

Algoritmy pro CSP (pokračování)

Prohledávání + konzistence

- Splňování podmínek **prohledáváním** prostoru řešení
 - podmínky jsou užívány pasivně jako test
 - přiřazuji hodnoty proměnných a zkouším co se stane
 - vestavěný prohledávací algoritmus Prologu: **backtracking**, triviální: **generuj & testuj**

Prohledávání + konzistence

- Splňování podmínek **prohledáváním** prostoru řešení
 - podmínky jsou užívány pasivně jako test
 - přiřazuji hodnoty proměnných a zkouším co se stane
 - vestavěný prohledávací algoritmus Prologu: **backtracking**, triviální: **generuj & testuj**
 - úplná metoda (nalezneme řešení nebo dokážeme jeho neexistenci)
 - zbytečně pomalé (exponenciální): procházím i „evidentně“ špatná ohodnocení

Prohledávání + konzistence

- Splňování podmínek **prohledáváním** prostoru řešení
 - podmínky jsou užívány pasivně jako test
 - přiřazují hodnoty proměnných a zkouším co se stane
 - vestavěný prohledávací algoritmus Prologu: **backtracking**, triviální: **generuj & testuj**
 - úplná metoda (nalezneme řešení nebo dokážeme jeho neexistenci)
 - zbytečně pomalé (exponenciální): procházím i „evidentně“ špatná ohodnocení
- **Konzistenční (propagační) techniky**
 - umožňují odstranění nekonzistentních hodnot z domény proměnných
 - neúplná metoda (v doméně zůstanou ještě nekonzistentní hodnoty)
 - relativně rychlé (polynomiální)

Prohledávání + konzistence

● Splňování podmínek **prohledáváním** prostoru řešení

- podmínky jsou užívány pasivně jako test
- přiřazují hodnoty proměnných a zkouším co se stane
- vestavěný prohledávací algoritmus Prologu: **backtracking**, triviální: **generuj & testuj**
- úplná metoda (nalezneme řešení nebo dokážeme jeho neexistenci)
- zbytečně pomalé (exponenciální): procházím i „evidentně“ špatná ohodnocení

● **Konzistenční (propagační) techniky**

- umožňují odstranění nekonzistentních hodnot z domény proměnných
- neúplná metoda (v doméně zůstanou ještě nekonzistentní hodnoty)
- relativně rychlé (polynomiální)

● Používá se **kombinace obou metod**

- postupné přiřazování hodnot proměnným
- po přiřazení hodnoty odstranění nekonzistentních hodnot konzistenčními technikami

Prohledávání do hloubky

- Základní prohledávací algoritmus pro problémy splňování podmínek
- **Prohledávání stavového prostoru do hloubky (*depth first search*)**
- Dvě fáze prohledávání s navracením
 - **dopředná fáze**: proměnné jsou postupně vybírány, rozšiřuje se částečné řešení přiřazením konzistentní hodnoty (pokud existuje) další proměnné
 - po vybrání hodnoty testujeme konzistenci
 - **zpětná fáze**: pokud neexistuje konzistentní hodnota pro aktuální proměnnou, algoritmus se vrací k předchozí přiřazené hodnotě

Prohledávání do hloubky

- Základní prohledávací algoritmus pro problémy splňování podmínek
- **Prohledávání stavového prostoru do hloubky (*depth first search*)**
- Dvě fáze prohledávání s navracením
 - **dopředná fáze**: proměnné jsou postupně vybírány, rozšiřuje se částečné řešení přiřazením konzistentní hodnoty (pokud existuje) další proměnné
 - po vybrání hodnoty testujeme konzistenci
 - **zpětná fáze**: pokud neexistuje konzistentní hodnota pro aktuální proměnnou, algoritmus se vrací k předchozí přiřazené hodnotě
- Proměnné dělíme na
 - **minulé** – proměnné, které už byly vybrány (a mají přiřazenu hodnotu)
 - **aktuální** – proměnná, která je právě vybrána a je jí přiřazována hodnota
 - **budoucí** – proměnné, které budou vybrány v budoucnosti

Základní algoritmus prohledávání do hloubky

- Pro jednoduchost proměnné očíslováme a ohodnocujeme je v daném pořadí
- Na začátku voláno jako `Labeling(G, 1)`

```
procedure labeling(G, a)
  if a > |uzly(G)| then return uzly(G)
  for  $\forall x \in D_a$  do
    if consistent(G, a) then % consistent(G, a) je nahrazeno FC(G, a), LA(G, a), .
      R := labeling(G, a + 1)
      if R  $\neq$  fail then return R
  return fail
end labeling
```

Po přiřazení všech proměnných vrátíme jejich ohodnocení

- Procedury `consistent` uvedeme pouze pro binární podmínky

Backtracking (BT)

- Backtracking ověřuje v každém kroku konzistenci podmínek vedoucích z minulých proměnných do aktuální proměnné
- Backtracking tedy zajišťuje konzistenci podmínek
 - na všech minulých proměnných
 - na podmínkách mezi minulými proměnnými a aktuální proměnnou

Backtracking (BT)

- Backtracking ověřuje v každém kroku konzistenci podmínek vedoucích z minulých proměnných do aktuální proměnné
- Backtracking tedy zajišťuje konzistenci podmínek
 - na všech minulých proměnných
 - na podmínkách mezi minulými proměnnými a aktuální proměnnou

● procedure $BT(G, a)$

$Q := \{(V_i, V_a) \in \text{hrany}(G), i < a\}$ % hrany vedoucí z minulých proměnných do aktuální

$\text{Consistent} := \text{true}$

while Q není prázdná \wedge Consistent do

 vyber a smaž libovolnou hranu (V_k, V_m) z Q

$\text{Consistent} := \text{not revise}(V_k, V_m)$ % pokud vyřadíme prvek, bude doména prázdná

