KATEDRA INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI
www.inf.upol.cz

Jan Laštovička
https://www.inf.upol.cz/lide/jan-lastovicka
jan.lastovicka@upol.cz
17. listopadu 12, 771 46 Olomouc

Paradigmata programování 4 ⋄ poznámky k přednášce

11. Proměnné v dotazech

verze z 6. května 2025

1 Proměnné v cíli

Začněme motivačním příkladem, kde máme fakta:

1. Abel je dítětem Evy.
2. Abel je dítětem Adama.
3. Adam je muž.
4. Abel je muž.
5. Eva je žena.

a jediné pravidlo:

- Jestliže existuje ***y*** tak, že ***y*** je dítětem ***x*** a ***x*** je muž, pak ***x*** je otec.

Chceme zjistit, kdo je otcem.

Pravidlo je opět výrok tvaru:

- Pro každé ***var1***, ..., ***varm*** platí, že jestliže ***body1*** a ... a ***bodyn***, pak ***head***.

V pravidle

- ***bodyi*** a ***head*** jsou atomické výrokové formy
- a ***vari*** jsou všechny proměnné, které se v nich vyskytují.

Pravidlo stále zapisujeme seznamem `(<- head body1 ... bodyn)`. Omezení z minulé přednášky, které nařizovalo, aby proměnné v těle byly přítomny i v hlavě pravidla, zde již neplatí.

Například vezměme pravidlo z motivačního příkladu, které ale není v požadovaném tvaru:

- Pro každé x platí, že jestliže existuje y tak, že y je dítětem x a x je muž, pak x je otec.

Nejprve jej ekvivalentně vyjádříme výrokem v požadovaném tvaru:

- Pro každé x, y platí, že jestliže y je dítětem x a x je muž, pak x je otec.

Pravidlo zapíšeme seznamem `(<- (father ?x) (child ?y ?x) (male ?x))`.

Systém z motivačního příkladu zapíšeme následujícím programem.

```
(defpredicate
 (<- (child abel eva))
 (<- (child abel adam)))

(defpredicate
 (<- (male adam))
 (<- (male abel)))

(defpredicate
 (<- (female eva)))

(defpredicate
 (<- (father ?x) (child ?y ?x) (male ?x)))
```

Cíl má tvar:

- Existují ***var1***, ..., ***varm*** tak, že ***atom1*** a ... a ***atomn***.

Kde:

- ***atomi*** jsou atomické výrokové formy
- a ***vari*** jsou všechny proměnné, které se v nich vyskytují.

Jak jsme zvyklí, cíl zapíšeme seznamem `(? atom1 ... atomn)`. Například cíl „ Existuje x tak, že x je otec.“ zapíšeme seznamem `(? (father ?x))`.

2 Unifikace

Připomeňme si definici unifikátoru dvou výrazů z minulé přednášky. Substituce ***sub*** je **unifikátor** výrazů ***expr1*** a ***expr2***, jestliže aplikací ***sub*** na ***expr1*** a ***expr2*** dostaneme týž výraz. Příklady unifikátorů výrazů `(child ?x ?y)` a `(child abel ?z)`:

1. `(sub (?x . abel) (?y . ?z))`
2. `(sub (?x . abel) (?y . eva) (?z . eva))`

Jak následující příklady ukazují, unifikátor výrazů nemusí existovat.

1. `(a b)`, `(?x ?x)`
2. `(?x)`, `?x`

Výrazy jsou **unifikovatelné**, pokud existuje jejich unifikátor.

Vezměme unifikátory *sub1* a *sub2* výrazů *expr1* a *expr2*. Označme *expr3* výsledek aplikace *sub1* na *expr1* (nebo *expr2*) a *expr4* výsledek aplikace *sub2* na *expr1* (nebo *expr2*). Unifikátor *sub1* je **obecnější** než *sub2*, jestliže *expr4* je instance *expr3*. Například unifikátor výrazů `(child ?x ?y)` a `(child abel ?z)`:

- `(sub (?x . abel) (?y . ?z))`

je obecnější než:

- `(sub (?x . abel) (?y . eva) (?z . eva))`

Nejobecnější unifikátor dvou výrazů, je jejich unifikátor, který je obecnější než každý jejich unifikátor. Pokud jsou výrazy unifikovatelné, pak mají nejobecnější unifikátor. Například nejobecnější unifikátor výrazů:

- `(child ?x ?y)`
- `(child abel ?z)`

je:

- `(sub (?x . abel) (?y . ?z))`

Postupně si představíme algoritmus, který rozhodne, zda jsou dva výrazy *expr1* a *expr2*, na které je použita substituce *sub*, unifikovatelné. V kladném případě algoritmus vrátí jejich nejobecnější unifikátor.

