Programowanie w logice i funkcyjne (Prolog) (prezentacja niekompletna i przed korektą)

Konrad Zdanowski

Outline

Gdzie korzystać z Prologu?

- Sztuczna inteligencja
  - systemy eksperckie,
  - przetwarzanie języka,
  - obliczenia symboliczne,
  - wnioskowania,
  - ...
- Struktura bazy wiedzy (reguł) w Prologu jest zapisywana w pliku w sposób dość czytelny.
- Podstwowe zrozumienie programu w Prologu nie wymaga znajomości zaawansowanych technik.

Prolog – składnia

- Stałe
  - atomy – zaczynają się od małej litery: zero, ala, slonce, jan_kowalski, ...
  - dowolne ciągi znaków w apostrofach: 'Ala', 'jan kowalski', ...
  - stałe liczbowe: 20, 1.23, -1, 4.32e4, 3.45e-2, ...
- Zmienne – zaczynają się dużą literą lub podkreśleniem: X, Y, Z, Ala, _X, _Y, ...
- Zmienna specjalna: podkreślenie _, używamy jej, kiedy nazwa zmiennej nie ma znaczenia (każde wystąpienie _ to inna zmienna).

Termy w Prologu

- Term to

$$\mathit{Term} ::= \mathit{Zmienna} | \mathit{Atom} | \mathit{Liczba} | \text{Term\_złożony}.$$

- Term_złożony to

$$\text{Term\_złożony} ::= \mathit{Atom}(\mathit{Term}, \ldots, \mathit{Term}).$$

- Termy złożone charakteryzuje ich nazwa (Atom) oraz liczba argumentów.
- Czasem można spotkać nazwę struktury zamiast termy złożone.
- Ten sam atom może raz wystąpić w miejscu nazwy predykatu i w miejscu symbolu funkcyjnego, np.

```
predfun(ala,predfun(ala,ala)).
```

- Predykaty i symbole funkcyjne to atomy.
- Atomy specjalne, m. in.,
  - ':-' – operator odpowiadający implikacji,
  - '?-' – „symbol zachęty",
  - ',' – przecinek.
- Wyrażenia zbudowane z tych atomów to także termy złożone.
  - ```
    [debug]  ?- X=':-'(ble(ala),ble(ala)).
    X =  (ble(ala):-ble(ala)).
    ```
  - ```
    [debug]  ?- ':-'(ble(ala),  ','(ble(ala), ble(ala))) = X.
    X =  (ble(ala):-ble(ala), ble(ala)).
    ```
  - Podobnie '=' to atom czyli

    ```
    [debug]  ?- '='(X,':-'(f(zero), ','(f(jeden), f(dwa)))).
    X =  (f(zero):-f(jeden), f(dwa)).
    ```

Równość

- $t = s$ oznacza próbę unifikacji termów t i s.
- Niestety, w celu polepszenia efektywności, Prolog nie sprawdza, czy unifikując X z t, term t nie zawiera wystąpień X.
- Np. Prolog unifikuje X i $f(X)$,

```
[debug]  ?- X=f(X).
X = f(X).
```

- To podejście odróżnia unifikację w Prologu od logiki.
- Aby uniknąć takiej sytuacji można użyć

```
[debug]  ?- unify_with_occurs_check(X,f(X)).
false.
```

- Zaleca się jednak pisanie programów w sposób, który wykluczy możliwość unifikacji X z $f(X)$.

Klauzule

- Fakty w bazie wiedzy Prologu wyrażamy za pomocą klauzul.
- Klauzula ma postać: Cel (głowa) :- Ciało.
- Cel to predykat (term).
- Ciało to predykaty pogrupowane przy pomocy:
  - ’,’ – koniunkcji,
  - ’;’ – alternatywy,
  - ’->’ – implikacji.
  - ’not ’ – negacją,
  - inne.
- Ciało może być puste. Wtedy `Głowa.` odpowiada `Głowa :- true.`
- Na początek zajmiemy się tylko koniunkcją, żeby zachować bliski związek z formułami Herbrandowskimi.

