Deklaratywne podejście do wnioskowania o programach w języku logiki

Włodzimierz Drabent
Instytut Podstaw Informatyki PAN

Mirosława Miłkowska*
Instytut Informatyki, Uniwersytet Warszawski

Streszczenie

Praca omawia kwestię specyfikowania programów w języku logiki oraz przedstawia prostą, deklaratywną metodę dowodzenia poprawności i pełności programów w języku logiki.

1. Wprowadzenie

Poprawność programów ocenia się na podstawie analizy zgodności programu ze specyfikacją. Według pewnych opinii program w języku logiki jest swoją własną specyfikacją, a zatem w oczywisty sposób jest on poprawny. Czasami dodaje się, że dotyczy to tylko tzw. dobrze napisanych programów, programów deklaratywnych, bez sprecyzowania jednakże co dokładnie te pojęcia oznaczają. Według innych opinii wszelkie dowody poprawności programów w języku logiki, jako skomplikowane i zawiłe, będą jeszcze mniej przekonywujące od samych programów. Przypuszczalnie tego typu poglądy przyczyniły się do braku powszechnie przyjętych metod weryfikacji programów w języku logiki. Metody takie istnieją natomiast dla wielu innych paradygmatów programowania, przykładowo dla sekwencyjnych programów imperatywnych [6, 5], czy też dla programów funkcyjnych [3].

*Praca częściowo finansowana z grantu KBN nr 8 T11C 015 15.

Ponadto niektóre z proponowanych podejść (np. [2, 11]) odwołują się nie do deklaratywnej lecz do operacyjnej semantyki programów, niwecząc w ten sposób podstawową ideę programowania w logice wyrażoną przez R. Kowalskiego [9] w postaci równania *program = logika + sterowanie*. Podejście operacyjne jest ponadto bardziej złożone od deklaratywnego [7].

Pojęcie całkowitej poprawności programów obejmuje częściową poprawność oraz własność stopu. W przypadku programów w języku logiki na częściową poprawność składają się dwie własności:

- poprawność — wynik każdego skończonego obliczenia jest poprawny (tj. zgodny ze specyfikacją),

- pełność — każdy poprawny wynik może być uzyskany (o ile tylko obliczenie jest skończone).

Natomiast własność stopu oznacza tzw. czyste zakończenie programu — obliczenie musi być skończone i wolne od błędów wykonania takich jak np. grzęźnięcie (zapytanie negatywne nieustalone).

Wnioskowanie o własności stopu programu musi odwoływać się do semantyki operacyjnej, w szczególności uwzględniać stosowaną regułę obliczeniową (np. w Prologu na ogół jest stosowana reguła wyboru lewego atomu). Natomiast pozostałe własności programów, własności deklaratywne, niezależne od mechanizmu obliczeniowego (sterowania) mogą i powinny być dowodzone deklaratywnie.

Na przełomie lat 70-tych i 80-tych była rozważana prosta metoda dowodzenia poprawności programów bez negacji [4, 8]. Metoda ta nie została jednak w pełni doceniona i rozpowszechniona. Wręcz przeciwnie, inne podejście odwołujące się do semantyki operacyjnej zostało zaproponowane przez niektórych autorów [2, 11].

Niniejsza praca przedstawia prostą, deklaratywną metodę wnioskowania o poprawności i pełności programów w języku logiki. Prezentację rozpoczniemy od omówienia kwestii związanych ze specyfikowaniem programów. W kolejnym rozdziale przedstawimy metodę wnioskowania dla programów bez negacji, a następnie uogólnimy ją dla klasy programów z negacją. W następnym rozdziale wykażemy poprawność i pełność przykładowego programu z negacją. Pracę zakończymy informacjami na temat dalszych kierunków badań.

