Programowanie deklaratywne

Artur Michalski
Informatyka II rok

Plan wykładu

- Wprowadzenie do języka Prolog
- Budowa składniowa i interpretacja programów prologowych
- Listy, operatory i operacje arytmetyczne
- *Sterowanie mechanizmem nawrotów*
- Predefiniowane procedury prologowe
- Styl i technika programowania w Prologu

Sterowanie mechanizmem nawrotów

- Mechanizm odcięć (ang. *cuts*)
- Przykłady wykorzystywania odcięć w programie prologowym
- Negacja przez niepowodzenie
- Problemy związane z zastosowaniem odcięć i negacji w Prologu

Mechanizm odcięć (ang. *cuts*)

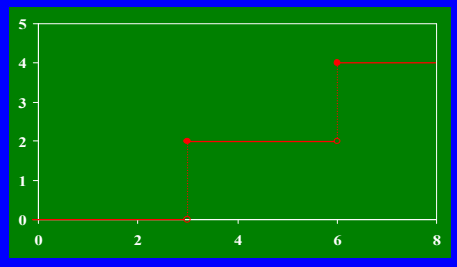
Mechanizm nawrotów:

- zapewnia weryfikację alternatywnych sposobów osiągnięcia celu,
- analizuje systematycznie wszystkie kolejne alternatywy,
- może okazać się nieefektywny, jeśli nie zostanie zastosowana odpowiednia strategia nawracania

Przykład 1

Zdefiniujemy funkcję:

```
f(X,0):- X<3.
f(X,2):- 3=<X,X<6.
f(X,4):- 6=<X.
```

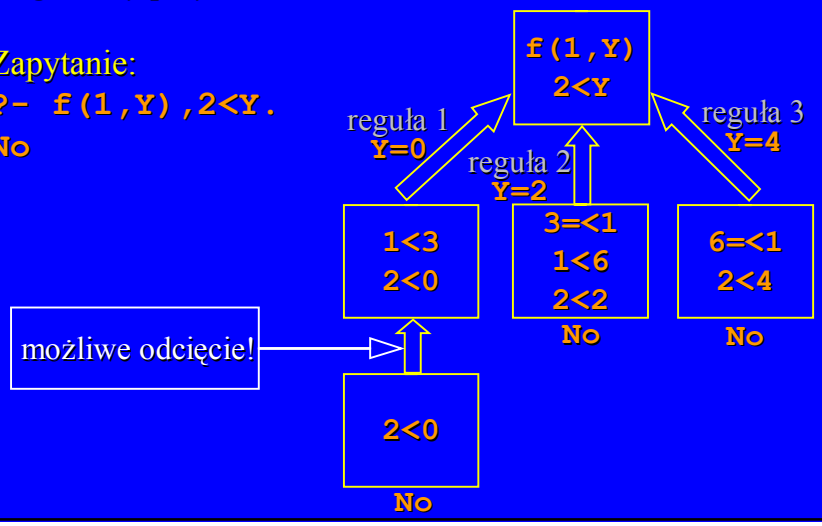


Mechanizm odcięć (ang. *cuts*)

ciąg dalszy przykładu 1:

Zapytanie:

```
?- f(1,Y),2<Y.
No
```



Mechanizm odcięć (ang. *cuts*)

ciąg dalszy przykładu 1:

Komentarz:

- Wszystkie trzy reguły wzajemnie się wykluczają (uzupełniają), więc któraś z nich zawsze zakończy się sukcesem
- Jeżeli warunki którejkolwiek z reguł są spełnione, nie ma potrzeby sprawdzać pozostałych alternatyw
- Nasze rozwiązanie jest zatem nieefektywne, bo po spełnieniu którejkolwiek z reguł następuje nawrót do pozostałych alternatyw

Mechanizm odcięć (ang. *cuts*)

ciąg dalszy przykładu 1:

Wersja poprawiona (z odcięciami):

```
f(X,0):- X<3,!.
f(X,2):- 3=<X,X<6,!.
f(X,4):- 6=<X.
```

Mechanizm odcięć uniemożliwia dokonanie nawrotu powyżej miejsca, w którym został wstawiony znak odcięcia (`!`).

