Paradigmata programování 4

Přednáška 11. Proměnné v dotazech

verze z 22. dubna 2025

Jan Laštovička

KATEDRA INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI

Obsah

Motivace

Motivace

Fakta:

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

Motivace

Fakta:

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

Pravidlo:

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

Motivace

Fakta:

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

Pravidlo:

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

Dotaz: Kdo je otec?

Pravidla

Pravidla

Pravidlo je opět výrok tvaru:

- Pro každé $var1, \ldots, varm$ platí, že jestliže $body1$ a $\ldots$ a $bodyn$, pak $head$.

Kde:

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

Pravidla

Pravidlo je opět výrok tvaru:

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

Kde:

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

Stále zapisujeme: `(<- head body1 ... bodyn)`

Pravidla

Pravidlo je opět výrok tvaru:

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

Kde:

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

Stále zapisujeme: `(<- head body1 ... bodyn)`

Omezení z minulé přednášky již neplatí.

Pravidla

Pravidlo je opět výrok tvaru:

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

Kde:

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

Stále zapisujeme: `(<- head body1 ... bodyn)`

Omezení z minulé přednášky již neplatí.

Například:

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

Což je ekvivalentní s:

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

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

Program

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

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

(defpredicate
 (<- (female eva)))

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

Cíle

Cíle

Cíl má tvar:

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

Kde:

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

Cíle

Cíl má tvar:

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

Kde:

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

Zapíšeme: `(? atom1 ... atomn)`

Cíle

Cíl má tvar:

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

Kde:

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

Zapíšeme: `(? atom1 ... atomn)`

Například:

- Existuje x tak, že x je otec.
  `... (? (father ?x))`

Obsah

Unifikátor

Unifikátor

Substituce *sub* je unifikátor výrazů *expr1* a *expr2*,

- jestliže aplikací *sub* na *expr1* a *expr2* dostaneme týž výraz.

Unifikátor

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))`

Unifikátor

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))`

Unifikátor výrazů nemusí existovat:

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

Unifikátor

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))`

Unifikátor výrazů nemusí existovat:

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

Výrazy jsou unifikovatelné, pokud existuje jejich unifikátor.

Obecnost unifikátorů

Obecnost unifikátorů

- *sub1*, *sub2* ... unifikátory výrazů *expr1* a *expr2*
- *expr3* ... výsledek aplikace *sub1* na *expr1* (nebo *expr2*)
- *expr4* ... výsledek aplikace *sub2* na *expr1* (nebo *expr2*)

Obecnost unifikátorů

- *sub1*, *sub2* ... unifikátory výrazů *expr1* a *expr2*
- *expr3* ... výsledek aplikace *sub1* na *expr1* (nebo *expr2*)
- *expr4* ... výsledek aplikace *sub2* na *expr1* (nebo *expr2*)

Unifikátor *sub1* je obecnější než *sub2*,

- jestliže *expr4* je instance *expr3*.

Obecnost unifikátorů

- *sub1*, *sub2* ... unifikátory výrazů *expr1* a *expr2*
- *expr3* ... výsledek aplikace *sub1* na *expr1* (nebo *expr2*)
- *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

Nejobecnější unifikátor

Pokud jsou výrazy unifikovatelné, pak mají nejobecnější unifikátor.

Nejobecnější 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))`

Unifikace

Unifikace

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

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

- Pokud jsou výrazy unifikovatelné, dostaneme nejobecnější unifikátor.

Unifikace

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

- Pokud jsou výrazy unifikovatelné, dostaneme nejobecnější unifikátor.
- Výchozí hodnota pro *sub* je prázdná substituce.

Unifikace proměnné a výrazu

Unifikace proměnné a výrazu

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

Přidání páru k substituci

Přidání páru zůstává stejné jako v minulé přednášce.

Přidání páru k substituci

Přidání páru zůstává stejné jako v minulé přednášce.

- *sub* ... (sub (*var1* . *val1*) ... (*varn* . *valn*))

Přidání páru k substituci

Přidání páru zůstává stejné jako v minulé přednášce.

- *sub* ... (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*.

Unifikace páru

Unifikace páru

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*.

Unifikace páru

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:

1. `(a . a)` a `(?x . ?x)`
2. `s (sub (?y . ?x))`

obdržíme:

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

Obsah

Zúžení substituce

Zúžení substituce

- sub ... substituce
- $vars$ ... seznam proměnných

Zúžením sub na $vars$ obdržíme substituci,

- jejíž dvojice jsou právě ty dvojice ze sub,
- které v *car* mají proměnnou náležící $vars$.

