Wstęp do teorii obliczeń

Paweł Urzyczyn
urzy@mimuw.edu.pl

19 czerwca 2017, godzina 14:47

1 Funkcje częściowo rekurencyjne

Pojęcie funkcji obliczalnej i częściowo obliczalnej może zostać zdefiniowane na wiele różnych sposobów. Dziś najczęściej odwołujemy się w tym celu do maszyn Turinga, ale czasem wygodnie jest użyć pochodzącej od Kleene'ego definicji funkcji (częściowo) rekurencyjnych. Rozważamy funkcje częściowe nad $\mathbb{N}$ o dowolnej liczbie argumentów. Funkcje częściowo rekurencyjne to funkcje otrzymane z funkcji bazowych (zero, następnik, rzutowania) za pomocą trzech operacji: składania, rekursji prostej i minimum. Sformułujemy teraz ścisłą definicję tych pojęć. Napis $f : \mathbb{N}^k \multimap \mathbb{N}$ czytamy „f jest k-argumentową funkcją częściową o argumentach i wartościach w $\mathbb{N}$".

Funkcje bazowe: Jako *funkcje bazowe* przyjmujemy:

- następnik, $succ(n) = n + 1$;
- rzuty, $\Pi^k_i(n_1, \ldots, n_k) = n_i$, gdzie $k \geq 1$ oraz $i \leq k$;
- funkcje stale równe zeru, $Z_k(n_1, \ldots, n_k) = 0$, gdzie $k \geq 1$.

Składanie: Funkcja $f : \mathbb{N}^k \multimap \mathbb{N}$ powstaje przez *składanie* funkcji $h : \mathbb{N}^\ell \multimap \mathbb{N}$ z funkcjami $g_1, \ldots, g_\ell : \mathbb{N}^k \multimap \mathbb{N}$, jeżeli

- $\mathrm{Dom}(f) = \{\vec{n} \mid \vec{n} \in \mathrm{Dom}(g_i)$ dla wszystkich i, oraz $(g_1(\vec{n}), \ldots, g_\ell(\vec{n})) \in \mathrm{Dom}(h)\}$;
- $f(\vec{n}) = h(g_1(\vec{n}), \ldots, g_\ell(\vec{n}))$, dla dowolnego $\vec{n} \in \mathrm{Dom}(f)$.

Rekursja: Funkcja $f : \mathbb{N}^{k+1} \multimap \mathbb{N}$ powstaje przez *rekursję prostą* z funkcji $h : \mathbb{N}^{k+2} \multimap \mathbb{N}$ i funkcji $g : \mathbb{N}^k \multimap \mathbb{N}$, jeżeli dla dowolnego $n \in \mathbb{N}$ i dowolnego $\vec{m} \in \mathbb{N}^k$ spełnione są równania

- $f(0, \vec{m}) = g(\vec{m})$;
- $f(succ(n), \vec{m}) = h(f(n, \vec{m}), n, \vec{m})$,

przy czym równanie uważamy za spełnione, gdy obie strony są określone i równe, lub obie strony są nieokreślone. Wartość wyrażenia $h(f(n, \vec{m}), n, \vec{m})$ jest określona wtedy i tylko wtedy gdy $(n, \vec{m}) \in \mathrm{Dom}(f)$ oraz $(f(n, \vec{m}), n, \vec{m}) \in \mathrm{Dom}(h)$.

Minimum: Funkcja $f : \mathbb{N}^k \multimap \mathbb{N}$ powstaje z funkcji $h : \mathbb{N}^{k+1} \multimap \mathbb{N}$ przez zastosowanie *operacji minimum*, co zapisujemy $f(\vec{n}) = \mu y[h(\vec{n}, y) = 0]$, gdy dla dowolnego $\vec{n} \in \mathbb{N}^k$:

- Jeśli istnieje takie m, że $h(\vec{n}, m) = 0$ oraz wszystkie wartości $h(\vec{n}, i)$ dla $i < m$ są określone i różne od zera, to $f(\vec{n}) = m$.
- Jeśli takiego m nie ma, to $f(\vec{n})$ jest nieokreślone.

Definicja 1.1 Klasa funkcji *częściowo rekurencyjnych* to najmniejsza klasa funkcji częściowych nad $\mathbb{N}$ zawierająca funkcje bazowe i zamknięta ze względu na składanie, rekursję prostą i operację minimum. Funkcje *rekurencyjne* to te funkcje częściowo rekurencyjne, które są całkowite (zawsze określone).

Klasa funkcji *pierwotnie rekurencyjnych* to najmniejsza klasa funkcji całkowitych nad $\mathbb{N}$ zawierająca funkcje bazowe i zamknięta ze względu na składanie i rekursję prostą.

Funkcje charakterystyczne: Z każdą relacją k-argumentową $r \subseteq \mathbb{N}^k$ wiążemy dwie funkcje:

$$c_r(\vec{n}) = \begin{cases} 0, & \text{jeśli } \vec{n} \in r; \\ 1, & \text{w przeciwnym przypadku,} \end{cases}$$

$$\chi_r(\vec{n}) = \begin{cases} 0, & \text{jeśli } \vec{n} \in r; \\ \text{nieokreślone}, & \text{w przeciwnym przypadku.} \end{cases}$$

Pierwsza z nich to zwykła funkcja charakterystyczna zbioru r, drugą nazywamy *częściową funkcją charakterystyczną* r. Relacje reprezentujemy za pomocą ich funkcji charakterystycznych. Piszemy np. w skrócie $\mu y[r(\vec{n}, y)]$ zamiast $\mu y[\chi_r(\vec{n}, y) = 0]$.

Definicja 1.2 Mówimy, że zbiór $A \subseteq \mathbb{N}^k$ jest *rekurencyjny*, gdy funkcja c_A jest rekurencyjna, a *rekurencyjnie przeliczalny*, gdy funkcja χ_A jest częściowo rekurencyjna. (W tym drugim przypadku mówi się też, że *relacja A jest częściowo rekurencyjna*.)

Fakt 1.3 *Każdy zbiór rekurencyjny jest rekurencyjnie przeliczalny.*

Dowód: Wynika to z równości $\chi_A(\vec{n}) = \mu y[c_A(\vec{n}) = 0]$, którą można ściśle wyrazić z pomocą rzutowań. Dla $A \subseteq \mathbb{N}$ mamy na przykład $\chi_A(n) = \mu y[c_A(\Pi^2_1(n, y)) = 0]$. □

Funkcje pierwotnie rekurencyjne

Najczęściej mamy do czynienia z funkcjami i relacjami pierwotnie rekurencyjnymi. Na przykład dodawanie określone jest przez rekursję prostą z pomocą funkcji następnika:

$$\begin{aligned} 0 + m &= m; \\ succ(n) + m &= succ(n + m). \end{aligned}$$

Definicja ta jest zgodna ze schematem rekursji prostej dla $k = 1$ oraz $g(m) = \Pi^1_1(m)$ i $h(l, n, m) = succ(\Pi^3_1(l, n, m))$. Podobnie definiujemy mnożenie i potęgowanie. Przykłady te

pokazują jak można manipulować argumentami: jeśli k-argumentowa funkcja f jest pierwotnie rekurencyjna, to także l-argumentowa funkcja $g(n_1,\ldots,n_l) = f(n_{i_1},\ldots,n_{i_k})$ jest pierwotnie rekurencyjna, bo jest złożeniem funkcji f z odpowiednimi rzutowaniami.

Każda funkcja stała $f(\vec{n}) = m$ jest pierwotnie rekurencyjna jako złożenie $f(\vec{n}) = succ^m(Z_k(\vec{n}))$. Aby określić funkcję poprzednika *pred* napiszemy najpierw równania

$$\begin{aligned} p(0,m) &= Z_1(m); \\ p(n+1,m) &= \Pi^3_2(p(n),n,m), \end{aligned}$$

a następnie użyjemy złożenia:

$$pred(n) := p(n,0) = p(n,Z_1(n)).$$

Teraz można już zdefiniować (nieujemne) odejmowanie:

$$\begin{aligned} m \dot{-} 0 &= m; \\ m \dot{-} (n+1) &= pred(m \dot{-} n). \end{aligned}$$

Następna prosta, ale pożyteczna funkcja to test na zero:

$$\begin{aligned} sg(0) &= 0; \\ sg(n+1) &= 1. \end{aligned}$$

Wśród najprostszych relacji pierwotnie rekurencyjnych mamy nierówność ostrą i nieostrą:

$$c_{\leq}(m,n) = sg(m \dot{-} n), \qquad c_{<}(m,n) = sg((m+1) \dot{-} n).$$

Operacje logiczne nie wyprowadzają poza klasę relacji pierwotnie rekurencyjnych:

$$c_{r\vee p}(\vec{n}) = c_r(\vec{n}) \cdot c_p(\vec{n}), \qquad c_{r\wedge p}(\vec{n}) = sg(c_r(\vec{n}) + c_p(\vec{n})), \qquad c_{\neg r} = 1 \dot{-} c_r(\vec{n}),$$

a więc np. równość, jako koniunkcja dwóch nierówności, też jest pierwotnie rekurencyjna. Klasa relacji pierwotnie rekurencyjnych jest też zamknięta ze względu na *kwantyfikatory ograniczone*. Oznacza to, że jeśli zdefiniujemy relację r warunkiem

$$r(n,\vec{m}) \equiv \exists y \leq n.\, s(y,\vec{m}),$$

w którym s jest pierwotnie rekurencyjna, to także r jest pierwotnie rekurencyjna. Istotnie:

$$\begin{aligned} r(0,\vec{m}) &= s(0,\vec{m}); \\ r(n+1,\vec{m}) &= r(n,\vec{m}) \vee s(n+1,\vec{m}). \end{aligned}$$

Zajmiemy się teraz jeszcze kilkoma sposobami definiowania funkcji pierwotnie rekurencyjnych. Najpierw definicje warunkowe. Jeśli funkcje g i h oraz relacja r są pierwotnie rekurencyjne i

$$f(\vec{n}) = \begin{cases} g(\vec{n}), & \text{jeśli } r(\vec{n}); \\ h(\vec{n}), & \text{w przeciwnym przypadku,} \end{cases}$$

to wtedy $f(\vec{n}) = g(\vec{n}) \cdot (1 \dot{-} c_r(\vec{n})) + h(\vec{n}) \cdot c_r(\vec{n})$ jest pierwotnie rekurencyjna. Możemy też definiować funkcje za pomocą *minimum ograniczonego*, rozumianego tak:

$$\mu y \leq n[r(y,\vec{m})] = \begin{cases} \mu y[y \leq n \wedge r(y,\vec{m})], & \text{jeśli minimum jest określone;} \\ 0, & \text{w przeciwnym przypadku.} \end{cases}$$

Minimum ograniczone definiujemy przez rekursję prostą (por. kwantyfikator ograniczony), więc jest to funkcja pierwotnie rekurencyjna, o ile tylko taka jest relacja r. Podobnie jest z operacją *maksimum ograniczonego*

$$\nu y \leq n[r(y, \vec{m})] \quad := \quad \mu y \leq n[r(y, \vec{m}) \wedge \forall z \leq n.\, r(z, \vec{m}) \to z \leq y].$$

Minimum i maksimum ograniczone to bardzo pożyteczne narzędzia, pozwalające łatwo definiować rozmaite funkcje, np. dzielenie całkowite to:

$$\left\lfloor \frac{m}{n} \right\rfloor = \nu y \leq m[y \cdot n \leq m].$$

Kodowanie słów

Dwie liczby naturalne można zakodować za pomocą jednej liczby, używając funkcji pary

$$\langle m, n \rangle = \left\lfloor \frac{(m+n)(m+n+1)}{2} \right\rfloor + m.$$

Funkcje odwrotne do funkcji pary to takie funkcje ℓ i r, że $\langle \ell(n), r(n) \rangle = n$ zachodzi dla wszystkich $n \in \mathbb{N}$. Zarówno $\langle\ ,\ \rangle$ jak i jej funkcje odwrotne, to funkcje pierwotnie rekurencyjne, na przykład $\ell(n) = \mu y \leq n[\exists x \leq n.\, \langle y, x \rangle = n]$. Jeśli trzeba zakodować ciąg liczb $a_0, \ldots, a_k$ o dowolnej długości $k+1$, to można użyć jednoznaczności rozkładu na czynniki pierwsze:

$$kod(a_0 \ldots a_k) = 2^{a_0} 3^{a_1} \cdots p_k^{a_k},$$

gdzie p_i dla $i \in \mathbb{N}$ to ciąg rosnący wszystkich liczb pierwszych. Uwaga: to kodowanie „skleja" ze sobą wszystkie ciągi kończące się zerami i nigdy nie daje w wyniku zera. Ale nam to akurat nie przeszkodzi, a nawet zaraz się przyda.

Aby swobodnie posługiwać się powyższym kodowaniem, zauważmy, że następujące funkcje i relacje są pierwotnie rekurencyjne:

- relacja „n jest liczbą pierwszą": $\neg\exists y \leq n \exists z \leq n(y \cdot z = n \wedge y \neq 1 \wedge z \neq 1)$;
- funkcja p_i dana[1] równaniami $p_0 = 2$, $p_{n+1} = \mu y \leq 2p_n[y \text{ pierwsze i } y > p_n]$;
- funkcja $@(m, n) := \mu y \leq m[p_n^y \mid m \wedge \neg(p_n^{y+1} \mid m)]$ — „n-ty wyraz ciągu o kodzie m";
- funkcja $m[i \leftarrow n] \quad := \quad \mu y \leq m \cdot p_i^n[\forall z \leq m(z \neq i \to @(y, z) = @(m, z)) \wedge @(y, i) = n]$ — „zmiana i-tego wyrazu w ciągu o kodzie m na liczbę n".

Maszyny Turinga

Przypomnijmy, że (deterministyczną, jednotaśmową) *maszynę Turinga nad alfabetem* $\mathcal{A}$ można zdefiniować jako krotkę $\mathcal{M} = \langle \Delta, Q, \delta, q_0, q_a, q_r \rangle$, gdzie:

- Δ jest skończonym alfabetem, zawierającym $\mathcal{A}$ oraz symbol $\mathsf{B} \notin \mathcal{A}$ (blank);

[1] Wiadomo, że pomiędzy liczbami m i $2m$ zawsze jest liczba pierwsza.

- Q jest skończonym zbiorem *stanów*;
- $q_0 \in Q$ jest stanem *początkowym*;
- $q_a \in Q$ jest stanem *akceptującym*;
- $q_r \in Q$ jest stanem *odrzucającym*;
- $\delta : (Q - \{q_a, q_r\}) \times \Delta \to \Delta \times Q \times \{-1, 0, +1\}$ jest *funkcją przejścia*.

Zakładając, że zbiory Δ i Q są rozłączne, można zdefiniować *konfigurację* maszyny jako trójkę postaci (q, i, w), gdzie $q \in Q$, $i \in \mathbb{N}$ oraz $w \in \Delta^*$, przy czym utożsamiamy konfiguracje (q, i, w) oraz $(q, i, w\mathsf{B})$. Interpretacja tej definicji jest następująca. Taśma maszyny jest nieskończona w prawo. Na początku taśmy zapisane jest słowo w, dalej w prawo są same blanki, a głowica maszyny znajduje się w pozycji i. Konfigurację postaci $\mathcal{C}_w = (q_0, 0, w)$, gdzie $w \in \mathcal{A}^*$, nazywamy *początkową*. Konfiguracje *akceptujące* (odp. *odrzucające*) są zaś postaci (q_a, i, w) (odp. (q_r, i, w)).

Jeśli $\mathcal{C} = (q, i, w)$ oraz $w = a_0 \dots a_k$, to *następna* konfiguracja $\mathcal{C}'$ jest określona tak:

- Jeśli $\delta(q, a) = (b, p, +1)$ to $\mathcal{C}' = (p, i + 1, w[i \leftarrow b])$;
- Jeśli $\delta(q, a) = (b, p, 0)$ to $\mathcal{C}' = (p, i, w[i \leftarrow b])$;
- Jeśli $\delta(q, a) = (b, p, -1)$ to $\mathcal{C}' = (p, i \dot{-} 1, w[i \leftarrow b])$.

Piszemy $\mathcal{C} \to_{\mathcal{M}} \mathcal{C}'$, a symbolem $\twoheadrightarrow_{\mathcal{M}}$ oznaczamy przechodnio-zwrotne domknięcie relacji $\to_{\mathcal{M}}$. Jeżeli $\mathcal{C}_w \twoheadrightarrow_{\mathcal{M}} \mathcal{C}'$, gdzie $\mathcal{C}'$ jest konfiguracją akceptującą (odp. odrzucającą) to mówimy, że maszyna *akceptuje* (odp. *odrzuca*) słowo w. Maszyna *zatrzymuje się* dla wejścia w, wtedy i tylko wtedy, gdy słowo w jest akceptowane lub odrzucane.

Obliczenie rozpoczynające się od konfiguracji $\mathcal{C}_0$ to skończony lub nieskończony ciąg konfiguracji $\mathcal{C}_0 \to_{\mathcal{M}} \mathcal{C}_1 \to_{\mathcal{M}} \mathcal{C}_2 \to_{\mathcal{M}} \cdots$ Obliczenie skończone jest *akceptujące* lub *odrzucające*, w zależności od ostatniego stanu. Oczywiście maszyna deterministyczna ma obliczenie nieskończone rozpoczynające się od $\mathcal{C}_w$ wtedy i tylko wtedy, gdy nie zatrzymuje się dla wejścia w.

Przez $L(\mathcal{M})$ oznaczamy język złożony ze wszystkich słów akceptowanych przez maszynę $\mathcal{M}$. Zbiór słów $L \subseteq \mathcal{A}^*$ jest *częściowo obliczalny* wtedy i tylko wtedy, gdy $L = L(\mathcal{M})$ dla pewnej maszyny $\mathcal{M}$. Jeśli dodatkowo maszyna $\mathcal{M}$ odrzuca dokładnie te słowa, które nie należą do L, to mówimy, że L jest *obliczalny*.

Fakt 1.4 *Następujące warunki są równoważne:*

1. *Z jest zbiorem obliczalnym;*
2. *$-Z$ jest zbiorem obliczalnym;*
3. *Z i $-Z$ są częściowo obliczalne;*

Dowód: Jeśli dwie różne deterministyczne maszyny akceptują języki Z i $-Z$, to należy uruchomić je jednocześnie i zobaczyć, która zaakceptuje słowo wejściowe (jedna musi). Tę pracę może wykonać jedna maszyna deterministyczna. □

Jeśli $\mathcal{A} = \{1\}$, a każdą liczbę naturalną n zinterpretujemy jako ciąg jedynek długości n, to język $L(\mathcal{A})$ można uważać za podzbiór $\mathbb{N}$. Możemy też użyć maszyny Turinga do definiowania funkcji $f_{\mathcal{M}} : \mathbb{N} \multimap \mathbb{N}$. Jeśli $\mathcal{M}$ zatrzymuje się dla wejścia n (tj. słowa 1^n), to za wartość funkcji $f_{\mathcal{M}}(n)$ przyjmujemy pozycję głowicy w odpowiedniej konfiguracji końcowej. W przeciwnym razie wartość $f_{\mathcal{M}}(n)$ uważamy za nieokreśloną. Rozdzielając ciągi jedynek znakiem $\sharp$, możemy także definiować maszyny akceptujące relacje wieloargumentowe i obliczające wieloargumentowe funkcje.

Definicja 1.5 Funkcja częściowa $f : \mathbb{N}^k \multimap \mathbb{N}$ jest *częściowo obliczalna*, wtedy i tylko wtedy, gdy jest postaci $f_{\mathcal{M}}$ dla pewnej maszyny $\mathcal{M}$. Jeśli przy tym maszyna $\mathcal{M}$ zatrzymuje się[2] dla dowolnego wejścia, to mówimy, że f jest *obliczalna*.

Fakt 1.6 *Każda funkcja (częściowo) rekurencyjna jest też (częściowo) obliczalna.*

Dowód: Indukcja ze względu na definicję funkcji rekurencyjnej (ćwiczenie). □

Wniosek 1.7 *Każdy zbiór rekurencyjny (rekurencyjnie przeliczalny) jest obliczalny (częściowo obliczalny).*

Kodowanie maszyn Turinga

Chcemy udowodnić twierdzenie odwrotne do faktu 1.6 (dla uproszczenia na razie dla funkcji jednoargumentowych). W tym celu najpierw umówimy się, że stany maszyny i symbole alfabetu są ponumerowane, przy czym symbol B ma numer 0. Od tej pory stany i symbole utożsamiamy z ich numerami. Konfiguracje reprezentujemy jako trójki $\langle q, i, w\rangle = \langle q, \langle i, w\rangle\rangle$, używając dwukrotnie zwykłej funkcji pary. Przypomnijmy, że jeśli $c = \langle q, i, w\rangle$ to $q = \ell(c)$, $i = \ell(r(c))$ i $w = r(r(c))$. Równość $c = \langle q, i, w\rangle$ jest więc relacją pierwotnie rekurencyjną, podobnie jak funkcja, która argumentowi $\langle q, i, w\rangle$ przypisuje trójkę $\langle p, i + \varepsilon, w[i \leftarrow b]\rangle$, gdzie $\varepsilon \in \{-1, 0, 1\}$. Dla danej konfiguracji c, kod następnej konfiguracji $N(c)$ wyraża się więc definicją warunkową:

$$N(c) = \begin{cases} \dots & \dots \\ \dots & \dots \\ \langle p, i+\varepsilon, w[i \leftarrow b]\rangle, & \text{jeśli } c = \langle q, i, w\rangle \text{ oraz } \delta(q, @(w, i)) = (p, b, \varepsilon), \\ \dots & \dots \\ \dots & \dots \end{cases}$$

o tylu wierszach ile jest możliwych „klauzul" w definicji funkcji przejścia. Operacja zmiany konfiguracji jest więc pierwotnie rekurencyjna. Możemy ją iterować, aby wyznaczyć kolejne konfiguracje obliczenia dla danego wejścia n:

$$\begin{aligned} C(0, n) &= \langle q_0, 0, kod(1^n)\rangle; \\ C(m+1, n) &= N(C(m, n)). \end{aligned}$$

Teraz zdefiniujemy relację: „$\mathcal{M}$ akceptuje wejście n po m krokach z głowicą w pozycji k":

[2] Można zakładać, że maszyna akceptuje każde słowo postaci $1^{m_1}\sharp 1^{m_2}\sharp \dots \sharp 1^{m_k}$ i odrzuca pozostałe.

$$t_{\mathcal{M}}(n,k,m) \equiv \exists y \leq C(m,n).\, C(m,n) = \langle q_a, k, y\rangle.$$

Możemy już napisać jaka funkcja jest obliczana przez maszynę $\mathcal{M}$.

$$f_{\mathcal{M}}(n) = \ell(\mu y[t_{\mathcal{M}}(n, \ell(y), r(y))]).$$

Powyższą konstrukcję łatwo jest uogólnić na funkcje wieloargumentowe (ćwiczenie). Uwzględniając fakt 1.6, udowodniliśmy więc, że

Twierdzenie 1.8 *Funkcje (częściowo) obliczalne i (częściowo) rekurencyjne to to samo.*

Jeśli dokładniej przyjrzymy się naszemu dowodowi, to zauważymy też następujący ważny fakt.

Twierdzenie 1.9 (o postaci normalnej Kleene'ego) *Każdą funkcję częściowo rekurencyjną φ można przedstawić w postaci*

$$\varphi(\vec{n}) = \ell(\mu y[g(\vec{n}, y) = 0]),$$

gdzie g jest funkcją pierwotnie rekurencyjną.

Fakt 1.10 *Następujące warunki są równoważne dla $A \subseteq \mathbb{N}$:*

1. *A jest rekurencyjnie przeliczalny.*
2. *A jest dziedziną pewnej funkcji częściowo rekurencyjnej.*
3. *Istnieje taki rekurencyjny zbiór $B \subseteq \mathbb{N}^2$, że dla wszystkich $n \in \mathbb{N}$ zachodzi równoważność*
$$n \in A \Leftrightarrow \exists z.\, (n,z) \in B.$$
4. *A jest pusty lub jest zbiorem wartości pewnej funkcji rekurencyjnej.*[3]
5. *A jest zbiorem wartości pewnej funkcji częściowo rekurencyjnej.*

Dowód: Implikacja (1)⇒(2) jest oczywista, a (2)⇒(5) wynika stąd, że $A = \mathrm{Dom}(\psi)$ implikuje $A = \mathrm{Rg}(\varphi)$, gdzie $\varphi(n) = n + Z_1(\psi(n))$. Dla dowodu implikacji (5)⇒(3) załóżmy, że $A = \mathrm{Rg}(\varphi)$. Na mocy twierdzenia Kleene'ego mamy $\varphi(x) = \ell(\mu y[g(x,y) = 0])$ dla pewnej rekurencyjnej funkcji g. Mamy teraz

$$n \in A \;\Leftrightarrow\; \exists x \exists y (g(x,y) = 0 \wedge \forall i < y (g(x,i) \neq 0) \wedge \ell(y) = n),$$

a więc można przyjąć

$$B = \{(n,z) \mid g(\ell(z), r(z)) = 0 \wedge \forall i < r(z)(g(\ell(z), i) \neq 0) \wedge \ell(r(z)) = n\}.$$

Aby pokazać (3)⇒(4), przypuśćmy, że $A \neq \varnothing$ i ustalmy jakieś $a \in A$. Wtedy $A = \mathrm{Rg}(f)$, dla

$$f(n) = \begin{cases} \ell(n), & \text{jeśli } (\ell(n), r(n)) \in B; \\ a, & \text{w przeciwnym przypadku.} \end{cases}$$

Na koniec, (4)⇒(1) wynika z równości $\chi_A(m) = Z_1(\mu y[f(y) = m])$ dla $A = \mathrm{Rg}(f)$. □

[3] To uzasadnia nazwę „rekurencyjnie przeliczalny".

Ćwiczenia

1. Udowodnić, że klasa funkcji pierwotnie rekurencyjnych jest zamknięta ze względu na minimum ograniczone i ograniczony kwantyfikator ogólny.
2. Pokazać, że jeśli f jest k-argumentową funkcją rekurencyjną to jest też rekurencyjną relacją $k+1$-argumentową.
3. Pokazać, że jeśli funkcja częściowa $f : \mathbb{N}^k \multimap \mathbb{N}$ jest rekurencyjnie przeliczalna jako relacja $k+1$-argumentowa to jest k-argumentową funkcją częściowo rekurencyjną.
4. Zdefiniować funkcje:
   - $|m|$ — „długość ciągu o kodzie m (bez końcowych zer)";
   - $m \bullet n$ — „dopisanie n na końcu ciągu o kodzie m";
   - $kod(1^n)$ — kod wejścia.

2 Inne definicje obliczalności

While-programy

While-programy to bardzo uproszczony model języka programowania, w którym programy są konstruowane za pomocą prostych pętli i w którym występuje tylko jeden typ danych (np. liczby naturalne, ale niekoniecznie).

Definicja 2.1 Ustalmy sygnaturę Σ. *While-program* nad Σ to wyrażenie jednej z nastepujących postaci:

- *Instrukcja przypisania:* $x := t$, gdzie x jest zmienną a t jest termem sygnatury Σ;
- *Złożenie:* $P_1; P_2$, gdzie P_1 i P_2 są while-programami;
- *Instrukcja warunkowa:* **if** α **then** P_1 **else** P_2 **fi**, gdzie α jest formułą otwartą a P_1 i P_2 są while-programami;
- *Pętla:* **while** α **do** P **od**, gdzie α jest formułą otwartą a P jest while-programem.

Intuicja związana z interpretacją while-programu w danej strukturze sygnatury Σ powinna być oczywista: wykonanie programu zmienia wartościowanie zmiennych w sposób odpowiedni do zawartych w nim instrukcji.

Definicja 2.2 Dla dowolnej struktury $\mathfrak{A}$ sygnatury Σ i dowolnego while-programu P definiujemy *relację wejścia-wyjścia*, którą oznaczamy przez $P^{\mathfrak{A}}$. Relacja ta zachodzi pomiędzy wartościowaniami w $\mathfrak{A}$ i jest oczywiście określona przez indukcję.[4]

- Relacja $(x := t)^{\mathfrak{A}}$ składa się z par postaci $(\varrho, \varrho[x \mapsto a])$, gdzie $a = [\![t]\!]_\varrho$.
- Relacja $(P_1; P_2)^{\mathfrak{A}}$ to złożenie relacji $P_2^{\mathfrak{A}} \bullet P_1^{\mathfrak{A}}$.

[4] Napis $\varrho[x \mapsto a]$ oznacza wartościowanie różniące się od ρ tylko tym, że $\varrho[x \mapsto a](x) = a$.

- Jeśli $P =$ **if** α **then** P_1 **else** P_2 **fi**, to $(\varrho, \varrho') \in P^{\mathfrak{A}}$ wtedy i tylko wtedy, gdy zachodzi jeden z przypadków
  - $(\varrho, \varrho') \in P_1^{\mathfrak{A}}$ oraz $\mathfrak{A}, \varrho \models \alpha$;
  - $(\varrho, \varrho') \in P_2^{\mathfrak{A}}$ oraz $\mathfrak{A}, \varrho \not\models \alpha$.
- Jeśli $P =$ **while** α **do** P_1 **od**, to $(\varrho, \varrho') \in P^{\mathfrak{A}}$ wtedy i tylko wtedy, gdy istnieje $m \geq 0$ i taki ciąg wartościowań $\varrho = \varrho_0, \varrho_1, \ldots, \varrho_m = \varrho'$, że
  - $(\varrho_i, \varrho_{i+1}) \in P_1^{\mathfrak{A}}$, dla $i = 0, \ldots, m-1$;
  - $\mathfrak{A}, \varrho_i \models \alpha$, dla $i = 0, \ldots, m-1$;
  - $\mathfrak{A}, \varrho_m \not\models \alpha$.

Oczywiście, dla danego ϱ istnieje co najwyżej jedno wartościowanie ϱ' o własności $(\varrho, \varrho') \in P^{\mathfrak{A}}$. Jeśli istnieje, to mówimy, że program P *zatrzymuje się* dla danych ϱ *z wynikiem* ϱ'. While-program, w którym wyróżnimy pewną liczbę zmiennych *wejściowych* oraz jedną zmienną *wyjściową* można więc uważać za definicję funkcji częściowej.

Definicja 2.3 Funkcja częściowa $f : \mathfrak{A}^n \multimap \mathfrak{A}$ jest *obliczalna w* $\mathfrak{A}$, wtedy i tylko wtedy, gdy istnieje while-program P i zmienne $x_1, \ldots, x_n, y$ o takich własnościach:

- Zmienne $x_1, \ldots, x_n$ są różne;
- P zatrzymuje się dla ϱ wtedy i tylko wtedy, gdy $f(\varrho(x_1), \ldots, \varrho(x_n))$ jest określone;
- Jeśli $(\varrho, \varrho') \in P^{\mathfrak{A}}$, to $\varrho'(y) = f(\varrho(x_1), \ldots, \varrho(x_n))$.

Fakt 2.4 *Funkcje (częściowo) obliczalne w strukturze* $\langle \mathbb{N}, 0, succ \rangle$*, to dokładnie funkcje (częściowo) rekurencyjne.*

Dowód: Ćwiczenie. □

Gramatyki typu zero

Przez *gramatykę typu zero* rozumiemy twór postaci $G = \langle \mathcal{A}, \mathcal{N}, P, \xi_0 \rangle$, w którym $\mathcal{A}$ jest *alfabetem terminalnym*, $\mathcal{N}$ jest *alfabetem nieterminalnym*, $\xi_0 \in \mathcal{N}$ to *symbol początkowy*, a *produkcje* (czyli *reguły*) ze zbioru P mają kształt $u \Rightarrow v$, gdzie $u, v \in (\mathcal{A} \cup \mathcal{N})^*$ są zupełnie dowolnymi słowami, byle tylko $u \neq \varepsilon$. Relacja redukcji $\to_G$ jest zdefiniowana podobnie jak dla gramatyk bezkontekstowych, mianowicie $x \to_G y$ (zapisywane też tak: $G \vdash x \to y$) zachodzi wtedy i tylko wtedy, gdy $x = x_1 u x_2$, $y = x_1 v x_2$, oraz $u \Rightarrow v$ jest pewną produkcją. Oczywiście notacja $\twoheadrightarrow_G$ oznacza istnienie (być może pustego) ciągu redukcji, a język generowany przez gramatykę G definiujemy tak:

$$L(G) = \{w \in \mathcal{A}^* \mid \xi_0 \twoheadrightarrow_G w\}.$$

Na przykład język $\{a^n b^n c^n \mid n \in \mathbb{N}\}$ jest generowany przez taką gramatykę typu zero:

$$\xi_0 \Rightarrow \varepsilon, \quad \xi_0 \Rightarrow \eta;$$
$$\eta \Rightarrow a\beta\eta c, \quad \eta \Rightarrow abc;$$
$$\beta a \Rightarrow a\beta, \quad \beta b \Rightarrow bb.$$

Twierdzenie 2.5 *Język L jest rekurencyjnie przeliczalny wtedy i tylko wtedy, gdy $L = L(G)$ dla pewnej gramatyki G.*

Dowód: ($\Leftarrow$) Niech $L = L(G)$. Konstruujemy maszynę niedeterministyczną $\mathcal{M}$ z jedną taśmą pomocniczą.[5] Na początku umieszcza się na tej taśmie symbol początkowy gramatyki, a następnie niedeterministycznie wykonuje redukcje. Tj. w każdej fazie pracy, maszyna wybiera redukcję $x \Rightarrow y$ i zastępuje na taśmie pewne wystąpienie słowa x przez y. (To może wymagać przesunięcia pozostałej zawartości taśmy w lewo lub w prawo.) Potem następuje sprawdzenie, czy zawartość obu taśm, wejściowej i roboczej, jest taka sama. Jeśli tak, to maszyna akceptuje.

