

Implementacja maszyny Turinga w maszynie RAM

Maszyna RAM

Niniejszy tekst ukazuje sposób implementacji maszyny Turinga w maszynie RAM. Maszynę RAM (*Random Access Machine*) możemy rozumieć jako abstrakcyjną maszynę wyposażoną w nieskończoną ilość komórek pamięci i możliwość wykonywania na tych komórkach 4 działań:

1. Wyzerowanie komórki o adresie n :
 $Z(0)$
2. Zwiększenie o jeden zawartości komórki o numerze n :
 $S(0)$
3. Skopiowanie zawartości komórki n do komórki m :
 $T(n, m)$
4. Porównanie zawartości komórek n i m . Jeśli zawierają tę samą liczbę, nastąpi skok do linii programu o numerze p :
 $I(n, m, p)$

Powyższy model maszyny RAM można nazwać modelem następnika (ze względu na operację S) z możliwością kopiowania (ze względu na operację T) – gdy zajrzemy na stronę angielskiej Wikipedii¹, to maszyna ta jest połączeniem modelu 2 i 3 z tejże strony.

Maszyna Turinga

Maszyna Turinga jest bardziej podstawowym modelem maszyny liczącej. Ten abstrakcyjny byt składa się z nieskończenie długiej taśmy (w poniższych rozważaniach taśma będzie miała początek, ale nie będzie miała końca), na której mogą być zapisywane określone symbole – na początku wszystkie pola są puste, więc jest na nich symbol oznaczany **B** – czyli *blank*. Maszyna może znajdować się w jednym z wielu stanów. Nad jednym z pól taśmy (na początku nad pierwszym z nich) znajduje się głowica maszyny. Maszyna odczytuje znak z taśmy i w zależności od stanu w którym się znajduje, zapisuje na taśmie inny znak i przesuwa głowicę w lewo lub w prawo (lub pozostawia ją na miejscu). To jaka kombinacja stanu i odczytanego znaku skutkuje jaką reakcją zapisuje się w programie dla maszyny. Oto przykładowy program:

$$q_0, B \rightarrow 1, R, q_1$$
$$q_1, B \rightarrow 1, _, q_f$$

q_0 to stan początkowy maszyny. Znajdując się w stanie początkowym z głowicą nad pustym polem, maszyna zapisuje do tego pola symbol „1”, przesuwa się w prawo (R – prawo, L – lewo) i zmienia swój stan na q_1 . Po przesunięciu znowu znajduje się nad pustym polem, tym razem posiadając stan q_1 . Tak więc znowu zmienia symbol tego pola na „1”, tym razem pozostawiając głowicę na miejscu ($_$) i przechodząc w stan q_f . Stan ten oznacza zakończenie wykonywania programu.

Implementacja

Reprezentacja programu

Ponieważ interesuje nas stworzenie programu maszyny RAM, który wykona program przeznaczony dla maszyny Turinga, należy przyjąć jakąś reprezentację programu maszyny T. w pamięci maszyny RAM (która – przypomnijmy – jest ciągiem komórek zawierających liczby). Ponieważ tak stanów jak i symboli może być przeliczalnie wiele, nie stanowi problemu przypisanie każdemu symbolowi

¹ http://en.wikipedia.org/wiki/Random_access_machine#Refresh:_The_counter_machine_model

i każdemu stanowi liczby naturalnej – i właśnie w ten sposób będą stany i symbole reprezentowane w pamięci maszyny RAM. Jedna linia programu maszyny Turinga może więc zostać zapisana następująco:

Przesunięcie w pamięci	Znaczenie
0	Aktualny stan
1	Zeskanowany symbol
2	Drukowany symbol
3	Przesunięcie (brak: 0, lewo: 1, prawo: 2)
4	Nowy stan

Zakładając, że stanowi q_0 , q_1 , q_f przypiszemy odpowiednio liczby 0, 1, 255, zaś symbolom B, 0 i 1 liczby 0, 48 i 49, program z akapitu „Maszyna Turinga” można zapisać następująco:

