Sterowanie w programowaniu w logice

- kolejność atomów w treści klauzuli (wybór atomów)
- kolejność klauzul w programie (wybór klauzul)
- mechanizmy sterowania (odcięcie)

Sterowanie w Prologu

$*$ reguła obliczeniowa: wybór skrajnie lewego atomu (LD-rezolucja, LDNF-rezolucja)

$*$ uporządkowanie klauzul (tekst programu)

$*$ przeszukiwanie w głąb

Znaczenie kolejności atomów w treści klauzuli, przykład

Program *slowsort*

- $sort(x, y) \leftarrow perm(x, y), sorted(y)$

generator permutacji, poszukiwanie właściwej
(czyli najprostsza wersja techniki *generuj i testuj*,
inaczej: wykonywalna specyfikacja)

- $sort(x, y) \leftarrow sorted(y), perm(x, y)$

nieskończony(!) generator posortowanych ciągów

Znaczenie kolejności klauzul — przykład

- Program $Member1$

$$member(x, y.z) \leftarrow member(x, z)$$
$$member(x, x.y) \leftarrow$$

$*$ zapytanie: $\leftarrow member(x, 1.2.3.4.5.6.7.nil)$

obliczenie: przejście do końca listy, pierwsza odpowiedź – ostatni element listy

$*$ zapytanie: $\leftarrow member(x, y)$

obliczenie: nieskończone, czyli brak jakiejkolwiek (!) odpowiedzi

- Program $Member2$

$$member(x, x.y) \leftarrow$$
$$member(x, y.z) \leftarrow member(x, z)$$

$*$ zapytanie: $\leftarrow member(x, 1.2.3.4.5.6.7.nil)$

obliczenie: pierwsza odpowiedź – pierwszy element listy, itd.

$*$ zapytanie: $\leftarrow member(x, y)$

obliczenie: nieskończenie wiele odpowiedzi, postaci:

$$y/x.y_1, \quad y/x_1.x.y_2, \quad y/x_1.x_2.x.y_3, \ \ldots$$

Odcięcie

narzędzie do sterowania
(bardzo kontrowersyjne lecz szeroko stosowane)

założenie: reguła obliczeniowa — wybór lewego atomu (Prolog)

odcięcie — **!**

– pozalogiczna anotacja programu (sterowanie)

– atom, który zawsze skutkuje (stała $true$)

semantyka programów z odcięciem nie zmienia się

znaczenie odcięcia:
zmniejszenie przestrzeni przeszukiwań

SLD-drzewo, rezolwenta, w której wybrany atom spełnia:
klauzula wybranego predykatu zawiera odcięcie

$\Longrightarrow$ pominięcie części poddrzewa, której korzeniem jest ta rezolwenta

Rodzaje odcięć:

– bezpieczne — pominięte poddrzewo nie zawiera gałęzi sukcesu

– niebezpieczne — istnieje gałąź sukcesu, czyli utrata pełności (SLD-rezolucja), a nawet poprawności (SLDNF-rezolucja)

(ang. green cut, red cut)