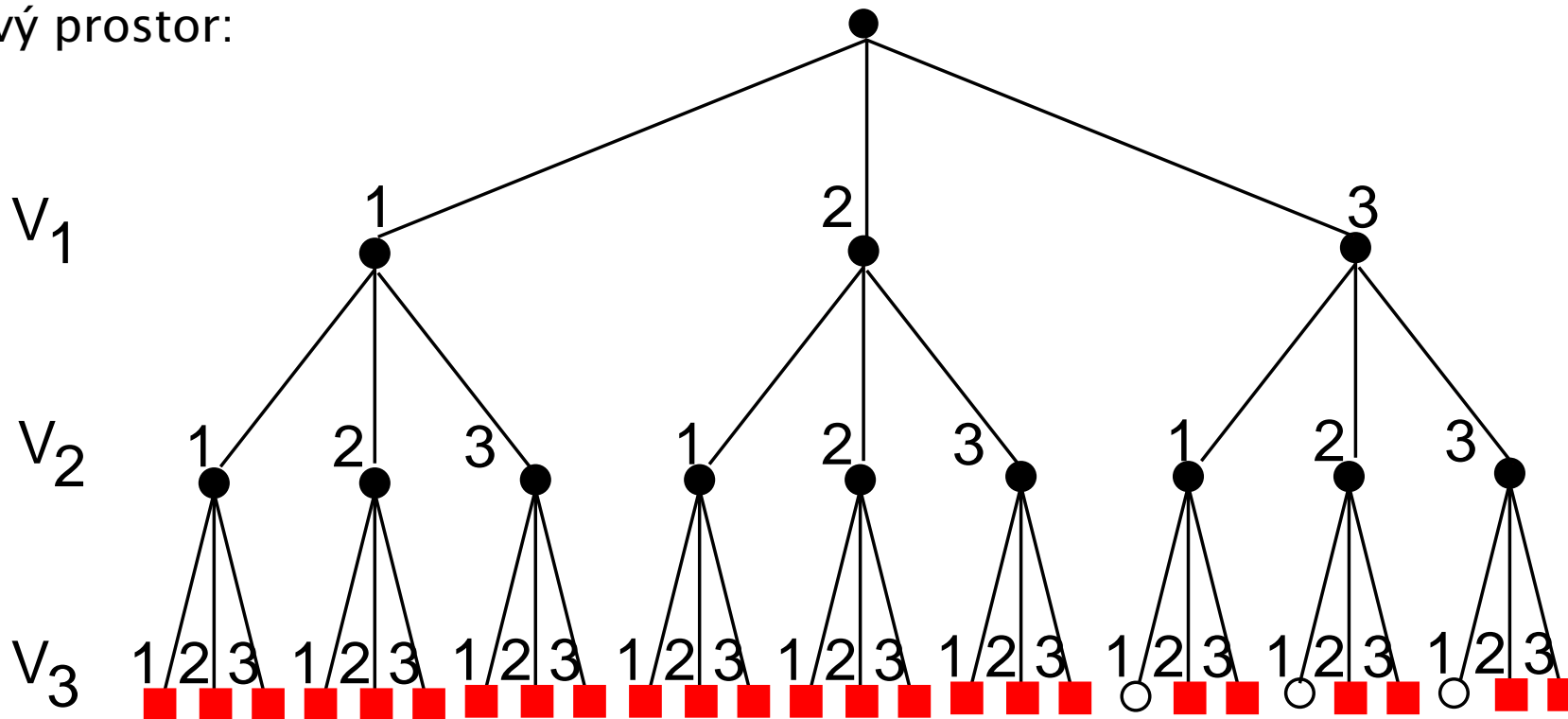
return Consistent

end BT

Příklad: backtracking

● Omezení: V_1, V_2, V_3 in $1 \dots 3$, $V_1 \neq 3 \times V_3$

● Stavový prostor:



- červené čtverečky: chybný pokus o instanciaci, řešení neexistuje
- nevyplněná kolečka: nalezeno řešení
- černá kolečka: vnitřní uzel, máme pouze částečné přiřazení

Kontrola dopředu (*FC – forward checking*)

- FC je rozšíření backtrackingu
- FC navíc zajišťuje konzistenci mezi aktuální proměnnou a budoucími proměnnými, které jsou s ní spojeny dosud nesplněnými podmínkami

Kontrola dopředu (*FC – forward checking*)

- FC je rozšíření backtrackingu
- FC navíc zajišťuje konzistenci mezi aktuální proměnnou a budoucími proměnnými, které jsou s ní spojeny dosud nesplněnými podmínkami

● procedure `FC(G, a)`

`Q := {(Vi, Va) ∈ hrany(G), i > a}` % přidání hran z budoucích do aktuální proměnné

`Consistent := true`

`while Q není prázdná ∧ Consistent do`

`vyber a smaž libovolnou hranu (Vk, Vm) z Q`

`if revise((Vk, Vm)) then`

`Consistent := (|Dk| > 0)` % vyprázdnění domény znamená nekonzistenci

`return Consistent`

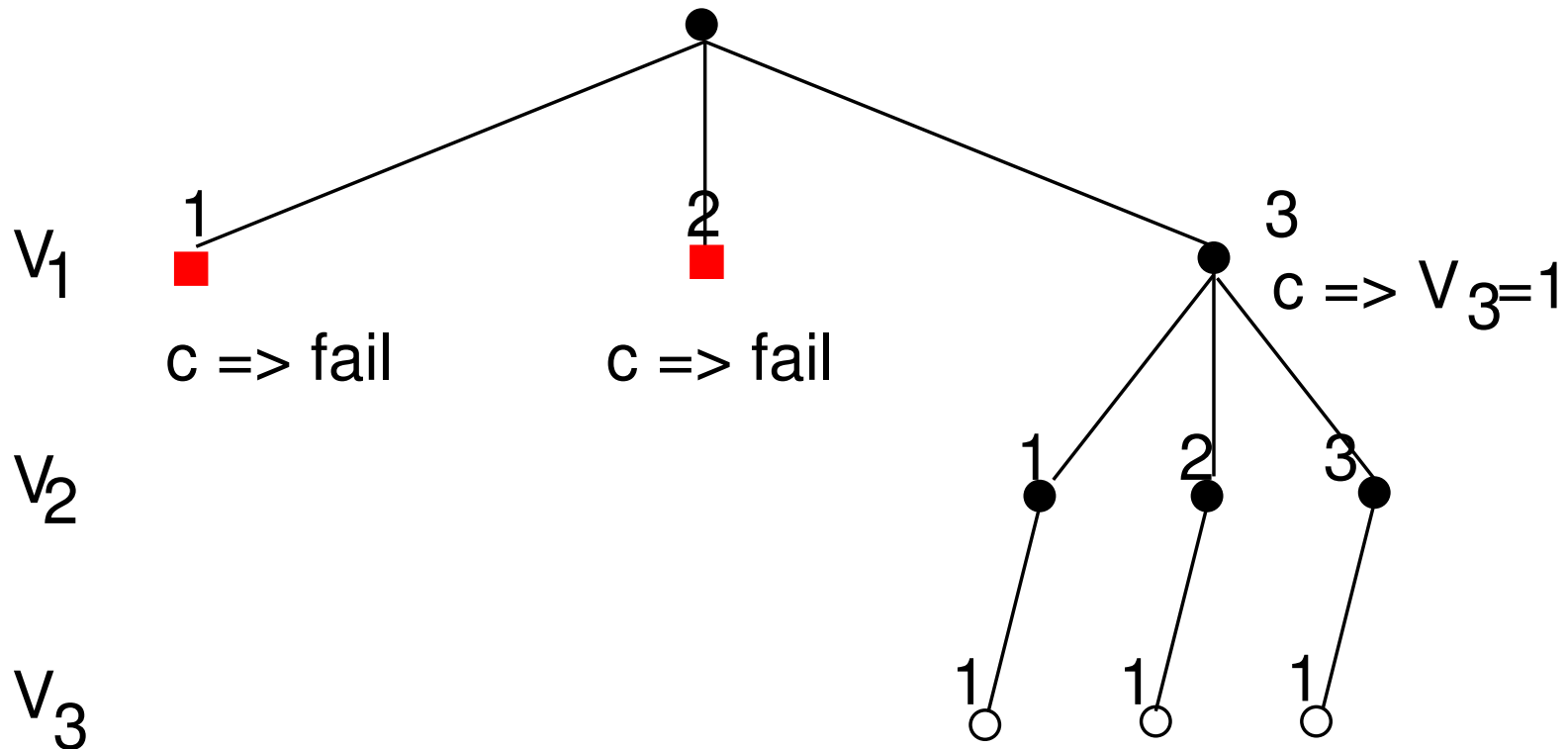
`end FC`

- Hrany z minulých proměnných do aktuální proměnné není nutno testovat

Příklad: kontrola dopředu

● Omezení: V_1, V_2, V_3 in $1 \dots 3$, $c : V_1 \# = 3 \times V_3$

● Stavový prostor:



Pohled dopředu (*LA – looking ahead*)

● LA je rozšíření FC, navíc ověřuje konzistenci hran mezi budoucími proměnnými

● procedure $LA(G, a)$

$Q := \{(V_i, V_a) \in \text{hrany}(G), i > a\}$ % začínáme s hranami do a

Consistent := true

while Q není prázdná \wedge Consistent do

 vyber a smaž libovolnou hranu (V_k, V_m) z Q

 if $\text{revise}(V_k, V_m)$ then

$Q := Q \cup \{(V_i, V_k) \mid (V_i, V_k) \in \text{hrany}(G), i \neq k, i \neq m, i > a\}$

