

Wnioskowanie o programach w programowaniu w logice

Włodzimierz Drabent

IPI PAN, seminarium zespołu, 30 marca 2017
Wersja 1.7, 15 kwietnia 2017

Tematyka

programowanie w logice

podstawy

sprawy dotychczas pomijane.

W. Drabent (2016). Correctness and Completeness of Logic Programs. *ACM Transactions on Computational Logic* 17(3), 32 p.

W. Drabent (2017). Logic + control: On program construction and verification. *CoRR*. 1110.4978. 29 p. W recenzji do TPLP.

W. Drabent, M. Miłkowska (2005). Proving Correctness and Completeness of Normal Programs – a Declarative Approach. *Theory and Practice of Logic Programming*, 5(6):669–711.

Plan

- ▶ programowanie w logice – wprowadzenie
 - ▶ przykłady: zagadka, kolejka
 - ▶ dwa poziomy odczytania programu (logic + control)
- ▶ poprawność i pełność programów
 - ▶ specyfikacje
 - ▶ dowodzenie poprawności
 - ▶ dowodzenie pełności
 - ▶ systematyczna konstrukcja programów
- ▶ rola przybliżonych specyfikacji
 - ▶ lokalizacja błędów w programach
- ▶ Przykład, SAT-solver w Prologu

Przepraszam za dwujęzyczność slajdów

Tradycyjne (*imperatywne*) języki programowania
– myślenie w terminach maszyny von Neumanna.

- ▶ Program jest **opisem akcji maszyny** (CPU + RAM).
- ▶ Główne pojęcie – zmienna; np. $x := x + 1$.
(Abstrakcja komórki pamięci.)

Ta sama nazwa oznacza różne rzeczy.

(W matematyce: $x' = x + 1$, lub $x_{i+1} = x_i + 1$,
Java: $x = x + 1$.)

Klasyczna praca:

John Backus,

*Can programming be liberated from the von Neumann style?
a functional style and its algebra of programs*

Comm.ACM 21 (August 1978), 613-641.

Tradycyjne (*imperatywne*) języki programowania
– myślenie w terminach maszyny von Neumanna.

- ▶ Program jest *opisem akcji maszyny* (CPU + RAM).
- ▶ Główne pojęcie – zmienna; np. $x := x + 1$.
(Abstrakcja komórki pamięci.)

Ta sama nazwa oznacza różne rzeczy.

(W matematyce: $x' = x + 1$, lub $x_{i+1} = x_i + 1$,

Java: $x = x + 1$.)

Klasyczna praca:

John Backus,

*Can programming be liberated from the von Neumann style?
a functional style and its algebra of programs*

Comm.ACM 21 (August 1978), 613-641.

Tradycyjne (*imperatywne*) języki programowania
– myślenie w terminach maszyny von Neumanna.

- ▶ Program jest *opisem akcji maszyny* (CPU + RAM).
- ▶ Główne pojęcie – zmienna; np. $x := x + 1$.
(Abstrakcja komórki pamięci.)

Ta sama nazwa oznacza różne rzeczy.

(W matematyce: $x' = x + 1$, lub $x_{i+1} = x_i + 1$,
Java: $x = x + 1$.)

Klasyczna praca:

John Backus,

*Can programming be liberated from the von Neumann style?
a functional style and its algebra of programs*

Comm. ACM 21 (August 1978), 613-641.

Programowanie deklaratywne:

CO ma być obliczone, ale niekoniecznie JAK

Program – opisem problemu

a nie opisem operacji/działań maszyny.

Declarative programming paradigms

- ▶ Spreadsheets
- ▶ DB query languages
- ▶ Logic programming (LP)
- ▶ Functional programming (FP)
- ▶ SAT-solving

Logic Programming

Program – a set of axioms
Results – its logical consequences
Computation – proof construction

Main programming language – Prolog

Declarative programming paradigms

- ▶ Spreadsheets
- ▶ DB query languages
- ▶ Logic programming (LP)
- ▶ Functional programming (FP)
- ▶ SAT-solving

Logic Programming

Program – a set of axioms
Results – its logical consequences
Computation – proof construction

Main programming language – Prolog

Programowanie w języku logiki

(programowanie w logice, *logic programming*, LP)

Program – zbiór aksjomatów (postaci $A_0 \leftarrow A_1, \dots, A_n$,
 A_i – atomy (formuły atomowe)).

Obliczenie – rezolucja; znajduje logiczne konsekwencje programu.

Zapytanie Q (postaci A_1, \dots, A_n).

Odpowiedzi $Q\theta$ t.ż. $P \models Q\theta$
(P – program, θ – podstawienie).

Każda obliczona odpowiedź spełnia ten warunek.

I odwrotnie. (Jeśli $P \models Q\theta$ to $Q\theta$ jest instancją odpowiedzi obliczonej.)

LP, Example

A grandchild of x is a child of a child of x .

$$\text{grandchild}(X, Z) \leftarrow \text{child}(X, Y), \text{child}(Y, Z).$$

We add facts to the program:

child(charlie, adam). *child(david, charlie).*
child(charlie, barbara). *child(eva, charlie).*

Queries and answers (computing consequences of the program):

?- grandchild(charlie, adam) .	no
?- grandchild(eva, adam) .	yes
?- grandchild(X, adam) .	X = david; X = eva
?- grandchild(eva, Y)
?- grandchild(X, Y)

LP, Ex. 2, puzzle

Build a sequence out of three 1's, three 2's, ..., three 9's, so that between each consecutive occurrences of i there are exactly i elements.

[1,9,1,2,1,8,2,4,6,2,7,9,4,5,8,6,3,4,7,5,3,9,6,8,3,5,7]

[1,8,1,9,1,5,2,6,7,2,8,5,2,9,6,4,7,5,3,8,4,6,3,9,7,4,3]

[1,9,1,6,1,8,2,5,7,2,6,9,2,5,8,4,7,6,3,5,4,9,3,8,7,4,3]

[3,4,7,8,3,9,4,5,3,6,7,4,8,5,2,9,6,2,7,5,2,8,1,6,1,9,1]

[3,4,7,9,3,6,4,8,3,5,7,4,6,9,2,5,8,2,7,6,2,5,1,9,1,8,1]

[7,5,3,8,6,9,3,5,7,4,3,6,8,5,4,9,7,2,6,4,2,8,1,2,1,9,1]

Notacja

Zmienne w programach – z dużej litery.

$[a_1, \dots, a_n]$ – lista z elementami a_1, \dots, a_n ($n \geq 0$)

$[]$ – lista pusta

$[h|t]$ – lista z głową h i ogonem t

$[h_1, h_2|t]$ – lista z głową h_1 i ogonem $[h_2|t]$, czyli $[h_1|[h_2|t]]$

LP, Ex. 2, puzzle

```

solution(S) ←
  sequence27(S),
  sublist([1, -, 1, -, 1], S),
  sublist([2, -, -, 2, -, -, 2], S),
  sublist([3, -, -, -, 3, -, -, -, 3], S),
  sublist([4, -, -, -, -, 4, -, -, -, -, 4], S),
  sublist([5, -, -, -, -, -, 5, -, -, -, -, -, 5], S),
  sublist([6, -, -, -, -, -, -, 6, -, -, -, -, -, -, 6], S),
  sublist([7, -, -, -, -, -, -, -, 7, -, -, -, -, -, -, -, 7], S),
  sublist([8, -, -, -, -, -, -, -, -, 8, -, -, -, -, -, -, -, -, 8], S),
  sublist([9, -, -, -, -, -, -, -, -, -, 9, -, -, -, -, -, -, -, -, -, 9], S).

```

```

sublist(Y, XYZ) ← app( -, YZ, XYZ ), app(Y, -, YZ ).

```

```

sequence27([-, -, -, -, -, -, -, -, -, -, -, -, -, -, -, -, -, -, -, -, -, -, -, -, -, -]).