2. Specyfikacje programów w języku logiki

Zacznijmy od określenia kilku podstawowych pojęć. Pełne informacje można znaleźć w pracach [10, 2]. *Zapytanie* jest to koniunkcja atomów (literałów pozytywnych) lub zanegowanych atomów (literałów negatywnych). Przykładem zapytania jest np. `sort([3,7,1,8,4],S)`[1]. Zapytanie ustalone (tj. nie zawierające zmiennych) może odnieść sukces (odpowiedź twierdząca na dane zapytanie), skończoną porażkę (odpowiedź przecząca) lub obliczenie może być nieskończone. W przypadku zapytania nieustalonego (jak w powyższym przykładzie) odniesienie sukcesu oznacza uzyskanie tzw. odpowiedzi obliczonej (`S = [1,3,4,7,8]` w powyższym przykładzie). *Odpowiedzią* na zapytanie nazywamy obliczoną instancję zapytania. W naszym przykładzie odpowiedzią jest `sort([3,7,1,8,4],[1,3,4,7,8])`.

2.1. Specyfikacje programów bez negacji

Kwestię specyfikowania programów w języku logiki rozpoczniemy od przedstawienia prostego przykładu programu bez negacji. Posłużymy się w tym celu klasycznym predykatem `member`. Relację `member(e, l)` można opisać następująco: `member(e, l)` wtw gdy `e` jest elementem listy `l`.

Standardowa definicja predykatu `member` (zapisana w Prologu) jest postaci:

```
member( E, [ E | L ] ).
member( E, [ Z | L ] ) :-  member( E, L ).
```

Powyższy program nie jest jednak zgodny z podanym opisem. Przykładowo zapytanie `member(5, [ 5 | a ])` odniesie sukces, mimo że obiekt `[ 5 | a ]` nie jest listą, gdyż `a` nie jest listą.

W rzeczywistości powyższy program nie definiuje relacji bycia elementem listy, lecz nadzbiór tej relacji. Jest to zjawisko charakterystyczne dla programowania w języku logiki. Rozważając poprawność programów będziemy zatem pomijać zapytania "źle utypowione" i odpowiedzi typu `member(5, [ 5 | a ])` traktować jako dozwolone. Poprawność predykatu `member` należy zatem badać w odniesieniu do następującej specyfikacji:

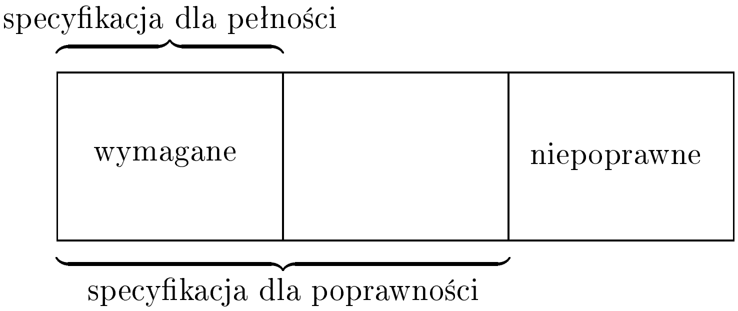
dla każdej odpowiedzi `member(e, l)`: jeśli `l` jest listą, to `e` jest elementem `l`.

Przechodząc do kwestii pełności zauważmy, że wymóg uzyskiwania odpowiedzi "źle utypowionych" nie miałby sensu. Program powinien natomiast obliczać wszystkie przypadki bycia elementem listy. A zatem, jeśli `L` jest listą i `e` jest elementem `L`, to program powinien udzielić takiej odpowiedzi, tzn. zapytanie `member(e, L)` powinno odnieść sukces, zaś na zapytanie `member(E, L)` powinniśmy uzyskać odpowiedź obliczoną `E = e`, a tym samym odpowiedź `member(e, L)`.

Analizując poprawność i pełność programów w języku logiki będziemy zatem w ogólności używać dwóch specyfikacji: specyfikacji dla poprawności oraz

[1] W Prologu identyfikator rozpoczynający się od wielkiej litery oznacza zmienną.

specyfikacji dla pełności, przy czym specyfikacja dla pełności stanowić będzie podzbiór specyfikacji dla poprawności. Ilustruje to poniższy diagram.



