MASZYNA RAM I PREDYKAT KLEENEGO

OPRACOWANIE PRZYGOTOWANE NA PODSTAWIE WYKŁADU
PROF. MARCINA MOSTOWSKIEGO „WPROWADZENIE DO TEORII OBLICZEŃ" PRZEZ
JEGO SKROMNEGO STUDENTA, PANICZA MACIEJA GODKA

1. Maszyna RAM

Maszyna RAM jest pewnym teoretycznym uogólnieniem rzeczywistej maszyny obliczeniowej zaprojektowanej przez Johna Von Neumanna. Abstrakcyjna maszyna RAM składa się z dowolnie dużej skończonej liczby rejestrów (komórek pamięci) r_i, mogących zapamiętać dowolnie dużą liczbę naturalną. Maszyna RAM w trakcie swojej pracy wykonuje jedną z czterech dostępnych instrukcji. (Instrukcję można pojmować jako funkcję przejścia ze stanu pamięci maszyny w pewnym kroku obliczenia t do stanu w kroku $t+1$, gdzie $t \in \mathbb{N}$.)

Definicja 1. Przez instrukcję maszyny RAM rozumieć będziemy jedną z następujących instrukcji:

$$I \in \left\{ \begin{array}{c} r_i := 0, \\ r_i := r_i + 1, \\ r_i := r_j, \\ \text{if } r_i = r_j \text{ goto } k \end{array} \right\}, \text{ gdzie } i, j, k \in \mathbb{N}.$$

W dalszej części tekstu semantyka powyższych instrukcji zostanie szczegółowo wyjaśniona. Na razie możemy nieformalnie powiedzieć, że

- instrukcja $r_i := 0$ (zerowanie) każe maszynie zapamiętać liczbę 0 w i-tym rejestrze pamięci,
- instrukcja $r_i := r_i + 1$ (zwiększanie o 1) każe maszynie zwiększyć zawartość rejestru r_i o jeden,
- instrukcja $r_i := r_j$ (przypisanie) każe maszynie przepisać zawartość komórki r_j do rejestru r_i, zaś
- instrukcja if $r_i = r_j$goto k (skok) każe maszynie wykonać skok do k-tej instrukcji programu, jeśli zawartości rejestrów r_i i r_j są sobie równe. W przeciwnym razie maszyna przejdzie do wykonania kolejnej instrukcji.

Definicja 2. Programem P na maszynę RAM nazwiemy skończony ciąg instrukcji maszyny RAM,

$$P = (I_1, I_2, ..., I_n), \text{ gdzie } n \in \mathbb{N}.$$

Maszyna RAM różni się koncepcyjnie od rozpowszechnionych współcześnie komputerów, ponieważ w przypadku maszyny RAM wykonywane instrukcje nie znajdują się w pamięci roboczej, przez co programy nie mają możliwości modyfikacji swojego kodu. W przypadku popularnych komputerów osobistych poszczególne komórki pamięci mogą być interpretowane jako kod programu, ponieważ zarówno kod, jak i dane są reprezentowane przez pewne ciągi zer i jedynek. O tym, czy dana komórka jest interpretowana jako program, decyduje specjalny rejestr, zwany *wskaźnikiem*

instrukcji albo *licznikiem rozkazów*, zawierający adres[1] kolejnej instrukcji do wykonania.

Chociaż na maszynie RAM program i dane są tworami odmiennej natury, opis działania maszyny RAM również wymaga wprowadzenia wskaźnika instrukcji, który przechowuje numer bieżącej instrukcji, a którego zawartość jest zwiększana o 1 po każdej instrukcji (oprócz instrukcji skoku).

Definicja 3. Konfiguracją (stanem) maszyny RAM ze względu na rejestry $r_1, r_2, ..., r_n$ nazwiemy ciąg $(k, a_1, a_2, ..., a_n)$.

Powyższą definicję można rozumieć w ten sposób, że stan maszyny w dowolnej chwili jest jednoznacznie wyznaczony przez pewien skończony ciąg $n+1$ liczb. Liczby $a_1, a_2, ..., a_n$ stanowią zawartości rejestrów $r_1, r_2, ..., r_n$, zaś liczba k odpowiada zawartości wskaźnika instrukcji.

