Pravidla hry

- **Přednáška:** účast není povinná
- **Cvičení:** zaměřeno na praktické aspekty
  - Zápočet udělen za zápočtový projekt
- **Zkouška.** Složky:

  (1) Bodování projektu, celkem až 40 bodů

  (2) Písemka, až 160 bodů

  (3) Písemná část: součet bodů za projekt a za písemku.
  Známka výborně za cca 150 bodů.

  (4) Ústní zkouška – jen pro nespokojené s písemnými výsledky

  (5) Známka bude zapsána pouze těm, kteří dostanou zápočet za projekt.

  Jediný termín písemky – **30. května 2002 od 13:00 v D1**.
  Žádná opravná písemka
  Alternativní termín písemky po dohodě jen v případech závažných důvodů pro neúčast (spíše formou delší ústní zkoušky).
  Jeden řádný termín ústní zkoušky pro každého zájemce.

- Průsvitky: `http://www.ics.muni.cz/people/matyska/vyuka/lp/prednaska.PDF`

Přehled přednášky

- Predikátový počet
- Negace v logickém programování
- Implementační techniky
- Logické programování s omezujícími podmínkami
- Paralelní logické programování

Deklarativní programování

- Specifikační jazyky
- Jasná sémantika
- Nevhodné pro procedurální postupy
- Kombinatorické problémy

Co dělat namísto **Jak dělat**.

Predikátový počet

Abeceda

(1) malá a velká písmena anglické abecedy

(2) číslice 0,1,$\cdots$,9

(3) znak podtrženík (\u201e_\u201c)

(Vlastní) symboly

posloupnost znaků abecedy začínající písmenem

Nevlastní symboly

( ) , .

Logické spojky

$\wedge$, $\vee$, $\neg$, $\Rightarrow$, $=$

Kvantifikátory

$\forall$, $\exists$

Vlastní symboly mohou reprezentovat:
proměnné, konstanty, funkce, predikáty

Logika I. řádu

Formule = řetězce tvořené

- predikátovými symboly (p, q, ...)
- funkčními symboly (f, g, ...)
- proměnnými (X, Y, ...)
- logickými operacemi ($\wedge$, $\vee$, $\Rightarrow$, ...)
- kvantifikátory ($\forall$, $\exists$)
- a nevlastními symboly ( ) , .

Logika I. řádu

Formule = řetězce tvořené

- predikátovými symboly (p, q, ...)
- funkčními symboly (f, g, ...)
- proměnnými (X, Y, ...)
- logickými operacemi ($\wedge$, $\vee$, $\Rightarrow$, ...)
- kvantifikátory ($\forall$, $\exists$)
- a nevlastními symboly ( ) , .

Dobře utvořená formule

Každá dobře utvořená formule (well founded formula, wff) označuje pravdivostní hodnotu (PRAVDA, NEPRAVDA) v závislosti na své struktuře a jako funkci **interpretace** predikátových a funkčních symbolů a volných proměnných.

Interpretace je dána neprázdnou množinou D a zobrazením, které

- každé konstantě c přiřadí nějaký prvek D
- každému n-árnímu funkčnímu symbolu f přiřadí n-ární operaci v D
- každému n-árnímu predikátovému symbolu p přiřadí n-ární relaci na D

Interpretace se nazývá **modelem** formule, je-li v ní tato formula pravdivá.

Interpretace je dána neprázdnou množinou D a zobrazením, které

- každé konstantě c přiřadí nějaký prvek D
- každému n-árnímu funkčnímu symbolu f přiřadí n-ární operaci v D
- každému n-árnímu predikátovému symbolu p přiřadí n-ární relaci na D

Interpretace se nazývá **modelem** formule, je-li v ní tato formula pravdivá.

Teorie je rekurzivní množina formulí, tzv. **axiomů**

Interpretace je dána neprázdnou množinou D a zobrazením, které

- každé konstantě c přiřadí nějaký prvek D
- každému n-árnímu funkčnímu symbolu f přiřadí n-ární operaci v D
- každému n-árnímu predikátovému symbolu p přiřadí n-ární relaci na D

Interpretace se nazývá **modelem** formule, je-li v ní tato formula pravdivá.

Teorie je rekurzivní množina formulí, tzv. **axiomů**

Modelem teorie je libovolná interpretace, která je modelem všech jejích axiomů.

Formule F je **pravdivá** (**platí**, **je splněna**) v teorii $\mathcal{F} \models F$, je-li pravdivá v každém z modelů teorie $\mathcal{F}$.

Formule F, pro kterou je libovolná interpretace modelem (resp. je pravdivá v každém modelu libovolné teorie), se nazývá **logicky pravdivá**, značíme $\models F$.

Zkoumání pravdivosti formulí

- **ve výrokovém počtu** to lze ověřováním v každém modelu
- obecně zjištění pravdivosti provádíme **důkazem**

Zkoumání pravdivosti formulí

- **ve výrokovém počtu** to lze ověřováním v každém modelu
- obecně zjištění pravdivosti provádíme **důkazem**

Důkaz = konečný rekurzivní objekt, odpovídající dokazované formuli F a jistým formulím $A_1, \cdots, A_n$ (předpokladům), který je sestaven podle určitých **inferenčních pravidel**.

Zkoumání pravdivosti formulí

- **ve výrokovém počtu** to lze ověřováním v každém modelu
- obecně zjištění pravdivosti provádíme **důkazem**

Důkaz = konečný rekurzivní objekt, odpovídající dokazované formuli F a jistým formulím $A_1, \cdots, A_n$ (předpokladům), který je sestaven podle určitých **inferenčních pravidel**.

Dokazatelnost

Existuje-li důkaz F z $A_1, \cdots, A_n$, říkáme, že F je **dokazatelná z formulí** $A_1, \cdots, A_n$ ($A_1, \cdots, A_n \vdash F$).

Zkoumání pravdivosti formulí

- **ve výrokovém počtu** to lze ověřováním v každém modelu
- obecně zjištění pravdivosti provádíme **důkazem**

Důkaz = konečný rekurzivní objekt, odpovídající dokazované formuli F a jistým formulím $A_1, \cdots, A_n$ (předpokladům), který je sestaven podle určitých **inferenčních pravidel**.

Dokazatelnost

Existuje-li důkaz F z $A_1, \cdots, A_n$, říkáme, že F je **dokazatelná z formulí** $A_1, \cdots, A_n$ $(A_1, \cdots, A_n \vdash F)$.

Teorém

Dokazatelné formule nazýváme **teorémy** (teorie $\mathcal{T}$). Pro predikátovou logiku prvního řádu je k dispozici úplná a korektní dokazatelnost, tj. pro teorii $\mathcal{T}$ s množinou axiomů $\mathcal{A}$ platí $\mathcal{T} \models F$ právě když $\mathcal{A} \vdash F$.

Herbrandovy interpretace

Herbrand ukázal, že při zkoumání pravdivosti není nutné uvažovat modely nad všemi interpretacemi, ale stačí se omezit na obor skládající se ze symbolických výrazů tvořených z predikátových a funkčních symbolů daného jazyka – **Herbrandovo univerzum** $\mathrm{U}(P)$ nad množinou P formulí je množina všech uzavřených termů, které mohou být tvořeny predikátovými a funkčními symboly z P.

Herbrandova báze $\mathrm{B}(P)$ je množina všech atomických formulí nad prvky $\mathrm{U}(P)$.

Herbrandova interpretace je libovolná interpretace, která přiřazuje

- proměnným prvky Herbrandova univerza
- konstantám sebe samé
- funkčním symbolům funkce, které symbolu f pro argumenty $t_1, \cdots, t_n$ přiřadí term $f(t_1, \cdots, t_n)$
- predikátovým symbolům libovolnou funkci z Herbrandova univerza do pravdivostních hodnot.

Herbrandův model je Herbrandova interpretace taková, že každá klauzule z P je v ní pravdivá.

Rezoluční princip

Požadavek: formule v **klauzulárním tvaru** (konjunktivní normální formě).

Základní pojmy:

Literál = elementární formule
$p(t_1, \cdots, t_n)$ – pozitivní
$\neg p(t_1, \cdots, t_n)$ – negativní

Klauzule = konečná množina literálů, reprezentující jejich disjunkci.
Klauzule je pravdivá, právě když je pravdivý alespoň jeden z jejích literálů.

Prázdná klauzule se značí $\square$ a je vždy nepravdivá.

Splnitelnost

Množina klauzulí reprezentuje konjunkci. Množina klauzulí je splnitelná, existuje-li interpretace, pro kterou je pravdivá.

Rezoluční princip

Požadavek: formule v **klauzulárním tvaru** (konjunktivní normální formě).

Základní pojmy:

Literál = elementární formule
$p(t_1, \cdots, t_n)$ – pozitivní
$\neg p(t_1, \cdots, t_n)$ – negativní

Klauzule = konečná množina literálů, reprezentující jejich disjunkci.
Klauzule je pravdivá, právě když je pravdivý alespoň jeden z jejích literálů.

Prázdná klauzule se značí $\square$ a je vždy nepravdivá.

Splnitelnost

Množina klauzulí reprezentuje konjunkci. Množina klauzulí je splnitelná, existuje-li interpretace, pro kterou je pravdivá.

Věta

Množina P klauzulí je splnitelná právě tehdy, když existuje její Herbrandův model.

Rezoluční princip II

Rezoluční princip = pravidlo, které umožňuje odvodit z $P \cup \{A\}$ a $\{\neg A\} \cup Q$ klauzuli $P \cup Q$, přitom $P \cup Q$ se nazývá **rezolventou** původních klauzulí

$$\frac{P \cup \{A\} \qquad \{\neg A\} \cup Q}{P \cup Q}.$$

Rezoluční princip II

Rezoluční princip = pravidlo, které umožňuje odvodit z $P \cup \{A\}$ a $\{\neg A\} \cup Q$ klauzuli $P \cup Q$, přitom $P \cup Q$ se nazývá **rezolventou** původních klauzulí

$$\frac{P \cup \{A\} \qquad \{\neg A\} \cup Q}{P \cup Q}.$$

Rezoluční důkaz formule F spočívá v demonstraci nesplnitelnosti $\neg F$.
Vyjdeme-li z klauzulí reprezentujících $\neg F$, musíme postupným uplatňováním rezolučního principu dospět k prázdné klauzuli $\square$.

Rezoluční princip II

Rezoluční princip = pravidlo, které umožňuje odvodit z $P \cup \{A\}$ a $\{\neg A\} \cup Q$ klauzuli $P \cup Q$, přitom $P \cup Q$ se nazývá **rezolventou** původních klauzulí

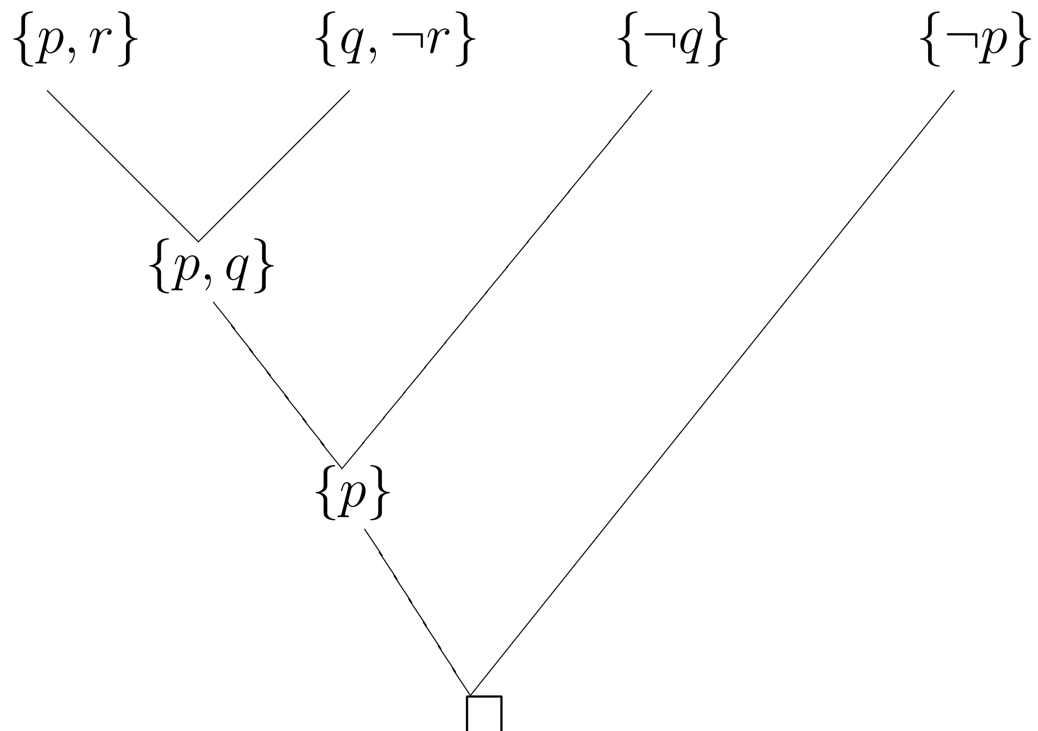
$$\frac{P \cup \{A\} \qquad \{\neg A\} \cup Q}{P \cup Q}.$$

Rezoluční důkaz formule F spočívá v demonstraci nesplnitelnosti $\neg F$. Vyjdeme-li z klauzulí reprezentujících $\neg F$, musíme postupným uplatňováním rezolučního principu dospět k prázdné klauzuli $\square$.

Rezoluční důkaz klauzule C z množiny klauzulí S je binární strom, ve kterém je kořen označen klauzulí C, listy klauzulemi z S a pro každý uzel u, který není listem, s bezprostředními potomky v_1 a v_2 označenými klauzulemi C_1 a C_2 je uzel u označen rezolventou klauzulí C_1 a C_2.

Rezoluční princip III

Příklad: $S = \{\{p, r\}, \{q, \neg r\}, \{\neg q\}, \{\neg p\}\}$
$C = \square$



Substituce

Substituce je libovolná funkce Θ zobrazující výrazy do výrazů podle následujících pravidel:

$\Theta(E) = E$ pro libovolnou konstantu E

$\Theta(f(E_1, \cdots, E_n)) = f(\Theta(E_1), \cdots, \Theta(E_n))$ pro libovolný funkční symbol f

$\Theta(p(E_1, \cdots, E_n)) = p(\Theta(E_1), \cdots, \theta(E_n))$ pro libovolný predikátový symbol p

Substituce je tedy homomorfismus výrazů, který zachová vše kromě proměnných – ty lze nahradit čímkoliv. Speciální náhrada proměnných proměnnými se nazývá **přejmenování proměnných**.

Substituce zapisujeme zpravidla ve tvaru seznamu

$$[X_1/\xi_1, \cdots, X_n/\xi_n]$$

kde X_i jsou proměnné a ξ_i substituované termy.

Příklad:

$$p(X)[X/f(a)] \equiv p(f(a))$$

Unifikace

Ztotožnění dvou literálů p, q pomocí vhodné substituce σ takové, že $p\sigma = q\sigma$ nazýváme **unifikací** a příslušnou substituci **unifikátorem**.

Definice 1 **Unifikátorem** množiny S literálů nazýváme substituce Θ takovou, že množina

$$S\Theta = \{t\Theta | t \in S\}$$

má jediný prvek.

Unifikace

Ztotožnění dvou literálů p, q pomocí vhodné substituce σ takové, že $p\sigma = q\sigma$ nazýváme **unifikací** a příslušnou substituci **unifikátorem**.

Definice 1 **Unifikátorem** množiny S literálů nazýváme substituce Θ takovou, že množina

$$S\Theta = \{t\Theta | t \in S\}$$

má jediný prvek.

Definice 2 Unifikátor σ množiny S nazýváme **nejobecnějším unifikátorem** (**mgu**), jestliže pro libovolný unifikátor Θ existuje substituce λ taková, že $\Theta = \sigma\lambda$.

Rezoluce v logice I. řádu

Rezoluční princip je pravidlo, které

- připraví příležitost pro uplatnění vlastního rezolučního pravidla nalezením vhodného unifikátoru
- provede rezoluci a získá rezolventu
- používá pouze nejobecnější unifikátory

Rezoluce v logice I. řádu

Rezoluční princip je pravidlo, které

- připraví příležitost pro uplatnění vlastního rezolučního pravidla nalezením vhodného unifikátoru
- provede rezoluci a získá rezolventu
- používá pouze nejobecnější unifikátory

Symbolický zápis:

$$\frac{P \cup \{A_1, \cdots, A_m\} \qquad \{\neg B_1, \cdots, \neg B_n\} \cup Q}{P\rho\sigma \cup Q\sigma}$$

kde ρ je přejmenováním proměnných takové, že obě množiny nemají společné proměnné, a σ je nejobecnější unifikátor množiny $\{A_1\rho, \cdots, A_m\rho, B_1, \cdots, B_n\}$.

Varianty rezoluční metody

- **Sémantická rezoluce**
  Zvolíme libovolnou interpretaci a pro rezoluci používáme jen takové klauzule, z nichž alespoň jedna je v této interpretaci nepravdivá.

- Použití **oporných množin** (Support set)
  Rozlišujeme mezi axiomy a dokazovaným teorémem. **Oporná množina** T množiny klauzulí S ($T \subset S$) je taková množina, že $S \subsetneq T$ je bezesporná. Při rezoluci bereme jen klauzule z T a/nebo z předchozí rezolventy.

- $\boldsymbol{P_1}$ **rezoluce**
  Jedna z klauzulí, účastnící se rezoluce musí být pozitivní.

- $\boldsymbol{N_1}$ **rezoluce**
  Jedna z klauzulí, účastnící se rezoluce musí být negativní.

- **Lineární rezoluce**
  V každém kroku kromě prvního můžeme použít bezprostředně předcházející rezolventu a k tomu buď některou z klauzulí vstupní množiny S nebo některou z předcházejících rezolvent.

Neúplné varianty rezoluce

- **Jednotková rezoluce**
  Alespoň jedna klauzule, použitá při rezoluci, je jednoprvková.
- **Vstupní rezoluce**
  Alespoň jedna z klauzulí, použitá při rezoluci, pochází z výchozí vstupní množiny S.

Obě uvedené alternativy jsou úplné na identické třídě klauzulí.

Hornovy klauzule

Definice 3 **Hornova klauzule** je libovolná klauzule s nejvýše jedním pozitivním literálem.

- Hornova klauzule s právě jedním pozitivním literálem se nazývá **programová klauzule**

  $$\{q, \neg p_1, \neg p_2, \cdots, \neg p_n\} \equiv q \leftarrow p_1, \cdots, p_n$$

  Je-li $n > 0$, pak se tato klauzule nazývá **pravidlo**,
  pro $n = 0$ se nazývá **fakt**.

- Hornova klauzule bez pozitivního literálu se nazývá **cíl** (**cílová klauzule**).

Hornovy klauzule II

Základní tvrzení

Hornovy klauzule II

Základní tvrzení

Věta

Je-li S nesplnitelná množina Hornových klauzulí, pak S obsahuje alespoň jeden cíl a jeden fakt.

Hornovy klauzule II

Základní tvrzení

Věta

Je-li S nesplnitelná množina Hornových klauzulí, pak S obsahuje alespoň jeden cíl a jeden fakt.

Věta

Existuje-li rezoluční důkaz prázdné množiny z množiny S Hornových klauzulí, pak tento rezoluční strom má v listech jedinou cílovou klauzuli.

