Paradigmata programování 4

Přednáška 3. Programová logika

verze z 27. února 2025

Jan Laštovička

KATEDRA INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI

Problémy

Problémy

Interpret Scheamu je pomalý.

Problémy

Interpret Scheamu je pomalý.

Makra se expandují při vyhodnocování.

Problémy

Interpret Scheamu je pomalý.

Makra se expandují při vyhodnocování.

Možná řešení:

- Expandovat makra před vyhodnocením výrazu.
- Přepsat kritická makra na speciální operátory.

Problémy

Interpret Scheamu je pomalý.

Makra se expandují při vyhodnocování.

Možná řešení:

- Expandovat makra před vyhodnocením výrazu.
- Přepsat kritická makra na speciální operátory.

Makro co upřednostňuje dříve vytvořené procesy.

Problémy

Interpret Scheamu je pomalý.

Makra se expandují při vyhodnocování.

Možná řešení:

- Expandovat makra před vyhodnocením výrazu.
- Přepsat kritická makra na speciální operátory.

Makro co upřednostňuje dříve vytvořené procesy.

Řešení:

- Proces počká na vytvoření všech procesů.

Ukázka:

```
(co (print 1) (print 2))
```

Obsah

Stav programu

. . . omezíme jen na proměnné

Například:

`x`	`1`
`y`	`1`

- vhodný stav pro program nebo výraz

Například: `(>= x y)`

Podmínky

Podmínka je jednoduchý výraz, jehož hodnotu chápeme jako logickou.

Podmínky

Podmínka je jednoduchý výraz, jehož hodnotu chápeme jako logickou.

Přidáme podmínky:

- `(not` *`condition`*`)`
- `(and` *`condition1 condition2 ...`*`)`
- `(or` *`condition1 condition2 ...`*`)`

Podmínky

Podmínka je jednoduchý výraz, jehož hodnotu chápeme jako logickou.

Přidáme podmínky:

- `(not` *`condition`*`)`
- `(and` *`condition1 condition2 ...`*`)`
- `(or` *`condition1 condition2 ...`*`)`

podmínka je splněna ve stavu = vyhodnotí se ve stavu na *Pravdu*

Podmínky

Podmínka je jednoduchý výraz, jehož hodnotu chápeme jako logickou.

Přidáme podmínky:

- `(not` *`condition`*`)`
- `(and` *`condition1 condition2 ...`*`)`
- `(or` *`condition1 condition2 ...`*`)`

podmínka je splněna ve stavu = vyhodnotí se ve stavu na *Pravdu*

Například podmínka `(and (= x 1) (> y 0))` je ve stavu

`x`	`1`
`y`	`1`

splněna.

Zesílení a oslabení podmínky

Zesílení a oslabení podmínky

Podmínka *condition1* je silnější než podmínka *condition2*,

- jestliže pro každý vhodný stav S platí následující implikace.
- Pokud stav S splňuje podmínku *condition1*,
- pak splňuje i podmínku *condition2*.

Zesílení a oslabení podmínky

Podmínka *condition1* je silnější než podmínka *condition2*,

- jestliže pro každý vhodný stav S platí následující implikace.
- Pokud stav S splňuje podmínku *condition1*,
- pak splňuje i podmínku *condition2*.

Například podmínka `(> x 1)` je silnější než podmínka `(>= x 1)`.

Zesílení a oslabení podmínky

Podmínka *condition1* je silnější než podmínka *condition2*,

- jestliže pro každý vhodný stav S platí následující implikace.
- Pokud stav S splňuje podmínku *condition1*,
- pak splňuje i podmínku *condition2*.

Například podmínka `(> x 1)` je silnější než podmínka `(>= x 1)`.

Podmínka *condition1* je slabší než podmínka *condition2*,

- jestliže podmínka *condition2* je silnější než podmínka *condition1*.

Zesílení a oslabení podmínky

Podmínka *condition1* je silnější než podmínka *condition2*,

- jestliže pro každý vhodný stav S platí následující implikace.
- Pokud stav S splňuje podmínku *condition1*,
- pak splňuje i podmínku *condition2*.

