

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 H lava :- 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**
a 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 ($S \neq \text{empty}$)` 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

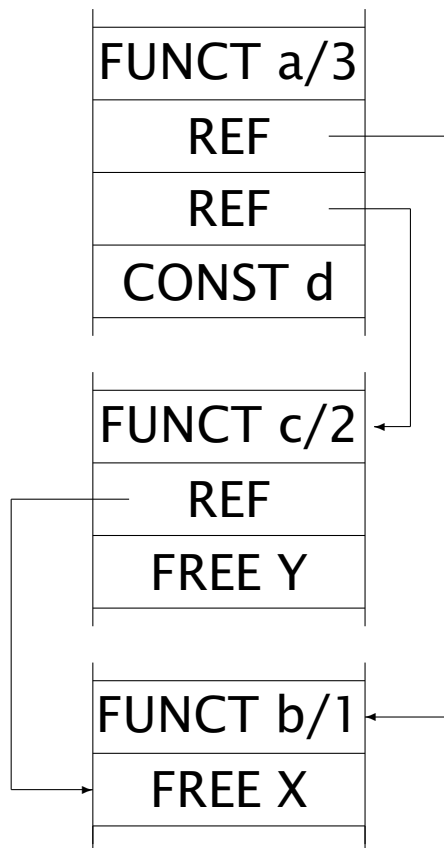
Složené objekty

- 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:

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

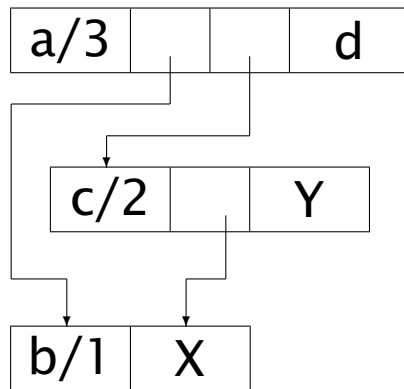


Kopírování struktur II

● 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

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

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

Implementace:

$\langle \&kostra_termu; \&rámec \rangle$

(& vrací adresu objektu)

Všechny instance sdílí společnou $kostru_termu \Rightarrow$ **sdílení struktur**

Srovnání: příklad

- Naivní srovnání: sdílení paměťově méně náročné

Srovnání: příklad

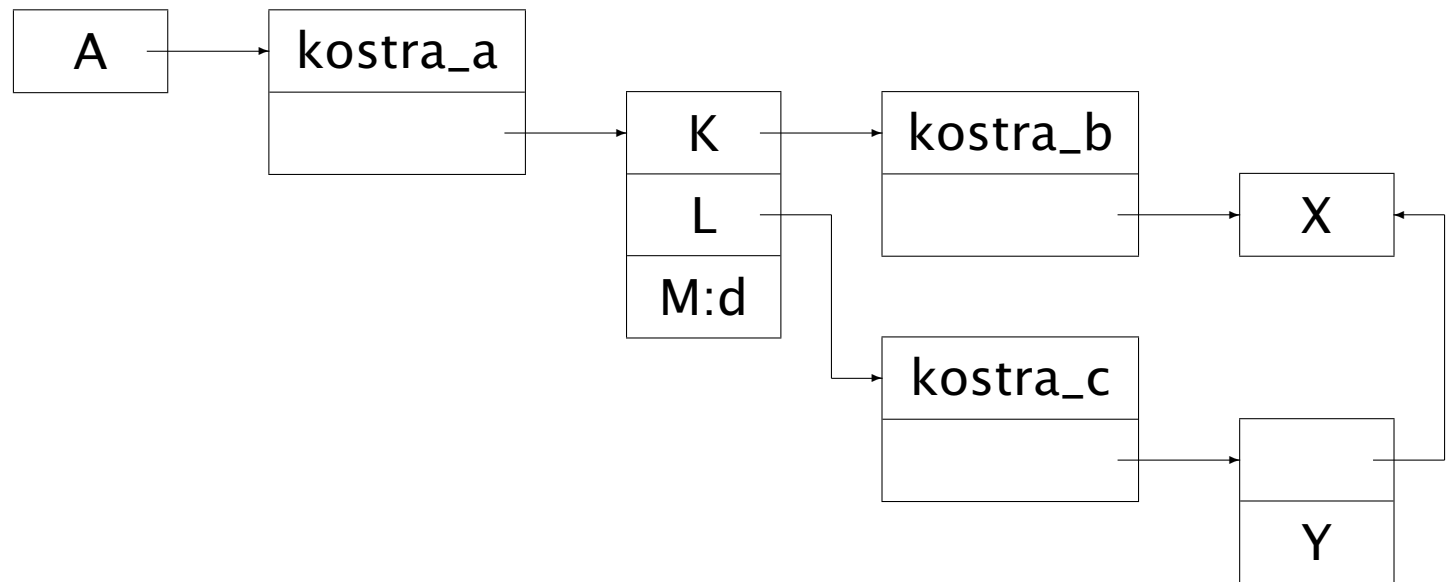
- Naivní srovnání: sdílení paměťově méně náročné
- Platí ale pouze pro rozsáhlé termy přítomné ve zdrojovém kódu

