Podsumowanie (16 czerwca, wykład 15)

12/06/20 10:23

Na zakończenie wykładu z **Programowania w Logice** chciałbym dokonać krótkiego podsumowania.

1. Omówiony został jeden z ciekawszych języków programowania, który wywodzi się bezpośrednio z logiki, a dokładniej z **rachunku predykatów pierwszego rzędu** [1].
2. **Prolog** jest przykładem języka deklaratywnego, w którym opisuje się problem bez podawania algorytmu jego rozwiązania [2].
3. Dostępne implementacje Prologu dostarczają nie tylko maszynę abstrakcyjną i kompilator na tę maszynę ale również liczne moduły ułatwiające praktyczne zastosowania tego języka (grafika, dostęp do baz danych, serwer/klient HTTP, parser SGML/XML, programowanie ograniczeń, przetwarzanie języka naturalnego, sieci semantyczne, obsługa SSL, …).
4. Szczególnie interesujące są moduły **CLPFD**, **CLPB**, **CLPQ**, **CLPR** i **CHR**, które dostarczają **programowanie ograniczeń**, w istotny sposób przyspieszające znajdowanie rozwiązań.
5. Programowanie ograniczeń dostarczane jest w postaci bibliotek i rozszerzeń dla wielu języków programowania. Należy jednak pamiętać, że po raz pierwszy opracowano je dla języka Prolog jako **CLP** (Constraint Logic Programming), w naturalny sposób rozszerzając deklaratywność tego języka (pierwsze implementacje to **Prolog III**, **CLP(R)** i **CHIP**).
6. Między innymi firma IBM dostarcza komercyjną bibliotekę **IBM ILOG CP Optimizer** dla języków C++ i Java [3]. Biblioteka ta jest omawiana na kursie **Technologia więzów** (kurs wybieralny na studiach drugiego stopnia).

Przykłady z biblioteki IBM ILOG CP Optimizer

Język C++

```
// -------------------------------------------------------------- -*- C++ -*-
// File: ./examples/src/cpp/color.cpp
// --------------------------------------------------------------------------
// Licensed Materials - Property of IBM
//
// 5724-Y48 5724-Y49 5724-Y54 5724-Y55 5725-A06 5725-A29
// Copyright IBM Corporation 1990, 2014. All Rights Reserved.
//
// Note to U.S. Government Users Restricted Rights:
// Use, duplication or disclosure restricted by GSA ADP Schedule
// Contract with IBM Corp.
// --------------------------------------------------------------------------

/* ------------------------------------------------------------

Problem Description
-------------------

The problem involves choosing colors for the countries on a map in
such a way that at most four colors (blue, white, yellow, green) are
used and no neighboring countries are the same color. In this exercise,
you will find a solution for a map coloring problem with six countries:
Belgium, Denmark, France, Germany, Luxembourg, and the Netherlands.

------------------------------------------------------------ */

#include <ilcp/cp.h>

const char* Names[] = {"blue", "white", "yellow", "green"};

int main(int , const char * []){
  IloEnv env;
  try {
    IloModel model(env);
    IloIntVar Belgium(env, 0, 3, "B"), Denmark(env, 0, 3, "DK"),
      France(env, 0, 3, "F"), Germany(env, 0, 3, "D"),
      Luxembourg(env, 0, 3, "L"), Netherlands(env, 0, 3, "NE");
    model.add(Belgium != France);
    model.add(Belgium != Germany);
    model.add(Belgium != Netherlands);
    model.add(Belgium != Luxembourg);
    model.add(Denmark != Germany );
    model.add(France != Germany);
    model.add(France != Luxembourg);
    model.add(Germany != Luxembourg);
    model.add(Germany != Netherlands);
    IloCP cp(model);
    if (cp.solve())
      {
      cp.out() << std::endl << cp.getStatus() << " Solution" << std::endl;
      cp.out() << "Belgium:     " << Names[cp.getValue(Belgium)] << std::endl;
      cp.out() << "Denmark:     " << Names[cp.getValue(Denmark)] << std::endl;
      cp.out() << "France:      " << Names[cp.getValue(France)] << std::endl;
      cp.out() << "Germany:     " << Names[cp.getValue(Germany)] << std::endl;
```

```
      cp.out() << "Luxembourg:  " << Names[cp.getValue(Luxembourg)]  << std::endl;
      cp.out() << "Netherlands: " << Names[cp.getValue(Netherlands)] << std::endl;
    }
  }
  catch (IloException& ex) {
    env.out() << "Error: " << ex << std::endl;
  }
  env.end();
  return 0;
}

/*
Feasible Solution
Belgium:     yellow
Denmark:     blue
France:      blue
Germany:     white
Luxembourg:  green
Netherlands: blue
*/
```