Například podmínka `(> x 1)` je silnější než podmínka `(>= x 1)`.

Podmínka *condition1* je slabší než podmínka *condition2*,

- jestliže podmínka *condition2* je silnější než podmínka *condition1*.

Podmínky *condition1* a *condition2* jsou ekvivalentní,

- jestliže podmínka *condition1* je silnější a současně slabší než podmínka *condition2*.

Zesílení a oslabení podmínky

Podmínka *condition1* je silnější než podmínka *condition2*,

- jestliže pro každý vhodný stav S platí následující implikace.
- Pokud stav S splňuje podmínku *condition1*,
- pak splňuje i podmínku *condition2*.

Například podmínka `(> x 1)` je silnější než podmínka `(>= x 1)`.

Podmínka *condition1* je slabší než podmínka *condition2*,

- jestliže podmínka *condition2* je silnější než podmínka *condition1*.

Podmínky *condition1* a *condition2* jsou ekvivalentní,

- jestliže podmínka *condition1* je silnější a současně slabší než podmínka *condition2*.

Například podmínky `(= x x)` a `t` jsou ekvivalentní.

Tvrzení

```
(assert condition)
```

1. Vyhodnotí se podmínka *condition*.
2. Pokud podmínka není splněna, vyvolá se chyba.
3. Jinak se neudělá nic.

Tvrzení

```
(assert condition)
```

1. Vyhodnotí se podmínka *condition*.
2. Pokud podmínka není splněna, vyvolá se chyba.
3. Jinak se neudělá nic.

Zapisujeme:

```
{condition}
```

Například:

```
{(> x 0)}
```

Obsah

Program

- posloupnost výrazů

Například:

```
{(> x 0)}
(set! x (+ x 1))
(set! x (+ x 1))
{(> x 0)}
```

- spuštění programu vede k postupnému vyhodnocování jeho výrazů

Předpoklad a závěr program

- programy budou vždy začínat a končit tvrzením
- podmínka prvního tvrzení se nazývá předpoklad (prekondice) programu
- podmínka posledního tvrzení se nazývá závěr (postkondice) programu

Například :

```
{(> x 0)}
(set! x (- x 1))
{(>= x 0)}
```

Předpoklad a závěr program

- programy budou vždy začínat a končit tvrzením
- podmínka prvního tvrzení se nazývá předpoklad (prekondice) programu
- podmínka posledního tvrzení se nazývá závěr (postkondice) programu

Například :

```
{(> x 0)}
(set! x (- x 1))
{(>= x 0)}
```

Neuvedením předpokladu nebo závěru programu myslíme podmínku `t`.

Například program:

```
(set! x (+ x 1))
```

má předpoklad i závěr `t`.

Pravdivost programu

Program je pravdivý, jestliže každé spuštění programu ve stavu splňujícího jeho předpoklad neskončí chybou.

Pravdivost programu

Program je pravdivý, jestliže každé spuštění programu ve stavu splňujícího jeho předpoklad neskončí chybou.

Příklady:

Pravdivost programu

Program je pravdivý, jestliže každé spuštění programu ve stavu splňujícího jeho předpoklad neskončí chybou.

Příklady:

```
{(> x 0)}
(set! x (+ x 1))
{(> x 0)}
```

Pravdivost programu

Program je pravdivý, jestliže každé spuštění programu ve stavu splňujícího jeho předpoklad neskončí chybou.

Příklady:

```
{(> x 0)}
(set! x (+ x 1))
{(> x 0)}
```

pravdivý

Pravdivost programu

Program je pravdivý, jestliže každé spuštění programu ve stavu splňujícího jeho předpoklad neskončí chybou.

Příklady:

```
{(> x 0)}
(set! x (+ x 1))
{(> x 0)}
```

pravdivý

```
{(> x 0)}
(set! x (- x 1))
{(> x 0)}
```

Pravdivost programu

Program je pravdivý, jestliže každé spuštění programu ve stavu splňujícího jeho předpoklad neskončí chybou.

