Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki
Instytut Informatyki

Praca magisterska

Automaty i programy z licznikami

Arkadiusz Wojna
Warszawa 1998/99

Promotor: dr hab. Paweł Urzyczyn

Za poświęcony czas i cenne uwagi dziękuję
profesorowi Pawłowi Urzyczynowi.

Spis rzeczy

ROZDZIAŁ 1

WSTĘP

Powstanie i rozwój komputerów w drugiej połowie XX wieku spowodował rozwój informatyki teoretycznej, której jednym z zadań jest usystematyzowanie wiedzy na temat obliczania. Aparatem teoretycznym wykorzystywanym w tym celu stał sie matematyczny opis, który pozwala na precyzyjne zdefiniowanie różnych modeli obliczeń oraz jednolite określanie ich właściwości i możliwości obliczeniowych. Szybko okazało się, że rzeczywiste języki używane w programowaniu są bardzo złożonymi obiektami z punktu widzenia ich opisu matematycznego. W związku z tym zaczęto tworzyć dużo prostsze, abstrakcyjne obiekty, w ten sposób jednak, aby zachowały one możliwości obliczeniowe rzeczywistych maszyn. W tym miejscu badania teoretyczne podążyły w kilku różnych kierunkach. Pierwszy, zapoczątkowany przez Alana Turinga [**15**] w 1936 roku, zajmuje się badaniem funkcji obliczanych przez automaty skończone, definiowane poprzez różne modyfikacje uniwersalnego modelu obliczeń na ciagach symboli — maszyny Turinga. Drugi nurt podążył w kierunku uniezależnienia się od dziedziny obliczeń i obejmuje programy definiowane niezależnie od struktury algebraicznej, na której operują. W rezultacie licznych badań wykształciły się definicje wielu klas programów, wśród nich: programy proste, programy z licznikami, stosami, kolejkami, tablicami lub innymi mechanizmami dodatkowymi, programy rekurencyjne, które są równoważne maszynom Turinga w ich dziedzinie obliczeń, oraz definicje efektywne — powszechnie uznawane za intuicyjnie definiujace wszystkie funkcje obliczalne we wszystkich modelach. Oba kierunki zmierzają do tego, aby jak najlepiej zbadać moc obliczeniową różnych modeli obliczeń i ułożyć je w hierarchię, jednocześnie pomagając w zrozumieniu, jak stosowane mechanizmy zwiększają moc obliczeniową, jak zależy ona od ich ilości i które ich elementy mają rzeczywiście istotne znaczenie, a które ułatwiają jedynie notację.

Niniejsza praca obejmuje tematykę badania mocy obliczeniowej modeli zawierających jeden ze wspomnianych mechanizmów matematycznych — liczniki. Rozważa dwa różne rodzaje liczników: pierwszy, który bedziemy nazywać dwukierunkowym, zawiera typowe operacje — zwiększenie i zmniejszenie o 1 oraz test na zero, drugi,

jednokierunkowy, ma swoje źródło w prostej strukturze liczb naturalnych z następującymi operacjami: zwiększenia o 1, zerowania i porównania wartości dwóch liczników. Praca przytacza wszystkie znane fakty dotyczące liczników dwukierunkowych i przytacza hierarchię dla modeli z licznikami tego typu, zarówno w teorii automatów jak i obliczeń w strukturach algebraicznych. Następnie pokazuje, jak dwa liczniki jednokierunkowe mogą symulować dowolną liczbę liczników dwukierunkowych. Twierdzenie to pozwala odnieść przedstawione fakty o licznikach dwukierunkowych do liczników jednokierunkowych.

Struktura niniejszej pracy jest nastepująca. Rozdział drugi podaje niezbędne definicje wszystkich obiektów matematycznych użytych w pracy. Rozdział trzeci i czwarty prezentują znane fakty o automatach licznikowych oraz programach z licznikami dwukierunkowymi, rozdział piąty przedstawia treść oraz dowód twierdzenia, które mówi, że dwa liczniki jednokierunkowe potrafią symulować dowolną liczbę liczników dwukierunkowych. W rozdziale szóstym zawarte są wszystkie ogólne wnioski w kontekście hierarchii automatów i klas programów, które wypływają z tego twierdzenia.

ROZDZIAŁ 2

DEFINICJE

W pierwszej kolejności przedstawimy definicje automatów taśmowych, które umożliwią nam sformułowanie i udowodnienie faktów dotyczących maszyn licznikowych.

DEFINICJA 2.1. Maszyna Turinga

Maszyna Turinga to automat $MT = \langle Q, \Sigma, \Gamma, \delta, q_0, b, F \rangle$ ze skończonym zbiorem stanów Q oraz taśmą i poruszającą się po niej głowicą. Taśma posiada początek z lewej strony i jest prawostronnie nieskończona. Podzielona jest na klatki, które zawierają pojedyncze symbole ze skończonego ustalonego alfabetu Γ. W czasie działania maszyny głowica wskazuje zawsze na jedną klatkę taśmy i może z niej odczytać symbol, zapisać symbol oraz przesunąć się w lewo bądź w prawo do sąsiedniej klatki.

Na początku działania maszyna otrzymuje wejście w postaci ciągu symboli alfabetu Σ umieszczonego na początku taśmy, wszystkie pozostałe klatki zawierają ustalony specjalny symbol $b \in \Gamma \setminus \Sigma$. Głowica jest ustawiona na pierwszą klatkę taśmy i automat znajduje się w stanie początkowym $q_0 \in Q$. W jednym kroku działania maszyna zmienia swój stan, zawartość klatki pod głowicą oraz położenie głowicy zależnie od aktualnego stanu i zawartości klatki. Zmiana ta jest dokonywana zgodnie z funkcją przejść maszyny

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{\leftarrow, \downarrow, \rightarrow\},$$

gdzie $\leftarrow$ i $\rightarrow$ oznaczają przemieszczenie głowicy o jedną klatkę odpowiednio w lewo lub prawo, a symbol $\downarrow$ oznacza, że głowica nie zmienia położenia. Jeśli maszyna znajduje się w stanie $q \in Q$, głowica wskazuje na symbol $a \in \Gamma$ oraz przyjmiemy, że $\delta(q, a) = (q', a', d)$, to w następnym kroku maszyna zapisuje symbol a' na taśmie, przechodzi do stanu q' i przesuwa głowicę zgodnie z kierunkiem d, z wyjątkiem sytuacji, kiedy głowica znajduje się na początku taśmy i kierunek jest $\leftarrow$. Wtedy pozostawia głowicę na pierwszej klatce taśmy.

Maszyna kończy działanie, jeśli znajdzie się w jednym ze stanów końcowych $F \subseteq Q$. Wynikiem działania maszyny jest maksymalny ciąg symboli należących do alfabetu Σ rozpoczynający się od początku taśmy w momencie osiągnięcia stanu końcowego. "Maksymalny" oznacza, że pierwszy symbol za słowem będącym wynikiem

nie należy już do alfabetu Σ. Zauważmy, że maszyna może nigdy nie osiągnąć stanu końcowego, mówimy wtedy, że wynik działania jest nieokreślony.

Funkcją definiowaną przez maszynę Turinga nazywamy funkcję

$$f : \Sigma^* \to \Sigma^* \cup \{\perp\}$$

o wartościach określonych następująco: dla każdego argumentu $w \in \Sigma^*$, jeśli maszyna Turinga uruchomiona na taśmie zawierającej słowo w jako wejście zatrzymuje się, to wartością funkcji $f(w)$ jest słowo bedące wynikiem działania maszyny, w przeciwnym przypadku $f(w) = \perp$.

Posługując się opisem maszyny Turinga w dalszej części pracy często będziemy wykorzystywali opis jej chwilowego stanu. Zauważmy, że w tym celu wystarczy podać aktualny stan, zawartość taśmy oraz położenie głowicy. Niech q — aktualny stan, $a_1 \cdots a_i \cdots a_j$ — zawartość taśmy i głowica wskazuje na i-tą klatkę taśmy. Taki stan chwilowy maszyny będziemy zapisywać $a_1 \cdots a_{i-1} q a_i \cdots a_j$. Aby zapis był jednoznaczny, możemy bez utraty ogólności założyć, że zbiór stanów Q i alfabet Γ są rozłączne.

Maszyny licznikowe, którymi będziemy zajmować się w dalszej części pracy, powstały z modyfikacji maszyn Turinga. Własności taśmy i głowic w tych zmodyfikowanych maszynach zostały tak dobrane, że zachowują się one jak liczniki.

Definicja 2.2. Maszyna licznikowa dwukierunkowa

Maszynę licznikową dwukierunkową $ML = \langle Q, C, \sigma, \delta, q_0, F \rangle$ o k licznikach definiujemy jak maszynę Turinga z tym, że zamiast jednej zwykłej głowicy na taśmie, mamy k głowic licznikowych C, które mają tę własność, że mogą jedynie poruszać się po taśmie, nie mogą natomiast zmieniać jej zawartości. Co więcej zawartość taśmy dla wszystkich maszyn licznikowych jest zawsze ta sama i opiera się na dwóch symbolach alfabetu: z i b. Pierwsza klatka taśmy zawiera symbol z, a wszystkie następne b. Wejściem maszyny licznikowej jest położenie pierwszej głowicy na taśmie, wszystkie pozostałe głowice ustawione są na pierwszej klatce taśmy. W kolejnych krokach maszyna licznikowa działa jak maszyna Turinga z tą różnicą, że głowica, która odczyta symbol w celu określenia zmiany stanu, jest wyznaczana przez funkcję $\sigma : Q \to C$. Funkcja przejścia $\delta : Q \times \{z, b\} \to Q \times C \times \{\leftarrow, \downarrow, \rightarrow\}$ na podstawie aktualnego stanu q oraz zawartość klatki wskazywanej przez głowicę $\sigma(q)$ określa, do jakiego stanu przechodzi automat, wskazuje głowicę, która jest przesuwana oraz kierunek jej przesunięcia. Wynikiem jest położenie pierwszej głowicy na taśmie w momencie osiagnięcia stanu końcowego. Zauważmy, że każdą głowicę możemy traktować jak licznik, którego wartość wyznacza jej odległość od pierwszej

klatki taśmy. Operacje, które wykonujemy, to zwiększenie licznika o 1, zmniejszenie o 1 oraz sprawdzenie czy wartość jest równa 0. Wejście oraz wynik są wtedy wartościami wyznaczanymi przez położenie pierwszej głowicy na taśmie w momencie odpowiednio rozpoczęcia oraz zakończenia działania maszyny. Podobnie jak przy maszynach Turinga maszyny licznikowe mogą się nigdy nie zatrzymać i wtedy wynik jest nieokreślony.

Funkcją definiowaną przez maszynę licznikową dwukierunkową nazywamy funkcję

$$f : \mathbf{N} \to \mathbf{N} \cup \{\perp\},$$

gdzie $\mathbf{N}$ to zbiór liczb naturalnych, okresloną analogicznie jak w maszynie Turinga.

DEFINICJA 2.3. Maszyna licznikowa jednokierunkowa

Maszynę licznikową jednokierunkową $ML = \langle Q, C, \sigma, \delta, q_0, F\rangle$ z k licznikami definiujemy podobnie jak maszynę licznikową dwukierunkową. Ma również jedną taśmę oraz zbiór k głowic C. Różnica polega na tym, że kolejny krok maszyny wyznaczany jest zależnie od tego, czy dwie głowice, ustalone dla stanu automatu $q \in Q$ zgodnie z funkcją $\sigma : Q \to C \times C$, wskazują na tą samą klatkę. Funkcja przejścia $\delta : Q \times \{=, \neq\} \to Q \times C \times \{\ll, \downarrow, \to\}$ umożliwia następujące operacje głowicy: powrót do początku taśmy $\ll$, zachowanie bieżącej pozycji $\downarrow$ oraz przesunięcie o jedną klatkę w prawo $\to$. Zauważmy, że w interpretacji licznikowej porównanie położeń głowic to porównanie wartości liczników, a operacje głowicy to ustawienie wartości licznika na 0, zachowanie poprzedniej wartości licznika oraz zwiększenie licznika o 1.

Funkcję $f : \mathbf{N} \to \mathbf{N} \cup \{\perp\}$ definiowaną przez maszynę licznikową jednokierunkową określamy podobnie jak w maszynie licznikowej dwukierunkowej.

W ten sposób zdefiniowaliśmy dwa modele maszyn licznikowych różniące się zestawem dostępnych działań na licznikach.

Druga część definicji dotyczy teorii obliczeń. W pierwszej kolejności przedstawimy podstawowe definicje teorii modeli algebraicznych oraz logiki pierwszego rzędu wykorzystywane do definicji klas programów oraz obliczeń.

DEFINICJA 2.4. Sygnatura

Sygnaturą nazywamy dowolny skończony zbiór symboli funkcyjnych oraz relacyjnych

$$\Sigma = \bigcup_{n \in N} \Sigma_n \cup \bigcup_{n \in N \setminus \{0\}} \Sigma_n^R,$$

gdzie Σ_n — zbiór symboli funkcyjnych n-argumentowych, Σ_n^R — zbiór symboli relacyjnych n-argumentowych i zbiory te są parami rozłączne.

DEFINICJA 2.5. Struktura algebraiczna

Strukturą algebraiczną lub modelem $\mathcal{A}$ sygnatury $\Sigma = \{f_1, \ldots, f_s, r_1, \ldots, r_t\}$ nazywamy krotkę $\langle A, f_1^{\mathcal{A}}, \ldots, f_s^{\mathcal{A}}, r_1^{\mathcal{A}}, \ldots, r_t^{\mathcal{A}} \rangle$, gdzie A to dowolny zbiór nazywany też nośnikiem struktury, natomiast pozostałe elementy to dowolne funkcje i relacje zachowujące następującą własność. Jeśli f_i jest symbolem funkcyjnym n-argumentowym, to $f_i^{\mathcal{A}}$ jest funkcją n-argumentową

$$f_i^{\mathcal{A}} : A^n \to A$$

oraz jeśli r_i jest symbolem relacyjnym n-argumentowym, to $r_i^{\mathcal{A}}$ jest relacją n-argumentową

$$r_i^{\mathcal{A}} : A^n \to \{true, false\}.$$

DEFINICJA 2.6. Zbiór termów

Zbiorem termów sygnatury $\Sigma = \{f_1, \ldots, f_s, r_1, \ldots, r_t\}$ nad zbiorem zmiennych $Var = \{x_1, \ldots, x_p\}$ nazywamy najmniejszy zbiór napisów $\mathcal{T}_\Sigma^{Var}$ spełniający warunki:

1. $Var \subseteq \mathcal{T}_\Sigma^{Var}$;
2. jeśli $t_1, \ldots, t_n \in \mathcal{T}_\Sigma^{Var}$ i $f \in \Sigma_n$, to $f(t_1, \ldots, t_n) \in \mathcal{T}_\Sigma^{Var}$.