Przykład

$A \leftarrow B, C$

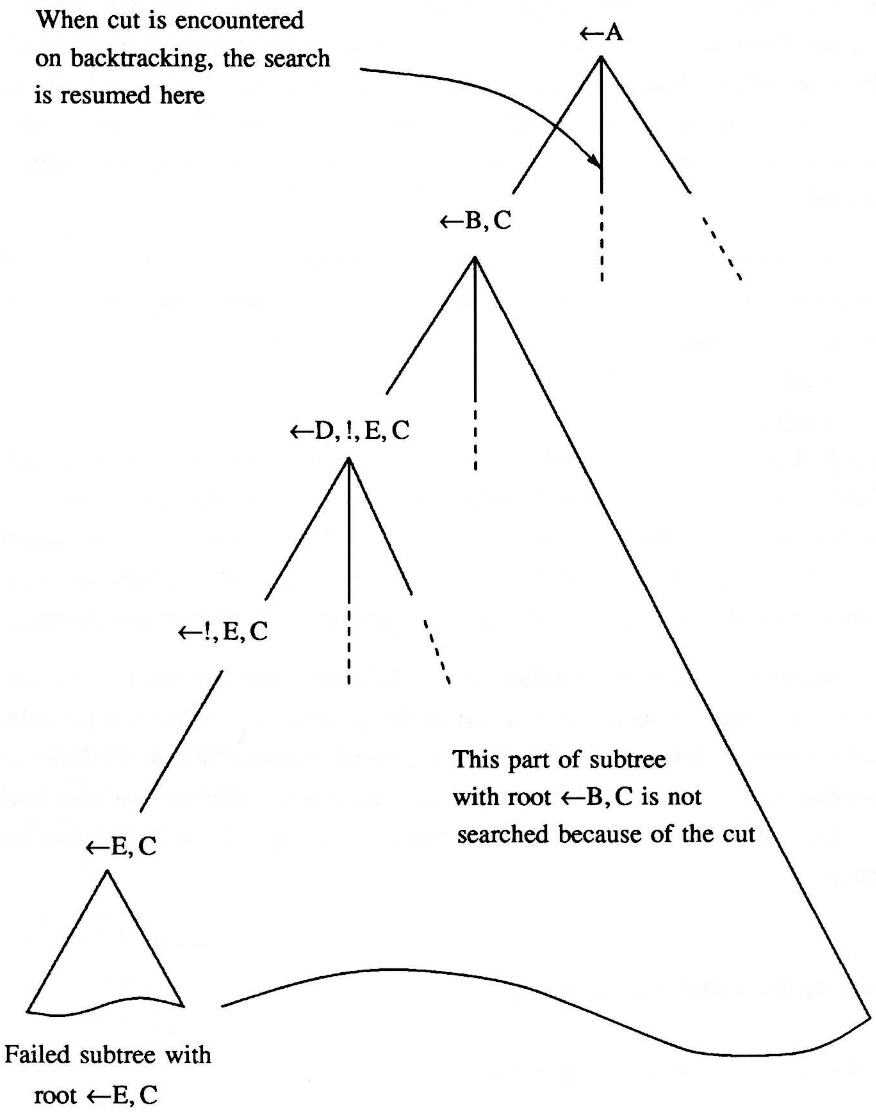
....

$B \leftarrow D, !, E$

....

$D \leftarrow$

....



(Rysunek pochodzi z książki Johna Lloyda ,,Foundations of Logic Programming".)

Bezpieczne odcięcie — wzrost sprawności

Przykład

$$
\begin{aligned}
&partition(nil, x, nil, nil) \leftarrow\, !\\
&partition(x.l, y, x.l1, l2) \leftarrow x \leq y, !, partition(l, y, l1, l2)\\
&partition(x.l, y, l1, x.l2) \leftarrow x > y, partition(l, y, l1, l2)
\end{aligned}
$$

Niebezpieczne odcięcie — utrata pełności
(pominięcie odpowiedzi lub odpowiedź ,,Nie")
co gorsza: programy niepoprawne deklaratywnie

Przykład

$$
\begin{aligned}
&max(x, y, y) \leftarrow x \leq y, !\\
&max(x, y, x) \leftarrow
\end{aligned}
$$

deklaratywnie — nie jest to relacja max
proceduralnie — uwzględniając strategię przeszukiwania, regułę obliczeniową i porządek klauzul: (prawie) max

czy prawdą jest, że $\leftarrow$ max(3, 5, 3) ?
odpowiedź: TAK!

Postępowanie w takich sytuacjach:

— inne mechanizmy (np. konstrukcja if-then-else)

— przeniesienie uzgadniania z nagłówka do treści klauzuli

(np. max(x, y, z) $\leftarrow$ x $\leq$ y, !, z = y)

Zasady korzystania z odcięcia:
– nie używać o ile nie jest to konieczne
– stosować rozsądnie (!)
– dodawać po napisaniu programu
– umieszczać **bezpośrednio** po zatwierdzeniu wyboru klauzuli

Odcięcie w programach z negacją

program (bez negacji) — możliwość utraty pełności
program z negacją — możliwość utraty poprawności (!)

Przykład

$$\begin{aligned}
&subset(x, y) \leftarrow \neg p(x, y)\\
&p(x, y) \leftarrow member(z, x), \neg member(z, y)\\
&member(x, x.y) \leftarrow\ !\\
&member(x, y.z) \leftarrow member(x, z)
\end{aligned}$$

G: $\leftarrow subset(1.2.3.nil, 1.nil)$

SLDNF-dowód (skrócony):

$G_0 = G$

$G_1 = \leftarrow \neg p(1.2.3.nil, 1.nil)$

$G1_0 = \leftarrow p(1.2.3.nil, 1.nil)$

$G1_1 = \leftarrow member(z, 1.2.3.nil), \neg member(z, 1.nil)$

$G1_2 = \leftarrow \neg member(1, 1.nil)$

$G2_0 = \leftarrow member(1, 1.nil)$

$G2_1 = \Box$

porażka

$G_2 = \Box$

czyli ,,udowodniliśmy", że: 1.2.3.nil jest podzbiorem 1.nil!

przyczyna błędu: niepoprawne skonstruowanie drzewa porażki dla zapytania $\leftarrow p(1.2.3.nil, 1.nil)$