 Consistent := $(|D_k| > 0)$

return Consistent

end LA

Pohled dopředu (*LA – looking ahead*)

- LA je rozšíření FC, navíc ověřuje konzistenci hran mezi budoucími proměnnými

- procedure $LA(G, a)$

$Q := \{(V_i, V_a) \in \text{hrany}(G), i > a\}$ % začínáme s hranami do a

Consistent := true

while Q není prázdná \wedge Consistent do

 vyber a smaž libovolnou hranu (V_k, V_m) z Q

 if $\text{revise}((V_k, V_m))$ then

$Q := Q \cup \{(V_i, V_k) \mid (V_i, V_k) \in \text{hrany}(G), i \neq k, i \neq m, i > a\}$

 Consistent := $(|D_k| > 0)$

return Consistent

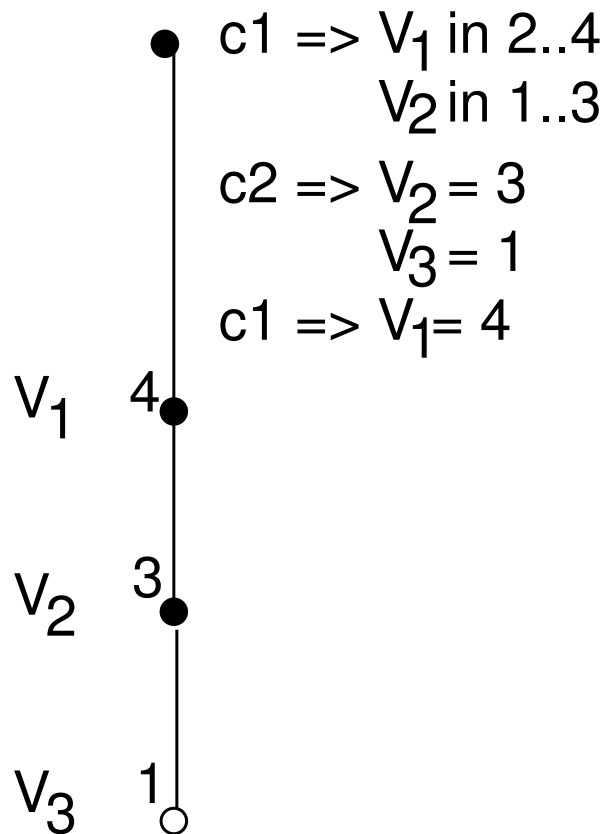
end LA

- Hrany z minulých proměnných do aktuální proměnné opět netestujeme
- Tato LA procedura je založena na AC-3, lze použít i jiné AC algoritmy
- **LA udržuje hranovou konzistenci:** protože ale $LA(G, a)$ používá AC-3, musíme **zajistit iniciální konzistenci** pomocí AC-3 ještě před startem prohledávání

Příklad: pohled dopředu (pomocí AC-3)

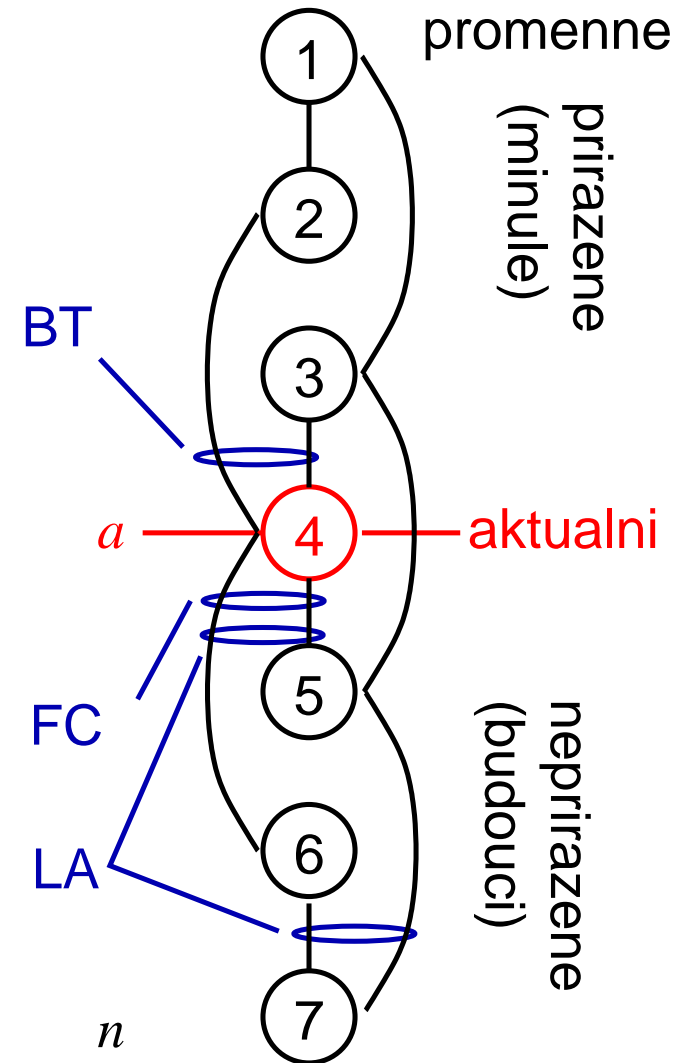
- Omezení: $V_1, V_2, V_3 \in 1 \dots 4$, $c1 : V_1\# > V_2$, $c2 : V_2\# = 3 \times V_3$
- Stavový prostor

(spouští se iniciální konzistence se před startem prohledávání)



Přehled algoritmů

- **Backtracking (BT)** kontroluje v kroku a podmínky $c(V_1, V_a), \dots, c(V_{a-1}, V_a)$ z minulých proměnných do aktuální proměnné



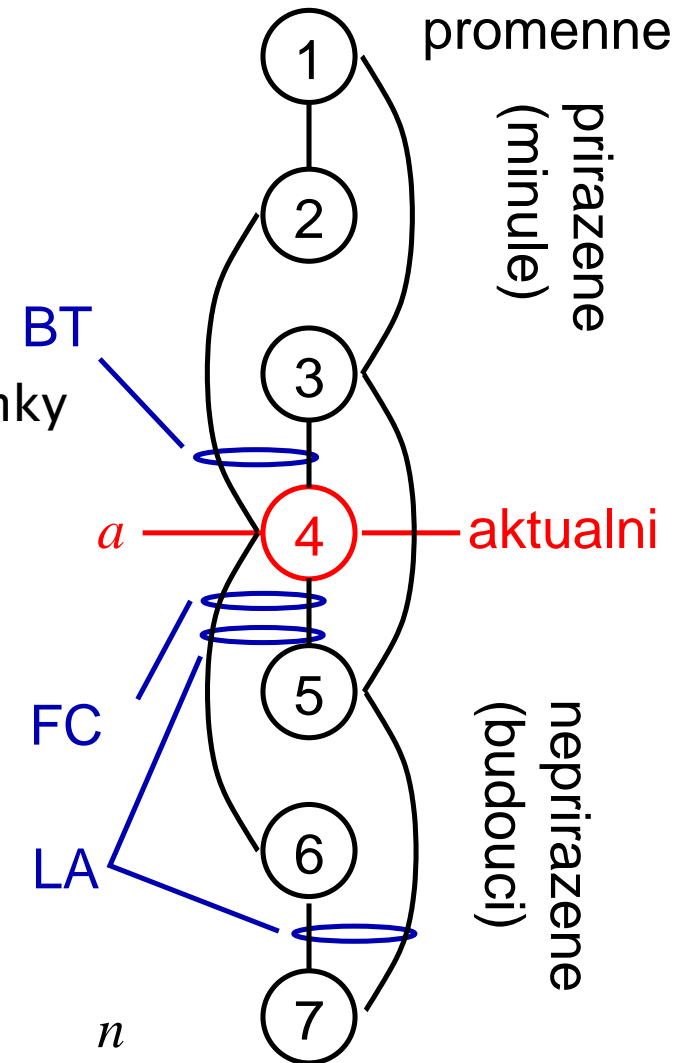
Přehled algoritmů

- **Backtracking (BT)** kontroluje v kroku a podmínky $c(V_1, V_a), \dots, c(V_{a-1}, V_a)$