Příklady:

```
{(> x 0)}
(set! x (+ x 1))
{(> x 0)}
```

pravdivý

```
{(> x 0)}
(set! x (- x 1))
{(> x 0)}
```

Obsah

Pravidlo nic nedělání

Pravidlo nic nedělání

`nil` ... příkaz nic nedělání

Pravidlo nic nedělání

`nil` ... příkaz nic nedělání

Pro každou podmínku *condition* platí:

$$\{condition\}\ \texttt{nil}\ \{condition\}$$

Pravidlo nic nedělání

`nil` ... příkaz nic nedělání

Pro každou podmínku *condition* platí:

```
{condition}
nil
{condition}
```

Například:

```
{(= x 1)}
nil
{(= x 1)}
```

Pravidlo přiřazení

Pravidlo přiřazení

Pro každou podmínku $condition$ a příkaz přiřazení `(set!` var $expr$`)` platí:

$$\{condition'\}$$
$$\texttt{(set!}\ var\ expr\texttt{)}$$
$$\{condition\}$$

kde $condition'$ vznikne z $condition$ nahrazením všech výskytů proměnné var za $expr$.

Pravidlo přiřazení

Pro každou podmínku *condition* a příkaz přiřazení `(set!` *var expr*`)` platí:

$$\{condition'\}$$
$$\texttt{(set!}\ var\ expr\texttt{)}$$
$$\{condition\}$$

kde *condition'* vznikne z *condition* nahrazením všech výskytů proměnné *var* za *expr*.

Například:

```
{      }
(set! x (+ x 1))
{(= x 1)}
```

Pravidlo přiřazení

Pro každou podmínku *condition* a příkaz přiřazení `(set! var expr)` platí:

$$\{condition'\}$$
$$\texttt{(set! } var\ expr\texttt{)}$$
$$\{condition\}$$

kde *condition'* vznikne z *condition* nahrazením všech výskytů proměnné *var* za *expr*.

Například:

```
{(= (+ x 1) 1)}
(set! x (+ x 1))
{(= x 1)}
```

Pravidlo přiřazení

Pro každou podmínku *condition* a příkaz přiřazení `(set! var expr)` platí:

$$\{condition'\}$$
$$\texttt{(set! } var\ expr\texttt{)}$$
$$\{condition\}$$

kde *condition'* vznikne z *condition* nahrazením všech výskytů proměnné *var* za *expr*.

Například:

```
{(= x 0)}
(set! x (+ x 1))
{(= x 1)}
```

Zesílení předpokladu

Zesílení předpokladu

Pokud $\begin{matrix}\{pre1\}\\ program\end{matrix}$ a $pre2$ je silnější než $pre1$, pak $\begin{matrix}\{pre2\}\\ program\end{matrix}$.

Zesílení předpokladu

Pokud $\begin{matrix}\{pre1\}\\ program\end{matrix}$ a *pre2* je silnější než *pre1*, pak $\begin{matrix}\{pre2\}\\ program\end{matrix}$.

Například z $\begin{matrix}\texttt{\{t\}}\\ \texttt{(set! x 1)}\\ \texttt{\{(= x 1)\}}\end{matrix}$ plyne $\begin{matrix}\texttt{\{(= x 2)\}}\\ \texttt{(set! x 1)}\\ \texttt{\{(= x 1)\}}\end{matrix}$.

Oslabení závěru

Oslabení závěru

Pokud $\begin{array}{l} \mathit{program} \\ \{\mathit{post1}\} \end{array}$ a $\mathit{post2}$ je slabší než $\mathit{post1}$, pak $\begin{array}{l} \mathit{program} \\ \{\mathit{post2}\} \end{array}$.

Oslabení závěru

Pokud $\begin{array}{l} \mathit{program} \\ \{\mathit{post1}\} \end{array}$ a *post2* je slabší než *post1*, pak $\begin{array}{l} \mathit{program} \\ \{\mathit{post2}\} \end{array}$.

