Miscelánea Matemática **56** (2013) 145–178 SMM

La máquina de Turing en el ámbito de los lenguajes de programación

Favio Ezequiel Miranda Perea
Departamento de Matemáticas, Facultad de Ciencias UNAM,
Circuito Exterior S/N, Cd. Universitaria 04510,
México D.F., México
favio@ciencias.unam.mx

Araceli Liliana Reyes Cabello
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
liliana.mar@gmail.com

Rafael Reyes Sánchez
Departamento de Matemáticas, Facultad de Ciencias UNAM,
Circuito Exterior S/N, Cd. Universitaria 04510,
México D.F., México
rreyes.rs@gmail.com

Lourdes del Carmen González Huesca
Laboratoire Preuves, Programmes et Systèmes,
Equipe πr^2. INRIA 23 avenue d'Italie,
CS 81321 - 75214 Paris Cedex 13, Francia
lgonzale@pps.univ-paris-diderot.fr

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 esta protagonista de las teorías de

computabilidad y complejidad computacional a un ámbito más familiar para el estudiante de computación. Para lo cual utilizaremos técnicas e instrucciones de programación usuales, como el enunciado de asignación o la iteración, así como conceptos de teoría de lenguajes de programación. Se enfatizará la equivalencia entre la máquina de Turing, los lenguajes estructurados y los no estructurados. De esta manera obtenemos como resultado no solo 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 de Turing. Si bien esto acerca el maravilloso invento de Alan Turing al ámbito de la programación, este lenguaje aún 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

En términos generales podemos decir que una máquina de Turing es un dispositivo capaz de procesar cualquier clase de datos, codificados mediante palabras o cadenas formadas por símbolos dados. Esta máquina abstracta consta de una cinta infinita hacia ambos lados que está divida en casillas o celdas, las cuales contienen a lo más un símbolo (ver figura 1); también tiene una cabeza de lectura y escritura que se mueve a lo largo de la cinta. Su funcionamiento es secuencial y está dado por la llamada función de transición que se representa mediante una tabla finita de reglas, llamadas de ejecución o transición, que describen la acción que debe ejecutarse considerando el estado actual y el símbolo escrito en la celda donde se encuentra la cabeza, con lo cual se determinan el nuevo estado, el símbolo que se debe escribir y el desplazamiento de la cabeza hacia alguna de las celdas vecinas.

Al principio, la máquina se encuentra en un estado inicial fijo, la cadena o dato de entrada se asume escrito en algún lugar de la cinta y la cabeza se encuentra una celda a la izquierda del primer símbolo del mismo. Cabe resaltar que en todas las celdas de la cinta, excepto aquellas que contienen la cadena, se encuentra escrito el símbolo especial $\sqcup$, llamado *blanco*, por lo tanto, al inicio de la ejecución la cabeza estará leyendo un blanco. La ejecución es dirigida por la función de transición y termina cuando no se ha definido ninguna regla para el estado y símbolo actual, es decir cuando no es posible ejecutar una acción

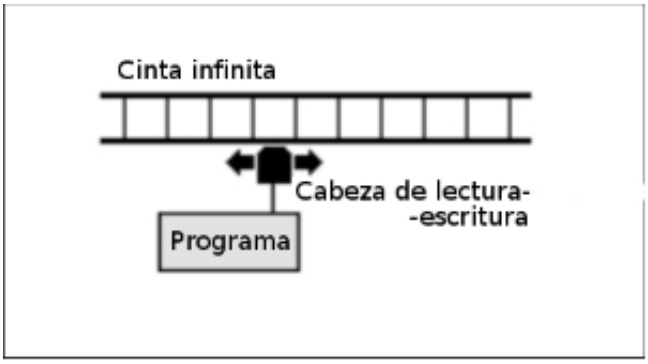

Figura 1:

más. Decimos que una cadena es aceptada por la máquina cuando la ejecución de esta termina y se encuentra en el estado distinguido como final. Adicionalmente la cadena final escrita sobre la máquina puede considerarse como la respuesta o dato de salida del proceso.

Con respecto a los símbolos permitidos en la cinta, en este artículo consideramos únicamente al siguiente alfabeto $\Sigma = \{0, 1, \sqcup\}$. Al conjunto de todas las cadenas que se pueden formar sobre un alfabeto Σ, incluyendo a la cadena vacía, lo denotaremos por Σ^*, por ejemplo $0101110 \sqcup 0 \in \Sigma^*$. No obstante, las cadenas de entrada solo se forman con 0 y 1, siendo el símbolo blanco un auxiliar para la ejecución. La aparente restricción a este alfabeto simple solo es aparente pues está demostrado que cualquier dato puede ser codificado de esta manera. Piense el lector que por más sofisticada sea su computadora actual, su funcionamiento básico es a partir de instrucciones de encendido (1) o apagado (0).