DEFINICJA 2.7. Zbiór formuł otwartych pierwszego rzędu

Zbiorem formuł otwartych pierwszego rzędu sygnatury $\Sigma = \{f_1, \cdots, f_s, r_1, \cdots, r_t\}$ nad zbiorem zmiennych $Var = \{x_1, \cdots, x_p\}$ nazywamy najmniejszy zbiór napisów $\mathcal{F}_\Sigma^{Var}$ spełniający warunki:

1. $\bot \in \mathcal{F}_\Sigma^{Var}$;
2. jeśli $t_1, \cdots, t_n \in \mathcal{T}_\Sigma^{Var}$, $r \in \Sigma_n^R$, to $r(t_1, \ldots, t_n) \in \mathcal{F}_\Sigma^{Var}$;
3. jeśli $t_1, t_2 \in \mathcal{T}_\Sigma^{Var}$, to $t_1 = t_2 \in \mathcal{F}_\Sigma^{Var}$;
4. jeśli $\phi, \psi \in \mathcal{F}_\Sigma^{Var}$, to $\neg\phi, \phi \vee \psi, \phi \wedge \psi, \phi \to \psi \in \mathcal{F}_\Sigma^{Var}$.

DEFINICJA 2.8. Wartościowanie

Wartościowaniem w modelu $\mathcal{A} = \langle A, f_1^{\mathcal{A}}, \ldots, f_s^{\mathcal{A}}, r_1^{\mathcal{A}}, \ldots, r_t^{\mathcal{A}} \rangle$ ustalonej sygnatury $\Sigma = \{f_1, \ldots, f_s, r_1, \ldots, r_t\}$ nad zbiorem zmiennych $Var = \{x_1, \ldots, x_p\}$ nazywamy dowolną funkcję

$$v : \mathcal{T}_\Sigma^{Var} \cup \mathcal{F}_\Sigma^{Var} \to A \cup \{true, false\}$$

spełniającą warunki:

1. $v(f(t_1, \ldots, t_n)) = f^{\mathcal{A}}(v(t_1), \ldots, v(t_n))$ dla każdych $t_1, \ldots, t_n \in \mathcal{T}_\Sigma^{Var}$ i każdego n-argumentowego symbolu funkcyjnego f;

2. $v(\bot) = false$;
3. $v(r(t_1, \cdots, t_n)) = r^{\mathcal{A}}(v(t_1), \ldots, v(t_n))$ dla każdych $t_1, \ldots, t_n \in \mathcal{T}_\Sigma^{Var}$ i każdego n-argumentowego symbolu relacyjnego r;
4. dla każdych $t_1, t_2 \in \mathcal{T}_\Sigma^{Var}$

$$v(t_1 = t_2) = \begin{cases} true & \text{gdy } v(t_1) = v(t_2) \\ false & \text{w przeciwnym przypadku} \end{cases};$$

5. dla każdej pary formuł otwartych pierwszego rzędu $\phi, \psi \in \mathcal{F}_\Sigma^{Var}$

$$v(\neg\phi) = \begin{cases} true & \text{gdy } v(\phi) = false \\ false & \text{w przeciwnym przypadku} \end{cases},$$

$$v(\phi \vee \psi) = \begin{cases} true & \text{gdy } v(\phi) = true \text{ lub } v(\psi) = true \\ false & \text{w przeciwnym przypadku} \end{cases},$$

$$v(\phi \wedge \psi) = \begin{cases} true & \text{gdy } v(\phi) = true \text{ i } v(\psi) = true \\ false & \text{w przeciwnym przypadku} \end{cases},$$

$$v(\phi \to \psi) = \begin{cases} true & \text{gdy } v(\phi) = false \text{ lub } v(\psi) = true \\ false & \text{w przeciwnym przypadku} \end{cases}.$$

W dalszej części pracy będziemy zakładać, że sygnatura $\Sigma = \{f_1, \ldots, f_s, r_1, \ldots, r_t\}$ jest ustalona. Następna część definicji to klasy programów rozważane w pracy.

DEFINICJA 2.9. Program prosty

Programem prostym $fd = (Ins, Var)$ nazywamy zbiór instrukcji Ins posiadających unikalne etykiety i działających na ustalonym skończonym zbiorze zmiennych Var. Każda instrukcja przyjmuje jedną z niżej wyszczególnionych postaci, w których f oznacza dowolny symbol funkcyjny, r — dowolny symbol relacyjny, zmienne x_i to dowolne zmienne ze zbioru Var, a id, id' oraz id'' to dowolne etykiety instrukcji:

(a) $id : START(x_1, \ldots, x_p); GO\,TO\,id'$
(b) $id : x_1 := x_2; GO\,TO\,id'$
(c) $id : x_0 := f(x_1, \ldots, x_n); GO\,TO\,id'$
(d) $id : IF\,x_1 = x_2\,THEN\,id'\,ELSE\,id''$
(e) $id : IF\,r(x_1, \ldots, x_n)\,THEN\,id'\,ELSE\,id''$
(f) $id : STOP(x_0)$

W każdym programie występuje dokładnie jedna instrukcja typu (a) oraz dokładnie jedna instrukcja typu (f). Zmienne $x_1, \ldots, x_p$ występujące w instrukcji typu (a)

nazywamy zmiennymi wejściowymi programu, a x_0 w instrukcji typu (f) zmienną wyjściową.

Program prosty działa w ustalonym modelu $\mathcal{A} = \langle A, f_1^{\mathcal{A}}, \ldots, f_s^{\mathcal{A}}, r_1^{\mathcal{A}}, \ldots, r_t^{\mathcal{A}} \rangle$ w następujący sposób. Jako wejście dostaje ciąg wartości $a_1, \ldots, a_p \in A$, które zostają przypisane odpowiednio na zmienne wejściowe $x_1, \ldots, x_p$ z instrukcji startowej (a), pozostałe zmienne pozostają nieokreślone. Stany obliczeń po kolejnych krokach możemy opisać przez parę $\langle id_i, v_i \rangle$, w której id_i oznacza etykietę następnej instrukcji do wykonania, a $v_i : Var \to A \cup \{\perp\}$ to wartościowanie zmiennych w bieżącym kroku. Symbol $\perp$ oznacza, że wartość zmiennej jest nieokreślona. Stan obliczeń po wykonaniu poszczególnych instrukcji określamy następująco:

1. po instrukcji startowej

$$id : START(x_1, \ldots, x_p); GO\,TO\,id'$$

program przechodzi do instrukcji o etykiecie $id_0 = id'$, a wartościowanie v_0 określamy tak:

$$v_0(x) = \begin{cases} a_i & \text{dla } x = x_i \\ \perp & \text{dla } x \neq x_1, \cdots, x_p \end{cases};$$

2. po instrukcji

$$id : x_1 := x_2; GO\,TO\,id'$$

program przechodzi do instrukcji o etykiecie $id_i = id'$, a wartościowanie v_i określamy tak:

$$v_i(x) = \begin{cases} v_{i-1}(x) & \text{dla } x \neq x_1 \\ v_{i-1}(x_2) & \text{dla } x = x_1 \end{cases};$$

3. po instrukcji

$$id : x_0 := f(x_1, \ldots, x_n); GO\,TO\,id'$$

program przechodzi do instrukcji o etykiecie $id_i = id'$, a wartościowanie v_i określamy tak:

$$v_i(x) = \begin{cases} v_{i-1}(x) & \text{dla } x \neq x_0 \\ f^{\mathcal{A}}(v_{i-1}(x_1), \ldots, v_{i-1}(x_n)) & \text{dla } x = x_0, \text{ gdy } v_{i-1}(x_1), \ldots, v_{i-1}(x_n) \neq \perp \\ \perp & \text{dla } x = x_0, \text{ gdy } v_{i-1}(x_j) = \perp \text{ dla pewnego } j \end{cases};$$

4. po instrukcji

$$id : IF\,x_1 = x_2\,THEN\,id'\,ELSE\,id''$$

etykietą następnej instrukcji w przypadku, gdy $v_{i-1}(x_1), v_{i-1}(x_2) \neq \perp$ oraz $v_{i-1}(x_1) = v_{i-1}(x_2)$, jest $id_i = id'$, w przeciwnym przypadku jest $id_i = id''$, wartości zmiennych pozostają bez zmian:

$$v_i = v_{i-1};$$

5. po instrukcji

$$id : IF\, r(x_1, \ldots, x_n)\, THEN\, id'\, ELSE\, id''$$

etykietą następnej instrukcji w przypadku, gdy $v_{i-1}(x_1), \ldots, v_{i-1}(x_n) \neq \perp$ oraz $r^{\mathcal{A}}(v_{i-1}(x_1), \ldots, v_{i-1}(x_n)) = true$, jest $id_i = id'$, w przeciwnym przypadku jest $id_i = id''$, wartości zmiennych pozostają bez zmian:

$$v_i = v_{i-1};$$

6. po instrukcji końcowej

$$id : STOP(x_0)$$

program kończy działanie, wynikiem programu jest bieżąca wartość zmiennej wyjściowej $v_{i-1}(x_0)$.

Podobnie jak maszyna licznikowa, program prosty może nigdy nie osiągnąć instrukcji końcowej. W tym przypadku analogicznie powiemy, że wynik działania programu jest nieokreślony.

Funkcję definiowaną w modelu $\mathcal{A}$ przez program prosty fd o p zmiennych wejściowych definiujemy jako funkcję

$$fd^{\mathcal{A}} : A^p \to A \cup \{\perp\}$$

w ten sposób, że dla $a_1, \ldots, a_p \in A$ wartością funkcji $fd^{\mathcal{A}}(a_1, \ldots, a_p)$ jest wynik działania programu fd dla tych wartości, jeśli program kończy działanie, w przeciwnym przypadku $fd^{\mathcal{A}}(a_1, \ldots, a_p) = \perp$.

Klasę wszystkich programów prostych oznaczamy FD.

DEFINICJA 2.10. Program z licznikami dwukierunkowymi

Program z k licznikami dwukierunkowymi $fdc = (Ins, Var, Cnt)$ definiujemy podobnie jak program prosty, z tą różnicą, że oprócz zbioru zmiennych Var posiada on zbiór k liczników Cnt i poza instrukcjami typu (a), (b), (c), (d), (e) i (f) może jeszcze zawierać następujące instrukcje licznikowe, w których, jak poprzednio, id, id' oraz id'' oznaczają dowolne etykiety instrukcji, natomiast $c_0 \in Cnt$ oznacza dowolny licznik:

(g) $id: c_0 := c_0 + 1; GO\,TO\,id'$

(h) $id: c_0 := c_0 - 1; GO\,TO\,id'$

(i) $id: IF\,c_0 = 0\,THEN\,id'\,ELSE\,id''$

Kolejne stany obliczeń programu opisywane są tym razem przez trójki $\langle id_i, v_i, vc_i \rangle$, gdzie id_i oraz v_i jak poprzednio oznaczają etykietę nastepnej instrukcji oraz wartości zmiennych Var, natomiast $vc_i : Cnt \to \mathbf{N}$ to wartości liczników w danym kroku obliczeń. Po wszystkich instrukcjach typu (a), (b), (c), (d), (e) i (f) kolejne instrukcje id_i wykonywane przez program oraz wartości zmiennych v_i definiowane są identycznie jak w programie prostym, natomiast wartości liczników po instrukcji startowej przyjmują wartość 0:

$$vc_0(c) = 0 \text{ dla } c \in Cnt$$

i przy wykonywaniu instrukcji nielicznikowych nie zmieniają się:

$$vc_i = vc_{i-1}.$$

Działanie instrukcji licznikowych jest następujące:

1. po instrukcji

   $$id: c_0 := c_0 + 1; GO\,TO\,id'$$

   program przechodzi do instrukcji o etykiecie $id_i = id'$, wartościowanie v_i się nie zmienia:

   $$v_i = v_{i-1},$$

   natomiast wartości liczników vc_i określamy tak:

   $$vc_i(c) = \begin{cases} vc_{i-1}(c) & \text{dla } c \neq c_0 \\ vc_{i-1}(c) + 1 & \text{dla } c = c_0 \end{cases};$$

2. po instrukcji

   $$id: c_0 := c_0 - 1; GO\,TO\,id'$$

   program przechodzi do instrukcji o etykiecie $id_i = id'$, wartościowanie v_i się nie zmienia:

   $$v_i = v_{i-1},$$

   natomiast wartości liczników vc_i określamy tak:

   $$vc_i(c) = \begin{cases} vc_{i-1}(c) & \text{dla } c \neq c_0 \text{ dla } c = c_0 = 0 \\ vc_{i-1}(c) - 1 & \text{dla } c = c_0 > 0 \end{cases};$$

3. po instrukcji

$$id : IF\, c_0 = 0\, THEN\, id'\, ELSE\, id''$$

etykietą następnej instrukcji w przypadku, gdy $vc_{i-1}(c_0) = 0$, jest $id_i = id'$, w przeciwnym przypadku jest $id_i = id''$, wartości zmiennych oraz liczników pozostają bez zmian:

$$\begin{aligned} v_i &= v_{i-1} \\ vc_i &= vc_{i-1}. \end{aligned}$$

Wynikiem programu z licznikami dwukierunkowymi podobnie jak w programie prostym jest wartość zmiennej występującej w instrukcji końcowej (f) w momencie zakończenia programu lub wynik jest nieokreślony, jeśli program nie zatrzymuje się.

Funkcję definiowaną przez program z licznikami dwukierunkowymi w ustalonym modelu $\mathcal{A}$ definiuje się identycznie jak dla programu prostego.

Klasę wszystkich programów z licznikami dwukierunkowymi oznaczamy FDC^{II}, a dla ustalonego k naturalnego klasę wszystkich programów, które mają nie więcej niż k liczników dwukierunkowych, oznaczamy FDC^{II}_k.

DEFINICJA 2.11. Program z licznikami jednokierunkowymi

Program z k licznikami jednokierunkowymi $fdc = (Ins, Var, Cnt)$ definiuje się analogicznie jak program z k licznikami dwukierunkowymi. Różni sie on jedynie zestawem instrukcji licznikowych, które mogą wystepować w programie. Zamiast dodawania (g), odejmowania (h) oraz testu na zero (i), mamy dodawanie (g), zerowanie (j) i porównanie dwóch liczników (k):

(g) $id : c_0 := c_0 + 1;\, GO\, TO\, id'$
(j) $id : c_0 := 0;\, GO\, TO\, id'$
(k) $id : IF\, c_1 = c_2\, THEN\, id'\, ELSE\, id''$

Symbole c_0, c_1 i c_2 oznaczają dowolne liczniki ze zbioru Cnt, a id, id' oraz id'', jak zawsze, etykiety dowolnych instrukcji. Działanie instrukcji (g) jest takie jak w liczniku dwukierunkowym, a dwie pozostałe działają w następujący sposób:

1. po instrukcji

$$id : c_0 := 0;\, GO\, TO\, id'$$

program przechodzi do instrukcji o etykiecie $id_i = id'$, wartościowanie v_i się nie zmienia:

$$v_i = v_{i-1},$$

natomiast wartości liczników vc_i określamy tak:

$$vc_i(c) = \begin{cases} vc_{i-1}(c) & \text{dla } c \neq c_0 \\ 0 & \text{dla } c = c_0 \end{cases} ;$$

2. po instrukcji

$$id : IF\, c_1 = c_2\, THEN\, id'\, ELSE\, id''$$

etykietą następnej instrukcji w przypadku, gdy $vc_{i-1}(c_1) = vc_{i-1}(c_2)$, jest $id_i = id'$, w przeciwnym przypadku jest $id_i = id''$, wartości zmiennych oraz liczników pozostają bez zmian:

$$\begin{aligned} v_i &= v_{i-1} \\ vc_i &= vc_{i-1}. \end{aligned}$$

Klasę wszystkich programów z licznikami jednokierunkowymi oznaczamy FDC^I, a dla ustalonego k naturalnego klasę wszystkich programów, które mają nie więcej niż k liczników jednokierunkowych, oznaczamy FDC^I_k.