Język Java

```
// --------------------------------------------------------------*- Java -*-
// File: ./examples/src/java/Color.java
// --------------------------------------------------------------------------
// Licensed Materials - Property of IBM
//
// 5724-Y48 5724-Y49 5724-Y54 5724-Y55 5725-A06 5725-A29
// Copyright IBM Corporation 1990, 2017. All Rights Reserved.
//
// Note to U.S. Government Users Restricted Rights:
// Use, duplication or disclosure restricted by GSA ADP Schedule
// Contract with IBM Corp.
// --------------------------------------------------------------------------

/* ------------------------------------------------------------

Problem Description
-------------------

The problem involves choosing colors for the countries on a map in
such a way that at most four colors (blue, white, yellow, green) are
used and no neighboring countries are the same color. In this exercise,
you will find a solution for a map coloring problem with six countries:
Belgium, Denmark, France, Germany, Luxembourg, and the Netherlands.

------------------------------------------------------------ */

import ilog.cp.*;
import ilog.concert.*;

public class Color {
    public static String[] Names = {"blue", "white", "yellow", "green"};
    public static void main(String[] args) {
        try {
            IloCP cp = new IloCP();
            IloIntVar Belgium = cp.intVar(0, 3);
            IloIntVar Denmark = cp.intVar(0, 3);
            IloIntVar France = cp.intVar(0, 3);
            IloIntVar Germany = cp.intVar(0, 3);
            IloIntVar Luxembourg = cp.intVar(0, 3);
            IloIntVar Netherlands = cp.intVar(0, 3);


            cp.add(cp.neq(Belgium , France));
            cp.add(cp.neq(Belgium , Germany));
            cp.add(cp.neq(Belgium , Netherlands));
            cp.add(cp.neq(Belgium , Luxembourg));
            cp.add(cp.neq(Denmark , Germany));
            cp.add(cp.neq(France , Germany));
            cp.add(cp.neq(France , Luxembourg));
            cp.add(cp.neq(Germany , Luxembourg));
            cp.add(cp.neq(Germany , Netherlands));

            if (cp.solve())
                {
```

```
                    System.out.println();
                    System.out.println( "Belgium:     " + Names[(int)cp.getValue(Belgium)]);
                    System.out.println( "Denmark:     " + Names[(int)cp.getValue(Denmark)]);
                    System.out.println( "France:      " + Names[(int)cp.getValue(France)]);
                    System.out.println( "Germany:     " + Names[(int)cp.getValue(Germany)]);
                    System.out.println( "Luxembourg:  " + Names[(int)cp.getValue(Luxembourg)]);
                    System.out.println( "Netherlands: " + Names[(int)cp.getValue(Netherlands)]);
                }
        } catch (IloException e) {
            System.err.println("Error " + e);
        }
    }
}
```

Język CPO (modele dla programu CPoptimizer)

```
// Decision variables:
Belgium = intVar(1..4);
Denmark = intVar(1..4);
France = intVar(1..4);
Germany = intVar(1..4);
Luxembourg = intVar(1..4);
Netherlands = intVar(1..4);

/* Constraints: */
Belgium != France;
Belgium != Germany;
Belgium != Netherlands;
Belgium != Luxembourg;
Denmark != Germany;
France != Germany;
France != Luxembourg;
Germany != Luxembourg;
Germany != Netherlands;

parameters {
  SearchType = DepthFirst;
}
```

Dziękuję wszystkim za udział w kursie. Życzę powodzenia podczas sesji i udanego odpoczynku podczas wakacji. Mam nadzieję, że spotkamy się w październiku na uczelni, a nie tylko zdalnie przez sieć.