($\Rightarrow$) Niech $L = L(\mathcal{M})$ i załóżmy, że $\mathcal{M}$ jest deterministyczną maszyną jednotaśmową. Bez straty ogólności zakładamy, że maszyna nigdy nie pisze symbolu B. Definiujemy gramatykę G, której alfabet nieterminalny składa się ze stanów i symboli maszyny $\mathcal{M}$, oraz dodatkowo z nawiasów kwadratowych (na oznaczenie początku i końca konfiguracji).

Zbiór produkcji gramatyki G dzieli się na dwie części. Pierwszą grupę stanowią produkcje pozwalające wyprowadzić dowolne słowo postaci $w\,[q_0 w]$ (ćwiczenie: jak to zrobić?). Druga grupa produkcji pozwala na symulację obliczenia maszyny w prawej części słowa. Są to takie produkcje:

- $qa \Rightarrow bp$, gdy $\delta(q,a) = (b,p,+1)$;
- $qa \Rightarrow pb$, gdy $\delta(q,a) = (b,p,0)$;
- $q] \Rightarrow bp]$, gdy $\delta(q,\mathsf{B}) = (b,p,+1)$;
- $q] \Rightarrow pb]$, gdy $\delta(q,\mathsf{B}) = (b,p,0)$;
- $cqa \Rightarrow pcb$ i $[qa \Rightarrow [pb$, gdy $\delta(q,a) = (b,p,-1)$;
- $cq] \Rightarrow pcb]$, gdy $\delta(q,\mathsf{B}) = (b,p,-1)$;
- $aq_a \Rightarrow q_a$ i $q_a a \Rightarrow q_a$, gdy $a \neq [,]$;
- $[q_a] \Rightarrow \varepsilon$.

Zauważmy, że wówczas zachodzi równoważność:

$$[q_0 w] \twoheadrightarrow_G [q_a] \quad \text{wtedy i tylko wtedy gdy} \quad w \in L(\mathcal{M}),$$

bo każdy ciąg postaci $[q_0 w] \to_G x_1 \to_G x_2 \to_G \cdots \to_G [q_a]$ musi reprezentować obliczenie maszyny dla słowa w, zakończone „wycieraniem" wszystkich symboli oprócz q_a.

[5] Czytelnik wie skądinąd, że języki akceptowane przez takie maszyny są częściowo obliczalne.

Jeśli odpowiednio dobierzemy produkcje z pierwszej grupy, to każdy ciąg redukcji, w którym występują słowa zawierające stany maszyny $\mathcal{M}$, musi zaczynać się od $\xi_0 \twoheadrightarrow_G w\,[\![q_0 w]\!]$, dla pewnego w.

A zatem wyprowadzenie w gramatyce G, kończące się słowem terminalnym może wyglądać tylko tak:

$$\xi_0 \twoheadrightarrow w\,[\![q_0 w]\!] \twoheadrightarrow w\,[\![q_a]\!] \to w,$$

co oznacza, że $L(G) = L(\mathcal{M})$. □

Rachunek lambda

Lambda-termy definiuje się zwykle tak (przyjmujemy pewien ustalony nieskończony zbiór *zmiennych przedmiotowych*):

- zmienne przedmiotowe są termami;
- jeśli M i N są termami, to (MN) też;
- jeśli M jest termem i x jest zmienną, to $(\lambda x M)$ jest termem.

Konwencje notacyjne:

- opuszczamy zewnętrzne nawiasy;
- aplikacja wiąże w lewo, tj. MNP oznacza $(MN)P$;
- piszemy $\lambda x_1 \ldots x_n.M$ zamiast $\lambda x_1 \ldots \lambda x_n M$.

Operator lambda-abstrakcji, λ, wiąże zmienne, tj. wszystkie wystąpienia x w termie $\lambda x M$ uważa się za *związane*. Zmienne *wolne* definiuje się tak:

- $FV(x) = \{x\}$;
- $FV(MN) = FV(M) \cup FV(N)$;
- $FV(\lambda x M) = FV(M) - \{x\}$.

Dwa termy uważa się za identyczne, jeśli różnią się tylko nazwami zmiennych związanych. Podstawienie $M[N/x]$ definiuje się tak, że N jest wstawiane tylko na wolne wystąpienia x, a zmienne związane ulegają w razie potrzeby przemianowaniu, tak aby nie doprowadzić do konfuzji. Na przykład: $((\lambda y.xy)x)[y/x] = (\lambda z.yz)y$. Szczegółowe definicje pomijamy.

Dla lambda termów określa się relację *beta-redukcji* jako najmniejszą relację $\to_\beta$, taką, że:

- $(\lambda x M)N \to_\beta M[N/x]$;
- jeśli $M \to_\beta M'$ to $MN \to_\beta M'N$, $NM \to_\beta NM'$ oraz $\lambda x M \to_\beta \lambda x M'$.

Teraz pokażemy jak w rachunku lambda można zinterpretować pewne proste konstrukcje. Zaczniemy od wartości logicznych

$$\mathbf{true} = \lambda xy.x \qquad\qquad \mathbf{false} = \lambda xy.y$$

i instrukcji warunkowej

$$\mathbf{if}\ P\ \mathbf{then}\ Q\ \mathbf{else}\ R = PQR.$$

Łatwo sprawdzić, że $\mathbf{if\ true\ then}\ Q\ \mathbf{else}\ R \twoheadrightarrow_\beta Q$ oraz $\mathbf{if\ false\ then}\ Q\ \mathbf{else}\ R \twoheadrightarrow_\beta R$.

Następna konstrukcja to para uporządkowana. Przyjmiemy

$$\begin{array}{rcl} \langle M, N\rangle & = & \lambda x.xMN; \\ \pi_i & = & \lambda x_1x_2.x_i \text{ dla } i = 1, 2. \end{array}$$

Jak należy się spodziewać, mamy $\langle M_1, M_2\rangle\pi_i \twoheadrightarrow_\beta M_i$. Zauważmy jednak, że nie zachodzi równość $\langle M\pi_1, M\pi_2\rangle =_\beta M$, tj. nasza para uporządkowana nie jest *surjektywna*.

Liczby naturalne reprezentujemy w rachunku lambda jako tzw. *liczebniki Churcha*:

$$c_n = \lambda fx.f^n(x),$$

gdzie notacja $f^n(x)$ oznacza oczywiście term $f(f(\cdots(x)\cdots))$, w którym f występuje n razy. Zwykle zamiast c_n będziemy pisać $\mathbf{n}$, co jest mniej precyzyjne, ale wygodne. Więc na przykład:

$$\begin{array}{rcl} \mathbf{0} & = & \lambda fx.x; \\ \mathbf{1} & = & \lambda fx.fx; \\ \mathbf{2} & = & \lambda fx.f(fx), \quad \text{i tak dalej.} \end{array}$$

Powiemy, że funkcja częściowa $f : \mathbb{N}^k \rightharpoonup \mathbb{N}$ jest *definiowalna* w beztypowym rachunku lambda (λ-*definiowalna*) jeżeli istnieje term zamknięty F, spełniający nastepujące warunki dla dowolnych $n_1, \ldots, n_k \in \mathbb{N}$:

- Jeżeli $f(n_1, \ldots, n_k) = m$, to $F\mathbf{n}_1 \ldots \mathbf{n}_k =_\beta \mathbf{m}$;
- Jeżeli $f(n_1, \ldots, n_k)$ jest nieokreślone, to $F\mathbf{n}_1 \ldots \mathbf{n}_k$ nie ma postaci normalnej.

Mówimy oczywiście, że F *definiuje* lub *reprezentuje* funkcję f w rachunku lambda. Kilka przykładów termów definiujących pewne funkcje mamy poniżej.

Przykład 2.6

- Nastepnik: $\mathbf{succ} \equiv \lambda nfx.f(nfx)$;
- Dodawanie: $\mathbf{add} \equiv \lambda mnfx.mf(nfx)$;
- Mnożenie: $\mathbf{mult} \equiv \lambda mnfx.m(nf)x$;
- Potęgowanie: $\mathbf{exp} \equiv \lambda mnfx.mnfx$;
- Test na zero: $\mathbf{zero} \equiv \lambda m.m(\lambda y.\mathbf{false})\mathbf{true}$;

- Funkcja k-argumentowa stale równa zeru: $\mathbf{Z}_k = \lambda m_1 \ldots m_k.\mathbf{0}$;
- Rzut k-argumentowy na i-tą współrzędną: $\mathbf{\Pi}^i_k = \lambda m_1 \ldots m_k.m_i$.

Udowodnimy teraz, że każda funkcja częściowo rekurencyjna jest definiowalna w rachunku lambda. Zaczynamy od najłatwiejszego.

Lemat 2.7 *Funkcje bazowe są lambda-definiowalne.*

Dowód: Przykład 2.6. □

Lemat 2.8 *Jeśli funkcja całkowita f powstaje przez składanie lambda-definiowalnych funkcji całkowitych, to też jest lambda-definiowalna.*

Dowód: Oczywiste. Ale tylko dlatego, że mowa o funkcjach całkowitych. □

Lemat 2.9 *Jeśli funkcja całkowita f powstaje przez rekursję prostą z lambda-definiowalnych funkcji całkowitych, to też jest lambda-definiowalna.*

Dowód: To już nie jest oczywiste. Załóżmy, że f jest zdefiniowana równaniami

$$\begin{array}{lcl} f(0,\vec{n}) & = & g(\vec{n}); \\ f(n+1,\vec{n}) & = & h(f(n,\vec{n}),n,\vec{n}), \end{array}$$

i że funkcje g i h są definiowalne odpowiednio za pomocą termów G i H. Zdefiniujmy pomocnicze termy

$$\begin{array}{lcl} \mathbf{Step} & \equiv & \lambda p.\langle \mathbf{succ}(p\pi_1), H(p\pi_2)(p\pi_1)x_1 \ldots x_m\rangle; \\ \mathbf{Init} & \equiv & \langle \mathbf{0}, Gx_1 \ldots x_m\rangle. \end{array}$$

Funkcja f jest wtedy definiowalna termem

$$F \equiv \lambda x x_1 \ldots x_m.x\,\mathbf{Step}\,\mathbf{Init}\pi_2.$$

Ta definicja wyraża następujący algorytm obliczania wartości funkcji f: generujemy ciąg par

$$(0,a_0),(1,a_1),\ldots,(n,a_n),$$

gdzie $a_0 = g(n_1,\ldots,n_m)$, $a_{i+1} = h(a_i,i,n_1,\ldots,n_m)$ i na koniec $a_n = f(n,n_1,\ldots,n_m)$. □

Wniosek 2.10 *Funkcje pierwotnie rekurencyjne są lambda-definiowalne.*

Ponieważ mamy twierdzenie Kleene'ego, więc pozostaje już tylko pokazać lambda-definiowalność funkcji określonych przez jednorazowe zastosowanie minimum.

Twierdzenie 2.11 *Funkcje lambda-definiowalne to dokładnie funkcje częściowo rekurencyjne.*

Dowód: Implikacja z lewej do prawej wynika stąd, że proces ewaluacji termu do postaci normalnej może być zrealizowany za pomocą maszyny Turinga. Dowodzimy implikacji odwrotnej. Niech $f(\vec{n}) = \ell(\mu y[g(\vec{n}, y) = 0])$, gdzie g jest funkcją pierwotnie rekurencyjną. Na mocy wniosku 2.10, zarówno g jak ℓ są lambda-definiowalne pewnymi termami G i L. Określimy pomocniczy term

$$W \equiv \lambda y.\textbf{if } \textbf{zero}(G\vec{x}y) \textbf{ then } \lambda w.Ly \textbf{ else } \lambda w.w(\textbf{succ}\, y)w.$$

Funkcja f jest definiowana termem

$$F \equiv \lambda\vec{x}.W\mathbf{0}W.$$

Rzeczywiście, przypuśćmy najpierw, że $\mu y[f(\vec{n}, y) = 0] = n$, oraz $\ell(n) = r$. Wtedy mamy taki ciąg redukcji:

$$F\vec{\mathbf{n}} \twoheadrightarrow_\beta \overline{W}\mathbf{0}\overline{W} \twoheadrightarrow_\beta \overline{W}\mathbf{1}\overline{W} \twoheadrightarrow_\beta \cdots \twoheadrightarrow_\beta \overline{W}\mathbf{n}\overline{W} \twoheadrightarrow_\beta L\mathbf{n} \twoheadrightarrow_\beta \mathbf{r}, \tag{1}$$

gdzie $\overline{W}$ oznacza wynik podstawienia $\vec{\mathbf{n}}$ na $\vec{x}$.

Jeśli minimum jest nieokreślone, to znaczy, że wszystkie wartości $g(\vec{n}, m)$ są określone i różne od zera, bo funkcja g jest całkowita. Wtedy mamy nieskończony ciąg redukcji

$$F\vec{\mathbf{n}} \twoheadrightarrow_\beta \overline{W}\mathbf{0}\overline{W} \twoheadrightarrow_\beta \overline{W}\mathbf{1}\overline{W} \twoheadrightarrow_\beta \cdots \twoheadrightarrow_\beta \overline{W}\mathbf{n}\overline{W} \twoheadrightarrow_\beta \cdots$$

Można pokazać, że żadna inna redukcja termu $F\vec{\mathbf{n}}$ nie prowadzi do postaci normalnej. □

Ćwiczenia

1. Napisać gramatykę typu zero generującą język złożony ze wszystkich słów $w\overline{w} \in \{0, 1, 2\}^*$, gdzie słowo $\overline{w}$ powstaje z w przez zamianę każdej jedynki na zero i każdej dwójki na jedynkę.
2. Udowodnić, że dodanie instrukcji **go to** do języka **while**-programów nie rozszerza klasy funkcji obliczalnych.
3. Napisać lambda-term definiujący funkcję $\lfloor \frac{m}{n} \rfloor$ dwóch zmiennych m i n.
4. Niech $\mathbf{Y} = \lambda f(\lambda x f(xx))(\lambda x f(xx))$. Pokazać, że $\mathbf{Y}F =_\beta F(\mathbf{Y}F)$ dla dowolnego termu F.

3 Funkcje pierwotnie i elementarnie rekurencyjne

Dla wszystkich $n \in \mathbb{N}$ definiujemy funkcje $a_n : \mathbb{N} \to \mathbb{N}$:

$$\begin{aligned} a_0(x) &= x + 1; \\ a_{n+1}(x) &= a_n^{x+1}(1), \end{aligned}$$

gdzie a_n^{x+1} oznacza $(x+1)$-krotną iterację funkcji a_{n+1}. A więc $a_1(x) = x + 2$, $a_2(x) = 2x + 3$, co jeszcze wyglada dość niegroźnie. Ale dalej $a_3(x) > 2^x$, a oszacowanie dla $a_4(x)$ wymaga potęgowania iterowanego x razy. Wszystkie funkcje a_n są jednak pierwotnie rekurencyjne, bo są określone przez rekursję prostą.

Lemat 3.1 *Funkcje a_n mają następujące własności (dla wszystkich $x \in \mathbb{N}$):*

1. $x < a_n(x)$;
2. $x < a_n^x(1)$;
3. $a_n(x) < a_n(x+1)$.

Dowód: Dowód jest przez jednoczesną indukcję ze względu na n. Oczywiście (1) i (3) zachodzi dla $n = 0$. Zauważmy też, że (1) implikuje (2). Istotnie, jeśli $x < a_n(x)$ dla dowolnego x, to $1 < a_n(1) < a_n(a_n(1)) < \cdots$ więc $a_n^x(1)$ jest równe co najmniej $x+1$.

Kolej na krok indukcyjny. Zaczynamy od części (3):

$$a_{n+1}(x+1) = a_n^{x+2}(1) = a_n^{x+1}(a_n(1)) > a_n^{x+1}(1) = a_{n+1}(x).$$

Teraz część (1):

$$a_{n+1}(x) = a_n^{x+1}(1) = a_n(a_n^x(1)) > a_n(x) > x. \qquad \square$$

Wniosek 3.2 *Funkcje a_n są rosnące, oraz $a_n(x) < a_m(x)$, gdy $n < m$.*

Dowód: Zauważmy, że $a_{n+1}(x) = a_n(a_n^x(1)) > a_n(x)$. $\square$

Lemat 3.3 *Następujące nierówności zachodzą dla wszystkich $n, x \in \mathbb{N}$:*

1. $a_n^2(x) < a_{n+2}(x)$;
2. $a_n^{x+1}(x) \leq a_{n+3}(x)$.

Dowód: Zaczynamy od nierówności $a_m(x+1) \leq a_m(a_m^x(1)) = a_{m+1}(x)$, która zachodzi dla wszystkich m. Stąd $a_n(a_n(x)) < a_n(a_{n+1}(x)) = a_n^{x+2}(1) = a_{n+1}(x+1) \leq a_{n+2}(x)$. Dalej mamy $a_n^{x+1}(x) < a_n^{x+1}(a_n^x(1)) = a_n^{2x+1}(1) < (a_n^2)^{x+1}(1) < a_{n+2}^{x+1}(1) = a_{n+3}(x)$. $\square$

Twierdzenie 3.4 *Dla dowolnej funkcji pierwotnie rekurencyjnej $f : \mathbb{N}^k \to \mathbb{N}$ istnieje taka liczba n, że $f(\vec{x}) \leq a_n(\max \vec{x})$ dla wszystkich $\vec{x} \in \mathbb{N}^k$.*

Dowód: Dowód jest przez indukcję ze względu na definicję funkcji f. Oczywiście funkcje bazowe są ograniczone przez funkcję następnika a_0. Jeśli funkcja f jest złożeniem postaci $f(\vec{x}) = h(g_1(\vec{x}), \ldots, g_\ell(\vec{x}))$, to niech m będzie dostatecznie duże na to, aby $g_i(\vec{x}) \leq a_m(\max \vec{x})$ oraz $h(\vec{y}) \leq a_m(\max \vec{y})$ dla dowolnych $\vec{x}, \vec{y}$. Teraz $f(\vec{x}) \leq a_m(a_m(\max \vec{x})) < a_{m+2}(\max \vec{x})$.

Jeśli f jest określona równaniami rekursji prostej:

$$\begin{aligned} f(0, \vec{y}) &= g(\vec{y}); \\ f(x+1, \vec{y}) &= h(f(x, \vec{y}), x, \vec{y}), \end{aligned}$$

oraz m jest dostatecznie duże, to przez indukcję ze względu na x udowodnimy

$f(x, \vec{y}) \leq a_m^{x+1}(\max \vec{y}) \leq a_{m+3}(max\{x, \vec{y}\})$. □

Twierdzenie 3.4 mówi, że funkcje a_n *majoryzują* klasę funkcji pierwotnie rekurencyjnych. Jeśli teraz określimy dwuargumentową *funkcję Ackermanna* $A(n, x)$ równaniami[6]

$$\begin{aligned} A(0, x) &= x + 1; \\ A(n + 1, 0) &= A(n, 1); \\ A(n + 1, x + 1) &= A(n, A(n + 1, x)), \end{aligned}$$

to łatwo stwierdzimy, że $A(n, x) = a_n(x)$. A zatem mamy:

Fakt 3.5 *Funkcja Ackermanna nie jest pierwotnie rekurencyjna.*

Dowód: Funkcja $a(x) = A(x, x) = a_x(x)$ nie jest ograniczona przez żadną z funkcji a_n. □

Następujący fakt charakteryzuje funkcje pierwotnie rekurencyjne w nieco przewrotny sposób. Jest jednak użyteczny, bo w myśl twierdzenia 3.4 „czas pierwotnie rekurencyjny" to to samo, co czas ograniczony przez którąś z funkcji a_n.

Twierdzenie 3.6 *Funkcja jest pierwotnie rekurencyjna wtedy i tylko wtedy, gdy jest obliczalna w czasie pierwotnie rekurencyjnym.*

Dowód: Podamy tylko szkic dowodu, zostawiając szczegóły Czytelnikowi. Implikacja z lewej do prawej wymaga konstrukcji maszyny Turinga obliczającej daną funkcję. Stosujemy indukcję ze względu na definicję funkcji, podobnie jak w dowodzie twierdzenia 3.4. Podobne też są oszacowania czasu obliczenia przez odpowiednie a_n.

Jeśli zaś funkcja f, obliczalna w czasie pierwotnie rekurencyjnym, zostanie przedstawiona w postaci normalnej Kleene'ego (twierdzenie 1.9), to użyte tam minimum będzie ograniczone, a zatem sama funkcja musi być pierwotnie rekurencyjna. □

Funkcje elementarnie rekurencyjne

Ważną podklasę funkcji pierwotnie rekurencyjnych stanowią funkcje *elementarnie rekurencyjne*. Istnieje kilka różnych definicji, my wybierzemy najprostszą.

Definicja 3.7 Niech $exp_0(x) = x$ oraz $exp_{m+1}(x) = 2^{exp_m(x)}$. Mówimy, że funkcja $f : \mathbb{N}^k \to \mathbb{N}$ jest *elementarnie rekurencyjna*, gdy jest obliczalna w czasie exp_m dla pewnego m.

Nietrudno pokazać, że funkcje exp_m majoryzują klasę funkcji elementarnie rekurencyjnych, a zatem funkcja a_4 nie jest już elementarna, podobnie jak funkcja $e(n) = exp_n(1)$

$$e(n) = 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}} \Big\} n$$

czyli „słupek dwójek wysokości n".

[6] Uwaga: to nie jest rekursja prosta. Mamy tu zagnieżdżoną rekursję ze względu na dwa parametry.

Ćwiczenia

1. Skąd wiadomo, że funkcja Ackermanna jest rekurencyjna?
2. Czy funkcja Ackermanna, rozumiana jako trójargumentowa relacja, jest pierwotnie rekurencyjna?
3. Niech e_0 będzie funkcją następnika i niech $e_{n+1}(x) = e_n^x(x)$. Udowodnić, że dla każdej funkcji pierwotnie rekurencyjnej f istnieje takie n, że $f(\vec{x}) \leq e_n(\max \vec{x})$, gdy $\max \vec{x} \geq 2$. Wywnioskować, że funkcja $e_\omega(x) = e_x(x)$ nie jest pierwotnie rekurencyjna.
4. W Definicji 2.1 zamieniamy pętlę **while** na pętlę postaci **for** $i = 1$ **to** x **do** P **od**, przy czym zmienna sterująca i nie występuje w P. Zdefiniować semantykę takiego języka w strukturze liczb naturalnych i pokazać, że funkcje obliczalne w tym języku to dokładnie funkcje pierwotnie rekurencyjne.
5. Udowodnić, że funkcja jest elementarnie rekurencyjna wtedy i tylko wtedy, gdy jest obliczalna w czasie elementarnie rekurencyjnym.

4 Numeracja

Funkcje częściowo rekurencyjne i zbiory rekurencyjnie przeliczalne można numerować na różne sposoby. Można na przykład zdefiniować numerację maszyn Turinga, lub przypisywać numery k-argumentowym funkcjom częściowo rekurencyjnym przez indukcję ze względu na ich definicje. Zaczynamy wtedy od przypisania numerów funkcjom bazowym:

- Z_k ma numer $\langle 0, 0\rangle$;
- Π_i^k ma numer $\langle 0, i\rangle$;
- *succ* ma numer $\langle 0, 2\rangle$.

Funkcję k-argumentową o numerze n będziemy oznaczać przez $\varphi_n^{(k)}$, pomijając górny indeks jeśli jest to nieszkodliwe. Jeśli teraz funkcja k-argumentowa ψ jest złożeniem postaci

$$\psi(\vec{x}) = \varphi_n^{(m)}(\varphi_{\ell_1}^{(k)}(\vec{x}), \ldots, \varphi_{\ell_m}^{(k)}(\vec{x})),$$

to za numer funkcji ψ możemy przyjąć liczbę $\langle 1, \langle m, \langle n, \langle \ell_1, \ldots, \langle \ell_{m-1}, \ell_m\rangle\rangle\rangle\rangle\rangle$. Podobnie, funkcja o $k+1$ argumentach i numerze $\langle 2, \langle m, n\rangle\rangle$ to funkcja zdefiniowana przez rekursję prostą z funkcji $\varphi_m^{(k)}$ i $\varphi_n^{(k+2)}$. A funkcji $\mu y[\varphi_n^{(k+1)}(\vec{x}, y) = 0]$ można przypisać numer $\langle 3, n\rangle$. Aby nie zostały żadne wolne numery przyjmujemy, że wszystkie liczby innej postaci niż wymienione wyżej są numerami funkcji Z_k. Niewiele to zmienia, bo i tak

każda funkcja częściowo rekurencyjna ma nieskończenie wiele numerów,

numerujemy bowiem w istocie nie funkcje, ale *algorytmy*.

W istocie nie ma większego znaczenia jaki dokładnie przyjęliśmy sposób numeracji. Można np. zamiast numerów funkcji używać ich definicji *in extenso* i nie wpłynęłoby to na otrzymane rezultaty. Używanie liczb naturalnych często jednak upraszcza sformułowanie tych rezultatów.

Zbiory rekurencyjnie przeliczalne numerujemy tak jak ich częściowe funkcje charakterystyczne: zbiór W_n to dziedzina funkcji φ_n.

Funkcje uniwersalne

Funkcja dwuargumentowa $\Phi(m,n) = \varphi_m^{(1)}(n)$ jest częściowo rekurencyjna. Nazywamy ją *funkcją uniwersalną* dla klasy funkcji częściowo rekurencyjnych. Dla całkowitych funkcji obliczalnych takiej funkcji nie ma (nie da się ich sensownie ponumerować).

Fakt 4.1 *Nie istnieje funkcja uniwersalna dla klasy funkcji rekurencyjnych, tj. nie istnieje dwuargumentowa funkcja rekurencyjna $\Psi : \mathbb{N}^2 \to \mathbb{N}$ o tej własności, że każda funkcja rekurencyjna $f : \mathbb{N} \to \mathbb{N}$ ma dla pewnego n postać*

$$f(x) = \Psi(n, x).$$

Dowód: W przeciwnym razie funkcja $f(x) = \Psi(x,x)+1$ też miałaby numer, tj. mielibyśmy $f(x) = \Psi(n,x)$, skąd w szczególności $\Psi(n,n) + 1 = f(n) = \Psi(n,n)$. □

s-m-n-twierdzenie

Twierdzenie o tej dziwacznej nazwie jest intuicyjnie tak naturalne, że zwykle posługujemy się nim nie zauważając tego. Rozpatrzmy na początek wersję dwuwymiarową.

Fakt 4.2 *Jeśli $\varphi : \mathbb{N}^2 \rightharpoonup \mathbb{N}$ jest częściowo rekurencyjna, to istnieje taka rekurencyjna (całkowita) funkcja s, że*

$$\varphi(x,y) = \varphi_{s(x)}(y)$$

zachodzi dla dowolnych x, y. (Inaczej możemy napisać że $\varphi_{s(x)} = \lambda y.\varphi(x,y)$.)

Dowód: Dla danego $x \in \mathbb{N}$, liczba $s(x)$ jest numerem definicji funkcji

$$\xi = \lambda y.\varphi(x, \Pi_1^1(y)).$$

Taki numer można efektywnie obliczyć z danego x. Przy naszej numeracji[7] mamy

$$s(x) = \langle 1, \langle 2, \langle p, \langle l(x), \langle 0, 1\rangle\rangle\rangle\rangle\rangle,$$

gdzie p jest (ustalonym) numerem funkcji φ, natomiast $l(x)$ jest numerem funkcji stale równej x, tj. funkcji $\eta = \lambda y.succ^x(Z_1(y))$. Aby obliczyć $l(x)$, zauważmy, że:

- $l(0)$ to numer funkcji Z_1, czyli $l(0) = \langle 0, 0\rangle$;
- $l(x+1)$ to numer funkcji $\lambda y.succ(x)$, czyli liczba $\langle 1, \langle 1, \langle\langle 0, 2\rangle, l(x)\rangle\rangle\rangle$.

Widzimy więc, że funkcja l jest definiowalna przez rekursję prostą (i składanie). Szczegóły pozostawiamy czytelnikowi. Uwaga: nasza definicja funkcji l nie jest w istocie zgodna ze schematem rekursji prostej (por. definicję poprzednika w rozdziale 1). □

Podobnie udowodnimy też ogólniejszą wersję powyższego faktu.

[7] Jeśli utożsamiać definicje funkcji z maszynami Turinga, to $s(x)$ jest numerem maszyny, która dla danego wejścia y dopisuje z przodu x i uruchamia maszynę obliczającą φ.

Twierdzenie 4.3 (s-m-n-twierdzenie) *Dla dowolnych $m, n \geq 1$ istnieje funkcja rekurencyjna $s_n^m : \mathbb{N}^{m+1} \to \mathbb{N}$, spełniająca dla dowolnych $i, \vec{x}, \vec{y}$ równość*

$$\varphi_i^{(m+n)}(\vec{x}, \vec{y}) = \varphi_{s_n^m(i,\vec{x})}^{(n)}(\vec{y})$$

Jeśli więc mówimy, że numer funkcji $\lambda y.y^n$, albo numer zbioru $\{k \mid k \text{ podzielne przez } n\}$ jest efektywnie obliczalny z n, to w istocie korzystamy z s-m-n-twierdzenia. W pierwszym przypadku mamy bowiem funkcję rekurencyjną s spełniającą równość $\varphi_{s(n)}(y) = y^n$, a w drugim mamy funkcję s o własności $\varphi_{s(n)}(y) = \chi_W(n, y)$, gdzie $W = \{\langle n, k\rangle \mid k \text{ podzielne przez } n\}$.

Uwaga: Należy pamiętać, że każda funkcja częściowo rekurencyjna ma nieskończenie wiele numerów. Nasze s-m-n-twierdzenie mówi, że istnieje algorytm pozwalający zawsze znaleźć *jakiś* numer poszukiwanej funkcji.

Twierdzenie o rekursji

Jako rozgrzewkę proponujemy czytelnikowi rozwiązanie ćwiczenia 3. W razie kłopotów należy przeczytać dowód poniższego twierdzenia. Bywa ono nazywane „twierdzeniem o punkcie stałym", ale w istocie nie stwierdza istnienia punktu stałego, bo przypisanie $\varphi_n \mapsto \varphi_{f(n)}$ nie jest dobrze określonym przekształceniem na funkcjach. Z warunku $\varphi_n = \varphi_m$ nie musi bowiem wynikać $\varphi_{f(n)} = \varphi_{f(m)}$.

Twierdzenie 4.4 (Twierdzenie o rekursji) *Dla każdej rekurencyjnej funkcji $f : \mathbb{N} \to \mathbb{N}$ istnieje taka liczba n, że $\varphi_n = \varphi_{f(n)}$.*

Dowód: Rozpatrzmy funkcję $\psi(x, y) = \varphi_{f(\varphi_x(x))}(y)$. Na mocy s-m-n-twierdzenia mamy $\psi(x, y) = \varphi_{s(x)}(y)$, dla pewnej rekurencyjnej funkcji s. (Uwaga: $s(x) \neq f(\varphi_x(x))$!) Oczywiście funkcja s też ma numer, powiedzmy, że $s = \varphi_m$. Dla dowolnych x, y zachodzi więc równość $\varphi_{\varphi_m(x)}(y) = \varphi_{f(\varphi_x(x))}(y)$, w szczególności $\varphi_{\varphi_m(m)}(y) = \varphi_{f(\varphi_m(m))}(y)$. A więc można przyjąć $n = s(m) = \varphi_m(m)$ i mamy $\varphi_n = \varphi_{f(n)}$. □

Powyższy dowód opiera się na takim samym pomyśle jak definicja kombinatora punktu stałego $\mathbf{Y} = \lambda f(\lambda x f(xx))(\lambda x f(xx))$ w rachunku lambda (ćwiczenie 2.4). Podobieństwo będzie bardziej widać, jeśli zamiast $\varphi_n(m)$ napiszemy $\underline{n}(m)$ i użyjemy nieformalnej lambda-notacji.

Rozpatrzmy funkcję $\lambda x f(\underline{x}(x))$ i przypuśćmy, że ma ona numer m, tj. $\underline{m} = \lambda x f(\underline{x}(x))$. Podstawiając m w miejsce x dostajemy równość $\underline{m}(m) = f(\underline{m}(m))$. Stąd $n = f(n)$ dla $n = \underline{m}(m)$. Wyrażenie $\underline{m}(m)$ uderzająco przypomina lambda-term $\mathbf{Y}f =_\beta (\lambda x.f(xx))(\lambda x.f(xx))$.

Oczywiście nie każda funkcja ma punkt stały, a błąd polega na tym, że wartość $\underline{m}(m)$ może być nieokreślona. Zamiast równości $\underline{m}(x) = f(\underline{x}(x))$ możemy jednak wziąć słabszy warunek $\underline{\underline{m}(x)} = \underline{f(\underline{x}(x))}$. Dzięki s-m-n-twierdzeniu mamy *całkowitą* funkcję $\underline{m}$, która spełnia ten warunek. Wtedy dla $n = \underline{m}(m)$ otrzymamy $\underline{n} = \underline{f(n)}$.

Wniosek 4.5 *Jeśli $\varphi : \mathbb{N}^2 \to \mathbb{N}$ jest częściowo rekurencyjna, to $\varphi_n = \lambda x.\varphi(n, x)$, dla pewnego n. W szczególności:*

- *Istnieje takie n, że $\varphi_n(x) = x^n$ dla dowolnego x.*

- *Istnieje takie k, że $W_k = \{k\}$.*
- *Istnieje takie m, że $\varphi_m(x) = m$ dla dowolnego x.*

Dowód: Niech s będzie taką funkcją obliczalną, że $\varphi_{s(n)} = \lambda x.\varphi(n, x)$. Należy zastosować twierdzenie 4.4 do funkcji s. □

Ostatnia część wniosku 4.5 to rozwiązanie ćwiczenia 3 w języku programowania, gdzie programami są numery algorytmów. Niezależnie od danych wejściowych, program φ_m generuje w wyniku własny numer. Rozwiązanie w Pascalu to już tylko sprawa implementacji.

