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

Aktuální témata z informatiky ◇ poznámky k semináři

Úvod do teorie typů

verze z 9. listopadu 2023

1 Základy

K zápisu matematických objektů používáme *výrazy*. Výrazy jsou různých *typů*. Fakt, že výraz a je typu A symbolicky zapíšeme

$$a : A.$$

Pokud existuje výraz typu A, říkáme, že typ A je *osídlený*, jinak je typ A *prázdný*.

Podle potřeby se na typ A můžeme dívat také jako na *tvrzení*. V takovém případě výrazy typu A považujeme za *důkazy* pravdivosti tvrzení A. Pokud existuje důkaz tvrzení A, říkáme, že A je *pravdivé*, jinak je A *nepravdivé*.

Jednotkový typ značíme $\mathbf{1}$. Existuje jediný výraz $*$, který je typu $\mathbf{1}$, tedy $* : \mathbf{1}$. Typ $\mathbf{1}$ odpovídá *pravdě*. *Nulový typ* značíme $\mathbf{0}$. Žádný výraz není nulového typu. Typ $\mathbf{0}$ odpovídá *nepravdě*.

Vezměme typy A a B, pak můžeme vytvořit typ $A \times B$, který se nazývá *součin* typů A a B. Pokud $a : A$ a $b : B$, pak dvojice $(a, b) : A \times B$. Součin $A \times B$ odpovídá konjunkci tvrzení A a B. Dále můžeme vytvořit typ $A + B$, který se nazývá *součet* typů A a B. Pokud $a : A$, pak *levá injekce* $inl(a) : A + B$. Pokud $b : B$, pak *pravá injekce* $inr(b) : A + B$. Součet $A + B$ odpovídá disjunkci tvrzení A a B.

Pokud máme typy A a B, tak můžeme vytvořit typ $A \to B$ *funkcí* s definičním oborem A a oborem hodnot B. Funkci $f : A \to B$ můžeme *aplikovat* na hodnotu z definičního oboru $a : A$ a obdržíme hodnotu z oboru hodnot označenou $f(a) : B$. Tvrzení $A \to B$ odpovídá implikaci „Jestliže A, pak B.“ Funkční typ je asociativní zprava, tedy $A \to B \to C$ znamená $A \to (B \to C)$. Pokud $f : A \to B \to C$, pak pro $a : A$ je $f(a) : B \to C$ a pro $b : B$ je $f(a)(b) : C$, kde $f(a)(b)$ přirozeně zapíšeme jako $f(a, b)$.

Fakt, že dva výrazy a_1 a a_2 (stejného typu) jsou z *definice rovné*, zapíšeme $a_1 \equiv a_2$.

Funkci můžeme definovat uvedením rovností, které určí, jak se funkce aplikuje. Například funkce identity $id_A : A \to A$ na libovolném typu A:

$$id_A(x) \equiv x$$

nebo funkce $const_{A*} : A \to \mathbf{1}$, která vždy vrací $*$:

$$const_{A*}(x) \equiv *$$

Negaci definujeme jako $\neg A \equiv A \to 0$ a ekvivalenci jako $A \leftrightarrow B \equiv (A \to B) \times (B \to A)$.

Typ *přirozených čísel* označíme $\mathbb{N}$. Položíme $0 : \mathbb{N}$ a $succ : \mathbb{N} \to \mathbb{N}$, kde $succ$ je funkce přiřazující číslu jeho následníka. Zavedeme značení $succ(0) \equiv 1$, $succ(1) \equiv 2$, ...

Rekurzivní funkce můžeme přímo definovat pomocí rovností. Například funkci $double \colon \mathbb{N} \to \mathbb{N}$, která zdvojnásobí zadané číslo, určí následující rovnosti.

$$\begin{aligned} double(0) &\equiv 0 \\ double(succ(n)) &\equiv succ(succ(double(n))) \end{aligned}$$

Funkci sčítání čísel $add \colon \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ můžeme definovat rovnostmi:

$$\begin{aligned} add(0, n) &\equiv n \\ add(succ(m), n) &\equiv succ(add(m, n)) \end{aligned}$$

Výraz $add(m, n)$ přirozeně zapíšeme jako $m + n$.

