Typový systém

Typový systém

Přiřazování typů termům bylo zatím popsána jen neformálně.

Je možné ho popsat více formálně pomocí tzv. **typového systému**.

Pro ukázku si ukážeme, jak může takový typový systém vypadat.

Poznámka: To, co si zde teď popíšeme, není komletní typový systém Coqu, ale jen jeho fragment, který byl navíc oproti skutečnému typovému systému používanému Coqem poměrně výrazně zjednodušen.

Typový systém

Neprve si připomeňme tu část syntaxe termů, kterou jsme si zatím popsali:

$$
\begin{array}{rcll}
t & ::= & x & \text{— proměnná} \\
 & | & c & \text{— konstanta} \\
 & | & t_1\ t_2 & \text{— aplikace} \\
 & | & \textbf{fun}\ x : A \Rightarrow t_1 & \text{— abstrakce (tj. } \lambda x\!:\!A.\ t_1\text{)} \\
 & | & \textbf{let}\ x : t_1 := t_2\ \textbf{in}\ t_3 & \text{— lokální definice} \\
 & | & \textbf{fix}\ f\ t_1 & \text{— operátor pevného bodu (fixpoint)} \\
 & | & t_1 : A & \text{— explicitní specifikace typu} \\
 & | & \ldots &
\end{array}
$$

Globální environment a lokální kontext

- **Globální environment** E — posloupnost deklarací následujících tří druhů:
  - předpoklady: $(c : T)$
  - definice: $(c := t : T)$
  - definice induktivních typů

- **Lokální kontext** Γ — posloupnost deklarací následujících dvou druhů:
  - předpoklady: $(x : T)$
  - definice: $(x := t : T)$

Příklad: Lokální kontext $[x : A;\ y := u : B;\ z : C]$

Pravidla typového systému

Var: $\dfrac{(x : A) \in \Gamma}{E[\Gamma] \vdash x : A}$ **Const:** $\dfrac{(c : A) \in E}{E[\Gamma] \vdash c : A}$

Lam: $\dfrac{E[\Gamma :: (x : A)] \vdash t_1 : B}{E[\Gamma] \vdash \lambda x{:}A.\, t_1 \ : \ A \to B}$

App: $\dfrac{E[\Gamma] \vdash t_1 : A \to B \qquad E[\Gamma] \vdash t_2 : A}{E[\Gamma] \vdash t_1\ t_2 \ : B}$

Let: $\dfrac{E[\Gamma] \vdash t_1 : A \qquad E[\Gamma :: (x : A)] \vdash t_2 : B}{E[\Gamma] \vdash \textbf{let}\ x : A := \ t_1\ \textbf{in}\ t_2 \ : \ B}$

Fix: $\dfrac{E[\Gamma :: (f : A)] \vdash t_1 : A}{E[\Gamma] \vdash \textbf{fix}\ f\ \ t_1 \ : A}$ **Cast:** $\dfrac{E[\Gamma] \vdash t_1 : A}{E[\Gamma] \vdash (t_1 : A) : A}$

Typy

Typy v Coqu je možné zhruba rozdělit do tří druhů:

- dependent product
- induktivně (a koinduktivně) definované typy
- typová universa

Standardní funkční typ, jehož prvky jsou všechny funkce z A do B (kde A a B jsou typy), označovaný zápisem

$$A \to B$$

je speciálním případem dependent product.

Typy

Typy v Coqu nejsou definovány syntakticky (tj. jako speciální syntaktická kategorie), jak je tomu ve většině variant typovaného λ-kalkulu či teorie typů.

Ze syntaktického hlediska jsou typy reprezentovány stejným druhem termů jako ostatní druhy objektů.

To, jestli daný term t reprezentuje typ, je dáno tím, jestli je možné v daném typovém systému odvodit

$$E[\Gamma] \vdash t : \mathbf{Type}$$

kde **Type** je příkladem tzv. **typového universa**, tj. typu, jehož prvky jsou jiné typy.

Typové universum Prop

Podobně to, zda daný term t reprezentuje výrok (proposition), není definováno syntakticky, ale tím, jestli je možné odvodit v typovém systému

$$E[\Gamma] \vdash t : \mathbf{Prop}$$

kde **Prop** je typové universum, jehož prvky jsou výroky.

Poznámka: Ve skutečnosti jsou i výroky v Coqu chápány jako speciální případ typů.

Typová universa

Typová universa jsou speciální typy, jejichž prvky jsou jiné typy.

- Každý typ patří do některého typového universa.
- Universa jsou dány v Coqu „napevno“ — uživatel je nemá možnost definovat ani měnit.
- Jsou to jediné typy, které existují ještě předtím, než se začnou načítat standardní knihovny.
- Universa tvoří nekonenečnou hierarchii (detaily se zde nebudeme zabývat).

Typová universa

Z uživatelského hlediska jsou důležité tzv. **sorts** — klíčová slova označující jednotlivé druhy typových univers:

- **Prop** — universum, jehož prvky jsou výroky
- **Set** — universum, jehož prvky jsou „obyčejné“ datové typy (běžná data, funkce, . . .)
- **Type** — obecně typové universum obsahující libovolný druh typů

Induktivní typy

Jedním z nejjednoduších příkladů induktivních typů jsou výčtové typy:

```
Inductive day : Set :=
  | Monday : day
  | Tuesday : day
  | Wednesday : day
  | Thursday : day
  | Friday : day
  | Saturday : day
  | Sunday : day.
```

- day je název nově definovaného induktivního typu
- Monday, Tuesday, . . . , Sunday jsou názvy **konstruktorů** tohoto typu

Induktivní typy

```
Definition next_workday (d : day) : day :=
  match d with
  | Monday ⇒ Tuesday
  | Tuesday ⇒ Wednesday
  | Wednesday ⇒ Thursday
  | Thursday ⇒ Friday
  | Friday ⇒ Monday
  | Saturday ⇒ Monday
  | Sunday ⇒ Monday
  end.
```

Induktivní typy

```
Inductive bool : Set :=
  | true : bool
  | false : bool.
```

```
Definition andb (x y : bool) : bool :=
  match x with
  | true ⇒ y
  | false ⇒ false
  end.
```

Induktivní typy

Konstrukce

if t_1 **then** t_2 **else** t_3

je zkrácený způsob zápisu pro

match t_1 **with**
| true $\Rightarrow$ t_2
| false $\Rightarrow$ t_3
end

Poznámka: Ve skutečnosti se dá **if** používat nejen pro testování hodnot typu bool, ale i libovolného jiného induktivního datového typu, který má dva konstruktory.

