
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

3. Programová logika

verze z 27. února 2025

V této přednášce se seznámíme s *Programovou logikou*, která se též nazývá *Hoareho logika* pojmenovaná podle tvůrce, kterým je britský logik Tony Hoare.

1 Podmínky

V zápisu stavu programu pro zjednodušení nebudeme uvádět následující příkazy procesů. V zápisu uvedeme tedy jen proměnné a jejich hodnoty. Například:

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

Vhodným stavem pro program (nebo výraz) myslíme stav, který obsahuje proměnné potřebné pro vykonání programu (nebo vyhodnocení výrazu) a tyto proměnné mají hodnoty správného typu. Například výše uvedený stav je vhodný pro výraz `(>= x y)`.

Podmínka je jednoduchý výraz, jehož hodnotu chápeme jako logickou. Pro vytváření složených podmínek si zavedeme podmínky *negace*, *konjunkce* a *disjunkce* podmínek:

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

Pokud se podmínka ve stavu programu vyhodnotí na *Pravdu* (hodnotu různou od `nil`), tak říkáme, že je ve stavu **splněna**, nebo, že stav **splňuje** podmínku. Například podmínka `(and (= x 1) (> y 0))` je ve stavu

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

splněna.

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

Zavedeme si výraz ověřující splnění zadané podmínky:

```
(assert condition)
```

Výraz se vyhodnotí následovně.

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

Zápis tvrzení si můžeme zjednodušit tak, že podmínku tvrzení uzavřeme do složených závorek:

```
{condition}
```

Například:

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

2 Program a jeho pravdivost

Program se skládá z neprázdné konečné posloupnosti výrazů. Například:

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

Spuštění programu v nějakém stavu vede k postupnému vyhodnocování jeho výrazů. Programy tedy můžeme spouštět v různých počátečních stavech.

Programy budou vždy začínat a končit tvrzením. Podmínka prvního tvrzení se nazývá **předpoklad** nebo také **prekondice** programu a podmínka posledního tvrzení se nazývá **závěr** nebo také **postkondice** programu.

Například program:

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

má předpoklad `(> x 0)` a závěr `(>= x 0)`.

Neuvedením předpokladu nebo závěru programu myslíme podmínku `t`. Například program:

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

má předpoklad i závěr `t`.

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

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

a nepravdivého programu:

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

Předchozí program je nepravdivý, protože když jej spustíme ve stavu, kde `x` má hodnotu 1, skončí chybou vyvolanou posledním tvrzením.

3 Pravida

Následuje soubor pravidel, jejichž používáním máme zaručeno, že obdržené programy jsou pravdivé.

Pravidlo nic nedělání

Výraz

```
nil
```

chápeme jako atomický příkaz, která nic neudělá.

Pro každou podmínku *condition* platí:

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

Například:

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

Pravidlo přiřazení

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

```
{condition'}
(set! var expr)
{condition}
```

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

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

Předpoklad nahradíme za ekvivalentní jednodušší podmínku:

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

Zesílení předpokladu

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

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

Pokud
```
program
{post1}
```
a *post2* je slabší než *post1*, pak
```
program
{post2}
```
.

Například z

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

plyne

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

Skládání programů

Pokud
```
prog1
{cond}
```
a
```
{cond}
prog2
```
, pak
```
prog1
{cond}
prog2
```
.

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}
```

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

```
{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))}
```

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**. Například:

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

Podmínka tvrzení, která v důkazu předchází příkaz se nazývá **předpoklad příkazu**. Například příkaz `(set! y 2)` z předchozího důkazu má předpoklad `(= x 1)`.

Důkaz *proof* nazveme **důkazem pravdivosti programu** *program*, jestliže dodáním tvrzení do programu *program* obdržíme důkaz *proof*. Například předchozí důkaz je důkazem pravdivosti programu:

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

4 Paralelní pravidla

Zavedeme si pravidla pro souběžné vykonávání programů. 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)}
```

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. Červeně obarvený příkaz nezachovává červeně obarvenou podmínku.

Souběžnost

Pokud
```
{pre1}      {pre2}
prog1    a  prog2
{post1}     {post2}
```
jsou nenarušující se důkazy, pak

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

Předchozí program čitelněji zapíšeme tabulkou:

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

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

Zapsáno tabulkou:

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

Složené atomické operace

Program v hranatých závorkách se vykoná atomicky:

```
[program]
```

Například:

```
                {(= x 0)}
                    |
[(set! x (+ x 1))]  |  [(set! x (+ x 1))]
                    |
                {(= x 2)}
```

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)}
```
.

5 Invarianty

Invariant programu je podmínka, která je splněna v každém jeho stavu. Podmínka je invariantem, právě když je splněna v počátečním stavu a je zachována každým příkazem programu.

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

```
                let ((x 1))
                 {(> x 0)}
                     |
{(> x 0)}            |  {(> x 0)}
[(set! x (+ x 1)]    |  [(set! x (+ x 2))]
{(> x 0))}           |  {(> x 0)}
                     |
                {(> x 0)}
```

6 Implementace

Zavedeme dynamickou proměnnou `*atomic*`, která bude určovat, zda se vykonává atomický kód. Změníme implementaci funkce `execute` tak, že čas do spuštění plánovače snižujeme pouze v případě, že globální proměnná `*atomic*` je *Nepravda*.