Například z
```
{t}
(set! x 1)
{(= x 1)}
```
plyne
```
{t}
(set! x 1)
{(or (= x 1) (= y 1))}
```

Skládání programů

Pokud $\begin{matrix} prog1 \\ \{cond\} \end{matrix}$ a $\begin{matrix} \{cond\} \\ prog2 \end{matrix}$, pak $\begin{matrix} prog1 \\ \{cond\} \\ prog2 \end{matrix}$.

Skládání programů

Pokud $\begin{matrix} prog1 \\ \{cond\} \end{matrix}$ a $\begin{matrix} \{cond\} \\ prog2 \end{matrix}$, pak $\begin{matrix} prog1 \\ \{cond\} \\ prog2 \end{matrix}$.

Například z
```
{t}
(set! x 1)
{(= x 1)}
```
a
```
{(= x 1)}
(set! y 2)
{(and (= x 1) (= y 2))}
```

plyne
```
{t}
(set! x 1)
{(= x 1)}
(set! y 2)
{(and (= x 1) (= y 2))}
```

Větvení

Pokud `{(and pre cond)} prog1 {post}` a `{(and pre (not cond))} prog2 {post}`, pak

```
{pre}
(if cond
  (begin
    {(and pre cond)}
    prog1
    {post})
  (begin
    {(and pre (not cond))}
    prog2
    {post}))
{post}
```

Příklad větvení

```
          {(>= x 0)}     {(< x 0)}
Například z nil          a (set! x (- 0 x))  plyne
          {(>= x 0)}     {(>= x 0)}
```

```
{t}
(if (>= x 0)
  (begin
    {(>= x 0)}
    nil
    {(>= x 0)})
  (begin
    {(< x 0)}
    (set! x (- 0 x))
    {(>= x 0)}))
{(>= x 0)}
```

Cyklus

Pokud
```
{(and inv cond)}
prog
{inv}
```
, pak
```
{inv}
(while cond
  {(and inv cond)}
  prog
  {inv})
{(and inv (not cond))}
```

Cyklus

Pokud
```
{(and inv cond)}
prog
{inv}
```
, pak
```
{inv}
(while cond
  {(and inv cond)}
  prog
  {inv})
{(and inv (not cond))}
```

Například z
```
{(< x 5}
(set! x (+ x 1))
{(<= x 5)}
```
plyne
```
{(<= x 5)}
(while (< x 5)
  {(< x 5)}
  (set! x (+ x 1))
  {(<= x 5)})
{(= x 5)}
```

Důkaz

Programy, které vzniknou použitím pravidel programové logiky, nazýváme důkazy.

Důkaz

Programy, které vzniknou použitím pravidel programové logiky, nazýváme důkazy.

Například:

```
{t}
(set! x 1)
{(= x 1)}
(set! y 2)
{(and (= x 1) (= y 2))}
```

předpoklad příkazu = podmínka tvrzení, které jej předchází

Důkaz pravdivosti programu

proof je důkazem pravdivosti *program*

... *proof* vznikne dodáním tvrzení do *program*

Například:

```
{t}
(set! x 1)
{(= x 1)}
(set! y 2)
{(and (= x 1) (= y 2))}
```

je důkazem pravdivosti:

```
{t}
(set! x 1)
(set! y 2)
{(and (= x 1) (= y 2))}
```

Obsah

Zachování podmínky

Příkaz *statement* s předpokladem *pre* zachovává podmínku *cond* jestliže:

```
{(and cond pre)}
statement
{cond}
```

Zachování podmínky

Příkaz *statement* s předpokladem *pre* zachovává podmínku *cond* jestliže:

```
{(and cond pre)}
statement
{cond}
```

Například příkaz `(set! x (- x 1))`
s předpokladem `(> x 0)`
zachovává podmínku `(>= x 0)`, protože:

```
{(> x 0)}
(set! x (- x 1))
{(>= x 0)}
```

Nenarušování důkazů

Dva důkazy se nenarušují, jestliže

- podmínka každého tvrzení z prvního důkazu je zachována všemi příkazy z druhého důkazu
- a naopak podmínka každého tvrzení z druhého důkazu je zachována všemi příkazy z prvního důkazu.