Inne zastosowania twierdzenia o rekursji moga polegać na dowodzeniu poprawności niektórych definicji rekurencyjnych. Na przykład częściową obliczalność funkcji Ackermanna[8] można wykazać stosując twierdzenie o rekursji w następujący sposób. Niech $\Phi(m, n, x)$ będzie trójargumentową funkcją uniwersalną (dla funkcji dwuargumentowych). Określimy trójargumentową operację $B(m, n, x)$ za pomocą zwykłej definicji warunkowej:

$$\begin{aligned} B(m, 0, x) &= x + 1; \\ B(m, n + 1, 0) &= \Phi(m, n, 1); \\ B(m, n + 1, x + 1) &= \Phi(m, n, \Phi(m, n + 1, x)). \end{aligned}$$

Niech f będzie taką (całkowitą) funkcją, że $\Phi(f(m), x, y) = B(m, x, y)$. Dwuargumentowa wersja twierdzenia o rekursji zastosowana do funkcji f mówi, że istnieje taka liczba m, że dla dowolnych n, x zachodzi $\Phi(m, n, x) = \Phi(f(m), n, x) = B(m, n, x)$. Oznacza to, że $A(n, x)$ można zdefiniować jako $B(m, n, x)$.

Ćwiczenia

1. Udowodnić, że funkcja s_n^m, o której mowa w s-m-n-twierdzeniu, jest elementarnie rekurencyjna.
2. Wywnioskować z s-m-n-twierdzenia, że
   (a) numer zbioru $W_n \cup W_m$;
   (b) numer funkcji $\lambda x(\varphi_n(x) \cdot \varphi_m(x))$,

   są efektywnie obliczalne z m i n.
3. Napisać program, który drukuje własny tekst. Nie wolno odwoływać się do szczegółów implementacji (nazwy pliku, miejsca w pamięci itp.). Zadanie łatwe w Lispie, trudne w Pascalu.
4. Dlaczego w dowodzie twierdzenia o rekursji funkcje s i $\lambda x.\, \varphi_x(x)$ są na pewno różne?
5. Niech $\psi_n(x) = \Psi(n, x)$, gdzie Ψ jest taką funkcją częściowo rekurencyjną, że
   - istnieje funkcja rekurencyjna f spełniająca warunek $\varphi_n = \psi_{f(n)}$ dla wszystkich n.

   Pokazać, że każda funkcja częściowo rekurencyjna ψ ma nieskończenie wiele wystąpień w ciągu ψ_n. *Wskazówka:* W przeciwnym razie zbiór $\{k \mid \varphi_k = \psi\}$ byłby rekurencyjny.

[8] To, że funkcja Ackermanna jest częściowo rekurencyjna jest dla nas oczywiste, ale tylko dlatego, że znamy maszyny Turinga.

5 Nierozstrzygalność

Zbiór $\mathbb{N}^{\mathbb{N}}$ jest mocy continuum, a zbiór wszystkich funkcji rekurencyjnych jest zaledwie przeliczalny. Z pewnością więc istnieją funkcje całkowite, które nie są rekurencyjne. Możemy jednak podać konkretny przykład:

$$f(x) = \begin{cases} 0, & \text{jeśli } \varphi_x(x) \text{ jest określone;} \\ 1, & \text{w przeciwnym przypadku.} \end{cases}$$

Gdyby f była rekurencyjna, to funkcja

$$\psi(x) = \begin{cases} 0, & \text{jeśli } \varphi_x(x) \text{ jest nieokreślone;} \\ \text{nieokreślone}, & \text{w przeciwnym przypadku.} \end{cases}$$

byłaby częściowo rekurencyjna, zatem $\psi = \varphi_k$ dla pewnego k. Podstawiając k w miejsce x otrzymujemy sprzeczność: $\varphi_k(k)$ jest określone wtedy i tylko wtedy, gdy jest nieokreślone.

Zastosowana powyżej technika przekątniowa pozwala także wywnioskować, że nie każdą funkcję częściowo rekurencyjną można rozszerzyć do funkcji rekurencyjnej. Inaczej mówiąc: to, że pewne algorytmy są częściowe, jest nieuniknione.

Twierdzenie 5.1 *Istnieje taka funkcja częściowo rekurencyjna $\varphi : \mathbb{N} \rightharpoonup \mathbb{N}$, że żadna funkcja całkowita $f : \mathbb{N} \to \mathbb{N}$ zawierająca φ nie jest rekurencyjna.*

Dowód: Wystarczy przyjąć $\varphi(x) = \varphi_x(x) + 1$. Jeśli $\varphi \subseteq f$, gdzie f jest rekurencyjna, to dla pewnego k mamy $f = \varphi_k$. Ponieważ f jest całkowita, więc $\varphi_k(k)$ jest określone. Stąd wynika, że $\varphi(k)$ jest też określone i mamy $\varphi_k(k) = f(k) = \varphi(k) = \varphi_k(k) + 1$. □

Metodą przekątniową łatwo też udowodnimy istnienie zbiorów nierekurencyjnych.

Twierdzenie 5.2 *Zbiory $K = \{n \mid \varphi_n(n) \text{ określone}\}$ i $S = \{\langle m, n\rangle \mid \varphi_n(m) \text{ określone}\}$ nie są rekurencyjne.*[9]

Twierdzenie 5.2 zwykle formułujemy tak: Pytanie czy dana funkcja częściowo rekurencyjna jest określona dla danego argumentu stanowi *problem nierozstrzygalny*. Jest to problem stopu dla maszyn Turinga wyrażony w języku funkcji częściowo rekurencyjnych. Ponieważ zbiory K i S są oczywiście rekurencyjnie przeliczalne, więc ich dopełnienia nie mogą być rekurencyjnie przeliczalne. A więc problem nieokreśloności funkcji nie jest nawet częściowo rozstrzygalny.

Dowody nierozstrzygalności konkretnych problemów zwykle polegają na *redukcji* jednego problemu do innego. Mówimy, że zbiór A *redukuje się* (lub jest *sprowadzalny*) do zbioru B, i piszemy $A \leq B$, gdy istnieje funkcja obliczalna f o takiej własności:[10]

$$x \in A \quad \text{wtedy i tylko wtedy, gdy} \quad f(x) \in B.$$

W praktyce oznacza to tyle, że istnieje algorytm przekształcający każdą możliwą instancję x problemu A w instancję $f(x)$ problemu B, w ten sposób, że pytanie „Czy $x \in A$?" można sprowadzić do pytania „Czy $f(x) \in B$?".

[9] Możemy także napisać $K = \{n \mid n \in W_n\}$ i $S = \{\langle m, n\rangle \mid m \in W_n\}$.

[10] Relację $\leq$ oznacza się też przez $\leq_m$ (od *many-one reducibility*). Jeśli zaś funkcja f jest różnowartościowa, to można napisać $A \leq_1 B$.

Fakt 5.3 *Niech $A \leq B$. Jeśli B jest rozstrzygalny (rekurencyjnie przeliczalny) to A też jest rozstrzygalny (rekurencyjnie przeliczalny). Ponadto, jeśli $-B$ jest rekurencyjnie przeliczalny, to $-A$ też jest rekurencyjnie przeliczalny.*

Dowód: Łatwy. □

Wniosek 5.4 *Zbiór $A = \{n \mid \varphi_n$ jest funkcją całkowitą$\}$ nie jest rekurencyjny. Dokładniej, zbiór $-A$ nie jest rekurencyjnie przeliczalny.*

Dowód: Pokażemy, że $K \leq A$, skąd na mocy faktu 5.3 otrzymamy tezę. Dla $x \in \mathbb{N}$ rozpatrzmy funkcję $\vartheta_x(y) = Z_1(\varphi_x(x))$. W zależności od tego, czy $x \in K$ jest to funkcja stała lub nigdzie nie określona. Funkcja ϑ_x jest częściowo rekurencyjna, a jej numer jest obliczalny z x, tj. $\vartheta_x = \varphi_{s(x)}$ dla pewnej rekurencyjnej funkcji s. Ponadto, funkcja ϑ_x jest całkowita wtedy i tylko wtedy, gdy określone jest $\varphi_x(x)$. A więc $K \leq A$, bo

$$x \in K \quad \text{wtedy i tylko wtedy, gdy} \quad s(x) \in A.$$ □

Twierdzenie Rice'a

Następne twierdzenie dostarcza prostego warunku wystarczającego na nierekurencyjność. Powiemy, że zbiór $A \subseteq \mathbb{N}$ jest *nietrywialny*, gdy $A \neq \varnothing$ i $A \neq \mathbb{N}$. Zbiór A jest zaś *adekwatny*, gdy z warunków $n \in A$ i $W_n = W_m$ wynika $m \in A$. Inaczej mówiąc, zbiór adekwatny określa pewną własność zbiorów rekurencyjnie przeliczalnych niezależną od wyboru indeksu. Poniższe twierdzenie możemy więc odczytać tak: Każda nietrywialna własność zbiorów rekurencyjnie przeliczalnych jest nierozstrzygalna.

Twierdzenie 5.5 (Rice'a) *Żaden zbiór nietrywialny i adekwatny nie jest rekurencyjny.*

Dowód: Niech $X \subseteq \mathbb{N}$ będzie nietrywialny i adekwatny. Bez straty ogólności możemy przyjąć, że zbiór pusty nie ma własności X, tj. że $m \notin X$, gdy $W_m = \varnothing$ (w przeciwnym razie zamiast X rozważamy zbiór $-X$). Załóżmy jeszcze, że $k \in X$.

Pokażemy, że $K \leq X$. W tym celu, dla dowolnego n rozpatrzmy maszynę Turinga M^n, która dla dowolnego wejścia x:

- najpierw oblicza $\varphi_n(n)$;
- a potem (jeśli $\varphi_n(n)$ było określone) sprawdza, czy $x \in W_k$.

Maszyna M^n akceptuje pewien zbiór $W_{f(n)}$ i właśnie stwierdziliśmy, że

$$W_{f(n)} = \begin{cases} W_k, & \text{jeśli } n \in K; \\ \varnothing, & \text{w przeciwnym przypadku.} \end{cases}$$

Inaczej mówiąc $f(n) \in X$ wtedy i tylko wtedy, gdy $n \in K$, co dowodzi, że $K \leq X$. □

Wniosek 5.6 *Następujące zbiory nie są rekurencyjne:*

1. $\{n \mid W_n$ *jest skończony*$\}$;
2. $\{n \mid W_n$ *jest ko-skończony*$\}$;
3. $\{n \mid W_n = \varnothing\}$;
4. $\{n \mid W_n$ *jest rekurencyjny*$\}$;
5. $\{n \mid 0 \in W_n\}$.

Przykłady takie, jak we wniosku 5.6 można łatwo mnożyć. Twierdzenie Rice'a nie jest jednak aż tak negatywne, jak może się wydawać. Własności algorytmów (maszyn Turinga) nie zawsze są adekwatne, tj. nie są własnościami rozpoznawanych przez nie zbiorów. Twierdzenie Rice'a nie stosuje się więc np. do problemu: *Czy dana niedeterministyczna maszyna Turinga ma dla każdego wejścia chociaż jedno obliczenie nieskończone?*.

Ćwiczenia

1. Skąd wiadomo, że funkcja f w dowodzie twierdzenia Rice'a jest rekurencyjna?
   Wskazówka: Z s-m-n-twierdzenia.
2. Sformułować i udowodnić twierdzenie Rice'a dla relacji wieloargumentowych i dla funkcji.
3. Udowodnić, że następujące zbiory są nierekurencyjne:
   - $\{n \mid \varphi_n(0)$ jest określone$\}$;
   - $\{n \mid \varphi_n$ jest monotoniczna$\}$;
   - $\{n \mid \mathrm{Dom}(\varphi_n) = \varnothing\}$;
   - $\{n \mid \varphi_n$ jest różnowartościowa$\}$.
4. Udowodnić, że $S \leq_m K$. *Wskazówka:* Dla danych n, k niech $A_{n,k} = \mathbb{N}$, gdy $n \in W_k$ oraz niech $A_{n,k} = \varnothing$, w przeciwnym przypadku. Jeśli teraz $A_{n,k} = W_{s(n,k)}$ to $\langle n, k\rangle \in S$ zachodzi wtedy i tylko wtedy, gdy $s(n, k) \in K$.

6 Stopnie nierozstrzygalności

Typową metodą dowodu nierozstrzygalności danego zbioru A jest redukcja postaci $S \leq_m A$ lub $S \leq_m -A$. Zatem znane problemy nierozstrzygalne są w istocie co najmniej tak trudne jak problem stopu. Pokażemy teraz, że istnieją jednak zbiory nierekurencyjne, które nie mają tej własności, tj. nie są *zupełne* w klasie zbiorów rekurencyjnie przeliczalnych ze względu na m-sprowadzalność.

Mówimy, że zbiór $A \subseteq \mathbb{N}$ jest *produktywny*, gdy istnieje taka częściowo rekurencyjna funkcja ψ, że $\psi(i) \in A - W_i$ (w szczególności $\psi(i)$ jest określone) dla dowolnego $W_i \subseteq A$.

Przykładem zbioru produktywnego jest $-K = \{n \mid n \notin W_n\}$ (przy funkcji identycznościowej).

Lemat 6.1

1. *Każdy zbiór produktywny zawiera nieskończony podzbiór rekurencyjnie przeliczalny.*
2. *Jeśli $-K \leq_m A$ to A jest produktywny.*

Dowód: (1) Niech A będzie produktywny i niech ψ będzie odpowiednią funkcją. Ponieważ zbiór pusty jest oczywiście zawarty w A, więc $\psi(i) \in A$, gdy i jest numerem zbioru pustego. Zbiór $\{\psi(i)\}$ też ma jakiś numer j, więc $\psi(j) \in A$, przy czym $\psi(i) \neq \psi(j)$. I tak dalej.[11]

(2) Niech f będzie taką funkcją, że $n \in -K$ wtedy i tylko wtedy, gdy $f(n) \in A$, i niech $W_i \subseteq A$. Zbiór $f^{-1}(W_i) \subseteq -K$ jest rekurencyjnie przeliczalny, co więcej (na mocy s-m-n-twierdzenia) jego numer $g(i)$ jest obliczalny i mamy $g(i) \in -K - W_{g(i)}$. Zatem $f(g(i)) \in A - W_i$. □

Twierdzenie 6.2 *Istnieje rekurencyjnie przeliczalny ale nierekurencyjny zbiór, do którego problem stopu nie jest m-sprowadzalny.*

Dowód: Problem stopu $S = \{\langle m, n\rangle \mid m \in W_n\}$, jako zbiór rekurencyjnie przeliczalny, jest obrazem pewnej funkcji rekurencyjnej f. Rozpatrzmy funkcję częściową

$$j(k) = \mu x\,[\, r(f(x)) = k \wedge \ell(f(x)) > 2k\,]$$

i rekurencyjnie przeliczalny zbiór $X = \mathrm{Rg}(\ell \circ f \circ j)$. Zauważmy, że $n \in X$ zachodzi wtedy i tylko wtedy, gdy dla pewnego k para $\langle n, k\rangle$ jest, w kolejności wyznaczonej przez f, pierwszym elementem zbioru S spełniającym warunek $n > 2k$. Ponieważ wśród liczb od 0 do $2k$ występuje co najwyżej k elementów zbioru X, więc $-X$ jest zbiorem nieskończonym.

Zauważmy dalej, że w każdym zbiorze nieskończonym W_k są liczby większe od $2k$, zatem $j(k)$ jest określone i mamy $X \cap W_k \neq \varnothing$. A więc $-X$ nie ma nieskończonego podzbioru rekurencyjnie przeliczalnego. Wyciągamy stąd dwa wnioski. Po pierwsze, sam zbiór $-X$, jako nieskończony, nie jest rekurencyjnie przeliczalny, a więc X nie jest rekurencyjny. Po drugie, z lematu 6.1 wynika, że $-K \not\leq_m -X$, czyli $K \not\leq_m X$. □

Jeśli $A \leq_m B$ oraz $B \leq_m A$, to mówimy, że zbiory A i B mają ten sam m-*stopień* i piszemy $A \equiv_m B$. Z twierdzenia 6.2 wynika, że struktura m-stopni jest nietrywialna: istnieją co najmniej trzy różne m-stopnie rekurencyjnie przeliczalne. Istotnie, każdy zbiór rekurencyjnie przeliczalny jest m-sprowadzalny do problemu stopu,[12] mamy bowiem

$$x \in W_n \qquad \text{wtedy i tylko wtedy, gdy} \qquad \varphi_n(x) \text{ jest określone.}$$

Z drugiej strony, każdy zbiór rekurencyjny jest m-sprowadzalny do dowolnego zbioru nietrywialnego. Mamy więc najmniejszy m-stopień zbiorów rekurencyjnych, największy m-stopień problemu stopu, wiemy też, że istnieją stopnie pośrednie.

Oprócz m-sprowadzalności rozważa się też inne pojęcia redukcji, w szczególności redukcje *w sensie Turinga*: piszemy $A \leq_T B$ gdy istnieje maszyna Turinga rozpoznająca zbiór A

[11] Ściśle rzecz biorąc, należy zdefiniować przez indukcję dwie funkcje rekurencyjne f i g, w ten sposób, że $W_{g(i)} = \{f(0), \ldots, f(i-1)\}$ dla dowolnego i, oraz $\psi(g(i)) = f(i)$.

[12] A więc także do K, patrz ćwiczenie 4 do rozdziału 5.

z pomocą „wyroczni" odpowiadającej na pytania postaci „czy dane słowo należy do B?". Mówimy wtedy też, że A jest *T-sprowadzalny* do B. Różnica między $\leq_T$ i $\leq_m$ polega m.in. na tym, że liczba możliwych „zapytań o B" jest dowolna i nie jest a priori ograniczona. Zatem $A \leq_m B$ implikuje $A \leq_T B$, ale niekoniecznie na odwrót. Zauważmy, np. że $-K \leq_T K$.

Analogicznie do m-stopni definiujemy T-*stopnie*, zwane czasem po prostu *stopniami nierozstrzygalności*, jako klasy abstrakcji relacji

$$A \equiv_T B \qquad \text{wtedy i tylko wtedy, gdy} \qquad A \leq_T B \text{ oraz } B \leq_T A.$$

Każdy T-stopień jest sumą pewnych m-stopni, a więc problemy, które nie są m-równoważne mogą jednak być T-równoważne. Z twierdzenia 6.2 nie wynika więc, że istnieją nietrywialne rekurencyjnie przeliczalne T-stopnie. To pytanie nazywano kiedyś *problemem Posta*.

6.1 Rozwiązanie problemu Posta

Skonstruujemy teraz takie dwa zbiory rekurencyjnie przeliczalne A i B, że $A \not\leq_T B$ oraz $B \not\leq_T A$. Aby tak było, musimy pokazać, że żadna maszyna z wyrocznią A nie rozstrzyga zbioru B i na odwrót.

Oznaczmy przez M_e^X deterministyczną maszynę Turinga o numerze e, która używa wyroczni X. Naszym celem jest konstrukcja zbiorów A i B jako sum ciągów przybliżeń A_n i B_n. Jednocześnie definiujemy ciąg „świadków" $F(j)$, spełniających następujące równoważności:[13]

$$\begin{aligned} F(2e) \in B \quad &\text{wtedy i tylko wtedy, gdy} \quad M_e^A \text{ odrzuca } F(2e)\,; \\ F(2e+1) \in A \quad &\text{wtedy i tylko wtedy, gdy} \quad M_e^B \text{ odrzuca } F(2e+1)\,. \end{aligned}$$

Oczywistym problemem jest to, że dodanie nowego elementu do zbioru A lub B może popsuć dotychczasowe własności „świadków". Zatem ciąg F będzie „granicą" ciągów F_n uzyskiwanych w poszczególnych fazach konstrukcji (w razie potrzeby znajdujemy nowych świadków).

Konstrukcja A_n, B_n i F_n przebiega przez indukcję ze względu na n i zaczyna się od

$$A_0 = \varnothing = B_0, \qquad F_0(x) = 2^x.$$

Krok indukcyjny dla $n > 0$ zależy od pierwszej współrzędnej liczby n interpretowanej jako para $n = \langle l, r\rangle$.

Przypadek parzysty. Załóżmy, że $n = \langle 2e, \text{coś}\rangle$ oraz

1. $F_n(2e) \notin B_n$;
2. maszyna $M_e^{A_n}$ odrzuca wejście $F_n(2e)$.

Obliczenie maszyny $M_e^{A_n}$ odrzucające $F_n(2e)$ jest skończone, zatem zadaje wyroczni A_n tylko skończenie wiele pytań. Niech k będzie takie, że wszystkie odwołania do A_n dotyczą liczb mniejszych od k. Definiujemy teraz $A_{n+1} = A_n$, $B_{n+1} = B_n \cup \{F_n(2e)\}$, oraz

$$F_{n+1}(x) = \begin{cases} 3^k \cdot F_n(x), & \text{jeśli } x > 2e \text{ i } x \text{ jest nieparzyste;} \\ F_n(x), & \text{w przeciwnym przypadku.} \end{cases}$$

Zauważmy, że liczba $3^k \cdot F_n(x)$ jest większa od k. Ewentualne późniejsze dodanie jej do zbioru A_n nie zmieni więc odpowiedzi wyroczni na pytania o liczby mniejsze niż k.

[13] Przez „odrzucanie" rozumiemy w tym dowodzie zatrzymanie maszyny w stanie odrzucającym.

Przypadek nieparzysty. Niech teraz $n = \langle 2e+1, \text{coś} \rangle$. Jeżeli zachodzą warunki

1. $F_n(2e+1) \notin A_n$;
2. maszyna $M_d^{B_e}$ odrzuca wejście $F_n(2e+1)$,

to $B_{n+1} = B_n$, $A_{n+1} = A_n \cup \{F_n(2e+1)\}$, oraz

$$F_{n+1}(x) = \begin{cases} 3^k \cdot F_n(x), & \text{jeśli } x > 2e+1 \text{ i } x \text{ jest parzyste;} \\ F_n(x), & \text{w przeciwnym przypadku,} \end{cases}$$

gdzie k jest takie, że obliczenie $M_e^{B_n}$ na wejściu $F_n(2e+1)$ odwołuje się do wyroczni B_n tylko dla argumentów mniejszych od k.

Przypadek trywialny. Jeśli nie zachodzi żaden z powyższych przypadków (warunki 1–2 nie są spełnione) to wszystko zostaje bez zmian: $A_{n+1} = A_n$, $B_{n+1} = B_n$ i $F_{n+1} = F_n$.

Przypadek parzysty i nieparzysty są do siebie całkowicie dualne. W dalszym ciągu zwykle rozważamy tylko jeden z nich. Ponieważ ciągi zbiorów A_n i B_n są wstępujące, więc możemy od razu zdefiniować

$$A = \bigcup_{n\in\mathbb{N}} A_n \qquad \text{oraz} \qquad B = \bigcup_{n\in\mathbb{N}} B_n.$$

Procedura opisana powyżej jest efektywna, więc zbiory A i B są rekurencyjnie przeliczalne.

Lemat 6.3 *Dla dowolnego x istnieje takie n_x, że $F_n(x) = F_{n_x}(x)$ dla wszystkich $n \geq n_x$.*

Dowód: Indukcja ze względu na x. Przypuśćmy, że x jest parzyste, oraz $F_{n+1}(x) \neq F_n(x)$. Wtedy $\ell(n)$ jest nieparzyste i mniejsze od x, oraz $F_n(\ell(n)) \in A_{n+1} - A_n$. Jeśli ponadto $F_{m+1}(x) \neq F_m(x)$ dla pewnego $m \neq n$ to mamy też $F_m(\ell(m)) \in A_{m+1} - A_m$. Oznacza to, że $F_n(\ell(n)) \neq F_m(\ell(m))$. W ten sposób każdemu n, takiemu że $F_{n+1}(x) \neq F_n(x)$, przypisujemy inny element zbioru $\{F_m(y) \mid m \in \mathbb{N} \wedge y < x\}$. Z założenia indukcyjnego wynika, że ten zbiór jest skończony, a więc x wartość $F_n(x)$ może się zmieniać tylko skończenie wiele razy. □

Z powyższego lematu wynika, że dla dowolnego x wartość $F_n(x)$ „ustala się" od pewnego miejsca, możemy więc zdefiniować:

$$F(x) = F_n(x) \text{ dla dostatecznie dużych } n.$$

Lemat 6.4 *Jeżeli $F_n(x) = F_m(y)$ to $x = y$. W szczególności F jest różnowartościowa.*

Dowód: Mamy $F_n(x) = 2^x \cdot 3^{\text{coś}}$, a to jednoznacznie określa x. □

Lemat 6.5 *Jeśli M_e^A odrzuca wejście $F(2e)$, to $F(2e) \in B$.*

Dowód: Obliczenie maszyny M_e^A dla wejścia $F(2e)$ używa wyroczni A tylko dla argumentów mniejszych od pewnego k. Weźmy takie $n > k$, że $F(2e) = F_n(2e)$ oraz

$$A \cap \{0, 1, \ldots, k\} = A_n \cap \{0, 1, \ldots, k\}.$$

Wtedy z punktu widzenia naszego obliczenia wyrocznia A_n jest równie dobra jak wyrocznia A, a więc maszyna $M_e^{A_n}$ też odrzuca wejście $F(2e)$. Jeśli $F(2e) = F_n(2e) \in B_n$ to już jest dobrze, bo $B_n \subseteq B$. W przeciwnym razie stosuje się przypadek parzysty i mamy $F(2e) \in B_{n+1}$. □

Lemat 6.6 *I na odwrót: jeśli $F(2e) \in B$ to maszyna M_e^A odrzuca $F(2e)$.*

Dowód: Załóżmy, że $F(2e) \in B$ i niech n będzie takie, że $F(2e) \in B_{n+1} - B_n$. Dla n zachodzi wtedy przypadek parzysty, a z lematu 6.4 wynika, że $\ell(n) = 2e$. Maszyna $M_e^{A_n}$ odrzuca więc $F(2e)$ używając wyroczni A_n dla argumentów mniejszych od pewnego k. Aby stwierdzić, że maszyna M_e^A też odrzuca $F(2e)$, wystarczy sprawdzić, że

$$A \cap \{0, 1, \ldots, k\} = A_n \cap \{0, 1, \ldots, k\}.$$

Przypuśćmy, że $z \in A - A_n$, czyli istnieje takie $m > n$, że $z \in A_{m+1} - A_m$. Wtedy $\ell(m) = 2d+1$ dla pewnego d i dla m zachodzi przypadek nieparzysty. Ponieważ wartość $F(2e) = F_n(2e)$ „już się ustaliła", więc $F_m(2e) = F_n(2e)$. To znaczy, że $2e < 2d + 1$, bo inaczej mielibyśmy $F_{m+1}(2e) > F_m(2e)$.

A zatem $z = F_m(2d + 1) \geq F_{n+1}(2d + 1) = 3^k \cdot F_n(2d + 1) > k$, zbiory A i A_n mogą się więc różnić tylko powyżej k. □

Twierdzenie 6.7 (A.A. Muchnik, R.M. Friedberg) *Istnieją nieporównywalne rekurencyjnie przeliczalne T-stopnie.*

Dowód: Jeśli $B \leq_T A$, to pewna maszyna M_e^A rozstrzyga zbiór B. Z lematów 6.5 i 6.6 otrzymujemy sprzeczność: maszyna M_e^A odrzuca $F(2e)$ wtedy i tylko wtedy, gdy akceptuje $F(2e)$. Analogicznie dowodzimy, że $A \not\leq_T B$. □

Z twierdzenia 6.7 wynika, że istnieją nierozstrzygalne problemy decyzyjne, których nierozstrzygalności nie można jednak udowodnić metodą redukcji z problemu stopu. Autorowi niniejszego nie jest jednak znany żaden nierozstrzygalny problem „matematyczny" (np. kombinatoryczny), który nie jest zupełny w sensie Turinga.

Ćwiczenia

1. Uogólnić lemat 6.1(2): Jeśli B jest produktywny i $B \leq_m A$, to A jest produktywny.
2. Funkcje i zbiory *rekurencyjne względem* (całkowitej) funkcji g definiujemy tak, jak funkcje rekurencyjne, ale dodając g do funkcji bazowych. Pokazać, że $A \leq_T B$ wtedy i tylko wtedy, gdy zbiór A jest rekurencyjny względem funkcji charakterystycznej c_B zbioru B.
3. *Warunkiem tablicowym* nazwiemy skończony zbiór $Z \subseteq \mathbb{N}$ i kombinację boolowską pytań postaci „$z \in B$?", gdzie $z \in Z$. Warunek jest *spełniony* dla B, gdy taka kombinaacja ma wartość logiczną jeden. Zbiór A jest *tablicowo sprowadzalny* do zbioru B, co zapisujemy $A \leq_{\mathrm{tt}} B$, gdy istnieje rekurencyjna funkcja f, która każdemu $x \in \mathbb{N}$ przypisuje taki warunek $f(x)$, że dla dowolnego x,

   $x \in A$ wtedy i tylko wtedy, gdy warunek $f(x)$ jest spełniony dla B.

   Niech teraz $K^* = \{n \in \mathbb{N} \mid n \in K \text{ oraz warunek o numerze } \varphi_n(n) \text{ nie jest spełniony dla } K\}$. Udowodnić, że $K^* \leq_T K$ ale $K^* \not\leq_{\mathrm{tt}} K$. (A zatem tt-stopnie są „drobniejsze" niż T-stopnie.)

7 Hierarchia arytmetyczna

Problem stopu jest wprawdzie nierozstrzygalny, ale jest rekurencyjnie przeliczalny, tj. rozstrzygalny częściowo. Może jednak być gorzej.

Fakt 7.1 *Zbiór* $A = \{n \mid \varphi_n \text{ jest funkcją całkowitą}\}$ *i jego dopełnienie nie są rekurencyjnie przeliczalne.*

Dowód: Mamy już wniosek 5.4, więc wystarczy pokazać, że A nie jest rekurencyjnie przeliczalny. W przeciwnym razie A byłby obrazem pewnej rekurencyjnej funkcji g. Wtedy funkcja $H(n,x) = \Phi(g(n),x) = \varphi_{g(n)}(x)$ byłaby uniwersalna w klasie funkcji rekurencyjnych, co jest niemożliwe (fakt 4.1). □

Własności, które nie są rekurencyjne, można klasyfikować ze względu na stopień skomplikowania formuł wyrażających ich definicje w języku formalnej arytmetyki. Przez *arytmetykę* rozumiemy tu teorię w języku pierwszego rzędu zawierającym dwuargumentowe symbole funkcyjne $+$ i $\cdot$, jednoargumentowy symbol $\boldsymbol{s}$ dla następnika i stałą 0. Jedynym symbolem relacyjnym jest znak równości. Dla dowolnej liczby $n \in \mathbb{N}$, skrót $\underline{n}$ oznacza term $\boldsymbol{s}(\boldsymbol{s}(\cdots \boldsymbol{s}(0)\cdots))$, gdzie symbol $\boldsymbol{s}$ występuje n razy. Strukturę $\mathcal{N} = \langle \mathbb{N}, +, \cdot, \boldsymbol{s}, 0\rangle$, w której symbole arytmetyki są interpretowane „jak zwykle", nazywamy *standardowym modelem arytmetyki*.

Mówimy, że k-argumentowa relacja nad $\mathbb{N}$ jest *arytmetyczna*, gdy istnieje formuła $\varphi(\vec{x})$, która ma (co najwyżej) k zmiennych wolnych $\vec{x}$, i spełnia dla dowolnego $\vec{n} \in \mathbb{N}^k$ warunek

$$\vec{n} \in r \quad \text{wtedy i tylko wtedy, gdy} \quad \mathcal{N} \models \varphi(\underline{\vec{n}}).$$

Funkcja jest *arytmetyczna*, gdy jest arytmetyczna jako relacja. Inaczej można powiedzieć, że funkcje (relacje) arytmetyczne, to te funkcje (relacje), które można zdefiniować za pomocą formuł arytmetyki pierwszego rzędu.

Nastepujące twierdzenie jest słabą wersją tzw. twierdzenia Gödla o reprezentacji.

Twierdzenie 7.2 *Każda funkcja rekurencyjna jest arytmetyczna.*

Dowód: Nietrudno sprawdzić, że funkcje bazowe (stała zero, rzuty, następnik) są arytmetyczne, oraz że funkcja otrzymana przez złożenie funkcyj arytmetycznych musi być arytmetyczna. Dla funkcji f zdefiniowanej przez minimum efektywne

$$f(\vec{x}) = \mu y\,(g(\vec{x},y) = 0),$$

gdzie g jest definiowalna formułą $\psi(\vec{x},y,z)$, możemy napisać formułę

$$\varphi(\vec{x},y) \equiv \psi(\vec{x},y,0) \wedge \forall z(z < y \to \neg\psi(\vec{x},y,0)),$$

w której $z < y$ jest skrótem wyrażenia $\exists u(z + u = y \wedge u \neq 0)$. Pozostaje przypadek funkcji określonej przez rekursję prostą:

$$\begin{aligned} f(0,\vec{x}) &= g(\vec{x}); \\ f(s(m),\vec{x}) &= h(\vec{x},m,f(m,\vec{x})). \end{aligned}$$

Tutaj musimy się odwołać pewnego tricku z teorii liczb, a mianowicie do *funkcji beta* Gödla:

$$\beta(x, y, i) = x \bmod (y(i+1)+1).$$

Najważniejsza własność funkcji beta jest znana jako „chińskie twierdzenie o resztach".

Dla dowolnego skończonego ciągu $k_0, k_1, \ldots, k_n$ istnieją takie liczby a, b, że $\beta(a, b, i) = k_i$ dla $i = 0, \ldots, n$.