Můžeme také použít *rekurzivní princip* pro přirozená čísla. Pokud chceme definovat funkci $f \colon \mathbb{N} \to C$, kde C je libovolný typ, stačí uvést $c_0 : C$ a $c_s : \mathbb{N} \to C \to C$. Funkce f pak bude dána rovnostmi:

$$\begin{aligned} f(0) &\equiv c_0 \\ f(succ(n)) &\equiv c_s(n, f(n)) \end{aligned}$$

Například pro $C \equiv \mathbb{N}$, $c_0 \equiv 0$ a $c_s(n, y) \equiv succ(succ(y))$ obdržíme funkci $double$. Pro definici funkce sčítání $add \colon \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ stačí použít rekurzivní princip pro:

$$\begin{aligned} C &\equiv \mathbb{N} \to \mathbb{N} \\ c_0 &: \mathbb{N} \to \mathbb{N} \\ c_0(n) &\equiv n \\ c_s &: \mathbb{N} \to (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N}) \\ c_s(m, g)(n) &\equiv succ(g(n)) \end{aligned}$$

Symbolem $\mathcal{U}$ značíme typ (*malých*) *typů*. Všechny dosud uvedené typy jsou typu $\mathcal{U}$. Například $\mathbf{0}, \mathbf{1}, \mathbb{N}, \mathbb{N} \to \mathbb{N} : \mathcal{U}$. Typ malých typů není malý typ. Neplatí tedy, že $\mathcal{U} : \mathcal{U}$.

Pokud A je typ, $B \colon A \to \mathcal{U}$, pak

$$\prod_{x:X} B(x)$$

je *závislostní typ funkcí*.

Pokud za předpokladu, že $x : A$ je výraz Φ typu $B(x)$, pak můžeme vytvořit funkci $f : \prod_{x:X} B(x)$ danou rovností $f(x) \equiv \Phi$. Uvedenou funkci můžeme zapsat *abstrakcí*:

$$\lambda x.\Phi : \prod_{x:X} B(x)$$

Například funkce identity:

$$id : \prod_{A:\mathcal{U}} A \to A$$

Pro libovolný typ A máme $id(A) : A \to A$. Identita zadaná abstrakcí

$$\lambda A.(\lambda x.x) : \prod_{A:\mathcal{U}} A \to A,$$

kde $\lambda A.(\lambda x.x)$ lze zkrátit vynecháním závorek na $\lambda A.\lambda x.x$.

Rekurzivní princip pro přirozená čísla zachtíme jedinou funkcí:

$$rec_{\mathbb{N}} : \prod_{C:\mathcal{U}} C \to (\mathbb{N} \to C \to C) \to (\mathbb{N} \to C)$$

danou rovnostmi:

$$\begin{aligned} rec_{\mathbb{N}}(C, c_0, c_s, 0) &\equiv c_0 \\ rec_{\mathbb{N}}(C, c_0, c_s, succ(n)) &\equiv c_s(n, rec_{\mathbb{N}}(C, c_0, c_s, n)) \end{aligned}$$

Funkce *double* a *add* můžeme také definovat přímo jako:

$$\begin{aligned} double &\equiv rec_{\mathbb{N}}(\mathbb{N}, 0, \lambda n.\lambda y.succ(succ(y))), \\ add &\equiv rec_{\mathbb{N}}(\mathbb{N} \to \mathbb{N}, \lambda n.n, \lambda m.\lambda g.\lambda n.succ(g(n))). \end{aligned}$$

Pokud A je typ a $a, b : A$, pak

$$a =_A b$$

je *typ rovnosti* a *a* b. Často místo $a =_A b$ píšeme jen $a = b$. Pro každý typ A a $a : A$ je

$$refl_A(a) : a =_A a.$$

Místo $refl_A(a)$ často píšeme jen $refl(a)$.

Tvrzení „Pro každé číslo n platí, že $0 + n$ je rovno n“ je vyjádřeno typem

$$\prod_{n:N} 0 + n = n.$$

Tvrzení je pravdivé, protože

$$\lambda n.refl(n) : \prod_{n:N} 0 + n = n.$$

Princip indukce na přirozených číslech je zachycen funkcí

$$ind_{\mathbb{N}} : \prod_{C:\mathbb{N}\to\mathcal{U}} C(0) \to \left(\prod_{n:\mathbb{N}} C(n) \to C(succ(n))\right) \to \prod_{n:\mathbb{N}} C(n).$$

danou rovnostmi:

$$ind_{\mathbb{N}}(C, c_0, c_s, 0) \equiv c_0$$
$$ind_{\mathbb{N}}(C, c_0, c_s, succ(n)) \equiv c_s(n, ind_{\mathbb{N}}(C, c_0, c_s, n))$$

Princip, že z rovnosti dvou čísel plyne i rovnost jejich následníků, vyjadřuje funkce:

$$ap_{succ} : \prod_{m:\mathbb{N}} \prod_{n:\mathbb{N}} (m = n) \to (succ(m) = succ(n))$$

Pro

$$C : \mathbb{N} \to \mathcal{U}$$
$$C(n) \equiv (n + 0 = n)$$
$$c_0 : (0 + 0 = 0)$$
$$c_0 \equiv refl(0)$$
$$c_s : \prod_{n:\mathbb{N}} (n + 0 = n) \to (succ(n) + 0 = succ(n))$$
$$c_s(n, p) \equiv ap_{succ}(n + 0, n, p)$$

je

$$ind_{\mathbb{N}}(C, c_0, c_s) : \prod_{n:\mathbb{N}} n + 0 = n.$$

Tedy pro každé číslo n platí, že $n + 0$ je rovno n.