Ahora estamos listos para establecer la definición formal 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,*
- δ *es una función de transición cuyo dominio es un subconjunto de* $Q \times \Sigma$ *y cuyo contradominio es* $Q \times \Sigma \times M$*. De tal forma que si* δ *está definida*[1] *para el par* $(\ell, s) \in Q \times \Sigma$ *y* $\delta(\ell, s) = (\ell', s', m)$ *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 sea 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 instantánea 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*

$$\langle q, ls'\underline{s}r \rangle \vdash \langle p, l\underline{s'}sr \rangle$$

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 relación obtenida al componer la relación* $\vdash$ *un número finito de veces. Es decir, si* c, c' *son dos configuraciones instantáneas entonces* $c \vdash^\star c'$ *si y solo si existen configuraciones instantáneas* $c_0, \ldots, c_n$ *tales que* $c_0 = c$*,* $c_n = c'$ *y* $c_0 \vdash c_1, c_1 \vdash$

[1] El lector experimentado observará entonces que δ es una función parcial con dominio $Q \times \Sigma$, lo cual se suele denotar como $\delta : Q \times \Sigma \rightharpoonup Q \times \Sigma \times M$

$c_2, \ldots, c_{n-1} \vdash c_n$. *Obsérvese que en particular se cumple* $c \vdash^\star c$ *para cualquier configuración instantánea* c. [2]

El lenguaje aceptado por una máquina de Turing es intuitivamente el conjunto de todas las cadenas cuya ejecución termina en el estado final de la máquina y se define formalmente 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*

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

Ejemplo 4. *La siguiente máquina de Turing* $T = \langle \{0, 1, \sqcup\}, \{0, 1, 2, 3\}, 0, 3, \delta \rangle$ *verifica que existe 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. Verifiquemos, por ejemplo que* $\langle 0, \sqcup 01011 \rangle \vdash^\star \langle 3, \sqcup 01011 \sqcup \rangle$ *y por lo tanto* $01011 \in L(T)$. *En la figura 1 en vez de usar las configuraciones directamente apelamos a la idea intuitiva de la cinta donde la flecha indica el sector donde está la cabeza de la máquina.*

Una vez expuestas las nociones necesarias de máquinas de Turing pasamos a dar la definición particular de lenguaje de programación que utilizaremos en este artículo.

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

[2] El lector experimentado podrá observar que $\vdash^\star$ es simplemente la cerradura reflexiva y transitiva de la relación $\vdash$.

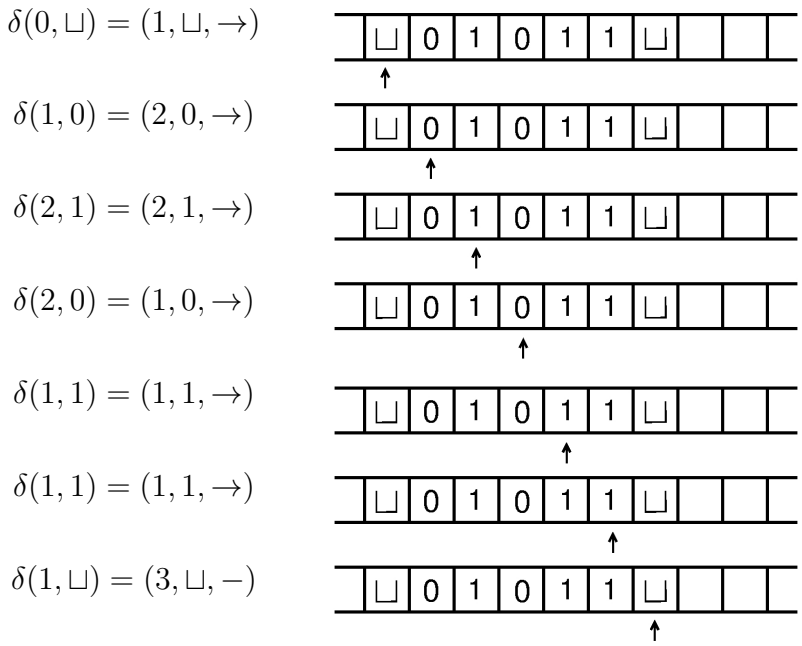

Figura 1.

- $\mathcal{P}_\mathcal{L} \neq \varnothing$ *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* $\mathcal{L}$*, que recibe un programa* $p \in \mathcal{P}_\mathcal{L}$ *cuya imagen* $[\![p]\!]_\mathcal{L}$ *es nuevamente una función, definida en un subconjunto del conjunto de datos* $\mathcal{D}_\mathcal{L}$ *y cuya imagen también está en* $\mathcal{D}_\mathcal{L}$*; de tal forma que para cualquier* $x \in \mathcal{D}_\mathcal{L}$*, si* $[\![p]\!]_\mathcal{L}(x)$ *está definida entonces* $[\![p]\!]_\mathcal{L}(x) \in \mathcal{D}_\mathcal{L}$ *es el resultado de la ejecución del programa* p *al recibir* x *como dato de entrada.*[3]

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

[3]El lector experimentado reconocerá que $[\![\cdot]\!]_\mathcal{L}$ es una función total cuyo dominio es el conjunto de programas $\mathcal{P}_\mathcal{L}$ y cuyo contradominio es el conjunto de funciones parciales de datos en datos. Es decir, $[\![\cdot]\!]_\mathcal{L} : \mathcal{P}_\mathcal{L} \rightarrow (\mathcal{D}_\mathcal{L} \rightharpoonup \mathcal{D}_\mathcal{L})$

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, donde el significado de una instrucción en un lenguaje de programación se da mediante el cómputo inducido al ejecutarla en una máquina abstracta como lo es una máquina de Turing. El concepto exacto de semántica operacional que usaremos es reminiscente de la definición de autómata finito o máquina de Turing y es el siguiente.