Zúžení substituce

- *sub* ... substituce
- *vars* ... seznam proměnných

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)))`

Stav

Stav

Stav je seznam `(state` *goal sub*`)`,

1. kde *goal* je cíl
2. a *sub* je substituce.

Stav

Stav je seznam `(state` *goal sub*`)`,

1. kde *goal* je cíl
2. a *sub* je substituce.

Například:

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

Stav

Stav je seznam `(state` *goal sub*`)`,

1. kde *goal* je cíl
2. a *sub* je substituce.

Například:

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

Výchozí stav pro dotaz *query*: `(state` *query* `(sub))`

Stav

Stav je seznam `(state` *goal sub*`)`,

1. kde *goal* je cíl
2. a *sub* je substituce.

Například:

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

Výchozí stav pro dotaz *query*: `(state` *query* `(sub))`

Například:

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

Stav

Stav je seznam `(state` *goal sub*`)`,

1. kde *goal* je cíl
2. a *sub* je substituce.

Například:

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

Výchozí stav pro dotaz *query*: `(state` *query* `(sub))`

Například:

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

Pokud *goal* je prázdný cíl, pak *sub* je odpověď.

Stav

Stav je seznam `(state` *goal sub*`)`,

1. kde *goal* je cíl
2. a *sub* je substituce.

Například:

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

Výchozí stav pro dotaz *query*: `(state` *query* `(sub))`

Například:

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

Pokud *goal* je prázdný cíl, pak *sub* je odpověď.

Například:

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

Čerstvá pravidla

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.

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 ?x0) (child ?y0 ?x0) (male ?x0))`

je čerstvým pravidlem vzhledem k:

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

Pravidlo z programu:

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

Úsudek

Úsudek

Na stav:

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

lze použít čerstvé pravidlo:

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

jestliže *atom1* je unifikovatelný s *head*.

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.

Příklady úsudků

Příklady úsudků

Dotaz: `(? (father ?x))`

- Proměnné v dotazu: `(?x)`

Příklady úsudků

Dotaz: `(? (father ?x))`

- Proměnné v dotazu: `(?x)`

1. Výchozí stav:
   - `(state (? (father ?x)) (sub))`

   Čerstvé pravidlo:
   - `(<- (father ?x0) (child ?y0 ?x0) (male ?x0))`

   Unifikátor:
   - `(sub (?x0 . ?x))`

   Výsledný stav:
   - `(state (? (child ?y0 ?x) (male ?x)) (sub))`

Příklady úsudků

Dotaz: `(? (father ?x))`

- Proměnné v dotazu: `(?x)`

1. Výchozí stav:
   - `(state (? (father ?x)) (sub))`

   Čerstvé pravidlo:
   - `(<- (father ?x0) (child ?y0 ?x0) (male ?x0))`

   Unifikátor:
   - `(sub (?x0 . ?x))`

   Výsledný stav:
   - `(state (? (child ?y0 ?x) (male ?x)) (sub))`
2. Stav:
   - `(state (? (child ?y0 ?x) (male ?x)) (sub))`

   Čerstvé pravidlo:
   - `(<- (child abel eva))`

   Unifikátor:
   - `(sub (?y0 . abel) (?x . eva))`

   Výsledný stav:
   - `(state (? (male eva)) (sub (?x . eva)))`

Strom úsudků

Strom úsudků

- Strom úsudků se tvoří stejně jako v minulé přednášce.
- Hodnoty uzlů jsou stavy.

Strom úsudků

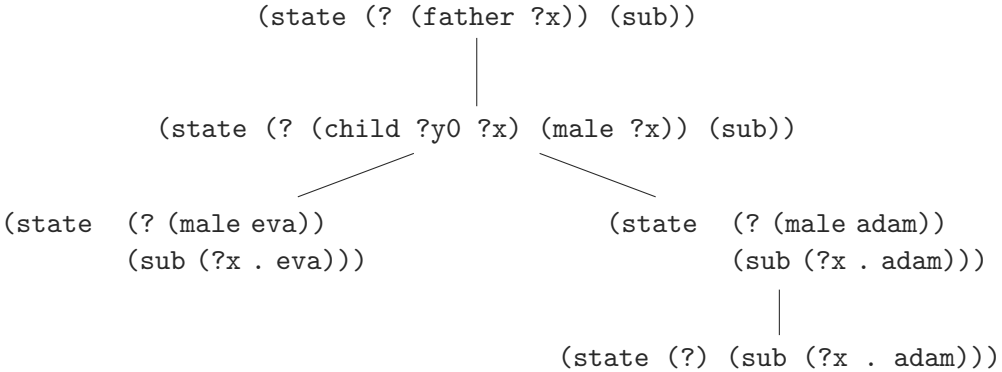
- Strom úsudků se tvoří stejně jako v minulé přednášce.
- Hodnoty uzlů jsou stavy.

