -$ y $s = s'$ entonces
$$\mathcal{S}_{q,s} =_{def} \texttt{goto}\, \widehat{\mathcal{I}_p}$$
- Finalmente el programa p_T se define como sigue: sean $\mathcal{I}_j$ las instrucciónes correspondientes a los m estados en Q. Suponemos que δ se compone de k transiciones y sean $\mathcal{S}_i$ con $1 \leq i \leq k$ las secuencias correspondientes a dichas transiciones.

$$p_T =_{def} 1 : \mathcal{I}_1; 2 : \mathcal{I}_2, \ldots, m : \mathcal{I}_m; m+1 : \mathcal{S}_1; \ell_2 : \mathcal{S}_2, \ldots, \ell_k : \mathcal{S}_k; \ell_{k+1} : \texttt{halt}$$

donde $\ell_2 = |\mathcal{S}_1| + m + 1$ y$\ell_{j+1} = \ell_j + |\mathcal{S}_j|$ para $1 < j < k + 1$.

De la construcción es claro que si $(q_0, \sqcup\underline{x}) \vdash^\star (q_f, w\underline{s}\, y)$ entonces $p \rhd (1, \sqcup\underline{x}) \rightarrow^\star (\ell_{k+1}, w\underline{s}\, y)$ y por lo tanto $x \in L(T)$ implica que $[\![p_T]\!](x)$ existe pues $[\![p_T]\!](x) = y$. Los detalles se dejan como ejercicio al lector. □

Veamos ahora un ejemplo de este proceso de transformación obteniendo a partir de una máquina el programa correspondiente.

Ejemplo 3. *La siguiente máquina verifica la paridad de los unos en una cadena, devolviendo 0 si hay un número impar de unos y 1 en otro caso.*

δ	0	1	$\sqcup$
1	-	-	$(2, \sqcup, \rightarrow)$
2	$(2, 0, \rightarrow)$	$(3, 1, \rightarrow)$	$(4, 0, \rightarrow)$
3	$(3, 0, \rightarrow)$	$(2, 1, \rightarrow)$	$(4, 1, \leftarrow)$

siendo 1 *el estado inicial y* 4 *el estado final.*

Al aplicar la transformación anterior a esta máquina se obtiene el siguiente programa:

```
1:if ⊔ then goto 4 else goto 22
2:case s of
   0 => goto 6
   1 => goto 13
   ⊔ => goto 16
3:case s of
   0 => goto 11
```

```
    1 => goto 8
   ⊔ => goto 19
 4:right
 5:goto 2
 6:right
 7:goto 2
 8:write 1
 9:right
10:goto 3
11:right
12:goto 3
13:write 1
14:right
15:goto 2
16:write 0
17:left
18:goto 22
19:write 1
20:left
21:goto 22
22:halt
```

Para terminar con la prueba de equivalencia discutimos ahora la transformación inversa.

Proposición 2 (De LTuring a $\mathcal{MT}$)**.** *Sea p un programa en* LTuring*. Existe una máquina de Turing clásica T_p tal que si $[\![p]\!](x) = y$ entonces $x \in L(T)$. Más aún, si $p \rhd_{LTuring} (1, \sqcup x) \to^\star (m+1, w\underline{s}y)$ entonces $(q_1, \sqcup x) \vdash^\star (q_f, w\underline{s}y)$.*

Demostración. Sea $p = 1 : \mathcal{I}_1; \ldots; m : \mathcal{I}_m; m+1 : \texttt{halt}$. Tomemos $Q = \{1, \ldots, m+1\}$ siendo $m+1$ el estado final y 1 el estado inicial. Definimos para cada instrucción $\mathcal{I}_\ell$ un conjunto de transiciones como sigue:

- Si $\mathcal{I}_\ell = \texttt{right}$ entonces
$$\forall s \in \Sigma\big(\delta(\ell, s) = (\ell+1, s, \to)\big)$$
- Si $\mathcal{I}_\ell = \texttt{left}$ entonces
$$\forall s \in \Sigma\big(\delta(\ell, s) = (\ell+1, s, \leftarrow)\big)$$
- Si $\mathcal{I}_\ell = \texttt{write}\, s'$ entonces
$$\forall s \in \Sigma\big(\delta(\ell, s) = (\ell+1, s', -)\big)$$
- Si $\mathcal{I}_\ell = \texttt{if}\, s' \,\texttt{then goto}\, \ell' \,\texttt{else goto}\, \ell''$ entonces
$$\begin{array}{rcl} \delta(\ell, s') & = & (\ell', s', -) \\ \delta(\ell, s) & = & (\ell'', s, -), \quad \text{si } s \neq s' \end{array}$$

- Si $\mathcal{I}_\ell = \texttt{goto}\,\ell'$ entonces

$$\forall s \in \Sigma\big(\delta(\ell, s) = (\ell', s, -)\big)$$

La definición de la función de transición deja claro que se está simulando fielmente al programa original. La verificación formal de este hecho se dejan como ejercicio al lector. □

Veamos como funciona la transformación anterior en el programa correspondiente al ejemplo 2.

Ejemplo 4. *La siguiente máquina de Turing corresponde al programa* `swap`.

δ	0	1	$\sqcup$
1	$(5, 0, -)$	$(2, 1, -)$	$(5, \sqcup, -)$
2	$(3, 0, -)$	$(3, 0, -)$	$(3, 0, -)$
3	$(4, 0, \rightarrow)$	$(4, 1, \rightarrow)$	$(4, \sqcup, \rightarrow)$
4	$(1, 0, -)$	$(1, 1, -)$	$(1, \sqcup, -)$
5	$(6, 0, -)$	$(9, 1, -)$	$(9, \sqcup, -)$
6	$(7, 1, -)$	$(7, 1, -)$	$(7, 1, -)$
7	$(8, 0, \rightarrow)$	$(8, 1, \rightarrow)$	$(8, \sqcup, \rightarrow)$
8	$(1, 0, -)$	$(1, 1, -)$	$(1, \sqcup, -)$
9	$(10, 0, \leftarrow)$	$(10, 1, \leftarrow)$	$(10, \sqcup, \leftarrow)$
10	$(9, 0, -)$	$(9, 1, -)$	$(11, \sqcup, -)$

Hasta ahora hemos logrado el primero de nuestros objetivos al mostrar un lenguaje de programación equivalente a la Máquina de Turing. Sin embargo, LTuring resulta casi tan rudimentario como ésta para ser considerado una alternativa en lo que respecta a servir como un sistema de programación. Presentamos ahora un lenguaje imperativo que le parecerá muy familiar al lector y que resultará ser nuevamente equivalente a $\mathcal{MT}$.