Specyfikację predykatu `member` możemy zatem przedstawić jako następującą parę zbiorów:

$$specS = \{member(e, l) \mid l \text{ jest listą} \rightarrow e \in l\}$$
$$specC = \{member(e, l) \mid l \text{ jest listą} \wedge e \in l\}$$

gdzie *specS* oznacza specyfikację dla poprawności, a *specC* specyfikację dla pełności (skrót S pochodzi od *soundness*, zaś C od *completeness*). *specS* jest zbiorem tych atomów, które mogą odnieść sukces, zaś *specC* zbiorem atomów, które muszą odnieść sukces.

Specyfikacje predykatów mogą być, tak jak w powyższym przykładzie, interpretacjami, mogą również być przedstawiane w postaci teorii. W dalszej części pracy będziemy zatem używać ogólniejszej notacji $spec \models A$, co odczytujemy jako A jest prawdziwe w (interpretacji, teorii) *spec*, zamiast $A \in spec$.

2.2. Specyfikacje programów z negacją

Wprowadzenie negacji do zapytania lub do programu oznacza konieczność rozszerzenia pojęcia poprawności oraz pełności. W programowaniu w logice jest stosowana reguła negacji przez skończoną porażkę [10], zgodnie z którą negację zapytania wyprowadzamy, jeśli zapytanie poniesie skończoną porażkę. Pojęcie poprawności oraz pełności programu dotyczy wszystkich skończonych obliczeń — zakończonych sukcesem lub porażką.

Kwestię specyfikacji dla programów z negacją rozpoczniemy od krótkiej analizy zapytań zawierających negatywne literały, pozostając przy programach bez negacji. Niech program bez negacji P będzie poprawny względem specyfikacji *specS* oraz pełny względem *specC*. Jeśli pewne zapytanie atomowe A poniesie skończoną porażkę, to $specC \models \neg A$ (z pełności względem *specC*). Skończona porażka odpowiada zatem poprawności zanegowanego zapytania względem specyfikacji dla pełności. Rozważmy zapytanie Q złożone z dwóch literałów: pozytywnego A i negatywnego $\neg B$ (czyli $Q = A, \neg B$). Jeśli Q odniesie sukces, to uzyskana odpowiedź będzie prawdziwa w interpretacji, w której literały pozytywne interpretujemy względem *specS*, zaś literały negatywne względem *specC*. Jeśli natomiast zapytanie Q poniesie porażkę, to $\neg Q$ powinno być prawdziwe w interpretacji, w której literały pozytywne interpretujemy względem *specC*, zaś literały negatywne względem *specS*.

Uwzględnianie właściwych specfikacji (*specS* lub *specC*) uzyskamy poprzez odpowiednie przemianowywanie nazw predykatów.

Specyfikacja, względem której będzie dokonywana analiza poprawności programu z negacją powinna opisywać dwa zbiory ustalonych atomów: zbiór atomów, które mogą odnieść sukces oraz zbiór atomów, które mogą ponieść porażkę. Zbiory te nie muszą być rozłączne, albowiem nie musimy wprowadzać konkretnych wymagań np. w stosunku do "źle utypowionych" zapytań. Natomiast specyfikacja dla pełności powinna obejmować dwa rozłączne zbiory: ustalonych atomów, które powinny odnieść sukces oraz tych, które powinny ponieść skończoną porażkę. Dwa zbiory określające pełność programu mogą zatem stanowić dopełnienie zbiorów określających poprawność.

Specyfikacją programu z negacją nazywamy parę $(specS, specC)$, gdzie *specS* i *specC* są interpretacjami takimi, że $specC \subseteq specS$.

Zbiór *specS* wraz z dopełnieniem zbioru *specC* tworzy specyfikację poprawności, natomiast *specC* wraz z dopełnieniem *specS* — specyfikację pełności. Przykładowo poprawność i pełność programu P względem specyfikacji $spec = (specS, specC)$ dla ustalonego zapytania A oznacza spełnienie następujących warunków:

— jeśli A odniesie sukces, to $specS \models A$,
— jeśli A poniesie porażkę, to $specC \models \neg A$,
— jeśli $specC \models A$, to A odnosi sukces,
— jeśli $specS \models \neg A$, to A ponosi skończoną porażkę.