z minulých proměnných do aktuální proměnné

- **Kontrola dopředu (FC)** kontroluje v kroku a podmínky $c(V_{a+1}, V_a), \dots, c(V_n, V_a)$

z budoucích proměnných do aktuální proměnné

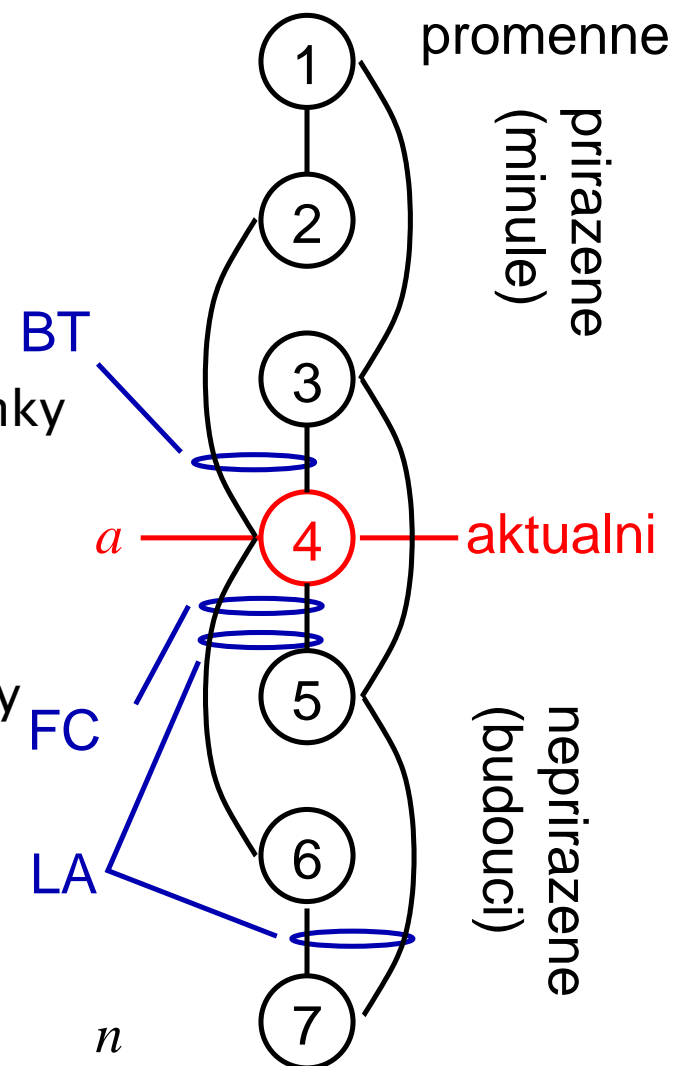


Přehled algoritmů

- **Backtracking (BT)** kontroluje v kroku a podmínky $c(V_1, V_a), \dots, c(V_{a-1}, V_a)$ z minulých proměnných do aktuální proměnné

- **Kontrola dopředu (FC)** kontroluje v kroku a podmínky $c(V_{a+1}, V_a), \dots, c(V_n, V_a)$ z budoucích proměnných do aktuální proměnné

- **Pohled dopředu (LA)** kontroluje v kroku a podmínky $\forall l(a \leq l \leq n), \forall k(a \leq k \leq n), k \neq l: c(V_k, V_l)$ z budoucích proměnných do aktuální proměnné a mezi budoucími proměnnými



Cvičení

1. Jak vypadá stavový prostor řešení pro následující omezení

$A \in 1..4, B \in 3..4, C \in 3..4, B \neq C, A \neq C$

při použití kontroly dopředu a uspořádání proměnných A,B,C? Popište, jaký typ propagace proběhne v jednotlivých uzlech.

2. Jak vypadá stavový prostor řešení pro následující omezení

$A \in 1..4, B \in 3..4, C \in 3..4, B \neq C, A \neq C$

při použití pohledu dopředu a uspořádání proměnných A,B,C? Popište, jaký typ propagace proběhne v jednotlivých uzlech.

3. Jak vypadá stavový prostor řešení pro následující omezení

$\text{domain}([A,B,C],0,1), A \neq B-1, C \neq A*A$

při použití backtrackingu a pohledu dopředu a uspořádání proměnných A,B,C? Popište, jaký typ propagace proběhne v jednotlivých uzlech.

Cvičení

1. Jaká jsou pravidla pro konzistenci mezí u omezení $X \neq Y + 5$? Jaké typy propagací pak proběhnou v následujícím příkladě při použití konzistence mezí?

$X \text{ in } 1..20, Y \text{ in } 1..20, X \neq Y + 5, Y \neq > 10.$

2. Ukažte, jak je dosaženo hranové konzistence v následujícím příkladu:

$\text{domain}([X,Y,Z], 1, 5), X \neq < Y, Z \neq Y + 1 .$

Implementace Prologu

Literatura:

- Matyska L., Toman D.: Implementační techniky Prologu, Informační systémy, (1990), 21–59.
<http://www.ics.muni.cz/people/matyska/vyuka/1p/1p.html>

Opakování: základní pojmy

- Konečná množina klauzulí **Hlava** :- Tělo tvoří **program P**.
- **Hlava** je literál
- **Tělo** je (eventuálně prázdná) konjunkce literálů $T_1, \dots, T_a, a \geq 0$
- **Literál**
je tvořen m -árním predikátovým symbolem (m/p) a m termy (argumenty)
- **Term** je konstanta, proměnná nebo složený term.
- **Složený term**
s n termy na místě argumentů
- **Dotaz (cíl)** je neprázdná množina literálů.

Interpretace

Deklarativní sémantika:

Hlava platí, platí-li jednotlivé literály těla.

Interpretace

Deklarativní sémantika:

Hlava platí, platí-li jednotlivé literály těla.

Procedurální (imperativní) sémantika:

Entry: Hlava::

```
{  
    call  $T_1$   
    :  
    call  $T_a$   
}
```

Volání procedury s názvem Hlava uspěje, pokud uspěje volání všech procedur (literálů) v těle.

Interpretace

Deklarativní sémantika:

Hlava platí, platí-li jednotlivé literály těla.

Procedurální (imperativní) sémantika:

Entry: Hlava::

```
{  
    call  $T_1$   
    :  
    call  $T_a$   
}
```

Volání procedury s názvem Hlava uspěje, pokud uspěje volání všech procedur (literálů) v těle.

Procedurální sémantika = podklad pro implementaci

Abstraktní interpret

Vstup: Logický program P a dotaz G .