```

```

app([], L, L).

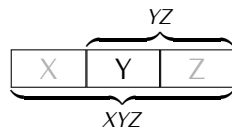
```

```

app([H|K], L, [H|M]) ← app( K, L, M ).

```

LP, Ex. 2, puzzle



```

solution(S) ←
  sequence27(S),
  sublist([1, -, 1, -, 1], S),
  sublist([2, -, -, 2, -, -, 2], S),
  sublist([3, -, -, -, 3, -, -, -, 3], S),
  sublist([4, -, -, -, -, 4, -, -, -, -, 4], S),
  sublist([5, -, -, -, -, -, 5, -, -, -, -, -, 5], S),
  sublist([6, -, -, -, -, -, -, 6, -, -, -, -, -, -, 6], S),
  sublist([7, -, -, -, -, -, -, -, 7, -, -, -, -, -, -, -, 7], S),
  sublist([8, -, -, -, -, -, -, -, -, 8, -, -, -, -, -, -, -, -, 8], S),
  sublist([9, -, -, -, -, -, -, -, -, -, 9, -, -, -, -, -, -, -, -, -, 9], S).
  
```

```

sublist(Y, XYZ) ← app( -, YZ, XYZ ), app(Y, -, YZ ).
  
```

```

sequence27([-, -, -, -, -, -, -, -, -, -, -, -, -, -, -, -, -, -, -, -, -, -, -, -]).
  
```

```

app([], L, L).
  
```

```

app([H|K], L, [H|M]) ← app( K, L, M).
  
```

LP, Ex. 2, puzzle

```

solution(S) ←
  sequence27(S),
  sublist([1, -, 1, -, 1], S),
  sublist([2, -, -, 2, -, -, 2], S),
  sublist([3, -, -, -, 3, -, -, -, 3], S),
  sublist([4, -, -, -, -, 4, -, -, -, -, 4], S),
  sublist([5, -, -, -, -, -, 5, -, -, -, -, -, 5], S),
  sublist([6, -, -, -, -, -, -, 6, -, -, -, -, -, -, 6], S),
  sublist([7, -, -, -, -, -, -, -, 7, -, -, -, -, -, -, -, 7], S),
  sublist([8, -, -, -, -, -, -, -, -, 8, -, -, -, -, -, -, -, -, 8], S),
  sublist([9, -, -, -, -, -, -, -, -, -, 9, -, -, -, -, -, -, -, -, -, 9], S).

```



```

sublist(Y, XYZ) ← app( -, YZ, XYZ ), app(Y, -, YZ ).
sequence27([-, -, -, -, -, -, -, -, -, -, -, -, -, -, -, -, -, -, -, -, -, -]).
app([], L, L).
app([H|K], L, [H|M]) ← app( K, L, M).

```


Przykład, kolejka

Para list $[e_1, \dots, e_n, e_{n+1}, \dots] - [e_{n+1}, \dots]$ reprezentuje
kolejkę e_1, \dots, e_n .

(Ogólniej i dokładniej: para termów $[e_1, \dots, e_n | t] - t$, "lista różnicowa")

Program + zapytanie/odpowiedź:

empty($A - A$).

deq($E, [E | A] - B, A - B$).

enq($E, A - [E | B], A - B$).

Obliczenia leniwe (e pobrany przed jego wstawieniem)

| ?- *deq*($X, A - A, Q1$).

$A = [X | A], \quad Q1 = A - [X | A] \quad \text{kolejka o długości } -1 !$

| ?- *deq*($X, A - A, Q1$), *enq*($a, Q1, Q2$).

$A = [a | A], \quad X = a, \quad Q1 = A - [a | A], \quad Q2 = A - A$

Przykład, kolejka

Para list $[e_1, \dots, e_n, e_{n+1}, \dots] - [e_{n+1}, \dots]$ reprezentuje
kolejkę e_1, \dots, e_n .

(Ogólniej i dokładniej: para termów $[e_1, \dots, e_n | t] - t$, "lista różnicowa")

Program + zapytanie/odpowiedź:

empty($A - A$).

deq($E, [E | A] - B, A - B$).

enq($E, A - [E | B], A - B$).

Obliczenia leniwe (e pobrany przed jego wstawieniem)

| ?- *deq*($X, A - A, Q1$).

$A = [X | _A], \quad Q1 = _A - [X | _A] \quad \text{kolejka o długości } -1 !$

| ?- *deq*($X, A - A, Q1$), *enq*($a, Q1, Q2$).

$A = [a | _A], \quad X = a, \quad Q1 = _A - [a | _A], \quad Q2 = _A - _A$

Przykład, kolejka

Para list $[e_1, \dots, e_n, e_{n+1}, \dots] - [e_{n+1}, \dots]$ reprezentuje
kolejkę e_1, \dots, e_n .

(Ogólniej i dokładniej: para termów $[e_1, \dots, e_n | t] - t$, "lista różnicowa")

Program + zapytanie/odpowiedź:

empty($A - A$).

deq($E, [E | A] - B, A - B$).

enq($E, A - [E | B], A - B$).

| ?- *enq*(4, [1,2,3|T]-T, Q).

Q = [1,2,3,4|_A]-_A, T = [4|_A]

Obliczenia leniwe (e pobrany przed jego wstawieniem)

| ?- *deq*(X, A - A, Q1).

A = [X|_A], Q1 = _A - [X|_A] kolejka o długości -1 !

| ?- *deq*(X, A - A, Q1), *enq*(a, Q1, Q2).

A = [a|_A], X = a, Q1 = _A - [a|_A], Q2 = _A - A

Przykład, kolejka

Para list $[e_1, \dots, e_n, e_{n+1}, \dots] - [e_{n+1}, \dots]$ reprezentuje
kolejkę e_1, \dots, e_n .

(Ogólniej i dokładniej: para termów $[e_1, \dots, e_n | t] - t$, "lista różnicowa")

Program + zapytanie/odpowiedź:

empty(A-A).

deq(E, [E|A]-B, A-B).

enq(E, A-[E|B], A-B).

| ?- enq(4, [1,2,3|T]-T, Q).

Q = [1,2,3,4|_A]-_A, T = [4|_A]

| ?- deq(X, [1,2,3|T]-T, Q1).

X = 1, Q1 = [2,3|T]-T

Obliczenia leniwe (e pobrany przed jego wstawieniem)

| ?- deq(X, A-A, Q1).

A = [X|_A], Q1 = _A-[X|_A] kolejka o długości -1 !

| ?- deq(X, A-A, Q1), enq(a, Q1, Q2).

A = [a|_A], X = a, Q1 = _A-[a|_A], Q2 = _A-_A

Przykład, kolejka

Para list $[e_1, \dots, e_n, e_{n+1}, \dots] - [e_{n+1}, \dots]$ reprezentuje
kolejkę e_1, \dots, e_n .

(Ogólniej i dokładniej: para termów $[e_1, \dots, e_n | t] - t$, "lista różnicowa")

Program + zapytanie/odpowiedź:

empty(A-A).

deq(E, [E|A]-B, A-B).

enq(E, A-[E|B], A-B).

| ?- enq(4, [1,2,3|T]-T, Q).

Q = [1,2,3,4|_A]-_A, T = [4|_A]

| ?- deq(X, [1,2,3|T]-T, Q1).

X = 1, Q1 = [2,3|T]-T

Obliczenia leniwe (e pobrany przed jego wstawieniem)

| ?- deq(X, A-A, Q1).

A = [X|_A], Q1 = _A-[X|_A] kolejka o długości -1 !

| ?- deq(X, A-A, Q1), enq(a, Q1, Q2).

A = [a|_A], X = a, Q1 = _A-[a|_A], Q2 = _A-_A

Dwa poziomy odczytania programu

deklaratywny – zbiór aksjomatów,
operacyjny – opis obliczeń.