- Každá odpověď ve stromu je (vypočítanou) odpovědí na dotaz.
- Pokud existuje aspoň jedna odpověď, pak je cíl dotazu splněný.

Příklad

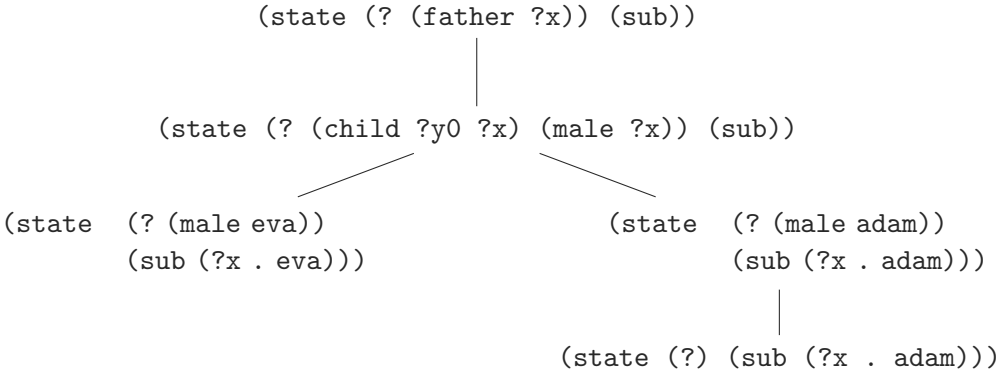
Příklad

Strom úsudků pro `(? (father ?x))`:



Příklad

Strom úsudků pro `(? (father ?x))`:



- Cíl `(? (father ?x))` je splněn.
- Jediná odpověď na dotaz `(? (father ?x))` je `(sub (?x . adam))`.

Více odpovědí na dotaz

Více odpovědí na dotaz

Pravidla:

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

Více odpovědí na dotaz

Pravidla:

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

Dotaz: `(? (member ?x (a b)))`

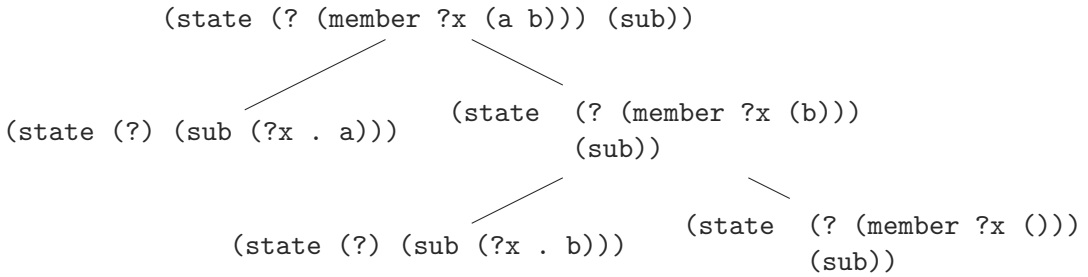
Odpovědi:

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

Strom úsudků

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

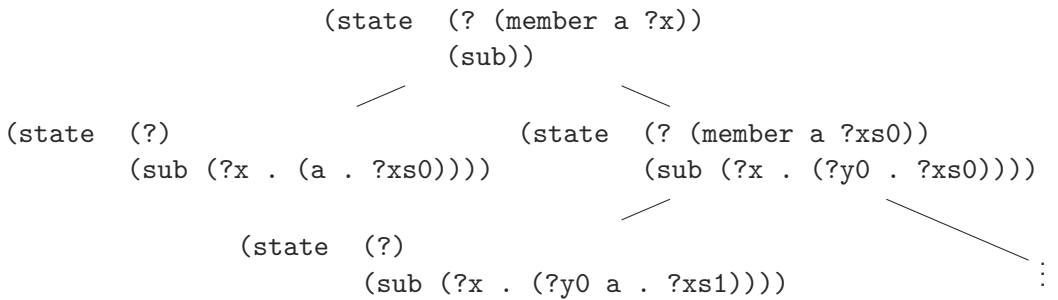
Strom úsudků pro `(? (member ?x (a b)))`:



Nekonečný strom úsudků

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

Strom úsudků pro `(? (member a ?x))`:



Nekonečně mnoho odpovědí

Odpovědi pro `(? (member a ?x))`:

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

⋮

Obsah

Seznamy

Seznamy

V programu můžeme seznamy reprezentovat přímo.

Například: `(a b c)`

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.