Definicja 4. Niech będzie dany program wykorzystujący rejestry $r_1, r_2, ..., r_n$, $P = (I_1, I_2, ..., I_t)$. Obliczeniem programu P dla wejścia $b_1, b_2, ..., b_s, s \leq n$ nazywamy ciąg konfiguracji $(c_0, c_1, ..., c_{m-1})$ takich, że

$$c_j = (k_j, a_{1,j}, a_{2,j}, ..., a_{n,j}) \text{ dla } j = 0, 1, ..., m, \text{ gdzie}$$

$$k_0 = 1,$$

$$a_{i,0} = \begin{cases} b_i & \text{dla } i = 1, 2, ..., s \\ 0 & \text{dla } i = s+1, ..., n \end{cases},$$

a ponadto dla dowolnego $j = 0, 1, ..., m-1$ oraz dla dowolnych naturalnych $x, y \in [1, n]$ oraz $z \in [1, t]$, przyjmując $i = 1, 2, ..., n$:

- jeśli I_{k_j} jest „$r_x := 0$”, to

$$k_{j+1} = k_j + 1,$$

$$a_{i,j+1} = \begin{cases} 0 & \text{dla } i = x \\ a_{i,j} & \text{dla } i \neq x \end{cases};$$

- jeśli I_{k_j} jest „$r_x := r_x + 1$”, to

$$k_{j+1} = k_j + 1,$$

$$a_{i,j+1} = \begin{cases} a_{i,j} + 1 & \text{dla } i = x \\ a_{i,j} & \text{dla } i \neq x \end{cases};$$

- jeśli I_{k_j} jest „$r_x := r_y$”, to

$$k_{j+1} = k_j + 1,$$

$$a_{i,j+1} = \begin{cases} a_{y,j} & \text{dla } i = x \\ a_{i,j} & \text{dla } i \neq x \end{cases};$$

[1]Omówienie maszyny RAM nie wymaga wprowadzenia pojęcia adresu, ponieważ używamy tylko pewnych systematycznych nazw komórek, lub też – w terminologii stosowanej przez informatyków – adresowania bezpośredniego. Dla wyjaśnienia, czym jest adres w pamięci, możemy jednak przyjąć, że adresem komórki r_i jest i.

- jeśli I_{j_k} jest „if $r_x := r_y$ goto z”, to

$$a_{i,j+1} = a_{i,j},$$

$$k_{j+1} = \begin{cases} z, & \text{jeśli } a_{x,j} = a_{y,j} \\ k_j + 1, & \text{jeśli } a_{x,j} \neq a_{y,j} \end{cases}.$$

Powyższa definicja określa w sposób formalny semantykę instrukcji maszyny RAM.

Fakt 5. *Dla dowolnego programu P wykorzystującego rejestry $r_1, r_2, ..., r_n$, z wejściem $b_1, b_2, ..., b_s$ $(s \leq n)$ istnieje co najwyżej jedno obliczenie.*

Kodowanie maszyny RAM i Predykat T Kleenego

Dowolne skończone ciągi liczb naturalnych można zakodować jako pojedynczą liczbę naturalną. Do kodowania pomocne jest zasadnicze twierdzenie arytmetyki.

Twierdzenie. *(Zasadnicze twierdzenie arytmetyki) Dla każdej liczby naturalnej $n > 1$ istnieje $k < n$ takie, że dla rosnącego ciągu $(p_{(1)}, p_{(2)}, ..., p_{(k)})$ k początkowych liczb pierwszych oraz dla pewnego jedynego ciągu liczb naturalnych $(q_1, q_2, ..., q_k)$*

$$n = p_{(1)}^{q_1} \cdot p_{(2)}^{q_2} \cdot ... \cdot p_{(k)}^{q_k} = \prod_{i=1}^{k} p_{(i)}^{q_i},$$

Innymi słowy, każdą liczbę naturalną większą od 1 można jednoznacznie przedstawić w postaci iloczynu naturalnych potęg liczb pierwszych.

Zasadnicze twierdzenie arytmetyki nie tylko stwierdza o możliwości kodowania ciągu, ale – co więcej – daje wskazówkę, jak można takie kodowanie przeprowadzić. Pomysł będzie polegał na tym, aby zadany ciąg liczb zakodować w wykładnikach kolejnych liczb pierwszych.