Ze względu na zupełnie inną strukturę programów oddzielnie zdefiniujemy klasę definicji efektywnych - uważaną za intuicyjnie definiującą wszystkie funkcje obliczalne w dowolnych strukturach.

DEFINICJA 2.12. Definicja efektywna

Definicją efektywną p-argumentową nazywamy funkcję rekurencyjną $ed : \mathbf{N} \to \mathcal{F} \times \mathcal{T}$, gdzie $\mathcal{F}$ — zbiór formuł otwartych pierwszego rzędu nad zbiorem p zmiennych wejściowych Var, a $\mathcal{T}$ — zbiór termów nad tym samym zbiorem zmiennych Var. Funkcja rekurencyjna, to taka funkcja, dla której istnieje maszyna Turinga wypisująca kolejne pary argument-wartość na taśmie w ten sposób, że dla każdego argumentu odpowiadająca mu para zostanie wypisana dokładnie raz.

Funkcją definiowaną przez p-argumentową definicję efektywną ed w ustalonym modelu $\mathcal{A} = \langle A, f_1^{\mathcal{A}}, \ldots, f_s^{\mathcal{A}}, r_1^{\mathcal{A}}, \ldots, r_t^{\mathcal{A}} \rangle$ nazywamy funkcję $ed^{\mathcal{A}} : A^p \to A \cup \{\perp\}$ określoną następująco: niech $Var = \{x_1, \ldots, x_p\}$ — zbiór zmiennych wejściowych ed, $v_{\overrightarrow{a}}$ — wartościowanie w $\mathcal{A}$ takie, że $v_{\overrightarrow{a}}(x_i) = a_i$ dla $i = 1, \cdots, p$, wtedy

1. jeśli istnieje $n \in \mathbf{N}$ takie, że $ed(n) = \langle \phi, t \rangle$ i $v_{\overrightarrow{a}}(\phi) = true$ oraz n jest najmniejsze spełniające ten warunek, to $ed^{\mathcal{A}}(\overrightarrow{a}) = v_{\overrightarrow{a}}(t)$
2. w przeciwnym przypadku $ed^{\mathcal{A}}(\overrightarrow{a}) = \perp$.

Wynikiem obliczenia definicji efektywnej ed dla wartości wejściowych

$$\overrightarrow{a} = (a_1, \ldots, a_p) \in A^p$$

nazywamy wartość

$$ed^{\mathcal{A}}(a_1, \ldots, a_p).$$

Klasę wszystkich definicji efektywnych oznaczamy ED.

Dotychczas wprowadziliśmy definicje różnych klas programów: FD, FDC_k^I, FDC^I, FDC_k^{II}, FDC^{II}, ED. W kolejnych rozdziałach będziemy pokazywać różne zależności zachodzące pomiędzy klasami, dlatego zdefiniujemy związki, którymi się będziemy posługiwać w opisie hierarchii klas programów.

DEFINICJA 2.13. Sprowadzalność pomiędzy klasami programów

Powiemy, że klasa programów $\mathcal{P}_2$ jest co najmniej tak silna jak klasa programów $\mathcal{P}_1$:

$$\mathcal{P}_1 \preceq \mathcal{P}_2$$

wtedy i tylko wtedy, gdy dla każdego p-argumentowego programu $P_1 \in \mathcal{P}_1$ istnieje p-argumentowy program $P_2 \in \mathcal{P}_2$ taki, że w każdym modelu

$$\mathcal{A} = \left\langle A, f_1^{\mathcal{A}}, \ldots, f_s^{\mathcal{A}}, r_1^{\mathcal{A}}, \ldots, r_t^{\mathcal{A}} \right\rangle$$

dla każdych wartości wejściowych $\overrightarrow{a} = (a_1, \ldots, a_p) \in A^p$

1. albo wyniki obliczeń P_1 i P_2 są oba określone i równe

$$P_1^{\mathcal{A}}(\overrightarrow{a}) = P_2^{\mathcal{A}}(\overrightarrow{a});$$

2. albo oba nieokreślone

$$P_1^{\mathcal{A}}(\overrightarrow{a}) = \bot \wedge P_2^{\mathcal{A}}(\overrightarrow{a}) = \bot.$$

DEFINICJA 2.14. Równoważność pomiędzy klasami programów

Powiemy, że klasy $\mathcal{P}_1$, $\mathcal{P}_2$ są równoważne:

$$\mathcal{P}_1 \equiv \mathcal{P}_2,$$

jeśli $\mathcal{P}_1 \preceq \mathcal{P}_2$ i $\mathcal{P}_2 \preceq \mathcal{P}_1$.

DEFINICJA 2.15. Ostra sprowadzalność pomiędzy klasami programów

Powiemy, że klasa $\mathcal{P}_2$ jest istotnie silniejsza od klasy $\mathcal{P}_1$:

$$\mathcal{P}_1 \prec \mathcal{P}_2$$

wtedy i tylko wtedy, jeśli $\mathcal{P}_1 \preceq \mathcal{P}_2$ oraz istnieje program $P_2 \in \mathcal{P}_2$ taki, że żaden program z klasy $\mathcal{P}_1$ nie definiuje identycznej funkcji we wszystkich strukturach, tzn.

dla każdego $P_1 \in \mathcal{P}_1$ istnieje struktura $\mathcal{A}$, w której funkcje definiowane przez oba programy się różnią:

$$P_1^{\mathcal{A}} \not\equiv P_2^{\mathcal{A}}.$$

ROZDZIAŁ 3

MASZYNY LICZNIKOWE DWUKIERUNKOWE

W pierwszej części tego rozdziału będziemy zmierzali do tego, aby pokazać, że maszyna z 2 licznikami dwukierunkowymi potrafi symulować maszynę Turinga, wynik sformułowany i udowodniony przez Marvina Minsky'ego [**11**].

Aby jednak można było mówić o symulacji jednej maszyny przez drugą, musimy rozwiązać problem różnych dziedzin wartości wejściowych, gdyż maszyna Turinga operuje na słowach, natomiast maszyna licznikowa na liczbach naturalnych. Niech $MT = \langle Q, \Sigma, \Gamma, \delta, q_0, b, F\rangle$ — maszyna Turinga. Skończone słowo $w = w_1 w_2 \cdots w_n$ zapisane na taśmie maszyny MT możemy interpretować jako zapis liczby naturalnej w systemie liczbowym o podstawie $m = |\Gamma|$, jeśli ustalimy wzajemnie jednoznaczne przyporządkowanie $\sigma : \Gamma \to \{0, \ldots, m-1\}$:

$$\sigma^*(w) = \sigma(w_1) + m\sigma(w_2) + \cdots + m^{n-1}\sigma(w_n).$$

Aby można było traktować słowa wejściowe dla maszyn Turinga jako skończone, ustalmy, że symbol pusty b ma kod $\sigma(b) = 0$.

Na początku pokażemy, że maszyna licznikowa potrafi zasymulować proste operacje maszyny Turinga na liczniku c_1 zawierającym kod ciągu symboli przy pomocy jednego dodatkowego licznika c_2.

Dodanie nowego symbolu na poczatek ciągu realizujemy w ten sposób, że przepisujemy wartość c_1 na c_2 zerując jednocześnie c_1, następnie na c_1 wstawiamy wartość c_2 pomnożoną przez m w ten sposób, że przy każdym zmniejszeniu c_2 o 1 zwiększamy c_1 o m. Wykonujemy to, aż c_2 osiągnie 0. Na koniec dodajemy do c_1 wartość będacą kodem nowego symbolu.

Usunięcie symbolu z początku ciągu realizujemy podobnie. Przepisujemy c_1 na c_2 zerując c_1, następnie próbujemy zmniejszyć c_2 o m i jeśli nam się uda, to zwiększamy c_1 o 1, jeśli zaś c_2 osiągnie 0, to kończymy działanie.

Aby rozpoznać, jaki symbol jest zakodowany na początku ciągu symboli w, musimy policzyć $\sigma^*(w) \mod m$, wynik będzie kodem pierwszego symbolu. Robimy to w ten sposób, że przepisujemy c_1 na c_2 zerując c_1, a następnie w cyklu przechodzimy po m stanach maszyny, zwiększając w każdym kroku c_1 o 1 i zmniejszając c_2 o 1. Jeśli w momencie wyzerowania c_2 osiągniemy j-ty stan w cyklu m stanów, to j jest

kodem pierwszego symbolu ze słowa w. Zauważmy, że w tym momencie na c_1 znów bedziemy mieć taką wartość, jak przed wyliczaniem kodu.

LEMAT 3.1. *Dla każdej maszyny Turinga MT istnieje taka maszyna trójlicznikowa dwukierunkowa ML, że uruchomiona z kodem dowolnego słowa wejściowego dla maszyny Turinga, symuluje jej działanie i albo na wyjściu zwraca kod wyniku działania maszyny Turinga, jeśli ta kończy działanie, albo podobnie jak maszyna Turinga nie zatrzymuje się.*

DOWÓD. Dla ustalonej maszyny Turinga MT zdefiniujemy, jak będzie działać maszyna trójlicznikowa. Jeśli przyjmiemy, że bieżący stan obliczeń MT ma postać

$$a_1 \cdots a_{i-1} q a_i \cdots a_j,$$

to na pierwszym liczniku c_1 będziemy przechowywać liczbę kodującą ciąg symboli zaczynający się od klatki wskazywanej przez bieżące położenie głowicy czyli

$$c_1 = \sigma(a_i) + m\sigma(a_{i+1}) + \cdots + m^{j-i}\sigma(a_j),$$

a na liczniku c_2 będziemy przechowywać liczbę kodującą ciąg symboli przed głowicą taśmy, z tą różnicą, że będziemy go traktować jako zapis liczby od końca, czyli

$$c_2 = \sigma(a_{i-1}) + m\sigma(a_{i-2}) + \cdots + m^{i-2}\sigma(a_1).$$

Trzeci licznik c_3 będzie służył jako pomocniczy do symulacji przejść maszyny MT. Ponieważ początkowe położenie głowicy w maszynie Turinga jest na pierwszej klatce taśmy, więc wartość licznika c_1 odpowiada kodowi słowa wejściowego maszyny Turinga MT. Bez utraty ogólności możemy przyjąć, że maszyna MT, którą będziemy symulować, po obliczeniu wyniku na taśmie, najpierw zamienia wszystkie symbole za słowem wynikowym na symbole puste b, a następnie wraca głowicą na początek taśmy. W ten sposób, na końcu działania maszyny licznikowej, licznik c_1 będzie zawierał dokładnie kod słowa wynikowego MT.

Teraz musimy pokazać, jak maszyna licznikowa symuluje kroki maszyny Turinga, w taki sposób, by liczniki c_1 i c_2 zawierały zawsze aktualny opis stanu obliczeń maszyny MT. Załóżmy, że maszyna licznikowa musi zasymulować przejście maszyny Turinga opisane w następujący sposób:

$$\delta(q, a) = (q', a', d).$$

Stan symulowanej maszyny Turinga będziemy pamiętać na stanach maszyny licznikowej, tzn. każdemu stanowi q maszyny Turinga będzie odpowiadać osobna grupa stanów w maszynie licznikowej, która będzie symulować przejście z tego stanu q. Robimy to w następujący sposób. Najpierw rozpoznajemy, jaki symbol jest pod głowicą przez rozponanie pierwszego symbolu słowa kodowanego przez c_1. Wtedy

maszyna już wie, że ma zapisać symbol a', poruszyć się w kierunku d i przejść do stanu q'. W tym celu symulujemy usunięcie pierwszego symbolu z c_1 i dołączenie symbolu a' na początek, następnie, jeśli mieliśmy poruszyć sie w lewo, symulujemy usunięcie pierwszego symbolu z c_2 i dodanie go do c_1, jeśli zaś w prawo, to robimy odwrotnie. Na końcu symulujemy przejście do stanu q'.

Przy wszystkich symulacjach tych prostych operacji używamy c_3 jako licznika pomocniczego. □

TWIERDZENIE 3.2. *Dla każdej maszyny Turinga MT istnieje taka maszyna dwulicznikowa dwukierunkowa ML, że jeśli na wejściu dostaje wartość $2^{\sigma^*(w)}$, gdzie w —dowolne słowo wejściowe dla maszyny MT, to symuluje działanie maszyny MT i albo na wyjściu zwraca wartość $2^{\sigma^*(u)}$, jeśli ta kończy działanie z wynikiem u, albo podobnie jak maszyna MT nie zatrzymuje się.*

DOWÓD. Pokażemy, że maszyna dwulicznikowa potrafi symulować maszynę trójlicznikową. Niech ML' — maszyna trójlicznikowa symulujacą maszynę MT. Załóżmy, że c_1, c_2 i c_3 — liczniki maszyny ML' oraz c_4 i c_5 — liczniki maszyny ML. W każdym kroku symulacji maszyny ML' licznik c_4 będzie przechowywać wartości liczników c_1, c_2 i c_3 w wykładnikach liczb pierwszych przypisanych do tych liczników, występujących w rozkładzie wartości c_4 na czynniki pierwsze:

$$c_4 = 2^{c_1}3^{c_2}5^{c_3}.$$

Zauważmy, że w momencie rozpoczęcia symulacji własność ta jest zachowana, gdyż $c_1 = \sigma^*(w)$, $c_2 = 0$ i $c_3 = 0$. Oznaczmy przez p_i liczbę pierwszą związaną z i-tym licznikiem: 2 dla c_1, 3 dla c_2 i 5 dla c_3. Poszczególne operacje maszyny trójlicznikowej na i-tym liczniku będziemy symulować następująco. Sprawdzenie, czy wartość c_i jest 0, realizujemy tak. W każdym kroku zmniejszamy c_4 o 1 jednocześnie zwiększając c_5 o 1. Wykonujemy to w cyklach p_i kroków. Jeśli c_4 osiągnie wartość 0 w środku cyklu, to oznacza, że jego początkowa wartość nie dzieliła się przez p_i, a więc zapamiętujemy na stanach maszyny, że test c_i na 0 wypadł pozytywnie, jeśli zaś osiągnie 0 na końcu cyklu, to pamiętamy, że test na 0 był negatywny. Następnie odtwarzamy poprzednią wartość licznika c_4 z c_5. Aby zasymulować podniesienie i-tego licznika o 1, musimy pomnożyć c_4 przez p_i. Realizujemy to przez przepisanie c_4 na c_5 i wyzerowanie c_4, a następnie przy każdorazowym zmniejszeniu c_5 o 1 zwiększamy c_4 o p_i, aż c_5 osiągnie wartość 0. Zmniejszenie i-tego licznika symulujemy tak: najpierw robimy test c_i na 0 i jeśli pozytywny, to nic więcej nie robimy, bo odejmowanie na wyzerowanym liczniku nic nie robi, jeśli zaś negatywny, to dalej analogicznie jak przy podniesieniu licznika, tylko teraz musimy podzielić c_4 przez p_i. Zatem przepisujemy c_4 na c_5 zerując c_4 i w każdym kroku zmniejszamy c_5 o p_i zwiekszając jednocześnie c_4 o 1.