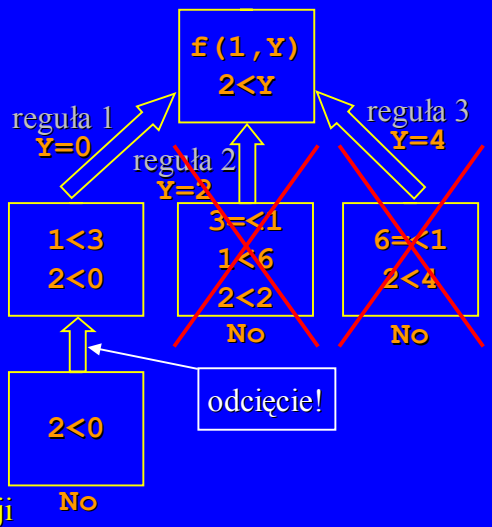
Mechanizm odcięć (ang. *cuts*)

ciąg dalszy przykładu 1:

Zapytanie:

```
?- f(1,Y),2<Y.
No
```

Uzyskany rezultat jest taki sam, lecz zmniejszył się koszt weryfikacji celu - wzrosła efektywność programu. Zastosowanie odcięć spowodowało zmiany tylko w interpretacji proceduralnej programu.



Mechanizm odcięć (ang. *cuts*)

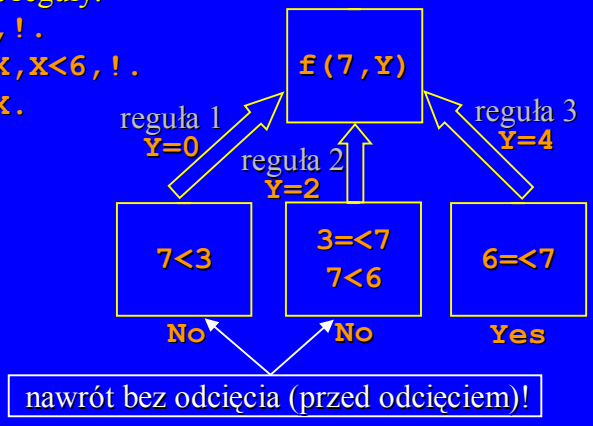
Przykład 2

Poprawiona wersja reguły:

```
f(X,0):- X<3,!.
f(X,2):- 3=<X,X<6,!.
f(X,4):- 6=<X.
```

inne zapytanie:

```
?- f(7,Y).
Y=4
Yes
```



Mechanizm odcięć (ang. *cuts*)

ciąg dalszy przykładu 2:

Komentarz:

- Jeżeli warunek pierwszej reguły nie jest spełniony (`7<3`), to pierwszy warunek drugiej reguły (`3=<7`) jest na pewno prawdziwy, gdyż stanowi jego logiczne dopełnienie; wniosek: pierwszy warunek drugiej reguły jest nadmiarowy (zbędny)
- Podobna sytuacja ma miejsce w przypadku warunku trzeciej reguły (`6=<X`): nie spełnienie drugiego warunku drugiej reguły (`X<6`) oznacza automatycznie prawdziwość warunku trzeciej reguły, więc sprawdzanie tego warunku (`6=<X`) jest nadmiarowe

Mechanizm odcięć (ang. *cuts*)

ciąg dalszy przykładu 2:

Definicja wersji „minimalistycznej” reguły:
Jeżeli X<3, to Y=0,
w przeciwnym przypadku jeżeli X<6, to Y=2,
w przeciwnym przypadku Y=4.

Zapis w Prologu:

```
f(X,0):- X<3,!.
f(X,2):- X<6,!.
f(X,4).
```

Mechanizm odcięć (ang. *cuts*)

ciąg dalszy przykładu 2:
Niebezpieczeństwa takiego użycia mechanizmu odcinania

```
f(X,0):- X<3.
f(X,2):- X<6.
f(X,4).
```

gdyby zabrakło odcięć!

to przy zapytaniu:

```
?- f(1,Y).
Y=0;
Y=2;
Y=4;
No
```

błąd!

błąd!

Zastosowanie odcięć może powodować zmiany nie tylko w interpretacji proceduralnej, lecz również w interpretacji deklaratywnej programu prologowego.

Mechanizm odcięć (ang. *cuts*)

Formalna interpretacja odcięć

Odcięcie symbolizuje cel, który jest natychmiast spełniony, gdy tylko zostanie osiągnięty znak odcięcia. Wszystkie cele, które zostały do tej chwili spełnione (w klauzuli zawierającej odcięcie!) nie będą analizowane powtórnie w celu weryfikacji alternatywnych definicji, odpowiadających im klauzul (czyli alternatywnych sposobów ich spełnienia).

