IV. LINGUAGENS DO TIPO 0 E MÁQUINAS DE TURING

4.1 A Máquina de Turing Universal

Sempre se pensou apenas nas máquinas de Turing como aceitadores de linguagem. É também importante usar estas máquinas como dispositivos que computam funções numéricas, isto é, que mapeiam $\mathbf{N}^k \rightarrow \mathbf{N}$. Concordamos então em codificar o conjunto dos números naturais na notação unária. Então o código para 0 é 1, o código para 1 é 11, 2 é 111, 3 é 1111, etc. Escreve-se n^ para simbolizar o n codificado em unário.

Usando a notação unária, pode-se dar uma semântica teorético-numérica para as máquinas de Turing. Ou seja, dada uma máquina de Turing sobre um alfabeto que inclui o símbolo 1, pode-se dizer como interpretar o comportamento de uma máquina de Turing tal que ela possa ser pensada como um dispositivo que computa uma função teorético-numérica. Como se verá, uma única máquina de Turing de acordo com a convenção adotada realmente computa uma função teorético-numérica (diferente) $\mathbf{N}^k \rightarrow \mathbf{N}$ para toda aridade k.

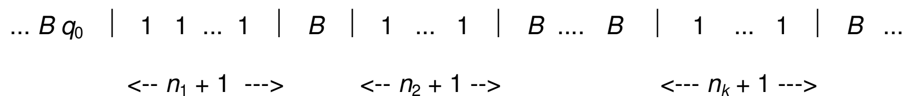


Figura 2. Descrição instantânea inicial para computar $\varphi_M^{(k)}(n_1, ..., n_k)$.

1. Definição. Uma máquina de Turing M computa uma função $\varphi_M^{(k)}$ de aridade k como se segue. Na entrada $(n_1, ..., n_k)$, $n_1, ..., n_k$ são colocados na fita de M em unário, separados por brancos simples (Figura 2). A cabeça de M é colocada sobre o 1 mais a esquerda de n_1^, e o controle de estados finitos de M é colocado em q_0. Em outras palavras, M tem DI inicial

$$q_0 n_1\hat{}\,B n_2\hat{}\,B \ldots B n_k\hat{}$$

Se e quando M terminar o processamento, os 1's na fita são contados (eles podem não aparecer em um bloco) e seu total é o valor de $\varphi_M^{(k)}(n_1, ..., n_k)$. Se M nunca pára, diz-se que $\varphi_M^{(k)}(n_1, ..., n_k)$ é *indefinido*. Refere-se a $\varphi_M^{(k)}$ como a (k-ésima) *semântica* de M.

2. Exemplo. A máquina de Turing sem nenhuma quíntupla (isto é, ela pára qualquer que seja a DI) computa a função sucessora $\varphi^{(1)}(n) = s(n) = n + 1$. Entretanto, o leitor deve checar que, como uma calculadora para uma função de duas variáveis, ela computa a função $\varphi^{(2)}(x, y) = x + y + 2$.

3. Exemplo. Considere a seguinte máquina de Turing.

$$(q_0\ 1\ q_1\ 1\ R)$$

$$(q_1\ 1\ q_0\ 1\ R)$$
$$(q_1\ B\ q_1\ B\ R)$$

Esta máquina de Turing computa a seguinte função de uma variável

$$\varphi_M^{(1)}(n) = \begin{cases} n + 1, & \text{se } n \text{ é ímpar;} \\ \perp, & \text{caso contrário} \end{cases}$$

onde a notação "$\perp$" é uma abreviação para "é indefinido". Isto é, aqui a semântica de M é uma *função parcial*: para algumas entradas, um valor é retornado, mas para outras - neste caso, todos os argumentos pares - a função é indefinida. Este fenômeno é um fato inegável da vida em ciência da computação. Existem programas perfeitamente legais em qualquer linguagem de programação (suficientemente rica) que falha ao retornar valores para algumas ou possivelmente todas as entradas. Isto é porque a noção matemática correspondente de uma função parcial é o estabelecimento apropriado para a teoria de algoritmos abstrata e as funções que eles computam.

4. Definição. Uma *função parcial* é uma função que pode ou não ser definida para todos os seus argumentos. Especificamente, onde uma função "ordinária" g: $A \to B$ atribui um valor $g(x)$ em B para cada x em A, uma função parcial φ: $A \to B$ atribui um valor $\varphi(x)$ apenas para os x's em algum subconjunto dom(φ) de A, chamado de *domínio da definição* de φ. Se $x \notin$ dom(φ), diz-se que φ é indefinido ou não especificado para aquele valor. Note que deve-se referir a A como o domínio de φ: $A \to B$, e B como o contra-domínio. O conjunto $\{ \varphi(x) \mid x \in \text{dom}(\varphi)\}$ é chamado de *faixa* de φ e é denotado por $\varphi(A)$. Se o dom(φ) = A, isto é, φ atribui um valor em B para todo $x \in A$, então φ é chamado de *função total*.

A mínima função parcial definida é a função vazia. É escrita $\perp$, com o dom($\perp$) = $\emptyset$, tal que $\perp(n) = \perp$ para todo n em **N**. As máximas funções parciais definidas são as funções totais, tal como a função sucessora, $s(n) = n + 1$, para todo n em **N**. Entre elas há funções parciais definidas parcialmente, tal como a função computada no Exemplo 3.

5. Definição. Uma função parcial φ: $N \to N$ é *Turing-computável* se ela for $\varphi_M^{(1)}$ para alguma máquina de Turing.

Agora de uma forma óbvia pode-se falar sobre as funções teorético-numéricas PASCAL-computáveis, as funções teorético-numéricas FORTRAN-computáveis, etc. Qual é o relacionamento entre estas classes? Conclui-se que estas classes de funções coincidem com as funções Turing-computáveis. Depois de meio século de estudos detalhados de vários sistemas de computação e as funções que eles computam, achou-se que as funções computáveis são invariantes ao longo de uma faixa larga de diferentes mecanismos de definição - cada sistema formal estudado foi mostrado computar ou todas as funções Turing-computáveis ou algum subconjunto delas. Isto levou o lógico matemático Alonzo Church a formular a *tese de Church*, que diz que *todos* os mecanismos de computação suficientemente poderosos definem a mesma classe de funções computáveis.