Funkcja beta, jak łatwo widzieć, jest arytmetyczna. Niech $B(x, y, i, z)$ będzie odpowiednią formułą i niech f będzie funkcją określoną przez rekursję prostą, jak powyżej. Załóżmy, że $\varphi(\vec{x}, y)$ i $\psi(\vec{x}, y, z, u)$ są formułami definiującymi odpowiednio g i h. Wtedy formuła definiująca funkcję f jest taka:

$$\exists a, b\, [\exists w(B(a, b, 0, w) \wedge \varphi(\vec{x}, w)) \wedge B(a, b, y, z) \\ \wedge \forall i(i < y \to \exists u, v(B(a, b, i, u) \wedge B(a, b, \boldsymbol{s}(i), v) \wedge \psi(\vec{x}, i, u, v)))].$$

Powyższa formuła wyraża następującą równoważność: $f(\vec{x}, y) = z$ wtedy i tylko wtedy, gdy istnieją takie liczby $k_0, k_1, \ldots, k_y$, że $k_0 = g(\vec{x})$ i $k_y = z$, a dla dowolnego $i = 0, \ldots, y-1$ zachodzi $k_{i+1} = h(\vec{x}, i, k_i)$. □

Z powyższego wynika, że każdy zbiór rekurencyjny jest arytmetyczny. Na mocy faktu 1.10(3), wystarczy jeden kwantyfikator egzystencjalny, aby przejść od zbiorów rekurencyjnych do rekurencyjnie przeliczalnych. A zatem mamy:

Wniosek 7.3 *Wszystkie funkcje częściowo rekurencyjne też są arytmetyczne.*

Zdefiniowanie innych zbiorów arytmetycznych może wymagać większej liczby kwantyfikatorów. Na przykład przynależność liczby n do zbioru A z faktu 7.1 można wyrazić tak:

Dla dowolnego wejścia x istnieje takie y, że obliczenie $\varphi_n(x)$ kończy się po y krokach.

Jak wiadomo, każdą formułę pierwszego rzędu można równoważnie przedstawić w *preneksowej postaci normalnej* $Q_1 x_1 Q_2 x_2 \ldots Q_n x_n\, \psi$, gdzie każde Q_i to $\forall$ albo $\exists$, a formuła ψ jest otwarta (nie zawiera kwantyfikatorów). Nam wystarcza, aby definiowała relację obliczalną. Wtedy możliwe jest dalsze uproszczenie, mamy bowiem takie równoważności:

$$\begin{aligned} \mathcal{N} &\models \forall x_1 \forall x_2\, \varphi(x_1, x_2) \leftrightarrow \forall x \varphi(\ell(x), r(x)); \\ \mathcal{N} &\models \exists x_1 \exists x_2\, \varphi(x_1, x_2) \leftrightarrow \exists x \varphi(\ell(x), r(x)), \end{aligned}$$

a zatem istotne są tylko te prefiksy kwantyfikatorowe, w których kwantyfikatory ogólne występują na przemian ze szczegółowymi. Prowadzi to do następującej klasyfikacji zbiorów, którą nazywamy *hierarchią arytmetyczną* lub *hierarchią Kleene'ego-Mostowskiego*.

Definicja 7.4 Niech $A \subseteq \mathbb{N}$. Zbiór A należy do klasy Σ_n (ozn. też Σ^0_n), wtedy i tylko wtedy, gdy dla pewnego rekurencyjnego zbioru $B \subseteq \mathbb{N}^{n+1}$ i wszystkich $y \in \mathbb{N}$ zachodzi

$$y \in \mathcal{A} \quad \text{wtedy i tylko wtedy, gdy} \quad \exists x_1 \forall x_2 \exists x_3 \ldots Q x_n.\, \langle x_1, x_2, \ldots, x_n, y\rangle \in B,$$

gdzie Q powyżej to $\forall$ lub $\exists$, zależnie od parzystości n. Dualnie, A jest zbiorem klasy Π_n (ozn. też Π^0_n), gdy dla pewnego rekurencyjnego $B \subseteq \mathbb{N}^{n+1}$ (i odpowiedniego Q) mamy

$$y \in \mathcal{A} \quad \text{wtedy i tylko wtedy, gdy} \quad \forall x_1 \exists x_2 \forall x_3 \ldots Q x_n.\, \langle x_1, x_2, \ldots, x_n, y\rangle \in B.$$

Dodatkowo definiujemy $\Delta_n = \Sigma_n \cap \Pi_n$ (piszemy też Δ^0_n).

Nietrudno teraz zauważyć, że

- Każdy zbiór arytmetyczny należy do jednej z klas Σ_n i Π_n.
- Zbiory rekurencyjne tworzą klasę $\Delta_0 = \Sigma_0 = \Pi_0$.
- Klasa Σ_1 to klasa zbiorów rekurencyjnie przeliczalnych.
- Do klasy Π_1 należą dopełnienia zbiorów rekurencyjnie przeliczalnych.
- A zatem $\Delta_1 = \Delta_0$, na mocy faktu 1.4.
- Zbiór A z faktu 7.1 jest elementem klasy Π_2, a jego dopełnienie należy do Σ_2.

Zbiory arytmetyczne można numerować w podobny sposób jak numeruje się zbiory rekurencyjnie przeliczalne. W numeracji określonej poniżej, symbol $\mathrm{V}_{n,m}$ oznacza zbiór z klasy Σ_n o numerze m, a symbol $\Lambda_{n,m}$ oznacza zbiór z klasy Π_n o numerze m.

$$\begin{aligned} \mathrm{V}_{1,m} &= W_m; \\ \Lambda_{n,m} &= -\mathrm{V}_{n,m}; \\ \mathrm{V}_{n+1,m} &= \{x \in \mathbb{N} \mid \exists y\, \langle x, y\rangle \in \Lambda_{n,m}\} \end{aligned}$$

Odpowiednikiem problemu stopu w klasie Σ_n jest zbiór $S_n = \{\langle x, y\rangle \mid x \in V_{n,y}\}$.

Fakt 7.5 *Dla dowolnego n zbiór S_n jest* zupełny *w klasie Σ_n, tj. $S_n \in \Sigma_n$ oraz $A \leq S_n$ dla dowolnego $A \in \Sigma_n$.*

Dowód: Zauważmy, że $\langle x, y\rangle \in S_{n+1}$ wtedy i tylko wtedy, gdy $\langle\langle x, z\rangle, y\rangle \notin S_n$ dla pewnego z. Przez indukcję można więc łatwo pokazać, że $S_n \in \Sigma_n$ dla wszystkich n. Dalej, jeśli $A = \mathrm{V}_{n,m}$, to $x \in A$ jest równoważne $\langle x, m\rangle \in S_n$. □

Hierarchia arytmetyczna jest ostra, w następującym sensie:

Fakt 7.6 *Dla każdego $n \geq 1$ zachodzą inkluzje $\Sigma_n, \Pi_n \subsetneq \Sigma_n \cup \Pi_n \subsetneq \Delta_{n+1}$.*

Dowód: Rozpatrzmy zbiór $K_n = \{m \in \mathbb{N} \mid \langle m, m\rangle \in S_n\}$. Wówczas oczywiście $K_n \in \Sigma_n$. Mamy jednak $K_n \notin \Pi_n$, w przeciwnym razie mielibyśmy bowiem $-K_n \in \Sigma_n$ skąd $-K_n = \mathrm{V}_{n,m}$ dla pewnego m. Wtedy $m \in K_n$ byłoby równoważne temu, że $m \notin K_n$.

A zatem $\Sigma_n \not\subseteq \Pi_n$, co implikuje $\Pi_n \subsetneq \Sigma_n \cup \Pi_n$. Podobnie jest dla Σ_n. Inkluzja $\Sigma_n \cup \Pi_n \subseteq \Delta_{n+1}$ wynika natychmiast z tego, że dostawienie dodatkowego kwantyfikatora (nie wiążącego żadnej zmiennej) nie zmienia znaczenia formuły. Pozostaje pokazać, że i ta inkluzja jest ostra.

Zacznijmy od tego, że klasy Σ_n i Π_n są zamknięte ze względu na sumę zbiorów. Mamy bowiem takie tautologie pierwszego rzędu:

$$\begin{aligned} \exists x \varphi(x) \vee \exists x \psi(x) &\leftrightarrow \exists x(\varphi(x) \vee \psi(x)); \\ \forall x \varphi(x) \vee \forall x \psi(x) &\leftrightarrow \forall x \forall y(\varphi(x) \vee \psi(y)). \end{aligned}$$

Zatem np. alternatywę dwóch formuł postaci $\exists x_1 \forall x_2 \ldots \varphi(x_1, x_2, \ldots)$ i $\exists x_1 \forall x_2 \ldots \psi(x_1, x_2, \ldots)$ można zapisać jako jedną formułę $\exists x_1 \forall x_2 \forall y_2 \ldots (\varphi(x_1, x_2, \ldots) \vee \psi(x_1, y_1, \ldots))$. Podobnie jest z iloczynem zbiorów, a więc klasy Δ_n są ciałami zbiorów (algebrami Boole'a). Tymczasem suma $\Sigma_n \cup \Pi_n$ ciałem zbiorów nie jest i z tego wynika nasza teza.

Aby to udowodnić, rozpatrzmy dwa zbiory $X = \{2k \mid k \in K_n\}$ oraz $Y = \{2k+1 \mid k \notin K_n\}$. Wtedy $X \leq K_n$ oraz $Y \leq -K_n$, skąd $X \in \Sigma_n$ i $Y \in \Pi_n$ (zob. ćwiczenie 3). Z drugiej strony zarówno $K_n \leq X \cup Y$ jak i $-K_n \leq X \cup Y$, gdyby więc $X \cup Y$ należało do Σ_n to mielibyśmy $-K_n \in \Sigma_n$, czyli sprzeczność. Podobnie, gdyby $X \cup Y \in \Pi_n$, to $K_n \in \Pi_n$ i też byłoby źle. Zatem $X \cup Y \notin \Sigma_n \cup \Pi_n$, a więc klasa $\Sigma_n \cup \Pi_n$ nie jest zamknięta ze względu na sumę. □

Przykłady

Fakt 7.7 *Zbiór* $A = \{n \mid \varphi_n$ jest funkcją całkowitą$\} = \{n \mid W_n = \mathbb{N}\}$ *jest zupełny w klasie* Π_2.

Dowód: Że nasz zbiór jest w klasie Π_2, to już wiemy. Załóżmy więc, że $X \in \Pi_2$. Wtedy istnieje takie $Z \in \Sigma_1$, że $x \in X$ wtedy i tylko wtedy, gdy wszystkie pary postaci $\langle x, y\rangle$ są w zbiorze Z. Jeśli teraz $\varphi_{f(x)} = \lambda y.\, \chi_Z(x, y)$ to $x \in X$ wtedy i tylko wtedy, gdy $\varphi_{f(x)}$ jest funkcją całkowitą. Mamy więc redukcję X do A. □

Fakt 7.8 *Zbiór* $B = \{n \mid W_n$ jest nieskończony$\}$ *jest zupełny w klasie* Π_2.

Dowód: Zbiór B jest w Π_2, bo własność $n \in B$ można wypowiedzieć tak: *Dla dowolnego* x *istnieje takie* $y > x$, *że* $y \in W_n$. Zupełność wynika z następującej redukcji $A \leq B$. Dla danego n niech $Z_n = \{x \mid \{0, \ldots, x-1\} \subseteq W_n\}$. Mamy $n \in A$ wtedy i tylko wtedy, gdy Z_n jest nieskończony, co oznacza redukcję $A \leq B$, bo numer zbioru Z_n jest obliczalny. □

Wniosek 7.9 *Zbiór* $-B = \{n \mid W_n$ jest skończony$\}$ *jest zupełny w klasie* Σ_2.

Następny będzie przykład zupełny w klasie Σ_3, ale to wymaga pewnych przygotowań.

Lemat 7.10 *Klasy* Π_n *i* Σ_n *są zamknięte ze względu na kwantyfikatory ograniczone.*

Dowód: Klasa zbiorów rekurencyjnych jest zamknięta ze względu na kwantyfikatory ograniczone, więc dla $n = 0$ teza zachodzi. Dalej użyjemy indukcji. Przypuśćmy, że klasa Σ_n jest zamknięta ze względu na kwantyfikatory ograniczone i rozpatrzmy własność $P(x, y)$ postaci $\exists v \leq y \forall z\, \phi(x, y, z, v)$, gdzie formuła $\forall z\, \phi(x, y, z, v)$ ma $n+1$ kwantyfikatorów (tj. definiuje zbiór klasy Π_{n+1}). Własność taką możemy równoważnie wyrazić formułą $\forall u \exists v \leq y \forall z \leq u\, \phi(x, y, z, v)$, a na mocy założenia indukcyjnego wyrażenie $\exists v \leq y \forall z \leq u\, \phi(x, y, z, v)$ definiuje zbiór klasy Σ_n.

Jeszcze łatwiej jest w przypadku własności postaci $\forall v \leq y \forall z\, \phi(x, y, z, v)$. Zatem pokazaliśmy tezę dla Π_{n+1}, a dla Σ_{n+1} argumentacja jest dualna. □

Symbol $\forall^\infty x$ czytamy „dla prawie wszystkich x". Zastępuje on wyrażenie $\exists y \forall x (x \geq y \to \cdots)$.

Lemat 7.11 *Niech $X = \{x \mid \forall^\infty w \forall v.\, \langle w, v, x\rangle \in Z\}$, gdzie $Z \subseteq \mathbb{N}^2$ jest rekurencyjnie przeliczalny. Istnieje taki rekurencyjnie przeliczalny zbiór $T \subseteq \mathbb{N}^3$, że $X = \{x \mid \forall^\infty z(\langle z, x\rangle \in T)\}$.*

Dowód: Na początek zauważmy, że następujące formuły są prawdziwe w $\mathcal{N}$:

$$\begin{aligned} \forall^\infty x \forall y\, \varphi(x,y) &\leftrightarrow \forall^\infty z(\varphi(\ell(z), r(z)) \vee \exists w{<}r(z).\, \neg\varphi(\ell(z), w));\\ \forall^\infty x(\forall y \varphi(x,y) \vee \exists y \psi(x,y)) &\leftrightarrow \forall^\infty z(\varphi(\ell(z), r(z)) \vee \exists w{<}r(z)\, \neg\varphi(\ell(z), w) \vee \exists w \psi(\ell(z), w)). \end{aligned}$$

Użyjemy powyższych równoważności do przeformułowania warunku „$\forall^\infty w \forall v.\, \langle w, v, x\rangle \in Z$”. Najpierw zastosujemy pierwszą równoważność, zauważając przy tym, że wyrażenie postaci $\exists w < r(z)\, \langle \ell(z), w, x\rangle \notin Z$ definiuje zbiór klasy Π_1 (lemat 7.10). Możemy więc zastosować drugą równoważność i otrzymamy

$$x \in X \quad \text{wtedy i tylko wtedy, gdy} \quad \forall^\infty z(\langle z, x\rangle \in Q_1 \vee \langle z, x\rangle \in Q_2 \vee \exists w \langle z, x, w\rangle \in Q_3),$$

gdzie Q_1, Q_2, Q_3 są rekurencyjne. A zatem poszukiwanym zbiorem jest:

$$T = \{\langle z, x\rangle \mid \langle z, x\rangle \in Q_1 \vee \langle z, x\rangle \in Q_2 \vee \exists w \langle z, x, w\rangle \in Q_3\}. \qquad \square$$

Fakt 7.12 *Zbiór $C = \{n \mid W_n$ jest ko-skończony$\}$ jest zupełny w klasie Σ_3.*

Dowód: Ko-skończoność zbioru W_n wyraża zdanie: *Istnieje takie x, że każde $y \geq x$ należy do W_n*. Mamy więc $C \in \Sigma_3$. Dla $X \in \Sigma_3$ określimy teraz redukcję $X \leq C$. Niech $Y \in \Pi_2$ będzie takim zbiorem, że $x \in X$ wtedy i tylko wtedy, gdy $\exists y\, \langle x, y\rangle \in Y$.

Własność $\exists y\, \langle x, y\rangle \in Y$ jest równoważna stwierdzeniu $\forall^\infty w \exists y{\leq}w.\, \langle x, y\rangle \in Y$, a ponieważ klasa Π_2 jest zamknięta ze względu na kwantyfikatory ograniczone, więc możemy napisać, że

$$x \in X \quad \text{wtedy i tylko wtedy, gdy} \quad \forall^\infty w \forall v.\, \langle w, v, x\rangle \in Z,$$

dla pewnego rekurencyjnie przeliczalnego Z. Użyjemy teraz lematu 7.11 i dostaniemy:

$$x \in X \quad \text{wtedy i tylko wtedy, gdy} \quad \forall^\infty z(\langle z, x\rangle \in T),$$

gdzie T jest pewnym zbiorem rekurencyjnie przeliczalnym. Jeśli $T_x = \{z \mid \langle z, x\rangle \in T\}$, to

$$x \in X \quad \text{wtedy i tylko wtedy, gdy} \quad T_x \text{ jest ko-skończony,}$$

A więc $X \leq C$, bo numer zbioru T_x zależy w sposób obliczalny od x. $\square$

Wniosek 7.13 *Zbiór $D = \{n \mid W_n$ jest rekurencyjny$\}$ jest zupełny w klasie Σ_3.*

Dowód: Sprawdzenie, że $D \in \Sigma_3$ pozostawiamy Czytelnikowi (ćwiczenie 6). Dowód zupełności polega na redukcji $C \leq D$. Dla dowolnego $x \in \mathbb{N}$ rozpatrzmy zbiór

$$Z(x) = \{\langle u, v\rangle \mid v \in W_x \vee (u < v \wedge u \in K)\}$$

Zbiór $Z(x)$ jest rekurencyjnie przeliczalny, a jego numer zależy w sposób obliczalny od x. Ponadto zbiór W_x jest ko-skończony wtedy i tylko wtedy, gdy zbiór $Z(x)$ jest rekurencyjny. Istotnie, przypuśćmy najpierw, że W_x jest ko-skończony. Wtedy nierozstrzygalny drugi człon alternatywy w definicji $Z(x)$ dotyczy w istocie tylko skończenie wielu par $\langle u, v\rangle$, a każdy

zbiór skończony jest rekurencyjny. Zatem $Z(x)$ jest rekurencyjny. Załóżmy więc, że $-W_x$ jest zbiorem nieskończonym. Wtedy mamy takie równoważności

$$\begin{aligned} u \in K &\Leftrightarrow \forall v(u < v \to \langle u, v\rangle \in Z(x)); \\ u \notin K &\Leftrightarrow \exists v(u < v \wedge \langle u, v\rangle \notin Z(x)). \end{aligned}$$

Jeśli $Z(x)$ jest zbiorem rekurencyjnym, to z warunku powyżej wynika rekurencyjna przeliczalność zbioru $-K$. □

Ćwiczenia

1. Udowodnić chińskie twierdzenie o resztach. A jak się nie uda, to znaleźć w książkach.
2. Udowodnić, że zbiór zdań prawdziwych w standardowym modelu arytmetyki jest nierozstrzygalny. *Wskazówka:* Użyć wniosku 7.3.
3. Udowodnić, że jeśli $A \leq B \in \Sigma_n$ to $A \in \Sigma_n$ oraz, że $A \leq B \in \Pi_n$ implikuje $A \in \Pi_n$.
4. Udowodnić, że zbiór $\{\langle x, y\rangle \mid W_x = W_y\}$ jest zupełny w klasie Π_2.
5. Uzupełnić dowód lematu 7.11.
6. Udowodnić, że zbiór $D = \{x \mid W_x$ jest rekurencyjny$\}$ należy do Σ_3. *Wskazówka:* Zauważyć, że $x \in D$ wtedy i tylko wtedy, gdy $\exists n \forall m(m \notin W_x \leftrightarrow m \in W_n)$. Nastepnie sprowadzić tę formułę do postaci normalnej z prefiksem typu $\exists\forall\exists$.
7. Udowodnić, że jeśli zbiór A jest rekurencyjny względem funkcji f (por. ćwiczenie 6.2), to A jest definiowalny formułą $\varphi(\vec{x}, f)$ z nową stałą funkcyjną f.

8 Zbiory analityczne

Formuły arytmetyki pierwszego rzędu, w których pod kwantyfikatorami występują zmienne indywiduowe (o wartościach liczbowych), definiują zbiory arytmetyczne. O języku *drugiego rzędu* mówimy wtedy, gdy kwantyfikatory wiążą zmienne przebiegające wartości, które są funkcjami lub relacjami. Dla naszych celów wystarczy założyć, że w formułach mogą występować zmienne funkcyjne (których znaczeniem może być dowolna całkowita funkcja z $\mathbb{N}$ do $\mathbb{N}$) i kwantyfikatory $\forall f$ i $\exists f$, wiążące takie zmienne.

Funkcja $f : \mathbb{N} \to \mathbb{N}$ może na przykład reprezentować nieskończone obliczenie maszyny Turinga, w ten sposób, że $f(n)$ jest kodem n-tej konfiguracji występującej w takim obliczeniu). Nietrudno napisać formułę stwierdzającą, że tak jest: trzeba napisać, że $f(0)$ to kod konfiguracji początkowej, i że dla dowolnego n, konfiguracja $f(n+1)$ powstaje w jednym kroku z $f(n)$. Dzięki temu możemy np. zdefiniować taki zbiór

$$\begin{aligned} H = \{k \mid &\text{ maszyna } M_k \text{ ma nieskończone obliczenie,} \\ &\text{w którym stan } q_0 \text{ występuje nieskończenie wiele razy}\}. \end{aligned}$$

Definicja 8.1 Zbiór $A \subseteq \mathbb{N}$ jest *analityczny*, wtedy i tylko wtedy, gdy istnieje taka formuła $\varphi(x)$ drugiego rzędu, o co najwyżej jednej zmiennej wolnej x, że dla dowolnego $n \in \mathbb{N}$:

$$n \in A \quad \text{wtedy i tylko wtedy, gdy} \quad \mathcal{N} \models \varphi(\underline{n}).$$

Jeśli formuła $\varphi(x)$ ma postać $\exists f_1 \forall f_2 \ldots Q_n f_n \, \psi(f_1, \ldots, f_n, x)$, gdzie wszystkie kwantyfikatory funkcyjne (ogólne i szczegółowe na przemian) występują na początku,[14] a w ψ już takich kwantyfikatorów nie ma, to mówimy, że zbiór A jest *klasy* Σ^1_n. Analogicznie, zbiór klasy Π^1_n to zbiór, który można zdefiniować formułą postaci $\forall f_1 \exists f_2 \ldots Q_n f_n \, \psi(f_1, \ldots, f_n, x)$.

A zatem zbiór H powyżej jest klasy Σ^1_1, bo do jego zdefiniowania wystarczy jeden kwantyfikator funkcyjny $\exists$.

Nazwa „zbiór analityczny" bierze się stąd, że równoważną definicję tego pojęcia można podać, odwołując się do dziedziny $\mathbb{R}$ liczb rzeczywistych, dokładniej do modelu $\langle \mathbb{R}, +, \cdot, 0, 1, \mathit{int} \rangle$, w którym dodatkowy jednoargumentowy predykat int wyróżnia liczby naturalne (tj. $r \in \mathit{int}$ zachodzi wtedy i tylko wtedy, gdy r jest liczbą naturalną). Zbiory analityczne to dokładnie te zbiory, które można w tym modelu zdefiniować formułą pierwszego rzędu postaci $\mathit{int}(x) \wedge \varphi(x)$. Dowód tego faktu, który można znaleźć np. w książce Rogersa, polega na reprezentowaniu liczb rzeczywistych przez ciągi liczb naturalnych, np. przy pomocy tzw. ułamków ciągłych (łańcuchowych).

Fakt 8.2 *Każdy zbiór analityczny jest klasy* Σ^1_n *lub* Π^1_n *dla pewnego* n. *Co więcej, dla* $n > 0$ *można go zdefiniować formułą (o jednej zmiennej wolnej* x*) postaci*

$$Q_1 f_1 Q_2 f_2 \ldots Q_n f_n Q' y \, P(f_1, f_2, \ldots, f_n, y, x),$$

gdzie kwantyfikator Q' *jest indywiduowy, a warunek* $P(f_1, f_2, \ldots, f_n, y, x)$ *jest rekurencyjny względem*[15] *funkcji* $f_1, f_2, \ldots, f_n$.

Dowód: Oczywiście najpierw sprowadzamy formułę do preneksowej postaci normalnej. Dwa kwantyfikatory tego samego rodzaju możemy „sklejać" stosując funkcję pary, podobnie jak w przypadku arytmetyki pierwszego rzędu. Przy tym kwantyfikatory indywiduowe możemy permutować z funkcyjnymi stosując ćwiczenie 1. Wreszcie „końcówkę" postaci $\forall f \exists x \forall y$ można skrócić zamieniając ją kolejno na „końcówkę" postaci $\forall f \forall g \exists x$ a potem postaci $\forall F \exists x$. Szczegóły pozostawiamy Czytelnikowi. □

Klasa $\Sigma^1_0 = \Pi^1_0$ to klasa zbiorów arytmetycznych. Zbiory klasy $\Delta^1_1 = \Sigma^1_1 \cap \Pi^1_1$ nazywamy *hiperarytmetycznymi*. Z poniższego twierdzenia wynika, że klasa zbiorów hiperarytmetycznych jest istotnie szersza niż klasa zbiorów arytmetycznych. Można pokazać np. że zbiór (numerów) zdań prawdziwych w standardowym modelu arytmetyki $\langle \mathbb{N}, +, \cdot, \boldsymbol{s}, 0 \rangle$ jest hiperarytmetyczny. A z twierdzenia Tarskiego (o niewyrażalności pojęcia prawdy) wiemy, że nie da się go zdefiniować w języku arytmetyki.

Twierdzenie 8.3 (o hierarchii analitycznej)

Dla wszystkich $n \geq 0$ *zachodzą ostre inkluzje* $\Sigma^1_n, \Pi^1_n \subsetneq \Sigma^1_n \cup \Pi^1_n \subsetneq \Sigma^1_{n+1} \cap \Pi^1_{n+1}$.

Dowód twierdzenia o hierarchii opuszczamy. Pokażemy za to konkretny przykład.

Fakt 8.4 *Zbiór* H *określony na początku tego rozdziału jest* Σ^1_1*-zupełny.*

[14] Rodzaj kwantyfikatora Q_n zależy od parzystości n.
[15] Por. ćwiczenie 6.2.

Dowód: Aby naszkicować dowód posłużymy się faktem 8.2 i ćwiczeniem 2 z Rozdziału 6. Wynika z nich, że każdy zbiór $A \in \Sigma^1_1$ można zdefiniować formułą postaci $\exists f \forall y P(f, x, y)$, gdzie warunek $P(f, x, y)$ można rozpoznawać maszyną Turinga z wyrocznią dla funkcji f.

Pokażemy, że dla każdego takiego zbioru A zachodzi $A \leq_m H$. Dla danego $m \in \mathbb{N}$, aby sprawdzić, czy $m \in A$, skonstruujemy maszynę $M(m)$ spełniającą równoważność:

$$m \in A \qquad \text{wtedy i tylko wtedy, gdy numer maszyny } M(m) \text{ należy do } H.$$

Maszyna $M(m)$ dla kolejnych liczb $y = 0, 1, 2, \ldots$ usiłuje sprawdzać warunek $P(f, m, y)$, „zgadując" niedeterministycznie potrzebną informację o funkcji f (i zapisując to co już zgadła dla ew. ponownego użycia). Po każdej fazie pracy (pozytywnym zweryfikowaniu warunku $P(f, m, y)$, dla kolejnej wartości y) maszyna wchodzi w stan q_0 i dopiero potem rozpoczyna następną fazę. Szczegóły konstrukcji pomijamy. □

Ćwiczenia

1. Uzupełnić dowód faktu 8.2. Wskazówka: formułę postaci $\forall x \exists f\, P(f, x)$ można zastąpić równoważną formułą postaci $\exists F \forall x\, P(\lambda y.F(x, y), x)$, z formułą o prefiksie $\exists x \forall f$ można postąpić podobnie, używając prawa De Morgana.
2. Niech $\varphi_m(x)$ oznacza m-tą (w pewnej numeracji) formułę o jednej zmiennej wolnej x. Jeżeli $S : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ jest taką funkcją, że $S(m, n) = 0$ wtedy i tylko wtedy, gdy $\mathbb{N} \models \varphi_m(\underline{n})$, to powiemy, że S jest *definicją prawdy*. Udowodnić, że nie istnieje arytmetyczna definicja prawdy.
3. Udowodnić, że zbiór zdań prawdziwych w $\mathbb{N}$ jest hiperarytmetyczny. *Wskazówka:* Istnieje taka formuła $\psi(X)$, że $\mathbb{N} \models \psi(S)$ wtedy i tylko wtedy, gdy S jest definicją prawdy.
4. Udowodnić, że zbiór tautologii klasycznej logiki drugiego rzędu nie jest analityczny. *Wskazówka:* Zdefiniować prawdę drugiego rzędu.

9 Problemy decyzyjne

Oczywiście sens pojęcia obliczalności jest taki: zbiór słów Z jest obliczalny wtedy, gdy istnieje algorytm, który dla danego słowa w zawsze poprawnie odpowiada „tak" lub „nie" na pytanie, czy $w \in Z$.

Takie pytania nazywamy często „problemami decyzyjnymi". Formalnie, *problem decyzyjny* to po prostu zbiór słów nad ustalonym alfabetem.

W praktyce „problemami decyzyjnymi" nazywamy na przykład takie pytania:

– „Czy dany graf skończony ma ścieżkę Hamiltona?"

– „Czy dana formuła rachunku predykatów jest tautologią?"

gdzie daną wejściową, czyli *instancją* problemu, nie jest słowo, ale jakiś obiekt kombinatoryczny. Oczywiście skończony graf można łatwo przedstawić za pomocą słowa (trzeba po prostu wypisać jego wierzchołki i krawędzie). Podobnie można też postąpić z innymi skończonymi obiektami. Ale trzeba tu zrobić dwie uwagi:

Po pierwsze, nasze pojęcie „(nie)rozstrzygalnego problemu decyzyjnego" ma zastosowanie tylko do tych obiektów matematycznych, które dają się przedstawić w *skończony* sposób.

Nie jest więc problemem decyzyjnym zadanie:

– „Czy dany graf nieskończony jest spójny?"

O ile powyższa obserwacja jest dość oczywista, następna uwaga dotyczy pułapki, w którą wpadają nawet doświadczeni badacze. Otóż np. w zadaniu dotyczącym formuł rachunku predykatów[16] instancją problemu nie jest dowolne słowo, ale poprawnie zbudowana formuła. Zatem nie jest to, ściśle biorąc, problem decyzyjny. Ale oczywiście możemy łatwo wybrnąć z sytuacji zadając nieco zmodyfikowane pytanie:

– „Czy dane słowo jest tautologią rachunku predykatów?".

Łatwo, bo istnieje prosty algorytm sprawdzający czy dane słowo jest formułą, i w istocie nie ma znaczenia, które pytanie postawimy.

W praktyce nikt nie formułuje problemów decyzyjnych w taki sposób. Ważne tylko, żeby pytanie:

– „Czy dane słowo jest poprawną instancją problemu?"

(tj. formułą, reprezentacją grafu itp.) samo było rozstrzygalne.[17] Zwykle tak jest i to w oczywisty sposób. Ale może tak nie być. Na przykład to pytanie nie stanowi poprawnego problemu decyzyjnego:

– „Czy dana tautologia (klasycznego) rachunku predykatów jest także tautologią intuicjonistyczną?"

Albo to:

– „Dana jest maszyna $\mathcal{M}$, zatrzymująca się dla słowa pustego. Czy $\mathcal{M}$ akceptuje słowo puste?"

O tym, że mamy tu tylko pozorną „rozstrzygalność" świadczy to, że dla powyższego problemu nie można sensownie określić złożoności obliczeniowej.

Fakt 9.1 *Niech f będzie dowolną funkcją rekurencyjną. Każdy algorytm odpowiadający poprawnie na pytanie „*Czy maszyna $\mathcal{M}$ akceptuje słowo puste?*" zawsze gdy $\mathcal{M}$ zatrzymuje się dla słowa pustego, wymaga więcej niż $f(n)$ kroków dla pewnego wejścia rozmiaru n.*

Dowód: Niech L będzie językiem obliczalnym, który nie należy do klasy $\textsc{Dtime}(2^{f(2n)})$ i niech M_L będzie maszyną rozpoznającą L. Dla danego słowa w, niech M_w będzie maszyną, która zawsze się zatrzymuje i akceptuje dowolne wejście x wtedy i tylko wtedy, gdy $w \in L$ (jej działanie nie zależy od danej wejściowej x). Mając dane w, można skonstruować taką maszynę w czasie równym długości słowa w plus pewna stała — wystarczy użyć maszyny M_L.

Przypuśćmy teraz, że mamy algorytm, który w czasie $f(n)$ rozwiązuje nasz problem. Aby stwierdzić, czy $w \in L$ wystarczy teraz uruchomić ten algorytm dla maszyny M_w. Łącznie z konstrukcją samej maszyny wymaga to czasu rzędu $2n + f(2n)$, gdzie n to długość słowa w, a więc mniej niż $2^{f(2n)}$. □

Z dotychczasowych naszych rozważań wynika nierozstrzygalność szeregu ważnych problemów

[16] Nazwa „problem decyzyjny" (Entscheidungsproblem) została po raz pierwszy użyta właśnie w odniesieniu do tego zadania.

[17] Jeśli interesuje nas nie tylko rozstrzygalność ale także złożoność problemu, to sprawdzenie poprawności instancji powinno być zadaniem o pomijalnej złożoności.

decyzyjnych. Pierwszą grupę stanowią własności języków i funkcji obliczanych przez maszyny Turinga (lub też inne równoważne modele obliczeń), zwłaszcza różne warianty problemu stopu. Ich nierozstrzygalność wynika z twierdzenia Rice'a.

- Czy dana maszyna Turinga zatrzymuje się dla danego wejścia?
- Czy dana maszyna Turinga zatrzymuje się dla słowa pustego?
- Czy dana maszyna Turinga akceptuje język pusty (nieskończony, ko-skończony, rekurencyjny, itd.)?
- Czy dana maszyna Turinga oblicza funkcję rosnącą?