```
(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*)))
```

Následující tři slova slouží postupně k zapnutí atomického vykonávání kódu, k jeho vypnutí a k zjištění, zda se kód vykonává atomicky.

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

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

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

Například atomické zvýšení hodnoty proměnné `n` provedeme programem:

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

Speciální operátor `atomic`

Ve Scheme si zavedeme speciální operátor `atomic`, který atomicky vyhodnotí postupně své argumenty. Syntax:

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

Vyhodnocení vede na vykonání:

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

Při použití `await` uvnitř `atomic` narazíme na problém, že při nesplnění podmínky `await` program niky neskončí. Například následující proces neskončí v případě, že `n` není kladné číslo, a to ani když by se souběžně vykonávaný proces chystal `n` na kladné číslo nastavit.

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

Řešením je v případě nesplnění podmínky atomičnost na chvíli přerušit, aby mohl procesor získat jiný proces. Do definice slova `:await` přidáme následující část.

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

Čekání je zařízeno slovem `:rnd` na získání náhodného čísla a sloven `:delay` čekajícím zadanou dobu.

Například následující program 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)))))
```

Přerušení atomického vykonávání operátorem `await` může vést k chybám. Například kód prvního procesu v následujícím programu se nevykoná atomicky.

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

Abychom se podobným chybám vyhnuli, zavedeme si následující pravidlo. V rámci výrazu `atomic` může být výraz `await` pouze jeho prvním argumentem. Tedy:

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

V souladu s teoretickou částí kód uzavřený v hranatých závorkách znamená atomické vykonávání. Přesněji

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

zapisuje výraz

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

Použití:

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

Předchozí výraz bude mít vždy hodnotu 200.

Dříve představený operátor `assert` definujeme jako speciální operátor:

```
(assert cond)
```

Předchozí výraz můžeme pohodlněji zapsat pomocí složených závorek:

```
{cond}
```

Například následující program může skončit chybou v případě, že dojde k souběhu jeho procesů.

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

Otázky a úkoly na cvičení

1. Použijte pravidlo přiřazení k odvození předpokladů následujících programů.

   (a)

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

   (b)

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

   (c)

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

2. Vypracujte důkaz následujícího programu.

   ```
   {(and (= x x0) (= y y0))}
   (set! t x)
   (set! x y)
   (set! y t)
   {(and (= x y0) (= y x0))}
   ```

3. Rozhodněte, zda se následující dva důkazy nenarušují. V případě, že se narušují, upravte důkazy tak, aby se nenarušovaly.

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

4. Vypracujte důkazy následujících programů.

   (a)

```
                {t}
                 |
(set! x z)       | (set! y z)
                 |
 {(and (= x z) (= y z))}
```

   (b)

```
                {t}
                 |
(set! z x)       | (set! z y)
                 |
  {(or (= z x) (= z y))}
```

   (c)

```
                {(= x 0)}
                    |
[(set! x (+ x 1))]  | [(set! x (+ x 2))]
                    |
                {(= x 3)}
```

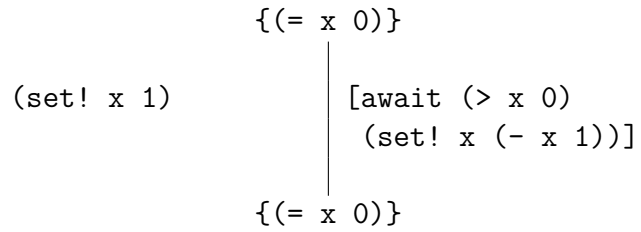
   (d)

```
                {(= x 0)}
                    |
[(set! x (+ x 1))]  | [(set! x (+ x 1))]
                    |
                {(= x 2)}
```

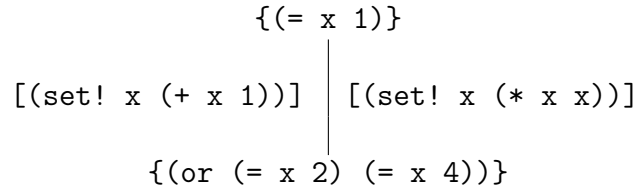
   (e)

```
                {(= x 0)}
                    |
(set! t1 (+ x 1))   | (set! t2 (+ x 1))
(set! x t1)         | (set! x t2)
                    |
          {(or (= x 1) (= x 2))}
```

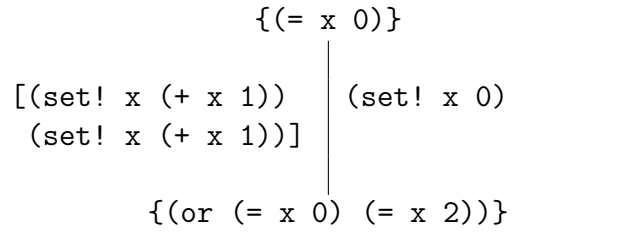
(f)



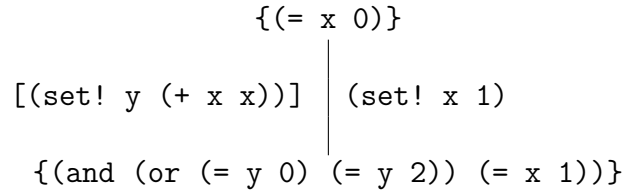
(g)



(h)



(i)



5. Přepište některé z výše uvedených programu do Scheme a spuštěním ověřte jejich pravdivost.

6. Přepište pro zvýšení efektivity makra `and` a `or` na speciální operátory.

7. Dokázali byste přepsat makro `co` na speciální operátor?