Srovnání: příklad

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

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

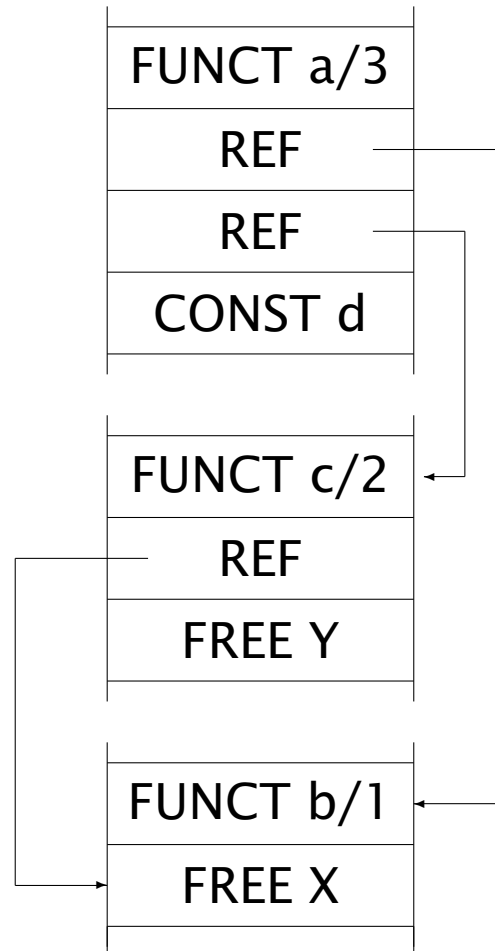
- Sdílení termů:



Srovnání: příklad – pokračování

● Kopírování struktur:

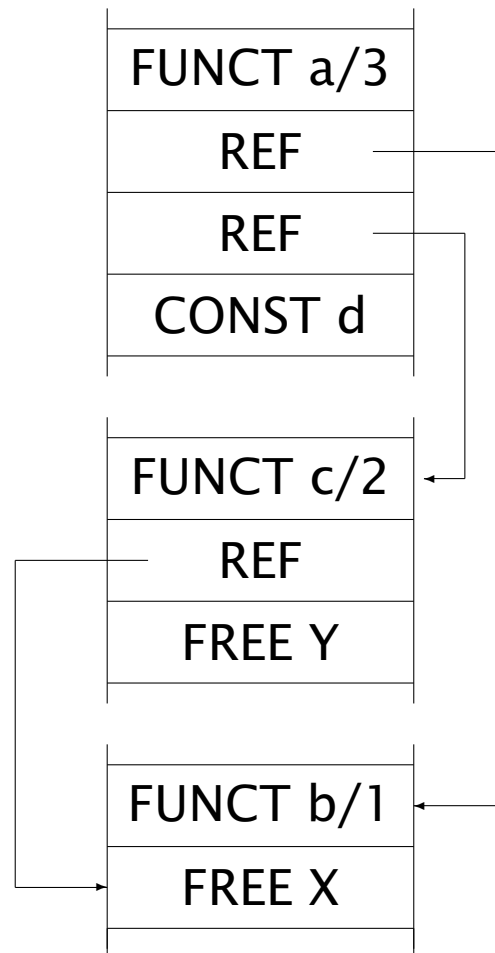
$A = a(K, L, M)$, $K = b(X)$, $L = c(X, Y)$, $M = d$



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 instanciace 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 uchovávan 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
 - $a(X) \text{ :- } X = b(c, Y), Y = d. \quad ?- W = b(Z, e), a(W). \quad (\text{viz instanciace } Z)$
- 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
- **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

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

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

Warrenův abstraktní počítač, WAM I.