Hornovy klauzule II

Základní tvrzení

Věta

Je-li S nesplnitelná množina Hornových klauzulí, pak S obsahuje alespoň jeden cíl a jeden fakt.

Věta

Existuje-li rezoluční důkaz prázdné množiny z množiny S Hornových klauzulí, pak tento rezoluční strom má v listech jedinou cílovou klauzuli.

Věta

Množina S Hornových klauzulí je nesplnitelná, právě když existuje rezoluční důkaz prázdné klauzule z S pomocí jednotkové rezoluce.

Hornovy klauzule II

Základní tvrzení

Věta

Je-li S nesplnitelná množina Hornových klauzulí, pak S obsahuje alespoň jeden cíl a jeden fakt.

Věta

Existuje-li rezoluční důkaz prázdné množiny z množiny S Hornových klauzulí, pak tento rezoluční strom má v listech jedinou cílovou klauzuli.

Věta

Množina S Hornových klauzulí je nesplnitelná, právě když existuje rezoluční důkaz prázdné klauzule z S pomocí jednotkové rezoluce.

Věta

Množina S Hornových klauzulí je nesplnitelná právě když existuje rezoluční důkaz prázdné klauzule z S pomocí vstupní rezoluce.

Hornovy klauzule III

Důsledek

Je-li P množina programových Hornových klauzulí a $\mathcal{G}$ cílová klauzule, pak existuje lineární rezoluční důkaz prázdné klauzule z $P \cup \{\mathcal{G}\}$ začínající cílovou klauzulí $\mathcal{G}$.

Hornovy klauzule a Herbrandův model

Je-li S množina programových klauzulí a M libovolná množina Herbrandových modelů, pak průnik těchto modelů je opět Herbrandův model množiny S.

Důsledek

Existuje **nejmenší Herbrandův model** množiny S, který značíme $\boldsymbol{M(S)}$.

Existence $M(S)$ je důsledkem tvaru Hornových klauzulí.

Logické programy

- **Logickým programem** rozumíme libovolnou konečnou množinu programových Hornových klauzulí.
- **Deklarativní sémantikou** logického programu P rozumíme jeho minimální Herbrandův model $M(P)$.
- **Operační sémantikou** logického programu P rozumíme množinu $O(P)$ všech atomických formulí bez proměnných, které lze pro nějaký cíl $\mathcal{G}$[1] odvodit nějakým rezolučním důkazem ze vstupní množiny $P \cup \{\mathcal{G}\}$.
- pro libovolný logický program P platí

$$M(P) = O(P)$$

[1]tímto výrazem jsou míněny všechny cíle, pro něž zmíněný rezoluční důkaz existuje.

Lineární rezoluce se selekčním pravidlem

SLD-rezoluce (**Selected Linear resolution for Definite clauses**)

Definice 4 (1) **Selekční pravidlo** $\boldsymbol{R}$ je funkce, která každé neprázdné klauzuli Q přiřazuje nějaký z jejích literálů $R(Q) \in Q$.

(2) Důkazem prázdné klauzule lineární rezolucí se selekčním pravidlem R (SLD-rezolucí) rozumíme rezoluční důkaz prázdné klauzule z množiny $P \cup \{\mathcal{G}\}$, kde $\mathcal{G}$ je cílová klauzule, odpovídající posloupnosti

$$\mathcal{G}_0, \mathcal{G}_1, \cdots, \mathcal{G}_n, \mathcal{G}_{n+1}$$

kde $\mathcal{G}_0 = \mathcal{G}$ a $\mathcal{G}_{n+1} = \Box$ a pro libovolné i, $0 \leq i \leq n$ je $\mathcal{G}_{i+1}$ rezolventou získanou z $\mathcal{G}_i$ a některé z programových klauzulí na literálu $R(\mathcal{G}_i)$.

Vlastnosti SLD-rezoluce

- je úplná
- v každém kroku jsou generovány výhradně negativní klauzule
- žádná klauzule v SLD-rezoluční posloupnosti není programovou klauzulí

Efektivita SLD-rezoluce je závislá na

- selekčním pravidle R
- způsobu výběru příslušné programové klauzule pro tvorbu rezolventy

SLD-rezoluce konstruuje po částech hledané termy.

Každý krok SLD-rezoluce vytváří novou unifikační substituce σ_i, která potenciálně instanciuje proměnné ve vstupní cílové klauzuli. Kompozice těchto unifikací

$$\alpha = \sigma_0\sigma_1 \cdots \sigma_n$$

se nazývá **výsledná substituce** (answer substitution) a její aplikace na $\mathcal{G}$ vytváří hledaný protipříklad demonstrující nesplnitelnost $P \cup \{\mathcal{G}\}$.

Strom výpočtu (SLD-strom)

je strom tvořený všemi možnými výpočetními posloupnostmi logického programu P vzhledem k cíli $\mathcal{G}$.

Kořeny stromy jsou programmové klauzule a cílová klauzule $\mathcal{G}$, v uzlech jsou rezolventy. Výchozím kořenem rezoluce je cílová klauzule $\mathcal{G}$.

Listy jsou dvojího druhu:

- označené prázdnou klauzulí – jedná se o **úspěšné uzly** (succes nodes)
- označené neprázdnou klauzulí – jedná se o **neúspěšné uzly** (failure nodes).

Úplnost SLD-rezoluce zaručuje **existenci** cesty od kořene k úspěšnému uzlu pro každý možný výsledek příslušející cíli $\mathcal{G}$.

Výpočetní strategie

Korektní výpočetní strategie prohledávání stromu výpočtu musí zaručit, že se každý (konečný) výsledek nalézt v konečném čase.

Výpočetní strategie

Korektní výpočetní strategie prohledávání stromu výpočtu musí zaručit, že se každý (konečný) výsledek nalézt v konečném čase.

Korektní výpočetní strategie = prohledávání stromu do šířky

Problémy

- Exponenciální paměťová náročnost
- Složité řídící struktury

Výpočetní strategie

Korektní výpočetní strategie prohledávání stromu výpočtu musí zaručit, že se každý (konečný) výsledek nalézt v konečném čase.

Korektní výpočetní strategie = prohledávání stromu do šířky

Problémy

- Exponenciální paměťová náročnost
- Složité řídící struktury

Použitelná výpočetní strategie = prohledávání stromu do hloubky

- Jednoduché řídící struktury (zásobník)
- Lineární paměťová náročnost
- Avšak: neúplné na nekonečných stromech (zacyklení)

Výpočetní strategie

Korektní výpočetní strategie prohledávání stromu výpočtu musí zaručit, že se každý (konečný) výsledek nalézt v konečném čase.

Korektní výpočetní strategie = prohledávání stromu do šířky

Problémy

- Exponenciální paměťová náročnost
- Složité řídící struktury

Použitelná výpočetní strategie = prohledávání stromu do hloubky

- Jednoduché řídící struktury (zásobník)
- Lineární paměťová náročnost
- Avšak: neúplné na nekonečných stromech (zacyklení)

V Prologu: prohledávání stromu do hloubky
$\Longrightarrow$ neúplnost použité výpočetní strategie.

Shrnutí

Mějme množinu P programových klauzulí a cílovou klauzuli $\mathcal{G}$. jaký je význam rezolučního SLD-důkazu prázdné klauzule z množiny $S = P \cup \{\mathcal{G}\}$?

Dokazujeme nesplnitelnost

(1) $P \wedge (\forall \vec{x})(\neg\mathcal{G}_1(\vec{x}) \vee \neg\mathcal{G}_2(\vec{x}) \vee \cdots \vee \neg\mathcal{G}_n(\vec{x}))$
kde $\mathcal{G} = \{\neg\mathcal{G}_1, \neg\mathcal{G}_2, \cdots, \neg\mathcal{G}_n\}$ a $\vec{x}$ je vektor proměnných v $\mathcal{G}$.
nesplnitelnost (1) je ekvivalentní tvrzení (2)

(2) $P \vdash \neg\mathcal{G}$
neboli

(3) $P \vdash (\exists \vec{x})(\mathcal{G}_1(\vec{x}) \wedge \cdots \wedge \mathcal{G}_n(\vec{x}))$

a jedná se tak o důkaz existence vhodných objektů, které na základě vlastností množiny P splňují konjunkci predikátů v cílové klauzuli.

Důkaz nesplnitelnosti $P \cup \{\mathcal{G}\}$ znamená nalezení protipříkladu – ten po extrakci z důkazového stromu konstruuje termy, které splňují konjunkci v (3).

SLDNF – negace v logickém programování

Negativní literály: pozice určena definicí Hornových klauzulí

$\Longrightarrow$ nelze vyvodit *negativní* informaci z logického programu (každý predikát definuje úplnou relaci; negativní literál *není* logickým důsledkem programu).

SLDNF – negace v logickém programování

Negativní literály: pozice určena definicí Hornových klauzulí

$\Longrightarrow$ nelze vyvodit *negativní* informaci z logického programu (každý predikát definuje úplnou relaci; negativní literál *není* logickým důsledkem programu).

Řešení převzato z databází:
Předpoklad uzavřeného světa (Closed World Assumption, CWA):
Určitý vztah platí *pouze* když je vyvoditelný z programu.

SLDNF – negace v logickém programování

Negativní literály: pozice určena definicí Hornových klauzulí

$\Longrightarrow$ nelze vyvodit *negativní* informaci z logického programu (každý predikát definuje úplnou relaci; negativní literál *není* logickým důsledkem programu).

Řešení převzato z databází:

Předpoklad uzavřeného světa (Closed World Assumption, CWA):

Určitý vztah platí *pouze* když je vyvoditelný z programu.

„Inferenční pravidlo“(A je (uzavřený) term): $\dfrac{\mathrm{P} \not\models A}{\neg A}$

SLDNF – negace v logickém programování

Negativní literály: pozice určena definicí Hornových klauzulí

$\Longrightarrow$ nelze vyvodit *negativní* informaci z logického programu (každý predikát definuje úplnou relaci; negativní literál *není* logickým důsledkem programu).

Řešení převzato z databází:
Předpoklad uzavřeného světa (Closed World Assumption, CWA):
Určitý vztah platí *pouze* když je vyvoditelný z programu.

„Inferenční pravidlo“(A je (uzavřený) term): $\dfrac{\mathrm{P} \not\models A}{\neg A}$

Problém: není rozhodnutelné, zda daná atomická formule je logickým důsledkem daného logického programu.

SLDNF – negace v logickém programování

Negativní literály: pozice určena definicí Hornových klauzulí

$\Longrightarrow$ nelze vyvodit *negativní* informaci z logického programu (každý predikát definuje úplnou relaci; negativní literál *není* logickým důsledkem programu).

Řešení převzato z databází:
Předpoklad uzavřeného světa (Closed World Assumption, CWA):
Určitý vztah platí *pouze* když je vyvoditelný z programu.

„Inferenční pravidlo“(A je (uzavřený) term): $\dfrac{\mathrm{P} \not\models A}{\neg A}$

Problém: není rozhodnutelné, zda daná atomická formule je logickým důsledkem daného logického programu.

CWA v logickém programování obecně nepoužitelná.

SLDNF – negace v logickém programování II

Speciální případ: **Definitivně neúspěšný** (finitely failed) SLD strom cíle $\leftarrow A$.

SLDNF – negace v logickém programování II

Speciální případ: **Definitivně neúspěšný** (finitely failed) SLD strom cíle $\leftarrow \boldsymbol{A}$.

Důsledek: A není logickým důsledkem programu P.

SLDNF – negace v logickém programování II

Speciální případ: **Definitivně neúspěšný** (finitely failed) SLD strom cíle $\leftarrow \boldsymbol{A}$.

Důsledek: A není logickým důsledkem programu P.

Důsledek 2: $\forall(\neg A)$

SLDNF – negace v logickém programování II

Speciální případ: **Definitivně neúspěšný** (finitely failed) SLD strom cíle $\leftarrow \boldsymbol{A}$.

Důsledek: A není logickým důsledkem programu P.

Důsledek 2: $\forall(\neg A)$

Normální cíle: Cíle, které obsahují negativní literály.

SLDNF – negace v logickém programování II

Speciální případ: **Definitivně neúspěšný** (finitely failed) SLD strom cíle $\leftarrow \boldsymbol{A}$.

Důsledek: A není logickým důsledkem programu P.

Důsledek 2: $\forall(\neg A)$

Normální cíle: Cíle, které obsahují negativní literály.

Negace jako neúspěch (Negation as Failure, NF):

$$\frac{\leftarrow A_1, \ldots, A_{i-1}, \neg A_i, A_{i+1}, \ldots, A_n \quad \leftarrow A_i \text{ definitivně neúspěšný v P}}{\leftarrow A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_n}$$

Zúplnění Logického Programu

Podstata: převod všech *if* příkazů v logickém programu na *iff*

Zúplnění Logického Programu

Podstata: převod všech *if* příkazů v logickém programu na *iff*

Definice: Nechť P je logický program a p/n je predikátový symbol. Pak *definice* $def(p/n)$ predikátu p/n je množina všech klauzulí ve tvaru $p(t_1, \ldots, t_n) \leftarrow B$ pro libovolné B a $t_1, \ldots, t_n$.

Zúplnění Logického Programu II

Definice: Nechť p/n je predikátový symbol programu P a nechť $X_1, \dots, X_n$ jsou „nové" proměnné, které se nevyskytují nikde v P.

- Nechť C je klauzule ve tvaru
  $p(t_1, \dots, t_n) \leftarrow L_1, \dots, L_m$
  kde $m \geq 0$, $t_1, \dots, t_n$ jsou termy a $L_1, \dots, L_m$ jsou literály. Pak označme $\mathrm{E}(C)$ výraz
  $\exists Y_1, \dots, Y_k (X_1 = t_1, \dots, X_n = t_n, L_1, \dots, L_m)$
  kde $Y_1, \dots, Y_k$ jsou všechny proměnné v C.
- Nechť $def(p/n) = \{C_1, \dots, C_n\}$. Pak formuli $\mathrm{IF}(p, \mathrm{P})$ získáme následujícím postupem:
  $p(X_1, \dots, X_n) \leftarrow \mathrm{E}(C_1) \vee \mathrm{E}(C_2) \vee \dots \vee \mathrm{E}(C_j)$ pro $j > 0$ a
  $p(X_1, \dots, X_n) \leftarrow \Box$ pro $j = 0$

$\mathrm{IF}(\mathrm{P})$ množina všech formulí $\mathrm{IF}(q, \mathrm{P})$ pro všechny predikátové symboly q v programu P.

$\mathrm{IFF}(\mathrm{P})$: spojka $\leftarrow$ v $\mathrm{IF}(\mathrm{P})$ je nahrazena spojkou $\leftrightarrow$

Clarkova Teorie Rovnosti

Definice: *Clarkova teorie rovnosti* (Clark's Euqality Theory, CET). $X \neq Y$ je zkrácený zápis $\neg(X = Y)$ a předpokládejme, že všechny formule jsou univerzálně kvantifikovány:

(1) $X = X$

(2) $X = Y \rightarrow Y = X$

(3) $X = Y \wedge Y = Z \rightarrow X = Z$

(4) Pro každý funktor f/m:
$X_1 = Y_1 \wedge \cdots \wedge X_m = Y_m \rightarrow f(X_1, \ldots, X_m) = f(Y_1, \ldots, Y_m)$

(5) Pro každý predikátový symbol p/m:
$X_1 = Y_1 \wedge \cdots \wedge X_m = Y_m \rightarrow (p(X_1, \ldots, X_m) \rightarrow p(Y_1, \ldots, Y_m))$

(6) Pro všechny různé funktory f/m a g/n, $(m, n \geq 0)$:
$f(X_1, \ldots, X_m) \neq g(Y_1, \ldots, Y_n)$

(7) Pro všechny funktory f:
$f(X_1, \ldots, X_m) = f(Y_1, \ldots, Y_m) \rightarrow X_1 = Y_1 \wedge \cdots \wedge X_m = Y_m$

(8) Pro každý term $t[X]$ obsahující X jako vlastní podterm:
$t[X] \neq X$

Zúplnění Logického Programu III

Zúplnění programu P je: $\mathrm{comp}(\mathrm{P}) := \mathrm{IFF}(\mathrm{P}) \cup \mathrm{CET}$

Zúplnění Logického Programu III

Zúplnění programu P je: $\mathrm{comp}(P) := \mathrm{IFF}(P) \cup \mathrm{CET}$

Základní vlastnost: $\mathrm{comp}(P) \models P$

Správnost NF pravidla

Nechť P je logický program a G cíl ve tvaru $\leftarrow A_1, \ldots, A_m$. Existuje-li výpočetní pravidlo R takové, že SLD strom pro cíl G a program P konstruovaný pravidlem R je definitivně neúspěšný, pak $\forall\neg(A_1 \wedge \cdots \wedge A_m)$ je logickým důsledkem $\mathrm{comp}(\mathrm{P})$.

Správnost NF pravidla

Nechť P je logický program a G cíl ve tvaru $\leftarrow A_1, \ldots, A_m$. Existuje-li výpočetní pravidlo R takové, že SLD strom pro cíl G a program P konstruovaný pravidlem R je definitivně neúspěšný, pak $\forall\neg(A_1 \wedge \cdots \wedge A_m)$ je logickým důsledkem $\mathrm{comp(P)}$.

Problémy:
důkaz existence definitivně neúspěšného SLD stromu

Správnost NF pravidla

Nechť P je logický program a G cíl ve tvaru $\leftarrow A_1, \ldots, A_m$. Existuje-li výpočetní pravidlo R takové, že SLD strom pro cíl G a program P konstruovaný pravidlem R je definitivně neúspěšný, pak $\forall \neg (A_1 \wedge \cdots \wedge A_m)$ je logickým důsledkem $\mathrm{comp}(\mathrm{P})$.

Problémy:
důkaz existence definitivně neúspěšného SLD stromu

korektní a nekorektní (fair and unfair) výpočetní pravidla.

Správnost NF pravidla

Nechť P je logický program a G cíl ve tvaru $\leftarrow A_1, \ldots, A_m$. Existuje-li výpočetní pravidlo R takové, že SLD strom pro cíl G a program P konstruovaný pravidlem R je definitivně neúspěšný, pak $\forall\neg(A_1 \wedge \cdots \wedge A_m)$ je logickým důsledkem $\mathrm{comp}(\mathrm{P})$.

Problémy:
důkaz existence definitivně neúspěšného SLD stromu

korektní a nekorektní (fair and unfair) výpočetní pravidla.

Úplnost NF pravidla

Obecně nelze nalézt každou správnou odpověď: SLDNF je *test*, nedokáže *konstruovat* výslednou substituci.

Normální a stratifikované programy

Normální program: obsahuje negativní literály v pravidlech.

Problém: existence zúplnění, která nemají žádný model:

$p \leftarrow \neg p$

Normální a stratifikované programy

Normální program: obsahuje negativní literály v pravidlech.

Problém: existence zúplnění, která nemají žádný model:

$p \leftarrow \neg p$

Definice: Normální program je *stratifikovaný*, právě tehdy, lze-li množinu predikátových symbolů programu P rozdělit do disjunktních množin $S_0, \ldots, S_m$ takových, že platí
$p(\ldots) \leftarrow L_1, \ldots, L_j \in \mathrm{P}$
a $p \in S_k$, $(0 \leq k \leq m)$, pak pro každé $L_i \in \{L_1, \ldots, L_j\}$:

- je-li L_i tvaru $q(\ldots)$, pak q je obsaženo v $S_0 \cup \ldots \cup S_k$;
- je-li L_i tvaru $\neg q(\ldots)$, pak q je obsaženo v $S_0 \cup \ldots \cup S_{k-1}$.