Bibliografia

1. John Wylie Lloyd. **Foundation of Logic Programming**. Springer-Verlan, 1993.
2. Robert Kowalski. **Logika w rozwiązywania zadań**. WNT, 1989. **(PDF**)
3. **IBM ILOG CPLEX Optimization Studio. CP Optimizer User's Manual**.

```
solve(Tiles, Board) :-
    board(Board),
    insert(Board, Tiles).
```

zauważymy, że nie ma w nim pierwszego etapu generowania rozwiązania i drugiego testowania. Po przygotowaniu predykatem **board/1** szkieletu planszy (pola zawierające zmienne, które połączono warunkami unifikacji) następuje wypełnienie szkieletu przez wkładanie do niego klocków predykatem **insert/2** (przegląd z nawrotami).

W prosty sposób można przekształcić predykat **solve/2**, tak by był zgodny z metodą generowania i testowania:

```
solve(Tiles, Board) :-
    insert(Board, Tiles),
    board(Board).
```

jednak zmiana taka bardzo wydłużyłaby czas poszukiwania rozwiązania (ponad 3 miliony kroków wnioskowania zamiast 469):

```
?- solve(t3).
% 3,260,122 inferences, 0.494 CPU in 0.501 seconds (99% CPU, 6596419 Lips)
true ;
% 62,058 inferences, 0.017 CPU in 0.020 seconds (82% CPU, 3749728 Lips)
false.
```

Pamiętajmy, metoda generowania i testowania to pełen przegląd. Stosujmy ją w ostateczności. Jeśli można zaproponować lepszy sposób poszukiwania rozwiązania, to postarajmy się go znaleźć.

Dlaczego pokazana na wykładzie wersja predykatu **solve/2** działa dużo szybciej niż ta zgodna z metodą generowania i testowania? Otóż zanim przystępujemy do wypełniania planszy klockami, narzucono już wszystkie równości na odpowiednie zmienne odpowiadające stykającym się ćwiartkom kwadratów. Zatem już częściowo wypełniona plansza może okazać się niedopuszczalna i Prolog wycofa się z tej gałęzi poszukiwań. W wersji pokazanej powyżej, każde umieszczenie klocków na planszy jest dopuszczalne i dopiero po wypełnieniu całej planszy predykatem **insert/2** sprawdzany jest predykatem **board/1** warunek zgodności kolorów na styku klocków.

Tags: nawracanie , odcięcie , generowanie i testowanie , metapredykat , negacja

Struktury danych (17 marca, wykład 3)

16/03/20 14:20

- język Java
- komunikat
- korutyna
- lista otwarta
- listy różnicowe
- logika Hoare'a
- maszyna abstrakcyjna
- metapredykat
- mutex
- najsłabszy warunek wstępny
- nawracanie
- negacja
- niedeterminizm
- odcięcie
- odraczanie celu
- ograniczenia arytmetyczne
- ograniczenia liniowe
- ograniczenie
- ograniczenie globalne
- optymalizacja
- otwieranie strumienia
- podsumowanie
- podział prostokąta
- predykat dynamiczny
- produkcja gramatyki
- programowanie obiektowe
- programowanie ograniczeń
- propagowanie ograniczenia
- pseudokod
- punkt wyboru
- rachunek predykatów
- reifikacja ograniczenia
- reprezentacja termów
- strategia sterowania etykietowaniem
- struktury danych
- strumień termów
- symbole nieterminalne
- symbole terminalne
- śledzenie działania
- TANTRIX Discovery
- termy
- unifikacja
- WAM
- wątek
- wejście/wyjście
- węzeł celtycki
- wzajemne wykluczenie
- zamykanie strumenia
- zmiana strumienia

FEEDS

- RSS Feed

 Contact Me

Tags: podsumowanie , język CPO , język C++ , język Java , rachunek predykatów , programowanie ograniczeń

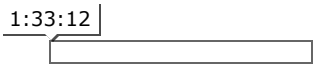
Comments

Przykłady programów (9 czerwca, wykład 14)

08/06/20 10:25

Programowanie w logice - wykład 14

from **Przemysław Kobylański**



(Obejrzano 1 razy)

Tags: pseudokod , formalna weryfikacja , TANTRIX Discovery , węzeł celtycki , podział prostokąta , logika Hoare'a , najsłabszy warunek wstępny

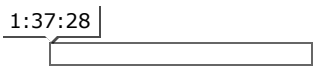
0 Comments

Globalne ograniczenia kombinatoryczne (2 czerwca, wykład 13)