Navržen D.H.D. Warrenem v roce 1983, modifikace do druhé poloviny 80. let

Datové oblasti:

● **Oblast kódu** (programová databáze)

- separátní oblasti pro uživatelský kód (modifikovatelný) a vestavěné predikáty (nemění se)
- obsahuje rovněž všechny statické objekty (texty atomů a funktorů apod.)

● **Lokální zásobník (*Stack*)**

● **Stopa (*Trail*)**

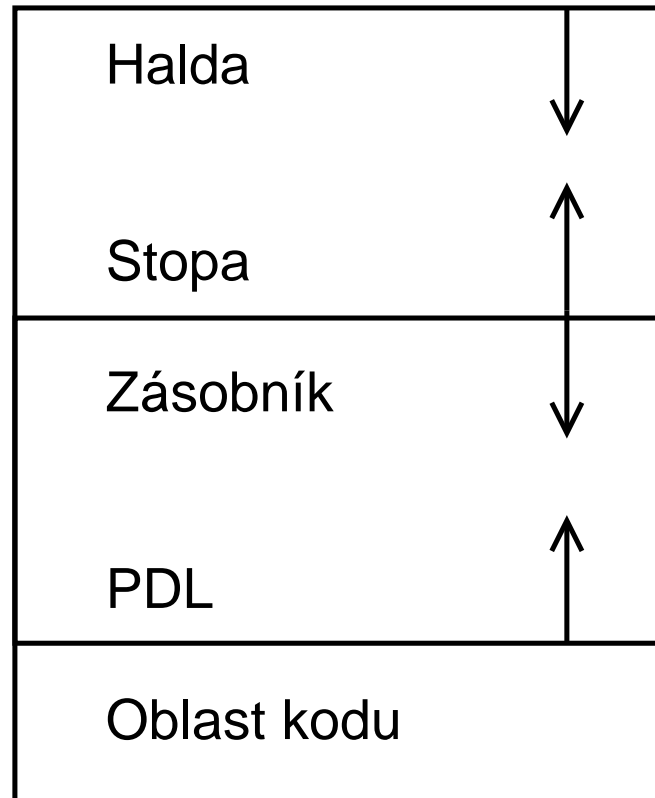
● **Globální zásobník n. halda (*Heap*)**

● **Pomocný zásobník (*Push Down List, PDL*)**

- pracovní paměť abstraktního počítače
- použitý v unifikaci, syntaktické analýze apod.

Rozmístění datových oblastí

● Příklad konfigurace



● Halda i lokální zásobník musí růst stejným směrem

- Ize jednoduše porovnat stáří dvou proměnných srovnáním adres využívá se při zabrání vzniku visících odkazů

Registry WAMu

● Stavové registry:

P čítač adres (Program counter)

CP adresa návratu (Continuation Pointer)

E ukazatel na nejmladší okolí (Environment)

B ukazatel na nejmladší bod volby (Backtrack point)

TR vrchol stopy (TRail)

H vrchol haldy (Heap)

HB vrchol haldy v okamžiku založení posledního bodu volby (Heap on Backtrack point)

S ukazatel, používaný při analýze složených termů (Structure pointer)

CUT ukazatel na bod volby, na který se řezem zařízne zásobník

● **Argumentové registry:** A1, A2, . . . (při předávání parametrů n. pracovní registry)

● **Registry pro lokální proměnné:** Y1, Y2, . . .