RAM	0	0	48	2	1	1	0	49	0	255
Turing	q_0	B	0	R	q_1	q_1	B	1	–	q_f

Taśma

Taśma, na której operuje maszyna Turinga powinna być nieskończona. Jako, że pamięć maszyny RAM również jest nieskończona (a dokładniej, moc zbioru komórek pamięci to alef 0), nie stanowi problemu zapisanie tej taśmy w pamięci – z dokładnością do jednego faktu. Taśma maszyny Turinga nie ma początku ani końca. Natomiast reprezentacja tej taśmy w pamięci maszyny RAM ma swój określony początek. Rozwiązanie tego problemu jest proste. Ponumerujemy kolejne symbole na taśmie liczbami całkowitymi. Miejscu „początkowemu” przypiszmy 0, symbole na prawo oznaczymy jako 1, 2, 3, itd., zaś na lewo -1, -2. Bijekcja obliczająca z numeru miejsca na taśmie numer miejsca w pamięci wygląda następująco:

$$f(x) = x*2, \text{ dla } x \geq 0$$

$$f(x) = x*2-1, \text{ dla } x < 0$$

$$f^{-1}(x) = x/2, \text{ dla } x \text{ parzystego}$$

$$f^{-1}(x) = -(x+1)/2, \text{ dla } x \text{ nieparzystego}$$

Ułożenie danych z taśmy w pamięci jest intuicyjnie oczywiste:

Pozycja w pamięci maszyny RAM	0	1	2	3	4	5	6	7	8	9	10	...
Pozycja na taśmie maszyny Turinga	0	-1	1	-2	2	-3	3	-4	4	-5	5	...

Algorytm

Ponieważ działanie maszyny Turinga nie jest przesadnie skomplikowane, można je zapisać w prosty sposób przy użyciu kodu w języku C:

```
int turing(int current_state, int finish_state, int *program, int
*program_end, int *tape)
{
    int *pc;
    int *current_pos = tape+2; //zaczynamy od pierwszego
miejsca na tasmie
    int absolute_pos;
    int left_shift = 1;
    int right_shift = 2;
```

```

while(current_state != finish_state)
{
    for(pc = program ; pc != program_end ; pc += 5)
    {
        if(*pc == current_state &&
            *(pc+1) == *current_pos)
            break;
    }
    if(pc == program_end)
        return -1;
    *current_pos = *(pc+2);

    absolute_pos = current_pos-tape;
    if(*(pc+3) == left_shift)
    {
        if(absolute_pos==0)
            current_pos=tape+1;
        else if((absolute_pos%2)==0)
            current_pos-=2;
        else if((absolute_pos%2)==1)
            current_pos+=2;
    }
    if(*(pc+3) == right_shift)
    {
        if(absolute_pos==1)
            current_pos=tape;
        else if((absolute_pos%2)==0)
            current_pos+=2;
        else if((absolute_pos%2)==1)
            current_pos-=2;
    }
    current_state = *(pc+4);
}
return 0;
}

```

Jest to gotowa funkcja – przykładowe jej wykorzystanie znajduje się w pliku **simulator.c**.

Nie pozostaje więc nic innego niż zaimplementować ów kod przy użyciu instrukcji maszyny RAM. Nie jest to niestety aż tak proste. W powyższym programie wykorzystywane są dwie tablice – tablica `tape` i `program`. Tablica jest pewnym obszarem w pamięci, początek tego obszaru wskazuje nazwa tablicy (właśnie tak – napis `tape` to zmienna, zawierająca liczbę określającą adres pierwszej komórki tablicy `tape[]`). Aby uzyskać dostęp do n -tej komórki tablicy należy obliczyć jej adres w pamięci, czyli dodać n^2 do adresu początku tablicy. Z tego powodu mogą stosować notację z gwiazdkami, zamiast po „normalnie” odwołać się do któregoś elementu tablicy. Gwiazdka powoduje wyłuskanie zawartości spod danego dalej adresu. Innymi słowy: `program[1]==*(program+1)`. Notacja ta przyda się w dalszej implementacji, dlatego warto zgodzić się na delikatne zaciemnienie powyższego algorytmu – ten koszt się zwróci.

W czasie wykonywania programu będziemy chcieli uzyskać dostęp do różnych komórek pamięci – nasze n będzie się zmieniać. Innymi słowy, w trakcie pisania programu, nie możemy na sztywno wpisać (na przykład w instrukcję `T`) adresu komórki do lub z której chcemy skopiować zawartość –

2 Dla uproszczeni założmy, że jednej komórce w pamięci odpowiada jedna komórka tablicy. Zwykle sprawy są bardziej skomplikowane – gdy na przykład jednej komórce tablicy odpowiadają 4 bajty w pamięci.

adres tej komórki musi zostać wyliczony w trakcie działania programu.