Definición 6. *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*[4] *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 intencional es*

  $$p \rhd s \to s' \text{ si y solo si la ejecución del programa } p \in \mathcal{P}_{\mathcal{L}} \text{ causa la transición del estado } s \text{ al estado } s'$$

- *$\mathsf{final} : \mathcal{P}_{\mathcal{L}} \to \mathcal{P}(\mathcal{E})$, donde $\mathcal{P}(\mathcal{E})$ es el conjunto potencia de $\mathcal{E}$, es la función que define estados finales. Es decir, un estado $s \in \mathcal{E}$ es final para el programa p si y solo 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.*

Se observa que nuestra definición involucra no solo un concepto de estado, como es el caso en las máquinas de Turing clásicas, sino

[4] En inglés *stores*

también un concepto de memoria a partir del cual se definen los estados. Esto es debido a que, como veremos más adelante, trabajaremos con lenguajes imperativos cuya semántica involucra un manejo explícito de la memoria de una máquina, modelada aquí mediante nuestra noción abstracta de memoria.

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

Definición 7. *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:*

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

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 a la composición de la relación $p \rhd \cdot \to \cdot$ consigo misma un número finito de veces. Es decir $p \rhd s \to s'$ si y solo si existen estados $s_1, s_2, \ldots, s_n$ tales que $s_1 = s$, $s_n = s'$ y $p \rhd s_1 \to s_2, \ldots, p \rhd s_{n-1} \to s_n$.[5]

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 por medio de un programa.

Definición 8. *El lenguaje* LTuring *se define como sigue:*

- *Datos:* $\mathcal{D}_{\mathcal{L}} = \{0, 1, \sqcup\}^\star$
- *Programas: un programa $p \in \mathcal{P}_{\mathcal{L}}$ es una secuencia finita de instrucciones etiquetadas:*

$$\begin{aligned} 1 &: \mathcal{I}_1; \\ 2 &: \mathcal{I}_2; \\ &\vdots \\ m &: \mathcal{I}_m; \\ m+1 &: \texttt{halt} \end{aligned}$$

[5] Nuevamente estamos ante una cerradura reflexiva y transitiva, en este caso la de la relación binaria $p \rhd \cdot \to \cdot$.

donde cada $\mathcal{I}_k$ *es una de las siguientes sentencias:* `right`, `left`, `goto` ℓ, `write` s, *o bien,* `if` s `then goto` ℓ `else goto` ℓ', *con* $s \in \{0, 1, \sqcup\}$ *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* $p = 1 : \mathcal{I}_1; \ldots; m : \mathcal{I}_m; m+1 :$ `halt`.
    - *Si* $\mathcal{I}_\ell =$ `right` *entonces* $p \rhd \langle \ell, L\underline{s}s'R\rangle \to \langle \ell + 1, Ls\underline{s'}R\rangle$
    - *Si* $\mathcal{I}_\ell =$ `right` *entonces* $p \rhd \langle \ell, L\underline{s}\rangle \to \langle \ell + 1, Ls\underline{\sqcup}\rangle$
    - *Si* $\mathcal{I}_\ell =$ `left` *entonces* $p \rhd \langle \ell, Ls\underline{s'}R\rangle \to \langle \ell + 1, L\underline{s}s'R\rangle$
    - *Si* $\mathcal{I}_\ell =$ `left` *entonces* $p \rhd \langle \ell, \underline{s}R\rangle \to \langle \ell + 1, \underline{\sqcup}sR\rangle$
    - *Si* $\mathcal{I}_\ell =$ `write` s *entonces* $p \rhd \langle \ell, L\underline{s'}R\rangle \to \langle \ell + 1, L\underline{s}R\rangle$
    - *Si* $\mathcal{I}_\ell =$ `goto` ℓ' *entonces* $p \rhd \langle \ell, L\underline{s}R\rangle \to \langle \ell', L\underline{s}R\rangle$
    - *Si* $\mathcal{I}_\ell =$ `if` s `then goto` ℓ' `else goto` ℓ'' *entonces* $p \rhd \langle \ell, L\underline{s}R\rangle \to \langle \ell', L\underline{s}R\rangle$
    - *Si* $\mathcal{I}_\ell =$ `if` s `then goto` ℓ' `else goto` ℓ'' *entonces* $p \rhd \langle \ell, L\underline{s'}R\rangle \to \langle \ell'', L\underline{s'}R\rangle$, *para* $s \neq s'$.
  - *Estados finales:* $\mathsf{final}(p) = \{\langle m+1, \sigma\rangle | \sigma \in \mathcal{S}\}$.
  - *Función de inicialización:* $\mathsf{init} : \mathcal{P}_{\mathcal{L}} \times \mathcal{D}_{\mathcal{L}} \to \mathcal{E}$, $\mathsf{init}(p, x) = \langle 1, \underline{\sqcup}x\rangle$
  - *Función de salida:* $\mathsf{output} : \mathcal{P}_{\mathcal{L}} \times \mathcal{E} \to \mathcal{D}_{\mathcal{L}}$, $\mathsf{output}(p, \langle \ell, L\underline{s}R\rangle) = R$

Veamos ahora un ejemplo de programación en LTURING.