ALGORITHM = LOGIC + CONTROL

[Robert Kowalski, 1974]

Poziom operacyjny: informacja o sterowaniu
zawarta w kolejności zapisu i pewnych dodatkowych deklaracjach.

Ważne:

Poziomy te można rozważać niezależnie.

Poprawność programów to własność poziomu deklaratywnego.

Dwa poziomy odczytania programu

deklaratywny – zbiór aksjomatów,
operacyjny – opis obliczeń.

ALGORITHM = LOGIC + CONTROL

[Robert Kowalski, 1974]

Poziom operacyjny: informacja o sterowaniu
zawarta w kolejności zapisu i pewnych dodatkowych deklaracjach.

Ważne:

Poziomy te można rozważać niezależnie.

Poprawność programów to własność poziomu deklaratywnego.

Nie myślimy w terminach opisu obliczeń maszyny.

Zmienne nie jak w programowaniu imperatywnym.

(Można też programować operacyjnie, zapominając o 1. poziomie.)

Dwa poziomy odczytania programu

deklaratywny – zbiór aksjomatów,
operacyjny – opis obliczeń.

ALGORITHM = LOGIC + CONTROL

[Robert Kowalski, 1974]

Poziom operacyjny: informacja o sterowaniu
zawarta w kolejności zapisu i pewnych dodatkowych deklaracjach.

Ważne, ale często pomijane:

Poziomy te można rozważać niezależnie.

Poprawność programów to własność poziomu deklaratywnego.

Nie myślimy w terminach opisu obliczeń maszyny.

Zmienne nie jak w programowaniu imperatywnym.

(Można też programować operacyjnie, zapominając o 1. poziomie.)

Sterowanie (w jęz. prog. Prolog)

Kolejność implikacji,
prześlanek w implikacjach,
dodatkowe konstrukcje.

Semantyka operacyjna, czyli obliczenia w LP (wzmianka)

SLD-resolution, szczególny przypadek rezolucji.

Inaczej: **konstrukcja** od końca **dowodu** za pom. reguły odrywania
dla pewnej instancji zapytania Q .

Można zobrazować jako **drzewo SLD** (*SLD-tree*),

węzły – zapytania, korzeń – Q ,

puste zapytanie – odpowiedź znaleziona,

w każdym niepustym węźle *wybrany atom*

Dla danego P, Q możliwe różne drzewa SLD,
każde zawiera wszystkie odpowiedzi.

Zwykle: obliczenie – przeszukiwanie drzewa wgłąb
(z nawracaniem).

Rezolucja SLD, krok obliczeń

Uzgadnianie (unifikacja, *unification*)

Dwa wyrażenia A, B , ich **mgu** –

najbardziej ogólne podstawienie θ t.ż. $A\theta = B\theta$

mgu – most general unifier (najogólniejsze podstawienie uzgadniające).

$Q = A_1, \dots, A_i, \dots, A_m$ – zapytanie, A_i – wybrany atom

$C = H \leftarrow B_1, \dots, B_n$ – klauzula z programu

θ – mgu A_i i H

$Q' = (A_1, \dots, A_{i-1}, B_1, \dots, B_n, A_{i+1}, \dots, A_m)\theta$

– **rezolwent** Q i C wzgl. A_i z mgu θ

Własność:

$C \models Q' \rightarrow Q\theta$

Gdy mamy wiele kroków $(Q_0, \theta_1, Q_1, \dots, \theta_n, Q_n)$,

$\text{Program} \models Q_n \rightarrow Q_0\theta_1 \cdots \theta_n$

Rezolucja SLD, krok obliczeń

Uzgadnianie (unifikacja, *unification*)

Dwa wyrażenia A, B , ich **mgu** –

najbardziej ogólne podstawienie θ t.ż. $A\theta = B\theta$

mgu – most general unifier (najogólniejsze podstawienie uzgadniające).

$Q = A_1, \dots, A_i, \dots, A_m$ – zapytanie, A_i – wybrany atom

$C = H \leftarrow B_1, \dots, B_n$ – klauzula z programu

θ – mgu A_i i H

$Q' = (A_1, \dots, A_{i-1}, B_1, \dots, B_n, A_{i+1}, \dots, A_m)\theta$

– **rezolwent** Q i C wzgl. A_i z mgu θ

Własność:

$C \models Q' \rightarrow Q\theta$

Gdy mamy wiele kroków $(Q_0, \theta_1, Q_1, \dots, \theta_n, Q_n)$,

$\text{Program} \models Q_n \rightarrow Q_0\theta_1 \dots \theta_n$

Spojrzenie programisty

Termy – struktury danych

Predykat – procedura

Klauzule programu rozpoczynające się od p
– deklaracja procedury p .

Atom $p(\dots)$ w przesłankach klauzuli – wołanie procedury p .

Atom $p(\dots)$ wybranie w węźle drzewa SLD – wywołanie p .

t_i w $p(t_1, \dots, t_n)$ – parametr we./wy.

Rola unifikacji (uzgadniania).

- ▶ Przekazywanie argumentów (we., wy.)
- ▶ Sprawdzanie warunków
- ▶ Konstrukcja struktur danych
- ▶ Dekonstrukcja struktur danych (dostęp)

Spojrzenie programisty

Termy – struktury danych

Predykat – procedura

Klauzule programu rozpoczynające się od p
– deklaracja procedury p .

Atom $p(\dots)$ w przesłankach klauzuli – wołanie procedury p .

Atom $p(\dots)$ wybranie w węźle drzewa SLD – wywołanie p .

t_i w $p(t_1, \dots, t_n)$ – parametr we./wy.

Rola unifikacji (uzgadniania).

- ▶ Przekazywanie argumentów (we., wy.)
- ▶ Sprawdzanie warunków
- ▶ Konstrukcja struktur danych
- ▶ Dekonstrukcja struktur danych (dostęp)

Spojrzenie programisty, c.d.

Zmienna w programowaniu imperatywnym – pojemnik na dane.

Zmienna w LP:

może nie mieć wartości,

wartość nie może się zmienić, może być uszczegółowiona, np.

$X \dots X = Y \dots X = Y = f(-, -) \dots X = Y = f(a, g(-)) \dots$

Poprawność programów

Wyniki – logiczne konsekwencje programu.

Nie zawsze jasne, czym one są.

Jak o nich wnioskować?

W programowaniu

imperatywnym: częściowa poprawność + terminacja.

Poprawność programów

Wyniki – logiczne konsekwencje programu.

Nie zawsze jasne, czym one są.

Jak o nich wnioskować?

W programowaniu

imperatywnym: częściowa poprawność + terminacja.

Poprawność – zgodność wyników programu ze specyfikacją.

Pełność – program da wszystkie wyniki wymagane specyfikację.
przez

Poprawność programów

Wyniki – logiczne konsekwencje programu.

Nie zawsze jasne, czym one są.

Jak o nich wnioskować?

W programowaniu

imperatywnym:

częściowa poprawność + terminacja.

W LP:

poprawność

pełność

Poprawność – zgodność wyników programu ze specyfikacją.

Pełność – program da wszystkie wyniki wymagane specyfikacją.
przez

Poprawność programów

Wyniki – logiczne konsekwencje programu.

Nie zawsze jasne, czym one są.

Jak o nich wnioskować?

W programowaniu
imperatywnym:

częściowa poprawność + terminacja.

W LP:

poprawność pełność

zupełna poprawność?

ogólna? mocna?
dwustronna?

Poprawność – zgodność wyników programu ze specyfikacją.

Pełność – program da wszystkie wyniki wymagane specyfikację.
przez

Specyfikacja

- zbiór atomów bez zmiennych (ang. *ground*), $S \subseteq \mathcal{HB}$.
(Nieformalnie – zbiór najprostszyc odpowiedzi.)

