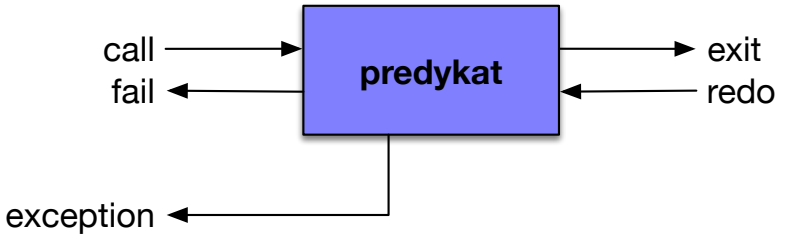
Programowanie w Logice

Śledzenie programu

Przemysław Kobylański

Śledzenie programu

Model pudełkowy



Śledzenie programu

Model pudełkowy

- wyjątek ma postać dowolnego termu
- do zgłaszania wyjątku służy predykat `throw(Exception)`
- z celem `Goal` można związać obsługę `Handler` wyjątku pasującego do wzorca `Pattern` za pomocą meta-predykatu `catch(Goal, Pattern, Handler)`

Śledzenie programu

Model pudełkowy

Example (Bezpieczne obliczanie wartości wyrażeń)

Wartością zeroargumentowej funkcji `nan` jest `NaN`.

```
eval(Expression, Value) :-
    catch(Value is Expression, _, Value is nan).

?- eval(2+2, X).
X = 4.

?- eval(sqrt(-1), X).
X = 1.5NaN.

?- eval(1/0, X).
X = 1.5NaN.

?- eval(sqrt(-1), X), eval(X+1, Y).
X = Y, Y = 1.5NaN.
```

Śledzenie programu

Model pudełkowy

- meta-predykaty catch/3 można oczywiście zagnieżdżać: `catch(catch(catch(..., ..., ...), ..., ...), ..., ...)`

Example (Zagnieżdżona obsługa wyjątków)

```
catch(catch(catch(Goal,
                  Pattern_1,
                  Handler_1),
            Pattern_2,
            Handler_2),
      Pattern_3,
      Handler_3)
```

Śledzenie programu

Model pudełkowy

Example (Zagadka)

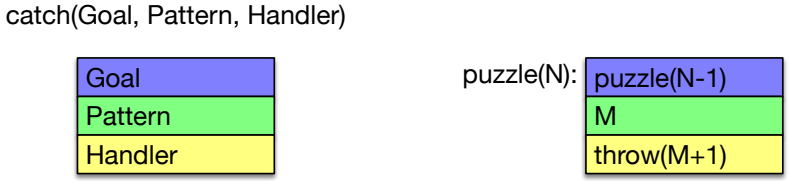
Co jest efektem wywołania `puzzle(N)`, dla całkowitego $N \geq 0$?

```
puzzle(0) :-
    throw(0).
puzzle(N) :-
    catch((N1 is N-1, puzzle(N1)),
          M,
          (M1 is M+1, throw(M1))).
```

Śledzenie programu

Model pudełkowy

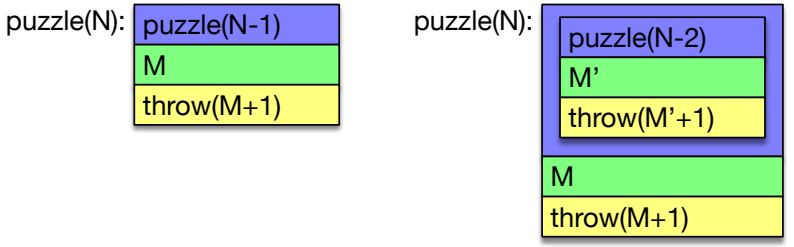
Example (Rozwiązanie zagadki)