02/06/20 11:28

Programowanie w logice - wykład 13

from **Przemysław Kobylański**



(Obejrzano 4 razy)

Tags: ograniczenie globalne , etykietowanie zmiennych , strategia sterowania etykietowaniem

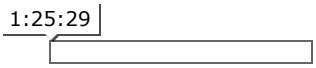
0 Comments

Proste ograniczenia (26 maja, wykład 12)

25/05/20 11:57

Programowanie w logice - wykład 12

from **Przemysław Kobylański**



(Obejrzano 17 razy)

Tags: ograniczenia arytmetyczne , ograniczenia liniowe , reifikacja ograniczenia

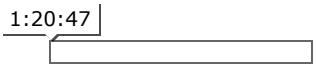
0 Comments

Zmienne, dziedziny i ograniczenia (19 maja, wykład 11)

19/05/20 07:07

Programowanie w logice - wykład 11

from **Przemysław Kobylański**



(Obejrzano 24 razy)

O atrybutach zmiennych można poczytać w rozdziale **Attributed variables** a o module CLP(FD) w dodatku **Library(clpfd)** podręcznika SWI-Prologu.

Tags: atrybut zmiennej , dziedzina , ograniczenie , propagowanie ograniczenia , etykietowanie zmiennych , optymalizacja

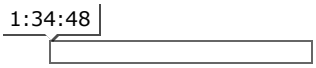
0 Comments

Korutyny i wątki (12 maja, wykład 10)

11/05/20 09:08

Programowanie w logice - wykład 10

from **Przemysław Kobylański**



(Obejrzano 25 razy)

O korutynach można poczytać w rozdziale **Corouting** a o wątkach w rozdziale **Multithreaded applications** podręcznika SWI-Prologu.

Tags: odraczanie celu , korutyna , wątek , komunikat , mutex , wzajemne wykluczenie , strumień termów , lista otwarta

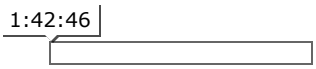
0 Comments

Interfejs graficzny (5 maja, wykład 9)

05/05/20 09:29

Programowanie w logice - wykład 9

from **Przemysław Kobylański**



(Obejrzany 13 razy)

O bibliotece XPCE można dodatkowo poczytać w podręczniku: Jan Wielemaker , Anjo Anjewierden. **Programming in XPCE/Prolog**.

Tags: interfejs graficzny , biblioteka XPCE , programowanie obiektowe

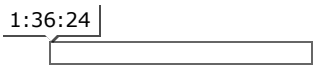
0 Comments

Gramatyki metamorficzne (28 kwietnia, wykład 8)

27/04/20 11:31

Programowanie w logice - wykład 8

from **Przemysław Kobylański**



(Obejrzany 31 razy)

Więcej o gramatykach metamorficznych możecie przeczytać w dziewiątym rozdziale książki W.F. Clocksin, C.S. Mellish. **Programming in Prolog** (odnośnik powinien działać również spoza kampusu PWr).

Tags: gramatyka bezkontekstowa , gramatyka metamorficzna , produkcja gramatyki , alfabet , symbole terminalne , symbole nieterminalne

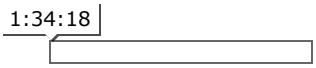
0 Comments

Śledzenie programów (21 kwietnia, wykład 7)

22/04/20 10:42

Programowanie w logice - wykład 7

from **Przemysław Kobylański**



(Obejrzany 22 razy)

Więcej o śledzeniu programów i szukaniu w nich błędów możecie przeczytać w ósmym rozdziale książki W.F. Clocksin, C.S. Mellish. **Programming in Prolog** (odnośnik powinien działać również spoza kampusu PWr).

Tags: śledzenie działania , niedeterminizm , punkt wyboru

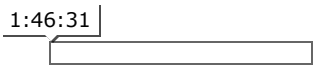
0 Comments

Przykłady programów (7 kwietnia, wykład 6)

06/04/20 23:00

Programowanie w logice - wykład 6

from **Przemysław Kobylański**



(Obejrzany 41 razy)

Jeśli kogoś interesuje jak działa WAM (Warren Abstract Machine) dla języka Prolog, to polecam raport Hassan Aït-Kaci **Warren's Abstract Machine: A Tutorial Reconstruction**.