5. While: Un lenguaje imperativo estructurado

En [3], Jones desarrolla las teorías de la computabilidad y complejidad prescindiendo de la Máquina de Turing y utilizando en su lugar al lenguaje While. La versión discutida aquí difiere ligeramente de la original para lograr que sea un caso particular de nuestra definición 4. Empezamos definiendo al conjunto particular de datos $\mathbb{D}$ que utilizaremos para definir al lenguaje While.

5.1. Árboles binarios. Un dato para While está representados mediante un árbol binario, construidos a partir de un elemento atómico llamado `nil`; esta clase de árboles sólo tienen etiquetas en las hojas y se genera con la siguiente definición:

Definición 8. *El conjunto de árboles $\mathbb{D}$ se define recursivamente como:*

- `nil` *es un elemento de* $\mathbb{D}$,
- *si* d_1 *y* d_2 *son elementos de* $\mathbb{D}$ *entonces* $(d_1.d_2)$ *son elementos de* $\mathbb{D}$;
- $\mathbb{D}$ *es el conjunto más pequeño que satisface lo anterior.*

La elección de esta estructura de datos en vez de números naturales y/o booleanos, usados en lenguajes similares, quedará clara más adelante. Veamos ahora como esta estructura es suficientemente poderosa para representar a los valores booleanos y a los números naturales

Definición 9. *Los valores de verdad* `true` *y* `false` *se definen como:*

- `false = nil`
- `true = (nil.nil)`

En el caso de los números naturales la idea es representar el número natural n mediante un árbol de tamaño n, construido de la siguiente forma:

Definición 10. *Definimos* $\underline{n} = \texttt{nil}^n$ *donde*

$$\begin{array}{rcl} \texttt{nil}^0 & = & \texttt{nil} \\ \texttt{nil}^{n+1} & = & (\texttt{nil.nil}^{\texttt{n}}) \end{array}$$

y sea $\mathcal{N} = \{\underline{n} | n \in \mathbb{N}\}$. *Los elementos de* $\mathcal{N}$ *se llaman numerales.*

Por ejemplo, el numeral correspondiente al número natural 4 es $\underline{4} = (\texttt{nil}.(\texttt{nil}.(\texttt{nil}.(\texttt{nil}.\texttt{nil}))))$. Por simplicidad escribiremos $0, 1, 2, \ldots$ en vez de $\underline{0}, \underline{1}, \underline{2}, \ldots$ o $\texttt{nil}^0$,$\texttt{nil}^1$, $\texttt{nil}^2, \ldots$.

Una vez definido el conjunto de datos podemos dar la definición del lenguaje.

Definición 11. *El lenguaje* WHILE *se define como sigue:*

- $\mathcal{D}_{\mathcal{L}} = \mathbb{D}$.
- $\mathcal{P}_{\mathcal{L}}$ *se define mediante la siguiente gramática*
  $\mathcal{P}_{\mathcal{L}} \ni p ::= \texttt{read}\, X;\, C_1, \ldots, C_m\, ; \texttt{write}\, Y$
  $C ::= X := e \mid \texttt{while}\, e\, \texttt{do}\ \vec{C}\ \texttt{end}$
  $e, f ::= X \mid \texttt{d} \mid \texttt{cons}\ e\ f \mid \texttt{hd}\, e \mid \texttt{tl}\, e \mid =?\, e\ f$
  donde $\vec{C}$ *es una secuencia de comandos de la forma* $\vec{C} =_{def} C_1; C_2; \ldots; C_n$.
  La longitud o número de lineas en una secuencia de comandos, denotada $|\vec{C}|$ *es un concepto de importancia y se define recursivamente como sigue:*
  - $|X := e| = 1$
  - $|\texttt{while}\, e\, \texttt{do}\ \vec{C}\ \texttt{end}| = |\vec{C}| + 1$
  - $|C_1; \ldots; C_n| = |C_1| + \ldots + |C_n|$
- *La función semántica se define mediante la siguiente semántica operacional:*
  - *Memorias:* $\mathcal{S} = \{\sigma |\ \sigma : Var \to \mathbb{D}\}$
  - *Estados:* $\mathcal{E} = \mathbb{N} \times \mathcal{S}$.
  - *Evaluación de expresiones: la relación de transición se servirá de la siguiente función* $\mathsf{ev} : \mathcal{E} \to Expr \to \mathbb{D}$ *donde para cada estado* s *escribimos* ev_s *en vez de* $\mathsf{ev}\ s$, *así* $\mathsf{ev}_s : Expr \to \mathbb{D}$
    - $\mathsf{ev}_s(X) = \sigma(X)$ *donde* $s = \langle \ell, \sigma \rangle$.
    - $\mathsf{ev}_s(d) = d$
    - $\mathsf{ev}_s(\texttt{cons}\, e\, f) = ((\mathsf{ev}_\sigma(e)).(\mathsf{ev}_\sigma(f)))$

- $\mathsf{ev}_s(\mathtt{hd}\, e) = nil$, *si* $\mathsf{ev}_\sigma\, e = nil$
- $\mathsf{ev}_s(\mathtt{hd}\, e) = t$, *si* $\mathsf{ev}_\sigma\, e = (t.r)$
- $\mathsf{ev}_s(\mathtt{tl}\, e) = nil$, *si* $\mathsf{ev}_\sigma\, e = nil$
- $\mathsf{ev}_s(\mathtt{tl}\, e) = r$, *si* $\mathsf{ev}_\sigma\, e = (t.r)$
- $\mathsf{ev}_s(=?\, e\ f) = true$, *si* $\mathsf{ev}_\sigma\, e = \mathsf{ev}_\sigma\, f$
- $\mathsf{ev}_s(=?\, e\ f) = false$, *si* $\mathsf{ev}_\sigma\, e \neq \mathsf{ev}_\sigma\, f$