Np. $S = \{ \text{solution}(l) \in \mathcal{HB} \mid l \text{ jest listą, } |l| = 27, \dots \dots \} \cup$
 $\{ \text{sequence27}(l) \in \mathcal{HB} \mid l \text{ jest listą, } |l| = 27 \} \cup$
 $\left\{ \text{sublist}(s, l) \in \mathcal{HB} \mid \begin{array}{l} l = [a_1, \dots, a_n] \Rightarrow \\ \exists i, j \in \{1, \dots, n\} : s = [a_i, \dots, a_j] \end{array} \right\} \cup$
 $\{ \text{app}(k, l, m) \in \mathcal{HB} \mid \dots \}$

Df.: Poprawność: $M_P \subseteq S$ Pełność: $S \subseteq M_P$

gdzie P – program, S – specyfikacja.

M_P – odpowiedzi dla P atomowe, bez zmiennych,

$M_P = \{A \in \mathcal{HB} \mid A \text{ jest odpowiedzią } P\}$ (najmniejszy model Herbrand).

Specyfikacja

- zbiór atomów bez zmiennych (ang. *ground*), $S \subseteq \mathcal{HB}$.
(Nieformalnie – zbiór najprostszyc odpowiedzi.)

Np. $S = \{ \text{solution}(l) \in \mathcal{HB} \mid l \text{ jest listą, } |l| = 27, \dots \dots \} \cup$
 $\{ \text{sequence27}(l) \in \mathcal{HB} \mid l \text{ jest listą, } |l| = 27 \} \cup$
 $\left\{ \text{sublist}(s, l) \in \mathcal{HB} \mid \begin{array}{l} l = [a_1, \dots, a_n] \Rightarrow \\ \exists i, j \in \{1, \dots, n\} : s = [a_i, \dots, a_j] \end{array} \right\} \cup$
 $\{ \text{app}(k, l, m) \in \mathcal{HB} \mid \dots \}$

Df.: Poprawność: $M_P \subseteq S$ Pełność: $S \subseteq M_P$

gdzie P – program, S – specyfikacja.

M_P – odpowiedzi dla P atomowe, bez zmiennych,

$M_P = \{A \in \mathcal{HB} \mid A \text{ jest odpowiedzią } P\}$ (najmniejszy model Herbranda).

Własność poprawności i pewności

Poprawność: $M_P \subseteq S$

Pełność: $S \subseteq M_P$

opisują odpowiedzi programu, czyli to, o co chodzi:

Tw.:

P poprawny wzgl. S ,
 $Q\theta$ odpowiedź dla $P \Rightarrow S \models Q\theta$

P pełny wzgl. S ,
 $S \models Q\theta$,
 $Q\theta$ bez zmiennych¹ $\Rightarrow Q\theta$ odpowiedź dla P

¹ Zw. z zagadnieniem, kiedy M_P charakteryzuje logiczne konsekwencje P .
W.D. Theory and Practice of Logic Programming 16(4):498-508.

Własność poprawności i pewności

Poprawność: $M_P \subseteq S$

Pełność: $S \subseteq M_P$

opisują odpowiedzi programu, czyli to, o co chodzi:

Tw.:

P poprawny wzgl. S ,
 $Q\theta$ odpowiedź dla $P \Rightarrow S \models Q\theta$

P pełny wzgl. S ,
 $S \models Q\theta$,
 $Q\theta$ bez zmiennych¹ $\Rightarrow Q\theta$ odpowiedź dla P

¹ Zw. z zagadnieniem, kiedy M_P charakteryzuje logiczne konsekwencje P .
 W.D. Theory and Practice of Logic Programming 16(4):498-508.

Kryterium poprawności

Tw.: $S \models P \Rightarrow P$ poprawny względem S .

$S \models P$ inaczej:

Dla każdej instancji bez zmiennych $H \leftarrow B_1, \dots, B_n$ implikacji z P ,
jeśli $B_1, \dots, B_n \in S$, to $H \in S$.

Example correctness proof

$$[B_1, \dots, B_n \in S \Rightarrow H \in S, \text{ for each } \dots H \leftarrow B_1, \dots, B_n.]$$

Program + specification:

$$\text{SPLIT: } s([], [], []). \quad (1)$$

$$s([X|Xs], [X|Ys], Zs) \leftarrow s(Xs, Zs, Ys). \quad (2)$$

$$S = \{ s(l, l_1, l_2) \mid l, l_1, l_2 \text{ are lists, } 0 \leq |l_1| - |l_2| \leq 1 \},$$

where $|l|$ – the length of a list l .

Proof:

Consider a ground instance $s([h|t], [h|t_2], t_1) \leftarrow s(t, t_1, t_2)$ of (2).

Assume $s(t, t_1, t_2) \in S$. Thus $[h|t], [h|t_2], t_1$ are lists.

Let $m = |t_1| - |t_2|$. As $m \in \{0, 1\}$, we have $||h|t_2|| - |t_1| = 1 - m \in \{0, 1\}$.

So the head $s([h|t], [h|t_2], t_1)$ is in S . The proof for (1) is trivial.

Thus program SPLIT correct w.r.t. specification S .

Uwaga

Przedstawiony warunek dostateczny poprawności $S \models P$
[Clark'79]

uważam, że prosty, naturalny, łatwy w stosowaniu 😊

zapoznany

Proponuje się inne metody, niedeklaratywne 😞

(np. monografia [Apt'97])

≈ metoda Hoare'a zastosowana do pewnego konkretnego sterowania.)

Pełność

[Każdy wynik wymagany przez specyfikację jest wynikiem programu]

$$[S \subseteq M_P]$$

Uwagi

Na ogół pomijana (!) ☹

Np.

Podręcznik [Hogger'84] – tylko definicja

Monografia [Apt'97] – brak.

Wnioskowanie o pełności

Pojęcie pomocnicze: **semi-pełność** (półpełność?)

= pełność dla wszystkich zapytań,
dla których istnieje obliczenie skończone.

Tzn. istnieje skończone drzewo przeszukiwań (SLD-tree).

Czyli $\text{pełność} = \text{semi-pełność} + \text{terminacja}$.

Wnioskowanie o semi-pełności

Df.: Atom A jest **pokryty** wzgl. specyfikacji S przez klauzulę C jeśli ma ona instancję $A \leftarrow B_1, \dots, B_n$, gdzie $B_1, \dots, B_n \in S$.

Nieformalnie: C może wyprodukować A z S .

Df.: A pokryty wzgl. S przez program $P \Leftrightarrow A$ pokryty wzgl. S przez $C \in P$

Tw.: S – specyfikacja, P – program;
każdy $A \in S$ pokryty przez P wzgl. $S \Rightarrow P$ **semi-pełny** wzgl. S

Przykład. Konstrukcja programu i dowód semi-pełności

Sprawdzanie czy element listy. Specyfikacja

$$S_M^0 = \{ \text{member}(t_i, [t_1, \dots, t_n]) \in \mathcal{HB} \mid 0 < n, 1 \leq i \leq n \}$$

Warunek na semi-pełność:

każdy $A \in S_M^0$ pokryty przez P wzgl. $S_M^0 \Rightarrow P$ semi-pełny wzgl. S_M^0

$i = 1$: każdy $\text{member}(t_1, [t_1, \dots, t_n])$ pokryty przez

$$\text{member}(X, [X|L]).$$

$1 < i \leq n$:

każdy $\text{member}(t_i, [t_1, \dots, t_n])$ ($1 < i \leq n$) pokryty przez

$$\text{member}(X, [Y|L]) \leftarrow \text{member}(X, L).$$

Program M gotowy wraz z dowodem semi-pełności.

Powszechnie używany, standardowy.