Induktivní typy

Obecnější příklad induktivního typu:

Inductive T : **Set** :=
$\quad | \; c_1 : A_1 \to A_2 \to T$
$\quad | \; c_2 : B_1 \to B_2 \to B_3 \to T$
$\quad | \; c_3 : T$
$\quad | \; c_4 : C_1 \to T$
$\quad | \; c_5 : D_1 \to D_2 \to T.$

Induktivní typy

Alternativní způsob zápisu:

$$
\begin{aligned}
&\textbf{Inductive } T : \textbf{Set} := \\
&\quad \mid\ c_1\ (a_1 : A_1)\ (a_2 : A_2) : T \\
&\quad \mid\ c_2\ (b_1 : B_1)\ (b_2 : B_2)\ (b_3 : B_3) : T \\
&\quad \mid\ c_3 : T \\
&\quad \mid\ c_4\ (c : C_1) : T \\
&\quad \mid\ c_5\ (d_1 : D_1)\ (d_2 : D_2) : T.
\end{aligned}
$$

Induktivní typy

Alternativní způsob zápisu (název typu u konstruktorů je možno vynechat):

Inductive T : **Set** :=
- $c_1\ (a_1 : A_1)\ (a_2 : A_2)$
- $c_2\ (b_1 : B_1)\ (b_2 : B_2)\ (b_3 : B_3)$
- c_3
- $c_4\ (c : C_1)$
- $c_5\ (d_1 : D_1)\ (d_2 : D_2)$.

Induktivní typy

Reprezentace množiny $\mathsf{nat} \times \mathsf{bool}$:

```
Inductive PairNatBool : Set :=
  | pairNB : nat → bool → PairNatBool.
```

```
pairNB 5 false : PairNatBool
```

Induktivní typy

```
Inductive nat : Set :=
  | O : nat
  | S : nat → nat.
```

```
Fixpoint add (x y : bool) : bool :=
  match x with
  | O ⇒ y
  | S x' ⇒ S (add x' y)
  end.
```

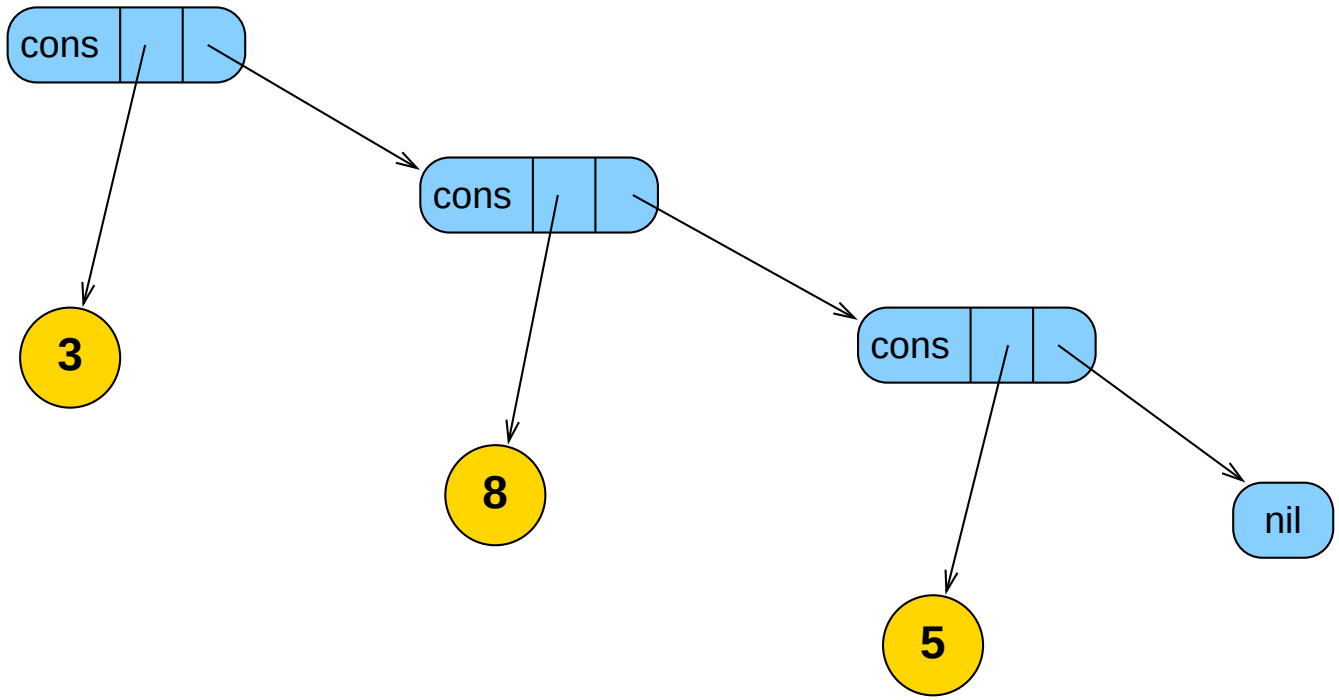
Induktivní typy

Seznam tvořený přirozenými čísly:

```
Inductive natlist : Set :=
  | nil : natlist
  | cons : nat → natlist → natlist.
```

Induktivní typy

Intuitivní představa seznamu tvořeného prvky $[3, 8, 5]$.



cons 3 (cons 8 (cons 5 nil))

Průchod seznamem

Příklad: Jedna možnost, jak spočítat součet prvků v seznamu:

```
Fixpoint sum_rec (ℓ : natlist) : nat :=
  match ℓ with
  | nil ⇒ 0
  | cons x ℓ′ ⇒ x + sum_rec ℓ′
  end.
```

Příklad použití **rekurze**.

