„być liczbą naturalną" i in. nie są wyrażalne przy pomocy formuł języka pierwszego rzędu.

Rozważmy język arytmetyki, który zawiera stałe 0 i *Max*, jednoargumentowy funktor *s* oraz znak równości =. Wykażemy, że następujący zbiór formuł AxA, w odpowiednim języku algorytmicznym, stanowi specyfikację klasy arytmetyk **integer**:

A1 $(\forall x)\,(x = Max \equiv s(x) = 0)$
A2 $Max \neq 0$
A3 $(\forall x, y)(s(x) = s(y) = x = y)$
A4 $(\forall x)(y := 0;$ **while** $\neg x = y$ **do** $y := s(y)$ **od) true**

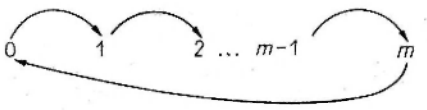
TWIERDZENIE 5.2

Każdy model przedstawionego układu aksjomatów AxA jest izomorficzny z pewną strukturą $\mathbf{N}_m$ (por. rys. 5.2) taką, że

$$\mathbf{N}_m = \langle\{0, ..., m\}, 0, m, +_1, =\rangle$$

gdzie $\{0, ..., m\}$ jest $m+1$-elementowym podzbiorem zbioru liczb naturalnych, 0 i m są wyróżnionymi stałymi, a $+_1$ jednoargumentową funkcją określoną następująco:

$+_1(i) = i+1$ dla $i < m$
$+_1(m) = 0$



RYS. 5.2

DOWÓD

Na początek zauważmy, że dla dowolnej liczby naturalnej $m > 0$, struktura $\mathbf{N}_m$ jest modelem zbioru aksjomatów AxA. Rzeczywiście, aksjomat A1 jest prawdziwy w $\mathbf{N}_m$, ponieważ funkcja $+_1$ daje w wyniku 0 tylko dla argumentu m. Z przyjętej definicji $+_1$ jest funkcją całkowitą i różnowartościową co implikuje prawdziwość aksjomatów A2 i A3 w strukturze $\mathbf{N}_m$. Na koniec aksjomat A4 jest prawdziwy w $\mathbf{N}_m$ ponieważ uniwersum struktury jest skończone i każdy element można otrzymać z dowolnego innego elementu przez zastosowanie, skończonej liczby razy, operacji $+_1$ (por. rys. 5.2)

Rozważmy dowolny model **A** zbioru AxA

$$\mathbf{A} = \langle A, a_0, a_{max}, s_{\mathbf{A}}, =\rangle$$

gdzie a_0, a_{max} są odpowiednio interpretacjami stałych 0 i *Max* z rozważanego języka. Niech m będzie najmniejszą liczbą naturalną taką, że

$s_{\mathbf{A}}^{m}(a_0) = a_{max}$

Takie m istnieje, gdyż dla wartościowania v takiego, że $v(y) = a_0$, $v(x) = a_{max}$ mamy, na mocy aksjomatu A4,

$\mathbf{A}, v \models$ **while** $\neg x = y$ **do** $y := s(y)$ **od true**

Niech h będzie odwzorowaniem struktury $\mathbf{N}_m$ w $\mathbf{A}$ określonym następująco:

$h(0) = a_0 \quad h(i) = s_{\mathbf{A}}^{i}(a_0)$ dla $i \leqslant m$

Czy h jest odwzorowaniem na zbiór A?

Weźmy dowolny element $a \in A$. Na mocy aksjomatu A4 istnieje najmniejsze i takie, że $s_{\mathbf{A}}^{i}(a_0) = a$. Gdyby $i > m$, to

$$a = s_{\mathbf{A}}^{i-m-1}(s_{\mathbf{A}}(s_{\mathbf{A}}^{m}(a_0))) = s_{\mathbf{A}}^{i-m-1}(s_{\mathbf{A}}(a_{max}))$$

Stąd na mocy aksjomatu A1 mielibyśmy $a = s_{\mathbf{A}}^{i-m-1}(a_0)$, czyli i nie jest najmniejszą liczbą naturalną taką, że $s_{\mathbf{A}}^{i}(a_0) = a$. Otrzymana w ten sposób sprzeczność dowodzi, że $i \leqslant m$. Wykazaliśmy tym samym, że istnieje takie i, dla którego $h(i) = a$ i w konsekwencji udowodniliśmy, że h jest odwzorowaniem na zbiór A.

Czy h jest funkcją różnowartościową?

Niech $i \leqslant j \leqslant m$ oraz $i \neq j$. Gdyby $h(i) = h(j)$ wówczas $s_{\mathbf{A}}^{i}(a_0) = s_{\mathbf{A}}^{j}(a_0)$. Stąd na mocy aksjomatu A3 byłoby $a_0 = s_{\mathbf{A}}^{j-i}(a_0)$, a na mocy aksjomatu A1, $s_{\mathbf{A}}^{j-i-1}(a_0) = a_{max}$. Ale $j-i-1 < m$. Otrzymaliśmy sprzeczność z przyjętą wcześniej definicją liczby m. Wynika stąd, że dla $i \neq j$ musi być $h(i) \neq h(j)$, tzn. h jest funkcją różnowartościową.

Czy h jest izomorfizmem?

Weźmy dowolną liczbę naturalną $0 \leqslant i < m$. Wówczas mamy:

$$h(+_1(i)) = h(i+1) = s_{\mathbf{A}}^{i+1}(a_0) = s_{\mathbf{A}}(s_{\mathbf{A}}^{i}(a_0)) = s_{\mathbf{A}}(h(i))$$

oraz

$$h(+_1(m)) = h(0) = a_0 = s_{\mathbf{A}}(a_{max}) = s_{\mathbf{A}}(h(a_{max}))$$

Wykazaliśmy w ten sposób, że wszystkie modele zbioru AxA ustalonej mocy są izomorficzne. □

W podobny sposób można zaksjomatyzować (wyspecyfikować) inne typy pierwotne dowolnego języka programowania, nawet tak skomplikowane, jak struktura liczb zmiennoprzecinkowych.

UWAGA

Twierdzenia postaci 5.1, 5.2 (i podobne) będą nazywane *twierdzeniami o reprezentacji.*

Kolejki 5.3

W tym punkcie omówimy strukturę kolejek (por. p. 2.3). Czym jest kolejka wie każdy. Zastanówmy się jednak, w jaki sposób opisać pojęcie kolejki tak, by opis ten był wystarczająco precyzyjny.

Przez kolejkę chcemy rozumieć obiekt, który powstaje w wyniku operacji typowych dla kolejek i który sam jest argumentem dla innych operacji na kolejkach. Chcemy więc opisać strukturę, w której obiektami są kolejki i ich elementy, a na tych obiektach są dokonywane operacje: wstaw element do kolejki (wynik tej operacji jest nową kolejką), usuń z kolejki jej pierwszy element (wynik i argument są kolejkami), podaj pierwszy element z kolejki (argument jest kolejką, a wynik jest elementem) oraz relacje: sprawdź czy kolejka jest pusta, porównaj dwa elementy.

Czy taki opis jest wystarczający? Nie. Nie każdą bowiem strukturę danych, w której występują: jedna operacja dwuargumentowa, dwie operacje jednoargumentowe oraz jedna relacja jednoargumentowa, chcemy uważać za strukturę kolejek. Należy więc wyspecyfikować jakie własności mają mieć operacje, o których była mowa.

Niech L będzie językiem pierwszego rzędu dwusortowym, w którym występują zmienne indywiduowe typu E i typu Q oraz następujące funktory i predykaty:

o typu $Q \to Q$
p typu $E \times Q \to Q$
f typu $Q \to E$
em typu $Q \to B_0$
$\underset{E}{=}$ typu $Q \times Q \to B_0$

Oznaczmy przez AxQ następujący zbiór formuł w języku algorytmicznym $L(\pi)$:

Q1 $(em(q) \Rightarrow eq(q, o(p(e, q))))$

Q2 $(\neg em(q) \Rightarrow eq(p(e, o(q)), o(p(e, q))))$

Q3 $(em(q) \Rightarrow (e \underset{E}{=} f(p(e, q))))$

Q4 $(\neg em(q) \Rightarrow (f(p(e, q)) \underset{E}{=} f(q)))$

Q5 $\neg em(p(e, q))$

Q6 **while** $\neg em(q)$ **do** $q := o(q)$ **od true**

Q7
```
eq(q1, q2) ≡ begin
                bool := true;
                while ¬em(q1) ∧ ¬em(q2) ∧ bool
                do
                  if ¬f(q1) =E f(q2)
                  then
                    bool := false
```

fi;
$q1 := o(q1)$;
$q2 := o(q2)$
od $(bool \wedge em(q1) \wedge em(q2)$

Q8 $\neg em(q) \equiv ok(o(q))$

Q9 $\neg em(q) \equiv ok(f(q))$

Q10 $ok(p(e, q))$

Teorię algorytmiczną w języku $L(\pi)$, której zbiorem aksjomatów specyficznych jest zbiór AxQ, będziemy nazywać algorytmiczną teorią kolejek ATQ.

TWIERDZENIE 5.3

Zbiór formuł AxQ jest niesprzeczny.

DOWÓD

Twierdzimy, że dla dowolnego zbioru E, standardowa struktura kolejek

$$\langle E \cap Seq(E); put, out, first; empty, \underset{E}{=}, \underset{Q}{=} \rangle$$

jest modelem zbioru AxQ (por. definicję 2.13). Zakładamy, że interpretacjami funktorów p, o i f są odpowiednio *put*, *out* i *first*. $\underset{Q}{=}$ jest interpretacją predykatu *eq*, a *empty* jest interpretacją predykatu *em*. Prześledźmy sens poszczególnych aksjomatów.

AKSJOMAT Q1

Usunięcie pierwszego elementu z kolejki powstającej przez wstawienie elementu do kolejki pustej daje w wyniku kolejkę pustą.

AKSJOMAT Q2

Operacje dodawania elementu i usuwania elementu, wykonywane na niepustej kolejce, są przemienne.

AKSJOMAT Q3

Jeżeli do pustej kolejki wstawimy jeden element, to jest on elementem pierwszym.

AKSJOMAT Q4

Jeżeli kolejka nie jest pusta, to operacja wstawiania do kolejki nowego elementu nie zmienia elementu pierwszego.

AKSJOMAT Q5

Kolejka, do której wstawiliśmy jakikolwiek element, jest niepusta.

AKSJOMAT Q6

Każda kolejka ma tylko skończoną liczbę elementów.

AKSJOMAT Q7

Dwie kolejki są równe tylko wtedy, gdy są tej samej długości i na tych samych pozycjach zawierają te same elementy. Aksjomaty Q8, Q9, Q10 opisują dziedziny operacji. Operacja wstawiania jest zawsze określona. Operacje usuwania i wskazywania pierwszego elementu są określone tylko na kolejkach niepustych.

Jest oczywiste, że wszystkie te zdania są prawdziwe, gdy przez kolejkę rozumiemy dowolny skończony ciąg, w którym operacja wstawiania polega na dopisaniu wskazanego elementu na końcu ciągu, a operacja usuwania na pominięciu pierwszego elementu. □

Zwróćmy uwagę, że równość kolejek nie jest przyjmowana jako pojęcie pierwotne. Zdefiniowano ją algorytmicznie. Dalej wykażemy, że program występujący w aksjomacie Q7 nie zapętla się w żadnym modelu zbioru formuł Q1–Q10. Uwaga ta ma większy sens niż to się może zdawać na pierwszy rzut oka. Znane są systemy, w których operacje są efektywne, a pomimo to porównywanie elementów nie jest operacją efektywną.

LEMAT 5.2

Formuła **K true** jest twierdzeniem teorii ATQ, gdzie K oznacza następujący program

```
begin
  bool := true;
  while ¬em(q1) ∧ ¬em(q2) ∧ bool
    do
      if ¬f(q1) ≡_E f(q2)
      then
        bool := false
      fi;
      q1 := o(q1);
      q1 := o(q2)
    od
end
```

Dowód

Oznaczamy przez K_1 i K_2 następujące programy:

K_1: **if** $\neg em(q1) \wedge \neg em(q2)$ **then** **if** $\neg f(q1) \underset{E}{=} f(q2)$ **then** $bool := \mathbf{false}$ **fi** **fi**

K_2: **if** $\neg em(q2)$ **then** $q2 := o(q2)$ **fi**

Na mocy aksjomatu AxQ6

$$ATQ \vdash \mathbf{while}\ \neg em(q1)\ \mathbf{do}\ q1 := o(q1)\ \mathbf{od}\ \mathbf{true}$$

Ponieważ

$$ATQ \vdash K_1\,\mathbf{true}, \qquad ATQ \vdash K_2\,\mathbf{true}$$

a zmienna $q1$ nie występuje ani w programie K_1 ani w K_2, zatem na mocy reguły pomocniczej udowodnionej w przykładzie 4.9 oraz reguły modus ponens R1 mamy

$$ATQ \vdash \mathbf{while}\ \neg em(q1)\ \mathbf{do}\ K_1;\ q1 := o(q1);\ K_2\ \mathbf{od}\ \mathbf{true}$$

Jednakże formuła

$$((\neg em(q1) \wedge \neg em(q2) \wedge bool) \Rightarrow \neg em(q1))$$

jest tautologią na mocy aksjomatu Ax5, więc na mocy reguły udowodnionej w przykładzie 4.6 mamy

$$ATQ \vdash \mathbf{while}\ \neg em(q1) \wedge \neg em(q2) \wedge bool\ \mathbf{do}\ K_1;\ q1 := o(q1);\ K_2\ \mathbf{od}\ \mathbf{true}$$

Stosując dwukrotnie regułę z przykładu 4.10 oraz regułę R2 w postaci

$$\frac{\alpha,\ \mathbf{M}\,\mathbf{true}}{\mathbf{M}\alpha}$$

otrzymamy ostatecznie tezę $ATQ \vdash K\,\mathbf{true}$. □

Następujący lemat ustala podstawowe własności predykatu *eq*.

Lemat 5.3

Dla dowolnych $q, q', q'' \in V_Q$ i dowolnych $e, e' \in V_E$

(1) $ATQ \vdash eq(q, q)$

(2) $ATQ \vdash eq(q, q') \Rightarrow eq(q', q)$

(3) $ATQ \vdash (eq(q, q') \wedge eq(q, q'')) \Rightarrow eq(q', q'')$

(4) $\mathrm{ATQ} \vdash (e \underset{E}{=} e' \wedge eq(q, q')) = eq(p(e, q), p(e', q'))$

(5) $\mathrm{ATQ} \vdash (eq(q, q') \wedge \neg em(q)) \Rightarrow (eq(o(q), o(q')) \wedge f(q) \underset{E}{=} f(q'))$

(6) $\mathrm{ATQ} \vdash eq(q, q') \Rightarrow em(q) \equiv em(q')$

(7) $\mathrm{ATQ} \vdash (em(q) \wedge em(q')) \Rightarrow eq(q, q')$ ■

Wykażemy, że zbiór aksjomatów AxQ jest zupełną specyfikacją pojęcia kolejki i że jest on przydatny w analizie algorytmów używających kolejki jako struktury danych. W dalszym ciągu pokażemy też, w jaki sposób można użyć tych aksjomatów w procesie weryfikacji poprawności danej implementacji struktury kolejek.

Następujące twierdzenie będziemy nazywać twierdzeniem o reprezentacji kolejek.

TWIERDZENIE 5.4

Niech

$$\mathbf{A} = \langle A \cup S, p_{\mathbf{A}}, o_{\mathbf{A}}, f_{\mathbf{A}}, em_{\mathbf{A}}, \underset{A}{=}, eq_{\mathbf{A}} \rangle$$

będzie dowolnym modelem zbioru formuł AxQ i niech $\approx$ będzie relacją kongruencji określoną następująco:

$$x \approx y \quad \text{wtw} \quad x, y \in A \text{ i } x \underset{A}{=} y \text{ lub } x, y \in S \text{ i } eq_{\mathbf{A}}(x, y)$$

Wtedy struktura standardowa wyznaczona przez ten sam zbiór elementów A

$$\langle A \cup Seq(A), put, out, first, empty, \underset{A}{=}, \underset{Q}{=} \rangle$$

jest izomorficzna z $\mathbf{A}/\approx$.

DOWÓD

Oznaczmy przez p', o', f', em', $='$, eq' operacje i relacje systemu $\mathbf{A}/\approx$ oraz przez [x] klasę abstrakcji relacji $\approx$ wyznaczoną przez dowolny element x ze zbioru $A \cup S$. Zauważmy, że na mocy twierdzenia 4.5 struktura $\mathbf{A}/\approx$ jest modelem właściwym zbioru aksjomatów AxQ (relacja eq' jest identycznością w $(A \cup S)/\approx$). Wynika stąd, że w $\mathbf{A}/\approx$ istnieje dokładnie jeden element $s \in S/\approx$ taki, że $em'(s)$ (por. lemat 5.3 własność (6)). Oznaczmy ten jedyny element przez $\emptyset'$.

Niech h będzie funkcją odwzorowującą zbiór $A \cup Seq(A)$ w zbiór $(A \cup S)/\approx$ taką, że

$$h(a) = [a] \qquad h(\emptyset) = \emptyset'$$
$$h((a_0, \ldots, a_n)) = p'([a_n], p'([a_{n-1}], \ldots, p'([a_0], \emptyset') \ldots))$$

dla $a \in A$ oraz dla dowolnego ciągu $(a_0, \ldots, a_n) \in Seq(A)$.

Pokażemy, że h jest odwzorowaniem na $\mathbf{A}/\approx$, tzn. pokażemy, że dla dowolnej kolejki $s \in S/\approx$ istnieje ciąg elementów $a_0, \ldots, a_n$ zbioru A taki, że $s = h((a_0, \ldots, a_n))$. Niech v będzie wartościowaniem w zbiorze $(A \cup S)/\approx$ i niech $v(q) = s$, $s \neq \emptyset'$. Na mocy aksjomatu Q6

$$\mathbf{A}/\approx, v \models \textbf{while } \neg em(q) \textbf{ do } q := o(q) \textbf{ od true}$$

Istnieje zatem takie i, że dla wszystkich $j \leqslant i$

$$\mathbf{A}/\approx, v \models em(o^{i+1}(q)) \wedge \neg em(o^j(q))$$

Oznaczmy dla $0 \leqslant j \leqslant i$

$$[a_j] \stackrel{\text{df}}{=} f'(o'^j(s))$$
$$s_j \stackrel{\text{df}}{=} p'([a_j], p'([a_{j-1}], \ldots, p'([a_1], p'[a_0], \emptyset'))\ldots))$$

Przyjmijmy ponadto, że $v(e_j) = [a_j]$ oraz $v(q') = s_i$.

Na mocy aksjomatu Q5 $\neg em'(s_j)$ dla $0 \leqslant j \leqslant i$, a stąd i z aksjomatów Q4 i Q3 otrzymujemy

$$f'(s_i) = f'(p'([a_{i-1}], \ldots, p'([a_1], p'([a_0], \emptyset'))\ldots)) = f'(p'([a_0], \emptyset')) = [a_0]$$

Czyli

$$f'(s_i) = f'(s)$$

Korzystając z aksjomatu Q2 oraz lematu 5.3 otrzymujemy kolejno

$$\mathbf{A}/\approx, v \vdash eq(o(q'), p(e_i, o(p(e_{i-1}, \ldots p(e_1, p(e_0, \emptyset))\ldots))))$$
$$\ldots$$
$$\mathbf{A}/\approx, v \vdash eq(o(q'), p(e_i, p(e_{i-1}, \ldots p(e_1, o(p(e_0, \emptyset)))\ldots))))$$
$$\mathbf{A}/\approx, v \vdash eq(o(q'), p(e_i, p(e_{i-1}, \ldots, p(e_1, \emptyset)\ldots)))$$

Powtarzając to rozumowanie dla dowolnego $j \leqslant i$

$$\mathbf{A}/\approx, v \vdash eq(o^j(q'), p(e_i, p(e_{i-1}, \ldots p(e_j, \emptyset)\ldots)))$$

Stąd na mocy aksjomatów Q2 i Q3 mamy

$$f'(o'^j(s_i)) = [a_j] = f'(o'^j(s))$$

Rozważmy teraz program występujący w aksjomacie Q7. Przy wartościowaniu początkowym v obliczenie programu zakończy się dokładnie po $i+1$ krokach oraz instrukcja $bool :=$ **false** nie będzie nigdy wykonana. Wynika stąd, że po zakończeniu obliczenia będzie spełniony warunek

$$(bool \wedge em(q) \wedge em(q'))$$

tzn. na mocy aksjomatu Q7

$$\mathbf{A}/\approx, v \vdash eq(q, q')$$

W konsekwencji $h((a_0, \ldots a_i)) = s_i = s$. Wykazaliśmy w ten sposób, że h jest odwzorowaniem zbioru $A \cup Seq(A)$ na zbiór $(A \cup S)/\approx$.

Następnym krokiem będzie wykazanie, że h jest odwzorowaniem różnowartościowym. Niech seq_1 i seq_2 będą dwoma różnymi elementami zbioru $Seq(A)$. Przyjmijmy, że $a_j \neq b_j$ oraz

$$seq_1 \stackrel{df}{=} (a_0, \ldots, a_j, \ldots, a_n) \quad seq_2 \stackrel{df}{=} (a_0, \ldots, a_{j-1}, b_j \ldots b_m)$$
$$s_1 \stackrel{df}{=} h(seq_1) \quad s_2 \stackrel{df}{=} h(seq_2)$$

Załóżmy ponadto, że wartościowanie v jest określone następująco:

$$v(e_j) = [a_j],\ v(e'_j) = [b_j],\ v(q1) = s_1,\ v(q2) = s_2$$

Na mocy poprzednich rozważań mamy

$$\mathbf{A}/\approx, v \vdash eq(o^i(q1), p(e_n, \ldots, p(e_i, \emptyset)\ldots)) \quad \text{dla } i \leqslant n$$
$$\mathbf{A}/\approx, v \vdash eq(o^i(q2), p(e'_m, \ldots, p(e'_i, \emptyset)\ldots)) \quad \text{dla } i \leqslant m$$
$$\mathbf{A}/\approx, v \vdash f(o^i(q1)) \underset{E}{=} f(o^i(q2)) \quad \text{dla } i < j$$
$$non\ \mathbf{A}/\approx, v \vdash f(o^j(q1)) \underset{E}{=} f(o^j(q2))$$

Analizując program występujący w aksjomacie Q7 dochodzimy do wniosku, że j-ta iteracja programu M w strukturze $\mathbf{A}/\approx$ przy wartościowaniu v zakończy się wartościowaniem, w którym zmienna *bool* ma wartość **0**. Będzie to więc ostatnia iteracja programu M oraz

$$non\ \mathbf{A}/\approx, v \vdash \mathrm{M}^j(bool \wedge em(q1) \wedge em(q2))$$

Wynika stąd na mocy aksjomatu Q7, że

$$\mathbf{A}/\approx, v \vdash \neg eq(q1, q2)$$

a co za tym idzie $s_1 \neq s_2$. Oznacza to, że

$$h(seq1) \neq h(seq2)$$

Pozostaje pokazać, że h jest homomorfizmem. Rozważmy dowolny ciąg $seq \stackrel{df}{=} (a_0, \ldots, a_n)$ ze zbioru $Seq(A)$

$$h(first(seq)) = h(a_0) = f'(p'(h(a_0), \emptyset')) = f'(p'(h(a_n), \ldots, p'(h(a_0), \emptyset')\ldots)) = f'(h(seq))$$
$$h(put(a, seq)) = h((a_0, \ldots, a_n, a)) = p'(h(a), p'(h(a_n), \ldots, p'(h(a_0), \emptyset')\ldots)) = p'(h(a), h(seq))$$
$$h(out(seq)) = h((a_1, \ldots, a_n)) = p'(h(a_n), \ldots, p'(h(a_1), \emptyset)\ldots)) = p'(h(a_n), \ldots, p'(h(a_1), o'(p'(h(a_0), \emptyset'))\ldots)) = o'(p'(h(a_n), \ldots, p'(h(a_1), p'(h(a_0), \emptyset'))\ldots)) = o'(h(seq))$$

Te równości dowodzą, że h jest izomorfizmem odwzorowującym strukturę standardową kolejek na strukturę $\mathbf{A}/\approx$, co kończy dowód twierdzenia o reprezentacji. □

Twierdzenie 5.4 przekonuje nas, że udało się podać specyfikację struktury kolejek wolną od sprzeczności, a zarazem w pełni opisującą rodzinę możliwych implementacji tego pojęcia.

Kolejki priorytetowe 5.4

Niech L będzie językiem pierwszego rzędu, w którym występują zmienne indywiduowe dwóch typów E i PQ oraz następujące funktory i predykaty:

in	typu	$E \times PQ \to PQ$
del	typu	$E \times PQ \to PQ$
min	typu	$PQ \to E$
mb	typu	$E \times PQ \to B_0$
em	typu	$PQ \to B_0$
les	typu	$E \times E \to B_0$
$\underset{E}{=}$	typu	$E \times E \to B_0$
eq	typu	$PQ \times PQ \to B_0$

Dla prostoty, zmienne typu E będziemy oznaczać przez e, e' itd., a zmienne typu PQ przez q, q' itd.

Teorią kolejek priorytetowych ATPQ będziemy nazywać teorię algorytmiczną w języku $L(\pi)$ z równością, której aksjomaty specyficzne stanowi następujący zbiór formuł AxPq (oraz zbiór aksjomatów dla równości, por. p. 4.4):

Pq1 $les(e, e)$
$((les(e, e') \wedge les(e', e)) \Rightarrow e' \underset{E}{=} e)$
$((les(e, e_1) \wedge les(e_1, e_2)) \Rightarrow les(e, e_2))$
$(les(e, e') \vee les(e', e))$

Pq2 **while** $\neg em(q)$ **do** $q := del(min(q), q)$ **od true**

Pq3 $(\neg em(q) \Rightarrow (mb(e, q) \Rightarrow les(min(q), e)))$

Pq4 $mb(e, in(e, q))$

Pq5 $(\neg e \underset{E}{=} e' \Rightarrow (mb(e', q) \equiv mb(e', in(e, q))))$

Pq6 $\neg mb(e, del(e, q))$

Pq7 $(\neg e \underset{E}{=} e' \Rightarrow (mb(e', q) \equiv mb(e', del(e, q))))$

Pq8 $mb(e, q) \equiv$ **begin**
$e1 := e;$
$q1 := q;$
$bool :=$ **false**;
while $(\neg em(q1) \wedge \neg bool)$
do

$e2 := min(q1);$
$bool := (e1 \underset{E}{=} e2);$
$q1 := del(e2, q1)$
od
end *bool*

Pq9 $eq(q, q') =$ **begin**
$q1 := q; q2 := q';$
$bool :=$ **true**;
while $(bool \wedge \neg em(q1) \wedge \neg em(q2)$
do
$e1 := min(q1);$
$bool := bool \wedge mb(e1, q2);$
if *bool*
then
$q1 := del(e1, q1);$
$q2 := del(e1, q2);$
fi
od
end $(bool \wedge em(q1) \wedge em(q2))$

Pq10 $\neg em(q) \equiv ok(min(q)) \equiv (e := min(q))$ **true**

Pq11 $ok(in(e, q)) \equiv ok(del(e, q)) \equiv$ **true**

DEFINICJA 5.2

Każdy model zbioru aksjomatów AxPq będziemy nazywać kolejką priorytetową. ■

TWIERDZENIE 5.5

Zbiór aksjomatów AxPq jest niesprzeczny. Inaczej mówiąc, istnieje co najmniej jeden model zbioru AxPq.

DOWÓD

Następująca struktura jest modelem teorii ATPQ:

$$\langle A \cup Fin(A), insert, delete, minimum, member, empty, \leqslant, \underset{A}{=}, = \rangle$$

gdzie

A jest dowolnym zbiorem;
$Fin(A)$ jest zbiorem skończonych podzbiorów zbioru A;
$insert(e, q) = q \cup \{e\}$;
$member(e, q) = e \in q$;

$delete(e, q) = q - \{e\}$;
$empty(q) \equiv (q = \emptyset)$;
$\leqslant$ relacja liniowego porządku w zbiorze A;
$minimum(q)$ najmniejszy w sensie porządku $\leqslant$ element zbioru q;
$\underset{A}{=}$, $=$ relacje identyczności w A i w $Fin(A)$.

Tak zdefiniowaną strukturę będziemy nazywać standardową strukturą kolejek priorytetowych.

Proste sprawdzenie prawdziwości aksjomatów w tej strukturze pozostawiamy czytelnikowi. □

LEMAT 5.4

Programy występujące w aksjomatach Pq8 i Pq9 nie zapętlają się w żadnym modelu teorii ATPQ.
Dowód lematu wynika wprost z aksjomatu Pq2 i reguł pomocniczych przedstawionych w przykładach 4.9 i 4.6 (por. analogiczny dowód lematu 5.2). ■

Następujący lemat podaje proste przykłady twierdzeń teorii ATPQ. Wykorzystamy je w dalszej części rozdziału.

LEMAT 5.5

(1) $ATPQ \vdash \neg em(in(e, q))$
(2) $ATPQ \vdash (em(q) \equiv (\forall e) \neg mb(e, q)) \wedge (\neg em(q) \Rightarrow mb(min(q), q))$
(3) $ATPQ \vdash mb(e, in(e', q)) \equiv (e \underset{E}{=} e' \vee mb(e, q))$
(4) $ATPQ \vdash mb(e, del(e', q)) \equiv (\neg e \underset{E}{=} e' \wedge mb(e, q))$
(5) $ATPQ \vdash (em(q) \Rightarrow (mb(e, in(e', q)) \Rightarrow e \underset{E}{=} e'))$
(6) $ATPQ \vdash (em(del(e, q)) \Rightarrow (mb(e', q) \Rightarrow e \underset{E}{=} e'))$
(7) $ATPQ \vdash (em(q) \Rightarrow min(in(e, q)) \underset{E}{=} e)$
(8) $ATPQ \vdash (em(del(e, q)) \Rightarrow min(q) \underset{E}{=} e)$

DOWÓD

Oznaczmy przez γ formułę $(\neg em(q1) \wedge \neg bool)$ oraz przez K i M następujące programy:

```
M: begin                    K: begin
     e2 := min(q1);               bool := false;
     bool := (e1 =E e2);          q1 := in(e, q);
     q1 := del(e2, q1)            e1 := e
   end                         end
```

Ad (1)
Na mocy aksjomatu Pq4

$$\mathrm{ATPQ} \vdash mb(e, in(e, q))$$

Stąd i z aksjomatów Pq8 i Ax21

$$\mathrm{ATPQ} \vdash \mathrm{K}((\neg\gamma \wedge bool) \vee (\gamma \wedge \mathrm{M}(\textbf{while } \gamma \textbf{ do } \mathrm{M} \textbf{ od } bool)))$$

Stosując aksjomaty Ax15 i Ax18 oraz korzystając z tego, że formuła $\mathrm{K}(\neg\gamma \wedge bool) \equiv$ **false** jest tautologią logiki algorytmicznej otrzymujemy

$$\mathrm{ATPQ} \vdash \neg em(in(e, q)) \wedge \mathrm{K}(\mathrm{M}(\textbf{while } \gamma \textbf{ do } \mathrm{M} \textbf{ od } bool))$$

co na mocy Ax2 i reguły R1 daje

$$\mathrm{ATPQ} \vdash \neg em(in(e, q))$$

Ad (2)
Najpierw zauważmy, że na mocy praw de Morgana oraz prawa transpozycji (por. lemat 2.3) własność (2) jest równoważna następującej

$$\mathrm{ATPQ} \vdash (\exists e)\, mb(e, q) \equiv \neg em(q)$$

Dowód formuły $((\exists e)\, mb(e, q) \Rightarrow \neg em(q))$ pomijamy, ponieważ jest podobny do dowodu własności (1). Rozważmy formułę $\neg em(q) \Rightarrow (\exists e)\, mb(e, q)$. Oznaczmy przez K′ program

begin $e1 := min(q)$; $q1 := q$; $bool :=$ **false end**

Na mocy aksjomatu Pq10 mamy

$$\mathrm{ATPQ} \vdash (\neg em(q) \Rightarrow (\neg em(q) \wedge min(q) \underset{E}{=} min(q)))$$

Łatwo zauważyć, wobec aksjomatu Ax18, że formuła występująca w następniku jest równoważna formule

$$\mathrm{K}'(\gamma \wedge \mathrm{M}bool)$$

a ta z kolei, na mocy reguły pomocniczej przedstawionej w przykładzie 4.11, jest równoważna formule

$$\mathrm{K}'(\gamma \wedge \mathrm{M}(\textbf{while } \gamma \textbf{ do } \mathrm{M} \textbf{ od } bool))$$

Zastosowanie aksjomatów Ax21 i Ax2 pozwala wywnioskować, że

$$\mathrm{ATPQ} \vdash (\neg em(q) \Rightarrow \mathrm{K}'(\textbf{while } \gamma \textbf{ do } \mathrm{M} \textbf{ od } bool))$$

czyli na mocy aksjomatu Pq8

$$\mathrm{ATPQ} \vdash (\neg em(q) \Rightarrow mb(min(q), q))$$

Ponieważ formuła $mb(min(q), q) \Rightarrow (\exists e)\, mb(e, q)$ jest tautologią rachunku kwantyfikatorów, a więc i twierdzeniem logiki algorytmicznej, zatem otrzymaliśmy ostatecznie

$$ATPQ \vdash (\neg em(q) \Rightarrow (\exists e)\, mb(e, q))$$

Ad (3)
Stosując aksjomat logiki Ax8 do aksjomatu kolejek priorytetowych Pq5 otrzymujemy

$$ATPQ \vdash (\neg e \underset{E}{=} e' \wedge mb(e, q)) \Rightarrow mb(e, in(e', q))$$

Z aksjomatu Pq4 mamy

$$ATPQ \vdash (e \underset{E}{=} e' \Rightarrow mb(e, in(e', q))$$

a stąd i z aksjomatu Ax4 otrzymujemy

$$ATPQ \vdash (((\neg e \underset{E}{=} e' \wedge mb(e, q)) \vee e \underset{E}{=} e') \Rightarrow mb(e, in(e', q)))$$

Przez proste przekształcenie poprzednika w tym twierdzeniu dostajemy

$$ATPQ \vdash (e \underset{E}{=} e' \vee mb(e, q)) \Rightarrow mb(e, in(e', q))$$

Z drugiej strony z aksjomatu Pq5 mamy

$$ATPQ \vdash (\neg e \underset{E}{=} e' \Rightarrow (mb(e, in(e', q)) \Rightarrow mb(e, q)))$$

Aksjomaty rachunku zdań pozwalają przepisać tę formułę w postaci

$$ATPQ \vdash mb(e, in(e', q)) \Rightarrow (e \underset{E}{=} e' \vee mb(e, q))$$

co razem z poprzednio udowodnioną implikacją kończy dowód własności (3).

Dowód własności (4) przebiega analogicznie do przedstawionego, z tym, że jest oparty na aksjomatach Pq6 i Pq7.

Ad (5)
Na mocy udowodnionej już własności (3) mamy

$$ATPQ \vdash (em(q) \wedge mb(e, in(e', q))) \Rightarrow (em(q) \wedge e \underset{E}{=} e' \vee em(q) \wedge mb(e, q))$$

Ale z własności (2) wynika, że $ATPQ \vdash (em(q) \Rightarrow \neg mb(e, q))$, a z aksjomatu Ax6 mamy $\vdash ((em(q) \wedge e \underset{E}{=} e') \Rightarrow e \underset{E}{=} e')$. Zatem możemy wywnioskować, że

$$ATPQ \vdash (em(q) \Rightarrow (mb(e, in(e', q)) \Rightarrow e \underset{E}{=} e'))$$

Dowód własności (6) opiera się na własności (4) i jest analogiczny do dowodu (5).

Ad (7)
Z własności (2) mamy

$$ATPQ \vdash (\neg em(q) \Rightarrow mb(min(q), q))$$

Stosując regułę R2 i aksjomat Ax18 mamy

$$ATPQ \vdash (\neg em(in(e, q)) \Rightarrow mb(min(in(e, q)), in(e, q)))$$

Z własności (1)

$$\text{ATPQ} \vdash mb(min(in(e, q), in(e, q)))$$

a stąd i z własności (5)

$$\text{ATPQ} \vdash (em(q) \Rightarrow e \underset{E}{=} min(in(e, q)))$$

Dowód własności (8), podobny do przeprowadzonego, pomijamy. □

Dalej wykażemy, że *eq* ma zwykłe własności relacji równości, tzn. własności antysymetrii, zwrotności, przechodniości i ekstensjonalności, por. p. 4.4.

LEMAT 5.6

W każdym modelu teorii ATPQ są prawdziwe następujące formuły:

(1) $eq(q, q') = (\forall e)(mb(e, q) = mb(e, q'))$
(2) $eq(q, q)$
(3) $(eq(q, q') \Rightarrow eq(q', q))$
(4) $(eq(q, q') \wedge eq(q', q'') \Rightarrow eq(q, q''))$

DOWÓD.

Dowód formuły (1) znajduje się w Dodatku A. Dowód formuł (2), (3), (4) wynika natychmiast z równoważności (1) i praw rachunku kwantyfikatorów. □

Z lematu 5.6 wynika natychmiast, że w każdym modelu **A** teorii ATPQ predykat *eq* musi być interpretowany jako pewna relacja równoważności w zbiorze typu *PQ*. Wykażemy, że relacja $eq_{\mathbf{A}}$ jest kongruencją w **A**.

LEMAT 5.7

Następujące formuły w języku L są prawdziwe w każdym modelu teorii ATPQ:

(1) $(eq(q, q') \Rightarrow min(q) \underset{E}{=} min(q'))$
(2) $(eq(q, q') \Rightarrow em(q) = em(q'))$
(3) $(e \underset{E}{=} e' \wedge eq(q, q')) \Rightarrow eq(in(e, q), in(e', q'))$
(4) $(e \underset{E}{=} e' \wedge eq(q, q')) \Rightarrow eq(del(e, q), del(e', q'))$

DOWÓD

Dowody formuł (1)–(4) są oparte na równoważności

$$eq(q, q') = (\forall e)(mb(e, q) \equiv mb(e, q'))$$

udowodnionej w Dodatku A. Dla przykładu rozważmy formułę (4). Oznaczmy przez α formułę $(mb(e'', q) = mb(e'', q'))$. Ponieważ

$$\vdash (e \underset{E}{=} e' \wedge \alpha) \Rightarrow ((\neg e \underset{E}{=} e'' \wedge mb(e'', q)) \equiv (\neg e' \underset{E}{=} e'' \wedge mb(e'', q')))$$

zatem korzystając z następującej tautologii logiki klasycznej

$$(\forall x)(\gamma(x) \Rightarrow \beta(x)) \Rightarrow ((\forall x)\, \gamma(x) \Rightarrow (\forall x)\, \beta(x))$$

otrzymamy

$$\vdash (e \underset{E}{=} e' \wedge (\forall e'')\, \alpha) \Rightarrow (\forall e'')((\neg e \underset{E}{=} e'' \wedge mb(e'', q)) = (\neg e' \underset{E}{=} e'' \wedge mb(e', q'))$$

Korzystając z własności (4) lematu 5.5 udowodnimy w teorii ATPQ formułę

$$(e \underset{E}{=} e' \wedge eq(q, q')) \Rightarrow (\forall e'')\, (mb(e'', del(e, q)) = mb(e'', del(e' q')))$$

Stosując jeszcze równoważność (1) z lematu 5.6 otrzymujemy ostatecznie

$$\text{ATPQ} \vdash ((e \underset{E}{=} e' \wedge eq(q, q')) \Rightarrow eq(del(e, q), del(e', q'))) \qquad \square$$

Następujące twierdzenie nosi nazwę twierdzenia o reprezentacji teorii kolejek priorytetowych. Twierdzenie to pokazuje, że aksjomaty PQ odzwierciedlają dokładnie wszystkie te własności, które ma model standardowy.

TWIERDZENIE 5.6

Niech **A** będzie dowolnym modelem teorii ATPQ

$$\mathbf{A} = \langle A \cup S, in_{\mathbf{A}}, del_{\mathbf{A}}, min_{\mathbf{A}}, mb_{\mathbf{A}}, em_{\mathbf{A}}, \leqslant, \underset{A}{=}, eq_{\mathbf{A}} \rangle$$

gdzie $\leqslant$ jest relacją liniowego porządku w A — interpretacją predykatu *les*. Niech $\approx$ będzie relacją w $A \cup S$ określoną następująco: dla dowolnych elementów a, a' zbioru $A \cup S$

$$a \approx a' \quad \text{wtw} \quad a, a' \in A \text{ i } a \underset{A}{=} a' \quad \text{lub} \quad a, a' \in S \text{ i } eq_{\mathbf{A}}(a, a')$$

Wówczas $\approx$ jest kongruencją w A, a struktura ilorazowa $\mathbf{A}/\approx$ oznaczona dalej jako **B** (por. p. 2.2),

$$\mathbf{B} = \langle A/_{\approx} \cup S/_{\approx}, in_{\mathbf{B}}, del_{\mathbf{B}}, min_{\mathbf{B}}; mb_{\mathbf{B}}, em_{\mathbf{B}}, \leqslant, \underset{B}{=}, eq_{\mathbf{B}} \rangle$$

jest izomorficzna ze strukturą standardową

$$\langle A/_{\approx} \cup Fin(A/_{\approx}), insert, delete, minimum; member, empty, \leqslant, \underset{A/\approx}{=}, = \rangle$$

DOWÓD

Niech h będzie odwzorowaniem ze zbioru $A/_{\approx} \cup S/_{\approx}$ w zbiór $A/_{\approx} \cup Fin(A/_{\approx})$ określonym następująco:

$h(a) = a$ dla $a \in A/_{\approx}$
$h(s) = \{a \in A/_{\approx} : mb_{\mathbf{B}}(a, s)\}$ dla $s \in S/_{\approx}$

Zauważmy najpierw, że zbiór $h(s)$ jest zbiorem skończonym, dla dowolnego s. Rzeczywiście, na mocy aksjomatu Pq2 istnieje takie $n \in N$, że

$$\mathbf{B}, v \models (q := del(min(q), q))^n em(q)$$

gdzie $v(q) = s$. Z aksjomatu Pq8 dla każdego $a \in A/_{\approx}$ mamy

$$mb_{\mathbf{B}}(a, s) \equiv (\exists i)\left(s := del_{\mathbf{B}}(min_{\mathbf{B}}(s), s)^i (min(s) \underset{B}{=} a)\right)$$

Zatem dla każdego a, jeżeli $mb_{\mathbf{B}}(a, s)$, to $i < n$. Wynika stąd, że tylko skończona ilość elementów a spełnia warunek $mb_{\mathbf{B}}(a, s)$, a co za tym idzie zbiór $h(s)$ jest skończony. W szczególności, jeżeli $n = 0$, to na mocy własności (2) z lematu 5.5, $-mb_{\mathbf{B}}(a, s)$ dla dowolnego a i w konsekwencji $h(s)$ jest, w tym przypadku, zbiorem pustym.

W ten sposób pokazaliśmy, że jeżeli $em_{\mathbf{B}}(s)$, to $h(s) = \emptyset$. Istnienie takiego s wynika, z aksjomatu Pq2. Na mocy aksjomatu Pq9 i definicji relacji eq w strukturze ilorazowej $\mathbf{B}$, jeżeli $em_{\mathbf{B}}(s)$ i $em_{\mathbf{B}}(s')$, to $s = s'$. Zatem istnieje dokładnie jedno takie s, dla którego $em_{\mathbf{B}}(s)$ i dla którego $h(s) = \emptyset$. Oznaczmy to jedyne s przez $\emptyset_{\mathbf{B}}$.

Rozważmy dwa dowolne elementy $s1$ i $s2$ i przypuśćmy, że

$$h(s1) = h(s2) = \{a_1, \ldots, a_n\}$$

Wtedy na mocy definicji funkcji h, $mb_{\mathbf{B}}(a_i, s1) = mb_{\mathbf{B}}(a_i, s2)$ dla $i \leqslant n$ oraz dla dowolnego a różnego od wszystkich a_i, $-mb_{\mathbf{B}}(a, s1)$ i $-mb_{\mathbf{B}}(a, s2)$. Stąd prawdziwe jest zdanie

$$(\forall a)\, mb_{\mathbf{B}}(a, s1) = mb_{\mathbf{B}}(a, s2)$$

Na mocy własności (1) z lematu 5.6 otrzymujemy $s1 = s2$. Wykazaliśmy w ten sposób różnowartościowość funkcji h.

Funkcja h jest odwzorowaniem ną zbiór $A/_{\approx} \cup Fin(A/_{\approx})$. Dla dowodu weźmy dowolny zbiór $\{a_1, \ldots, a_n\} \in Fin(A_{\approx})$ oraz element s taki, że

$$s \overset{df}{=} in_{\mathbf{B}}(a_1, in_{\mathbf{B}}(a_2, \ldots, in_{\mathbf{B}}(a_n, \emptyset_{\mathbf{B}}) \ldots))$$

Korzystając z własności (3) z lematu 5.5 otrzymujemy

$$\begin{aligned} mb_{\mathbf{B}}(a, s) &= \left(a \underset{B}{=} a_1 \vee mb_{\mathbf{B}}(a, in_{\mathbf{B}}(a_2, \ldots, in_{\mathbf{B}}(a_n, \emptyset_{\mathbf{B}}) \ldots))\right) \equiv \ldots \\ &= (a \underset{B}{=} a_1 \vee a \underset{B}{=} a_2 \vee \ldots \vee a \underset{B}{=} a_n) \end{aligned}$$

Stąd $h(s) = \{a_1, \ldots, a_n\}$.

Pozostało wykazać, że funkcja h zachowuje relacje i operacje systemu $\mathbf{B}$. Równoważności

$mb_{\mathbf{B}}(a, s) \equiv member(h(a), h(s))$
$em_{\mathbf{B}}(s) \equiv empty(h(s))$
$eq_{\mathbf{B}}(s, s') = h(s) = h(s')$

wynikają z przeprowadzonych rozważań.

Dla dowodu równości $h(min_{\mathbf{B}}(s)) = minimum(h(s))$ niech $minimum(h(s)) = a_1$ oraz weźmy $h(s) = \{a_1, ..., a_n\}$. Na mocy aksjomatu Pq3, $min_{\mathbf{B}}(s) \leqslant a_i$ i $min_{\mathbf{B}}(s) \in \{a_1, ..., a_n\}$. Zatem $minimum(h(s)) = a_1 - min_{\mathbf{B}}(s) = h(min_{\mathbf{B}}(s))$.

Na mocy lematu 5.5 następujące równoważności są prawdziwe w **B**:

$$mb_{\mathbf{B}}(a', in_{\mathbf{B}}(a, s)) = (a' \underset{B}{=} a \vee mb_{\mathbf{B}}(a', s)) \equiv (a \underset{B}{=} a' \vee a' \in h(s)) = \equiv a' \in \{a\} \cup h(s)$$

Czyli

$$h(in_{\mathbf{B}}(a, s)) = insert(h(a), h(s))$$

Analogicznie, korzystając z własności (4) lematu 5.5 otrzymamy

$$h(del_{\mathbf{B}}(a, s)) = delete(h(a), h(s))$$

Rozważania te dowodzą, że h jest izomorfizmem odwzorującym strukturę **B** na odpowiadającą jej strukturę standardową. □

Znaczenie twierdzenia 5.6 polega na tym, że pozwala ono weryfikować własności różnych implementacji kolejki priorytetowej albo na gruncie teorii ATPQ, albo zamiennie, na gruncie jednego modelu — modelu standardowego.