1. Inicializuj množinu cílů S literály z dotazu G ; $S := G$
2. `while (S != empty) do`
3. Vyber $A \in S$ a dále vyber klauzuli $A' : -B_1, \dots, B_n$ ($n \geq 0$) z programu P takovou, že $\exists \sigma : A\sigma = A'\sigma$; σ je nejobecnější unifikátor.
Pokud neexistuje A' nebo σ , ukonči cyklus.

Abstraktní interpret

Vstup: Logický program P a dotaz G .

1. Inicializuj množinu cílů S literály z dotazu G ; $S := G$
2. `while (S != empty)` do
3. Vyber $A \in S$ a dále vyber klauzuli $A' : -B_1, \dots, B_n$ ($n \geq 0$) z programu P takovou, že $\exists \sigma : A\sigma = A'\sigma$; σ je nejobecnější unifikátor.
Pokud neexistuje A' nebo σ , ukonči cyklus.
4. Nahrad' A v S cíli B_1 až B_n .
5. Aplikuj σ na G a S .
6. `end while`
7. Pokud $S == \text{empty}$, pak výpočet úspěšně skončil a výstupem je G se všemi aplikovanými substitucemi.
Pokud $S \neq \text{empty}$, výpočet končí neúspěchem.

Abstraktní interpret – pokračování

Kroky (3) až (5) představují **redukci** (logickou inferenci) cíle A.

Počet redukcí za sekundu (LIPS) == indikátor výkonu implementace

Abstraktní interpret – pokračování

Kroky (3) až (5) představují **redukci** (logickou inferenci) cíle A.

Počet redukcí za sekundu (LIPS) == indikátor výkonu implementace

Věta

Existuje-li instance G' dotazu G , odvoditelná z programu P v konečném počtu kroků, pak bude tímto interpretem nalezena.

Nedeterminismus interpretu

1. **Selekční pravidlo:** výběr cíle A z množiny cílů S

● neovlivňuje výrazně výsledek chování interpretu

2. **Způsob prohledávání stromu výpočtu:** výběr klauzule A' z programu P

● je velmi důležitý, všechny klauzule totiž nevedou k úspěšnému řešení

Nedeterminismus interpretu

1. **Selekční pravidlo:** výběr cíle A z množiny cílů S

● neovlivňuje výrazně výsledek chování interpretu

2. **Způsob prohledávání stromu výpočtu:** výběr klauzule A' z programu P

● je velmi důležitý, všechny klauzule totiž nevedou k úspěšnému řešení

Vztah k úplnosti:

1. Selekční pravidlo neovlivňuje úplnost

● možno zvolit libovolné v rámci SLD rezoluce

2. Prohledávání stromu výpočtu do šířky nebo do hloubky

Nedeterminismus interpretu

1. **Selekční pravidlo:** výběr cíle A z množiny cílů S

● neovlivňuje výrazně výsledek chování interpretu

2. **Způsob prohledávání stromu výpočtu:** výběr klauzule A' z programu P

● je velmi důležitý, všechny klauzule totiž nevedou k úspěšnému řešení

Vztah k úplnosti:

1. Selekční pravidlo neovlivňuje úplnost

● možno zvolit libovolné v rámci SLD rezoluce

2. Prohledávání stromu výpočtu do šířky nebo do hloubky

„Prozření” – automatický výběr správné klauzule

● vlastnost abstraktního interpretu, kterou ale reálné interprety nemají

Prohledávání do šířky

1. Vybereme všechny klauzule A'_i , které je možno unifikovat s literálem A
 - necht' je těchto klauzulí q
2. Vytvoříme q kopií množiny S
3. V každé kopii redukuje A jednou z klauzulí A'_i .
 - aplikujeme příslušný nejobecnější unifikátor

Prohledávání do šířky

1. Vybereme všechny klauzule A'_i , které je možno unifikovat s literálem A
 - necht' je těchto klauzulí q
2. Vytvoříme q kopií množiny S
3. V každé kopii redukuje A jednou z klauzulí A'_i .
 - aplikujeme příslušný nejobecnější unifikátor
4. V následujících krocích redukuje všechny množiny S_i současně.

Prohledávání do šířky

1. Vybereme všechny klauzule A'_i , které je možno unifikovat s literálem A
 - necht' je těchto klauzulí q
2. Vytvoříme q kopií množiny S
3. V každé kopii redukuje A jednou z klauzulí A'_i .
 - aplikujeme příslušný nejobecnější unifikátor
4. V následujících krocích redukuje všechny množiny S_i současně.
5. Výpočet ukončíme úspěchem, pokud se alespoň jedna z množin S_i stane prázdnou.

Prohledávání do šířky

1. Vybereme všechny klauzule A'_i , které je možno unifikovat s literálem A
 - necht' je těchto klauzulí q
 2. Vytvoříme q kopií množiny S
 3. V každé kopii redukuje A jednou z klauzulí A'_i .
 - aplikujeme příslušný nejobecnější unifikátor
 4. V následujících krocích redukuje všechny množiny S_i současně.
 5. Výpočet ukončíme úspěchem, pokud se alespoň jedna z množin S_i stane prázdnou.
-
- Ekvivalence s abstraktnímu interpretem
 - pokud jeden interpret neuspěje, pak neuspěje i druhý
 - pokud jeden interpret uspěje, pak uspěje i druhý

Prohledávání do hloubky

1. Vybereme všechny klauzule A'_i , které je možno unifikovat s literálem A.
2. Všechny tyto klauzule zapíšeme na zásobník.
3. Redukci provedeme s klauzulí na vrcholu zásobníku.

Prohledávání do hloubky

1. Vybereme všechny klauzule A'_i , které je možno unifikovat s literálem A .
2. Všechny tyto klauzule zapíšeme na zásobník.
3. Redukci provedeme s klauzulí na vrcholu zásobníku.
4. Pokud v nějakém kroku nenajdeme vhodnou klauzuli A' , vrátíme se k předchozímu stavu (tedy anulujeme aplikace posledního unifikátoru σ) a vybereme ze zásobníku další klauzuli.

Prohledávání do hloubky

1. Vybereme všechny klauzule A'_i , které je možno unifikovat s literálem A .
2. Všechny tyto klauzule zapíšeme na zásobník.
3. Redukci provedeme s klauzulí na vrcholu zásobníku.
4. Pokud v nějakém kroku nenajdeme vhodnou klauzuli A' , vrátíme se k předchozímu stavu (tedy anulujeme aplikace posledního unifikátoru σ) a vybereme ze zásobníku další klauzuli.
5. Pokud je zásobník prázdný, končí výpočet neúspěchem.
6. Pokud naopak zredukujeme všechny literály v S , výpočet končí úspěchem.

Prohledávání do hloubky

1. Vybereme všechny klauzule A'_i , které je možno unifikovat s literálem A .
2. Všechny tyto klauzule zapíšeme na zásobník.
3. Redukci provedeme s klauzulí na vrcholu zásobníku.
4. Pokud v nějakém kroku nenajdeme vhodnou klauzuli A' , vrátíme se k předchozímu stavu (tedy anulujeme aplikace posledního unifikátoru σ) a vybereme ze zásobníku další klauzuli.
5. Pokud je zásobník prázdný, končí výpočet neúspěchem.
6. Pokud naopak zredukujeme všechny literály v S , výpočet končí úspěchem.