W ten sposób potrafimy symulować operacje maszyny trójlicznikowej. Zauważmy, że przy symulacji maszyny Turinga MT maszyna trójlicznikowa ML' kończy działanie z wyzerowanymi licznikami c_2 i c_3. To oznacza, że po zakończeniu działania maszyny dwulicznikowej ML, wartość c_4 będzie równa 2^{c_1} i w ten sposób udowodniliśmy twierdzenie 3.2. □

Formułując powyższe twierdzenie założyliśmy, że maszyna dwulicznikowa dostaje wartość odpowiedniej funkcji kodującej słowa wejściowe maszyn Turinga. Pokażemy, że można efektywnie obliczać wartość tej funkcji ze słowa wejściowego, a także generować słowo wyjściowe z wartości takiej funkcji używając taśm: wejściowej i wyjściowej, o bardzo ograniczonych możliwościach.

Zauważmy, że wystarczy pokazać, jak maszyna trójlicznikowa ze słowa wejściowego w dla maszyny Turinga wylicza wartość $\sigma^*(w)$ na swoim pierwszym liczniku, a nastepnie w jaki sposób generuje słowo wyjściowe u z wartości $\sigma^*(u)$. Wtedy maszyna dwulicznikowa, symulując maszynę trójlicznikową identycznie jak to jest opisane w dowodzie twierdzenia 3.2, będzie umiała ze słowa wejściowego w wyliczyć wartość $2^{\sigma^*(w)}$ na swoim pierwszym liczniku, a na koniec działania z wartości $2^{\sigma^*(u)}$ wygenerować słowo wyjściowe u.

Skoncentrujmy sie zatem na maszynie trójlicznikowej. Jak już wspomnieliśmy, dodajemy jej dwie taśmy: taśmę wejściową, którą możemy jedynie czytać i po której możemy się poruszać jedynie w prawo czytając każdy symbol tylko raz, oraz taśmę wyjściową, którą możemy jedynie zapisywać i po której również możemy poruszać się jedynie w prawo każdą klatkę zapisując tylko raz. Przyjmijmy, że na wejściu, zamiast kodu słowa wejściowego $w = w_1 w_2 \cdots w_n$ na pierwszym liczniku, dostajemy jego oryginalną postać na taśmie wejściowej, natomiast wszystkie liczniki mają wartość początkową 0.

Przyjmując, że $m = |\Gamma|$, najpierw na liczniku c_2 obliczymy wartość

$$\sigma^*(w_n w_{n-1} \cdots w_1) = \sigma(w_n) + m\sigma(w_{n-1}) + \cdots + m^{n-1}\sigma(w_1).$$

Zauważmy, że jeśli na c_1 będzie 0, to ten stan symuluje stan maszyny Turinga, która przeczytała słowo wejściowe i głowice ma ustawioną na pierwszej klatce za słowem wejściowym. Wtedy wystarczy, że zasymulujemy powrót głowicy na początek taśmy i otrzymamy wartość $\sigma^*(w)$ na liczniku c_1.

Wartość $\sigma^*(w_n w_{n-1} \cdots w_1)$ na c_2 obliczamy w następujący sposób. Czytamy pierwszy symbol w i wyliczamy jego kod na c_2. Następnie w każdym kroku, czytamy kolejny symbol z taśmy wejściowej, i jeśli należy do słowa wejściowego, to symulujemy operację dodania symbolu na początek słowa zakodowanego na c_2. Kończymy,

gdy z taśmy wejściowej odczytamy symbol pusty b. Potem musimy jeszcze zasymulować powrót głowicy na początek taśmy.

Generowanie słowa wyjściowego z wartości c_1 wykonujemy poprzez proces odwrotny. W każdym kroku rozpoznajemy kod pierwszego symbolu ze słowa zakodowanego na c_1, wypisujemy symbol na taśmie wyjściowej i symulujemy usunięcie pierwszego symbolu z kodowanego słowa. Kończymy, kiedy c_1 osiagnie wartość 0.

Pokazaliśmy, że maszyny dwulicznikowe potrafią symulować maszyny Turinga. Posiadają one jeszcze jedną ważną cechę maszyn Turinga. Problem stopu dla maszyn dwulicznikowych jest nierozstrzygalny [**11**]. Aby udowodnić ten fakt, musimy najpierw precyzyjnie zdefiniować dwie rzeczy. Po pierwsze, w jaki sposób bedziemy kodować maszynę licznikową na taśmie maszyny Turinga, po drugie, jak interpretować maszynę Turinga jako maszynę rozwiązującą problem decyzyjny.

Załóżmy, że w alfabecie maszyny Turinga mamy symbole 0 i 1. Liczby naturalne możemy kodować jako ciąg zer i jedynek. Możemy przyjąć, że stany kodowanej maszyny licznikowej to są kolejne liczby naturalne od 0 do $n-1$, a stany akceptujące, to m najmniejszych liczb naturalnych (oczywiście $m \leq n$). Symbole $\leftarrow$, $\rightarrow$ i 0 niech oznaczają odpowiednio operacje opuszczenia, podniesienia i testu licznika na 0. Maszynę licznikową kodujemy następująco:

$$\#n\#m\#k\#(0, j_0, op_0, p_0, q_0)(1, j_1, op_1, p_1, q_1)\cdots(n-1, j_{n-1}, op_{n-1}, p_{n-1}, q_{n-1})\#.$$

Kolejne fragmenty kodu oddzielamy specjalnym symbolem $\#$. Najpierw kodujemy liczbę stanów n, potem liczbę stanów akceptujących m, nastepnie liczbę liczników maszyny k, a dalej kodujemy operacje maszyny wykonywane w poszczególnych stanach jako ciąg krotek (i, j_i, op_i, p_i, q_i). Każdemu stanowi maszyny od 0 do $n-1$ odpowiada dokładnie jedna krotka, w której i to numer tego stanu, j_i to numer licznika, na którym jest wykonywana operacja, op_i to symbol wykonywanej operacji, a p_i i q_i to numery stanów, do których maszyna przechodzi w następnym kroku: do stanu p_i w przypadku operacji $\leftarrow$, $\rightarrow$ lub gdy test na 0 jest pozytywny, a do stanu q_i, gdy operacja op_i jest testem na 0 i wynik jest negatywny.

Umiemy już kodować maszynę licznikową, omówimy teraz, jak będziemy interpretować maszynę Turinga jako maszynę rozwiązującą problem decyzyjny. Przyjeliśmy, że w alfabecie maszyny występuje symbol 1. Jeśli maszyna się zatrzymuje, to powiemy, że odpowiada: tak, jeśli na pierwszej klatce taśmy zwraca 1, a odpowiada: nie, jeśli na pierwszej klatce znajduje się dowolny inny symbol.

Możemy teraz problem stopu dla maszyn dwulicznikowych sformułować precyzyjnie: czy istnieje maszyna Turinga, która dostawszy na wejściu kod maszyny dwulicznikowej ML oraz liczbę naturalną n odpowiada: tak, jeśli maszyna ML zatrzymuje

się dla wejścia n, i odpowiada: nie, jeśli maszyna ML nie zatrzymuje się dla wejścia n.

Nieistnienie takiej maszyny można w prosty sposób pokazać opierając się na znanym fakcie o nierozstrzygalności problemu stopu dla maszyn Turinga [**15**].

Po drodze posłużymy się dodatkowym lematem:

LEMAT 3.3. *Nie istnieje maszyna Turinga, która na wejściu dostawszy kod maszyny trójlicznikowej dwukierunkowej ML oraz liczbę naturalną n odpowiada: tak, jeśli maszyna ML zatrzymuje się dla wejścia n, a odpowiada: nie, jeśli maszyna ML nie zatrzymuje się dla wejścia n.*

DOWÓD. Załóżmy, że taka maszyna istnieje. Jeśli pokażemy, że istnieje maszyna Turinga, która kod dowolnej maszyny Turinga MT zamienia na kod maszyny licznikowej ML symulującej maszynę MT, a słowo wejściowe na kod tego słowa, to po połączeniu tych dwóch maszyn dostalibyśmy maszynę rozstrzygającą problem stopu dla maszyn Turinga.

Maszynę Turinga kodujemy podobnie jak maszynę licznikową. Najpierw liczbę stanów n, potem liczbę stanów akceptujących m, liczbę symboli alfabetu k i na końcu funkcję przejść δ:

$$\#n\#m\#k\#(0,0,s_{0,0},q_{0,0})\cdots(n-1,k-1,s_{n-1,k-1},q_{n-1,k-1})\#.$$

W tym miejscu musimy przyjąć, że symbole alfabetu będziemy również kodować przez kolejne liczby naturalne. Tym razem krotki $(i,j,s_{i,j},q_{i,j})$ przebiegają po wszystkich kombinacjach stanów i od 0 do $n-1$ i symboli j od 0 do $k-1$. Każda krotka określa nowy symbol $s_{i,j}$, który ma być zapisany na miejsce poprzedniego, oraz stan $q_{i,j}$, do którego maszyna Turinga przechodzi w nastepnym kroku.

Kod maszyny trójlicznikowej symulującej zakodowaną maszynę Turinga będziemy budować zgodnie z ideą zawartą w dowodzie lematu 3.1. Poszczególne stany maszyny Turinga będziemy zamieniać na duże fragmenty maszyny licznikowej, a te będą składane z mniejszych fragmentów symulujących proste operacje składające się na cały krok maszyny Turinga. Te podstawowe operacje to dodanie, usunięcie oraz rozpoznanie symbolu z początku ciągu. Dodanie symbolu to pętla stanów, w której przepisujemy wartość z pierwszego licznika c_1 na drugi c_2 testując, czy c_1 się wyzerował, potem znów pętla, w której mnożymy wartość c_2 przez k odkładając ją z powrotem na c_1, na końcu ciąg stanów, który dodaje do c_1 kod dodawanego symbolu. Usunięcie symbolu to na początku tak samo — pętla przepisująca c_1 na c_2, potem w drugiej pętli zmniejszamy wartość c_2 k razy przepisując ją z powrotem na c_1. Rozpoznanie symbolu to znów przepisanie c_1 na c_2, a potem w pętli k stanów

przywracamy początkową wartość c_1 pamiętając, w którym z nich skończyliśmy. W ten sposób na stanach pamiętamy, jaki symbol rozpoznaliśmy.

Zauważmy, że w niektórych pętlach liczba stanów jest stała, a w niektórych jest zależna tylko od liczby symboli alfabetu kodowanej maszyny. Zatem mając daną liczbę symboli wszystkie te operacje możemy efektywnie wygenerować w postaci kodu stanów i operacji maszyny trójlicznikowej, a te możemy połączyć w kod symulujący całe kroki maszyny Turinga, które z kolei możemy połączyć w kod maszyny trójlicznikowej symulującej całą maszynę Turinga. Ostateczną liczbę stanów musimy policzyć po wygenerowaniu wszystkich fragmentów maszyny. Stany akceptujące maszyny trójlicznikowej zaznaczamy zgodnie z odpowiadającymi im stanami akceptującymi maszyny Turinga. Potem musimy przenumerować je tak, aby spełniały założenie, że zbiór stanów akceptujących tworzą najmniejsze liczby naturalne.

Równie prosto możemy efektywnie zamienić słowo wejściowe na jego kod. Jest to po prostu zamiana liczby zapisanej w systemie liczbowym o podstawie k na liczbę zapisaną w systemie dwójkowym.

Ponieważ wszystkie operacje opisane powyżej można wykonać efektywnie, więc istnieje maszyna Turinga, która potrafi zamieniać kod dowolnej maszyny Turinga na kod maszyny trójlicznikowej, a słowo wejściowe na kod tego słowa, więc gdybyśmy mieli maszynę Turinga rozstrzygającą problem stopu dla maszyn trójlicznikowych, to potrafilibyśmy zbudować maszynę Turinga rostrzygającą problem stopu dla maszyn Turinga. □

TWIERDZENIE 3.4. *Nie istnieje maszyna Turinga, która na wejściu dostawszy kod maszyny dwulicznikowej dwukierunkowej ML oraz liczbę naturalną n odpowiada: tak, jeśli maszyna ML zatrzymuje się dla wejścia n, a odpowiada: nie, jeśli maszyna ML nie zatrzymuje się dla wejścia n.*

DOWÓD. Schemat dowodu jest analogiczny jak w dowodzie lematu 3.3. Różnica polega na tym, że teraz pokazujemy, jak efektywnie zamienić kod maszyny trójlicznikowej na kod symulującej ją maszyny dwulicznikowej. W ten sposób, jeśli istniałaby maszyna Turinga rostrzygająca problem stopu dla maszyn dwulicznikowych, to potrafilibyśmy przy jej pomocy zbudować maszynę Turinga rozstrzygająca problem stopu dla maszyn trójlicznikowych, co byłoby sprzeczne z lematem 3.3.

Przypomnijmy, że maszyna dwulicznikowa przechowuje wartość liczników maszyny trójlicznikowej c_1, c_2 i c_3 w następującej postaci

$$2^{c_1}3^{c_2}5^{c_3}.$$

Zauważmy, że operacje podniesienia, opuszczenia i testu licznika na 0 w maszynie trójlicznikowej maszyna dwulicznikowa symuluje odpowiednio poprzez pomnożenie przez ustaloną liczbę pierwszą, podzielenie przez nią i sprawdzenie podzielności wartości licznika przez tą liczbę. Są to proste pętle o niezmiennej liczbie stanów, wystarczy więc zamienić każdą operacje maszyny trójlicznikowej na odpowiadającą jej pętlę maszyny dwulicznikowej i uzupełnić je o operacje przejść pomiędzy nimi zgodnie z przejściami maszyny trójlicznikowej. Podobnie jak poprzednio, na końcu musimy zliczyć liczbę wszystkich stanów oraz odpowiednio je przenumerować, tak, aby spełniały założenia kodu maszyny licznikowej. Zamiana kodu słowa wejściowego polega na wyliczeniu wartości 2^{c_1} na podstawie wartości c_1. □

W ten sposób przedstawiliśmy najważniejsze znane fakty na temat maszyn licznikowych dwukierunkowych: wystarczalność dwóch liczników do symulacji maszyny Turinga oraz nierozstrzygalność problemu stopu już dla maszyn dwulicznikowych. Pozwalają one nam posługiwać się maszyną dwulicznikową dwukierunkową jako uniwersalnym modelem obliczeń przy odpowiednio zakodowanych danych wejściowych.

ROZDZIAŁ 4

PROGRAMY Z LICZNIKAMI DWUKIERUNKOWYMI

Dziedziną maszyn Turinga jest zawsze zbiór słów nad skończonym alfabetem, dziedziną maszyn licznikowych — liczby naturalne. W tym rozdziale zajmiemy się inną, bardziej ogólną i abstrakcyjną koncepcją obliczeń wykorzystującą mechanizm liczników dwukierunkowych. Będziemy badać moc obliczeniową modeli obliczeń operujących na dowolnych strukturach algebraicznych. Tego typu modelem jest program z licznikami dwukierunkowymi. Pokażemy, jak ilość liczników, z której program może korzystać, wpływa na jego możliwości obliczeniowe i przedstawimy zależności pomiędzy różnymi klasami programów, które wprowadzają usystematyzowaną hierarchię obrazującą moc obliczeniową każdej z nich. To w tym celu przedstawilismy definicje klas programów prostych FD oraz definicji efektywnych ED.