TWIERDZENIE 5.7

Następujące formuły są twierdzeniami teorii ATPQ:

(1) $mb(e, q) \Rightarrow eq(in(e, del(e, q)), q)$
(2) $\neg mb(e, q) \Rightarrow eq(del(e, in(e, q)), q)$
(3) $eq(in(e, q), in(e, q')) \Rightarrow ((mb(e, q) = mb(e, q')) \Rightarrow eq(q, q'))$
(4) $eq(del(e, q), del(e, q')) \Rightarrow ((mb(e, q) \equiv mb(e, q')) \Rightarrow eq(q, q'))$
(5) $mb(e, q) \Rightarrow eq(q, in(e, q))$
(6) $eq(q, del(e, q)) \Rightarrow \neg mb(e, q)$

SZKIC DOWODU

Korzystając z twierdzenia 5.6, wystarczy sprawdzić prawdziwość formuł (1)–(6) w dowolnym modelu standardowym. □

Drzewa binarne 5.5

W tym punkcie podamy specyfikację struktury skończonych drzew binarnych, w których liściom przyporządkowano pewne elementy zwane atomami. Rozważmy język algorytmiczny dwusortowy z równością. Niech V_A będzie zbiorem zmiennych indywiduowych typu A, a V_T będzie zbiorem zmiennych indywiduowych typu T tego języka. Zakładamy, że język ma następujące funktory i predykaty:

cons	typu	$T \times T \to T$
left	typu	$T \to T$
right	typu	$T \to T$
atom	typu	$T \to B_0$
val	typu	$T \to A$
new	typu	$A \to T$
$=$	typu	$T \times T \cup A \times A \to B_0$

Następujący zbiór formuł AxTr będziemy nazywać zbiorem aksjomatów drzew binarnych:

Tr1 $atom(t) \vee t = cons(left(t), right(t))$

Tr2 $t_1 = left(cons(t_1, t_2))$

Tr3 $t_2 = right(cons(t_1, t_2))$

Tr4 $\neg atom(cons(t_1, t_2)) \wedge atom(new(a))$

Tr5 **while** $\neg atom(t)$
do
if $atom(left(t))$
then
$t := right(t)$
else
$t := cons(left(left(t)), cons(right(left(t)), right(t)))$
fi
od true

Tr6 $(atom(t) = new(val(t)) = t) \wedge val(new(a)) = a$

Tr7 $(new(a) = new(a') \Rightarrow a = a')$

Tr8 $(const(t_1, t_2) = cons(t_3, t_4) \Rightarrow (t_1 = t_3 \wedge t_2 = t_4))$

Tr9 $ok(left(t)) = ok(right(t)) = \neg atom(t)$

Tr10 $ok(val(t)) = atom(t)$

Tr11 $ok(const(t_1, t_2)) \equiv$ **true**

Teorię algorytmiczną, której zbiorem aksjomatów specyficznych jest zbiór AxTr, będziemy nazywać algorytmiczną teorią drzew binarnych ATBT. Standardowy model tej teorii jest oparty na zbiorze tzw. *S*-wyrażeń (*S*-wyrażenia stanowią sematyczną bazę języka programowania Lisp [36]).

DEFINICJA 5.3

Zbiór $S(A)$, S-wyrażeń nad zbiorem A, jest najmniejszym zbiorem takim, że:

(1) dla każdego $a \in A$, wyrażenie (a) należy do $S(A)$,

(2) dla dowolnych dwóch S-wyrażeń τ_1 i τ_2, wyrażenie (τ_1, τ_2) jest również elementem zbioru $S(A)$. ■

Niech $cons_S$, $left_S$, $right_S$, new_S, val_S, $atom_S$, (por. p. 2.2) oznaczają operacje i relacje w zbiorze S-wyrażeń nad A takie, że dla dowolnych $a \in A$ oraz $\tau_1, \tau_2 \in S(A)$

$$atom_S(t) \quad \text{wtw} \quad (\exists a \in A)\, t = (a)$$

$$new_S(a) \stackrel{\text{df}}{=} (a)$$

$$cons_S(\tau_1, \tau_2) \stackrel{\text{df}}{=} (\tau_1, \tau_2)$$

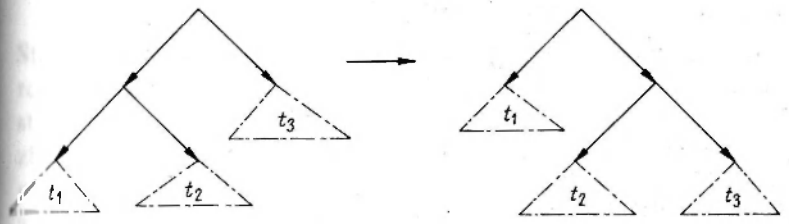
$$left_S(\tau) \stackrel{\text{df}}{=} \begin{cases} \tau_1 & \text{gdy } \tau = (\tau_1, \tau_2) \\ \text{nieokreślone} & \text{w przeciwnym razie} \end{cases}$$

$$right_S(\tau) \stackrel{\text{df}}{=} \begin{cases} \tau_2 & \text{gdy } \tau = (\tau_1, \tau_2) \\ \text{nieokreślone} & \text{w przeciwnym razie} \end{cases}$$

$$val_S(\tau) \stackrel{\text{df}}{=} \begin{cases} a & \text{gdy } \tau = (a) \\ \text{nieokreślone} & \text{w przeciwnym razie} \end{cases}$$

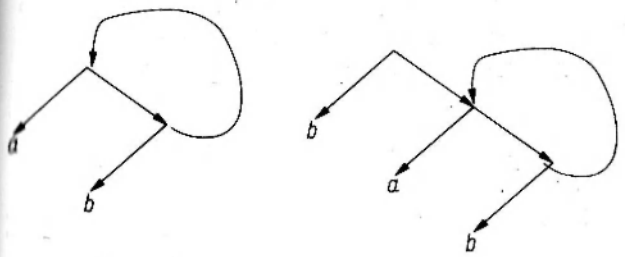
Nietrudno zauważyć, że struktura

$$S(A) = \langle A \cup S(A), cons_S, left_S, right_S, new_S, val_S; atm_S, = \rangle$$



RYS. 5.3

jest modelem aksjomatów Tr1–Tr4 i Tr6–Tr11. Zwróćmy uwagę na aksjomat Tr5 (5). Semantyczną treść formuły Tr5 łatwiej zrozumieć, gdy zauważy się na czym polega efekt wykonania instrukcji iterowanej w programie zawartym w tym aksjomacie (rys. 5.3).



RYS. 5.4

Jest oczywiste, że aksjomat Tr5 odrzuca drzewa nieskończone oraz elementy podobne do tego z rys. 5.4. Program występujący w Tr5 nie zakończy obliczeń dla takich danych początkowych.

Uważny czytelnik z łatwością wykaże teraz następujący fakt.

LEMAT 5.8

Dla dowolnego zbioru A struktura

$$\mathbf{S}(A) = \langle A \cup S(A), cons_{\mathbf{S}}, left_{\mathbf{S}}, right_{\mathbf{S}}, new_{\mathbf{S}}; val_{\mathbf{S}}; atom_{\mathbf{S}}, = \rangle$$

jest modelem algorytmicznej teorii drzew binarnych ATBT. ■

Jak pokazuje następujące twierdzenie o reprezentacji, struktura S-wyrażeń jest charakterystyczna dla klasy modeli teorii ATBT.

TWIERDZENIE 5.8

Każdy model teorii ATBT, właściwy dla identyczności, jest izomorficzny z pewną strukturą S-wyrażeń.

DOWÓD

Niech

$$\mathbf{B} = \langle A \cup B, cons_{\mathbf{B}}, left_{\mathbf{B}}, right_{\mathbf{B}}, new_{\mathbf{B}}, val_{\mathbf{B}}; atom_{\mathbf{B}}, = \rangle$$

będzie dowolnym modelem zbioru AxTr. Zdefiniujmy odwzorowanie h ze zbioru $A \cup S(A)$ w zbiór $A \cup B$ następująco:

$$h(a) \stackrel{\text{df}}{=} a$$
$$h((a)) \stackrel{\text{df}}{=} new_{\mathbf{B}}(a)$$
$$h((\tau_1, \tau_2)) \stackrel{\text{df}}{=} cons_{\mathbf{B}}(h(\tau_1), h(\tau_2))$$

dla dowolnych $a \in A$, $\tau_1, \tau_2 \in S(A)$. Z aksjomatów Tr9–Tr11 wynika, że funkcje $cons_{\mathbf{B}}$, $new_{\mathbf{B}}$ są różnowartościowe. Zatem h jest też funkcją różnowartościową.

Na mocy aksjomatu Tr1 każdy element zbioru B jest albo wynikiem operacji *new* na pewnym elemencie zbioru A, albo wynikiem operacji *cons* na pewnych elementach zbioru B. Stąd i z definicji operacji $new_{\mathbf{S}}$ oraz $cons_{\mathbf{S}}$ w zbiorze S-wyrażeń otrzymujemy, że funkcja h odwzorowuje $A \cup S(A)$ na zbiór $A \cup B$.

Wprost z definicji funkcji h wynika, że zachowuje ona operacje $new_{\mathbf{S}}$ i $cons_{\mathbf{S}}$, tzn.

$$h(cons_{\mathbf{S}}(\tau_1, \tau_2)) = cons_{\mathbf{B}}(h(\tau_1), h(\tau_2))$$
$$h(new_{\mathbf{S}}(a)) = new_{\mathbf{B}}(h(a))$$

Na mocy aksjomatów Tr2, Tr3 i podanych równości, dla $\tau = (\tau_1, \tau_2)$ otrzymujemy

$$h(left_S(\tau)) = h(left_S(cons_S(\tau_1, \tau_2))) = h(\tau_1) = left_B(cons_B(h(\tau_1), h(\tau_2))) = \\ = left_B(h(cons_S(\tau_1, \tau_2))) = left_B(h(\tau))$$

$$h(right_S(\tau)) = h(right_S((\tau_1, \tau_2))) = h(\tau_2) = right_B(cons_B(h(\tau_1), h(\tau_2))) = \\ = right_B(h(cons_S(\tau_1, \tau_2))) = right_B(h(\tau))$$

Na mocy aksjomatów Tr6 dla $\tau = (a)$ mamy

$$h(val_S(\tau)) = h(val_S((a))) = h(a) = a = val_B(new_B(a)) = val_B(h(new_S(a))) = \\ = val_B(h(\tau))$$

$$atom_S(\tau) = new_S(val_S(\tau)) = \tau = h(new_S(val_S(\tau))) = h(\tau) \equiv \\ \equiv new_B(val_B(h(\tau))) = h(\tau) \equiv atom_B(h(\tau))$$

Rozważania te dowodzą, że funkcja h jest izomorfizmem odwzorowującym strukturę $\bar{S}(A)$ na strukturę **B**. □

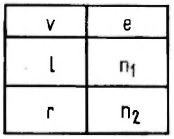
Modele teorii ATBT są strukturami uniwersalnymi, ich znaczenie może być porównywane ze znaczeniem arytmetyki liczb naturalnych, w których można zaimplementować wiele różnych struktur. W następnym punkcie zajmiemy się specyfikacją struktury drzew binarnych poszukiwań bardziej przydatnej do celów implementacji kolejek priorytetowych.

Drzewa binarnych poszukiwań 5.6

Standardowa struktura drzew binarnych poszukiwań była opisana w rozdz. 2. (por. definicję 2.18). Obecnie podamy algorytmiczną specyfikację tej struktury. Podobnie jak w poprzednich rozdziałach, specyfikacja będzie zbiorem aksjomatów w języku algorytmicznym. Przedstawiona tutaj aksjomatyzacja będzie związana ze znaną metodą implementacji drzew BST zrealizowanych za pomocą następującej (lub podobnej) deklaracji.

unit *Node*: **class**(v:E); *variable* l, r: Node **end** Node;

gdzie E oznacza pewien zbiór (tutaj zbiór obiektów typu E) uporządkowany liniowo przez pewną relację $\underset{E}{\leqslant}$. Intuicje programistyczne związane z tą deklaracją każą nam traktować ją jako opis zbioru obiektów, których struktura jest przedstawiona na rys. 5.5.



Rys. 5.5

Dowolny obiekt tej postaci można traktować jako wartościowanie trzech jego atrybutów v, l, r. Ponadto przyjmuje się, że wśród obiektów znajduje się obiekt pusty **none**, dla którego wartości atrybutów v, l, r nie są określone.

Na obiektach różnych od **none** możemy wykonywać operacje odczytania wartości atrybutów v, l, r oraz operacje ul, ur, zmiany wartości atrybutów l i r w obiekcie.

Z przedstawioną deklaracją typu wiąże się następująca struktura danych:

$$\langle E \cup Node,\ v,\ l,\ r,\ new,\ ul,\ ur;\ isnone,\ \underset{E}{\leqslant}, = \rangle$$

gdzie

$new\colon E \to Node$ $\qquad$ $v\colon Node \to E$

$l\colon Node \to Node$ $\qquad$ $r\colon Node \to Node$

$ul\colon Node \times Node \to Node$ $\qquad$ $ur\colon Node \times Node \to Node$

$isnone\colon Node \to B_0$

$=$ oraz $\underset{E}{\leqslant}$ są odpowiednio relacjami binarnymi w $E \cup Node$ i w E.

DEFINICJA 5.4

Struktura danych o podanej wyżej sygnaturze będzie nazywana strukturą drzew binarnych poszukiwań, jeżeli spełnia następujące aksjomaty:

Bst1 $v(new(e)) = e \wedge isnone(l(new(e))) \wedge isnone(r(new(e)))$

Bst2 $(mb(e, l(n)) \Rightarrow e < v(n)) \wedge (mb(e, r(n)) \Rightarrow v(n) < e)$

gdzie predykat $mb(e, n)$ jest zdefiniowany następująco:

```
mb(e, n) ≝ begin
             n1 := n; bool := false;
             while ¬isnone(n1) ∧ ¬bool
             do
               if v(n1) = e
               then
                 bool := true
               else
                 if e < v(n1) then n1 := l(n1) else n1 := r(n1) fi
               fi
             od
           end bool
```

Bst3 $(\neg isnone(n) \wedge \neg isnone(l(n))) \Rightarrow v(l(n)) < v(n)$

Bst4 $(\neg isnone(n) \wedge \neg isnone(r(n))) \Rightarrow v(n) < v(r(n))$

Bst5 $n = n' \equiv ((isnone(n) \wedge isnone(n')) \vee (v(n) = v(n') \wedge l(n) = l(n') \wedge$
$\wedge r(n) = r(n')))$

Bst6 $(isnone(n) \vee \mathbf{K}\,\mathbf{true})$,

gdzie

```
K: begin
     n' := n;
     while ¬ isnone(n')
     do
       if isnone(l(n'))
       then
         n1 := r(n')
       else
         n1 := new(v(l(n')));
         n1 := ul(l(l(n')), n1);
         n2 := new(v(n'));
         n2 := ul(r(l(n')), n2);
         n2 := ur(r(n'), n2);
         n1 := ur(n2, n1)
       fi;
       n' := n1
     od
   end
```

Bst7 $((r(n) = n'' \wedge v(n) = e \wedge \mathbf{K}_l\, bool) \vee isnone(n')) \Rightarrow$
$\Rightarrow (n3 := ul(n', n))(r(n3) = n'' \wedge v(n3) = e \wedge l(n3) = n')$

gdzie

```
K_l: begin
       n2 := n';
       while ¬ isnone(r(n2))
       do
         n2 := r(n2)
       od;
       bool := v(n2) < v(n);
     end
```

Bst8 $((l(n) = n'' \wedge v(n) = e \wedge \mathbf{K}_r\, bool) \vee isnone(n')) \Rightarrow$
$\Rightarrow (n3 := ur(n', n))(l(n3) = n'' \wedge v(n3) = e \wedge r(n3) = n')$

gdzie

```
K_r: begin
       n2 := n';
       while ¬ isnone(l(n2))
       do
```

$n2 := l(n2)$
do;
$bool := v(n) < v(n2)$;
end

Bst9 aksjomaty równości w zbiorze $E \cup Node$ i liniowego porządku $\leqslant_E$ w zbiorze E;

Bst10 grupa czterech aksjomatów określających dziedziny operacji

$\neg isnone(n) \equiv ok(l(n)) = ok(r(n)) \equiv ok(v(n))$
$(\neg isnone(n) \wedge (K_l bool \vee isnone(n')) \equiv ok(ul(n', n))$
$(\neg isnone(n) \wedge (K_r bool \vee isnone(n')) = ok(ur(n', n))$
$ok(new(e)) \equiv \mathbf{true}$ ■

Teorię algorytmiczną, której aksjomatami specyficznymi są formuły Bst1–Bst10*, będziemy nazywać algorytmiczną teorią drzew binarnych poszukiwań w skrócie ATBST.

TWIERDZENIE 5.9

Teoria algorytmiczna ATBST drzew binarnych poszukiwań jest niesprzeczna.

DOWÓD

Niech Et będzie niepustym zbiorem, a $\mathbf{M}$ standardową strukturą drzew binarnych poszukiwań

$$\mathbf{M} = \langle Et \cup BST, val, left, right, new, upl, upr; isnone, \lesssim_{Et}, = \rangle$$

(por. definicję 2.18). Nietrudno sprawdzić, że wszystkie aksjomaty drzew BST są prawdziwe w tej strukturze. Znaczenie aksjomatów w strukturze $\mathbf{M}$ jest następujące:

Bst1. W nowo utworzonym obiekcie typu *Node* atrybuty l i r mają wartość ( ), tak więc każdy obiekt otrzymany jako wartość wyrażenia $new(e)$ reprezentuje liść.

Bst2–Bst4. Dla każdego niepustego drzewa n, każdy element jego lewego poddrzewa jest mniejszy niż wartość zapisana w korzeniu i każdy element prawego poddrzewa n jest większy niż wartość zapisana w korzeniu drzewa.

Bst6. Każdy niepusty element n jest korzeniem skończonego drzewa binarnego (por. p. 5.5).

* Aksjomaty Bst3 i Bst4 można udowodnić na podstawie pozostałych aksjomatów.

Bst7. Jeżeli największy element w drzewie n' jest mniejszy niż wartość zapisana w korzeniu drzewa n lub gdy n' jest postaci ( ), to przypisanie n' jako lewego syna obiektu n jest dobrze określoną operacją, a pozostałe atrybuty obiektu n są nie zmienione.

Bst8. Jeżeli najmniejszy element w drzewie n' jest większy niż wartość etykiety wierzchołka drzewa n lub gdy n' jest postaci ( ), to przypisanie n' jako prawego syna obiektu n jest dobrze określoną operacją, pozostałe atrybuty obiektu n są nie zmienione.

Sens pozostałych aksjomatów jest oczywisty.

Wszystkie wymienione tu własności są prawdziwe w strukturze **M**, zatem struktura standardowa jest modelem algorytmicznej teorii ATBST drzew binarnych poszukiwań. □

LEMAT 5.9

Następujące formuły są twierdzeniami teorii ATBST:

(1) $n = n' \Rightarrow (\forall e)(mb(e, n) = mb(e, n'))$

(2) $\neg isnone(n) \Rightarrow ur(r(n), ul(l(n), new(v(n)))) = n$

(3) $ur(n', ul(n'', n)) = n3 = ul(n'', ur(n', n)) = n3$ ■

Dowód lematu pomijamy.

TWIERDZENIE 5.10 (o reprezentacji)

Każdy model aksjomatów Bst1–Bst10 jest izomorficzny z pewnym modelem standardowym.

DOWÓD

Niech

$$\mathbf{A} = \langle Et \cup A, v_{\mathbf{A}}, l_{\mathbf{A}}, r_{\mathbf{A}}, new_{\mathbf{A}}, ul_{\mathbf{A}}, ur_{\mathbf{A}}; isnone_{\mathbf{A}}, \underset{Et}{\leqslant}, = \rangle$$

będzie dowolnym modelem teorii ATBST oraz niech struktura

$$\mathbf{M} = \langle Et \cup BST, val, left, right, new, upl, upr; isnone, \underset{Et}{\leqslant}, = \rangle$$

będzie modelem standardowym opartym na tym samym zbiorze elementów Et uporządkowanym liniowo przez relację $\underset{Et}{\leqslant}$.

Z aksjomatu Bst1 wynika istnienie w strukturze **A** elementu, dla którego relacja $isnone_{\mathbf{A}}$ jest spełniona. Na mocy Bst5 istnieje dokładnie jeden taki element. Oznaczmy go przez $none_{\mathbf{A}}$. Przyjmijmy następującą definicję odwzorowania h:

$h(e) \stackrel{\text{df}}{=} e$ dla $e \in Et$
$h(none_{\mathbf{A}}) \stackrel{\text{df}}{=} ()$
$h(n) \stackrel{\text{df}}{=} (h(l(n))\, h(v(n))\, h(r(n)))$ dla $n \neq none_{\mathbf{A}}$ i $n \notin Et$

Jeżeli $n \neq none_{\mathbf{A}}$, to na mocy definicji funkcji h, $h(n) \neq ()$. Wynika stąd, że funkcje *val*, *left*, *right* są określone dla $h(n)$. Na mocy aksjomatu Bst10 również $v_{\mathbf{A}}(n)$, $r_{\mathbf{A}}(n)$, $l_{\mathbf{A}}(n)$ mają wartości określone. Z definicji operacji w strukturze standardowej i definicji funkcji h mamy

$val(h(n)) = h(v_{\mathbf{A}}(n))$
$right(h(n)) = h(r_{\mathbf{A}}(n))$
$left(h(n)) = h(l_{\mathbf{A}}(n))$

Na mocy aksjomatu Bst1

$$(h(none_{\mathbf{A}})\, h(e)\, h(none_{\mathbf{A}})) = (h(l_{\mathbf{A}}(new_{\mathbf{A}}(e)))\, h(v_{\mathbf{A}}(new_{\mathbf{A}}(e)))\, h(r_{\mathbf{A}}(new_{\mathbf{A}}(e))))$$

Zatem

$$new(h(e)) = new(e) = ((\,)\, e\, (\,)) = (h(none_{\mathbf{A}})\, h(e)\, h(none_{\mathbf{A}})) = h(new_{\mathbf{A}}(e))$$

Rozważmy operację *ur*. Pokażemy najpierw, że wynik operacji *ur* w strukturze **A** jest określony dla argumentów n', n wtedy i tylko wtedy, gdy wynik operacji *upr* w strukturze **M** jest określony dla argumentów $h(n')$, $h(n)$. Gdyby $ur_{\mathbf{A}}(n', n)$ nie było określone, to na mocy aksjomatu Bst10 $n' \neq none_{\mathbf{A}}$ i formuła $\mathrm{K}_{\mathrm{r}}bool$ nie byłaby spełniona. Aksjomat Bst6 gwarantuje, że wszystkie obliczenia programu K_{r} są skończone, zatem istnieje takie i, że $v(l^i(n')) \leqslant v(n)$. Ponieważ funkcja h jest identycznością na zbiorze Et, mamy również

$$h(v(l^i(n'))) \leqslant h(v(n))$$

Stąd $val(left^i(h(n'))) \leqslant val(h(n))$ co oznacza, że wartość funkcji *upr*, nie jest określona dla argumentów $h(n')$, $h(n)$. Podobne rozumowanie pozwoli pokazać, że jeśli $upr(h(n'), h(n))$ jest nieokreślone, to również $ur_{\mathbf{A}}(n', n)$ nie jest określone.

Przypuśćmy teraz, że wartość $ur_{\mathbf{A}}(n1, n)$ jest dobrze określona w strukturze **A**. Z aksjomatu Bst8 mamy

$l_{\mathbf{A}}(ur_{\mathbf{A}}(n1, n)) = l_{\mathbf{A}}(n)$
$r_{\mathbf{A}}(ur_{\mathbf{A}}(n1, n)) = n1$

Stąd na mocy definicji funkcji h

$$h(ur_{\mathbf{A}}(n1, n)) = (h(l_{\mathbf{A}}(n))\, h(v_{\mathbf{A}}(n))\, h(n1))$$

a na mocy udowodnionych własności

$$h(ur_{\mathbf{A}}(n1, n)) = upr(h(n1), h(n))$$

Przeprowadzenie podobnego rozumowania dla pozostałych relacji i operacji pozostawiamy czytelnikowi. W dalszej części dowodu będziemy posługiwać się funkcją gł określającą głębokość wyrażenia *BST* Przyjmijmy

$$gł(()) \stackrel{df}{=} 0$$
$$gł(\tau e \eta) \stackrel{df}{=} \max(gł(\tau), gł(\eta)) + 1$$

dla dowolnych *BST* wyrażeń τ, η i dla dowolnego $e \in Et$.

Przez indukcję ze względu na głębokość wyrażenia *BST* wykażemy, że h odwzorowuje zbiór A na *BST*. Własność

$$\tau \in BST \text{ i } gł(\tau) = j \text{ implikuje } (\exists n \in A)\, h(n) = \tau \tag{5.1}$$

jest prawdziwa dla $j = 0$. Załóżmy, że własność (5.1) zachodzi dla wszystkich j mniejszych niż pewna ustalona liczba naturalna k.

Rozważmy wyrażenie $BST(\tau e \eta)$ o głębokości k. Oczywiście $gł(\tau)$, $gł(\eta) < k$, zatem z założenia indukcyjnego otrzymujemy

$$(\exists x, y \in A)\, h(x) = \tau \wedge h(y) = \eta$$

Niech $n \stackrel{df}{=} ul_A(x, ur_A(y, new_A(e)))$. Wtedy

$$h(n) = h(ul_A(x, ur_A(y, new_A(e)))) = upl(h(x), upr(h(y), h(new_A(e)))) =$$
$$= upl(\tau, upr(\eta, (()e()))) = (\tau e \eta)$$

Wynika stąd, że własność (5.1) zachodzi dla dowolnego wyrażenia $\tau \in BST$, tzn. h odwzorowuje zbiór A na *BST*.

Podobnie, przez indukcję ze względu na głębokość wyrażenia *BST* wykażemy, że h jest funkcją równowartościową.

Z definicji funkcji h, dla $n1$ takiego, że $gł(h(n1)) = 0$, $h(n1) = h(n2)$ implikuje $n1 = n2$. Załóżmy, że dla wszystkich n', n takich, że $gł(h(n')) < < j, j > 0$,

$$h(n') = h(n) \Rightarrow n' = n$$

Rozważmy $n1$ takie, że $gł(h(n1)) = j$ oraz $h(n1) = h(n2)$. Z definicji funkcji h i z aksjomatu Bst5 mamy

$$h(l_A(n1)) = h(l_A(n2))$$
$$h(r_A(n1)) = h(r_A(n2))$$
$$h(v_A(n1)) = h(v_A(n2))$$

Ponieważ głębokość wyrażeń $h(l_A(n1))$, $h(l_A(n2))$, $h(r_A(n1))$, $h(r_A(n2))$, $h(v_A(n1))$, $h(v_A(n2))$ jest mniejsza niż j, zatem z założenia indukcyjnego otrzymujemy

$$l_A(n1) = l_A(n2),\ r_A(n1) = r_A(n2),\ v_A(n1) = v_A(n2)$$

Stąd na mocy aksjomatu Bst5 otrzymujemy $n1 = n2$. Na mocy zasady indukcji, dla dowolnych $n1$, $n2$

$h(n1) = h(n2) \Rightarrow n1 = n2$

tzn. h jest funkcją różnowartościową. Co kończy dowód twierdzenia o reprezentacji. □

Interpretacja 5.7

Punkt obecny i następny są poświęcone problemowi implementacji struktury kolejek priorytetowych w strukturze drzew binarnych poszukiwań.

Problem, czy można zaimplementować pewną strukturę **A** w strukturze **B** występuje w informatyce wielokrotnie. Rozwiązaniem jest pewien moduł implementacyjny składający się z definicji typów, klas i procedur (funkcji). Rozwiązaniem poprawnym jest moduł, o którym można udowodnić, że poprawnie implementuje strukturę **A**, tzn. jest zgodny z zadaną specyfikacją struktury **A**. Rozwiązaniem efektywnym jest moduł, o którym można powiedzieć, że koszt wykonania procedur implementujących nie jest nadmierny lub jest tak niski jak to możliwe.

Formalnym odpowiednikiem zadania implementacji struktury **A** w strukturze **B** jest zadanie interpretacji teorii T(**A**) struktury **A** w teorii T(**B**) struktury **B**.

Niech Ax**A** będzie specyfikacją struktury danych **A** (lub klasy struktur **K**), a Ax**B** niech będzie specyfikacją struktury danych **B** (lub klasy struktur **K**′). Rozpatrzmy teorie algorytmiczne $T1 = \langle L1, \vdash, Ax\mathbf{A}\rangle$ i $T2 = \langle L2, \vdash, Ax\mathbf{B}\rangle$ o aksjomatach specyficznych Ax**A** i Ax**B**, odpowiednio. Nie zakładamy równości języków L1 i L2 tych teorii. Jednak zakładamy zgodność typów funktorów i predykatów występujących w obu językach równocześnie.

DEFINICJA 5.5

Będziemy mówić, że *teoria* T1 jest *interpretowalna* w teorii T2 wtedy i tylko wtedy, gdy istnieje zbiór DF formuł taki, że

(1) dla każdego funktora φ z języka L1, który nie występuje równocześnie w języku L2, istnieje term $\tau_\varphi \in L2$ taki, że formuła

$$(\forall x_1, \ldots, x_n)\, \varphi(x_1, \ldots, x_n) = \tau_\varphi(x_1, \ldots, x_n)$$

należy do zbioru DF;

(2) dla każdego predykatu ϱ z języka L1, nie występującego równocześnie w L2, istnieje formuła $\alpha_\varrho \in L2$ taka, że formuła

$$(\forall x_1, \ldots, x_m)\, \varrho(x_1, \ldots, x_m) = \alpha_\varrho(x_1, \ldots, x_m)$$

należy do zbioru DF;

(3) $Ax\mathbf{B} \cup DF \vdash Ax\mathbf{A}$

(tzn. aksjomaty specyficzne teorii T1 można udowodnić w teorii $\langle L1 \cup L2, \mathbf{C}', Ax\mathbf{B} \cup DF\rangle$, która jest rozszerzeniem teorii T2 o nowe aksjomaty specyficzne DF). ■

Powiemy, że teoria T1 jest algorytmicznie interpretowalna w teorii T2 wtedy i tylko wtedy, gdy termy i formuły, występujące w definicji 5.5, są odpowiednio wyrażeniami algorytmicznymi postaci $K\tau$ i $M\alpha$, gdzie τ jest termem klasycznym, α formułą otwartą, a K i M są programami.

Zwróćmy uwagę, że jeżeli pewna struktura danych **C** jest modelem zbioru aksjomatów Ax**B**, to istnieje struktura **C**′, będąca modelem zbioru $Ax\mathbf{B} \cup DF$. Rzeczywiście, wystarczy jedynie rozszerzyć sygnaturę struktury **C**′ o funkcje $\varphi_{\mathbf{C}'}$, i predykaty $\varrho_{\mathbf{C}'}$, oraz przyjąć ich znaczenie następująco:

$$(\forall c_1, \ldots, c_m)\, \varrho_{\mathbf{C}'}(c_1, \ldots, c_m) \stackrel{\text{df}}{=} \alpha_{\varrho\mathbf{C}}(v)$$
$$(\forall c_1, \ldots, c_n)\, \varphi_{\mathbf{C}'}(c_1, \ldots, c_n) \stackrel{\text{df}}{=} \tau_{\varphi\mathbf{C}}(v)$$

Wynika stąd następujący fakt.

LEMAT 5.10

Zbiór $Ax\mathbf{B} \cup DF$ ma model wtedy i tylko wtedy, gdy Ax**B** ma model. ■

LEMAT 5.11

Jeżeli teoria T1 jest interpretowalna w niesprzecznej teorii T2, to jest niesprzeczna.

Dowód wynika natychmiast z poprzedniego lematu i pojęcia interpretowalności. ■

Wracając do naszego przypadku, implementacji struktury kolejek priorytetowych w strukturze drzew binarnych poszukiwań, chcemy zbudować moduł implementacyjny wraz z gwarancją jakości, z dowodem jego poprawności.

Wprowadzimy kolejno definicje operacji i relacji występujących w strukturze kolejek priorytetowych i wykażemy, że korzystając z tych definicji, można udowodnić własności wymienione w specyfikacji kolejek priorytetowych.

Dokładniej, udowodnimy, że aksjomaty teorii kolejek priorytetowych są twierdzeniami teorii, którą uzyskujemy przez dodanie do aksjomatów teorii drzew binarnych poszukiwań, dodatkowych aksjomatów definiujących operacje struktury kolejek priorytetowych.

Dla uproszczenia przedstawionych dalej definicji i zgodnie z przyzwyczajeniami programistów będziemy stosowali następującą notację:

$l(n)$	$n.l$
$r(n)$	$n.r$
$v(n)$	$n.v$
$n := ul(n', n)$	$n.l := n'$
$n := ur(n', n)$	$n.r := n'$

Ponadto wykorzystamy oznaczenie **zerwij** dla programu nigdzie nieokreślonego (por. p. 8.2).

DEFINICJA 5.6

$$min(n) \stackrel{\text{df}}{=} \textbf{(if } isnone(n) \textbf{ then zerwij else } n1 := n \textbf{ fi}$$
$$\textbf{(while } \neg isnone(n1.l) \textbf{ do } n1 := n1.l \textbf{ od } n1.v))$$

LEMAT 5.12

W każdym modelu teorii ATBST i dla każdego n takiego, że $\neg isnone(n)$, wartość $min(n)$ jest określona.

Dla dowodu wystarczy zauważyć, że w dowolnej strukturze standardowej dla ATBST, każde obliczenie instrukcji **while** $\neg isnone(n)$ **do** $n1 := n1.l$ **od** jest skończone. Teza lematu wynika z twierdzenia o reprezentacji drzew binarnych poszukiwań (por. twierdzenie 5.10). ■

DEFINICJA 5.7

```
member(e, n) ≡ begin n1 := n; result := false;
                     while ¬result ∧ ¬isnone(n1)
                     do
                       if e = n1.v
                       then result := true
                       else
                         if e < n1.v then n1 := n1.l else n1 := n1.r fi
                       fi
                     od
               end result
```
■

LEMAT 5.13

Program występujący w definicji 5.7 zatrzymuje się dla każdych danych początkowych, w każdej strukturze drzew binarnych poszukiwań. ■

LEMAT 5.14

W dowolnym modelu teorii ATBST, jeżeli wartość $min(n)$ jest określona, to jest spełniona formuła:

$$(\forall e)(member(e, n) \Rightarrow min(n) \leqslant e)$$ ■

DEFINICJA 5.8

```
insert(e, n) ≝ begin n1 := n; bool := false; n3 := n1;
                 while (¬isnone(n1) ∧ ¬bool)
                 do
                   n2 := n1;
                   if e = n1.v
                   then
                     bool := true
                   else
                     if e < n1.v
                     then
                       n1 := n1.l
                     else
                       n1 := n1.r
                     fi
                   fi
                 od;
                 if ¬bool
                 then
                   if isnone(n2)
                   then
                     n3 := new(e)
                   else
                     if e < n2.v
                     then
                       n2.l := new(e)
                     else
                       n2.r := new(e)
                     fi
                   fi
                 fi
               end n3
```
■

LEMAT 5.15

Niech M oznacza program występujący w definicji 5.8. Dla każdego $e \in E$, dla każdego n mamy

(1) M *member*(e, n),
(2) dla każdego $e' \neq e$ *member*$(e', n) \equiv$ M *member*$(e', n3)$ ■

Następną definicję, ze względu na jej długość podajemy w zapisie nieformalnym. Procedurę usuwania elementu czytelnik znajdzie w p. 5.8.

DEFINICJA 5.9

delete$(e, n) \stackrel{\text{df}}{=}$ **begin**
{szukaj e}
{przypuśćmy, że e znaleziono w wierzchołku $n1$ a $n2$ jest ojcem $n1$}
{jeżeli $n1$ jest liściem — usuń $n1$}
{jeżeli $n1$ ma tylko jednego syna — uczyń $n2$ ojcem tego syna}
{jeżeli $n1$ ma dwu synów — znajdź najmniejszy element *min*$(n1.r)$ w prawym poddrzewie drzewa $n1$. Usuń ten element z poddrzewa $n1.r$ i umieść go w korzeniu poddrzewa oznaczonego $n1$}
$n3 :=$ drzewo skonstruowane powyżej
end $n3$ ■

LEMAT 5.16

Niech K oznacza program naszkicowany w definicji 5.9. Dla każdego $e \in E$, dla każdego $n \in N$
(1) K $\neg$*member*$(e, n3)$,
(2) dla każdego $e' \neq e$, *member*$(e', n) =$ K *member*$(e', n3)$. ■

Łącząc obserwacje zgromadzone w powyższych lematach otrzymujemy następujące twierdzenie:

TWIERDZENIE 5.11

Wszystkie aksjomaty teorii kolejek priorytetowych są wyprowadzalne (są twierdzeniami) w teorii drzew binarnych poszukiwań, uzupełnionej aksjomatami definiującymi operacje *insert*, *delete*, *min*, *member*, *empty*. ■

Wynika stąd (na mocy twierdzenia o pełności), że jeżeli jest dany pewien model dla teorii drzew binarnych poszukiwań, to można za jego pomocą zbudować model dla algorytmicznej teorii kolejek priorytetowych. W tym przypadku konstrukcja jest prosta, należy określić wartości operacji *insert* (i innych) zgodnie z przyjętą definicją, i tak:

Dla każdego zestawu (e, n) kładziemy $insert_{\mathrm{M}}(e, n)$ jako równe wartości wyrażenia występującego po prawej stronie definicji 5.8 obliczonej dla

danych e, n. Podobnie postępujemy z innymi funktorami zdefiniowanymi wcześniej. Jest oczywiste, że struktura drzew binarnych poszukiwań spełnia formuły — definicje nowych funktorów. A więc otrzymaliśmy model dla teorii drzew binarnych poszukiwań, który równocześnie jest modelem dla aksjomatów definiujących nowe funktory. Zgodnie z twierdzeniem sformułowanym powyżej, struktura ta jest również modelem dla teorii kolejek priorytetowych. Spełnia ona specyfikację struktury kolejek priorytetowych. Zauważmy, że definicje przez nas użyte są algorytmiczne. Co więcej są to procedury obliczania wartości funkcji *insert*, *delete*, *min* i relacji *member*. Pozwala to nam wysłowić tezę o poprawności modułu implementującego, który prezentujemy w następnym punkcie.

Implementacja 5.8

Obecnie przedstawimy moduł implementujący strukturę kolejek priorytetowych. Moduł ten jest zapisany w języku programowania Loglan 82 [9]. Wyniki poprzedniego punktu, twierdzenie 5.11 o interpretowalności teorii kolejek priorytetowych w teorii drzew binarnych poszukiwań, posłużą do prawie mechanicznego utworzenia modułu implementującego, w tym przypadku — klasy KolPrio. Z dyskusji przeprowadzonej w poprzednim punkcie wynika, że jest to implementacja poprawna, tzn. spełniająca wszystkie warunki wymienione w specyfikacji kolejek priorytetowych (por. aksjomaty teorii kolejek priorytetowych, p. 5.4)

```
unit KolPrio: class (type El; function less(e, e': El): Boolean);

  unit node: class (v: El);
    var l, r: node;
  end node;

  unit min: function (n:node): El;
  begin
      while n.l ≠ none do n := n.l od;
      result := n.v
  end min;

  unit member: function (e: El, n:node): Boolean;
    var n1: node, bool1:Boolean;
  begin
    n1 := n; bool1 := false;
    while n1 ≠ none ∧ ¬bool1
    do
      if n1.v = e
      then
```

```
        bool1 := true
      else
        if less(e, n1.v)
        then
          n1 := n1.l
        else
          n1 := n1.r
        fi
      fi
    od;
    result := bool1
end member;

unit empty: function (n: node): Boolean;
begin
    if n = none
    then
      result := true
    else
      result := false
    fi
end empty;

unit insert: function (e: El, n: node): node;
    var n1, n2, n3: node, bool1: Boolean;
begin
    n1 := n; n3 := n; bool1 := false;
    while ¬n1 = none ∧ ¬bool1
    do
      n2 := n1;
      if e = n1.v
      then
        bool1 := true {znaleziono e}
      else {szukaj dalej}
        if less(e, n1.v)
        then {w lewo}
          n1 := n1.l
        else {w prawo}
          n1 := n1.r
        fi
      fi
    od;
    if ¬bool1 {nie znaleziono e, trzeba wstawić}
    then
```

```
      if n3 = none {gdy n jest drzewem pustym}
      then
        n3 := new node(e)
      else {wstawiamy e jako odpowiedniego syna n2}
        if less(e, n2.v)
        then
          n2.l := new node(e)
        else
          n2.r := new node(e)
        fi
      fi
    fi;
    result := n3
end insert;

unit delete: function (e: El, n: node): node;
    var n1, n2, n3, n4, n5: node,
          bool1, leftson: Boolean;
begin
    n1 := n; n3 := n; bool1 := false;
    while ¬n1 = none ∧ ¬bool1
    do
      n2 := n1;
      if e = n1.v
      then
        bool1 := true
      else
        if less(e, n1.v)
        then
          n1 := n1.l
        else
          n1 := n1.r
        fi
      fi
    od;
    if bool1
    then {e znaleziono w n1, n2 jest ojcem n1}
      if less(e, n2.v)
      then
        leftson := true
      else
        leftson := false
      fi {leftson ≡ n1 jest lewym synem n2};
```

```
        if n1.l = none ∧ n1.r = none
        then {n1 jest liściem}
          if leftson then n2.l := none else n2.r := none fi;
        else {n1 nie jest liściem}
          if n1.l = none
          then {n1 nie ma lewego syna}
            if n1 = n
            then
              n3 := n1.r
            else
              if leftson
              then
                n2.l := n1.r
              else
                n2.r := n1.r
              fi {leftson?}
            fi {n = n1?}
          else {n1 ma lewego syna}
            if n1.r = none
            then {n1 nie ma prawego syna}
              if n1 = n
              then {znaleziono w korzeniu}
                n3 := n1.l
              else
                if leftson then n2.l := n1.l else n2.r := n1.l fi
              fi
            else {n1 ma dwóch synów}
              n4 := n1.r
              while n4.l ≠ none
              do
                n5 := n4;
                n4 := n4.l
              od;
              n5.l := n4.r; n1.v := n4.v
            fi {n1.r = none?}
          fi {n1.l = none?}
        fi {n1.l = none ∧ n1.r = none?}
      fi {bool1?}
      result := n3
    end delete
  end KolPrio
```

Moduł implementujący KolPrio może być stosowany wielokrotnie dla różnych programów abstrakcyjnych. Należy po prostu poprzedzić taki program abstrakcyjny następującym prefiksem:

```
pref KolPrio (aktualny typ El, aktualna procedura porównywania)
     block
          ...{program abstrakcyjny, por. p. 5.1}
     end  {blok prefiksowany [9]}.
```

Dopisanie jednej tylko linii ustalającej nazwę modułu implementującego i parametrów aktualnych tego modułu powoduje, że operacje struktury kolejek priorytetowych występujące w programie abstrakcyjnym są realizowane zgodnie ze znaczeniem nadanym im przez moduł implementujący KolPrio.

Moduł symulacyjny 5.9

W tym punkcie zastanowimy się nad zadaniami i strukturą uniwersalnego modułu symulacji procesów. Zadanie modułu polega na dostarczeniu użytkownikowi pojęcia procesu (symulowanego) i odpowiednich operacji na procesach. Zakłada się, że równocześnie może istnieć wiele procesów. Całość obliczenia symulacyjnego oglądana z zewnątrz przypomina grę, w której nasze-procesy po kolei odgrywają swoje role, ustępują miejsca procesom bardziej aktualnym i oczekują właściwego momentu na wznowienie swoich czynności. Są więc potrzebne dwa mechanizmy:

(1) mechanizm umożliwiający przełączanie pomiędzy różnymi procesami symulującymi;

(2) mechanizm ustalający kolejność, w jakiej procesy dochodzą do głosu, tzn. stają się aktywne.

Najlepszym rozwiązaniem pierwszego zadania jest mechanizm współprogramów. O współprogramach piszemy obszerniej w Dodatku C. Współprogramy (ang. coroutines) są narzędziem programowania pozwalającym na „rozpisywanie obliczenia na głosy". Można dzięki niemu utworzyć pewną liczbę obiektów i traktować je jako procesy pracujące w trybie quasi-współbieżnym.

Poniżej podajemy specyfikację systemu zarządzania symulacją procesów. Specyfikacja ta nie jest całkowicie sformalizowana. Sądzimy, że czytelnik zechce to zaakceptować i sam potrafi, w razie potrzeby, sformułować własności operacji rządzących symulacją procesów w pełniejszy sposób. Naszym zdaniem tak właśnie powinno się postępować w praktyce produkcji oprogramowania, nie jest bowiem realistyczne wymaganie, by specyfikacja była zawsze konstruowana jako twór zupełnie formalny. Formalizacja jest

i powinna być środkiem, do którego uciekamy się w ostateczności wówczas, gdy nie umiemy naszych wątpliwości rozstrzygnąć inaczej.

Przez system zarządzania symulacją procesów będziemy rozumieć następujący system algebraiczny:

$$\langle SP \cup SQS \cup EVN \cup T,\ current,\ time,\ schedule,\ run,\ hold,\ passivate,\ cancel;\ idle?,\ terminated? \rangle$$

gdzie SP oznacza zbiór procesów symulowanych, SQS jest oznaczeniem osi czasu, EVN jest zbiorem notatek o zdarzeniach, T jest jakąś strukturą danych reprezentującą czas. Typy operacji są określone następująco:

$current$: $SQS \to SP$
$time$: $SQS \to T$
$schedule$: $SP \times T \to SQS$
$hold$: $T \to SQS$
$idle?$: $SP \to B_0$
$terminated?$: $SP \to B_0$
run: $SP \to SQS$
$passivate$: $SP \to SP$
$cancel$: $SP \to SP$

Żądamy przy tym spełnienia następujących postulatów:

S1 SQS jest kolejką priorytetową $\langle EVN \cup SQS,\ ins,\ del,\ min;\ mb,\ \leqslant,\ = \rangle$

S2 $EVN = SP \times T$

S3 $time(sqs) = min(sqs).t$
$current(sqs) = min(sqs).sp$

S4 $(\forall sqs)\ \neg idle(p) = (\exists t)\, mb((p, t), sqs)$

S5 $mb((p, t), sqs) \wedge mb((p, t'), sqs) \Rightarrow t = t'$

S6 $(sqs = o \wedge idle(p) \wedge \neg terminated(p)) \Rightarrow$
$\Rightarrow (\textbf{call}\ sqs.schedule(p, t))(\neg idle(p) \wedge sqs = ins((p, t), o))$

S7 $(sqs = o \wedge \neg idle(p) \wedge \neg terminated(p)) \Rightarrow$
$\Rightarrow (call\ sqs.\ schedule(p, t))(\neg idle(p) \wedge sqs =$
$= ins((p, t), del((p, time), o))$

S8 $terminated(p) \Rightarrow (call\ sqs.\ schedule(p, t))\{\text{ERROR}\}$

S9 $(\textbf{call}\ sqs.\ hold(t))\alpha \equiv (\textbf{call}\ sqs.\ schedule(current, time + t))\alpha$
dla każdej formuły α

S10 $(current = p' \wedge \neg terminated(p)) \Rightarrow ((\textbf{call}\ sqs.\ run(p))\alpha =$
$= (\textbf{call}\ sqs.\ schedule(p, time))\alpha)$ dla każdej formuły α

S11 $(sqs = o \wedge mb((p, t'), sqs)) \Rightarrow (\textbf{call}\ sqs.\ run(p))(current = p \wedge sqs =$
$= ins((p, time), del((p, t'), o))$

S12 $(p = current \wedge sqs = o) \Rightarrow (\textbf{call}\ p.\ passivate)(idle(p) \wedge sqs =$
$= del((p, time), o))$

S13 $sqs = o \Rightarrow (\mathbf{cali}\ sqs.cancel(p))(idle(p) \wedge sqs = del((p, t), o) \wedge terminated(p)))$

Podstawową własnością systemu symulacji jest następująca równoważność: proces symulowany p jest aktywnym obiektem współprogramu $\equiv$ zawiadomienie o zdarzeniu (p, t) jest najwcześniejszym elementem w kolejce priorytetowej *sqs*. (Jak widać z aksjomatu S5, jest co najwyżej jedno takie t, że para (p, t) należy do *sqs*.) Własność tę zapewniono przez połączenie struktury współprogramów ze strukturą kolejki priorytetowej.

Klasa Simulation implementująca strukturę procesów symulacyjnych jest zbyt obszerna, by ją tu reprodukować.