Postup **unifikace výrazů** *expr1* a *expr2* se substitucí *sub*:

1. Vytvoříme nový výraz *expr1'* vzniklý aplikací *sub* na *expr1*.
2. Vytvoříme nový výraz *expr2'* vzniklý aplikací *sub* na *expr2*.
3. Jestliže *expr1'* = *expr2'*, výsledkem je substituce *sub*.
4. Je-li *expr1'* proměnná, výsledkem je unifikace proměnné *expr1'* a *expr2'* se *sub*.
5. Je-li *expr2'* proměnná, výsledkem je unifikace proměnné *expr2'* a *expr1'* se *sub*.
6. Jsou-li *expr1'* a *expr2'* páry, výsledkem je unifikace párů *expr1'* a *expr2'* se *sub*.

7. Jinak nejsou unifikovatelné

Unifikace proměnné *var* a výrazu *expr* se substitucí *sub* probíhá takto:

1. zkontroluje se, zda se *var* nevyskytuje v *expr*.
2. Pokud se vyskytuje, nejsou unifikovatelné,
3. jinak se přidá pár `(var . expr)` k substituci *sub*.

Přidání páru k substituci zůstává stejné jako v minulé přednášce. Pro úplnost postup zopakujeme. Symbol *sub* označuje substituci `(sub (var1 . val1) ... (varn . valn))`.

Přidání páru `(var . val)` k substituci *sub* probíhá takto:

1. zkontroluje se, zda *var* není rovno některému *vari* ze substituce. Pokud je, je to chyba,
2. zkontroluje se, zda se *var* nevyskytuje v *val*. Pokud se vyskytuje, je to chyba,
3. na všechna *vali* se aplikuje substituce `(sub (var . val))`,
4. pár se přidá zepředu k *sub*.

Postup **unifikace párů** `(a1 . a2)` s `(b1 . b2)` se substitucí *sub*:

1. Unifikujeme *a1* a *b1* se *sub*, pokud lze unifikovat, obdržíme substituci *sub2*, jinak nelze unifikovat.
2. Unifikujeme *a2* a *b2* se *sub2*, pokud lze unifikovat, obdržíme substituci *sub3*, jinak nelze unifikovat.
3. Výsledkem je substituce *sub3*.

Například unifikací párů:

- `(a . a)` a `(?x . ?x)`
- s `(sub (?y . ?x))`

obdržíme:

- `(sub (?x . a) (?y . a))`

3 Odpověď na dotaz

Nejprve si zavedeme jeden pomocný pojem. Vezměme substituci *sub* a seznam proměnných *vars*. **Zúžením** *sub* na *vars* obdržíme substituci, jejíž dvojice jsou právě ty dvojice ze *sub*, které v *car* mají proměnnou náležící *vars*. Například zúžením:

- `(sub (?x . a) (?y . b) (?z . c)))`

na:

- `(?x ?z)`

obdržíme:

- `(sub (?x . a) (?z . c)))`

Aby bylo možné odpovědět na dotaz uživatele, je potřeba ve stromě úsudků k cílům přiřadit vznikající odpověď, která má podobu substituce. **Stav** je seznam `(state` *goal sub*`)`, kde *goal* je cíl a *sub* je substituce. Například:

- `(state (? (male eva)) (sub (?x . eva)))`

Výchozím stavem pro dotaz *query* je stav `(state` *query* `(sub))`. Například:

- `(state (? (father ?x)) (sub))`

je výchozí stav pro dotaz `(? (father ?x))`. Pokud je ve stavu `(state` *goal sub*`)` cíl *goal* prázdný, pak se *sub* nazývá **odpověď**. Například pro stav:

- `(state (?) (sub (?x . abel)))`

je odpověď `(sub (?x . abel))`.