Každé S_i $(0 \leq i \leq m)$ se nazývá *stratum* (úroveň).

Normální a stratifikované programy

Normální program: obsahuje negativní literály v pravidlech.

Problém: existence zúplnění, která nemají žádný model:

$p \leftarrow \neg p$

Definice: Normální program je *stratifikovaný*, právě tehdy, lze-li množinu predikátových symbolů programu P rozdělit do disjunktních množin $S_0, \ldots, S_m$ takových, že platí

$p(\ldots) \leftarrow L_1, \ldots, L_j \in \mathrm{P}$

a $p \in S_k$, $(0 \leq k \leq m)$, pak pro každé $L_i \in \{L_1, \ldots, L_j\}$:

- je-li L_i tvaru $q(\ldots)$, pak q je obsaženo v $S_0 \cup \ldots \cup S_k$;
- je-li L_i tvaru $\neg q(\ldots)$, pak q je obsaženo v $S_0 \cup \ldots \cup S_{k-1}$.

Každé S_i $(0 \leq i \leq m)$ se nazývá *stratum* (úroveň).

Program je m-*stratifikovaný* iff m je nejmenší index takový, že $S_0 \cup \ldots \cup S_m$ je množina všech predikátových symbolů z P.

Normální a stratifikované programy

Normální program: obsahuje negativní literály v pravidlech.

Problém: existence zúplnění, která nemají žádný model:

$p \leftarrow \neg p$

Definice: Normální program je *stratifikovaný*, právě tehdy, lze-li množinu predikátových symbolů programu P rozdělit do disjunktních množin $S_0, \ldots, S_m$ takových, že platí
$p(\ldots) \leftarrow L_1, \ldots, L_j \in \mathrm{P}$
a $p \in S_k$, $(0 \leq k \leq m)$, pak pro každé $L_i \in \{L_1, \ldots, L_j\}$:

- je-li L_i tvaru $q(\ldots)$, pak q je obsaženo v $S_0 \cup \ldots \cup S_k$;
- je-li L_i tvaru $\neg q(\ldots)$, pak q je obsaženo v $S_0 \cup \ldots \cup S_{k-1}$.

Každé S_i $(0 \leq i \leq m)$ se nazývá *stratum* (úroveň).
Program je m-*stratifikovaný* iff m je nejmenší index takový, že $S_0 \cup \ldots \cup S_m$ je množina všech predikátových symbolů z P.

Věta: Zúplnění každého stratifikovaného programu má Herbrandův model.

Normální a stratifikované programy

Normální program: obsahuje negativní literály v pravidlech.

Problém: existence zúplnění, která nemají žádný model:

$p \leftarrow \neg p$

Definice: Normální program je *stratifikovaný*, právě tehdy, lze-li množinu predikátových symbolů programu P rozdělit do disjunktních množin $S_0, \ldots, S_m$ takových, že platí

$p(\ldots) \leftarrow L_1, \ldots, L_j \in \mathrm{P}$

a $p \in S_k$, $(0 \leq k \leq m)$, pak pro každé $L_i \in \{L_1, \ldots, L_j\}$:

- je-li L_i tvaru $q(\ldots)$, pak q je obsaženo v $S_0 \cup \ldots \cup S_k$;
- je-li L_i tvaru $\neg q(\ldots)$, pak q je obsaženo v $S_0 \cup \ldots \cup S_{k-1}$.

Každé S_i $(0 \leq i \leq m)$ se nazývá *stratum* (úroveň).

Program je m-*stratifikovaný* iff m je nejmenší index takový, že $S_0 \cup \ldots \cup S_m$ je množina všech predikátových symbolů z P.

Věta: Zúplnění každého stratifikovaného programu má Herbrandův model.

Stratifikované programy obecně nemají *jedinečný* minimální Herbrandův model.

SLDNF resoluce pro normální programy

SLDNF odvození
Neúspěšné SLDNF odvození

SLDNF resoluce pro normální programy

SLDNF odvození
Neúspěšné SLDNF odvození

Definice: **SLD$^+$ odvození.** Nechť P je normální program, G_0 normální cíl a $\mathcal{R}$ výpočetní pravidlo. SLD$^+$-odvození G_0 je buď konečná posloupnost

$$\langle G_0; C_0 \rangle, \ldots, \langle G_{i-1}; C_{i-1} \rangle, G_i$$

nebo nekonečná posloupnost

$$\langle G_0; C_0 \rangle, \langle G_1; C_1 \rangle, \langle G_2; C_2 \rangle, \ldots$$

kde v každém kroku $m + 1 (m \geq 0)$, $\mathcal{R}$ vybírá pozitivní literál v G_m a dospívá k G_{m+1} obvyklým způsobem.

Konečné SLD$^+$-odvození může být:

(1) *Úspěšné.* $G_i = \square$

(2) *Neúspěšné.*

(3) *Blokované:* G_i je negativní (např. $\neg A$).

SLDNF resoluce pro normální programy II

Definice: **Úrovně cílů.** Nechť P je normální program, G_0 normální cíl a $\mathcal{R}$ výpočetní pravidlo. **Úroveň** G_0 je

- $\mathbf{0}$ $\Leftrightarrow$ žádné SLD$^+$-odvození s pravidlem $\mathcal{R}$ není blokováno.
- $\boldsymbol{k+1}$ $\Leftrightarrow$ maximální úroveň cílů $\leftarrow A$ které ve tvaru $\neg A$ blokují SLD$^+$-odvození G_0 je k.

SLDNF odvození

Definice: SLDNF-odvození. Nechť P je normální program, G_0 normální cíl a $\mathcal{R}$ výpočetní pravidlo. Množina SLDNF-odvození a podmnožina neúspěšných SLDNF-odvození cíle G_0 jsou takové nejmenší množiny, že:

- Každé SLD$^+$-odvození G_0 je SLDNF-odvození G_0.
- Je-li SLD$^+$-odvození $\langle G_0; C_0\rangle, \ldots, G_i$ blokováno na $\neg A$ a úroveň cíle $\leftarrow A$ je k, pak
  - je-li každé úplné SLDNF-odvození $\leftarrow A$ neúspěšné (pod $\mathcal{R}$), pak $\langle G_0; C_0\rangle, \ldots, \langle G_i, @\rangle, (\leftarrow L_1, \ldots, L_{m-1}, L_{m+1}, \ldots, L_n)$ je SLDNF-odvození cíle G_0 (předpokládáme, že G_i je tvaru $\leftarrow L_1, \ldots, L_{m-1}, \neg A, L_{m+1}, \ldots, L_n$); symbol „@“ označuje prázdnou cílovou substituci;
  - existuje-li SLDNF-odvození $\leftarrow A$ (pod $\mathcal{R}$) s prázdnou cílovou substitucí, pak $\langle G_0; C_0\rangle, \ldots, G_i$ je neúspěšné SLDNF-odvození.
- Je-li $\langle G_0; C_0\rangle, \ldots, G_i$ SLDNF-odvození G_0 a $\langle G_i; C_i\rangle, \ldots$ je SLDNF-odvození G_i, pak „složení“ $\langle G_0; C_0\rangle, \ldots, \langle G_i; C_i\rangle, \ldots$ je (nekonečné) SLDNF-odvození G_0.
- Je-li $\langle G_0; C_0\rangle, \ldots, \langle G_{i-1}; C_{i-1}\rangle, G_i$ SLDNF-odvození cíle G_0 pro každé $i \geq 0$, pak $\langle G_0; C_0\rangle, \langle G_1; C_1\rangle, \ldots$ je nekonečné SLDNF-odvození.

SLDNF odvození II

Konečné SLDNF-odvození může být:

(1) *Úspěšné:* $G_i = \Box$.

(2) *Neúspěšné.*

(3) *Uvázlé (flounder):* G_i je negativní $(\neg A)$ a $\leftarrow A$ je úspěšné s neprázdnou cílovou substitucí.

(4) *Blokované:* G_i je negativní $(\neg A)$ a $\leftarrow A$ nemá konečnou úroveň.

SLDNF odvození – vlastnosti

Věta: Správnost SLDNF-odvození. Nechť P je normální program, $\leftarrow B$ je normální cíl a $\mathcal{R}$ je výpočetní pravidlo. Je-li θ cílová substituce SLDNF-odvození cíle $\leftarrow B$, pak $B\theta$ je logickým důsledkem comp(P).

SLDNF odvození – vlastnosti

Věta: Správnost SLDNF-odvození. Nechť P je normální program, $\leftarrow B$ je normální cíl a $\mathcal{R}$ je výpočetní pravidlo. Je-li θ cílová substituce SLDNF-odvození cíle $\leftarrow B$, pak $B\theta$ je logickým důsledkem comp(P).

Úplnost: – SLDNF-odvození **není** úplné.

Implementace Prologu

Základní pojmy

Konečná množina klauzulí `Hlava :- Tělo` tvoří **program** `P`.

`Hlava` je literál

`Tělo` je (eventálně prázdná) konjunkce literálů $T_1, \ldots T_a$, $a \geq 0$

Literál je tvořen m-árním predikátovým symbolem (m/p) a m termy (argumenty)

Term je konstanta, proměnná nebo složený term.

Složený term tvoří funktor f s aritou n (píšeme f/n) 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

 $\vdots$

 `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`$_1$`,...,B`$_n$ $(n \geq 0)$ z programu `P` takovou, že $\exists\sigma$: `A`σ = `A'`σ; σ je nejobecnější unifikátor.
Pokud neexistuje `A'` nebo σ, ukonči cyklus.

(4) Nahraď `A` v `S` cíli `B`$_1$ až `B`$_n$.

(5) Aplikuj σ na `G` a `S`.

(6) `end while`

(7) Pokud `S==empty`, pak výpočet úspěšně skončil a výstupem je `G` se všemi aplikovanými substitucemi.
Pokud `S!=empty`, výpočet končí neúspěchem.

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`,...,`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) Nahraď `A` v `S` cíli B_1 až B_n.

(5) Aplikuj σ na `G` a `S`.

(6) `end while`

(7) Pokud `S==empty`, pak výpočet úspěšně skončil a výstupem je `G` se všemi aplikovanými substitucemi.
Pokud `S!=empty`, výpočet končí neúspěchem.

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 interpetu

(1) **Selekční pravidlo:** Výběr cíle `A` z množiny cílů `S`

(2) **Způsob prohledávání stromu výpočtu:** Výběr klauzule `A'` z programu `P`.

Nedeterminismus interpetu

(1) **Selekční pravidlo:** Výběr cíle `A` z množiny cílů `S`

(2) **Způsob prohledávání stromu výpočtu:** Výběr klauzule `A'` z programu `P`.

Vztah k úplnosti:

(1) Selekční pravidlo neovlivňuje úplnost (možno zvolit libovolné)

(2) Prohledávání stromu výpočtu do šířky nebo do hloubky

Nedeterminismus interpetu

(1) **Selekční pravidlo:** Výběr cíle `A` z množiny cílů `S`

(2) **Způsob prohledávání stromu výpočtu:** Výběr klauzule `A'` z programu `P`.

Vztah k úplnosti:

(1) Selekční pravidlo neovlivňuje úplnost (možno zvolit libovolné)

(2) Prohledávání stromu výpočtu do šířky nebo do hloubky

„Prozření“ – automatický výběr správné klauzule

- Prohledávání do šířky:
  (1) Vybereme všechny klauzule `A'`$_i$, které je možno unifikovat s literálem `A`. Nechť je těchto klauzulí q.
  (2) Vytvoříme q kopií množiny `S`
  (3) V každé kopii redukujeme `A` jednou z klauzulí `A'`$_i$.
  (4) V následujících kopiích redukujme všechny množiny `S`$_i$ současně.
  (5) Výpočet ukončíme úspěchem, pokud se alespoň jedna z množim `S`$_i$ stane prázdnou.

- Prohledávání do šířky:

(1) Vybereme všechny klauzule `A'`$_i$, které je možno unifikovat s literálem `A`. Nechť je těchto klauzulí q.

(2) Vytvoříme q kopií množiny `S`

(3) V každé kopii redukujeme `A` jednou z klauzulí `A'`$_i$.

(4) V následujících kopiích redukujme všechny množiny `S`$_i$ současně.

(5) Výpočet ukončíme úspěchem, pokud se alespoň jedna z množim `S`$_i$ stane prázdnou.

- 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.

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 (atomy):**
  - konstanty
  - čísla
  - volná proměnná
  - odkaz (reference)

Reprezentace objektů

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

Typy objektů

- **Primitivní (atomy):**
  - konstanty
  - čísla
  - volná proměnná
  - odkaz (reference)
- **Složené (strukturované) objekty:**
  - Struktury
  - Seznamy

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

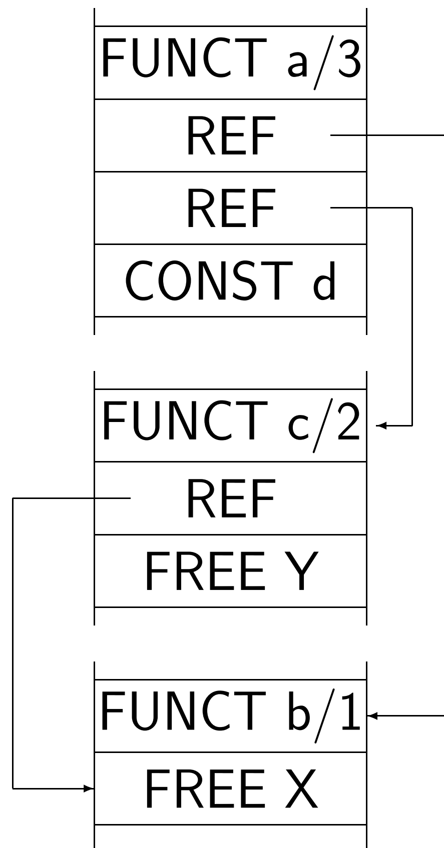
Složené objekty:

- Kopírování struktur
- Sdílení struktur

Kopírování struktur

Příklad:

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

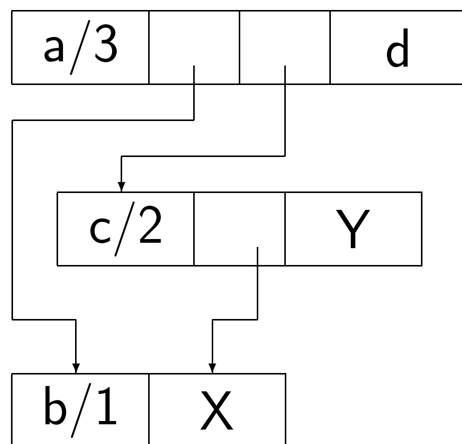


Kopírování struktur II

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 DAG (Directed Acyclic Graph):



Vykopírována každá instance ⟹ **kopírování struktur**

Termy ukládány na **globální zásobník**

Sdílení struktur

Instance termu:

`< kostra_termu; rámec >`

`kostra_termu` je zdrojový term s očíslovanými proměnnými

`rámec` 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

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

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)

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` $\Longrightarrow$ **Sdílení struktur**

Srovnání

Naivní: sdílení pamět'ově méně náročné

Srovnání

Naivní: sdílení paměťově méně náročné

Platí pouze pro rozsáhlé termy přítomné ve zdrojovém kódu

Srovnání

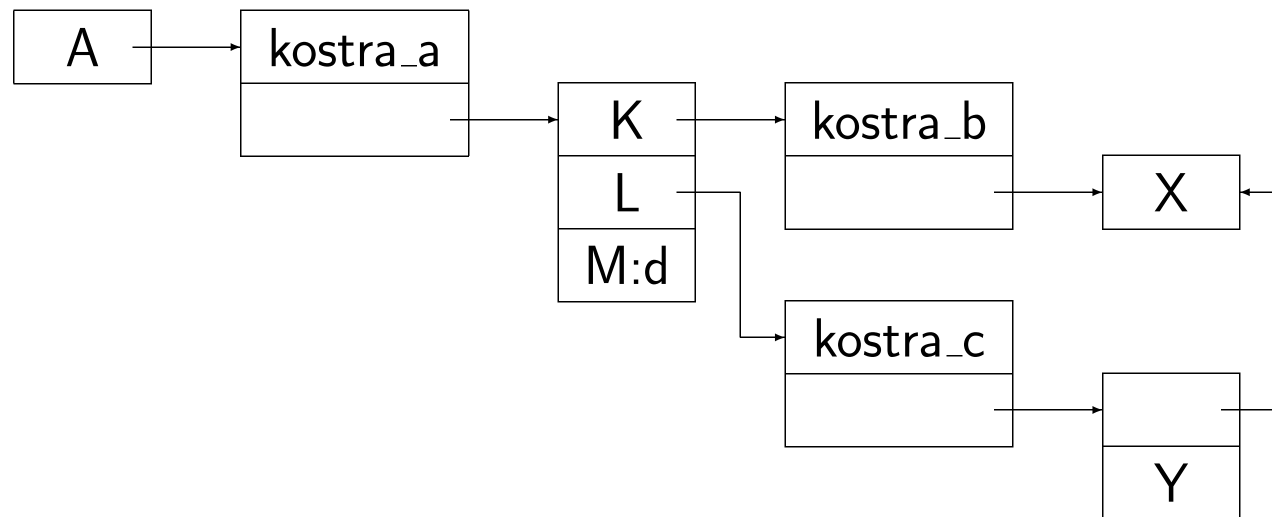
Naivní: sdílení paměťově méně náročné

Platí pouze pro rozsáhlé termy přítomné ve zdrojovém kódu

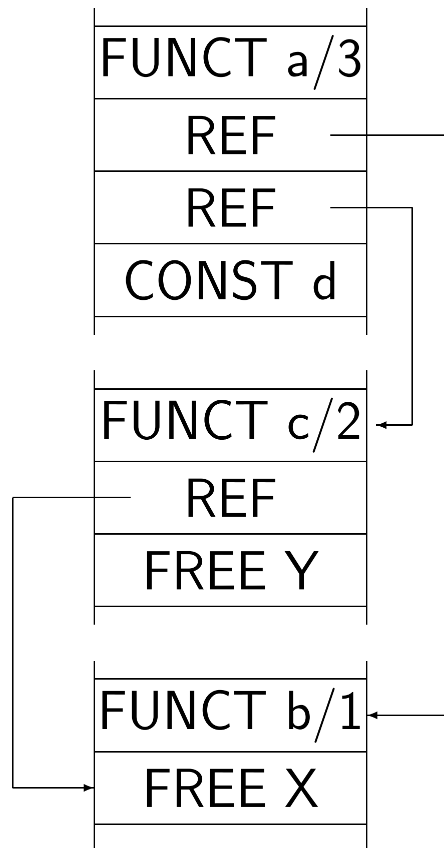
Postupná tvorba termů:

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

Sdílení termů:



Kopírování struktur:



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

Srovnání II

Lokalita přístupů do paměti

- Sdílení struktur: přístupy rozptýleny po paměti
- Kopírování struktur: lokalizované přístupy

Srovnání II

Lokalita přístupů do paměti

- Sdílení struktur: přístupy rozptýleny po paměti
- Kopírování struktur: lokalizované přístupy

Z praktického hlediska neexistuje mezi těmito přístupy zásadní rozdíl.