Nenarušování důkazů

Dva důkazy se nenarušují, jestliže

- podmínka každého tvrzení z prvního důkazu je zachována všemi příkazy z druhého důkazu
- a naopak podmínka každého tvrzení z druhého důkazu je zachována všemi příkazy z prvního důkazu.

Například:

```
{(= x 0)}        {(= x 0)}
(set! x 1)   a   (set! y x)
{(= x 1)}        {(= y 0)}
```

Nenarušování důkazů

Dva důkazy se nenarušují, jestliže

- podmínka každého tvrzení z prvního důkazu je zachována všemi příkazy z druhého důkazu
- a naopak podmínka každého tvrzení z druhého důkazu je zachována všemi příkazy z prvního důkazu.

Například:

```
{(= x 0)}        {(= x 0)}
(set! x 1)   a   (set! y x)
{(= x 1)}        {(= y 0)}
```

se narušují.

Souběžnost

Souběžnost

Pokud $\begin{matrix}\{pre1\}\\ prog1\\ \{post1\}\end{matrix}$ a $\begin{matrix}\{pre2\}\\ prog2\\ \{post2\}\end{matrix}$ jsou nenarušující se důkazy, pak

```
{(and pre1 pre2)}
(co
  (begin
    {pre1}
    prog1
    {post1})
  (begin
    {pre2}
    prog2
    {post2})
{(and post1 post2)}
```

Příklad souběžnosti

Například z

```
{t}
(set! x 1)
{(= x 1)}
```

a

```
{t}
(set! y 1)
{(= y 1)}
```

plyne

```
{t}
(co
  (begin
    {t}
    (set! x 1)
    {(= x 1)})
  (begin
    {t}
    (set! y 1)
    {(= y 1)}))
{(and (= x 1) (= y 1))}
```

Složené atomické operace

Program v hranatých závorkách se vykoná atomicky:

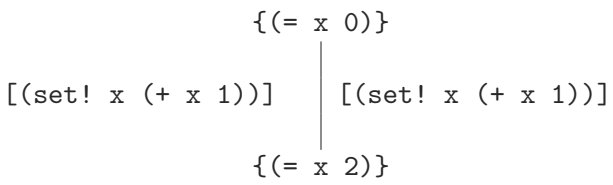
$$[program]$$

Složené atomické operace

Program v hranatých závorkách se vykoná atomicky:

```
[program]
```

Například:



Pravidlo sladění

Pokud $\begin{array}{l}\texttt{\{(and } pred\ cond\texttt{)\}} \\ prog \\ \texttt{\{}post\texttt{\}}\end{array}$, pak $\begin{array}{l}\texttt{\{}pred\texttt{\}} \\ \texttt{[(await } cond\texttt{)} \\ \ prog'\texttt{]} \\ \texttt{\{}post\texttt{\}}\end{array}$,

kde $prog'$ vznikne z $prog$ odstraněním všech tvrzení.

Pravidlo sladění

Pokud
```
{(and pred cond)}
prog
{post}
```
, pak
```
{pred}
[(await cond)
 prog']
{post}
```
,

kde `prog'` vznikne z `prog` odstraněním všech tvrzení.

Například z
```
{(= x 0)}
(set! x (+ x 1))
{(= x 1)}
(set! x (+ x 1))
{(= x 2)}
```
plyne
```
{t}
[(await (= x 0))
 (set! x (+ x 1))
 (set! x (+ x 1))]
{(= x 2)}
```
.

Obsah

Invarianty

Invariant programu je podmínka, která je splněna v každém jeho stavu.

Invarianty

Invariant programu je podmínka, která je splněna v každém jeho stavu.

Podmínka je invariantem, právě když

1. je splněna v počátečním stavu
2. a je zachována každým příkazem programu.