- Není úplné, tj. nemusí najít všechna řešení
- Nižší paměťová náročnost než prohledávání do šířky
- Používá se v Prologu

Reprezentace objektů

- Beztypový jazyk
- Kontrola „typů“ za běhu výpočtu
- Informace o struktuře součástí objektu

Reprezentace objektů

- Beztypový jazyk
- Kontrola „typů“ za běhu výpočtu
- Informace o struktuře součástí objektu

Typy objektů

- **Primitivní objekty:**
 - konstanta
 - číslo
 - volná proměnná
 - odkaz (reference)

Reprezentace objektů

- Beztypový jazyk
- Kontrola „typů“ za běhu výpočtu
- Informace o struktuře součástí objektu

Typy objektů

● Primitivní objekty:

- konstanta
- číslo
- volná proměnná
- odkaz (reference)

● Složené (strukturované) objekty:

- struktura
- seznam

Reprezentace objektů II

Příznaky (tags):

Objekt	Příznak
volná proměnná	FREE
konstanta	CONST
celé číslo	INT
odkaz	REF
složený term	FUNCT

Reprezentace objektů II

Příznaky (tags):

Objekt	Příznak
volná proměnná	FREE
konstanta	CONST
celé číslo	INT
odkaz	REF
složený term	FUNCT

Obsah adresovatelného slova: **hodnota** a **příznak**.

Reprezentace objektů II

Příznaky (tags):

Objekt	Příznak
volná proměnná	FREE
konstanta	CONST
celé číslo	INT
odkaz	REF
složený term	FUNCT

Obsah adresovatelného slova: **hodnota** a **příznak**.

Primitivní objekty uloženy přímo ve slově

Reprezentace objektů II

Příznaky (tags):

Objekt	Příznak
volná proměnná	FREE
konstanta	CONST
celé číslo	INT
odkaz	REF
složený term	FUNCT

Obsah adresovatelného slova: **hodnota** a **příznak**.

Primitivní objekty uloženy přímo ve slově

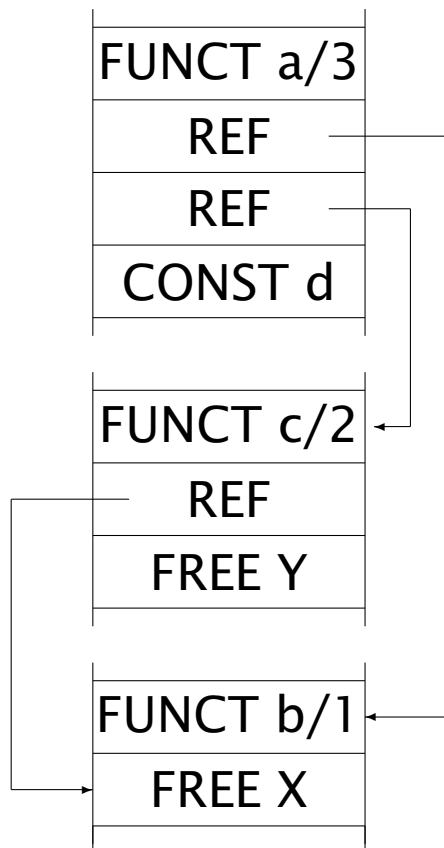
Složené objekty

- jsou instance termu ve zdrojovém textu, tzv. zdrojového termu
- zdrojový term bez proměnných \Rightarrow každá instancie ekvivalentní zdrojovému termu
- zdrojový term s proměnnými \Rightarrow dvě instance se mohou lišit aktuálními hodnotami proměnných, jedinečnost zajišťuje kopírování struktur nebo sdílení struktur

Kopírování struktur

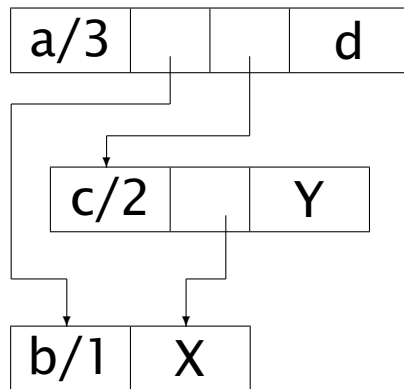
Příklad:

$a(b(X), c(X, Y), d),$



Kopírování struktur II

- Term F s aritou A reprezentován A+1 slovy:
 - funktor a arita v prvním slově
 - 2. slovo nese první argument (resp. odkaz na jeho hodnotu) :
 - A+1 slovo nese hodnotu A-tého argumentu
- Reprezentace vychází z orientovaných acyklických grafů:



- Vykopírována každá instance ⇒ **kopírování struktur**
- Termy ukládány na **globální zásobník**

Sdílení struktur

- Vychází z myšlenky, že při reprezentaci je třeba řešit přítomnost proměnných
- Instance termu

< kostra_termu; rámeč >

- kostra_termu je zdrojový term s očíslovanými proměnnými
- rámeč je vektor aktuálních hodnot těchto proměnných
 - i -tá položka nese hodnotu i -té proměnné v původním termu

Sdílení struktur II

Příklad:

$a(b(X), c(X, Y), d)$

reprezentuje

$\langle a(b(\$1), c(\$1, \$2), d) ; [\text{FREE}, \text{FREE}] \rangle$

kde symbolem $\$i$ označujeme i -tou proměnnou.

Sdílení struktur II

Příklad:

`a(b(X), c(X, Y), d)`

reprezentuje

`< a(b($1), c($1, $2), d) ; [FREE, FREE] >`

kde symbolem `$i` označujeme i -tou proměnnou.

Implementace:

`< &kostra_termu; &rámec >` (& vrací adresu objektu)

Všechny instance sdílí společnou `kostru_termu` ⇒ **sdílení struktur**

Srovnání: příklad

- Naivní srovnání: sdílení paměťově méně náročné

Srovnání: příklad

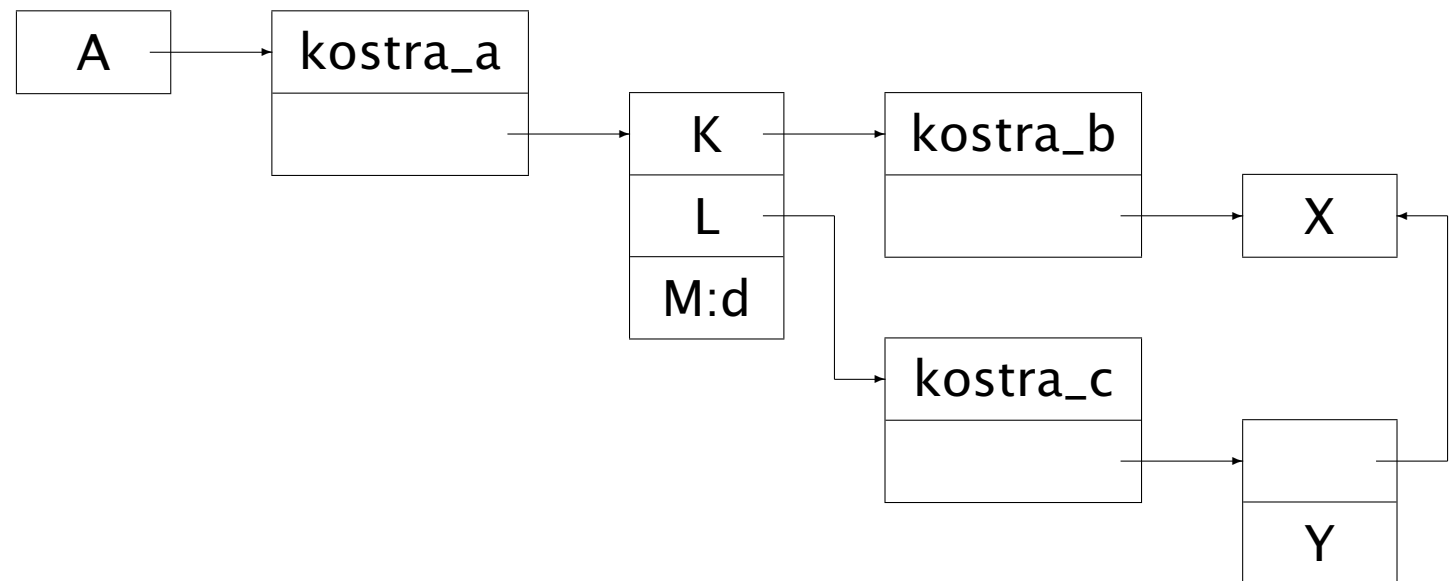
- Naivní srovnání: sdílení paměťově méně náročné
- Platí ale pouze pro rozsáhlé termy přítomné ve zdrojovém kódu