Przykład. Konstrukcja programu i dowód semi-pełności

Sprawdzanie czy element listy. Specyfikacja

$$S_M^0 = \{ \text{member}(t_i, [t_1, \dots, t_n]) \in \mathcal{HB} \mid 0 < n, 1 \leq i \leq n \}$$

Warunek na semi-pełność:

każdy $A \in S_M^0$ pokryty przez P wzgl. $S_M^0 \Rightarrow P$ semi-pełny wzgl. S_M^0

$i = 1$: każdy $\text{member}(t_1, [t_1, \dots, t_n])$ pokryty przez

$$\text{member}(X, [X|L]).$$

$1 < i \leq n$:

każdy $\text{member}(t_i, [t_1, \dots, t_n])$ ($1 < i \leq n$) pokryty przez

$$\text{member}(X, [Y|L]) \leftarrow \text{member}(X, L).$$

Program M gotowy wraz z dowodem semi-pełności.

Powszechnie używany, standardowy.

Przykład. Konstrukcja programu i dowód semi-pełności

Sprawdzanie czy element listy. Specyfikacja

$$S_M^0 = \{ \text{member}(t_i, [t_1, \dots, t_n]) \in \mathcal{HB} \mid 0 < n, 1 \leq i \leq n \}$$

Warunek na semi-pełność:

każdy $A \in S_M^0$ pokryty przez P wzgl. $S_M^0 \Rightarrow P$ semi-pełny wzgl. S_M^0

$i = 1$: każdy $\text{member}(t_1, [t_1, \dots, t_n])$ pokryty przez

$$\text{member}(X, [X|L]).$$

$1 < i \leq n$:

każdy $\text{member}(t_i, [t_1, \dots, t_n])$ ($1 < i \leq n$) pokryty przez

$$\text{member}(X, [Y|L]) \leftarrow \text{member}(X, L).$$

Program M gotowy wraz z dowodem semi-pełności.

Powszechnie używany, standardowy.

Przykład. Konstrukcja programu i dowód semi-pełności

Sprawdzanie czy element listy. Specyfikacja

$$S_M^0 = \{ \text{member}(t_i, [t_1, \dots, t_n]) \in \mathcal{HB} \mid 0 < n, 1 \leq i \leq n \}$$

Warunek na semi-pełność:

każdy $A \in S_M^0$ pokryty przez P wzgl. $S_M^0 \Rightarrow P$ semi-pełny wzgl. S_M^0

$i = 1$: każdy $\text{member}(t_1, [t_1, \dots, t_n])$ pokryty przez

$$\text{member}(X, [X|L]).$$

$1 < i \leq n$:

każdy $\text{member}(t_i, [t_1, \dots, t_n])$ ($1 < i \leq n$) pokryty przez

$$\text{member}(X, [Y|L]) \leftarrow \text{member}(X, L).$$

Program M gotowy wraz z dowodem semi-pełności.

Powszechnie używany, standardowy.

Przykład. Konstrukcja programu i dowód semi-pełności

Sprawdzanie czy element listy. Specyfikacja

$$S_M^0 = \{ \text{member}(t_i, [t_1, \dots, t_n]) \in \mathcal{HB} \mid 0 < n, 1 \leq i \leq n \}$$

Warunek na semi-pełność:

każdy $A \in S_M^0$ pokryty przez P wzgl. $S_M^0 \Rightarrow P$ semi-pełny wzgl. S_M^0

$i = 1$: każdy $\text{member}(t_1, [t_1, \dots, t_n])$ pokryty przez

$$\text{member}(X, [X|L]).$$

$1 < i \leq n$:

każdy $\text{member}(t_i, [t_1, \dots, t_n])$ ($1 < i \leq n$) pokryty przez

$$\text{member}(X, [Y|L]) \leftarrow \text{member}(X, L).$$

Program M gotowy wraz z dowodem semi-pełności.

Powszechnie używany, standardowy.

Przykład, c.d.

Oczywiście program powinien być też poprawny.

Program $\{ \text{member}(X, Y). \}$ też jest pełny wzgl. S_M^0 ,
a bezużyteczny.

Przykład, c.d.

Specyfikacja i program:

$$S_M^0 = \{ \text{member}(t_i, [t_1, \dots, t_n]) \in \mathcal{HB} \mid 1 \leq i \leq n \}$$

$$M: \quad \text{member}(E, [E|L]). \quad (1)$$

$$\text{member}(E, [H|L]) \leftarrow \text{member}(E, L). \quad (2)$$

Ale M **niepoprawny** wzgl. S_M^0

bowiem $M \models \text{member}(a, [a|b])$ ($[a|b]$ nie jest listą).

Nic nie szkodzi, M poprawny względem

$$\begin{aligned} S_M &= \left\{ m(t, l) \in \mathcal{HB} \mid \begin{array}{l} l \text{ jest listą} \Rightarrow \\ t \text{ jest elementem } l \end{array} \right\} \\ &= \{ m(t, s) \in \mathcal{HB} \mid s \text{ nie jest listą} \} \cup S_M^0 \end{aligned}$$

Przykład, c.d.

Specyfikacja i program:

$$S_M^0 = \{ \text{member}(t_i, [t_1, \dots, t_n]) \in \mathcal{HB} \mid 1 \leq i \leq n \}$$

$$M: \quad \text{member}(E, [E|L]). \quad (1)$$

$$\text{member}(E, [H|L]) \leftarrow \text{member}(E, L). \quad (2)$$

Ale M **niepoprawny** wzgl. S_M^0

bowiem $M \models \text{member}(a, [a|b])$ ($[a|b]$ nie jest listą).

Nic nie szkodzi, M poprawny względem

$$\begin{aligned} S_M &= \left\{ m(t, l) \in \mathcal{HB} \mid \begin{array}{l} l \text{ jest listą} \Rightarrow \\ t \text{ jest elementem } l \end{array} \right\} \\ &= \{ m(t, s) \in \mathcal{HB} \mid s \text{ nie jest listą} \} \cup S_M^0 \end{aligned}$$

Przykład, c.d.

Specyfikacja i program:

$$S_M^0 = \{ \text{member}(t_i, [t_1, \dots, t_n]) \in \mathcal{HB} \mid 1 \leq i \leq n \}$$

$$M: \quad \text{member}(E, [E|L]). \quad (1)$$

$$\text{member}(E, [H|L]) \leftarrow \text{member}(E, L). \quad (2)$$

Ale M **niepoprawny** wzgl. S_M^0

bowiem $M \models \text{member}(a, [a|b])$ ($[a|b]$ nie jest listą).

Nic nie szkodzi, M poprawny względem

$$\begin{aligned} S_M &= \left\{ m(t, l) \in \mathcal{HB} \mid \begin{array}{l} l \text{ jest listą} \Rightarrow \\ t \text{ jest elementem } l \end{array} \right\} \\ &= \{ m(t, s) \in \mathcal{HB} \mid s \text{ nie jest listą} \} \cup S_M^0 \end{aligned}$$

Specyfikacje przybliżone


W przykładzie:

$$S_M^0 = \{ \text{member}(t_i, [t_1, \dots, t_n]) \in \mathcal{HB} \mid 1 \leq i \leq n \}$$

$$S_M = \{ \text{member}(t, s) \in \mathcal{HB} \mid s \text{ nie jest listą} \} \cup S_M^0$$

Program poprawny wzgl. S_M i pełny wzgl. S_M^0 .

Jest to sytuacja typowa. Choć teoria LP nieczęsto to przyznaje.

 Wiemy, co program **musi** produkować (S_M^0),
co program **może** produkować (S_M).

Reszta nieważna.

Specyfikacje przybliżone

W przykładzie:

$$S_M^0 = \{ \text{member}(t_i, [t_1, \dots, t_n]) \in \mathcal{HB} \mid 1 \leq i \leq n \}$$

$$S_M = \{ \text{member}(t, s) \in \mathcal{HB} \mid s \text{ nie jest listą} \} \cup S_M^0$$

Program poprawny wzgl. S_M i pełny wzgl. S_M^0 .