Um conceito familiar em ciência da computação é que quando um computador, M, for suficientemente de "propósito geral", um programa escrito em qualquer outra máquina pode ser recodificado para fornecer um programa para M que computará a mesma função. Aqui apresentamos um resultado de 1936 devido a A. M. Turing que antecipa o computador digital por quase uma década e ainda carrega a idéia inicial da sentença anterior: ou seja, que existe uma máquina de Turing U que é *universal*, no sentido de que o comportamento de qualquer outra máquina M pode ser codificado como uma cadeia $e(M)$ tal que U processará qualquer cadeia da forma $(e(M), w)$ da forma como w seria processado por M; diagramaticamente, significa

$$\text{se} \quad w \Rightarrow^*_M w' \quad \text{então} \ (e(M), w) \Rightarrow^*_U w'$$

Antes de construir a máquina universal U, precisamos mostrar como enumerar todas as descrições da máquina de Turing. Por que? Porque uma máquina de Turing é apresentada como uma lista de quíntuplas, a enumeração deve ser uma listagem de listas de quíntuplas. Além disso, a enumeração será feita de uma forma *efetiva*, tal que dado um inteiro k pode-se algoritmicamente achar a k-ésima máquina de Turing, e dado uma máquina de Turing, pode-se algoritmicamente achar k, sua posição na enumeração.

Começamos descrevendo uma versão especial do "programa vazio", uma máquina de Turing que é indefinida para todas as aridades e todas as entradas.

$$(q_0\ 1\ q_0\ 1\ R)$$
$$(q_0\ B\ q_0\ B\ R)$$

Claramente, para qualquer entrada envolvendo apenas 1's e brancos, este programa "roda" para sempre. Denota-se esta máquina como M_R, já que ela sempre se move para a direita.

Agora, damos a listagem sistemática de todas as máquinas de Turing:

$$M_0, M_1, ..., M_k, ...$$

assumindo que os símbolos da fita são apenas B e 1, e todos os estados são codificados na forma q_k onde k é um número natural escrito na notação decimal. Aqui, a máquina M_k é determinada como se segue. Associa-se um código parecido com ASCII com todo símbolo que pode aparecer em uma quíntupla. O seguinte quadro dá uma destas combinações.

100000	0
100001	1
100010	2
...	
101001	9
110000	*q*
110001	1 (o símbolo da fita)
110010	*B*
111000	*R*
111001	*L*

111010 N

Usando este código pode-se associar uma cadeia binária com qualquer quíntupla simplesmente concatenando as cadeias de 6 bits associadas com cada símbolo. Por exemplo:

$(q_2\ 1\ q_{11}\ B\ L)$

110000:100010:110001:110000:100001:100001:110010:111001
q 2 1 q 1 1 B L

(Os dois-pontos ":" não são parte da cadeia - estão aqui apenas para ajudar a leitura.)

Uma vez estabelecida uma codificação para as quíntuplas, pode-se codificar uma máquina de Turing completa sem ambigüidade através da concatenação de cadeias de bits de suas quíntuplas individuais. A cadeia concatenada resultante, interpretada como um número binário, é o código da máquina de Turing. O número natural n é uma *descrição de máquina legal* se ele corresponde a um conjunto de quíntuplas que são determinísticas no sentido descrito na seção anterior.

Portanto, tem-se a seguinte listagem de máquinas de Turing,

6. $M_0, M_1, ..., M_n, ...$

onde M_n é a máquina com código binário n, se n for uma descrição de máquina legal, e a máquina fixa M_R para a função vazia, caso contrário. Chama-se n o índice da máquina M_n. (Note que M_R aparece muito freqüentemente na lista.)

Além da listagem das máquinas de Turing pode-se falar sobre uma listagem das funções Turing-computáveis. As funções Turing-computáveis de uma variável são enumeradas como se segue:

7. $\varphi_0, \varphi_1, ..., \varphi_n, ...$

onde φ_n é a função de uma variável $\varphi_{Mn}^{(1)}$ computada pela máquina de Turing M_n.

Antes de apresentar o resultado principal, façamos várias observações sobre a listagem da nossa máquina de Turing **6** e a listagem de funções computáveis **7**. Primeiro, a listagem demonstra que há apenas contavelmente muitas máquinas de Turing e funções associadas. Segundo, note que toda função Turing-computável aparece infinitamente freqüente na listagem φ_n. Isto acontece porque, dada qualquer máquina M_n, considere o conjunto de estados de M_n, Q. Para qualquer estado p não em Q (e há infinitamente muitos destes), considere a máquina consistindo de M_n e da quíntupla adicional $(p\ B\ p\ B\ N)$. Esta nova máquina, M', faz exatamente o que M_n faz pois o estado p nunca pode ser alcançado. Mas para cada escolha do estado p, M' tem um índice diferente.

Finalmente, note que listamos apenas as funções de uma variável computadas pelas nossas máquinas de Turing. Pode-se dar uma listagem também para as funções de *k* variáveis para qualquer $k > 1$. Escreve-se como $\varphi_n^{(k)}$.

Vamos voltar a máquina de Turing universal *U*. Primeiro, vamos mostrar que não há perda de generalidade na simulação de máquinas tendo apenas dois símbolos, *B* e 1, em seus alfabetos de entrada. Segundo, vamos desenhar uma máquina universal *U*^ que usa três cabeças de fita para acompanhar sua tarefa. Finalmente, vamos mostrar que *U*^ pode ser substituída por uma *U* de cabeça única, ainda que sofrendo um grande aumento no tempo de computação.