Klauzule

- Klauzule w bazie wiedzy reprezentują zależności logiczne pomiędzy predykatami.
- Należy jednak pamiętać, że predykaty i symbole funkcyjne w sensie logicznym w Prologu reprezentowane są przez termy.
- Ten sam term może wystąpić w obu rolach.

Przykład

- Niech baza wiedzy zawiera

```
predfun(ola,ola).
predfun(ala,predfun(ola,ola)):-predfun(ola,ola).
```

- Wtedy zapytanie `?-predfun(X,Y)` otrzyma dwie odpowiedzi:

```
X = Y, Y = ola ;
X = ala, Y = predfun(ola, ola).
```

- `predfun` wystąpił jako symbol funkcyjny i jako predykat.

Zapytania

- Do bazy wiedzy można kierować zapytania.
- Zapytanie ma (oczywiście) postać termu.
- Prolog stara się wynioskować term opierając się na wczytanej bazy wiedzy.
- Zapytanie możemy zadać w konsoli w postaci

  ```
  ?- zapytanie.
  ```

  albo możemy wpisać je do bazy wiedzy w postaci

  ```
  :-zapytanie.
  ```

- W tym drugim przypadku Prolog podczas wczytywania bazy stara się również wywnioskować dany term.

Baza wiedzy

- Klauzule z bazy wiedzy zapisujemy do pliku z rozszerzeniem .pl
- Plik ten wczytujemu instrukcją `consult('plik.pl').`
- Po zmianie pliku instrukcja `make.` wczyta zmienione pliki.
- Możemy użyć też `consult`.
- Podczas pracy z interpreterem możemy dodać klauzule do bazy wiedzy (`assert, asserta, assertz`) lub usunąć (`retract`).

Hello world!

- `print(t)` wypisuje term t.
- ```
  ?- write("Hello world!").
  Hello world!
  true.
  ```
- write(tekst) możemy umieścić też w bazie wiedzy. Aby print zostało wykonane podczas wczytywania pliku, trzeba wpisać je jako cel do uzgodnienia (wynioskowania):

```
:-write(tekst).
```
```

Outline

Rezolucja w Prologu

- Niech teoria T reprezentuje bazę wiedzy.
- Niech $\varphi_1(x, y, z), \varphi_2(x, y, z)$ to koniunkcja formuł atomowych będąca zapytaniem do Prologu.
- Jest to równoważne pytaniu czy

$$T \vdash \exists x \exists y \exists z (\varphi_1(x, y, z) \wedge \varphi_2(x, y, z)).$$

- Dowód rezolucyjny jest dowodem nie wprost więc próbujemy uzyskać sprzeczność z T i $\neg \exists x \exists y \exists z (\varphi_1(x, y, z) \wedge \varphi_2(x, y, z))$.
- Jest to równoważne dodaniu do T formuły $\forall x, y, z (\neg \varphi_1 \vee \neg \varphi_2)$.
- Prolog stara się wykonać SLD-rezolucje z T i jednej negatywnej klauzuli $\{\neg \varphi_1(x, y, z), \neg \varphi_2(x, y, z)\}$.

Rezolucja w Prologu

- Prolog stara się udowodnić przy pomocy rezolucji pytanie zadane do bazy wiedzy.
- Co więcej podczas wywodu rezolucyjnego konstruuje podstawienie, które wskaże świadków dla zmiennych występujących w pytaniu.
- Aby kontrolować Prolog trzeba znać sposób konstrukcji wywodu rezolucyjnego.
- Na kolejnym slajdzie przedstawię algorytm (w wersji rekurencyjnej) SLD rezolucji realizowanej w Prologu.

Rezolucja w Prologu – algorytm rekurencyjny

- Wejście: ciąg formuł atomowych $\varphi_1(\bar{x}), \ldots, \varphi_k(\bar{x})$ i podstawienie θ.
- Cel: wykazać sprzeczność klauzuli $\{\neg\varphi_1(\bar{x}), \ldots, \neg\varphi_k(\bar{x})\}\theta$ z bazą wiedzy i podać wartościowanie, które konstruuje świadków.
  1. Jeśli ciąg formuł jest pusty to zwróć (**sukces**, θ).
  2. Weź pierwszą formułę z ciągu $\varphi_1(\bar{x})\theta$ i postaraj się się ją zunifikować przy pomocy MGU z celem formuły z T (po urozłącznieniu zmiennych tej formuły z $\varphi_1\theta$).
     Formuły z T przeglądaj zgodnie z ich kolejnością w bazie wiedzy.
  3. Zaznacz miejsce p w T gdzie znalazłeś formułę $\psi : -\overline{\psi}.$ taką, że dla MGU τ, $\varphi_1\theta\tau == \psi\tau$.
  4. Wywołaj się rekurencyjnie dla $\overline{\psi}, \varphi_2, \ldots, \varphi_k$ i podstawienia $\theta \circ \tau$.
  5. Jeśli wywołanie zwróci (**sukces,** τ) to zwróć (**sukces**,τ).
  6. Jeśli wywołanie zwróci **porażka** to szukaj innej unifikacji $\varphi_1\theta$ od miejsca p.
  7. Jeśli nie udało się zuunifikować $\varphi_1\theta$ to zwróć **porażka**.

Rezolucja w Prologu – własności algorytmu 1

- Przeglądamy formuły w kolejności ich występowania w bazie wiedzy.
- Kolejność formuł ma znaczenie, może spowodować wystąpienie nieskończonej pętli podczas konstruowania dowodu.
- Należy wpisywać najpierw przypadki „bazowe”, które są prawdziwe bezwarunkowo, mają postać: $\mathtt{p}(\mathtt{t}_1, \ldots, \mathtt{t}_k)$.

Rezolucja w Prologu – własności algorytmu 2

- Prolog unifikuje używając MGU.
- Wynika z tego, że nie wystarczy tylko raz próbować unifikacji z daną formułą w bazie wiedzy.
- Jeśli wywód przy pomocy MGU nie zakończy się sukcesem, to żaden inny unifikator nie osiągnie sukcesu (z własności MGU).
- Dlatego Prolog może przejść do szukania następnej formuły z bazy wiedzy w algorytmie.

Rezolucja w Prologu – własności algorytmu 3

- Po zwróceniu sukcesu Prolog zwraca też podstawienie, za zmienne z zapytania, dla którego może udowodnić zapytanie.
- Podstawienie to jest złożeniem podstawień, które Prolog wykonał podczas wywodu pustej klauzuli.
- Możemy kazać Prologowi wyszukać kolejne rozwiązanie (spacja, średnik).
- Prolog szuka wtedy kolejnej formuły w bazie wiedzy, z którą może zunifikować pierwszą formułę zapytania.

Kolejność klauzul ma znaczenie – przykład

- Rozważmy bazę wiedzy postaci