Řízení výpočtu

- Dopředný výpočet
  Po úspěchu (úspěšná redukce)
  Klasické volání rekurzivních procedur
- Zpětný výpočet (backtracking)
  Po neúspěchu vyhodnocení literálu (neúspěšná redukce)

Aktivační záznam

- Sdílení kódu: Aktivace sdílí společný kód
  liší se obsahem **aktivačního záznamu**
- **Lokální zásobník**
- Dopředný výpočet
  - Stav výpočtu v okamžiku volání procedury
  - Lokální proměnné
  - Pomocné proměnné (à la registry)

Aktivační záznam

- Sdílení kódu: Aktivace sdílí společný kód
  liší se obsahem **aktivačního záznamu**
- **Lokální zásobník**
- Dopředný výpočet
  - Stav výpočtu v okamžiku volání procedury
  - Lokální proměnné
  - Pomocné proměnné (à la registry)
- Zpětný výpočet (backtracking)
  - Hodnoty parametrů v okamžiku zavolání procedury
  - Následující klauzule pro zpracování při neúspěchu

Aktivační záznam

- Sdílení kódu: Aktivace sdílí společný kód
  liší se obsahem **aktivačního záznamu**
- **Lokální zásobník**
- Dopředný výpočet
  - Stav výpočtu v okamžiku volání procedury
  - Lokální proměnné
  - Pomocné proměnné (à la registry)
- Zpětný výpočet (backtracking)
  - Hodnoty parametrů v okamžiku zavolání procedury
  - Následující klauzule pro zpracování při neúspěchu
- **Stopa** (trail)
  - Adresy instanciovaných proměnných
  - Odkaz na aktuální vrchol uchováván v aktivačním záznamu procedury

  Roll-back (obnovení původních) hodnot proměnných po neúspěchu

Okolí a bod volby

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

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 II

Důsledky:

- Samostatná práce s každou částí aktivačního záznamu (optimalizace)

Okolí a bod volby II

Důsledky:

- Samostatná práce s každou částí aktivačního záznamu (optimalizace)
- Alokace pouze okolí pro deterministické procedury

Okolí a bod volby II

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 půrocedury)

Řez

Odstranění alternativních větví výpočtu

⇒ odstranění odpovídajících bodů volby

⇒ změna ukazatele na „nejmladší“ bod volby

Řez

Odstranění alternativních větví výpočtu

⇒ odstranění odpovídajících bodů volby

⇒ změna ukazatele na „nejmladší“ bod volby

⇒ Vytváření deterministických procedur

⇒ Optimalizace využití zásobníku

Interpret

Základní principy:

- Klauzule uloženy jako termy
- **Programová databáze** – charakter haldy
- Klauzule prořetězeny podle pořadí načtení

Interpret

Základní principy:

- Klauzule uloženy jako termy
- **Programová databáze** – charakter haldy
- Klauzule prořetězeny podle pořadí načtení

Vyhodnocení dotazu: volání procedur řízené unifikací

Interpret – Základní princip

(1) Vyber redukovaný literál („první“, tj. nejlevější literál cíle)

(2) Lineárním průchodem od začátku databáze najdi klauzuli, jejíž hlava má stejný funktor a stejný počet argumentů jako redukovaný literál

(3) V případě nalezení klauzule založ bod volby procedury

(4) Založ dále okolí klauzule (velikost odvozena od počtu lokálních proměnných v klauzuli)

(5) Proveď unifikaci literálu a hlavy klauzule

(6) V případě úspěchu přidej všechny literály klauzule k cíli („doleva“, tj. na místo redukovaného literálu).
Pokud je tělo prázdné, výpočet se s úspěchem vrací do klauzule, jejíz adresa je v aktuálním okolí.

(7) V případě neúspěchu unifikace se z bodu volby obnoví stav a pokračuje se v hledání další vhodné klauzule v databázi.

(8) Pokud není nalezena odpovídající klauzule, výpočet se vrací na předchozí bod volby (krátí se lokální i globální zásobník).

(9) Výpočet končí úspěchem, jsou-li úspěšně redukovány všechny literály v cíli.

(10) Výpočet končí neúspěchem, pokud již neexistuje bod volby, k němuž by se výpočet mohl vrátit.

Interpret – vlastnosti

- Lokální i globální zásobník
  - při dopředném výpočtu roste
  - při zpětném výpočtu se zmenšuje

  Lokální zásobník se může zmenšit při dopředném úspěšném výpočtu deterministické procedury.

Interpret – vlastnosti

- Lokální i globální zásobník
  - při dopředném výpočtu roste
  - při zpětném výpočtu se zmenšuje

  Lokální zásobník se může zmenšit při dopředném úspěšném výpočtu deterministické procedury.

- Unifikace argumentů hlavy – obecný unifikační algoritmus
  Současně poznačí adresy unifikovaných proměnných do stopy.

Interpret – vlastnosti

- Lokální i globální zásobník
  - při dopředném výpočtu roste
  - při zpětném výpočtu se zmenšuje

  Lokální zásobník se může zmenšit při dopředném úspěšném výpočtu deterministické procedury.

- Unifikace argumentů hlavy – obecný unifikační algoritmus
  Současně poznačí adresy unifikovaných proměnných do stopy.

„Interpret“:

```
interpret(Query, Vars) :- call(Query), success(Query, Vars).
interpret(_,_) :- failure.
```

Optimalizace: Indexace

- Provázání klauzulí se stejným funktorem a aritou hlavy (tvoří jednu **procedúru**)
- Hash tabulka pro vyhledání první klauzule
- Možno rozhodnout (parciálně) determinismus procedury

Indexace argumentů

```
a(1) :- q(1).
a(a) :- b(X).
a([A|T]) :- c(A,T).
```

- Obecně nedeterministická
- Při volání s alespoň částečně instanciovaným argumentem vždy deterministická (pouze jedna klauzule může uspět)

Indexace argumentů

```
a(1) :- q(1).
a(a) :- b(X).
a([A|T]) :- c(A,T).
```

- Obecně nedeterministická
- Při volání s alespoň částečně instanciovaným argumentem vždy deterministická (pouze jedna klauzule může uspět)
- Indexace podle prvního argumentu
  Základní typy zřetězení:
  - podle pořadí klauzulí (aktuální argument je volná proměnná)
  - dle konstant (aktuální je argument konstanta)
  - formální argument je seznam (aktuální argument je seznam)
  - dle struktur (aktuální argument je struktura)

  Seznam speciální případ struktury.

Indexace argumentů II

- Složitější indexační techniky
  - Podle všech argumentů
  - Podle nejvíce diskriminujícího argumentu
  - Kombinace argumentů (indexové techniky z databází)

Tail Recursion Optimization, TRO

Iterace prováděna pomocí rekurze ⇒ lineární paměťová náročnost cyklů

Tail Recursion Optimization, TRO

Iterace prováděna pomocí rekurze ⇒ lineární paměťová náročnost cyklů

TRO:

Okolí se odstraní *před* rekurznivním voláním posledního literálu klauzule, pokud je klauzule resp. její volání deterministické.

Řízení se nemusí vracet:

- V případě úspěchu se rovnou pokračuje
- V případě neúspěchu se vrací na předchozí bod volby („nad“ aktuální klauzulí)

Rekurzivně volaná klauzule může být volána přímo z kontextu volající klauzule.

TRO – příklad

Program:

```
append([], L, L).
append([A|X], L, [A|Y]) :- append(X, L, Y).
```

Dotaz:

```
?- append([a,b,c], [x], L).
```

TRO – příklad

Program:

```
append([], L, L).
append([A|X], L, [A|Y]) :- append(X, L, Y).
```

Dotaz:

```
?- append([a,b,c], [x], L).
```

append volán rekurzivně 4krát

- Bez TRO: 4 okolí, lineární paměťová náročnost
- S TRO: 1 okolí, konstatní paměťová náročnost

Last Call Optimization

TRO pouze speciální případ
obecné optimalizace posledního volání (LCO)

Last Call Optimization

TRO pouze speciální případ
obecné optimalizace posledního volání (LCO)

Okolí (před redukcí posledního literálu) odstraňováno vždy, když leží na vrcholu zásobníku.

Last Call Optimization

TRO pouze speciální případ
obecné optimalizace posledního volání (LCO)

Okolí (před redukcí posledního literálu) odstraňováno vždy, když leží na vrcholu zásobníku.

Nutné úpravy interpretu

- Disciplina směrování ukazatelů
  - Vždy „mladší“ ukazuje na „starší“
  - Z lokálního do globálního zásobníku

  Odstranění „visících odkazů“ (dangling references)

Last Call Optimization

TRO pouze speciální případ
obecné optimalizace posledního volání (LCO)

Okolí (před redukcí posledního literálu) odstraňováno vždy, když leží na vrcholu zásobníku.

Nutné úpravy interpretu

- Disciplina směrování ukazatelů
  - Vždy „mladší“ ukazuje na „starší“
  - Z lokálního do globálního zásobníku

  Odstranění „visících odkazů“ (dangling references)
- „Globalizace“ lokálních proměnných: lokální proměnné posledního literálu
  - Nutno přesunout na globální zásobník
  - Pouze pro neinstanciované proměnné

Překlad

Motivace:

- Dosažení vyšší míry optimalizace
- Kompaktní kód
- Částečná nezávislost na hardware

Překlad

Motivace:

- Dosažení vyšší míry optimalizace
- Kompaktní kód
- Částečná nezávislost na hardware

Etapy překladu:

(1) Zdrojový text $\Rightarrow$ kód abstraktního počítače

(2) Kód abstraktního počítače $\Rightarrow$ kód (instrukce) cílového počítače

Překlad

Motivace:

- Dosažení vyšší míry optimalizace
- Kompaktní kód
- Částečná nezávislost na hardware

Etapy překladu:

(1) Zdrojový text $\Rightarrow$ kód abstraktního počítače

(2) Kód abstraktního počítače $\Rightarrow$ kód (instrukce) cílového počítače

Výhody:

- Snazší přenos jazyka (nutno přepsat jen druhou část)
- Kód abstraktního počítače možno navrhnout s ohledem na jednoduchost překladu; prostor pro strojově nezávislou optimalizaci

Překlad

Motivace:

- Dosažení vyšší míry optimalizace
- Kompaktní kód
- Částečná nezávislost na hardware

Etapy překladu:

(1) Zdrojový text ⇒ kód abstraktního počítače

(2) Kód abstraktního počítače ⇒ kód (instrukce) cílového počítače

Výhody:

- Snazší přenos jazyka (nutno přepsat jen druhou část)
- Kód abstraktního počítače možno navrhnout s ohledem na jednoduchost překladu; prostor pro strojově nezávislou optimalizaci

Překlad Prologu založen na principu existence abstraktního počítače
V dalším se věnujeme jeho odvození a vlastnostem

Parciální vyhodnocení

- Forma zpracování programu, tzv. transformace na úrovni zdrojového kódu
- Dosazení známých hodnot vstupních parametrů a vyhodnocení všech operací nad nimi.

Parciální vyhodnocení – příklad

```
a(X,Y) :- b(X), c(X,Y).
a(X,Y) :- b(Y), c(Y,X).

b(1). b(2). b(3). b(4).

c(1,2). c(1,3). c(1,4).
c(2,3). c(2,4). c(3,4).
```

Dotaz: `?- a(2,Z).` lze společně s uvedeným programem parciálně vyhodnotit na nový program

```
a'(3).   a'(4).   a'(1).
```

a nový dotaz

```
?- a'(Z).
```

Parciální vyhodnocení – příklad

```
a(X,Y) :- b(X), c(X,Y).
a(X,Y) :- b(Y), c(Y,X).

b(1). b(2). b(3). b(4).

c(1,2). c(1,3). c(1,4).
c(2,3). c(2,4). c(3,4).
```

Dotaz: `?- a(2,Z).` lze společně s uvedeným programem parciálně vyhodnotit na nový program

```
a'(3).   a'(4).   a'(1).
```

a nový dotaz

```
?- a'(Z).
```

Je evidentní, že dotaz nad parciálně vyhodnoceným programem bude zpracován mnohem rychleji (efektivněji) než v případě původního programu.

Parciální vyhodnocení II

Konstrukce překladače: parciálním vyhodnocením interpretu

Parciální vyhodnocení II

Konstrukce překladače: parciálním vyhodnocením interpretu

Problémy:

- Příliš složitá operace (vyhodnocení se musí provést vždy znovu pro každý nový program)
- Výsledný program příliš rozsáhlý
- Nedostatečná dekompozice (zejméně při použití zdrojového jazyka pro implementace i interpretu)

Parciální vyhodnocení II

Konstrukce překladače: parciálním vyhodnocením interpretu

Problémy:

- Příliš složitá operace (vyhodnocení se musí provést vždy znovu pro každý nový program)
- Výsledný program příliš rozsáhlý
- Nedostatečná dekompozice (zejména při použití zdrojového jazyka pro implementace i interpretu)

Vhodnější: využití („ručního“) parciálního vyhodnocení pro návrh abstraktního počítače.

Parciální vyhodnocení Prologu

Cílová operace: unifikace.

Důvod:

- Řízení výpočtu poměrně podrobné i v interpretu
- Unifikace v interpretu atomickou operací
- Unifikace v interpretu nahrazuje řadu podstatně jednodušších operací (testy, přiřazení, . . .)

Parciální vyhodnocení Prologu

Cílová operace: unifikace.

Důvod:

- Řízení výpočtu poměrně podrobné i v interpretu
- Unifikace v interpretu atomickou operací
- Unifikace v interpretu nahrazuje řadu podstatně jednodušších operací (testy, přiřazení, . . .)

Zviditelnění unifikace: transformací zdrojového programu

- Termy reprezentujeme kopírováním struktur na globálním zásobníku
- Parametry procedur jsou vždy umístěny na globální zásobník (predikátem `put/2`) a předávány jsou pouze adresy
- Formálním parametrem procedury jsou pouze volné proměnné, které se v hlavě vyskytují pouze jednou
- Všechny unifikace jsou explicitně zachyceny voláním predikátu `unify/2`

Parciální unifikace Prologu II

Příklad: append/3 s explicitní unifikací:

```
append(A1, A2, A3) :-
    unify(A1,[]),
    unify(A2,L),
    unify(A3,L).

append(A1, A2, A3) :-
    unify(A1,[A|X]),
    unify(A2,L),
    unify(A3,[A|Y]),
    put(X,B1),
    put(L,B2),
    put(Y,B3),
    append(B1,B2,B3).
```

Parciální unifikace Prologu II

Příklad: append/3 s explicitní unifikací:

```
append(A1, A2, A3) :-
    unify(A1,[]),
    unify(A2,L),
    unify(A3,L).

append(A1, A2, A3) :-
    unify(A1,[A|X]),
    unify(A2,L),
    unify(A3,[A|Y]),
    put(X,B1),
    put(L,B2),
    put(Y,B3),
    append(B1,B2,B3).
```

Cíl: parciálně vyhodnotit predikáty `unify/2` a `put/2`

Parciální vyhodnocení – Pomocné termy a predikáty

- term `$addr$(A)` – odkaz na objekt s adresou `A`
- predikát `is_addr(T,P)` – je-li `T` ve tvaru `$addr$(A)`, pak `P` se unifikuje s hodnotou na adrese `A` (jinak predikát selže)
- predikát `:=(V,T)` – přiřadí volné proměnné pgmV term `T`; `V` musí být volná proměnná.
- predikát `repres(A,Tag,Value)` – uloží do proměnné `Tag` příznak a do proměnné `Value` hodnotu slova na adrese `A`.
  `A` musí být adresa na globálním zásobníku, `Tag` i `Value` musí být volné proměnné.
- je-li `A` adresa a `i` celočíselná konstanta, pak výraz `A+i` reprezentuje adresu o `i` slov vyšší (ukazatelová aritmetika)

Parciální vyhodnocení Prologu – odvození instrukcí

`unify(A,T)` unifikuje term na adrese `A` (aktuální parametr) s termem `T` (formální parametr).

Podle hodnoty `T` mohou nastat následující 4 případy:

(1) `T` je volná proměnná: výsledkem je instanciace

```
unify(A,T) :- var(T),
    (var(A), create_var(A);
     true),
     T := $addr$(A).
```

Disjunkce garantuje, že `A` je korektní adresa na globálním zásobníku: nutný run-time test. Lze proto přepsat na

```
unify(A,T) :-
    var(T),
    unify_var(A,T).
```

kde `unify_var/2` vloží do `T` odkaz nebo založí novou proměnnou.

Parciální vyhodnocení Prologu – odvození instrukcí II

(2) T je konstanta: výsledkem je test nebo přiřazení

```
unify(A,T) :- atomic(T),
    ((var(A), create_var(A),
      instantiate_const(A,T));
     (repres(A,Tag,Value),
      Tag == 'FREE',
       instantiate_const(A,T);
      Tag == 'CONST',
       Value == T
    )).
```

kde `instantiate_const/2` uloží do slova s adresou A hodnotu T.
Opět možno přepsat do kompaktního tvaru

```
unify(A,T) :-
    atomic(T),
    unify_const(A,T).
```

kde `unify_const/2` provede příslušný test nebo přiřazení.

Parciální vyhodnocení Prologu – odvození instrukcí III

(3) `T` je složený term: dvoufázové zpracování, v první fázi test nebo založení funktoru,v druhé rekurzivní unifikace argumentů

```
unify(A,T) :-
    struct(T),
    functor(T,F,N),
    unify_struct(F,N,A),
    T =.. [_|Tl],
    unify_args(Tl,A+1).
```

Predikát `unify_struct/3` je analogický výše použitým predikátům `unify_var/2` a `unify_const/2`.

Druhá fáze

```
unify_args([],_).
unify_args([T|Tl], A) :-
    unify(A,T),
    unify_args(Tl,A+1).
```

Parciální vyhodnocení Prologu – odvození instrukcí IV

(4) `T` je odkazem: nutno použít obecnou unifikaci (není žádná informace pro parciální vyhodnocení)

```
unify(A,T) :-
    is_addr(T,P),
    unification(A,P).
```

Parciální vyhodnocení Prologu – odvození instrukcí V

Predikát put/2 je jednodušší (nikdy nepotřebuje unifikaci)

```
put(B,T) :- is_addr(T,B).
put(B,T) :- var(T),
    create_var(B),
    T := $addr$(B).
put(B,T) :- atomic(T),
    create_const(B,T).
put(B,T) :- struct(T),
    create_struct(B,T).
```

Parciální vyhodnocení Prologu – odvození instrukcí VI

Parciální vyhodnocení první klauzule programu append/3 upraví

```
unify(A1,[]) na unify_const(A1,[])
unify(A2,L)  na L:=$addr$(A2)
unify(A3,L)  na is_addr(L,T), unification(T,A3)
```

Posloupnost `L:=$addr$(A2), is_addr(L,T)` odpovídá přejmenování `T` na `A2`.

Parciální vyhodnocení Prologu – odvození instrukcí VII

Výsledný tvar programu append/3

```
append(A1, A2, A3) :-
    unify_const(A1,[]),
    unification(A2,A3).
append(A1, A2, A3) :-
    unify_struct('.',2,A1),
        unify_var(A,A1+1),
        unify_var(X,A1+2),
    unify_var(L,A2),
    unify_struct('.',2,A3),
        unification(A1+1,A3+1),
        unify_var(Y,A3+2),
    append(A1+2,A2,A3+2).
```

Většina původních unifikací převedena na jednodušší operace; unifikace v posledním kroku je nezbytná.