3. Wnioskowanie o programach bez negacji

W niniejszym rozdziale przedstawimy metodę dowodzenia poprawności i pełności dla programów bez negacji. Metoda ta jest prostsza od ogólniejszej metody wnioskowania o programach z negacją. Negacja występuje co prawda w wielu programach, jednakże na ogół jest zlokalizowana w obrębie niewielkiej liczby predykatów. Własności pozostałych predykatów można dowodzić stosując prostszą metodę dla programów bez negacji.

Pełną postać wszystkich twierdzeń, ich dowody oraz wykazanie, że podejście operacyjne nie jest mocniejsze od prezentowanego podejścia deklaratywnego można znaleźć w pracy [7].

Program bez negacji jest *poprawny* względem specyfikacji *specS* wtw gdy każda odpowiedź programu Q spełnia warunek $specS \models Q$.

Twierdzenie (Poprawność, programy bez negacji)
Niech P — program bez negacji, *specS* — specyfikacja. Jeśli

$$specS \models P,$$

to program P jest poprawny względem specyfikacji *specS*.

Wykazanie poprawności programu bez negacji wymaga zatem (niezależnej) analizy wszystkich klauzul programu. Na przykład obie klauzule predykatu `member` w oczywisty sposób spełniają specyfikację podaną w p. 2.1.

Program jest *pełny* dla zapytania Q_0, względem specyfikacji *specC*, jeśli dla każdej instancji Q zapytania Q_0 takiej, że $specC \models Q$, Q jest instancją pewnej odpowiedzi na zapytanie Q_0.

Na przykład dla zapytania $Q =$ `member(E,[1,E,3])` i specyfikacji *specC* z p. 2.1 zachodzi $specC \models member(1, [1, 1, 3])$ i $specC \models member(2, [1, 2, 3])$. `member(1,[1,1,3])` jest odpowiedzią na Q (`E = 1` jest odpowiedzią obliczoną), zaś `member(2,[1,2,3])` nie jest odpowiedzią na zapytanie Q, jest natomiast instancją odpowiedzi `member(X,[1,X,3])`.

W dowodzie pełności programu P korzystamy z konstrukcji ONLY-IF(P) [1]. Intuicyjnie ONLY-IF(P) oznacza odwrócenie implikacji we wszystkich klauzulach programu P i połączenie klauzul definiujących ten sam predykat. Przykładowo ONLY-IF(P) dla programu z p. 2.1 jest następującej postaci:

$$\begin{aligned} member(x, y) \rightarrow \quad & (\exists\, E, L\ x = E \ \wedge\ y = [E|L]) \vee \\ & (\exists\, E, Z, L\ x = E \ \wedge\ y = [Z|L] \ \wedge\ member(E, L)) \end{aligned}$$

Dodatkowo jest potrzebna specyfikacja dla równości, która może być następującej postaci:

$$spec_{=} = \{t{=}t \mid t \text{ jest termem ustalonym}\}.$$

Podamy teraz warunek konieczny i wystarczający dla pełności programu.

Twierdzenie (Pełność, programy bez negacji)
Niech P — program bez negacji, Q — zapytanie, *specC* — specyfikacja. Jeśli

(i) $specC \cup spec_{=} \models$ ONLY-IF(P) oraz

(ii) obliczenie jest skończone (istnieje skończone SLD-drzewo dla P i Q),

to program P jest pełny dla zapytania Q względem *specC*.

3.1. Wnioskowanie o programach z negacją

Wprowadźmy dla każdego symbolu predykatywnego p występującego w programie nowy symbol p'. Niech $\mathcal{Q}$ oznacza formułę (zapytanie lub klauzulę programu) lub zbiór formuł (program). Przez $\mathcal{Q}'$ będziemy oznaczać formułę (zbiór formuł) uzyskany z $\mathcal{Q}$ poprzez zamianę symbolu p występującego w każdym negatywnym literale symbolem p' (dla każdego p). Analogicznie $\mathcal{Q}''$ uzyskujemy poprzez zamianę symbolu p występującego w pozytywnym literale symbolem p'. Jeśli $\mathcal{I}$ oznacza specyfikację, to $\mathcal{I}'$ jest interpretacją uzyskaną z $\mathcal{I}$ poprzez zamianę każdego symbolu p symbolem p'.