Ejemplo 9. *El siguiente programa* `swap` *invierte una cadena de ceros y unos.*

```
1: right;
2: if 1 then goto 3 else goto 6
3: write 0;
4: right;
5: goto 2;
```

```
 6: if 0 then goto 7 else goto 10
 7: write 1;
 8: right;
 9: goto 2;
10: left;
11: if ⊔ then goto 12 else goto 10
12: halt
```

El lector puede verificar por ejemplo que

$$\texttt{swap} \rhd \langle 1, \sqcup 1010\rangle \to^\star \langle 11, \sqcup 0101\rangle$$

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 10 (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*

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

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

Demostración. Dada la máquina de Turing T, definiremos un programa p_T en LTuring. La idea para construir p_T es simular cada transición de la máquina de Turing mediante una secuencia de instrucciones del lenguaje LTuring. Para esto usamos la siguiente notación: si $m \in \{\to, \leftarrow\}$ entonces definimos m^g como $(\to)^g = \texttt{right}$ y $(\leftarrow)^g = \texttt{left}$. Por otra parte si $\mathcal{J}$ es una instrucción o secuencia de instrucciones en LTuring, 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.

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

  El lector debe observar que si bien la instrucción `case` no es válida en la sintaxis de LTURING, se trata de un macro definido de la manera usual a partir de instrucciones condicionales `if`.

- 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

$$\mathcal{S}_{q,s} = \texttt{write}\, s'; m^g; \texttt{goto}\, \widehat{\mathcal{I}_p}$$

  - Si $m = -$ y $s \neq s'$ entonces

$$\mathcal{S}_{q,s} = \texttt{write}\, s'; \texttt{goto}\, \widehat{\mathcal{I}_p}$$

  - Si $m \in \{\rightarrow, \leftarrow\}$ y $s = s'$ entonces

$$\mathcal{S}_{q,s} = m^g; \texttt{goto}\, \widehat{\mathcal{I}_p}$$

  - Si $m = -$ y $s = s'$ entonces

$$\mathcal{S}_{q,s} = \texttt{goto}\, \widehat{\mathcal{I}_p}$$

- Finalmente el programa p_T se define como sigue: sean $\mathcal{I}_j$ las instrucciones 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.

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

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 $\langle q_0, \sqcup x\rangle \vdash^\star \langle q_f, w \underline{s}\, y\rangle$ entonces $p \rhd \langle 1, \sqcup x\rangle \rightarrow^\star \langle \ell_{k+1}, w \underline{s}\, y\rangle$ 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 11. *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 12 (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 $p \rhd_{LTuring} \langle 1, \sqcup x\rangle \to^\star \langle m+1, w\underline{s}y\rangle$ entonces $\langle q_1, \sqcup x\rangle \vdash^\star \langle q_f, w\underline{s}y\rangle$.*

Demostración. Sea $p = 1 : \mathcal{I}_1; \ldots; m : \mathcal{I}_m; m+1 :$ `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 $\mathcal{I}_\ell$ es `right` entonces

$$\forall s \in \Sigma\big(\delta(\ell, s) = (\ell+1, s, \to)\big)$$

- Si $\mathcal{I}_\ell$ es `left` entonces

$$\forall s \in \Sigma\big(\delta(\ell, s) = (\ell+1, s, \leftarrow)\big)$$

- Si $\mathcal{I}_\ell$ es `write` s' entonces

$$\forall s \in \Sigma\big(\delta(\ell, s) = (\ell+1, s', -)\big)$$

- Si $\mathcal{I}_\ell$ es `if` s' `then goto` ℓ' `else goto` ℓ'' entonces

$$\begin{array}{rcl} \delta(\ell, s') & = & (\ell', s', -) \\ \delta(\ell, s) & = & (\ell'', s, -), \quad \text{si } s \neq s' \end{array}$$

- Si $\mathcal{I}_\ell$ es `goto` ℓ' 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 deja como ejercicio al lector. □

Veamos cómo funciona la transformación anterior en el programa correspondiente al ejemplo 9.

Ejemplo 13. *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 aquella 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 5. 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 representado mediante un árbol binario, construido a partir de un elemento atómico llamado `nil`; esta clase de árboles solo tienen etiquetas en las hojas y se genera con la siguiente definición:

Definición 14. *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* $(d_1.d_2)$ *es un elemento de* $\mathbb{D}$;
- $\mathbb{D}$ *es el conjunto más pequeño que satisface lo anterior.*

La intuición es que dados dos árboles d_1, d_2, el árbol cuyo subárbol izquierdo (derecho) es d_1 (d_2) se denota con $(d_1.d_2)$. La elección de esta estructura de datos en lugar de valores booleanos y/o números naturales usados en lenguajes similares, quedará clara más adelante. Veamos ahora cómo esta estructura es suficientemente poderosa para representar a los valores booleanos y a los números naturales.