W tym rozdziale będziemy zakładać, że sygnatura $\Sigma = \{f_1, \ldots, f_s, r_1, \ldots, r_t\}$ jest ustalona.

Najpierw pokażemy twierdzenie Garlanda i Luckhama [**4**], które mówi, że liczniki dwukierunkowe istotnie zwiekszają moc obliczeniową programów.

TWIERDZENIE 4.1. *Klasa programów z licznikami dwukierunkowymi FDC^{II} jest istotnie silniejsza od klasy programów prostych FD*

$$FD \prec FDC^{II}.$$

DOWÓD. Związek

$$FD \preceq FDC^{II}$$

jest oczywisty, ponieważ kazdy program prosty jest również programem z licznikami.

Musimy zbudować program z licznikami o takiej własności, której nie ma żaden program prosty.

Najpierw pokażemy, że klasa programów prostych FD ma rozstrzygalny problem stopu na skończonych strukturach.

Program prosty kodujemy na taśmie jako ciąg instrukcji z etykietami, natomiast skończoną strukturę w ten sposób, że podajemy ilość elementów, a następnie po kolei kodujemy wszystkie funkcje i relacje wyliczając wszystkie możliwe kombinacje argumentów z wartościami.

Załóżmy, że program prosty ma k instrukcji i n zmiennych, a struktura ma c elementów. Problem stopu rozwiązujemy następująco. Symulujemy działanie programu przez $kc^n + 1$ kroków, jeśli symulacja zakończyła się, to odpowiadamy: tak, jeśli nie, to odpowiadamy: nie. Zauważmy, że istnieje tylko kc^n róznych stanów programu, więc w $kc^n + 1$ krokach ten sam stan musi powtórzyć się dwa razy. A to wskazuje, że program będzie sie pętlił.

Zdefiniujemy teraz ciąg struktur algebraicznych nad jednoelementową sygnaturą $\Sigma = \{f\}$. Niech $\mathcal{A}_n = \langle \{0, \ldots, n-1\}, f_n \rangle$ z funkcjami określonymi następująco:

$$f_n(i) = i \mod n,$$

a zbiór $\mathbf{Z}$ niech będzie zbiorem rekurencyjnie przeliczalnym, ale nie rekurencyjnym. Nie istnieje program prosty, który dla dowolnych wartości wejściowych zatrzymuje się dokładnie tylko w tych strukturach $\mathcal{A}_n$, dla których $n \in \mathbf{Z}$. Gdyby tak było, to zbiór $\mathbf{Z}$ byłby rekurencyjny. Niech fd — program prosty zatrzymujący się dokładnie tylko w tych strukturach $\mathcal{A}_n$, dla których $n \in \mathbf{Z}$. Maszyna Turinga rozpoznająca zbiór $\mathbf{Z}$ działałaby tak. Najpierw z liczby wejściowej n generuje kod interpretacji $\mathcal{A}_n$, następnie kod programu fd, a potem uruchamia maszynę rozwiązującą problem stopu dla programów prostych i odpowiada tak samo.

Pokażemy, że istnieje program z licznikami dwukierunkowymi posiadający tę szczególną własność, że zatrzymuje się tylko w strukturach $\mathcal{A}_n$, dla których $n \in \mathbf{Z}$.

W lemacie 3.1 pokazaliśmy, że maszyny licznikowe dwukierunkowe potrafią symulować maszyny Turinga. Zatem maszyny licznikowe równie dobrze jak maszyny Turinga potrafią wyliczać kolejne elementy zbiorów rekurencyjnych. Ale programy z licznikami mogą wykonywać dokładnie te same operacje na licznikach co maszyny na taśmach licznikowych, więc one również potrafią generować zbiory rekurencyjne.

Korzystając z tego możemy już zdefiniować nasz program z licznikami. Najpierw na liczniku c_1 wylicza on wielkość struktury $\mathcal{A}_n$, na której operuje, obliczając kolejne wartości $f_n^k(i)$ aż do momentu, gdy po raz drugi natknie sie na wartość i. Następnie generuje kolejne liczby ze zbioru $\mathbf{Z}$ i sprawdza, czy są równe wartości c_1, używając dodatkowego licznika. Jeśli znajdzie taką liczbę, zatrzymuje się. □

Wykazaliśmy ogólny fakt, że programy z licznikami dwukierunkowymi potrafią liczyć więcej niż programy proste. Teraz pokażemy twierdzenie Plaisteda [**13**], które mówi, że dodanie jednego licznika do programów prostych nie zmienia ich mocy obliczeniowej. Twierdzenie to zostało udowodnione dla programów bez instrukcji porównania wartości zmiennych. Ponieważ programy zdefiniowane w tej pracy dopuszczają tą instrukcję, więc nam wystarczy prostsza wersja tego twierdzenia. Wpierw jednak pokażemy jeden lemat.

LEMAT 4.2. *Niech sygnatura $\Sigma = \{f, r\}$ zawiera symbol funkcyjny jednoargumentowy f i symbol relacyjny jednoargumentowy r. Zdefiniujmy program z jednym licznikiem dwukierunkowym $fdc \in FDC_1^{II}$ następująco:*

$$0: \; START(x); \; GO\,TO\,1$$

$$1: \; c := c + 1; \; GO\,TO\,2$$

$$2: \; IF\,r(x)\,THEN\,5\,ELSE\,3$$

$$3: \; c := c + 1; \; GO\,TO\,4$$

$$4: \; x := f(x); \; GO\,TO\,2$$

$$5: \; c := c - 1; \; GO\,TO\,6$$

$$6: \; IF\,c = 0\,THEN\,8\,ELSE\,7$$

$$7: \; x := f(x); \; GO\,TO\,2$$

$$8: \; STOP(x).$$

Istnieje program prosty $fd \in FD$ obliczający tą samą funkcję co fdc w każdym modelu $\mathcal{A}$.

DOWÓD. Program fdc działa następująco: zatrzymuje się dla wejścia a z wartością $f^m(a)$ w momencie, w którym liczba elementów w ciągu a, $f(a), \ldots, f^m(a)$ spełniających relację r jest większa niż liczba elementów niespełniających jej. Wartość m jest najmniejsza z możliwych spełniających warunek zatrzymania. Zauważmy, że wartość końcowa licznika c jest nie większa niż $m + 1$.

Licznik w programie prostym fd będziemy symulować przez wartości ciągu a, $f(a)$, $f^2(a), \ldots$ na dodatkowej zmiennej y. Test na 0 to sprawdzenie, czy y zawiera a. Zwiększenie licznika realizujemy przez wykonanie instrukcji

$$y := f(y),$$

natomiast opuszczenie robimy tak: najpierw sprawdzamy, czy $y \neq a$ i wtedy obliczamy $f^{-1}(y)$ w ten sposób, że na dodatkowych zmiennych liczymy kolejne pary $f^i(a)$ i $f^{i+1}(a)$ do momentu, kiedy $y = f^{i+1}(a)$. Wtedy $f^i(a)$ przepisujemy na zmienną y. Mogą wystąpić dwie sytuacje. Pierwsza, to kiedy wszystkie elementy ciągu a, $f(a)$, $f^2(a), \ldots$ są różne, druga, kiedy w ciągu pojawia się pętla, tzn. istnieją k, l takie, że

$$f^{k+l}(a) = f^k(a).$$

Jednak w przypadku, w którym zmienna symulująca licznik y osiąga taką powtarzającą się wartość, pokażemy, że symulowany program fdc musi sie pętlić. Przyjmijmy, że k i l są najmniejszymi możliwymi liczbami określonymi jak wyżej, tzn. ciąg

a, $f(a), \ldots, f^k(a), \ldots, f^{k+l-1}(a)$ jest maksymalnym ciągiem różnych elementów i przyjrzyjmy się momentowi, w którym licznik y po raz pierwszy osiąga wartość $f^{k+l}(a)$. Do tego momentu symulacja licznika była poprawna. Załóżmy, że w tym momencie $x = f^m(a)$. Wiemy, że $k + l \leq m + 1$. To znaczy, że kolejna wartość $x = f^{m+1}(a)$ już raz wystąpiła w obliczeniu, przy czym wartość licznika przy poprzednim wystąpieniu była mniejsza. Stąd dalej program będzie działał cyklicznie na pętli $f^{m+1-l}(a), \ldots, f^m(a)$ stopniowo zwiększając wartość licznika.

Wystarczy więc rozpoznać moment, kiedy zmienna symulująca licznik pierwszy raz wchodzi w pętlę. W tym celu trzymamy dotychczasową maksymalną wartość licznika $f^{max}(a)$ na dodatkowej zmiennej i za każdym razem, kiedy ją przekraczamy, sprawdzamy, czy nowa wartość licznika $f^{max+1}(a)$ nie wystąpiła już wcześniej w ciągu a, $f(a), \cdots, f^{max}(a)$. Jeśli wystąpiła, to symulujemy zapętlenie fd. □

Korzystając z lematu 4.2 udowodnimy twierdzenie Plaisteda.

TWIERDZENIE 4.3. *Dla każdego programu z co najwyżej jednym licznikiem dwukierunkowym $fdc \in FDC_1^{II}$ istnieje program prosty $fd \in FD$ obliczający tą samą funkcję w każdym modelu $\mathcal{A}$.*

DOWÓD. Idea dowodu jest następująca. Zdefiniujemy strukturę $\mathcal{A}_{fdc}$ stanów programu fdc oraz pewne operacje w niej, które łatwo symulować przy pomocy elementów programu prostego, a następnie zbudujemy program prosty działający na strukturze $\mathcal{A}_{fdc}$, który symuluje działanie programu fdc.

Możemy założyć, że program fdc nigdy nie opuszcza licznika o wartości 0. Wystarczy wstawić test na 0 przed każdym opuszczeniem licznika.

Strukturę $\mathcal{A}_{fdc} = \langle A_{fdc}, f, g, r \rangle$ określamy następująco.

Dziedzina A_{fdc} zawiera wszystkie pary (i, v), gdzie i — numer instrukcji licznikowej $c := c + 1$ lub $c := c - 1$, a v — wartościowanie zmiennych w programie fdc.

Operację f definiujemy tak: $f(i, v) = (j, w)$, jeśli j jest następną instrukcją opuszczenia lub podniesienia licznika, do której program fdc dochodzi po uruchomieniu od instrukcji i ze stanem zmiennych v, przy założeniu, że wartość licznika jest różna od 0. Wartościowanie w to stan zmiennych programu w momencie osiągnięcia instrukcji j. W przypadku, gdy tak uruchomiony program nie osiąga żadnej instrukcji licznikowej, wartość funkcji jest nieokreślona.

Operację g definiujemy tak samo jak f z tą różnicą, że teraz zakładamy, że wartość licznika jest równa 0.

Test r ma wartość *true* na (i, v), jeśli instrukcja i jest opuszczeniem licznika, zaś $false$, jeśli i jest podniesieniem licznika.

Niech $dec \in FD$ — program prosty symulujący program z licznikiem jak w lemacie 4.2 z operacjami f i r jak wyżej. Jeśli uruchomimy go na dowolnym stanie obliczeń $(i_0, v_0) \in A_{fdc}$ programu fdc, to przyjmując, że uruchamiamy go w momencie, w którym stan licznika c ma wartość 1, symuluje on przechodzenie fdc do kolejnych instrukcji licznikowych

$$(i_0, v_0),\ (i_1, v_1), \ldots ,\ (i_n, v_n)$$

do momentu, w którym licznik c zostanie więcej razy opuszczony niz podniesiony, a więc, kiedy pierwszy raz osiagnie wartość 0. Jeśli w trakcie działania zostanie wywołane obliczenie nieokreślonej wartości f, to znaczy, że program fdc kończy działanie lub pętli się nigdy nie osiągając następnej instrukcji licznikowej.

Zatem dec symuluje fazę obliczeń programu fdc od wartości licznika 1 do wartości 0. Teraz możemy złożyć to w jeden program $fd_{fdc} \in FD$ symulujacy działanie fdc:

$$0 :\ START(x);\ GO\,TO\ 1$$

$$1 :\ dec;\ GO\,TO\ 2$$

$$2 :\ x := g(x);\ GO\,TO\ 1.$$

Wartością początkową jest stan obliczeń programu fdc z momentu osiagniecia pierwszej instrukcji podniesienia licznika. Dalej w pętli wykonuje na przemian fazę przejścia od wartości licznika 1 do wartości 0 i odwrotnie. Istotne jest tu założenie, że program fdc nie opuszcza zerowego licznika. Dzięki temu mamy pewność, że operacja $x := g(x)$ symuluje zawsze przejście do instrukcji podniesienia, gdyż symuluje zawsze fazę od wartości licznika równej 0. Wywołanie obliczenia niezdefiniowanej wartości funkcji f w podprogramie dec lub niezdefiniowanej wartości g wskazuje na to, że w danej fazie program fdc kończy działanie lub pętli się nie osiagając nigdy następnej instrukcji licznikowej.

Mamy więc program prosty fd_{fdc} nad strukturą $\mathcal{A}_{fdc}$ stanów obliczeń fdc symulujący jego działanie. Wystarczy teraz zamienić instrukcję startową oraz instrukcje zawierające wołanie operacji z tej struktury na odpowiednie fragmenty programu fdc i otrzymamy program prosty obliczający taką sama funkcję jak fdc. Stany obliczeń $(i, v) \in A_{fdc}$ pamiętamy nastepująco: numery instrukcji licznikowych pamiętamy w strukturze programu fd — oddzielne fragmenty dla każdej instrukcji, natomiast wartościowanie v będziemy trzymali na zmiennych.

Instrukcję startową zastępujemy fragmentem fdc od instrukcji startowej do instrukcji podniesień licznika usuwając wszystkie opuszczenia oraz testy na 0, które zastępujemy przez połączenia pozytywne.

Funkcję f zastepujemy dla każdej instrukcji licznikowej przez fragment fdc prowadzący od tej instrukcji do innych instrukcji licznikowych. Testy na 0 zastępujemy przez połączenia negatywne.

Funkcję g tak samo, tylko testy na 0 zastępujemy przez połączenia pozytywne.

Test r zastępujemy przez połączenie negatywne lub pozytywne zależnie od tego, czy odnosimy sie do instrukcji podniesienia czy opuszczenia licznika. To możemy określić w momencie definiowania programu, gdyż ta informacja zawarta jest w strukturze fd.

W ten sposób otrzymaliśmy program prosty fd, który oblicza tą samą funkcję co program z jednym licznikiem fdc w każdym modelu $\mathcal{A}$. □

WNIOSEK 4.4. *Klasy programów prostych FD i programów z jednym licznikiem dwukierunkowym FDC_1^{II} są równoważne*

$$FD \equiv FDC_1^{II}.$$

Udowodniliśmy, że jeden licznik dodany do klasy programów prostych nie zwiększa jej mocy obliczeniowej. Dopełnieniem tego faktu jest twierdzenie Marvina Minsky'ego [**11**], które mówi, że dwa liczniki już wystarczą, by zastąpić ich dowolną ilość.