Synteza 5.10

Spróbujmy teraz zgromadzić razem moduły opisane w poprzednich punktach tego rozdziału i przedstawić strukturę całego programu. Po pierwsze, zakładamy, że czytelnik potrafi sam skonstruować klasę Kolejki i poprzedzić moduł KolPrio prefiksem Kolejki. Przypuśćmy, że wiemy jak wyglądają wnętrza modułów Symulacja, Biuro, Bank, które poniżej są tylko naszkicowane. Wtedy struktura programu symulacji pracy oddziału bankowego będzie następująca:

```
program BANKSYM;
  unit KOLEJKI: class;
    unit element; class ... end element;
    unit kolejka; class ... end kolejka;
    unit wstaw: function (e: element, k:kolejka): kolejka;
      ...
    end wstaw;
    unit usuń: function(k:kolejka): kolejka; ... end usuń;
    unit pierwszy (k:kolejka): function element; ... end pierwszy;
    unit pusta (k:kolejka): function Boolean; ... end pusta;
  end KOLEJKI;
  unit KolPrio: KOLEJKI class;
    {tu należy wpisać treść klasy z p. 5.8}
    ...
  end KolPrio;
  unit SYMULACJA: KolPrio class;
    unit simproces: class ... end simproces;
    ...
  end SYMULACJA;
  unit BIURO: SYMULACJA class;
    {tę klasę można rozwinąć w konkretne biuro, np. bank}
    unit Klient: simproces class; ... end Klient;
```

```
        unit Obsługa: simproces class; ... end Obsługa;
        ...
    end BIURO;
    unit BANK: BIURO class;
        unit KlientBanku: Klient class; ... end KlientBanku
        ...
    end BANK;
begin {tu rozpoczynają się instrukcje}
    pref BANK block
        {tu rozpoczyna się program symulacji oddziału bankowego}
        var k1, k2: KlientBanku, u1: UrzędnikBanku, o:Okienko;
    begin
        WchodziKlient(k1);
        StajePrzedOkienkiem(o);
        ...
    end BANK_block
end programu BANKSYM.
```

Przypatrzmy się strukturze tego programu. Składa się on z sześciu modułów, tak jak to zaplanowaliśmy na początku. Pięć modułów to deklaracje klas, deklaracje te są powiązane pomiędzy sobą relacją „być rozszerzeniem klasy...". Mówimy też, że jedna klasa jest prefiksowana nazwą innej klasy [9]. Szósty moduł, to jedyna instrukcja (złożona) programu BANKSYM, jest to instrukcja bloku (a więc może ona być dowolnie dużym programem). Blok ten, rozważany osobno, jest programem abstrakcyjnym, ponieważ występują w nim pojęcia spoza pierwotnych struktur danych języka Loglan. A jednak program ten może być wykonywany. Poprzedzając go dwoma słowami **pref**BANK wskazaliśmy na jego środowisko, w którym można nadać znaczenie wszystkim terminom użytym w tym bloku. Modułowi BANK nie można nadać sensownego znaczenia bez modułu BIURO itd.

Zauważmy, że moduł BIURO (wraz z prefiksującym go łańcuchem modułów) może być użyty wielokrotnie do wielu innych zadań. Można definiować różne biura i wykonywać odpowiednie programy w towarzystwie modułów opisujących te biura.

Z kolei, jeśli zatrzymamy naszą uwagę na module SYMULACJA, to stwierdzamy z łatwością, że moduł ten jest właściwie definicją i implementacją języka programowania przeznaczonego do symulowania różnych procesów. Oferuje on ogólne pojęcie procesu symulowanego (simproces), któremu to pojęciu użytkownik dodaje nowych cech, rozwijając definicje potrzebnych mu procesów. Moduły KolPrio i KOLEJKI znajdują jeszcze szersze pole zastosowań.

Można stąd wnioskować, że został spełniony postulat wielokrotnej stosowalności (ang. reusability) modułów. Jest to ważne wymaganie przemysłu oprogramowania.

Rozważania tego rozdziału dowodzą, że logika algorytmiczna może stanowić dogodną bazę teoretyczną do formułowania wymagań stawianych oprogramowaniu. Zaoferowane przez nas narzędzia logiki są wystarczające, by uzyskać precyzyjne specyfikacje (por. twierdzenie 5.2 o kategoryczności arytmetyki algorytmicznej liczb naturalnych), a z drugiej strony są elastyczne (por. twierdzenie o reprezentacji dla kolejek priorytetowych (5.6), drzew binarnych (5.8), kolejek (5.4).

Algorytmiczne aksjomaty struktur danych z jednej strony, umożliwiają charakteryzację właściwych struktur danych, z drugiej znacznie ułatwiają zadanie dowodzenia poprawności programu.

Na koniec chcemy wspomnieć o jeszcze jednej korzyści z takiego podejścia: badanie interpretowalności jednej teorii algorytmicznej w drugiej może doprowadzić do wytworzenia modułów implementujących struktury danych (por. 5.8) w sposób gwarantujący ich poprawność.

Spróbujmy podsumować. Nasza propozycja sprowadza się do rozwijania algorytmicznych teorii struktur danych i polega na

(1) potraktowaniu struktury danych jako wielosortowego systemu algebraicznego;

(2) specyfikacji struktur przez aksjomaty algorytmiczne;

(3) analizie (weryfikacji) semantycznych własności programów na gruncie logiki algorytmicznej;

(4) zastosowaniu teorii interpretacji jako formalnego narzędzia pozwalającego badać implementowalność jednej struktury w innych;

(5) wykorzystaniu możliwości formalnego wyprowadzania (konstruowania) modułów implementujących wraz z argumentami o poprawności tak otrzymanej implementacji.

W trakcie badań metamatematycznych okazało się, że wiele interesujących i ważnych struktur nie może być zaksjomatyzowanych (wyspecyfikowanych) w logice pierwszego rzędu. Wprowadzono więc inne, mocniejsze systemy, np. logiki infinistyczne, dopuszczające nieskończone alternatywy i koniunkcje. Okazuje się, że wszystkie znane przykłady struktur, które mają specyfikację w języku z nieskończonymi alternatywami, mają również specyfikację w języku logiki algorytmicznej. A więc nieskończone formuły nie są konieczne.

Wiadomo, że zbiory formuł pierwszego rzędu nie mają dostatecznie dużej siły wyrażania własności struktur algebraicznych. Pomimo to w literaturze rozważa się specyfikacje zapisywane w języku równości, równości warunkowych (formuły Horna) lub w pełnym języku pierwszego rzędu z kwantyfikatorami i negacją. Co z tego ma wyniknąć? Oczywiście klasy modeli dla takich układów aksjomatów są zbyt wielkie. Zawierają struktury nie zamierzone. Przyjmuje się wtedy dodatkowy mechanizm wyróżniający modele zamierzone. Zadanie aksjomatycznego opisu struktury danych zostaje w ten sposób chlubnie rozwiązane, to prawda. Tylko co z tego wynika

dla naszego programisty. Jemu jest potrzebna taka specyfikacja abstrakcyjnej struktury danych, która, po pierwsze, obejmuje dokładnie zamierzoną klasę modeli, po drugie, może być użyta do weryfikacji poprawności przedstawionych realizacji, po trzecie, może być użyta w procesie analizy poprawności programów abstrakcyjnych. Przy tym powinno się osiągnąć wszystkie trzy cele łącznie. Aksjomaty równościowe są sformułowane w ubogim podzbiorze formuł pierwszego rzędu i same nie mogą stanowić opisu nawet klasy struktur będących arytmetykami komputerów. Nie udaje się z nich wyprowadzić niektórych semantycznych własności programów.

Prezentowany przez nas sposób myślenia o specyfikacji struktur danych pozwala połączyć w jedną całość kilka spraw, dla których powstały oddzielne i konkurujące ze sobą teorie [2, 3, 4, 13, 17, 19, 24, 27, 43, 48]. Aksjomaty algorytmiczne umożliwiają dokładne określenie klas struktur danych, logika algorytmiczna dostarcza narzędzi do analizowania własności programów i struktur, interpretacja teorii algorytmicznej w innej teorii pozwala wykazać poprawność implementacji. Z kolei, narzędzia programistyczne Loglanu umożliwiają sprawną realizację modułów implementujących i algorytmów m.in. dzięki mechanizmom klas i obiektów, współprogramów, składaniu deklaracji klas (prefiksowaniu), obsłudze sytuacji wyjątkowych i sygnałów.

O procedurach 6

Wprowadzenie 6.1

W tym rozdziale prezentujemy dwa różne podejścia do problemu definiowania semantyki procedur. W pierwszej części rozważymy tzw. procedury niefunkcyjne. Procedura jest traktowana jako nowa konstrukcja programowcza, która dość istotnie wzbogaca język programów iteracyjnych. Semantyka klasy programów z procedurami niefunkcyjnymi jest rozszerzeniem semantyki operacyjnej programów iteracyjnych przedstawionej w rozdz. 3. Przedstawiono system formalny, który w sposób jednoznaczny wyznacza wspomnianą semantykę operacyjną. System ten może więc być traktowany jako aksjomatyczna definicja semantyki programów z procedurami.

Część druga rozdziału dotyczy procedur funkcyjnych. Sposób ich przedstawienia jest nieco bardziej abstrakcyjny. Procedurę traktujemy jako pewną uwikłaną definicję funkcji czy relacji. System procedur jest, w tym przypadku, układem aksjomatów specyficznych pewnej teorii algorytmicznej. Semantykę języka algorytmicznego przedstawiono tu w nieco inny niż do tej pory sposób. Zaproponowano, tzw. pojęcie obliczenia formalnego, które jest niczym innym jak dowodem formalnym, że rozważane wyrażenie przyjmuje pewną wartość.

Bloki 6.2

Niech L będzie językiem pierwszego rzędu (por. p. 2.4), a w π klasą deterministycznych while-programów nad L (por. p. 3.2).

DEFINICJA 6.1

Klasą programów z blokami nad L nazywamy najmniejszy zbiór π_1 zawierający π i zamknięty ze względu na reguły składania programów, rozgałęzienia, iteracji oraz ze względu na następującą regułę bloku: jeżeli $\mathbf{x}$ jest ciągiem zmiennych i $M \in \pi_1$, to wyrażenie (tzw. instrukcja bloku) postaci

block var x; begin M **endbl**

jest elementem zbioru π_1. ■

Część instrukcji bloku **var x**; będziemy nazywać deklaracją zmiennych. Zmienne wymienione na liście **x** będą miały charakter pomocniczy, będziemy je nazywać zmiennymi lokalnymi bloku. Pozostałe zmienne będziemy nazywać globalnymi dla rozważanego bloku.

PRZYKŁAD 6.1

```
begin
    x := 5; y := 3;
    block var x;
    begin
        x := 15; y := x
    endbl;
    x := x + 1
end.
```

W tym przykładzie mamy do czynienia z jedną instrukcją bloku, którego deklaracja składa się z jednej tylko zmiennej x. Zmienna y jest zmienną globalną w stosunku do bloku, w którym występuje. Natomiast zmienna x pełni dwie role: w instrukcjach $x := 5$ i $x := x+1$ występuje jako zmienna globalna, a w instrukcji $x := 15$, wewnątrz bloku, występuje jako zmienna lokalna. Obserwujemy tu zjawisko tzw. zasłaniania zmiennych: zmienna globalna x jest zasłonięta przez zmienną lokalną tak długo, jak długo są wykonywane instrukcje bloku, w którym ta zmienna lokalna została zadeklarowana. □

UWAGA

W powyższym przykładzie pominięto dla uproszczenia niektóre nawiasy **begin...end**. W dalszym ciągu będziemy z zasady pomijać te nawiasy, które nie są konieczne do jednoznacznego odczytania struktury programu. ■

Oznaczmy przez $L(\pi_1)$ język algorytmiczny zbudowany zgodnie z zasadami podanymi w p. 3.5 z tym, że w formułach algorytmicznych postaci $M\alpha$, M będzie oznaczać teraz program z klasy π_1. Przyjmijmy, że w języku L, nad którym zbudowano język algorytmiczny, występuje stała θ, która będzie pełniła w tym rozdziale pewną specjalną rolę.

W dalszym ciągu będziemy się posługiwać następującymi oznaczeniami i definicjami przyjętymi w p. 3.6. Jeżeli **x** jest wektorem $x_1, x_2 ..., x_n$, a **u** odpowiadającym mu wektorem $u_1, u_2, ..., u_n$, to wyrażenie **x** := **u**, jest skrótem dla ciągu instrukcji przypisania

$x_1 := u_1; x_2 := u_2; \ldots; x_n := u_n$

Znaczeniem programów z klasy π_1 w strukturze **A** jest, tak jak dla klasy π, funkcja częściowa w zbiorze wszystkich wartościowań w **A**. Semantyka, zarówno instrukcji przypisania, jak i konstrukcji programotwórczych składania, rozgałęzienia i iteracji, jest określona zgodnie z poprzednio przyjętymi zasadami (por. rozdz. 3). Semantykę programów rozszerzymy na instrukcje bloku w następujący sposób:

DEFINICJA 6.2

Dla dowolnego wartościowania v w strukturze danych **A**,

$$(\textbf{block var x; begin M endbl})_A(v) \overset{\text{if}}{=} v_1$$

wtedy i tylko wtedy, gdy istnieje wartościowanie v_2 takie, że

(1) $v_2 = \mathrm{M}_A(v')$, gdzie $v'(z) = v(z)$ dla $z \notin \mathbf{x}$ i $v'(\mathbf{x}) = \theta$

oraz

(2) $v_1(\mathbf{x}) = v(\mathbf{x})$ i $v_1(z) = v_2(z)$ dla $z \notin \mathbf{x}$. ■

Mówiąc nieformalnie, na zewnątrz instrukcji bloku

block var x; begin M endbl

widać zmiany tylko tych zmiennych, które nie są w nim zadeklarowane jako lokalne. Zmiany zmiennych lokalnych nie są widoczne po wykonaniu instrukcji bloku.

PRZYKŁAD 6.2

Załóżmy, że struktura, w której będziemy realizować następujący program jest strukturą liczb rzeczywistych **R** z naturalną interpretacją występujących w programie stałych i operacji. Niech ponadto stała θ oznacza 0. Prześledzimy w tym przykładzie mechanizm zasłaniania zmiennych.

```
begin
  x := 100;
  block var x;
    begin
      x := 1;
      block var x;
        begin
          x := x + 10
        endbl;
        x := x + 1
  endbl;
```

$x := x+1$
end

Zmienna x, występująca w tym programie, wielokrotnie zmienia swoje wartości. W najbardziej wewnętrznym bloku wartością zmiennej x jest 10, natomiast po jego wykonaniu odsłania się wartość zmiennej x z bloku zewnętrznego, zatem przed opuszczeniem zewnętrznej instrukcji bloku wartością x jest 2. Ta z kolei wartość zmiennej x nie jest widoczna po wykonaniu obu instrukcji bloku. Ostatecznie, wartościowanie końcowe spełnia warunek $x = 101$. □

Zauważmy, że jeżeli w pewnym wartościowaniu wartości zmiennych u i z są takie same oraz z, $u \notin V(M)$, a $x \in V(M)$, to wartość zmiennej z po wykonaniu programu M, w którym zmienną x zastąpiono przez z, jest taka sama jak wartość zmiennej u po wykonaniu programu M, w którym x zastąpiono przez u. Mówiąc nieformalnie, wynik programu nie zależy od użytych nazw zmiennych, lecz od ich wartości. W lemacie 6.1 sformułujemy tę zależność bardziej precyzyjnie.

LEMAT 6.1

Niech M będzie dowolnym programem z klasy π_1 oraz $\mathbf{u}$, $\mathbf{x}$ odpowiadającymi sobie skończonymi ciągami zmiennych tego samego typu (por. p. 3.6), takimi, że $\mathbf{x} \cap \mathbf{u} = \emptyset$ i $\mathbf{u} \cap V(M) = \emptyset$. Wówczas, dla dowolnych wartościowań v, v' w dowolnej ekspresywnej strukturze danych $\mathbf{A}$, jeżeli $v = v'$ *off*$(\mathbf{x} \cup \mathbf{u})$ (por. z definicją 3.8) oraz $v'(\mathbf{u}) = v(\mathbf{x})$, to

(1) wartościowanie $\mathrm{M}_{\mathbf{A}}(v)$ jest określone wtedy i tylko wtedy, gdy wartościowanie $\mathrm{M}(\mathbf{x}/\mathbf{u})_{\mathbf{A}}(v')$ są określone;

(2) jeśli oba wartościowania $v_1 = \mathrm{M}_{\mathbf{A}}(v)$ i $v_2 = \mathrm{M}(\mathbf{x}/\mathbf{u})_{\mathbf{A}}(v')$ są określone, to $v_1 = v_2$ *off*$(\mathbf{x} \cup \mathbf{u})$ oraz $v_2(\mathbf{u}) = v_1(\mathbf{x})$, $v_2(\mathbf{x}) = v'(\mathbf{x})$, $v_1(\mathbf{u}) = v(\mathbf{u})$.

DOWÓD

Dowód lematu przebiega przez indukcję ze względu na stopień komplikacji programu i opiera się na analogicznym spostrzeżeniu dotyczącym zbioru termów i zbioru formuł otwartych.
Dla dowolnego termu lub formuły otwartej w, dowolnego wektora zmiennych $\mathbf{u}$ odpowiadającego wektorowi $\mathbf{x}$, takiego, że $\mathbf{u} \cap \mathbf{x} = \emptyset$, $\mathbf{u} \cap V(w) = \emptyset$, i dla dowolnych wartościowań v, v', jeżeli $v = v'$ *off*$(\mathbf{x} \cup \mathbf{u})$ i $v'(\mathbf{u}) = v(\mathbf{x})$, to $w_{\mathbf{A}}(v) = w(\mathbf{x}/\mathbf{u})_{\mathbf{A}}(v')$.

Na mocy przytoczonego faktu wywnioskujemy natychmiast prawdziwość lematu 6.1 dla instrukcji przypisania.

Załóżmy, że lemat jest prawdziwy dla programów M_1 i M_2. Rozważmy

ogram M postaci **begin** M_1; M_2 **end**. Na mocy założenia, jeżeli istnieją rtościowania $M_{1A}(v)$ i $M_{1A}(\mathbf{x/u})(v')$, to też spełniają założenia lematu. obec tego i wobec założenia indukcyjnego dla programu M_2 warlowanie $v_1 = M_{2A}(M_{1A}(v))$ istnieje tylko wtedy, gdy wartościowanie $= M_2(\mathbf{x/u})_A(M_1(\mathbf{x/u})_A(v))$ istnieje oraz jest spełniony warunek (2) lematu.

Jeżeli M jest postaci **if** γ **then** M_1 **else** M_2 **fi**, to w zależności od γ, $M_A(v)$ st równe albo $M_{1A}(v)$, albo $M_{2A}(v)$. Zatem na mocy założenia indukcyjnego rzymujemy tezę lematu.

Jeżeli M jest postaci **while** γ **do** M_1 **od**, to $M_A(v)$ jest z definicji równe **if** γ **then** M_1 **fi**$^i_A(v)$ dla pewnego $i \in N$. Na mocy poprzednich rozważań lemat 6.1 jest już udowodniony dla programów postaci **if** γ **then** M_1 **fi**^{i}, a więc jest też udowodniony dla M.

Niech M będzie postaci **block var y; begin** K **endbl**. Z założenia indukcyjnego lemat 6.1 jest prawdziwy dla programu K, instrukcji przypisania instrukcji złożonej. Zatem jest prawdziwy dla programu M' postaci

begin z := θ; K(**y/z**); **z** := $\tau_{v(\mathbf{z})}$ **end**

gdzie $\mathbf{z} \cap (v(K) \cup \mathbf{y} \cup \mathbf{u}) = \emptyset$.

Niech v, v' będą wartościowaniami w **A** zdefiniowanymi jak w sformułowaniu lematu 6.1. Na mocy założenia indukcyjnego wartościowanie $v_1 = M'_A(v)$ istnieje wtedy i tylko wtedy, gdy wartościowanie $v_2 = M'(\mathbf{x/u})_A(v')$ istnieje oraz, gdy oba istnieją $v_1 = v_2$ *off* $(\mathbf{x} \cup \mathbf{u})$, $v_2(\mathbf{u}) = v_1(\mathbf{x})\, v_2(\mathbf{x}) = v'(\mathbf{x})$, $v_1(\mathbf{u}) = v(\mathbf{u})$. Jednak wyrażenie M' (**x/u**) jest identyczne z wyrażeniem

begin z := θ; K(**x/u, y/z**); **z** := $\tau_{v(\mathbf{z})}$ **end**

Stąd na mocy definicji 6.2, $M'(\mathbf{x/u})_A(v') = M(\mathbf{x/u})_A(v')$ oraz $M'_A(v) = M_A(v)$. Zatem warunek (2) lematu jest spełniony również dla wartościowań $M(\mathbf{x/u})_A(v')$ i $M_A(v)$, co należało udowodnić. □

Pojęcie obliczenia programu z klasy π_1 można zdefiniować, tak jak w przypadku while-programów, za pomocą relacji bezpośredniego następstwa w zbiorze konfiguracji (por. definicję 3.4). W tym celu wystarczy zdefiniować bezpośredni następnik dla konfiguracji z instrukcją bloku. Niech **A** będzie strukturą danych dla języka L. Zakładamy w dalszym ciągu, że wszystkie rozważane w tym rozdziale struktury są strukturami ekspresywnymi (por. p. 4.6). Wynika stąd, że dla każdego elementu a struktury **A** istnieje w języku L term τ_a definiujący ten element (tzn. taki, że dla każdego wartościowania v, $\tau_{aA}(v) = a$).

DEFINICJA 6.3

Niech M_1 będzie instrukcją bloku postaci

block var x; begin M **endbl**

a *Rest* dowolną listą instrukcji. Bezpośrednim następnikiem konfiguracji $\langle v, \mathrm{M}_1; Rest\rangle$ w strukturze **A** jest konfiguracja $\langle v, \mathbf{u} := \theta; \mathrm{M}(\mathbf{x}/\mathbf{u}); \mathbf{u} := \tau_{v(u)}; Rest\rangle$ dla pewnego **u** takiego, że $\mathbf{u} \cap V(\mathrm{M}_1) = \emptyset$ ■

PRZYKŁAD 6.3

Obliczenie programu M przedstawionego w przykładzie 6.1, w strukturze liczb naturalnych **N** ze zwykłą interpretacją stałych i operacji +, jest następujące:

$\langle v, \mathrm{M}\rangle$

$\left\langle \begin{array}{|c|c|c|}\hline x & y & u \\ \hline 5 & - & a \\ \hline \end{array}, \textbf{begin } y := 3; \textbf{block var } x; \textbf{ begin } x := 15; \right.$
$\left. y := x \textbf{ endbl}; x := x+1 \textbf{ end} \right\rangle$

$\left\langle \begin{array}{|c|c|c|}\hline x & y & u \\ \hline 5 & 3 & a \\ \hline \end{array}, \textbf{begin block var } x; \textbf{ begin } x := 15; y := x \right.$
$\left. \textbf{endbl}; x := x+1 \textbf{ end} \right\rangle$

$\left\langle \begin{array}{|c|c|c|}\hline x & y & u \\ \hline 5 & 3 & a \\ \hline \end{array}, \textbf{block var } x; \textbf{ begin } x := 15; y := x \textbf{ endbl}; x := x+1 \right\rangle$

$\left\langle \begin{array}{|c|c|c|}\hline x & y & u \\ \hline 5 & 3 & a \\ \hline \end{array}, u := \theta; \textbf{ begin } u := 15; y := u \textbf{ end}; u := \tau_a; x := x+1 \right\rangle$

$\left\langle \begin{array}{|c|c|c|}\hline x & y & u \\ \hline 5 & 3 & \theta \\ \hline \end{array}, \textbf{begin } u := 15; y := u \textbf{ end}; u := \tau_a; x := x+1 \right\rangle$

$\left\langle \begin{array}{|c|c|c|}\hline x & y & u \\ \hline 5 & 3 & \theta \\ \hline \end{array}, u := 15; y := u; u := \tau_a; x := x+1 \right\rangle$

$\left\langle \begin{array}{|c|c|c|}\hline x & y & u \\ \hline 5 & 3 & 15 \\ \hline \end{array}, y := u; u := \tau_a; x := x+1 \right\rangle$

$\left\langle \begin{array}{|c|c|c|}\hline x & y & u \\ \hline 5 & 15 & 15 \\ \hline \end{array}, u := \tau_a; x := x+1 \right\rangle$

$\left\langle \begin{array}{|c|c|c|}\hline x & y & u \\ \hline 5 & 15 & a \\ \hline \end{array}, x := x+1 \right\rangle$

$\left\langle \begin{array}{|c|c|c|}\hline x & y & u \\ \hline 6 & 15 & a \\ \hline \end{array}, \right\rangle$ □

Definicje 6.2 i 6.3 określają semantykę instrukcji bloku na dwa sposoby: algebraiczny i obliczeniowy (przy pomocy pojęcia obliczenia). Lemat 6.2 stwierdza ich równoważność.

LEMAT 6.2

Dla dowolnego programu $M \in \pi_1$ i dowolnego wartościowania v w strukturze $\mathbf{A}$, $v' \in M_{\mathbf{A}}(v)$ wtedy i tylko wtedy, gdy istnieje skończone udane obliczenie programu M w strukturze $\mathbf{A}$, które przekształca wartościowanie początkowe v w wartościowanie końcowe v'.

DOWÓD

Lemat jest oczywiście prawdziwy dla instrukcji przypisania. Załóżmy jego prawdziwość dla programu K (założenie indukcyjne) i rozważmy przypadek, gdy M jest postaci

block var x; begin K **endbl**

Niech $\mathbf{u}$ będzie wektorem zmiennych odpowiadającym wektorowi $\mathbf{x}$ takim, że $\mathbf{u} \cap V(M) = \emptyset$. Mamy wtedy

$v' \in M_{\mathbf{A}}(v)$ wtw
$v'(z) = v(z)$ dla $z \in \mathbf{x}$ oraz $(\exists v_1)\, v_1 = K_{\mathbf{A}}((\mathbf{x} := \theta)_{\mathbf{A}}(v))$ i $v' = v_1\ off\ \mathbf{x}$ wtw

(na mocy lematu 6.1)

$(\exists v_2)\, v_2 = (\mathbf{begin}\ \mathbf{u} := \theta;\ K(\mathbf{x}/\mathbf{u})\ \mathbf{end})_{\mathbf{A}}(v)$ i $v' = v_2\ off\ (\mathbf{u} \cup \mathbf{x})$,
$v'(z) = v(z)$ dla $z \in (\mathbf{u} \cup \mathbf{x})$ wtw

(na mocy założenia indukcyjnego) dla wartościowania początkowego v, istnieje skończone, udane obliczenie programu **begin** $\mathbf{u} := \theta$; $K(\mathbf{x}/\mathbf{u})$ **end**, którego wynikiem jest v_2 oraz $v' = v_2\ off\ (\mathbf{u} \cup \mathbf{x})$, $v'(z) = v(z)$ dla $z \in \mathbf{u} \cup \mathbf{x}$ wtw istnieje skończone, udane obliczenie programu

begin $\mathbf{u} := \theta$; $K(\mathbf{x}/\mathbf{u})$; $\mathbf{u} := \tau_{v(u)}$ **end**

dla wartościowania początkowego v, oraz jego wynikiem jest v' wtw istnieje skończone, udane obliczenie programu

block var x; begin K **endbl**

dla wartościowania początkowego v, którego wynikiem jest v'.
Dyskusja pozostałych przypadków jest prostsza i pozostawiamy ją czytelnikowi (por. lemat 3.1). □

Aksjomat bloku 6.3

Oznaczmy przez $AL(\pi_1)$ system formalny logiki algorytmicznej dla klasy programów π_1. System ten jest zdefiniowany przez zbiór schematów aksjomatów Ax i reguł wnioskowania RW (por. rozdz. 4) oraz następujący aksjomat bloku:

Axbl: $((\mathbf{u} = \tau) \Rightarrow (\textbf{block var } \mathbf{x}; \textbf{ begin } M \textbf{ endbl } \alpha \equiv$
$\equiv \textbf{begin } \mathbf{u} := \theta; M(\mathbf{x}/\mathbf{u}); \mathbf{u} := \tau \textbf{ end } \alpha))$

gdzie τ jest dowolnym ciągiem termów odpowiadającym $\mathbf{u}$, α jest dowolną formułą, M dowolnym programem oraz

$V(\tau) \cap V(M\mathbf{u}) = \emptyset$, $V(M\mathbf{x}) \cap V(\mathbf{u}) = \emptyset$.

Przytoczona tu postać aksjomatu może wydawać się nadmiernie skomplikowaną. Korzystamy z niej w dowodzie lematu 6.4, by wykazać, że udało się nam podać aksjomatyczną definicję semantyki programów iteracyjnych z blokami. W dowodach własności semantycznych programów z blokami można wykorzystywać uproszczoną wersję aksjomatu bloku (por. przykład 6.4). Jest oczywiste, że można dobrać zmienne $\mathbf{u}$ w taki sposób, by nie występowały ani w instrukcji bloku, ani w formule α. Wtedy można pominąć instrukcję $\mathbf{u} := \tau$ i konsekwentnie warunek $\mathbf{u} = \tau$. Aksjomat przyjmuje wtedy prostszą postać

block var x; **begin** M **andbl** $\alpha \equiv$ **begin** $\mathbf{u} := \theta$; M(x/u) **end** α

PRZYKŁAD 6.4

Następujący ciąg równoważnych formuł stanowi dowód tego, że, po wykonaniu programów z przykładu 6.1, jest spełniony warunek $(x = 6 \wedge y = 15)$.

(1) $6 = 6 \wedge 15 = 15$ (aksjomaty równości)

(2) $5+1 = 6 \wedge 15 = 15$ (prawa arytmetyki)

(3) $(x := 5)(x+1 = 6 \wedge 15 = 15)$ {Ax 18}

(4) $(x := 5)(y := 3)(x+1 = 6 \wedge 15 = 15)$ {Ax 18}

(5) $(x := 5)(y := 3)(u := \theta)(x+1 = 6 \wedge 15 = 15)$ {Ax 18}

(6) $(x := 5)(y := 3)(u := \theta)(u := 15)(x+1 = 6 \wedge u = 15)$ {Ax 18}

(7) $(x := 5)(y := 3)(u := \theta)(u := 15)(y := u)(x+1 = 6 \wedge y = 15)$ {Ax 18}

(8) $(x := 5)(y := 3)$ **block var** x; **begin** $x := 15$; $y := x$ **endbl**
$(x+1 = 6 \wedge y = 15)$ {Ax bl}

(9) $(x := 5)(y := 3)$ **block var** x; **begin** $x := 15$; $y := x$ **endbl**
$(x := x+1)(x = 6 \wedge y = 15)$ {Ax 18}

(10)
```
begin
  x := 5; y := 3;
  block var x
  begin
    x := 15; y := x
  end;
```

$x := x + 1$
end $(x = 6 \wedge y = 15)$ {Ax 19}
□

Należy teraz wykazać, że system logiczny $\mathrm{AL}(\pi_1)$ jest niesprzeczny i pełny. Dowód słuszności aksjomatyzacji można ograniczyć do następującego lematu, gdyż razem z faktami udowodnionymi w rozdz. 4, pozwala on wykazać, że każde twierdzenie logiki $\mathrm{AL}(\pi_1)$ jest tautologią.

LEMAT 6.3

Dla dowolnej struktury danych **A**, $\mathbf{A} \models \mathrm{Axbl}$.

DOWÓD

Niech **A** będzie ustaloną strukturą semantyczną. Przypuśćmy, że dla pewnego v takiego, że $\mathbf{A}, v \models (u = \tau)$, $\mathbf{u} \cap (V(\mathrm{M}) \cup V(\tau)) = \emptyset$, mamy

$\mathbf{A}, v \models$ **block var x; begin** M **endbl** α

Niech wtedy wartościowanie v' takie, że

$(v, v') \in$ **block var x; begin** M **endbl**$_{\mathbf{A}}$ i $\mathbf{A}, v' \models \alpha$

Z definicji instrukcji bloku wynika, że wartościowanie

$v_1 =$ **begin x** $:= \theta$; M **end**$_{\mathbf{A}}(v)$

jest określone oraz $v'(x) = v(x)$ dla $x \in \mathbf{x}$, $v' = v_1$ *off* **x**. Na mocy lematu 6.1 niech wartościowanie $v_2 =$ **begin u** $:= \theta$; M**(x/u) end**$_{\mathbf{A}}(v)$ oraz $v_2 = v'$ *off* **(x ∪ u)**, a ponieważ $\mathbf{x} \cap (V(\mathrm{M}(\mathbf{x}/\mathbf{u})) \cup \mathbf{u}) = \emptyset$, to $v_2(x) = v'(x)$. Z założenia mamy **u**, $V(\tau)$, $V(\mathrm{M})$ są parami rozłączne, zatem otrzymujemy $v' =$ **begin u** $:= \theta$; M**(x/u)**; **u** $:= \tau$ **end**$_{\mathbf{A}}(v)$, a co za tym idzie

$\mathbf{A}, v \models$ **begin u** $:= \theta$; M**(x/u)**; **u** $:= \tau$ **end** α

Jeżeli $\mathbf{A}, v \vdash \mathbf{u} = \tau$ oraz $\mathbf{A}, v \models$ **block var x; begin** M **andbl** α, to przedstawione rozumowanie można odwrócić, co kończy dowód lematu 6.3. □

Dla logiki algorytmicznej $\mathrm{AL}(\pi_1)$ można udowodnić, analogicznie jak w przypadku AL, twierdzenie o pełności. Przytoczymy je bez dowodu.

TWIERDZENIE 6.1

Dla dowolnej formuły α następujące warunki są równoważne:
(1) α jest twierdzeniem teorii T opartej na logice $\mathrm{AL}(\pi_1)$;
(2) α jest prawdziwa w każdym modelu teorii T. ■

W rozdz. 4 wykazaliśmy, że system AL(π) definiuje jednoznacznie znaczenie konstrukcji programotwórczych składania, rozgałęziania i iteracji. Obecnie udowodnimy, że przyjęty tutaj aksjomat bloku wyznacza jednoznacznie znaczenie instrukcji bloku.

LEMAT 6.4

Niech **A** będzie strukturą ekspresywną, a $\langle \mathbf{A}, \mathrm{I}, \models \rangle$ strukturą semantyczną spełniającą warunki wymienione w p. 4.5. Jeżeli $\langle \mathbf{A}, \mathrm{I}, \models \rangle$ jest modelem dla AL(π_1), to

$$\mathrm{I}(\textbf{block var x; begin M endbl}) = \mathrm{I}(\mathbf{u} := \theta) \circ \mathrm{I}(\mathrm{M}(\mathbf{x}/\mathbf{u})) \circ \mathrm{I}(\mathbf{u} := \tau_{v(u)})$$

gdzie $\mathbf{u} \cap V(\mathrm{M}) = \emptyset$, $\mathbf{u} \cap \mathbf{x} = \emptyset$.

DOWÓD

Niech $(v, v') \in \mathrm{I}(\textbf{block var x; begin M endbl})$ i niech α będzie formułą taką, że $\mathbf{A}, v' \models \alpha$. Załóżmy ponadto, że $\mathbf{u} \cap V(\mathrm{M}) = \emptyset$. Na mocy tych założeń, istnieje term $\tau_{v(u)}$ taki, że

$$\mathbf{A}, v \models \mathbf{u} = \tau_{v(u)} \quad \text{oraz} \quad \mathbf{A}, v \models \textbf{block var x; begin M endbl } \alpha$$

Ponieważ $\mathbf{A} \models$ Axbl, więc

$$\mathbf{A}, v \models \textbf{begin } \mathbf{u} := \theta; \mathrm{M}(\mathbf{x}/\mathbf{u}); \mathbf{u} := \tau_{v(u)} \textbf{ end } \alpha$$

Stąd istnieje wartościowanie v_1 takie, że

$$(v, v_1) \in \mathrm{I}(\mathbf{u} := \theta; \mathrm{M}(\mathbf{x}/\mathbf{u}); \mathbf{u} := \tau_{v(u)}) \text{ i } \mathbf{A}, v_1 \models \alpha$$

Ponieważ α jest dowolną formułą, więc wartościowania v' i v_1 są nieseparowalne (por. lemat 4.5), a co za tym idzie $v' = v_1$. Ostatecznie

$$(v, v') \in \mathrm{I}(\mathbf{u} := \theta; \mathrm{M}(\mathbf{x}/\mathbf{u}); \mathbf{u} := \tau_{v(u)}).$$

Odwrotnie, jeżeli $(v, v') \in \mathrm{I}(\mathbf{u} := \theta; \mathrm{M}(\mathbf{x}/\mathbf{u}); \mathbf{u} := \tau_{v(u)})$, $\mathbf{A}, v' \models \alpha$ oraz $\mathbf{u} \cap V(\mathrm{M}) = \emptyset$, to wobec $\mathbf{A}, v \models \mathbf{u} = \tau_{v(u)}$, mamy

$$\mathbf{A}, v \models \textbf{block var x; begin M endbl } \alpha$$

Istnieje zatem wartościowanie v_1 takie, że $\mathbf{A}, v_1 \models \alpha$ oraz $(v, v_1) \in \mathrm{I}(\textbf{block var x; begin M endbl})$. Ponieważ α jest dowolną formułą, to $v_1 = v'$. Ostatecznie

$$(v, v') \in \mathrm{I}(\textbf{block var x; begin M endbl}) \qquad \square$$

UWAGA

Wybór zmiennych **u** w sformułowaniu lematu 6.4 jest nieistotny, gdyż na mocy lematu 6.1 oraz twierdzenia 4.11

$$\mathbf{I}(\mathbf{u} := 0;\ \mathbf{M}(\mathbf{x}/\mathbf{u});\ \mathbf{u} := \tau_{v(u)}) = \mathbf{I}(\mathbf{z} := 0;\ \mathbf{M}(\mathbf{x}/\mathbf{z});\ \mathbf{z} := \tau_{v(z)})$$

dla dowolnych $\mathbf{u}$, $\mathbf{z}$ takich, że $(\mathbf{u} \cup \mathbf{z}) \cap V(\mathbf{M}) = \emptyset$. ■

Podprogramy i procedury 6.4

W tym punkcie będziemy rozważać klasę programów π_2, która mówiąc nieformalnie, powstaje z klasy π_1 przez dodanie nowej konstrukcji programotwórczej, zwanej instrukcją procedury. Instrukcja ta nie tylko istotnie rozszerza język, ale także umożliwia uproszczenie samej struktury programu.

PRZYKŁAD 6.5

Rozwiązanie układu n równań z n niewiadomymi można sprowadzić do wielokrotnego obliczania wyznacznika. Zamiast powtarzać algorytm obliczania wyznacznika wszędzie tam gdzie jest potrzebna jego wartość, można najpierw zdefiniować procedurę, której treścią będzie algorytm obliczania wyznacznika, a potem odwoływać się do tej definicji zawsze, gdy chcemy znać wartość konkretnego wyznacznika. □

Powszechnie jest stosowana metoda nadawania nazwy pojęciom na tyle skomplikowanym, że jest kłopotliwe wielokrotne ich objaśnianie. Procedura jest nazwą pewnego pojęcia zdefiniowanego algorytmicznie.

PRZYKŁAD 6.6

Niech zadanie polega na obliczeniu funkcji Fibonacciego $Fib(n)$ zdefiniowanej dla $n \in N$ za pomocą następujących zależności rekurencyjnych

$$Fib(0) \overset{\text{df}}{=} 0$$
$$Fib(1) \overset{\text{df}}{=} 1$$
$$Fib(n+2) = Fib(n) + Fib(n+1) \quad \text{dla } n > 1.$$

Program obliczania wartości tej funkcji może wyglądać następująco:

```
begin
    a := 0;
    if n = 0 then wynik := 0
    else
        b := 1; i := 1;
        while i ⩽ n
        do
```

```
            c := b; b := a+b;
            a := c; i := i+1
        od;
        wynik := b
    fi
end
```

W algorytmie opisanym tym programem za każdym razem, gdy chcemy policzyć wartość funkcji dla pewnego n, zaczynamy obliczenia od początku. Tymczasem dla obliczenia $Fib(n+2)$ wystarczy znać dwie wartości $Fib(n)$ i $Fib(n+1)$. Inny algorytm obliczania wartości tej funkcji przedstawimy w następnym przykładzie. □

Procedury (niefunkcyjne), o których mowa w tym rozdziale, dopuszczają odwoływanie się do swojej treści umożliwiając w ten sposób, pisanie algorytmów rekurencyjnych. Jeżeli w procedurze nie występuje odwołanie do jej treści ani do treści żadnej innej procedury, to nazwiemy ją podprogramem. Syntaktycznie procedura składa się z dwu części: części definiującej oraz instrukcji wykonania. Te dwie części będziemy nazywać odpowiednio deklaracją procedury i instrukcją procedury.

Wprowadzenie procedur wymaga rozszerzenia rozważanego do tej pory języka o zbiór nazw dla procedur, tzw. identyfikatorów procedur. Z każdym identyfikatorem procedury związemy parę liczb naturalnych, którą będziemy nazywać typem identyfikatora. Będzie on określał ilość i rodzaj parametrów procedury.

DEFINICJA 6.4

Deklaracją procedury będziemy nazywać każde wyrażenie postaci

procedure p **(in x; out y); begin M endproc**

gdzie p jest identyfikatorem procedury typu (n, m), **M** jest ciągiem programów z π'_2 (definicję klasy π'_2 podamy dalej), a **x**, **y** są ciągami zmiennych o długości odpowiednio n i m.
Zmienne **x**, **y** są parametrami formalnymi procedury (odpowiednio wejściowymi i wyjściowymi), a **M** treścią procedury p. Wyrażenie p **(in x; out y)**, występujące w deklaracji procedury, będziemy nazywać nagłówkiem procedury. ■

DEFINICJA 6.5

Instrukcją procedury nazywamy każde wyrażenie postaci p **(a, b)** gdzie p jest identyfikatorem procedury typu (n, m), **a** jest (odpowiadającym **x**) n-elementowym ciągiem termów, a **b** (odpowiadającym **y**) m-elementowym ciągiem

zmiennych. Ciągi **a**, **b** będziemy nazywać parametrami aktualnymi procedury p. ■

PRZYKŁAD 6.7

Rozważmy następującą deklarację procedury:

```
procedure Fib(in n; out w);
var i, k;
begin
    if n = 0 then w := 0 else
        if n = 1 then w := 1 else
            Fib(n-2, i);
            Fib(n-1, k);
            w := i+k
        fi
    fi
endproc
```

W tym przykładzie *Fib* jest identyfikatorem procedury typu (1, 1). Zmienne n, w są parametrami formalnymi procedury, i, k są jej zmiennymi lokalnymi. Treść procedury stanowi instrukcja warunkowa. Wyrażenia $Fib(n-2; i)$ i $Fib(n-1; k)$, występujące w treści procedury, są instrukcjami procedury, której parametrami aktualnymi są: term $n-2$ i zmienna i w pierwszym przypadku oraz term $n-1$ i zmienna k w drugim przypadku. □

Zbiór π'_2 zdefiniujemy teraz jako najmniejszy zbiór wyrażeń zawierający instrukcje przypisania i instrukcje procedury, zamknięty ze względu na znane reguły składania, rozgałęziania, iteracji oraz następującą regułę tworzenia bloku:

jeżeli **dekl** jest ciągiem (być może pustym) deklaracji procedur o różnych identyfikatorach, **z** jest ciągiem różnych zmiennych (być może pustym), a **M** ciągiem wyrażeń z π'_2, to wyrażenie **block var z; dekl; begin M endbl**, jest elementem π'_2.

Wielkościami lokalnymi bloku

block var z; deki; begin M andbl

są zmienne występujące w ciągu **z** i nazwy procedur zadeklarowanych w ciągu **dekl**. Wielkościami globalnymi tego bloku są wszystkie inne zmienne występujące w tym bloku.

Wielkościami lokalnymi deklaracji procedury

procedure p**(in x; out y); var z; dekl; begin M endproc**

są nazwy procedur zadeklarowanych w ciągu **dekl**, zmienne występujące

w ciągu **z** i nazwy parametrów formalnych występujące w ciągach **x** i **y**. Pozostałe wielkości występujące w tej deklaracji procedury są globalne (nielokalne) dla niej.

W dalszym ciągu słowo *moduł* będzie oznaczać blok lub deklarację procedury.

DEFINICJA 6.6

Klasą π_2 programów z procedurami będziemy nazywać największy podzbiór zbioru π_2' taki, że każdy jego element M spełnia następujące warunki:

(1) każde wystąpienie instrukcji procedury jest zawarte albo w deklaracji procedury o tym samym identyfikatorze, albo w bloku, w którym występuje deklaracja tej procedury;

(2) każde dwa moduły zawarte w wyrażeniu M mają rozłączne zbiory wielkości lokalnych w tych modułach. ■

Czytelnik może żachnąć się na ten drugi warunek i zawołać, iż ogranicza się swobodę układania programów. Wcale tak nie jest. Każdy program może być automatycznie i w łatwy sposób przekształcony do równoważnego mu programu spełniającego oba wyliczone przez nas warunki.

Zadanie określenia semantyki programów klasy π_2 jest znacznie bardziej skomplikowane niż w przypadku klasy π_1. Podstawowa idea polega na zastąpieniu instrukcji procedury jej zmodyfikowaną treścią. Wprowadzenie pojęcia obliczenia poprzedzimy kilkoma niezbędnymi oznaczeniami.

Rozważmy następującą deklarację procedury

```
procedure p(in x; out y);
begin
  M
endproc                                    (6.1)
```

Niech **a** będzie ciągiem termów, **b** zaś ciągiem zmiennych odpowiadających **x** i **y**. Przez *Mod*(**a**, **b**, *p*) będziemy rozumieli następujący program:

```
block var u, w;
begin
    u := a;
    M(x/u, y/w);
    b := w
endbl
```

gdzie **u** jest wektorem zmiennych odpowiadającym **x**, a **w** jest wektorem odpowiadającym **y** oraz $(\mathbf{u} \cup \mathbf{w}) \cap (\mathbf{a} \cup \mathbf{b} \cup V(\mathrm{M})) = \emptyset$.

Niech M będzie programem zawierającym instrukcję procedury $p(\mathbf{a}, \mathbf{b})$ oraz niech odpowiadającą tej instrukcji deklaracją w M będzie (6.1). Przez

$Mod(\mathrm{M})$ oznaczmy program, powstający z M przez tekstowe zastąpienie rozważanego wystąpienia instrukcji procedury jej zmodyfikowaną treścią $Mod(\mathbf{a}, \mathbf{b}, p)$.

Niech **A** będzie strukturą danych rozważanego języka pierwszego rzędu L. Zakładamy, tak jak w poprzednim punkcie, że **A** jest strukturą ekspresywną (por. p. 4.5). Relację bezpośredniego następstwa $\mapsto$ w zbiorze wszystkich konfiguracji w **A**, zdefiniowaną jak w rozdz. 3 (por. z definicją 6.3), rozszerzymy w następujący sposób. Dla dowolnego wartościowania v w strukturze **A** przyjmujemy

$$\langle v, \textbf{block var z; dekl; begin M endbl}; \ldots\rangle \mapsto$$
$$\langle v, \textbf{block dekl; begin u} := \theta; \mathbf{M(z/u)}; \mathbf{u} := \tau_{v(u)} \textbf{ endbl}; \ldots\rangle$$

dla dowolnego **u** takiego, że $\mathbf{u} \cap V(\mathbf{M}) = \emptyset$ i $\mathbf{z} \neq \emptyset$

$$\langle v, \textbf{block dekl; begin M endbl}; \ldots\rangle \mapsto \langle v, \mathbf{M}; \ldots\rangle$$

gdy w **M** nie występuje żadna instrukcja procedury zadeklarowana w rozważanym bloku

$$\langle v, \textbf{block dekl; begin } p(\mathbf{a}, \mathbf{b}); \mathbf{M} \textbf{ endbl}; \ldots\rangle \mapsto$$
$$\langle v, \textbf{block dekl; begin } Mod(\mathbf{a}, \mathbf{b}, p); \mathbf{M} \textbf{ endbl}; \ldots\rangle$$

Jeżeli K jest instrukcją złożoną, przypisania, warunkową, iteracji lub instrukcją bloku oraz jeśli $\langle v, \mathrm{K}\rangle \mapsto \langle v'. \mathrm{K}'\rangle$, to

$$\langle v, \textbf{block dekl; begin K; M endbl}; \ldots\rangle \mapsto$$
$$\langle v', \textbf{block dekl; begin K' M endbl}; \ldots\rangle$$

Pojęcie obliczenia programu definiujemy tak samo jak dla klasy π oraz przyjmujemy, że dla dowolnego programu $\mathrm{M} \in \pi_2$ i dowolnych wartościowań v, v' w strukturze **A**, $(v, v') \in \mathrm{M}_{\mathrm{A}}$ wtedy i tylko wtedy, gdy istnieje udane skończone obliczenie M przekształcające wartościowanie początkowe v w wartościowanie końcowe v'.