Průchod seznamem

Příklad: Alternativní možnost, jak spočítat součet prvků v seznamu:

```
Fixpoint sum_iter_loop (ℓ : natlist) (acc : nat) : nat :=
    match ℓ with
    | nil ⇒ acc
    | cons x ℓ′ ⇒ sum_iter_loop ℓ′ (acc + x)
    end.

Definition sum_iter (ℓ : natlist) : nat := sum_iter_loop ℓ 0.
```

Příklad použití rekurze, která je podobná **iteraci**.
Jedná se o tzv. **tail rekurzi**.

Induktivní typy

Induktivně definované typy se dají použít pro vytváření libovolných (acyklických) **datových struktur**, např. pro různé druhy stromů.

Příklad: Definice datové struktruktury pro reprezentaci aritmetických výrazů obsahujících konstanty typu nat a operace plus, mínus, krát a děleno:

```
Inductive arith_op := Plus | Minus | Times | Div.

Inductive Arith_expr :=
    | AE_Num : nat → Arith_expr
    | AE_BinOp : Arith_expr → arith_op → Arith_expr → Arith_expr.
```

Přepisování termů

Jak už bylo řečeno, termy se vyhodnocují postupným přepisováním podle přepisovacích pravidel:

- α-konverze (alfa)
- β-redukce (beta)
- δ-redukce (delta)
- ζ-redukce (zeta)
- match-redukce
- fix-redukce
- ι-redukce (iota)
- ...

Přepisování budeme zapisovat takto:

$$t \longmapsto t'$$

Přepisování termů

- α-**konverze** (αlfa):

  přejmenování lokálních proměnných (použitých ve **fun**, **let**, **match**, apod.)

- β-redukce (βeta):

  odpovídá volání funkce

  $$(\lambda x{:}A.\, t_1)\ t_2 \longmapsto t_1[t_2/x]$$

- δ-redukce (δelta):

  nahrazení konstanty termem, který je jí přiřazen

  Poznámka: Toto provádí např. taktika unfold.

Přepisování termů

- ζ-redukce (zeta)

  nahrazení lokální proměnné její hodnotou, tj. termem který je jí přiřazen

$$\textbf{let } x := t_1 \textbf{ in } t_2 \quad \longmapsto \quad t_2[t_1/x]$$

- match-redukce:

  v **match** t_1 **with** vybere na základě konstruktoru, kterým byl vytvořen term t_1 příslušnou větev a dosadí hodnoty argumentů, se kterými byl konstruktor vytvořen do příslušných lokálních proměnných reprezentujících parametry této větve

Přepisování termů

- fix-redukce:

  jedno „rozvinutí“ rekurzivně definované funkce

$$\textbf{fix}\ f\ t_1 \longmapsto t_1[\,(\textbf{fix}\ f\ t_1)\,/\,f\,]$$

- ι-redukce (iota)

  match-redukce a fix-redukce společně

Uživatelsky definované notace

Uživatelsky definované notace

Pomocí příkazu **Notation** je možné si definovat vlastní notace.

Příklad:

```
Notation "x && y" := (andb x y).
Notation "x || y" := (orb x y).
```

Místo termu (kde a a b mohou být libovolné termy):

```
andb a b
```

je pak možné psát:

```
a && b
```

Uživatelsky definované notace

- Notace nejsou součástí jádra systému — vnitřně je term vždy uložen bez použití notací.
- Notace slouží pro snadnější a čitelnější zápis termů, ať už při jejich zápisu ve zdrojovém kódu nebo při jejich vypisování systémem (např. při použití příkazů jako **Print** a **Check**, v důkazech v interaktivním dokazovacím módu, apod.)
- V IDE je možné v menu (položka *View/Display notation*) nebo pomocí klávesové zkratky přepínat mezi tím, zda se mají či nemají při vypisování termů používat notace.

Uživatelsky definované notace

Příkaz **Notation** provede dvě věci:

- rozšíří gramatiku jazyka o nové pravidlo, které určuje, jak může vypadat term
- spojí příslušnou nově definovanou notaci s odpovídajícím významem (např. zápis x && y s funkcí andb)

Uživatelsky definované notace

Příkaz **Notation** umožňuje pro nově definovanou notaci specifikovat řadu vlastností, jako např. její **prioritu** a **asociativitu**:

Příklad:

Notation "x + y" := (plus x y) (at level 50, left associativity).
Notation "x >> y" := (rshift x y) (at level 55, left associativity).

Priorita (parametr "at level"):

- priorita je reprezentována číslem v intervalu 0 až 100
  (je možná též speciální priorita 200)
- nižší číslo znamená vyšší prioritu

Příklad:

$a >> b + c$ $\longleftarrow$ bude interpretováno jako $a >> (b + c)$

Uživatelsky definované notace

Asociativita:

- "left associativity" — asociativní směrem doleva

  Příklad:

$a >> b >> c >> d \quad \longleftarrow$ interpretováno jako $((a >> b) >> c) >> d$

- right associativity" — asociativní směrem doprava

  Příklad:

Notation "x # y" := (concat x y) (at level 35, right associativity).