Mechanizm odcięć (ang. *cuts*)

Formalna interpretacja odcięć

Załóżmy, że dana jest klauzula **H** postaci:
H:- B1,B2,...,Bm,!,...,Bn.
oraz dany cel **G**, który został dopasowany do klauzuli **H**. W momencie, w którym zostanie osiągnięty znak odcięcia (**!**), wszystkie podcele **B1,B2,...,Bm** są już spełnione. Po przekroczeniu znaku odcięcia rozwiązania tych podcelów zostają „zamrożone", zaś alternatywne sposoby ich spełnienia nie będą analizowane. Również cel **G** jest już możliwy do spełnienia tylko przez klauzulę **H** i inne reguły zdefiniowane później niż **H**, nagłówek których pasuje do **G** nie będą sprawdzane.

Mechanizm odcięć (ang. *cuts*)

Przykład

Zbiór klauzul:

```
c:- p,q,r,!,s,t,u.
c:- v.
a:- b,c,d.  % definicje b i d gdzieś poniżej
```

Cel:

```
?- a.
```

Odcięcie w pierwszej regule **c** uniemożliwia nawrót do innych dopasowań dla podcelów **p,q,r** , jak również do drugiej reguły **c**, jeżeli pierwsza byłaby spełniona. Nawroty są jednak nadal możliwe w ramach podcelów **s,t,u**. Podobnie odcięcie wpłynie tylko na sposób osiągania celu **c**, natomiast dla celu głównego **a** pozostaje ono „niewidoczne", więc nawrót w ramach listy warunków **b,c,d** jest nadal możliwy.

Przykłady wykorzystywania odcięć w programie prologowym

Obliczanie maksimum

Klauzula **max(X,Y,Max)** ma zwracać spośród dwóch wartości **X** i **Y** wartość większą jako **Max**.

Definicja 1
Max=X, o ile
X jest większe lub równe Y
lub
Max=Y, o ile
X jest mniejsze od Y.

Zapis w Prologu:

```
max(X,Y,X):- X>=Y.
max(X,Y,Y):- X<Y.
```

Definicja 2
Max=X, o ile
X jest większe lub równe Y
w przeciwnym przypadku
Max=Y.

Zapis w Prologu:

```
max(X,Y,X):- X>=Y,!.
max(X,Y,Y).
```

Przykłady wykorzystywania odcięć w programie prologowym

Szukanie pierwszego wystąpienia na liście

```
member(X,[X|L]).
member(X,[Y|L]):- member(X,L).
```

Dotychczasowa wersja operacji **member** jest „niedeterministyczna”- znalezione zostanie dowolne wystąpienie elementu na liście. Znalezienie tylko pierwszego wystąpienia wymaga innej wersji:

```
member1(X,[X|L]):- !.
member1(X,[Y|L]):- member1(X,L).
```

Przykładowe zapytanie:

```
?- member1(X,[a,b,c]).
X=a;
No
```

nawrót nie powoduje wybrania reszty elementów; brak niedeterminizmu!

Przykłady wykorzystywania odcięć w programie prologowym

Wstawianie elementu do listy bez powtórzeń

Klauzula **add1(X,L,L1)** umieszcza element **X** na liście **L**, o ile go jeszcze tam nie ma i w efekcie powstaje lista **L1**.

Definicja

Jeżeli X należy do L, to L1=L
w przeciwnym przypadku L1
jest równe L powiększonemu o X.

Zapis w Prologu:

```
add1(X,L,L):- member(X,L),!.
add1(X,L,[X|L]).
```

Przykłady wykorzystywania odcięć w programie prologowym

Wstawianie elementu do listy bez powtórzeń c.d.

Przykładowe zapytania:

```
?- add1(a,[b,c],L).
L=[a,b,c]
Yes

?- add1(X,[b,c],L).
L=[b,c]
X=b
Yes

?- add1(a,[b,c,X],L).
L=[b,c,a]
X=a
Yes
```

Brak odcięcia w definicji spowodowałby błędy logiczne!

```
?- add1(a,[a,b,c],L).
L=[a,b,c];
L=[a,a,b,c]
Yes
```

błąd!

Przykłady wykorzystywania odcięć w programie prologowym

Grupowanie w kategorie

Treść zadania

Dysponujemy bazą danych o rozgrywkach tenisowych w pewnym klubie sportowym, reprezentowanych w postaci faktów: **beat(<zwycięzca>,<pokonany>)**. Naszym celem jest podział graczy na trzy kategorie:
winner - gracze, którzy wygrali wszystkie swoje mecze,
fighter - gracze, którzy część meczy wygrali a część przegrali,
sportsman - gracze, którzy przegrali wszystkie swoje mecze.