Definición 15. *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 16. *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 $\underline{4} =$ (`nil`.(`nil`.(`nil`.(`nil`.`nil`)))). Por simplicidad escribiremos $0, 1, 2, \ldots$ como representación de $\underline{0}, \underline{1}, \underline{2}, \ldots$ o $\texttt{nil}^0$,$\texttt{nil}^1$, $\texttt{nil}^2$, ...

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

Definición 17. *El lenguaje* WHILE *se define como sigue:*

- $\mathcal{D}_{\mathcal{L}} = \mathbb{D}$.
- *Los programas de* $\mathcal{P}_{\mathcal{L}}$ *son secuencias de instrucciones definidas de la siguiente manera:*

$$\texttt{read}\, X;\, C_1, \ldots, C_m\, ;\texttt{write}\, Y$$

donde cada C_i *puede ser:*

- *Asignación:* $X := e$
- *Ciclo:* `while` e `do` $\vec{C}$ `end`

Asimismo tenemos que:

- $\vec{C}$ *representa una secuencia de comandos de la forma:* $C_1; C_2; \ldots; C_n$,
- e *y* f *representan expresiones que pueden ser:*
  - X, *una variable*
  - `d`, *una constante que denota datos de* $\mathbb{D}$
  - `cons` e f, *la operación de construcción de árboles.*
  - `hd` e, *el subárbol izquierdo de* e.
  - `tl` e, *el subárbol derecho de* e.
  - $=?\, e\ f$, *verificación de igualdad de los árboles* e, f.

  Cabe señalar que a los conjuntos de variables y expresiones del lenguaje While *se les denota por* Var *y* Expr, *respectivamente.*

Por otro lado, la longitud o número de líneas 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:* $\mathcal{S} = \{\sigma |\ \sigma : Var \to \mathbb{D}\}$, *observemos que* σ *es una función que recibe una variable y devuelve el dato almacenado en ésta.*
  - *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* $s = \langle \ell, \sigma \rangle$[6].

[6] ℓ sirve para identificar la instrucción que se está ejecutando en un momento determinado y σ es el estado de memoria en dicho momento.

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

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

    donde $\sigma[X/\mathsf{d}] : Var \to D$ *denota a la memoria que resulta al actualizar* σ *con el valor* d *para la variable* X*, coincidiendo el valor de las demás variables con el dictado por* σ.
  - *Si* $\mathcal{I}_\ell = \mathtt{while}\, e\, \mathtt{do}\ \vec{D}\ \mathtt{end}$ *con* $|\vec{D}| = k$ *y* $\mathsf{ev}_s(e) = \mathtt{nil}$*, 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* $\mathsf{ev}_s(e) \neq \mathtt{nil}$*, 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:* $\mathsf{init} : \mathcal{P}_\mathcal{L} \times \mathcal{D}_\mathcal{L} \to \mathcal{E}$*, donde si se cumple que* $p = \mathtt{read}\, X; \vec{C}; \mathtt{write}\, Y$ *y* $Vars(p) = \{X, Y, Z_1, \ldots, Z_n\}$ *se define como el conjunto de todas las variables que figuran en* p *entonces*

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

  Es decir, la inicialización de p *en el dato* d *consiste en crear un estado cuya primera componente es* 1 *y cuya segunda componente es una memoria que asigna el dato de entrada* d *a la variable de lectura* X *y el dato* $\mathtt{nil}$ *a las restantes variables que figuren en* p*, incluida la variable de escritura* Y.

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

El siguiente ejemplo tiene como fin mostrar como se pueden definir las instrucciones `if` e `if-else` en el lenguaje While.

Ejemplo 18. *La siguiente secuencia de instrucciones simula el comportamiento de una instrucción* `if`*, es decir, se ejecutará* `C` *si y solo si el resultado de la evaluación de* `E` *es* `true`*.*

```
Z := E;
while Z do
  Z := false;
  C;
end
```

De manera similar las siguientes instrucciones simulan el comportamiento de una instrucción `if-else`*, esto es, se ejecutará* `C1` *si el resultado de la evaluación de* `E` *es* `true` *y* `C2` *en caso contrario.*

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

A continuación se presentas algunos programas simples en el lenguaje While.