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:

- `(a b c)` ... $cons(a, cons(b, cons(c, nil)))$

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:

- `(a b c)` ... $cons(a, cons(b, cons(c, nil)))$

Zavedeme relační symbol $list$ pro správně utvořené seznamy.

Teorie seznamů:

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

Model 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

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

Přirozená čísla s nulou

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

Přirozená čísla s nulou

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

Například:

- `(succ (succ zero))` je číslo dva

Přirozená čísla s nulou

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 $succ$.

Přirozená čísla s nulou

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

Model čísel

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

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

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:

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

Obsah

Proměnné v dotazu

Proměnné v dotazu

Připomeňme, že program chápeme jako teorii.

Proměnné v dotazu

Připomeňme, že program chápeme jako teorii.

Například:

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

Proměnné v dotazu

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 $\psi_1, \ldots, \psi_n$.

Proměnné v dotazu

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: $\varphi = (\exists x, y) member(1, cons(x, cons(y, nil)))$

Substituce

Substituce

Substituci chápeme jako množinu jejích dvojic.

Substituce

Substituci chápeme jako množinu jejích dvojic.

Například:

- `(sub (?x . a) (?y . b))` ... $\{\langle x, a\rangle, \langle y, b\rangle\}$

Substituce

Substituci chápeme jako množinu jejích dvojic.

Například:

- `(sub (?x . a) (?y . b))` ... $\{\langle x, a\rangle, \langle y, b\rangle\}$

Aplikaci substituce θ na formuli bez kvantifikátorů φ značíme $\varphi\theta$.

Například:

Substituce

Substituci chápeme jako množinu jejích dvojic.

Například:

- `(sub (?x . a) (?y . b))` ... $\{\langle x, a\rangle, \langle y, b\rangle\}$

Aplikaci substituce θ na formuli bez kvantifikátorů φ značíme $\varphi\theta$.

Například:

- $\theta = \{\langle x, a\rangle, \langle y, b\rangle\}$
- $\varphi = member(a, cons(x, cons(y, cons(z, nil))))$
- $\varphi\theta = member(a, cons(a, cons(b, cons(z, nil))))$

Uzávěry

Uzávěry

- φ ... formule bez kvantifikátorů
- $x_1, \ldots, x_n$ jsou všechny proměnné z φ

Uzávěry

- φ ... formule bez kvantifikátorů
- $x_1, \ldots, x_n$ jsou všechny proměnné z φ

Zápis:

- $\forall\varphi$

je zkratkou za:

- $(\forall x_1, \ldots, x_n)\varphi$

Uzávěry

- φ ... formule bez kvantifikátorů
- $x_1, \ldots, x_n$ jsou všechny proměnné z φ

Zápis:

- $\forall\varphi$

je zkratkou za:

- $(\forall x_1, \ldots, x_n)\varphi$

Zápis:

- $\exists\varphi$

je zkratkou za:

- $(\exists x_1, \ldots, x_n)\varphi$

Logické odpovědi

Logické odpovědi

Substituce θ je logickou odpovědí na $\exists\varphi$ vzhledem k T,

- jestliže $T \vDash \forall(\varphi\theta)$.

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:

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

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

Korektnost

Korektnost

Platí:

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

Korektnost

Platí:

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

Například program:

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

Dotaz:

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

Vypočítaná odpověď:

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

Z programu plyne:

- `(member a (a ?y))` ... $(\forall y) member(a, cons(a, cons(y, nil)))$

Úplnost

Úplnost

Platí:

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 $\varphi\theta_2$.

Úplnost

Platí:

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 $\varphi\theta_2$.

Například program:

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

Dotaz:

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

Logická odpověď:

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

Vypočítaná odpověď:

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

Platí, že `(member a (a b))` je instancí `(member a (a ?y))`.

Obsah

Líné stromy úsudků

Líné stromy úsudků

Uzel líného stromu úsudků je seznam tvaru:

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

kde

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

Líné stromy úsudků

Uzel líného stromu úsudků je seznam tvaru:

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

kde

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

- Průchodem získáme proud odpovědí.

Líné stromy úsudků

Uzel líného stromu úsudků je seznam tvaru:

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

kde

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

- Průchodem získáme proud odpovědí.
- Průchodem do šířky získáme každou odpověď.

Líné stromy úsudků

Uzel líného stromu úsudků je seznam tvaru:

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

kde

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

- Průchodem získáme proud odpovědí.
- Průchodem do šířky získáme každou odpověď.
- Pokud neexistuje odpověď a strom je nekonečný, výpočet nikdy neskončí.

Líné stromy úsudků

Uzel líného stromu úsudků je seznam tvaru:

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

kde

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

- Průchodem 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.