- *Relación de transición: sea* $p = \mathtt{read}\, X; \vec{C}; \mathtt{write}\, Y$.
  - *Si* $\mathcal{I}_\ell = X := e$ *y* $\mathsf{ev}_{\langle \ell, \sigma \rangle}(e) = d$ *entonces*

    $$p \rhd \langle \ell, \sigma \rangle \to \langle \ell + 1, \sigma[X/d] \rangle$$

    donde $\sigma[X/d]$ *denota a la memoría que resulta al actualizar* σ *con el valor* d *para la variable* X.
  - *Si* $\mathcal{I}_\ell = \mathtt{while}\, e\ \mathtt{do}\ \ \vec{D}\ \mathtt{end}$ *con* $|\vec{D}| = k$ *y* $\mathsf{ev}_s(e) = nil$, *entonces*

    $$p \rhd \langle \ell, \sigma \rangle \to \langle \ell + k + 1, \sigma \rangle$$

  - *Si* $\mathcal{I}_\ell = \mathtt{while}\, e\ \mathtt{do}\ \ \vec{D}\ \mathtt{end}$ *con* $|\vec{D}| = k$ *y* $\mathsf{ev}_s(e) \neq nil$, *entonces*

    $$p \rhd \langle \ell, \sigma \rangle \to \langle \ell + k + 1, \sigma'' \rangle,$$

    donde
    - $p \rhd \langle \ell + 1, \sigma \rangle \to^\star \langle \ell + k, \sigma' \rangle$
    - $p \rhd \langle \ell, \sigma' \rangle \to \langle \ell + k + 1, \sigma'' \rangle$
- *Estados finales:* $\mathsf{final}(p) = \{\langle m + 1, \sigma \rangle \mid \sigma \in \mathcal{S}\}$, *donde* $m = |\vec{C}|$.
- *Función de inicialización:* $\mathsf{init} : \mathcal{P}_\mathcal{L} \times \mathcal{D}_\mathcal{L} \to \mathcal{E}$,

$$\mathsf{init}(p, d) = \big\langle 1, [X \mapsto d, Y \mapsto \mathtt{nil}, Z_1 \mapsto \mathtt{nil}, \ldots, Z_n \mapsto \mathtt{nil}] \big\rangle$$

  donde $p = \mathtt{read}\, X; \vec{C}; \mathtt{write}\, Y$ *y* $Vars(p) = \{X, Y, Z_1, \ldots, Z_n\}$.
- *Función de salida:* $\mathsf{output} : \mathcal{P}_\mathcal{L} \times \mathcal{E} \to \mathcal{D}_\mathcal{L}$, $\mathsf{output}(p, \sigma) = \sigma(Y)$

A continuación se presentan algunos programas en el lenguaje WHILE

Ejemplo 5. *Las instrucciones condicional* `if` *e* `if-else` *se pueden expresar mediante la instrucción* `while` *de la siguiente forma: Dada la siguiente secuencia de instrucciones, se ejecutará* `C` *si y sólo si el resultado de la evaluación de* `E` *es* `true`.

```
Z := E;   ( * if E then C *)
while Z do
  Z := false;
  C;
end
```

En las siguientes instrucciones se ejecutará `C1` *si el resultado de la evaluación de* `E` *es* `true` *y* `C2` *en caso contrario.*

```
Z := E;   ( * if E then C1 else C2 *)
W := true;
while Z do
```

```
  Z := false;
  W := false;
  C1;
end
while W do
  W := false;
  C2;
end
```

Ejemplo 6. *Los siguientes programas calculan el sucesor y el predecesor de un numeral.*

```
read X;  (* suc *)
  Y := cons nil X;
write Y
```

```
read X;  (* pred *)
  Y := tl X;
write Y;
```

Ejemplo 7. *El siguiente programa realiza la suma dos numerales. Aquí XY es una variable que recibe a un dato de entrada el cual debe ser de la forma $(n.m)$ para devolver $n + m$.*

```
read XY;  (* add X Y *)
  X := hd XY;
  Y := tl XY;
  while X do
    Y := cons nil Y;
    X := tl X;
  end
write Y
```

Ejemplo 8. *Este programa obtiene la reversa de un elemento de $\mathbb{D}$, si la entrada es $d_1.d_2.\ldots.d_n.nil$ entonces la salida será $d_n.d_{n-1}.\ldots.d_1.nil$:*

```
read X;
  Y := nil;
  while X do
    Y := cons (hd X) Y;
    X := tl X;
  end
write Y
```

En nuestra siguiente sección presentamos un lenguaje reminiscente de los antiguos lenguajes no estructurados como Fortran o Basic.

6. Goto: un lenguaje imperativo no estructurado

Para probar la equivalencia entre $\mathcal{MT}$ y While nos serviremos de un lenguaje que manipule los mismos datos que While pero que incluye una instrucción explícita para modificar el flujo del programa. Obsérvese que la ejecución en While es estrictamente secuencial mientras que una máquina de Turing o un programa en LTuring puede saltar arbitrariamente entre

cualesquiera dos de sus estados de la misma manera que el lenguaje discutido en esta sección.

Definición 12. *El lenguaje* GOTO *se define como sigue:*

- $\mathcal{D}_{\mathcal{L}} = \mathbb{D}$
- $\mathcal{P}_{\mathcal{L}}$ *se define mediante la siguiente gramática*
  $\mathcal{P}_{\mathcal{L}} \ni p ::= \texttt{read}\, X; 1 : \mathcal{I}_1; \ldots; m : \mathcal{I}_m; \texttt{write}\, Y$
  $\mathcal{I} ::= X := e \mid \texttt{if}\, X\, \texttt{then goto}\, \ell\, \texttt{else goto}\, \ell' \mid \texttt{goto}\, \ell$
  $e, f ::= X \mid \texttt{d} \mid \texttt{cons}\ Y\ Z \mid \texttt{hd}\, X \mid \texttt{tl}\, X \mid =?\, Y\ Z$
  donde $\ell, \ell' \in \mathbb{N}$. *Obsérvese que, a diferencia de* WHILE, *en este lenguaje en cada expresión* e *hay exactamente una presencia de operador y la longitud* $|\vec{\mathcal{I}}|$ *de una secuencia de instrucciones es simplemente el número de instrucciones que la componen.*

La función semántica se define mediante la siguiente semántica operacional:

- *Memorias:* $\mathcal{S} = \{\sigma |\ \sigma : Var \to \mathbb{D}\}$
- *Estados:* $\mathcal{E} = \mathbb{N} \times \mathcal{S}$.
- *Evaluación de expresiones: si* $s = \langle \ell, \sigma \rangle$ *entonces definimos la función de evaluación* $ev_s : Expr \to \mathcal{D}_{\mathcal{L}}$ *como sigue:*
  - $ev_s(X) = \sigma(X)$
  - $ev_s(d) = d$
  - $ev_s(\texttt{hd}\, X) = nil$, *si* $\sigma(X) = nil$
  - $ev_s(\texttt{hd}\, X) = d$, *si* $\sigma(X) = (d.e)$
  - $ev_s(\texttt{tl}\, X) = nil$, *si* $\sigma(X) = nil$
  - $ev_s(\texttt{tl}\, X) = e$, *si* $\sigma(X) = (d.e)$
  - $ev_s(\texttt{cons}\, X\ Y) = (d.e)$ *si* $\sigma(X) = d,\ \sigma(Y) = e$
  - $ev_s(=?X\ Y) = true$ *si* $\sigma(X) = \sigma(Y)$.
  - $ev_s(=?X\ Y) = false$ *si* $\sigma(X) \neq \sigma(Y)$.