Nosso primeiro estágio é simples. Se substituirmos cada símbolo de um alfabeto de fita grande por uma cadeia distinta de *m* símbolos de $\{1, B\}^m$, podemos simular nossa máquina original *M* por uma nova máquina *M*^ que simula um único movimento de *M* em no máximo 3*m* movimentos. Pense na fita da máquina como dividida em blocos de *m* quadrados consecutivos, tal que cada bloco contenha o código para um único símbolo do alfabeto original Σ' (e com B^m codificando o *B* de Σ'). Então a nova máquina *M*^ começa seu ciclo de simulação no quadrado mais a esquerda de um bloco. Ele então lê a direita *m* passos para inferir o símbolo de Σ' codificado. Depois, ele armazena o estado novo desejado de *M* e completa seu ciclo não movendo sua cabeça (para simular um *R*); *m* quadrados a esquerda (para simular um *N*); ou 2*m* quadrados a esquerda (para simular um *L*). Como ela codifica diretamente este processo na forma quíntupla tem-se o seguinte:

8. Lema. *Dada qualquer máquina de Turing* $M = (Q, \Sigma, q_0, q_a, \delta)$ *e qualquer mapa um por um* $h: \Sigma \rightarrow \{1, B\}^m$ *que envia B para* B^m*, existe uma máquina de Turing*

$$M\hat{} = \{Q\hat{}, \{1\}, q_0', q_a', \delta\hat{}\}$$

que simula M de uma forma precisa: existe um codificação $h_q: Q \rightarrow Q\hat{}$ *de estados de M tal que para toda DI* $\alpha q \beta$ *de M podemos obter a codificação de seu sucessor em* 3*m passos pelo uso de M*^; *isto é*

$$\delta_M(\alpha q \beta) = \alpha' q' \beta' \Rightarrow \delta_{M\hat{}}^{3m}(h(\alpha) h_q(q) h(\beta)) = h(\alpha') h_q(q') h(\beta')$$

<u>PROVA</u>: Por uniformidade, fazemos o "ciclo" de *M*^ para *m* passos quando simulamos um movimento *N*, e para 2*m* passos quando simulamos um movimento a direita tal que a simulação de um passo-*M* sempre leva 3*m* passos-*M*^. ◊

Nós agora vamos mostrar como qualquer máquina de Turing com um alfabeto de 2 símbolos pode ser simulada por uma máquina de Turing *U*^ que tem três cabeças com as quais ela pode percorrer as fitas (que nós convenientemente representamos como três trilhas de uma única fita), como mostrado no Figura 3.

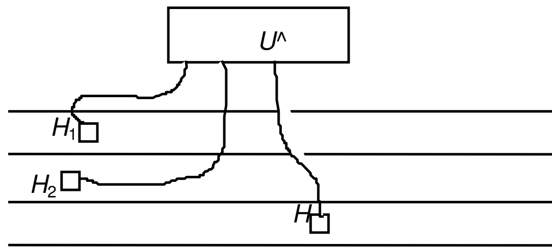


Figura 3

A idéia é que H_1 situada na primeira trilha da fita de tal forma que a cabeça de *M* se situa na sua fita binária. *U^* então consulta as quíntuplas de *M*, usando H_2 para ler sua codificação na trilha 2, para comandar o comportamento de H_1 na trilha 1; ele usa H_3 na trilha 3 para computações subsidiárias requeridas para reposicionar H_2 na codificação de quíntupla correta para o próximo ciclo de simulação.

Suponha que nós temos uma máquina com *p* cabeças percorrendo uma única fita na qual são impressos símbolos do alfabeto Σ'. A qualquer tempo a caixa de controle estará no estado *q* de *Q* e receberá como entrada os *p* símbolos percorridos pelas suas cabeças, isto é, um elemento de $(\Sigma')^p$. A saída da unidade de controle é um elemento de $((\Sigma')^p \times M) \cup \{halt\}$, onde $M = \{I \mid I$ é uma instrução possível às cabeças para mover ao máximo a distância 1$\}$. Então, a máquina de Turing é especificada como usual pelas quíntuplas

$$q_i\, x_j\, q_l\, x_k\, I$$

com a única diferença de que os *x*'s e os *I*'s são "vetores," e que devemos empregar uma convenção para resolver conflitos se duas cabeças tentam imprimir símbolos diferentes num único "quadrado." Uma computação de tal máquina começa com a atribuição de um estado à unidade de controle e a atribuição das posições iniciais para as cabeças. Como de costume, é assumido que a fita tem no máximo finitamente muitos quadrados não brancos. Então a computação prossegue da forma usual, parando quando e apenas quando nenhuma quíntupla começando com q_ix_j é aplicável.

Nossa tarefa é mostrar que tal computação pode ser simulada, de forma adaptável, em uma máquina de Turing "ordinária" (isto é, com $p = 1$) tal que o número de passos necessários para tal simulação é limitado.

9. Lema. *Suponha que uma máquina de Turing generalizada M^ tenha* (a) *p rastreadores em uma única fita de uma dimensão ou* (b) *um rastreador em cada uma das p fitas. Então M^ pode ser simulada por uma máquina de Turing ordinária M de tal forma que um único passo de M^, quando a porção ativa de sua fita for de n quadrados, pode ser simulada em no máximo* $2n + 2p$ *passos.*

<u>PROVA</u>: Seja nossa máquina *M^* de *p* cabeças com alfabeto Σ' em uma fita única no caso (a). No caso (b) simplesmente colocamos as *p* fitas lado a lado para formar

uma única fita com alfabeto $\Sigma' = \Sigma'_1 \times ... \times \Sigma'_p$ onde Σ'_j é o alfabeto na fita *j*. Em qualquer caso, simulamos *M^* com uma máquina de uma cabeça *M* com alfabeto $(\Sigma' \times S_p \cup \{*\})$ onde S_p é o conjunto de todos os subconjuntos de $\{1, 2, ..., p\}$, e $*$ marca o final esquerdo da fita. Usamos o alfabeto de fita aumentado para codificar a colocação das cabeças na fita. Por exemplo, se a cabeça 1 e a cabeça 3 de *M^* percorre (rastreia) o símbolo x_2, codificamos este fato na fita de *M* usando o símbolo composto $(x_2, \{1,3\})$ ao invés de x_2. Escrevemos x para $(x,\emptyset)$.