Dla dowolnego programu M oznaczmy przez $\mathrm{M}^{\emptyset}$ program otrzymany z M przez równoczesne zastąpienie wszystkich instrukcji procedury stale zapętlającym się programem $\emptyset$ i przez usunięcie wszystkich deklaracji procedury.

LEMAT 6.5

Dla dowolnego programu $\mathrm{M} \in \pi_2$ i dowolnych wartościowań v, v' w strukturze **A**

$$v' \in \mathrm{M}_{\mathrm{A}}(v) \quad \text{wtw} \quad v' \in Mod(\mathrm{M})_{\mathrm{A}}(v) \tag{6.2}$$
$$v' \in \mathrm{M}^{\emptyset}_{\mathrm{A}}(v) \quad \text{implikuje} \quad v' \in \mathrm{M}_{\mathrm{A}}(v) \tag{6.3}$$

DOWÓD

Dowód pierwszej własności wynika natychmiast z definicji pojęcia obliczenia. Dla dowodu drugiej własności zauważmy, że jeżeli istnieje udane skończone obliczenie programu $\mathrm{M}^{\emptyset}$, prowadzące z wartościowania v do wartościowania v', to w tym obliczeniu nie mogło dojść do wykonania instrukcji $\emptyset$. Zatem obliczenie programu M nigdy nie dojdzie do wykonania instrukcji procedury. Obliczenie programu M różni się od obliczenia programu $\mathrm{M}^{\emptyset}$ przy tym samym wartościowaniu początkowym jedynie tym, że przechowuje informację o definicjach procedur tak długo, jak długo występują w nim instrukcje procedury.

Pełny dowód, przebiegający przez indukcję ze względu na postać programu, pomijamy. □

LEMAT 6.6

Dla dowolnego programu $\mathrm{M} \in \pi_2$ i dla dowolnych wartościowań v, v' w strukturze $\mathbf{A}$

$$v' \in \mathrm{M}_A(v) \quad \text{wtw} \quad \text{istnieje } i \text{ takie, że } v' \in (Mod^i(\mathrm{M}))^{\emptyset}_A(v)$$

DOWÓD

Implikacja $\Leftarrow$ wynika natychmiast z poprzedniego lematu, jeżeli bowiem $v' \in (Mod^i(\mathrm{M}))^{\emptyset}_A(v)$, to z własności (6.3)

$$v' \in Mod^i(\mathrm{M})_A(v)$$

i z własności (6.2) $v' \in \mathrm{M}_A(v)$.

Rozważmy implikację $\Rightarrow$. Niech $v' \in \mathrm{M}_A(v)$. Istnieje zatem obliczenie programu M, które przekształca v w wartościowanie końcowe v'. Jeżeli w obliczeniu programu M nie doszło do wykonania żadnej instrukcji procedury, to $i = 0$ i teza lematu jest konsekwencją przyjętej deklaracji procedury (6.1). W przeciwnym razie, niech

$$p_1(\mathbf{a}_1, \mathbf{b}_1), \ldots, p_n(\mathbf{a}_n, \mathbf{b}_n)$$

będą wszystkimi wystąpieniami instrukcji procedury, które są wykonywane w rozważanym obliczeniu. Oznaczmy przez K program $Mod^n(\mathrm{M})$, gdzie dla $i \leqslant n$, $Mod^i(\mathrm{M})$ powstaje z $Mod^{i-1}(\mathrm{M})$ przez tekstowe zastąpienie wystąpienia $p_i(\mathbf{a}_i, \mathbf{b}_i)$ instrukcji procedury jej zmodyfikowaną treścią $Mod(\mathbf{a}_i, \mathbf{b}_i, p_i)$. Obliczenie programu $Mod^n(\mathrm{M})$ przebiega dokładnie tak, jak obliczenie programu M i $v' \in Mod^n(\mathrm{M})_A(v)$. Chociaż w $Mod^n(\mathrm{M})$ mogą wystąpić jeszcze instrukcje procedury, to na mocy założenia, nie dojdzie do ich wykonania, jeżeli rozważamy obliczenie programu K przy wartościowaniu v. Zatem $v' \in \mathrm{K}^{\emptyset}_A(v)$, co należało pokazać. □

Aksjomaty procedur 6.5

Oznaczmy przez $AL(\pi_2)$ logikę algorytmiczną dla klasy programów π_2 (por. p. 6.4) w języku L. Zbiór aksjomatów i reguł wnioskowania logiki $AL(\pi_2)$ jest rozszerzeniem aksjomatyzacji podanej w rozdz. 4 o następujące aksjomaty i regułę wnioskowania:

$$\begin{aligned}\text{Axproc}\quad & \mathrm{M}\alpha \equiv \mathit{Mod}(\mathrm{M})\,\alpha \\ & (\mathrm{M}^{\emptyset}\alpha \Rightarrow \mathrm{M}\alpha) \\ & (u = \tau \Rightarrow (\textbf{block var z; dekl; begin M endbl } \alpha \equiv \\ & \qquad \equiv \textbf{block deki; begin u} := \theta;\ \mathrm{M}(\mathbf{z/u});\ \mathbf{u} := \tau \textbf{ endbl}))\,\alpha\end{aligned}$$

gdy $\mathbf{u} \cap V(\mathrm{M}) = \emptyset$, $\mathbf{z} \neq \emptyset$ i $V(\tau) \cap (V(\mathrm{M}) \cup V(\mathbf{u})) = \emptyset$,

$$\textbf{block deki; begin } \mathrm{M1} \textbf{ endbl } \alpha = \textbf{begin } \mathrm{M1} \textbf{ end } \alpha$$

$$\text{Rproc}\quad \frac{\{(\mathrm{K}^{\emptyset}\alpha \Rightarrow \beta)\}\ \mathrm{K} \in \mathit{Der}(\mathrm{M})}{(\mathrm{M}\alpha \Rightarrow \beta)}$$

We wszystkich wyrażeniach M jest dowolnym programem z klasy π_2, M1 zaś dowolnym programem, w którym nie występują instrukcje procedur odpowiadające deklaracjom **dekl**.

Der (M) jest najmniejszym zbiorem programów zawierających M i takich, że jeżeli $\mathrm{K} \in \mathit{Der}(\mathrm{M})$, to $\mathit{Mod}(\mathrm{K}) \in \mathit{Der}(\mathrm{M})$.

LEMAT 6.7

Dowolna struktura danych **A** jest modelem dla logiki $AL(\pi_2)$.

DOWÓD

Prawdziwość aksjomatów $AL(\pi_2)$ w strukturze **A** wynika z lematu 6.5 i twierdzenia o słuszności aksjomatyzacji dla logiki $AL(\pi_1)$. Słuszność reguły Rproc wynika wprost z lematu 6.6. □

Dla logiki rozważanej w tym punkcie zachodzi również twierdzenie o pełności stwierdzające, że wszystko co jest prawdziwe w przyjętej semantyce, można udowodnić ze zbioru aksjomatów za pomocą reguł systemu $AL(\pi_2)$.

W dalszym ciągu rozważań zajmiemy się zbadaniem czy system $AL(\pi_2)$ można traktować jako definicję semantyki. Niech **A** będzie strukturą ekspresywną, a $\langle \mathbf{A}, \mathrm{I}, \models \rangle$ strukturą semantyczną zdefiniowaną tak, jak w p. 4.6.

Nasze rozważania rozpoczniemy od prostego faktu wynikającego z przyjętych własności struktury semantycznej.

LEMAT 6.8

Jeżeli $\langle \mathbf{A}, \mathrm{I}, \models \rangle$ jest modelem dla logiki $\mathrm{AL}(\pi_2)$, to dla dowolnego programu $\mathrm{M} \in \pi_2$

$$\mathrm{I}(\mathrm{M}) = \mathrm{I}(Mod(\mathrm{M}))$$
$$\mathrm{I}(\mathrm{M}^{\emptyset}) \subset \mathrm{I}(\mathrm{M})$$ ■

LEMAT 6.9

Jeżeli $\langle \mathbf{A}, \mathrm{I}, \models \rangle$ jest modelem logiki $\mathrm{AL}(\pi_2)$, to dla dowolnego programu M zachodzi następująca równość

$$\mathrm{I}(\mathrm{M}) = \bigcup_{\mathrm{K} \in Der(\mathrm{M})} \mathrm{I}(\mathrm{K}^{\emptyset})$$

DOWÓD

Niech $(v, v') \in \mathrm{I}(\mathrm{K}^{\emptyset})$ dla pewnego $\mathrm{K} \in Der$ (M) takiego, że $\mathrm{K} = Mod^{i}(\mathrm{M})$. Wtedy na mocy lematu 6.8, $(v, v') \in \mathrm{I}(Mod^{i}(\mathrm{M}))$, a co za tym idzie $(v, v') \in \mathrm{I}(\mathrm{M})$. Dowodzi to inkluzji

$$\bigcup_{\mathrm{K} \in Der} \mathrm{I}(\mathrm{K}^{\emptyset}) \subset \mathrm{I}(\mathrm{M})$$

Dla dowodu drugiej części lematu, niech $(v_1, v_2) \in \mathrm{I}(\mathrm{M})$ i niech α będzie formułą taką, że $\mathbf{A}, v_2 \models \alpha$. Załóżmy że β jest formułą, która opisuje wartościowanie v_1 ze względu na wszystkie zmienne $x_1, \ldots, x_n$ występujące w M

$$\beta = (x_1 = \tau_{v(x_1)} \wedge \ldots \wedge x_n = \tau_{v(x_n)})$$

Zatem $\mathbf{A}, v_1 \models \mathrm{M}\alpha$ oraz $\mathbf{A}, v_1 \models \beta$ i *non* $\mathbf{A}, v_1 \models (\mathrm{M}\alpha \Rightarrow \neg\beta)$. Na mocy reguły Rproc istnieje $\mathrm{K} \in Der(\mathrm{M})$ i wartościowanie v' takie, że $\mathrm{K} = Mod^{i}(\mathrm{M})$ dla pewnego i oraz

$$non\ \mathbf{A}, v' \models (\mathrm{K}^{\emptyset}\alpha \Rightarrow \neg\beta) \tag{6.4}$$

Stąd $\mathbf{A}, v' \models \beta$ i w konsekwencji wartościowanie v' jest identyczne z wartościowaniem v_1 na wszystkich zmiennych ze zbioru $V(\mathrm{M})$. Na mocy własności (6.4) $\mathbf{A}, v_1 \models \mathrm{K}^{\emptyset}\alpha$, istnieje więc wartościowanie v_2' takie, że

$$(v_1, v_2') \in \mathrm{I}(\mathrm{K}^{\emptyset}) \text{ i } \mathbf{A}, v_2' \models \alpha$$

Jeżeli jako α weźmiemy formułę opisującą wartościowanie v_2 dla wszystkich występujących w M zmiennych, to otrzymamy równość $v_2 = v_2'$ na zbiorze $V(\mathrm{M})$. Wartościowania v_2 i v_2' są identyczne na zbiorze zmiennych, które nie występują w $V(\mathrm{M})$. Zatem $(v_1, v_2) \in \mathrm{I}(\mathrm{K}^{\emptyset})$ i w konsekwencji

$$(v_1, v_2) \in \bigcup_{\mathrm{K} \in Der(\mathrm{M})} \mathrm{I}(\mathrm{K}^{\emptyset})$$ □

Z lematu 6.9 wynika, że logika algorytmiczna AL(π_2) definiuje jednoznacznie semantykę programów z procedurami niefunkcyjnymi. Co więcej, na mocy lematu 6.6 znaczenie programu wyznaczone przez aksjomatyzację jest dokładnie takie, jakie wyznacza pojęcie obliczenia zdefiniowane w p. 6.3.

Procedury funkcyjne. Obliczenia formalne 6.6

Niech $\varphi_1, ..., \varphi_k$ oznaczają funktory, a $\varrho_1, ..., \varrho_l$, predykaty nie należące do języka algorytmicznego L(π), w którym dopuszczono termy algorytmiczne (por. p. 3.5). Przyjmijmy, że funktor φ_i jest n_i-argumentowy dla $i = 1, ..., k$, a predykat ϱ_j jest m_j-argumentowy dla $j = 1, ..., l$. Przez L' będziemy oznaczać rozszerzenie języka L otrzymane przez dołączenie funktorów $\varphi_1, ..., \varphi_k$ i predykatów $\varrho_1, ..., \varrho_l$ do alfabetu języka L.

DEFINICJA 6.7

Systemem procedur definiujących funktory $\varphi_1, ..., \varphi_k$ i predykaty $\varrho_1, ..., \varrho_l$ będziemy nazywać układ (*) postaci

$$\begin{aligned}
&\varphi_1(x_1, ..., x_{n1}) = \mathrm{K}_1\,\tau_1 \\
&\ldots\ldots\ldots\ldots\ldots \\
&\varphi_k(x_1, ..., x_{nk}) = \mathrm{K}_k\,\tau_k \\
&\varrho_1(x_1, ..., x_{m1}) \equiv \mathrm{M}_1\,\alpha_1 \\
&\ldots\ldots\ldots\ldots\ldots \\
&\varrho_l(x_1, ..., x_{ml}) \equiv \mathrm{M}_l\,\alpha_l
\end{aligned} \qquad (*)$$

gdzie $\mathrm{K}_1, ..., \mathrm{K}_k$, $\mathrm{M}_1, ..., \mathrm{M}_l$ są programami w języku L', $\tau_1, ..., \tau_k$ są termami, $\alpha_1, ..., \alpha_l$ są formułami w języku L' oraz dla dowolnego $i \leqslant k$, $V(\mathrm{K}_i\,\tau_i) = \{x_1, ..., x_{ni}\}$ i dla dowolnego $j \leqslant l$, $V(\mathrm{M}_j\,\alpha_j) = \{x_1, ..., x_{mj}\}$. ■

W dalszym ciągu wprowadzimy semantykę języka L' opartą na pojęciu obliczenia formalnego.

Niech **A** będzie strukturą danych dla języka L (a zatem funktory $\varphi_1, ..., \varphi_k$ oraz predykaty $\varrho_1, ..., \varrho_l$ nie mają w **A** ustalonej interpretacji).

DEFINICJA 6.8

Triadą w A będziemy nazywali dowolną trójkę postaci $\langle v, exp, w \rangle$, gdzie v jest wartościowaniem w struktrze **A**, *exp* jest dowolnym wyrażeniem języka L' poprawnie zbudowanym (tzn. termem, formułą lub programem), a w jest albo elementem struktury **A**, albo wartością boolowską, albo wartościowaniem w **A**.

Triady postaci $\langle v, z, v(z)\rangle$ lub $\langle v, \varphi, \varphi_{\mathbf{A}}\rangle$, gdzie z jest zmienną, φ zero-argumentowym funktorem, a $\varphi_{\mathbf{A}}$ interpretacją tego funktora (tzn. stałą), będziemy nazywać *triadami elementarnymi.* ■

W zbiorze triad określimy pewne reguły przekształcenia zwane dalej regułami obliczeń formalnych. Każda reguła składa się z przesłanek i z wniosku. Niech ROF oznacza zbiór następujących reguł obliczania formalnego.

$$R-\varphi \quad \frac{\langle v, \tau_1, w_1\rangle \ldots \langle v, \tau_n, w_n\rangle}{\langle v, \varphi(\tau_1, \ldots, \tau_n), w\rangle}$$

gdzie φ jest funktorem z L, a w jest wartością operacji $\varphi_{\mathbf{A}}$ dla argumentów $w_1, \ldots, w_n$

$$R-\varrho \quad \frac{\langle v, \tau_1, w_1\rangle \ldots \langle v, \tau_n, w_n\rangle}{\langle v, \varrho(\tau_1, \ldots, \tau_n), w\rangle}$$

gdzie ϱ jest predykatem z L, a w jest wartością relacji $\varrho_{\mathbf{A}}$ dla argumentów $w_1, \ldots, w_n$

$$R-\neg_1 \quad \frac{\langle v, \alpha, \mathbf{true}\rangle}{\langle v, \neg\alpha, \mathbf{false}\rangle} \qquad \text{i} \qquad R-\neg_2 \quad \frac{\langle v, \alpha, \mathbf{false}\rangle}{\langle v, \neg\alpha, \mathbf{true}\rangle}$$

$$R-\vee_1 \quad \frac{\langle v, \alpha, \mathbf{true}\rangle}{\langle v, \alpha \vee \beta, \mathbf{true}\rangle} \qquad \text{i} \qquad R-\vee_2 \quad \frac{\langle v, \beta, \mathbf{true}\rangle}{\langle v, \alpha \vee \beta, \mathbf{true}\rangle}$$

$$R-\vee_3 \quad \frac{\langle v, \alpha, \mathbf{false}\rangle \langle v, \beta, \mathbf{false}\rangle}{\langle v, \alpha \vee \beta, \mathbf{false}\rangle}$$

$$R-\wedge_1 \quad \frac{\langle v, \alpha, \mathbf{false}\rangle}{\langle v, \alpha \wedge \beta, \mathbf{false}\rangle}$$

$$R-\wedge_2 \quad \frac{\langle v, \beta, \mathbf{false}\rangle}{\langle v, \alpha \wedge \beta, \mathbf{false}\rangle}$$

$$R-\wedge_3 \quad \frac{\langle v, \alpha, \mathbf{true}\rangle \langle v, \beta, \mathbf{true}\rangle}{\langle v, \alpha \wedge \beta, \mathbf{true}\rangle}$$

$$R-p \quad \frac{\langle v, ex, w\rangle}{\langle v, x := ex, v'\rangle}$$

gdzie $v'(x) = w$ i $v'(z) = v(z)$ dla $z \neq x$

$$R-R \quad \frac{\langle v, (x_1 := \tau_1 \ldots x_{mi} := \tau_{mi}) M_i \alpha_i, w\rangle}{\langle v, \varrho_i(\tau_1, \ldots, \tau_{mi}), w\rangle}$$

dla $i = 1, \ldots, l$

$$R-F \quad \frac{\langle v, (x_1 := \tau_1 \ldots x_{nj} := \tau_{nj}) \mathrm{K}_j \tau_j, w\rangle}{\langle v, \varphi_j(\tau_1, \ldots, \tau_{nj}), w\rangle}$$

dla $j = 1, \ldots, k$

$$R-\mathrm{K}\tau \quad \frac{\langle v, \mathrm{K}, v'\rangle \langle v', \tau, w\rangle}{\langle v, \mathrm{K}\tau, w\rangle}$$

$$R-\mathrm{K}\alpha \quad \frac{\langle v, \mathrm{K}, v\rangle \langle v', \alpha, w\rangle}{\langle v, \mathrm{K}\alpha, w\rangle}$$

$$R-\textbf{if}-\textbf{true} \quad \frac{\langle v, \gamma, \textbf{true}\rangle \langle v, \mathrm{K}, v'\rangle}{\langle v, \textbf{if } \gamma \textbf{ then K else M fi}, v'\rangle}$$

$$R-\textbf{if}-\textbf{false} \quad \frac{\langle v, \gamma, \textbf{false}\rangle \langle v, \mathrm{M}, v'\rangle}{\langle v, \textbf{if } \gamma \textbf{ then K else M fi}, v'\rangle}$$

$$R-\textbf{begin} \quad \frac{\langle v, \mathrm{K}, v'\rangle \langle v', \mathrm{M}, v''\rangle}{\langle v, \textbf{begin K; M end}, v''\rangle}$$

$$R-\textbf{while}-\textbf{false} \quad \frac{\langle v, \gamma, \textbf{false}\rangle}{\langle v, \textbf{while } \gamma \textbf{ do K od}, v\rangle}$$

$$R-\textbf{while}-\textbf{true} \quad \frac{\langle v, \gamma, \textbf{true}\rangle \langle v, \mathrm{K}, v''\rangle \langle v'', \textbf{while } \gamma \textbf{ do K od}, v'\rangle}{\langle v, \textbf{while } \gamma \textbf{ do K od}, v'\rangle}$$

DEFINICJA 6.9

Powiemy, że triada $\langle v, ex, w\rangle$ ma obliczenie formalne w strukturze **A** z użyciem systemu procedur (*), $\mathbf{A} \underset{(*)}{\models} \langle v, ex, w\rangle$, wtedy i tylko wtedy, gdy istnieje ciąg triad $s_1, \ldots, s_n$ taki, że

(1) dla dowolnego $i \leqslant n$ albo s_i jest triadą elementarną, albo wnioskiem w pewnej regule obliczania formalnego $r \in \mathrm{ROF}$, w której przesłankami są pewne triady poprzedzające triadę s_i,

(2) $s_n = \langle v, ex, w\rangle$ ■

DEFINICJA 6.10

Jeżeli $\mathbf{A} \underset{(*)}{\models} \langle v, ex, w\rangle$, to powiemy, że w jest wartością wyrażenia ex przy wartościowaniu v w strukturze **A**, wyznaczoną za pomocą systemu procedur (*). ■

PRZYKŁAD 6.8

Niech L będzie językiem arytmetyki, w którym nie występuje symbol funkcyjny f, a **N** niech będzie strukturą liczb naturalnych. Niech system procedur Proc zawiera tylko jedną procedurę postaci

$$f(n) = \textbf{if } n = 0 \textbf{ then } z := 1 \textbf{ else } z := n \times f(n-1) \textbf{ fi } z$$

Przedstawiamy teraz obliczenie formalne dla triady $\langle v, f(n), 1\rangle$, gdzie $v(n) = 1$. Kroki obliczenia są numerowane kolejnymi liczbami naturalnymi. Każdy krok obliczenia zawiera komentarz wskazujący nazwę użytej reguły i numery przesłanek

1	$\left\langle v: \begin{array}{c\|c} z & n \\ \hline 5 & 1 \end{array}, n, 1 \right\rangle$	$\{e\}$
2	$\langle v, 0, 0 \rangle$	$\{e\}$
3	$\langle v, n = 0, \textbf{false} \rangle$	$\{1, 2, R-\varrho\}$
4	$\langle v, 1, 1 \rangle$	$\{e\}$
5	$\langle v, n-1, 0 \rangle$	$\{1, 4, R-\varphi\}$
6	$\left\langle v, n := n-1, v_1: \begin{array}{c\|c} z & n \\ \hline 5 & 0 \end{array} \right\rangle$	$\{5, R-p\}$
7	$\langle v_1, n, 0 \rangle$	$\{e\}$
8	$\langle v_1, n = 0, \textbf{true} \rangle$	$\{2, 7, R-\textbf{if}-\textbf{true}\}$
9	$\langle v_1, 1, 1 \rangle$	$\{e\}$
10	$\left\langle v_1, z := 1, v_2: \begin{array}{c\|c} z & n \\ \hline 1 & 0 \end{array} \right\rangle$	$\{8, R-p\}$
11	$\langle v_1, \textbf{if } n = 0 \textbf{ then } z := 1 \textbf{ else } z := n * f(n-1) \textbf{ fi}, v_2 \rangle$	$\{8, 10, R-\textbf{if}-\textbf{true}\}$
12	$\langle v_2, z, 1 \rangle$	$\{e\}$
13	$\langle v_1, \textbf{if } n = 0 \textbf{ then } z := 1 \textbf{ else } z := n * f(n-1) \textbf{ fi}, z, 1 \rangle$	$\{11, 12, R-\mathrm{K}\tau\}$
14	$\langle v, (n := n-1)(\textbf{if } n = 0 \textbf{ then } z := 1 \textbf{ else } z := n * f(n-1) \textbf{ fi } z), 1 \rangle$	$\{13, 6, R-\mathrm{K}\tau\}$
15	$\langle v, f(n-1), 1 \rangle$	$\{14, R-F\}$
16	$\langle v, n * f(n-1), 1 \rangle$	$\{1, 15, R-\varphi\}$
17	$\left\langle v, z := n * f(n-1), v': \begin{array}{c\|c} z & n \\ \hline 1 & 1 \end{array} \right\rangle$	$\{16, R-p\}$
18	$\langle v, \textbf{if } n = 0 \textbf{ then } z := 1 \textbf{ else } z := n * f(n-1) \textbf{ fi}, v' \rangle$	$\{3, 17, R-\textbf{if}-\textbf{false}\}$
19	$\langle v', z, 1 \rangle$	$\{e\}$
20	$\langle v, \textbf{if } n = 0 \textbf{ then } z := 1 \textbf{ else } z := n * f(n-1) \textbf{ fi } z, 1 \rangle$	$\{18, 19, R-K\tau\}$
21	$\langle v, f(1), 1 \rangle$	$\{20, R-F\}$

□

Obliczenia formalne wykazują wiele podobieństw do dowodów. Co prawda, w triadach występują nie tylko wyrażenia, ale i jednostki semantyczne: wartościowania i wartości wyrażeń. Jest jednak oczywiste, że można rozważać odpowiednio bogatszy język.

Obliczenia formalne mogą stanowić punkt wyjścia do „programowania w logice" innego niż w Prologu. Do pomyślenia jest zautomatyzowany sposób wyznaczania wartości polegający na poszukiwaniu (z nawrotami, ang. backtracking) obliczenia formalnego, gdy dane są wartościowanie i wyrażenie. W odróżnieniu od Prologu metoda oparta na wywodzeniu obliczeń formalnych umożliwia obliczanie wartościowań wynikowych dla programów.

LEMAT 6.10

Jeżeli dwie triady $\langle v, ex, w_1 \rangle$ i $\langle v, ex, w_2 \rangle$ mają obliczenia formalne, to $w_1 = w_2$. ■

Wynika stąd, że wartości przypisane wyrażeniom są określone w jednoznaczny sposób. Czy można oczekiwać, że obliczenia formalne będą wyznaczane w sposób automatyczny? Rzeczywiście, jeżeli jest dana para: wartościowanie v i wyrażenie ex, to trzeci element triady-wartość w- może być wyprowadzony tylko na jeden z kilku sposobów. Nietrudno będzie zapamiętać, jakiego wyboru dokonujemy i jakie możliwości pozostają do zbadania. Wydaje się, że w odróżnieniu od metod znanych wcześniej, ta metoda umożliwi ocenę czasu, w jakim powinny się zamknąć poszukiwania obliczenia formalnego.

Spójrzmy teraz na układ (*) procedur jako na układ aksjomatów. Załóżmy, że jest dana pewna struktura danych **A** dla języka L. Układ (*) wprowadza nowe funktory i predykaty, a więc nowy język L'. Czy określa on i to jednoznacznie znaczenie nowych symboli? Rozważmy następujące przykłady.

PRZYKŁAD 6.9

Niech **A** będzie dowolnie ustaloną strukturą danych dla języka L. Rozważmy formuły

$$\varphi(x, y) = \mathrm{K}\tau$$
$$\varrho(x, y, z) = \mathrm{M}\alpha$$

zakładamy, że ϱ i φ nie występują ani w $\mathrm{K}\tau$ ani w $\mathrm{M}\alpha$. Z poprzednich rozważań (por. p. 5.7) wynika, że formuły te jednoznacznie definiują rozszerzenie **A'** struktury **A**, które jest modelem tych formuł. **A'** jest strukturą danych dla języka L'. Oczywiście zdarza się, że nawt uwikłane (rekurencyjne) definicje mogą jednoznacznie określać rozszerzenie **A'**. W strukturze liczb naturalnych z poprzednikiem -1 i dodawaniem $+$, równość

$$h(x, y) = \textbf{if } y = 0 \textbf{ then } z := 0 \textbf{ else } z := h(x, y-1) + x \textbf{ fi } z$$

określa funkcję (jaką?). □

PRZYKŁAD 6.10

Zauważmy, że mogą istnieć sprzeczne układy procedur, np.

$$\varrho(x) \equiv \neg\varrho(x)$$

Taki układ nie daje możliwości zbudowania jakiegokolwiek obliczenia formalnego dla żadnej triady postaci $\langle v, \varrho(x), w\rangle$ dla żadnego w. Traktowany jako układ aksjomatów, jest sprzeczny. Czytelniku, zwróć uwagę, że własność sprzeczności może być ukryta głęboko w zbiorze równości i równoważności (*). □

PRZYKŁAD 6.11

Istnieją układy procedur niesprzeczne, mające więcej niż jeden model, np.

$$\varrho(x, y) = \varrho(x, y)$$

Przy każdej interpretacji predykatu ϱ formuła ta jest prawdziwa. Zwróćmy uwagę, że nie istnieje obliczenie formalne dla żadnej triady postaci $\langle v, \varrho(x, y), w\rangle$. □

Na zakończenie tego punktu zanotujmy kilka faktów, które podajemy bez dowodów.

(1) Niech **A** będzie strukturą danych dla języka L i niecz (*) oznacza układ procedur. Obliczenia formalne w strukturze **A** pozwalają zdefiniować pewne rozszerzenie **B** struktury **A**. **B** jest strukturą danych dla języka L rozszerzonego o funktory i predykaty wprowadzone przez układ procedur (*).

(2) Rozszerzenie **B** jest modelem układu procedur (*), o ile układ ten jest niesprzeczny.

(3) Każdy układ procedur (*) można w efektywny sposób przekształcić do niesprzecznego układu aksjomatów (**) dodając nowe predykaty wyrażające określoność terminów definiowanych w układzie (*).

(4) Rozszerzenie **B** wyznaczone za pomocą obliczeń formalnych jest najmniejszym modelem układu (**) zawierającym strukturę **A**.

Współbieżność 7

Programy współbieżne 7.1

Niech L będzie językiem pierwszego rzędu (por. p. 2.4). Klasę π deterministycznych while-programów nad L rozszerzymy obecnie o zbiór programów współbieżnych pozwalających na równoczesne wykonywanie wielu akcji. Rozszerzenie to otrzymamy przez dołączenie nowej konstrukcji programotwórczej

cobegin ... coend

DEFINICJA 7.1

Klasą programów współbieżnych π_3 nazywamy najmniejszy zbiór wyrażeń taki, że

(1) $\pi \subset \pi_3$;

(2) π_3 jest zbiorem zamkniętym ze względu na operacje programotwórcze składania, rozgałęziania, iteracji;

(3) jeżeli $K_1, ..., K_n$ są programami z klasy π_3, to wyrażenie **cobegin** $K_1 \| K_2 \| ... \| K_n$ **coend** jest programem z klasy π_3. ■

Instrukcje przypisania, rozgałęziania i iteracji będziemy nazywać instrukcjami prostymi w odróżnieniu od instrukcji złożonej **begin** ... **end** i instrukcji współbieżnej **cobegin** ... **coend**.

PRZYKŁAD 7.1

Następujące wyrażenie jest przykładem programu współbieżnego należącego do klasy π_3.

cobegin
  while true do *oblicz* (*e*); *put* (*e*, *s*) **od** ‖
  while true do if ¬ *empty* (*s*) **then** *e* := *get* (*s*) **fi**; *drukuj* (*e*) **od**
coend □

Składowe $K_1, ..., K_n$ programu współbieżnego

cobegin $K_1 \parallel K_2 \parallel ... \parallel K_n$ **coend**

będziemy nazywać procesami. Z każdym procesem będzie związany procesor odpowiedzialny za wykonanie procesu.

Zauważmy, że w definicji programu współbieżnego nie zakładamy, że zbiory zmiennych są rozłączne. Wprost przeciwnie: współpraca między procesami będzie polegała na wymianie informacji poprzez wspólną pamięć. W przypadku programu deterministycznego akcje programu były wykonywane sekwencyjnie, zgodnie ze strukturą syntaktyczną programu. Idea semantyki programów współbieżnych polega na dopuszczeniu do równoczesnego wykonania pewnych akcji. Jednak już proste przykłady nastręczają wiele trudności. Rozważmy program **cobegin** $x := 2 \parallel x := 12$ **coend**. Wyniku równoczesnego wykonania dwu instrukcji przypisania $x := 2$ i $x := 12$ nie można przewidzieć. Zależy to np. od szybkości działania poszczególnych procesorów. Nie można też stwierdzić, że wartością x będzie 2 lub 12. Podobna sytuacja zdarza się, gdy chcemy równocześnie zbadać test zależny np. od x i podstawić na x nową wartość.

W konsekwencji dochodzimy do wniosku, że pewne akcje nie powinny być wykonywane równocześnie, jeżeli chcemy mieć możność analizowania programu, badania jego własności.

DEFINICJA 7.2

Powiemy, że instrukcje proste K i M są w konflikcie wtedy i tylko wtedy, gdy K jest instrukcją przypisania $x := \tau$, a M przyjmuje jedną z następujących postaci: $x := w$, $z := w(x)$, **if** $\gamma(x)$ **then** ... **fi**, **while** $\gamma(x)$ **do** ... **od**, gdzie x, z są dowolnymi zmiennymi, a $w(x)$ i $\gamma(x)$ są wyrażeniami, w których występuje zmienna x.

Zbiór instrukcji deterministycznych prostych jest zbiorem konfliktowym wtedy i tylko wtedy, gdy istnieje w nim chociaż jedna para instrukcji w konflikcie. ■

PRZYKŁAD 7.2

Niech będą dane dwa zbiory instrukcji I_1 i I_2

$$I_1 = \{x := y + z, \textbf{while } x > 0 \textbf{ do } y := \tau \textbf{ od}\}$$
$$I_2 = \{x := y + z, \textbf{while } y < 0 \textbf{ do } x := \tau \textbf{ od}\}$$

I_1 jest zbiorem konfliktowym, a I_2 jest zbiorem niekonfliktowym. Zauważmy, że programy występujące po słowie **do** w instrukcji **while** nie mają wpływu na konfliktowość rozważanych zbiorów. □

DEFINICJA 7.3

Powiemy, że I jest maksymalnym niekonfliktowym podzbiorem zbioru J instrukcji prostych wtedy i tylko wtedy, gdy $I \subset J$ i dla dowolnego I' takiego, że $I \neq I'$ i $I \subset I' \subset J$, I' jest zbiorem konfliktowym. ■

PRZYKŁAD 7.3

Zbiór instrukcji przypisania $\{z := x * y,\ x := z+1,\ y := y * z\}$ ma dwa maksymalne niekonfliktowe podzbiory

$$\{z := x * y\} \quad \text{i} \quad \{x := z+1,\ y := y * z\}$$ □

Semantyka MAX 7.2

Semantyka programów współbieżnych, przedstawiona w tym punkcie, jest oparta na dwu założeniach:

(1) procesory wykonują akcje swojego procesu, jeśli tylko jest to możliwe, tzn. jeżeli wykonanie akcji nie prowadzi do konfliktu;

(2) procesory mogą się różnić szybkością wykonywania akcji.

Proces obliczania programu będzie się odbywać w kolejnych krokach, które będą polegały na przejściu od jednej konfiguracji do innej. Kroki te są wyznaczone kolejnymi zmianami stanu pamięci lub sterowania. W konsekwencji pierwszego założenia, w każdym kroku rozpoczyna obliczenie maksymalny niekonfliktowy zbiór instrukcji prostych. W konsekwencji drugiego założenia, akcje, których wykonanie rozpoczęło się równocześnie, niekoniecznie zostaną równocześnie zakończone. W związku z tym, w każdej konfiguracji każdy procesor może być w jednym z dwu stanów: albo jest gotowy do wykonania akcji, ale jej nie rozpoczął, albo jest w trakcie wykonywania akcji.

PRZYKŁAD 7.4

Rozważmy przykład, w którym trzy procesy są gotowe do wykonania odpowiednio instrukcji $x := 5$, $z := y$, $y := 3$. W myśl intuicyjnych zasad, które przedstawiliśmy powyżej, albo rozpocznie się wykonywanie instrukcji $x := 5$ i $z := y$, albo instrukcji $x := 5$ i $y := 3$, gdyż są to jedyne zbiory maksymalne instrukcji niekonfliktowych. Zatem są możliwe co najmniej dwa różne obliczenia rozważanego programu. Co więcej, ze względu na różny czas wykonywania poszczególnych instrukcji, nie jest jasne, która z instrukcji zakończy swoją akcję wcześniej, a w związku z tym nie jest jasne, jaka będzie wartość zmiennej z po wykonaniu wszystkich akcji. □

Obliczenie programu, w którym występuje instrukcja **cobegin**... nie jest zdeterminowane syntaktycznie. Nie można z góry przewidzieć, które akcje będą wykonane wcześniej, a które później. Zamiast jednego obliczenia, jak w przypadku while-programów, teraz, dla jednego stanu początkowego, możemy otrzymać zbiór różnych obliczeń. Obliczenia te mogą prowadzić do różnych wyników. Zatem program nie definiuje już funkcji częściowej, jak to było w przypadku while-programów, a pewną relację binarną w zbiorze stanów.

Zanim zdefiniujemy formalne pojęcie obliczenia (a zatem semantykę programów współbieżnych) wprowadzimy kilka pojęć pomocniczych.

Oznaczmy przez First (M) zbiór wystąpień instrukcji prostych programu M, stanowiących początki wszystkich „widocznych" procesów. W poniższej definicji będziemy dla prostoty identyfikować instrukcję z jej wystąpieniem, jednak zakładamy, że zbiór First (M) ma informację o kontekście, z którego dana instrukcja pochodzi.

First (**begin** K; M **end**) = First (K)
First (**while** γ **do** M **od**) = {**while** γ **do** M **od**}
First (**if** γ **then** K **else** M **fi**) = {**if** γ **then** K **else** M **fi**}
First (s) = $\{s\}$
First (**cobegin** $M_1 \| \ldots \| M_n$ **coend**) = $\bigcup_{i \leqslant n}$ First (M_i)

W tej i we wszystkich następnych definicjach litery K, M oznaczają programy ze zbioru π_3, γ jest formułą otwartą języka L, a s jest instrukcją przypisania.

Niech $J \subset$ First (M) i niech v będzie wartościowaniem w strukturze **A**. Oznaczmy przez Rest (v, J, M) program otrzymany z programu M przez wykonanie instrukcji należących do zbioru J. Funkcja Rest opisuje lokalne zachowanie procesów w ustalonej strukturze danych. Oczywiście zachowanie to zależy nie tylko od wartościowania, ale i od samej struktury **A**. Dla uproszczenia zapisu, w przedstawionej poniżej definicji funkcji Rest pominęliśmy parametr **A**.

(1) Jeżeli K jest instrukcją prostą i *non* K $\in J$, to

Rest (v, J, K) = K

(2) Jeżeli K jest instrukcją prostą i K $\in J$, to

(a) albo K jest instrukcja przypisania i wtedy

Rest (v, J, K) = $\emptyset$

(b) albo K = **if** γ **then** K **else** M **fi** i wtedy

$$Rest(v, J, K) = \begin{cases} M_1 & \text{gdy } \mathbf{A}, v \models \gamma \\ M_2 & \text{gdy } non\,\mathbf{A}, v \models \gamma \end{cases}$$

(c) albo K = **while** γ **do** M **od** i wtedy

$$Rest(v, J, \mathrm{K}) = \begin{cases} \emptyset \text{ gdy } non\,\mathbf{A}, v \models \gamma \\ \textbf{begin } \mathrm{M}; \textbf{ while } \gamma \textbf{ do } \mathrm{M} \textbf{ od end} \text{ gdy } \mathbf{A}, v \models \gamma \end{cases}$$

$$(3)\ Rest(v, J, \textbf{begin } \mathrm{M}_1; \mathrm{M}_2 \textbf{ end}) = \begin{cases} \textbf{begin } Rest(v, J, \mathrm{M}_1);\ \mathrm{M}_2 \textbf{ end} \\ \qquad \text{gdy } Rest(v, J, \mathrm{M}_1) \neq \emptyset \\ \mathrm{M}_2 \text{ w przeciwnym razie} \end{cases}$$

$$(4)\ Rest(v, J, \textbf{cobegin } \mathrm{M}_1 \parallel \ldots \parallel \mathrm{M}_n \textbf{ coend}) = \begin{cases} \emptyset \text{ jeśli } Rest(v, J, \mathrm{M}_i) = \emptyset \text{ dla } i \leqslant n \\ \textbf{cobegin } Rest(v, J, \mathrm{M}_{i1}) \\ \qquad \parallel \ldots \parallel Rest(v, J, \mathrm{M}_{ik}) \textbf{ coend} \\ \qquad \text{gdy } Rest(v, J, \mathrm{M}_i) \neq \emptyset \\ \qquad \text{dla } i \in \{i_1, \ldots, i_k\} \end{cases}$$

Na koniec oznaczmy przez $J_A(v)$ wartościowanie otrzymane z v przez wykonanie wszystkich instrukcji przypisania ze zbioru J (kolejność nie ma znaczenia).

DEFINICJA 7.4

Konfiguracją w strukturze **A** będziemy nazywać trójkę uporządkowaną $\langle v, J, \mathrm{M}\rangle$, gdzie M jest programem, v wartościowaniem w **A**, a J zbiorem instrukcji takich, że $J \subset \mathrm{First}(\mathrm{M})$. Zbiór J będziemy nazywać zbiorem instrukcji aktywnych programu M. ■

DEFINICJA 7.5

Relacją bezpośredniego następstwa w strukturze **A** nazywamy relację binarną $\rightarrow$ w zbiorze wszystkich konfiguracji w **A** taką, że dla dowolnych v, M

(1) jeżeli I jest maksymalnym niekonfliktowym podzbiorem zbioru First(M), to dla dowolnego J, $J \subset I$ i $J \neq \emptyset$

$$\langle v, I, \mathrm{M}\rangle \rightarrow \langle J_A(v), I - J, Rest(v, J, \mathrm{M})\rangle$$

(2) jeżeli I nie jest maksymalnym niekonfliktowym podzbiorem zbioru First(M), to

$$\langle v, I, \mathrm{M}\rangle \rightarrow \langle v, I', \mathrm{M}\rangle$$

dla dowolnego zbioru I' będącego maksymalnym niekonfliktowym rozszerzeniem I w First(M). ■

Oznaczmy przez $\twoheadrightarrow$ przechodnie domknięcie relacji bezpośredniego następstwa $\rightarrow$.

DEFINICJA 7.6

Obliczeniem w semantyce MAX programu M w strukturze **A** przy wartościowaniu początkowym v nazywamy maksymalny dobrze uporządkowany w sensie relacji $-*\rightarrow$ podzbiór zbioru wszystkich konfiguracji w **A**, którego elementem pierwszym jest trójka $\langle v, \emptyset, \mathrm{M}\rangle$. ■

PRZYKŁAD 7.5

Rozważmy program M następującej postaci:

cobegin $D := a*d;\quad D := D-c*b \,\|$
$x := e*d;\quad x := x-b*f;\quad x := x/D \,\|$
$y := a*f;\quad y := y-c*e;\quad y := y/D$
coend,

którego zadaniem jest rozwiązanie układu równań

$$a*x+b*y = e$$
$$c*x+d*y = f$$

Następujący ciąg konfiguracji jest jednym z możliwych obliczeń tego programu w strukturze liczb rzeczywistych **R** przy dowolnym wartościowaniu v takim, że parametry a, b, c, d, e, f są odpowiednio równe 1, 2, 3, 4, 5, 6. Dla uproszczenia w każdej konfiguracji będziemy podawać tylko wartości zmiennych x, y, D, a elementy zbioru instrukcji aktywnych oznaczymy symbolem $\circ$ w tekście programu.

$\langle v, \mathrm{M}\rangle$

$\langle v,$ **cobegin** $\circ D := a*d;\ D := D-c*b \,\|$
$\circ x := e*d;\ x := x-b*f;\quad x := x/D \,\|$
$\circ y := a*f;\ y := y-c*e;\quad y := y/D$ **coend**$\rangle$

$\langle \begin{array}{c|c|c} x & y & D \\ \hline - & - & 4 \end{array}$, **cobegin** $D := D-c*b \,\|$
$\circ x := e*d;\ x := x-b*f;\ x := x/D \,\|$
$\circ y := a*f;\ y := y-c*e;\ y := y/D$ **coend**$\rangle$

$\langle \begin{array}{c|c|c} x & y & D \\ \hline - & - & 4 \end{array}$, **cobegin** $\circ D := D-c*b \,\|$
$\circ x := e*d;\ x := x-b*f;\ x := x/D \,\|$
$\circ y := a*f;\ y := y-c*e;\ y := y/D$ **coend**$\rangle$

$\langle \begin{array}{c|c|c} x & y & D \\ \hline 20 & 6 & -2 \end{array}$, **cobegin** $x := x-b*f;\ x := x/D \,\|$
$y := y-c*e;\ y := y/D$ **coend**$\rangle$

$\langle \begin{array}{c|c|c} x & y & D \\ \hline 20 & 6 & -2 \end{array}$, **cobegin** $\circ x := x-b*f;\ x := x/D \,\|$
$\circ y := y-c*e;\ y := y/D$ **coend**$\rangle$

$$\left\langle \frac{x \mid y \mid D}{8 \mid -9 \mid -2}, \textbf{cobegin } x := x/D \parallel y := y/D \textbf{ coend} \right\rangle$$

$$\left\langle \frac{x \mid y \mid D}{8 \mid -9 \mid -2}, \textbf{cobegin } \circ\, x := x/D \parallel \circ\, y := y/D \textbf{ coend} \right\rangle$$

$$\left\langle \frac{x \mid y \mid D}{-4 \mid 4{,}5 \mid -2}, \right\rangle$$

□

Inne koncepcje semantyki 7.3

Strategia semantyki MAX jest oparta na dwu niedeterministycznych wyborach:

(1) na wyborze niedeterministycznego niekonfliktowego zbioru instrukcji, które będą aktywne w rozważanym kroku;

(2) na wyborze zbioru instrukcji, które zakończą akcję w rozważanym kroku.

UWAGA

Słowo wybór może sugerować, że ktoś lub coś dokonuje takiego wyboru, że istnieje centralny dyrygent wykonawców (procesorów). Otóż nie zakładamy istnienia takiego dyrygenta. Słowo wybór pojawia się tutaj tylko dla wygody, by uniknąć dłuższych omówień. Semantyka, którą tu prezentujemy jest tylko matematycznym modelem zjawisk zachodzących w świecie fizycznie działających urządzeń. Należy brać pod uwagę zwłaszcza wyścigi sygnałów wysyłanych przez procesory (w imieniu procesów) oraz niejednakowe prędkości procesorów. ■

Oba te niedeterministyczne wybory są istotne. Jeżeli zmodyfikujemy semantykę MAX odrzucając jeden z nich, to uzyskana semantyka będzie miała własności istotnie inne niż MAX.

W dalszym ciągu tego rozdziału będziemy rozważać kilka typów semantyk zdefiniowanych za pomocą pojęcia obliczenia, w których lokalne zachowanie procesów jest opisane standardowo, np. przez funkcję Rest (por. 7.2). Różnica między rozważanymi semantykami będzie polegać tylko na różnych zasadach globalnego współdziałania.

DEFINICJA 7.7

Powiemy, że semantyka S_2 jest rozszerzeniem semantyki S_1 i piszemy $S_1 \subset S_2$ wtedy i tylko wtedy, gdy dla dowolnego M, relacja wejścia-wyjścia przypi-

sana programowi M w semantyce S_1 jest zawarta w relacji wejścia-wyjścia programu M w semantyce S_2 (por. przykłady 7.6, 7.7). Jeżeli $non\, S_1 \subset S_2$ lub $non\, S_2 \subset S_1$, to powiemy, że semantyki S_1 i S_2 są istotnie różne $S_1 \neq S_2$. ■

DEFINICJA 7.8

Relacją bezpośredniego następstwa w semantyce SMAX będziemy nazywać relację binarną $\rightarrow$ w zbiorze wszystkich konfiguracji taką, że dla dowolnej struktury danych **A**, dowolnego wartościowania v i dowolnego maksymalnego niekonfliktowego zbioru I zawartego w First(M)

$$\langle v, \emptyset, \mathrm{M}\rangle \rightarrow \langle I_{\mathrm{A}}(v), \emptyset, \mathrm{Rest}(v, I, \mathrm{M})\rangle$$ ■

Aby pokazać różnicę między semantyką MAX i SMAX rozważymy następujący prosty przykład. (Ponieważ w każdej konfiguracji obliczenia w semantyce SMAX drugim elementem jest zbiór pusty, będziemy ten zbiór konsekwentnie pomijać).