Program P jest *poprawny* względem specyfikacji $spec = (specS, specC)$ wtw gdy

(i) każda odpowiedź Q spełnia: $specS \cup specC' \models Q'$,

(ii) jeśli Q ponosi skończoną porażkę, to $specS \cup specC' \models \neg Q''$.

A zatem jeśli program P jest poprawny względem specyfikacji $spec = (specS, specC)$, to każda odpowiedź Q spełnia następujące warunki: dla każdego pozytywnego literału A występującego w Q zachodzi $specS \models A$, a dla każdego negatywnego literału $\neg A$ z Q zachodzi $specC \models \neg A$.

Program P jest *pełny* względem specyfikacji $spec = (specS, specC)$ dla zapytania Q wtw gdy

(i) $specS \cup specC' \models \neg Q'$ implikuje, że Q ponosi skończoną porażkę;

(ii) $specC \cup specS' \models Q'\sigma$ implikuje, że na zapytanie Q uzyskamy odpowiedź $Q\theta$ bardziej ogólną od $Q\sigma$.

W przypadku zapytań złożonych z jednego ustalonego literału warunek pełności można przedstawić w następujący sposób:

$$\begin{array}{lcl} specC \models A & \Longrightarrow & A \text{ odnosi sukces} \\ specS \models \neg A & \Longrightarrow & A \text{ ponosi skończoną porażkę} \end{array}$$

Twierdzenie (Poprawność i pełność, programy z negacją)
Niech P — program z negacją, $spec = (specS, specC)$ — specyfikacja.

1. Jeśli

   (a) $specS \cup specC' \models P'$

   (b) $specS \cup specC' \cup spec_= \models \text{ONLY-IF}(P'')$

   to program P jest poprawny względem specyfikacji $spec$.

2. Jeśli warunki (a) i (b) są spełnione oraz obliczenie dla zapytania Q jest skończone, to program P jest pełny dla zapytania Q względem specyfikacji $spec = (specS, specC)$.

4. Przykład

Sposób specyfikowania oraz metodę dowodzenia poprawności i pełności programów w języku logiki zilustrujemy na przykładzie programu definiującego relację bycia podzbiorem, przy założeniu, że podzbiór jest reprezentowany jako lista bez powtórzeń. Niech P oznacza następujący program:

```
subset( [], L ).
subset( [H | T], LH ) :-
        select( H, LH, L ), subset( T, L ), not member( H, T ).
select( H, [ H | L ], L ).
select( H, [ X | L ], [ X | LH ] ) :-  select( H, L, LH ).
member( X, [ X | L ] ).
member( X, [ Y | L ] ) :-  member( X, L ).
```

Specyfikację, względem której będziemy analizowali nasz program, definiujemy w sposób następujący:

$$spec = (specS,\ specC),$$

$$specS = sS_m \cup sS_{sel} \cup sS_{subs}, \quad specC = sC_m \cup sC_{sel} \cup sC_{subs},$$

gdzie (sS_m, sC_m), (sS_{sel}, sC_{sel}) i (sS_{subs}, sC_{subs}) są następującymi specyfikacjami predykatów $member/2$, $select/2$ oraz $subset/2$, przy czym $l \approx m$ oznacza, że listy l i m zawierają dokładnie te same elementy, zaś $listd(l)$ oznacza, że l jest listą zawierającą różne elementy (bez powtórzeń).