  Obsérvese que la función ev_s *no es recursiva a diferencia de la función correspondiente en el lenguaje* WHILE.
- *Relación de transición: sea* $p = \texttt{read}\, X; 1 : \mathcal{I}_1; \ldots; m : \mathcal{I}_m; \texttt{write}\, Y$.
  - *Si* $\mathcal{I}_\ell =_{def} X := e$ *y* $\mathsf{ev}_{\langle \ell, \sigma \rangle}(e) = d$ *entonces*
    $$p \rhd \langle \ell, \sigma \rangle \to \langle \ell + 1, \sigma[X/d] \rangle$$
  - *Si* $\mathcal{I}_\ell = \texttt{if}\, X\, \texttt{then goto}\, \ell'\, \texttt{else goto}\, \ell''$ *y* $\sigma(X) = \texttt{nil}$ *entonces*
    $$p \rhd \langle \ell, \sigma \rangle \to \langle \ell'', \sigma \rangle$$
  - *Si* $\mathcal{I}_\ell = \texttt{if}\, X\, \texttt{then goto}\, \ell'\, \texttt{else goto}\, \ell''$ *y* $\sigma(X) \neq \texttt{nil}$ *entonces*
    $$p \rhd \langle \ell, \sigma \rangle \to \langle \ell', \sigma \rangle$$
  - *Si* $\mathcal{I}_\ell = \texttt{goto}\, \ell'$ *entonces*
    $$p \rhd \langle \ell, \sigma \rangle \to \langle \ell', \sigma \rangle$$
- *Estados finales:* $\mathsf{final}(p) = \{\langle m + 1, \sigma \rangle \mid \sigma \in \mathcal{S}\}$.
- *Función de inicialización:* $\mathsf{init} : \mathcal{P}_{\mathcal{L}} \times \mathcal{D}_{\mathcal{L}} \to \mathcal{E}$
  $$\mathsf{init}(p, d) = \big\langle 1, [X \mapsto d, Z_1 \mapsto \texttt{nil}, \ldots, Z_n \mapsto \texttt{nil}] \big\rangle$$

donde $p = \texttt{read}\, X; \vec{C}; \texttt{write}\, Y$ *y* $Vars(p) = \{X, Y, Z_1, \ldots, Z_n\}$.

- *Función de salida:* $\mathsf{output} : \mathcal{P}_{\mathcal{L}} \times \mathcal{E} \to \mathcal{D}_{\mathcal{L}},\ \mathsf{output}(p, \sigma) = \sigma(Y)$

Veamos un ejemplo de programa GOTO.

Ejemplo 9. *El siguiente programa* `reverse` *implementa la función reversa.*

```
read X
1: Y := nil;
2: if X then goto 3 else goto 7;
3: Z := hd X;
4: Y := cons Z Y;
5: X := tl X;
6: goto 2;
write Y
```

Obsérvese que la etiqueta `7` no existe en el programa anterior. Sin embargo, la instrucción `goto 7` no es incorrecta pues su ejecución causará que el programa se detenga en un estado de la forma $\langle 7, \sigma \rangle$ el cual es final y por lo tanto no habrá un error en tiempo de ejecución.
Si comparamos el ejemplo anterior con su contraparte en WHILE (ejemplo 8) podemos vislumbrar un método para modelar la instrucción `while` mediante la combinación del condicional `if` y el `goto`. Esto nos hace conjeturar que todo programa en WHILE puede simularse en GOTO. Esta idea es la base de la equivalencia de ambos lenguajes, la cual presentamos en la siguiente sección

7. EQUIVALENCIAS

Nos ocupamos ahora en mostrar que todos los modelos de cómputo discutidos previamente son equivalentes. Para esto nos serviremos ampliamente del siguiente concepto de compilador.

Definición 13. *Sean* $\mathcal{S} = \langle \mathcal{P}_{\mathcal{S}}, \mathcal{D}_{\mathcal{S}}, [\![\cdot]\!]_{\mathcal{S}} \rangle$ *y* $\mathcal{T} = \langle \mathcal{P}_{\mathcal{T}}, \mathcal{D}_{\mathcal{T}}, [\![\cdot]\!]_{\mathcal{T}} \rangle$ *dos lenguajes de programación. Un compilador de* $\mathcal{S}$ *en* $\mathcal{T}$ *es un par de funciones* $\mathcal{C} = \langle F, c \rangle$ *tal que* $F : \mathcal{P}_{\mathcal{S}} \to \mathcal{P}_{\mathcal{T}},\ c : \mathcal{D}_{\mathcal{S}} \to \mathcal{D}_{\mathcal{T}}$ *y se cumple que*

$$\forall p \in \mathcal{P}_{\mathcal{S}} \forall x \in \mathcal{D}_{\mathcal{S}} \big(c([\![p]\!]_{\mathcal{S}}(x)) = [\![F(p)]\!]_{\mathcal{T}}(c(x)) \big)$$

o equivalentemente,

$$\forall p \in \mathcal{P}_{\mathcal{S}} \forall x \in \mathcal{D}_{\mathcal{S}} \forall y \in \mathcal{D}_{\mathcal{T}} \big(\ [\![p]\!]_{\mathcal{S}}(x) = y \ \textit{si y sólo si}\ [\![F(p)]\!]_{\mathcal{T}}(c(x)) = c(y) \big)$$

Para mostrar las equivalencias necesarias basta entonces con construir compiladores entre cualesquiera dos lenguajes.

7.1. De WHILE a GOTO. En esta sección describimos a detalle la idea vislumbrada en el ejemplo 9 acerca de la simulación de un ciclo `while` usando el condicional y el salto en GOTO. Antes de esto es adecuado restringir a WHILE de forma que cada expresión contenga sólo una presencia de operador como en GOTO.

Definición 14. *El lenguaje* WHILE1 *se define como la restricción de la sintaxis del lenguaje* WHILE *de manera que los únicos programas sintácticamente válidos son aquellos donde cada expresión contiene un único operador. Siendo la semántica idéntica a la definida para* WHILE.

Esta restricción no causa una pérdida de expresividad de acuerdo a la siguiente

Proposición 3. *Cualquier programa p en* WHILE *es equivalente a un programa en* WHILE1.

Demostración. Basta transformar instrucciones que contengan expresiones con más de un operador mediante el uso de variables auxilares, por ejemplo $X := \texttt{cons}\, X(\texttt{hd}\, Y)$ se convierte en $W := \texttt{hd}\, Y; X := \texttt{cons}\, X\ W$. Los detalles se dejan como ejercicio al lector. □

Por lo anterior para construir un compilador de WHILE a GOTO, basta construir un compilador de WHILE1 a GOTO, tarea que realizamos a continuación.