Jiný příklad

```
a(c,s(f),d,X) :- g(X).
```

Procedurální pseudokód (testy a přiřazení) a kód abstraktního počítače:

```
procedure a(X,Y,Z,A) is              | a(A1, A2, A3, A4) :-
    if ( X == 'c' &&                 |     unify_const(c,A1),
         Z == 'd' &&                 |     unify_struct(s,1,A2),
         ( is_struct(Y,'s',1 &&      |         unify_const(f,A2+1),
           first_arg(Y) == 'f' ))    |     unify_const(d,A3),
    then                             |     unify_var(X,A4),
        call g(Z)                    |     g(A4).
    else                             |
        call fail                    |
end procedure
```

Warrenův abstraktní počítač, WAM

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

Warrenův abstraktní počítač, WAM

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

- 5. datových oblastí:
  1. **Oblast kódu** (programová databáze); obsahuje rovněž všechny statické objekty (texty atomů a funktorů apod.)
  2. **Lokální zásobník**
  3. **Stopa**
  4. **Globální zásobník**
  5. **Pomocný zásobník** (Push Down List, PDL); pracovní paměť abstraktního počítače (použitý v unifikaci, syntaktické analýze apod.)

- **Stavové registry**:

  P čitač adres (**P**rogram counter)
  CP adresa návratu (**C**ontinuation **P**ointer)
  E ukazatel na nejmladší okolí (**E**nvironment)
  B ukazatel na nejmladší bod volby (**B**acktrack point)
  TR vrchol stopy (**TR**ail)
  H vrchol haldy (**H**eap)
  HB vrchol haldy v okamžiku založení posledního bodu volby (**H**eap on **B**acktrack point)
  S ukazatel, používaný při analýze složených termů (**S**tructure pointer)
  CUT ukazatel na bod volby, na který se řezem zařízne zásobník

- **Stavové registry**:

P	čitač 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,...`

- Instrukce WAMu

get instrukce		*put instrukce*		*unify instrukce*	
get_var	Ai,Y	put_var	Ai,Y	unify_var	Y
get_value	Ai,Y	put_value	Ai,Y	unify_value	Y
get_const	Ai,C	put_unsafe_value	Ai,Y	unify_local_value	Y
get_nil	Ai	put_const	Ai,C	unify_const	C
get_struct	Ai,F/N	put_nil	Ai	unify_nil	
get_list	Ai	put_struct	Ai,F/N	unify_void	N
		put_list	Ai		

instrukce řízení		*indexační instrukce*			
allocate		try_me_else	Next	try	Next
deallocate		retry_me_else	Next	retry	Next
call	Proc/N,A	trust_me_else	fail	trust	fail
execute	Proc/N				
proceed		cut_last		switch_on_term	L1,L2,L3,L4
		save_cut	Y	switch_on_const	Table
		load_cut	Y	switch_on_struct	Table

WAM – Instrukce

- `get` instrukce – unifikace aktuálních a formálních parametrů
- `put` instrukce – příprava argumentů před voláním podcíle
- `unify` instrukce – zpracování složených termů. Používají registr S jako druhý argument (volání instrukce `unify` zvětší hodnotuS o jedničku)
- indexační instrukce – indexace klauzulí a manipulace s body volby
- instrukce řízení běhu – předávání řízení a explicitní manipulace s okolím.

WAM – indexace

- Provázání klauzulí: instrukce `XX_me_else`:
  - první klauzule: `XX` je `try`; založí bod volby
  - poslední klauzule: `XX` je `trust`; zruší bod volby
  - ostatní klauzule: `XX` je `retry`; znovupoužije bod volby
- 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

Procedure

```
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
```

WAM – řez

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

WAM – řez

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

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

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

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`.

```
a(X,Y) :- b(X), !, c(Y).
a(2,Y) :- !, c(Y).
a(X,Y) :- d(X,Y).
```

dá

```
save_cut Y2; get A2, Y2; 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/1
```

WAM – řízení výpočtu

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

WAM – řízení výpočtu

- `call Proc,N`: zavolá `Proc`, `N` udává počet lokálních proměnných (odpovídá velikosti zásobníku) Možná optimalizace:

  ```
  a(A,B,C,D) :- b(D), c(A,C), d(B), e(A), f.
  allocate
  call b/1,4
  call c/2,3
  call d/2,2
  call e/1,1
  deallocate
  execute f/0
  ```

  Postupně se lokální zásobník zkracuje

WAM – řízení výpočtu

- `call Proc,N`: zavolá `Proc`, `N` udává počet lokálních proměnných (odpovídá velikosti zásobníku) Možná optimalizace:

  ```
  a(A,B,C,D) :- b(D), c(A,C), d(B), e(A), f.
  allocate
  call b/1,4
  call c/2,3
  call d/2,2
  call e/1,1
  deallocate
  execute f/0
  ```

  Postupně se lokální zásobník zkracuje

- `execute Proc`: ekvivalentní příkazu goto

WAM – řízení výpočtu

- `call Proc,N`: zavolá `Proc`, `N` udává počet lokálních proměnných (odpovídá velikosti zásobníku) Možná optimalizace:

  ```
  a(A,B,C,D) :- b(D), c(A,C), d(B), e(A), f.
  allocate
  call b/1,4
  call c/2,3
  call d/2,2
  call e/1,1
  deallocate
  execute f/0
  ```

  Postupně se lokální zásobník zkracuje
- `execute Proc`: ekvivalentní příkazu goto
- `proceed`: zpracování faktů

WAM – řízení výpočtu

- `call Proc,N`: zavolá `Proc`, `N` udává počet lokálních proměnných (odpovídá velikosti zásobníku) Možná optimalizace:

  ```
  a(A,B,C,D) :- b(D), c(A,C), d(B), e(A), f.
  allocate
  call b/1,4
  call c/2,3
  call d/2,2
  call e/1,1
  deallocate
  execute f/0
  ```

  Postupně se lokální zásobník zkracuje

- `execute Proc`: ekvivalentní příkazu goto
- `proceed`: zpracování faktů
- `allocate`: alokuje okolí (pro některé klauzule není třeba, proto explicitně generováno)

WAM – řízení výpočtu

- `call Proc,N`: zavolá `Proc`, `N` udává počet lokálních proměnných (odpovídá velikosti zásobníku) Možná optimalizace:

  ```
  a(A,B,C,D) :- b(D), c(A,C), d(B), e(A), f.
  allocate
  call b/1,4
  call c/2,3
  call d/2,2
  call e/1,1
  deallocate
  execute f/0
  ```

  Postupně se lokální zásobník zkracuje

- `execute Proc`: ekvivalentní příkazu goto
- `proceed`: zpracování faktů
- `allocate`: alokuje okolí (pro některé klauzule není třeba, proto explicitně generováno)
- `deallocate`: uvolní okolí (je-li to možné, tedy leží-li na vrcholu zásobníku)

WAM – optimalizace

- Indexace klauzulí
- generování optimální posloupnosti instrukcí WAMu
- odstranění redundancí při generování cílového kódu.

WAM – optimalizace

- Indexace klauzulí
- generování optimální posloupnosti instrukcí WAMu
- odstranění redundancí při generování cílového kódu.

Příklad:

```
a(X,Y,Z) :-
        b(f,X,Y,Z).
```

Naivní vs. optimalizovaný kód (počet registrů a tedy i počet instrukcí/přesunů v paměti):

```
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   A5,A2     |     execute     b/4
put_value   A6,A3     |
put_value   A7,A4     |
execute     b/4       |
```

WAM – optimalizace II

Mělké navracení

WAM – optimalizace II

Mělké navracení

Detekce determinismu

WAM – zpětný překlad

Vestavěné predikáty `clause/2`, `retract/2`, `listing` a podobné, pracující se zdrojovým kódem.

WAM – zpětný překlad

Vestavěné predikáty `clause/2`, `retract/2`, `listing` a podobné, pracující se zdrojovým kódem.

- Kopie zdrojového kódu součástí programu

WAM – zpětný překlad

Vestavěné predikáty `clause/2`, `retract/2`, `listing` a podobné, pracující se zdrojovým kódem.

- Kopie zdrojového kódu součástí programu
- **Zpětný překlad**

WAM – zpětný překlad

Vestavěné predikáty `clause/2`, `retract/2`, `listing` a podobné, pracující se zdrojovým kódem.

- Kopie zdrojového kódu součástí programu
- **Zpětný překlad**

(1) Vybereme klauzuli a nahradíme všechna volání instrukcí `call`, `execute` a `proceed` novými instrukcemi `de_call`, `de_execute` a `de_proceed` (analogicky s vestavěnými predikáty).

(2) Spustíme vyhodnocení klauzule tím, že ji zavoláme na cíl s volnými proměnnými na místě argumentů.

(3) Instrukce `get` a `put` (a příslušné `unify`) zkontruují argumenty.

(4) Nové instrukce `de_AAA` pak zkonstruují na globáloním zásobníku příslušné termy, odpovídající predikátům (jména jsou známa z argumentů těchto instrukcí).

(5) Instrukce `de_execute` či `de_proceed` ukončí zpětnou kompilaci; na globálním zásobníku je term odpovídající zdrojovému textu klauzule.

Logické programování s omezujícími podmínkami

Constraint Logic Programming

Cíle:

- Využít syntaktické a výrazové přednosti LP
- Dosáhnout kvalitatvně vyšší efektivity ve srovnání s backtrackingem

(Pure) Logic Programming

+:

- Declarativeness
- Simple Programming
- Simple & Clear Semantics
- Declarative = Operational Semantics

−:

- Everything must be mapped to Herbrand Universe
- Low level programming
- Not fully implemented
- Forced explicit representation of objects/results
- Depth-first search procedure
- Generate and test paradigm

Conclusion:

Interesting approach, but not strong enough for all problems.

CLP($\mathcal{A}$) Language family: Inference Mechanism
Constraint Solver(s)

Unification

Decomposable to two parts:

(1) Decision Procedure, i.e., *is* `t=s` *solvable?*

(2) Explicit Solution Representation, i.e., *the mgu of* `t=s`

Unification is a special case of "Constraint" (equality over terms of Herbrand Universe)

Basic idea behind Constraint Logic Programming:

Replace Unification by Constraint Satisfaction

This leads to introduction of interpreted objects with predefined semantics

Explicit vs Implicit representation:

- Explicit can be inconvenient, e.g.:

  All numbers between 1 and 1000 but not 17.
  explicit: 1,2,. . . ,16,18,. . . ,1000
  implicit: $X : 1 \leq X \leq 1000 \wedge X \neq 17$

- Explicit is impossible for infinite sets, e.g.:

  All points on the ring (in 2D).
  explicit: impossible
  implicit: $[X, Y] : X * X + Y * Y = r * r$

Consistency Techniques

- More efficient search methods: complement backtracking by **consistency techniques**.
- Constraints are evaluated deterministically *before* a nondeterministic process of a search.
- "Constrain and generate" instead of "Generate and test"

```
sort(L, S) :-
     permute(L, S),
     sorted(S).
```

```
sort(L, S) :-
     sorted(S),
     permute(L, S).
```

```
sorted([]).
sorted([_]).
sorted([A,B|T]) :-
     A >= B,
     sorted([B|T]).
```

$\mathcal{O}(n!)$ $\mathcal{O}(n^2)$

Logic Programs

- **Domain of computation:**
  - a Herbrand Universe (i.e., uninterpreted terms)
- **Syntax:**
  - definite clauses
- **Operational interpretation:**
  - unification
  - SLD resolution
- **Declarative interpretation:**
  - Truth obtained by successful derivations
  - Falsity obtained by finitely failed derivations
- **Output:**
  - an mgu

Constraint Logic Programs

- **Domain of computation:**
  - any ***solution compact*** structure $\mathcal{A}$
- **Syntax:**
  - definite clauses **with constraints**
- **Operational interpretation:**
  - constraint solving
  - SLD resolution
- **Declarative interpretation:**
  - Truth obtained by successful derivations
  - Falsity obtained by finitely failed derivations
- **Output:**
  - constraints

There is no restriction to

- Herbrand Universe
- Unification
- Equations

From Unification to Constraints

UNIFICATION

- Equality Theory

- The question:
  - is `t=s` solvable?
  - what are the solutions?
- The required answer:
  - a complete set U of mgu's

Problems:

- E cannot capture uncountable structures
- E cannot naturally capture some countable structures (e.g., the rational trees of Prolog-II)
- U may be infinite
- U may not exists

From Unification to Constraints

CONSTRAINTS

- Theory $\mathcal{T}$ and (algebraic) structure $\mathcal{A}$
- $\mathcal{T}$ is **satisfaction-complete** w.r.t. $\mathcal{A}$:
  $\mathcal{T} \models c \quad \Leftrightarrow \mathcal{A} \models c$ and
  $\mathcal{T} \models \neg c \Leftrightarrow \mathcal{A} \models \neg c$
- $\mathcal{A}$ is **solution-compact**, i.e.:
  $\forall a \in \mathcal{A},\ a = \cap c_i$
  $\forall a,\ \neg a = \cup c_i$
  for some finite or infinite set of constraints c_i

- The question:
  - is c solvable?
- The required answer:
  - YES or NO

A **structure** $\mathcal{A}$ consists of

- a set $D(\mathcal{A})$, called the *domain* of $\mathcal{A}$
- a collection Σ of n-ary *functions*:
  $\mathcal{A}^n \rightarrow \mathcal{A}$
- a collection Π (which constains $= /2$) of n-ary *relations*:
  $\mathcal{A}^n \rightarrow \{\text{TRUE, FALSE}\}$

Constraints

- a $\mathcal{A}$-*term* is either

  (1) a constant $t \in \Sigma$, or

  (2) a variable, e.g. V, or

  (3) term of the form $f(V_1, \ldots, V_n)$ where $f \in \Sigma$ is n-ary functor and $V_1, \ldots, V_n$ are $\mathcal{A}$-terms.

- a $\mathcal{A}$-*constraint* of the form $p(V_1, \ldots, V_n)$ where $p \in \Pi$ is n-ary and $V_1, \ldots, V_n$ are $\mathcal{A}$-terms.

Solution Compactness

A structure $\mathcal{A}$ is *solution compact* if

- Every element in $D(\mathcal{A})$ is definable by a (not necessarily finite) set of $\mathcal{A}$-constraints:

$$\forall d \in D(\mathcal{A}),\ \exists C_i :\ d = \cap C_i$$

- Every two distinct elements in $D(A)$ are separable by two disjoint finite sets of $\mathcal{A}$-constraints:

$$\forall e,\ \exists C_i :\ \neg e = \cup C_i$$

Solution compact structures $\mathcal{A}$ are such that any element in $D(\mathcal{A})$ can be *approximate* to any accuracy by a (possibly infinite) set of constraints.

Solution Compact Structures – Examples

The PROLOG Structure:

- the Herbrand Universe: $\{a, f(a), f(f(a)), f(f(f(a))), \ldots\}$
- $\Sigma = \{a, f\}$ where
  a is a constant and f is a map: $a \mapsto f(a), f(a) \mapsto f(f(a)), \ldots$
- Π is simply $\{=\}$

This structure is (trivially) solution compact because it is countable.

A PROLOG-WITH-EQUALITY Structure:

Let E be an equality theory.

- the domain: T(Σ)/E
- $\Sigma =$ some functor alphabet
- Π is simply $\{=\}$

This structure is again (trivially) solution compact because it is countable.

The Structure of Natural Numbers:

- the natural numbers: $\{0, 1, 2, \ldots\}$
- $\Sigma = \{0, 1, +, -\}$
- $\Pi = \{=, \leq, <\}$

Solution compactness is immediate

The CLP Scheme

CLP($\mathcal{A}$) is a generic language
each instance of the Scheme is a programming language, obtained by specifying a *solution-compact* (possibly many sorted) structure.

A CLP($\mathcal{A}$) program consists of a finite number of rules:

$$\boxed{A_0 :- c_1, c_2, \ldots, c_n, A_1, A_2, \ldots, A_m}$$

where

$n \geq 0,\ m \geq 0,$

c_i are *constraints* over the $\mathcal{A}$-relations and $\mathcal{A}$-terms,

A_i are *atoms* over $\mathcal{A}$-terms.

A CLP($\mathcal{A}$) query has the form:

$$?!- c_1, c_2, \ldots, c_n, A_1, A_2, \ldots, A_m$$

where also $n + m > 0$

Operational Model

- Subgoal Selection Strategy
  $\mathcal{P}$ is a program
  $\mathcal{G}$ is a goal with
  - atoms $A = \{a_1, \ldots, a_n\}$ $(n \geq 0)$
  - solved constraints $C = \{c_1, \ldots, c_m\}$ $(m \geq 0)$
  - delayed constraints $D = \{d_1, \ldots, d_k\}$ $(k \geq 0)$.
- Derivation step $\mathcal{G} \rightarrow \mathcal{G}'$:
  - d_i selected: $\mathcal{G}'$ is $\langle A, C \cup d_i, D' \rangle$ where D' is D without d_i. $C \cup d_i$ is solvable.
  - a_i selected: $\mathcal{G}'$ is $\langle A', C, D' \rangle$ where A' is A with the a_i replaced by atoms of the body of clause from $\mathcal{P}$, D' is D plus constraints of the body of clause from $\mathcal{P}$ plus constraint of matching a_i with clause head. C must be solvable.

Difference from Prolog selection strategy:

- no head unification, just construction of delayed constraints
- selection of delayed constraint

Derivation Sequence: sequence of goals connected by derivation steps.

- Successful:
  finite with last $\mathcal{G}$ equals $\langle\emptyset, C, \emptyset\rangle$
- Conditionally Successful:
  finite with last $\mathcal{G}$ equals $\langle\emptyset, C, D\rangle$, $D \neq \emptyset$
- Finitely Failed:
  finite, $\mathcal{G}$ equals $\langle A, C, D\rangle$, $A \neq \emptyset$ and no derivation possible

Answer Constraints: $C \cup D$ in (conditionally) successful sequences.

Answer constraints represent *output* of running CLP program.

Implementation conditions:

- Sufficient expressive power of $D(\mathcal{A})$
- Existence of efficient (and complete) constraint solver
  - Incrementality of the solver
- Large area of potential applications

CLP($\mathcal{R}$) Language

Structure:

- Two Sorted Domain:

  Uninterpreted terms and Real arithmetic terms

- Arithmetic Terms (may not contain functors)

  $2.7828,\ V - 1,\ 6 * V - 3.14 * U * U + Z,\ \ldots$

- Terms with functors (may contain arithmetic terms)

  $b,\ g(a, b),\ g(f(a), h(b)), \ldots$

  $f(2.78283),\ g(3 + V, a),$ complex$(2, 7.14),\ [2, 17.4, 3|$ Tail$],$

- Aritmetic relations: $=,\ <,\ >,\ \leq,\ \geq$

  $3 < 7,\ V + 2 \leq X - 1,\ \ldots$

- Functor relations: $=$

  $f(a) = A,\ g(X) = g(j),\ \ldots$

```
fib(0,1).
fib(1,1).
fib(N,F) :-
      N > 1,
      N1 is N-1,
      N2 is N-2,
      fib(N1,F1),
      fib(N2,F2),
      F is F1+F2.
```

Prolog program

```
fib(0,1).
fib(1,1).
fib(N,X1+X2) :-
      N > 1,
      fib(N-1,X1),
      fib(N-2,X2).
```

CLP($\mathcal{R}$) program

`?- fib(5,X).`

Prolog	CLP($\mathcal{R}$)
`X = 8`	`X = 8`

`?- fib(X,8)`

Prolog	CLP($\mathcal{R}$)
error	`X = 5`

`?- B >= 80, B <= 90, fib(A,B).`

Prolog	CLP($\mathcal{R}$)
error	`A = 10, B = 89`

General scheme of a CLP($\mathcal{A}$) system

CLP($\mathcal{R}$) as an approximation of CLP($\mathcal{A}$) implementation:

- Non-arithmetic equations solved by unification without “occurs-check”.
- “Left-to-right” subgoal selection strategy.
- “Top-to-bottom” rule selection search.
- Floating point representation of real numbers.
- Only linear constraint solver implemented.