PRZYKŁAD 7.6

Niech M oznacza program

cobegin $x := 1;\ x := 2;\ x := y \,\|\, y := 3;\ y := 4$ **coend**

Niech v będzie dowolnym wartościowaniem w strukturze liczb rzeczywistych **R**. Jedyne możliwe obliczenie w semantyce SMAX ma następującą postać:

$\langle v,$ **cobegin** $x := 1;\ x := 2;\ x := y \,\|\, y := 3;\ y := 4$ **coend**$\rangle$

$\langle \begin{array}{c|c} x & y \\ \hline 1 & 3 \end{array},$ **cobegin** $x := 2;\ x := y \,\|\, y := 4$ **coend**$\rangle$

$\langle \begin{array}{c|c} x & y \\ \hline 2 & 4 \end{array},$ **cobegin** $x := y$ **coend**$\rangle$

$\langle \begin{array}{c|c} x & y \\ \hline 4 & 4 \end{array}, \rangle$

W semantyce MAX program M ma kilka możliwych obliczeń. Jedno z nich wygląda następująco:

$\langle v,$ **cobegin** $x := 1;\ x := 2;\ x := y \,\|\, y := 3;\ y := 4$ **coend**$\rangle$
$\langle v,$ **cobegin** $\circ\, x := 1;\ x := 2;\ x := y \,\|\, \circ\, y := 3;\ y := 4$ **coend**$\rangle$

$\langle \begin{array}{c|c} x & y \\ \hline 1 & v(y) \end{array}$, **cobegin** $x := 2$; $x := y \| \circ y := 3$; $y := 4$ **coend**$\rangle$

$\langle \begin{array}{c|c} x & y \\ \hline 1 & v(y) \end{array}$, **cobegin** $\circ\, x := 2$; $x := y \| \circ y := 3$; $y := 4$ **coend**$\rangle$

$\langle \begin{array}{c|c} x & y \\ \hline 2 & 3 \end{array}$, **cobegin** $x := y \| y := 4$ **coend**$\rangle$

$\langle \begin{array}{c|c} x & y \\ \hline 2 & 3 \end{array}$, **cobegin** $\circ\, x := y \| y := 4$ **coend**$\rangle$

$\langle \begin{array}{c|c} x & y \\ \hline 3 & 3 \end{array}$, **cobegin** $y := 4$ **coend**$\rangle$

$\langle \begin{array}{c|c} x & y \\ \hline 3 & 3 \end{array}$, **cobegin** $\circ\, y := 4$ **coend**$\rangle$

$\langle \begin{array}{c|c} x & y \\ \hline 3 & 4 \end{array}$, $\rangle$

Aby podkreślić różnicę w wykonywaniu programu M w zależności od wybranej semantyki, zauważmy następujące algorytmiczne własności. W semantyce SMAX wynik obliczenia programu M ma własność $(x = y)$, a w semantyce MAX można uzyskać wynik spełniający formułę $(x \neq y)$. □

Z tych rozważań można wyciągnąć następujący wniosek:

$$\text{SMAX} \neq \text{MAX}$$

Jednocześnie, porównując relacje bezpośredniego następstwa w obu semantykach, zauważymy, że

$$\text{SMAX} \subset \text{MAX}$$

DEFINICJA 7.9

Relacją bezpośredniego następstwa w semantyce ARB nazywamy relację binarną $\rightarrow$ w zbiorze konfiguracji taką, że dla dowolnej struktury A, dowolnego w niej wartościowania v, dowolnego programu M i dowolnego niekonfliktowego zbioru $J \subset \text{First}(\text{M})$

$$\langle v, \emptyset, \text{M}\rangle \rightarrow \langle J_{\text{A}}(v), \emptyset, \text{Rest}(v, J, \text{M})\rangle$$ ■

Jako prostą konsekwencję tej definicji otrzymujemy

$$\text{SMAX} \subset \text{MAX} \subset \text{ARB}$$

Co więcej, pokażemy, że ARB jest różna od obu poprzednio omówionych semantyk. Dla dowodu rozważmy następujący przykład.

PRZYKŁAD 7.7

Niech p, q będą zmiennymi zdaniowymi, a M następującym programem:

cobegin $p := \mathbf{false} \parallel q := \mathbf{true}$; $p := q$ **coend**

W dowolnej strukturze danych w semantyce ARB są możliwe między innymi następujące obliczenia:

Θ_1:

$\langle v,$ **cobegin** $p := \mathbf{false} \parallel q := \mathbf{true}$; $p := q$ **coend**$\rangle$

$\langle \begin{array}{c|c} p & q \\ \hline 0 & 1 \end{array},$ **cobegin** $p := q$ **coend**$\rangle$

$\langle \begin{array}{c|c} p & q \\ \hline 1 & 1 \end{array}, \rangle$

Θ_2;

$\langle v,$ **cobegin** $p := \mathbf{false} \parallel q := \mathbf{true}$; $p := q$ **coend**$\rangle$

$\langle \begin{array}{c|c} p & q \\ \hline v(p) & 1 \end{array},$ **cobegin** $p := \mathbf{false} \parallel p := q$ **coend**$\rangle$

$\langle \begin{array}{c|c} p & q \\ \hline 1 & 1 \end{array},$ **cobegin** $p := \mathbf{false}$ **coend**$\rangle$

$\langle \begin{array}{c|c} p & q \\ \hline 1 & 1 \end{array}, \rangle$

Poniżej przedstawiamy wszystkie możliwe obliczenia programu M w semantyce MAX.

Θ'_1:

$\langle v,$ **cobegin** $p := \mathbf{false} \parallel q := \mathbf{true}$; $p := q$ **coend**$\rangle$

$\langle v,$ **cobegin** $\circ\, p := \mathbf{false} \parallel \circ\, q := \mathbf{true}$; $p := q$ **coend**$\rangle$

$\langle \begin{array}{c|c} p & q \\ \hline 0 & v(q) \end{array},$ **cobegin** $\circ\, q := \mathbf{true}$; $p := q$ **coend**$\rangle$

$\langle \begin{array}{c|c} p & q \\ \hline 0 & 1 \end{array},$ **cobegin** $p := q$ **coend**$\rangle$

$\langle \begin{array}{c|c} p & q \\ \hline 0 & 1 \end{array},$ **cobegin** $\circ\, p := q$ **coend**$\rangle$

$\langle \begin{array}{c|c} p & q \\ \hline 1 & 1 \end{array}, \rangle$

Θ'_2:

$\langle v,$ **cobegin** $p := \mathbf{false} \parallel q := \mathbf{true}$; $p := q$ **coend**$\rangle$

$\langle v,$ **cobegin** $\circ\, p := \mathbf{false} \parallel \circ\, q := \mathbf{true}$; $p := q$ **coend**$\rangle$

$$\left\langle \begin{array}{c|c} p & q \\ \hline v(p) & 1 \end{array}, \textbf{cobegin} \circ p :\!- \textbf{false} \parallel p := q \textbf{ coend} \right\rangle$$

$$\left\langle \begin{array}{c|c} p & q \\ \hline 0 & 1 \end{array}, \textbf{cobegin } p := q \textbf{ coend} \right\rangle$$

$$\left\langle \begin{array}{c|c} p & q \\ \hline 0 & 1 \end{array}, \textbf{cobegin} \circ p := q \textbf{ coend} \right\rangle$$

$$\left\langle \begin{array}{c|c} p & q \\ \hline 1 & 1 \end{array}, \right\rangle$$

θ'_3:

$\langle v, \textbf{cobegin } p := \textbf{false} \parallel q := \textbf{true}; p := q \textbf{ coend} \rangle$

$\langle v, \textbf{cobegin} \circ p := \textbf{false} \parallel \circ q := \textbf{true}; p := q \textbf{ coend} \rangle$

$$\left\langle \begin{array}{c|c} p & q \\ \hline 0 & 1 \end{array}, \textbf{cobegin } p := q \textbf{ coend} \right\rangle$$

$$\left\langle \begin{array}{c|c} p & q \\ \hline 0 & 1 \end{array}, \textbf{cobegin} \circ p := q \textbf{ coend} \right\rangle$$

$$\left\langle \begin{array}{c|c} p & q \\ \hline 1 & 1 \end{array}, \right\rangle$$

Podsumowując zauważmy, że w semantyce ARB jest możliwe obliczenie, w wyniku którego zmienna p ma wartość prawda i możliwe jest obliczenie, w wyniku którego p ma wartość fałsz. W semantyce MAX wszystkie możliwe obliczenia dają w wyniku wartościowanie, w którym p ma wartość prawda. □

We wszystkich semantykach, o których była do tej pory mowa, ilość użytych procesów była dowolna. Jednak w praktyce, fizyczne możliwości posiadanego sprzętu narzucają ograniczenie na liczbę, biorących udział w obliczeniu programu współbieżnego, procesorów. W dalszym ciągu tego punktu przedyskutujemy konsekwencje takiego ograniczenia.

Rozważmy semantyki MAX(n), SMAX(n) i ARB(n) dla $n \in N$, które różnią się odpowiednio od MAX, SMAX i ARB jedynie tym, że w każdym kroku obliczenia może być aktywnych co najwyżej n procesorów.

Z przykładów rozważanych poprzednio możemy wyciągnąć następujący wniosek, dla dowolnego n naturalnego:

$$\text{SMAX}(n) \neq \text{MAX}(n) \neq \text{ARB}(n)$$

oraz

$$\text{SMAX}(n) \subset \text{MAX}(n) \subset \text{ARB}(n)$$

PRZYKŁAD 7.8

Niech p będzie zmienną zdaniową, a $\mathbf{R}$ strukturą liczb rzeczywistych. Porównamy zachowanie się następującego programu M w różnych semantykach i przy różnych ograniczeniach na liczbę procesorów.

M: **cobegin**
$p := \mathbf{false} \,\|$
$x_1 := x_1 + 1$; **while** p **do** $x_1 := x_1 + 1$ **od** $\|$
. . . . . .
$x_n := x_n + 1$; **while** p **do** $x_n := x_n + 1$ **od**
coend

Niech v będzie wartościowaniem takim, że $v(p) = 1$ i $v(x_i) = 0$ dla $i \leqslant n$.

SMAX$(n+1)$

W semantyce SMAX z $n+1$ procesorami program M ma tylko jedno obliczenie:

$\langle v, \mathrm{M} \rangle$

$\langle \begin{array}{c|c|c|c} p & x_1 & \ldots & x_n \\ \hline 0 & 1 & & 1 \end{array},$ **cobegin** **while** p **do** $x_1 := x_1 + 1$ **od** $\|$
. . . . . .
while p **do** $x_n := x_n + 1$ **od coend** $\rangle$

$\langle \begin{array}{c|c|c|c} p & x_1 & \ldots & x_n \\ \hline 0 & 1 & & 1 \end{array}, \rangle$

SMAX(n)

Jeżeli ograniczymy ilość procesorów do n, to przy tej samej strategii otrzymamy wiele różnych obliczeń. Jedno z nich może wyglądać następująco:

$\langle v, \mathrm{M} \rangle$

$\langle \begin{array}{c|c|c|c} p & x_1 & \ldots & x_n \\ \hline 1 & 1 & & 1 \end{array},$ **cobegin** $p := \mathbf{false} \,\|$
while p **do** $x_1 := x_1 + 1$ **od** $\|$
. . . . . .
while p **do** $x_n := x_n + 1$ **od coend** $\rangle$

$\langle \begin{array}{c|c|c|c} p & x_1 & \ldots & x_n \\ \hline 1 & 1 & & 1 \end{array},$ **cobegin**
$p := \mathbf{false} \,\|$
$x_1 := x_1 + 1$; **while** p **do** $x_1 := x_1 + 1$ **od** $\|$
. . . . . .
$x_n := x_n + 1$; **while** p **do** $x_n := x_n + 1$ **od coend** $\rangle$

$$\left\langle \begin{array}{c|c|c|c} p & x_1 & \cdots & x_n \\ \hline 1 & 2 & & 2 \end{array}, \right.$$ **cobegin** $p := \mathbf{false} \parallel$
while p **do** $x_1 := x_1 + 1$ **od** $\parallel$
. . . . . .
while p **do** $x_n := x_n + 1$ **od coend**$\rangle$

$$\left\langle \begin{array}{c|c|c|c} p & x_1 & \cdots & x_n \\ \hline 0 & 2 & & 2 \end{array}, \right\rangle$$

Zauważmy, że każde obliczenie w semantyce SMAX(n) prowadzi do wartościowania, w którym wszystkie zmienne x_i mają taką samą wartość i, w zależności od długości obliczenia mogą to być dowolne liczby naturalne.

MAX($n+1$)

Jedyne obliczenie w tej semantyce różni się nieistotnie od obliczenia w semantyce SMAX($n+1$).

$\langle v, \mathrm{M} \rangle$
$\langle v,$ **cobegin**
$\circ\, p := \mathbf{false} \parallel$
$\circ\, x_1 := x_1 + 1;$ **while** p **do** $x_1 := x_1 + 1$ **od** $\parallel$
. . . . . .
$\circ\, x_n := x_n + 1;$ **while** p **do** $x_n := x_n + 1$ **od coend**$\rangle$

$$\left\langle \begin{array}{c|c|c|c} p & x_1 & \cdots & x_n \\ \hline 0 & 1 & & 1 \end{array}, \right.$$ **cobegin while** p **do** $x_1 := x_1 + 1$ **od** $\parallel$
. . . . . .
while p **do** $x_n := x_n + 1$ **od coend**$\rangle$

$$\left\langle \begin{array}{c|c|c|c} p & x_1 & \cdots & x_n \\ \hline 0 & 1 & & 1 \end{array}, \right.$$ **cobegin** $\circ$ **while** p **do** $x_1 := x_1 + 1$ **od** $\parallel$
. . . . . .
$\circ$ **while** p **do** $x_n := x_n + 1$ **od coend**$\rangle$

$$\left\langle \begin{array}{c|c|c|c} p & x_1 & \cdots & x_n \\ \hline 0 & 1 & & 1 \end{array}, \right\rangle$$

MAX(n)

Jedno z możliwych obliczeń programu M w semantyce MAX z n procesorami może mieć następującą postać:

$\langle v, \mathrm{M} \rangle$
$\langle v,$ **cobegin**
$p := \mathbf{false} \parallel$
$\circ\, x_1 := x_1 + 1;$ **while** p **do** $x_1 := x_1 + 1$ **od** $\parallel$
. . . . . .
$\circ\, x_n := x_n + 1;$ **while** p **do** $x_n := x_n + 1$ **od coend**$\rangle$

$\langle \begin{array}{c|c|c|c|c} p & x_1 & x_2 & \ldots & x_n \\ \hline 1 & 0 & 1 & & 1 \end{array}$, **cobegin**
$p := \mathbf{false} \parallel$
$\circ\, x_1 := x_1 + 1$; **while** p **do** $x_1 := x_1 + 1$ **od** $\parallel$
while p **do** $x_2 := x_2 + 1$ **od** $\parallel$
. . . . . .
while p **do** $x_n := x_n + 1$ **od coend**$\rangle$

$\langle \begin{array}{c|c|c|c|c} p & x_1 & x_2 & \ldots & x_n \\ \hline 1 & 0 & 1 & & 1 \end{array}$, **cobegin**
$p := \mathbf{false} \parallel$
$\circ\, x_1 := x_1 + 1$; **while** p **do** $x_1 := x_1 + 1$ **od** $\parallel$
$\circ$ **while** p **do** $x_2 := x_2 + 1$ **od** $\parallel$
. . . . . .
$\circ$ **while** p **do** $x_n := x_n + 1$ **od coend**$\rangle$

$\langle \begin{array}{c|c|c|c|c} p & x_1 & x_2 & \ldots & x_n \\ \hline 1 & 0 & 1 & & 1 \end{array}$, **cobegin**
$p := \mathbf{false} \parallel$
$\circ\, x_1 := x_1 + 1$; **while** p **do** $x_1 := x_1 + 1$ **od** $\parallel$
$x_2 := x_2 + 1$; **while** p **do** $x_2 := x_2 + 1$ **od** $\parallel$
. . . . . .
$x_n := x_n + 1$; **while** p **do** $x_n := x_n + 1$ **od**
coend$\rangle$

$\langle \begin{array}{c|c|c|c|c} p & x_1 & x_2 & \ldots & x_n \\ \hline 1 & 0 & 1 & & 1 \end{array}$, **cobegin**
$p := \mathbf{false} \parallel$
$\circ\, x_1 := x_1 + 1$; **while** p **do** $x_1 := x_1 + 1$ **od** $\parallel$
$\circ\, x_2 := x_2 + 1$; **while** p **do** $x_2 := x_2 + 1$ **od** $\parallel$
. . . . . .
$\circ\, x_n := x_n + 1$; **while** p **do** $x_n := x_n + 1$ **od**
coend$\rangle$

$\langle \begin{array}{c|c|c|c|c} p & x_1 & x_2 & \ldots & x_n \\ \hline 1 & 1 & 2 & & 2 \end{array}$, **cobegin**
$p := \mathbf{false} \parallel$
while p **do** $x_1 := x_1 + 1$ **od** $\parallel$
while p **do** $x_2 := x_2 + 1$ **od** $\parallel$
. . . . . .
while p **do** $x_n := x_n + 1$ **od**
coend$\rangle$

$\langle \begin{array}{c|c|c|c|c} p & x_1 & x_2 & \ldots & x_n \\ \hline 1 & 1 & 2 & & 2 \end{array}$, **cobegin**
$\circ\, p := \mathbf{false} \parallel$
while p **do** $x_1 := x_1 + 1$ **od** $\parallel$
while p **do** $x_2 := x_2 + 1$ **od** $\parallel$

. . . . . .
while p **do** $x_n := x_n + 1$ **od**
coend⟩

⟨ $\begin{array}{c|c|c|c|c} p & x_1 & x_2 & \ldots & x_n \\ \hline 0 & 1 & 2 & & 2 \end{array}$, **cobegin**
while p **do** $x_1 := x_1 + 1$ **od** ‖
while p **do** $x_2 := x_2 + 1$ **od** ‖
. . . . . .
while p **do** $x_n := x_n + 1$ **od**
coend⟩

⟨ $\begin{array}{c|c|c|c|c} p & x_1 & x_2 & \ldots & x_n \\ \hline 0 & 1 & 2 & & 2 \end{array}$, **cobegin**
∘**while** p **do** $x_1 := x_1 + 1$ **od** |
∘**while** p **do** $x_2 := x_2 + 1$ **od** |
. . . . . .
∘**while** p **do** $x_n := x_n + 1$ **od**
coend⟩

⟨ $\begin{array}{c|c|c|c|c} p & x_1 & x_2 & \ldots & x_n \\ \hline 0 & 1 & 2 & & 2 \end{array}$, ⟩ . □

Na zakończenie zauważmy następujące proste fakty zachodzące dla dowolnej liczby naturalnej n

$\mathrm{ARB}(n) \subset \mathrm{ARB}(n+1)$
$non\,\mathrm{SMAX}(n) \subset \mathrm{SMAX}(n+1)$
$non\,\mathrm{MAX}(n) \subset \mathrm{MAX}(n+1)$

W powyższych rozważaniach zawarliśmy definicje różnych semantyk dla tej samej klasy programów π_2. Wykazaliśmy, że semantyki te nie są równoważne. Czy można twierdzić, że istnieje jedna, najlepsza semantyka programów współbieżnych? Wydaje się nam, że nie. Tym którzy uważają, że różnice między przytoczonymi semantykami są nieistotne, dedykujemy rozważania zawarte w następnym punkcie.

Semantyki MAX i ARB w sieciach Petriego 7.4

W tym punkcie przedyskutujemy różnice między semantyką MAX i ARB na gruncie sieci Petriego, by pokazać, że różnica między tymi semantykami jest głębsza, niż to by wynikało z podanych poprzednio przykładów.

DEFINICJA 7.9

Siecią Petriego będziemy nazywać system

$$PN = \langle PL, TR, BACK, FOR, m_0 \rangle$$

w którym PL, TR są odpowiednio skończonymi zbiorami miejsc i przejść, $PL \cap TR = \emptyset$, a BACK, FOR, m_0 są następującymi funkcjami:

$$BACK: PL \times TR \to N$$
$$FOR: TR \times PL \to N$$
$$m_0: PL \to N$$

Funkcję m_0 będziemy nazywać markowaniem początkowym sieci. ■

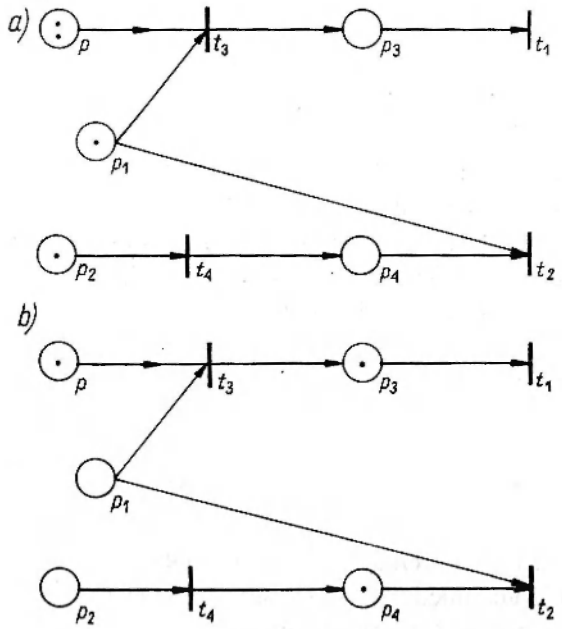
Sieć Petriego zwykle jest przedstawiana w postaci *grafu dwudzielnego*, którego zbiorem wierzchołków jest zbiór $TR \cup PL$, a zbiorem krawędzi zbiór

$$\{(p, t): BACK(p, t) > 0\} \cup \{(t, p): FOR(t, p) > 0\}$$

Wartości funkcji BACK i FOR są etykietami krawędzi. Dla uproszczenia rysunków, etykiety równe 1 są pomijane.

PRZYKŁAD 7.9

Graf przedstawiony na rys. 7.1a jest siecią Petriego, w której $\{p, p_1, p_2, p_3, p_4\}$ jest zbiorem miejsc $\{t_1, t_2, t_3, t_4\}$ jest zbiorem przejść. Markowanie



RYS. 7.1

początkowe jest przedstawione za pomocą kropek przyporządkowanych miejscom. □

Przejścia w sieciach Petriego odpowiadają akcjom, a miejsca stanom. Funkcje BACK i FOR narzucają warunki, w jakich działają przejścia, a markowanie początkowe sieci ustala stan, w jakim znajduje się sieć. Działanie całej sieci Petriego polega na kolejnych zmianach markowania wyznaczonych przez kolejne „odpalanie" pewnych przejść.

DEFINICJA 7.10

Przejścia $t_1, ..., t_n$ w sieci Petriego PN = ⟨PL, TR, BACK, FOR, m_0⟩ mogą zostać odpalone równocześnie przy markowaniu m wtedy i tylko wtedy, gdy dla każdego $p \in PL$

$$m(p) \geqslant \sum_{i \leqslant n} \mathrm{BACK}(p, t_i)$$ ■

Powiemy, że zbiór przejść $\{t_1, ..., t_n\}$ sieci PN jest zbiorem konfliktowym przy markowaniu m wtedy i tylko wtedy, gdy nie można równocześnie odpalić przejść $t_1, ..., t_n$.

PRZYKŁAD 7.10

Rozważmy sieć przedstawioną na rys. 7.1a. Przy zadanym, jak na rysunku markowaniu istnieją trzy zbiory niekonfliktowe $\{t_3\}$, $\{t_4\}$, $\{t_3, t_4\}$. □

DEFINICJA 7.11

Markowanie $m' : PL \to N$ jest wynikiem równoczesnego odpalenia przejść $t_1, ..., t_n$ przy markowaniu m w sieci PN wtedy i tylko wtedy, gdy

(1) przejścia $t_1, ..., t_n$ mogą być odpalone równocześnie;

(2) dla każdego $p \in PL$

$$m'(p) = m(p) - \sum_{i \leqslant n} \mathrm{BACK}(p, t_i) + \sum_{i \leqslant n} \mathrm{FOR}(t_i, p)$$ ■

PRZYKŁAD 7.11

W wyniku równoczesnego odpalenia przejść t_3 i t_4 nowe markowanie sieci z rys. 7.1a wygląda tak, jak na rys. 7.1b. Jedynym niekonfliktowym zbiorem przejść jest teraz zbiór $\{t_4\}$. □

DEFINICJA 7.12

Ciąg par $\langle m_0, c_0\rangle$, $\langle m_1, c_1\rangle$, $\langle m_2, c_2\rangle$... będziemy nazywać obliczeniem w sieci Petriego PN = $\langle$PL, TR, BACK, FOR, $m_0\rangle$ w semantyce ARB (MAX) wtedy i tylko wtedy, gdy dla każdego j

(1) c_j jest dowolnym (maksymalnym) zbiorem niekonfliktowym przejść przy markowaniu m_j;

(2) m_{j+1} jest wynikiem równoczesnego odpalenia przejść ze zbioru c_j przy markowaniu m_j. ■

PRZYKŁAD 7.12

Kontynuując rozważania związane z siecią PN (rys. 7.1a), zaobserwujmy następujące obliczenia w semantyce MAX i w semantyce ARB:

MAX:

$$\left\langle \begin{array}{c|c|c|c|c} p & p_1 & p_2 & p_3 & p_4 \\ \hline 2 & 1 & 1 & 0 & 0 \end{array}, \{t_3, t_4\} \right\rangle$$

$$\left\langle \begin{array}{c|c|c|c|c} p & p_1 & p_2 & p_3 & p_4 \\ \hline 1 & 0 & 0 & 1 & 1 \end{array}, \{t_1\} \right\rangle$$

$$\left\langle \begin{array}{c|c|c|c|c} p & p_1 & p_2 & p_3 & p_4 \\ \hline 1 & 0 & 0 & 0 & 1 \end{array}, \right\rangle$$

ARB:

$$\left\langle \begin{array}{c|c|c|c|c} p & p_1 & p_2 & p_3 & p_4 \\ \hline 2 & 1 & 1 & 0 & 0 \end{array}, \{t_4\} \right\rangle$$

$$\left\langle \begin{array}{c|c|c|c|c} p & p_1 & p_2 & p_3 & p_4 \\ \hline 2 & 1 & 0 & 0 & 1 \end{array}, \{t_2\} \right\rangle$$

$$\left\langle \begin{array}{c|c|c|c|c} p & p_1 & p_2 & p_3 & p_4 \\ \hline 2 & 0 & 0 & 0 & 0 \end{array}, \right\rangle$$

□

Rozważmy jeszcze raz sieć z rysunku 7.1a i przeanalizujmy jej działanie w zależności od zawartości miejsca p. W semantyce MAX zachowanie sieci można opisać instrukcją warunkową

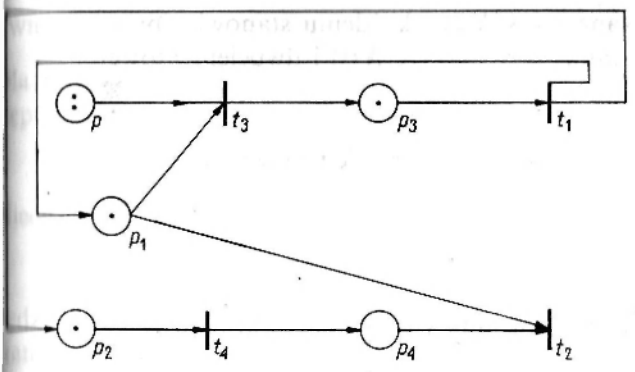
if $p > 0$ **then** t_1 **else** t_2 **fi**.

W semantyce ARB zachowanie tej sieci jest nieco inne. Jeżeli $p > 0$, to zostanie odpalone albo przejście t_1, albo przejście t_2, w przeciwnym razie t_2.

Rozważmy teraz sieć przedstawioną na rys. 7.2. Przy założeniu, że funkcje FOR i BACK są stale równe 1, sieć ta działa w semantyce MAX jak program

begin while $p > 0$ **do** t_1 **od**; t_2 **end**

Wynika stąd, że w semantyce MAX każda funkcja częściowo rekurencyjna może być wyliczona przez pewną sieć Petriego. Zatem, problem stopu dla sieci z semantyką MAX jest nierozstrzygalny. Jednak, jak wynika z twierdzenia Meyra [39], problem stopu dla sieci Petriego z semantyką ARB jest rozstrzygalny. W konsekwencji istnieje sieć, której zachowanie w semantyce MAX, nie może być symulowane przez żadną sieć w semantyce ARB. Powyższe rozumowanie oraz przykłady zaczerpnęliśmy z pracy Burkharda [10].



RYS. 7.2

Logika modalna jako narzędzie analizy programów współbieżnych 7.5

Niedeterminizm obliczeń współbieżnych uzasadnia potrzebę skonstruowania formalnego opisu znaczenia programów współbieżnych. Powtarzanie doświadczenia obliczeniowego nie musi prowadzić do tych samych wyników. Potrzeba aksjomatycznej specyfikacji znaczenia programów współbieżnych jest większa niż dla programów sekwencyjnych. Opis semantyki powinien być oderwany od pojęć i terminów semantycznych, takich jak stan pamięci, sterowania i in. Zadanie skonstruowania takiego opisu udało się rozwiązać dla programów sekwencyjnych (por. p. 4.5). Dla programów współbieżnych jest to znacznie trudniejsze, tym bardziej że nie mamy do tej pory należytego aparatu logiki programów współbieżnych.

W tym punkcie przedstawimy prostą logikę modalną pierwszego rzędu [12, 29], którą następnie zastosujemy do (specyfikacji) opisu matematycznego modelu współbieżności MAX.

Niech L^m będzie rozszerzeniem języka pierwszego rzędu L o skończony zbiór zmiennych zdaniowych v_{st} oraz operatory modalne $\Box$ i $\Diamond$. Zbiór formuł F^m języka L^m, oprócz formuł klasycznych języka L (por. p. 2.4), zawiera formuły postaci $\Box\alpha$ oraz $\Diamond\alpha$ dla dowolnej formuły $\alpha \in F^m$. Natomiast zbiór termów języka L^m jest identyczny ze zbiorem termów języka L.

DEFINICJA 7.13

Strukturą semantyczną dla języka L^m będziemy nazywać system postaci $\mathfrak{A} = \langle S, \{\mathbf{A}(s): s \in S\}, R, w\rangle$ taki, że

(1) S jest niepustym zbiorem stanów;

(2) dla każdego $s \in S$, $\mathbf{A}(s)$ jest strukturą danych dla L (dla uproszczenia funkcje przyporządkowane funktorom niech będą całkowite);

(3) R jest relacją binarną w S;

(4) w jest funkcją zdefiniowaną na S, która każdemu stanowi s przyporządkowuje pewne wartościowanie w strukturze $\mathbf{A}(s)$ i dwuelementowej algebrze Boole'a $\mathbf{B}_0$. ■

Przy ustalonej strukturze semantycznej $\mathfrak{A}$ znaczenie wyrażeń języka L^m jest zdefiniowane następująco:

$$\tau_{\mathbf{A}}(s) = \tau_{\mathbf{A}(s)}(w(s))$$
$$\mathfrak{A}, s \models \alpha \equiv \mathbf{A}(s), w(s) \models \alpha$$

dla dowolnego termu τ i dla dowolnej formuły α w języku L. Ponadto,

$$\mathfrak{A}, s \models \Box\alpha = R(s) \neq \emptyset \text{ i } (\forall s')(s' \in R(s) \text{ implikuje } \mathfrak{A}, s' \models \alpha)$$
$$\mathfrak{A}, s \models \Diamond\alpha = (\exists s')(s' \in R(s) \text{ i } \mathfrak{A}, s' \models \alpha)$$

dla dowolnej formuły $\alpha \in L^m$.

Powiemy, że formuła modalna α jest prawdziwa w strukturze semantycznej $\mathfrak{A}$ (lub że $\mathfrak{A}$ jest modelem formuły α), w skrócie $\mathfrak{A} \models \alpha$, jeżeli dla każdego stanu s tej struktury $\mathfrak{A}$, $s \models \alpha$. Formuła modalna α jest semantyczną konsekwencją zbioru formuł Z w języku modalnym L^m, w skrócie $Z \models \alpha$, wtedy i tylko wtedy, gdy dla każdej struktury semantycznej $\mathfrak{A}$, która jest modelem zbioru Z, $\mathfrak{A} \models \alpha$.

Problem syntaktycznej charakteryzacji operacji semantycznej konsekwencji rozwiązano pozytywnie. Poniżej przedstawiamy finitystyczny system formalny ML wyznaczony przez aksjomaty Ax_m oraz reguły wnioskowania R_m.

Ax_m: aksjomaty Ax1 – Ax11 (por. p. 4.2) oraz

$(\forall x)\,\alpha(x) \Rightarrow \alpha(x/\tau)$ dla dowolnego termu τ

$(\forall x)\,\alpha(x) \equiv \neg(\exists x)\,\neg\alpha(x)$

$\Box\alpha \Rightarrow \Diamond\alpha$

$\Box(\alpha \wedge \beta) \equiv (\Box\alpha \wedge \Box\beta)$

$\neg\Diamond\alpha \equiv (\neg\Box\,\mathbf{true} \wedge \Box\neg\alpha)$

$\neg\Diamond\,\mathbf{false}$

R_m:

$$\frac{(\alpha \Rightarrow \beta),\ \alpha}{\beta}$$

$$\frac{\alpha(x)}{(\forall x)\,\alpha(x)}$$

$$\frac{(\alpha \Rightarrow \beta)}{(\Diamond\alpha \Rightarrow \Diamond\beta)}$$

Niech $\vdash_m$ oznacza operację syntaktycznej konsekwencji wyznaczoną przez aksjomaty Ax_m i reguły wnioskowania R_m (por. definicję 4.4).

WIERDZENIE 7.1

la dowolnego zbioru formuł Z i dla dowolnej formuły α zachodzi następująca własność

$$Z \vdash_m \alpha \quad \text{implikuje} \quad Z \models \alpha \qquad \square$$

Niech

$$\mathfrak{A} = \langle S, \{\mathfrak{A}(s) : s \in S\}, R, w \rangle$$

dzie dowolną strukturą semantyczną dla języka L^m i niech dla dowolnego anu $s_o \in S$, $\mathfrak{A}(s_o)$ oznacza strukturę

$$\langle S_o, \{\mathfrak{A}(s) : s \in S_o\}, R_o, w_o \rangle$$

definiowaną następująco

$$S_o = \{s \in S : s_o R^{rtc} s\}$$

dzie R^{rtc} jest zwrotnym i przechodnim domknięciem relacji R,

$$R_o \stackrel{df}{=} R/S_o \times S_o \quad \text{i} \quad w_o \stackrel{df}{=} w_o/S_o.$$

MAT 7.1

la dowolnego zbioru formuł Z, jeżeli $\mathfrak{A}$ jest modelem Z, to podstruktura (s_o) jest także modelem zbioru Z. ■

Klasa modeli języka L^m jest bardzo duża. Do celów charakteryzacji mantyki procesów współbieżnych ograniczymy tę klasę do struktur semantycznych związanych z jedną tylko strukturą danych.

EFINICJA 7.14

Strukturę semantyczną $\mathfrak{A} = \langle S, \{\mathbf{A}(s) : s \in S\}, R, w \rangle$ nazywamy jednorodną wtedy i tylko wtedy, gdy dla wszystkich $s \in S$ struktury danych $\mathbf{A}(s)$ są identyczne. Strukturą jednorodną, związaną ze strukturą danych $\mathbf{A}$, będziemy oznaczać krótko przez $\mathfrak{A}(\mathbf{A})$. ■

Aksjomaty modalne obliczeń współbieżnych 7.6

W tym punkcie przedstawimy jednolitą metodę pozwalającą skonstruować, dla dowolnego programu współbieżnego M, pewien zbiór formuł w języku modalnym L^m (por. p. 7.5), zwany dalej aksjomatami programu M. Będziemy go oznaczać Ax(dM). W następnym punkcie wykażemy, że formuły te charakteryzują obliczenia programu M w semantyce MAX.

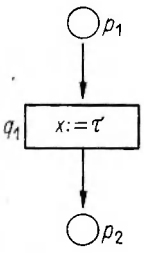
Niech M będzie dowolnie ustalonym programem współbieżnym. Programowi M przyporządkujemy pewien graf dM, zwany diagramem programu M. Diagram jest grafem dwudzielnym, tzn. zbiór jego wierzchołków składa się z dwu rozłącznych podzbiorów (na rysunkach kółka i prostokąty), a krawędzie łączą zawsze wierzchołki z różnych podzbiorów. W każdym diagramie istnieje dokładnie jeden wierzchołek wejściowy i jeden wierzchołek wyjściowy. Niech V_{st} będzie skończonym zbiorem zmiennych zdaniowych, które nie występują w programie M. Zmienne V_{st}, nazywane w dalszym ciągu zmiennymi sterowania, będą etykietami wierzchołków diagramu. Do oznaczenia zmiennych sterowania etykietujących wierzchołki ze zbioru kółek będzie używana litera p z odpowiednimi indeksami, a do oznaczenia zmiennych sterowania etykietujących wierzchołki ze zbioru prostokątów będzie używana litera q z indeksami.

Diagram dM jest zdefiniowany przez indukcję ze względu na strukturę programu M następująco.

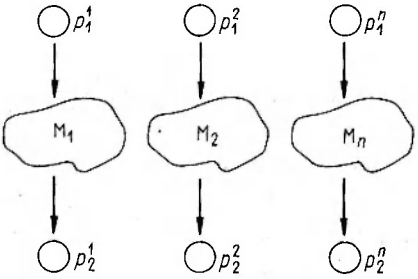
DEFINICJA 7.15

(1) Jeżeli M jest instrukcją przypisania postaci $x := \tau$, to rys. 7.3 przedstawia diagram programu M, w którym p_1 jest etykietą wejścia, p_2 etykietą wyjścia, a q_1 jest etykietą prostokąta z instrukcją $x := \tau$.

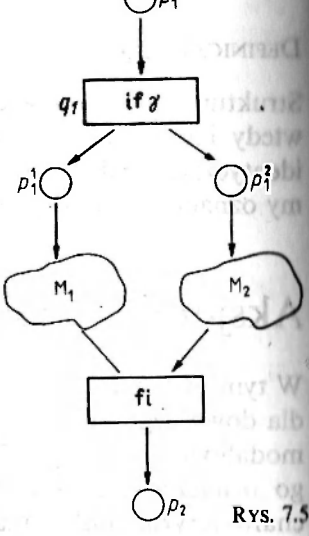
(2) Jeżeli dM_1 i dM_2 są diagramami programów M_1 i M_2 przedstawionymi na rys. 7.4 takimi, że zbiory ich zmiennych sterowania są rozłączne i zmienne p_1, p_2, q_1 nie występują w tych diagramach, to rys. 7.5 przedstawia diagram programu **if** γ **then** M_1 **else** M_2 **fi**, w którym utożsamiono wyjście diagramu dM_2 z wyjściem diagramu dM_1 i oznaczono ten wierzchołek etykietą p_2.



Rys. 7.3



Rys. 7.4



Rys. 7.5

(3) Jeżeli dM_1 i dM_2 są diagramami programów M_1 i M_2 (rys. 7.4) takimi, że zbiory ich zmiennych sterowania są rozłączne i zmienna p nie występuje w tych diagramach, to rys. 7.6 przedstawia diagram dM programu **begin** M_1; M_2 **end**, w którym utożsamiono wejście diagramu dM_2 z wyjściem diagramu dM_1 i oznaczono ten wierzchołek etykietą p.

(4) Jeżeli dM_1 jest diagramem programu M_1 przedstawionym na rys. 7.4 oraz p_1, p_2, q_1 nie występują w diagramie dM_1, to graf dM przedstawiony na rys. 7.7 jest diagramem programu **while** γ **do** M **od**, w którym utożsamiono wyjście diagramu dM_1 z wejściem diagramu dM i oznaczono je przez p_1.

(5) Jeżeli dM_i dla $i \leqslant n$ są diagramami programów $M_1, \ldots, M_n$ przedstawionymi na rys. 7.4 takimi, że ich zbiory zmiennych sterowania są rozłączne oraz zmienne p_1, p_2, q_1, q_2 nie występują w żadnym z diagramów dM_i, to graf przedstawiony na rys. 7.8 jest diagramem programu **cobegin** $M_1 \| \ldots \| M_n$ **coend.** ■

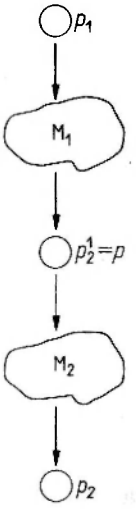
Niech dM będzie diagramem programu M. Niech P i Q tworzą rozłączne zbiory zmiennych sterowania, odpowiadające dwóm typom wierzchołków diagramu dM. Jeżeli wierzchołki $w1$ i $w2$ są połączone krawędzią w diagramie dM ($w1$ poprzedza $w2$) i $p \in P$ (lub dualnie $q \in Q$) jest etykietą wierzchołka $w1$, to q_p (lub dualnie p_q) oznacza etykietę wierzchołka $w2$. Dla dowolnej zmiennej $q \in Q$, będącej etykietą pewnej instrukcji K, przez V_q oznaczymy zbiór zmiennych występujących w K.

Zbiór Ax (dM) składa się z

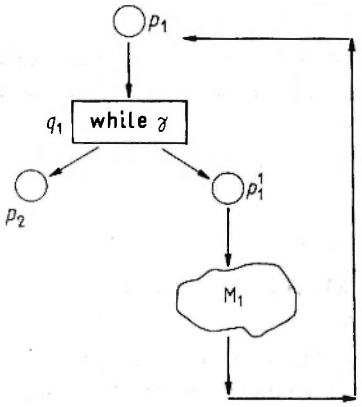
(1) aksjomatów lokalnych Lok(dM),

(2) aksjomatów globalnych, w ich skład wchodzą m.in.

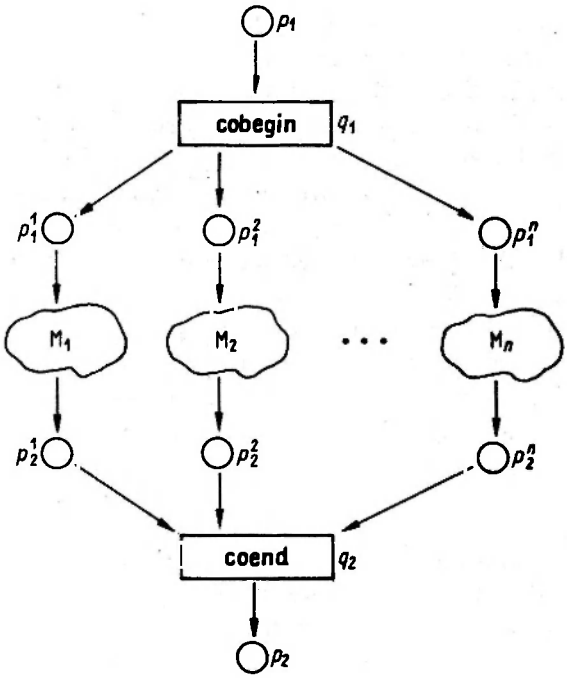
(a) aksjomaty konfliktu, Konf(dM), które opisują wszystkie możliwe sytuacje konfliktowe w M,



Rys. 7.6



Rys. 7.7



Rys. 7.8

(b) aksjomaty Ctrl(dM) opisujące zbiór możliwych stanów sterowania,

(c) aksjomaty semantyki MAX, które opisują wzajemne zachowanie się procesów.

Wszystkie aksjomaty zależą od postaci programu M, dla którego są budowane. Podamy teraz rekurencyjną definicję tych formuł.

Aksjomaty lokalne Lok(dM)

(a) Niech $M = (x := \gamma)$ oraz niech dM będzie grafem przedstawionym na rys. 7.3. Wtedy Lok(dM) składa się z następujących schematów formuł:

$$p_1 \Rightarrow \Box(p_1 \vee \neg p_1 \wedge q_1) \tag{7.1}$$
$$(q_1 \wedge \alpha(x/\tau)) \Rightarrow \Box(q_1 \wedge \alpha(x/\tau) \vee \neg q_1 \wedge p_2 \wedge \alpha(x))$$

gdzie x jest jedyną zmienną wolną w α (7.2)

(b) Niech $M =$ **if** γ **then** M_1 **else** M_2 **fi** oraz niech dM_1, dM_2 będą diagramami przedstawionymi na rys. 7.4 oraz niech dM będzie grafem przedstawionym na rys. 7.5. Wtedy Lok(dM) składa się z formuł Lok(dM_1), Lok(dM_2) (w których zmienne p_2^1, p_2^2 zastąpiono nową zmienną p) oraz następujących schematów:

$$p_1 \Rightarrow \Box(p_1 \vee q_1 \wedge \neg p_1) \tag{7.3}$$

$$(q_1 \wedge \gamma \wedge \alpha) \Rightarrow \Box((q_1 \wedge \gamma \vee \neg q_1 \wedge p_1^1) \wedge \alpha) \tag{7.4}$$

$$(q_1 \wedge \neg\gamma \wedge \alpha) \Rightarrow \Box((q_1 \wedge \neg\gamma \neg q_1 \wedge p_1^2) \wedge \alpha)$$

dla dowolnej formuły α takiej, że $V(\alpha) \subset V(\gamma)$ (7.5)

(c) Niech M = **begin** M_1; M_2 **end** i niech dM będzie grafem z rys. 7.6. Wtedy Lok(dM) składa się z formuł Lok(dM_1) i Lok(dM_2), w których zmienne p_2^1 (wyjście programu M_1), p_1^2 (wejście programu M_2) zastąpiono nową zmienną p.

(d) Niech M = **while** γ **do** M_1 **od** i niech dM będzie grafem przedstawionym na rys. 7.7, a dM_1 grafem z rys. 7.4. Wtedy zbiór Lok(dM) składa się z formuł Lok(dM_1) (w których p_2^1 zastąpiono przez p_1) i następujących schematów formuł:

$$p_1 \Rightarrow \Box(p_1 \vee q_1 \wedge \neg p_1) \tag{7.6}$$

$$(q_1 \wedge \gamma \wedge \alpha) \Rightarrow \Box((q_1 \wedge \gamma \vee \neg q_1 \wedge p_1^1) \wedge \alpha) \tag{7.7}$$

$$(q_1 \wedge \neg\gamma \wedge \alpha) \Rightarrow \Box((q_1 \wedge \neg\gamma \vee \neg q_1 \wedge p_2) \wedge \alpha)$$

dla dowolnej formuły α takiej, że $V(\alpha) \subset V(\gamma)$ (7.8)

(e) Niech M = **cobegin** $M_1 \| \ldots \| M_n$ **coend** i niech dM_i będą diagramami przedstawionymi na rys. 7.4, a dM będzie diagramem przedstawionym na rys. 7.8. Wtedy Lok(dM) składa się ze zbioru formuł $\bigcup \{\text{Lok}(dM_i) : i \leqslant n\}$ oraz następujących schematów formuł:

$$p_1 \Rightarrow \Box(q_1 \wedge \neg p_1) \tag{7.9}$$

$$q_1 \Rightarrow \Box(\neg q_1 \wedge p_1^1 \wedge p_1^2 \wedge \ldots \wedge p_1^n) \tag{7.10}$$

$$(p_2^1 \wedge p_2^2 \wedge \ldots \wedge p_2^n) \Rightarrow \Box(q_2 \wedge \neg p_2^1 \wedge \ldots \wedge \neg p_2^n) \tag{7.11}$$

$$q_2 \Rightarrow \Box(\neg q_2 \wedge p_2) \tag{7.12}$$

Aksjomat konfliktu

Oznaczmy przez Konf(dM) lub krótko Konf, jeśli wiadomo o który diagram chodzi, alternatywę wszystkich formuł postaci $(q \wedge q')$ takich, że q jest etykietą pewnej instrukcji postaci $x := \tau$, a $q' \in Q$ jest etykietą instrukcji

"**if** γ" i $x \in V(\gamma)$ lub
"**while** γ" i $x \in V(\gamma)$ lub
"$y := \eta$" i $x \in V(\eta)$ lub
"$x := \eta$"

w innym procesie. Aksjomat konfliktu ma wówczas następującą postać:

$$\neg\text{Konf}(dM) \tag{7.13}$$

Aksjomaty semantyki MAX Max(dM)

Dla uproszczenia zapisu, przyjęliśmy następujące oznaczenie

$$st(X, Y) \stackrel{\text{df}}{=} \bigwedge_{x \in X} x \wedge \bigwedge_{x \in Y - X} \neg x$$

gdzie X i Y są dowolnymi podzbiorami zbioru zmiennych sterowania. Niech P' i Q' oznaczają odpowiednio podzbiory zbiorów zmiennych sterowania P i Q.

$$(st(P', P) \wedge st(Q', Q)) \Rightarrow (max \equiv \bigwedge \{\text{Konf}(q_p/\textbf{true})) : p \in P', q_p \in V(\text{Konf})\} \tag{7.14}$$

$$(st(P', P) \wedge st(Q', Q) \wedge max \wedge \alpha) \Rightarrow \Box(\neg st(Q', Q) \wedge st(P', P) \wedge \alpha) \tag{7.15}$$

dla dowolnej formuły zdaniowej α takiej, że $V(\alpha) \subset V \cup P' \cup (Q - Q')$

$$(st(P', P) \wedge st(Q', Q) \wedge max \wedge \alpha) \Rightarrow \Diamond(st(Q' - J, Q) \wedge st(P' \cup \{p_q : q \in J\}, P) \wedge \wedge \neg \Diamond(st(Q' - J, Q) \wedge \neg\alpha)) \tag{7.16}$$

dla dowolnego $J \subset Q'$ i dowolnej formuły α takiej, że $V(\alpha) \subset V - \bigcup \{V_q : q \in J\}$

$$(st(P', P) \wedge st(Q', Q) \wedge \neg max \wedge \alpha) \Rightarrow \Box(max \wedge \neg st(P, \emptyset) \wedge st(Q', \emptyset) \wedge \alpha) \tag{7.17}$$

dla dowolnej formuły α takiej, że $V(\alpha) \cap P' \cup (Q - Q') = \emptyset$

$$(st(P', P) \wedge st(Q', Q) \wedge \neg max) \Rightarrow \Diamond(st(P' - J, P) \wedge st(Q' \cup J', Q)) \tag{7.18}$$

dla dowolnego J, $J \subset P'$, i $J' = \{q_p : p \in J\}$.