$a \# b \# c \# d \quad \longleftarrow$ interpretováno jako $a \# (b \# (c \# d))$

- no associativity" — žádná asociativita — je třeba explicitně psát závorky

Uživatelsky definované notace

Aktuální podobu gramatiky termů se všemi momentálně definovanými notacemi (včetně informací o prioritě a asociativitě jednotlivých pravidel) je možné vypsat příkazem:

Print Grammar constr.

Poznámka: Slovo "constr" zde slouží jako název neterminálu, který v gramatice Coqu reprezentuje libovolný term.

Místo constr je možné též použít následující názvy neterminálů:

- pattern — syntaxe vzorů používaných při pattern machingu
- vernac — syntaxe příkazů
- tactic — syntaxe uživatelsky definovaných taktik

Uživatelsky definované notace

- Všechny notace na stejné úrovni priority musí používat stejný druh asociativity.
- Dříve definovanou notaci je možné předefinovat, tj. je možné přiřadit dané notaci jiný term, pomocí kterého bude interpretována.

  Priorita a asociativita dané notace však musí zůstat stejná.
- Pomocí příkazu **Reserved Notation** je možné rozšířit gramatiku o danou notaci (spolu s případným specifikováním její priority, asociativity a dalších vlastností) bez toho, aby byl této notaci přiřazen nějaký význam — tj. term, pomocí kterého bude interpretována:

```
Reserved Notation "e ==> n" (at level 90, left associativity).
```

Tento term pak může být přiřazen dané notaci později:

```
Notation "e ==> n" := (eval_expr e n).
```

Uživatelsky definované notace

- Řetězec reprezentující syntaxi dané notace je interpretován jako posloupnost **tokenů** oddělených mezerami.
- Je možné používat (téměř) libovolné znaky.
- Tokeny, jejichž zápis odpovídá syntaxi identifikátorů, jsou interpretovány jako **parametry**, které je možné použít v příslušném termu přiřazeném notaci:

  — při použití dané notace budou tyto parametry nahrazeny skutečnými termy, které se nachází na jejich místě
- Pomocí uzavření do apostrofů (znak ’) je možné vyznačit, že daná posloupnost znaků nemá být interpretována jako parametr:

Notation "’LET’ x ’:=’ E" := (Assign x E) (at level 72).

Uživatelsky definované notace

Příklad: notace pro uspořádanou dvojici

Notation "(x , y)" := (pair x y).

Příklad: notace pro seznamy

Notation "x :: s" := (cons x s) (at level 60, right associativity).
Notation "[ ]" := nil.
Notation "[x ; .. ; y]" := (cons x .. (cons y nil) ..).

Místo cons 1 (cons 2 (cons 3 nil)) je pak možné psát, např.

- 1 :: 2 :: 3 :: [ ]
- [1; 2; 3]

Uživatelsky definované notace

Notace je možné definovat i pro konstrukce, které zahrnují vázané lokální proměnné, např.:

Notation "'sigma' x : A , B" := (sigT (**fun** $x : A \Rightarrow B$))
(at level 200, x name, A at level 200, right associativity).

Zápis

sigma n : nat, $n > 5$

pak bude reprezentovat term

sigT (**fun** n : nat $\Rightarrow$ gt n 5)

Poznámka: x name znamená, že x musí být identifikátor (jméno proměnné), ne libovolný term

Uživatelsky definované notace

Kromě příkazu **Notation** je možné zavádět novou notaci pro infixové operátory také pomocí příkazu **Infix**.

Například příkaz

```
Infix "/\" := and (at level 80, right associativity).
```

provede to samé, co příkaz

```
Notation "x /\ y" := (and x y) (at level 80, right associativity).
```

Uživatelsky definované notace

Pomocí příkazu **Notation** je kromě notací definujících nová pravidla gramatiky též možné definovat **zkratky** (**abbreviations**) pro libovolné termy.

Příklad:

```
Notation NList := (list nat).
Notation reflexive R := (∀x, R x x).
```

Poznámka: Tento typ notací nemodifikuje gramatiku termů.

Uživatelsky definované notace

- Často se hodí, aby používání dané notace bylo omezeno jen na určité úseky zdrojového kódu.
- Zároveň se někdy hodí možnost používat jednu a tu samou notaci v několika různých významech.

  (např. symbol "*" použitý jednou pro násobení na přirozených číslech, jindy jako násobení celých čísel, jindy pro násobení matic, jindy jako symbol pro kartézský součin, apod.)

Pro oba tyto účely slouží tzv. **notation scopes**.

- Každý notation scope má své jméno.
- Je možné si definovat vlastní notation scopes, stejně jako používat notation scopes definované ve standardních knihovnách.

Uživatelsky definované notace

Nový notation scope je možné definovat pomocí příkazu

Declare Scope my_scope.

kde my_scope je jméno tohoto nově definovaného scope.

Při definování nových notací je pak možné uvést scope, ve kterém bude mít daná notace přiřazený uvedený význam:

Notation "$x * y$" := (my_multiplication x y) : my_scope.

Uživatelsky definované notace

Notace definované v určitém scope nebudou používány, dokud nebude příslušný scope „otevřen“ pomocí příkazu

Open Scope my_scope.

- Současně může být otevřeno více různých scopes.
- Pokud se stejná notace (typicky s různými přiřazenými významy) vyskytuje ve více různých otevřených scopes, má vždy vyšší prioritu ten scope, který otevřen později.

Otevřený scope je možné „zavřít“ (tj. pozastavit používání notací z daného scope) pomocí příkazu

Close Scope my_scope.

Uživatelsky definované notace

Příkazy **Open Scope** a **Close Scope** slouží k vymezení delších úseků kódu, ve kterých se mají notace z příslušného scope používat.