● abstraktní znázornění lok. proměnných na zásobníku

Typy instrukcí WAMu

- **get instrukce** – unifikace aktuálních a formálních parametrů
 - vykonávají činnost analogickou instrukcím `unify` u parc. vyhodnocení
 - obecná unifikace pouze při `get_value`
- **put instrukce** – příprava argumentů před voláním podcíle
 - žádná z těchto instrukcí nevolá obecný unifikační algoritmus
- **unify instrukce** – zpracování složených termů
 - jednoargumentové instrukce, používají registr `S` jako druhý argument
 - počáteční hodnota `S` je odkaz na 1. argument
 - volání instrukce `unify` zvětší hodnotu `S` o jedničku
 - obecná unifikace pouze při `unify_value` a `unify_local_value`
- **Indexační instrukce** – indexace klauzulí a manipulace s body volby
- **Instrukce řízení běhu** – předávání řízení a explicitní manipulace s okolím

Instrukce WAMu

get instrukce

get_var	Ai, Y
get_value	Ai, Y
get_const	Ai, C
get_nil	Ai
get_struct	Ai, F/N
get_list	Ai

put instrukce

put_var	Ai, Y
put_value	Ai, Y
put_unsafe_value	Ai, Y
put_const	Ai, C
put_nil	Ai
put_struct	Ai, F/N
put_list	Ai

unify instrukce

unify_var	Y
unify_value	Y
unify_local_value	Y
unify_const	C
unify_nil	
unify_void	N

instrukce řízení

allocate	
deallocate	
call	Proc/N, A
execute	Proc/N
proceed	

indexační instrukce

try_me_else	Next	try	Next
retry_me_else	Next	retry	Next
trust_me_else	fail	trust	fail
cut_last		switch_on_term	Var, Const, List, Struct
save_cut	Y	switch_on_const	Table
load_cut	Y	switch_on_struct	Table

Instrukce unify, get, put

- Větší počet typů objektů
 - rozlišeny atomy, čísla, `nil` \equiv prázdný seznam, seznam speciální druh složeného termu
- `unify_void` umožní přeskočit anonymních proměnné ve složených termech
- `put_unsafe_value` pro optimalizaci práce s lokálními proměnnými při TRO
 - `a(X) :- b(X,Y), !, a(Y).`
 - při TRO nesmí být lokální proměnné posledního literálu (Y) na lokálním zásobníku
 - kompilátor může všechny **nebezpečné (*unsafe*)** výskyty lok. proměnných detekovat při překladu (jsou to poslední výskyty lok. proměnných) a generuje složitější instrukce `put_unsafe_value`, které provádějí test umístění

Instrukce unify, get, put

- Větší počet typů objektů
 - rozlišeny atomy, čísla, nil \equiv prázdný seznam, seznam speciální druh složeného termu
- `unify_void` umožní přeskočit anonymních proměnné ve složených termech
- `put_unsafe_value` pro optimalizaci práce s lokálními proměnnými při TRO
 - $a(X) :- b(X,Y), !, a(Y).$
 - při TRO nesmí být lokální proměnné posledního literálu (Y) na lokálním zásobníku
 - kompilátor může všechny **nebezpečné (*unsafe*)** výskyty lok. proměnných detekovat při překladu (jsou to poslední výskyty lok. proměnných) a generuje složitější instrukce `put_unsafe_value`, které provádějí test umístění
- `unify_local_value` kvůli TRO jako `put_unsafe_value`
 - $a(X) :- d(X), b(s(Y),X).$ objekt přístupný přes Y opět nesmí být na lok. zásobníku
doba života $s/1$ může být delší než doba života okolí na něž se Y odkazuje
 - `unify_local_value` testují umístění a pokud nutné přesouvají objekty na haldu

WAM – indexace

- Provázání klauzulí: instrukce `XX_me_else`:
 - první klauzule: `try_me_else`; založí bod volby
 - poslední klauzule: `trust_me_else`; zruší nejmladší bod volby
 - ostatní klauzule: `retry_me_else`; znovu použije nejmladší bod volby po neúspěchu

WAM – indexace

- Provázání klauzulí: instrukce `XX_me_else`:
 - první klauzule: `try_me_else`; založí bod volby
 - poslední klauzule: `trust_me_else`; zruší nejmladší bod volby
 - ostatní klauzule: `retry_me_else`; znovu použije nejmladší bod volby po neúspěchu
- Provázání podmnožiny klauzulí (podle argumentu):
 - `try`
 - `retry`
 - `trust`