Aksjomaty możliwych stanów sterowania Ctrl(dM)

$$(p_1 \Rightarrow \neg z) \text{ dla dowolnej zmiennej } z \neq p_1,\ z \in P \cup Q \tag{7.19}$$

$$\text{Ctrl}(\text{dM}) = \bigvee_{X \in ST(dM)} st(X, P \cup Q) \tag{7.20}$$

gdzie ST(dM) jest rodziną podzbiorów zbioru $P \cup Q$, zdefiniowaną przez indukcję ze względu na strukturę programu M następująco:

(1) Jeżeli dM jest diagramem programu $\text{M} = x := \tau$, przedstawionym na rys. 7.3, to

$$ST(\text{dM}) \stackrel{\text{df}}{=} \{\{p_1\}, \{q_1\}, \{p_2\}\}$$

(2) Jeżeli dM jest diagramem programu $\text{M} = \textbf{if } \tau \textbf{ then } \text{M}_1 \textbf{ else } \text{M}_2 \textbf{ fi}$, przedstawionym na rys. 7.5, to

$$ST(\text{dM}) \stackrel{\text{df}}{=} ST(\text{dM}_1) \cup ST(\text{dM}_2) \cup \{\{p_1\}, \{q_1\}, \{p_2\}\}$$

(3) Jeżeli dM jest diagramem programu $\text{M} = \textbf{begin } \text{M}_1; \text{M}_2 \textbf{ od}$, przedstawionym na rys. 7.6, to

$$ST(\text{dM}) \stackrel{\text{df}}{=} ST(\text{dM}_1) \cup ST(\text{dM}_2) \cup \{\{p_1\}, \{p\}, \{p_2\}\}$$

(4) Jeżeli dM jest diagramem programu M = **while** τ **do** M_1 **od fi**, przedstawionym na rys. 7.7, to

$$ST(\mathrm{dM}) \stackrel{\mathrm{df}}{=} ST(\mathrm{dM}_1) \cup \{\{p_1\}, \{q_1\}, \{p_2\}\}$$

(5) Jeżeli dM jest diagramem programu M = **cobegin** $M_1 \| \ldots \| M_2$ **coend**, przedstawionym na rys. 7.8, to

$$ST(\mathrm{dM}) \stackrel{\mathrm{df}}{=} \{\{p_1\}, \{p_2\}, \{q_1\}, \{q_2\}\} \cup \{X_1 \cup X_2 \cup \ldots \cup X_n : X_i \in ST(\mathrm{dM}_i)\}$$

Intuicje związane ze zbiorem $ST(\mathrm{dM})$ są następujące. Każdy ze zbiorów tej rodziny zawiera te zmienne sterowania, które (potencjalnie) mogą być równocześnie prawdziwe. W ten sposób każdy taki zbiór opisuje jakiś stan sterowania programu M, a Ctrl(dM) — wszystkie możliwe stany sterowania programu M.

Intuicyjnie, formuła Konf(dM) wskazuje, które akcje nie mogą być wykonywane równocześnie. Zatem $\neg$ Konf(dM) gwarantuje, że zawsze mamy do czynienia ze stanem bezkonfliktowym.

Aksjomaty lokalne opisują jak zmienia się stan w zależności od wykonywanej akcji.

Pierwsza grupa aksjomatów MAX (7.14) definiuje nasze rozumienie maksymalności. Zbiór instrukcji działających jest maksymalny wtedy i tylko wtedy, gdy każde rozszerzenie tego zbioru prowadzi do konfliktu. Zmienna zdaniowa *max* wskazuje czy można zbiór aktywnych instrukcji rozszerzyć nie powodując konfliktu.

Aksjomaty z grupy (7.15) zapewniają, że przejście ze stanu, w którym pewien maksymalny zbiór instrukcji jest aktywny, do stanu następnego może nastąpić tylko wtedy, gdy zakończy się wykonywanie co najmniej jednej aktywnej instrukcji.

Ostatnia grupa (7.17) zapewnia, że jeśli aktualnie nie jesteśmy w stanie maksymalnie zaangażowanych procesów, to następny stan będzie miał tę własność. Ponadto własność (7.18) gwarantuje, że żadna zmienna programu nie zmieni swojej wartości. Zmiany są możliwe jedynie w obrębie zmiennych sterowania (por. własność 7.17).

W następnym punkcie zajmiemy się zbadaniem klasy modeli zbioru Ax(dM) i spróbujemy odpowiedzieć na pytanie czy aksjomaty te definiują semantykę MAX obliczeń współbieżnych.

Aksjomaty modalne definiują semantykę programu współbieżnego 7.7

Niech M będzie ustalonym programem zbudowanym nad językiem pierwszego rzędu L i niech dM będzie ustalonym jego diagramem. W tym punkcie

zbiór zmiennych jest rozszerzeniem zbioru zmiennych języka L o zmienne sterowania V_{st} występujące w diagramie dM, i wykażemy, że struktura ta jest modelem zbioru aksjomatów Ax(dM).

Dla ustalenia uwagi niech zbiór V_{st} będzie sumą rozłącznych zbiorów P i Q. Niech ponadto **A** będzie dowolną strukturą danych dla L. Przyjmijmy

$$\mathbf{Comp}(\mathbf{A}, \mathrm{M}) \stackrel{\mathrm{df}}{=} \langle S, R, w \rangle$$

gdzie

S jest zbiorem wszystkich konfiguracji, które występują w dowolnym obliczeniu programu M w strukturze **A**,

R jest relacją następstwa $\rightarrow$ w zbiorze konfiguracji (por. p. 7.2),

w jest funkcją przyporządkowującą stanom wartościowania w następujący sposób: $w(\langle v, J, K \rangle) \stackrel{\mathrm{df}}{=} v^+$ gdzie

$v^+(z) = v(z)$ dla dowolnej zmiennej indywiduowej $z \in V$,

$v^+(q) = \mathbf{1}$ wtw q jest etykietą instrukcji ze zbioru J,

$v^+(p) = \mathbf{1}$ wtw p jest etykietą instrukcji ze zbioru First(K)$-J$,

$v^+(max) = \mathbf{1}$ wtw zbiór J jest maksymalnym, niekonfliktowym podzbiorem zbioru First(K).

Pokażemy, że w strukturze **Comp**(**A**, M) są spełnione wszystkie, podane w poprzednim punkcie, aksjomaty Ax(dM).

LEMAT 7.2

$$\mathbf{Comp}(\mathbf{A}, \mathrm{M}) \models \neg \mathrm{Konf}$$

DOWÓD

Rozważmy dowolnie ustalony stan $s = \langle v, J, \mathrm{K} \rangle$ struktury **Comp**(**A**, M) i przypuśćmy, że **Comp**(**A**, M), $s \models$ Konf. Istnieje wtedy para zmiennych zdaniowych $q_1, q_2 \in V_{st}$ taka, że **Comp**(**A**, M), $s \models (q_1 \wedge q_2)$, tzn. instrukcje K_1, K_2 odpowiadające etykietom q_1, q_2 są w konflikcie. Zgodnie z definicją struktury **Comp**(**A**, M), K_1, $\mathrm{K}_2 \in J$. Zatem J jest zbiorem konfliktowym; sprzeczność z definicją semantyki MAX. □

LEMAT 7.3

$$\mathbf{Comp}(\mathbf{A}, \mathrm{M}) \models \mathrm{Max(dM)}$$

DOWÓD

Niech $s = \langle v, J, \mathrm{K} \rangle$ będzie dowolnym stanem struktury **Comp**(**A**, M). Załóżmy ponadto, że

$$\mathbf{Comp}(\mathbf{A}, \mathrm{M}), s \models (st(P', P) \wedge st(Q', Q))$$

dla pewnych zbiorów P' i Q' takich, że $P' \subset P, Q' \subset Q$.

(1) Niech $s \models max$. Wtedy zbiór J aktywnych instrukcji jest zbiorem maksymalnym. Inaczej mówiąc, zbiór $J \cup \{K'\}$ jest konfliktowy dla dowolnej instrukcji $K' \in \mathrm{First}(\mathrm{K}) - J$. Rozważmy dowolną zmienną $q_p \in V(\mathrm{Konf})$ dla pewnego $p \in P'$. Oznaczmy przez K_1 instrukcję odpowiadającą etykiecie q_p. Mamy wtedy

$$s \models p \quad \text{oraz} \quad s \models \neg q_p$$

Niech formuła Konf będzie postaci

$$(q_p \wedge q_1) \vee \ldots \vee (q_p \wedge q_n) \vee \beta$$

gdzie β nie zawiera wystąpień zmiennej q_p, tzn. $q_1, \ldots, q_n$ są etykietami wszystkich instrukcji, które są w konflikcie z q_p. Gdyby żadna ze zmiennych q_i nie należała do Q', to zbiór $J \cup \{\mathrm{K}_1\}$ byłby niekonfliktowy. Zatem zbiór J nie byłby maksymalny, wbrew założeniu. Wynika stąd, że istnieje i takie, że $q_i \in Q'$, tzn. $s \models q_i$. Zatem dla dowolnego $q_p \in V(\mathrm{Konf})$ takiego, że $p \in P'$ istnieje $q' \in V(\mathrm{Konf})$, dla którego

$$s \models (q_p \wedge q')(q_p/\mathbf{true})$$

W konsekwencji $s \models \mathrm{Konf}(q_p/\mathbf{true})$.

(2) Przypuśćmy, że

$$s \models \neg max$$

Wtedy zbiór J nie jest maksymalnym, bezkonfliktowym zbiorem instrukcji. Istnieje więc w zbiorze First (K) $-J$ instrukcja, np. K′, która dołączona do zbioru J nie powoduje konfliktu. Niech etykietą tej instrukcji będzie q_p, wówczas etykieta p poprzedzająca q_p, w diagramie dM należy do zbioru P'. Mamy zatem

$$s \models p$$

Przedstawmy formułę Konf w postaci

$$(q_p \wedge q_1) \vee \ldots \vee (q_p \wedge q_n) \vee \beta$$

gdzie $q_p \notin V(\beta)$. Ponieważ zbiór J jest bezkonfliktowy więc z lematu 7.2 mamy $s \models \neg\mathrm{Konf}$. Zatem dla wszystkich $i \leqslant n$ musi być $s \models \neg q_i$, gdyż w przeciwnym razie zbiór $J \cup \{\mathrm{K}'\}$ byłby konfliktowy, wbrew założeniu. Wynika stąd, że dla wszystkich $i \leqslant n$, $q_i \notin Q'$, a więc $s \models \neg\mathrm{Konf}(q_p/\mathbf{true})$. Ostatecznie dla wszystkich $q_p \in V(\mathrm{Konf})$ takich, że $p \in P'$

$$\mathbf{Comp}(\mathbf{A}, \mathrm{M}), s \models \neg\mathrm{Konf}(q_p/\mathbf{true})$$

Łącznie, przypadki (1) i (2) składają się na dowód prawdziwości formuły (7.14). Prawdziwość pozostałych formuł (7.15)–(17.18) z grupy Max(dM) można udowodnić w podobny sposób (dowód pozostawiamy czytelnikowi). □

LEMAT 7.4

Comp(A, M) $\models$ Lok(dM)

DOWÓD

Niech dM będzie diagramem instrukcji przypisania $x := \tau$, przedstawionym na rys. 7.3. Rozważmy dowolny stan $s = \langle v, J, K \rangle$ taki, że $s \models (st(P', P) \wedge st(Q', Q))$.
Przypuśćmy, że $s \models p_1$. Z definicji struktury **Comp**(A, M) instrukcja, której etykietą jest p_1, należy do zbioru First(K) $-J$. Jeżeli $s \models max$, to na mocy powyższych rozważań dla każdego bezpośredniego następnika s' stanu s, $s' \models p_1$. Jeżeli $s \models \neg max$ wtedy jest możliwe, że instrukcja $(x := \tau)$ będzie wybrana do wykonania. Zatem we wszystkich możliwych bezpośrednich następnikach s' stanu s, $s' \models (p_1 \vee q_1 \wedge \neg p_1)$, co dowodzi prawdziwości formuły (7.1).

Niech $\alpha(x)$ będzie formułą z jedną wolną zmienną indywiduową x taką, że $s \models q_1 \wedge \alpha(x/\tau)$. Jeżeli $s \models \neg max$, to $s' \models (q_1 \wedge \alpha(x/\tau))$ dla dowolnego bezpośredniego następnika s' stanu s. Jeżeli $s \models max$, to instrukcja $(x := \tau)$ może zostać wykonana w tym kroku obliczenia. Zatem jej etykieta zostanie usunięta ze zbioru Q' i etykietę następnika instrukcji $(x := \tau)$ dołączy się do zbioru P'. Ponadto, ponieważ $v(x/\tau) \models \alpha(x)$, więc dla wszystkich następników s' stanu s, $s' \models \alpha(x)$. Ostatecznie

$$s' \models (\neg q_1 \wedge p_2 \wedge \alpha(x))$$

Dla $s \models max$ jest również możliwe, że $(x := \tau)$ nie zostanie wykonana w tym kroku obliczenia. Dla takiego następnika s' stanu s, $s' \models q_1$. Instrukcje wykonane w tym kroku nie mogą zmienić żadnej zmiennej ze zbioru $V(\tau) \cup \{x\}$, z powodu konfliktu. Zatem

$$s' \models (q_1 \wedge \alpha(x/\tau))$$

Dokładną analizę pozostałych formuł z grupy aksjomatów lokalnych pozostawiamy czytelnikowi. □

LEMAT 7.5

Comp(A, M) $\models$ Ctrl(dM)

DOWÓD

Prawdziwość aksjomatu stanu początkowego (7.19) wynika z definicji konfiguracji początkowej dowolnego obliczenia. W pozostałych formułach tej

grupy jest wyrażona idea, że wystąpienie instrukcji nie może być równocześnie aktywne (markowane przez ∘) i gotowe do wykonania. Ponadto, w każdym sekwencyjnym procesie co najwyżej jedna akcja jest aktywna.

Ścisły dowód przebiega przez indukcję ze względu na strukturę programu, ze względu na proste i nieciekawe szczegóły techniczne, jest pominięty. □

Lematy 7.2–7.5 stanowią dowód następującego twierdzenia:

TWIERDZENIE 7.2

Dla dowolnej struktury **A**, **Comp**(**A**, M) jest modelem zbioru Ax(dM),

$$\mathbf{Comp}(\mathbf{A}, \mathrm{M}) \models \mathrm{Ax}(\mathrm{dM})$$ ■

Analizując dokładniej dowody lematów 7.2–7.5 łatwo zauważymy następujący prosty fakt. Dla dowolnej struktury danych **A** i dla dowolnej konfiguracji początkowej $\langle v, \mathrm{M}\rangle$ w **A**, podstruktura **Comp**(**A**, M, v) wyznaczona przez $\langle v, \mathrm{M}\rangle$ jest modelem zbioru formuł Ax(dM) (por. lemat 7.1).

DEFINICJA 7.16

Strukturę **Comp**(**A**, M, v) będziemy nazywać minimalnym modelem obliczeniowym wyznaczonym przez **A** i v. ■

Obecnie mamy zamiar wykazać, że twierdzenie w pewnym sensie odwrotne do twierdzenia 7.2 jest także prawdziwe. Pokażemy, że model zbioru aksjomatów Ax(dM) może być zredukowany do minimalnego modelu obliczeniowego.

DEFINICJA 7.17

Model jednorodny (por. def. 7.14) $\mathfrak{A}(\mathbf{A}) = \langle S, R, w\rangle$ zbioru Ax(dM) taki, że

(1) *non* $\mathfrak{A}(\mathbf{A}) \models \neg p_1$

gdzie p_1 jest etykietą początkową diagramu d(M);

(2) $s_1 \neq s_2$ wtedy i tylko wtedy, gdy istnieje formuła $\alpha \in \mathrm{L}^{\mathrm{m}}$ taka, że $s_1 \models \alpha$ i $s_2 \models \neg\alpha$ dla dowolnych stanów s_1 i s_2;

będziemy nazywać modelem właściwym zbioru Ax(dM). ■

TWIERDZENIE 7.3

Niech $\mathfrak{A}(\mathbf{A}, s_0)$ będzie dowolnym modelem właściwym zbioru Ax(dM) wyznaczonym przez stan s_0 taki, że $s_0 \models p_1$. Wówczas model $\mathfrak{A}(\mathbf{A}, s_0)$ (w skrócie $\mathfrak{A}$) jest izomorficzny z modelem obliczeniowym **Comp**(**A**, M, v_0), gdzie $v_0 = w(s_0)/V$.

Dowód

Niech s będzie dowolnie ustalonym stanem struktury $\mathfrak{A}$ i niech

$$\mathfrak{A}, s \models st(P', P) \wedge st(Q', Q)$$

Przyjmijmy

$$h(s) \stackrel{\text{df}}{=} \langle v, J, \mathrm{K} \rangle$$

gdzie v jest obcięciem wartościowania $w(s)$ do zbioru zmiennych V, J jest zbiorem tych instrukcji programu M, których etykiety, q należą do zbioru Q', K jest programem, który pozostaje do wykonania, gdy zbiór instrukcji początkowych First(M) składa się z tych wystąpień instrukcji, których etykiety p należą do zbioru P'.

Zauważmy, że na mocy aksjomatu (7.13) i definicji odwzorowania h, $\mathfrak{A}, s \models \neg \mathrm{Konf}(\mathrm{dM})$ wtedy i tylko wtedy, gdy J jest zbiorem niekonfliktowym. Ponadto, $\mathfrak{A}, s \models max$ tylko wtedy, gdy J jest zbiorem maksymalnym, co wynika z aksjomatu (7.14) i definicji funkcji h.

Niech sRs' dla pewnego s' i niech $h(s) \in \mathbf{Comp}(\mathbf{A}, \mathrm{M}, v_0)$. Wykażemy, że

$$h(s') \in \mathbf{Comp}(\mathbf{A}, \mathrm{M}, v_0) \tag{7.21}$$
$$h(s) \multimap\!\!\ast\!\!\rightarrow h(s') \tag{7.22}$$

Rozważmy dwa przypadki: $\mathfrak{A}, s \models max$ i $\mathfrak{A}, s \models \neg max$.

Niech $\mathfrak{A}, s \models max$. Na mocy aksjomatu (7.15) istnieje $Q'' \subset Q'$ takie, że

$$Q'' \neq \emptyset \quad \text{oraz} \quad \mathfrak{A}, s' \models st(Q' - Q'', Q)$$

Rozważmy konfigurację $h(s') = \langle v', J', \mathrm{K}' \rangle$. Na mocy aksjomatu (7.16) wartości zmiennych sterowania ze zbioru P' są nadal prawdziwe, a ponadto prawdziwe są zmienne p_q dla $q \in Q''$. Jeżeli $q \in J$ oraz q jest etykietą instrukcji $(x := \tau)$ (por. rys. 7.3), to na mocy aksjomatów lokalnych przejść, dla dowolnej formuły α takiej, że $V(\alpha) = \{x\}$

$$s \models \alpha(x/\tau) \quad \text{implikuje} \quad s' \models p_2 \wedge \alpha(x)$$

Zatem $w(s')(x) = \tau_{\mathrm{A}}(v)$ i $w(s') \models p_2$. Na mocy aksjomatu (7.16) żadna zmienna indywiduowa ze zbioru $V - \bigcup \{V_q : q \in J\}$ nie zmieni wartości, a na mocy aksjomatów lokalnych wszystkie zmiany są związane z wykonaniem instrukcji przypisania etykietowanych przez q, $q \in J$. Ostatecznie, v' jest wynikiem wykonania wszystkich instrukcji ze zbioru I na wartościowaniu v. Zatem

$$\mathrm{K}' = \mathrm{Rest}(v, \mathrm{I}, \mathrm{K})$$

gdzie I jest zbiorem instrukcji odpowiadających etykietom $q \in Q''$. Stąd $h(s') \in \mathbf{Comp}(\mathbf{A}, \mathrm{M}, v_0)$ oraz $h(s) \multimap\!\!\ast\!\!\rightarrow h(s')$.

Niech $\mathfrak{A}, s \models \neg max$. Na mocy aksjomatu (7.18) istnieje $P'' \subset P'$ takie, że

$$P'' \neq \emptyset \quad \text{oraz} \quad \mathfrak{A}, s' \models st(P' - P'', P)$$

Rozważmy konfigurację $h(s') = \langle v', J', K' \rangle$. Na mocy aksjomatu (7.17) wartościowania $w(s')$ i $w(s)$ są identyczne na zbiorze $V \cup Q' \cup (P - P')$. Zatem przy przejściu od s do s' żadna zmienna indywiduowa nie zostanie zmieniona, wszystkie instrukcje aktywne w K i wszystkie instrukcje, które nie są gotowe do wykonania w K pozostaną takimi w K'. Ponadto, na mocy aksjomatu (7.17) wszystkie możliwe zmiany są związane ze zbiorem $P' \cup (Q - Q')$.

Na mocy aksjomatu (7.14), zbiór instrukcji odpowiadających etykietom ze zbioru Q' nie jest maksymalny i na mocy aksjomatu (7.17), zbiór instrukcji odpowiadających etykietom q takim, że $w(s') \models q$ jest maksymalnym niekonfliktowym zbiorem. Dowodzi to, że konfigurację $h(s')$ otrzymaliśmy z $h(s)$ zgodnie z definicją semantyki MAX (por. z definicją 7.5).

Aby zakończyć dowód twierdzenia 7.3 zauważmy, że na mocy podobnych argumentów wykażemy

$$h(s) \multimap h(s') \Rightarrow sR\,s'$$

dla dowolnych s, s'.
Ponieważ h jest odwzorowaniem różnowartościowym i przekształca $\mathfrak{A}(\mathbf{A}, s_0)$ na strukturę $\mathbf{Comp}(\mathbf{A}, \mathbf{M}, v_0)$ zatem h jest izomorfizmem. □

Uwaga

Analogiczny wynik może być otrzymany dla semantyki ARB; można podać taki zbiór aksjomatów modalnych, który charakteryzuje zbiór obliczeń dowolnego programu w semantyce ARB. ■

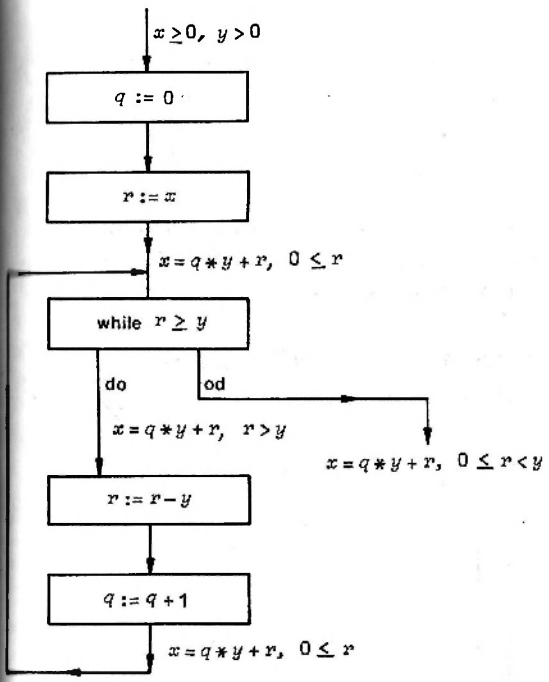
O innych logikach programów 8

Floyda opisy programów 8.1

W roku 1967 Floyd [19] zaproponował pewną metodę analizy znaczenia programów wykorzystującą pojęcie *opisu diagramu programu.* Opisem diagramu programu (por. rozdz. 3) nazywamy odwzorowanie, przyporządkowujące jego krawędziom formuły pierwszego rzędu. Jeżeli opis ma tę własność, że dla każdej krawędzi diagramu formuła jej przypisana jest spełniona, gdy podczas obliczenia programu przechodzimy wzdłuż tej krawędzi, to opis taki nazwiemy *dopuszczalnym.* Opisy dopuszczalne są pomocne w ustalaniu częściowej poprawności programów. Mianowicie, jeżeli program ma opis dopuszczalny, to jest on częściowo poprawny względem formuł przyporządkowanych krawędziom wejściowej i wyjściowej diagramu programu. Okazało się, że istnieje dość prosty syntaktyczny warunek wystarczający na to, by opis programu był dopuszczalny. Teoria przedstawiona przez Floyda wzbudziła duże zainteresowanie, była to jedna z pierwszych matematycznych teorii dotyczących semantyki programów. W pracy [22] można znaleźć wyniki dotyczące opisów programów, główny jej wynik to stwierdzenie, że warunki: opis jest akceptowalny w pewnej teorii i opis jest dopuszczalny we wszystkich modelach tej teorii, są równoważne.

W tym punkcie przedstawimy koncepcje Floyda w ujęciu nieco zmienionym. Skorzystaliśmy z wyników Banachowskiego [6] i jego sugestii, by rozważać programy strukturalne. Nasza definicja opisu różni się nieco od poprzednich i, mamy nadzieję, jest bardziej czytelna.
Rozważmy przykład opisu programu dzielenia liczb całkowitych (rys. 8.1).

Można zauważyć, że w trakcie każdego wykonywania tego programu jest zachowana następująca własność: ilekroć podczas obliczenia przechodzimy wzdłuż jakiejś krawędzi, tylekroć aktualny stan pamięci spełnia formułę przypisaną tej krawędzi. W szczególności, gdy program kończy swoje obliczenia, wtedy jest spełniona formuła przypisana krawędzi wyjściowej. Formuła ta wyraża własność następującą: liczba q jest ilorazem dwu liczb całkowitych x i y, a liczba r jest resztą z dzielenia.



Rys. 8.1

DEFINICJA 8.1

Opisem programu (ang. an annotated program) będziemy nazywać wyrażenie zdefiniowane przez indukcję, ze względu na długość programu, następująco:

Niech α i β będą dowolnymi formułami (języka logiki algorytmicznej), niech s będzie instrukcją przypisania.

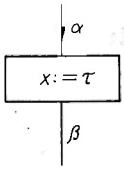
(1) Wyrażenie $\{\alpha\}\, s\, \{\beta\}$ jest opisem programu s.

Niech M1′ i M2′ będą opisami programów M1 i M2.

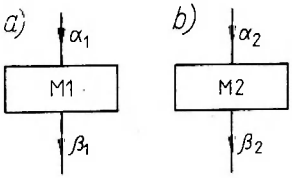
(2) Wyrażenie $\{\alpha\}$ **if** γ **then** M1′ **else** M2′ **fi** $\{\beta\}$ jest opisem programu **if** γ **then** M1 **else** M2 **fi**.

(3) Wyrażenie $\{\alpha\}$ **while** γ **do** M1′ **od** $\{\beta\}$ jest opisem programu **while** γ **do** M1 **od**.

(4) Wyrażenie $\{\alpha\}$ **begin** M1′; M2′ **end** $\{\beta\}$ jest opisem programu **begin** M1; M2 **end**. ■



Rys. 8.2

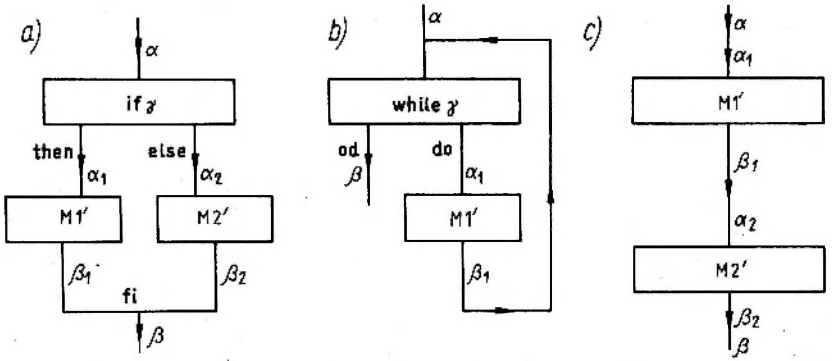


Rys. 8.3

Formuły α i β będą nazywane odpowiednio formułą wejściową i formułą wyjściową opisu programu. Najlepiej pojęcie opisu programu przedstawić na rysunkach. Opis programu atomowego, czyli instrukcji przypisania $x := \tau$ przedstawia rys. 8.2.

Niech dane będą dwa opisy M1′ i M2′ programów M1 i M2 (rys. 8.3).

Wtedy opisy programów złożonych mają postać jak na rys. 8.4. ■



Rys. 8.4

DEFINICJA 8.2

Warunkiem weryfikacyjnym dla opisu programu M′ nazywamy formułę VC(M′) określoną przez indukcję jak następuje:

(1) Jeżeli M′ jest postaci $\{\alpha\}\, s\, \{\beta\}$ gdzie s jest pewną instrukcją przypisania, a α, β są dowolnymi formułami, to $\mathrm{VC}(M') = (\alpha \Rightarrow s\beta)$.

Niech M1′ i M2′ będą opisami programów M1 i M2, niech α_i będzie warunkiem wejściowym opisu Mi′, β_i niech będzie warunkiem wyjściowym tego opisu, $i = 1, 2$.

(2) Jeżeli M′ jest postaci $\{\alpha\}$ **if** γ **then** M1′ **else** M2′ **fi** $\{\beta\}$, to
$\mathrm{VC}(M') = \mathrm{VC}(M1') \wedge \mathrm{VC}(M2') \wedge ((\alpha \wedge \gamma) \Rightarrow \alpha_1) \wedge ((\alpha \wedge \neg\gamma) \Rightarrow \alpha_2) \wedge ((\beta_1 \vee \beta_2) \Rightarrow \beta)$.

(3) Jeżeli M′ jest postaci $\{\alpha\}$ **begin** M1′; M2′ **end** $\{\beta\}$, to
$\mathrm{VC}(M') = \mathrm{VC}(M1') \wedge \mathrm{VC}(M2') \wedge (\alpha \Rightarrow \alpha_1) \wedge (\beta_1 \Rightarrow \alpha_2) \wedge (\beta_2 \Rightarrow \beta)$.

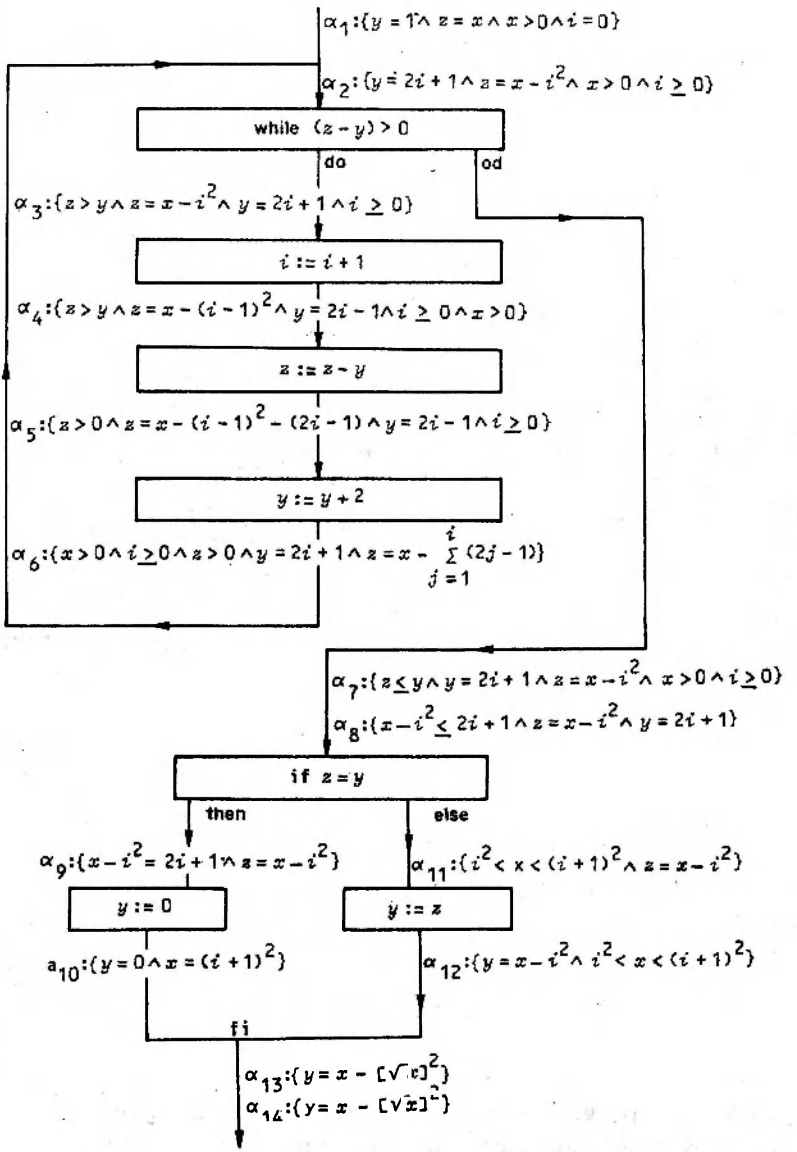
(4) Jeżeli M′ jest postaci $\{\alpha\}$ **while** γ **do** M1′ **od** $\{\beta\}$, to
$\mathrm{VC}(M') = \mathrm{VC}(M1') \wedge (((\alpha \vee \beta_1) \wedge \gamma) \Rightarrow \alpha_1) \wedge (((\alpha \vee \beta_1) \wedge \neg\gamma) \Rightarrow \beta)$. ■

PRZYKŁAD 8.1

Rozważmy następujący program M:

```
begin
  while (z−y) > 0
  do
    i:= i+1;
    z:= z−y;
    y:= y+2
  od;
  if z = y then y:= 0 else y:= z fi
end
```

i jego opis (rys. 8.5). □



Rys. 8.5

EFINICJA 8.3

opuszczalny warunek weryfikacyjny VC(M′) dla opisu programu M′ jest strukturze danych **A** wtedy i tylko wtedy, gdy jest formułą prawdziwą w **A**. ■

PRZYKŁAD 8.2

arunkiem weryfikacyjnym dla przedstawionego opisu programu jest natępująca formuła:

$$\begin{aligned}&VC(M) \equiv (\alpha_1 \Rightarrow \alpha_2) \wedge (\alpha_7 \Rightarrow \alpha_8) \wedge (\alpha_{13} \Rightarrow \alpha_{14}) \wedge (\alpha_5 \Rightarrow (y := y+2)\alpha_6) \wedge \\ &\wedge (\alpha_3 \Rightarrow (i := i+1)\alpha_4) \wedge (\alpha_4 \Rightarrow (z := z-y)\alpha_5) \wedge (((\alpha_2 \vee \alpha_6) \wedge z-y > 0) \Rightarrow \alpha_3) \wedge \\ &\wedge (((\alpha_2 \vee \alpha_6) \wedge z-y \leqslant 0) \Rightarrow \alpha_7) \wedge (\alpha_9 \Rightarrow (y := 0)\alpha_{10}) \wedge (\alpha_{11} \Rightarrow (y := z)\alpha_{12}) \wedge \\ &\wedge ((\alpha_8 \wedge z = y) \Rightarrow \alpha_9) \wedge ((\alpha_8 \wedge z \neq y) \Rightarrow \alpha_{11}) \wedge ((\alpha_{10} \vee \alpha_{12}) \Rightarrow \alpha_{13})\end{aligned}$$

Zwróćmy uwagę na to, że formuła VC(M) jest koniunkcją trzynastu mplikacji i że wszystkie one mają nieskomplikowaną strukturę. Każda z nich opisuje pewien wierzchołek w diagramie programu. Wierzchołkom z testami **if** lub **while**) odpowiadają dwie implikacje, wierzchołkom zawierającym nstrukcje podstawienia odpowiadają pojedyncze implikacje. Każda z tych mplikacji wyraża następującą prostą własność semantyczną obliczenia programu M: jeżeli obliczenie programu M osiągnęło dany wierzchołek c i jeśli bieżąca konfiguracja (stan) obliczenia spełnia warunek opisany w poprzedniku implikacji — warunek ten jest alternatywą warunków przypisanych krawędziom prowadzącym do wierzchołka c — to, po wykonaniu instrukcji (bądź testu) zawartej w wierzchołku c, nowy stan obliczenia spełnia formułę przypisaną krawędzi wychodzącej z wierzchołka c. Gdy wierzchołek jest testem i wychodzą z niego dwie krawędzie, mamy dwie mplikacje. Dla każdej krawędzi wychodzącej z wierzchołka testującego warunek można powtórzyć zdanie, jakie sformułowaliśmy dla instrukcji podstawienia. □

LEMAT 8.1

Jeżeli warunek weryfikacyjny VC(M′) programu M jest dopuszczalny w strukturze **A**, to program M jest częściowo poprawny ze względu na formułę α wejściową dla opisu M′ i formułę β wyjściową dla tego opisu

$$\mathbf{A} \models VC(M') \text{ pociąga } \mathbf{A} \models ((\alpha \wedge M\text{true}) \Rightarrow M\beta)$$ ■

LEMAT 8.2

Warunek weryfikacyjny VC(M′) programu M jest dopuszczalny w strukturze **A** wtedy i tylko wtedy, gdy w strukturze **A** jest prawdziwa formuła $(\alpha M \Rightarrow \beta)$

przypomnijmy, αM oznacza formułę — najmocniejszy następnik warunku α względem programu M. ■

DEFINICJA 8.4

Akceptowalny warunek weryfikacyjny VC(M) programu M jest w teorii $T = \langle L, C, A \rangle$, gdy jest on twierdzeniem tej teorii. ■

LEMAT 8.3

Dla danego opisu programu M, jego warunek weryfikacyjny jest akceptowalny w teorii T wtedy i tylko wtedy, gdy jest dopuszczalny w każdym modelu teorii T. ■

Właściwie jest to prosty wniosek z twierdzenia o pełności (twierdzenia 4.2). Lemat ten pozwala przekonać się o częściowej poprawności programu M za pomocą dowodzenia pewnej liczby implikacji. Warunek weryfikacyjny jest koniunkcją implikacji o dość prostej budowie. Dzięki temu na ogół nietrudno każdą taką implikację bądź udowodnić, bądź wykazać, że nie jest ona prawdziwa. Ta cecha przyczyniła się do znacznej popularności metody Floyda. Niewiele później powstał system aksjomatyczny dowodzenia częściowej poprawności opracowany przez Hoare'a [27]. Dla uwidocznienia podobieństwa idei zawartych w obu tych systemach używa się często nazwy logika Floyda-Hoare'a.

Logika Hoare'a 8.2

W roku 1969 Hoare [27] podał system reguł wnioskowania o częściowej poprawności programów. Wyrażeniami rozważanego języka są: programy, formuły pierwszego rzędu i trójki postaci $\alpha\{M\}\beta$, gdzie α i β są formułami pierwszego rzędu, a M jest programem. Trójka $\alpha\{M\}\beta$ jest wyrażeniem logicznym i wyraża własność częściowej poprawności programu M względem warunku początkowego α i warunku końcowego β. Interpretuje się ją w ten sposób, że przyjmuje ona dla pewnego stanu v wartość prawda wtedy i tylko wtedy, gdy

nie jest prawdą, że warunek wstępny α jest spełniony przez stan v,
albo gdy program M nie zatrzymuje się,
albo gdy warunek wstępny α jest spełniony przez stan v oraz program M zatrzymuje się i w stanie końcowym jest spełniony warunek końcowy β.

Inaczej mówiąc, trójka $\alpha\{M\}\beta$ wyraża tę samą własność co formuła algorytmiczna $(\alpha \wedge \mathbf{M}\mathbf{true}) \Rightarrow M\beta$. Stąd wynika, że rachunek Hoare'a jest zawarty w AL. Poniżej podamy bardziej szczegółowe uzasadnienie tej tezy.

Zacznijmy od przedstawienia formalnej konstrukcji systemu Hoare'a.

Aksjomaty

Aksjomat instrukcji przypisania

H1 $\alpha(x/\tau)\{x := \tau\}\alpha(x)$

Aksjomat instrukcji zerwij

H2 $\alpha\{\mathbf{zerwij}\}\mathbf{false}$

Aksjomat instrukcji zostaw

H3 $\alpha\{\mathbf{zostaw}\}\alpha$

Reguły wnioskowania

Reguła instrukcji złożonej

$$\text{H4}\quad \frac{\alpha\{K_1\}\alpha',\ \ \alpha'\{K_2\}\beta}{\alpha\{\mathbf{begin}\ K_1;\ K_2\ \mathbf{end}\}\beta}$$

Reguła instrukcji rozgałęzienia

$$\text{H5}\quad \frac{(\alpha\wedge\gamma)\{K\}\beta,\ \ (\alpha\wedge\neg\gamma)\{M\}\beta}{\alpha\{\mathbf{if}\ \gamma\ \mathbf{then}\ K\ \mathbf{else}\ M\ \mathbf{fi}\}\beta}$$

Reguła instrukcji powtarzania (iteracji)

$$\text{H6}\quad \frac{(\alpha\wedge\gamma)\{K\}\alpha}{\alpha\{\mathbf{while}\ \gamma\ \mathbf{do}\ K\ \mathbf{od}\}(\neg\gamma\wedge\alpha)}$$

Reguła konsekwencji

$$\text{H7}\quad \frac{\alpha'\Rightarrow\alpha,\ \alpha\{K\}\beta,\ \beta\Rightarrow\beta'}{\alpha'\{K\}\beta'}$$

Wykażemy teraz, że aksjomaty i reguły H1–H7 mogą być wyprowadzone na gruncie logiki algorytmicznej. Każde więc rozumowanie, opierające się na regułach Hoare'a, jest również poprawnym rozumowaniem w logice algorytmicznej.

Dowody słuszności reguł Hoare'a na gruncie AL

Wiemy, że reguły H4–H7 są semantycznie słuszne [2], tzn. z prawdziwych przesłanek nie można za pomocą tych reguł wyciągnąć fałszywych wniosków lub, inaczej mówiąc, prawdziwość przesłanek pociąga za sobą prawdziwość wniosku w każdej regule. Można też sprawdzić, że wyrażenia H1–H3 są zawsze prawdziwe, są tautologiami. Na podstawie twierdzenia (4.2) o pełności dla logiki algorytmicznej można stwierdzić, że aksjomaty te i reguły wnioskowania dają się wyprowadzić formalnie w AL.

Dowód aksjomatu H1

Aksjomat H1 jest równoważny formule $\alpha(x/\tau) \wedge (x := \tau)\,\mathbf{true} \Rightarrow (x := \tau)\,\alpha$, która wynika wprost z aksjomatu algorytmicznego Ax18: $(x := \tau)\,\alpha \equiv \alpha(x/\tau) \wedge \wedge (x := \tau)\,\mathbf{true}$. Aksjomat ten jest mocniejszy od aksjomatu H1 częściowej poprawności.

Dowód reguły H4

Dowód polega na wykazaniu, że formuła

$$\alpha_1 \wedge \mathbf{begin}\, K_1; K_2\, \mathbf{end}\, \mathbf{true} \Rightarrow \mathbf{begin}\, K_1; K_2\, \mathbf{end}\, \alpha_3$$

ma dowód oparty na następujących dwóch formułach —przesłankach:

$$(\alpha_1 \wedge K_1\,\mathbf{true} \Rightarrow K_1\,\alpha_2), \quad (\alpha_2 \wedge K_2\,\mathbf{true} \Rightarrow K_2\,\alpha_3)$$

Stosując regułę algorytmiczną R2 (por. z definicją 4.2) z drugiej przesłanki otrzymujemy $(K_1\,\alpha_2 \wedge K_1(K_2\,\mathbf{true}) \Rightarrow K_1(K_2\,\alpha_3))$. Łącząc to z pierwszą przesłanką $(\alpha_1 \wedge K_1\,\mathbf{true} \Rightarrow K_1\,\alpha_2)$ i stosując aksjomaty rachunku zdań, wyprowadzamy $\alpha_1 \wedge K_1(K_2\,\mathbf{true}) \Rightarrow K_1(K_2\,\alpha_3)$. Stąd, na mocy aksjomatu Ax19, otrzymujemy wniosek, który należało udowodnić. □

Dowód reguły H7

Należy wykazać, że formuła $\alpha' \wedge K\mathbf{true} \Rightarrow K\beta'$ ma dowód z przesłanek wymienionych w badanej regule. Z przesłanki $\alpha' \Rightarrow \alpha$ nietrudno wyprowadzić formułę $\alpha' \wedge K\mathbf{true} \Rightarrow \alpha \wedge K\mathbf{true}$. Stosując regułę R2 dowodzimy formułę $K\beta \Rightarrow K\beta'$. Ale formuła $\alpha \wedge K\mathbf{true} \Rightarrow K\beta$, to druga przesłanka badanej reguły dowodzenia. Stosując dwukrotnie aksjomat Ax1 otrzymujemy, najpierw $\alpha' \wedge K\mathbf{true} \Rightarrow K\beta$, a potem $\alpha' \wedge K\mathbf{true} \Rightarrow K\beta'$. □

Dowód reguły H5

Z pierwszej przesłanki wyprowadzamy formułę

$$((\alpha \wedge \gamma) \wedge \mathbf{K}\,\mathbf{true}) \Rightarrow ((\mathbf{K}\beta) \wedge \gamma)$$

Z drugiej otrzymujemy

$$((\alpha \wedge \neg\gamma) \wedge \mathbf{M}\,\mathbf{true}) \Rightarrow ((\mathbf{M}\beta) \wedge \neg\gamma)$$

Stąd stosując następujące prawo rachunku zdań

$$((\delta \Rightarrow \sigma) \wedge (\delta' \Rightarrow \sigma')) \Rightarrow (\delta \vee \delta' \Rightarrow \sigma \vee \sigma')$$

uzyskujemy

$$((\alpha \wedge \gamma) \wedge \mathbf{K}\,\mathbf{true}) \vee ((\alpha \wedge \neg\gamma) \wedge \mathbf{M}\,\mathbf{true}) \Rightarrow (\gamma \wedge \mathbf{K}\beta \vee \neg\gamma \wedge \mathbf{M}\beta)$$

Teraz już widać, że w następniku tej implikacji mamy **if** γ **then** K **else** M **fi** β, (a stosując prawo rozdzielności i wyciągając α przed nawias, widzimy, że poprzednik implikacji jest równoważny formule ($\alpha \wedge$ **if** γ **then** K **else** M **fi true**), co kończy dowód. □

Dowód reguły H6

Reguła H6, pozwalająca wnioskować o częściowej poprawności instrukcji iteracji na podstawie faktu, że formuła α jest niezmiennikiem instrukcji iterowanej, może być udowodniona za pomocą własności najsłabszego warunku wstępnego; por. Dijkstra [18]. Dowód poprawności tej reguły podaliśmy w rozdz. 4, por. przykład 4.5. □

Ostrzeżenie

Popularność, jaką cieszy się formalizm Hoare'a, każe nam za O'Donnellem [42] przytoczyć następujące ostrzeżenie. Wprowadzanie reguł wnioskowania o programach powinno odbywać się z należytą troską o niesprzeczność systemu. Rozważmy następującą regułę (por. [2]):

$$\textit{Function}\ \frac{\alpha\{\mathrm{P}\}\beta}{(\forall x)\ (\alpha \Rightarrow \beta(f(x)/y))}$$

gdzie f jest zdefiniowane za pomocą następującej deklaracji:

f: **function** (x); **local** $z_1, \ldots, z_n$;
P; **return** (y)
end

a α i β nie zawierają $z_1, \ldots, z_n$ jako zmiennych wolnych [2]. Zauważono, że dodanie tej reguły do poprzednich, prowadzi do sprzeczności. Rozważmy mianowicie następującą definicję:

f: **function** (x); **zerwij**; **return** (y) **end**

i wnioskowanie

true {**zerwij**} **false**	*aksjomat* **zerwij**
$(\forall x)$ **true** $\Rightarrow$ **false**	*reguła Function*
false	

Dowód prowadzący do **false** istnieje nawet wtedy, gdy funkcja jest częściowo określona i jej nieokreśloność ma miejsce tylko w jednym punkcie.