Śledzenie programu

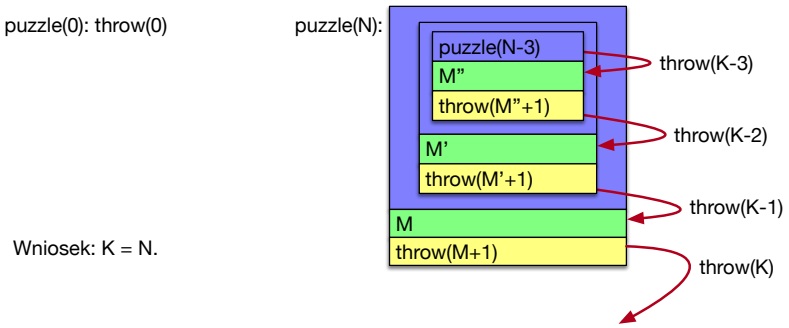
Model pudełkowy

Example (Rozwiązanie zagadki cd.)



Śledzenie programu
Model pudełkowy

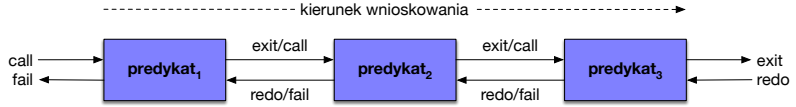
Example (Rozwiązanie zagadki cd.)



Odpowiedź: wywołanie `puzzle(N)` jest równoważne `throw(N)`.

Śledzenie programu
Model pudełkowy

Do końca kursu nie będziemy korzystać w programach z wyjątków, zatem można przyjąć, że w modelu pudełkowym mamy tylko cztery porty: call, exit, redo i fail.



Śledzenie programu
Predykaty trace i debug

Dwa tryby śledzenia działania programu:

trace w tym trybie drukowany jest ślad przejścia przez dany port i system czeka na wprowadzenie przez użytkownika komendy co dalej robić

debug w tym trybie drukowany jest ślad przejścia przez dany port ale system nie czeka na komendę (automatycznie kontynuuje)

- predykat `trace/0` włącza tryb śledzenia
- predykat `notrace/0` wyłącza tryb śledzenia
- predykat `debug/0` włącza tryb debuggera
- predykat `nodebug/0` wyłącza tryb debuggera

Śledzenie programu
Predykaty trace i debug

Dostępne komendy w trybie śledzenia:

```
+:                    spy          -:              no spy
/c|e|r|f|u|a goal:    find         .:              repeat find
a:                    abort        A:              alternatives
b:                    break        c (ret, space): creep
[depth] d:            depth        e:              exit
f:                    fail         [ndepth] g:     goals (backtrace)
h (?):                help         i:              ignore
l:                    leap         L:              listing
n:                    no debug     p:              print
r:                    retry        s:              skip
u:                    up           w:              write
m:                    exception details
C:                    toggle show context
```

Śledzenie programu

Predykaty trace i debug

- predykat `trace(Pred)` włącza tryb debugowania wszystkich portów danego predykatu
- predykat `trace(Pred, Ports)` włącza tryb debugowania wybranych portów danego predykatu

Śledzenie programu

Predykaty trace i debug

Example (Predykat trace/0)

```
p(a). p(b). q(X) :- p(X).

?- trace.
true.

[trace]  ?- q(X).
   Call: (8) q(_7652) ? creep          % naciśnięto Enter
   Call: (9) p(_7652) ? creep
   Exit: (9) p(a) ? creep
   Exit: (8) q(a) ? creep
X = a ;
   Redo: (9) p(_7652) ? creep
   Exit: (9) p(b) ? creep
   Exit: (8) q(b) ? creep
X = b.
```

Śledzenie programu

Predykaty trace i debug

Example (Predykat trace/1)

```
p(a). p(b). q(X) :- p(X).

?- trace(p).
%         p/1: [call,redo,exit,fail]
true.

[debug]  ?- q(X).
 T Call: (9) p(_7524)
 T Exit: (9) p(a)
X = a ;
 T Redo: (9) p(_7524)
 T Exit: (9) p(b)
X = b.
```

Śledzenie programu

Predykaty trace i debug

Example (Predykat trace/2)

```
p(a). p(b). q(X) :- p(X).

?- trace(p, [call]).
%         p/1: [call]
true.

[debug]  ?- q(X).
 T Call: (9) p(_7524)
X = a ;
X = b.
```

Śledzenie programu

Predykaty trace i debug

Example (Generowanie permutacji)

Rozpatrzmy następujący predykat perm/2:

```
perm([], []).
perm([A | B], C) :-
    select(A, C, D),
    perm(B, D).
```

Druga klauzula mówi aby:

- rozłożyć daną listę na głowę A i ogon B
- wstawić A do wynikowej permutacji C między elementy listy D
- niech D będzie permutacją ogona B

Śledzenie programu

Predykaty trace i debug

Example (Generowanie permutacji cd.)