Uwaga: problem stopu jest nierozstrzygalny także dla pewnych ustalonych maszyn. Tj. można skonstruować konkretną maszynę M dla której pytanie

- Czy dane słowo jest akceptowane przez maszynę M?

jest nierozstrzygalne. Dalej mamy różne problemy dotyczące maszyn Turinga, których nierozstrzygalność nie wynika wprost z twierdzenia Rice'a, ale z odpowiedniej redukcji:

- Czy dana maszyna Turinga ma obliczenie, w którym występuje dany stan?
- Czy dana niedeterministyczna maszyna Turinga ma obliczenie nieskończone?
- Czy dana niedeterministyczna maszyna Turinga ma obliczenie nieskończone, w którym dany stan powtarza się nieskończenie wiele razy?

Ten ostatni problem jest *wysoce nierozstrzygalny:* nie mieści się w hierarchii arytmetycznej. Każdy model pojęcia obliczalności ma oczywiście swoją wersję problemu stopu, na przykład nierozstrzygalne są problemy:

- Czy dana gramatyka typu zero generuje dane słowo?
- Czy w danej gramatyce typu zero można z danego słowa w otrzymać w drodze redukcji dane słowo v? (Problem osiągalności dla gramatyk typu zero.)
- Czy dany while-program zatrzymuje się dla danej wartości wejściowej?

Rachunek lambda też ma swoje przyjemności, między innymi takie:

- Dane termy M i N. Czy $M \twoheadrightarrow_\beta N$?
- Dane termy M i N. Czy $M =_\beta N$?
- Dany term M. Czy M ma postać normalną?
- Dany term M. Czy M ma własność silnej normalizacji?

Uwaga: nierozstrzygalność trzech pierwszych problemów wynika wprost z twierdzenia 2.11, ale nierozstrzygalność czwartego problemu wymaga delikatniejszej analizy konstrukcji użytej w dowodzie tego twierdzenia.

Dwa problemy nierozstrzygalne, o których będzie teraz mowa, są szczególnie często używane w dowodach nierozstrzygalności innych problemów.

Automaty dwulicznikowe

Przez *automat dwulicznikowy* rozumiemy krotkę $\mathcal{A} = \langle Q, q_0, q_f, I\rangle$, w której Q jest skończonym zbiorem stanów, q_0 jest stanem początkowym, q_f stanem końcowym, a I jest zbiorem *instrukcji*, z których każda jest jednej z postaci

1. $(q : c_i := c_i + 1;$ `goto` $p)$;
2. $(q : c_i := c_i - 1;$ `goto` $p)$;
3. $(q :$ `if` $c_i = 0$ `then goto` p `else goto` $r)$;

gdzie $i = 1, 2$ oraz $q, p, r \in Q$. *Konfiguracja* takiego automatu $\mathcal{A}$ to trójka $\langle q, m, n\rangle$ złożona ze stanu $q \in Q$ i liczb $m, n \in \mathbb{N}$, stanowiących bieżącą zawartość liczników c_1 i c_2. Konfiguracja *początkowa* ma postać $\mathcal{C}_0 = \langle q_0, 0, 0\rangle$, a konfiguracje $\langle q_f, m, n\rangle$ są *końcowe*.

Znaczenie instrukcji jest oczywiste. Piszemy $\mathcal{C}_1 \twoheadrightarrow_{\mathcal{A}} \mathcal{C}_2$, gdy $\mathcal{C}_2$ można otrzymać z $\mathcal{C}_1$ po pewnej liczbie kroków.

Fakt 9.2 *Problem stopu dla automatów dwulicznikowych* (czy $\mathcal{C}_0 \twoheadrightarrow_{\mathcal{A}} \mathcal{C}$, dla pewnej konfiguracji końcowej?) *jest nierozstrzygalny.*

Dowód: Jest to dosyć łatwa konsekwencja nierozstrzygalności problemu stopu dla while-programów. Najpierw łatwo pokazujemy że automat z wieloma licznikami potrafi naśladować obliczenie while-programu. Potem zawartość k liczników reprezentujemy za pomocą jednego, używając tricku z Rozdziału 1:

$$kod(a_0 \dots a_k) = 2^{a_0} 3^{a_1} \cdots p_k^{a_k},$$

Polecenie zwiększenia lub zmniejszenia wartości a_i o jeden sprowadza się teraz do pomnożenia lub podzielenia wartości licznika przez p_i. Można to łatwo zrobić, używając drugiego licznika jako pomocniczego. □

Problem odpowiedniości Posta

Problem odpowiedniości Posta (PCP) jest klasycznym przykładem problemu nierozstrzygalnego. Formułujemy go tak:

Dany jest skończony zbiór par słów $\{(x_i, y_i) \mid i = 1, \dots, n\}$ (*system par Posta*[18]).

[18] Nazwa „system Posta" używana też jest w innym znaczeniu.

Czy istnieje taki niepusty ciąg $i_1, i_2, \ldots, i_m$, że zachodzi równość
$$x_{i_1} x_{i_2} \ldots x_{i_m} = y_{i_1} y_{i_2} \ldots y_{i_m} \text{?}$$

Poszukiwany ciąg $i_1, i_2, \ldots, i_m$ nazywamy *rozwiązaniem* systemu par Posta. Także słowo

$$x_{i_1} x_{i_2} \ldots x_{i_m}$$

bywa wtedy nazywane rozwiązaniem.

System par Posta $\{(x_i, y_i) \mid i = 1, \ldots, n\}$ przedstawia się często w taki sposób:

x_1	x_2	$\ldots$	x_n
y_1	y_2	$\cdots$	y_n

Na przykład taki system:

a^2	b^2	ab^2
a^2b	ba	b

ma rozwiązanie 1213 (bo $a^2 \cdot b^2 \cdot a^2 \cdot ab^2 = a^2b^2a^3b^2 = a^2b \cdot ba \cdot a^2b \cdot b$), a systemy

a^2b	a
a^2	ba^2

ba^2	ba^2
b	a^2b

nie mają rozwiązań.

Twierdzenie 9.3 *Problem odpowiedniości Posta jest nierozstrzygalny.*

Dowód: Redukcja z problemu osiągalności dla gramatyk typu zero (strona 37). Niech $G = \langle \mathcal{A}, \mathcal{N}, R, \xi_0 \rangle$ bedzie gramatyką typu zero i niech $\Sigma = \mathcal{A} \cup \mathcal{N}$. Ustalmy słowa $w, v \in \Sigma^*$. Skonstruujemy (efektywnie[19]) system par Posta P, który będzie miał rozwiązanie wtedy i tylko wtedy, gdy $G \vdash w \twoheadrightarrow v$.

Alfabet naszego systemu par będzie taki:

$$\Delta = \Sigma \cup \{\overline{a} \mid a \in \Sigma\} \cup \{*, \overline{*}\} \cup \{\texttt{[}, \texttt{]}\}.$$

Jeśli $u = a_1 \ldots a_r$ to symbol $\overline{u}$ oznacza oczywiście słowo $\overline{a}_1 \ldots \overline{a}_r$.

Reguły przedstawimy tabelką, w której a oznacza dowolny symbol alfabetu Σ, a $\alpha \Rightarrow \beta$ jest dowolną produkcją gramatyki. Należy to rozumieć tak, że do systemu P należą wszystkie pary zgodne z szablonem w którejś kolumnie tabeli.

$\texttt{[}w*$	a	$\overline{a}$	$*$	$\overline{*}$	$\texttt{]}$	$\texttt{]}$	β	$\overline{\beta}$
$\texttt{[}$	$\overline{a}$	a	$\overline{*}$	$*$	$\overline{*}v\texttt{]}$	$*\overline{v}\texttt{]}$	$\overline{\alpha}$	α

[19] Tj. opiszemy w istocie algorytm realizujący tę konstrukcję.

Na przykład jeśli $w = baca$, $v = owad$, a regułami naszej gramatyki są

$$ba \Rightarrow ow \qquad ca \Rightarrow ad,$$

to mamy taki system par Posta:

$[baca*$	a	$\overline{a}$	$\cdots$	$*$	$\overline{*}$	ow	$\overline{ow}$	ad	$\overline{ad}$	]	]
[	$\overline{a}$	a	$\cdots$	$\overline{*}$	$*$	$\overline{ba}$	ba	$\overline{ca}$	ca	$\overline{*}owad]$	$*\overline{owad}]$

Załóżmy teraz, że $w = w_0 \to_G w_1 \to_G w_2 \to_G \cdots \to_G w_k = v$, i dla ustalenia uwagi przyjmijmy, że k jest parzyste. Wtedy nasz system par Posta można rozwiązać tak:

$$[w_0 * \cdot \overline{w}_1\overline{*} \cdot w_2 * \cdots * \overline{w}_{k-1}\overline{*} \cdot w_k \cdot] = [\cdot w_0 * \cdot \overline{w}_1\overline{*} \cdot w_2 * \cdots * \overline{w}_{k-1} \cdot \overline{*}w_k]$$

Rzeczywiście, jeśli mamy np. $w_0 = x\alpha y \to_G x\beta y = w_1$, przez zastosowanie produkcji $\alpha \Rightarrow \beta$, to słowa w_0* i $\overline{w}_1\overline{*}$ można przedstawić jako konkatenacje odpowiadających sobie słów z dolnej i górnej części tabelki. Podobnie dla słów $\overline{w}_1\overline{*}$ i w_2* i tak dalej.

Na dodatek, jeśli mamy jakiekolwiek rozwiązanie systemu P, to takie rozwiązanie musi wyznaczać pewne wyprowadzenie $w \twoheadrightarrow_G v$. Rozwiązanie musi się zaczynać od pary ($[w*$, [), jest to bowiem jedyna para, która zaczyna się tak samo (po to są kreski). Jedynym sposobem zgodnego przedłużenia słów z tej pary jest dopisanie do dolnego [słowa $w*$. To jednak oznacza, że do słowa $[w*$ musimy dopisać słowo postaci $\overline{u}_1\overline{*}$ gdzie $w \twoheadrightarrow_G u_1$. Zatem mamy teraz słowa

$$[w * \overline{u}_1\overline{*} \qquad\qquad [w*$$

i musimy do drugiego z nich dopisać $\overline{u}_1\overline{*}$. Ale wtedy do pierwszego dopiszemy u_2*, gdzie $u_1 \twoheadrightarrow_G u_2$ i tak dalej. To się może skończyć tylko użyciem jednej z par zawierających], co oznacza, że $w \twoheadrightarrow v$.

Uwaga: Warunek $u_1 \twoheadrightarrow_G u_2$ powyżej nie musi oznaczać $u_1 \to u_2$. Na przykład przekształcenie bacy w owada może wyglądać nie tylko tak:

$$[baca * \overline{owca*}owad],$$

ale też na przykład tak:

$$[baca * \overline{baca*}owad * \overline{owad}]. \qquad \square$$

Przykładem zastosowania PCP jest taki fakt:

Twierdzenie 9.4 *Problem pustości dla języków kontekstowych jest nierozstrzygalny.*

Dowód: Dla danego systemu par Posta P, można łatwo skonstruować maszynę Turinga, rozpoznającą dokładnie te słowa, które są rozwiązaniami tego systemu. Maszyna ta nie używa więcej taśmy, niż zajmowało słowo wejściowe. Modyfikując nieco konstrukcję z dowodu twierdzenia 2.5, możemy z tej maszyny uzyskać gramatykę monotoniczną generującą zbiór rozwiązań systemu P. W ten sposób zredukujemy problem odpowiedniości Posta do problemu pustości dla języków kontekstowych. $\square$

Systemy Thuego

Gramatykę typu zero, w której nie rozróżnia się symboli terminalnych i nieterminalnych, oraz nie wyróżnia się symbolu początkowego, nazywamy *systemem pół-thue'owskim*.[20] A więc system pół-thue'owski to po prostu skończony zbiór T reguł postaci „$w \Rightarrow v$", gdzie w, v są słowami nad ustalonym alfabetem. Relacja $w \to_T v$ zachodzi wtedy i tylko wtedy, gdy $w = uw_1u'$, $v = uw_2u'$, oraz $w_1 \Rightarrow w_2$ jest w T. Jej domknięcie przechodnio-zwrotne oznaczamy jak zwykle przez $\twoheadrightarrow_T$, a najmniejsza relacja równoważności zawierająca $\to_T$ będzie oznaczana przez $\twoheadleftarrow\!\!\!\twoheadrightarrow_T$. *System Thuego*[21] to taki system pół-thue'owski T, w którym wszystkie reguły są odwracalne, tj. jeśli „$w \Rightarrow v$" $\in T$ to także „$v \Rightarrow w$" $\in T$.

Przez *problem słów* dla systemu Thuego T rozumiemy następujący problem decyzyjny: dane słowa w i v, czy $w \twoheadleftarrow\!\!\!\twoheadrightarrow_T v$?

Twierdzenie 9.5 (Post) *Istnieją systemy Thuego o nierozstrzygalnym problemie słów.*

Dowód: Pokażemy najpierw pewien system półthue'owski, dla którego nierozstrzygalny jest problem osiągalności (czasem nazywany problemem słów): dane słowa w i v, czy $w \twoheadrightarrow_T v$?

Niech $\mathcal{M}$ będzie deterministyczną maszyną Turinga, dla której nierozstrzygalny jest problem stopu (tj. język $L(\mathcal{M})$) Dla uproszczenia załóżmy, że $\mathcal{M}$ ma tylko jeden stan końcowy q_a (akceptujący).

Rozpatrzmy alfabet skladający się ze stanów i symboli maszyny $\mathcal{M}$, oraz dodatkowo z nawiasów kwadratowych (na oznaczenie początku i końca konfiguracji). System półthue'owski P składa się z reguł:

- $qa \Rightarrow bp$, gdy $\delta(q,a) = (b,p,+1)$;
- $qa \Rightarrow pb$, gdy $\delta(q,a) = (b,p,0)$;
- $q\texttt{]} \Rightarrow bp\texttt{]}$, gdy $\delta(q,\mathsf{B}) = (b,p,+1)$;
- $q\texttt{]} \Rightarrow pb\texttt{]}$, gdy $\delta(q,\mathsf{B}) = (b,p,0)$;
- $cqa \Rightarrow pcb$ i $\texttt{[}qa \Rightarrow \texttt{[}pb$, gdy $\delta(q,a) = (b,p,-1)$;
- $cq\texttt{]} \Rightarrow pcb\texttt{]}$, gdy $\delta(q,\mathsf{B}) = (b,p,-1)$;
- $aq_a \Rightarrow q_a$ i $q_aa \Rightarrow q_a$ (dla *wszystkich* a, także dla $a = \texttt{[}, \texttt{]}$).

Wówczas zachodzi równoważność:

$$\texttt{[}q_0w\texttt{]} \twoheadrightarrow_P q_a \quad \text{wtedy i tylko wtedy gdy} \quad w \in L(\mathcal{M}),$$

bo każdy ciąg postaci $\texttt{[}q_0w\texttt{]} \to_P x_1 \to_P x_2 \to_P \cdots \to_P q_a$ musi reprezentować obliczenie maszyny dla słowa w, zakończone „wycieraniem" wszystkich symboli oprócz q_a (patrz dowód twierdzenia 2.5). W istocie mamy jednak lepszą własność:

[20] ang.: *semi-Thue system*.
[21] ang.: *Thue system*.

$$[q_0 w] \twoheadleftarrow\!\!\twoheadrightarrow_P q_a \quad \text{wtedy i tylko wtedy gdy} \quad w \in L(\mathcal{M}).$$

Przypuśćmy bowiem, że $[q_0 w] \twoheadleftarrow\!\!\twoheadrightarrow_P q_a$. Mamy ciąg:

$$[q_0 w] = x_0 \sim x_1 \sim x_2 \sim \cdots \sim x_n = q_a,$$

gdzie każde $\sim$ to albo $\to_P$ albo ${}_P\!\leftarrow$. Niech to będzie *najkrótszy* taki ciąg. Łatwo widzieć, że każde x_i zawiera dokładnie jedno wystąpienie pewnego stanu q. Zauważmy, że jeśli

$$x_{i-1} \ {}_P\!\leftarrow x_i \to_P \ x_{i+1},$$

to w słowie x_i występuje q_a. Inaczej x_i odpowiada niekońcowej konfiguracji maszyny i z powodu determinizmu mamy $x_{i-1} = x_{i+1}$. Ale wtedy nasz ciąg można skrócić. A więc nasza redukcja jest postaci:

$$[q_0 w] = x_0 \twoheadrightarrow_P x_m \sim x_{m+1} \sim \cdots \sim x_n = q_a,$$

przy czym x_m jest konfiguracją końcową. Najkrótszy sposób na otrzymanie q_a z konfiguracji końcowej to po prostu eliminacja pozostałych symboli (wytarcie każdego symbolu wymaga co najmniej jednego kroku). Skoro więc nasz ciąg jest najkrótszy, to faktycznie mamy redukcje $[q_0 w] = x_0 \twoheadrightarrow_P x_m \twoheadrightarrow_P x_n = q_a$. A zatem $[q_0 w] \twoheadleftarrow\!\!\twoheadrightarrow_P q_a$ faktycznie oznacza $[q_0 w] \twoheadrightarrow_P q_a$.

Niech teraz T będzie systemem Thuego otrzymanym przez dodanie odwróconych wersji wszystkich reguł. Następujące trzy warunki są równoważne:

$$[q_0 w] \twoheadleftarrow\!\!\twoheadrightarrow_T q_a \qquad [q_0 w] \twoheadleftarrow\!\!\twoheadrightarrow_P q_a \qquad w \in L(\mathcal{M}).$$

A zatem problem słów dla T jest nierozstrzygalny. □

Problem słów dla systemów Thuego jest też nazywany *problemem słów w półgrupach*. O systemie Thuego T można bowiem myśleć jak o zbiorze aksjomatów równościowych nad sygnaturą złożoną ze słowa pustego i liter alfabetu (stałych) oraz dwuargumentowej operacji konkatenacji. Ponieważ jednak konkatenacja słów jest łączna, a ε jest jej elementem neutralnym, więc faktycznie mówimy tu o zbiorze zdań T^+ otrzymanym przez dodanie do T aksjomatów *półgrupy z jednością*:

$$\forall xyz(x \cdot (y \cdot z) \approx (x \cdot y) \cdot z);$$
$$\forall x(\varepsilon \cdot x \approx x), \quad \forall x(x \cdot \varepsilon \approx x).$$

Wniosek 9.6 *Zbiór tautologii klasycznej logiki pierwszego rzędu jest nierozstrzygalny.*

Dowód: Z twierdzenia 9.5 wynika nierozstrzygalność następującego problemu:

Dany zbiór zdań T^+ wyznaczony przez system Thuego T, czy $T^+ \models w = v$?

Ale zbiór T^+ jest skończony; jeśli więc φ jest koniunkcją wszystkich zdań z T^+, to nasze zadanie możemy sformułować jako pytanie o prawdziwość zdania $\varphi \to w = v$. □

Analogicznie jak dla półgrup, rozważa się też *problem słów w grupach*. Tym razem sygnatura zawiera dodatkowo operację unarną $()^{-1}$ i mamy dodatkowe aksjomaty:

$$x^{-1} \cdot x = \varepsilon, \quad x \cdot x^{-1} = \varepsilon.$$

Problem słów dla grup też jest w ogólności nierozstrzygalny, pokazał to Nowikow w 1952 roku. Ale ten dowód jest dużo trudniejszy (problem był znany już 40 lat wcześniej).

Ćwiczenia

1. Udowodnić nierozstrzygalność problemu stopu dla zmodyfikowanych automatów dwulicznikowych, których dozwolone instrukcje są nastepujące:

   (a) $(q : c_i := c_j + 1;$ `goto` $p)$;

   (b) $(q : c_i := 0;$ `goto` $p)$;

   (c) $(q :$ `if` $c_i = c_j$ `then goto` p `else goto` $r)$;

   Wskazówka: Parę liczb $\langle a_0, a_1\rangle$ można reprezentować nie tylko przez $2^{a_0}3^{a_1}$ ale także przez każdą z liczb postaci $2^{a_0}3^{a_1}5^x$. Aby obniżyć a_0 o jeden, nie trzeba takiej liczby zmniejszać, wystarczy ją pomnożyć przez $5/2$.

2. Po co te kreski w dowodzie twierdzenia 9.3?

3. *Automat kolejkowy* dysponuje kolejką (słowem nad ustalonym alfabetem) na której może wykonywać takie operacje:

   - dodaj literę a na końcu kolejki;
   - usuń pierwszą literę z kolejki;
   - zmień stan w zależności od tego jaka litera znajduje się na początku kolejki.

   Udowodnić nierozstrzygalność problemu stopu dla automatów kolejkowych: *czy dany automat uruchomiony z pustą kolejką początkową osiągnie stan końcowy?*

4. Udowodnić nierozstrzygalność problemu stopu dla automatów z dwoma stosami, na których można przechowywać litery ze skończonego alfabetu. Automat może wykonywać operacje *push* i *pop* i zmieniać stan w zależności od tego, co znajduje się na wierzchołku pierwszego stosu.

5. Udowodnić rozstrzygalność problemu stopu dla automatów z jednym stosem.

6. Udowodnić, że dla dowolnego systemu Thuego T relacja $w \mathrel{\twoheadleftarrow\!\!\!\!\twoheadrightarrow_T} v$ zachodzi wtedy i tylko wtedy, gdy równość $w = v$ można wyprowadzić ze aksjomatów półgrupy rozszerzonych o aksjomaty równościowe wyznaczone przez reguły systemu T.

10 Układanie kafelków

Istnieje wiele wariantów problemu układania kafelków (ang.: *tiling problem*). Zwykle zakłada się, że dany jest skończony zbiór T *kafelków* i relacje $H, V \subseteq T \times T$ zgodności poziomej i pionowej. *Pokryciem* zbioru $A \subseteq \mathbb{Z} \times \mathbb{Z}$ nazwiemy taką funkcję $P : A \to T$, że:

$$\langle P(x,y), P(x+1,y)\rangle \in H, \text{ zawsze gdy } \langle x,y\rangle \in A \text{ i } \langle x+1,y\rangle \in A;$$

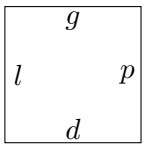
$$\langle P(x,y), P(x,y+1)\rangle \in V, \text{ zawsze gdy } \langle x,y\rangle \in A \text{ i } \langle x,y+1\rangle \in A.$$

Pokrycie może spełniać jakiś *warunek początkowy*, np. $P(0,0) = t_0$.

Jeśli $T \subseteq C^4$, gdzie C jest zbiorem *kolorów*, to mamy do czynienia z *dominem*. Relacje H, V definiujemy wtedy następująco:

$$\begin{aligned} \langle t, t'\rangle \in H &\quad\Leftrightarrow\quad p(t) = l(t');\\ \langle t, t'\rangle \in V &\quad\Leftrightarrow\quad g(t) = d(t'), \end{aligned}$$

gdzie $g(t)$, $p(t)$, $d(t)$, $l(t)$ oznaczają cztery współrzędne klocka t interpretowane jako kolory czterech boków kwadratu (w kolejności góra, prawo, dół, lewo).



Dwa najważniejsze przykłady problemu układania kafelków są takie:

A) Dany jest zbiór T i relacje H, V; czy istnieje pokrycie całej płaszczyzny $\mathbb{Z} \times \mathbb{Z}$?

B) Dany jest zbiór T i relacje H, V, oraz kafelek początkowy $t_0 \in T$; czy istnieje pokrycie P pierwszej ćwiartki $\mathbb{N} \times \mathbb{N}$ spełniające warunek początkowy $P(0,0) = t_0$?

Oba powyższe problemy są nierozstrzygalne. Dowód nierozstrzygalności problemu w wersji (A) jest jednak znacznie trudniejszy niż dla wersji (B). Dlatego zajmiemy się tym drugim.

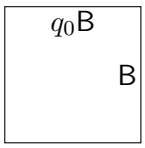
Kodowanie maszyny Turinga: Aby udowodnić nierozstrzygalność problemu (B) zredukujemy problem stopu dla deterministycznych maszyn Turinga do nieistnienia pokrycia. To jest, dla danej deterministycznej maszyny M skonstruujemy efektywnie domino T wraz z klockiem początkowym $t_0 \in T$ i udowodnimy taki fakt:

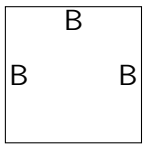
Lemat 10.7 *Następujące warunki są równoważne:*

1) *Maszyna M ma nieskończone obliczenie dla pustego słowa wejściowego.*
2) *Istnieje takie pokrycie P zbioru $\mathbb{N} \times \mathbb{N}$ klockami domina T, że $P(0,0) = t_0$.*

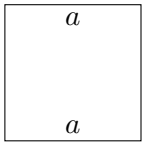
Zbiór kolorów domina T składa się z następujących elementów:
– symbole alfabetu maszyny M;
– kolory postaci qa, gdzie q jest stanem i a symbolem alfabetu M;
– kolory postaci $\overleftarrow{q}$ i $\overrightarrow{q}$, gdzie q jest stanem M;
– jeden kolor „neutralny” (tego koloru są na obrazkach krawędzie bez oznaczeń).

Teraz opiszemy klocki domina T. Zaczniemy od dwóch szczególnych klocków:

$t_0 =$ 

$t_1 =$ 

Powyżej q_0 to stan początkowy. Dla każdego symbolu a z alfabetu maszyny mamy klocek:

$t_a =$ 

Dalsze klocki reprezentują funkcję przejścia maszyny. Jeśli $\delta(q,a) = (b,p,+1)$, to dla dowolnego c z alfabetu maszyny mamy dwa klocki:[22]

[22] Jeśli $t^R_{qa} = t^R_{q'a'}$, to mamy wspólny „prawy” klocek dla dwóch różnych instrukcji. Ale to nic nie szkodzi.

$$t^L_{qa} = \boxed{\begin{array}{cc} b & \\ & \overrightarrow{p} \\ qa & \end{array}} \qquad t^R_{qa} = \boxed{\begin{array}{cc} & pc \\ \overrightarrow{p} & \\ & c \end{array}}$$

Analogicznie, dla $\delta(q,a) = (b,p,-1)$:[23]

$$t^L_{qa} = \boxed{\begin{array}{cc} pc & \\ & \overleftarrow{p} \\ c & \end{array}} \qquad t^R_{qa} = \boxed{\begin{array}{cc} & b \\ \overleftarrow{p} & \\ & qa \end{array}}$$

W przypadku $\delta(q,a) = (b,p,0)$ mamy tylko jeden klocek:

$$t_{qa} = \boxed{\begin{array}{c} pb \\ \\ qa \end{array}}$$

Dowód lematu 10.7 naszkicujemy w trudniejszym kierunku (2) $\Rightarrow$ (1). Załóżmy że pokrycie P spełnia warunek początkowy $P(0,0) = t_0$. Do klocka t_0 pasuje z prawej strony tylko klocek t_1, tj. musi być $P(1,0) = t_1$. Ale do klocka t_1 pasuje z prawej też tylko t_1, więc $P(n,0) = t_1$ dla wszystkich n. A zatem pierwsza „warstwa" naszego pokrycia wygląda tak:

$$\boxed{\begin{array}{cc} q_0\mathsf{B} & \\ & \mathsf{B} \end{array}}\boxed{\begin{array}{cc} \mathsf{B} & \\ \mathsf{B} & \mathsf{B} \end{array}}\boxed{\begin{array}{cc} \mathsf{B} & \\ \mathsf{B} & \mathsf{B} \end{array}}\boxed{\begin{array}{cc} \mathsf{B} & \\ \mathsf{B} & \mathsf{B} \end{array}}\cdots$$

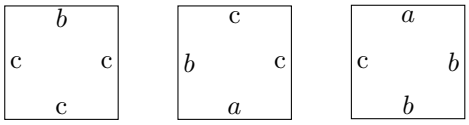
Z górnej krawędzi tej warstwy odczytamy ciąg $q_0\mathsf{BBBB}\ldots$, który stanowi zapis konfiguracji początkowej maszyny. Przypuśćmy teraz (indukcja), że ciąg kolorów na górnej krawędzi warstwy m jest postaci $a_1,\ldots,a_n,qa,b_1,\ldots b_k,\mathsf{B},\mathsf{B},\ldots$ i przedstawia konfigurację otrzymaną po m krokach obliczenia maszyny M. W szczególności, istnieje dokładnie jedno takie n, że klocek $P(n,m)$ ma na górnej krawędzi kolor postaci qa. Wtedy klocek $P(n,m+1)$ musi mieć qa na dolnej krawędzi, czyli musi być postaci t_{qa}, t^L_{qa} lub t^R_{qa}. W pierwszym przypadku, dla wszystkich pozycji $n' \neq n$ musimy mieć $P(n',m+1) = t_x$ dla odpowiedniego symbolu x. W pozostałych dwóch przypadkach z lewej lub z prawej strony klocka $P(n,m+1)$ stoi odpowiednio klocek t^R_{qa} lub t^L_{qa}, bo żaden inny nie pasuje. A inne klocki też muszą być postaci t_x. W ten sposób „przenosimy" informację z dolnej krawędzi warstwy $m+1$ do górnej krawędzi, zmieniając przy tym zapisaną konfigurację.

Pokazaliśmy przez indukcję, że poprawne pokrycie zbioru $\{0,\ldots,m\}\times\mathbb{N}$ oznacza, że maszyna wykonuje co najmniej $m+1$ kroków dla pustego słowa wejściowego. Skoro więc istnieje pokrycie całej pierwszej ćwiartki, to obliczenie musi być nieskończone. Istotnie, funkcja przejścia M nie jest określona dla stanu końcowego q_a, a więc nie ma klocka, który na dolnej krawędzi miałby kolor postaci $q_a a$.

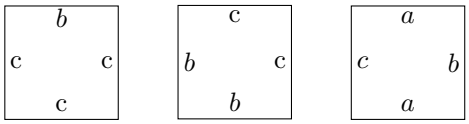
[23] Bez straty ogólności zakładamy, że maszyna nigdy nie wykonuje takiego kroku, gdy głowica znajduje się nad pierwszą klatką taśmy.

Ćwiczenia

1. Skonstruować pokrycie $\mathbb{Z} \times \mathbb{Z}$ takimi klockami:



2. Pokazać, że nie da się pokryć płaszczyzny takimi klockami:



3. Zmodyfikować definicję domina w ten sposób, aby można było kafelki obracać. Czy problemy (A) i (B) są nadal nierozstrzygalne?
4. Zmodyfikować definicję domina w ten sposób, aby zamiast kolorów zgodność klocków była zdeterminowana przez kształt krawędzi („puzzle"). Czy problemy (A) i (B) są nadal nierozstrzygalne? Czy odpowiedź zależy od dopuszczalności obracania klocków?
5. Pokazać, że dla dowolnych T, H, V pokrycie płaszczyzny istnieje wtedy i tylko wtedy, gdy istnieje pokrycie dowolnie dużego kwadratu.
6. Pokazać, że dopełnienia problemów (A) i (B) są rekurencyjnie przeliczalne.

11 Das Entscheidungsproblem

Wykorzystując problem układania kafelków pokażemy nierozstrzygalność klasycznej logiki pierwszego rzędu, a ściślej problemu decyzyjnego *„czy dana formuła jest tautologią?"*

W tym celu dla danego zbioru kafelków T z relacjami H, V i wyróżnionym $t_0 \in T$ skonstruujemy efektywnie formułę Φ, która jest spełnialna wtedy i tylko wtedy, gdy istnieje pokrycie P zbioru $\mathbb{N} \times \mathbb{N}$ kafelkami z T, spełniające warunek początkowy $P(0,0) = t_0$.

W ten sposób problem istnienia pokrycia sprowadzimy do problemu spełnialności, a problem nieistnienia pokrycia – do problemu prawdziwości.[24]

Sygnatura formuły Φ jest czysto relacyjna (nie ma symboli funkcyjnych) i składa się z:

– jednego jednoargumentowego symbolu relacyjnego Z;
– symboli relacyjnych dwuargumentowych R i S;
– po jednym dwuargumentowym symbolu K_t, dla każdego $t \in T$.

Sama formuła Φ jest koniunkcją postaci $\Phi = \varphi_0 \wedge \varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \varphi_4 \wedge \psi_0 \wedge \psi_1 \wedge \psi_2 \wedge \vartheta_H \wedge \vartheta_V$.

[24] Formuła Φ nie jest spełnialna wtedy i tylko wtedy, gdy $\neg\Phi$ jest tautologią.