$$\begin{array}{lcl}
sS_m & = & \{member(x,l) \mid l \text{ jest listą} \rightarrow x \in l\} \\
sC_m & = & \{member(x,l) \mid l \text{ jest listą} \wedge x \in l\} \\
sS_{sel} & = & \{select(e,l,m) \mid l \text{ jest listą } \rightarrow e \in l \wedge m \text{ jest listą} \wedge l \approx [e|m]\} \\
sC_{sel} & = & \{select(e,l,m) \mid l \text{ i } m \text{ są listami takimi, że} \\
 & & \quad l = [e_1, \ldots, e_i, e, e_{i+1}, \ldots e_k], m = [e_1, \ldots, e_i, e_{i+1}, \ldots e_k], \\
 & & \quad 0 \leq i \leq k\} \\
sS_{subs} & = & \{subset(l,m) \mid m \text{ jest listą} \rightarrow listd(l) \wedge l \subseteq m\} \\
sC_{subs} & = & \{subset(l,m) \mid m \text{ jest listą} \wedge listd(l) \wedge l \subseteq m\}
\end{array}$$

Najpierw wykażemy, że program P spełnia warunek (a) twierdzeń o poprawności i pełności. Rozważmy predykat $subset/2$. W przypadku pierwszej klauzuli $subset/2$ powinniśmy wykazać, że $subset([\,], l) \in sS_{subs}$, co w oczywisty sposób zachodzi, gdyż pusta lista oznacza zbiór pusty, który jest podzbiorem każdego zbioru i nie zawiera powtarzających się elementów. Przejdźmy do drugiej klauzuli predykatu $subset/2$. Załóżmy, że

$$specS \cup specC' \models select(h,lh,l), subset(t,l), not\ member'(h,t). \qquad (A1)$$

Chcemy pokazać, że $subset([h|t], lh) \in sS_{subs}$. Przyjmijmy zatem, że lh jest listą (w przeciwnym przypadku warunek jest w oczywisty sposób spełniony).

Z założenia $(A1)$ wynika że:

(1) $select(h, lh, l) \in sS_{sel}$, a ponieważ lh jest listą, to $h \in lh$ oraz l jest listą taką, że $lh \approx [h|l]$;

(2) $subset(t, l) \in sS_{subs}$, a skoro l jest listą (z (1)), to zachodzi $listd(t)$ oraz $t \subseteq l$, stąd i z (1) otrzymujemy $[h|t] \subseteq lh$;

(3) $member(h, t) \notin sC_m$ czyli $h \notin t$ (ponieważ t jest listą), stąd i z (2) otrzymujemy $listd([h|t])$.

Wykazaliśmy zatem, że $[h|t] \subseteq lh$ oraz $listd([h|t])$, co kończy dowód warunku (a) dla klauzul predykatu $subset/2$. Wykażemy teraz spełnienie warunku (b) przez predykat $subset/2$, czyli pokażemy, że

$$specS \cup specC' \cup spec_{=} \models subset'(S, M) \rightarrow S = [\,] \vee$$
$$\exists H, T, L\, (S = [H|T] \wedge select'(H, M, L) \wedge subset'(T, L) \wedge not\, member(H, T))$$

Załóżmy, że s, m spełniają warunek $specS \cup specC' \models subset'(s, m)$, czyli $subset(s, m) \in sC_{subs}$. A zatem m jest listą, s jest listą składającą się z różnych elementów oraz $s \subseteq m$. Przypadek, gdy $s = [\,]$ jest oczywisty. Załóżmy zatem, że s jest listą niepustą, czyli $s = [h|t]$. Skoro $h \in s$ oraz $s \subseteq m$, to $h \in m$ i m jest postaci: $m = [m_1, \ldots, m_{i-1}, h, m_{i+1}, \ldots, m_k]$. Weźmy za listę l listę $[m_1, \ldots, m_{i-1}, m_{i+1}, \ldots, m_k]$. W oczywisty sposób zachodzi: $select(h, m, l) \in sC_{sel}$. Z warunku $listd(s)$, otrzymujemy $listd(t)$ oraz $h \notin t$, a zatem $member(h, t) \notin sS_m$. Z definicji listy l oraz warunku $s = [h|t] \subseteq m$ wynika, że $t \subseteq l$, czyli $subs(t, l) \in sC_{subs}$. W ten sposób wykazaliśmy spełnienie prawej strony implikacji. Wykazanie spełnienia warunków (a) i (b) przez pozostałe predykaty jest znacznie prostsze i je pomijamy. W ten sposób wykazaliśmy, że program P jest poprawny względem podanej specyfikacji *spec*.