Někdy se hodí možnost specifikovat, že notace z daného scope se mají používat jen v rámci určitého termu (který sám může být součástí nějakého většího termu).

K tomu slouží následující notace pro zápis termů používající tzv. **delimiting key** — tj. (typicky krátký) identifikátor označující příslušný scope, který se má použít pro daný term.

Uživatelsky definované notace

- Delimiting key (zde nazvaný ms) pro daný scope my_scope se definuje příkazem

Delimit Scope my_scope **with** ms.

- To, že se má daný scope my_scope, který má přiřazen delimiting key ms, použít pro interpretaci významu daného termu, se zapisuje následujícím způsobem:

$(\cdots)\%\mathrm{ms}$

- Přiřazení daného delimiting key danému scope je možné zrušit pomocí příkazu

Undelimit Scope my_scope.

Uživatelsky definované notace

Daný notation scope je také možné „svázat“ s určitým datovým typem A pomocí příkazu

Bind Scope my_scope **with** A.

- Ve všech termech, které se pak budou nacházet na pozici argumentu funkce, kde je očekávána hodnota uvedeného typu A, bude automaticky použit pro jejich interpretaci uvedený scope my_scope.

Příklad:

Bind Scope nat_scope **with** nat.

Uživatelsky definované notace

- **Print Scopes**. — vypíše všechny aktuálně deklarované notation scopes (včetně všech v nich definovaných notací a informací o případném přiřazeném delimiting key a svázaném typu)
- **Print Scope** my_scope. — vypíše informace o daném konkrétním scope
- **Print Visibility**. — vypíše informace o momentálně otevřených notation scopes a seznam těch notací z nich, které by aktuálně byly použity.
  (Tj. nezobrazuje ty notace, které byly „překryty“ notacemi z nějakého později otevřeného scope.)
- **Print Visibility** my_scope. — ukáže, jak by vypadala situace ohledně viditelnosti jednotlivých notací, pokud by byl nově otevřen scope my_scope.

Moduly

Moduly

Systém **modulů** slouží k lepšímu strukturování kódu a k tomu, aby nedocházelo ke kolizím jmen použitých pro definované konstanty, datové typy, taktiky, apod.

Základní použití modulů vypadá následovně:

Module A.

$\vdots$

Definition $c := \cdots$

$\vdots$

End A.

- Na konstantu c definovanou v rámci modulu A se pak mimo tento modul můžeme odkazovat pomocí $A.c$.

Moduly

Obecně může modul obsahovat libovolnou posloupnost

- konstant
- definic induktivních typů
- dokázaných tvrzení (což je speciální typ konstant)
- modulů (tj. moduly mohou být libovolně zanořeny)
- notací
- nově definovaných taktik
- ...

Jména různých objektů definovaných v rámci jednoho modulu nebudou kolidovat se jmény jiných objektů definovaných mimo tento modul.

Moduly

Příklad:

Module A.

Definition $c := \cdots$

Module B.
Definition $c := \cdots$
End B.

Module C.
Definition $c := \cdots$
End C.

End A.

- Tři různé vzájemně nesouvisející definice konstanty c, na které je možné se odkázat mimo modul A pomocí $A.c$, $A.B.c$ a $A.C.c$.

Moduly

- V rámci daného modulu je možné se odkazovat na objekty definované v rámci tohoto modulu jejich krátkými jmény bez uvedení jména modulu:

  Příklad: V rámci modulu A je možné se odkazovat na konstantu c definovanou v tomto modulu pomocí c.

  Je ovšem možné použít i delší název $A.c$.

- Výše uvedené platí i uvnitř modulu, který je definován uvnitř jiného modulu.

  Příklad: Uvnitř modulu B, který je definován uvnitř modulu A, může zápis c odkazovat k hodnotě $A.c$ definované v modulu A.

  Pokud je však v rámci modulu definována konstanta, která se také jmenuje c, bude c odkazovat na $A.c$ jen v části kódu do místa definice této konstanty. Po této definici bude c odkazovat na $A.B.c$.

Moduly

Pomocí příkazu

Import *A*.

kde *A* je název modulu, je možné do aktuálního modulu **importovat** jména všech objektů z modulu *A*:

- je tím myšleno to, že v rámci aktuálního modulu je pak možné používat krátké názvy všech objektů z modulu *A*.

  Například místo *A.c* je pak možné psát jen *c*.

- Pokud bude v aktuálním modulu definován objekt se stejným jménem jako nějaký objekt z importovaného modulu *A*, bude mít při použití krátkého názvu přednost objekt definovaný v aktuálním modulu.

Moduly

Příklad:

Module A.

Definition $c := \cdots$

End A.

Module B.

Import A.

Definition $d := c$. $\longleftarrow$ c se odkazuje k $A.c$

Definition $c := \cdots$

Definition $e := c$. $\longleftarrow$ c se odkazuje k $B.c$

End B.

Moduly

Příkaz

Export *A*.

se chová podobně jako příkaz **Import**.

Přidává však následující věc:

- Pokud je v modulu *B* použit příkaz **Import** *A* a následně je modul *B* importován do modulu *C*, do modulu *C* jsou importovány pouze názvy z modulu *B*, nikoli z modulu *A*.
- Oproti tomu, pokud je v modulu *B* použit příkaz **Export** *A* a následně je modul *B* importován do modulu *C*, do modulu *C* jsou importovány jak názvy z modulu *B*, tak z modulu *A*.

Moduly

Příklad:

Module A.

Definition $c := \cdots$

End A.

Module B.

Export A.

...

End B.

Module C.

Import B.

Definition $d := c$ $\longleftarrow$ c se odkazuje k $A.c$

End C.