Co to jest relatywna pełność?

Wobec stwierdzonej niemożliwości podania pełnego i skończonego zestawu aksjomatów i reguł dla częściowej poprawności Cook [14] zaproponował rozważenie innej definicji pełności. Przyjmijmy jako aksjomaty wszystkie formuły pierwszego rzędu prawdziwe w pewnej klasie struktur danych K. Niech Z oznacza zbiór tych formuł. Przyjmijmy dalej, że w tej klasie struktur danych każda formuła postaci $\alpha\{M\}\beta$ ma równoważną jej formułę pierwszego rzędu γ tzn., że równoważność $\gamma = \alpha\{M\}\beta$ jest prawdziwa w tej klasie struktur. Jeżeli każda formuła częściowej poprawności, prawdziwa w klasie K, może być wyprowadzona ze zbioru Z, to mówimy *o relatywnej pełności.*

Badaniom relatywnej pełności poświęcono wiele prac [3, 14, 24]. Większość z nich zajmuje się problemem istnienia aksjomatyzacji relatywnie pełnych. Okazuje się, że systemy aksjomatyczne częściowej poprawności, tworzone dla bogatszych języków programowania, nie mogą cieszyć się własnością relatywnej pełności.

W praktyce programistycznej (a także matematycznej) rzadkie są przypadki, w których można przyjąć, że jest znana pełna prawda o strukturze danych. Na ogół teoria struktury jest czymś, co stale się rozwija i jest wzbogacane przez pracę całych pokoleń badaczy. Uważamy, że do tego stale rozszerzającego się zbioru prawd, powinny wchodzić także własności algorytmiczne. Przyjmowanie nierealnego założenia, że jest nam dana wyrocznia orzekająca o każdej formule arytmetyki liczb naturalnych czy formuła ta jest prawdziwa, czy nie, stawia całą sprawę analizy programów na głowie. Zwróćmy uwagę, że ponadto przyjmuje się, iż formuły algorytmiczne są sprowadzalne do formuł pierwszego rzędu. Celem pracy programistów jest badanie semantycznych własności programów w pewnej klasie K struktur danych. Ponieważ z założenia te własności są redukowalne do formuł pierwszego rzędu, zatem pozostaje jedynie przetłumaczyć formułę algorytmiczną na formułę pierwszego rzędu. Wykonalność tego tłumaczenia jest zagwarantowana przez wcześniejsze założenia. Potem trzeba tylko odwołać się do wyroczni, która orzeknie czy badana formuła pierwszego rzędu jest prawdziwa, czy nie. Ale czy taką wyrocznię można skonstruować? Badania wykazują, że nie. Dla struktury liczb naturalnych

zbiór formuł prawdziwych jest zbiorem hiperarytmetycznym [33], tzn. leży bardzo wysoko w hierarchii Kleene-Mostowskiego i nie ma mowy o tym, by wyrocznia odpowiadająca na pytanie o przynależność do takiego zbioru, dała się przybliżyć jakimś algorytmem. Dla klasy kolejek priorytetowych nad skończonymi zbiorami elementów, zbiór ten leży znacznie niżej, jest to dopełnienie pewnego zbioru rekurencyjnie przeliczalnego. Ale i ten zbiór także nie może mieć mechanicznej wyroczni. Cóż więc pozostaje? Trzeba podjąć trud dowodzenia pewnej formuły pierwszego rzędu.

Programistów interesują własności algorytmiczne, własności programów. Poznawanie tych własności daje możność stosowania ich w dowodach nowych faktów. Mogą to być twierdzenia o tradycyjnym statycznym charakterze, mogą to też być własności programów. Tak, naszym zdaniem, powinien wyglądać naturalny rozwój wiedzy o strukturze danych, rozwój algorytmicznych teorii struktur danych. Taki proces może wnosić nowe fakty do teorii opartych na języku klasycznej logiki pierwszego rzędu. Na przykład, istotnym wzbogaceniem teorii liczb naturalnych byłaby odpowiedź na pytanie: czy program

```
while n ≠ 1
do if n podzielne przez 2
   then
        n := n div 2
   else
        n := n * 3 + 1
   fi
od
```

zatrzymuje swoje obliczenia dla każdej liczby naturalnej n? Z drugiej strony, być może właściwym podejściem do tego problemu okaże się wykorzystanie faktu, że program

```
begin z := 0; while z ≠ x do z := z + 1 od
```

zatrzymuje się dla każdej liczby naturalnej x. Nie jest pewne, czy własność zatrzymywania się poprzedniego programu jest konsekwencją aksjomatów Peano. Na pewno jednak algorytmiczne aksjomaty liczb naturalnych stanowią zbiór aksjomatów wystarczający na to, by własność ta była jego *semantyczną konsekwencją.*

Zbieranie faktów prawdziwych, w interesujących nas klasach struktur danych, jest zadaniem na długie lata dla sporej społeczności badaczy nie może być tu mowy o powierzeniu tego zadania automatowi lub wyroczni.

O rachunku Dijkstry 8.3

W roku 1975 Dijkstra [17] zaproponował rachunek tzw. transformacji predykatów. W rozumowaniach swoich wychodzi on z obserwacji, że programy mogą być pojmowane jako transformacje warunków, tzn. predykatów spełnianych przez stany. W szczególności zajął się *najsłabszym warunkiem wstępnym*. W wydanej wkrótce potem, popularnej dziś książce [18] przedstawił swoje poglądy i wiele przykładów analizy semantycznych własności programów, analizy opartej na aksjomatach najsłabszego warunku wstępnego. W książce rozważa się język programowania tzw. instrukcji dozorowanych, niedeterministycznych. Tu przedstawimy rachunek Dijkstry dostosowany do języka programów iteracyjnych, deterministycznych. Uważamy bowiem, że w ten sposób łatwiej będzie skoncentrować się na sprawach najważniejszych. Nieformalne określenie najsłabszego warunku wstępnego w pracy [18] brzmi jak następuje: Warunek charakteryzujący zbiór wszystkich stanów początkowych, takich że podjęcie obliczeń w dowolnym z nich doprowadzi z pewnością do ich pomyślnego zakończenia w stanie końcowym spełniającym zadany warunek ostateczny, nazywamy *najsłabszym warunkiem wstępnym dla danego warunku ostatecznego*. W pracy Dijkstry nie znajdujemy innej definicji tego pojęcia. (Zwracamy uwagę czytelnika na określenie najsłabszego warunku wstępnego, które podaliśmy w rozdz. 3.4). Natomiast autor podaje własności, jakie ma spełniać najsłabszy warunek wstępny. Po przetłumaczeniu ich na język używany w tej książce brzmią one następująco:

(Wł1) $\mathrm{M}\mathbf{false} = \mathbf{false}$

(Wł2) $$\frac{(\alpha \Rightarrow \beta)}{(\mathrm{M}\alpha \Rightarrow \mathrm{M}\beta)}$$

(Wł3) $\mathrm{M}(\alpha \wedge \beta) \equiv \mathrm{M}\alpha \wedge \mathrm{M}\beta$

(Wł4) $\mathrm{M}(\alpha \vee \beta) \equiv \mathrm{M}\alpha \vee \mathrm{M}\beta$

(Wł5) W książce [18] własność tę sformułowano następująco:

Dla dowolnego programu M i dowolnego nieskończonego ciągu formuł $\alpha_0, \alpha_1, \alpha_2, \ldots$, takiego że dla wszystkich stanów jest spełniona własność

$$(\alpha_r \Rightarrow \alpha_{r+1}) \quad \text{dla } r \geqslant 0$$

mamy dla wszystkich stanów

$$\mathrm{M}((\exists r \geqslant 0)\,\alpha_r) = ((\exists s \geqslant 0)\,\mathrm{M}\alpha_s)$$

Z wysłowieniem tej własności (zwanej także własnością ciągłości) w języku logiki algorytmicznej mamy nieco kłopotów. W przyjętym przez nas języku nie występuje nieskończona alternatywa formuł. Notacja $(\exists r \geqslant 0\,\alpha_r)$ użyta w książce jest myląca, jednak jej semantyczny sens jest dla nas oczywisty: formuła ta jest spełniona wówczas, gdy istnieje takie $r > 0$, że jest spełniona

formuła α_r, r-ta formuła w ciągu formuł $\alpha_1, \alpha_2, \ldots$. Właściwie należałoby stwierdzić, że mamy do czynienia z nieskończonymi alternatywami i napisać

$$\mathrm{M}(\bigvee_{r\geq 0}\alpha_r) \equiv (\bigvee_{s\geq 0}\mathrm{M}\alpha_s)$$

Zwróćmy też uwagę na to, że zmienna r nie musi wcale występować w żadnej z tych formuł. W języku pierwszego rzędu poprawna formuła o zbliżonej formie to $\exists_{r\geq 0}\alpha(r)$, ale ma ona zupełnie inny sens i nie ma związku z ciągiem $\{\alpha_r\}$ formuł, o którym mówi się w przesłance własności (Wł5).

Wszystko to prowadzi do konkluzji, że (Wł5) jest regułą wnioskowania o nieskończonej liczbie przesłanek. Należałoby ją więc zapisać następująco:

$$\frac{\{\alpha_r \Rightarrow \alpha_{r+1}\}_{r\in N}}{\mathrm{M}(\bigvee_{r\geq 0}\alpha_r) = (\bigvee_{s\geq 0}\mathrm{M}\alpha_s)}$$

Jeżeli dla każdej liczby naturalnej r prawdziwa jest implikacja $(\alpha_r \Rightarrow \alpha_{r+1})$, to prawdziwa jest też równoważność

$$\mathrm{M}(\bigvee_{r\geq 0}\alpha_r) \equiv (\bigvee_{s\geq 0}\mathrm{M}\alpha_s)$$

W dalszym ciągu pokażemy, że w przypadku programów deterministycznych, konkluzja w tej regule wnioskowania jest tautologią.

Omawiając po kolei konstrukcje występujące w języku programowania Dijkstra podaje dalsze własności najsłabszego warunku wstępnego i traktuje je jako definicje znaczenia odpowiednich konstrukcji programotwórczych języka programowania. Bez szkody dla ogólności naszych rozważań możemy, stosowany przez autora język instrukcji dozorowanych, zastąpić językiem programów iteracyjnych, deterministycznych. Obok instrukcji przypisania, jako programy atomowe, wystąpią jeszcze dwie instrukcje **zostaw** i **zerwij**. Właściwie można się bez nich obejść lub przyjąć, że **zostaw** jest oznaczeniem programu $x := x$ dla pewnej dowolnie ustalonej zmiennej x, a **zerwij** jest oznaczeniem dla instrukcji, która nigdy nie daje wyniku, np. $x := x/0$, albo programu **while true do od**, ale oznaczenia te mogą być przydatne w dalszych naszych rozważaniach.

Oprócz wyliczonych pięciu własności autor podaje jeszcze sześć schematów aksjomatów, których zadaniem jest zdefiniowanie semantyki konstrukcji występujących w języku programowania.

(A1) **zostaw** $\alpha \equiv \alpha$

(A2) **zerwij** $\alpha \equiv$ **false**

(A3) $(x := \tau)\alpha \equiv \alpha(x/\tau)$

(A4) **begin** M_1; M_2 **end** $\alpha \equiv \mathrm{M}_1(\mathrm{M}_2\,\alpha)$

(A5) **if** γ **then** M_1 **else** M_2 **fi** $\alpha = ((\gamma \wedge \mathrm{M}_1\alpha) \vee (\neg\gamma \wedge \mathrm{M}_2\alpha))$

(A6) dla instrukcji iteracyjnej Dijkstra podaje następującą definicję najsłabszego warunku wstępnego:

while γ **do** M **od** $\alpha \equiv (\exists k \geqslant 0)\, H_k(\alpha)$

gdzie formuły H_k są określone indukcyjnie jak następuje:

$$H_0(\alpha) = \alpha \wedge \neg\gamma$$
$$H_{k+1}(\alpha) \equiv (\textbf{if } \gamma \textbf{ then } \mathrm{M} \textbf{ fi})\, H_k(\alpha) \vee H_0(\alpha).$$

Przypatrzmy się bliżej tym formułom. Formuła H_0 nie budzi żadnych wątpliwości. Czym jest formuła H_1? Stosując aksjomat (A5) i proste przekształcenia rachunku zdań widzimy, że

$$\begin{aligned} H_1(\alpha) &\equiv (\textbf{if } \gamma \textbf{ then } M \textbf{ fi})(\alpha \wedge \neg\gamma) \vee (\alpha \wedge \neg\gamma) \\ &\equiv ((\gamma \wedge M(\alpha \wedge \neg\gamma) \vee \neg\gamma \wedge (\alpha \wedge \neg\gamma) \vee (\alpha \wedge \neg\gamma) \\ &\equiv ((\gamma \wedge M \neg\gamma \wedge M\alpha) \vee (\neg\gamma \wedge \alpha)) \end{aligned}$$

Ogólnie, kolejne formuły H_k wyrażają się jako coraz dłuższe alternatywy coraz dłuższych koniunkcji

$$\begin{aligned} H_k(\alpha) &= ((\neg\gamma \wedge \alpha) \vee \\ &\quad \vee (\gamma \wedge M \neg\gamma \wedge M\alpha) \vee \\ &\quad \ldots \\ &\quad \vee (\gamma \wedge M\gamma \wedge MM\gamma \wedge \ldots \wedge M^{k-1}\gamma \wedge M^k \neg\gamma \wedge M^k\alpha)) \equiv \\ &\equiv (\textbf{if } \gamma \textbf{ then } \mathrm{K} \textbf{ fi})^k (\neg\gamma \wedge \alpha) \end{aligned}$$

UWAGA

Człon $H_0(\alpha)$ w indukcyjnej definicji formuł $H_k(\alpha)$ jest zbędny. ■

Zauważmy, że każda z tych formuł ma inną strukturę i że formuły H_k nie zawierają zmiennej k. Stosowanie notacji z kwantyfikatorem $(\exists_{k \geqslant 0} H_k(\alpha))$ należy więc traktować jako nieformalny zapis nieskończonej alternatywy formuł H_k, tym bardziej, że każda z tych formuł ma inną budowę. Przypomnijmy, że w rachunku kwantyfikatorów pierwszego rzędu wyrażenie postaci $(\exists x)\alpha(x)$ jest formułą pierwszego rzędu wtedy, gdy $\alpha(x)$ jest formułą pierwszego rzędu. Mamy tu do czynienia z pomieszaniem formuł i ich oznaczeń. Wyrażenie H_k jest oznaczeniem formuły, a nie formułą. Z rachunku kwantyfikatorów [45] zaś przypominamy sobie, że formuła z kwantyfikatorem $(\exists x)\alpha(x)$ ma wartość równą kresowi górnemu wartości formuł $\alpha(x/\tau)$, gdzie τ jest dowolnym termem (lub, jeśli czytelnik woli, wyrażeniem arytmetycznym). A choć długości tych formuł mogą się różnić i chociaż nie istnieje ograniczenie na długość formuły $\alpha(x/\tau)$, to każda z tych formuł ma jednakową strukturę logiczną, tyle samo operatorów logicznych i kwantyfikatorów. W przypadku formuł H_k ich struktura komplikuje się ze wzrostem k. Nie można więc stosować kwantyfikatora $(\exists k)$. Zresztą zmienna k wcale nie występuje w tych formułach.

Chcielibyśmy zwrócić uwagę czytelnika na fakt, że semantyczna treść aksjomatu A6 jest wiernie wyrażona przez formułę

$$(*) \quad \textbf{while } \gamma \textbf{ do } K \textbf{ od } \alpha \equiv \bigcup \textbf{if } \gamma \textbf{ then } K \textbf{ fi } (\alpha \wedge \neg\gamma) \quad (*)$$

Prawdziwość tej formuły wykazaliśmy w lemacie 3.3. Na mocy twierdzenia (4.2) o pełności wiemy, że równoważność ta ma też dowód z aksjomatów logiki algorytmicznej. W dowodzie formuły (*) stosujemy aksjomat Ax21 i reguły R3 i R4 logiki algorytmicznej. Z aksjomatu wynika, że dla każdej liczby naturalnej $i \in N$

$$(\textbf{if } \gamma \textbf{ then } K \textbf{ fi})^i (\neg\gamma \wedge \alpha) \Rightarrow \textbf{while } \gamma \textbf{ do } K \textbf{ od } \alpha$$

Dowód tego faktu podaliśmy w przykładzie 4.4. Po zastosowaniu reguły wnioskowania R4 wyprowadzamy implikację

$$\bigcup \textbf{if } \gamma \textbf{ then } K \textbf{ fi} (\alpha \wedge \neg\gamma) \Rightarrow \textbf{while } \gamma \textbf{ do } K \textbf{ od } \alpha$$

Implikację odwrotną dowodzimy równie łatwo. Dla każdej liczby naturalnej i następująca formuła jest tautologią

$$(\textbf{if } \gamma \textbf{ then } K \textbf{ fi})^i (\neg\gamma \wedge \alpha) \Rightarrow (\textbf{if } \gamma \textbf{ then } K \textbf{ fi})^i (\neg\gamma \wedge \alpha)$$

Z aksjomatu Ax22 i rachunku zdań wynika, że dla każdej liczby naturalnej i tautologią jest implikacja

$$(\textbf{if } \gamma \textbf{ then } K \textbf{ fi})^i (\neg\gamma \wedge \alpha) \Rightarrow \bigcup \textbf{if } \gamma \textbf{ then } K \textbf{ fi } (\alpha \wedge \neg\gamma)$$

Fakt ten pozwala nam zastosować regułę R3 i udowodnić formułę

$$\textbf{while } \gamma \textbf{ do } K \textbf{ od } \alpha \Rightarrow \bigcup \textbf{if } \gamma \textbf{ then } K \textbf{ fi } (\alpha \wedge \neg\gamma)$$

Łatwo zauważyć, że formuły (Wł3), (Wł4), (A3), (A4), (A5) są znanymi już nam aksjomatami AL. Można zadawać sobie pytanie czy formalizm Dijkstry i logika algorytmiczna są równoważne. Otóż, przyjmując do wiadomości uwagi poczynione na temat języków programowania i formuł z nieskończonymi alternatywami, potrafimy wyprowadzić wszystkie własności wymienione przez Dijkstrę z aksjomatów logiki algorytmicznej. Co więcej *własność ciągłości* (Wł5) jest do udowodnienia w rozszerzeniu logiki algorytmicznej dopuszczającym nieskończone alternatywy. Bez żadnych dodatkowych założeń udowodnimy, że

$$M(\bigvee_{r \geqslant 0} \alpha_r) = (\bigvee_{s \geqslant 0} M\alpha_s)$$

Przypuśćmy, że dla pewnego dowolnie wybranego wartościowania zachodzi $\mathbf{A}, v \models M(\bigvee_{r \geqslant 0} \alpha_r)$. Wtedy, z definicji semantyki formuł postaci $K\beta$, zachodzi też $\mathbf{A}, \mathbf{M}_{\mathbf{A}}(v) \models \bigvee_{r \geqslant 0} \alpha_r$. Ale formuła $\bigvee_{r \geqslant 0} \alpha_r$ jest spełniona wówczas, gdy dla pewnego $s \geqslant 0$ jest spełniona formuła α_s. Mamy więc $\mathbf{A}, \mathbf{M}_{\mathbf{A}}(v) \models \alpha_s$ czyli, ponownie stosując definicję znaczenia formuł postaci $K\beta$, $\mathbf{A}, v \models M\alpha_s$ dla

pewnego $s \geqslant 0$. A jeżeli tak, to $\mathbf{A}, v \models (\bigvee_{s \geqslant 0} \mathrm{M}\alpha_s)$. Implikacji w drugą stronę dowodzimy równie łatwo.

Następne nasze uwagi dotyczą własności (Wł1) zwanej przez Dijkstrę *prawem wykluczania cudów*. Wobec tego, że implikacja **false** $\Rightarrow$ M **false** jest tautologią należy tylko udowodnić implikację odwrotną (M**false**) $\Rightarrow$ **false**. Przypomnijmy, że $(\alpha \wedge \neg\alpha) =$ **false**. Zauważmy, że dla dowolnego programu M i dowolnej formuły α

$$\mathrm{M}(\alpha \wedge \neg\alpha) \equiv \mathrm{M}\alpha \wedge \mathrm{M}\neg\alpha$$

jest to aksjomat Ax14.
Na mocy innego aksjomatu $\mathrm{M}\neg\alpha \Rightarrow \neg\mathrm{M}\alpha$. A więc

$$\mathrm{M}(\alpha \wedge \neg\alpha) \Rightarrow \mathrm{M}\alpha \wedge \neg\mathrm{M}\alpha$$

co, jak można łatwo sprawdzić, jest konsekwencją dwu ostatnich formuł. Po zastosowaniu aksjomatu $(\alpha \wedge \neg\alpha) \Rightarrow \beta$ otrzymujemy potrzebną nam implikację M **false** $\Rightarrow$ **false** co kończy formalny dowód prawa wykluczania cudów z aksjomatów AL.

Warto jeszcze powrócić do własności (Wł2) i (Wł5) — ciągłości, są to reguły wnioskowania. Własność ciągłości jest przy tym regułą o nieskończonej ilości przesłanek. Wykazaliśmy przed chwilą, że przesłanki te nie są niezbędne dla zachowania (Wł5) (rozdzielność programu z nieskończoną alternatywą). Ale sama nieskończona alternatywa powinna być jakoś scharakteryzowana i nie jest trudne do odgadnięcia, że w tym celu jest potrzebna jakaś reguła o nieskończonej ilości przesłanek. W logice algorytmicznej rozszerzonej o nieskończone alternatywy można wyprowadzić równoważność (A6) z aksjomatu Ax21 i reguły R3 (por. definicję 4.2). Nie sądzimy jednak, by warto było dokonywać rozszerzenia języka, które jest zbędne. W świetle naszych wcześniejszych rozważań widać wyraźnie, iż rozważania dotyczące programów iteracyjnych nie wymagają wprowadzania do języka formuł o nieskończonej długości.

Wracając do pytania o równoważność rachunku Dijkstry i logiki algorytmicznej, odpowiedź na nie brzmi: tak, oba te formalizmy są równoważne, jeśli przyjąć do wiadomości nasze uwagi o (Wł5) oraz uzupełnić ten rachunek aksjomatami i regułami wnioskowania dla kwantyfikatorów, i funktorów logicznych.

Nasuwa się z kolei pytanie: czy stwierdzenie Dijkstry, iż własności (Wł1)–(Wł5) i (A1)–(A6) definiują semantykę, można uznać za uzasadnione. Harel [24] badał podobne zagadnienie i doszedł do wniosku, że spośród wielu możliwych strategii przeglądania drzewa niedeterministycznych obliczeń jest tylko jedna spełniająca te wszystkie własności. A więc, konkluduje Harel, aksjomaty Dijkstry definiują semantykę obliczeń niedeterministycznych. Nam sprawa ta nie wydawała się do końca wyjaśniona i stąd wzięły się wyniki zawarte w p. 4.5. Wskazują one na mocniejsze konsekwencje przyjęcia

aksjomatów algorytmicznych. Znaczenie konstrukcji złożenia, rozgałęzienia i iteracji, a także instrukcji atomowych (przypisania) jest jednoznacznie określone przez wymaganie, by realizacja języka programowania spełniała aksjomaty AL. Badania, czy podobny rezultat da się także osiągnąć dla logiki algorytmicznej programów niedeterministycznych, są trudniejsze i w tej chwili nie znamy jeszcze pełnej odpowiedzi.

8.4 Logika dynamiczna

Logikę dynamiczną, w skrócie DL, wprowadził w roku 1976 Pratt dla badania semantycznych własności programów [43]. Najważniejsze idee DL wyłożył Harel w pracy [24]. Zgodnie z intencją twórców DL ma na celu dostarczenie precyzyjnego aparatu matematycznego do badania własności programów. Jednym z podstawowych jej zadań jest skonstruowanie podstaw matematycznych umożliwiających formułowanie i dowodzenie zdań postaci „dany program jest poprawny". Twórcy DL nie przewidują natychmiastowej stosowalności teorii przez nich rozwijanej w rozwoju metod konstruowania i dowodzenia własności programów. Twórcy logiki dynamicznej sądzą, że jej zastosowania będą miały charakter pośredni i długofalowy jako, że prowadzą do wyodrębnienia i lepszego zrozumienia tych pojęć i metod, które są podstawowe w procesie wnioskowania o własnościach programów. Uważają oni ponadto, że badania w logice dynamicznej mają implikacje w konwencjonalnych dziedzinach logiki i filozofii, ponieważ dostarczają narzędzi do wnioskowania o ogólnych sytuacjach dynamicznych.

Nazwa logika dynamiczna doskonale podkreśla różnicę między statyczną logiką klasyczną a logikami algorytmicznymi, do których DL też się zalicza. Aby obliczyć wartość formuły klasycznej odwołujemy się do jednego stanu — danego wartościowania zmiennych. W logikach programów, a w szczególności w DL, przy obliczaniu wartości formuły odwołujemy się do stanów pamięci, które dynamicznie pojawiają się na różnych etapach obliczenia programu.

W logice dynamicznej, tak jak w logice algorytmicznej, pojęcie programu jest jednym ze składników języka. Programy występują explicite w formułach. Jednakże samo pojęcie programu jest nieco inne niż przedstawiono w rozdz. 3. Jedna z istotnych różnic polega na przyjęciu operacji niedeterministycznego wyboru jako operacji na programach.

Niech $L_=$ będzie językiem pierwszego rzędu (bez zmiennych zdaniowych) (por. p. 2.5), w którym występuje dwuargumentowy symbol równości $=$.

DEFINICJA 8.5

Zbiór DL-programów (tzw. programów regularnych) jest najmniejszym zbiorem zamkniętym ze względu na następujące reguły tworzenia programów:

(1) wszystkie instrukcje przypisania postaci $x := \tau$ dla dowolnej zmiennej indywiduowej x i dowolnego termu τ (por. definicję 2.21), są DL-programami;

(2) jeżeli α jest formułą klasyczną, to $\alpha?$ jest DL-programem;

(3) jeżeli K i M są DL-programami, to wyrażenia (K; M), $(K \cup M)$, K* są DL-programami.

DL-program postaci $\alpha?$ nazywamy *testem*, a DL-program postaci K* nazywamy *iteracją programu* K. Symbol $\cup$ jest operacją niedeterministycznego wyboru. Jeżeli K, M są DL-programami, to $(K \cup M)$ jest DL-programem niedeterministycznym. ■

PRZYKŁAD 8.3

Jeżeli γ jest formułą pierwszego rzędu w języku $L_=$, a K, M są DL-programami, to wyrażenie

$$((\gamma?; K) \cup (\neg\gamma?; M))$$

jest DL-programem. □

DEFINICJA 8.6

Zbiór DL-formuł jest najmniejszym zbiorem wyrażeń spełniającym następujące warunki:

(1) dla dowolnego n-argumentowego predykatu ϱ i dowolnych termów $\tau_1, \ldots, \tau_n$ wszystkie formuły atomowe postaci

$$\varrho(\tau_1, \ldots, \tau_n)$$

są DL — formułami oraz

(2) jeżeli α, β są DL-formułami, a x jest zmienną indywiduową, to wyrażenia $\neg\alpha, (\alpha \vee \beta), (\alpha \wedge \beta), (\alpha \Rightarrow \beta), (\exists x)\alpha, (\forall x)\alpha$ są DL-formułami,

(3) jeżeli K jest programem, a α jest DL-formułą, to wyrażenia $\langle K \rangle \alpha$ $[K]\alpha$ są DL-formułami. ■

PRZYKŁAD 8.4

Jeżeli α jest DL-formułą, a K, L są DL-programami, to wyrażenie

$$(\langle K \rangle \alpha \Rightarrow \langle K \cup M \rangle \alpha)$$

jest DL-formułą. □

Idea semantyki DL-programów jest oparta na pojęciu relacji wejścia-wyjścia przypisanej programowi.

Niech **A** oznacza system relacyjny tego samego typu co rozważany język pierwszego rzędu (por. p. 2.4). Dla każdego funktora n-argumentowego

φ istnieje n-argumentowa funkcja (całkowita) $\varphi_{\mathbf{A}}$ w systemie **A**, stanowiąca jego interpretację, oraz dla każdego m-argumentowego predykatu ϱ istnieje m-argumentowa relacja, której funkcja charakterystyczna $\varrho_{\mathbf{A}}$ jest interpretacją predykatu ϱ. System relacyjny **A** wyznacza jednoznacznie interpretację DL-programów jako relację wejścia-wyjścia między stanami pamięci, tzn. wartościowaniami. Dla dowolnego DL-programu M oznaczmy odpowiadającą mu relację przez $\mathrm{M}_{\mathbf{A}}$.

$$(\mathrm{K};\mathrm{M})_{\mathbf{A}} \stackrel{\mathrm{df}}{=} \{(v, v'): (\exists v'')(v, v'') \in \mathrm{K}_{\mathbf{A}} \text{ i } (v'', v') \in \mathrm{M}_{\mathbf{A}}\}$$
$$(\mathrm{K} \cup \mathrm{M})_{\mathbf{A}} \stackrel{\mathrm{df}}{=} \{(v, v'): (v, v'') \in \mathrm{K}_{\mathbf{A}} \text{ lub } (v, v') \in \mathrm{M}_{\mathbf{A}}\}$$
$$(x := \tau)_{\mathbf{A}} \stackrel{\mathrm{df}}{=} \{(v, v'): v = v' \text{ off } x \text{ i } v'(x) = \tau_{\mathbf{A}}(v)\}$$
$$(\alpha?)_{\mathbf{A}} \stackrel{\mathrm{df}}{=} \{(v, v): \mathbf{A}, v \models \alpha\}$$
$$(\mathrm{K}^*)_{\mathbf{A}} \stackrel{\mathrm{df}}{=} \{(v, v'): (\exists i \geqslant 0)(v, v') \in \mathrm{K}^i_{\mathbf{A}}\}$$

W powyższych definicjach K, M są dowolnymi DL-programami, x jest zmienną indywiduową, α dowolną DL-formułą, a τ dowolnym termem. Tak jak w poprzednich rozdziałach tej książki, jeżeli $(v, v') \in \mathrm{K}_{\mathbf{A}}$, to v' będziemy nazywać wynikiem programu K dla danych początkowych v.

Zauważmy, że w wyniku przyjęcia dwóch operacji niedeterministycznych, $\cup$ i $*$, relacja przypisana DL-programom nie musi być funkcją częściową, DL-program może mieć więcej niż jeden wynik.

PRZYKŁAD 8.5

M: $((x := 0); (x := x + 1)^*)$

DL-program M zdefiniowany wcześniej ma w strukturze liczb naturalnych N ze zwykłą interpretacją symboli $+$, 1, 0 nieskończenie wiele wyników. Wartością zmiennej x po wykonaniu programu M może być dowolna liczba naturalna. □

Zauważmy ponadto, że DL-program może nie mieć wyniku. DL-program postaci (**false?**) ma nieokreślony wynik dla dowolnych danych początkowych.

Podobnie jak w rozdz. 3 możemy wprowadzić pojęcie obliczenia DL-programu. Wymieniamy dość oczywiste różnice. Bezpośrednim następnikiem konfiguracji $\langle v, (\mathrm{K} \cup \mathrm{M})\rangle$ jest zarówno konfiguracja $\langle v, \mathrm{K}\rangle$, jak i $\langle v, \mathrm{M}\rangle$, a bezpośrednim następnikiem konfiguracji $\langle v, \mathrm{K}^*\rangle$ jest konfiguracja postaci $\langle v, \mathrm{K}^i\rangle$ dla dowolnej liczby naturalnej i.

Zauważmy, że DL-program nie ma obliczeń nieskończonych. Obliczenia DL-programów można jedynie podzielić na skończone udane, tzn. dające wynik, i skończone nieudane.

PRZYKŁAD 8.6

Rozważmy DL-program postaci

K: $((\gamma?; s) \cup (\neg\gamma?; s'))$

gdzie s, s' są instrukcjami przypisania, a γ jest formułą otwartą pierwszego rzędu. Relacja wejścia-wyjścia zdefiniowana przez ten program w dowolnej strukturze danych dla języka $L_=$ jest taka sama, jak relacja wejścia-wyjścia zdefiniowana przez program

M: **if** γ **then** s **else** s' **fi**

tzn. dokładnie te same pary wartościowań należą do interpretacji obu programów. Jednakże zbiory obliczeń programów są różne. Dla dowolnego wartościowania v takiego, że $\mathbf{A}, v \models \gamma$, program M ma tylko jedno obliczenie skończone i udane:

$\langle v, M\rangle, \langle v, s\rangle, \langle s_A(v), \emptyset\rangle$

Natomiast program K może mieć dwa obliczenia: skończone udane Θ_1 i skończone nieudane Θ_2,

Θ_1: $\langle v, K\rangle, \langle v, \gamma?; s\rangle, \langle v, s\rangle, \langle s_A(v), \emptyset\rangle$
Θ_2: $\langle v, K\rangle, \langle v, \neg\gamma?; s'\rangle$ □

PRZYKŁAD 8.7

DL-program

$((\gamma?, M)^*; \neg\gamma?)$

wyznacza taką samą relację wejścia-wyjścia, jak program

while γ **do** M **od**

Jednakże, program **while true do** M **od** ma tylko jedno obliczenie nieskończone, podczas gdy ((**true**?; M)*; ¬**true**?) ma nieskończenie wiele skończonych obliczeń nieudanych. □

Relacja $\models$ jest rozszerzeniem klasycznej relacji spełniania o następujące definicje:

$\mathbf{A}, v \models \langle M\rangle \alpha$ wtw $(\exists v')((v, v') \in M_A$ i $\mathbf{A}, v' \models \alpha)$
$\mathbf{A}, v \models [M]\alpha$ wtw $(\forall v')((v, v') \in M_A \Rightarrow \mathbf{A}, v' \models \alpha)$

PRZYKŁAD 8.8

Wyrażenie

$[(x := x+1)^*](\langle((y = y+1) \cup (y = y-1))^*\rangle y = x)$

st DL-formułą prawdziwą w strukturze liczb całkowitych ze zwykłą ıterpretacją symboli $+$, $-$, $=$. □

Za pomocą formuł logiki dynamicznej możemy wyrazić wiele własności rogramów, o których była wcześniej mowa.

$\langle M \rangle$**true** wyraża istnienie skończonego, udanego obliczenia progra- ıu M;

$\langle M \rangle \alpha$ — istnieje wynik programu M spełniający warunek α;

$(\alpha \Rightarrow [M]\beta)$ — program M jest częściowo poprawny ze względu na ′arunek początkowy α i warunek końcowy β.

$\langle M \rangle \langle \mathbf{K}(\mathbf{x}/\mathbf{y}) \rangle (\mathbf{x} = \mathbf{y})$ — programy K i M są równoważne ze względu na biór zmiennych **x**.

Jednym z głównych wyników logiki dynamicznej jest twierdzenie o tzw. rytmetycznej pełności [24]. Twierdzenie to pozwala zastąpić rozumowania emantyczne w strukturach zawierających arytmetykę, syntaktycznymi do- vodami w systemie aksjomatycznym, w którym wszystkie reguły wnio- kowania są skończone. Podstawą do dalszych rozważań jest definicja truktury arytmetycznej [24].

DEFINICJA 8.7

ɔtrukturą arytmetyczną nazywamy dowolny system relacyjny, którego uni- versum zawiera zbiór liczb naturalnych N i taki, że jego sygnatura zawiera

(1) dwuargumentowe funkcje $+$, $*$, które rozważane w zbiorze liczb ıaturalnych, są odpowiednio operacjami dodawania i mnożenia;

(2) stałe 0, 1, które są odpowiednio zerem i jedynką w zbiorze N;

(3) jednoargumentową relację *nat* taką, że dla dowolnego elementu ıniwersum struktury x, $nat(x)$ wtw x jest liczbą naturalną;

(4) trójargumentową relację $R(x, i, y)$ taką, że $R(x, i, y)$ wtw x jest i-tym ɛlementem ciągu o kodzie y. ■

Dla klasy bogatych struktur, jakimi są struktury arytmetyczne, zachodzi następująca własność:

LEMAT 8.4

Dla dowolnej DL-formuły α istnieje formuła β w języku pierwszego rzędu $L_=$ taka, że

$$\mathbf{A} \models (\alpha \equiv \beta)$$

dla dowolnej struktury arytmetycznej **A**. ■

Lemat ten stwierdza, że w gruncie rzeczy, w strukturach arytmetycznych (za pomocą języka DL) można opisać tylko te własności, które są wyrażalne w języku pierwszego rzędu.

Oznaczmy przez AxDL następujący zbiór DL-formuł:

(T) wszystkie tautologie klasycznego rachunku zdań,

(←R) $[x := \tau]\alpha = \alpha(x/\tau)$ dla dowolnej formuły α języka $L_=$ ze zmienną wolną x,

(?R) $[\beta?]\,\alpha \equiv (\beta \Rightarrow \alpha)$

(;R) $[K; M]\,\alpha \equiv [K]([M]\,\alpha)$

($\cup$R) $[K \cup M]\,\alpha \equiv ([K]\,\alpha \wedge [M]\,\alpha)$

(R) $[K]\,\alpha \equiv \neg\langle K\rangle \neg\alpha$

Niech ponadto r — DL oznacza następujący zbiór reguł wnioskowania:

$$\text{(MP)}\quad \frac{\alpha,\ (\alpha \Rightarrow \beta)}{\beta}$$

$$(\exists)\quad \frac{(\alpha \Rightarrow \beta)}{((\exists x)\alpha \Rightarrow (\exists x)\,\beta)}$$

$$\text{(G)}\quad \frac{(\alpha \Rightarrow \beta)}{([K]\,\alpha \Rightarrow [K]\,\beta)}$$

$$(*)\quad \frac{(\alpha \Rightarrow [K]\,\alpha)}{(\alpha \Rightarrow [K^*]\,\alpha)}$$

$$\text{(C*)}\quad \frac{(nat\,(n) \Rightarrow (\gamma\,(n+1) \Rightarrow \langle K\rangle\,\gamma\,(n)))}{(nat\,(n) \Rightarrow (\gamma\,(n) \Rightarrow \langle K^*\rangle\,\gamma\,(0)))}$$

W tych schematach γ jest formułą pierwszego rzędu ze zmienną wolną n taką, że $n \notin V(K)$, a α i β są dowolnymi DL-formułami, a K i M są dowolnymi DL-programami.

System formalny wyznaczony przez powyższe aksjomaty i reguły wnioskowania nazywamy *logiką dynamiczną* DL. Pojęcie dowodu formalnego w logice dynamicznej jest takie, jak w logice klasycznej. Dowód jest ciągiem formuł prowadzącym od aksjomatu do dowodzonej formuły i takim, że każdy jego następny element wynika z poprzednich przez zastosowanie pewnej reguły wnioskowania.

PRZYKŁAD 8.9

Następujący ciąg formuł jest dowodem formalnym formuły

$$[M^*](\alpha \Rightarrow [M]\,\alpha) \Rightarrow (\alpha \Rightarrow [M^*]\,\alpha)$$

zwanej aksjomatem indukcji w logice dynamicznej DL:

$$\vdash (((\alpha \Rightarrow [M]\alpha) \wedge \alpha) \Rightarrow \alpha) \qquad \{(T)\}$$
$$\vdash ([M^*]((\alpha \Rightarrow [M]\alpha) \wedge \alpha) \Rightarrow [M^*]\alpha) \qquad \{(G)\}$$
$$\vdash ([M^*](\alpha \Rightarrow [M]\alpha) \Rightarrow (\alpha \Rightarrow [M^*]\alpha)) \qquad \square$$

Niech dla dowolnej struktury arytmetycznej **A**, $F(\mathbf{A})$ oznacza zbiór wszystkich formuł pierwszego rzędu prawdziwych w **A**. Następujące twierdzenie wskazuje na związek semantycznej i syntaktycznej operacji konsekwencji w logice dynamicznej. Twierdzenie to nazywamy twierdzeniem o arytmetycznej pełności DL [24].

TWIERDZENIE 8.1

Dla dowolnej formuły α i dowolnej struktury arytmetycznej **A**,

$$\mathbf{A} \models \alpha \quad \text{wtw} \quad F(\mathbf{A}) \vdash \alpha \qquad \blacksquare$$

Na zakończenie tej krótkiej prezentacji logiki dynamicznej przedstawimy przykład dowodu formalnego w tym systemie. Dowód ten pochodzi z pracy [25].

PRZYKŁAD 8.10

Udowodnimy, stosując przedstawiony system dedukcyjny, że formuła

$$(\alpha \Rightarrow \langle (K \cup M)^* \rangle \beta)$$

gdzie

$$\alpha = (z = 1 \wedge x = a \wedge y = b)$$
$$\beta = (y = 0 \wedge z = a^b)$$
$$K: \quad ((y > 0 \wedge p(y))?; \quad x := x^2; \; y := y/2)$$
$$M: \quad (np(y)?; z := z * x; \; y := y - 1)$$

jest prawdziwa we wszystkich strukturach arytmetycznych, w których predykaty $p(y)$, $np(y)$ wyrażają odpowiednio własności: y jest liczbą parzystą y jest liczbą nieparzystą.

Oznaczmy przez $\delta(n)$ formułę pierwszego rzędu postaci

$$(z * x^y = a^b \wedge n = |\log_2 y| + bin(y))$$

dzie $bin(y)$ oznacza liczbę jedynek w rozwinięciu dwójkowym y. Formuły

$$((z * x^y = a^b \wedge 0 = |\log_2 y| + bin(y)) \Rightarrow (y = 0 \wedge z = a^b))$$
$$((z = 1 \wedge x = a \wedge y = b) \Rightarrow (\exists n)(z * x^y = a^b \wedge n = |\log_2 y| + bin(y)))$$

są oczywiście prawdziwe w każdej arytmetycznej strukturze, a więc na mocy twierdzenia 8.1,

$$\mathrm{F}(\mathbf{A}) \models (\delta(0) \Rightarrow \beta) \tag{1}$$
$$\mathrm{F}(\mathbf{A}) \vdash (\alpha \Rightarrow (\exists n)\,\delta(n)) \tag{2}$$

Dalej pokażemy, że dla dowolnego naturalnego n, formuła

$$\mathrm{F}(\mathbf{A}) \vdash (\delta(n+1) \Rightarrow \langle \mathbf{K} \cup \mathbf{M} \rangle \delta(n)) \tag{3}$$

ma dowód w systemie DL ze zbioru założeń F(**A**).

Ponieważ dla dowolnego parzystego n,

$$bin(n) = bin(n/2) \quad \text{oraz} \quad |\log_2 y| = 1 + |\log_2 (y/2)|$$

zatem formuła

$$(z * x^y = a^b \wedge n+1 = |\log_2 y| + bin(y) \wedge y > 0 \wedge p(y)) \Rightarrow$$
$$\Rightarrow (z * (x^2)^{y/2} = a^b \wedge n = |\log_2 (y/2)| + bin(y/2) \wedge y > 0 \wedge p(y))$$

jest prawdziwa w każdej strukturze arytmetycznej, tzn. można ją udowodnić w systemie DL wzbogaconym o zbiór aksjomatów F(**A**). Na mocy aksjomatu (?R)

$$\mathrm{F}(\mathbf{A}) \vdash (z * (x^2)^{y/2} = a^b \wedge n = |\log_2 (y/2)| + bin(y/2) \wedge y > 0 \wedge p(y)) \Rightarrow$$
$$\Rightarrow \langle y > 0 \wedge p(y))? \rangle (z * (x^2)^{y/2} = a^b \wedge n = |\log_2 (y/2)| + bin(y/2))$$

a na mocy aksjomatu ($\leftarrow$ R) mamy

$$\mathrm{F}(\mathbf{A}) \vdash (z * (x^2)^{y/2} = a^b \wedge n = |\log_2 (y/2)| + bin(y/2)) \Rightarrow$$
$$\Rightarrow \langle x := x^2;\ y := y/2 \rangle (z * x^y = a^b \wedge n = |\log_2 y| + bin(y))$$

Z twierdzeń (1), (2), (3), aksjomatu (;R) i po zastosowaniu reguły (G) otrzymujemy

$$\mathrm{F}(\mathbf{A}) \vdash ((\delta(n+1) \wedge y > 0 \wedge p(y)) \Rightarrow \langle \mathbf{K} \rangle \delta(n)) \tag{4}$$

Analogiczne rozważania przeprowadzimy, gdy wartością y będzie liczba nieparzysta. Ponieważ dla dowolnego nieparzystego n

$$bin(n) = 1 + bin(n-1) \quad \text{oraz} \quad |\log_2 y| = |\log_2 (y-1)|$$

zatem formuła

$$(z * x^y = a^b \wedge n+1 = |\log_2 y| + bin(y) \wedge np(y)) \Rightarrow$$
$$\Rightarrow (z * x * x^{y-1} = a^b \wedge n = |\log_2 (y-1)| + bin(y-1) \wedge np(y))$$

jest prawdziwa w każdej strukturze arytmetycznej, tzn. ma dowód ze zbioru aksjomatów F(**A**).

Na mocy aksjomatu (?R)

$$F(A) \vdash (z * x * x^{y-1} = a^b \wedge n = |\log_2(y-1)| + bin(y-1) \wedge np(y)) \Rightarrow$$
$$\Rightarrow \langle np(y)? \rangle (z * x * x^{y-1} = a^b \wedge n = |\log_2(y-1)| + bin(y-1))$$

a na mocy aksjomatu ($\leftarrow$ R) i reguły (G) otrzymujemy

$$F(A) \vdash (z * x * x^{y-1} = a^b \wedge n = |\log_2(y-1)| + bin(y-1) \wedge np(y)) \Rightarrow$$
$$\Rightarrow \langle np(y)? \rangle \langle z := z * x \rangle \langle y := y-1 \rangle (z * x * x^y = a^b \wedge n = |\log_2 y| + bin(y))$$

Z udowodnionych formuł, aksjomatu (;R) i praw logiki klasycznej otrzymujemy

$$F(A) \vdash ((\delta(n+1) \wedge np(y)) \Rightarrow \langle M \rangle \delta(n)) \tag{5}$$

Twierdzenia (4) i (5) oraz aksjomat ($\cup$ R) pozwalają udowodnić formułę (3). Stąd na mocy reguły indukcji (C*) oraz twierdzeń (1) i (2) otrzymujemy

$$F(A) \vdash (\alpha \Rightarrow \langle (K \cup M)^* \rangle \beta)$$

Udowodniliśmy w ten sposób, że DL-program $(K \cup M)^*$ oblicza poprawnie wartości funkcji x^y dla dowolnych y naturalnych, tzn. DL-program $(K \cup M)^*$ jest poprawny ze względu na warunek początkowy α i końcowy β we wszystkich arytmetycznych strukturach danych. □

Logika dynamiczna jest od ponad dziesięciu lat szeroko rozwijana przez wielu autorów. Znane są różne jej warianty w zależności od zestawu oferowanych operacji na programach i od klasy dopuszczalnych modalności. Nie sposób wymienić ich wszystkich w tym niewielkim opracowaniu. Zainteresowanych czytelników odsyłamy do bogatej literatury [23, 24, 25, 43].