Čerstvá pravidla

Pravidlo nazveme **čerstvým** vzhledem k stavu, jestliže:

1. neobsahuje žádnou proměnnou, která se ve stavu vyskytuje,
2. a vzniklo záměnou proměnných z nějakého pravidla v programu.

Například:

- `(<- (father ?x1) (child ?y ?x1) (male ?x1))`

je čerstvým pravidlem vzhledem k:

- `(state (? (father ?x)) (sub))`

Použité pravidlo z programu:

- `(<- (father ?x) (child ?y ?x) (male ?x))`

Úsudek

Na stav:

- `(state (? atom1 atom2 ... atomn) sub)`

lze použít čerstvé pravidlo:

- `(<- head body1 ... bodym)`

jestliže *atom1* je unifikovatelný s *head*.

Použitím pravidla obdržíme stav:

- `(state (? body1' ... bodym' atom2' ... atomn') sub')`,

kde

- *unifier* je nejobecnější unifikátor *head* a *atom1* se *sub*,
- *bodyi'* je výsledek aplikace *unifier* na *bodyi*,
- *atomi'* je výsledek aplikace *unifier* na *atomi*,
- *sub'* je zúžení substituce *unifier* na *query-vars*
- a *query-vars* je seznam proměnných v dotazu.

Například uvažujme dotaz `(? (father ?x))`. Seznam `(?x)` je seznamem proměnných v dotazu. Níže naleznete dva příklady úsudků.

1. Výchozí stav:

   - `(state (? (father ?x)) (sub))`

   Čerstvé pravidlo:

   - `(<- (father ?x1) (child ?y ?x1) (male ?x1))`

   Unifikátor:

   - `(sub (?x1 . ?x))`

   Výsledný stav:

   - `(state (? (child ?y ?x) (male ?x)) (sub))`

2. Stav:

   - `(state (? (child ?y ?x) (male ?x)) (sub))`

Čerstvé pravidlo:

- `(<- (child abel eva))`

Unifikátor:

- `(sub (?y . abel) (?x . eva))`

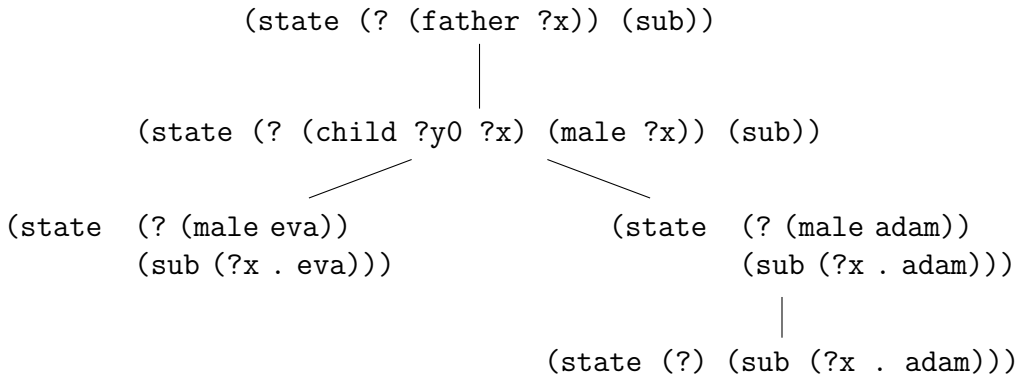
Výsledný stav:

- `(state (? (male eva)) (sub (?x . eva)))`

Strom úsudků

Stromy úsudků tvoříme stejně, jako v minulé přednášce, až na to, že hodnoty uzlů jsou stavy. Každá odpověď ve stromu je (**vypočítanou**) **odpovědí** na dotaz. Pokud existuje aspoň jedna odpověď, pak je cíl dotazu **splněný**.

Strom úsudků pro `(? (father ?x))`:



Tedy cíl `(? (father ?x))` je splněn a jediná odpověď na dotaz `(? (father ?x))` je `(sub (?x . adam))`.