Mais formalmente, se *M^* tem a fita não branca

$$x_1x_2...x_m$$

com a cabeça *j* percorrendo x_{ij}, deixamos *M* ter a sua cabeça percorrendo o quadrado mais a esquerda de

$$(x_1, A_1)(x_2, A_2)...(x_m, A_M)$$

onde $A_k = \{j \mid k = i_j\}$.

Agora suponha que o alfabeto de fita de *M^* consiste dos dígitos 1 a 9 e suponha que abreviamos a símbolo $(x,\emptyset)$ de *M* escrevendo apenas *x*. Então, codificamos a configuração *M^*

```
          3     1,2
          ↓     ↓
    3 7 4 1 2 9 1 3 4 2 2
```

com a configuração *M*

```
↓
∗ 3 7 4 (1,{3}) 2 9 (1,{1,2}) 3 4 2 2
```

Então, a informação da cabeça é codificada na fita. Agora é necessário projetar a lógica tal que esta informação seja atualizada depois de cada ciclo de simulação. A caixa de controle de *M* pode ser pensada como contendo *p* registradores que armazenarão os *p* símbolos percorridos de *M^*.

Em uma simulação, *M* move ↓ para a direita de $*$ até que tenha preenchido todos os *p* registradores. Ele então “conhece” os novos valores necessários e move ↓ para a esquerda, mudando os quadrados apropriadamente, até que tenha retornado a $*$ (que pode estar deslocado um quadrado a esquerda).

Se o quadrado mais a direita no qual o segundo símbolo é não nulo é *n* quadrados a direita de $*$, então a simulação levará no máximo $2n + 2p$ passos, os passos $2p$ extras sendo necessários para simular o pior reposicionamento possível das cabeças. ◊

Note que este resultado suporta a tese de Church de que qualquer algoritmo codificado em qualquer linguagem de computador, real ou imaginária, é em princípio programável por máquinas de Turing. Imagine que uma fita da máquina de Turing é uma palavra de memória - de fato, uma palavra de memória não limitada. Agora suponha que consideramos uma máquina de fita de 512K/cabeça. Se imaginarmos que a fita 1 é usada como um armazenamento de massa infinito e as outras fitas são meramente consideradas como palavras de memória, então teremos uma máquina com uma linguagem de máquina esquisita mas útil e uma memória infinita. Tal linguagem é adequada para tornar real um algoritmo escrito em qualquer linguagem de computador.

10. Teorema. (O Teorema da Máquina Universal). *Existe uma máquina de Turing universal U tal que*

$$\varphi_U^{(2)}(x, y) = \varphi_x^{(1)}(y)$$

O *teorema da máquina universal* estabelece que existe uma única máquina de Turing que, quando considerada como um dispositivo que calcula uma função de duas variáveis, age como um interpretador: ela trata seu primeiro argumento como um código de programa e aplica este código ao seu segundo argumento. Note que Turing publicou este resultado em 1936 (quando ele tinha 24 anos), cerca de 8 anos antes do primeiro computador eletrônico programado ser construído, e bem antes de um interpretador real ter sido projetado.

4.2. Máquinas de Turing Não Determinísticas

Com nossa experiência em reduzir máquinas de Turing multi-cabeças para máquinas de uma cabeça, podemos concluir que as máquinas de Turing não determinísticas aceitam os mesmos conjuntos das máquinas de Turing determinísticas. Primeiro, diz-se que uma cadeia *w* é aceita por uma máquina de Turing de 2 cabeças e 2 fitas se, quando *w* é escrita na fita 1 e a cabeça 1 percorre o símbolo mais a esquerda de *w* no estado q_0, e a fita 2 está inicialmente em branco, *M* eventualmente pára em seu estado de aceitação q_a. Então segue do Lema 4.1.9 que

1. Teorema. *Seja L uma linguagem que possa ser reconhecida por uma máquina de Turing de k cabeças e k fitas. Então L =* $T(M)$ *para alguma máquina de Turing de 1 cabeça e 1 fita.*

2. Definição. Uma máquina *M* é chamada de *máquina de Turing não determinística* se ela incluir quíntuplas que especifiquem movimentos múltiplos para um dado par estado/símbolo, isto é, se ela é definida como na Definição 3.2.2 exceto que agora δ mapeia $Q \times \Sigma'$ a subconjuntos de $Q \times \Sigma' \times \{L, N, R\}$.

Claramente, podemos definir DIs, transições e DIs de parada (*halting*) como antes. Dizemos que uma máquina de Turing não determinística *M aceita* uma cadeia *w* se existir *alguma* cadeia de transições da máquina na entrada *w* que alcança uma DI de parada que inclui o estado de aceitação q_a. A definição de aceitação está de acordo com a definição de aceitação não determinística dada para aceitadores de estados finitos não determinísticos. O próximo resultado mostra que, assim como com os AEFs, o não determinismo não adiciona potência computacional nova ao modelo determinístico original. (Isto *não* é verdadeiro para todas as máquinas que estudamos - autômatos de pilha não determinísticos (capítulo 2) *são* mais potentes que os autômatos de pilha determinísticos.)

3. Teorema. *Seja M uma máquina de Turing não determinística que aceita L. Então existe uma máquina determinística M' que também aceita L.*

PROVA: Vamos construir uma máquina determinística *M'* de 2 fitas e 2 cabeças que age como um interpretador para *M*. Em sua segunda fita *M'* mantém um registro das possíveis DIs ativas de *M* em qualquer instante de sua computação. Por exemplo, se *M* inclui as instruções q_0

$$(q_0\ a\ q_i\ b\ R)$$

$$(q_0\ a\ q_j\ c\ L)$$

então depois de uma execução de instrução na entrada *aa*, a fita 2 de *M'* se parece com

$$\#\ b\ q_i\ a\ \#\ q_j\ B\ c\ a\ \#$$

Usamos o símbolo # para separar as codificações das DIs. Suponha que em algum instante a fita 2 se pareça com

(∗) $\#\ DI_1\ \#\ DI_2\ \#\ ...\ \#\ DI_n\ \#$

Então *M'* opera para formar o próximo conjunto de DIs processando (∗) da esquerda para a direita. Quando processar o *j*-ésimo bloco, *M'* pode