Srovnání: příklad

- Naivní srovnání: sdílení paměťově méně náročné
- Platí ale pouze pro rozsáhlé termy přítomné ve zdrojovém kódu
- Postupná tvorba termů:

$A = a(K, L, M)$, $K = b(X)$, $L = c(X, Y)$, $M = d$

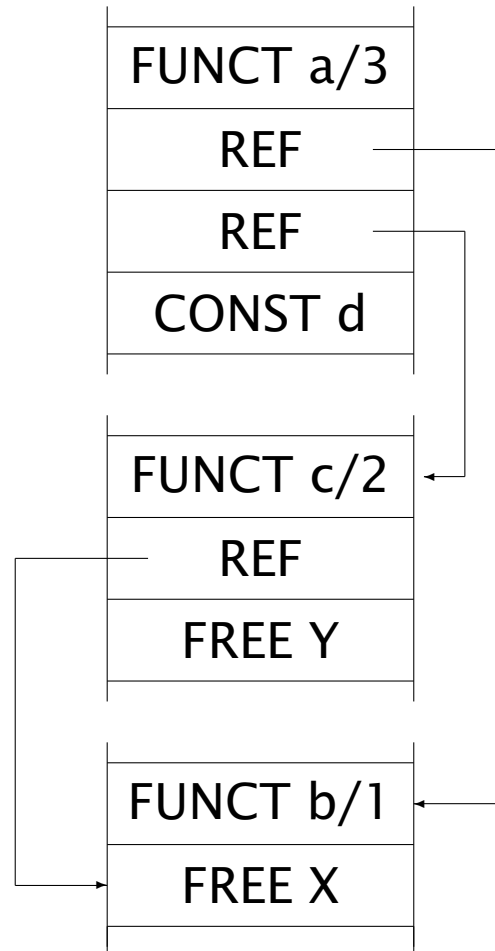
- Sdílení termů:



Srovnání: příklad – pokračování

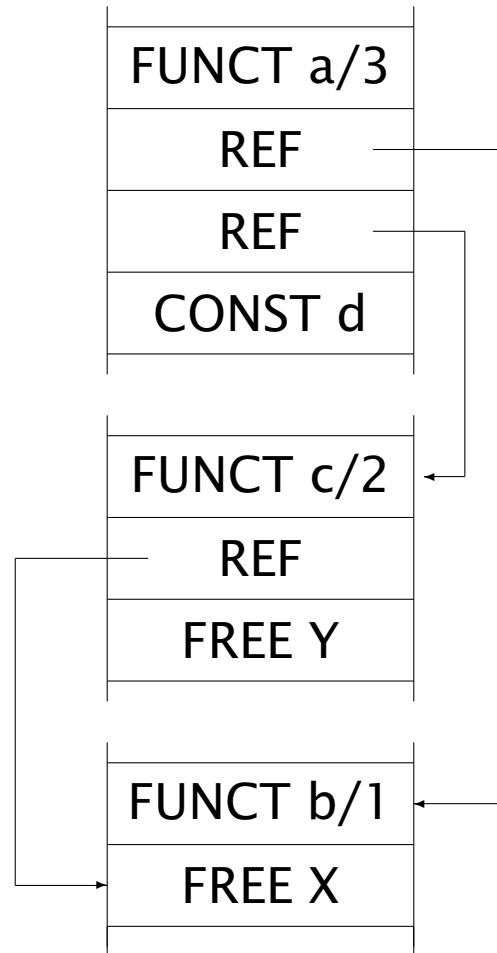
● Kopírování struktur:

$A = a(K, L, M)$, $K = b(X)$, $L = c(X, Y)$, $M = d$



Srovnání: příklad – pokračování

● Kopírování struktur: $A = a(K, L, M), K = b(X), L = c(X, Y), M = d$



tj. identické jako přímé vytvoření termu $a(b(X), c(X, Y), d)$

Srovnání II

- **Složitost algoritmů pro přístup k jednotlivým argumentům**
 - sdílení struktur: nutná víceúrovňová nepřímá adresace
 - kopírování struktur: bez problémů
 - jednodušší algoritmy usnadňují i optimalizace

Srovnání II

● Složitost algoritmů pro přístup k jednotlivým argumentům

- sdílení struktur: nutná víceúrovňová nepřímá adresace
- kopírování struktur: bez problémů
- jednodušší algoritmy usnadňují i optimalizace

● Lokalita přístupů do paměti

- sdílení struktur: přístupy rozptýleny po paměti
- kopírování struktur: lokalizované přístupy
- při stránkování paměti – rozptýlení vyžaduje přístup k více stránkám

Srovnání II

- **Složitost algoritmů pro přístup k jednotlivým argumentům**
 - sdílení struktur: nutná víceúrovňová nepřímá adresace
 - kopírování struktur: bez problémů
 - jednodušší algoritmy usnadňují i optimalizace
- **Lokalita přístupů do paměti**
 - sdílení struktur: přístupy rozptýleny po paměti
 - kopírování struktur: lokalizované přístupy
 - při stránkování paměti – rozptýlení vyžaduje přístup k více stránkám
- Z praktického hlediska neexistuje mezi těmito přístupy zásadní rozdíl

Řízení výpočtu

● Dopředný výpočet

- po úspěchu (úspěšná redukce)
 - jednotlivá volání procedur skončí úspěchem
- klasické volání rekurzivních procedur

Řízení výpočtu

● Dopředný výpočet

- po úspěchu (úspěšná redukce)
 - jednotlivá volání procedur skončí úspěchem
- klasické volání rekurzivních procedur

● Zpětný výpočet (backtracking)

- po neúspěchu vyhodnocení literálu (neúspěšná redukce)
 - nepodaří se unifikace aktuálních a formálních parametrů hlavy
- návrat do bodu, kde zůstala nevyzkoušená alternativa výpočtu
 - je nutná obnova původních hodnot jednotlivých proměnných
 - po nalezení místa s dosud nevyzkoušenou klauzulí pokračuje dále dopředný výpočet

Aktivační záznam

- Volání (=aktivace) procedury
- Aktivace **sdílí společný kód**, liší se obsahem **aktivačního záznamu**
- Aktivační záznam uložen na **lokálním zásobníku**

Aktivační záznam

- Volání (=aktivace) procedury
- Aktivace **sdílí společný kód**, liší se obsahem **aktivačního záznamu**
- Aktivační záznam uložen na **lokálním zásobníku**
- Dopředný výpočet
 - stav výpočtu v okamžiku volání procedury
 - aktuální parametry
 - lokální proměnné
 - pomocné proměnné ('a la registry)

Aktivační záznam

- Volání (=aktivace) procedury
- Aktivace **sdílí společný kód**, liší se obsahem **aktivačního záznamu**
- Aktivační záznam uložen na **lokálním zásobníku**
- Dopředný výpočet
 - stav výpočtu v okamžiku volání procedury
 - aktuální parametry
 - lokální proměnné
 - pomocné proměnné ('a la registry)
- Zpětný výpočet (backtracking)
 - hodnoty parametrů v okamžiku zavolání procedury
 - následující klauzule pro zpracování při neúspěchu