Adresowanie pośrednie

Ujawnia się tu duża uciążliwość RAM maszyny – nie posiada ona instrukcji operujących na adresowaniu pośrednim. Adresowanie pośrednie polega na tym, że chcemy zadziałać na komórce, której adres zawarty jest w innej komórce. Adresowanie pośrednie zapisuje się używając nawiasów kwadratowych. Na przykład dla fragmentu pamięci:

Adres	12	13	14	15	16	17	18
Zawartość	17	0	0	0	0	58	0

Wykonanie instrukcji z adresowaniem bezpośrednim (czyli „normalnym”):

T(12,14)

zaowocuje skopiowaniem treści komórki 12 do komórki 14:

Adres	12	13	14	15	16	17	18
Zawartość	17	0	17	0	0	58	0

Natomiast gdybyśmy wykonali instrukcję z adresowaniem pośrednim:

T([12],14)

to maszyna sprawdzi, jaka liczba jest w komórce 12 i potraktuje tę właśnie liczbę jako adres komórki, z której należy skopiować zawartość. Zostanie więc skopiowana treść komórki o adresie 17:

Adres	12	13	14	15	16	17	18
Zawartość	17	0	58	0	0	58	0

Dopiero wykorzystanie adresowania pośredniego umożliwia implementację tablicy bez konieczności zapisywania w treści programu jawnego (i żmudnego) odwołania do każdego elementu tejże.

Poświęciliśmy tyle uwagi adresowaniu pośredniemu, gdyż kolejnym krokiem implementacji (po wypisaniu algorytmu w C) będzie zapisanie algorytmu dla RAM maszyny, która umożliwia wykorzystanie tego właśnie typu adresowania (w konwencji z nawiasami kwadratowymi). Będzie to kolejny krok w kierunku implementacji dla maszyny, o której napisałem we wstępie.

Implementacja dla maszyny RAM z adresowaniem pośrednim

Wprowadźmy kilka stałych, które wskazywać będą na konkretne komórki pamięci i pozwolą uczynić program nieco bardziej czytelnym:

Nazwa	Opis
<i>current_state</i>	Aktualny stan maszyny.
<i>finish_state</i>	Stan końcowy.
<i>program</i>	Początek programu.
<i>program_end</i>	Koniec programu.
<i>tape</i>	Początek taśmy.
<i>left_shift</i>	Kod przejścia w lewo.
<i>right_shift</i>	Kod przejścia w prawo.
<i>current_pos</i>	Aktualna pozycja na taśmie.

<i>pc</i>	Aktualnie przetwarzana instrukcja.
<i>n, p, q</i>	Zmienne pomocnicze.
<i>C0-C5</i>	Stałe 0-5.
<i>absolute_pos</i>	<i>current_pos-tape</i>
<i>absolute_pos_mod</i>	$(current_pos-tape) \bmod 2$

Parametrami, które powinien uzupełnić użytkownik są *program_length*, *tape*, *program*, *current_state*, *finish_state*. Powinien oczywiście również wprowadzić program o długości pod adres *program* (i jeśli chce na taśmie umieścić jakieś symbole, to powinien to zrobić pod wskazanym przez siebie adresem *tape*).

Tyle słów. Przejdźmy do implementacji.

```
// Etykiety
current_state=0
finish_state=1
program=2
program_end=3
tape=4
left_shift=5
right_shift=6
current_pos=7
pc=8
n=9
p=10
q=11
C0=12
C1=13
C2=14
C3=15
C4=16
C5=17
absolute_pos=18
absolute_pos_mod=19

// C0...5=0...5
Z(C0)
Z(C1)
S(C1)
Z(C2)
S(C2)
S(C2)
T(C2,C3)
S(C3)
T(C3,C4)
S(C4)
T(C4,C5)
S(C5)

// left_shift=1; right_shift=2; current_pos=tape+2
T(C1,left_shift)
T(C2,right_shift)
T(tape,current_pos)
```

```

S(current_pos)
S(current_pos)

while:
I(current_state, finish_state, end)
T(program, pc)

for:
I(pc, program_end, undefined_state)
I([pc], current_state, state_ok)
I(0, 0, continue)

state_ok:
T(pc, n)
S(n)
I([n], [current_pos], found_instruction)

continue:
S(pc)
S(pc)
S(pc)
S(pc)
S(pc)
I(0, 0, for)

found_instruction:
// Drukuj symbol
S(n)
T([n], [current_pos])

S(n)

// absolute_pos = current_pos - tape
// absolute_pos_mod = absolute_pos % 2
Z(p)
Z(absolute_pos)
Z(absolute_pos_mod)
T(tape, p)
subtract_pos_loop:
I(p, current_pos, subtract_pos_end)
S(p)
S(absolute_pos)
S(absolute_pos_mod)
// if(absolute_pos_mod==2) absolute_pos_mod=0;
I(absolute_pos_mod, C2, zero_mod)
I(0, 0, subtract_pos_loop)
zero_mod:
Z(absolute_pos_mod)
I(0, 0, subtract_pos_loop)
subtract_pos_end:

// Czy ruch w lewo?
I([n], left_shift, move_left)