Jest to sytuacja typowa. Choć teoria LP nieczęsto to przyznaje.



Wiemy, co program **musi** produkować (S_M^0),
co program **może** produkować (S_M).

Reszta nieważna.

Nota bene, dokładna specyfikacja M :

$$\{ \text{member}(t_i, [t_1, \dots, t_i|s]) \in \mathcal{HB} \mid i \geq 0 \}$$

Ale ogólnie:

dokładna specyfikacja może być trudna i niewygodna do uzyskania,
i niepotrzebna.

Specyfikacje przybliżone

W przykładzie:

$$S_M^0 = \{ \text{member}(t_i, [t_1, \dots, t_n]) \in \mathcal{HB} \mid 1 \leq i \leq n \}$$

$$S_M = \{ \text{member}(t, s) \in \mathcal{HB} \mid s \text{ nie jest listą} \} \cup S_M^0$$

Program poprawny wzgl. S_M i pełny wzgl. S_M^0 .

Jest to sytuacja typowa. Choć teoria LP nieczęsto to przyznaje.



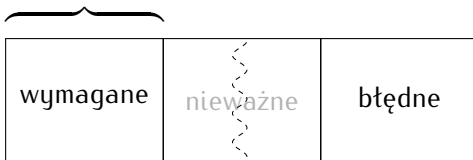
Wiemy, co program **musi** produkować (S_M^0),
co program **może** produkować (S_M).

Reszta nieważna.

Df.: Specyfikacja przybliżona – para
specyfikacja dla pełności + specyfikacja dla poprawności.

Specyfikacje przybliżone

specyfikacja dla pełności



specyfikacja dla poprawności

Uwaga o typach

W odpowiedniej logice z typami
przybliżone specyfikacje niepotrzebne
w **niektórych** przypadkach.

Np. gdy drugi argument $member(t, s)$ musi być listą.

Propozycje LP z typami istnieją, ale nie są popularne.

Prolog – język programowania bez typów.

Lokalizacja błędów w programach

algorithmic debugging, declarative diagnosis

Algorytmy lokalizowania w programie
przyczyny niepoprawności/niepełności
wykrytej przez test

Zadają zapytania o zgodność atomu ze specyfikacją

☹ Nie stosowane
bo wymagano od programisty znajomości specyfikacji dokładnej

Np. pytanie o $member(a, [a|b])$ – jak odpowiedzieć?

😊 Rozwiązanie problemu: specyfikacje przybliżone
w lokalizacji niepoprawności – specyfikacja dla poprawności
w lokalizacji niepełności – specyfikacja dla pełności

Lokalizacja błędów w programach

algorithmic debugging, declarative diagnosis

Algorytmy lokalizowania w programie
przyczyny niepoprawności/niepełności
wykrytej przez test

Zadają zapytania o zgodność atomu ze specyfikacją

☹ Nie stosowane
bo wymagano od programisty znajomości specyfikacji dokładnej

Np. pytanie o $\text{member}(a, [a|b])$ – jak odpowiedzieć?

😊 Rozwiązanie problemu: specyfikacje przybliżone
w lokalizacji niepoprawności – specyfikacja dla poprawności
w lokalizacji niepełności – specyfikacja dla pełności

SAT-solver, Howe & King

HOWE, J. M. AND KING, A. [A pearl on SAT and SMT solving in Prolog](#).
Theor. Comput. Sci. 435, 43–55, 2012.

Program w Prologu, [22 linie](#) (+7 pustych), 84 słowa, 791 znaków
DPLL + *watched literals* + *unit propagation* (w tym spacje itp)

Przedstawiam jako [program w logice](#) z dodanym sterowaniem. ▶ por.

[Systematyczna konstrukcja](#) ze specyfikacji, jednocześnie z dowodami.

Dowody poprawności, pełności, terminacji, ...
[na poziomie programu w logice.](#)

SAT-solver, Howe & King

Reprezentacja formuł CNF (w postaci koniunktywnej normalnej)

Wartości logiczne – stałe true, false

Literały $x, \neg x$ – pary true- X , false- X

Wartościowanie – podstawienie (np. $\{X/\text{false}, Y/\text{true}\}$)

A więc literał prawdziwy jest postaci $t-t$

Klauzula $(L_1 \vee \dots \vee L_n)$ – lista reprezentacji literałów

Formuła $(C_1 \wedge \dots \wedge C_n)$ – lista reprezentacji klauzul

A więc formuła f spełnialna jeśli

$\exists \theta$ lista $f\theta$ ma w każdym elemencie element postaci $t-t$

SAT-solver, specyfikacja

$[f$ spełnialna jeśli $\exists \theta$ lista $f\theta$ ma w każdym elemencie element postaci $t-t$]

Prawdziwe klauzule

$$L_1 = \{[t_1, \dots, t_j, t-t|s] \in \mathcal{HU} \mid j \geq 0\}$$

$$L_1^0 = \{[t_1-t'_1, \dots, t_n-t'_n] \in \mathcal{HU} \mid n > 0, \exists_i t_i = t'_i\}$$

Prawdziwe formuły

$$L_2 = \{[s_1, \dots, s_n] \in \mathcal{HU} \mid n \geq 0, s_1, \dots, s_n \in L_1\}$$

$$L_2^0 = \{[s_1, \dots, s_n] \in \mathcal{HU} \mid n \geq 0, s_1, \dots, s_n \in L_1^0\},$$

Specyfikacja przybliżona

$$S_1 = \{sat_cl(s) \mid s \in L_1\} \cup \{sat_cnf(u) \mid u \in L_2\}.$$

$$S_1^0 = \{sat_cl(s) \mid s \in L_1^0\} \cup \{sat_cnf(u) \mid u \in L_2^0\}.$$

P poprawny wzgl. S_1 i pełny wzgl. $S_1^0 \Rightarrow$

f spełnialna $\Leftrightarrow sat_cnf(f)$ ma odpowiedź



SAT-solver, pierwszy program

Będzie poprawny i pełny wzgl. S_1 j.w.

$$S_1 = \{ \text{sat_cl}(s) \mid s \in L_1 \} \cup \{ \text{sat_cnf}([s_1, \dots, s_n]) \mid s_1, \dots, s_n \in L_1 \}$$

gdzie $L_1 = \{ [t_1, \dots, t_j, t-t|s] \in \mathcal{HU} \mid j \geq 0 \}$

Zgodnie z warunkiem dla semi-pełności,
by pokryć każdy $\text{sat_cl}([t_1, \dots, t_j, t-t|s])$,

dla $j = 0$: $\text{sat_cl}([Pol-Pol|Pairs])$.

dla $j > 0$: $\text{sat_cl}([H|Pairs]) \leftarrow \text{sat_cl}(Pairs)$.

Warunek dla poprawności również spełniony

Podobnie by pokryć każdy $\text{sat_cnf}([s_1, \dots, s_n])$

dla $n = 0$: $\text{sat_cnf}([])$.

dla $n > 0$: $\text{sat_cnf}([Cl|Cls]) \leftarrow \text{sat_cl}(Cl), \text{sat_cnf}(Cls)$.

SAT-solver, pierwszy program

Będzie poprawny i pełny wzgl. S_1 j.w.

$$S_1 = \{ \text{sat_cl}(s) \mid s \in L_1 \} \cup \{ \text{sat_cnf}([s_1, \dots, s_n]) \mid s_1, \dots, s_n \in L_1 \}$$

$$\text{gdzie } L_1 = \{ [t_1, \dots, t_j, t-t|s] \in \mathcal{HU} \mid j \geq 0 \}$$

Zgodnie z warunkiem dla semi-pełności,

by pokryć każdy $\text{sat_cl}([t_1, \dots, t_j, t-t|s])$,

dla $j = 0$: $\text{sat_cl}([Pol-Pol|Pairs])$.

dla $j > 0$: $\text{sat_cl}([H|Pairs]) \leftarrow \text{sat_cl}(Pairs)$.

