

3 Stosy i nie tylko

A flow-diagram with a *stack* (or a *push-down store*) is defined as a flow-diagram which may involve two additional instruction types:

- $i : \text{push}(x); \text{goto } j;$
- $i : x := \text{pop}; \text{goto } j;$

for an arbitrary variable x .

Operational semantics for flowcharts with stacks is defined as for flowcharts but modified as follows: A state (in a structure A) is now of the form $\langle i, v, s \rangle$, where i and v are as before and s (the stack) is a sequence (possibly empty) of elements of A . (The convention is that the stack grows from left to right.) In the initial state we have $s = \varepsilon$.

The effect of executing “ $\text{push}(x); \text{goto } j$ ” on $\langle i, v, s \rangle$ is $\langle j, v, s \cdot v(x) \rangle$, and the effect of executing “ $i : x := \text{pop}; \text{goto } j$ ” on $\langle i, v, s \cdot a \rangle$ is $\langle j, v[a/x], s \rangle$. Executing a *pop* on empty stack results in an infinite loop, i.e., we assume $\langle i, v, \varepsilon \rangle \longrightarrow \langle i, v, \varepsilon \rangle$.

Twierdzenie 1 *Flow-diagramy ze stosem obliczają w dowolnej strukturze te same funkcje co flow-diagramy z procedurami.*

Dowód: Porządny dowód tego twierdzenia jest dość skomplikowany, my zrobimy tylko pobieżny szkic obu translacji. Założymy od razu, że mamy do czynienia ze strukturami wyposażonymi w dostatecznie wiele różnych stałych aby można było ich używać jako nazw procedur, zmiennych i etykiet instrukcji.¹

Część 1: Przypuśćmy, że $P = \langle P_1, \dots, P_n \rangle$ jest programem z procedurami. (Bez straty ogólności można zakładać, że procedura P_1 (program główny) nigdy nie jest wywoływana rekurencyjnie.) Skonstruujemy równoważny flow-diagram ze stosem S . Będzie on miał prawie wszystkie te instrukcje co procedury P_1, \dots, P_n i jeszcze trochę. Oczywiście wołania procedur i instrukcje **stop** trzeba zastąpić operacjami na stosie.

Poniżej (m, i) jest etykietą i -tej instrukcji procedury P_m . Wołanie procedury postaci

$$(m, i) : \quad x_p := P_\ell(x_{i_1}, \dots, x_{i_r}); \text{ go to } j$$

zastępujemy ciągiem instrukcji

$$\text{push}(x_1); \text{push}(x_2); \dots; \text{push}(x_k); \text{push}(m); \text{push}(p); \text{push}(j); \\ z_1 := x_{i_1}; \dots; z_r := x_{i_r}; z_{r+1} := z_1; \dots; z_s := z_1; \text{ go to } (\ell, 1),$$

¹W istocie można sobie poradzić bez tego. Najpierw trzeba zauważyć, że wystarczy mieć dwie rozróżnialne wartości, potem skonstruować podprogram zwany *lokator*, który potrafi takie dwie wartości znaleźć, lub sam wykonać żadaną symulację. Bo jeśli wszystkie wartości zmiennych zachowują się tak samo, to ta symulacja musi być trywialna.

gdzie x_1, \dots, x_k i z_1, \dots, z_s to wszystkie zmienne odpowiednio w P_m i P_ℓ , przy tym z_1, \dots, z_r są wejściowe. Instrukcje $push(m)$; $push(p)$; $push(j)$ należy rozumieć jako polecenia odłożenia na stos elementów struktury używanych do identyfikacji procedury, zmiennej i instrukcji. W ten sposób zapisujemy na stosie bieżącą aktywację i przechodzimy do nowej procedury.

W razie natrafienia na instrukcję **stop** programu głównego kończy się też obliczenie programu ze stosem. Instrukcja postaci $(m, i) : \mathbf{stop}(x_j)$, dla $m \neq 1$, jest zastąpiona przez:

$$v := x_j; l := pop; p := pop; n := pop; y_k := pop; y_{k-1} := pop; \dots; y_1 := pop; y_p := v; \mathbf{go\ to} (n, l).$$

Powyższe „instrukcje” są skrótowym i nieformalnym zapisem następujących operacji. Najpierw zapamiętujemy wartość wynikową, używając pomocniczej zmiennej v . Potem odczytujemy ze stosu numer p zmiennej, na którą należy tę wartość podstawić, oraz identyfikację instrukcji (n, j) do której należy wrócić. Następnie odtwarzamy ze stosu wartości zmiennych y_1, \dots, y_k procedury P_n , poprawiamy wartość zmiennej y_p i powracamy do wykonywania P_n w miejscu, w którym wywołano zakończoną właśnie procedurę. Oczywiście nie możemy bezpośrednio posłużyć się np. numerem p jako parametrem instrukcji $y_p := v$, nie wiemy też a priori, gdzie podstawiać wartości pobrane ze stosu itd. Dlatego powyższy „ciąg” jest w istocie skomplikowanym układem instrukcji warunkowych, wybierającym jedną z wielu możliwości.