Odpovědí na dotaz může být více. Například uvažujme predikát:

```
(defpredicate
  (<- (member ?x (?x . ?xs)))
  (<- (member ?x (?y . ?xs))
      (member ?x ?xs)))
```

Dotazem `(? (member ?x (a b)))` získáme dvě odpovědi:

- `(sub (?x . a))`
- `(sub (?x . b))`

Skutečně, stačí si prohlédnout strom úsudků:

```
(state (? (member ?x (a b))) (sub))

(state (?) (sub (?x . a)))   (state  (? (member ?x (b)))
                                     (sub))

                  (state (?) (sub (?x . b)))   (state  (? (member ?x ()))
                                                       (sub))
```

Jak následující příklad ukazuje, strom úsudků může být nekonečný. Strom úsudků pro `(? (member a ?x))`:

```
                    (state  (? (member a ?x))
                            (sub))

 (state  (?)                      (state  (? (member a ?xs0))
         (sub (?x . (a . ?xs0))))         (sub (?x . (?y0 . ?xs0))))

               (state  (?)
                       (sub (?x . (?y0 a . ?xs1))))                 ⋮
```

Strom obsahuje nekonečně mnoho odpovědí:

1. `(sub (?x . (a . ?xs0)))`
2. `(sub (?x . (?y0 a . ?xs1))`
3. `(sub (?x . (?y0 ?y1 a . ?xs2))`
   ⋮

4 Reprezentace seznamů a čísel

Seznamy

V programu můžeme seznamy reprezentovat přímo. Například `(a b c)`. V predikátové logice je reprezentujeme pomocí

1. binárního funkčního symbolu *cons*
2. a konstanty *nil*.

Například seznam `(a b c)` reprezentujeme termem $cons(a, cons(b, cons(c, nil)))$.

Zavedeme relační symbol *list* pro správně utvořené seznamy. Teorie seznamů

$$\{list(nil), (\forall x, y)(list(y) \rightarrow list(cons(x, y)))\}$$

popisuje význam predikátu *list*. Konkrétně term L je správně utvořený seznam, právě když $list(L)$ vyplývá z teorie seznamů.

Model teorie seznamů

Zamýšleným modelem M teorie seznamů jsou konečné posloupnosti.

1. nil^M je prázdná posloupnost
2. $cons^M$ je funkce přidání prvku na začátek posloupnosti

Například:

- $cons(a, cons(b, cons(c, nil)))^M = (a, b, c)$

Přirozená čísla s nulou

V programu čísla reprezentujeme následujícími objekty.

1. Číslo nula označíme `zero`.
2. Pokud ***n*** je číslo, pak `(succ` ***n*** `)` je následník ***n***.

Například `(succ (succ zero))` je číslo dva. Podobně v predikátové logice uvažujeme:

1. konstantu *zero*
2. a unární funkční symbol *succ*.

Teorie čísel (relační symbol *number*):

$$\{number(zero), (\forall x)(number(x) \rightarrow number(succ(x)))\}$$

Model čísel

Zamýšlený model M teorie čísel je množina přirozených čísel s nulou a

1. $zero^M = 0$
2. $succ^M(n) = n + 1$

Například:

- $succ(succ(zero))^M = 2$

5 Pohled predikátové logiky

Připomeňme, že program chápeme jako teorii. Například:

$$T = \{(\forall x, y)member(x, cons(x, y)),$$
$$(\forall x, y, z)(member(x, y) \rightarrow member(x, cons(z, y)))\}$$

Cíl

- `(?` ψ_1 `...` ψ_n`)`

je formule

- $\varphi = (\exists x_1, \ldots, x_n)(\psi_1 \wedge \cdots \wedge \psi_n)$,

kde $x_1, \ldots, x_n$ jsou všechny proměnné ve $\psi_1, \ldots, \psi_n$.
Například: $\varphi = (\exists x, y)member(1, cons(x, cons(y, nil)))$

Substituci chápeme jako množinu jejích dvojic. Například substituci `(sub (?x . a) (?y . b))` chápeme jako množinu $\{\langle x, a\rangle, \langle y, b\rangle\}$. Aplikaci substituce θ na formuli bez kvantifikátorů φ značíme $\varphi\theta$.

Například aplikací substituce $\theta = \{\langle x, a\rangle, \langle y, b\rangle\}$ na formuli

$$\varphi = member(a, cons(x, cons(y, cons(z, nil))))$$

získáme formuli $\varphi\theta = member(a, cons(a, cons(b, cons(z, nil))))$.