Warunek dla poprawności również spełniony

Podobnie by pokryć każdy $\text{sat_cnf}([s_1, \dots, s_n])$

dla $n = 0$: $\text{sat_cnf}([])$.

dla $n > 0$: $\text{sat_cnf}([Cl|Cls]) \leftarrow \text{sat_cl}(Cl), \text{sat_cnf}(Cls)$.

SAT-solver, pierwszy program

Będzie poprawny i pełny wzgl. S_1 j.w.

$$S_1 = \{ \text{sat_cl}(s) \mid s \in L_1 \} \cup \{ \text{sat_cnf}([s_1, \dots, s_n]) \mid s_1, \dots, s_n \in L_1 \}$$

gdzie $L_1 = \{ [t_1, \dots, t_j, t-t|s] \in \mathcal{HU} \mid j \geq 0 \}$

Zgodnie z warunkiem dla semi-pełności,
by pokryć każdy $\text{sat_cl}([t_1, \dots, t_j, t-t|s])$,

dla $j = 0$: $\text{sat_cl}([Pol-Pol|Pairs])$.

dla $j > 0$: $\text{sat_cl}([H|Pairs]) \leftarrow \text{sat_cl}(Pairs)$.

Warunek dla poprawności również spełniony

Podobnie by pokryć każdy $\text{sat_cnf}([s_1, \dots, s_n])$

dla $n = 0$: $\text{sat_cnf}([])$.

dla $n > 0$: $\text{sat_cnf}([Cl|Cls]) \leftarrow \text{sat_cl}(Cl), \text{sat_cnf}(Cls)$.

SAT-solver, pierwszy program

Będzie poprawny i pełny wzgl. S_1 j.w.

$$S_1 = \{ \text{sat_cl}(s) \mid s \in L_1 \} \cup \{ \text{sat_cnf}([s_1, \dots, s_n]) \mid s_1, \dots, s_n \in L_1 \}$$

gdzie $L_1 = \{ [t_1, \dots, t_j, t-t|s] \in \mathcal{HU} \mid j \geq 0 \}$

Zgodnie z warunkiem dla semi-pełności,
by pokryć każdy $\text{sat_cl}([t_1, \dots, t_j, t-t|s])$,

dla $j = 0$: $\text{sat_cl}([Pol-Pol|Pairs])$.

dla $j > 0$: $\text{sat_cl}([H|Pairs]) \leftarrow \text{sat_cl}(Pairs)$.

Warunek dla poprawności również spełniony

Podobnie by pokryć każdy $\text{sat_cnf}([s_1, \dots, s_n])$

dla $n = 0$: $\text{sat_cnf}([])$.

dla $n > 0$: $\text{sat_cnf}([Cl|Cls]) \leftarrow \text{sat_cl}(Cl), \text{sat_cnf}(Cls)$.

SAT-solver, konstrukcja efektywnego programu

Nasz program prosty, ale nieefektywny. Zmienimy procedurę

$$\text{sat_cl}([Pol-Pol|Pairs]).$$

$$\text{sat_cl}([H|Pairs]) \leftarrow \text{sat_cl}(Pairs).$$

by można dodać sprytne sterowanie.

Klauzule nieunarne – oddzielnie; nowe procedury sat_cla , sat_clb .

$$S_{11} = \{ \text{sat_cla}(s), \text{sat_clb}(s) \mid \text{sat_cl}(s) \in S_1 \}$$

$$S_{11}^0 = \{ \text{sat_cla}(s), \text{sat_clb}(s) \mid \text{sat_cl}(s) \in S_1^0, |s| > 1 \}$$

$$\text{sat_cl}([Pol-Pol]).$$

$$\text{sat_cl}([Pol1-Var1, Pol2-Var2|Pairs]) \leftarrow$$

$$\text{sat_cla}([Pol1-Var1, Pol2-Var2|Pairs]).$$

► por

SAT-solver, konstrukcja efektywnego programu

Nasz program prosty, ale nieefektywny. Zmienimy procedurę

$$\begin{aligned} & \text{sat_cl}([Pol-Pol|Pairs]). \\ & \text{sat_cl}([H|Pairs]) \leftarrow \text{sat_cl}(Pairs). \end{aligned}$$

by można dodać sprytne sterowanie.

Klauzule nieunarne – oddzielnie; nowe procedury *sat_cla*, *sat_clb*.

$$S_{11} = \{ \text{sat_cla}(s), \text{sat_clb}(s) \mid \text{sat_cl}(s) \in S_1 \}$$

$$S_{11}^0 = \{ \text{sat_cla}(s), \text{sat_clb}(s) \mid \text{sat_cl}(s) \in S_1^0, |s| > 1 \}$$

► por.

$$\text{sat_cl}([Pol-Pol]).$$

$$\begin{aligned} & \text{sat_cl}([Pol1-Var1, Pol2-Var2|Pairs]) \leftarrow \\ & \text{sat_cla}([Pol1-Var1, Pol2-Var2|Pairs]). \end{aligned}$$

Sterowanie:

użyć *sat_cla*(...) tylko gdy *Var1* lub *Var2* nie jest zmienną;
to implementuje *watched literals*;

SAT-solver, konstrukcja efektywnego programu

Nasz program prosty, ale nieefektywny. Zmienimy procedurę

```
sat_cl([Pol-Pol|Pairs]).
sat_cl([H|Pairs]) ← sat_cl(Pairs).
```

by można dodać sprytne sterowanie.

Klauzule nieunarne – oddzielnie; nowe procedury *sat_cla*, *sat_clb*.

$$S_{11} = \{ \text{sat_cla}(s), \text{sat_clb}(s) \mid \text{sat_cl}(s) \in S_1 \}$$

$$S_{11}^0 = \{ \text{sat_cla}(s), \text{sat_clb}(s) \mid \text{sat_cl}(s) \in S_1^0, |s| > 1 \}$$

► por.

```
sat_cl([Pol-Pol]).
```

```
sat_cl([Pol1-Var1, Pol2-Var2|Pairs]) ←
```

```
sat_cla([Pol1-Var1, Pol2-Var2|Pairs]).
```

Sterowanie:

użyć *sat_cla*(...) tylko gdy *Var1* lub *Var2* nie jest zmienną;
to implementuje *watched literals*;

SAT-solver, konstrukcja efektywnego programu

Klauzule nieunarne – oddzielnie; nowe procedury *sat_cla*, *sat_clb*.

$$S_{11} = \{ sat_cla(s), sat_clb(s) \mid sat_cl(s) \in S_1 \}$$

$$S_{11}^0 = \{ sat_cla(s), sat_clb(s) \mid sat_cl(s) \in S_1^0, |s| > 1 \}$$

▶ por.

sat_cl([*Pol*-*Pol*]).

sat_cl([*Pol1*-*Var1*, *Pol2*-*Var2*|*Pairs*]) ←

sat_cla([*Pol1*-*Var1*, *Pol2*-*Var2*|*Pairs*]).

Sterowanie:

użyć *sat_cla*(...) tylko gdy *Var1* lub *Var2* nie jest zmienną;
to implementuje *watched literals*;

SAT-solver, konstrukcja efektywnego programu

Klauzule nieunarne – oddzielnie; nowe procedury *sat_cla*, *sat_clb*.

$$S_{11} = \{ sat_cla(s), sat_clb(s) \mid sat_cl(s) \in S_1 \}$$

$$S_{11}^0 = \{ sat_cla(s), sat_clb(s) \mid sat_cl(s) \in S_1^0, |s| > 1 \}$$

▶ por.

sat_cl([*Pol-Pol*]).

sat_cl([*Pol1-Var1*, *Pol2-Var2*|*Pairs*]) ←

sat_cla([*Pol1-Var1*, *Pol2-Var2*|*Pairs*]).