TWIERDZENIE 4.5. *Klasa programów z dwoma licznikami dwukierunkowymi FDC_2^{II} jest równoważna klasie programów z dowolna ilością liczników dwukierunkowych FDC^{II}*

$$FDC_2^{II} \equiv FDC^{II}.$$

DOWÓD. Pomysł dowodu jest identyczny jak w dowodzie twierdzenia 3.2, w którym maszyną dwulicznikową symulowaliśmy maszynę trójlicznikową.

Niech $fdc \in FDC^{II}$ będzie programem z n licznikami. Bedziemy go symulować za pomocą programu $fdc_2 \in FDC_2^{II}$ z dwoma licznikami d_1 i d_2, w ten sposób, że na liczniku d_1 będziemy przechowywać wartości wszystkich liczników $c_1, c_2, \ldots, c_n$ programu fdc:

$$d_1 = 2^{c_1}3^{c_2}\cdots p_n^{c_n}.$$

Tak, jak w dowodzie twierdzenia 3.2, test na 0 licznika c_i robimy w cyklu p_i instrukcji sprawdzając, czy przepisywanie zakończyło się na ostatniej. Podniesienie c_i to pomnożenie d_1 przez p_i, a opuszczenie licznika c_i to zrobienie testu na 0, i jesli negatywny, to dzielimy d_1 przez p_i. We wszystkich tych operacjach używamy d_2 jako licznika pomocniczego.

Do kompletności symulacji potrzeba jeszcze podnieść raz d_1 na początku programu fdc_2, aby nadać mu początkową wartość 1. □

Wskazaliśmy zależności programów z licznikami dwukierunkowymi FDC^{II} w stosunku do słabszej klasy programów prostych FD. Teraz w oparciu o prace Friedmana [**3**] i Kfoury'ego [**7**] pokażemy, jaka zależność zachodzi pomiędzy klasą programów FDC^{II} a klasą efektywnych definicji ED.

TWIERDZENIE 4.6. *Klasa definicji efektywnych ED jest istotnie silniejsza od klasy programów z licznikami dwukierunkowymi FDC^{II}*

$$FDC^{II} \prec ED.$$

DOWÓD. W pierwszej części pokażemy, że dla każdego programu z licznikami $fdc \in FDC^{II}$ istnieje efektywna defnicja $ed \in ED$ definiująca tą samą funkcję.

Budujemy ją w następujący sposób. Najpierw rozwijamy graf programu fdc w nieskończony program drzewiasty fdc' zaczynając od instrukcji startowej w wierzchołku drzewa. Każdej instrukcji w drzewie odpowiada jedna instrukcja w grafie programu i ma ona zero, jeden lub dwa następniki zależnie od tego, czy jest to instrukcja zakończenia, przypisania, czy wyboru. Do każdej instrukcji na poziomie i tworzymy nowe instrukcje na poziomie $i+1$ o identycznej treści jak następniki odpowiadające jej w grafie programu fdc, tylko z nowymi identyfikatorami. Tak zbudowane drzewo działa identycznie jak odpowiadający mu program.

Z drzewa fdc' możemy teraz usunąć wszystkie instrukcje licznikowe w następujący sposób. Wiedząc, że wartości liczników na początku są równe 0, możemy w każdej instrukcji drzewa określić dokładnie ich wartość, zatem znamy wyniki wszystkich testów na 0. Odcinamy gałęzie, które nie zostaną wybrane i pozostawiamy resztę. W ten sposób działanie już nie zależy od liczników. Usuwamy instrukcje licznikowe otrzymując nowe drzewo fdc''.

Przyjmijmy, że zmiennymi wejściowymi programu były zmienne $x_1, x_2, \ldots, x_n$. Załóżmy przez chwilę, że program został uruchomiony w strukturze termów $\mathcal{T}^{Var}$ nad zbiorem zmiennych wejściowych $x_1, x_2, \ldots, x_n$ z tymi zmiennymi na wejściu. Idąc w dół struktury drzewa możemy dla każdej instrukcji określić, jakie termy są aktulanie przypisane na zmienne. W ten sposób w każdej instrukcji wyboru możemy zależność od bieżących wartości zmiennych zastąpić zależnością od termów i w ten sposób uzyskamy warunki w formie formuł otwartych pierwszego rzędu $\mathcal{F}^{Var}$ nad zbiorem zmiennych $x_1, x_2, \ldots, x_n$. Podobnie w instrukcjach zakończenia zmienną wyjściową możemy zastąpić termem będącym jej bieżącą wartością.

Teraz możemy już zdefiniować definicję efektywną. Będzie to zbiór par $\langle \phi, t \rangle$, przy czym każdej skończonej ścieżce drzewa odpowiada dokładnie jedna para $\langle \phi, t \rangle$ w ten sposób, że formuła ϕ jest koniunkcją wszystkich warunków testowych występujących na ścieżce, włączonych bez lub z negacją zależnie od tego, czy ścieżka wychodzi przez wynik pozytywny czy przez negatywny z instrukcji testowej. Term t jest wartością zmiennej wyjściowej z instrukcji $STOP$ na końcu ścieżki.

Pozostało jeszcze do wyjaśnienia, w jaki sposób efektywnie generujemy pary $\langle \phi, t \rangle$. Załóżmy, że na taśmie mamy kod programu fdc w postaci ciągu jego instrukcji. Możemy generować kolejne stany obliczeń programu przeszukując drzewo wszerz. W każdym stanie pamiętamy stan obliczeń $\langle i, v, \phi \rangle$, w którym i to numer aktualnej instrukcji, v to termy będące aktualnymi wartościami zmiennych, a ϕ to formuła opisującą koniunkcję warunków testowych, które obliczenie musiało spełnić, aby dojść do aktualnego stanu. W każdym kroku bierzemy kolejny stan obliczeń, generujemy nowe stany zgodnie ze strukturą programu i wstawiamy je na koniec kolejki. Jesli i jest instrukcją wyboru, to powstają dwa nowe stany obliczeń, w pierwszym dodajemy warunek testowy do formuły ϕ, a w drugim jego negację. Jeśli i jest przypisaniem, to zmienia się wartość jednej ze zmiennych w wartościowaniu v. Jeśli zaś natrafiamy na instrukcję końcową, to wypisujemy parę $\langle \phi, t \rangle$, gdzie ϕ to formuła z aktualnego stanu, a t to term będący bieżącą wartością zmiennej wyjściowej. W tym przypadku nie generujemy nowego stanu obliczeń.

Udowodniliśmy, że

$$FDC^{II} \preceq ED.$$

Pokażemy teraz, że istnieje definicje efektywna $ed \in ED$ nierównoważna żadnemu programowi z licznikami dwukierunkowymi .

Niech sygnatura $\Sigma = \langle c, l, r, g, p \rangle$ zawiera jedną stałą c, dwie funkcje jednoargumentowe l i r, jedną funkcję dwuargumentową g oraz relację jednoargumentową p, $\mathcal{A}_n$ — ciąg algebr termów nad sygnaturą Σ, w których dla ustalonego n relacja p jest spełniona dla wszystkich termów algebry $\mathcal{A}_n$ z wyjątkiem c, $l(c)$, $l^2(c), \ldots, l^{n-1}(c)$. Definicję efektywną ed definiujemy w ten sposób, że w algebrze $\mathcal{A}_n$ pierwszą spełnialną formułą będzie n-ta formuła definicji ed, natomiast term t_n będzie drzewem binarnym wysokości n zbudowanym z symbolu funkcyjnego g, w każdym liściu zawierającym unikalny podterm o długości n zbudowany z unarnych symboli l i r tak, że wyznacza on ścieżkę z korzenia do danego liścia, przy interpetacji, że l to skręt w lewo, a r — w prawo. Pierwsze trzy formuły definicji efektywnej wyglądają

następująco:

$$\begin{aligned}
\langle \phi_0, t_0 \rangle &= \langle p(c), c \rangle \\
\langle \phi_1, t_1 \rangle &= \langle p\left(l(c)\right), g\left(l(c), r(c)\right) \rangle \\
\langle \phi_2, t_2 \rangle &= \langle p\left(l\left(l(c)\right)\right), g\left(g\left(l\left(l(c)\right), l\left(r(c)\right)\right), g\left(r\left(l(c)\right), r\left(r(c)\right)\right)\right) \rangle
\end{aligned}$$

i dalej budujemy je analogicznie. Tak zdefiniowane formuły są efektywnie generowalne. Pokażemy, że nie istnieje program z licznikami, który w każdej algebrze $\mathcal{A}_n$ dawałby w wyniku term t_n.

Zauważmy, że każdy podterm termu t_n zawierający przynajmniej jedną pełną ścieżkę unarną złożoną z symboli l i r jest unikalny wśród wszystkich podtermów t_n. Przez indukcję pokażemy, że obliczenie każdego termu t_n wymaga $n+1$ zmiennych. Dokładniej, że w obliczeniu istnieje moment, w którym program musi na swoich zmiennych przechowywać $n+1$ różnych podtermów t_n. Przyjmijmy dodatkowo, że żadne obliczenie nie wykonuje zbędnych kroków.

Dla $n = 0$ teza zachodzi.

Załóżmy, że obliczenie t_n wymaga $n+1$ zmiennych. Rozważmy dowolne obliczenie t_{n+1}. Niech t_n^l, t_n^r oznaczają odpowiednio lewe i prawe poddrzewo t_{n+1}:

$$t_{n+1} = g(t_n^l, t_n^r).$$

Aby policzyć t_{n+1}, musimy policzyć oba poddrzewa t_n^l, t_n^r. Zauważmy, że od termu t_n różnią sie tylko tym, że t_n^l ma na początku każdego liścia jeden dodatkowy symbol l, a t_n^r ma r, więc ich obliczenie również wymaga $n+1$ zmiennych. Niech t_x — pierwszy unikalny term obliczony w procesie liczenia t_{n+1}. Załóżmy, że to jest podterm poddrzewa t_n^l. Zauważmy, że już do końca obliczania t_n program musi przechowywać jeden z termów drzewa t_n^l znajdujących się na ścieżce pomiędzy t_x i korzeniem poddrzewa t_n^l. Ponadto każdy z tych termów jest unikalny, więc nie może być wykorzystany w obliczeniu poddrzewa t_n^r. Z drugiej strony, w wyliczaniu termu t_n^r musi istnieć moment, w którym program przechowuje $n+1$ podtermów t_n^r. To oznacza, że w tym momencie program musi przechowywać $n+2$ termy. To samo dowodzimy symetrycznie w przypadku, gdy pierwszym unikalnym podtermem, który jest wyliczany, jest podterm t_n^r. W ten sposób pokazaliśmy krok indukcyjny.

Udowodniliśmy, że program, który wylicza term t_n w algebrze termów, musi mieć $n+1$ zmiennych.

Załóżmy, że istnieje program z licznikami dwukierunkowymi $fdc \in FDC^{II}$, który definiuje identyczną funkcję. Niech program fdc ma n zmiennych. Wynikiem ed w algebrze $\mathcal{A}_n$ jest term t_n. Aby go policzyć, fdc musiałby mieć $n+1$ zmiennych. Sprzeczność. □

Możemy teraz zebrać wszystkie twierdzenia z tego rozdziału w jeden ogólny wniosek.

WNIOSEK 4.7. *Istnieje następujący łańcuch zależności pomiędzy klasami programów:*

$$FD \equiv FDC_1^{II} \prec FDC_2^{II} \equiv FDC^{II} \prec ED.$$

Wiemy, że klasa definicji efektywnych ED, która jest uważana za klasę definiującą wszystkie obliczalne funkcje w dowolnych strukturach, jest istotnie silniejsza od klasy programów z licznikami dwukierunkowymi FDC^{II}. Pokażemy teraz jeszcze jedną ważną własność klasy FDC^{II} sformułowaną przez Kfoury'ego [**7**], a udowodnioną przez Friedmana [**3**], która opisuje te struktury, na których programy z licznikami dwukierunkowymi są uniwersalne.

TWIERDZENIE 4.8. *Dla każdej definicji efektywnej $ed \in ED$ istnieje program z licznikami dwukierunkowymi $fdc \in FDC^{II}$, który oblicza identyczną funkcję jak ed we wszystkich strukturach algebraicznych $\mathcal{A}$ spełniających następujący warunek: dla każdego całkowitego $k \geq 1$ istnieje całkowite $n \geq k$ takie, że dla każdych $a_1, \ldots, a_k \in A$ i dla każdego termu b nad zbiorem zmiennch $x_1, \ldots, x_k$ istnieje program prosty $fd \in FD$ k-argumentowy z n zmiennymi, który oblicza wartość termu $v(b)$ dla wartości wejściowych $a_1, \ldots, a_k$ i wartościowania v takiego, że $v(x_i) = a_i$.*

DOWÓD. Dowód tego twierdzenia jest długi i przedstawimy tylko jego szkic.

Chcemy symulować definicję efektywną ed przy pomocy programu z licznikami dwukierunkowymi. Skonstruujemy program $fdc \in FDC^{II}$, który będzie miał tą własność, że będzie obliczał identyczną funkcję jak ed na wszystkich strukturach spełniających warunek twierdzenia, natomiast na pozostałych może się zapętlić.

Pierwszą rzecz, którą zrobimy, to przyjmiemy, że zamiast liczników będziemy posługiwać się taśmą maszyny Turinga. Możemy tak założyć, bo z lematu 3.1 z trzeciego rozdziału wiemy, że liczniki potrafią symulować maszynę Turinga.

Dowolny term t możemy bezpośrednio zapisać na taśmie. Załóżmy przez moment, że istnieje program z licznikami fdc', który mając zakodowany term t nad zbiorem zmiennych $x_1, \ldots, x_k$, potrafi obliczyć jego wartość dla argumentów $a_1, \ldots, a_k$. Wtedy definicję efektywną symulujemy tak. Generujemy kolejno pary $\langle \phi_i, t_i \rangle$ i w każdym kroku liczymy najpierw wartość ϕ_i w ten sposób, że wyliczamy indukcyjnie wartości kolejnych podformuł. Jeśli musimy policzyć wartość relacji $r(t_1, \ldots, t_m)$, to obliczamy kolejno wartości termów $t_1, \ldots, t_m$ zapamiętując wyniki na kolejnych m zmiennych i na końcu liczymy wartość relacji r zapamiętując wynik. Jeśli otrzymamy pozytywną wartość formuły ϕ_i, to wyliczamy wartość termu t_i i zwracamy wynik, w przeciwnym przypadku przechodzimy do następnej pary $\langle \phi_{i+1}, t_{i+1} \rangle$. Wartości termów liczymy przy pomocy podprogramu fdc'. Zauważmy, że potrzebujemy tylko

tyle dodatkowych zmiennych, ile wynosi maksymalna arność relacji struktury, a więc ustaloną liczbę.

To, co musimy pokazać, to jak liczyć wartość zakodowanego termu t.

Najpierw powiemy, jak policzyć wartość termu $\overline{t}$ o złożoności rejestrowej n, tzn. takiego, którego wartość można policzyć programem prostym o n zmiennych w algebrze termów. Będziemy wyliczać term $\overline{t}$ indukcyjnie z wartości kolejnych podtermów. Najpierw indukcyjnie w głąb drzewa $\overline{t}$ wyliczamy, ile zmiennych zajmuje wyliczenie poszczególnych podtermów, a następnie wyliczamy je w kolejności od tych, które wymagają najwięcej zmiennych do tych, które wymagają najmniej. O kolejności wyliczania podtermów decydujemy tak w każdym poddrzewie $\overline{t}$. Taki sposób wyliczania termu zajmuje minimalną liczbę zmiennych, więc w szczególności nie więcej niż n.