Aktivační záznam a roll-back

● Neúspěšná klauzule mohla nainstanciovat nelokální proměnné

● $a(X) \text{ :- } X = b(c, Y), Y = d.$ $?- W = b(Z, e), a(W).$

Aktivační záznam a roll-back

● Neúspěšná klauzule mohla nainstanciovat nelokální proměnné

● $a(X) \text{ :- } X = b(c, Y), Y = d.$ $?- W = b(Z, e), a(W).$ (viz instanciacce Z)

Aktivační záznam a roll-back

- Neúspěšná klauzule mohla nainstanciovat nelokální proměnné
 - $a(X) \text{ :- } X = b(c, Y), Y = d. \quad ?- W = b(Z, e), a(W). \quad (\text{viz instanciace } Z)$
 - Při návratu je třeba obnovit (**roll-back**) původní hodnoty proměnných
 - Využijeme vlastností logických proměnných
 - instanciovat lze pouze volnou proměnnou
 - jakmile proměnná získá hodnotu, nelze ji změnit jinak než návratem výpočtu
- ⇒ původní hodnoty všech proměnných odpovídají volné proměnné

Aktivační záznam a roll-back

- Neúspěšná klauzule mohla nainstanciovat nelokální proměnné
 - $a(X) \text{ :- } X = b(c, Y), Y = d. \quad \text{?- } W = b(Z, e), a(W). \quad (\text{viz instanciace } Z)$
- Při návratu je třeba obnovit (**roll-back**) původní hodnoty proměnných
- Využijeme vlastností logických proměnných
 - instanciovat lze pouze volnou proměnnou
 - jakmile proměnná získá hodnotu, nelze ji změnit jinak než návratem výpočtu

⇒ původní hodnoty všech proměnných odpovídají volné proměnné
- **Stopa** (trail): zásobník s adresami instanciovaných proměnných
 - ukazatel na aktuální vrchol zásobníku uchováván v aktivačním záznamu
 - při neúspěchu jsou hodnoty proměnných na stopě v úseku mezi aktuálním a uloženým vrcholem zásobníku změněny na „volná“

Aktivační záznam a roll-back

- Neúspěšná klauzule mohla nainstanciovat nelokální proměnné
 - $a(X) :- X = b(c, Y), Y = d. \quad ?- W = b(Z, e), a(W). \quad$ (viz instanciace Z)
- Při návratu je třeba obnovit (**roll-back**) původní hodnoty proměnných
- Využijeme vlastností logických proměnných
 - instanciovat lze pouze volnou proměnnou
 - jakmile proměnná získá hodnotu, nelze ji změnit jinak než návratem výpočtu

⇒ původní hodnoty všech proměnných odpovídají volné proměnné
- **Stopa** (trail): zásobník s adresami instanciovaných proměnných
 - ukazatel na aktuální vrchol zásobníku uchováván v aktivačním záznamu
 - při neúspěchu jsou hodnoty proměnných na stopě v úseku mezi aktuálním a uloženým vrcholem zásobníku změněny na „volná“
- **Globální zásobník**: pro uložení složených termů
 - ukazatel na aktuální vrchol zásobníku uchováván v aktivačním záznamu
 - při neúspěchu vrchol zásobníku snížen podle uschované hodnoty v aktivačním záznamu

Okolí a bod volby

Aktivační záznam úspěšně ukončené procedury nelze odstranit z lokálního zásobníku \Rightarrow **rozdělení aktivačního záznamu:**

- **okolí** (environment) – informace nutné pro dopředný běh programu
- **bod volby** (choice point) – informace nezbytné pro zotavení po neúspěchu

Okolí a bod volby

Aktivační záznam úspěšně ukončené procedury nelze odstranit z lokálního zásobníku \Rightarrow **rozdělení aktivačního záznamu:**

- **okolí** (environment) – informace nutné pro dopředný běh programu
- **bod volby** (choice point) – informace nezbytné pro zotavení po neúspěchu
- ukládány na lokální zásobník
- samostatně provázány (odkaz na předchozí okolí resp. bod volby)

Okolí a bod volby

Aktivační záznam úspěšně ukončené procedury nelze odstranit z lokálního zásobníku \Rightarrow **rozdělení aktivačního záznamu:**

- **okolí** (environment) – informace nutné pro dopředný běh programu
- **bod volby** (choice point) – informace nezbytné pro zotavení po neúspěchu
- ukládány na lokální zásobník
- samostatně provázány (odkaz na předchozí okolí resp. bod volby)

Důsledky:

- samostatná práce s každou částí aktivačního záznamu (optimalizace)

Okolí a bod volby

Aktivační záznam úspěšně ukončené procedury nelze odstranit z lokálního zásobníku \Rightarrow **rozdělení aktivačního záznamu:**

- **okolí** (environment) – informace nutné pro dopředný běh programu
- **bod volby** (choice point) – informace nezbytné pro zotavení po neúspěchu
- ukládány na lokální zásobník
- samostatně provázány (odkaz na předchozí okolí resp. bod volby)

Důsledky:

- samostatná práce s každou částí aktivačního záznamu (optimalizace)
- alokace pouze okolí pro deterministické procedury

Okolí a bod volby

Aktivační záznam úspěšně ukončené procedury nelze odstranit z lokálního zásobníku \Rightarrow **rozdělení aktivačního záznamu:**

- **okolí** (environment) – informace nutné pro dopředný běh programu
- **bod volby** (choice point) – informace nezbytné pro zotavení po neúspěchu
- ukládány na lokální zásobník
- samostatně provázány (odkaz na předchozí okolí resp. bod volby)

Důsledky:

- samostatná práce s každou částí aktivačního záznamu (optimalizace)
- alokace pouze okolí pro deterministické procedury
- možnost odstranění okolí po úspěšném vykonání (i nedeterministické) procedury (pokud okolí následuje po bodu volby dané procedury)
 - pokud je okolí na vrcholu zásobníku

Řez

● Prostředek pro ovlivnění běhu výpočtu programátorem

● `a(X) :- b(X), !, c(X). a(3).`
`b(1). b(2).`
`c(1). c(2).`

Řez

- Prostředek pro ovlivnění běhu výpočtu programátorem

- $a(X) \text{ :- } b(X), !, c(X). \quad a(3).$
 $b(1). \quad b(2).$
 $c(1). \quad c(2).$

- Řez: neovlivňuje dopředný výpočet, má vliv pouze na zpětný výpočet

- Odstranění alternativních větví výpočtu

⇒ odstranění odpovídajících bodů volby

- tj. odstranění bodů volby mezi současným vrcholem zásobníku a bodem volby procedury, která řez vyvolala (včetně bodu volby procedury s řezem)

⇒ změna ukazatele na „nejmladší“ bod volby

Řez

- Prostředek pro ovlivnění běhu výpočtu programátorem

- $a(X) \text{ :- } b(X), !, c(X). \quad a(3).$
 $b(1). \quad b(2).$
 $c(1). \quad c(2).$

- Řez: neovlivňuje dopředný výpočet, má vliv pouze na zpětný výpočet

- Odstranění alternativních větví výpočtu

⇒ odstranění odpovídajících bodů volby

- tj. odstranění bodů volby mezi současným vrcholem zásobníku a bodem volby procedury, která řez vyvolala (včetně bodu volby procedury s řezem)

⇒ změna ukazatele na „nejmladší“ bod volby

⇒ Vytváření deterministických procedur

⇒ Optimalizace využití zásobníku