```

```

I(0,0,check_right)

// Ruch w lewo
move_left:
I(absolute_pos,C0,add_one)
I(absolute_pos_mod,C0,subtract_two)
I(absolute_pos_mod,C1,add_two)

check_right:
I([n],right_shift,move_right)
I(0,0,change_state)

move_right:
I(absolute_pos,C1,subtract_two)
I(absolute_pos_mod,C0,add_two)
T(tape,current_pos)
I(0,0,change_state)

// Odejmij 2 od current_pos
subtract_two:
Z(p)
T(C2,q)
subtract_two_loop:
I(q,current_pos,subtract_two_end)
S(p)
S(q)
I(0,0,subtract_two_loop)
subtract_two_end:
T(p,current_pos)
I(0,0,change_state)

// Dodaj 1 do current_pos
add_one:
S(current_pos)
I(0,0,change_state)

// Dodaj 2 do current_pos
add_two:
S(current_pos)
S(current_pos)
I(0,0,change_state)

// Ustaw nowy stan maszyny
change_state:
S(n)
T([n],current_state)

// Powrot do glownej petli
I(0,0,while)

// Koniec
end:
undefined_state:

```

T(0,0)

Program ten można przetestować korzystając z mojej implementacji maszyny RAM, obsługującej także rozkazy z adresowaniem pośrednim³. Natomiast na stronie projektu, czyli http://newton.net.pl/turing_in_ram znajduje się powyższy listing i przykładowe dane testowe.

Jak widać, adresowanie pośrednie wykorzystywane jest tu dość często. Symulator ma tę zaletę, że nie trzeba zmieniać jego treści bez względu na długość tabelki przejść stanów maszyny Turinga i bez względu na długość taśmy – która teoretycznie może być obustronnie nieskończona.

Implementacja bez adresowania pośredniego

Naszym celem jest wszakże zapisanie symulatora dla maszyny RAM bez możliwości wykorzystania adresowania pośredniego. Program stanie się mniej elastyczny – jego długość będzie zależała od maksymalnej przewidywanej długości taśmy i długości tabelki stanów maszyny Turinga.

Aby to zrobić, należy rozwiązać pętlę `for` sprawdzającą kolejne rozkazy i każdy rozkaz sprawdzać oddzielnym fragmentem kodu. Nie jest to eleganckie rozwiązanie, jednak w tej sytuacji jedynie możliwe. Rozwiązanie to co gorsza trzeba będzie powtórzyć – aby odczytać lub nadrukować nową wartość w odpowiednie miejsce taśmy.

```
// Etykiety
current_state=0
finish_state=1
program_end=2
program=3
tape=4
left_shift=5
right_shift=6
current_pos=7
current_symbol=8
pc=9
n=10
p=11
q=12
put_symbol_to=13
C0=14
C1=15
C2=16
C3=17
C4=18
C5=19
absolute_pos=20
absolute_pos_mod=21

// C0...5=0...5
Z(C0)
Z(C1)
S(C1)
Z(C2)
S(C2)
S(C2)
T(C2,C3)
S(C3)
```

³ <http://newton.net.pl/jramachine>

```

T(C3,C4)
S(C4)
T(C4,C5)
S(C5)

// left_shift=1; right_shift=2;
T(C1,left_shift)
T(C2,right_shift)

// program=$PROGRAM
Z(program)
// for(i=0;i<$PROGRAM;i+=1)
// {
S(program)
// }

// tape=$TAPE
Z(tape)
// for(i=0;i<$TAPE;i+=1)
// {
S(tape)
// }
T(tape,current_pos)
S(current_pos)
S(current_pos)

while:
I(current_state,finish_state,end)
T(program,pc)

I(0,0,get_symbol)

for:
// for(i=$PROGRAM;i<$MAX_PROGRAM_END;i+=5)
// {
I(pc,program_end,undefined_state)

I($i,current_state,state_ok_$i)
I(0,0,continue_$i)

state_ok_$i:
I($i+1,current_symbol,found_instruction_$i)
I(0,0,continue_$i)
found_instruction_$i:

// Zapamietaj pozycje glowicy
T(current_pos,put_symbol_to)

// absolute_pos = current_pos - tape
// absolute_pos_mod = absolute_pos % 2
Z(p)
Z(absolute_pos)
Z(absolute_pos_mod)