Część 2: Teraz niech S będzie programem ze stosem o zmiennych x_1, \dots, x_k . Dla uproszczenia założymy (bez straty ogólności), że w S jest dokładnie jedna instrukcja pop postaci

$$i_0: x_1 := pop; \mathbf{go\ to} i_0 + 1,$$

i że wszystkie instrukcje $push$ mają postać

$$j: push(x_2); \mathbf{go\ to} j + 1.$$

Także bez straty ogólności można zakładać, że program S zawsze zatrzymuje się z pustym stosem (ćwiczenie). Najpierw pokażemy jak symulować taki program S za pomocą funkcji „wielowymiarowych”, wywoływanych tak:

$$(x_1, \dots, x_k) := P(x_1, \dots, x_k).$$

Instrukcję $push$ jak wyżej zastępujemy wołaniem

$$j: (x_1, \dots, x_k) := Push_j(x_1, \dots, x_k); \mathbf{go\ to} i_0 + 1,$$

a instrukcję pop zamieniamy na

$$\mathbf{stop}(top, x_2, \dots, x_k).$$

Ciało każdej procedury $Push_j$ zaczyna się tak:

$$\mathbf{start}(x_1, \dots, x_k); top := x_2; \mathbf{go\ to} j + 1,$$

i zawiera poza tym wszystkie instrukcje programu S .

Nasza symulacja oparta jest na takim prostym pomysle: każde odłożenie kolejnej wartości na stos odpowiada otwarciu nowej aktywacji procedury, która nadaje tę wartość pomocniczej zmiennej top . Zdjęcie wartości ze stosu jest symulowane zakończeniem bieżącej aktywacji.

Pozostaje pytanie jak zamienić wielowymiarowe funkcje na zwykłe: można to zrobić wielokrotnie wywołując tę samą procedurę, traktując coraz to inną zmienną jako wynikową. Nic złego się nie stanie, bo nie ma efektów ubocznych. \square

Rozpoznawanie skończoności

Pokażemy, że istnieje program rekurencyjny z równością (a więc i program ze stosem), zatrzymujący się dokładnie na tych danych wejściowych, które generują skończone podstruktury

W tym celu, dla dowolnej struktury \mathcal{A} i wektora $v = (a_0, \dots, a_{k-1}) \in \mathcal{A}^k$ zdefiniujemy ciąg b_i^v elementów \mathcal{A} (a w istocie podstruktury generowanej w \mathcal{A} przez a_0, \dots, a_{k-1}). Bez straty ogólności możemy przyjąć, że wszystkie operacje struktury \mathcal{A} mają tę samą liczbę argumentów p . Powiedzmy, że te operacje są ponumerowane: f_1, \dots, f_n .

Jako b_0^v przyjmujemy element a_0 . Jeśli b_0^v, \dots, b_m^v są już określone, to mamy dwa przypadki:

- Istnieje takie j , że $a_j \notin \{b_0^v, \dots, b_m^v\}$. Wtedy jako b_{m+1}^v bierzemy takie $a_j \notin \{b_0^v, \dots, b_m^v\}$, że j jest najmniejsze możliwe.
- Nie ma takiego j , tj. wszystkie generatory są już w zbiorze $\{b_0^v, \dots, b_m^v\}$. Wtedy definiujemy b_{m+1}^v jako $f_r(b_{i_1}^v, \dots, b_{i_p}^v)$, gdzie (i_1, \dots, i_p, r) jest najwcześniejszym leksykograficznie ciągiem o tej własności, że $f_r(b_{i_1}^v, \dots, b_{i_p}^v) \notin \{b_0^v, \dots, b_m^v\}$.

Oto własności ciągu b_m^v : Jeśli podstruktura generowana w \mathcal{A} przez a_0, \dots, a_{k-1} jest skończona, to ciąg b_m^v jest skończony i jego wartościami są wszystkie elementy tej podstruktury. Jeśli podstruktura jest nieskończona, to ciąg b_m^v jest nieskończony. W obu przypadkach ciąg ten jest różnowartościowy (bez powtórzeń).

Na rysunku 1 mamy rekurencyjną procedurę NEXT, która w dowolnej strukturze \mathcal{A} , dla danych $a_0, \dots, a_{k-1}, b_m^v$ oblicza b_{m+1}^v . Stąd natychmiast wynika nasze twierdzenie: wystarczy iterować NEXT tak długo, aż nie da się uzyskać nowego elementu (wynik jest równy ostatniej z wartości wejściowych).