Zobaczmy jakie są permutacje listy [1,2,3]:

```
?- perm([1, 2, 3], X).
X = [1, 2, 3] ;
Could not reenable global-stack
...
ERROR: Out of global-stack.
ERROR: No room for exception term.  Aborting.
...
% Execution Aborted
?-
```

Dlaczego predykat nie zadziałał zgodnie z naszymi oczekiwaniami?

Śledzenie programu

Predykaty trace i debug

Example (Generowanie permutacji cd.)

Prześledźmy działanie programu:

```
?- trace.
[trace]  ?- perm([1, 2, 3], X).
   Call: (8) perm([1, 2, 3], _7892) ? creep
   Call: (9) lists:select(1, _7892, _8132) ? creep
   Exit: (9) lists:select(1, [1|_8116], _8116) ? creep
   Call: (9) perm([2, 3], _8116) ? creep
   Call: (10) lists:select(2, _8116, _8138) ? creep
   Exit: (10) lists:select(2, [2|_8122], _8122) ? creep
   Call: (10) perm([3], _8122) ? creep
   Call: (11) lists:select(3, _8122, _8144) ? creep
   Exit: (11) lists:select(3, [3|_8128], _8128) ? creep
   Call: (11) perm([], _8128) ? creep
   Exit: (11) perm([], []) ? creep
   Exit: (10) perm([3], [3]) ? creep
   Exit: (9) perm([2, 3], [2, 3]) ? creep
   Exit: (8) perm([1, 2, 3], [1, 2, 3]) ? creep
X = [1, 2, 3] ;
```

Na razie wszystko zgodnie z planem.

Śledzenie programu

Predykaty trace i debug

Example (Generowanie permutacji cd.)

Kontynuujemy śledzenie:

```
Redo: (11) lists:select(3, [_8126|_8128], _8150) ? creep
Exit: (11) lists:select(3, [_8126, 3|_8134], [_8126|_8134]) ? creep
%  wstawienie na drugą pozycję ___/
Call: (11) perm([], [_8126|_8134]) ? creep
Fail: (11) perm([], [_8126|_8134]) ? creep
Redo: (11) lists:select(3, [_8126, _8132|_8134], [_8126|_8140]) ? creep
Exit: (11) lists:select(3, [_8126, _8132, 3|_8146], [_8126, _8132|_8146]) ? cre
%       wstawienie na trzecią pozycję ___/
Call: (11) perm([], [_8126, _8132|_8146]) ? creep
Fail: (11) perm([], [_8126, _8132|_8146]) ? creep
Redo: (11) lists:select(3, [_8126, _8132, _8144|_8146], [_8126, _8132|_8152]) ?
Exit: (11) lists:select(3, [_8126, _8132, _8144, 3|_8158], [_8126, _8132, _8144
%              wstawienie na czwartą pozycję ___/
Call: (11) perm([], [_8126, _8132, _8144|_8158]) ? creep
Fail: (11) perm([], [_8126, _8132, _8144|_8158]) ? creep
Redo: (11) lists:select(3, [_8126, _8132, _8144, _8156|_8158], [_8126, _8132, _
Exit: (11) lists:select(3, [_8126, _8132, _8144, _8156, 3|_8170], [_8126, _8132
%                      wstawienie na piątą pozycję ___/
```

Znaleźliśmy przyczynę problemu.

Śledzenie programu

Predykaty trace i debug

Example (Generowanie permutacji cd.)