(1) apagar o bloco se nenhuma transição for possível; ou
(2) substituir o bloco por um novo conjunto finito de blocos de DIs que representam as possíveis novas transições de *M* na DI dada. ◊

4. Teorema. *Uma linguagem é do tipo* 0 *se e somente se ela for o conjunto de aceitação de alguma máquina de Turing*.

4.3. O Problema da Parada (*Halting*) e a Indecidibilidade

Tendo demonstrado a potência e flexibilidade de computação efetiva, como capturadas nas máquinas de Turing, mostraremos agora que mesmo esta potência e flexibilidade têm limites. Uma questão muito prática para o cientista de computador é: este programa processará dados da forma pretendida? Mesmo mais fundamental, talvez, do que determinar se ou não um programa está correto é dizer se ou não ele terminará, fornecendo a resposta correta. Rescrevendo a última questão em termos das máquinas de Turing, então, teremos o seguinte:

O Problema da Parada (*Halting*). Dada uma máquina de Turing M_n com semântica φ_n e uma cadeia de dados *w*, $\varphi_n(w)$ é definida? Isto é, M_n sempre pára se iniciado no estado q_0 percorrendo o quadrado mais a esquerda de *w*?

Seria muito bom se pudéssemos achar algum testador de *halting* universal que, quando dado um número *n* e uma cadeia *w*, poderia efetivamente nos dizer se ou não a máquina M_n sempre parará se sua entrada de dados for *w*. Entretanto, o teorema 3 nos dirá que tal máquina não existe. Antes de mostrarmos que *nenhum* procedimento efetivo pode resolver o problema do *halting*, vejamos por que o procedimento *óbvio* falha. Vamos pegar a máquina universal *U* e "rodá-la" na fita ($e(M_n)$, *w*). Quando ela pára, o controle é transferido para uma subrotina que imprime um 1 para significar que a computação de M_n em *w* parou realmente. Mas quando o controle pode ser transferido para uma subrotina que imprime um 0 para significar que M_n nunca pára com os dados *w*? Depois de muitas simulações por *U*, é possível concluir que M_n *nunca* parará em *w*, ou que, depois de muito tempo, as computações pararão? Esta abordagem claramente falha. Mas para provar que *todas* as abordagens que tentam decidir a terminação algoritmicamente necessariamente falharão em uma tarefa bem menor, e para prová-lo devemos usar um descendente do argumento diagonal usado por Cantor para mostrar que nenhuma enumeração poderia incluir todos os números reais.

Para antever nosso estudo do problema do *halting*, primeiro forneceremos um resultado que relaciona nossa enumeração das máquinas de Turing com nossa discussão de funções numéricas. Lembre-se que uma função Turing-computável ψ é total se ela retorna uma saída para toda entrada.

2. Teorema. *Não existe nenhuma função Turing-computável total g que enumera as funções Turing-computáveis totais no seguinte sentido: a função Turing-computável ψ é total se e somente se ψ for igual à função $\varphi_{g(n)}$ computada por $M_{g(n)}$ para algum n.*

<u>PROVA</u>: Nesta prova usaremos a noção de que, se consideramos um procedimento efetivo para computar uma função, então de fato ela é Turing-computável (Vide discussão da tese de Church).

Se g é uma função Turing-computável total, então o seguinte é um procedimento efetivo para computar uma função numérica *total* que chamaremos de h: dado n, compute $g(n)$, então compute $\varphi_{g(n)}(n) = \varphi_U^{(2)}(g(n), n)$, e então some um a este resultado.

Portanto, se $\varphi_{g(n)}$ é *total* para todo n podemos concluir que

$$h(n) = \varphi_{g(n)}(n) + 1$$

é uma função Turing-computável total. Mas este h não pode ser igual a $\varphi_{g(n0)}$ para qualquer n_0, para então termos a contradição $h(n_0) = \varphi_{g(n0)}(n_0) = \varphi_{g(n0)}(n_0) + 1$. Portanto concluímos que para qualquer função numérica Turing-computável total g: $\mathbf{N} \to \mathbf{N}$, a enumeração $\varphi_{g(0)}, \varphi_{g(1)}, \varphi_{g(2)}, \ldots$ não pode obter todas (e apenas) as funções totais. ◊

Vamos agora ao resultado central desta seção.

3. Teorema. (A Insolubilidade do Problema do *Halting*). *Considere a função* halt: $\mathbf{N} \to \mathbf{N}$ *definida como*:

$$\text{halt}(x) = \begin{cases} 1, & \text{se } M_x \text{ pára na entrada } x; \\ 0, & \text{se } M_x \text{ nunca pára em } x. \end{cases}$$

Então halt *não é Turing-computável*.

Antes de provar este resultado, vamos fazer algumas observações. Primeiro, note que *halt* é uma função total com certeza. Ela tem um valor definido para toda entrada. O que estamos tentando mostrar é que *halt não* é computável: não existe nenhum método (formalmente expresso como uma máquina de Turing) para calcular os valores de *halt*. Note também que fomos muito conservadores no Teorema 3. Apenas consideramos a não computabilidade de Turing. De fato, se a tese de Church estiver subscrita, *halt* não é computável por nenhum algoritmo especificado em qualquer linguagem de programação.

<u>PROVA DO TEOREMA</u>: Suponha que construamos uma matriz $\mathbf{N} \times \mathbf{N}$, estabelecendo a entrada (i, j) para $\downarrow$ se a computação de $\varphi_i(j)$ termina, enquanto estabelecemos a $\uparrow$ caso contrário. Para construir uma função φ não no conjunto de φ_i's adotamos o

argumento diagonal de Cantor: movendo para baixo na diagonal, “invertemos as setas”, dando $\varphi(x)$ o comportamento do *halting* oposto a $\varphi_x(x)$:

$$\varphi(x) = \begin{cases} 1, & \text{se } \varphi_x(x) \text{ não for definido;} \\ \perp, & \text{se } \varphi_x(x) \text{ for definido.} \end{cases}$$

Mas isto apenas diz que

$$\varphi(x) = \begin{cases} 1, & \text{se } halt(x) = 0; \\ \perp, & \text{se } halt(x) = 1. \end{cases}$$

Agora se *halt* for computável, existe certamente uma máquina de Turing que computa φ. Entretanto, nosso argumento diagonal garante que φ *não* é computável, se φ for φ_j para algum j teríamos $\varphi_j(j) = 1$ se $\varphi_j(j) = \perp$ enquanto $\varphi_j(j) = \perp$ se $\varphi_j(j) = 1$ - uma contradição. Assim nenhum método de cálculo para *halt* pode existir.