Sterowanie:

użyć *sat_cla*(...) tylko gdy *Var1* lub *Var2* nie jest zmienną;
to implementuje *watched literals*;

mieć literał bez zmiennej na początku listy

sat_cla([*P1-V1*, *P2-V2*|*Pairs*]) ← *sat_clb*([*P1-V1*, *P2-V2*|*Pairs*])

sat_cla([*P1-V1*, *P2-V2*|*Pairs*]) ← *sat_clb*([*P2-V2*, *P1-V1*|*Pairs*])

sat_clb([*Pol-Pol*|_]).

sat_clb([_|*Pairs*]) ← *sat_cl*(*Pairs*).

SAT-solver, konstrukcja efektywnego programu

Klauzule nieunarne – oddzielnie; nowe procedury *sat_cla*, *sat_clb*.

$$S_{11} = \{ \text{sat_cla}(s), \text{sat_clb}(s) \mid \text{sat_cl}(s) \in S_1 \}$$

$$S_{11}^0 = \{ \text{sat_cla}(s), \text{sat_clb}(s) \mid \text{sat_cl}(s) \in S_1^0, |s| > 1 \}$$

▶ por.

sat_cl([*Pol-Pol*]).

sat_cl([*Pol1-Var1*, *Pol2-Var2*|*Pairs*]) ←

sat_cla([*Pol1-Var1*, *Pol2-Var2*|*Pairs*]).

Sterowanie:

użyć *sat_cla*(...) tylko gdy *Var1* lub *Var2* nie jest zmienną;
to implementuje *watched literals*;

mieć literał bez zmiennej na początku listy

sat_cla([*P1-V1*, *P2-V2*|*Pairs*]) ← *sat_clb*([*P1-V1*, *P2-V2*|*Pairs*])

sat_cla([*P1-V1*, *P2-V2*|*Pairs*]) ← *sat_clb*([*P2-V2*, *P1-V1*|*Pairs*])

sat_clb([*Pol-Pol*|_]).

sat_clb([_|*Pairs*]) ← *sat_cl*(*Pairs*).

SAT-solver, konstrukcja efektywnego programu

Klauzule nieunarne – oddzielnie; nowe procedury *sat_cla*, *sat_clb*.

$$S_{11} = \{ sat_cla(s), sat_clb(s) \mid sat_cl(s) \in S_1 \}$$

$$S_{11}^0 = \{ sat_cla(s), sat_clb(s) \mid sat_cl(s) \in S_1^0, |s| > 1 \}$$

▶ por.

sat_cl([*Pol-Pol*]).

sat_cl([*Pol1-Var1, Pol2-Var2*|*Pairs*]) ←

sat_cla([*Pol1-Var1, Pol2-Var2*|*Pairs*]).

mieć literał bez zmiennej na początku listy

sat_cla([*P1-V1, P2-V2*|*Pairs*]) ← *sat_clb*([*P1-V1, P2-V2*|*Pairs*])

sat_cla([*P1-V1, P2-V2*|*Pairs*]) ← *sat_clb*([*P2-V2, P1-V1*|*Pairs*])

sat_clb([*Pol-Pol*|_]).

sat_clb([_|*Pairs*]) ← *sat_cl*(*Pairs*).

Sterowanie wybierze właściwą implikację dla *sat_cla*

(która wywoła *sat_clb* bez zmiennej w głowie listy);

porzuci drugą implikację dla *sat_clb*, jeśli pierwsza skutkuje.

Uwagi. SAT-solver, konstrukcja

Konstrukcja programu niejako sterowana
warunkiem dostatecznym na semi-pełność wzgl. $S_1^0 \cup S_{11}^0$.

Wraz z konstrukcją – dowody semi-pełności i poprawności
wzgl. $S_1^0 \cup S_{11}^0$ i $S_1 \cup S_{11}$, niewidoczne na slajdach.

Właściwy program nieco bardziej rozbudowany.

Zamiast jednego $[P2-V2, P1-V1|Pairs]$ – pięć argumentów
by wykorzystać pewne optymalizacje kompilatorów (tzw. indeksowanie).

Dodatkowy fragment, by obliczenia nie ugrzęzły z braku klauzuli unarnej.

Implementuje DPLL z *watched literals* i *unit propagation*
(propagacja tylko na *watched literals*)

Uwagi. SAT-solver, konstrukcja

Konstrukcja programu niejako sterowana
warunkiem dostatecznym na semi-pełność wzgl. $S_1^0 \cup S_{11}^0$.

Wraz z konstrukcją – dowody semi-pełności i poprawności
wzgl. $S_1^0 \cup S_{11}^0$ i $S_1 \cup S_{11}$, niewidoczne na slajdach.

Właściwy program nieco bardziej rozbudowany.

Zamiast jednego $[P2-V2, P1-V1|Pairs]$ – pięć argumentów
by wykorzystać pewne optymalizacje kompilatorów (tzw. indeksowanie).

Dodatkowy fragment, by obliczenia nie ugrzęzły z braku klauzuli unarnej.

Implementuje DPLL z *watched literals* i *unit propagation*
(propagacja tylko na *watched literals*)

Uwagi. SAT-solver, konstrukcja

W artykule przedstawiłem pełną **konstrukcję** tego programu, jako program w logice + dodane sterowanie.

Dowody poprawności, pełności, terminacji, ...
na poziomie programu w logice, w pełni deklaratywne.

Dodanie sterowania zachowuje poprawność i terminację
nie zachowuje pełności

bowiem obcinamy część przestrzeni poszukiwań (SLD-tree).


Przedstawiłem warunek konieczny na zachowanie pełności.
Zastosowany w tym przypadku.

W trakcie konstrukcji **zmienia się semantyka** programu

Np. jedynie pierwszy program ma odpowiedź `sat_cl([true-true|1])`.

A więc *“semantic-preserving program transformations”* nieadekwatne.


Podsumowanie

 Główna część rozumowania o programach
możliwa na poziomie **deklaratywnym**/logicznym 😊
w **abstrakcji** od semantyki operacyjnej.

Gdyby tak nie było, to LP nie zasługiwałoby na uznanie
za deklaratywny paradygmat programowania.

A często proponuje się metody zw. z semantyką operacyjną.

Podsumowanie

 Główna część rozumowania o programach
możliwa na poziomie **deklaratywnym**/logicznym 😊
w **abstrakcji** od semantyki operacyjnej.

Gdyby tak nie było, to LP nie zasługiwałoby na uznanie
za deklaracyjny paradygmat programowania.


A często proponuje się metody zw. z semantyką operacyjną.

Uwaga: Przy dowodzie pełności korzystamy z terminacji 😞
(pojęcia operacyjnego).

Tu był przypadek szczególny: deklaracyjny dowód terminacji.



Ale w praktyce terminacja i tak potrzebna.

Podsumowanie

 Główna część rozumowania o programach
możliwa na poziomie **deklaratywnym**/logicznym 😊
w **abstrakcji** od semantyki operacyjnej.

Gdyby tak nie było, to LP nie zasługiwałoby na uznanie
za deklaracyjny paradygmat programowania.

A często proponuje się metody zw. z semantyką operacyjną.

-  Metoda dla semi-pełności \rightsquigarrow sposób konstruowania programów
wraz z dowodami poprawności, semi-pełności
-  Uważam, że przedstawione metody nadają się do stosowania w praktyce
(pewnie na poziomie nieformalnym).

Tematy na c.d.

Wnioskowanie o programach z negacją (może też CLP, CHR, ASP)

Implementacja lokalizacji błędów w programach.

Nauczanie. Dalsze przykłady.

Formalizacja specyfikacji.

Formalizacja dowodów (sprawdzanie, wspomaganie konstruowania,...)