Ejemplo 19. *Los siguientes programas calculan el sucesor y el predecesor de un numeral, respectivamente.*

```
read X;
  Y := cons nil X;
write Y
```

```
read X;
  Y := tl X;
write Y;
```

Ejemplo 20. *El programa que se muestra a continuación recibe un dato de la forma* $(n.m)$ *y devuelve la suma* $n + m$*.*

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

Ejemplo 21. *Este programa obtiene la reversa de un elemento de* $\mathbb{D}$*, si la entrada es* $d_1.d_2.\ldots.d_n.\texttt{nil}$ *entonces la salida será* $d_n.d_{n-1}.\ldots.d_1.\texttt{nil}$*:*

```
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 que lo hace el lenguaje que se discutirá en esta sección.

Definición 22. *El lenguaje* GOTO *se define como sigue:*

- $\mathcal{D}_{\mathcal{L}} = \mathbb{D}$
- *Un programa* $p \in \mathcal{P}_{\mathcal{L}}$ *es una secuencia de instrucciones de la*

siguiente forma:

$$\begin{array}{l} \texttt{read}\, X; \\ 1 : \mathcal{I}_1; \\ 2 : \mathcal{I}_2; \\ \quad \vdots \\ m : \mathcal{I}_m; \\ \texttt{write}\, Y \end{array}$$

donde $\ell, \ell' \in \mathbb{N}$ *y cada* $\mathcal{I}_k$ *es una de las siguientes sentencias:*

- *Asignación:* $X := e$
- *Salto:* $\texttt{goto}\, \ell$
- *Salto condicional:* $\texttt{if}\, X\, \texttt{then goto}\, \ell\, \texttt{else goto}\, \ell'$.

De igual forma X *representa una variable y* e *denota una expresión que pueden ser:*

- X, *variable.*
- $\texttt{d}$, *constante.*
- $\texttt{cons}\ Y\ Z$, *construcción de un árbol.*
- $\texttt{hd}\, X$, *subárbol izquierdo de* X.
- $\texttt{tl}\, X$, *subárbol derecho de* X.
- $=?\, Y\ Z$, *verificación de igualdad de* Y, Z.

Obsérvese que, a diferencia de While, *en este lenguaje en cada expresión* e *hay exactamente una presencia de un 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* $\mathsf{ev}_s : Expr \to \mathcal{D}_{\mathcal{L}}$ *como sigue:*
  - $\mathsf{ev}_s(X) = \sigma(X)$

- $\mathsf{ev}_s(\mathsf{d}) = \mathsf{d}$
- $\mathsf{ev}_s(\mathtt{hd}\, X) = \mathtt{nil}$, *si* $\sigma(X) = \mathtt{nil}$
- $\mathsf{ev}_s(\mathtt{hd}\, X) = \mathsf{d}$, *si* $\sigma(X) = (\mathsf{d}.e)$
- $\mathsf{ev}_s(\mathtt{tl}\, X) = \mathtt{nil}$, *si* $\sigma(X) = \mathtt{nil}$
- $\mathsf{ev}_s(\mathtt{tl}\, X) = e$, *si* $\sigma(X) = (\mathsf{d}.e)$
- $\mathsf{ev}_s(\mathtt{cons}\, X\ Y) = (\mathsf{d}.e)$ *si* $\sigma(X) = \mathsf{d},\ \sigma(Y) = e$
- $\mathsf{ev}_s(=?X\ Y) = \mathtt{true}$ *si* $\sigma(X) = \sigma(Y)$.
- $\mathsf{ev}_s(=?X\ Y) = \mathtt{false}$ *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* $p = \mathtt{read}\, X; 1 : \mathcal{I}_1; \ldots; m : \mathcal{I}_m; \mathtt{write}\, Y$.
  - *Si* $\mathcal{I}_\ell = X := e$ *y* $\mathsf{ev}_{\langle \ell, \sigma \rangle}(e) = \mathsf{d}$ *entonces*
    $$p \rhd \langle \ell, \sigma \rangle \to \langle \ell + 1, \sigma[X/\mathsf{d}] \rangle$$
  - *Si* $\mathcal{I}_\ell = \mathtt{if}\, X\ \mathtt{then}\, \mathtt{goto}\ \ell'\ \mathtt{else}\, \mathtt{goto}\ \ell''$ *y* $\sigma(X) = \mathtt{nil}$ *entonces*
    $$p \rhd \langle \ell, \sigma \rangle \to \langle \ell'', \sigma \rangle$$
  - *Si* $\mathcal{I}_\ell = \mathtt{if}\, X\ \mathtt{then}\, \mathtt{goto}\ \ell'\ \mathtt{else}\, \mathtt{goto}\ \ell''$ *y* $\sigma(X) \neq \mathtt{nil}$ *entonces*
    $$p \rhd \langle \ell, \sigma \rangle \to \langle \ell', \sigma \rangle$$
  - *Si* $\mathcal{I}_\ell = \mathtt{goto}\, \ell'$ *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:* $\mathsf{init} : \mathcal{P}_\mathcal{L} \times \mathcal{D}_\mathcal{L} \to \mathcal{E}$
  $$\mathsf{init}(p, \mathsf{d}) = \big\langle 1, [X \mapsto \mathsf{d}, Z_1 \mapsto \mathtt{nil}, \ldots, Z_n \mapsto \mathtt{nil}] \big\rangle$$
  donde $p = \mathtt{read}\, X; \vec{C}; \mathtt{write}\, Y$ *y* $Vars(p) = \{X, Y, Z_1, \ldots, Z_n\}$.