Interface module:

- Test simple relations `A op B` where both `A` and `B` are ground arithmetic expressions and op is an interpreted relation.
- Perform simple assignments `V = E` where `E` is a ground arithmetic expression and `V` is a variable.
- Transform to canonical form.

Complication: Equation of the form `X = E` where `X` is a non-solver variable and `E` is non-ground arithmetic term.

a) to bind

- cyclic binding (`X=f(X)`, e.g., `X=3*X+7`)
- thrashing (unnecessary increase of complexity)

b) to pass to solver

Constraint Solver in CLP($\mathcal{R}$)

- The linear equation solver.
- The linear inequality solver.
- The delaying module (dealing with non-linear constraints).

Consequences of delaying non-linear constraints:

- Incompleteness:

  ```
  ?- Y=3/X, X+Y=4.
  ```

  gives the answer

  ```
  X = 4.0-Y,
  3 = Y*(4.0-Y).
  **Maybe
  ```

  while there are two solutions:

  ```
  X = 1, Y = 3,
  X = 3, Y = 1.
  ```

- Incorrectness:

  ```
  ?- Y*Y + 9 = 0.
  ```

  gives the answer

  ```
  -9.0 = Y*Y.
  **Maybe
  ```

  while there is no solution in the real domain.

Precision of Computation

Floating point approximation of real numbers:

- Solver of Sets of Linear Equations
  Partially solvable by advanced methods
- "Chaotic" behaviour

Equations:

$$X_{n+1} = (R+1)X_n - R(X_n X_n) \tag{1}$$
$$X_{n+1} = (R+1)X_n - (RX_n)X_n \tag{2}$$
$$X_{n+1} = ((R+1) - RX_n)X_n \tag{3}$$
$$X_{n+1} = RX_n + (1 - RX_n)X_n \tag{4}$$
$$X_{n+1} = X_n - R(X_n - X_n X_n) \tag{5}$$

```
it(R,I,X1,X2,X3,X4,X5) :-
   eq1(R,X1,Y1),
   eq2(R,X2,Y2),
   eq3(R,X3,Y3),
   eq4(R,X4,Y4),
   eq5(R,X5,Y5),

   print(I,X1,X2,X3,X4,X5),
   it(R,I,Y1,Y2,Y3,Y4,Y5).
```

```
it(R,I,X1,X2,X3,X4,X5) :-
   eq1(R1,X1,Y1),
   eq2(R1,X2,Y2),
   eq3(R1,X3,Y3),
   eq4(R1,X4,Y4),
   eq5(R1,X5,Y5),
   R1 = R,
   print(I,X1,X2,X3,X4,X5),
   it(R,I,Y1,Y2,Y3,Y4,Y5).
```

Results of computation:

Computations for first program

n	Eq. 1	Eq. 2	Eq. 3	Eq. 4	Eq. 5	Expected[a]
0	**0.500000**	**0.500000**	**0.500000**	**0.500000**	**0.500000**	**0.500000**

Results of computation:

Computations for first program

n	**Eq. 1**	**Eq. 2**	**Eq. 3**	**Eq. 4**	**Eq. 5**	**Expected**[a]
0	**0.500000**	**0.500000**	**0.500000**	**0.500000**	**0.500000**	**0.500000**
20	**0.418895**	**0.418895**	**0.418895**	**0.418895**	**0.418895**	**0.418895**

Results of computation:

Computations for first program

n	Eq. 1	Eq. 2	Eq. 3	Eq. 4	Eq. 5	Expected[a]
0	0.500000	0.500000	0.500000	0.500000	0.500000	0.500000
20	0.418895	0.418895	0.418895	0.418895	0.418895	0.418895
50	0.067567	0.063747	0.059988	0.061491	0.061503	0.062236

Results of computation:

Computations for first program

n	Eq. 1	Eq. 2	Eq. 3	Eq. 4	Eq. 5	Expected[a]
0	0.500000	0.500000	0.500000	0.500000	0.500000	0.500000
20	0.418895	0.418895	0.418895	0.418895	0.418895	0.418895
50	0.067567	0.063747	0.059988	0.061491	0.061503	0.062236
80	0.553038	0.817163	0.021952	1.310402	0.018439	0.119640

Results of computation:

Computations for first program

n	Eq. 1	Eq. 2	Eq. 3	Eq. 4	Eq. 5	Expected[a]
0	**0.500000**	**0.500000**	**0.500000**	**0.500000**	**0.500000**	**0.500000**
20	**0.418895**	**0.418895**	**0.418895**	**0.418895**	**0.418895**	**0.418895**
50	**0.067567**	**0.063747**	**0.059988**	**0.061491**	**0.061503**	**0.062236**
80	**0.553038**	**0.817163**	**0.021952**	**1.310402**	**0.018439**	**0.119640**
100	**0.000067**	**0.119457**	**1.256496**	**0.658047**	**1.042823**	**0.742887**

Results of computation:

Computations for first program

n	**Eq. 1**	**Eq. 2**	**Eq. 3**	**Eq. 4**	**Eq. 5**	**Expected**[a]
0	**0.500000**	**0.500000**	**0.500000**	**0.500000**	**0.500000**	**0.500000**
20	**0.418895**	**0.418895**	**0.418895**	**0.418895**	**0.418895**	**0.418895**
50	**0.067567**	**0.063747**	**0.059988**	**0.061491**	**0.061503**	**0.062236**
80	**0.553038**	**0.817163**	**0.021952**	**1.310402**	**0.018439**	**0.119640**
100	**0.000067**	**0.119457**	**1.256496**	**0.658047**	**1.042823**	**0.742887**

Computations for second program

n	**Eq. 1**	**Eq. 2**	**Eq. 3**	**Eq. 4**	**Eq. 5**	**Expected**[a]
0	**0.500000**	**0.500000**	**0.500000**	**0.500000**	**0.500000**	**0.500000**

Results of computation:

Computations for first program

n	**Eq. 1**	**Eq. 2**	**Eq. 3**	**Eq. 4**	**Eq. 5**	**Expected**[a]
0	**0.500000**	**0.500000**	**0.500000**	**0.500000**	**0.500000**	**0.500000**
20	**0.418895**	**0.418895**	**0.418895**	**0.418895**	**0.418895**	**0.418895**
50	**0.067567**	**0.063747**	**0.059988**	**0.061491**	**0.061503**	**0.062236**
80	**0.553038**	**0.817163**	**0.021952**	**1.310402**	**0.018439**	**0.119640**
100	**0.000067**	**0.119457**	**1.256496**	**0.658047**	**1.042823**	**0.742887**

Computations for second program

n	**Eq. 1**	**Eq. 2**	**Eq. 3**	**Eq. 4**	**Eq. 5**	**Expected**[a]
0	**0.500000**	**0.500000**	**0.500000**	**0.500000**	**0.500000**	**0.500000**
20	**0.418895**	**0.418895**	**0.418895**	**0.418895**	**0.418895**	**0.418895**

Results of computation:

Computations for first program

n	Eq. 1	Eq. 2	Eq. 3	Eq. 4	Eq. 5	Expected[a]
0	0.500000	0.500000	0.500000	0.500000	0.500000	0.500000
20	0.418895	0.418895	0.418895	0.418895	0.418895	0.418895
50	0.067567	0.063747	0.059988	0.061491	0.061503	0.062236
80	0.553038	0.817163	0.021952	1.310402	0.018439	0.119640
100	0.000067	0.119457	1.256496	0.658047	1.042823	0.742887

Computations for second program

n	Eq. 1	Eq. 2	Eq. 3	Eq. 4	Eq. 5	Expected[a]
0	0.500000	0.500000	0.500000	0.500000	0.500000	0.500000
20	0.418895	0.418895	0.418895	0.418895	0.418895	0.418895
50	0.061503	0.061503	0.061485	0.060712	0.061503	0.062236

Results of computation:

Computations for first program

n	Eq. 1	Eq. 2	Eq. 3	Eq. 4	Eq. 5	Expected[a]
0	0.500000	0.500000	0.500000	0.500000	0.500000	0.500000
20	0.418895	0.418895	0.418895	0.418895	0.418895	0.418895
50	0.067567	0.063747	0.059988	0.061491	0.061503	0.062236
80	0.553038	0.817163	0.021952	1.310402	0.018439	0.119640
100	0.000067	0.119457	1.256496	0.658047	1.042823	0.742887

Computations for second program

n	Eq. 1	Eq. 2	Eq. 3	Eq. 4	Eq. 5	Expected[a]
0	0.500000	0.500000	0.500000	0.500000	0.500000	0.500000
20	0.418895	0.418895	0.418895	0.418895	0.418895	0.418895
50	0.061503	0.061503	0.061485	0.060712	0.061503	0.062236
80	0.018439	0.018439	1.200312	1.184312	0.017072	0.119640

Results of computation:

Computations for first program

n	Eq. 1	Eq. 2	Eq. 3	Eq. 4	Eq. 5	Expected[a]
0	0.500000	0.500000	0.500000	0.500000	0.500000	0.500000
20	0.418895	0.418895	0.418895	0.418895	0.418895	0.418895
50	0.067567	0.063747	0.059988	0.061491	0.061503	0.062236
80	0.553038	0.817163	0.021952	1.310402	0.018439	0.119640
100	0.000067	0.119457	1.256496	0.658047	1.042823	0.742887

Computations for second program

n	Eq. 1	Eq. 2	Eq. 3	Eq. 4	Eq. 5	Expected[a]
0	0.500000	0.500000	0.500000	0.500000	0.500000	0.500000
20	0.418895	0.418895	0.418895	0.418895	0.418895	0.418895
50	0.061503	0.061503	0.061485	0.060712	0.061503	0.062236
80	0.018439	0.018439	1.200312	1.184312	0.017072	0.119640
100	0.878859	0.878859	0.028512	1.230244	0.437689	0.742887

[a] Values computed using rationals with up to 10000 digits.

Similar Approaches

- CLP(Σ^*) over regular sets
  - Regular Set Inclusion
  - Regular Set Decomposition
- CLPS over sets
  - Set unification
  - Set Membership
  - Three basic interpreted functors
  - All non-ground set relations delayed

- CLP(Boolean)
  - Boolean unification
  - Example:

```
add(I1,I2,I3,O1,O2) :-
    X1 = I1 ⊕ I2,
    X2 = I1 ∧ I2,
    X3 = I3 ∧ I2,
    O1 = X1 ⊕ I3,
    O2 = X2 ∨ X3.
```

The query `?- add(a,b,c,O1,O2).` produces

```
O1 = a⊕b⊕c,
O2 = a∧b ⊕ a∧c ⊕ b∧c
```

Programming Methodology

- Hierarchical reasoning
  - Local properties directly represented by constraints
  - Rules specify interaction between modules
  - Global properties generated implicitly
- Constraint Propagation
- Combinatorial Search
- Constraints as Output
  - Compact representation of infinite number of solutions
  - Output can be used as input

Programming Styles:

- Computational Model
  - Propagating the Constraints
  - Making Choices
- Basic Approach
  - Generating domain-variables
  - Generating constraints on the variables
  - Making choices

Constraint Satisfaction

Constraint Satisfaction Problem:

Variables $I = \{X_1, X_2, \dots, X_n\}$
Domains $D_1, D_2, \dots, D_n$
Constraint $c(X_{i_1}, X_{i_2}, \dots, X_{i_k}) \subset D_{i_1} \times D_{i_2} \times D_{i_k}$ specifies compatible values of variables
C is a set of constraints

CSP is given stating I and C and a solution is an assignment of values to all variables (satisfying all the constraints).

Only unary and binary constraints $\rightarrow$ *Constraint Graph*

- Decidable
- Exponential complexity

Solution methods

- Generate and Test
- Backtracking
  - Thrashing
  - Redundant work
- Consistency techniques
  - Node Consistency
  - Arc Consistency

Search Methods

CSP:

Variables: $x_1, \ldots, x_n$
Domains: $D_1, \ldots, D_n$
Constraints: $c_{ij}(x_i, x_j)$ between x_i, x_j, $i < j$

Task: Find values $v_1, \ldots, v_n$ for which $P_n(v_1, \ldots, v_n)$ holds.

Serach Methods — Iteration step

$P_{k+1}(v_1, \ldots, v_k, v_{k_1}) \Longrightarrow P_k(v_1, \ldots, v_k)$

$P_k(v_1, \ldots, v_k)$ is true for all $k < n$ iff:

- **Generate and test**:

  (1) $v_i \in D_i$ for $1 \leq i \leq k$

- **Standard backtracking**:

  (1) condition 1 of generate and test holds;

  (2) $\forall i, j : (1 \leq i < j \leq k)\ c_{ij}(v_i, v_j)$ holds.

- **Forward checking**:

  (1) conditions 1 and 2 of standard backtracking hold;

  (2) $\forall l : (k < l \leq n) \exists v_l \in D_l : c_{1l}(v_1, v_l), \ldots, c_{kl}(v_k, v_l)$ hold.

- **Looking ahead**:

  (1) conditions 1 and 2 of forward checking hold;

  (2) $\forall l : (k < l \leq n) \exists v_l \in D_l$ and it is possible to find values $vk + 1, \ldots, v_{l-1}, v_{l+1}, \ldots, v_n$ which satisfy $c_{k+1l}(v_{k+1}, v_l), \ldots, c_{l-1l}(v_{l-1}, v_l), c_{ll+1}(v_l, v_{l+1}), \ldots, c_{ln}(v_l, v_n)$

Consistency Techniques

- Unary constraints: removing values from domain
- Binary constraints:
  - $\langle V_i, V_j \rangle$ is arc consistent if $\forall a \in D_i : \exists b \in D_j : c(V_i, V_j)$ is satisfied
  - Arc consistency is *directional*
  - Propagation of Constraints — reduction by one arc is *propagated* to other arcs
    - complexity $\mathcal{O}(ed)$, where e number of binary constraints and d domain size
- Strong K-consistency

Arc consistency unsufficient to replace backtracking

Arc consistent CSP can have:

(1) no solution

(2) exactly one solution

(3) more than one solution

Constraint Satisfaction in LP

- **Purpose:**
  - enhance backtracking by more efficient search method(s)

New definition of constraint:

Predicate p/n is a *constraint* iff for any ground terms $t_1, \ldots, t_n$ either $p(t_1, \ldots, t_n)$ has a successful refutation or finitely failed derivations.

- Domain variable
- New Inference Rules
  - Forward Checking — both sound and complete
  - Looking Ahead — sound but not complete
- Generating Values
  - first-fail principle
- Optimization Problems

CHIP

- **Domain declaration:**

$$\texttt{domain p(d}_1\texttt{, ..., d}_n\texttt{).}$$

where d_i is $\begin{cases} \text{constant } \texttt{h} \\ \{a_1, \ldots, a_n\}, a_i \text{ is constant or string} \\ \texttt{l..u}, \texttt{l,u} \text{ integers}, \texttt{l} < \texttt{u} \end{cases}$

- **Forward declaration:**

$$\texttt{forward p(a}_1\texttt{, ..., a}_n\texttt{)}$$

where a_i is g or d.

- **Lookahead declaration:**

$$\texttt{lookahead p(a}_1\texttt{, ..., a}_n\texttt{)}$$

where a_i is g or d.

Extensions — HCLP($\mathcal{R}$)

- Hierarchical CLP($\mathcal{R}$)
  - Over-constrained systems
  - "Soft" constraints — labels on constraints
  - Hierarchy Comparators
    - Splitting
    - Averaging
  - Complex and dependent on comparators
  - Possible applications in:
    - Typesetting & Font Design
    - Graphical Applications
    - Scheduling
    - CAD

Example:

Query:

```
?- strong:X=3, strong:X=4.
```

There are two answers for splitting comparators:

`X=3` and `X=4`

and one answer for averaging comparators:

`X=3.5`

Extensions — CHRs

- User Defined Constraints
  - Specialized constraint solving algorithm hard to encode
    - `alldif(List)`
    - `atmost(N, List, Value)`
    - `element(N, List, Value)`
  - (Almost) impossible to modify existing constraint solver
    It is hard to
    - build a new constraint solver
    - extend the constraint solver
    - specialize the constraint solver
    - combine different constraint solvers

Constraint Handling Rules

- Simplification Rule
  $A_1, \ldots, A_i \Leftrightarrow C_1, \ldots, C_j | B_1, \ldots, B_k$
- Propagation Rule
  $A_1, \ldots, A_i \Rightarrow C_1 \ldots, C_j | B_1, \ldots, B_k$

They are *guarded rules*.

- Call Declaration
  `callable A if` $\mathtt{C}_1$`,...,`$\mathtt{C}_j$

All the rules are multiheaded — this is needed for global constraint satisfaction.

Classical CLP Systems

- CLP($\mathcal{R}$)
  IBM T.J. Watson Research Center
- CHIP
  ECRC $\rightarrow$ COSYTEC $\rightarrow$ ILOG
  Base system with CS
  Domains: Rationals with arbitrary precision
  Boolean unification
  Advanced consistence predicates (cummulative, ...)
  delay primitive (daemons)
- CHARME
  Bull
  Constraint programming language with logical variable
- Prolog-III
  University of Marseille
  Domains: Rationals with arbitrary precision
  Boolean constraints (saturation method)
  Finite strings (restricted string unification)

Academic CLP Systems

- BNR-Prolog
  Bell-Northern Research, Ottawa
  Real intervals

- CAL (Contrainte Avec Logique)
  ICOT
  Domains: Non-linear real equations (Gröbner bases)
  Boolean constraints (modified Gröbner bases)

- ECL^{i}PSe
  ECRC
  Sepia-Prolog with delays, coroutining, and integer finite domains (using meta-structures).

- RISC-CLP
  RISC, Linz
  General arithmetic constraints over real (Tarski's quantifier elimination)

- SICStus-CLP($\mathcal{R}$), SICStus-CLP($\mathcal{Q}$), and FD
  University of Vienna
  Attributed variables

- Many other LP systems incorporate constraint satisfaction features.

Summary

- General Scheme for Constraints in Logic Programming
  - Improved Declarativeness
  - Use of efficient solvers for specific domains
- Active use of constraints to reduce search space
- Larger potencial application areas
  - Engineering
  - Operational Research
    - Schedulling
    - Cutting problems
    - Traffic Assignment
    - Graph Colouring
  - Circuit Design, Verification & Testing
  - Finance
    - Option Trading
    - Investment Planning

Concurrent Logic programming

Základní motivace

Využití víceprocesorových systémů

Proč logické programování:

- Jasná a jednoduchá sémantika
- Logické proměnné (jedinečné přiřazení)
- Nezávislost výsledku na pořadí vyhodnocení (podcílů)

Jednotlivé podcíle mohou být (za určitých podmínek) paralelně zpracovávány.

Typy programů

Transformační (uzavřené) programy – jejich úkolem je transformovat vstup na výstup.

Zahájení i ukončení jasně definované akce

Příklad: program na výpočet daní.