Twierdzenie 2 *Dla dowolnego k istnieje k -argumentowy program z procedurami P , zatrzymujący się w dowolnej strukturze \mathcal{A} dla danych a_0, \dots, a_{k-1} wtedy i tylko wtedy, gdy podstruktura generowana w \mathcal{A} przez a_0, \dots, a_{k-1} jest skończona.*

Obliczenia w strukturze liczb naturalnych

W ogólności rekursja (stos) istotnie zwiększa siłę obliczeniową języka flow-diagramów. Jednak w strukturze liczb naturalnych $\mathcal{N} = \langle \mathbb{N}, s, 0, = \rangle$, gdzie s oznacza funkcję następnika jest inaczej.

Twierdzenie 3 *Każdy program ze stosem jest równoważny w \mathcal{N} pewnemu flow-diagramowi.*

Dowód: Operacje na stocie symulujemy używając dodatkowej zmiennej s , korzystając z funkcji pary \langle , \rangle i operacji odwrotnych ℓ, r . Operację $push(x)$ zastępuje $s := \langle s, x \rangle$, a zamiast $x := pop$ wykonujemy $x := r(s)$; $s := \ell(s)$. \square

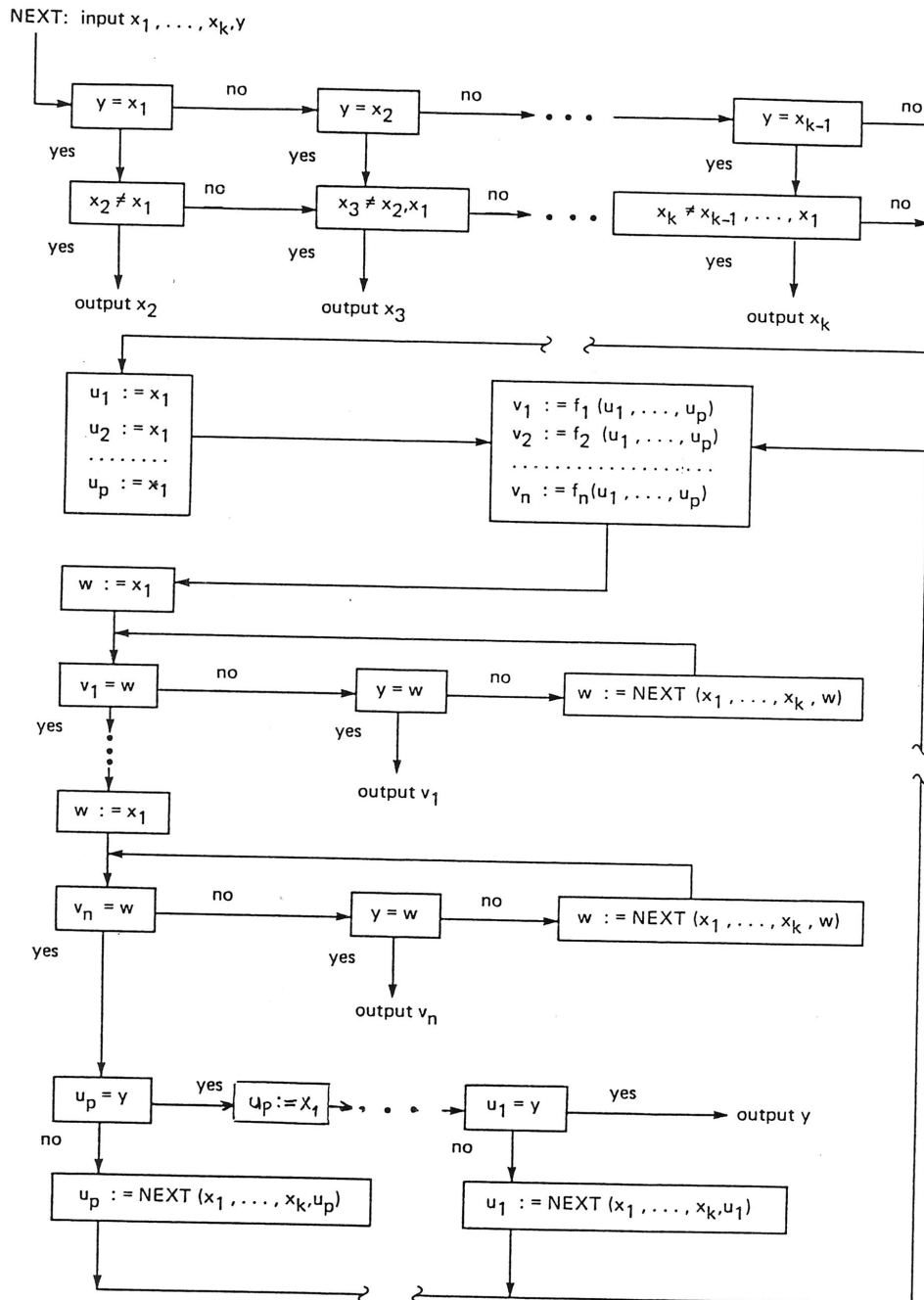


Figure 1: Program NEXT