Początkowe człony koniunkcji są takie:

$$\begin{aligned}
\varphi_0 &= \forall x\, R(x,x);\\
\varphi_1 &= \forall xyz(R(x,y) \wedge R(y,z) \to R(x,z);\\
\varphi_2 &= \forall xy(S(x,y) \to R(x,y));\\
\varphi_3 &= \forall x \exists y(S(x,y) \wedge \neg R(y,x));\\
\varphi_4 &= \exists x Z(x).
\end{aligned}$$

Lemat 11.8 *Jeśli model $\mathcal{A}$ spełnia koniunkcję $\varphi_0 \wedge \varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \varphi_4$, to istnieje taki nieskończony ciąg $(a_n)_{n\in\mathbb{N}}$ różnych elementów $\mathcal{A}$, że $a_0 \in Z^{\mathcal{A}}$ oraz $\langle a_n, a_{n+1}\rangle \in S^{\mathcal{A}}$, dla wszystkich $n \in \mathbb{N}$.*

Dowód: Formuły φ_0 i φ_1 wyrażają zwrotność i przechodniość relacji $R^{\mathcal{A}}$. Formuła φ_2 wymusza zawieranie $S^{\mathcal{A}} \subseteq R^{\mathcal{A}}$, a formuła φ_4 gwarantuje niepustość zbioru $Z^{\mathcal{A}}$. Jako a_0 wybieramy dowolny element tego zbioru. Dalej postępujemy przez indukcję: jeśli elementy a_i, dla $i \leq n$, są już określone, to a_{n+1} wybieramy tak, żeby $\langle a_n, a_{n+1}\rangle \in S^{\mathcal{A}} - R^{\mathcal{A}}$. Jest to możliwe, bo $\mathcal{A} \models \varphi_3$. Pozostaje jeszcze sprawdzić, że $a_{n+1} \neq a_i$ dla $i \leq n$. Ale jeśli $i \leq n$, to para $\langle a_0, a_i\rangle$ należy do domknięcia przechodniego relacji $S^{\mathcal{A}}$, a więc także do relacji $R^{\mathcal{A}}$. □

Kolej na dwie następne składowe formuły Φ. Ich sens wyraża oczywisty lemat 11.9.

$$\begin{aligned}
\psi_0 &= \forall xy(Z(x) \wedge Z(y) \to K_{t_0}(x,y));\\
\psi_1 &= \forall xy \bigvee\{K_t(x,y) \mid t \in T\};\\
\psi_2 &= \bigwedge\{\forall xy(K_t(x,y) \to \neg K_s(x,y)) \mid t,s \in T \;\&\; t \neq s\}.
\end{aligned}$$

Lemat 11.9 *Jeśli model $\mathcal{A}$ spełnia koniunkcję $\psi_0 \wedge \psi_1 \wedge \psi_2$, to każda para $\langle a,b\rangle \in \mathcal{A}^2$ należy do dokładnie jednej z relacji $K_t^{\mathcal{A}}$. Przy tym jeśli $a \in Z^{\mathcal{A}}$, to $\langle a,a\rangle \in K_{t_0}^{\mathcal{A}}$.*

Pozostają nam jeszcze dwie formuły składowe, które zapewniają zgodność w pionie i w poziomie.

$$\begin{aligned}
\vartheta_H &= \forall xyz(S(xz) \to \bigvee\{K_t(x,y) \wedge K_s(z,y) \mid \langle t,s\rangle \in H\});\\
\vartheta_V &= \forall xyz(S(yz) \to \bigvee\{K_t(x,y) \wedge K_s(x,z) \mid \langle t,s\rangle \in V\}).
\end{aligned}$$

Lemat 11.10 (główny) *Formuła Φ jest spełnialna wtedy i tylko wtedy, gdy istnieje pokrycie P zbioru $\mathbb{N}\times\mathbb{N}$ kafelkami z T, spełniające warunek początkowy $P(0,0) = t_0$.*

Dowód: Łatwiejsza jest implikacja ($\Leftarrow$). Jeśli P jest pokryciem, to można zdefiniować model $\mathcal{A}$ przyjmując jako dziedzinę zbiór $\mathbb{N}$. Symbol S interpretujemy jako relację następnika, symbol R jako $\leq$, a Z jako $\{0\}$. Do tego oczywiście $K_t^{\mathcal{A}} = \{\langle x,y\rangle \mid P(x,y) = t\}$. Sprawdzenie, że $\mathcal{A} \models \Phi$ jest rutynowe.

Trudniejsza implikacja ($\Rightarrow$) wymaga pokazania, że każdy model $\mathcal{A}$ formuły Φ indukuje pokrycie. Niech $(a_n)_{n\in\mathbb{N}}$ będzie ciągiem elementów $\mathcal{A}$, o którym mowa w lemacie 11.8. (Takich ciągów może być wiele, ale każdy jest równie dobry.) Lemat 11.9 pozwala dla $i,j \in \mathbb{N}$ zdefiniować $P(i,j)$ jako jedyny element $t \in T$ o własności $\langle a_i, a_j\rangle \in K_t^{\mathcal{A}}$. Zauważmy, że $P(0,0) = t_0$.

Tak określona funkcja $P : \mathbb{N} \times \mathbb{N} \to T$ jest pokryciem, bo $\mathcal{A} \models \vartheta_H \wedge \vartheta_V$. Istotnie, weźmy dowolne liczby $i, j \in \mathbb{N}$. Ponieważ $\langle a_i, a_{i+1} \rangle \in S^{\mathcal{A}}$ oraz $\mathcal{A} \models \vartheta_H \wedge \vartheta_H$, więc spełniona musi być alternatywa w konkluzji formuły ϑ_H, tj. dla pewnej pary $\langle t, s \rangle \in H$ zachodzi $\langle a_i, a_j \rangle \in K_t^{\mathcal{A}}$ oraz $\langle a_{i+1}, a_j \rangle \in K_s^{\mathcal{A}}$. Znaczy to tyle samo co $\langle P(i,j), P(i+1,j) \rangle \in H$. Zgodność w pionie wynika w podobny sposób ze spełnienia formuły ϑ_V. □

Ćwiczenia

1. Sygnatura formuły Φ zależy od liczby kafelków w zbiorze T. Pokazać, że logika pierwszego rzędu jest nierozstrzygalna dla ustalonej sygnatury. (Wskazówka: użyć trójargumentowego symbolu relacyjnego.)
2. Kodując kilka relacji za pomocą jednej pokazać, że logika pierwszego rzędu jest nierozstrzygalna dla sygnatury, w której występuje tylko jeden symbol relacyjny i to dwuargumentowy.

12 Problem wnioskowania w rachunku zdań

Przez *problem wnioskowania w rachunku zdań* rozumiemy tu następujący problem decyzyjny:

Czy daną formułę rachunku zdań można wywnioskować z danych schematów aksjomatów?

Ściślej: dla zbioru schematów aksjomatów zdaniowych Γ i formuły φ, pytamy, czy φ ma dowód (w sensie Hilberta) z aksjomatów Γ. Przyjmujemy, że jedyną regułą wnioskowania jest modus ponens, a za aksjomat uważamy każdą formułę, którą otrzymamy przez podstawienie z jakiegoś schematu w Γ.

Twierdzenie 12.11 (Linial i Post, 1949)

Problem wnioskowania w rachunku zdań jest nierozstrzygalny.

Twierdzenie 12.11 udowodnimy dla formuł zbudowanych wyłącznie z implikacji. W tym celu zredukujemy problem stopu dla automatów dwulicznikowych do problemu wnioskowania w rachunku zdań. Zaczniemy od wprowadzenia trzech skrótów:

$$\iota(\alpha) = \alpha \to \alpha, \qquad \delta(\alpha) = \alpha \to \alpha \to \alpha, \qquad \sigma(\alpha) = (\alpha \to \alpha) \to \alpha.$$

Kodem liczby n nazwiemy dowolną formułę postaci: $\iota(\alpha_1) \to \cdots \to \iota(\alpha_n) \to \delta(\beta)$, czyli postaci

$$(\alpha_1 \to \alpha_1) \to \cdots \to (\alpha_n \to \alpha_n) \to (\beta \to \beta \to \beta).$$

Jeśli przy tym $\alpha_i = \alpha$ dla wszystkich i, to taki kod oznaczymy przez $\lambda_n(\alpha)$.

Formuła otrzymana przez podstawienie z kodu liczby n też jest kodem liczby n, zatem z poniższego lematu wynika, że kody liczb nie są unifikowalne z formułami postaci $\iota(\alpha)$.

Lemat 12.12 *Kody liczb nie są postaci $\iota(\alpha)$, dla żadnego α.*

Dowód: Indukcja ze względu na n. Kod zera ma postać $\beta \to \beta \to \beta$. Gdyby zachodziła równość $\beta \to \beta \to \beta = \alpha \to \alpha$, to mielibyśmy $\beta = \alpha = \beta \to \beta$, skąd wynika $\beta = \beta \to \beta$.

Kod liczby $n+1$ ma postać $\iota(\alpha_1) \to \tau$, gdzie τ jest kodem liczby n. Równość $\iota(\alpha_1) \to \tau = \iota(\alpha)$ implikuje $\iota(\alpha_1) = \alpha = \tau$, co jest sprzeczne z założeniem indukcyjnym o n. □

Lemat 12.13 *Kody różnych liczb są różne.*

Dowód: Niech $n \neq 0$ i niech τ będzie kodem n. Wtedy $\tau = \iota(\alpha) \to \rho$, gdzie ρ jest kodem liczby $n-1$. Gdyby $\tau = \beta \to \beta \to \beta$, to $\rho = \beta \to \beta$, co jest niemożliwe z lematu 12.12. Zatem kod niezerowej liczby nie może być jednocześnie kodem zera.

Jeśli liczby $n, m > 0$ mają takie same kody, to także $n-1$ i $m-1$ mają takie same kody, możemy więc postępować przez indukcję ze względu na $\min(m, n)$. □

Kodowanie automatu: Ustalmy automat dwulicznikowy $\mathcal{A}$. Bez straty ogólności załóżmy, że automat wykonuje operację $c_i := c_i - 1$ tylko wtedy, gdy wartość licznika c_i jest niezerowa. Wygodnie jest przyjąć taką konwencję, że stany naszego automatu to liczby od 1 do F, przy czym 1 jest stanem początkowym, a F jest stanem akceptującym.

Kodem konfiguracji $\mathcal{C} = \langle q, n_1, n_2 \rangle$ niech będzie dowolna formuła postaci

$$\lambda_q(\alpha) \to \tau_1 \to \tau_2 \to \sigma(\xi),$$

gdzie τ_1 i τ_2 są odpowiednio kodami liczb n_1 i n_2. Oczywiście $\lambda_q(\alpha)$ koduje stan.

Lemat 12.14 *Kody konfiguracji nie są żadnej z postaci* $\beta \to \gamma \to \sigma(\xi)$, $\gamma \to \sigma(\xi)$ *ani* $\sigma(\xi)$.

Dowód: W pierwszym przypadku drugi licznik unifikuje się z $\xi \to \xi$, w drugim to samo dotyczy pierwszego licznika, a w trzecim kodu stanu. Stosujemy lemat 12.12. □

Aksjomaty: Każdej instrukcji automatu odpowiada jeden lub dwa schematy aksjomatów:

1. Dla instrukcji $(q : c_1 := c_1 + 1;\ \mathsf{goto}\ p)$ przyjmujemy schemat aksjomatu:
$$[\lambda_p(\alpha) \to (\iota(\varepsilon) \to \beta) \to \gamma \to \sigma(\xi)] \to [\lambda_q(\alpha) \to \beta \to \gamma \to \sigma(\xi)].$$
2. Dla $(q : c_1 := c_1 - 1;\ \mathsf{goto}\ p)$ też mamy jeden schemat aksjomatu:
$$[\lambda_p(\alpha) \to \beta \to \gamma \to \sigma(\xi)] \to [\lambda_q(\alpha) \to (\iota(\varepsilon) \to \beta) \to \gamma \to \sigma(\xi)].$$
3. Ale dla testu $(q : \mathsf{if}\ c_1 = 0\ \mathsf{then\ goto}\ p\ \mathsf{else\ goto}\ r)$ są dwa schematy:
$$[\lambda_p(\alpha) \to \delta(\beta) \to \gamma \to \sigma(\xi)] \to [\lambda_q(\alpha) \to \delta(\beta) \to \gamma \to \sigma(\xi)],$$
$$[\lambda_r(\alpha) \to (\iota(\varepsilon) \to \beta) \to \gamma \to \sigma(\xi)] \to [\lambda_q(\alpha) \to (\iota(\varepsilon) \to \beta) \to \gamma \to \sigma(\xi)].$$
4. Dla instrukcji dotyczących c_2 aksjomaty są analogiczne.
5. Na koniec jeszcze schemat $\lambda_F(\alpha) \to \beta \to \gamma \to \sigma(\xi)$.

Oznaczmy przez Δ zbiór wszystkich instancji powyższych schematów (tj. zbiór wszystkich formuł, które można z nich otrzymać w drodze podstawienia. Oto główny lemat:

Lemat 12.15 *Dla dowolnej konfiguracji $\mathcal{C} = \langle q, n_1, n_2\rangle$ i dowolnej formuły φ, która jest kodem $\mathcal{C}$, następujące warunki są równoważne:*

1. *Automat $\mathcal{A}$ akceptuje konfigurację $\mathcal{C}$ (tj. $\mathcal{C} \twoheadrightarrow_{\mathcal{A}} \mathcal{C}_F$, gdzie $\mathcal{C}_F$ końcowa).*
2. *Formuła φ ma dowód z aksjomatów Δ.*

Dowód: $(1 \Rightarrow 2)$ Indukcja ze względu na długość obliczenia. Załóżmy, że automat $\mathcal{A}$ akceptuje konfigurację $\mathcal{C} = \langle q, n_1, n_2\rangle$, oraz ψ_1 i ψ_2 są kodami n_1 i n_2. Mamy pokazać, że formuła $\lambda_q(\alpha) \to \psi_1 \to \psi_2 \to \sigma(\xi)$ ma dowód. Jeśli $\mathcal{C}$ jest konfiguracją końcową, to wystarczy przywołać ostatni aksjomat na liście powyżej.

Przypuśćmy, że $\mathcal{C} \to \mathcal{C}' = \langle p, n_1, n_2 + 1\rangle$ wskutek wykonania instrukcji $q : c_2 := c_2 + 1;$ goto p. Wtedy mamy taki aksjomat:

$$[\lambda_p(\alpha){\to}\psi_1{\to}(\iota(\varepsilon){\to}\psi_2){\to}\sigma(\xi)] \to [\lambda_q(\alpha){\to}\psi_1{\to}\psi_2{\to}\sigma(\xi)].$$

Formuła $\lambda_p(\alpha){\to}\psi_1{\to}(\iota(\varepsilon){\to}\psi_2){\to}\sigma(\xi)$ jest kodem $\mathcal{C}'$, więc ma dowód z założenia indukcyjnego. Zatem $\lambda_q(\alpha) \to \psi_1 \to \psi_2 \to \sigma(\xi)$ otrzymamy przez odrywanie. Dla pozostałych instrukcji postępowanie jest podobne.

$(2 \Rightarrow 1)$ Indukcja ze względu na długość dowodu. Najkrótszy dowód polega na przywołaniu aksjomatu. Jeśli kod konfiguracji jest instancją schematu (5), to musi to być konfiguracja końcowa (dla $q \neq F$ formuły $\lambda_q(\alpha)$ i $\lambda_F(\alpha')$ nie są unifikowalne). Niemożliwe zaś, aby kod był instancją innego schematu, wówczas bowiem formuła $\lambda_q(\alpha)$ (pierwsza przesłanka w kodzie) unifikuje się z formułą o kształcie $\lambda_p(\alpha') \to \beta \to \gamma \to \sigma(\xi)$. Ponieważ $q \geq 1$, więc pierwszą przesłanką w $\lambda_q(\alpha)$ jest $\alpha \to \alpha$. Mamy więc równanie $\alpha \to \alpha = \lambda_p(\alpha')$, sprzeczne z lematem 12.12.

A zatem dowód polega na (być może kilkakrotnym) zastosowaniu reguły modus ponens do jednego z aksjomatów. Zauważmy jednak, że kod konfiguracji nie może być otrzymany przez odrywanie ze schematu postaci (5), ani przez wielokrotne odrywanie ze schematu postaci (1–4). Wyklucza to lemat 12.14.

Zatem mamy jednorazowe zastosowanie modus ponens do aksjomatu, np. takiego:

$$[\lambda_p(\alpha){\to}\delta(\beta){\to}\gamma{\to}\sigma(\xi)] \to [\lambda_q(\alpha){\to}\delta(\beta){\to}\gamma{\to}\sigma(\xi)].$$

Wtedy $q \neq F$, a pierwszy licznik w konfiguracji $\mathcal{C}$ jest zerem, bo tylko zero ma kod $\delta(\beta)$. Zatem test na zero jest poprawny i $\mathcal{C} = \langle q, 0, n_2\rangle \to_{\mathcal{A}} \mathcal{C}' = \langle p, 0, n_2\rangle$. Ponadto formuła $\lambda_p(\alpha) \to \delta(\beta) \to \gamma \to \sigma(\xi)$ ma krótszy dowód i jest kodem konfiguracji $\mathcal{C}'$. Z założenia indukcyjnego automat akceptuje $\mathcal{C}'$ a więc i $\mathcal{C}$. □

Dowód twierdzenia 12.11: Z lematu 12.15 otrzymujemy równoważność: automat akceptuje konfigurację początkową wtedy i tylko wtedy, gdy jej kod ma dowód z aksjomatów Δ. W ten sposób problem stopu zredukowaliśmy do wnioskowania w rachunku zdań. □

Ćwiczenia

1. Co się zmieni jeśli instrukcja ($q : c_i := c_i - 1$; goto p) może być wykonana także dla $c_i = 0$ (w zależności od semantyki tej instrukcji)?
2. Pokazać, że problem wnioskowania staje się rozstrzygalny (w PTIME) jeśli za aksjomaty uznajemy tylko elementy skończonego zbioru formuł Δ (a nie wszystkie ich instancje).

13 Totalność, mortalność i terminalność

Przez *problem totalności* dla maszyn Turinga rozumiemy pytanie:

Dana deterministyczna maszyna $\mathcal{M}$. Czy $\mathcal{M}$ zatrzymuje się dla dowolnego słowa?

Z faktu 7.1 natychmiast wynika, że problem totalności nie jest rekurencyjnie przeliczalny ani nie jest dopełnieniem problemu rekurencyjnie przeliczalnego.

Mówimy, że deterministyczna maszyna Turinga $\mathcal{M}$ jest *terminalna* jeżeli dla dowolnej konfiguracji $\mathcal{C}$ (niekoniecznie początkowej), obliczenie maszyny $\mathcal{M}$ rozpoczynające się od $\mathcal{C}$ jest skończone.

Twierdzenie 13.1 *Problem terminalności dla maszyn Turinga (czy dana maszyna jest terminalna?) nie jest rekurencyjnie przeliczalny ani nie jest dopełnieniem problemu rekurencyjnie przeliczalnego.*

Dowód: Dowód opiera się na redukcji problemu totalności do problemu terminalności. Dla danej maszyny $\mathcal{M}$ konstruujemy taką maszynę $\mathcal{N}$, żeby zachodził warunek:

$\mathcal{M}$ jest totalna wtedy i tylko wtedy, gdy $\mathcal{N}$ jest terminalna.

Oczywiście maszyna $\mathcal{N}$ musi naśladować obliczenia maszyny $\mathcal{M}$. Trudność przy tej redukcji polega na tym, że konfiguracja, w której uruchamiać będziemy maszynę $\mathcal{N}$, może być zupełnie dowolna, w szczególności może nie odpowiadać żadnej konfiguracji $\mathcal{M}$ osiągalnej z jakiejkolwiek konfiguracji początkowej. Działanie maszyny $\mathcal{N}$ jest następujące:

Maszyna $\mathcal{N}$ wyobraża sobie, że jej taśma jest podzielona na trzy części. Pierwsza przechowuje słowo wejściowe, druga przeznaczona jest na binarny licznik, a trzecia stanowi obszar roboczy. Maszyna $\mathcal{N}$ w obszarze roboczym symuluje zachowanie maszyny $\mathcal{M}$, zwiększając licznik o 1 po każdym kroku. W razie wykrycia jakiejkolwiek niezgodności pomiędzy swoimi oczekiwaniami i rzeczywistymi danymi zapisanymi na taśmie, maszyna $\mathcal{N}$ natychmiast się zatrzymuje. Kiedy wyczerpie się miejsce na licznik, maszyna dodaje do niego jedną klatkę, kopiuje słowo wejściowe do obszaru roboczego i zaczyna symulację od początku. W ten sposób maszyna $\mathcal{N}$ po skończonej liczbie kroków musi albo się zatrzymać albo rozpocząć prawidłową symulację maszyny $\mathcal{M}$, zaczynającą się od pewnej konfiguracji początkowej. Szczegóły konstrukcji pozostawiamy jako ćwiczenie. □

Jeśli uważnie przyjrzymy się temu dowodowi, to zobaczymy, że istotnie wykorzystuje on założenie o skończoności konfiguracji maszyny. Obszar roboczy jest skończony i można zawsze odszukać jego lewy koniec (albo pierwszy blank — można zakładać, że konfiguracje

maszyny $\mathcal{M}$ nigdy nie zawierają blanków „wewnętrznych"). Jeśli założyć, że konfiguracje maszyny Turinga mogą być nieskończone, to mówimy, że maszyna jest *mortalna*, gdy każde obliczenie maszyny $\mathcal{M}$ rozpoczynające się od dowolnej (także nieskończonej) konfiguracji musi zawsze być skończone.

Podobną własnością jest *ograniczoność* maszyny Turinga. Ma ona miejsce wtedy, gdy istnieje taka stała k, że liczba różnych konfiguracji przyjmowanych przez maszynę w dowolnym ciągu kolejnych kroków jest zawsze mniejsza lub równa k. Zauważmy, że maszyna ograniczona nie musi być mortalna.

Fakt 13.2 *Jeśli maszyna $\mathcal{M}$ jest mortalna, to istnieje taka stała k, że maszyna $\mathcal{M}$, uruchomiona w dowolnej (być może nieskończonej) konfiguracji, zatrzymuje się po co najwyżej k krokach.*

Dowód: Ćwiczenie. Wskazówka: użyć lematu Königa. □

Wniosek 13.3 *Problem mortalności (czy dana maszyna jest mortalna?) jest rekurencyjnie przeliczalny.*

Wniosek 13.4 *Każda maszyna mortalna jest ograniczona.*

Twierdzenie 13.5 (Philip K. Hooper) *Problem mortalności jest nierozstrzygalny.*

Pozostała część tego rozdziału to dowód (a właściwie szkic dowodu) twierdzenia Hoopera. Dowód jest w większości po angielsku, bo taki gotowy tekst istnieje, a lepszy taki, niż żaden.

The goal is to reduce the halting problem for two-counter automata (known to be undecidable) to the mortality problem. We start with an arbitrary two-counter automaton M.

Lemat 13.6 *One can assume that*

1. *M may execute at most two tests for zero in a row;*
2. *For each n there is a k, such that if M makes k moves when started on any ID, then the value of the second counter must exceed n at least once during these k moves.*

Dowód: An ID of the form $\langle q, c_1, c_2\rangle$ is represented by $\langle q, 0, 2^{c_1}3^{c_2}5^p\rangle$, where $p \geq 0$. A single move of the automaton is simulated in an obvious way, but in addition, after completing each step, the second counter is multiplied by 5. The construction is left to the reader. □

The next step is to construct a Turing Machine T which simulates the behaviour of M. This construction is quite standard, but essential for the considerations to follow, and thus we describe it in some detail.

The machine T has one tape, infinite in both directions. The set Q of states of T includes the set Q_M of states of M, called *main* states of T. The tape alphabet is $\{0, 1\}$,

with 0 used as the blank symbol. An ID of M of the form $\langle q, c_1, c_2\rangle$ will be represented as[25] $\langle 10^{c_1+1}, q, m, 10^{c_2+1}1\rangle$, and after simulation of a move of M resulting in $\langle p, c_1', c_2'\rangle$, the machine T arrives at $\langle 10^{c_1'+1}, q, m-c_1+c_1', 10^{c_2'+1}1\rangle$, i.e., the position of the leftmost "1" does not change. This allows us in a later construction to run a copy of T, using the part of tape to the left to store additional information.

The way T simulates M is as follows. To execute an operation on a counter, T performs the following actions:

- moves the middle "1" by one cell to the left or to the right, if necessary;
- moves the head to the rightmost "1", and adjusts its position (always by one cell to the right or to the left);
- returns to the middle "1".

To execute a test for zero, T moves its head two cells to the left or to the right and returns to the initial position. We assume that T halts if it discovers any kind of inconsistency, e.g., when there is no "0" between the middle and the rightmost 1's.

The machine T is obviously unbounded. The reason is that there are states q of T (called *right search* states), such that whenever T encounters "0" in state q, it moves the head right, and remains in state q. Similarly, there are *left search* states with the dual property. The sets of right search states and left search states are denoted Q_R and Q_L, respectively. The initial state is denoted by q_0.

Observe that, if T starts in an arbitrary ID, then in at most 10 steps it must enter a (left or right) search state. This is due to (1) in Lemma 13.6.

The crucial part of Hooper's proof is to construct another machine T^* which will simulate T without actually doing any unbounded search. For this we first define a machine $\overline{T}$, which is a "mirror copy" of T, with all its left moves replaced by right moves, and conversely. That is, the set of states of $\overline{T}$ is $\overline{Q} = \{\overline{q} \mid q \text{ is a state of } T\}$. Whenever T in a state q and reading a symbol a, prints b and moves right, changing state to p, then $\overline{T}$ in state $\overline{q}$, when reading a, will print b, move *left* and change state to $\overline{p}$.

The machine *T**

One may think of T^* as a program consisting of two recursive procedures P and $\overline{P}$, simulating the behaviour of T and $\overline{T}$, respectively. Each of these procedures may call itself or the other one. (This recursion will be used as a main trick in simulating the searching process.) An activation of P will be represented on the tape as a word of the form $1^x 0^{c_1+1} 10^{c_2+1} 1$, where $x \geq 2$ identifies the context in which the activation has been created. A further call to P will result in locating another word $1^y 0101$ (representing the initial ID of T) inside one of the segments of 0's between the 1's. Similarly, an activation of $\overline{P}$ will be represented by a word of the form $10^{c_2+1} 10^{c_1+1} 1^x$.

[25] Przez $\langle w, q, m, v\rangle$ oznaczamy tu konfigurację maszyny, która znajduje się w stanie q, ma głowicę umieszczoną w pozycji $m \in \mathbb{Z}$, gdzie znajduje się pierwsza litera słowa v zapisanego na taśmie po ostatniej literze słowa w. Zakładamy, że to, co znajduje się na lewo od w i na prawo od v jest dla nas nieistotne.

Let us describe the behaviour of T^* more precisely. The set S of states of T^* includes $Q \cup \overline{Q}$, and we distinguish the following particular subsets of S:

$S_M = Q_M \cup \overline{Q}_M$, the *main* states;

$S_R = Q_R \cup \overline{Q}_L \cup \{e_R^1, e_R^2\}$, the *right search* states;

$S_L = Q_L \cup \overline{Q}_R \cup \{e_L^1, e_L^2\}$, the *left search* states.

The new states e_R^1, e_R^2 and e_L^1, e_L^2 are called, respectively, right and left *erase* states. We assume that all the right and left search states are identified by integers greater than 1.

Assume now that T^* is in a state $q \in S_R$, numbered x, the head is positioned at the m-th cell containing a "0". Then T^* performs the following action:

- checks whether the nearest "1" to the right is at most $x + 4$ cells away;
- if so, it moves the head to the nearest "1" and enters q again;
- if not, it prints "01^x0101" at the cells $m, m+1, \ldots, m+x+4$, moves the head to the $(m+x+2)$-th cell, and changes the state to q_0. (This creates a new activation of P.)

Note that to perform the above, the machine uses $\mathcal{O}(x)$ brand new states, for each $q \in S_R$. If $q \in S_L$, then T^* behaves in a dual way, in particular, in step (3) above, it prints 10101^x0 at the cells $m-4-x, \ldots, m$, and enters state $\overline{q}_0$, at the $(m-x-2)$-th cell.

Now let the current state be $q \in S_R - \{e_R^1, e_R^2\}$, and let the symbol encountered on (m-th cell of) the tape be "1". (This represents a successful search.) Then T^* behaves as follows:

- checks whether the next symbol to the right is "0";
- if so, it executes the appropriate move of T or of $\overline{T}$ (determined by the transition function of T or $\overline{T}$ for q and "1");
- otherwise, it erases the "1" at the m-th cell, and enters the state e_L^1. (If "1" has been encountered at the $(m+1)$-st cell, it must be that the current activation of P has exhausted all the available space.)

The behaviour of T^* in a state $q \in S_L - \{e_L^1, e_L^2\}$ is defined again in a dual way.

If the machine encounters "1", when in the state e_L^1, it simply erases it, and enters e_L^2. When "1" (in the m-th cell) is encountered in state e_L^2, it should be the case that a whole activation of P has been erased except the initial 01^x. Using brand new states (no more than the largest possible x), the machine erases the 1's and enters the right search state identified by x at the cell $(m-x+1)$. Note that the erased activation of P has been created when the tape head was positioned at the "0" preceding "1^x". Thus, after completing one activation of P, the tape head moved one step to the right. Of course, the behaviour of the right erase states is dual.

We assume that T^* halts whenever it enters a final state of T or $\overline{T}$, or if it discovers any inconsistency.

Lemat 13.7 *Suppose that T^* is started at $\langle w, q, m, 0^n 1 v\rangle$, and that $q \in S_R$. Then, after some number of steps, T^* will either halt or will reach $\langle w 0^n, q, m+n, 1v\rangle$. That is, T^* is able to perform a successful right search. Similarly for left search: if T^* starts at $\langle w 1 0^{n-1}, q, m, 0v\rangle$, and $q \in S_L$ then it will either halt or reach $\langle w, q, m-n, 10^n 1 v\rangle$.*

Dowód: The proof goes by induction on n (we prove the claim in parallel for all search states). If q is numbered by x and $n \leq 4 + x$ then the claim is obvious. Otherwise, a new activation of P or $\overline{P}$ is created. The space available for this activation consists of $n - x$ cells, thus no search will be required on a distance longer than that. By the induction hypothesis, we may assume that each search was successful, and that our procedure activation is eventually erased. Thus, we arrive at $\langle w 0, q, m+1, 0^{n-1} v\rangle$ in the right search case, or at $\langle w 1 0^{n-2}, q, m-1, 00v\rangle$ otherwise, and we can use the induction hypothesis again. □

Clearly, the machine T^* has been constructed in an effective way from the two-counter automaton M. Thus, to complete the proof of Hooper's theorem, it remains to show:

Lemat 13.8 *M halts on the input $(0,0)$ iff T^* is mortal.*

Dowód: The "if" part is easy: assume that M does not halt. Then, by Lemma 13.6, the size of ID's of T^* obtained from $\langle 10, q_0, m, 101\rangle$, must grow infinitely, whence T^* must be unbounded and therefore immortal.

Proving the "only if" part is more delicate. Assume that M halts, and let K_1 be the space needed for T^* to complete a full simulation of the behaviour of M on the input $(0,0)$.

Oczywiście udana symulacja automatu M musi doprowadzić do zatrzymania maszyny T^*. A taka symulacja będzie miała miejsce zawsze, gdy maszyna T^* znajdzie się w stanie $s \in S_R$, mając na prawo co najmniej K_1+K_2 zer, gdzie K_2 jest ograniczeniem na długość ciągu jedynek kodującego stan s. Wystarczy więc pokazać, że maszyna, prędzej czy później musi wykonać przeszukiwanie w prawo (lub w lewo) na odległość co najmniej $K_1 + K_2$. Maszyna T^* musi oczywiście po skończonej liczbie kroków wejść w stan przeszukujący (lemat 13.6(1)) i wykonać przeszukiwanie na jakiś, być może krótki, dystans.

We shall show that if T^* performs a successful search at a distance n, then after a constant number of steps it must either halt or enter a search at a distance greater than n. Suppose we start in $\langle w, q, m, 0^n 1 v\rangle$, where $q \in S_R$. After reaching "1" at the $(m+n)$-th cell, the machine must enter an ID of one of the following forms (assuming it does not just halt):

1. $\langle w 0^n, p, m+n, 01v'\rangle$; here, the rightmost "1" has been moved one more step to the right, and p is a left search state;
2. $\langle w 0^{n-2}, p, m+n-1, 010v\rangle$, with $p \in S_L$; if the rightmost "1" has been moved one step to the left;
3. $\langle w 0^n, e^1_L, m+n, 0v\rangle$; here, a "1" was found in the $(m+n+1)$-st cell;
4. $\langle w 0^n 0^{x-1}, p, m+n+x, 0v'\rangle$ with $p \in S_L$; this happens when q is e^2_R;
5. $\langle w 0^{n+1}, e^2_R, m+n+1, v\rangle$; this happens when q is e^1_R;

In all cases except (2), it is clear that the next left search is at a distance at least $n+1$. The same holds also for case (2), if the last two symbols in w are 0's. Otherwise, after completing the left search, we enter an ID either of the form $\langle w', s, m-1, 10^{n-1}10v\rangle$ or of the form $\langle w', s, m-2, 10^n10v\rangle$, where $s \in S_M$. The machine then simulates the behaviour of M with the second counter set to $n-1$, or n, respectively. Assuming it will not halt, then, by Lemma 13.6(2), there is a constant K_3 such that after K_3 steps it must increase c_2 to at least $n+1$, i.e, a left search at a distance $n+1$ will eventually be executed.

A zatem, predzej czy później maszyna albo się zatrzyma, albo rozpocznie przeszukiwanie na odległość co najmniej $K_1 + K_2$. Ale to przeszukiwanie doprowadzi do pełnej symulacji akceptującego obliczenia automatu M i w konsekwencji do zatrzymania maszyny T^*. □

Ćwiczenia

1. Uzupełnić dowód twierdzenia 13.1. W szczególności wyjaśnić w jaki sposób maszyna $\mathcal{N}$ może rozpoznać lewy koniec taśmy. (Przy naszej definicji z rozdziału 1, maszyna mogłaby się zapętlić, próbując przesunąć głowicę poza koniec taśmy.)
2. Udowodnić fakt 13.2 oraz wnioski 13.3 i 13.4.
3. Wywnioskować nierozstrzygalność problemu ograniczoności
   (a) z dowodu twierdzenia 13.5;
   (b) z treści twierdzenia 13.5.

14 Semi-unifikacja

Ordinary (first-order) unification is solving equations between first-order terms. Semi-unification is solving inequalities. Let Σ be a first-order signature, and let t, s be two terms over Σ. We say that an inequality $t \leq s$ is *true* (holds in the term algebra) iff s is a substitution instance of t, i.e., there exists a substitution R with $R(t) = s$. A substitution S is a *solution* of the inequality $t \leq s$ iff $S(t) \leq S(s)$ is true.