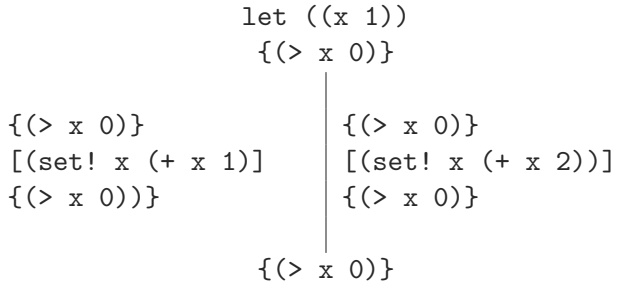
Invarianty

Invariant programu je podmínka, která je splněna v každém jeho stavu.

Podmínka je invariantem, právě když

1. je splněna v počátečním stavu
2. a je zachována každým příkazem programu.

Například `(> x 0)` je invariantem programu:



Obsah

Změna execute

```
(defun execute (&rest code)
  (let ((*ret* '())
        (*seek* '())
        (*rslt* '())
        (*exec* code)
        (*time* 0)
        (*atomic* nil)
        (*rprs-head* (list (make-idle-process)))
        (*rprs-tail* '()))
    (loop
       (cond
        ((null *exec*)
         (return))
        ((= *time* 0)
         (timer-handler))
        (t
         (exec-elem (pop *exec*))
         (unless *atomic*
           (decf *time*)))))
    (pop *rslt*)))
```

Slova na ovládání atomičnosti

```
(defprim :aon
  (setf *atomic* t))

(defprim :aoff
  (setf *atomic* nil))

(defprim :a?
  (push *atomic* *rslt*))
```

Použití:

```
:aon n :val 1 :+ n :set! :aoff
```

Speciální operátor `atomic`

Syntax:

```
(atomic expr1 expr2 ...)
```

Atomicky vyhodnotí výrazy: *expr1*, *expr2*, ...

Vyhodnocení vede na vykonání:

```
(begin expr1 expr2 ...) eval :val :aon :clsub :aoff
```

Změna await

Problém:

```
(atomic
  (await (< 0 n))
  (set! n (- n 1)))
```

Při nesplnění podmínky program nikdy neskončí.

Změna await

Problém:

```
(atomic
  (await (< 0 n))
  (set! n (- n 1)))
```

Při nesplnění podmínky program nikdy neskončí.

Řešení: Na chvíli přerušit atomičnost.

```
:a? :if :else :aoff 100 :rnd :delay :aon :then
```

Použití

Nikdy nevytiskne záporné číslo:

```
(let ((n 1))
  (co
   (atomic
    (await (< 0 n))
    (set! n (- n 1))
    (print n))
   (atomic
    (set! n (- n 1)))
   (atomic
    (set! n (+ n 1)))))
```

Omezení pro `await`

Argumenty v `atomic` se nevyhodnotí atomicky:

```
(let ((n 0))
  (co
   (atomic
    (set! n 1)
    (await (= n 2))
    (print "Není atomické"))
   (set! n 2)))
```

Omezení pro `await`

Argumenty v `atomic` se nevyhodnotí atomicky:

```
(let ((n 0))
  (co
   (atomic
    (set! n 1)
    (await (= n 2))
    (print "Není atomické"))
   (set! n 2)))
```

Pravidlo:
V rámci výrazu `atomic` může být výraz `await` pouze jeho prvním argumentem.
Tedy:

```
(atomic (await condition) body1 ... bodyn)
```

Hranaté závorky

Pro

```
(atomic expr1 expr2 ...)
```

zavedeme syntax:

```
[expr1 expr2 ...]
```

Použití:

```
(let ((n 0))
  (co
   (dotimes (i 100)
     [(set! n (+ n 1))])
   (dotimes (i 100)
     [(set! n (+ n 1))]))
  n)
```

Tvrzení

Speciální operátor `assert`:

```
(assert cond)
```

Syntax:

```
{cond}
```

Použití:

```
(let ((n 0))
  (co
    (begin
     (set! n 1)
     {(= n 1)})
    (set! n 2)))
```