```
p(ala).
p(X):-p(f(X)).
```

- Zapytanie `?- p(X).` zwróci odpowiedź `X=ala`.
- Kolejne próby uzgodnienia zakończą się nieskończoną pętlą i próbami udowodnienia `p(f(X)), p(f(f(X))), ...`.
- Gdyby zmienić kolejność klauzul Prolog od razu wpadnie w nieskończoną pętlę i nie znajdzie odpowiedzi.

Definicje nowych predykatów

- Prolog pozwala na definicje predykatów.
- Definicja charakteryzuje predykat poprzez klauzule, które go opisują w bazie wiedzy.
- Przy pomocy tych klauzul Prolog stara się udowonić predykat.
- Definiując klauzulą predykat p(...) można użyć tego predykatu w ciele klauzuli.
- Trzeba dbać o to, żeby definicja taka nie spowodowała nieskończonej pętli.

Definicje nowych predykatów – przykład

- Niech `edge(X,Y)` będzie relacją bycia krawędzią w drzewie (grafie bez cykli).
- Chcemy zdefiniować tranzytywne domknięcie relacji `edge`.
- Niech

```
edge(a,b).
edge(b,c).
edge(c,d).
```

Definicje nowych predykatów – przykład

- ```
  ts1_edge(X,Y):-edge(X,Y).
  ts1_edge(X,Y):-edge(X,Z), ts1_edge(Z,Y).

  ts2_edge(X,Y):-edge(X,Y).
  ts2_edge(X,Y):-ts2_edge(X,Z), ts2_edge(Z,Y).

  ts3_edge(X,Y):-edge(X,Z), ts3_edge(Z,Y).
  ts3_edge(X,Y):-edge(X,Y).