2 Typovaný Scheme

Pro použití stačí načíst soubor `typed-scheme.lisp`. Pro příklady se podívejte do `examples.lisp`.

Následující tabulka udává přepis výrazů teorie typů na výrazy typovaného Scheme:

$\mathbf{1}$	`one-type`
$*$	`star`
$\mathbf{0}$	`zero-type`
$A \to B$	`(-> A B)`
$f(x)$	`(f x)`
$\mathbb{N}$	`nat`
0	`zero`
$succ$	`succ`
$\mathcal{U}$	`u`
$\prod_{x:A} B(x)$	`(pi A (lambda (x) B(x)))`
$\lambda x.\Phi$	`(lambda (x) Φ)`
$rec_{\mathbb{N}}$	`rec-nat`
$a =_A b$	`(= A a b)`
$refl_A(a)$	`(refl A a)`
$ind_{\mathbb{N}}$	`ind-nat`
ap_{succ}	`ap-succ`

Součin a součet typů v typovaném Scheme nemáme.

Místo `lambda` můžeme psát λ. Symbol λ v editoru vložíme klávesovou zkratkou Control L.

Výraz

```
(e1 e2 e3 ...)
```

je zkratkou za

```
((e1 e2) e3 ...)
```

Výraz

```
(lambda (a1 a2 ...) expr)
```

je zkratkou za

```
(lambda (a1) (lambda (a2 ...) expr))
```

Následuje popis lispových funkcí pro práci s výrazy typovaného Scheme.

```
(ts-check expr type) => nil

  expr: výraz
  type: typ
```

Provede kontrolu, zda výraz ***expr*** je typu ***type***. Také kontroluje, zda ***type*** je správně utvořený typ. V případě selhání kontroly, vyvolá chybu.

Volání

```
(ts-check a A)
```

ověří, zda $a : A$.

```
(ts-infer-type expr) => type

  expr: výraz
  type: typ
```

Pokusí se odvodit typ výrazu ***expr***. V případě selhání vyvolá chybu.

```
(ts-equal-p expr1 expr2 &optional type) => bool

  expr1: výraz
  expr2: výraz
  type: typ
  bool: logická hodnota
```

Rozhodne, zda výrazy *expr1* a *expr2* jsou z definice rovné. Oba výrazy musí být typu *type*. Provede nejprve kontrolu, zda *type* je typ. V případě selhání vyvolá chybu. Pokud *type* není zadán, pokusí se odvodit typ výrazu *expr1*.

Volání

```
(ts-equal-p a b A)
```

rozhodne, zda $a \equiv b$, kde $a, b : A$.

```
(ts-assert-equal expr1 expr2 &optional type) => nil

  expr1: výraz
  expr2: výraz
  type: typ
  bool: logická hodnota
```

Provede kontrolu, zda výrazy *expr1* a *expr2* jsou z definice rovné. Oba výrazy musí být typu *type*. Také provede kontrolu, zda *type* je typ. V případě selhání vyvolá chybu. Pokud *type* není zadán, pokusí se odvodit typ výrazu *expr1*.

```
(ts-eval expr &optional type) => value

  expr: výraz
  value: hodnota
  type: typ
```

Spočítá hodnotu výrazu *expr*. Napřed provede kontrolu, zda *type* je typ a *expr* je výraz typu *type*. V případě selhání vyvolá chybu. Pokud *type* není zadán, pokusí se jej odvodit z výrazu *expr*.

```
(ts-define name expr &optional type) => nil

  name: proměnná
  expr: výraz
  type: typ
```

Prohlásí výrazy *name* a *expr* za z definice rovné. Nejprve provede kontrolu, zda *type* je opravdu typ a zda *expr* je typu *type*. V případě selhání vyvolá chybu. Pokud typ *type* není zadán, pokusí se jej odvodit z výrazu *expr*.

Volání

```
(ts-define a Φ A)
```

položí $a \equiv \Phi$, kde $\Phi : A$.

V typovaném Scheme pro jednoduchost platí, že $\mathcal{U} : \mathcal{U}$.

3 Úkol

Za použití indukčního principu pro přirozená čísla dokažte vámi zvolené tvrzení. Návrhy dvou tvrzení jsou uvedeny níže. Důkaz ověřte v typovaném Scheme. Důkaz i jeho ověření mi pošlete na e-mail nebo přes MS Teams a stavte se na konzultaci, kde mi důkaz vysvětlíte.

Návrhy tvrzení:

1. $\prod_{i:\mathbb{N}} \prod_{j:\mathbb{N}} \prod_{k:\mathbb{N}} i + (j + k) = (i + j) + k$
2. $\prod_{m:\mathbb{N}} \prod_{n:\mathbb{N}} m + succ(n) = succ(m + n)$