Rozważmy teraz zapytania postaci $subset(L, M)$, gdzie L jest zmienną, a M jest ustaloną listą. Jeśli udowodnimy, że dla tego rodzaju zapytań program poprawnie kończy swoje działanie (dla pewnej reguły obliczeniowej i pewnej strategii poszukiwania), to będziemy wiedzieli, że program jest pełny dla takich zapytań. A to oznacza, że dla dowolnej listy, reprezentującej pewien zbiór $\mathcal{Z}$, zostaną wygenerowane wszystkie listy (i wszystkie ich permutacje) reprezentujące wszystkie możliwe podzbiory zbioru $\mathcal{Z}$. Z drugiej strony, jeśli zażądamy podania wszystkich odpowiedzi na tego rodzaju zapytanie i obliczenie (poprawnie) zakończy się, to dzięki powyższej analizie będziemy wiedzieli, że wszystkie odpowiedzi wymagane przez specyfikację zostały udzielone. Tak na przykład będzie jeśli użyjemy SICStus Prologu [13].

5. Podsumowanie i dalsze prace

W niniejszej pracy przedstawiliśmy sposób specyfikowania programów w języku logiki oraz deklaratywną metodę dowodzenia poprawności i pełności programów w języku logiki.

Każdy programista pisząc program w Prologu powinien znać tzw. zamierzoną interpretację wszystkich pisanych przez siebie predykatów. Specyfikacja jest tylko formalnym zapisem takich interpretacji. Uważamy, że zaproponowana przez nas formalna metoda dowodzenia poprawności i pełności programów w logice jest formalizacją nieformalnego wnioskowania o programach, które powinien przeprowadzać każdy programista w celu przekonania się o poprawności i pełności swego programu. W wielu prostszych przypadkach nieformalna (ale dokładna) analiza może być całkowicie wystarczająca. Natomiast pełna formalna analiza może być konieczna w bardziej złożonych przypadkach. Użycie w tym celu systemu automatycznego dowodzenia twierdzeń (np. PVS [12]) wydaje się być możliwe, wymaga jednak dalszych prac.

Bibliografia

[1] K. R. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B*, chapter 10, pages 493–574. Elsevier Science Publishers B.V., 1990.

[2] K. R. Apt. *From Logic Programming to Prolog.* Prentice-Hall, 1997.

[3] R. Bird and Ph. Wadler. *Introduction to Functional Programming.* Prentice Hall, Englewood Cliffs, NJ, 1988.

[4] K. L. Clark. Predicate logic as computational formalism. Technical Report Technical Report 79/59, Imperial College, London, December 1979.

[5] Ole-Johan Dahl. *Verifiable Programming.* Prentice Hall, 1992.

[6] E. W. Dijkstra. *Umiejętność programowania.* WNT, 1985.

[7] W. Drabent and M. Miłkowska. Proving correctness and completeness of normal programs — a declarative approach. Praca przyjęta na ICLP'2001.

[8] C. J. Hogger. Derivation of logic programs. *Journal of ACM*, 28(2):372–392, 1981.

[9] R. Kowalski. Algorithm = logic + control. *Comm. ACM*, 22(7):424–436, 1979.

[10] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 2nd edition, 1987.

[11] D. Pedreschi and S. Ruggieri. Verification of logic programs. *Journal of Logic Programming*, 39(1-3):125–176, 1999.

[12] S.Owre, N.Shankar, J.M.Rushby, and D.W.J.Stringer-Calvert. *PVS Language Reference.* Computer Science Laboratory, SRI International, Menlo Park, CA, September 1998.

[13] Swedish Institute of Computer Science. *SICStus Prolog User's Manual*, 1995.