Definición 15. *Sea C un comando del lenguaje* WHILE1 *y ℓ una etiqueta. Definimos la secuencia de comandos etiquetados del lenguaje* GOTO $\widehat{\ell} : C^\star$, *donde $\widehat{\ell}$ es la etiqueta de la primera instrucción en $C^\star$, como sigue:*

- *Si $C =_{def} X := e$ entonces $\widehat{\ell} : C^\star =_{def} X := e$.*
- *Si $C =_{def} \texttt{while}\, X\, \texttt{do}\ D_1; \ldots; D_k\, \texttt{end}$ entonces*

$$\begin{aligned} C^\star \ =_{def}\ & \widehat{\ell} : \texttt{if}\, X\, \texttt{then goto}\, \widehat{\ell}+1\, \texttt{else goto}\, \widehat{\ell_{k+1}}+1; \\ & \ell+1 : D_1^\star; \\ & \widehat{\ell_2} : D_2^\star; \\ & \vdots \\ & \widehat{\ell_k} : D_k^\star \\ & \widehat{\ell_{k+1}} : \texttt{goto}\, \ell; \end{aligned}$$

El valor exacto de cada etiqueta $\widehat{j}$ en la definición anterior debe definirse sólamente hasta considerar a la secuencia $\widehat{j} : C^\star$ como parte de un programa completo. Intuitivamente dado el comando C la etiqueta inicial es ℓ si C está en la ℓ-ésima linea en el programa WHILE1 p en consideración. En tal caso $\widehat{\ell}$ es la etiqueta correspondiente a la primera instrucción de la secuencia $C^\star$ en el programa GOTO $\bar{p}$ obtenido a partir de p al aplicar la transformación anterior, en forma secuencial, a todos los comandos de p. Esta idea se formaliza en la siguiente definición.

Definición 16. *Sea $p =_{def} \texttt{read}\, X; C_1; \ldots; C_m; \texttt{write}\, Y$ un programa* WHILE1. *Definimos el programa p_G en* GOTO *como:*

$$p_G =_{def} \texttt{read}\, X; 1 : C_1^\star; \widehat{\ell_2} : C_2^\star; \ldots; \widehat{\ell_m} : C_m^\star; \texttt{write}\, Y$$

donde $\widehat{\ell_2} = |C_1^\star| + 1$ y $\widehat{\ell_{j+1}} = \widehat{\ell_j} + |C_j^\star|$ para $1 < j < m$.

Veamos ahora un ejemplo.

Ejemplo 10. *El siguiente es el programa* GOTO *para la suma de dos numerales obtenido a partir del programa del ejemplo 7 según la definición anterior.*

```
read XY;
 1: X:= hd XY;
 2: Y:= tl XY;
 3: if X then goto 4 else goto 7;
 4: Y:= cons nil Y;
 5: X:= tl X;
 6: goto 3;
write Y.
```

Por último mostramos la existencia de un compilador.

7.1.1. Compilación. Dado que las funciones semánticas de WHILE1 y GOTO se definen mediante una semántica operacional, la condición de compilación será consecuencia de un lema de simulación entre dichas semánticas.

Definición 17. *Sea* $s = \langle \ell, \sigma \rangle$ *un estado en* WHILE1*. Definimos el estado* $\bar{s}$ *en* GOTO *como* $\bar{s} =_{def} \langle \widehat{l}, \sigma \rangle$

Lema 1 (Simulación)**.** *Si* $p \rhd_{\text{WHILE1}} s \to s'$ *entonces* $p_G \rhd_{\text{GOTO}} \bar{s} \to^\star \bar{s'}$

Demostración. Por inducción sobre la relación $p \rhd_{\text{WHILE1}} s \to s'$. □

Proposición 4. *Sea* $\mathcal{C} = \langle F, c \rangle$ *con* $F : \mathcal{P}_{\text{WHILE1}} \to \mathcal{P}_{\text{GOTO}}$ *tal que* $F(p) = p_G$ *y* $c = id$ *la función identidad.* $\mathcal{C}$ *es un compilador de* WHILE1 *a* GOTO*.*

Demostración. Si $[\![p]\!]_{\text{WHILE1}}(x) = y$ basta ver que $[\![p_G]\!]_{\text{GOTO}}(x) = y$ puesto que c es la función identidad, pero esto es consecuencia inmediata del lema de simulación. □

7.2. De GOTO **a** WHILE**.** Esta implicación es de interés por si sola, pues nos proporcionará una prueba directa del llamado teorema de la programación estructurada de Böhm-Jacopini [1] que, a diferencia de la original, no recurre a diagramas de flujo. La idea es modelar las etiquetas de un programa GOTO mediante el uso de una variable contador en WHILE. En esta sección hacemos uso de la instrucción `case` definible facilmente en WHILE.

Definición 18 (Transformación de Böhm-Jacopini)**.** *Dada una instrucción etiquetada* $\ell : \mathcal{I}_\ell$ *del lenguaje* GOTO*. Definimos la secuencia de comandos* $\mathcal{I}_\ell^\star$ *del lenguaje* WHILE *de acuerdo a los siguientes casos:*

- *Si* $\mathcal{I}_\ell =_{def} X := e$ *entonces* $\mathcal{I}_\ell^\star =_{def} X := e; PC := \ell + 1$
- *Si* $\mathcal{I}_\ell =_{def}$ `if` X `then goto` ℓ' `else goto` ℓ'' *entonces*

$$\mathcal{I}_\ell^\star =_{def} \texttt{if}\ X\ \texttt{then}\ PC := \ell'\ \texttt{else}\ \ PC := \ell''$$

- *Si* $\mathcal{I}_\ell =_{def}$ `goto` ℓ' *entonces* $\mathcal{I}_\ell^\star =_{def} PC := \ell'$

donde PC es una variable nueva, es decir, una variable que no figura en $\mathcal{I}_\ell$ y cuyas presencias en $\mathcal{I}^\star_\ell$ son únicamente las mostradas en la definición.