Q: Jak naprawić predykat `perm/2`?
A: Wyznaczyć permutację D ogona B zanim będziemy na nią wstawiać głowę A.

```
perm([], []).
perm([A | B], C) :-
    perm(B, D),      %  zamiana kolejności
    select(A, C, D). %  dwóch warunków
```

Powyższa wersja jest poprawna:

```
?- perm([1, 2, 3], X).
X = [1, 2, 3] ;
X = [2, 1, 3] ;
X = [2, 3, 1] ;
X = [1, 3, 2] ;
X = [3, 1, 2] ;
X = [3, 2, 1] ;
false.
```

Śledzenie programu

Przepływ danych, punkt wyboru i niedeterminizm

- argument predykatu może wystąpić w jednym z poniższych trybów:
  - `+` argument powinien być ukonkretniony
  - `?` argument powinien być ukonkretniony lub być zmienną
  - `-` argument powinien być zmienną, która ukonkretni się gdy cel zakończy się sukcesem
- przykłady trybów wywołania:

```
append(+, +, +)   append([1,2], [3], [1,2,3])
append(+, +, -)   append([1,2], [3], X)
append(+, -, +)   append([1,2], X, [1,2,3])
append(-, -, +)   append(X, Y, [1,2,3])
```

Śledzenie programu

Przepływ danych, punkt wyboru i niedeterminizm

- jeśli pozostawiono możliwość nawrotu do danego punktu i wybranie innej ścieżki wnioskowania, to punkt ten nazywamy **punktem wyboru** (ang. *choice point*)

Możliwe tryby determinizmu:

- `det` gdy warunek jest spełniony jednokrotnie bez pozostawienia punktu wyboru
- `semidet` gdy warunek jest spełniony jednokrotnie bez pozostawienia punktu wyboru lub zawodzi
- `failure` gdy warunek zawsze zawodzi
- `nondet` bez ograniczeń na liczbę spełnień
- `multi` jak `nondet` ale spełniony co najmniej raz

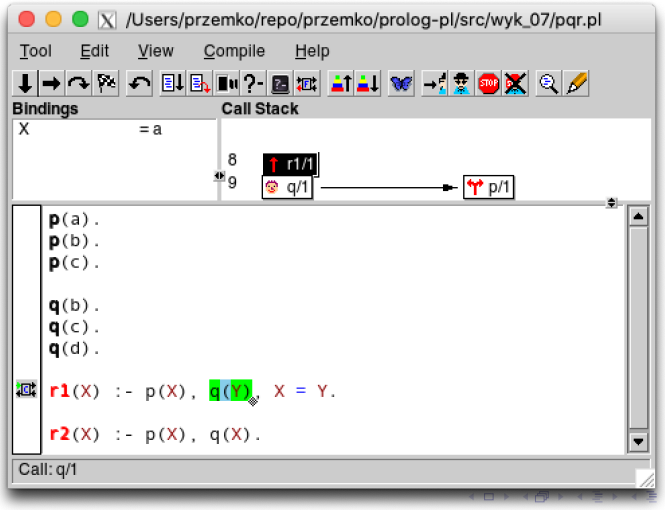
Przykłady trybów determinizmu:

det	`append(+,+,-)`	`append([1,2], [3], X)`
semidet	`append(+,-,+)`	`append([1], X, [1,2,3])`
multi	`append(-,-,+)`	`append(X, Y, [1,2,3])`
nondet	`append(?,?,+)`	

Śledzenie programu

Predykat gtrace

- W systemie SWI-Prolog dostępny jest graficzny debugger.
- Aby z niego skorzystać należy wywołać predykat `gtrace/0`.



Śledzenie programu

Kolejność warunków

Example (Skończone ciągi bitów)

Chcemy generować wszystkie skończone ciągi bitów 0/1.
Zaczniemy od zdefiniowania jednego bitu:

```
bit(0).
bit(1).
```

Warunek `bit(X)` w trybie `bit(-)` jest niedeterministyczny i generuje oba możliwe przypadki `X=0` i `X=1`.
Zdefiniujmy predykat wyrażający, że jego argument jest skończoną listą złożoną z bitów:

```
bits1([]).
bits1([X | Y]) :- bit(X), bits1(Y).
```

Śledzenie programu

Kolejność warunków

Example (Skończone ciągi bitów cd.)