Definicja 6. Niech dany będzie ciąg $(a_1, a_2, ..., a_n)$ liczb naturalnych. *Kodowaniem* tego ciągu nazwiemy liczbę l taką, że

$$l = c((a_i)_{i=1}^{n}) = p_{(1)}^{a_1+1} \cdot p_{(2)}^{a_2+1} \cdot ... \cdot p_{(n)}^{a_n+1},$$

gdzie $p_{(i)}$ oznacza i-tą liczbę pierwszą. Funkcję $c : \mathbb{N}^n \mapsto \mathbb{N}$ nazwiemy zaś *funkcją kodującą*. Dodatkowo moglibyśmy przyjąć, że liczba 1 koduje ciąg pusty (co jednak wymagałoby modyfikacji poniższych definicji). Zamiast tego będziemy zakładać, że operujemy na ciągach niepustych.

Zakodowawszy ciąg do postaci liczby, możemy mówić o jego własnościach w oparciu o własności liczb.

Możemy na przykład określić długość ciągu jako numer największej liczby pierwszej, która dzieli kod ciągu[2]:

$$lh(l) = \max\{i < l : (p_{(i)} \mid l)\}.$$

Zauważmy ponadto, że nie każda liczba naturalna koduje jakiś ciąg. W szczególności żadna niezerowa potęga naturalna liczby pierwszej większej od 2 nie koduje żadnego ciągu, ponieważ kod każdego ciągu musi zawierać w swoim rozkładzie na

[2]jeżeli ktoś martwi się własnościami obliczeniowymi zdefiniowanej w ten sposób funkcji długości, może ją alternatywnie zdefiniować jako $ln(c) = min\{i : p_{(i+1)} \nmid c\}$

liczby pierwsze wszystkie liczby pierwsze mniejsze od największej liczby pierwszej występującej w rozkładzie (tj. $p_{(ln(l))}$):

$$SEQ(l) \equiv \forall_{0<i\leq ln(l)}(p_{(i)}|l).$$

Co więcej, możemy nasz ciąg zdekodować, czyli zdefiniować funkcję, której wartością jest i-ty wyraz ciągu:

$$(l)_i = s \equiv (p_{(i)}^{s+1}) \mid (l \wedge p_{(i)}^{s+2} \nmid l).$$

Na mocy twierdzenia, dla $i = 1, ..., n$ zachodzi $a_i = (c((a_k)_{k=1}^n))_i$.

Skoro możemy kodować skończone ciągi liczb naturalnych, to możemy zapewne również kodować skończone ciągi skończonych ciągów liczb naturalnych. W szczególności zaś możemy zakodować obliczenia dowolnego obliczalnego programu P na maszynę RAM.

Taka operacja pozwala nam mówić o programach w języku arytmetyki, co osobom przyzwyczajonym do języka arytmetyki zapewne wydaje się wygodne, a może nawet eleganckie.

Prawdopodobnie taką właśnie osobą był Stephen Kleene, który zdefiniował predykat określający, czy dana liczba naturalna koduje obliczenie:

Definicja 7. Powiemy, że l jest kodem zakończonego obliczenia programu kodowanego przez e z wejściem $a_1, a_2, ..., a_n$ wtedy i tylko wtedy, gdy spełnione są następujące warunki:

- e jest kodowaniem ciągu,
- e jest kodowaniem programu,
- liczba l koduje warunki początkowe obliczenia programu e dla wejścia $a_1, a_2, ..., a_n$,
- liczba l koduje warunek końcowy dla programu e,
- wszystkie kolejne konfiguracje są zgodne z określoną w definicji 4 funkcją przejścia

Innymi krzaki,

$$T_n(e, a_1, a_2, ..., a_n, l) \equiv \left(\begin{array}{c} SEQ(e) \wedge PROG(e) \\ \wedge \\ START(e, a_1, a_2, ..., a_n, l) \\ \wedge \\ FINISH(e, l) \\ \wedge \\ \forall_{0<i<lh(l)} STEP(e, i, l) \end{array} \right).$$

$PROG(e)$ oznacza, że e jest kodem programu, czyli że i-ty element ciągu kodowanego przez e jest kodem instrukcji maszyny RAM:

$$PROG(e) \equiv_{df} \forall_{0<i\leq lh(e)} \left(Zero((e)_i) \vee Increase((e)_i) \vee Assign((e)_i) \vee Jump((e)_i)\right).$$