Definición 19. *Sea $p =_{def}$ `read` $X; 1 : \mathcal{I}_1; \ldots; m : \mathcal{I}_m;$ `write` Y un programa en* GOTO. *Definimos el programa p_W en* WHILE *como:*

$$
\begin{array}{rl}
p_W \;=_{def}\; & \texttt{read}\, X; \\
& PC := 1; \\
& \texttt{while}\; PC\; \texttt{do} \\
& \quad \texttt{case}\; PC\; \texttt{of} \\
& \qquad 1 \Rightarrow \mathcal{I}^\star_1; \\
& \qquad \vdots \\
& \qquad m \Rightarrow \mathcal{I}^\star_m; \\
& \qquad \texttt{otherwise} \Rightarrow PC := 0; \\
& \texttt{end}; \\
& \texttt{write}\, Y
\end{array}
$$

Ejemplo 11. *Veamos el resultado de aplicar la transformación de Böhm-Jacopini al programa* `reverse` *del ejemplo 9 para la reversa.*

```
read X;
  PC := 1;
  while PC do
   case PC of
    1 => Y:=nil;PC:=2;
    2 => if X then PC:=3 else PC:=7;
    3 => Z:=hd x; PC:=4;
    4 => Y:=cons Z Y;PC:=5;
    5 => X:=tl X; PC:= 6;
    6 => PC:=2;
    otherwise => PC:=0
  end
write Y
```

7.2.1. Compilación. Veamos ahora la existencia del compilador correspondiente.

Definición 20. *Sean $p =$ `read` $X, 1 : \mathcal{I}_1, \ldots, m : \mathcal{I}_m,$ `write` Y un programa y $s = \langle \ell, \sigma \rangle$ un estado en* GOTO. *Definimos el estado $\bar{s}$ en* WHILE *como $\bar{s} =_{def} \langle \widetilde{\ell}, \widetilde{\sigma} \rangle$ donde $\widetilde{1} = 4$, $\widetilde{\ell + 1} = \widetilde{\ell} + |\widetilde{\mathcal{I}_\ell}|$ para $1 < \ell \leq m$ y $\widetilde{\sigma} = \sigma[PC/n]$ donde n es un valor único que queda determinado hasta el momento de la ejecución.*

Lema 2 (Simulación)**.** *Si $p \rhd_{\text{GOTO}} s \to s'$ entonces $p_W \rhd_{\text{WHILE}} \bar{s} \to^\star \bar{s'}$*

Demostración. Por inducción sobre la relación $p \rhd_{\text{GOTO}} s \to s'$. □

Proposición 5. *Sea $\mathcal{C} = \langle F, c \rangle$ con $F : \mathcal{P}_{\text{GOTO}} \to \mathcal{P}_{\text{WHILE1}}$ tal que $F(p) = p_W$ y $c = id$ es la función identidad. $\mathcal{C}$ es un compilador de* GOTO *a* WHILE1

Demostración. Si $[\![p]\!]_{\text{GOTO}}(x) = y$ basta ver que $[\![p_W]\!]_{\text{WHILE1}}(x) = y$ puesto que c es la función identidad, pero esto es consecuencia inmediata del lema de simulación. □

Ahora podemos obtener de manera directa uno de los resultados más relevantes en la teoría de lenguajes de programación.

Teorema 1 (de la Programación Estructurada (Böhm-Jacopini))**.** *Cualquier programa no estructurado puede implementarse en un lenguaje estrictamente secuencial que contenga las siguientes instrucciones: secuencia (;), condicional (*`if`*) e iteración (*`while`*)*

Demostración. Es consecuencia inmediata de la equivalencia entre WHILE y GOTO. □

Pasemos ahora a discutir el proceso de transformación de un programa en LTURING al lenguaje GOTO.

7.3. De LTURING a GOTO. Para este caso necesitamos codificar los datos de LTURING como datos de GOTO de acuerdo a la siguiente

Definición 21. *La función de codificación* $(\cdot)^\dagger : \{0,1,\sqcup\}^\star \to \mathbb{D}$ *se define como* $w^\dagger = s_1^\dagger s_2^\dagger \dots s_n^\dagger$. *donde* $w = s_1 s_2 \dots s_n$ *y*

$$\sqcup^\dagger = nil,\ 0^\dagger = nil.nil,\ 1^\dagger = nil.(nil.nil)$$

La transformación se basa en la idea de simular una memoria $\sigma = L\underline{s}R$ de LTURING, mediante una memoria con tres variables Rt, Lf, C en GOTO de manera que C almacena al (código del) símbolo actual s, Lf a la cadena izquierda L y Rt a la cadena derecha R.

Definición 22. *Dada una instrucción etiquetada* $\ell : \mathcal{I}_\ell$ *en* LTURING*, definimos la secuencia de instrucciones etiquetadas* $\widehat{\ell} : \widetilde{\mathcal{I}}_\ell$ *en* GOTO*, donde* $\widehat{\ell}$ *es la etiqueta de la primera instrucción de* $\widetilde{\mathcal{I}}_\ell$*, como sigue:*

- *Si* $\mathcal{I}_\ell =_{def}$ `right` *entonces*

$$\begin{aligned}
\widehat{\ell} : \widetilde{\mathcal{I}}_\ell \ =_{def}\ & \widehat{\ell} : A_1 := nil \\
& \widehat{\ell}+1 : A_2 := (=?\ Rt\ A_1) \\
& \widehat{\ell}+2 : \texttt{if}\ A_2\ \texttt{then goto}\ \widehat{\ell}+3\ \texttt{else goto}\ \widehat{\ell}+6 \\
& \widehat{\ell}+3 : Lf := \texttt{cons}\ C\ Lf \\
& \widehat{\ell}+4 : C := \sqcup \\
& \widehat{\ell}+5 : \texttt{goto}\ \widehat{l+1} \\
& \widehat{\ell}+6 : Lf := \texttt{cons}\ C\ Lf \\
& \widehat{\ell}+7 : C := \texttt{hd}\ Rt \\
& \widehat{\ell}+8 : Rt := \texttt{tl}\ Rt
\end{aligned}$$

  donde las variables A_1, A_2 *son nuevas.*
- *Si* $\mathcal{I}_\ell =_{def}$ `left` *entonces* $\widehat{\ell} : \widetilde{\mathcal{I}}_\ell$ *se define análogamente al caso anterior.*

- *Si* $\mathcal{I}_\ell =_{def}$ `write` s *entonces* $\widehat{\ell} : \widetilde{\mathcal{I}}_\ell =_{def} \widehat{\ell} : C := s$
- *Si* $\mathcal{I}_\ell =_{def}$ `if` s `then goto` ℓ' `else goto` ℓ'' *entonces*

$$\begin{array}{rcl} \widehat{\ell} : \widetilde{\mathcal{I}}_\ell & =_{def} & \widehat{\ell} : A_1 := s \\ & & \widehat{\ell} + 1 : A_2 := (=?\ C\ A_1) \\ & & \widehat{\ell} + 2 : \texttt{if}\ A_2\ \texttt{then goto}\ \widehat{\ell'}\ \texttt{else goto}\ \widehat{\ell''} \end{array}$$

  donde las variables A_1, A_2 *son nuevas.*
- *Si* $\mathcal{I}_\ell =_{def}$ `goto` ℓ' *entonces* $\widetilde{\mathcal{I}}_\ell =_{def}$ `goto` $\widehat{\ell'}$

Obsérvese que dada ℓ la etiqueta $\widehat{\ell}$ no ha sido definida pues depende de un programa en consideración y se calcula de acuerdo a la siguiente definición.