Oczekujemy, że na pytanie `bits1(X)` otrzymamy w kolejnych odpowiedziach wszystkie skończone ciągi bitów.

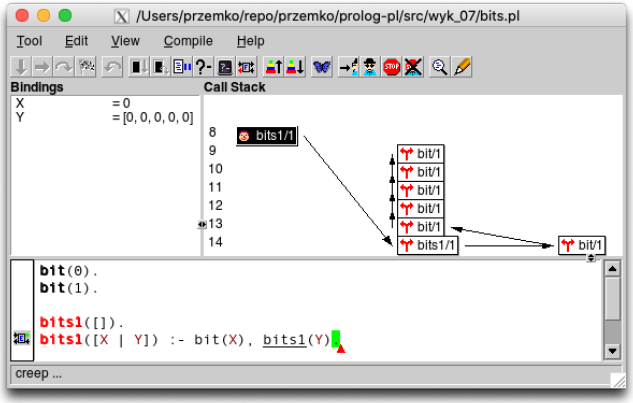
```
?- bits1(X).
X = [] ;
X = [0] ;
X = [0, 0] ;
X = [0, 0, 0] ;
X = [0, 0, 0, 0] ;
X = [0, 0, 0, 0, 0] ;
X = [0, 0, 0, 0, 0, 0] ;
...
```

Śledzenie programu

Kolejność warunków

Example (Skończone ciągi bitów cd.)

Zajrzyjmy do debuggera.



Przy każdej odpowiedzi ciągle przybywa punktów wyboru.

Śledzenie programu

Kolejność warunków

Example (Skończone ciągi bitów cd.)

Zastanówmy się nad ciałem drugiej klauzuli definiującej predykat `bits1/1`:

```
bits1([X | Y]) :- bit(X), bits1(Y).
```

Zanim wygeneruje się druga wartość `X` spełniająca warunek `bit(X)` muszą wcześniej wygenerować się wszystkie wartości `Y` spełniające warunek `bits1(Y)` a tych jest nieskończenie wiele.
W drugiej wersji zmienimy kolejność tych warunków:

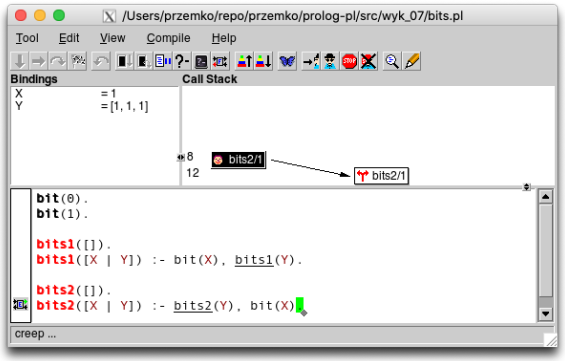
```
bits2([]).
bits2([X | Y]) :- bits2(Y), bit(X).
```

Śledzenie programu

Kolejność warunków

Example (Skończone ciągi bitów cd.)

Znowu zajrzyjmy do debuggera.



Tym razem pozostaje tyle punktów wyboru dla `bit/1` ile jest zer w odpowiedzi i jeszcze jeden punkt wyboru dla `bits2/1`.

Śledzenie programu

Kolejność warunków

Example (Skończone ciągi bitów cd.)

Początkowe odpowiedzi na pytanie `bits2(X)`:

```
?- bits2(X).
X = [] ;
X = [0] ;
X = [1] ;
X = [0, 0] ;
X = [1, 0] ;
X = [0, 1] ;
X = [1, 1] ;
X = [0, 0, 0] ;
X = [1, 0, 0] ;
X = [0, 1, 0] ;
X = [1, 1, 0] ;
X = [0, 0, 1] ;
X = [1, 0, 1] ;
X = [0, 1, 1] ;
X = [1, 1, 1] ;
X = [0, 0, 0, 0] ;
...
```