◊

Portanto, exibimos um problema geral em ciência da computação, natural, interessante e valioso, que não tem solução algorítmica. (Este resultado é extremamente fundamental e está fortemente relacionado com o resultado famoso de Gödel da incompletude das teorias formais da aritmética.)

Neste ponto vamos clarear nosso pensamento sobre conjuntos computáveis de números naturais.

4. Definição. Um conjunto $S \subset N$ é *decidível* se a pertinência em S pode ser determinada algoritmicamente. Isto é, S é decidível (ou *recursivo* ou *solúvel*) se a função característica de S,

$$\chi_S(x) = \begin{cases} 1, & \text{se } x \in S \\ 0, & \text{se } x \notin S \end{cases}$$

é Turing-computável.

Se S não é decidível, dizemos que S é *indecidível*, ou *insolúvel* ou *não recursivo* ou, em casos que requer ênfase considerável, *recursivamente insolúvel*.

Temos um exemplo de um conjunto indecidível: o conjunto

$$K = \{ n \mid \varphi_n(n) \text{ retorna um valor } \}$$

é recursivamente insolúvel. Nosso próximo resultado mostra como mapear a indecidibilidade de K na teoria da linguagem.

5. Definição. Seja G uma gramática do tipo 0 sobre algum alfabeto $V \cup \Sigma$. O *problema de dedução* para G é o problema de determinar, dadas as cadeias arbitrárias x, y em $(V \cup \Sigma)^*$, se $x \Rightarrow^* y$ em G.

6. Teorema. *O problema de dedução para gramáticas do tipo* 0 *é indecidível.*

Contrastando com o resultado anterior, temos o seguinte:

7. Teorema. *O problema de dedução para gramáticas sensíveis ao contexto é decidível.*

8. Corolário. *O problema de dedução para gramáticas livres de contexto é decidível.*

9. Definição. Dada uma classe de gramáticas C, o *problema de esvaziamento* para C é o problema de determinar para G arbitrária em C, se a linguagem $L(G)$ é vazia.

10. Teorema. *O problema de esvaziamento para gramáticas livres de contexto é decidível.*

Vamos agora discutir o "problema de correspondência de Post", um resultado de indecidibilidade clássico sobre equações de cadeia, descoberto por Emil L. Post, com aplicações diretas à teoria da linguagem.

11. Definição. Um *sistema de Post P* sobre um alfabeto finito Σ é um conjunto de pares ordenados (y_i, z_i), $1 \leq i \leq n$, onde y_i, z_i são cadeias em Σ^*. Um par (y_i, z_i) é algumas vezes chamado de uma *equação de Post*. O *problema da correspondência de Post* (PCP) é o problema de determinar, para um sistema de Post arbitrário P, se existem inteiros $i_1, ..., i_k$ tais que

$$y_{i1}\, y_{i2} \ldots y_{ik} = z_{i1}\, z_{i2} \ldots z_{ik}$$

Os i_j's não precisam ser distintos. Para um dado PCP, devemos referir a uma cadeia solução como um *cadeia de Post*.

12. Exemplo. Considere o sistema de Post sobre $\Sigma = \{0,1\}$:

$$P = \{(1,01),(\lambda,11),(0101,1),(111,\lambda)\}$$

Este sistema tem uma solução, já que

$$y_2\, y_1\, y_1\, y_3\, y_2\, y_4 = 110101111 = z_2\, z_1\, z_1\, z_3\, z_2\, z_4.$$

O seguinte teorema é um resultado fundamental.

13. Teorema. *O problema de correspondência de Post é insolúvel.*

Porque é orientado a equação, o PCP é particularmente bem adaptado para aplicações na teoria da linguagem.

14. Teorema. *O problema de decidir se uma gramática livre de contexto arbitrária é ambígua é indecidível.*

4.4. Técnicas para Construção de Máquinas de Turing

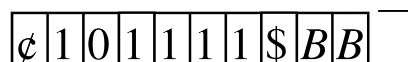
Uma máquina de Turing pode ser "programada" como um computador. Quando se especifica a função δ para uma máquina de Turing *T*, está-se escrevendo um programa para *T*. Algumas técnicas para construir Máquinas de Turing são:

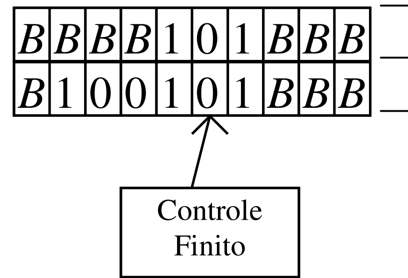
1. Armazenamento no Controle Finito. (O controle finito pode ser usado para armazenar uma quantidade finita de informação. Para fazer isto, o estado é escrito como um par de elementos, um exercendo controle e o outro armazenando um símbolo. Deve ser enfatizado que este arranjo é apenas para propósitos conceituais. Nenhuma modificação na definição da máquina de Turing é feita.) Considere a máquina de Turing $T = (Q, \{0,1\}, \{[q_0, B]\}, \{[q_1, B]\}, \delta)$, onde Q pode ser escrito como $\{q_0, q_1\} \times \{0, 1, B\}$. Isto é, Q consiste dos pares $[q_0, 0]$, $[q_0, 1]$, $[q_0, B]$, $[q_1, 0]$, $[q_1, 1]$ e $[q_1, B]$. *T* olha para o primeiro símbolo de entrada, grava-o em seu controle finito e checa que o símbolo não aparece em nenhum outro lugar na sua entrada. O segundo componente do estado grava o primeiro símbolo de entrada. Note que *T* aceita um conjunto regular, mas *T* servirá apenas para demonstração. *T* armazena o símbolo percorrido no segundo componente do estado e se move para a direita. O primeiro componente do estado de *T* se torna q_1. Se *T* tem um 0 armazenado e vê um 1, ou vice-versa, então *T* continua a mover para a direita. *T* entra no estado final $[q_1, B]$ se *T* alcança um símbolo branco sem ter encontrado uma segunda cópia do símbolo mais a esquerda. Se *T* alcança um branco no estado $[q_1, 0]$ ou $[q_1, 1]$, ele aceita. Para o estado $[q_1, 0]$ e símbolo 0 ou para o estado $[q_1, 1]$ e símbolo 1, δ não é definido, tal que se *T* vê o símbolo armazenado, ele pára sem aceitar. Define-se δ como se segue:

1. a) $\delta([q_0, B], 0) = ([q_1,0], 0, R)$
   b) $\delta([q_0, B], 1) = ([q_1,1], 1, R)$
   (*T* armazena o símbolo percorrido no segundo componente do estado e se move à direita. O primeiro componente do estado de *T* se torna q_1.)
2. a) $\delta([q_1, 0], 1) = ([q_1,0], 1, R)$
   b) $\delta([q_1, 1], 0) = ([q_1,1], 0, R)$
   (Se *T* tem um 0 armazenado e vê um 1, ou vice-versa, então *T* continua a se mover para a direita.)
3. a) $\delta([q_1, 0], B) = ([q_1, B], 0, L)$
   b) $\delta([q_1, 1], B) = ([q_1, B], 0, L)$
   (*T* entra no estado final $[q_1, B]$ se *T* alcança um símbolo branco sem ter primeiro encontrado uma segunda cópia do símbolo mais a esquerda.)

   Se *T* alcança um branco no estado $[q_1, 0]$ ou $[q_1, 1]$, ele aceita. Para o estado $[q_1, 0]$ e símbolo 0 ou para o estado $[q_1, 1]$ e símbolo 1, δ não é definido, tal que se *T* vir o símbolo armazenado, ele pára sem aceitá-lo.
   Em geral, pode-se permitir que o controle finito tenha *k* componentes, todos exceto um dos quais armazenam informação.

2. Trilhas Múltiplas. (A fita da máquina de Turing é dividida em *k* trilhas. Os símbolos na fita são considerados como *k*-tuplas - um componente para cada trilha.) Seja a fita de 3 trilhas da figura abaixo:





Esta fita pode ser imaginada como de uma máquina de Turing *T* que pega uma entrada binária maior que 2, escrita na sua primeira trilha, e determina se é um número primo. A entrada é envolvida por ¢ e $ na primeira trilha. Então, os símbolos de entrada permitidos são [¢, *B*, *B*], [0, *B*, *B*], [1, *B*, *B*] e [$, *B*, *B*]. Estes símbolos podem ser identificados com ¢, 0, 1 e $, respectivamente, quando vistos como símbolos de entrada. O símbolo branco pode ser representado por [*B*, *B*, *B*].

Para testar se sua entrada é um número primo, *T* primeiro escreve o número dois em binário na segunda trilha e copia a primeira trilha na terceira trilha. Então, a segunda trilha é subtraída, tantas vezes quanto possível, da terceira trilha, efetivamente dividindo a terceira trilha pela segunda e deixando o resto.

Se o resto for zero, o número na primeira trilha não é primo. Se o resto for diferente de zero, incremente o número da segunda trilha em um. Se agora a segunda trilha for igual à primeira, o número na primeira trilha é primo, porque ele não pode ser dividido por nenhum número entre um e ele mesmo. Se a segunda for menor que a primeira, toda a operação é repetida para o novo número na segunda trilha.

Na figura acima, *T* está tentando determinar se 47 é primo. *T* está dividindo por 5; como 5 foi subtraído duas vezes, o número 37 aparece na terceira trilha.

3. Cheque de Símbolos. O cheque de símbolos é um truque útil para visualizar como uma máquina de Turing reconhece linguagens definidas por cadeias repetidas, tais como

$$\{ww \mid w \text{ em } \Sigma^*\}, \quad \{wcy \mid w \text{ e } y \text{ em } \Sigma^*, w \neq y\} \quad \text{ou} \quad \{ww^R \mid w \text{ em } \Sigma^*\}$$

É também útil quando comprimentos de cadeias devem ser comparadas, tais como nas linguagens

$$\{a^ib^i \mid i \geq 1\} \quad \text{ou} \quad \{a^ib^jc^k \mid i \neq j \text{ ou } j \neq k\}.$$

Introduz-se uma trilha extra na fita que armazena um branco ou √. O √ aparece quando o símbolo abaixo dele foi considerado pela máquina de Turing em uma de suas comparações. Considere uma máquina de Turing $T = (Q, \Sigma, q_0, F, \delta)$, que reconhece a linguagem $\{wcw \mid w \text{ em } \{a, b\}^*\}$. Seja

$$Q = \{[q, d] \mid q = q_1, q_2, ..., q_9 \text{ e } d = a, b \text{ ou } B\}.$$

O segundo componente do estado é usado para armazenar um símbolo de entrada.

$$\Sigma = \{[B, d] \mid d = a, b \text{ ou } c\},$$
$$\Sigma' = \{[Y, d] \mid Y = B \text{ ou } \surd \text{ e } d = a, b, c \text{ ou } B\},$$
$$q_0 = [q_1, B] \quad \text{e} \quad F = \{[q_9, B]\}.$$

O símbolo branco é identificado com [*B*, *B*], *a* é identificado com [*B*, *a*], *b* é identificado com [*B*, *b*] e *c* é identificado com [*B*, *c*]. Define-se δ como

1. $$\delta([q_1, B], [B, d]) = ([q_2, d], [\surd, d], R)$$

para $d = a$ ou b. (*T* checa o símbolo percorrido na fita, armazena o símbolo no controle finito e se move para a direita.)