Definición 23. *Dado un programa* $p =_{def} 1 : \mathcal{I}_1, 2 : \mathcal{I}_2, \ldots, m : \mathcal{I}_m; m+1 :$ `halt` *en* LTuring *definimos el programa* p_G *en* Goto *como sigue:*

$$p_G =_{def} \texttt{read}\, Rt; 1 : \widetilde{\mathcal{I}_1}, \widehat{2} : \widetilde{\mathcal{I}_2}, \ldots, \widehat{m} : \widetilde{\mathcal{I}_m}; \texttt{write}\, Rt$$

donde $\widehat{2} = 1 + |\widetilde{\mathcal{I}_1}|$ *y* $\widehat{j+1} = |\widetilde{\mathcal{I}_j}| + \widehat{j},\ 1 < j < m.$

Dejamos como ejercicio obtener el programa $\texttt{swap}_G$ a partir del programa `swap` del ejemplo 2.

7.3.1. Compilación. Veamos ahora la existencia del compilador correspondiente.

Definición 24. *Dado un estado* $s =_{def} \langle \ell, L\underline{s}R \rangle$ *en* LTuring *definimos el estado* $\bar{s}$ *en* Goto *como* $\bar{s} =_{def} \langle \widehat{\ell}, \sigma \rangle$ *donde*

$$\sigma =_{def} [Lf \mapsto (L^{rev})^\dagger,\ C \mapsto s^\dagger,\ Rt \mapsto R^\dagger]$$

donde L^{rev} *denota a la reversa de la cadena* L*. ¿Por qué es esto necesario?*

Lema 3 (Simulación)**.** *Si* $p \rhd_{\text{LTuring}} s \to s'$ *entonces* $p_G \rhd_{\text{Goto}} \bar{s} \to^\star \bar{s'}$

Demostración. Por inducción sobre la relación $p \rhd_{\text{LTuring}} s \to s'$. □

Proposición 6. *Sea* $\mathcal{C} = \langle F, c \rangle$ *con* $F : \mathcal{P}_{\text{LTuring}} \to \mathcal{P}_{\text{Goto}},\ F(p) = p_G$ *y* c*.* $\mathcal{C}$ *es un compilador de* LTuring *a* Goto

Demostración. Si $[\![p]\!](x) = y$ entonces $p \rhd_{LTuring} (1, \underline{\sqcup} x) \to^\star (m+1, L\underline{s}\, y)$ y por el lema de simulación tenemos que $p_G \rhd_{Goto} (1, \sigma_0) \to^\star (\widehat{m+1}, \sigma_f)$ donde $\sigma_0 = [Lf \mapsto nil,\ C \mapsto nil,\ Rt \mapsto x^\dagger]$ y $\sigma_f = [Lf \mapsto (L^{rev})^\dagger,\ C \mapsto s^\dagger,\ Rt \mapsto y^\dagger]$, lo cual implica que $[\![p_G]\!](x^\dagger) = y^\dagger$. □

Nuestra última transformación cierra el círculo de implicaciones, constata la equivalencia entre $\mathcal{MT}$ y While y resulta ser un ejemplo interesante del uso de máquinas de Turing multicinta.

7.4. De GOTO a $\mathcal{MT}$. En esta sección bosquejamos el proceso de simulación de un programa GOTO mediante una máquina de Turing multicinta que incluye una cinta por cada variable del programa. Esto completa la equivalencia entre WHILE y $\mathcal{MT}$, dado que es bien sabido que las máquinas de Turing multicinta son equivalentes a aquellas con una sóla cinta, ver [4, 2] por ejemplo.

Definición 25 (Codificación)**.** *Sea* $\Sigma = \{0, (,), \cdot, \sqcup\}$ *El conjunto de datos de* GOTO*,* $\mathcal{D}_{Goto} = \mathbb{T}$ *se codifica en* Σ *mediante la función* $(\cdot)^{\ddagger} : \mathbb{D} \to \Sigma^{\star}$ *como sigue:*

- $nil^{\ddagger} = 0$
- $(x.y)^{\ddagger} = (x^{\ddagger} \,.\, y^{\ddagger})$

Por ejemplo el árbol $(nil.(nil.nil))$ se codifica con la cadena $(0.(0.0))$

La máquina de Turing multicinta correspondiente a un programa GOTO utiliza la codificación anterior. Antes de dar la definición es necesario definir algunos macros.

Proposición 7. *Sea* M *una máquina de Turing multicinta cuyo alfabeto codifica expresiones del lenguaje* GOTO*. Los siguientes macros son implementables.*

1. `C.E`*: mover la cabeza de todas las cintas al primer blanco a la izquierda de la cadena actual.*
2. `copy C to D`*: copia a la cinta* `D` *el contenido de la cinta* `C`*.*
3. `erase` C*: borra el contenido de la cinta* C*.*
4. `compute e in C`*: calcula el valor de la* GOTO*-expresión* e *(codificada), escribiendo el resultado en la cinta* C
5. `if C then goto q1 else goto q2` *: verifica si el contenido de la cinta* C *no es 0 (el código de nil) en cuyo caso se cambia el estado a* `q1` *y en caso contrario se cambia el estado a* `q2`*.*
6. `goto q`*: cambia el estado a* `q`

Demostración. Ejercicio. □

Ahora ya podemos definir la máquina de Turing multicinta que simule a un programa GOTO.

Definición 26. *Sea* $p = \texttt{read}\, X; 1 : \mathcal{I}_1, \ldots, m : \mathcal{I}_m; \texttt{write}\, Y$ *un programa en* GOTO *tal que* $Vars(p) = \{X, Y, Z_1, \ldots, Z_n\}$*. Definimos la máquina de Turing multicinta* $T_p = \langle \Sigma, Q, q_1, q_f, \delta \rangle$ *como sigue:*

- *Existen* $n+3$ *cintas, una por cada variable de* p*, denotada exactamente igual, más una cinta auxiliar denotada* $\mathcal{A}$*.*
- $\Sigma = \{(,), \cdot, 0, \sqcup\}$
- $Q = \{q_1 \ldots, q_m, q_f\} \cup Q'$ *donde existe un estado* q_ℓ *para cada instrucción* $\mathcal{I}_\ell$ *del programa* GOTO *y* Q' *es un conjunto de estados auxiliares utilizados únicamente en los macros de la proposición 7*