8.5 Logika temporalna

Logika temporalna jest uznawana dość powszechnie za najlepsze ze znanych narzędzie do specyfikacji i wnioskowania o programach współbieżnych. Dzięki występowaniu specjalnych operatorów pozwalających formalnie reprezentować własności prawdziwe na różnych etapach obliczenia programu oraz własności mówiące o kolejności występowania zdarzeń, logika ta pozwala analizować zachowanie programu w czasie obliczeń, a nie tylko relację wejścia-wyjścia programu. Logika klasyczna jest właściwa do opisu statycznej sytuacji, logika temporalna zaś umożliwia rozważania, jak zmienia się stan pamięci w miarę upływu czasu. Zasadnicza różnica między logiką temporalną a omawianymi wcześniej logikami polega na tym, że program nie jest wyrażony w języku tej logiki, nie występuje explicite w formułach temporalnych. Natomiast obliczenie programu lub zbiór możliwych obliczeń programu może stanowić bazę semantyczną dla tej logiki.

Język logiki temporalnej stanowi rozszerzenie języka logiki klasycznej o operatory modalne, do których najczęściej zalicza się $\Box$ — konieczne, $\Diamond$ — możliwe, $\circ$ — w następnej chwili, *until* — dopóki, *atnext* — następnym razem. Ponadto przyjmuje się, że w języku występują dwa typy zmiennych indywiduowych: zmienne lokalne i zmienne globalne. Zbiór termów i zbiór formuł elementarnych jest zbudowany tak, jak w języku pierwszego rzędu. Poniżej przedstawimy dokładną definicję zbioru TL-formuł pewnej logiki temporalnej.

DEFINICJA 8.8

Zbiór TL-formuł jest najmniejszym zbiorem wyrażeń zawierającym wszystkie formuły klasyczne pierwszego rzędu i takim, że jeżeli α i β są TL-formułami, a x jest zmienną globalną, to

$$(\alpha \wedge \beta),\ (\alpha \vee \beta),\ \neg \alpha,\ (\exists x)\alpha,\ (\forall x)\alpha,\ (\alpha \Rightarrow \beta)$$

oraz

$$\Diamond \alpha,\ \Box \alpha,\ \circ \alpha,\ (\alpha\ \textit{until}\ \beta),\ (\alpha\ \textit{atnext}\ \beta)$$

są TL-formułami. ■

Semantyka języka logiki temporalnej jest określana na różne sposoby w zależności od koncepcji czasu ukrytej w sformułowaniach typu „w następnej chwili", „zawsze w przyszłości", „kiedyś w przyszłości" itd. Najlepiej zbadana jest tzw. logika czasu liniowego i tę właśnie omówimy tu szerzej.

Niech **A** będzie ustaloną strukturą danych dla języka pierwszego rzędu L związanego z rozważanym językiem temporalnym.

DEFINICJA 8.9

Strukturą czasu liniowego $\mathfrak{M}$ nad strukturą danych **A** dla języka temporalnego L będziemy nazywać system $\langle \mathbf{A}, S, v \rangle$, gdzie S jest ciągiem wartościowań zmiennych lokalnych (tzn. ciągiem stanów systemu $\mathfrak{M}$), $S = \{s_0, s_1, \ldots\}$, a v jest wartościowaniem zmiennych globalnych.

Znaczenie termów i formuł jest określone rekurencyjnie następującymi definicjami:

$x_{\mathfrak{M}}(s_i) = v(x)$, gdy x jest zmienną globalną

$x_{\mathfrak{M}}(s_i) = s_i(x)$, gdy x jest zmienną lokalną

$\varphi(\tau_1, \ldots, \tau_n)_{\mathfrak{M}}(s_i) = \varphi_{\mathfrak{M}}(\tau_{1\mathfrak{M}}(s_i), \ldots, \tau_{1\mathfrak{M}}(s_i))$ dla dowolnego funktora n-argumentowego φ

$\mathfrak{M}, s_i \models \alpha = \mathbf{A}, s_i \models \alpha$, dla dowolnej formuły pierwszego rzędu α
$\mathfrak{M}, s_i \models \neg \alpha \equiv non\, \mathfrak{M}, s_i \models \alpha$
$\mathfrak{M}, s_i \models (\alpha \wedge \beta) \equiv \mathfrak{M}, s_i \models \alpha$ i $\mathfrak{M}, s_i \models \beta$
$\mathfrak{M}, s_i \models (\forall x)\, \alpha(x) \equiv \mathfrak{M}, s_i \models \alpha(x/a)$, dla dowolnego $a \in \mathbf{A}$
$\mathfrak{M}, s_i \models \circ \alpha = \mathfrak{M}, s_{i+1} \models \alpha$
$\mathfrak{M}, s_i \models \Box \alpha \equiv \mathfrak{M}, s_j \models \alpha$, dla wszystkich $j > i$
$\mathfrak{M}, s_i \models \Diamond \alpha \equiv \mathfrak{M}, s_j \models \alpha$, dla pewnego $j > i$
$\mathfrak{M}, s_i \models (\alpha\ until\ \beta) \equiv (\exists j > i)\, \mathfrak{M}, s_j \models \beta$ i dla wszystkich $i < k < j$, $\mathfrak{M}, s_j \models \alpha$
$\mathfrak{M}, s_i \models (\alpha\ atnext\ \beta) \equiv (\forall j > i)\, \mathfrak{M}, s_j \models \neg\beta$ lub
$\mathfrak{M}, s_k \models \alpha$, dla najmniejszego $k > i$, dla którego $\mathfrak{M}, s_k \models \beta$

owiemy, że formuła α jest prawdziwa w strukturze $\mathfrak{M}$, jeżeli

$\mathfrak{M}, s \models \alpha$

a każdego stanu s struktury $\mathfrak{M}$. ■

RZYKŁAD 8.11

L-formuła postaci

$\circ\alpha \equiv$ **false** *until* α

st prawdziwa w każdej strukturze temporalnej czasu liniowego. □

Zadanie syntaktycznej charakteryzacji zbioru formuł prawdziwych we szystkich strukturach temporalnych czasu liniowego jest równie trudne jak ıdanie aksjomatyzacji logiki algorytmicznej czy logiki dynamicznej. Co ięcej, nie jest możliwe podanie pełnego skończonego systemu aksjomatów la logiki temporalnej pierwszego rzędu [47]. Przytoczona poniżej aks- ıatyzacja logiki temporalnej pochodzi od Kroegera [32] i stanowi pełny /stem wtedy, gdy ograniczymy się tylko do temporalnego rachunku zdań zn. do zbioru formuł, w których występują jedynie zmienne zdaniowe spójniki logiczne).

KSJOMATY

Lt0 aksjomaty klasycznego rachunku zdań
Lt1 $\Box(\alpha \Rightarrow \beta) = (\Box\alpha \Rightarrow \Box\beta)$
Lt2 $\circ(\alpha \Rightarrow \beta) \Rightarrow (\circ\alpha \Rightarrow \circ\beta)$
Lt3 $\Box(\alpha \Rightarrow \circ\alpha) \Rightarrow (\circ\alpha \Rightarrow \Box\alpha)$
Lt4 $\Box\alpha \Rightarrow (\circ\alpha \wedge \circ\Box\alpha)$
Lt5 $\circ\neg\alpha \equiv \neg\circ\alpha$
Lt6 $\Box\neg\beta \Rightarrow (\alpha\ atnext\ \beta)$
Lt7 $(\alpha\ atnext\ \beta) \equiv \circ(\alpha \wedge \beta) \vee \circ(\neg\beta \wedge (\alpha\ atnext\ \beta))$
Lt8 $(\alpha\ until\ \beta) \equiv (\beta\ atnext\ (\alpha \Rightarrow \beta)) \wedge \Diamond\beta$

REGUŁY WNIOSKOWANIA

$$\text{MP}\ \frac{\alpha, (\alpha \Rightarrow \beta)}{\beta}$$

$$\text{r}\Box:\ \frac{\alpha}{\Box\alpha}$$

PRZYKŁAD 8.12

Następująca formuła jest twierdzeniem systemu TL

$$((\circ\alpha \Rightarrow \circ\beta) \Rightarrow \circ(\alpha \Rightarrow \beta))$$

DOWÓD

Niech γ oznacza formułę α lub formułę β.

(1) $(\alpha \wedge \beta) \Rightarrow \gamma$	{prawo rachunku zdań}
(2) $\Box((\alpha \wedge \beta) \Rightarrow \gamma)$	{z (1) i reguły r $\Box$}
(3) $\circ((\alpha \wedge \beta) \Rightarrow \gamma)$	{własność (2) i Lt4}
(4) $\circ(\alpha \wedge \beta) \Rightarrow \circ\gamma$	{z (3) i Lt2}
(5) $\circ(\alpha \wedge \beta) \Rightarrow (\circ\alpha \wedge \circ\beta)$	{z (4) i praw rachunku zdań}
(6) $\circ(\alpha \wedge \neg\beta) \Rightarrow (\circ\alpha \wedge \circ\neg\beta)$	{weźmy w (5) $\neg\beta$ zamiast β}
(7) $(\neg\circ\alpha \vee \circ\beta) \Rightarrow \circ\neg(\alpha \wedge \neg\beta)$	{(6), prawo transpozycji i Lt5}
(8) $(\circ\alpha \Rightarrow \circ\beta) \Rightarrow \circ(\alpha \Rightarrow \beta)$	{z (7) i praw rachunku zdań}

□

Możliwość zastosowania logiki temporalnej do badania własności programów współbieżnych przedstawimy w następującym przykładzie.

PRZYKŁAD 8.13

Rozważmy dowolny program współbieżny M postaci

cobegin $M_1 \| \ldots \| M_n$ **coend**

gdzie $M_1, \ldots, M_n$ są programami iteracyjnymi. Z każdym programem będziemy wiązać zbiór zmiennych zdaniowych jednoznacznie identyfikujących wystąpienia akcji procesów $M_1, \ldots, M_n$. Spełnienie zmiennej zdaniowej p_a, będącej oznaczeniem pewnego wystąpienia akcji a rozważanego programu, będziemy interpretować jako „akcja a jest wykonywana". Niech V_c oznacza zbiór tych zmiennych zdaniowych. Podstawowym założeniem modelu będzie warunek, że w każdym stanie modelu może być wykonywana dokładnie jedna akcja.

Formułę temporalną prawdziwą przy tym założeniu będziemy nazywać

M-prawdziwą. Każda taka formuła opisuje pewną własność programu M rozważanego w semantyce ARB, czyli w semantyce przeplatania atomowych akcji różnych procesów. Jeżeli jako zbiór stanów modelu weźmiemy zbiór stanów pewnego obliczenia programu M w takiej semantyce, to formuła temporalna prawdziwa w tym modelu mówi o własnościach rozważanego obliczenia. □

UWAGA

Dla ustalonej konfiguracji początkowej istnieje wiele dopuszczalnych obliczeń programu współbieżnego. Możemy mówić o drzewie obliczeń dopuszczalnych (możliwych). W związku z tym zaproponowano inny wariant logiki temporalnej, tzw. logikę czasu rozgałęzionego (ang. branching time temporal logic). Zainteresowanych odsyłamy do bogatej literatury (zob. [35]). ■

Dowód lematu 5.6

LEMAT 5.6

Następująca formuła jest twierdzeniem teorii ATPQ

$$eq(q, q') = (\forall e)(mb(e, q) \equiv mb(e, q'))$$

DOWÓD

Udowodnimy najpierw implikację $\Rightarrow$.

$$eq(q, q') \Rightarrow (\forall e)\,(mb(e, q) = mb(e, q')) \qquad (1)$$

Przyjmiemy następujące oznaczenia:

γ: $(bool \wedge \neg em(q1) \wedge \neg em(q2))$
α: $(bool \wedge em(q) \wedge em(q'))$
$\beta \stackrel{\text{df}}{=} (\forall e)(mb(e, q) \equiv mb(e, q'))$
IF $\stackrel{\text{df}}{=}$ **if** γ **then** M **fi**

M: **begin**
$e1 := min(q1)$;
$bool := bool \wedge mb(e1, q2)$;
$q1 := del(e1, q1)$;
$q2 := del(e1, q2)$
end

K: **begin**
$q1 := q$;
$q2 := q'$;
$bool :=$ **true**
end

Dzięki tym oznaczeniom aksjomat Pq9 (patrz p. 5.4) możemy przepisać jako

$$eq(q, q') \equiv \text{K}(\textbf{while } \gamma \textbf{ do M od } \alpha)$$

Udowodnimy najpierw, że dla każdej liczby naturalnej $i \in N$, następująca formuła jest twierdzeniem teorii kolejek priorytetowych:

$$\text{K}(\text{IF}^i(\neg\gamma \wedge \alpha)) \Rightarrow \beta \qquad (2)$$

Z tego faktu, po zastosowaniu reguły R3 (rozdz. 4), otrzymamy formułę (1).

Dowód wzoru (2) przebiega przez indukcję ze względu na i. Dla dowolnych q i e, na mocy aksjomatu Pq8, mamy

$$\mathrm{ATPQ} \vdash (em(q) \Rightarrow \neg mb(e, q))$$

Zatem

$$\mathrm{ATPQ} \vdash \mathrm{K}(em(q1) \wedge em(q2) \wedge bool) \Rightarrow (\forall e)(mb(e, q) = mb(e, q'))$$

Ponieważ $\vdash (\mathrm{K}(\neg\gamma \wedge \alpha) \equiv \alpha)$, to z tej implikacji otrzymujemy

$$\mathrm{ATPQ} \vdash (\mathrm{K}(\neg\gamma \wedge \alpha) \Rightarrow \beta)$$

czyli formuła (2) jest twierdzeniem teorii ATPQ dla $i = 0$. Załóżmy teraz, że formuła (2) jest twierdzeniem ATPQ dla wszystkich $j \leqslant i$, rozważmy formułę

$$\mathrm{K}(\textbf{if } \gamma \textbf{ then } \mathrm{M} \textbf{ fi})^{i+1}(\neg\gamma \wedge \alpha) \tag{3}$$

Po zastosowaniu aksjomatu Ax20 stwierdzamy, że formuła (3) jest równoważna formule

$$\mathrm{K}(\gamma \wedge \mathrm{M}(\mathrm{IF}^1(\neg\gamma \wedge \alpha)) \vee \neg\gamma \wedge (\mathrm{IF}^i(\neg\gamma \wedge \alpha)))$$

Po zastosowaniu aksjomatów Ax14 i Ax15 otrzymamy inną formułę równoważną formule (3)

$$(\neg em(q) \wedge \neg em(q') \wedge \mathrm{K}(\mathrm{M}(\mathrm{IF}^1(\neg\gamma \wedge \alpha)))) \vee \\ \vee ((em(q) \vee em(q')) \wedge \mathrm{K}(\mathrm{IF}^1(\neg\gamma \wedge \alpha)))$$

i na mocy założenia indukcyjnego, druga część tej alternatywy implikuje formułę

$$((em(q) \vee em(q')) \wedge \beta) \tag{4}$$

Rozważmy zatem formułę

$$(\neg em(q) \wedge \neg em(q') \wedge \mathrm{K}(\mathrm{M}(\mathrm{IF}^i(\neg\gamma \wedge \alpha)))) \tag{5}$$

Na mocy tautologii

$$\models (bool := \delta)\, \delta' = ((\delta \wedge (bool := \textbf{true})\, \delta') \vee (\neg\delta \wedge (bool := \textbf{false})\, \delta'))$$

(por. przykład 3.15) i aksjomatu Ax18 mamy

$$\mathrm{M}(\mathrm{IF}^1(\neg\gamma \wedge \alpha)) = (e1 := min(q1))((bool \wedge mb(e1, q2) \wedge \\ \wedge (bool := \textbf{true})\, \mathrm{K}'(\mathrm{IF}^i(\neg\gamma \wedge \alpha))) \vee (\neg(bool \wedge \\ \wedge mb(e1, q2)) \wedge (bool := \textbf{false})\, \mathrm{K}'(\mathrm{IF}^1(\neg\gamma \wedge \alpha))))$$

Idzie

$$\mathrm{K}' \stackrel{\mathrm{df}}{=} \textbf{begin } q1 := del(e1, q1);\ q2 := del(e1, q2) \textbf{ end}$$

Korzystając z przykładu 4.11 otrzymujemy

$$(bool := \textbf{false})\, \mathrm{K}'(\mathrm{IF}^1(\neg\gamma \wedge \alpha)) \equiv \textbf{false}$$

a zatem możemy pominąć drugą część powyższej alternatywy. Na mocy założenia indukcyjnego formuła (5) implikuje więc formułę

$$(\neg em(q) \wedge \neg em(q')) \wedge \mathrm{K}(e1 := min(q1))(bool \wedge mb(e1, q2) \wedge \\ \wedge (\forall e)\, mb(e, del(e1, q1)) \equiv mb(e, del(e1, q2)))$$

Po zastosowaniu aksjomatów Ax14 i Ax18 przekształcimy tę ostatnią formułę do postaci

$$(\neg em(q) \wedge \neg em(q') \wedge \beta)$$

Stąd i z (4) otrzymujemy, że w teorii ATPQ formuła (3) implikuje formułę

$$(em(q) \vee em(q') \wedge \beta) \vee (\neg em(q) \wedge \neg em(q') \wedge \beta)$$

Ostatecznie udowodniliśmy, że

$$\mathrm{ATPQ} \vdash (\mathrm{K}(\textbf{if } \gamma \textbf{ then } \mathrm{M} \textbf{ fi})^{i+1}(\neg\gamma \wedge \alpha) \Rightarrow \beta)$$

Stąd na mocy zasady indukcji matematycznej wynika własność (2) dla dowolnej liczby naturalnej i, a zastosowanie reguły R3 pozwala uzasadnić własność (1).

Dla dowodu implikacji przeciwnej

$$(\beta \Rightarrow eq(q, q'))$$

załóżmy, że dla pewnego wartościowania v w strukturze $\mathbf{A}$ będącej modelem teorii ATPQ mamy

$$\mathbf{A}, v \models \beta \qquad (6)$$

Ponieważ program **begin** K; **while** γ **do** M **od end** nie zapętla się przy dowolnym wartościowaniu początkowym (por. lemat 5.4) zatem, na mocy definicji semantyki, istnieje najmniejsza liczba naturalna n taka, że

$$\mathbf{A}, v \models (\mathrm{K}(\mathrm{M}^n \neg\gamma) \wedge \mathrm{K}(\mathrm{M}^i\gamma)) \quad \text{dla } i < n$$

Oznaczmy $v_0 \stackrel{\mathrm{df}}{=} \mathrm{K}_{\mathbf{A}}(v)$ oraz dla $0 \leqslant i \leqslant n$, $v_{i+1} \stackrel{\mathrm{df}}{=} \mathrm{M}_{\mathbf{A}}(v_i)$. W dalszym ciągu wykażemy, że $\mathbf{A}, v_n \models \alpha$. Zauważmy najpierw prosty fakt

$$\mathbf{A}, v_{i+1} \models bool \equiv \mathbf{A}, v_i \models (\mathrm{M}bool) = \mathbf{A}, v_i \models (bool \wedge mb(min(q1), q2))$$

Wynika stąd, że

$$\mathbf{A}, v_n \models bool \quad \text{wtw} \quad \mathbf{A}, v \models mb(min(q_0), q'_0) \wedge \\ \wedge mb(min(q_1), q'_1) \wedge \ldots \wedge mb(min(q_{n-1}), q'_{n-1})$$

gdzie

$q_0 \stackrel{df}{=} q$
$q'_0 \stackrel{df}{=} q'$
$q_{i+1} \stackrel{df}{=} del(min(q_i), q_i)$
$q'_{i+1} \stackrel{df}{=} del(min(q_i), q'_i)$

Fakt 1

$\mathbf{A}, v \models \neg(min(q_i) = min(q_{i-1}))$

Rzeczywiście, gdyby $\mathbf{A}, v \models min(del(e, q)) = e$ dla pewnego e, to na mocy lematu 5.5 (por. własność (2)) byłoby $\mathbf{A}, v \models mb(e, del(e, q))$, co jest sprzeczne z założeniem, że **A** jest modelem ATPQ i aksjomatu Pq6 w szczególności.

Fakt 2

$\mathbf{A}, v \models mb(min(q_i), q'_i)$ wtw $\mathbf{A}, v \models mb(min(q_i), q_i)$

Na mocy przyjętych oznaczeń i lematu 5.5 oraz faktu 1 mamy

$\mathbf{A}, v \models mb(min(q_i), q'_i)$ wtw
$\mathbf{A}, v \models \neg(min(q_\cdot) = min(q_{i-1})) \wedge mb(min(q_i), q'_{i-1})$ wtw
$\mathbf{A}, v \models mb(min(q_i), q'_{i-1})$ wtw ...wtw
$\mathbf{A}, v \models mb(min(q_i), q')$

owtarzając powyższe argumenty wykażemy, że

$\mathbf{A}, v \models mb(min(q_i), q)$ wtw $\mathbf{A}, v \models mb(min(q_i), q_i)$

Na mocy założenia (6)

$\mathbf{A}, v \models mb(min(q_i), q')$ wtw $\mathbf{A}, v \models mb(min(q_i), q)$

co kończy dowód faktu 2.

Korzystając z udowodnionej własności otrzymujemy

$\mathbf{A}, v_n \models bool$ wtw dla dowolnego $i < n$
$\mathbf{A}, v \models mb(min(q_i), q_i)$

onieważ **A** jest z założenia modelem teorii ATPQ, więc na mocy lematu 5.5

$\mathbf{A}, v_n \models bool$

Z założenia i z przeprowadzonego do tej pory dowodu mamy $\mathbf{A}, v_n \models$ $\models (\neg\gamma \wedge bool)$, tzn.

$$\mathbf{A}, v_n \models (em(q1) \vee em(q2)) \tag{7}$$

Fakt 3

$\mathbf{A}, v_n \models em(q1)$ wtw $\mathbf{A}, v_n \models em(q2)$

Rzeczywiście, gdyby *non* $\mathbf{A}, v_n \models em(q2)$, to na mocy własności (2) z lematu 5.5

byłoby $\mathbf{A}, v_n \models (\exists e)\, mb(e, q2)$. Używając argumentów podobnych do przedstawionych pokazalibyśmy

$$\mathbf{A}, v \models (\exists e)\, mb(e, del(min(q_{n-1}), q'_{n-1}))$$

Stąd i z faktu 2 otrzymamy $\mathbf{A}, v \models (\exists e)\, mb(e, del(min(q_{n-1}), q_{n-1}))$. W konsekwencji otrzymujemy

$$\mathbf{A}, v_n \models (\exists e)\, mb(e, q1)$$

Oznacza to na mocy lematu 5.5, że $\mathbf{A}, v_n \models \neg em(q1)$, co należało wykazać.

Z faktu 3 oraz własności (7) wynika natychmiast $\mathbf{A}, v_n \models \alpha$. Zatem w strukturze $\mathbf{A}$ przy wartościowaniu początkowym v jest spełniona formuła **begin** K; **while** γ **do** M **od end** α, a na mocy aksjomatu Pq9 jest spełniona również formuła $eq(q, q')$. Wykazaliśmy w ten sposób prawdziwość implikacji

$$\mathbf{A}, v \models (\beta \Rightarrow eq(q, q'))$$

Ponieważ zarówno struktura $\mathbf{A}$, jak i wartościowanie v było dowolne, więc na mocy twierdzenia o pełności logiki algorytmicznej wykazaliśmy, że

$$\mathrm{ATPQ} \vdash (\beta \Rightarrow eq(q, q'))$$

co łącznie z pierwszą częścią dowodu daje $\mathrm{ATPQ} \vdash (\beta = eq(q, q'))$. □

Dziwna implementacja kolejek

Czasami spotykamy się z uwagą „po co komuś aksjomat(y) zapewniający(e) skończoność kolejki (stosu, drzewa itp.)? Przecież, z natury rzeczy, w komputerze można zapisać tylko skończoną informację". Czy jednak na pewno można pominąć te elementy specyfikacji?

Mamy nadzieję, że czytelnik dostrzegł pożyteczną rolę aksjomatów algorytmicznych w dowodach poprawności (stopu, ...) programów. Chcemy teraz pokazać pewne społeczne lub ekonomiczne konsekwencje, jakie może pociągnąć rezygnacja z użycia aksjomatów algorytmicznych.

Przypuśćmy, że firma X zleciła firmie Y zadanie wykonania modułu implementującego strukturę kolejek. Załącznikiem do umowy, regulującym sposób odbioru pracy, stała się następująca specyfikacja. Moduł ma zawierać dwa typy danych: *elem* i *kolejka* oraz cztery funkcje: *first*, *put*, *out*, *empty*, w taki sposób, by zachodziły zależności

$$\neg\, empty(put(e, q))$$
$$\neg\, empty(q) \Rightarrow (out(put(e, q)) = put(e, out(q)))$$
$$empty(q) \Rightarrow (out(put(e, q)) = q)$$
$$\neg\, empty(q) \Rightarrow (first(put(e, q)) = first(q))$$
$$empty(q) \Rightarrow (first(put(e, q)) = e)$$
$$ok(first(q)) \langle=\rangle ok(out(q)) \langle=\rangle \neg\, empty(q)$$

i naturalnie, aksjomaty równości.

Co się stanie, gdy zleceniobiorca przedstawi firmie X następujący moduł i zażąda zapłaty?

```
unit KOLEJKI_DZIWNE: class;
  unit elem: clase; end elem;
  unit kolejka: class; end kolejka;
  var e0; elem,
      q0: kolejka;

  signal PustaKolejka;

  unit empty: function(q: kolejka): Boolean;
```

```
    begin
      result := false;
    end empty;

    unit first: function (q: kolejka): elem;
    begin
      result := e0
    end first;

    unit put: function (e: elem, q: kolejka): kolejka;
    begin
      result := q0
    end put;

    unit out: function (q : function (q: kolejka): kolejka;
    begin
      if empty(q)
      then
        raise PustaKolejka
      else
        result := q0
      fi
    end out;

  begin
      e0 := new elem;
      q0 := new kolejka;

  end KOLEJKI_DZIWNE
```

Czy zleceniodawca ma zapłacić czy nie? Rozsądek mówi mu, że wyrzuca pieniądze w błoto. (Zauważmy, że przy podanej implementacji program **while** ¬ *empty* (*q*) **do** *q* := *out* (*q*) **od** będzie miał niekończące się obliczenia). Umowa i specyfikacja będąca jej częścią nakazują zapłacić. Bo rzeczywiście ta implementacja spełnia warunki wyliczone w umowie. Określono dwa typy (co prawda w obu typach udaje się utworzyć tylko trywialne obiekty — ale można stworzyć inną implementację zawierającą zwykłe kolejki i jeszcze te dziwne). Operacje *empty*, *first*, *put* i *out* są określone (zawsze) i spełniają warunki umowy. Pierwszy warunek jest spełniony ponieważ każda kolejka jest niepusta. Warunek drugi zachodzi ponieważ wynikiem operacji *put* i *out* jest *q*0. Warunki trzeci i piąty są spełnione, bo nie ma kolejek pustych. Warunek czwarty jest spełniony w sposób oczywisty.

Kto tu był nieuczciwy? Czy rzeczywiście mowa jest tylko o oszukiwaniu i sposobach przeciwdziałania? Czy potrafisz czytelniku ułożyć taki moduł, który zawierałby zarówno zwykłe kolejki (skończone), jak i te dziwne, niestandardowe? A jeśli taki moduł powstanie przypadkiem, niechcący? Czy

można będzie wtedy bezpiecznie wnioskować o zachowaniu się programu korzystającego z takiej struktury tylko na podstawie specyfikacji, jaką przytoczyliśmy?

Operacje częściowe

Spotkaliśmy się z zarzutem, że w naszych programach i formułach funktory występują tylko z takimi argumentami, dla których są określone wyniki działań. *A to wcale nie jest przypadek. Zerwanie obliczeń jest zawsze sytuacją niepożądaną. Staramy się jej uniknąć i stosujemy trzy różne środki chroniące nas przed taką ewentualnością. Może czytelnicy zechcą skorzystać z naszych recept?*

Formuły. Spójrzmy na następujący aksjomat kolejek:

$$\neg\, empty(q) \Rightarrow (put(e, out(q)) = out(put(e, q)))$$

Nie możemy przyjąć jako aksjomatu samej tylko równości występującej w następniku implikacji $(put(e, out(q)) = out(put(e, q))$. Nie jest ona bowiem prawdziwa dla kolejki pustej. A dla pustej kolejki q zachodzi inna własność $(q = out(put(e, q)))$.

Programy. Uważamy za wskazane posługiwanie się instrukcją warunkową po to, by zapewnić, że obliczenie nie zostanie zerwane na skutek próby wykonania operacji na argumentach nie należących do dziedziny. Warto więc napisać

if $x \geqslant 0$ **then** $y := \sqrt{x}$ **fi** zamiast $y := \sqrt{x}$
if $\neg\, empty(q)$ **then** $q1 := out(q)$ **fi** zamiast $q1 := out(q)$

Sygnały. Chcemy zwrócić uwagę czytelnika na pożyteczne i rzadko stosowane narzędzie programistyczne, jakim są sygnały (często zwane sygnałami sytuacji wyjątkowych) i moduły ich obsługi. Posługiwanie się tym narzędziem nie jest bynajmniej łatwe, ale pozwala ono programiście systemowemu stworzyć moduły, które będą w stanie rozpoznać sytuację wyjątkową i zasygnalizować ją innemu modułowi (modułowi użytkownika), w którym zawarto odpowiednią receptę postępowania w takim przypadku.

O współprogramach

Współprogramy (ang. coroutines) są bardzo przydatnym, a mało znanym, narzędziem programowania. Dodatek ten ma na celu przybliżenie czytelnikowi własności i możliwości zastosowań współprogramów. Spośród powszechnie używanych języków programowania tylko Modula 2 daje możliwość wykorzystywania współprogramów*).

Wymieńmy kilka z bardzo licznych zastosowań współprogramów:

(1) programowanie gier — każdemu graczowi przyporządkowuje się wtedy inny współprogram (narzędzie wręcz idealne do tego celu);

(2) symulacja procesów (najliczniejsze, najbardziej istotne zastosowania współprogramów);

(3) implementacja procesów w komputerach jednoprocesorowych (współbieżność jest wtedy imitowana przez podział czasu komputera pomiędzy współprogramy);

(4) w wielu programach, w których użycie dwu współpracujących modułów (a dokładniej obiektów) pozwala na programowanie naturalne i bardziej czytelne, np. moduły analizy leksykalnej i składniowej w translatorach języków programowania lub moduły: zbierający dane z pomiarów i tworzący histogram.

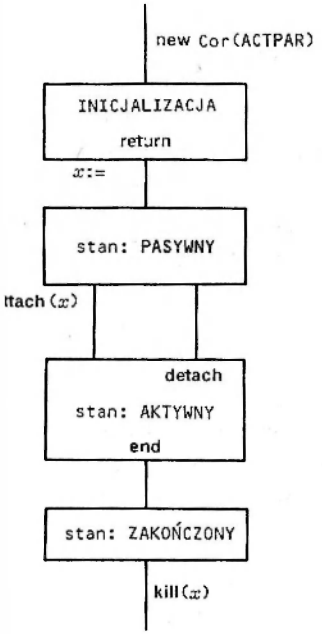
W tym dodatku przedstawimy współprogramy posługując się terminologią ustaloną w języku Loglan 82. Język ten zaprojektowano i zrealizowano w Instytucie Informatyki UW.

Słowo współprogram jest używane do oznaczenia zarówno modułu programu (który ma wiele własności klasy, a ponadto jest wyposażony w dwie dodatkowe operacje: **attach** i **detach**), jak i obiektu, który powstał według wzorca jakim jest moduł. W danym stanie obliczenia może istnieć równocześnie wiele różnych obiektów różnych współprogramów, ale tylko jeden z tych obiektów jest aktywny i wykonuje swe instrukcje. Zbiór utworzonych obiektów współprogramów można podzielić na kilka rozłącz-

*) Współprogramy były obecne już w Simuli 67. Loglan oferuje uporządkowany sposób współpracy z nimi, ale jak dotąd jest mało znany. W Adzie współprogramy trzeba wprowadzać bardzo okrężną drogą, jako procesy współbieżne... nie całkiem współbieżnie działające.

nych podzbiorów: współprogramy w stanie INICJALIZACJI, PASYWNE, AKTYWNY (jest zawsze tylko jeden proces aktywny), ZAKOŃCZONE. Reguły zmiany stanu są związane z operacjami: **new**, **attach**, **detach**, **return**, **end**, **kill**.

Niech Cor będzie nazwą pewnego modułu współprogramu. Instrukcja przypisania $x :=$ **new** Cor (⟨parametry aktualne⟩) powoduje utworzenie obiektu współprogramu Cor i jego inicjalizację. Polega to na wykonywaniu instrukcji współprogramu, faza inicjalizacji kończy się wraz z wykonaniem instrukcji **return**. Od tej chwili istnieje obiekt i można go nazwać, przypisać do zmiennej (np. przez $x :=$). Nowo utworzony obiekt jest w stanie pasywnym, można korzystać z niego tak, jak z obiektu klasy. Ponadto inny obiekt aktywny, powiedzmy współprogram y, może mu przekazać procesor wykonując instrukcję **attach**(x). Instrukcja ta ma podwójny efekt: współprogram y, w którym ta instrukcja została wykonana, przechodzi w stan pasywny, natomiast współprogram x — w stan aktywny. Pod warunkiem jednak, że obiekt przypisany zmiennej x był współprogramem w stanie pasywnym. W chwili rozpoczęcia pracy w obiekcie x zostają zapamiętane nazwa obiektu współprogramu y i instrukcja współprogramu y, którą należy wykonać po wykonaniu instrukcji **attach.** Instrukcja **attach** powoduje wznowienie wykonywania instrukcji w uaktywnionym obiekcie x współprogramu bezpośrednio po ostatnio wykonanej instrukcji. Tak więc, pierwsze wykonanie **attach** spowoduje wykonanie instrukcji występującej bezpośrednio po **return**.



Rys. C1

Druga operacja na współprogramach to **detach**. Instrukcja ta nie ma wyraźnego adresata jak instrukcja **attach**. Niech znowu *y* oznacza obiekt współprogramu zawierający właśnie wykonywaną instrukcję **detach**. Efektem instrukcji **detach** jest zawieszenie wykonywania instrukcji w obiekcie *y*, wznowienie (uaktywnienie) tego obiektu, który ostatnio uaktywnił współprogram *y*, zapamiętanie ostatnio wykonanej w *y* instrukcji.

Wyczerpanie listy instrukcji zawartych w obiekcie *y* współprogramu (oznaczone na rys. C1 przez **end**) pociąga za sobą przejście obiektu ze stanu aktywnego w stan zakończony i uaktywnienie obiektu, który ostatnio spowodował wznowienie obiektu *y*. Obiekt zakończony nie może być wznowiony ani instrukcją **detach**, ani instrukcją **attach**. Można z nim współpracować tak, jak z obiektem klasy, ale nie można mu przekazać sterowania (procesora). Można też obiekt taki zlikwidować, wykonując instrukcję **kill**(*y*).

Literatura

1. AHO A., HOPCROFT J., ULLMAN J.: *Projektowanie i analiza algorytmów komputerowych.* Warszawa, PWN 1983.
2. ALAGIĆ S., ARBIB M. A.: *Projektowanie programów poprawnych i dobrze zbudowanych.* Warszawa, WNT 1982.
3. APT K. R.: Ten years of Hoare's logic. A survey — Part 1. *ACM Trans. Prog. Iang. Syst.*, 1979, **3**, 431—483.
4. BACKHOUSE R. C.: Program construction and verification. Englewood Cliffs, New Jersey, Prentice-Hall 1986.
5. BANACHOWSKI L.: An axiomatic approach to the theory of data structures. *Bull. PAS*, 1975, **23**, 315–323.
6. BANACHOWSKI L.: Investigation of properties of programs by means of extended algorithmic logic. *Fundamenta Informaticae*, 1977, **1**, 93–119, 167–193.
7. BANACHOWSKI L., KRECZMAR A.: *Elementy analizy algorytmów.* Wyd. 2. Warszawa, WNT, 1989.
8. BANACHOWSKI L., KRECZMAR A., MIRKOWSKA G., RASIOWA H., SALWICKI A.: An introduction to algorithmic logic: mathematical investigations in the theory of programs. W Mathematical Foundations of Computer Science. A. Mazurkiewicz, Z. Pawlak (red.). *Banach Center Publications.* Warszawa, PWN 1977.
9. BARTOL W. M. i in.: *Report of Loglan programming language.* Warszawa, PWN 1983.
10. BURKHARD H. D.: On properties of parallelism in Petri nets under the maximum Firing Strategy. *Proc. Logics of programs and their applications.* Berlin Springer 1983, 86–98.
11. CHANG C. L., LEE R.: *Symbolic logic and mechanical theorem proving.* New York, Academic Press 1973.
12. CHELAS B. S.: *Modal logic, an introduction.* Cambridge, University Press 1980.
13. CONSTABLE R. L., O'DONNEL M. J.: *A programming logic.* Cambridge, Mass. Winthrop 1978.
14. COOK S.: Soundness and completeness of an axiom system for program verification. *SIAM Journal on Computing* 1978, 7, 70–90.
15. DAŃKO W.: Algorithmic properties of programs with tables. *Fundamenta Informaticae*, 1978, **1**, 379–398.
16. DAŃKO W.: A criterion of undecidability of algorithmic theories. *Proc. MFCS'80. LNCS 148.* Berlin Springer 1980, 205–216.
17. DIJKSTRA E. W.: On guarded commands, non-determinacy and formal derivation of programs, *CACM*, 1975, **18**, 453–457.
18. DIJKSTRA E. W.: *Umiejętność programowania.* Warszawa. WNT 1978.
19. FLOYD R. W.: Assigning meanings to programs. *Proc. Symp. Appl. Math.*, AMS, 1967, **19**, Mathematical Aspects of Computer Science. Schwartz J. T. (red.), 19–32.
20. GBURZYŃSKI P.: Badania eksperymentalne w dziedzinie automatycznego dowodzenia twierdzeń. Analiza porównawcza dwu metod. Praca doktorska. Uniwersytet Warszawski, Wydział Matematyki, Informatyki i Mechaniki 1982.

21. GRZEGORCZYK A.: *Zarys logiki matematycznej.* Warszawa, PWN 1973.
22. GÓRAJ A., MIRKOWSKA G., PALUSZKIEWICZ A.: *On the notion of description of programs.* Bull. PAS, Ser. Math., 1970, **18**, 499–506.
23. HAREL D.: *Algorithmics, the spirit of computing.* Wokingham, Addison-Wesley 1987.
24. HAREL D.: *First-order dynamic logic.* LNCS 68, Berlin, Springer 1979.
25. HAREL D.: Dynamic logic. W: *Handbook of philosophical logic*, Vol. II. Gabbay D., Guenther F. (red.), Dordrecht. Reidel Publ. Comp. 1982, 497–604.
26. HEHNER E. C. R.: *The logic of programming.* Englowood Cliffs, New Jersey, Prentice-Hall 1986.
27. HOARE C. A. R.: An axiomatic basis for computer programming CACM, 1969, **12**, 576–583.
28. HOARE C. A. R.: Proof of correctness of data representation. Acta Informatica 1972, **1**, 271–281.
29. HUGHES G. E., CRESSWELL M. J.: *A companion to modal logic.* London Methuen, 1984.
30. KLUŹNIAK F., SZPAKOWICZ S.: Prolog. Warszawa. WNT 1983.
31. KRECZMAR A.: Effectivity problems in algorithmic logic. *Fundamenta Informaticae*, 1977, **1**, 195–230.
32. KRÖGER F.: A generalized nexttime operator in temporal logic. *Journ. Comp. Syst. Sci.*, 1984, **29**, 80–98.
33. LYNDON R.: *O logice matematycznej.* Warszawa, PWN 1968.
34. MANNA Z., W'LDINGER R.: *The logical basis for computer programming.* Don Mills, Addison-Wesley 1985.
35. MANNA Z., PNUELI A.: The temporal logic of reactive and concurrent systems. New York, Springer-Verlag, 1992.
36. MARTINEK J.: Lisp. Warszawa, WNT 1982.
37. MAZUR S.: Computable analysis. *Dissertationes Math.*, 1963.
38. MEYER A., HALPERN J.: Axiomatic definition of programming languages: a theoretical assessment. *Proc. 7 ACM STOC*, 1980, **12**, 32–41.
39. MEYR E.: On Reachability problem in Nets. *Proc. ACM STOC*, 1981, **13**, 59–68.
40. MIRKOWSKA G., SALWICKI A.: *Algorithmic logic.* Warszawa, PWN, Dordrecht, Reidel Publ. Comp. 1987.
41. MOSTOWSKI A.: Logika matematyczna. *Monografie matematyczne.* Warszawa-Wrocław, Czytelnik 1948.
42. O'DONNELL M.: A critique of the foundations of Hoare-style programming logics. *Proc.* Logics of Programs 1981 (Kozen D. red.). LNCS 131. Berlin, Springer 1981, 349–374.
43. PRATT V.: Semantical considerations on Floyd-Hoare logic *Proc. 17th FOCS'76*, 1976, 109–121.
44. RASIOWA H.: *Wstęp do matematyki współczesnej.* Warszawa, PWN 1984, wyd. 8.
45. RASIOWA H., SIKORSKI R.: *Mathematics of metamathematics.* Warszawa, PWN 1968.
46. STARKE P.: Sieci Petri. Warszawa, PWN 1987.
47. SZAŁAS A.: Concerning the semantic consequence relation in first-order temporal logic. *TCS* 47, 1986, 329–334.
48. TURSKI W. M., MAIBAUM T. S. E.: The specification of computer programs. Wokingham Addison-Wesley 1987.

Skorowidz

T

W

Z

Algorithmic logic for programmers

Summary

This book in an introduction to logics of programs. It describes problems of program semantics, presents research results and applications of logics in software production. We have chosen algorithmic logic as the oldest and most developed branch of logics of programs. For computer science the algorithmic logic plays a role similar to that played by mathematical logic in mathematics. Limiting to a minimum abstract considerations concerning algorithmic logic itself as an object of studies, we present its applications in analysis of semantic properties of programs, specification of program modules and axiomatic definition of programming languages. We present comparisons with other logics of programs. As we believe that some information on metatheory of logics of programs is a necessary component of a common view of algorithmics, we present theorems on consistency and completeness of algorithmic logic. We argue that the ordinary first order logic is not an adequate tool for analysis of software.

The book is intended for programmers, system designers, computer scientists and students of computer science.

Логика для программистов

Резюме

Настоящая книга является введениеи в тематику логик программ, дает представление о проблемах связанных с семантикой программ, приводит результаты научных исследований, а также описывает приложения к производству математического обеспечения. Мы избрали для представления алгорифмическую логику, старейшую и лучше развитую из логик программ. Для работ в области программ и алгорифмов алгорифмическая логика имеет такое же значение как математическая логика для математики. Мы представляем читателю приложения алгорифмической логики к анализу семантических свойств программ, к спецификации модулей программ и к аксиоматическим определениям языков программирования. Мы представляем также другие логики программ, сравнивая их с алгорифмической логикой. Так как мы убеждены в том, что все работники в области математического обеспечения должны выработать свое собственное мнение об истине алгорифмики, мы попытались представить теоремы о совместимости и полноте алгорифмической логики и указать на примерах на недостаточность средств языка классической логики первого порядка для представления и анализа семантических свойств программ.

Книга предназначена для студентов информатики и математики, для программистов и научных работников.

Biblioteka Inżynierii Oprogramowania

Wykaz wydanych książek

S. Alagić, M. A. Arbib — Projektowanie programów poprawnych i dobrze zbudowanych
I. O. Angell — Wprowadzenie do grafiki komputerowej, wyd. 1 i 2
R. L. Baber — O oprogramowaniu inaczej
L. Banachowski, A. Kreczmar — Elementy analizy algorytmów, wyd. 1 i 2
L. Banachowski, A. Kreczmar, W. Rytter — Analiza algorytmów i struktur danych, wyd. 1 i 2
M. Ben-Ari — Podstawy programowania współbieżnego
J. Bielecki — System VSAM. Zasady stosowania w języku PL/I
J. Błażewicz — Złożoność obliczeniowa problemów kombinatorycznych
L. Bolc, M. Cichy, L. Różańska — Przetwarzanie języka naturalnego
S. Borak, J. Klaczak, S. Korczak, Z. Płoski — System operacyjny George 3, wyd. 1 i 2 rozszerzone
J. M. Brady — Informatyka teoretyczna w ujęciu programistycznym
K. L. Clark, F. G. Mc Cabe — Micro-Prolog
M. Dąbrowski, K. Laus-Mączyńska — Metody wyszukiwania i klasyfikacji informacji
C. Delobel, M. Adiba — Relacyjne bazy danych
P. Dembiński, J. Małuszyński — Matematyczne metody definiowania języków programowania
J. Deminet— System operacyjny RSX-11
E. W. Dijkstra — Umiejętność programowania, wyd. 1 i 2
S. Gasik, P. Kulczycki, K. Piasecki, J. Witaszek — PL/I F
P. Gizbert-Studnicki, J. Karczmarczuk — Snobol4
M. Głowacki — Systemy operacyjne DOS i OS
M. J. C. Gordon — Denotacyjny opis języków programowania
R. E. Griswold, M. T. Griswold — Icon
A. N. Haberman, D. E. Perry — Ada dla zaawansowanych
L. J. Hoffman — Poufność w systemach informatycznych
M. Iglewski, J. Madey, S. Matwin — Pascal. Język wzorcowy. Pascal 6000, wyd. 2

M. IGLEWSKI, J. MADEY, S. MATWIN — Pascal. Język wzorcowy. Pascal 360, wyd. 3 zmienione i 4
M. IGLEWSKI, J.MADEY, S. MATWIN — Pascal. Standard, wyd. 5 częściowe
W. ISZKOWSKI, M. MANIECKI — Programowanie współbieżne
R. JAGIELSKI — Tablice rozproszone
A. P. JERSZOW — Wprowadzenie do teorii programowania
C. B. JONES — Konstruowanie oprogramowania metodą systematyczną
A. KASSUR, P. PERKOWSKI — Obliczeniowe aspekty projektowania układów elektronicznych
B. W. KERNIGHAN, P. J. PLAUGER — Narzędzia programistyczne w Pascalu
B. W. KERNIGHAN, D. M. RITHIE — Język C, wyd. 1 i 2
F. KLUŹNIAK, S. SZPAKOWICZ — Prolog
H. KOPETZ — Niezawodność oprogramowania
R. KOWALSKI — Logika w rozwiązywaniu zadań
W. LIPSKI — Kombinatoryka dla programistów, wyd. 1 i 2
J. MARTINEK — Lisp. Opis, realizacja i zastosowania
G. J. MYERS — Projektowanie niezawodnego oprogramowania
L. NIEMCZYCKI — Oprogramowanie teleprzetwarzania maszyn Jednolitego Systemu
H. OKTABA, W. RATAJCZAK — Simula 67
J. OLSZEWSKI — Projektowanie struktur systemów operacyjnych
W. PACHELSKI — Fortran IV dla maszyn Odra serii 1300
W. PACHELSKI — Fortran IV dla maszyn Jednolitego Systemu, wyd. 1 i 2 poprawione i rozszerzone
T. PAVLIDIS — Grafika i przetwarzanie obrazów. Algorytmy
P. PERKOWSKI — Technika symulacji cyfrowej
Przegląd metod i algorytmów numerycznych, cz. 1 — J. i M. JANKOWSCY, wyd. 1 i 2 poprawione
Przegląd metod i algorytmów numerycznych, cz. 2 — M. DRYJA, J. i M. JANKOWSCY, wyd. 1 i 2 poprawione
I. C. PYLE — Ada
W. REISIG — Sieci Petriego. Wprowadzenie
P. P. SILVESTER — System operacyjny Unix
B. SZAFRAŃSKI, W. SKURZAK, W. SZYPUŁA — System operacyjny RT-11
D. VAN TASSEL — Praktyka programowania, wyd. 1 i 2 rozszerzone
D. C. TSICHRITZIS, F. H. LOCHOVSKY — Modele danych
W. M. TURSKI — Metodologia programowania, wyd. 1 i 2 rozszerzone
J. TYSZER — Symulacja cyfrowa
E. Ch. TYUGU — Programowanie z bazą wiedzy
J. D. ULLMAN — Systemy baz danych
W. M. WAITE, G. GOOS — Konstrukcja kompilatorów
J. WALASEK — Konwersacyjne otoczenie programowe Pascala
N. WIRTH — Modula 2, wyd. 1 i 2 poprawione

N. WIRTH — Wstęp do programowania systematycznego, wyd. 1 i 2
R. WIT — Metody programowania nieliniowego. Minimalizacja funkcji gładkich
K. ZORYCHTA, W. OGRYCZAK — Programowanie liniowe i całkowitoliczbowe