```

```

T(tape,p)
subtract_pos_loop_$i:
I(p,current_pos,subtract_pos_end_$i)
S(p)
S(absolute_pos)
S(absolute_pos_mod)
// if(absolute_pos_mod==2) absolute_pos_mod=0;
I(absolute_pos_mod,C2,zero_mod_$i)
I(0,0,subtract_pos_loop_$i)
zero_mod_$i:
Z(absolute_pos_mod)
I(0,0,subtract_pos_loop_$i)
subtract_pos_end_$i:

// Czy ruch w lewo?
I($i+3,left_shift,move_left_$i)
I(0,0,check_right_$i)

// Ruch w lewo
move_left_$i:
I(absolute_pos,C0,add_one_$i)
I(absolute_pos_mod,C0,subtract_two_$i)
I(absolute_pos_mod,C1,add_two_$i)

check_right_$i:
I($i+3,right_shift,move_right_$i)
I(0,0,change_state_$i)

move_right_$i:
I(absolute_pos,C1,subtract_two_$i)
I(absolute_pos_mod,C0,add_two_$i)
T(tape,current_pos)
I(0,0,change_state_$i)

// Odejmij 2 od current_pos
subtract_two_$i:
Z(p)
T(C2,q)
subtract_two_loop_$i:
I(q,current_pos,subtract_two_end_$i)
S(p)
S(q)
I(0,0,subtract_two_loop_$i)
subtract_two_end_$i:
T(p,current_pos)
I(0,0,change_state_$i)

// Dodaj 1 do current_pos
add_one_$i:
S(current_pos)
I(0,0,change_state_$i)

// Dodaj 2 do current_pos

```

```

add_two_$i:
S(current_pos)
S(current_pos)
I(0,0,change_state_$i)

// Ustaw nowy stan maszyny
change_state_$i:
T($i+4,current_state)

// Drukuj symbol i wroc do glownej petli
T($i+2,current_symbol)
I(0,0,put_symbol)
continue_$i:
S(pc)
S(pc)
S(pc)
S(pc)
S(pc)
S(pc)
// }
I(0,0,undefined_state)

get_symbol:
T(tape,n)
// for(i=$TAPE;i<=$TAPE_END;i+=1)
// {
I(n,current_pos,get_symbol_ok_$i)
I(0,0,get_symbol_next_$i)

get_symbol_ok_$i:
T($i,current_symbol)
I(0,0,for)

get_symbol_next_$i:
S(n)
// }
I(0,0,undefined_state)

put_symbol:
T(tape,n)
// for(i=$TAPE;i<=$TAPE_END;i+=1)
// {
I(n,put_symbol_to,put_symbol_ok_$i)
I(0,0,put_symbol_next_$i)

put_symbol_ok_$i:
T(current_symbol,$i)
I(0,0,while)

put_symbol_next_$i:
S(n)
// }
I(0,0,undefined_state)

```

```
// Koniec
end:
undefined_state:
T(0,0)
```

Należy się w tym momencie kilka słów wyjaśnienia. Po pierwsze, w miejsce PROGRAM, MAX_PROGRAM_END, TAPE oraz TAPE_END należy wstawić adresy początku i końca odpowiednio programu i taśmy. MAX_PROGRAM_END, jak sama nazwa wskazuje to numer komórki, do której każdy z programów musi się skończyć. Nic jednak nie stoi na przeszkodzie, by programy były krótsze. Poza tym użytkownik powinien uzupełnić w pamięci maszyny RAM parametry *current_state*, *finish_state* oraz *program_end*. Resztę zmiennych maszyna obliczy sama.

Komentarza wymaga także konstrukcja for(...). Jest to konstrukcja informująca, że w tym miejscu należy zwielokrotnić kod, podstawiając pod zmienną \$i numer przejścia. Najlepiej wytłumaczyć to na przykładzie. Fragment:

```
// for(i=0;i<3;i++)
// {
Z($i)
S($i)
// }
```

w rzeczywistości wygląda tak:

```
Z(0)
S(0)
Z(1)
S(1)
Z(2)
S(2)
```

Konstrukcja ta została dodana w celu pewnego uogólnienia schematu programu. Nic nie stoi na przeszkodzie, by z powyższego szablonu sam komputer wygenerował dla nas kod symulatora o zadanych parametrach. Przykładowy pre-kompilator znajduje się na wyżej wspomnianej stronie poświęconej powyższemu projektowi. Przykładowe jego wywołanie może wyglądać tak:

```
$ ./precompiler.rb PROGRAM=25 MAX_PROGRAM_END=95 TAPE=95 \
TAPE_END=110 listing_template.txt > direct.txt
```

Plik *direct.txt* wygenerowany w powyższej stronie, a także dane przykładowe także znajdują się na stronie projektu.