- *El estado inicial es q_1 y el estado final es q_f.*
- *Las acciones de la máquina al llegar a un estado correspondiente a una instruccion del programa* GOTO *son las siguientes:*
  - *Si $\mathcal{I}_\ell =_{def} X := e$ entonces*
    q_ℓ*:* `compute e in` $\mathcal{A}$*;* `erase` X*;* `copy` $\mathcal{A}$ `to X;` `C.E;` `goto` $q_{\ell+1}$
  - *Si $\mathcal{I}_\ell =_{def}$* `if` X `then goto` ℓ' `else goto` ℓ'' *entonces*
    q_ℓ*:* `compute X in` $\mathcal{A}$*;* `C.E;` `if` $\mathcal{A}$ `then goto` $q_{\ell'}$ `else goto` $q_{\ell''}$
  - *Si $\mathcal{I}_\ell =_{def}$* `goto` $q_{\ell'}$ *entonces*
    q_ℓ : `goto` $q_{\ell'}$

Ejemplo 12. *La máquina $T_{\texttt{reverse}}$ asociada al programa del ejemplo 9 se describe como sigue:*

```
q1: compute nil in A;erase Y;copy A to Y;C.E.;goto q2
q2: if X then goto q3 else goto q7
q3: compute hd X in A;erase Z;copy A to Z;C.E.;goto q4
q4: compute cons Z Y in A;erase Y;copy A to Y;C.E.;goto q5
q5: compute tl X in A;erase X;copy A to X;C.E.;goto q6
q6: goto q2
```

La correctud de esta transformación se sigue de la siguiente

Proposición 8. *Si $[\![p]\!](x) = y$ entonces $y \in L(T_p)$. Más aún, si la tupla $(q, w_X, w_Y, w_{Z_1}, \ldots, w_{Z_n}, w_{\mathcal{A}})$ es una configuración de T_p indicando que T_p se encuentra en el estado q siendo w_V el contenido de la cinta correspondiente a cada variable V o a la cinta auxiliar $\mathcal{A}$ entonces*

$$(q_1, (x^\ddagger, 0, \ldots, 0)) \vdash^\star (q_f, (w_X, y^\ddagger, \ldots, w_{Z_n}, w_{\mathcal{A}}))$$

Demostración. Ejercicio □

Con esto terminamos de presentar las transformaciones entre los distintos lenguajes discutidos, siendo una consecuencia inmediata la siguiente proposición que es una evidencia más de la validez de la tesis de Church-Turing, es decir, de la afirmación de que cualquier formalización razonable de la noción intuitiva de cómputo efectivo es equivalente a la noción de Turing computabilidad.

Proposición 9. *Los lenguajes* WHILE*,* GOTO *y* LTURING *son equivalentes a $\mathcal{MT}$.*

Demostración. De todos los resultados de esta sección tenemos que WHILE $\Leftrightarrow$ GOTO, GOTO $\Rightarrow$ $\mathcal{MT}$ y LTURING $\Rightarrow$ GOTO. Además en la sección 4.1 mostramos que LTURING $\Leftrightarrow$ $\mathcal{MT}$. Por lo tanto también podemos concluir que GOTO $\Leftrightarrow$ $\mathcal{MT}$. □

8. Comentarios finales

En este artículo hemos presentado tres lenguajes de programación equivalentes a la Máquina de Turing, el primero, LTURING, modela de manera

directa el mecanismo operacional de una Máquina de Turing, puesto que un programa en este lenguaje es esencialmente una definición de la función de transición de una máquina. Para lograr una identificación directa de la programación con las Máquinas de Turing hemos presentado el lenguaje WHILE así como su equivalencia con éstas usando como puente el lenguaje GOTO, formalismo reminiscente de los antiguos lenguajes no estructurados como COBOL, FORTRAN o BASIC. La equivalencia entre GOTO y WHILE es de interés por si sola pues nos permitió demostrar de una manera rigurosa y sin apelar al uso de diagramas de flujo, el famoso teorema de la programación estructurada de Böhm-Jacopini [1]. Algunas cuestiones interesantes acerca de los programas WHILE, por ejemplo, la existencia y programación de un auto intérprete, contraparte de la Máquina de Turing Universal, se han omitido. Sin embargo, el escenario queda listo pues la estructura de datos escogida, los árboles binarios, resulta excelente para este propósito dado que tal intérprete debe recibir como entrada a un programa WHILE y este puede codificarse de manera directa como un árbol binario, a saber el correspondiente árbol de sintaxis abstracta del programa, ver el capítulo 3 de [3]. Para profundizar en estos temas mencionamos que es posible desarrollar las teorías de computabilidad y complejidad basandose en el lenguaje WHILE en vez de en máquinas de Turing, esto se realiza en el libro [3], el cual también sirvió de base para nuestro desarrollo aunque nosotros hemos modificado las definiciones de lenguaje de programación y compilador de manera adecuada para hacer autocontenida la exposición. Deseamos que nuestro trabajo haya cumplido con el objetivo de acercar a las Máquinas de Turing al ámbito de la teoría y práctica de la programación actual y que resulte ser una invitación a la teoría de los lenguajes de programación.

REFERENCIAS

[1] C. Böhm, G. Jacopini. Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules. Communications of the ACM 9(5):366–371, 1966. doi:10.1145/355592.365646

[2] J. E. Hopcroft, R. Motwani, J. D. Ullman. Introduction to Automata Theory, Languages, and Computation. 2nd edition. Addison-Wesley Publishing Company. 2001.

[3] N.D. Jones. Computability and Complexity From a Programming Perspective. Foundations of Computing. MIT Press 1997. Versión actualizada disponible en http://www.diku.dk/~neil/Comp2book.html

[4] John Martin. Lenguajes formales y teoria de la computacion. Tercera edición. McGraw Hill 2004.

[5] A.M. Turing. On computable numbers with an application to the *Entscheidungsproblem*. En *Proceedings of the London Mathematical Society*, 42(2):230–265, 1936-7.

(Favio E. Miranda) Departamento de Matemáticas, Facultad de Ciencias UNAM, Circuito Exterior S/N, Cd. Universitaria 04510, México D.F., México
E-mail address: favio@ciencias.unam.mx

(A. Liliana Reyes) Colegio de Ciencia y Tecnología, Universidad Autónoma de la Ciudad de México, Plantel San Lorenzo Tezonco. Prolongación San Isidro 151. San Lorenzo Tezonco, Iztapalapa, 09790 México D.F., México
E-mail address: liliana.mar@gmail.com

(Rafael Reyes) Departamento de Matemáticas, Facultad de Ciencias UNAM, Circuito Exterior S/N, Cd. Universitaria 04510, México D.F., México
E-mail address: rreyes.rs@gmail.com

(L. González Huesca) Laboratoire Preuves, Programmes et Systèmes, Equipe πr^2. INRIA 23 avenue d'Italie, CS 81321 - 75214 Paris Cedex 13, Francia
E-mail address: lgonzale@pps.univ-paris-diderot.fr