Typy programů

Transformační (uzavřené) programy – jejich úkolem je transformovat vstup na výstup.

Zahájení i ukončení jasně definované akce

Příklad: program na výpočet daní.

Reaktivní (otevřené) programy – jejich úkolem je reagovat na události.

Zahájeny jednou (provždy), ukončení je v podstatě nežádoucí.

Příklad: operační systém.

Typy programů

Transformační (uzavřené) programy – jejich úkolem je transformovat vstup na výstup.

Zahájení i ukončení jasně definované akce

Příklad: program na výpočet daní.

Reaktivní (otevřené) programy – jejich úkolem je reagovat na události.

Zahájeny jednou (provždy), ukončení je v podstatě nežádoucí.

Příklad: operační systém.

Oba typy programů přinášejí odlišný přístup k paralelizaci:

- Transformační programy – „skrytý“ paralelismus. Cílem je maximální zrychlení beze změny sémantiky.
- Reaktivní programy – „zjevný“ (explicitní) paralelismus. Paralelní zpracování událostí, často spojeno i se změnou sémantiky (paralelismus se stává explicitním).

Typy programů II

Paralelní systémy (parallel systems) – pojem používaný (v kontextu logického programování) pro paralelní transformační programy

Konkurentní systémy (concurrent systems) – používán pro paralelní reaktivní programy

Poznámka: každá složka i paralelního systému je de-facto reaktivním programem (zpracovává nějakou posloupnost událostí).

Typy nedeterminismu

Don't Know nedeterminismus –

programátor se nestará o to, která z možných cest je správná (neví to). Program sám při vykonávání vybere (dříve či později) správnou cestu. Výsledky neúspěšných cest nehrají roli (nepřispívají k výsledku, k výsledné substituci).

Typy nedeterminismu

Don't Know nedeterminismus –
programátor se nestará o to, která z možných cest je správná (neví to). Program sám při vykonávání vybere (dříve či později) správnou cestu. Výsledky neúspěšných cest nehrají roli (nepřispívají k výsledku, k výsledné substituci).

Don't Care nedeterminismus –
volba (další cesty) je provedena na základě další, dodatečné znalosti. Pojem „neúspěšná cesta“ nemá přímý smysl.

Typy nedeterminismu

Don't Know nedeterminismus –

programátor se nestará o to, která z možných cest je správná (neví to). Program sám při vykonávání vybere (dříve či později) správnou cestu. Výsledky neúspěšných cest nehrají roli (nepřispívají k výsledku, k výsledné substituci).

Don't Care nedeterminismus –

volba (další cesty) je provedena na základě další, dodatečné znalosti. Pojem „neúspěšná cesta" nemá přímý smysl.

Hloupý Honza a rozcestí

- Don't know: vyzkouší postupně všechny cesty
- Don't care: počká, až mu kolemjdoucí karavana prozradí, kterou cestou se dostane k princezně

Typy nedeterminismu

Don't Know nedeterminismus –
programátor se nestará o to, která z možných cest je správná (neví to). Program sám při vykonávání vybere (dříve či později) správnou cestu. Výsledky neúspěšných cest nehrají roli (nepřispívají k výsledku, k výsledné substituci).

Don't Care nedeterminismus –
volba (další cesty) je provedena na základě další, dodatečné znalosti. Pojem „neúspěšná cesta" nemá přímý smysl.

Hloupý Honza a rozcestí

- Don't know: vyzkouší postupně všechny cesty
- Don't care: počká, až mu kolemjdoucí karavana prozradí, kterou cestou se dostane k princezně

- Don't know: typický pro sekvenční logické programy
- Don't care: typický pro paralelní (konkurentní) logické programy

Paralelismus v logickém programování

Or-paralelismus –

paralelně jsou spuštěny všechny klauzule, jejichž hlavy unifikují se zadaným cílem.

Paralelismus v logickém programování

Or-paralelismus –
paralelně jsou spuštěny všechny klauzule, jejichž hlavy unifikují se zadaným cílem.

And-paralelismus –
paralelně jsou spuštěny všechny podcíle právě vykonávané klauzule.

Paralelismus v logickém programování

Or-paralelismus –
paralelně jsou spuštěny všechny klauzule, jejichž hlavy unifikují se zadaným cílem.

And-paralelismus –
paralelně jsou spuštěny všechny podcíle právě vykonávané klauzule.

Plný paralelismus –
Neomezený and- a or-paralelismus.

Paralelismus v logickém programování

Or-paralelismus –
paralelně jsou spuštěny všechny klauzule, jejichž hlavy unifikují se zadaným cílem.

And-paralelismus –
paralelně jsou spuštěny všechny podcíle právě vykonávané klauzule.

Plný paralelismus –
Neomezený and- a or-paralelismus.

Uplatnitelné (alespoň teoreticky) pro transformační i reaktivní programy.

Problémy s plným paralelismem

Uvažujme paralelizaci „klasických“ sekvenčních logických programů (Prolog)

Problémy s plným paralelismem

Uvažujme paralelizaci „klasických" sekvenčních logických programů (Prolog)

- Or-paralelismus:
  - Zachování pořadí výsledků – paralelně spuštěná alternativa může dospět dříve k výsledku než „první" větev v sekvenčním modelu
  - Interakce s prostředím (např. vstupy a výstupy)
  - Rozštěpení proměnných – pro každý paralelní proces se musí vytvořit nový kontext

  Or-paralelismus „probouzí" prohledávání do šířky

Problémy s plným paralelismem II

- And-paralelismus:
  - Vazba proměnných

    Mejmě program

```
a(1).   a(2).   a(3).
b(3).   b(4).   b(5).
```

a dotaz

```
?- a(X), b(X).
```

And-paralelní zpracování (předpokládáme, že prohledávání programové databáze probíhá „standardním“ způsobem) naváže `X` na `1` (při vyhodnocení `a(1)`) a paralelně na 3 (při vyhodnocení `b(X)`).

Problémy s plným paralelismem II

- And-paralelismus:
  - Vazba proměnných
    Mejmě program

```
a(1).  a(2).  a(3).
b(3).  b(4).  b(5).
```

    a dotaz

```
?- a(X), b(X).
```

    And-paralelní zpracování (předpokládáme, že prohledávání programové databáze probíhá „standardním“ způsobem) naváže `X` na `1` (při vyhodnocení `a(1)`) a paralelně na 3 (při vyhodnocení `b(X)`).

    Problém: Jak poznat, kterou z těchto vazeb zrušit (a vybrat jinou)?

Problémy s plným paralelismem II

- And-paralelismus:
  - Vazba proměnných

    Mejmě program

    ```
    a(1).   a(2).   a(3).
    b(3).   b(4).   b(5).
    ```

    a dotaz

    ```
    ?- a(X), b(X).
    ```

    And-paralelní zpracování (předpokládáme, že prohledávání programové databáze probíhá „standardním“ způsobem) naváže `X` na `1` (při vyhodnocení `a(1)`) a paralelně na 3 (při vyhodnocení `b(X)`).

    Problém: Jak poznat, kterou z těchto vazeb zrušit (a vybrat jinou)?

Řešení: nový programovací model (a jazyk)

Strážené jazyky

Tělo klauzule rozdělíme na dvě části

```
H :- G | B.
```

kde

- `H` je hlava klauzule (literál)
- `G` je stráž klauzule (případné prázdná posloupnost literálů)
- | je uperátor upnutí
- `B` je tělo klauzule (případně prázdná posloupnost literálů)

Strážené jazyky

Tělo klauzule rozdělíme na dvě části

```
H :- G | B.
```

kde

- H je hlava klauzule (literál)
- G je stráž klauzule (případné prázdná posloupnost literálů)
- | je uperátor upnutí
- B je tělo klauzule (případně prázdná posloupnost literálů)

Omezený or-paralelismus:
Or-paralelně jsou spuštěny všechny větve, ale jakmile jedna z nich provede operátor upnutí (commit), jsou ostatní or-paralelní větve násilně ukončeny.

Strážené jazyky II

- Paralelně jsou vyhodnoceny pouze stráže

Strážené jazyky II

- Paralelně jsou vyhodnoceny pouze stráže
- Jak rozhodnout, která stráž má být úspěšná?
  - Vzájemná exkluzivita (pouze jedna stráž) odstraní nedeterminismus
  - Řešení: použití don't know nedeterminismu
    Honza a karavana: paralelně probíhající procesy

Strážené jazyky II

- Paralelně jsou vyhodnoceny pouze stráže
- Jak rozhodnout, která stráž má být úspěšná?
  - Vzájemná exkluzivita (pouze jedna stráž) odstraní nedeterminismus
  - Řešení: použití don't know nedeterminismu
    Honza a karavana: paralelně probíhající procesy
- And-paralelismus: změna sémantiky unifikace proměnných

Strážené jazyky II

- Paralelně jsou vyhodnoceny pouze stráže
- Jak rozhodnout, která stráž má být úspěšná?
  - Vzájemná exkluzivita (pouze jedna stráž) odstraní nedeterminismus
  - Řešení: použití don't know nedeterminismu
    Honza a karavana: paralelně probíhající procesy
- And-paralelismus: změna sémantiky unifikace proměnných
  - Explicitní: programátor specifikuje (konkrétní konstrukcí jazyka), zda daná unifikace může vázat proměnnou (dosadit hodnotu) nebo zda jde pouze o test (výpočet musí cekat).
  - Implicitní: v určitých syntakticky definovaných částech programu není povolena vazba proměnných (např. není povolena ve strážích). Hovoříme o vstupní unifikaci.

Strážené jazyky II

- Paralelně jsou vyhodnoceny pouze stráže
- Jak rozhodnout, která stráž má být úspěšná?
  - Vzájemná exkluzivita (pouze jedna stráž) odstraní nedeterminismus
  - Řešení: použití don't know nedeterminismu
    Honza a karavana: paralelně probíhající procesy
- And-paralelismus: změna sémantiky unifikace proměnných
  - Explicitní: programátor specifikuje (konkrétní konstrukcí jazyka), zda daná unifikace může vázat proměnnou (dosadit hodnotu) nebo zda jde pouze o test (výpočet musí cekat).
  - Implicitní: v určitých syntakticky definovaných částech programu není povolena vazba proměnných (např. není povolena ve strážích). Hovoříme o vstupní unifikaci.

  Oba přístupy v konkurentním logickém programování použity.

Procesová sémantika

Korespondence mezi pojmy z paralelního programování a z logického programování

proces	atomický cíl
síť procesů	konjunkce cílů
instrukce procesu	klauzule
komunikační kanál	logická proměnná
komunikace, přiřazení	obecná unifikace
synchronizace, předání parametrů	vstupní unifikace

Ploché (flat) jazyky

Další zjednodušení:

- Stráže jsou tvořeny vestavěnými predikáty (tj. nikoliv uživatelskými)
- Unifikace ve strážích nahrazena vstupní unifikací.

  Vstupní unifikace – probíhá jako obvyklá unifikace, avšak je-li v průběhu unifikace nutno dosadit hodnotu vstupní proměnné, je výpočet **pozastaven**.

Příklady stráží:

`X` je vstupní proměnná (vyskytuje se v hlavě, např. `a(X):-` nebo `a(s(X)):-`); pokud je při vykonání stráže tato proměnná neinstanciována, výpočet se pozastaví, dokud jí paralelní proces nedosadí hodnotu.

`X > 3`	jednoduchý test
`integer(X)`	jednoduchý test
`X = [A|B]`	test; pokud uspěje, je `A` hlava a `B` ocas seznamu v `X`.

Ploché (flat) jazyky

Další zjednodušení:

- Stráže jsou tvořeny vestavěnými predikáty (tj. nikoliv uživatelskými)
- Unifikace ve strážích nahrazena vstupní unifikací.

  Vstupní unifikace – probíhá jako obvyklá unifikace, avšak je-li v průběhu unifikace nutno dosadit hodnotu vstupní proměnné, je výpočet **pozastaven**.

Příklady stráží:
`X` je vstupní proměnná (vyskytuje se v hlavě, např. `a(X):-` nebo `a(s(X)):-`); pokud je při vykonání stráže tato proměnná neinstanciována, výpočet se pozastaví, dokud jí paralelní proces nedosadí hodnotu.

`X > 3`	jednoduchý test
`integer(X)`	jednoduchý test
`X = [A|B]`	test; pokud uspěje, je `A` hlava a `B` ocas seznamu v `X`.

Důvod zjednodušení

- Omezené prohledávání do hloubky
- Vstupní unifikace řeší synchronizaci and-paralelních procesů

Konkurentní logické programy

Nový přístup (ne pouhá paralelizace Prologu)

Konkurentní logické programy (Concurrent Logic Programs) používají and-paralelismus a omezený or-paralelismus.

V dalším představíme reprezentanta: Flat Concurrent Prolog

Syntaxe: Program je množina **strážených** klauzulí tvaru:

$$\texttt{H <- } \texttt{G}_1\texttt{,...,}\texttt{G}_n \texttt{ | } \texttt{B}_1\texttt{,...,}\texttt{B}_m\texttt{.} \qquad \texttt{n,m} \geq 0$$

kde `H` je **hlava**, $\texttt{G}_i$ jsou **stráže**, | je operátor **upnutí** a $\texttt{B}_i$ tvoří **tělo** klauzule.

Konkurentní logické programy

Nový přístup (ne pouhá paralelizace Prologu)

Konkurentní logické programy (Concurrent Logic Programs) používají and-paralelismus a omezený or-paralelismus.

V dalším představíme reprezentanta: Flat Concurrent Prolog

Syntaxe: Program je množina **strážených** klauzulí tvaru:

$$\texttt{H <- } G_1, \ldots, G_n \mid B_1, \ldots, B_m. \qquad n,m \geq 0$$

kde H je **hlava**, G_i jsou **stráže**, | je operátor **upnutí** a B_i tvoří **tělo** klauzule.

Deklarativně lze klauzuli číst:

H platí, platí-li G_i a B_j

Z hlediska chování je takto definována síť procesů, kde každé B_j je samostatný proces.

Flat Concurrent Prolog (FCP)

Procesy mohou **změnit stav**
rozštěpit se
skončit

Flat Concurrent Prolog (FCP)

Procesy mohou **změnit stav**
rozštěpit se
skončit

Procesy spolu komunikují prostřednictvím logických proměnných.

Flat Concurrent Prolog (FCP)

Procesy mohou **změnit stav**
rozštěpit se
skončit

Procesy spolu komunikují prostřednictvím logických proměnných.

Chování procesů:

`A <- B.` – změna stavu z `A` na `B`
`A <- B`$_1$`,...,B`$_n$`.` – rozštěpení `A` na procesy `B`$_1$`,...,B`$_n$
`A <- true.` – ukončení procesu

Příklad programování v FCP – změna stylu

Máme seznamy `Ys` a `Zs` a prvek `X`. Úkolem je vytvořit program `in_both/3`, který uspěje, je-li `X` prvkem obou seznamů, a program `in_either/3`, který uspěje, je-li `X` prvkem alespoň jednoho ze seznamů `Ys` a `Zs`.

Příklad programování v FCP – změna stylu

Máme seznamy `Ys` a `Zs` a prvek `X`. Úkolem je vytvořit program `in_both/3`, který uspěje, je-li `X` prvkem obou seznamů, a program `in_either/3`, který uspěje, je-li `X` prvkem alespoň jednoho ze seznamů `Ys` a `Zs`.

Původní program v Prologu

```
in_both(X,Ys,Zs) :- member(X,Ys), member(X,Zs).

in_either(X,Ys,Zs) :- member(X,Ys).
in_either(X,Ys,Zs) :- member(X,Zs).

member(X,[X|Ys]).
member(X,[_|Ys]) :- member(X,Ys).
```

Příklad programování v FCP II

Pseudosekvenční verze

V první klauzuli použijeme implicitní stráž (`true |`)

```
in_both(X,[X|Ys],Zs) <- member(X,Zs).
in_both(X,[Y|Ys],Zs) <- X\=Y | in_both(X,Ys,Zs).

in_either(X,[X|Ys],Zs).
in_either(X,[Y|Ys],Zs) <- X\=Y | in_either(X,Ys,Zs).
in_either(X,Ys,[X|Zs]).
in_either(X,Ys,[Y|Zs]) <- X\=Y | in_either(X,Ys,Zs).

member(X,[X|Ys]).
member(X,[Y|Ys]) <- X\=Y | member(X,Ys).
```

Příklad programování v FCP II

Pseudosekvenční verze

V první klauzuli použijeme implicitní stráž (`true |`)

```
in_both(X,[X|Ys],Zs) <- member(X,Zs).
in_both(X,[Y|Ys],Zs) <- X\=Y | in_both(X,Ys,Zs).

in_either(X,[X|Ys],Zs).
in_either(X,[Y|Ys],Zs) <- X\=Y | in_either(X,Ys,Zs).
in_either(X,Ys,[X|Zs]).
in_either(X,Ys,[Y|Zs]) <- X\=Y | in_either(X,Ys,Zs).

member(X,[X|Ys]).
member(X,[Y|Ys]) <- X\=Y | member(X,Ys).
```

Tato verze je nekorektní (není to správný program v FCP, přestože syntakticky je v pořádku), protože stále pracuje s pojmem neúspěchu. V FCP se mohou vyhodnotit jako neúspěšné pouze stráže, vyhodnocení těla musí být vždy úspěšné. To však nesplňuje ani predikát `member/2`, neboť volání `?-member(a,[1,2,3])` nemůže uspět, což je však v FCP chyba.

Příklad II – korektní verze v FCP

`member/2` je v Prologu použit jednou jako test (jsou-li prvek i seznam instanciovány), podruhé jako generátor (není-li prvek `X` instanciován).

Tato dualita je umožněna použitím don't know nedeterminismu.

FCP používá don't care nedeterminismus, musí proto oba případy explicitně odlišit; současně je třeba definovat pozitivní reakci v případě neúpěšného testu. Sémantiku predikátů je proto nutné poněkud opravit – namísto testu se generuje seznam všech prvků, které příslušnou podmínku (`in_both` nebo `in_either`) splňují:

Příklad II – korektní verze v FCP

member/2 je v Prologu použit jednou jako test (jsou-li prvek i seznam instanciovány), podruhé jako generátor (není-li prvek X instanciován).

Tato dualita je umožněna použitím don't know nedeterminismu.

FCP používá don't care nedeterminismus, musí proto oba případy explicitně odlišit; současně je třeba definovat pozitivní reakci v případě neúpěšného testu. Sémantiku predikátů je proto nutné poněkud opravit – namísto testu se generuje seznam všech prvků, které příslušnou podmínku (in_both nebo in_either) splňují:

```
in_both(Xs,[X|Ys],Zs) <- member(X,Zs,Xs\Xs1), in_both(Xs1,Ys,Zs).
in_both(Xs,[],_) <- Xs = [].

in_either(Xs,[X|Ys],Zs) <- Xs = [X|Xs1], in_either(Xs1,Ys,Zs).
in_either(Xs,[],Z) <- Xs = Z.

member(X,[X|Y],Xs\Xs1) <- X = [X|Xs1].
member(X,[Y|Ys],Xs\Xs1) <- X\=Y | member(X,Ys,Xs\Xs1).
member(X,[],Xs\Xs1) <- Xs = Xs1.
```

Příklad II – korektní verze v FCP

`member/2` je v Prologu použit jednou jako test (jsou-li prvek i seznam instanciovány), podruhé jako generátor (není-li prvek `X` instanciován).

Tato dualita je umožněna použitím don't know nedeterminismu.