Wynik ma być reprezentowany za pomocą relacji
class(<gracz>,<kategoria>).

Przykłady wykorzystywania odcięć w programie prologowym

Grupowanie w kategorie c.d.

Definicje

X należy do kategorii **`fighter`**, o ile istnieje taki Y, którego X pokonał i istnieje taki Z, który pokonał X.

X należy do kategorii **`winner`**, o ile istnieje taki Y, którego X pokonał i nie istnieje taki Z, który pokonał X.

X należy do kategorii **`sportsman`**, o ile nie istnieje taki Y, którego X pokonał i istnieje taki Z, który pokonał X.

definicje wymagają zastosowania logicznej negacji!

Przykłady wykorzystywania odcięć w programie prologowym

Grupowanie w kategorie c.d.

Definicja alternatywna (bez negacji)

Jeżeli X pokonał kogoś i został przez kogoś pokonany,
to należy do kategorii **`fighter`**,
w przeciwnym przypadku jeżeli X pokonał kogoś,
to należy do kategorii **`winner`**,
w przeciwnym przypadku jeżeli X został przez kogoś pokonany,
to należy do kategorii **`sportsman`**.

Zapis w Prologu

```
class(X,fighter):- beat(X,_),beat(_,X),!.
class(X,winner):- beat(X,_),!.
class(X,sportsman):- beat(_,X).
```

Przykłady wykorzystywania odcięć w programie prologowym

Grupowanie w kategorie c.d.

Przykład (niebezpieczeństwa wynikające z odcięć)

```
beat(tom,jim).
beat(ann,tom).
beat(pat,jim).

?- class(tom,C).
C=fighter;
No

?- class(tom,sportsman).
Yes
```

Dobrze!

Źle!

Negacja przez niepowodzenie

W Prologu istnieje systemowy sterujący predykat **`fail`**, który *nigdy nie jest spełniony* i prowadzi do porażki celu nadrzędnego. Stosowany jest do wymuszania następnych (alternatywnych) sposobów spełniania celu.

Przykład

Mamy bazę danych o osobach zapisaną w postaci relacji **`person`**. Należy znaleźć wszystkie osoby z naszej bazy.

Zapis w Prologu:

```
find:- person(X), fail.
find.
```

Błąd spowoduje nawrót i próbę spełnienia **`person`** w inny sposób!

Negacja przez niepowodzenie

Przykład

Wyrazić w Prologu zdanie „Maria lubi wszystkie zwierzęta oprócz węży”.

Jeżeli X jest wężem, to „Maria lubi X” nie jest prawdą, w przeciwnym przypadku jeżeli X jest zwierzęciem, to Maria lubi X.

Zapis w Prologu:

```
likes(mary,X):- snake(X),!,
                fail.
likes(mary,X):- animal(X).
```

`fail` powoduje porażkę, ale nawrót nie jest możliwy z powodu odcięcia!

Negacja przez niepowodzenie

Przykład

Predykat `different(X,Y)` jest spełniony, jeżeli `X` i `Y` są różne.

Interpretacja różnicy:
- `X` i `Y` to różne napisy
- `X` i `Y` to niedopasowane termy
- `X` i `Y` to wartości wyrażeń `X` i `Y` są różne

Dla interpretacji drugiej:

```
different(X,X):- !,fail.
different(X,Y).
```

albo:

```
different(X,Y):- X=Y,!,fail; true.
```

systemowy zawsze spełniony!

Negacja przez niepowodzenie

W Prologu istnieje systemowy predykat unarny `not(X)`, który jest spełniony, jeżeli `X` nie jest spełnione (nie daje się wywieść).

Definicja negacji
Jeżeli cel `X` jest spełniony, to cel `not(X)` nie jest spełniony, w przeciwnym przypadku `not(X)` jest spełnione.

Zapis w Prologu:

```
not(X):- X,!,fail.
not(X).
```

U podstaw teoretycznych negacji przez niepowodzenie leży założenie o „zamkniętości świata", które mówi, że jeżeli prawdziwości faktu nie można wykazać za pomocą dostępnych danych, to negacja faktu jest prawdziwa.