Moduly

Informaci o obsahu daného modulu je možné vypsat pomocí příkazu **Print**, případně **Print Module**:

Print *A*.
Print *A.B*.

nebo

Print Module *A*.
Print Module *A.B*.

Poznámka: Obojí má stejný význam.

Varianta s **Print Module** zdůrazňuje, že se odkazujeme na modul a ne na něco jiného (např. definici konstanty nebo induktivního typu).
Pokud se nejedná o modul, skončí tato varianta chybou.

Moduly

- Výpis informací o modulu příkazem **Print** zahrnuje přehled všech:
  - konstant
  - induktivních typů
  - modulů

  definovaných v rámci daného typu.
- Tento výpis nezahrnuje informace o notacích, taktikách a dalších objektech, které nejsou součástí jádra systému.

  Poznámka: Systém modulů je součástí jádra systému.
- Aktuálně definované moduly jsou součástí environmentu.

Moduly

- Modul je vždy vytvořen až v okamžiku provedení příkazu **End**, který ho ukončuje.
- Uvnitř modulu tedy není možné se odkazovat na tento modul, protože v dané chvíli ještě není vytvořen.
- Unitř modulu je ovšem možné se odkazovat pomocí jména tohoto modulu na objekty definované uvnitř tohoto modulu.

Příklad:

Module A.

Definition $c := \cdots$
Print $A.c$. $\longleftarrow$ toto je v pořádku
Print Module A. $\longleftarrow$ Chyba!

End A.

Print Module A. $\longleftarrow$ toto je v pořádku

Moduly

Informace o všech objektech definovaných v modulech, které dosud nebyly ukončeny příkazem **End** (a uvnitř kterých se tedy momentálně nacházíme), je možné vypsat příkazem

Print All.

Libraries

Speciálním případem modulů jsou moduly, které odpovídají celým souborům, tj.

- zdrojovým souborům s příponou `.v`
- z nich přeloženým binárním souborům s příponou `.vo`

Tyto moduly se označují jako **libraries** (tj. knihovny):

- Jejich definice nezačíná příkazem **Module** a nekončí příkazem **End**.
- Jejich začátkem je začátek souboru a koncem konec souboru.
- Jejich jméno je dáno názvem příslušného souboru.

  Například obsah souboru `Lists.v` (případně z něj přeloženého binárního souboru s názvem `Lists.vo`) odpovídá modulu s názvem Lists.

  (V případě, že soubor nebyl dosud pojmenován, použije se defaultní jméno Top.)

Libraries

- Libraries jsou jediným případem modulů, které samy nejsou obsaženy uvnitř žádného dalšího modulu.
- Všechny ostatní moduly jsou prvkem nějakého dalšího modulu.

Názvy libraries však nemusí být tvořeny jen názvy příslušných souborů, ale může jim předcházet prefix tvořený identifikátory oddělenými tečkami, specifikující plný název dané library v hierarchii všech libraries.

Příklad: Standardní knihovna obsahuje například následující libraries

- Coq.Init.Datatypes
- Coq.Lists.List

Volume 1 (Logical Foundations) série Software Foundations obsahuje například library

- LF.Lists

Libraries

Každá library je sama o sobě modul, prefixy v názvech libraries ale moduly nejsou (jak se můžeme přesvědčit například pomocí příkazu **Print**):

Print Module Coq.Init.Datatypes. ⟵ vypíše obsah modulu
Print Module Coq.Init. ⟵ Chyba!
Print Module Coq. ⟵ Chyba!

Poznámka: Názvy libraries, které jsou sočástí standardních knihoven Coqu, typicky začínají prefixem Coq.

Pro názvy libraries vytvářených uživatelem je tedy vhodné používat raději nějaký jiný prefix než Coq, aby nedocházelo ke konfliktu jmen se standardními knihovnami.

Libraries

Každý definovaný objekt (konstanta, induktivní typ, modul, . . .) má tak určeno jednoznačné **kanonické jméno**, které se skládá z kompletního názvu příslušné library, zanořených modulů a vlastního názvu.

Toto kanonické jméno je možné zjistit pomocí příkazu **About**, který vypíše o daném objektu také různé další informace, např. o jaký druh objektu se jedná (konstanta, induktivní typ, modul, . . .) či případně informaci o jeho typu.

Například

```
About nat.
```

vypíše informaci o tom, že kanonické jméno typu nat je

```
Coq.Init.Datatypes.nat
```

a některé další informace (např. že se jedná o induktivní typ, apod.)

Libraries

Při odkazování se na libraries není třeba používat kanonické jméno, ale libovolný sufix, který příslušnou library dostatečně identifikuje.

Příklad: Místo Coq.Lists.List stačí psát List nebo Lists.List.

```
Require List.
```

tedy provede to samé, co

```
Require Coq.Lists.List.
```

V případě potřeby je možné použít variantu **Require** s **From**, kde je možné specifikovat prefix, kterým musí kanonický název dané library začínat:

```
From Coq Require List.
```

Libraries

V příkazu **Require** je možné uvést navíc **Import** nebo **Export**.

```
Require Import Coq.Lists.List.
```

nebo

```
Require Export Coq.Lists.List.
```

Tyto příkazy provedou to samé, co by provedl příkaz **Require** následovaný příkazem **Import** nebo **Export** pro příslušný modul.

V rámci jednoho příkazu **Require** je možné uvést více libraries (oddělených mezerami) a vše je možné kombinovat se všemi výše uvedenými variantami:

```
From Coq Require Import Bool List Arith.
```

Libraries

- Pracovat je možné jen s těmi libraries, které jsou momentálně načteny v paměti.
- K načtení libovolné library do paměti slouží příkaz **Require**, který načte do paměti obsah přeloženého binárního souboru (s příponou `.vo`).
- To, které libraries jsou momentálně načteny v paměti, je součástí aktuálního environmentu.
- Seznam všech aktuálně načtených libraries je možné vypsat pomocí příkazu:

Print Libraries.