FCP používá don't care nedeterminismus, musí proto oba případy explicitně odlišit; současně je třeba definovat pozitivní reakci v případě neúpěšného testu. Sémantiku predikátů je proto nutné poněkud opravit – namísto testu se generuje seznam všech prvků, které příslušnou podmínku (`in_both` nebo `in_either`) splňují:

```
in_both(Xs,[X|Ys],Zs) <- member(X,Zs,Xs\Xs1), in_both(Xs1,Ys,Zs).
in_both(Xs,[],_) <- Xs = [].

in_either(Xs,[X|Ys],Zs) <- Xs = [X|Xs1], in_either(Xs1,Ys,Zs).
in_either(Xs,[],Z) <- Xs = Z.

member(X,[X|Y],Xs\Xs1) <- X = [X|Xs1].
member(X,[Y|Ys],Xs\Xs1) <- X\=Y | member(X,Ys,Xs\Xs1).
member(X,[],Xs\Xs1) <- Xs = Xs1.
```

Všechny predikáty vždy uspějí, výsledný prázdný seznam je ekvivalentem neúspěchu v Prologu.

Synchronizace Logickou Proměnnou

Logická proměnná – synchronizační kanál

Jeden proces proměnnou inicializuje, ostatní procesy na tuto událost čekají.

Synchronizace Logickou Proměnnou

Logická proměnná – synchronizační kanál

Jeden proces proměnnou inicializuje, ostatní procesy na tuto událost čekají.

Komunikační protokoly:

- **dvoubodová komunikace** $(1 \rightarrow 1)$

```
tree_sum(T,S) <- tree_sum(T,0,S).
tree_sum(tree(L,R), P, S) <-
                 tree_sum(L,P,P1), tree_sum(R,P1,S).
tree_sum(leaf(X),P,S) <- plus(X,P,S).
```

Synchronizace Logickou Proměnnou

Logická proměnná – synchronizační kanál

Jeden proces proměnnou inicializuje, ostatní procesy na tuto událost čekají.

Komunikační protokoly:

- **dvoubodová komunikace** $(1 \rightarrow 1)$

  ```
  tree_sum(T,S) <- tree_sum(T,0,S).
  tree_sum(tree(L,R), P, S) <-
                      tree_sum(L,P,P1), tree_sum(R,P1,S).
  tree_sum(leaf(X),P,S) <- plus(X,P,S).
  ```

- **broadcast komunikace** $(1 \rightarrow n)$
  Jednu hodnotu „čte“ více procesů

Synchronizace Logickou Proměnnou

Logická proměnná – synchronizační kanál

Jeden proces proměnnou inicializuje, ostatní procesy na tuto událost čekají.

Komunikační protokoly:

- **dvoubodová komunikace** $(1 \rightarrow 1)$

```
tree_sum(T,S) <- tree_sum(T,0,S).
tree_sum(tree(L,R), P, S) <-
                  tree_sum(L,P,P1), tree_sum(R,P1,S).
tree_sum(leaf(X),P,S) <- plus(X,P,S).
```

- **broadcast komunikace** $(1 \rightarrow n)$
  Jednu hodnotu „čte" více procesů
- **duplexní komunikace** $(1 \leftrightarrow 1)$
  procesy sdílí dvě proměnné

Synchronizace Logickou Proměnnou

Logická proměnná – synchronizační kanál

Jeden proces proměnnou inicializuje, ostatní procesy na tuto událost čekají.

Komunikační protokoly:

- **dvoubodová komunikace** $(1 \rightarrow 1)$

```
tree_sum(T,S) <- tree_sum(T,0,S).
tree_sum(tree(L,R), P, S) <-
                   tree_sum(L,P,P1), tree_sum(R,P1,S).
tree_sum(leaf(X),P,S) <- plus(X,P,S).
```

- **broadcast komunikace** $(1 \rightarrow n)$
  Jednu hodnotu „čte“ více procesů
- **duplexní komunikace** $(1 \leftrightarrow 1)$
  procesy sdílí dvě proměnné
- **více do jednoho** $(n \rightarrow 1)$
  n procesů zapíše do celkem n proměnných, jeden proces pak „čte“ všechny hodnoty

Synchronizace Logickou Proměnnou

Logická proměnná – synchronizační kanál

Jeden proces proměnnou inicializuje, ostatní procesy na tuto událost čekají.

Komunikační protokoly:

- **dvoubodová komunikace** $(1 \rightarrow 1)$

```
tree_sum(T,S) <- tree_sum(T,0,S).
tree_sum(tree(L,R), P, S) <-
                 tree_sum(L,P,P1), tree_sum(R,P1,S).
tree_sum(leaf(X),P,S) <- plus(X,P,S).
```

- **broadcast komunikace** $(1 \rightarrow n)$
  Jednu hodnotu „čte“ více procesů
- **duplexní komunikace** $(1 \leftrightarrow 1)$
  procesy sdílí dvě proměnné
- **více do jednoho** $(n \rightarrow 1)$
  n procesů zapíše do celkem n proměnných, jeden proces pak „čte“ všechny hodnoty
- **komunikace přes tabuli** $(m \leftrightarrow n)$
  kombinace výše uvedených

Synchronizace Logickou Proměnnou II

Všechny uvedené protokoly jsou realizovatelné pomocí **proudů** (streams) $\Longrightarrow$ generované seznamy.

Synchronizace Logickou Proměnnou II

Všechny uvedené protokoly jsou realizovatelné pomocí **proudů** (streams) $\Longrightarrow$ generované seznamy.

Každý komunikační krok je realizován pomocí jedné logické proměnné (případně jedné skupiny logických proměnných), ty jsou postupně ukládány do stále rostoucího seznamu.

Synchronizace Logickou Proměnnou II

Všechny uvedené protokoly jsou realizovatelné pomocí **proudů** (streams) $\Longrightarrow$ generované seznamy.

Každý komunikační krok je realizován pomocí jedné logické proměnné (případně jedné skupiny logických proměnných), ty jsou postupně ukládány do stále rostoucího seznamu.

Potenciálně neokonečná komunikace, v praxi omezena délkou seznamu (proudu) – systémy garbage collection pro uvolnění paměti zabírané již nevyužívanou částí proudu.

Synchronizace Logickou Proměnnou III

Protokoly s neúplnými zprávami:

- Zpětná komunikace (model „remote procedure call“)
- Dialog
- Výstavba (konfigurace) sítě
- Rekonfigurace sítě
- Komunikace s bufferem pevné délky (bounded-buffer)

Synchronizace Logickou Proměnnou III

Protokoly s neúplnými zprávami:

- Zpětná komunikace (model „remote procedure call“)
- Dialog
- Výstavba (konfigurace) sítě
- Rekonfigurace sítě
- Komunikace s bufferem pevné délky (bounded-buffer)

Výhody použití (sdílené) logické proměnné jako sdílené paměti:

- Možnost udržení historie komunikace programu
- Proud zpráv může být prohlížen a transformován (použitelné např. při ladění)

Využití proudů – Základní techniky

Konzumenti a producenti

- Konzument: proces, který čeká na instanciaci proměnné v hlavě nebo stráži – `c(a)`.

Využití proudů – Základní techniky

Konzumenti a producenti

- Konzument: proces, který čeká na instanciaci proměnné v hlavě nebo stráži – `c(a).`
- Producent: proces, který unifikuje proměnnou z hlavy (v těle) – `p(X) <- X=a.`

Využití proudů – Základní techniky

Konzumenti a producenti

- Konzument: proces, který čeká na instanciaci proměnné v hlavě nebo stráži – `c(a).`
- Producent: proces, který unifikuje proměnnou z hlavy (v těle) – `p(X) <- X=a.`

Nedeterminismus

Nedeterministická volba z více možností (ve stráži):

```
c(a).
c(b).
```

Využití proudů – Základní techniky

Konzumenti a producenti

- Konzument: proces, který čeká na instanciaci proměnné v hlavě nebo stráži – `c(a).`
- Producent: proces, který unifikuje proměnnou z hlavy (v těle) – `p(X) <- X=a.`

Nedeterminismus

Nedeterministická volba z více možností (ve stráži):

```
c(a).
c(b).
```

Nedeterministický výběr z více možností (za stráží):

```
p(X) <- X=a.
p(X) <- X=b.
```

Využití proudů II

- Produkce proudu:
  Seznam všech celých čísel mezi From a To:

```
integers(From,To,Ns) <- From > To | Ns=[].
integers(From,To,Ns) <- From =< To |
     Ns=[From|Ns1],
     From1 is From+1,
     integers(From1,To,Ns1).
```

Využití proudů II

- Produkce proudu:
  Seznam všech celých čísel mezi From a To:

```
integers(From,To,Ns) <- From > To | Ns=[].
integers(From,To,Ns) <- From =< To |
     Ns=[From|Ns1],
     From1 is From+1,
     integers(From1,To,Ns1).
```

Seznam všech Fibbonacciho čísel menších než N:

```
fib(N,Ns) <- fib1(N,0,1,Ns).
fib1(N,N1,N2,Ns) <- N =< N1 | Ns=[].
fib1(N,N1,N2,Ns) <- N > N1 | Ns =[N1|Ns1],
       N3 is N1+N2,
       fib1(N,N2,N3,Ns1).
```

Využití proudů III

- Konzumace proudu:
  Skalární součin dvou vektorů:

```
ip(Xs,Ys,S) <- ip1(Xs,Ys,0,S).
ip1([],[],P,S) <- P=S.
ip1([X|Xs],[Y|Ys],P,S) <-
      P1 is P+X*Y,
      ip1(Xs,Ys,P1,S).
```

Využití proudů IV – modifikace proudu

filtry; transducers

Vyjmi všechny násobky čísla N z proudu:

```
filter([X|In],P,Out) <- 0=/=X mod P | Out=[X|Out1],
        filter(In,P,Out1).
filter([X|In],P,Out) <- 0=:=X mod P |
        filter(In,P,Out).
filter([],P,Out) <- Out=[].
```

Využití proudů V – distribuce proudu

Rozdělení do dvou proudů podle zprávy:

```
distribute([send(1,X)|In],Out1,Out2) <-
        Out1=[X|Out11],
        distribute(In,Out11,Out2).
distribute([send(2,X)|In],Out1,Out2) <-
        Out2=[X|Out21],
        distribute(In,Out1,Out21).
distribute([],Out1,Out2) <- Out1=[], Out2=[].
```

Využití proudů VI – spojování proudů

Spojení dvou utříděných proudů – deterministické:

```
omerge([X|In1],[Y|In2],Out) <- X =< Y | Out=[X|Out1],
        omerge(In1,[Y|In2],Out1).
omerge([X|In1],[Y|In2],Out) <- X > Y | Out=[Y|Out1],
        omerge([X|In1],In2,Out1).
merge([],In2,Out) <- Out=In2.
merge(In1,[],Out) <- Out=In1.
```

Využití proudů VI – spojování proudů

Spojení dvou utříděných proudů – deterministické:

```
omerge([X|In1],[Y|In2],Out) <- X =< Y | Out=[X|Out1],
        omerge(In1,[Y|In2],Out1).
omerge([X|In1],[Y|In2],Out) <- X > Y | Out=[Y|Out1],
        omerge([X|In1],In2,Out1).
merge([],In2,Out) <- Out=In2.
merge(In1,[],Out) <- Out=In1.
```

Spojení dvou utříděných proudů – nedeterministické:

```
omerge([X|In1],[Y|In2],Out) <- X =< Y | Out=[X|Out1],
        omerge(In1,[Y|In2],Out1).
omerge([X|In1],[Y|In2],Out) <- X >= Y | Out=[Y|Out1],
        omerge([X|In1],In2,Out1).
merge([],In2,Out) <- Out=In2.
merge(In1,[],Out) <- Out=In1.
```

Problém and- a or-fairness (vyhladovění proudu)

Pokročilé techniky

- Hammingův problém: utříděná posloupnost všech čísel (bez duplikátů) tvaru $2^i 3^j 5^k$:

```
hamming(Xs) <-
    multiply([1|Xs],2,X2),
    multiply([1|Xs],3,X3),
    multiply([1|Xs],5,X5),
    omerge1(X2,X3,X23),
    omerge1(X23,X5,Xs).
```

kde omerge1 odstraňuje duplicitní prvky.

- Dynamická síť procesů:
  Eratosthenovo síto:

```
primes(N,Ps) <-
    integers(2,N,Ns), sift(Ns,Ps).

sift([P|Ns],Ps) <- Ps=[P|Ps1],
    filter(Ns,P,Ns1), sift(Ns1,Ps1).
sift([],Ps) <- Ps=[].
```

- Dynamická síť procesů:
  Eratosthenovo síto:

```
primes(N,Ps) <-
    integers(2,N,Ns), sift(Ns,Ps).

sift([P|Ns],Ps) <- Ps=[P|Ps1],
    filter(Ns,P,Ns1), sift(Ns1,Ps1).
sift([],Ps) <- Ps=[].
```

Násobení vektoru maticí:

```
vm(_,[],Zv) <- Zv=[].
vm(Xv,[Yv|Ym],Zv) <- Zv=[Z|Zv1],
    ip(Xv,Yv,Z), vm(Xv,Ym,Zv1).
```

- Dynamická síť procesů:
  Eratosthenovo síto:

```
primes(N,Ps) <-
    integers(2,N,Ns), sift(Ns,Ps).

sift([P|Ns],Ps) <- Ps=[P|Ps1],
    filter(Ns,P,Ns1), sift(Ns1,Ps1).
sift([],Ps) <- Ps=[].
```

Násobení vektoru maticí:

```
vm(_,[],Zv) <- Zv=[].
vm(Xv,[Yv|Ym],Zv) <- Zv=[Z|Zv1],
    ip(Xv,Yv,Z), vm(Xv,Ym,Zv1).
```

Násobení dvou matic:

```
mm([],_,Zm) <- Zm=[].
mm([Zv|Xm],Ym,Zm) <- Zm=[Zv|Zm1],
    vm(Xv,Ym,Zv), mm(Xm,Ym,Zm1).
```

Neúplné zprávy

- Monitory:

  Jednoduchý čitač:

```
counter(In) <- counter1(In,0).

counter1([clear|In],C) <- counter1(In,0).
counter1([add|In],C) <- C1 is C+1, counter1(In,C1).
counter1([read(X)|In],C) <- X=C, counter1(In,C).
counter1([],_).
```

Neúplné zprávy

- Monitory:

  Jednoduchý čitač:

```
counter(In) <- counter1(In,0).

counter1([clear|In],C) <- counter1(In,0).
counter1([add|In],C) <- C1 is C+1, counter1(In,C1).
counter1([read(X)|In],C) <- X=C, counter1(In,C).
counter1([],_).
```

  Sdílená fronta:

```
queue(In) <- queue1(In,Q-Q).

queue1([dequeue(X)|In],H-T) <- H=[X|H1],
    queue1(In,H1-T).
queue1([enqueue(X)|In],H-T) <- T=[X|T1],
    queue1(In,H-T1).
queue1([],H-T).
```

Distribuovaná detekce ukončení, klidu

Problém:
Jak garantovat, že všechny podprocesy daného procesu již skončily?

Distribuovaná detekce ukončení, klidu

Problém:
Jak garantovat, že všechny podprocesy daného procesu již skončily?

– Dva nové argumenty ke každému cíli: `Left`, `Right`

– Podle změny:

změna	předat beze změny
rozštěpení	vytvořit posloupnost do všech podcílů `(Left,Left1)`,...,`(Leftn,Right)`
zánik	unifikovat: `Left=Right`

Distribuovaná detekce ukončení, klidu

Problém:
Jak garantovat, že všechny podprocesy daného procesu již skončily?

– Dva nové argumenty ke každému cíli: `Left`, `Right`

– Podle změny:

změna	předat beze změny
rozštěpení	vytvořit posloupnost do všech podcílů `(Left,Left1),...,(Leftn,Right)`
zánik	unifikovat: `Left=Right`

– Volání cíle s kontrolou:

```
?- goal(Left, Right), Left=done, wait_for_done(Right).

wait_for_done(done) <- cíl goal včetně všech podcílů
                       skončil
```

- Vzájemné vyloučení

```
mutex(In) <- mutex1(In,done).

mutex1([lock(Reply)|In],done) <-
    Reply=granted(Done),
    mutex1(In,Done).
mutex1([],_).



p(Mutex,...) <- p_request(done,Mutex).

p_request(done,Mutex,...) <-
    Mutex=[lock(Reply)|Mutex1],
    p_wait(Reply,Mutex1,...).

p_wait(granted(Done),Mutex1,...) <-
    do critical operation,
p_request(Done,Mutex,...).
```

- Distribuované řízení událostmi

```
p_dormant([m(Left-Right,LeftEnd-RightEnd,...)|In],

        Out,...) <-
    Left=[X|Left1], Right=[X|Right1],
    ...,
    p_passive(In,Out,Left1-Right1,
        LeftEnd-RightEnd, ...).

p_passive([m(Left1-Right1,LeftEnd-RightEnd,...)|In],Out,

        Left-Right,LeftEnd-RightEnd,...) <-
    Left1=Right1,
    ...,
    p_passive(In,Out,Left-Right,LeftEnd-RightEnd,...).
p_passive(In,Out,Left-Right,
        [X|LeftEnd]-[X|RightEnd],...) <-
    ...,
    Out=[m(Left-Right,LeftEnd-RightEnd,...)|Out1],
    p_dormant(In,Out1,...).
```

- Buffer omezené délky:

```
bounded_buffer_network(...) <-
    buffer(N,H-T),
    consume(H-T, ...),
    produce(H, ...).
buffer(0,H-T) <- H=T.
buffer(N,H-T) <-
    N>0 |
    N1 is N-1, T=[message(_)|T1],
    buffer(N1,H-T1).
consume([message(X)|H]-T, ...) <-
    known(X),
    more X's wanted |
    T=[message(_)|T1],
    process X,
    consume(H-T1, ...).
consume(H-T, ...) <-
    no more X's wanted |
    T=[],
    process remaining X's in H.
```

```
produce([message(X)|In], ...) <-
    produce X,
    produce(In, ...).
produce([], ...).
```

- Meta-interpret:
  Programové klauzule `A <- G | B` jsou reprezentovány jako `clause(A,X) <- G | X=B1`, kde `B1` je konjunkce cílů `Goal(B)` (kromě predikátů '=' a '`true`').

```
reduce(true).
reduce(X=Y)       <- X=Y.
reduce((A,B))     <- reduce(A), reduce(B).
reduce(goal(A))   <- clause(A,B), reduce(B).
```

  Detekce ukončení:

```
reduce(A,Done) <- reduce1(A,done-Done).

reduce1(true,L-R)      <- L=R.
reduce1(X=Y,L-R)       <- (X,L)=(Y,R).
reduce1((A,B),L-R)     <- reduce1(A,L-M),
    reduce1(B,M-R).
reduce1(goal(A),L-R)   <- clause(A,B), reduce1(B,L-R).
```

Zpracování přerušení:

```
reduce(true,Is).
reduce(X=Y,Is)      <- X=Y.
reduce((A,B),Is)    <- reduce(A,Is), reduce(B,Is).
reduce(goal(A),Is) <- clause(A,B,Is), reduce(B,Is).
reduce(A,[I|Is])    <- serve_intr([I|Is],A).

serve_intr([abort|_],_).
serve_intr([suspend|Is],A) <- serve_intr(Is,A).
serve_intr([resume|Is],A)  <- reduce(A,Is).
```