DOPISANE

Jeszcze parę słów o złożoności obliczeniowej algorytmu unifikacji. Rozpatrzmy następujący przykład:

```
?- f(A, B, C, D) = f(a, g(A, A), g(B, B), g(C, C)).
A = a,
B = g(a, a),
C = g(g(a, a), g(a, a)),
D = g(g(g(a, a), g(a, a)), g(g(a, a), g(a, a))).
```

Wydawać by się mogło, że termy podstawiane pod kolejne zmienne A, B, C i D podwajają swoją długość. Sugerowałoby to wykładniczą złożoność pamięciową algorytmu unifikacji.

Jednak gdy przyjrzymy się algorytmowi unifikacji, to podczas jego działania nie są zajmowane nowe komórki pamięci. Jedynie przepinane są referencje prowadzące do miejsc zawierających struktury (unifikacja zmiennej i struktury) lub zmienne (unifikacja dwóch zmiennych).

Poniżej przedstawiona jest zawartość pamięci przed i po unifikacji termów **f(A, B, C, D)** i **f(a, g(A, A), g(B, B), g(C, C))**:

Powyższy rysunek w formacie PDF: **unifikacja.pdf**.

Tags: interpreter języka imperatywnego , interpreter języka deklaratywnego , unifikacja , maszyna abstrakcyjna , WAM , reprezentacja termów , predykat dynamiczny

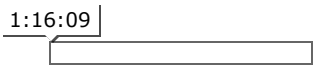
0 Comments

Wejście i wyjście (31 marca, wykład 5)

30/03/20 21:10

Programowanie w logice - wykład 5

from **Przemysław Kobylański**



(Obejrzany 45 razy)

Dodatkowo przeczytajcie piąty rozdział z książki W.F. Clocksin, C.S. Mellish. **Programming in Prolog** (odnośnik powinien działać również spoza kampusu PWr).

DOPISANE

Jeśli kogoś zainteresował problem samoreplikujących się programów, to polecam stronę **rosettacode/Quine** (dzisiaj dopisałem tam przykład w Prologu jaki pokazywałem na wykładzie).

Tags: wejście/wyjście , czytanie termu , drukowanie termu , czytanie znaku , drukowanie znaku , otwieranie strumienia , zamykanie strumenia , zmiana strumienia , efekt uboczny

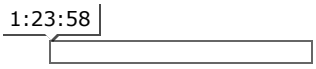
0 Comments

Poszukiwanie rozwiązań (24 marca, wykład 4)

23/03/20 19:30

Programowanie w logice - wykład 4

from **Przemysław Kobylański**



(Obejrzany 91 razy)

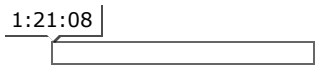
O generowaniu rozwiązań, nawracaniu i odcięciu można dodatkowo przeczytać w czwartym rozdziale książki W.F. Clocksin, C.S. Mellish. **Programming in Prolog** (odnośnik powinien działać również spoza kampusu PWr).

DOPISANE

W prezentację wkradł się błąd. Pokazany na koniec wykładu przykład gry Tetravex, wbrew temu co powiedziałem, nie jest rozwiązywany metodą generowania i testowania. Gdy popatrzymy na definicję predykatu **solve/2**:

Programowanie w logice - wykład 3

from **Przemysław Kobylański**



(Obejrzany 148 razy)

Dodatkowo przeczytajcie trzeci rozdział z książki W.F. Clocksin, C.S. Mellish. **Programming in Prolog** (odnośnik powinien działać również spoza kampusu PWr). Jeśli uda Wam się zamieścić poniżej pytania do wykładu, to postaram się na nie odpowiedzieć.

Tags: termy , unifikacja , listy różnicowe , struktury danych

1 Comment

ARCHIVES

- June 2020
- May 2020
- April 2020
- March 2020

TAGS

- alfabet
- atrybut zmiennej
- biblioteka XPCE
- czytanie termu
- czytanie znaku
- drukowanie termu
- drukowanie znaku
- dziedzina
- efekt uboczny
- etykietowanie zmiennych
- formalna weryfikacja
- generowanie i testowanie
- gramatyka bezkontekstowa
- gramatyka metamorficzna
- interfejs graficzny
- interpreter języka deklaratywnego
- interpreter języka imperatywnego
- język C++
- język CPO