Przykład 14.9

1. The inequality $f(x, y) \leq f(f(y, z), f(z, z))$ is true. The identity substitution is a solution of it. But not every substitution is a solution, take e.g., $S(x) = S(y) = x$ and $S(z) = z$.
2. The inequality $f(f(x, y), z) \leq f(z, f(x, y))$ is not true, but it has a solution S such that $S(z) = f(x_1, y_1)$ and $S(v) = v$, for all other variables v.
3. The inequality $f(f(x, y), z) \leq f(x, f(y, z))$ is not true and has no solution. Indeed, if S were a solution then, for some substitution R, we would have $R(S(f(x, y))) = S(x)$. This is impossible as the rhs would properly contain its own substitution instance.

The *semi-unification problem* is to decide whether there exists a common solution for a finite number of inequalities. In what follows, we assume that our signature consists of only one function symbol, denoted "$\rightarrow$", which is binary (cf. Exercise 6), and written in infix notation.

(The analogy with types is of course intentional.) Since it is known (Exercise 7) that solving two inequalities is as difficult as solving the general case, we will consider only pairs of inequalities of the form:

$$A \leq_0 A', \qquad B \leq_1 B'.$$

Here and below we use the indices 0 and 1 to identify the inequalities. We ask for the existence of three substitutions S, R_0 and R_1, such that we have

$$R_0(S(A)) = S(A'), \quad \text{and} \quad R_1(S(B)) = S(B').$$

It is convenient to think of the variables $\alpha, \beta, \ldots$ occurring in our inequalities as of unknown terms $S(\alpha), S(\beta), \ldots$ in very much the same way as we think of the x in "$x^2 + 2x \leq 3$" as of an unknown number (which later will turn out to be between -3 and 1). From this point of view it is natural to introduce new unknowns, $\alpha_0, \beta_0, \ldots$ and $\alpha_1, \beta_1, \ldots$, with the intended meaning $R_0(S(\alpha)), R_0(S(\beta)), \ldots$, and $R_1(S(\alpha)), R_1(S(\beta)), \ldots$, respectively. If we denote by t_i a term obtained from t by replacing all variables ν by ν_i, we can rewrite our inequalities as the following equations:

$$A_0 = A', \qquad B_1 = B'$$

We may now try to solve these equations by the ordinary elimination of unknowns, but we must keep in mind the intended relation between ν's and ν_i's. Thus, for instance, if we find out that α must equal $\beta_1 \to \beta_0$, we must also conclude that α_1 must be a term of the form $\beta_{11} \to \beta_{01}$. Clearly, this means introducing new unknowns ν_{ij}, which are thought of as the unknown terms $R_j(R_i(S(\nu)))$. Further steps may involve introducing arbitrarily many unknowns of the form ν_w, where $w = i_1 i_2 \ldots i_n$, to represent $R_{i_n}(\ldots R_{i_2}(R_{i_1}(S(\nu)))\ldots)$.

Thus, solving semi-unification is very much like solving ordinary unification, with a possibly unbounded number of unknowns. Instead of formally describing this simple algorithm, let us consider an example. The initial instance consists of the following two inequalities:

$$(\varepsilon \to \gamma) \to (\gamma \to \zeta) \leq_0 \alpha \to \beta, \qquad (\beta \to \delta) \to (\gamma \to \nu) \leq_1 \alpha \to \delta. \tag{*}$$

We write these inequalities in the form of equations:

$$(\varepsilon_0 \to \gamma_0) \to (\gamma_0 \to \zeta_0) = \alpha \to \beta, \qquad (\beta_1 \to \delta_1) \to (\gamma_1 \to \nu_1) = \alpha \to \delta,$$

and we eliminate the unknowns as follows:

$$\begin{aligned} \alpha &= \varepsilon_0 \to \gamma_0 = \beta_1 \to \delta_1; \\ \beta &= \gamma_0 \to \zeta_0; \\ \delta &= \gamma_1 \to \nu_1. \end{aligned}$$

From the last two of the above equations it follows that the variables β_1 and δ_1 occurring in the first line should also be eliminated:

$$\begin{aligned} \beta_1 &= \gamma_{01} \to \zeta_{01}; \\ \delta_1 &= \gamma_{11} \to \nu_{11}. \end{aligned}$$

Further, since $\gamma_0 = \delta_1$, we must also have $\gamma_{01} = \delta_{11}$, and the variable δ_{11} should be eliminated too, as we must have $\delta_{11} = \gamma_{111} \to \nu_{111}$. Note that these eliminations increase the size of

equations defining the unknowns which already have been eliminated; we finally get:

$$\begin{aligned}\delta &= \gamma_1 \to \nu_1;\\ \beta &= (\gamma_{11} \to \nu_{11}) \to \zeta_0;\\ \alpha &= ((\gamma_{111} \to \nu_{111}) \to \zeta_{01}) \to (\gamma_{11} \to \nu_{11})\end{aligned}$$

There is nothing more to eliminate at the right-hand side, and thus the above equations describe a solution of our initial system of inequalities (which is identity on ε, γ, ζ and ν). A solution will always be found in this way, provided one exists. If no solution exists, this process will continue forever, and the terms to be assigned to some of the initial unknowns will grow infinitely.

The reader is invited to define precisely the algorithm to solve semi-unification, and to prove that a solution, if exists, will always be found (Exercise 3). Otherwise, the algorithm keeps eliminating unknowns and the size of the "solution to be" grows infinitely. In fact, a solution in possibly infinite terms (trees) always exists for our signature, and our algorithm constructs this solution in an infinite number of steps. (If there are constants in the signature, then it may happen that even an infinite solution does not exist.)

In most cases, the algorithm can also detect the non-existence of a solution. This happens when one derives an equality of the form $\alpha_w = A$, with some α_{wu} occurring as a proper subterm of A. Such an equality, called a "loop", is unsatisfiable, as the solution for α_{wu} must be at least as large as the solution for α_w. Unfortunately, this is not the only possible cause of non-termination of our algorithm, because the semi-unification problem is undecidable. In fact, no explicit example is known of a system of inequalities of reasonable size, which would lead to non-termination and would not generate a loop. (One could extract such inequalities from Hooper's proof, but that would not be of reasonable size.)

Path equations: First, let us recall that terms in our signature are just simple types and thus can be identified with binary trees. A position of a subterm in a term can be identified in the ordinary way by a sequence of 0's and 1's representing the path leading „upward" from the occurrence of our subterm to the root. Let 0 stands for "left" and 1 for "right". Consider an equation of the form $\alpha = A$. If a variable β occurs in A at node Π, then we may write this information in the form of a *path equation* of the form $\beta = \Pi\alpha$. For example, we write $\beta = 001\alpha$ and $\beta = 11\alpha$ if we have $\alpha = \gamma \to ((\beta \to \gamma) \to \beta)$. Of course, we may also consider systems of path equations that may have solutions or not. In order to investigate such systems (involving our auxiliary variables of the form ν_w) it is convenient to allow the following general form of a path equation

$$\Pi\alpha_w = \Sigma\beta_v.$$

The informal meaning of such an equation is: "the subterm of α_w at node Π and the subterm of β_v at node Σ are the same".

Clearly, each instance of the semi-unification problem can be transformed into a system of path equations involving the original variables and auxiliary variables of the form α_i. These path equations are either of the form $\alpha = \Pi\beta_i$ or of the form $\alpha_i = \Pi y$. For instance, for the

inequalities (*) we obtain the following path equations:

$$\begin{array}{ll} \varepsilon_0 = 0\alpha; & \beta_1 = 0\alpha; \\ \gamma_0 = 1\alpha; & \delta_1 = 1\alpha; \\ \gamma_0 = 0\beta; & \gamma_1 = 0\delta; \\ \zeta_0 = 1\beta; & \nu_1 = 1\delta. \end{array}$$

We can now derive other path equations, for instance

$$\begin{array}{rcll} \gamma_{111} & = & 0\delta_{11} & (\text{because } \gamma_1 = 0\delta); \\ 0\delta_{11} & = & 01\alpha_1 & (\text{because } \delta_1 = 1\alpha); \\ 01\alpha_1 & = & 0\gamma_{01} & (\text{because } 1\alpha = \gamma_0); \\ 0\gamma_{01} & = & 00\beta_1 & (\text{because } \gamma_0 = 0\beta); \\ 00\beta_1 & = & 000\alpha & (\text{because } \beta_1 = 0\alpha). \end{array}$$

Finally, by transitivity, we conclude that $\gamma_{111} = 000\alpha$. Note that this equation means that the path 000 must be present in the unknown type to be substituted for α, and thus this type must be of depth at least 3. In general, unbounded path equations of the form $\gamma_w = \Pi\alpha$ preclude the existence of a finite solution. This observation can be made precise: one can design inference rules for path equations (Exercise 4) so that the following property holds:

Lemat 14.10 *A set of inequalities E has a solution if and only if there is a number n such that for every variable α occurring in E and for every derivable path equation of the form $\beta_w = \Pi\alpha$, the length of Π does not exceed n.*

For the right-to-left direction in Lemma 14.10 observe that the bound n on path equations forces the "potentially infinite solution" to be of depth at most n. In other words, our algorithm for solving semi-unification must terminate, as no unknown can grow infinitely.

Intercell Turing Machines: We now extract a machine from our path equations. Consider again the equation $000\alpha = \gamma_{111}$ of the previous example. For a better presentation, let us write the indices in path expression in the ordinary font, i.e., write e.g., $0\delta11$ instead of $0\delta_{11}$. We have the following equations:

$$\gamma111 = 0\delta11 = 01\alpha1 = 0\gamma01 = 00\beta1 = 000\alpha.$$

Observe that this looks very much like a recording of a machine computation where e.g. the sequence $0\gamma01$ represents an ID with 0 to the left and 01 to the right of the head and with the initial state denoted by γ:

$$\gamma111 \Rightarrow 0\delta11 \Rightarrow 01\alpha1 \Rightarrow 0\gamma01 \Rightarrow 00\beta1 \Rightarrow 000\alpha.$$

During this computation, the machine head moved three steps to the right. In general, a derivable path equation $\gamma_w = \Pi\alpha$ with the length of Π equal to n would give our machine a possibility to move its head n steps to the right, when started in an ID corresponding to γ_w. This observation is the key to the reduction of Turing Machine boundedness to semi-unification.

The technical construction to implement this idea must begin with an observation that the path expressions above do not really behave like ID's of an ordinary Turing Machine. First, the behaviour of our machine may depend equally well on the the first symbol to the right of the tape head and on the first symbol to the left. Thus, a more adequate model is an *intercell Turing Machine*, where the tape head is always positioned *between* two adjacent tape cells, rather than over a single cell, so that either one can be scanned at the next step.

Formally, an *intercell Turing Machine* (abbreviated ITM), over the alphabet $A = \{0, 1\}$ (where 0 is identified with blank) consists of a finite set of states Q, and a transition relation $T \subseteq Q \times \{-1, +1\} \times A \times A \times Q$. An instantaneous description (ID) of an ITM takes the form $\langle w_1, \alpha, m, w_2 \rangle$, and the understanding is that:

- the current state is α;
- the tape head is positioned between the $m-1$-th and m-th tape cell;
- the contents of the tape is $w_1 w_2$; the first symbol of w_2 occupying the m-th tape cell.

The *next move* relation $\vdash_Y$ on ID's of an intercell machine Y is defined as follows:

$$\langle w_1 a, \alpha, m, w_2 \rangle \vdash_Y \langle w_1, \beta, m-1, b w_2 \rangle, \quad \text{for} \quad \langle \alpha, -1, a, b, \beta \rangle \in T;$$

$$\langle w_1, \alpha, m, a w_2 \rangle \vdash_Y \langle w_1 b, \beta, m+1, w_2 \rangle, \quad \text{for} \quad \langle \alpha, +1, a, b, \beta \rangle \in T.$$

An ITM is *deterministic* iff there is at most one move possible from any given ID, the direction of the move being determined by the internal state. More formally, the set of states of a deterministic machine splits into two disjoint subsets Q_L and Q_R, so that

$$T \subseteq (Q_L \times \{-1\} \times A \times A \times Q) \cup (Q_R \times \{+1\} \times A \times A \times Q),$$

and of course T must be a partial function. It is a routine exercise to show that intercell Turing Machines can simulate ordinary Turing Machines, in particular the boundedness problem for deterministic intercell Turing Machines is undecidable.

Symmetric machines: The next special property is far less common. Our rewriting of path equations is entirely symmetric; it is actually a special kind of a Thue system. Thus we are interested only in *symmetric* machines, i.e., intercell Turing Machines with symmetric next move relations. It turns out that this restriction is not essential as far as the boundedness problem is concerned.

Lemat 14.11 *Let Y be a deterministic intercell Turing Machine, and let Y_S be an ITM whose next move relation is the symmetric closure of the next move relation of Y. The machine Y_S is bounded if and only if Y is bounded.*

Here is a proof hint: if Y_S can move its head $2\ell - 1$ cells to the left then the shortest possible such computation must consist of two phases, one being a legal computation of Y and the other being a reversed computation of Y; during one of these phases the tape head of Y is moved at a distance at least ℓ.

It follows that the boundedness problem for symmetric intercell TM's is also undecidable. It remains to show how to construct an instance of semi-unification so that it has a solution

iff a given symmetric intercell Turing Machine is bounded. The essence of our reduction is to simulate a machine move of the form e.g.,

$$\ldots\alpha 0\ldots \Leftrightarrow \ldots 1\beta\ldots$$

in such a way that the path equation

$$\alpha_0 = 1\beta$$

can be extracted from the constructed inequalities (actually, from the inequality numbered 0 in this case). This means that, for each tuple of the form $\langle\alpha, +1, 0, 1, \beta\rangle \in T$, we want the inequality number 0 to be of the form

$$\ldots(\ldots\to\alpha)\ldots \leq_0 \ldots\beta\ldots$$

where $(\ldots\to\alpha)$ and β occur at the same position. Let us make it more precise. With no loss of generality we can assume that the part of T corresponding to right moves, i.e., the subset $T\cap Q\times\{+1\}\times A\times A\times Q$ (we may forget now about the other part, because of the symmetry), is of the following form (i.e., the quintuples come in pairs):

$$\begin{array}{l}\{\langle\alpha^\ell, +1, 0, 1, \beta^\ell\rangle : \ell = 1,\ldots m\}\cup\\ \{\langle\gamma^\ell, +1, 0, 0, \beta^\ell\rangle : \ell = 1,\ldots m\}\cup\\ \{\langle\alpha^\ell, +1, 1, 1, \beta^\ell\rangle : \ell = m+1,\ldots M\}\cup\\ \{\langle\gamma^\ell, +1, 1, 0, \beta^\ell\rangle : \ell = m+1,\ldots M\}.\end{array}$$

We identify the internal states of our machine with type variables. Let $A^\ell = \alpha^\ell\to\gamma^\ell$. The inequalities are:

$$A^1\to A^2\to\cdots A^m \leq_0 \beta^1\to\beta^2\to\cdots\beta^m$$
$$A^{m+1}\to A^{m+2}\to\cdots A^M \leq_1 \beta^{m+1}\to\beta^{m+2}\to\cdots\beta^M$$

A solution of these inequalities exists if and only if our machine is bounded. A formal proof of this fact is based on the equivalence of the following conditions:

- A path equation $\Pi\alpha_w = \Sigma\beta_v$ is derivable;
- The instantaneous descriptions $\langle\Pi, \alpha, k, w\rangle$ and $\langle\Sigma, \beta, k', v\rangle$, where $k' = k - |\Pi| + |\Sigma|$, are reachable from each other.

We conclude with the following fact:

Twierdzenie 14.12 *The semi-unification problem is undecidable.*

Ćwiczenia

1. Prove that unification can be seen as a special case of semi-unification.
2. Consider a system of $2n+1$ inequalities of the form
$$f(x^0, f(x^0, x^0)) \leq f(x^0, y^0),\ f(x^i, y^i) \leq f(x^{i+1}, z^{i+1}),\ f(x^i, y^i) \leq f(z^{i+1}, y^{i+1}),$$
where $i = 0,\ldots,n-1$. Show that the system has a solution, but every such solution must assign to y^i a term of depth at least 2^i, and of length at least 2^{i+1}.

3. Design a partial algorithm for solving instances of the semi-unification problem, and prove that it is correct, i.e., it halts producing a solution if a solution exists. Moreover, the constructed solution S is *principal* in that all other solutions are of the form $S' \circ S$, for some S'. However, the converse is not true, i.e., $S' \circ S$ is not necessarily a solution for arbitrary S'.
4. Define a calculus of path equations (a set of rules to infer path equations) so that the derivable path equations are exactly those satisfied by all solutions of the initial instance of semi-unification, including the infinite "solution" obtained when the algorithm does not terminate. Complete the proof of Lemma 14.10.
5. Prove that the boundedness problem for deterministic intercell Turing Machines is undecidable. Then prove the same for symmetric machines.
6. Prove that "linear" semi-unification (each variable occurs at most once at the left-hand side of each inequality) is decidable. In particular, semi-unification with only unary function symbols is decidable. Hint: if the path equation $\Pi\alpha = \beta_w$ is derivable, then it can be obtained by a sequence of rewrites of the form $\Pi\gamma_{iw} \Rightarrow \Pi'\delta_w$, i.e., with the index shrinking at each step. If this rewriting is sufficiently long, then it must consist a segment of the form $\Pi(\beta_v)_w \Rightarrow \cdots \Rightarrow \Pi(\Pi'\beta)_w$ (think of Π as of a push-down store).
7. Prove that semi-unification with arbitrary number of inequalities can be reduced (in presence of at least binary function symbols) to semi-unification with two inequalities. Hint: code inequation numbers with strings of 0's and 1's and design the two new inequalities so that e.g. $R_0(R_1(R_0(R_0(S(x))))) = R_n(S(x))$, where x is a variable occurring in the original instance, and n is coded by 0010. (That is, the auxiliary variable x_n should be replaced by x_{0010}.)

15 Dodatek: Schematologia porównawcza

By a *signature* we mean a sequence σ of function symbols (including constants) and relation symbols, each symbol with a nonnegative arity. A signature may contain the distinguished symbol $=$ of equality. We always assume that σ contains at least one function symbol.

A *σ-structure* is an arbitrary set A, the *domain* of the structure, together with an appropriate interpretation of symbols in σ. That is, each function symbol f corresponds to a function f^A over A of the appropriate arity, while each relation symbol r corresponds to a relation r^A of the appropriate arity. The symbol $=$ is always interpreted as equality.

15.1 Flow-chart programs

A (deterministic) *flow-diagram* (or "flow-chart program") over σ, is a finite, directed graph of nodes labeled by instructions. The instructions may take one of the following forms:

1. Assignments: $x := c$, or $x := y$, or $x := f(x_1, \ldots, x_n)$;
2. Tests: $\varphi?$;
3. Initial and final instructions: $\mathbf{start}(x_1, \ldots, x_k)$, and $\mathbf{stop}(x)$.

Here, x, y, x_i, for all i, are variables, c is a constant, f is a function symbol in σ of arity n, and φ is an open (quantifier-free) formula over σ.[26] The variables listed after **start** are called

[26]Sometimes other kinds of formulas are considered as tests, e.g. arbitrary first-order formulas.

the *input variables* of the program, and the x in $\mathbf{stop}(x)$ is the *output variable*. We require that $k \geq 1$. Other variables are called *auxiliary* or *local*.

Each node labeled by an assignment or **start** has one outcoming edge. There are two edges outcoming from each test node — one for "yes" and one for "no", while **stop** has no outcoming edges at all. There is exactly one node labeled **start**, and this node has no incoming edges. The number of incoming edges is arbitrary, except **start** is the only initial node.

It is often convenient to number the nodes of a flow-diagram and present it as a sequence of numbered instructions of the following forms:

a) $1 : \mathbf{start}(x_1, \ldots, x_k);\ \mathbf{goto}\ j;$

b) $i : x := c;\ \mathbf{goto}\ j;$

c) $i : x := y;\ \mathbf{goto}\ j;$

d) $i : x := f(x_1, \ldots, x_n);\ \mathbf{goto}\ j;$

e) $i : \mathbf{if}\ \varphi\ \mathbf{then\ goto}\ j\ \mathbf{else\ goto}\ k;$

f) $i : \mathbf{stop}(x).$

A variant of the above definition may also be considered, where instructions of the form (f) are replaced by $\mathbf{stop}(\mathit{true})$ and $\mathbf{stop}(\mathit{false})$.

A flow-chart is called *simple* if the formulas φ in tests are always of the form $r(x_1, \ldots, x_n)$ or $\neg r(x_1, \ldots, x_n)$, where r is a relation symbol in σ of arity n.

15.2 While-programs

We define a while-program as a special case of a flow-chart program, which consists of a *segment of instructions* preceded by a start instruction and followed by a stop. A segment of instructions is either a single assignment, or a sequence of segments (denoted $B_1; B_2$), or a conditional written **if** φ **then** B_1 **else** B_2 **fi**, or a while-loop **while** φ **do** B_1 **od**.

15.3 Operational semantics

Consider a flow-chart P with input variables $x_1, \ldots, x_k$ and auxiliary variables $x_{k+1}, \ldots, x_m$. Let A be a σ-structure. A *state* (configuration, ID) of P in A is a pair $\langle i, v \rangle$, where i is an instruction number and $v \in A^m$ is a valuation of the variables $x_1, \ldots, x_m$. If $i = 1$, we assume that $v(x_j) = v(x_1)$, for all $j > k$, so that we can actually assume the initial valuation to be defined only on the input variables ($v \in A^k$). Every state $\langle i, v \rangle$ (except when i is a number of a stop instruction (f)) uniquely determines its succesor state $\langle i', v' \rangle$, which is written $\langle i, v \rangle \longrightarrow \langle i', v' \rangle$:

- $\langle 1, v \rangle \longrightarrow \langle j, v \rangle$, for an instruction of the form (a).
- If the i-th instruction is an assignment of the form (b–d) then $\langle i, v \rangle \longrightarrow \langle j, v' \rangle$, where v' differs from v only on x:

  b) $v'(x) = c^A$;

c) $v'(x) = v(y)$;
d) $v'(x) = v(f(x_1, \ldots, x_n))$.

- If the i-th instruction is a test (e) and $A, v \models r(x_1, \ldots, x_n)$ then $\langle i, v\rangle \longrightarrow \langle j, v\rangle$. Otherwise $\langle i, v\rangle \longrightarrow \langle k, v\rangle$.

A sequence of states $\langle i_1, v_1\rangle, \ldots, \langle i_n, v_n\rangle$ is called a *computation* iff $\langle i_\ell, v_\ell\rangle \longrightarrow \langle i_{\ell+1}, v_{\ell+1}\rangle$, for every $\ell = 1, \ldots n-1$. We write $\langle i_1, v_1\rangle \longrightarrow\!\!\!\!\!\rightarrow \langle i_n, v_n\rangle$.

If $x_1, \ldots, x_k$ are input variables of a program and $x_{k+1}, \ldots, x_m$, are its auxiliary variables, then the state $\langle 1, a_1, \ldots, a_k, a_1, \ldots, a_1\rangle$ is the initial state for the input $\mathbf{a} = \langle a_1, \ldots, a_k\rangle \in A^k$. This initial state will also be written as $\langle 1, \mathbf{a}\rangle$. Consider the maximal computation starting from $\langle 1, \mathbf{a}\rangle$. If this computation is finite, i.e., ends with $\langle i, v\rangle$, where i is a number of a stop instruction (f), then $v(x)$ is called the *output*, and we write $P^A(\mathbf{a}) = v(x)$. This defines a partial function

$$P^A : A^k \longrightarrow A.$$

In case stop instructions are of the form **stop**(*true*) and **stop**(*false*), we have instead a partial relation $P^A : A^k \longrightarrow \{true, false\}$.

We write $A, v \models P{\downarrow}$ when $P^A(\mathbf{a})$ is defined, and $A, v \models P{\uparrow}$ otherwise.

We say that two programs P and Q are *equivalent* (over a restricted class of structures) iff the functions P^A and Q^A are equal in every structure A (in the appropriate class).

15.4 Gra w kamyki

Gra w kamyki (ang. *pebble game*), to w istocie łamigłówka dla jednego gracza, polegająca na układaniu m kamyków (pionków, żetonów) w wierzchołkach zorientowanego grafu $G = (V, E)$, gdzie $E \subseteq V \times V$. Celem gry jest osiągnięcie wskazanego wierzchołka końcowego f. W dowolnym wierzchołku v można postawić pionek wtedy, gdy pokryte pionkami są wszystkie poprzedniki v, tj. wierzchołki z których wychodzą krawędzie prowadzące do v. Pionek postawiony w wierzchołku v może być nowy lub zdjęty z innego wierzchołka, np. z jednego z poprzedników v. Ściślej: *konfiguracja* gry to dowolny zbiór $S \subseteq V$, o co najwyżej m elementach, a przejście z konfiguracji S do S' jest dozwolone, gdy $S' \subseteq S \cup \{v\}$ oraz $\{w \mid (w, v) \in E\} \subseteq S$ i różnica $S - S'$ ma co najwyżej jeden element. Gra jest wygrana, jeśli od konfiguracji pustej potrafimy przejść do konfiguracji z wierzchołkiem f.

Złożoność kamyczkowa grafu G (z wyróżnionym wierzchołkiem f) to najmniejsza liczba m, przy której gra jest wygrana.

Uwaga: Złożoność kamyczkowa nie zmieni się, jeśli dopuścimy możliwość usuwania pionków z grafu, tj. przejście od konfiguracji S do $S' \subseteq S$.

Lemat 15.13 *Jeśli graf G_1 jest podgrafem grafu G_2 i ma ten sam wierzchołek końcowy, to złożoność kamyczkowa G_1 jest co najwyżej taka, jak złożoność kamyczkowa G_2.*

Dowód: Niech $G_1 = (V_1, E_1)$ i $G_2 = (V_2, E_2)$ przy czym $f \in V_1 \subseteq V_2$ i $E_1 \subseteq E_2$. Jeśli w grafie G_2 z m żetonami można przejść w jednym kroku od konfiguracji S do konfiguracji S', to w grafie G_1 można przejść od $S \cap V_1$ do $S' \cap V_1$ w co najwyżej jednym kroku. □

Niech teraz $T_n = \{w \in \{0,1\}^* \mid |w| \leq n\}$ oznacza pełne drzewo binarne wysokości n, w którym krawędzie prowadzą od synów do ojców, tj. $E = \{(wa, w) \mid |w| < n \wedge a \in \{0,1\}\}$. Jedynym wierzchołkiem końcowym jest korzeń drzewa (słowo puste ε).

Lemat 15.14 *Złożoność kamyczkowa grafu T_n jest równa $n+1$.*

Dowód: Pokażemy część trudniejszą: nie da się dojść do korzenia używając tylko n kamieni. Przypuśćmy, że się da i niech $\varnothing = S_1, S_2, \ldots, S_r \ni \varepsilon$ będzie takim ciągiem konfiguracji. Dowodzimy przez indukcję, że dla dowolnego $m = 1, \ldots, n+1$, któryś ze zbiorów S_i zawiera m-elementowy „przekrój" drzewa T_n, tj. taki podzbiór $\{w_1, \ldots, w_m\}$, że każdy liść T_n ma dokładnie jeden prefiks postaci w_i. Dla $m = n+1$ oznacza to, że potrzebujemy $n+1$ pionków.

Nietrudno pokazać, że wtedy $|w_i| \leq m-1$ dla wszystkich i (elementy m-elementowego przekroju są nie dalej niż $m-1$ kroków od korzenia).

Dla $m = 1$ takim przekrojem jest $\{\varepsilon\}$. W kroku indukcyjnym niech i będzie najmniejszą liczbą o tej własności, że S_i zawiera m-elementowy przekrój. Konfiguracja S_i powstaje z S_{i-1} przez dodanie jakiegoś wierzchołka v. Ten wierzchołek musi należeć do m-elementowego przekroju, bo takiego przekroju nie było w S_{i-1}. Wtedy jednak $v0, v1 \in S_{i-1}$ i mamy $m+1$-elementowy przekrój w S_{i-1}. □

15.5 Ograniczenia siły obliczeniowej

Język flow-diagramów ma dwa ograniczenia. Pierwsze dotyczy „pamięci algebraicznej" – skończona liczba zmiennych umożliwia przechowywanie w pamięci tylko skończonej liczby elementów modelu. Drugie ograniczenie dotyczy „sterowania" – mamy tylko skończenie wiele stanów. Okazuje się, że oba te ograniczenia są istotne. Jako pierwszy przykład rozpatrzmy funkcję zadaną przez taką definicję rekurencyjną:

$$F(x) = \mathsf{if}\ P(x)\ \mathsf{then}\ x\ \mathsf{else}\ g(F(g(x,c)), F(g(c,x))), \tag{$*$}$$

w języku gdzie P jest jednoargumentowym symbolem relacyjnym, g jest dwuargumentowym symbolem funkcyjnym, a c jest stałą. Niech teraz A_n będzie algebrą termów dla tego języka, w której relacja P zachodzi dla termów stałych z co najmniej n wystąpieniami symbolu g. Jeśli użyjemy skrótów $l(t) = g(t,c)$, $r(t) = g(c,t)$ i przepiszemy nasza definicję w ten sposób:

$$F(x) = \mathsf{if}\ P(x)\ \mathsf{then}\ x\ \mathsf{else}\ g(F(l(x)), F(r(x))),$$

to zauważymy, że $F(c)$ jest termem, który można przedstawić jako pełne drzewo binarne wysokości n (o wierzchołkach etykietowanych przez g) w którego liściach zaczepione jest 2^n różnych termów zbudowanych z operacji l i r. Ważne, że wszystkie te termy są różne, z czego wynika od razu, że złożoność kamyczkowa otrzymanego grafu jest co najmniej $n+1$. A więc nie można obliczyć wartości $F(c)$ używając n lub mniej zmiennych. Wynika stąd natychmiast:

Twierdzenie 15.15 *Nie istnieje flow-diagram, który w dowolnej strukturze oblicza funkcję F zadaną definicją $(*)$.*

Drugi przykład dotyczy sygnatury, w której są dwa jednoargumentowe symbole funkcyjne f i g i jeden jednoargumentowy symbol relacyjny R. Mamy taką rekurencyjną definicję relacji:

$$S(x) = \text{if } R(x) \text{ then } \mathit{true} \text{ else if } S(f(x)) \text{ then } S(g(x)) \text{ else } \mathit{false}. \qquad (**)$$

To jest oczywiście relacja częściowa, prawdziwa wszędzie tam, gdzie określona, ale nie zawsze określona. Program obliczający taką relację powinien się zatrzymywać dla wartości wejściowych spełniających tę relację, osiągając instrukcję końcową postaci **stop**(*true*) i zapętlać się w przeciwnym razie. Ale takiego programu nie ma.

Twierdzenie 15.16 *Nie istnieje flow-diagram, który w dowolnej strukturze oblicza relację S zadaną definicją $(**)$.*

Dowód: Przypuśćmy, że taki program P istnieje, i że ma k zmiennych oraz r instrukcji. Konfiguracje P są więc postaci $(i, a_1, \ldots, a_k)$. Rozpatrzmy strukturę T_n o dziedzinie $\{0,1\}^*$, w której relacja R^{T_n} jest spełniona dla wszystkich słów o długości równej n i tylko takich. Oczywiście P powinien się zatrzymywać dla wejścia ε, bo korzeń drzewa spełnia relację S.

Dla $a, b \in T_n$ przyjmijmy, że $a \sim b$ gdy a i b generują w T_n izomorficzne podstruktury. W praktyce oznacza to, że wysokości wierzchołków długości słów) a i b są albo równe albo obie większe od n. Relacja $\sim$ ma więc $n+2$ klasy abstrakcji. Rozszerzymy ją na konfiguracje programu przyjmując, że $(i, a_1, \ldots, a_k) \sim (j, b_1, \ldots, b_k)$, gdy $i = j$ oraz $a_1 \sim b_1, \ldots, a_k \sim b_k$. To rozszerzenie ma $r \cdot (n+2)^k$ klas abstrakcji.

Najważniejsza obserwacja jest taka: jeśli konfiguracje C_1 i C_2 są w relacji $\sim$, oraz program przechodzi z C_1 do C_1' i z C_2 do C_2', to także $C_1' \sim C_2'$. Dowód tego jest łatwy, bo jedyny dostępny test jest jednoargumentowy.

Z powyższego wynika, że jeśli w jakimś obliczeniu wystąpią dwie $\sim$-równoważne konfiguracje, to takie obliczenie musi się zapętlić. Żadne obliczenie skończone nie może składać się z więcej niż $r \cdot (n+2)^k$ kroków, także to, które uruchomimy dla wejścia ε. Liczba wszystkich elementów struktury, które występują w tym obliczeniu jest więc co najwyżej $kr \cdot (n+2)^k$. To oszacowanie zachodzi dla wszystkich T_n. Ale dla dostatecznie dużego n oznacza to, że wśród elementów wysokości n (jest ich 2^n) są takie, których nasz program nie „ogląda" akceptując ε. Przypuśćmy, że w_n jest takim elementem. A zatem obliczenie P na wejściu ε w T_n jest dokładnie takie samo jak w strukturze U_n, różniącej się od T_n tylko tym, że $w_n \notin R^{U_n}$. Ponieważ relacja S nie zachodzi w wierzchołku drzewa U_n, więc nasz program się „pomyli". □

15.6 Obliczenia z równością

Zbadamy zachowanie flow-diagramów na pełnych skończonych drzewach binarnych tj. strukturach postaci $\mathcal{T}_n = \langle T_n, f, g, = \rangle$, gdzie $T_n = \{w \in \{0,1\}^* \mid |w| \leq n\}$, oraz $f(w) = 0w$, $g(w) = 1w$, dla $|w| < n$ i $f(w) = g(w) = w$, dla $|w| = n$. Załóżmy, że mamy flow-diagram P o k zmiennych i r instrukcjach. Konfiguracje P to krotki postaci $\langle i, a_1, \ldots, a_k \rangle$, gdzie $i \leq r$ oraz $a_j \in T_n$ dla $j = 1, \ldots, k$. Termy w tej sygnaturze możemy utożsamiać ze słowami, pisząc np $001x$ zamiast $f(f(g(x)))$.