2. $\delta([q_2, d], [B, e]) = ([q_2, d], [B, e], R)$
para $d = a$ ou b, $e = a$ ou b. (T continua a se mover para a direita sobre símbolos não checados, procurando pelo c.)
3. $\delta([q_2, d], [B, c]) = ([q_3, d], [B, c], R)$
para $d = a$ ou b. (Ao achar c, T entra em um estado com primeiro componente q_3.)
4. $\delta([q_3, d], [\surd, e]) = ([q_3, d], [\surd, e], R)$
para $d = a$ ou b e $e = a$ ou b. (T se move para a direita sobre os símbolos checados.)
5. $\delta([q_3, d], [B, d]) = ([q_4, B], [\surd, d], L)$
para $d = a$ ou b. (T encontra um símbolo não checado. Se o símbolo não checado “casa” com o símbolo armazenado no controle finito, T o checa e começa a se mover para a esquerda. Se os símbolos não “casam”, T não tem um próximo movimento e então pára sem aceitação.)
6. $\delta([q_4, B], [\surd, d]) = ([q_4, B], [\surd, d], L)$
para $d = a$ ou b. (T se move para a esquerda sobre símbolos checados.)
7. $\delta([q_4, B], [B, c]) = ([q_5, B], [B, c], L)$
(T encontra o símbolo c.)
8. $\delta([q_5, B], [B, d]) = ([q_6, B], [B, d], L)$
para $d = a$ ou b. (Se o símbolo imediatamente à esquerda do c estiver não checado, T continua para a esquerda para encontrar o símbolo não checado mais à esquerda.)
9. $\delta([q_6, B], [B, d]) = ([q_6, B], [B, d], L)$
para $d = a$ ou b. (T continua para a esquerda.)
10. $\delta([q_6, B], [\surd, d]) = ([q_1, B], [\surd, d], R)$
para $d = a$ ou b. (T encontra um símbolo checado e se move para a direita, para pegar um outro símbolo para comparação. O primeiro componente do estado se torna q_1 novamente.)
11. $\delta([q_5, B], [\surd, d]) = ([q_7, B], [\surd, d], R)$
para $d = a$ ou b. (T estará no estado $[q_5, B]$ imediatamente depois de passar c se movendo para a esquerda. (Veja regra 7.) Se um símbolo checado aparece imediatamente à esquerda de c, todos os símbolos à esquerda de c foram checados. T deve testar se todos os símbolos à direita foram checados. Se foram, eles devem ser comparados propriamente com o símbolo à esquerda de c, tal que T aceitará.)
12. $\delta([q_7, B], [B, c]) = ([q_8, B], [B, c], R)$
(T se move para a direita sobre c.)
13. $\delta([q_8, B], [\surd, d]) = ([q_8, B], [\surd, d], R)$
para $d = a$ ou b. (T se move para a direita sobre símbolos checados.)
14. $\delta([q_8, B], [B, B]) = ([q_9, B], [\surd, B], L)$
(Se T acha $[B, B]$, o branco, ele pára e aceita. Se T achou um símbolo não checado quando seu primeiro componente do estado era q_8, ele deveria ter parado sem aceitar.)

4. Deslocamento. Uma máquina de Turing pode fazer espaço em sua fita deslocando todos os símbolos não brancos um número finito de células à direita. Para fazer isto, a cabeça da fita deve fazer uma excursão para a direita , repetidamente armazenando os símbolos lidos em seu controle finito e substituindo-os por símbolos das células a esquerda. A máquina de Turing pode então retornar às células vagas e imprimir símbolos de sua escolha. Se houver espaço disponível, ela pode empurrar blocos de símbolos à esquerda de uma forma parecida. Construímos parte de uma máquina de Turing, $T = (Q, \Sigma, q_0, F, \delta)$ que pode ocasionalmente ter uma necessida-

de de deslocar símbolos duas células para a direita. Suponha que Q contenha estados da forma $[q, A_1, A_2]$ para $q = q_1$ ou q_2, e A_1 e A_2 em Σ'. Seja B o branco e Y um símbolo especial não usado por T exceto no processo de deslocamento. Suponha que T começa o processo de deslocamento no estado $[q_1, B, B]$. As porções relevantes da função δ são as seguintes.

1. $\delta([q_1, B, B], A_1) = ([q_1, B, A_1], Y, R)$

para A_1 em $\Sigma' - \{B, Y\}$. (T armazena o primeiro símbolo lido no terceiro componente do seu estado. Y é impresso na célula percorrida e T se move para a direita.)

2. $\delta([q_1, B, A_1], A_2) = ([q_1, A_1, A_2], Y, R)$

para A_1 e A_2 em $\Sigma' - \{B, Y\}$. (T imprime um Y, armazena o símbolo sendo lido no terceiro componente, desloca o símbolo no terceiro componente para o segundo componente, e se move para a direita.)

3. $\delta([q_1, A_1, A_2], A_3) = ([q_1, A_2, A_3], A_1, R)$

para A_1, A_2 e A_3 em $\Sigma' - \{B, Y\}$. (T agora repetidamente lê um símbolo A_3, armazena-o no terceiro componente do estado, desloca o símbolo anterior do terceiro componente A_2 para o segundo componente, deposita o segundo componente anterior A_1 na célula percorrida, e se move para a direita. Deve estar claro que um símbolo será depositado duas células à direita da sua posição original.)

4. $\delta([q_1, A_1, A_2], B) = ([q_1, A_2, B], A_1, R)$

para A_1 e A_2 em $\Sigma' - \{B, Y\}$. (Quando um branco é visto na fita, os símbolos armazenados são depositados na fita.)

5. $\delta([q_1, A_1, B], B) = ([q_2, B, B], A_1, L)$

(Depois de todos os símbolos terem sido depositados, T faz o primeiro componente do estado igual a q_2 e moverá para a esquerda para achar um Y, que marca sua célula vaga mais à direita.)

6. $\delta([q_2, B, B], A) = ([q_2, B, B], A, L)$

para A em $\Sigma' - \{B, Y\}$. (T se move para a esquerda até que um Y seja encontrado. Quando Y é encontrado, T transferirá para um estado que assumiu-se existir em Q e termina suas outras funções.)