  ts3_edge(X,Y):-ts4_edge(X,Z), ts4_edge(Z,Y).
  ts4_edge(X,Y):-edge(X,Y).
  ```
- Która wersja jest lepsza?
  ```

Definicje nowych predykatów – przykład

- ```
  ts1_edge(X,Y):-edge(X,Y).
  ts1_edge(X,Y):-edge(X,Z), ts1_edge(Z,Y).

  ts2_edge(X,Y):-edge(X,Y).
  ts2_edge(X,Y):-ts2_edge(X,Z), ts2_edge(Z,Y).
  ```
- Kiedy `t` należy do tranzytywnego domknięcia `s`, to ts1_edge i ts2_edge zachowają się podobnie.
- Różnica pojawia się gdy atomy nie są ze sobą w relacji:
  - `:-ts1_edge(a,a).` odnosi porażkę.
  - `:-ts2_edge(a,a).` wpada w nieszkończoną pętlę.
  ```

Definicje nowych predykatów – przesłanianie nazw

- Definiując nowe predykaty możemy przysłonić nazwy predykatów z biblioteki Prologu.
- Dodajmy do bazy wiedzy `print('Ala ma kota').`
- Teraz `?- print(X).` uzgodni X ze stałą 'Ala ma kota' ale nic nie wydrukuje.

Równość czyli unifikacja

- W Prologu nie ma instrukcji przypisania.
- Równość, '=', ma w Prologu specjalne znaczenie.
- Uzgodnienie termu 't = s' powoduje unifikację termów.
- Oznacza to, że aktualne podstawienie Prologu zostaje rozszerzone o podstawienie θ takiego, że $t\theta$ i $s\theta$ to takie same termy.

Podwójna równość, ’==’, czyli równość termów

- Term ’t==s’ odnosi sukces wtedy, gdy termy s i t są równe przy aktualnie skonstruowanym podstawieniu.
- Uzgodnienie ’t=s’ rozszerza podstawienie.
- W przeciwieństwie do tego, ’==’ nie powoduje zmiany podstawienia.

’=’ i ’==’ – przykład

- `:- X==f(Y).` kończy się niepowodzeniem.
- `:- X=f(Y).` odnosi sukces.
- `:- X=f(Y), X==f(Y).` odnosi sukces.

’=’ i ’==’ – skok do negacji

- `not(term)` odnosi sukces kiedy próba wykazania `term` się nie powiedzie (negacja w Prologu).
- `:- not(X==f(Y)).` odnosi sukces.
- `:- not(X=f(Y)).` odnosi porażkę.
- `:- not(not(X=f(Y))).` odnosi sukces.
- `:- not(not(X=f(Y))), X==f(Y).` odnosi porażkę (dlaczego?).

Outline

Kontrola wykonywania rezolucji

- Podczas wykonywania programu Prolog może wpaść w nieskończoną pętlę.
- Nawet jeśli Prolog nie wpadnie w nieskończoną pętlę często może wykonywać niepotrzebnie obliczenia, które nie mogą zakończyć się sukcesem.
- Prolog może niepotrzebnie szukać rozwiązania, o którym można z góry powiedzieć, że nie istnieje.
- Aby zoptymalizować obliczenia Prologu, uniknąć nieskończonych pętli, niezbędna jest kontrola nad wykonywanym wnioskowaniem.

- Kolejność wyboru klauzul do unifikacji – zgodnie z opisem algorytmu.
- Nawracanie – zgodnie z opisem algorytmu.
- fail – wymuszona porażka lub wymuszenie nawrotu.
- cut (!) – odcięcie.

fail

- fail – ten cen nigdy nie kończy się sukcesem.
- Można go użyć do wymuszenia nawracania.
- Można go użyć do stworzenia pętli.

Pętla może przebiegać po termach

- ```
  osoba(ala).
  osoba(ola).
  osoba(alek).
  osoba(olek).

  loop:- osoba(X), write(X),nl, fail.
  loop:-true.
  ```
- ```
  ?- loop.
  ala
  ola
  alek
  olek
  true.
  ```

- Wbudowane konstrukcje pozwalające na wykonanie pętli numerycznej:
  - `between(Low,High,Value)`
- Przykład.