Teraz pokażemy, jak obliczyć wartość dowolnego termu t. Niech n jak w treści twierdzenia będzie taką liczbą, że dla każdej wartości $b \in A$ obliczalnej z argumentów $a_1, \ldots, a_k \in A$ w strukturze $\mathcal{A}$ istnieje program prosty fd o n zmiennych obliczający wartość b. To znaczy, że istnieje term $\overline{t}$ o złożoności rejestrowej n, którego wartość dla tych argumentów w strukturze $\mathcal{A}$ jest równa b. Szukamy go następująco. Najpierw ze struktury t generujemy ciąg wszystkich jego podtermów $t_1, \ldots, t_u$ w ten sposób, że dla $t_i = f(t_{i_1}, \ldots, t_{i_m})$ termy $t_{i_1}, \ldots, t_{i_m}$ występują już wcześniej w tym ciągu. Ostatni element t_u jest równy t. Teraz szukamy takiego ciągu termów $\overline{t_1}, \ldots, \overline{t_u}$ o złożoności rejestrowej n, że wartość każdego $\overline{t_i}^{\mathcal{A}}$ jest równa wartości $t_i^{\mathcal{A}}$. W ten sposób otrzymamy term $\overline{t_u}$ o złożoności rejestrowej n, którego wartość w strukturze $\mathcal{A}$ będzie równa wartości t. Ponieważ każda wartość $t_i^{\mathcal{A}}$ jest obliczalna przez program prosty o n zmiennych, więc taki ciąg istnieje. Szukamy go w następujący sposób. Generujemy po kolei wszystkie możliwe ciągi termów $\overline{t_1}, \ldots, \overline{t_u}$ o złożoności rejestrowej n — to jest zbiór rekurencyjny, więc możemy tak zrobić — i sprawdzamy po kolei, czy każda wartość $\overline{t_i}^{\mathcal{A}}$ jest równa $t_i^{\mathcal{A}}$. Jeśli chcemy sprawdzić term $t_i = f(t_{i_1}, \ldots, t_{i_m})$, to wiemy już, że wartości podtermów $t_{i_1}^{\mathcal{A}}, \ldots, t_{i_m}^{\mathcal{A}}$ są równe odpowiednio $\overline{t_{i_1}}^{\mathcal{A}}, \ldots, \overline{t_{i_m}}^{\mathcal{A}}$, więc wyliczamy te wartości, a także wartość $\overline{t_i}^{\mathcal{A}}$ i sprawdzamy, czy

$$f^{\mathcal{A}}(\overline{t_{i_1}}^{\mathcal{A}}, \ldots, \overline{t_{i_m}}^{\mathcal{A}}) \;=\; \overline{t_i}^{\mathcal{A}}.$$

Jeśli tak, to sprawdzamy zgodność następnej pary termów t_{i+1}, $\overline{t_{i+1}}$, jeśli nie, to generujemy następny ciąg $\overline{t_1}, \ldots, \overline{t_u}$. □

Ponieważ klasa programów z dwoma licznikami dwukierunkowymi FDC_2^{II} potrafi symulować klasę programów z dowolną ilością liczników dwukierunkowych FDC^{II}, więc identyczny wniosek możemy sformułować dla programów z dwoma licznikami.

WNIOSEK 4.9. *Dla każdej definicji efektywnej* $ed \in ED$ *istnieje program z dwoma licznikami dwukierunkowymi* $fdc \in FDC_2^{II}$*, który oblicza identyczną funkcję jak* ed *we wszystkich strukturach algebraicznych* $\mathcal{A}$ *spełniających następujący warunek:*
dla każdego całkowitego $k \geq 1$ *istnieje całkowite* $n \geq k$ *takie, że dla każdych* $a_1, \ldots, a_k \in A$ *i dla każdego termu* b *nad zbiorem zmiennch* $x_1, \ldots, x_k$ *istnieje program prosty* $fd \in FD$ k*-argumentowy z* n *zmiennymi, który oblicza wartość termu* $v(b)$ *dla wartości wejściowych* $a_1, \ldots, a_k$ *i wartościowania* v *takiego, że* $v(x_i) = a_i$*.*

W ten sposób zebraliśmy najważniejsze znane fakty o licznikach dwukierunkowych. Zauważmy, że dziedziny maszyn Turinga możemy traktować jak szczególne struktury oparte na zbiorach słów nad skończonym alfabetem z prostymi operacjami dodania, usunięcia i rozpoznania symbolu na początku słowa. Podobnie dziedziny maszyn licznikowych dwukierunkowych to liczby naturalne z działaniami następnika, poprzednika i testu na 0. W tym kontekście definicje efektywne możemy traktować jako uogólnienie maszyn Turinga na wszystkie struktury i analogicznie programy z licznikami jako uogólnienie maszyn licznikowych. I o ile w tych szczególnych przypadkach maszyny Turinga i maszyny licznikowe mają takie same możliwości, to w ogólnym przypadku mechanizm definicji efektywnych jest silniejszy od mechanizmu liczników. Uogólnieniem tego faktu jest ostatnie twierdzenie 4.8 opisujące struktury, na których możliwości obu modeli obliczeń pozostają takie same.

ROZDZIAŁ 5

TWIERDZENIE O LICZNIKACH JEDNOKIERUNKOWYCH

W tym rozdziale zajmiemy się innym modelem liczników — licznikami jednokierunkowymi. Podstawowymi operacjami liczników dwukierunkowych były podniesienie, opuszczenie i test na 0. W licznikach jednokierunkowych mamy do dyspozycji wyzerowanie licznika, podniesienie i porównanie wartości dwóch liczników. Zauważmy, że są to operacje z prostej struktury liczb naturalnych ze stałą 0, operacją następnika i równością.

Pokażemy nowe twierdzenie, które mówi, że dwa liczniki jednokierunkowe potrafią symulować dowolną liczbę liczników dwukierunkowych. Następny rozdział przedstawi, w jaki sposób na bazie tego twierdzenia wszystkie znane fakty o licznikach dwukierunkowych można odnieść do liczników jednokierunkowych.

TWIERDZENIE 5.1. *Dwa liczniki jednokierunkowe potrafią symulować dowolny program pc zbudowany z operacji na dowolnej liczbie liczników dwukierunkowych w ten sposób, że jeśli pc z n licznikami dwukierunkowymi startuje z wartością początkową k na pierwszym liczniku i daje w wyniku wartość l na pierwszym liczniku, to program z dwoma licznikami jednokierunkowymi mając na pierwszym liczniku wartość 2^k kończy działanie z wynikiem $2^l p_{n+1}^{d_1} p_{n+2}^{d_2} \cdots p_{2n}^{d_n}$ na pierwszym liczniku, gdzie $p_1, \ldots, p_n, p_{n+1}, \ldots, p_{2n}$ to kolejne liczby pierwsze, a $d_1, d_2, \ldots, d_n$ są pewnymi liczbami naturalnymi.*

DOWÓD. Przyjmijmy, że program pc operuje na n licznikach dwukierunkowych $c_1, c_2, \ldots, c_n$. Będziemy go symulowć przy pomocy dwóch liczników jednokierunkowych $c1$, $c2$ w ten sposób, że licznik $c1$ będzie przechowywać wartości $c_1, c_2, \ldots, c_n$ w wykładnikach kolejnych liczb pierwszych $2, 3, \ldots, p_n$ w nastepujący sposób

$$c1 = 2^{c_1} 3^{c_2} \cdots p_n^{c_n} p_{n+1}^{d_1} p_{n+2}^{d_2} \cdots p_{2n}^{d_n}.$$

W powyższym wzorze pojawia się n dodatkowych czynników, które będziemy wykorzystywać przy symulacji opuszczania liczników c_i. Ponieważ nie mamy operacji

opuszczenia na liczniku jednokierunkowym, więc będziemy ją symulować przez przeniesienie jedynki z wykładnika c_i do wykładnika d_i. Zauważmy, że dzięki temu operację opuszczenia będziemy mogli symulować poprzez zwiększanie $c1$. Licznik $c2$ będziemy wykorzystywać jako pomocniczy do obliczeń na $c1$.

Operację testu na 0

$$id: IF\, c_i = 0\, THEN\, id'\, ELSE\, id''$$

symulujemy przez sprawdzenie, czy $c1$ jest podzielne przez p_i. Robimy to w ten sposób, że w pętli p_i instrukcji podnosimy $c2$ aż do momentu, kiedy osiagnie wartość $c1$. Jeśli ją osiągnie na ostatniej instrukcji pętli, to c_i jest różne od 0, jeśli w środku, to jest równe 0. Cały podprogram $ZEROTEST(c_i, id, id', id'')$ symulujący test na 0 wygląda zatem tak:

$$\begin{array}{rl} id: & c2 := 0;\, GO\, TO\, id_1 \\ \left.\begin{array}{r} id_1: \\ id_2: \\ \vdots \\ id_{2p_i-3}: \\ id_{2p_i-2}: \\ id_{2p_i-1}: \end{array}\right\} 2p_i - 2 & \begin{array}{l} c2 := c2 + 1;\, GO\, TO\, id_2 \\ IF\, c2 = c1\, THEN\, id'\, ELSE\, id_3 \\ \vdots \\ c2 := c2 + 1;\, GO\, TO\, id_{2p_i-2} \\ IF\, c2 = c1\, THEN\, id'\, ELSE\, id_{2p_i-1} \\ c2 := c2 + 1;\, GO\, TO\, id_{2p_i} \end{array} \\ id_{2p_i}: & IF\, c2 = c1\, THEN\, id''\, ELSE\, id_1. \end{array}$$

Podniesienie licznika

$$id: c_i := c_i + 1;\, GO\, TO\, id'$$

symulujemy przez pomnożenie $c1$ przez p_i. Robimy to tak, że w kolejnych krokach zwiększamy $c1$ o $p_i - 1$, a $c2$ o p_i i kończymy w momencie, gdy wartości $c1$ i $c2$ się wyrównają. Cały podprogram $INCREMENT(c_i, id, id')$ symulujący podniesienie licznika wygląda tak:

$$\begin{array}{rl} id: & c2 := 0;\, GO\, TO\, id_1 \\ id_1: & IF\, c2 = c1\, THEN\, id'\, ELSE\, id_2 \\ \left.\begin{array}{r} id_2: \\ \vdots \\ id_{p_i}: \end{array}\right\} p_i - 1 & \begin{array}{l} c1 := c1 + 1;\, GO\, TO\, id_3 \\ \vdots \\ c1 := c1 + 1;\, GO\, TO\, id_{p_i+1} \end{array} \\ \left.\begin{array}{r} id_{p_i+1}: \\ \vdots \\ id_{2p_i}: \end{array}\right\} p_i & \begin{array}{l} c2 := c2 + 1;\, GO\, TO\, id_{p_i+2} \\ \vdots \\ c2 := c2 + 1;\, GO\, TO\, id_1. \end{array} \end{array}$$

Niech r oznacza ilość wykonanych kroków. Z warunku zakończenia

$$c1 + (p_i - 1) \cdot r = p_i \cdot r$$

dostajemy, że podprogram wykonuje liczbę kroków r równą wartości początkowej $c1$. Ponieważ w każdym kroku $c1$ zwiększa swoją wartość o $p_i - 1$, zatem jego nowa wartość jest p_i razy większa:

$$c1 := c1 + (p_i - 1) \cdot c1 = p_i \cdot c1.$$

Jeśli wartością początkową $c1$ było

$$c1 = 2^{c_1} \cdots p_i^{c_i} \cdots p_n^{c_n} p_{n+1}^{d_1} \cdots p_{2n}^{d_n},$$

to po wykonaniu symulacji otrzymamy

$$c1 = 2^{c_1} \cdots p_i^{c_i+1} \cdots p_n^{c_n} p_{n+1}^{d_1} \cdots p_{2n}^{d_n}.$$

Opuszczenie licznika

$$id : c_i := c_i - 1; GO\,TO\,id'$$

symulujemy przez jednoczesne podzielenie $c1$ przez p_i i pomnożenie przez p_{n+i}. Najpierw jednak sprawdzamy, czy wartośc c_i nie jest przypadkiem równa 0, wykorzystując podprogram $ZEROTEST$. Jeśli tak, to nic już nie robimy, jeśli nie, to postępujemy następująco. W kolejnych krokach zwiększamy $c1$ o $p_{n+i} - p_i$, a $c2$ o p_{n+i} i kończymy w momencie, gdy wartości $c1$ i $c2$ się wyrównają. Cały podprogram $DECREMENT(c_i, id, id')$ symulujący opuszczenie licznika wygląda tak:

$$\begin{array}{rrl}
 & id: & ZEROTEST(c_i, id, id', id_1) \\
 & id_1: & c2 := 0; GO\,TO\,id_2 \\
 & id_2: & IF\,c2 = c1\,THEN\,id'\,ELSE\,id_3 \\
p_{n+i} - p_i & id_3: & c1 := c1 + 1; GO\,TO\,id_4 \\
 & \vdots & \vdots \\
 & id_{p_{n+i}-p_i+2}: & c1 := c1 + 1; GO\,TO\,id_{p_{n+i}-p_i+3} \\
p_{n+i} & id_{p_{n+i}-p_i+3}: & c2 := c2 + 1; GO\,TO\,id_{p_{n+1}-p_i+4} \\
 & \vdots & \vdots \\
 & id_{2p_{n+i}-p_i+2}: & c2 := c2 + 1; GO\,TO\,id_1.
\end{array}$$

Niech r oznacza ilość wykonanych kroków. Z warunku zakończenia

$$c1 + (p_{n+i} - p_i) \cdot r = p_{n+i} \cdot r$$

dostajemy, że podprogram wykonuje liczbę kroków r, która jest dokładnie p_i razy mniejsza od wartości początkowej $c1$. Ponieważ w każdym kroku $c1$ zwiększa swoją

wartość o $p_{n+i} - p_i$, zatem jego nowa wartość jest $\frac{p_{n+i}}{p_i}$ razy większa:

$$c1 := c1 + (p_{n+i} - p_i) \cdot \frac{c1}{p_i} = \frac{p_{n+i}}{p_i} \cdot c1.$$

Jeśli wartością początkową $c1$ było

$$c1 = 2^{c_1} \cdots p_i^{c_i} \cdots p_n^{c_n} p_{n+1}^{d_1} \cdots p_{n+i}^{d_i} \cdots p_{2n}^{d_n},$$

to po wykonaniu symulacji otrzymamy

$$c1 = 2^{c_1} \cdots p_i^{c_i - 1} \cdots p_n^{c_n} p_{n+1}^{d_1} \cdots p_{n+i}^{d_i + 1} \cdots p_{2n}^{d_n}.$$

Symulując w wyżej opisany sposób działanie liczników dwukierunkowych z wartością początkową 2^k na pierwszym liczniku $c1$ otrzymamy w wyniku dokładnie to, co mówi twierdzenie:

$$2^l p_{n+1}^{d_1} p_{n+2}^{d_2} \cdots p_{2n}^{d_n},$$

gdzie l jest wartością wynikową działania liczników dwukierunkowych. □

ROZDZIAŁ 6

WNIOSKI DLA LICZNIKÓW JEDNOKIERUNKOWYCH

Opierając się na twierdzeniu 5.1 z poprzedniego rozdziału przedstawimy wypływające z niego wnioski dla maszyn licznikowych oraz programów z licznikami i uzupełnimy powszechnie znaną hierarchię klas programów sformułowaną w rozdziale czwartym. Wnioski te pozwalają ocenić, które elementy struktury automatów i programów z licznikami istotnie przyczyniają się do zwiększenia ich mocy obliczeniowej, a które służą jedynie do uproszczenia zapisu.