Nieco uwagi należy poświęcić sposobowi kodowania instrukcji. Spośród czterech prezentowanych tu instrukcji, dwie (zerowanie i inkrementacja) przyjmują jeden argument (indeks rejestru docelowego), jedna (przypisanie) przyjmuje dwa argumenty (indeks rejestru docelowego i źródłowego), i jedna (skok) przyjmuje trzy argumenty (dwa indeksy porównywanych rejestrów oraz numer instrukcji docelowej). Oznacza to, że będziemy musieli zakodować łącznie siedem wartości. Do ich zakodowania użyjemy kolejnych początkowych liczb pierwszych $p_{(1)}, p_{(2)}, ..., p_{(7)}$ (wybór zestawu liczb pierwszych użytych do kodowania instrukcji jest całkowicie arbitralny).

Oznaczając przez $\ulcorner\alpha\urcorner$ kod instrukcji α, możemy określić:

- $\ulcorner r_i := 0\urcorner = p_{(1)}^{i+1} = 2^{i+1}$,
- $\ulcorner r_i := r_i + 1\urcorner = p_{(2)}^{i+1} = 3^{i+1}$,
- $\ulcorner r_i := r_j\urcorner = p_{(3)}^{i+1} \cdot p_{(4)}^{j+1} = 5^{i+1} \cdot 7^{j+1}$,
- $\ulcorner$if $r_i = r_j$ goto $k\urcorner = p_{(5)}^{i+1} \cdot p_{(6)}^{j+i} \cdot p_{(7)}^{k+1} = 11^{i+1} \cdot 13^{j+1} \cdot 17^{k+1}$.

Wówczas predykaty określające, czy dana liczba jest kodem instrukcji, możemy zdefiniować następująco:

$$Zero(w) \equiv_{df} (p_{(1)}|w) \wedge \forall_{i<w}(p_{(i)}|w \to i = 1);$$
$$Increase(w) \equiv_{df} (p_{(2)}|w) \wedge \forall_{i<w}(p_{(i)}|w \to i = 2);$$
$$Assign(w) \equiv_{df} (p_{(3)}|w) \wedge (p_{(4)}|w) \wedge \forall_{i<w}(p_{(i)}|w \to i \in \{3,4\});$$
$$Jump(w) \equiv_{df} (p_{(5)}|w) \wedge (p_{(6)}|w) \wedge (p_{(7)}|w) \wedge \forall_{i<w}(p_{(i)}|w \to i \in \{5,6,7\}).$$

Zakładając, że w jest kodem instrukcji, możemy określić wartości argumentów tych instrukcji:

- dla zerowania:
$$Reg_Z(w) = s \equiv (Zero(w) \wedge (p_{(1)}^{s+1} \mid w) \wedge (p_{(1)}^{s+2} \nmid w));$$
- dla inkrementacji:
$$Reg_I(w) = s \equiv (Increase(w) \wedge (p_{(2)}^{s+1} \mid w) \wedge (p_{(2)}^{s+2} \nmid w));$$
- dla przypisania:
$$Reg_A^1(w) = s \equiv (Assign(w) \wedge (p_{(3)}^{s+1} \mid w) \wedge (p_{(3)}^{s+2} \nmid w)),$$
$$Reg_A^2(w) = s \equiv (Assign(w) \wedge (p_{(4)}^{s+1} \mid w) \wedge (p_{(4)}^{s+2} \nmid w));$$
- dla skoku:
$$Reg_J^1(w) = s \equiv (Jump(w) \wedge (p_{(5)}^{s+1}|w) \wedge (p_{(5)}^{s+2} \nmid w)),$$
$$Reg_J^2(w) = s \equiv (Jump(w) \wedge (p_{(6)}^{s+1} \mid w) \wedge (p_{(6)}^{s+2} \nmid w)),$$
$$Goto(w) = s \equiv (Jump(w) \wedge (p_{(7)}^{s+1} \mid w) \wedge (p_{(7)}^{s+2} \nmid w)).$$