```
?- between(3,5,X), write(X),nl,fail.
3
4
5
false.
```

- Później zobaczymy jak skonstruować pętlę ze zmienną numeryczną bez between; lub jak zdefiniować `between`.

Odcięcie

- Odcięcie, !, zawsze odnosi sukces, ale podczas uzgadniania celu odnosi sukces tylko raz i Prolog już nie próbuje uzgodnić inaczej celów, które wystąpiły przed odcięciem.
- `cel(X):- a(X),b(X),!,c(X),d(X).`
- Co więcej, po dotarciu do odcięcia, Prolog nie będzie próbował uzgodnić `cel(X)` korzystając z innych reguł zawierających `cel(X)` jako głowę reguły.
- Gdyby Prolog nie doszedł do odcięcia, mógłby skorzystać z innych reguł dla uzgodnienia `cel(X)`.

Odcięcie – dlaczego używamy

- Mówi Prologowi, że wybrał już właściwą (może jedyną) regułę do uzgodnienia celu.
  Podstawienie obliczone do tego momentu przez Prolog nie będzie zmieniane. Korzystamy z tego, gdy nie chcemy, żeby Prolog próbował uzgadniać cel w inny sposób, nawet jesli by mu się to udało.
- Mówi Prologowi, że nie uda się już uzgodnić celu w inny sposób.
- Może wymusić natychmiastowe zakończenie prób uzgodnienia celu w konfigurancji `cel :- a, b, !, fail.`

Odcięcie – przykład

- ```
  osoba(ala).
  osoba(ola).
  osoba(alek).
  osoba(olek).
  zwierze(ara).
  zwierze(pola).

  odciecie(X,Y):-osoba(X), X=Y.
  odciecie(X,Y):-osoba(X),!, zwierze(Y).
  odciecie(X,Y):-zwierze(X),osoba(Y).
  ```
- `?-odciecie(X,Y).` uzgodni wszystkie pary z pierwszej klauzuli, uzgodni pary `ala, ara` oraz `ala, pola`.
- Proszę zwrócić uwagę na kolejność wypisywania stałych.
  ```

Odcięcie – przykład

- ```
  osoba(ala).
  osoba(ola).
  osoba(alek).
  osoba(olek).
  zwierze(ara).
  zwierze(pola).
  odciecie(X,Y):-osoba(X), X=Y.
  odciecie(X,Y):-osoba(X),!, zwierze(Y).
  odciecie(X,Y):-zwierze(X),osoba(Y).
  celnadrzedny(X,Y):-odciecie(X,Y).
  celnadrzedny(X,Y):-zwierze(X),X=Y.
  ```
- `?-celnadrzedny(X,Y).`
  uzgodni wszystkie pary `(ala,ala), ...,(olek,olek),`
  uzgodni `(ala,ara), (ala,pola)`
  oraz pary `(ara,ara), (pola,pola).`
  ```

Odcięcie – przykład

- suma(N,X) – X jest sumą liczb od 1 do N
- ```
  suma(1,1) :- !.
  suma(N,X) :- N1 is N - 1, suma(N1,Y), X is N + Y.
  ```
- Bez odcięcia Prolog mógłby próbować uzgodnić `suma(1,X)` inaczej niż przy pomocy pierwszej klauzuli.
  ```

Odcięcie – przykład

- ```
  suma(1,1) :- !.
  suma(N,X) :- N1 is N - 1, suma(N1,Y), X is N + Y.

  suma_bez_odciecia(1,1).
  suma_bez_odciecia(N,X) :- N1 is N-1,
                            suma_bez_odciecia(N1,Y),
                            X is N+Y.
  ```
- `?- suma(1,X), fail.` kończy się porażką.
- `?-suma_bez_odciecia(1,X), fail.` wpada w nieskończoną pętlę.
  ```

Odcięcie – przykład

- Odcięcie zmienia ilość dostępnych wyborów.
- ```
  ilu_ro(adam,N):- !, N=0.
  ilu_ro(ewa,N):- !, N=0.
  ilu_ro(X,2).