Negacja przez niepowodzenie

Zastosowanie negacji

Negacja przez niepowodzenie nie jest dokładnym odzwierciedleniem negacji logicznej (matematycznej), co nie pozostaje bez wpływu na działanie programów prologowych.

Zastosowanie negacji do dotychczasowych przykładów:

```
likes(mary,X):- animal(X), not(snake(X)).

different(X,Y):- not(X=Y).

class(X,fighter):-beat(X,_),beat(_,X).
class(X,winner):-beat(X,_),not(beat(_,X)).
class(X,sportsman):-beat(_,X), not(beat(X,_)).
```

Problemy związane z zastosowaniem odcięć i negacji w Prologu

Zalety mechanizmu odcięć:

- odcięcia zwiększają efektywność programu prologowego - programista jawnie wskazuje, które rozwiązania alternatywne są niepotrzebne
- odcięcia pozwalają zapisać wzajemnie wykluczające się relacje (*jeżeli...to...w przeciwnym przypadku...*)

Wady:

- wpływ na proceduralną i deklaratywną interpretację programu prologowego prowadzący często do znacznych różnic między tymi interpretacjami
- zmiana porządku klauzul i celów wpływa jedynie na efektywność programu, gdy nie ma w nim odcięć, ale zmienia stronę deklaratywną, gdy odcięcia w nim występują

Problemy związane z zastosowaniem odcięć i negacji w Prologu

Wpływ odcięć na interpretację deklaratywną

```
p:- a,b.
p:- c.
```

Interpretacja deklaratywna: `p <=> (a ∧ b) ∨ c.`

Zmiana kolejności klauzul nie zmieni tej interpretacji!

```
p:- a,!,b.
p:- c.
```

Interpretacja deklaratywna: `p <=> (a ∧ b) ∨ (~a ∧ c).`

Jeżeli zmienimy kolejność reguł:

```
p:- c.
p:- a,!,b.
```

to interpretacja ma postać: `p <=> c ∨ (a ∧ b).`

Problemy związane z zastosowaniem odcięć i negacji w Prologu

Wpływ odcięć na interpretację deklaratywną

Wyróżniamy dwa rodzaje odcięć:

- odcięcia „czerwone” - to takie, które zmieniają interpretację deklaratywną programu, utrudniają jego zrozumienie i powodują utratę pewnych rozwiązań
- odcięcia „zielone” - takie, które nie wpływają na interpretację deklaratywna programu, nie zmniejszają jego czytelności i zachowują wszystkie rozwiązania (choć obcinają drzewo poszukiwań)

Zastosowanie odcięć „czerwonych” wymaga dużej ostrożności przy programowaniu.

Problemy związane z zastosowaniem odcięć i negacji w Prologu

Zastosowanie predykatu `not` niesie ze sobą wszystkie zagrożenia, wynikające z niewłaściwego zastosowania odcięć.

Przykład

Zapytanie w Prologu:

```
?- not(human(tom)).
Yes
```

Odpowiedź nie oznacza, że „`tom` nie jest człowiekiem”, lecz nie ma dość informacji, żeby stwierdzić, że „`tom` jest człowiekiem”. Interpreter nie próbuje dowieść negacji celu, lecz dowodzi celu prostego i jeśli wywód ten się nie powiedzie, zakłada, że negacja jest prawdziwa. Jest to przejaw założenia o zamkniętości świata.

Problemy związane z zastosowaniem odcięć i negacji w Prologu

Zastosowanie predykatu `not` niesie ze sobą wszystkie zagrożenia, wynikające z niewłaściwego zastosowania odcięć.

Przykład

```
good_standard(jeanluis).
expensive(jeanluis).
good_standard(francesco).
reasonable(R):- not(expensive(R)).
?- good_standard(X),reasonable(X).
X=francesco
?- reasonable(X),good_standard(X).
No
```

Różne odpowiedzi są efektem wiązania zmiennej `X` w pierwszym zapytaniu już w pierwszym celu i brakiem takiego wiązania w drugim zapytaniu. Wiązanie zmiennych zmienia „siłę” negacji.

Problemy związane z zastosowaniem odcięć i negacji w Prologu

Podsumowanie

Odcięcie i negacja powinny być wykorzystywane z należytą ostrożnością. Nie oznacza to jednak, iż należy z nich całkowicie zrezygnować. Są one często pomocne w zwiększaniu efektywności programu, a czasami wręcz niezbędne w znalezieniu rozwiązania w sensownym czasie. Problemy, które wynikają z ich zastosowania występują również w innych językach programowania deklaratywnego.