Libraries

- Každá library má v přeložené binární podobě uloženou také informaci, na kterých dalších libraries závisí.

  Příkaz **Require** rekurzivně načte nejprve ty libraries, na kterých daná library závisí. Pokud ony závisí na nějakých dalších libraries, načte nejprve je, atd.

- Každá library je načtena nejvýše jednou — pokud už v paměti načtená je, nenačítá se znova.

- Závislosti mezi libraries nesmí obsahovat cyklus.

- Příkaz **Print Libraries** vypisuje libraries v pořadí, kdy každá z libraries závisí jen na libraries, které jsou v tomto seznamu uvedeny před ní.

Libraries

To, kde konkrétně na disku se při použití příkazu **Require** hledají binární soubory (s příponou `.vo`) obsahující příslušné libraries, je určeno nastavením tzv. **LoadPath**:

- namapování **logických cest** (které odpovídají prefixům v názvech libraries) na **fyzické cesty** (které odpovídají názvům adresářů na disku)

Aktuální LoadPath je možné zjistit pomocí příkazu:

Print LoadPath.

Libraries

Příklad:

- logická cesta Coq.Lists může být mapována například na fyzickou cestu

  `/usr/lib/ocaml/coq/theories/Lists`

- library Coq.Lists.List se tak bude načítat ze souboru

  `/usr/lib/ocaml/coq/theories/Lists/List.vo`

Poznámka: Součástí instalace Coqu bývají typicky i zdrojové kódy knihoven, které bývají umístěny ve stejných adresářích jako přeložené binární soubory.

Zdrojový kód library Coq.Lists.List bude tedy pak pravděpodobně k dispozici v souboru

`/usr/lib/ocaml/coq/theories/Lists/List.v`

Libraries

Konkrétní nastavení LoadPath závisí samozřejmě na konkrétní instalaci Coqu a na tom, jak přesně je nastavena její konfigurace.

Nastavení cest ke standardním knihovnám většinou není důvod měnit.

Někdy ale chceme přidat nějaké vlastní libraries.

Rozšíření LoadPath o nějaké další položky je možné provést například:

- pomocí parametru příkazové řádky, který je předán programu **coqide** nebo **coqc**:

  `-Q` $\langle phys\_path \rangle$ $\langle log\_path \rangle$

  kde $\langle phys\_path \rangle$ je fyzická cesta a $\langle log\_path \rangle$ jí odpovídající logická cesta

- v konfiguračním souboru `_CoqProject`

Libraries

Příklad:

Řekněme, že chceme se soubory

`Basics.v`, `Induction.v`, `Lists.v`, ...

a jim odpovídajícími binárními soubory

`Basics.vo`, `Induction.vo`, `Lists.vo`, ...

které jsou všechny umístěny v aktuálním adresáři, pracovat jako s libraries pojmenovanými

LF.Basics, LF.Induction, LF.Lists, ...

Dosáhneme toho použitím následujícího parametru příkazové řádky:

```
-Q . LF
```

Libraries

Alternativně je možné místo

`-Q` ⟨*phys_path*⟩ ⟨*log_path*⟩

použít

`-R` ⟨*phys_path*⟩ ⟨*log_path*⟩

Význam parametru `-R` je podobný jako v případě parametru `-Q`.

- Parametr `-Q` přidá do LoadPath vždy jen jednu konkrétní uvedenou dvojici.
- Při použití `-R` se navíc rekurzivně projde uvedený adresář a do LoadPath je přidána odpovídající dvojice i pro každý z jeho podadresářů.

Libraries

Při startu systému jsou defaultně zavedeny některé standardní knihovny.

Pro tyto defaultně zavedené libraries není třeba provádět příkaz **Require**.

Konkrétně jde o libraries, jejichž kanonické názvy začínají prefixem Coq.Init, např.:

- Coq.Init.Logic — definice logický spojek, kvantifikátorů a dalších věcí souvisejících s logikou
- Coq.Init.Datatypes — definice základních datových typů jako jsou bool, nat, list, apod.
- Coq.Init.Nat — definice základních operací na přirozených číslech

Poznámka: Defaultně se načítá pouze library Coq.Init.Prelude.
Ta provede příkazy **Require** na ostatní libraries s prefixem Coq.Init.

Libraries

Poznámka: Pokud chceme v **coqide** otevřít a procházet zdrojové soubory těchto defaultně zaváděných libraries (tj. libraries s prefixem Coq.Init, je třeba při spuštění uvést následující parametr příkazové řádky:

```
-nois
```

Ten zajistí, že při spuštění nebudou defaultně načteny žádné knihovny.

(V opačném případě bude otevřený soubor kolidovat s již načtenou library a program skončí chybou.)

Libraries

Některé další užitečné příkazy týkající se libraries a souborů na disku:

- **Locate Library** *A*. — vypíše soubor, ve kterém se nachází library *A*
- **Locate File** "`...`" — vypíše cestu k danému souboru

Příklad:

```
Locate File "Nat.v".
```

Polymorfismus

Parametrický polymorfismus

Příklad:

$$\textbf{Definition}\ \text{twice}\ (A : \textbf{Type})\ (f : A \to A)\ (x : A) : A := f\ (f\ x).$$

Typ funkce twice je

$$\forall A : \textbf{Type}, (A \to A) \to A \to A$$

V syntaxi Coqu je tento typ zapsán takto:

```
forall A : Type, (A -> A) -> A -> A
```

Funkci twice je možné zavolat například takto:

```
twice nat succ 5
```

(Funkce succ zde musí být typu $\text{nat} \to \text{nat}$ a hodnota 5 typu nat.)