WAM – indexace

- Provázání klauzulí: instrukce `XX_me_else`:
 - první klauzule: `try_me_else`; založí bod volby
 - poslední klauzule: `trust_me_else`; zruší nejmladší bod volby
 - ostatní klauzule: `retry_me_else`; znovu použije nejmladší bod volby po neúspěchu
- Provázání podmnožiny klauzulí (podle argumentu):
 - `try`
 - `retry`
 - `trust`
- „Rozskokové” instrukce (dle typu a hodnoty argumentu):
 - `switch_on_term Var, Const, List, Struct`
výpočet následuje uvedeným návěstím podle typu prvního argumentu
 - `switch_on_YY`: hashovací tabulka pro konkrétní typ (konstanta, struktura)

Příklad indexace instrukcí

Proceduře

a(atom) :- body1.

a(1) :- body2.

a(2) :- body3.

a([X|Y]) :- body4.

a([X|Y]) :- body5.

a(s(N)) :- body6.

a(f(N)) :- body7.

odpovídají instrukce

a: switch_on_term L1, L2, L3, L4

L2: switch_on_const atom :L1a

1 :L5a

2 :L6a

L3: try L7a

trust L8a

L4: switch_on_struct s/1 :L9a

f/1 :L10a

L1: try_me_else L5

L1a: body1

L5: retry_me_else L6

L5a: body2

L6: retry_me_else L7

L6a: body3

L7: retry_me_else L8

L7a: body4

L8: retry_me_else L9

L8a: body5

L9: retry_me_else L10

L9a: body6

L10: trust_me_else fail

L10a: body7

WAM – řízení výpočtu

- `call Proc,N`: zavolá Proc, N udává počet lok. proměnných (odpovídá velikosti zásobníku)

WAM – řízení výpočtu

● `call Proc,N`: zavolá Proc, N udává počet lok. proměnných (odpovídá velikosti zásobníku)

Možná optimalizace: vhodným uspořádáním proměnných

Ize dosáhnout postupného zkracování lokálního zásobníku

`a(A,B,C,D) :- b(D), c(A,C), d(B), e(A), f.`

```
generujeme instrukce    allocate
                          call b/1,4
                          call c/2,3
                          call d/1,2
                          call e/1,1
                          deallocate
                          execute f/0
```

WAM – řízení výpočtu

- `call Proc, N`: zavolá Proc, N udává počet lok. proměnných (odpovídá velikosti zásobníku)

Možná optimalizace: vhodným uspořádáním proměnných

Ize dosáhnout postupného zkracování lokálního zásobníku

`a(A,B,C,D) :- b(D), c(A,C), d(B), e(A), f.`

```
generujeme instrukce    allocate
                          call b/1,4
                          call c/2,3
                          call d/1,2
                          call e/1,1
                          deallocate
                          execute f/0
```

- `execute Proc`: ekvivalentní příkazu `goto`

WAM – řízení výpočtu

- `call Proc, N`: zavolá Proc, N udává počet lok. proměnných (odpovídá velikosti zásobníku)

Možná optimalizace: vhodným uspořádáním proměnných

Ize dosáhnout postupného zkracování lokálního zásobníku

`a(A,B,C,D) :- b(D), c(A,C), d(B), e(A), f.`

```
generujeme instrukce  allocate
                        call b/1,4
                        call c/2,3
                        call d/1,2
                        call e/1,1
                        deallocate
                        execute f/0
```

- `execute Proc`: ekvivalentní příkazu `goto`

- `proceed`: zpracování faktů

WAM – řízení výpočtu

- `call Proc, N`: zavolá Proc, N udává počet lok. proměnných (odpovídá velikosti zásobníku)

Možná optimalizace: vhodným uspořádáním proměnných

Ize dosáhnout postupného zkracování lokálního zásobníku

$a(A,B,C,D) :- b(D), c(A,C), d(B), e(A), f.$

```
generujeme instrukce    allocate
                          call b/1,4
                          call c/2,3
                          call d/1,2
                          call e/1,1
                          deallocate
                          execute f/0
```

- `execute Proc`: ekvivalentní příkazu goto

- `proceed`: zpracování faktů

- `allocate`: alokuje okolí (pro některé klauzule netřeba, proto explicitně generováno)

WAM – řízení výpočtu

- `call Proc, N`: zavolá Proc, N udává počet lok. proměnných (odpovídá velikosti zásobníku)

Možná optimalizace: vhodným uspořádáním proměnných

Ize dosáhnout postupného zkracování lokálního zásobníku

$a(A,B,C,D) :- b(D), c(A,C), d(B), e(A), f.$

```
generujeme instrukce    allocate
                          call b/1,4
                          call c/2,3
                          call d/1,2
                          call e/1,1
                          deallocate
                          execute f/0
```