Ponadto możemy zdefiniować funkcję określającą maksymalny indeks rejestru użyty w programie e:

$$Regmax(w) = \begin{cases} Reg_Z(w), & \text{jeśli } Zero(w) \\ Reg_I(w), & \text{jeśli } Increase(w) \\ \max\{Reg_A^1(w), Reg_A^2(w)\}, & \text{jeśli } Assing(w) \\ \max\{Reg_J^1(w), Reg_J^2(w)\}, & \text{jeśli } Jump(w) \end{cases},$$

$$Regs(e) = \max_{i=1}^{lh(e)}\{Regmax((e)_i)\}.$$

Dysponując funkcją $Regs$, możemy doprecyzować znaczenie pozostałych predykatów użytych w definicji T_n. Przede wszystkim musimy pamiętać, że l jest kodowaniem ciągu ciągów o stałej długości, mianowicie ciągów konfiguracji określonych w definicji 4 na stronie 2. Odnosząc się do użytych tam oznaczeń, zapis $(l)_j$ będzie desygnował

konfigurację c_{j-1}, zaś zapis $((l)_j)_i$ będzie oznaczał k_{j-1} dla $i = 1$, oraz $a_{i-1,j-1}$ dla $i > 1$.

$$START(e, a_1, a_2, ..., a_n, l) \equiv \begin{pmatrix} \forall_{0<i\leq lh(l)} lh((l)_i) = max(n, Regs(e)) \\ \wedge \\ ((l)_1)_1 = 1 \\ \wedge \\ \forall_{1<i\leq lh((l)_1)}((l)_1)_i = \begin{cases} a_{i-1}, & \text{jeżeli } i \leq n \\ 0, & \text{jeżeli } i > n \end{cases} \end{pmatrix}$$

Powyższe sformułowanie odpowiada określeniu konfiguracji początkowej z definicji 4.

Program e kończy się, jeśli licznik rozkazów w ostatnim korku (w ostatniej konfiguracji) przekroczy długość programu:

$$FINISH(e, l) \equiv ((l)_{lh(l)})_1 \geq lh(e).$$

Wreszcie, należy zapewnić, że kody kolejnych stanów maszyny będą zgodne ze sformułowaną w definicji 4 semantyką:

$$STEP(e, i, l) \equiv \begin{pmatrix} Zero((e)_t) \wedge Trans_Z((l)_i, (l)_{i+1}, Reg_Z((e_t)) \\ \vee \\ Increase((e)_t) \wedge Trans_I((l)_i, (l)_{i+1}, Reg_I((e)_t)) \\ \vee \\ Assing((e)_t) \wedge Trans_A((l)_i, (l)_{i+1}, Reg^1_A((e)_t), Reg^2_A((e)_t)) \\ \vee \\ Jump((e)_t) \wedge Trans_J((l)_i, (l)_{i+1}, Reg^1_J((e)_t), Reg^2_J((e)_t), Goto((e)_t)) \end{pmatrix}_{t=((l)_i)_0},$$

gdzie:

$$Trans_Z(a, b, x) \equiv \begin{pmatrix} (b)_1 = (a)_1 + 1 \\ \wedge \\ \forall_{1<i\leq lh(b)}(b)_i = \begin{cases} 0, & \text{jeśli } i = x + 1 \\ (a)_i, & \text{jeśli } i \neq x + 1 \end{cases} \end{pmatrix},$$

$$Trans_I(a, b, x) \equiv \begin{pmatrix} (b)_1 = (a)_1 + 1 \\ \wedge \\ \forall_{1<i\leq lh(b)}(b)_i = \begin{cases} (a)_i + 1, & \text{jeśli } i = x + 1 \\ (a)_i, & \text{jeśli } i \neq x + 1 \end{cases} \end{pmatrix},$$

$$Trans_A(a, b, x, y) \equiv \begin{pmatrix} (b)_1 = (a)_1 + 1 \\ \wedge \\ \forall_{1<i\leq lh(b)}(b)_i = \begin{cases} (a)_y, & \text{jeśli } i = x + 1 \\ (a)_i, & \text{jeśli } i \neq x + 1 \end{cases} \end{pmatrix},$$

$$Trans_J(a, b, x, y, z) \equiv \begin{pmatrix} \forall_{1<i\leq lh(b)}(b)_i = (a)_i \\ \wedge \\ (b)_1 = \begin{cases} z, & \text{jeśli } (a)_{x+1} = (a)_{y+1} \\ (a)_1 + 1, & \text{jeśli } (a)_{x+1} \neq (a)_{y+1} \end{cases} \end{pmatrix}.$$