Vezměme formuli φ bez kvantifikátorů a označme $x_1, \ldots, x_n$ všechny proměnné formule φ. Zápis $\forall\varphi$ je zkratkou za formuli $(\forall x_1, \ldots, x_n)\varphi$ a zápis $\exists\varphi$ je zkratkou za formuli $(\exists x_1, \ldots, x_n)\varphi$. Například $\forall member(x, cons(x, y)$ je zkratkou za $(\forall x, y)member(x, cons(x, y))$.

Logické odpovědi

Substituce θ je **logickou odpovědí** na $\exists\varphi$ vzhledem k T, jestliže $T \models \forall(\varphi\theta)$. Příklady logických odpovědí pro dotaz:

- $(\exists x, y)member(a, cons(x, cons(y, nil)))$

vzhledem k T:

- $\{\langle x, a\rangle, \langle y, b\rangle\}$
- $\{\langle x, b\rangle, \langle y, a\rangle\}$
- $\{\langle x, a\rangle\}$
- $\{\langle y, a\rangle\}$

Korektnost

Korektnost našeho Prologu můžeme vyjádřit následujícím tvrzením.

> Každá vypočítaná odpověď na dotaz vzhledem k programu je logickou odpovědí.

Například vezměme program k teorii T:

```
(<- (member ?x (?x . ?xs)))
(<- (member ?x (?y . ?xs))
    (member ?x ?xs))
```

a položme dotaz:

- `(? (member a (?x ?y)))`

Jedna z vypočítaných odpovědí je:

- `(sub (?x . a))`

Dotaz vyjádřený formulí je $\exists member(a, cons(x, cons(y, nil)))$ vypočítanou odpovědí je substituce $\theta = \{\langle x, a\rangle\}$. Korektnost udává, že θ je i logickou odpovědí. Tedy, že formule

$$\forall(member(a, cons(x, cons(y, nil)))\theta) = (\forall y)member(a, cons(a, cons(y, nil)))$$

plyne z T.

Úplnost

Úplnost našeho Prologu vyjadřuje následující tvrzení.

> Jestliže je θ_1 je logická odpověď na $\exists\varphi$ vzhledem k T, pak existuje vypočítaná odpověď θ_2 na dotaz φ vzhledem k T taková, že $\varphi\theta_1$ je instancí $\varphi\theta_2$.

Ve stejném programu:

```
(<- (member ?x (?x . ?xs)))
(<- (member ?x (?y . ?xs))
    (member ?x ?xs))
```

položme například dotaz:

- `(? (member a (?x ?y)))`

Vezměme nějakou logickou odpověď:

- `(sub (?x . a) (?y . b))`

K ní jsme schopni najít vypočítanou odpověď:

- `(sub (?x . a))`

Platí, že `(member a (a b))` je instancí `(member a (a ?y))`. Totéž můžeme vyjádřit pomocí formulí. Dotaz je stále formule $\exists\varphi$, kde φ je $member(a, cons(x, cons(y, nil)))$, logická odpověď je substituce $\theta_1 = \{\langle x, a\rangle, \langle y, b\rangle\}$, vypočítaná odpověď je substituce $\theta_2 = \{\langle x, a\rangle\}$. Platí, že $\varphi\theta_1 = member(a, cons(a, cons(b, nil)))$ je instancí $\varphi\theta_2 = member(a, cons(a, cons(y, nil)))$.

6 Líné stromy úsudků

Protože stromy úsudků mohou být nekonečné, musíme přejít k líné implementaci stromů. Zde pouze nastíníme hlavní myšlenku líného stromu. Pro detailní rozpracování myšlenky se podívejte do zdrojového kódu přiloženého k přednášce. Uzel líného stromu úsudků je seznam tvaru:

```
(lazy-node state . children)
```

kde

- *`state`* je stav,
- *`children`* je příslib seznamu následníků.

Průchodem líným stromem úsudků získáme proud odpovědí. Průchodem do šířky získáme každou odpověď. Pokud neexistuje odpověď a strom je nekonečný, výpočet nikdy neskončí. Průchodem nekonečného stromu do hloubky nemusíme získat všechny odpovědi.