- `execute Proc`: ekvivalentní příkazu goto

- `proceed`: zpracování faktů

- `allocate`: alokuje okolí (pro některé klauzule netřeba, proto explicitně generováno)

- `deallocate`: uvolní okolí (je-li to možné, tedy leží-li na vrcholu zásobníku)

WAM – řez

Implementace řezu (opakování): odstranění bodů volby mezi současným vrcholem zásobníku a bodem volby procedury, která řez vyvolala (včetně bodu volby procedury s řezem)

Indexační instrukce znemožňují v době překladu rozhodnout, zda bude alokován bod volby

● příklad: $?- a(X) .$ může být nedeterministické, ale $?- a(1) .$ může být deterministické

WAM – řez

Implementace řezu (opakování): odstranění bodů volby mezi současným vrcholem zásobníku a bodem volby procedury, která řez vyvolala (včetně bodu volby procedury s řezem)

Indexační instrukce znemožňují v době překladu rozhodnout, zda bude alokován bod volby

● příklad: `?- a(X) .` může být nedeterministické, ale `?- a(1) .` může být deterministické

`cut_last: B := CUT`

`save_cut Y: Y := CUT`

`load_cut Y: B := Y`

WAM – řez

Implementace řezu (opakování): odstranění bodů volby mezi současným vrcholem zásobníku a bodem volby procedury, která řez vyvolala (včetně bodu volby procedury s řezem)

Indexační instrukce znemožňují v době překladu rozhodnout, zda bude alokován bod volby

● příklad: $?- a(X) .$ může být nedeterministické, ale $?- a(1) .$ může být deterministické

`cut_last: B := CUT` `save_cut Y: Y := CUT` `load_cut Y: B := Y`

Hodnota registru B je uchovávána v registru CUT instrukcemi `call` a `execute`.

Je-li řez prvním predikátem klauzule, použije se rovnou `cut_last`. V opačném případě se použije jako první instrukce `save_cut Y` a v místě skutečného volání řezu se použije `load_cut Y`.

WAM – řez

Implementace řezu (opakování): odstranění bodů volby mezi současným vrcholem zásobníku a bodem volby procedury, která řez vyvolala (včetně bodu volby procedury s řezem)

Indexační instrukce znemožňují v době překladu rozhodnout, zda bude alokován bod volby

● příklad: $?- a(X)$. může být nedeterministické, ale $?- a(1)$. může být deterministické

```
cut_last:   B := CUT           save_cut Y:   Y := CUT           load_cut Y:   B := Y
```

Hodnota registru B je uchovávána v registru CUT instrukcemi call a execute.

Je-li řez prvním predikátem klauzule, použije se rovnou cut_last. V opačném případě se použije jako první instrukce save_cut Y a v místě skutečného volání řezu se použije load_cut Y.

Příklad: $a(X,Z) :- b(X), !, c(Z)$.

$a(2,Z) :- !, c(Z)$.

$a(X,Z) :- d(X,Z)$. odpovídá

```
save_cut Y2; get A2,Y1; call b/1,2; load_cut Y2; put Y1,A1; execute c/1
```

```
get_const A1,2; cut_last; put A2,A1; execute c/1
```

```
execute d/2
```

WAM – optimalizace

1. Indexace klauzulí
2. Generování optimální posloupnosti instrukcí WAMu
3. Odstranění redundancí při generování cílového kódu.

WAM – optimalizace

1. Indexace klauzulí
2. Generování optimální posloupnosti instrukcí WAMu
3. Odstranění redundancí při generování cílového kódu.

● Příklad: $a(X,Y,Z) :- b(f,X,Y,Z)$.

naivní kód (vytvoří kompilátor pracující striktně zleva doprava) vs.

optimalizovaný kód (počet registrů a tedy i počet instrukcí/přesunů v paměti snížen):

get_var	A1,A5		get_var	A3,A4
get_var	A2,A6		get_var	A2,A3
get_var	A3,A7		get_var	A1,A2
put_const	A1,f		put_const	A1,f
put_value	A2,A5		execute	b/4
put_value	A3,A6			
put_value	A4,A7			
execute	b/4			