  ?- ilu_ro(X,Y).
  X = adam,
  Y = 0.
  ```

  Odcięcie sprawia, że Prolog nie znajduje innych podstawień.
- ```
  ?- ilu_ro(X,2).
  false.
  ```

  Odcięcie sprawia, że po próbie uzgodnienia z pierwszą klauzulą Prolog nie próbuje uzgodnień z innymi klauzulami.

Odcięcie a predykat ’not’

- Predykat ’not(P)’ (\+ P) odnosi sukces, gdy uzgodnienie P się nie powiedzie.
- Pozwala to na zdefiniowanie odcięcia.
- Zamiast

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

  możemy napisać

```
cel :- a, b.
cel :- \+ a, c.
```

- Druga wersja jest „bardziej logiczna” ale w pierwszej klauzuli `a` może być uzgadniane wiele razy.
- W drugiej wersji cel `a` jest uzgadniany również w drugiej klauzuli, co zmniejsza efektywność.
-
```
a(X):- ground(X), zwierze(X).
a(X):- \+ground(X), osoba(X).
```

Powtórzyc poprzedni przyklad z omowieniem tego, jak Prolog wybiera termy i dla jakich X a(X) sie powiedzie.

Outline

- Wczytywanie bazy wiedzy `consult(plik).`
- Śledzenie (trace, gtrace,notrace).
- Ustawianie punktów śledzenia (spy).
- Przerywanie wykonywania programu: Ctr+C.

Outline

Listy

- Pusta lista to [].
- Listę dzielimy na głowę i ogon i zapisujemy $[Glowa|Ogon]$.
- Listę możemy też zapisać $[a, b, c|Ogon]$ lub $[a, b, c] = [a, b, c|[]]$.

- Lista jest strukturą dynamiczną.
- Predykaty pisane dla list muszą uwzględniać rekurencyjną strukturę listy.
- Sprawdzenie czy coś jest elementem listy

```
member(X,[X|_]).
member(X,[_|O]):-member(X,O).
```

- Prawidłowe użycia

```
:-member(a,[c,b,a]).
:-member(X,[c,b,a]).
:-member(a,X).
```

- Co by się zmieniło w `:-member(a,X).` gdyby zastosować odcięcie w pierwszej klauzuli definicji czyli
  - `:-memeber(X,[X|_]):-!.`

Struktury i struktury dynamiczne

- Struktura może być reprezentowana termem.
- Np. `moja_struktura('Adam','Abacki',1.82,92)` reprezentuje Adama Abackiego o wzroście 1.82m i wadze 92kg.
- Drzewa w prologu reprezentujemy przez termy zagnieżdżone.
  - Węzeł drzewa to `node(Wartosc,LewePodrzewo,PrawePodrzewo).`
  - Przykład drzewa `node(2,node(1,null,null),node(3,null,null)).`
  - W jednym drzewie możemy przechowywać wartości „różnych typów”, np. `node(2,node(1.3,null,null),node(procesor,null,null)).`

Drzewo BST

```
insertBSTaux(N,null,treeBST(N,null,null)):- !.
insertBSTaux(N,treeBST(N,TLeft,TRight),
                treeBST(N,TLeft,TRight)):-!.
insertBSTaux(N,treeBST(M,TLeft1,TRight1),
                treeBST(M,TLeft2,TRight2)):-
                    N<M,
                    TRight1=TRight2,
                    insertBSTaux(N,TLeft1,TLeft2).
insertBSTaux(N,treeBST(M,TLeft1,TRight1),
                treeBST(M,TLeft2,TRight2)):-
                    N>M,
                    TLeft1=TLeft2,
                    insertBSTaux(N,TRight1,TRight2).

insertBST(N,T1,T2):- number(N), insertBSTaux(N,T1,T2).
```

Drzewo BST

```
?- insertBST(4,null,T1), insertBST(5,T1,T2),
   insertBST(1,T2,T3), insertBST(8,T3,T4).
T1 = treeBST(4, null, null),
T2 = treeBST(4, null, treeBST(5, null, null)),
T3 = treeBST(4, treeBST(1, null, null),
                treeBST(5, null, null)),
T4 = treeBST(4, treeBST(1, null, null),
                treeBST(5, null, treeBST(8, null, null)))
```

Koniec