- *Función de salida:* $\mathsf{output} : \mathcal{P}_\mathcal{L} \times \mathcal{E} \to \mathcal{D}_\mathcal{L},\ \mathsf{output}(p, \sigma) = \sigma(Y)$

Veamos un ejemplo de programa GOTO.

Ejemplo 23. *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 21) podemos vislumbrar un método para modelar la instrucción `while` mediante la combinación del condicional `if` y la instrucción `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 24. *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 cumplen con la siguiente especificación*

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

o equivalentemente,

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

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 23 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 solo una presencia de operador como en GOTO.

Definición 25. *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.

Ejemplo 26. *Este programa es equivalente al programa que calcula la reversa de un elemento de* $\mathbb{D}$ *visto en el ejemplo 21.*

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

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

Proposición 27. *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 28. *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 es una asignación $X := e$ entonces la instrucción $\widehat{\ell} : C^\star$ se define como $X := e$.*
- *Si C es un ciclo* `while` X `do` $D_1; \ldots; D_k$ `end` *entonces $C^\star$ es la siguiente secuencia de instrucciones:*

$$\begin{array}{l}\widehat{\ell} : \texttt{if}\, X\, \texttt{then goto}\, \widehat{\ell} + 1\, \texttt{else goto}\, \widehat{\ell_{k+1}} + 1;\\ \widehat{\ell} + 1 : D_1^\star;\\ \widehat{\ell_2} : D_2^\star;\\ \vdots\\ \widehat{\ell_k} : D_k^\star\\ \widehat{\ell_{k+1}} : \texttt{goto}\, \widehat{\ell};\end{array}$$

El valor exacto de cada etiqueta $\widehat{j}$ en la definición anterior debe definirse solamente 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 línea 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 29. *Sea $p =$* `read` $X; C_1; \ldots; C_m;$ `write` Y *un programa* While1*. Definimos el programa p_G en* Goto *como:*

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

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 30. *El siguiente es el programa* Goto *para la suma de dos numerales obtenido a partir del programa del ejemplo 20 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; para ello, es importante notar que a cada estado $s = \langle \ell, \sigma \rangle$ de WHILE1 se le asocia un estado $\bar{s} = \langle \widehat{\ell}, \sigma \rangle$ en GOTO.

Lema 31 (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[7] $p \rhd_{\text{WHILE1}} s \to s'$. □

Proposición 32. *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 33 (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* $\mathcal{I}_\ell$ *es una asignación* $X := e$ *entonces* $\mathcal{I}_\ell^\star$ *se define como la secuencia de instrucciones* $X := e; PC := \ell + 1$
- *Si* $\mathcal{I}_\ell$ *es un condicional* `if` X `then goto` ℓ' `else goto` ℓ'' *entonces* $\mathcal{I}_\ell^\star$ *es:*

$$\texttt{if}\ X\ \texttt{then}\ PC := \ell'\ \texttt{else}\ \ PC := \ell''$$

- *Si* $\mathcal{I}_\ell$ *es una instrucción* `goto` ℓ' *entonces* $\mathcal{I}_\ell^\star$ *es la asignación* $PC := \ell'$

[7] En este caso se trata de una inducción estructural sobre la definición de la relación $p \rhd_{\text{WHILE1}} s \to s'$, la cual puede reemplazarse por una inducción sobre el número de pasos en la derivación de la relación $p \rhd_{\text{WHILE1}} s \to s'$.

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

Definición 34. *Sea p un programa en* Goto *de la siguiente forma* `read` $X; 1 : \mathcal{I}_1; \ldots; m : \mathcal{I}_m;$ `write` Y. *Definimos el programa p_W en* While *como:*