Wpierw sformułujemy wnioski dotyczące maszyn licznikowych jednokierunkowych.

WNIOSEK 6.1. *Dla każdej maszyny Turinga MT istnieje taka maszyna dwulicznikowa jednokierunkowa ML, że jeśli na wejściu dostaje wartość $2^{\sigma^*(w)}$, gdzie w — dowolne słowo wejściowe dla maszyny MT, to symuluje działanie maszyny MT i albo na wyjściu zwraca wartość $2^{\sigma^*(u)}7^{d_1}11^{d_2}13^{d_3}$, gdzie d_1, d_2, d_3 są pewnym liczbami naturalnymi, jeśli ta kończy działanie z wynikiem u, albo podobnie jak maszyna MT nie zatrzymuje się.*

DOWÓD. Wniosek ten można pokazać korzystając z twierdzenia o dwóch licznikach jednokierunkowych oraz lematu 3.1 o maszynach trójlicznikowych dwukierunkowych. Lemat ten mówi, że maszyna trójlicznikowa potrafi symulować maszynę Turinga MT w ten sposób, że na pierwszym liczniku dostaje kod słowa wejściowego $\sigma^*(w)$ maszyny MT i w wyniku zwraca kod słowa wynikowego $\sigma^*(u)$. Jeśli teraz te trzy liczniki dwukierunkowe będziemy symulować dwoma licznikami jednokierunkowymi tak jak to jest opisane w twierdzeniu 5.1, a więc na pierwszym liczniku dostaniemy wartość $2^{\sigma^*(w)}$, to w wyniku maszyna dwulicznikowa jednokierunkowa zwróci nam wartość $2^{\sigma^*(u)}7^{d_1}11^{d_2}13^{d_3}$, gdzie $d_!$, d_2, d_3 są pewnymi liczbami naturalnymi. □

W rozdziale trzecim opisaliśmy jeszcze inny sposób symulacji maszyny Turinga. Mianowicie wyposażyliśmy maszynę dwulicznikową dwukierunkową ML w jednokierunkową taśmę wejściową tylko do odczytu oraz jednokierunkową taśmę wyjściową tylko do zapisu i pokazaliśmy, że możemy symulować maszynę Turinga MT w ten sposób, że maszyna ML otrzymuje słowo wejściowe w na jej taśmie wejściowej i wynik u działnia maszyny MT wypisuje na swojej taśmie wyjściowej.

Skorzystaliśmy wtedy z tego, że maszyna trójlicznikowa dwukierunkowa ML' potrafi symulować maszynę Turinga w ten sposób. Ale jeśli tak, to symulując maszynę ML' tak jak w twierdzeniu 5.1 o dwóch licznikach jednokierunkowych możemy identyczny wniosek sformułować dla maszyny z licznikami jednokierunkowymi. Mianowicie, dla każdej maszyny Turinga ML istnieje maszyna dwulicznikowa jednokierunkowa ML z jednokierunkową taśmą wejściową tylko do odczytu oraz jednokierunkową taśmą wyjściową tylko do zapisu, która, otrzymawszy na taśmie wejściowej słowo wejściowe w dla maszyny MT zwraca na taśmie wyjściowej wynik w działania maszyny MT.

W równie prosty sposób możemy pokazać, że problem stopu dla maszyn dwulicznikowych jednokierunkowych jest nierostrzygalny.

WNIOSEK 6.2. *Nie istnieje maszyna Turinga, która na wejściu dostawszy kod maszyny dwulicznikowej jednokierunkowej ML oraz liczbę naturalną n odpowiada: tak, jeśli maszyna ML zatrzymuje się dla wejścia n, a odpowiada: nie, jeśli maszyna ML nie zatrzymuje się dla wejścia n.*

DOWÓD. Lemat 3.3 mówi nam, że nie istnieje maszyna Turinga rostrzygająca problem stopu dla maszyn trójlicznikowych dwukierunkoych. Gdyby istniała taka maszyna MT dla maszyn dwulicznikowych jednokierunkowych, to analogicznie jak w dowodzie twierdzenia 3.4 o nierostrzygalności problemu stopu dla maszyn dwulicznikowych dwukierunkowych, moglibyśmy zbudować maszynę, która najpierw kod maszyny trójlicznikowej zamieniałaby na kod maszyny z dwoma licznikami jendokierunkowymi, a potem uruchamiałaby maszynę MT i dawała w wyniku identyczną odpowiedź. W ten sposób otrzymalibyśmy maszynę Turinga rostrzygającą problem stopu dla maszyn trójlicznikowych dwukierunkowych. □

Wnioski, że dwa liczniki każdego typu wystarczą do symulacji maszyn Turinga i oba modele są nierozstrzygalne wskazują na to, że moc obliczeniowa tych dwóch mechanizmów jest bardzo podobna.

Teraz przejdziemy do drugiej, bardziej abstrakcyjnej koncepcji obliczeń — programów z licznikami. Będziemy zmierzali do uzupełnienia hierarchii sformułowanej we wniosku 4.7 z rozdziału czwartego o klasy oparte na licznikach jednokierunkowych. W tym celu przedstawimy kilka prostych faktów stanowiących dopełnienie wiedzy na temat liczników jednokierunkowych.

WNIOSEK 6.3. *Dla każdego programu z licznikami dwukierunkowymi $fdc \in FDC^{II}$ istnieje program z dwoma licznikami dwukierunkowymi $fdc_2 \in FDC_2^I$ definiujący identyczną funkcję jak fdc.*

DOWÓD. Wniosek ten wypływa bezpośrednio z twierdzenia 5.1 z poprzedniego rozdziału. Wystarczy symulować operacje n liczników dwukierunkowych tak jak to jest opisane w dowodzie tego twierdzenia. Dopowiedzenia wymaga jedynie fakt, że na początku programu podnosimy raz pierwszy licznik $c1$, aby zasymulować początkową zerową wartość liczników dwukierunkowych. □

FAKT 6.4. *Dla każdego programu z co najwyżej jednym licznikiem jednokierunkowym $fdc \in FDC_1^I$ istnieje program prosty $fd \in FD$ definiujący identyczną funkcję jak fdc.*

DOWÓD. Wystarczy zauważyć, że jedyną instrukcją licznika jednokierunkowego, która wpływa na sterowanie w programie, jest instrukcja porównania wartości liczników. W tym przypadku, ponieważ jest tylko jeden licznik, więc porównanie go z samym sobą da zawsze wynik pozytywny. Możemy więc z programu usunąć wszystkie instrukcje licznikowe wyboru, a w konsekwencji również wszystkie pozostałe instrukcje licznikowe. W ten sposób otrzymamy program prosty działający tak samo. □

FAKT 6.5. *Dla każdego programu z licznikami jednokierunkowymi $fdc \in FDC^I$ istnieje program z licznikami dwukierunkowymi $fdc' \in FDC^{II}$ definiujący identyczną funkcję jak fdc.*

DOWÓD. Wartości liczników jednokierunkowych będziemy trzymać na odpowiadających im licznikach dwukierunkowych i użyjemy dodatkowego licznika c do symulacji operacji porównania wartości. Operację podniesienia licznika wykonujemy tak samo, operację wyzerowania realizujemy przez pętlę, która zmniejsza wartość licznika aż do osiągnięcia wartości 0, natomiast porównanie dwóch liczników jednokierunkowych c_i, c_j robimy tak. W każdym kroku opuszczamy wartości odpowiadających im liczników dwukierunkowych c_i', c_j' i podnosimy wartość c aż do momentu, w którym jeden z liczników c_i', c_j' osiągnie wartość 0. Jeśli na drugim też jest 0, to zapamiętujemy, że równe, w przeciwnym przypadku pamiętamy, że nie. Na końcu musimy przywrócić wartość c_i', c_j' używając c w ten sposób, że przy każdym opuszczeniu c podnosimy c_i', c_j' aż do momentu, w którym c osiągnie 0. □

Powyższe wnioski pozwalają sformułować następujące zależności

$$\begin{aligned} FDC^{II} &\preceq FDC_2^I \\ FDC_1^I &\equiv FD \\ FDC^I &\preceq FDC^{II}. \end{aligned}$$

Uwzględniając te zależności możemy już uzupełnić naszą hierarchię klas programów.

WNIOSEK 6.6. *Istnieje następujący łańcuch zależności pomiędzy klasami programów:*

$$FD \equiv FDC_1^I \equiv FDC_1^{II} \prec FDC_2^I \equiv FDC_2^{II} \equiv FDC^I \equiv FDC^{II} \prec ED.$$

W ten sposób udało się zbudować bogaty pomost pomiędzy jedną z najbardziej podstawowych, stosunkowo prostą klasą programów prostych FD, a klasą definicji efektywnych ED, która jest uważana powszechnie za klasę intuicyjnie definiującą wszystkie funkcje obliczalne we wszystkich strukturach. Umożliwia on ocenę, które elementy struktury programów istotnie przyczyniają się do zwiększenia mocy obliczeniowej, a które służą jedynie do uproszczenia ich zapisu. Jednocześnie udostępnia bogatszy wybór równoważnych mechanizmów obliczeniowych w badaniu i porównywaniu mocy modeli licznikowych z innymi modelami obliczeń.

W ostatniej części rozdziału czwartego przedstawiliśmy twierdzenie 4.8, które podaje warunek, jaki musi spełniać struktura algebraiczna, aby klasa programów z licznikami dwukierunkowymi FDC^{II} była w tej strukturze uniwersalna. Warunek uniwersalności możemy odnieść do wszystkich klas równoważnych klasie FDC^{II}.

WNIOSEK 6.7. *Klasy programów FDC_2^I, FDC_2^{II}, FDC^I i FDC^{II} są uniwersalne, to znaczy są równoważne klasie definicji efektywnych ED we wszystkich strukturach algebraicznych $\mathcal{A}$ spełniających następujący warunek:*
dla każdego całkowitego $k \geq 1$ istnieje całkowite $n \geq k$ takie, że dla każdych $a_1, \ldots, a_k \in A$ i dla każdego termu b nad zbiorem zmiennch $x_1, \ldots, x_k$ istnieje program prosty $fd \in FD$ k-argumentowy z n zmiennymi, który oblicza wartość termu $v(b)$ dla wartości wejściowych $a_1, \ldots, a_k$ i wartościowania v takiego, że $v(x_i) = a_i$.

Wniosek ten umożliwia uproszczenie badań nad obliczalnością w dużej klasie struktur, wśród których znajdują się wszystkie podstawowe struktury algebraiczne używane w matematyce, poprzez ograniczenie się do badania obliczalności klasy programów z dwoma licznikami jednokierunkowymi FDC_2^I lub klasy programów z dwoma licznikiami dwukierunkowymi FDC_2^{II}. Ze względu na uniwersalność tych dwóch klas w tych strukturach uzyskane rezultaty będą dotyczyć wszystkich funkcji obliczalnych w takich strukturach.

Wniosek ten to również uogólnienie faktu, że maszyny dwulicznikowe jednokierunkowe i dwukierunkowe potrafią symulować maszyny Turinga. Dziedziny tych maszyn to tak naprawdę struktury spełniające warunek uniwersalności programów z licznikami.

Rozwinięcie teorii obliczeń, zarówno ogólnej jak i tej dotyczącej modeli licznikowych, można znaleźć w literaturze wyszczególnionej w ostatniej części pracy. Treść pracy została oparta na artykułach Turinga [**15**], Minsky'ego [**11**], Friedmana [**3**], Plaisteda

[**13**], Garlanda i Luckhama [**4**] oraz Kfoury'ego [**7**]. Odwołania do nich możemy znaleźć przy odpowiednich twierdzeniach i faktach. Teoria modeli licznikowych przedstawiona jest również w pracach Fischera, Meyera i Rosenberga [**1**], [**2**]. Prace Hewitta i Patersona [**5**] oraz Shepherdsona i Sturgisa [**14**] dotyczą klasy programów rekurencyjnych. Praca doktorska McKaya [**10**] rozważa problem równoważności programów w różnych klasach. Natomiast książki Minsky'ego [**12**], Hopcrofta i Ullmana [**6**], Lewisa i Papadimitriou [**9**] oraz Kozena [**8**] to podręczniki zawierające ogólne wprowadzenie do teorii obliczeń pochodzące z różnych okresów rozwoju tego kierunku.

Literatura

[1] P. C. Fischer, *Turing machines with restricted memory access*, w "Information and Control", Nr 4, 1966, strony 364–379.

[2] P. C. Fischer, A. R. Meyer, A. L. Rosenberg, *Counter machines and counter languages*, w "Mathematical Systems Theory", Nr 3, 1968, strony 265–283.

[3] H. Friedman, *Algorithmic procedures, generalized Turing algorithms, and elementary recursion theory*, w "Logic Colloquium '69", North Holland, Amsterdam, 1971, strony 361–389.

[4] S. J. Garland, D. C. Luckham, *Program schemes, recursion schemes, and formal languages*, w "Journal of Computer and System Sciences", Nr 2, 1973, strony 119–160.

[5] C. E. Hewitt, M. S. Paterson, *Comparative schematology*, w "Record of Project MAC Conference on Concurrent Systems and Parallel Computation", ACM, New York, 1970, strony 119–128.

[6] J. E. Hopcroft, J. D. Ullman, *Introduction to automata theory, languages and computation*, Addison-Wesley Publishing Company, Inc., 1979.

[7] A. J. Kfoury, *Translatability of schemas over restricted interpretations*, w "Journal of Computer and System Sciences", Nr 8, 1974, strony 387–408.

[8] D. C. Kozen, *Automata and Computability*, Springer-Verlag, New York, Inc., 1997.

[9] H. R. Lewis, Ch. H. Papadimitriou, *Elements of the theory of computation*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

[10] R. I. McKay, *The equivalence of programme schemes in structures with equality*, PhD Thesis, University of Bristol, 1976.

[11] M. L. Minsky, *Recursive unsolvability of Post's problem of 'tag' and other topics in the theory of Turing machines*, w "Annals of Mathematics", Nr 3, 1961, strony 437–455.

[12] M. L. Minsky, *Computation: finite and infinite machines*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1967.

[13] D. A. Plaisted, *Flowchart schemata with counters*, w "Proceedings of the Fourth Annual ACM Symposium on the Theory of Computing", Denver, Colorado, 1972, strony 44–51.

[14] J. C. Shepherdson, H. E. Sturgis, *Computability of Recursive Functions*, w "Journal for the Association for Computing Machinery", Nr 10, 1963, strony 217–255.

[15] A. M. Turing, *On computable numbers with an application to the Entscheidungsproblem*, w "Proceedings of the London Mathematical Society", Nr 2, 1936, strony 230–265.