It is convenient to consider formal computations of programs that may or may not correspond to some real computations. Krotkę postaci $\langle i, t_1, \ldots, t_k \rangle$, gdzie $t_1, \ldots, t_k$ są termami

o zmiennych $x_1, \ldots, x_k$, nazwiemy *formalną konfiguracją*. W naszym przypadku, termy t_ℓ są zawsze postaci wx_j. Przez *formalne obliczenie* rozumiemy ciąg takich krotek postaci $C_\ell = \langle i_\ell, w_1^\ell x_{j_1}, \ldots, w_k^\ell x_{j_k}\rangle$, w którym $C_0 = \langle i_0, x_1, \ldots, x_k\rangle$, a przejścia od C_ℓ do $\mathcal{C}_{\ell+1}$ odbywają się zgodnie z instrukcją o numerze i_ℓ. Jeśli ma ona np. postać $x_2 := f(x_1)$; goto i', to wtedy $i_{\ell+1} = i'$, $w^{\ell+1} = 0w^\ell$, a pozostałe współrzędne $\mathcal{C}_{\ell+1}$ są takie same jak C_ℓ. Jeśli instrukcja o numerze i_ℓ jest testem postaci if $x_p = x_q$ then goto i' else goto i'', to termy w $\mathcal{C}_{\ell+1}$ są takie same jak C_ℓ, a numer $i_{\ell+1}$ jest albo równy i' albo i''. Przy tym obie wartości są dozwolone, niezależnie od termów występujących w C_ℓ (tu się żaden test nie wykonuje). Dla uproszczenia, formalnym obliczeniem będziemy też nazywali po prostu odpowiedni ciąg instrukcji $i_0, i_1, \ldots$ Każde „prawdziwe" obliczenie wyznacza swoje obliczenie formalne, ale niekoniecznie na odwrót.

Jeśli teraz $C_\ell = \langle i_\ell, w_1^\ell x_{j_1}, \ldots, w_k^\ell x_{j_k}\rangle$ w formalnym obliczeniu α, to napiszemy $v_\alpha(\ell, i) = j_i$. Sens: wartość zmiennej x_i jest w ℓ-tym kroku obliczenia „potomkiem" początkowej wartości x_{j_i}.

Przypuśćmy teraz, że $v_\alpha(\ell, i) = j$ i nasze formalne obliczenie jest wyznaczone przez rzeczywiste obliczenie programu P. Jeśli początkową wartością zmiennej x_j jest element a, a końcową wartością zmiennej x_i jest b, to mówimy, że b jest *osiągalne* w ℓ krokach z a.

Powiemy, że zmienne x_i i x_j są *sklejone* w m-tym kroku formalnego obliczenia $\alpha = i_0, i_1, \ldots$, gdy dla pewnych p, q:

- instrukcja i_{m-1} jest postaci if $x_p = x_q$ then goto i_m else goto ℓ;
- ponadto $v_\alpha(m-1, p) = i$ oraz $v_\alpha(m-1, q) = j$.

Uwaga: Jeśli obliczenie dla wejścia $a_1, \ldots, a_k$ wyznacza ciąg etykiet α i zmienne x_i i x_j są sklejone w m-tym kroku α, to $a_i = wa_j$ lub $a_j = wa_i$ dla pewnego w, zależnego tylko od α i od długości słów a_i i a_j.

Niech $\zeta_\alpha(i, j) = m$, gdy x_i i x_j są sklejone w m-tym kroku α i m jest najmniejszą liczbą o tej własności. Jeśli takiego m nie ma, to niech $\zeta_\alpha(i, j) = 0$.

Lemat 15.17 *Niech ciągi $a_1, \ldots, a_k$ i $b_1, \ldots, b_k$ będą takie, że $|a_i| = |b_i|$ dla $i = 1, \ldots, k$, i niech α i β będą formalnymi obliczeniami tej samej długości, zaczynającymi się od tej samej etykiety i wyznaczonymi przez rzeczywiste obliczenia z wartościami początkowymi $a_1, \ldots, a_k$ i $b_1, \ldots, b_k$. Jeśli $\zeta_\alpha(i, j) = \zeta_\beta(i, j)$ dla wszystkich i, j, to $\alpha = \beta$.*

Dowód: Niech $\alpha = i_0, \ldots i_{m-1}, i_m$ oraz $\beta = i'_0, \ldots i'_{m-1}, i'_m$. Dowód jest przez indukcję ze względu na m. Z założenia indukcyjnego wynika, że ciągi $i_0, \ldots i_{m-1}$ i $i'_0, \ldots i'_{m-1}$ są identyczne, w szczególności $i_{m-1} = i'_{m-1}$. Istotny przypadek jest wtedy, gdy instrukcja i_{m-1} jest postaci if $x_p = x_q$ then goto i_m else goto ℓ. Ponieważ oba obliczenia przebiegały dotąd tak samo, więc są takie i, j, że $v_\alpha(m-1, p) = i = v_\beta(m-1, p)$ i $v_\alpha(m-1, q) = j = v_\beta(m-1, q)$. Ponadto $0 < \zeta_\beta(i, j) = \zeta_\alpha(i, j) \leq m$, tj. zmienne x_i i x_j są sklejone teraz lub były już sklejone wcześniej – i to w ten sam sposób w obu obliczeniach (patrz uwaga powyżej). Tak czy owak, wynik testu $x_p = x_q$ jest zdeterminowany przez to sklejenie, czyli $i'_m = i_m$. □

Niech $B \subseteq T_n$ ma co najwyżej k elementów. Przez $V(B, m)$ oznaczymy zbiór wszystkich wartości osiągalnych w co najwyżej m krokach obliczenia programu P z elementów zbioru B.

Lemat 15.18 *Istnieje taki wielomian $p(n,m)$, że dla każdego n zbiór $V(B,m)$ ma nie więcej niż $p(n,m)$ elementów.*

Dowód: Z lematu 15.17 wynika, że liczba możliwych obliczeń wyznaczających różne obliczenia formalne długości m szacuje się przez liczbę możliwych rozkładów wysokości danych wejściowych $(n+1)^k$ pomnożoną przez liczbę możliwych funkcji ζ_α czyli przez $(m+1)^{k^2}$. Weźmy teraz dwa obliczenia długości m o tym samym obliczeniu formalnym, rozpoczynające się od konfiguracji, gdzie wartości zmiennych należące do B są takie same. (Takich rozmieszczeń wartości z B jest co najwyżej stała liczba $(k+1)^k$.) W tych dwóch obliczeniach występują te same elementy osiągalne z B i jest ich najwyżej $k(m+1)$. A więc łącznie mamy co najwyżej $(n+1)^k \cdot (m+1)^{k^2} \cdot (k+1)^k \cdot k(m+1)$ elementów osiągalnych w m krokach. □

Przez $G_i(n)$ oznaczymy maksymalną długość takiego obliczenia programu P w $\mathcal{T}_n$, w którym pewne i elementów $b_1, \ldots, b_i$ występuje we wszystkich konfiguracjach i żadna konfiguracja się nie powtarza.

Lemat 15.19 *Dla każdego $m = 0, \ldots, k$ istnieje taki wielomian $p_m(n)$, że $G_m(n) \leq p_m(n)$.*

Dowód: Indukcja. Zaczynamy od $G_k(n) \leq r \cdot k!$ (to stała), skończymy na $G_0(n)$. W kroku indukcyjnym chcemy oszacować $G_m(n)$ znając $G_{m+1}(n)$. Niech więc $B = \{b_1, \ldots, b_m\}$ występuje we wszystkich konfiguracjach długiego obliczenia i niech $a \notin B$ występuje w jednej z tych konfiguracji. Element a musiał się pojawić nie wcześniej niż $G_{m+1}(n)$ kroków temu przez podstawienie z wartości b, przy czym $|b| = |a| - 1$. Jeśli $b \notin B$, to powtarzamy to samo rozumowanie dla b i w końcu widzimy, że a zostało otrzymane w ciągu co najwyżej $|a| \cdot G_{m+1}(n)$ kroków z jakiegoś elementu B. Ale takich a jest co najwyżej $p(n, n \cdot G_{m+1}(n))$, więc tylko tyle różnych wartości może wystąpić (oprócz B) w obliczeniu. Ponieważ B ma $m \leq k$ elementów, więc liczba różnych konfiguracji tego obliczenia nie przekracza $r \cdot (k + p(n, n \cdot G_{m+1}(n)))^k$. □

Następujące twierdzenie wynika natychmiast z tego, że $G_0(n)$ jest wielomianowo ograniczone.

Twierdzenie 15.20 (J. Tiuryn) *Dla dowolnego flow-diagramu P jest taki wielomian $w(n)$, że każde skończone obliczenie programu P w strukturze $\mathcal{T}_n$ ma co najwyżej $w(n)$ kroków.*

Wniosek 15.21 *Nie istnieje flow-diagram z równością zatrzymujący się dokładnie dla tych danych, które generują skończone struktury.*

Dowód: W obliczeniu długości $G_0(n)$ w $\mathcal{T}_n$ (na wejściu ε) może wystąpić co najwyżej $k \cdot G_0(n)$ elementów. Tego jest oczywiście za mało, aby obejrzeć wszystkie liście $\mathcal{T}_n$. Zatem każdy program P pomija pewien liść d i nie potrafi odróżnić $\mathcal{T}_n$ od struktury o dziedzinie $T_n \cup \{wd \mid w \in \{0,1\}^*\}$, w ktorej $f(wd) = 0wd$ oraz $g(wd) = 0wd$, dla każdego w. □

15.7 Recursive schemes

The language of recursive schemes, except of variables and symbols from the signature σ, includes *defined function symbols* of arbitrary arity. *Terms* of the extended language are defined as usual with the additional conditions:

– If $t_1, \ldots, t_n$ are terms, and $\mathcal{F}$ is an n-ary defined symbol, then $\mathcal{F}(t_1, \ldots, t_n)$ is a term;

– If t_1, t_2 are terms, and φ is an atomic[27] formula, then **if** φ **then** t_1 **else** t_2 is a term.

A *function definition* is an expression of the form

$$\mathcal{F}(x_1, \ldots, x_n) = t,$$

where $\mathcal{F}$ is an n-ary defined function symbol, and t is a term, with all free variables among $x_1, \ldots, x_n$. A *recursive scheme* P is a collection of definitions:

$\mathcal{F}_1(x_1, \ldots, x_{n_1}) = t_1$;

$\mathcal{F}_2(x_1, \ldots, x_{n_2}) = t_2$;

...

$\mathcal{F}_k(x_1, \ldots, x_{n_k}) = t_k$,

where the $\mathcal{F}_i$'s are distinct defined function symbols of arities n_i, and no defined function symbol other than the $\mathcal{F}_i$'s may occur in the $t_1, \ldots, t_k$. The symbol $\mathcal{F}_1$ is the *main* defined symbol of P, and we say that P is n_1-ary.

If a program P uses defined function symbols $\mathcal{F}_1, \ldots, \mathcal{F}_k$, each of them could be chosen as the main symbol. We use the notation P_i to denote the program with main symbol $\mathcal{F}_i$. Then of course P_1 is P.

This syntax can be extended to allow defined relation symbols in a natural way.

Operational semantics

The operational semantics is defined with respect to a given σ-structure A. We extend the language of terms by adding special constants to denote all elements of A. We identify elements of A with these constants. A function definition $\mathcal{F}(x_1, \ldots, x_n) = t$ is understood as a rewrite rule for *closed* terms:

$$\mathcal{F}(A_1, \ldots, A_n) \Rightarrow t[A_1/x_1, \ldots, A_n/x_n].$$

For a given recursive scheme P, we consider the reduction system consisting of such rules for all definitions in P, and in addition the following rules:

- $f(a_1, \ldots, a_k) \Rightarrow a$, where f is a k-ary function symbol in the signature, $a_1, \ldots, a_k$ are in A, and $f^A(a_1, \ldots, a_k) = a$;
- **if** φ **then** t_1 **else** $t_2 \Rightarrow t_1$, if $A \models \varphi$ (the formula φ must be an atomic formula of the form $r(a_1, \ldots, a_n)$, where $a_1, \ldots, a_n \in A$);
- **if** φ **then** t_1 **else** $t_2 \Rightarrow t_1$, if $A \models \neg\varphi$.

We write $t \longrightarrow_P t'$ (or just $t \longrightarrow t'$ if a closed term t can be rewritten into t' by a single application of one of the above rules. The notation $t \longrightarrow_P^L t'$ ($t \longrightarrow^L t'$) means that t' was

[27] The formula φ may contain defined function symbols.

obtained from t by a leftmost reduction step (the subterm replaced in this step is the leftmost possible redex in t). As usual, the symbol $\longrightarrow\!\!\!\!\!\rightarrow$ (with superscripts and subscripts) denotes many-step reduction.

A k-ary recursive scheme P with main symbol F defines a partial function P^A given by

$$P^A(a_1,\ldots,a_k) = a, \text{ if and only if } \mathcal{F}(a_1,\ldots,a_k) \longrightarrow\!\!\!\!\!\rightarrow^L_P a.$$

Denotational semantics

We extend the structure A by adding a new element $\bot$ and a flat ordering on $A_\bot = A \cup \{\bot\}$ with the least element $\bot$. Every n-ary signature function f^A is identified with the strict operation $f^A : A^n_\bot \to A_\bot$ satisfying $f^A(\vec{a}) = \bot$, whenever at least one component of $\vec{a}$ is $\bot$.

An *environment* is a function E that assigns a monotone function $E(\mathcal{F})$ over $A_\bot$ to each defined function symbol $\mathcal{F}$. Given an environment E, we define the meaning $[\![t]\!]^E \in A_\bot$, for every closed term t:

- $[\![a]\!]^E = a$, for $a \in A$;
- $[\![f(t_1,\ldots,t_n)]\!]^E = f^A([\![t_1]\!]^E,\ldots,[\![t_n]\!]^E)$;
- $[\![\mathcal{F}(t_1,\ldots,t_n)]\!]^E = E(\mathcal{F})([\![t_1]\!]^E,\ldots,[\![t_n]\!]^E)$;
- $[\![\textbf{if } r(t_1,\ldots,t_n) \textbf{ then } s_1 \textbf{ else } s_2]\!]^E = \bot$, when $[\![t_i]\!]^E = \bot$, for some i;
- $[\![\textbf{if } r(t_1,\ldots,t_n) \textbf{ then } s_1 \textbf{ else } s_2]\!]^E = [\![s_1]\!]^E$, if $\langle [\![t_1]\!]^E,\ldots,[\![t_n]\!]^E\rangle \in r^A$;
- $[\![\textbf{if } r(t_1,\ldots,t_n) \textbf{ then } s_1 \textbf{ else } s_2]\!]^E = [\![s_2]\!]^E$, if $\langle [\![t_1]\!]^E,\ldots,[\![t_n]\!]^E\rangle \in A^n - r^A$.

Let P be a recursive program with function definitions $\mathcal{F}_i(x_1,\ldots,x_{n_i}) = t_i$, where $i = 1,\ldots,k$. The program P defines a continuous transformation $\mathcal{W}$ on vectors of monotone functions:

$$\mathcal{W} : [A^{n_1}_\bot \to A_\bot] \times \cdots \times [A^{n_k}_\bot \to A_\bot] \to [A^{n_1}_\bot \to A_\bot] \times \cdots \times [A^{n_k}_\bot \to A_\bot].$$

Let the vector of functions $\mathcal{L}_1,\ldots,\mathcal{L}_n$ be the least fixpoint of this transformation, and let L be the environment given by $L(\mathcal{F}_i) = \mathcal{L}_i$. The equivalence of operational and denotational semantics is expressed by:

$$P_i(a_1,\ldots,a_{n_i}) \longrightarrow^L a \text{ if and only if } [\![P_i(a_1,\ldots,a_{n_i})]\!]^L = a.$$

The above makes sense for $a = \bot$ if "$t \longrightarrow^L \bot$" is understood as "the leftmost reduction starting from t is infinite".

15.8 Flow-charts with procedures

Our procedures are actually call-by-value "functions" with local variables and no side-effects. A *flow-chart with procedures* over σ is a tuple $\langle P_1,\ldots,P_n\rangle$ of flow-charts, each over a signature

extended with the function symbols $P_1, \ldots, P_n$ of appropriate arities. For simplicity, we allow the new symbols to be used only in assignment instructions of the form:

$$i : x := P_r(x_1, \ldots, x_\ell);\ \textbf{goto}\ j.$$

A *state* of such a program is a sequence of *activations*, each of the form $\langle m, j, v\rangle$ where $m = 1, \ldots, n$, and j, v are as usual. We require that in a state of the form $\ldots \langle m, i, v\rangle\langle r, \ell, w\rangle \ldots$ it is always the case that the i-th instruction in P_m is a call to P_j as above. An initial (final) state consists of a single initial (final) activation of P_1, but we may have to consider computations starting from arbitrary states (for uniformity).

Consider a state q of the form

$$\langle m_1, j_1, v_1\rangle\langle m_2, j_2, v_2\rangle \cdots \langle m_{k-1}, j_{k-1}, v_{k-1}\rangle\langle m_k, j_k, v_k\rangle.$$

We define the successor state q' depending on the instruction j_k. If it is neither a call instruction nor a stop, the state q' is determined by an appropriate modification of the top activation $\langle m_k, j_k, v_k\rangle$. If it is a call of the form:

$$j_k : x := P_m(x_1, \ldots, x_\ell);\ \textbf{goto}\ j,$$

then

$$q' = q \cdot \langle m, 1, v_{k-1}(x_1), \ldots, v_{k-1}(x_\ell), v_{k-1}(x_1), \ldots, v_{k-1}(x_1)\rangle,$$

where the number of repetitions of $v_{k-1}(x_1)$ at the end equals the number of auxilary variables of P_m. Finally, suppose that j_k refers to $\mathit{stop}(x)$, and the instruction j_{k-1} in $P_{m_{k-1}}$ is

$$j_{k-1} : y := P_{m_k}(x_1, \ldots, x_\ell);\ \textbf{goto}\ j.$$

Then q' is

$$\langle m_1, j_1, v_1\rangle\langle m_2, j_2, v_2\rangle \cdots \langle m_{k-1}, j, v_{k-1}[v_k(x)/y]\rangle.$$

A *computation* is a sequence of states such that each consecutive two states are in this relation. Let $q = \langle m_1, j_1, v_1\rangle\langle m_2, j_2, v_2\rangle \cdots \langle m_{k-1}, j_{k-1}, v_{k-1}\rangle\langle m_k, j_k, v_k\rangle$. We say that a computation starting from q returns a as the output, if it ends with a state of the form $\langle m_1, j, v\rangle$, where j is a number of a $\mathit{stop}(x)$ instruction and $v(x) = a$.

Twierdzenie 15.22 *Dla każdego schematu rekurencyjnego istnieje flow-diagram z procedurami, który w każdej strukturze definiuje tę samą funkcję. I na odwrót.*

15.9 Stosy i nie tylko

A flow-diagram with a *stack* (or a *push-down store*) is defined as a flow-diagram which may involve two additional instruction types:

- $i : \mathit{push}(x);\ \textbf{goto}\ j;$
- $i : x := \mathit{pop};\ \textbf{goto}\ j;$

for an arbitrary variable x.

Operational semantics for flowcharts with stacks is defined as for flowcharts but modified as follows: A state (in a structure A) is now of the form $\langle i, v, s\rangle$, where i and v are as before and s (the stack) is a sequence (possibly empty) of elements of A. (The convention is that the stack grows from left to right.) In the initial state we have $s = \varepsilon$.

The effect of executing "$push(x)$; **goto** j" on $\langle i, v, s\rangle$ is $\langle j, v, s{\cdot}v(x)\rangle$, and the effect of executing "$i : x := pop$; **goto** j" on $\langle i, v, s \cdot a\rangle$ is $\langle j, v[a/x], s\rangle$. Executing a *pop* on empty stack results in an infinite loop, i.e., we assume $\langle i, v, \varepsilon\rangle \longrightarrow \langle i, v, \varepsilon\rangle$.

Twierdzenie 15.23 *Flow-diagramy ze stosem obliczają w dowolnej strukturze te same funkcje co flow-diagramy z procedurami.*

Dowód: Porządny dowód tego twierdzenia jest dość skomplikowany, my zrobimy tylko pobieżny szkic obu translacji. Załóżymy od razu, że mamy do czynienia ze strukturami wyposażonymi w dostatecznie wiele różnych stałych, aby można było ich używać jako nazw procedur, zmiennych i etykiet instrukcji.[28]

Część 1: Przypuśćmy, że $P = \langle P_1, \ldots, P_n\rangle$ jest programem z procedurami. (Bez straty ogólności można zakładać, że procedura P_1 (program główny) nigdy nie jest wywoływana rekurencyjnie.) Skonstruujemy równoważny flow-diagram ze stosem S. Będzie on miał prawie wszystkie te instrukcje co procedury $P_1, \ldots, P_n$ i jeszcze trochę. Oczywiście wołania procedur i instrukcje **stop** trzeba zastąpić operacjami na stosie.

Poniżej (m, i) jest etykietą i-tej instrukcji procedury P_m. Wołanie procedury postaci

$$(m, i): \quad x_p := P_\ell(x_{i_1}, \ldots, x_{i_r});\ \textbf{go to}\ j$$

zastepujemy ciągiem instrukcji

$$\begin{array}{l} push(x_1);\ push(x_2);\ \ldots\ ;\ push(x_k);\ push(m);\ push(p);\ push(j);\\ z_1 := x_{i_1};\ \ldots\ ;\ z_r := x_{i_r};\ z_{r+1} := z_1;\ \ldots\ ;\ z_s := z_1;\ \textbf{go to}\ (\ell, 1),\end{array}$$

gdzie $x_1, \ldots, x_k$ i $z_1, \ldots, z_s$ to wszystkie zmienne odpowiednio w P_m i P_ℓ, przy tym $z_1, \ldots, z_r$ są wejściowe. Instrukcje $push(m); push(p); push(j)$ należy rozumieć jako polecenia odłożenia na stos elementów struktury używanych do identyfikacji procedury, zmiennej i instrukcji. W ten sposób zapisujemy na stosie bieżącą aktywację i przechodzimy do nowej procedury.

W razie natrafienia na instrukcję **stop** programu głównego kończy się też obliczenie programu ze stosem. Instrukcja postaci (m, i) : **stop**(x_j), dla $m \neq 1$, jest zastąpiona przez:

$v := x_j;\ q := pop;\ p := pop;\ n := pop;\ y_k := pop;\ y_{k-1} := pop;\ \ldots;\ y_1 := pop;\ y_p := v;$ **go to** (n, q).

Powyższe „instrukcje" są skrótowym i nieformalnym zapisem następujących operacji. Najpierw zapamiętujemy wartość wynikową, używając pomocniczej zmiennej v. Potem odczytujemy ze stosu numer p zmiennej, na którą należy tę wartość podstawić, oraz identyfikację instrukcji (n, q) do której należy wrócić. Następnie odtwarzamy ze stosu wartości zmiennych $y_1, \ldots, y_k$ procedury P_n, poprawiamy wartość zmiennej y_p i powracamy do wykonywa-

[28] W istocie można sobie poradzić bez tego. Najpierw trzeba zauważyć, że wystarczy mieć dwie rozróżnialne wartości, potem skonstruować podprogram zwany *lokatorem*, który potrafi takie dwie wartości znaleźć, lub sam wykonać żądaną symulację. Bo jeśli wszystkie wartości zmiennych zachowują się tak samo, to ta symulacja musi być trywialna.

nia P_n w miejscu, w którym wywołano zakończoną właśnie procedurę. Oczywiście nie możemy bezpośrednio posłużyć się np. numerem p jako parametrem instrukcji $y_p := v$, nie wiemy też a priori, gdzie podstawiać wartości pobrane ze stosu itd. Dlatego powyższy „ciąg" jest w istocie skomplikowanym układem instrukcji warunkowych, wybierającym jedną z wielu możliwości.

Część 2: Teraz niech S będzie programem ze stosem o zmiennych $x_1, \ldots, x_k$. Dla uproszczenia załóżmy (bez straty ogólności), że w S jest dokładnie jedna instrukcja *pop* postaci

$$i_0\colon \quad x_1 := \mathit{pop};\ \textbf{go to}\ i_0 + 1,$$

i że wszystkie instrukcje *push* mają postać

$$j\colon \quad \mathit{push}(x_2);\ \textbf{go to}\ j + 1.$$

Także bez straty ogólności można zakładać, że program S zawsze zatrzymuje się z pustym stosem (ćwiczenie). Najpierw pokażemy jak symulować taki program S za pomocą funkcji „wielowymiarowych", wywoływanych tak:

$$(x_1, \ldots, x_k) := P(x_1, \ldots, x_k).$$

Instrukcję *push* jak wyżej zastępujemy wołaniem

$$j\colon \quad (x_1, \ldots, x_k) := \mathit{Push}_j(x_1, \ldots, x_k);\ \textbf{go to}\ i_0 + 1,$$

a instrukcję *pop* zamieniamy na

$$\textbf{stop}(\mathit{top}, x_2, \ldots, x_k).$$

Ciało każdej procedury Push_j zaczyna się tak:

$$\textbf{start}(x_1, \ldots, x_k);\ \mathit{top} := x_2;\ \textbf{go to}\ j + 1,$$

i zawiera poza tym wszystkie instrukcje programu S.

Nasza symulacja oparta jest na takim prostym pomyśle: każde odłożenie kolejnej wartości na stos odpowiada otwarciu nowej aktywacji procedury, która nadaje tę wartość pomocniczej zmiennej *top*. Zdjęcie wartości ze stosu jest symulowane zakończeniem bieżącej aktywacji.

Pozostaje pytanie jak zamienić wielowymiarowe funkcje na zwykłe: można to zrobić wielokrotnie wywołując tę samą procedurę, traktując coraz to inną zmienną jako wynikową. Nic złego się nie stanie, bo nie ma efektów ubocznych. □

15.10 Rozpoznawanie skończoności

Pokażemy, że istnieje program rekurencyjny z równością (a więc i program ze stosem), zatrzymujący się dokładnie na tych danych wejściowych, które generują skończone podstruktury.

W tym celu, dla dowolnej struktury $\mathcal{A}$ i wektora $v = (a_0, \ldots, a_{k-1}) \in \mathcal{A}^k$ zdefiniujemy ciąg b_i^v elementów $\mathcal{A}$ (a w istocie podstruktury generowanej w $\mathcal{A}$ przez $a_0, \ldots, a_{k-1}$). Bez straty ogólności możemy przyjąć, że wszystkie operacje struktury $\mathcal{A}$ mają tę samą liczbę argumentów p. Powiedzmy, że te operacje są ponumerowane: $f_1, \ldots, f_n$.

Jako b_0^v przyjmujemy element a_0. Jeśli $b_0^v, \ldots, b_m^v$ są już określone, to mamy dwa przypadki:

- Istnieje takie j, że $a_j \notin \{b_0^v, \ldots, b_m^v\}$. Wtedy jako b_{m+1}^v bierzemy takie $a_j \notin \{b_0^v, \ldots, b_m^v\}$, że j jest najmniejsze możliwe.

- Nie ma takiego j, tj. wszystkie generatory są już w zbiorze $\{b_0^v, \ldots, b_m^v\}$. Wtedy definiujemy b_{m+1}^v jako $f_r(b_{i_1}^v, \ldots, b_{i_p}^v)$, gdzie $(i_1, \ldots, i_p, r)$ jest najwcześniejszym leksykograficznie ciągiem o tej własności, że $f_r(b_{i_1}^v, \ldots, b_{i_p}^v) \notin \{b_0^v, \ldots, b_m^v\}$.

Oto własności ciągu b_m^v: Jeśli podstruktura generowana w $\mathcal{A}$ przez $a_0, \ldots, a_{k-1}$ jest skończona, to ciąg b_m^v jest skończony i jego wartościami są wszystkie elementy tej podstruktury. Jeśli podstruktura jest nieskończona, to ciąg b_m^v jest nieskończony. W obu przypadkach ciąg ten jest różnowartościowy (bez powtórzeń).

Na rysunku 1 mamy rekurencyjną procedurę NEXT, która w dowolnej strukturze $\mathcal{A}$, dla danych $a_0, \ldots, a_{k-1}, b_m^v$ oblicza b_{m+1}^v. Stąd natychmiast wynika nasze twierdzenie: wystarczy iterować NEXT tak długo, aż nie da się uzyskać nowego elementu (wynik jest równy ostatniej z wartości wejściowych).

Twierdzenie 15.24 *Dla dowolnego k istnieje k-argumentowy program z procedurami P, zatrzymujący się w dowolnej strukturze $\mathcal{A}$ dla danych $a_0, \ldots, a_{k-1}$ wtedy i tylko wtedy, gdy podstruktura generowana w $\mathcal{A}$ przez $a_0, \ldots, a_{k-1}$ jest skończona.*

15.11 Obliczenia w strukturze liczb naturalnych

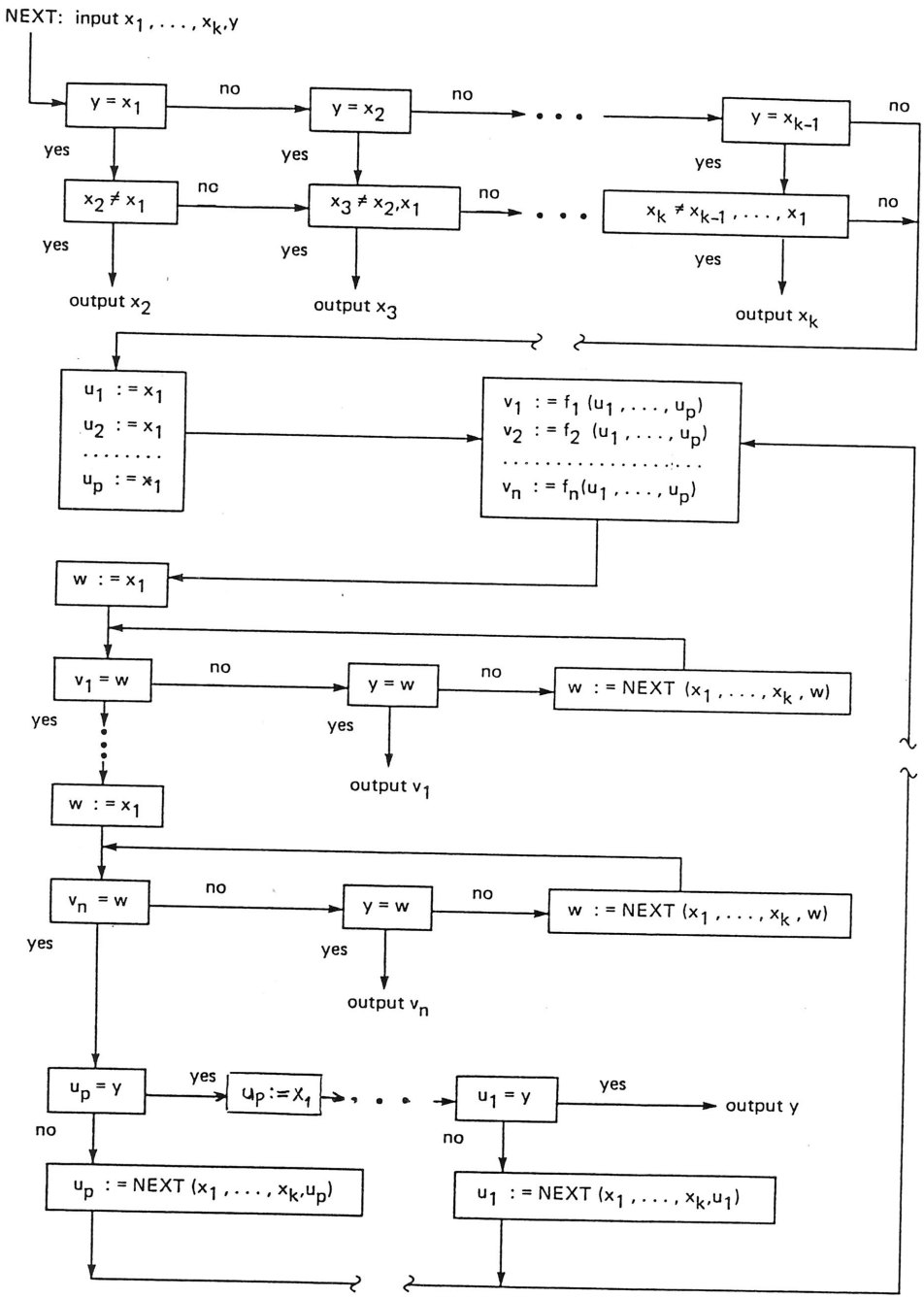
W ogólności rekursja (stos) istotnie zwiększa siłę obliczeniową języka flow-diagramów. Jednak w strukturze liczb naturalnych $\mathcal{N} = \langle \mathbb{N}, s, 0, = \rangle$, gdzie s oznacza funkcję następnika jest inaczej.

Twierdzenie 15.25 *Każdy program ze stosem jest równoważny w strukturze $\mathcal{N}$ pewnemu flow-diagramowi.*

Dowód: Operacje na stosie symulujemy używając dodatkowej zmiennej s, korzystając z funkcji pary $\langle\ ,\ \rangle$ i operacji odwrotnych ℓ, r. Operację $push(x)$ zastępuje $s := \langle s, x \rangle$, a zamiast $x := pop$ wykonujemy $x := r(s);\ s := \ell(s)$. □

Podziękowania

Za poprawki dziękuję Pani Karolinie Sołtys oraz Panom Danielowi Hansowi, Michałowi Kotowskiemu, Łukaszowi Kożuchowskiemu, Dariuszowi Leniowskiemu, Maciejowi Pazurkiewiczowi i Piotrowi Wilkinowi.



Obrazek 1: Program NEXT