```
read X;
PC := 1;
while PC do
  case PC of
    1 ⇒ I*_1;
    ⋮
    m ⇒ I*_m;
    otherwise ⇒ PC := 0;
end;
write Y
```

Aquí la instrucción `case` *no es válida en la sintaxis formal pero se trata nuevamente de un macro definible de la manera usual con instrucciones condicionales* `if`.

Ejemplo 35. *Veamos el resultado de aplicar la transformación de Böhm-Jacopini al programa* `reverse` *del ejemplo 23 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 36. *Sean* $p = \texttt{read}\, X, 1 : \mathcal{I}_1, \ldots, m : \mathcal{I}_m, \texttt{write}\, Y$ *un programa y* $s = \langle \ell, \sigma \rangle$ *un estado en* GOTO. *Definimos el estado* $\bar{s}$ *como la pareja* $\langle \widetilde{\ell}, \widetilde{\sigma} \rangle$ *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 37 (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 38. *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 39 (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. □

A continuación discutiremos 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 40. *La función de codificación* $(\cdot)^\dagger : \{0, 1, \sqcup\}^\star \to \mathbb{D}$ *se define como* $w^\dagger = s_1^\dagger s_2^\dagger \ldots s_n^\dagger$. *donde* $w = s_1 s_2 \ldots s_n$ *y*

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

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 41. *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* $\mathcal{I}_\ell$ *es* `right` *entonces la instrucción* $\widehat{\ell} : \widetilde{\mathcal{I}}_\ell$ *es la secuencia:*

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

  donde las variables A_1, A_2 *son nuevas.*

- *Si* $\mathcal{I}_\ell$ *es* `left` *entonces* $\widehat{\ell} : \widetilde{\mathcal{I}}_\ell$ *se define análogamente al caso anterior.*

- *Si* $\mathcal{I}_\ell$ *es* `write` s *entonces* $\widehat{\ell} : \widetilde{\mathcal{I}}_\ell$ *se define como* $\widehat{\ell} : C := s$

- *Si* $\mathcal{I}_\ell$ *es* `if` s `then goto` ℓ' `else goto` ℓ'' *entonces* $\widehat{\ell} : \widetilde{\mathcal{I}}_\ell$ *es la siguiente secuencia de instrucciones*

$$\begin{array}{l}
\widehat{\ell} : A_1 := s \\
\widehat{\ell} + 1 : A_2 := (=?\ C\ A_1) \\
\widehat{\ell} + 2 : \texttt{if}\ A_2\ \texttt{then goto}\ \widehat{\ell'}\ \texttt{else goto}\ \widehat{\ell''}
\end{array}$$

  donde las variables A_1, A_2 *son nuevas.*

- *Si* $\mathcal{I}_\ell$ *es* `goto` ℓ' *entonces* $\widetilde{\mathcal{I}}_\ell$ *es* `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 42. *Dado un programa* $p = 1 : \mathcal{I}_1, 2 : \mathcal{I}_2, \ldots, m : \mathcal{I}_m; m+1 :$ `halt` *en* LTuring *definimos el programa* p_G *en* Goto *como sigue:*

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

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 9.

7.3.1. Compilación

Veamos ahora la existencia del compilador correspondiente.

Definición 43. *Dado un estado* $s = \langle \ell, L\underline{s}R\rangle$ *en* LTURING *definimos el estado* $\bar{s}$ *en* GOTO *como el par* $\langle \widehat{\ell}, \sigma\rangle$ *donde*

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

aquí L^{rev} *denota a la reversa de la cadena* L*.*

Lema 44 (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 45. *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 $p \rhd_{LTuring} \langle 1, \sqcup\underline{x}\rangle \to^{\star} \langle m+1, L\underline{s}\, y\rangle$ y por el lema de simulación tenemos que $p_G \rhd_{Goto} \langle 1, \sigma_0\rangle \to^{\star} \langle \widehat{m+1}, \sigma_f\rangle$ donde $\sigma_0 = [Lf \mapsto \texttt{nil},\ C \mapsto \texttt{nil},\ Rt \mapsto x^{\dagger}]$ 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 sola cinta, ver [**4**, **2**] por ejemplo.

Definición 46 (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:*

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

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 47. *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 48. *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 47*
- *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 instrucción del programa* GOTO *son las siguientes:*
  - *Si* $\mathcal{I}_\ell$ *es de la forma* $X := e$ *entonces*

    $$q_\ell\text{:}\ \texttt{compute e in } \mathcal{A}\text{;}\ \texttt{erase } X\text{;}\ \texttt{copy } \mathcal{A} \texttt{ to X;}\ \texttt{C.E;}\ \texttt{goto } q_{\ell+1}$$

  - *Si* $\mathcal{I}_\ell$ *es de la forma* $\texttt{if } X \texttt{ then goto } \ell' \texttt{ else goto } \ell''$ *entonces*

    $$q_\ell\text{:}\ \texttt{compute X in } \mathcal{A}\text{;}\ \texttt{C.E;}\ \texttt{if } \mathcal{A} \texttt{ then goto } q_{\ell'} \texttt{ else goto } q_{\ell''}$$

  - *Si* $\mathcal{I}_\ell$ *es de la forma* $\texttt{goto } q_{\ell'}$ *entonces*

    $$q_\ell : \texttt{goto } q_{\ell'}$$

Ejemplo 49. *La máquina* $T_{\texttt{reverse}}$ *asociada al programa del ejemplo 23 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 puede demostrar de forma rutinaria definiendo la relación de transición de configuraciones instantaneas para máquinas multicinta de forma análoga a la definición 2.

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 que la siguiente afirmacion se cumple: cualquier formalización 'razonable' de la noción intuitiva de computo efectivo es equivalente a la noción de Turing computabilidad.

Proposición 50. *Los lenguajes* WHILE, GOTO *y* LTURING *son equivalentes a* $\mathcal{MT}$.

Demostración. Considerando 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 sí 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 autointérprete, contraparte de la máquina universal de Turing, 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 basándose en el lenguaje While en lugar de máquinas de Turing, lo cual se presenta en [**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.

Bibliografía

1. C. Böhm y G. Jacopini, Flow diagrams, Turing machines and languages with only two formation rules, *Communications of the ACM* **9(5)** (1966) 366–371.
2. J. E. Hopcroft, R. Motwani, y J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 2 ed., Addison-Wesley Publishing Company, 2001.

3. N. 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. J. Martin, *Lenguajes formales y teoria de la computacion*, 3 ed., McGraw Hill, 2004.
5. A. M. Turing, Proceedings of the london mathematical society, en *On computable numbers with an application to the* Entscheidungsproblem, tomo 42(2), 1936-7, 230–265.