Parametrický polymorfismus

Poznámky:

- V teorii typů se varianta typovaného lambda kalkulu, který je rozšířen o tento druh polymorfismu, označuje jako **systém F**.
- V Coqu (na rozdíl od systému F) se nejedná o samostatnou syntaktickou konstrukci, ale (podobně jako v případě „obyčejných“ funkcí) jde o speciální případ **dependentního produktu**.

Implicitní argumenty

U „typových“ argumentů často dává smysl nastavit je jako **implicitní**:

- pomocí speciálních závorek ($\{\cdots\}$ nebo $[\cdots]$) nebo
- pomocí příkazu **Arguments**

Tyto implicitní argumeny se pak při volání funkce neuvádí a Coq sám zkusí doplnit jejich hodnotu pomocí typové inference.

Příklad:

Definition twice $\{A : \mathbf{Type}\}$ $(f : A \to A)$ $(x : A) : A := f\ (f\ x)$.

Funkci twice je pak možné zavolat takto:

twice succ 5

Implicitní argumenty

- Implicitní argumenty nejsou v principu omezeny na „typové“ argumenty (tj. argumenty typu **Type**, **Set**, apod.).

  U argumentů „běžných“ typů ale většinou není možné automaticky dopočítat jejich hodnotu na základě hodnot a typů ostatních argumentů, které jsou explicitně uvedeny.

- Pokud chceme implicitní argumenty uvést explicitně (což je někdy potřeba, protože Coq nemusí být vždy schopen pomocí typové inference doplnit jejich hodnotu), můžeme použít zápis jména funkce se symbolem “@”:

```
@twice nat succ 5
```

Datové typy

- **Typová universa**
  - hierarchie typových univers je dána v jádru systému „napevno“
  - uživatel je nemá možnost definovat ani měnit
  - jsou to jediné typy, které existují ještě předtím, než se začnou načítat standardní knihovny
- **Induktivní datové typy**
  - definují se pomocí klíčových slov **Inductive**, **CoInductive** a **Variant**
  - každá taková definice zavádí nový typ (případně i více typů)
  - mohou se odkazovat k již dříve definovaným datovým typům
- **Dependentní produkt**
  - jeden konkrétní způsob, jak ze dvou existujících typů vytvořit nový typ
  - typ funkcí $A \to B$ z A do B je speciálním případem dependentního produktu

Induktivní typy

Seznam tvořený přirozenými čísly:

```
Inductive natlist : Set :=
  | nil : natlist
  | cons : nat → natlist → natlist.
```

Induktivní typy

Seznam tvořený přirozenými čísly:

Inductive natlist : **Set** :=
| nil : natlist
| cons : nat → natlist → natlist.

Induktivní typy mohou být parametrizovány:

Inductive list $(A : \textbf{Type})$: **Type** :=
| nil : list A
| cons : $A \to$ list $A \to$ list A.

Typy:

list : **Type** → **Type**

nil : $\forall A : \textbf{Type}$, list A
cons : $\forall A : \textbf{Type}$, $A \to$ list $A \to$ list A

Induktivní typy

Návratový typ, který by mohl být použit k reprezentaci výsledku výpočtu, který může skončit chybou:

$$\begin{aligned}&\textbf{Inductive}\ \mathsf{Result}\ (A : \textbf{Type}) : \textbf{Type} := \\ &\quad |\ \mathsf{OK} : A \to \mathsf{Result}\ A \\ &\quad |\ \mathsf{Err} : \mathsf{ErrorInfo} \to \mathsf{Result}\ A.\end{aligned}$$

Induktivní typy

Reprezentace regulárních výrazů:

Inductive RegExp $(S : \mathbf{Set}) : \mathbf{Set} :=$
- $\mid$ Empty : RegExp S
- $\mid$ Eps : RegExp S
- $\mid$ Symb : $S \to$ RegExp S
- $\mid$ Dot : RegExp $S \to$ RegExp $S \to$ RegExp S
- $\mid$ Plus : RegExp $S \to$ RegExp $S \to$ RegExp S
- $\mid$ Star : RegExp $S \to$ RegExp S.

Odpovídá standardní induktivní definici regulárních výrazů nad abecedou Σ:

$$
\begin{aligned}
e ::= \;& \emptyset \\
\mid\; & \varepsilon \\
\mid\; & a \qquad (\text{kde } a \in \Sigma) \\
\mid\; & e_1 \cdot e_2 \\
\mid\; & e_1 + e_2 \\
\mid\; & e_1^*
\end{aligned}
$$

Induktivní typy

Některé standardní typy:

- Kartézský součin ($A \times B$):

  Inductive prod $(A\ B : \mathbf{Type}) : \mathbf{Type}\ :=$
  $\quad \mid$ pair $: A \to B \to$ prod $A\ B$.

- Disjunktní sjednocení ($A + B$):

  Inductive sum $(A\ B : \mathbf{Type}) : \mathbf{Type}\ :=$
  $\quad \mid$ inl $: A \to$ sum $A\ B$
  $\quad \mid$ inr $: B \to$ sum $A\ B$.

Induktivní typy

- Volitelná hodnota typu A (tj. taková, která nemusí existovat):

  Inductive option $(A : \mathbf{Type}) : \mathbf{Type} :=$
  $\quad | \ \mathrm{Some} : A \to \mathrm{option}\ A$
  $\quad | \ \mathrm{None} : \mathrm{option}\ A.$

- Jednoprvkový typ:

  Inductive unit : **Set** :=
  $\quad | \ \mathrm{tt} : \mathrm{unit}.$

- Prázdný typ:

  Inductive Empty_set : **Set** := .

Induktivní typy

Může být najednou definováno několik induktivních typů, které odkazují na sebe navzájem:

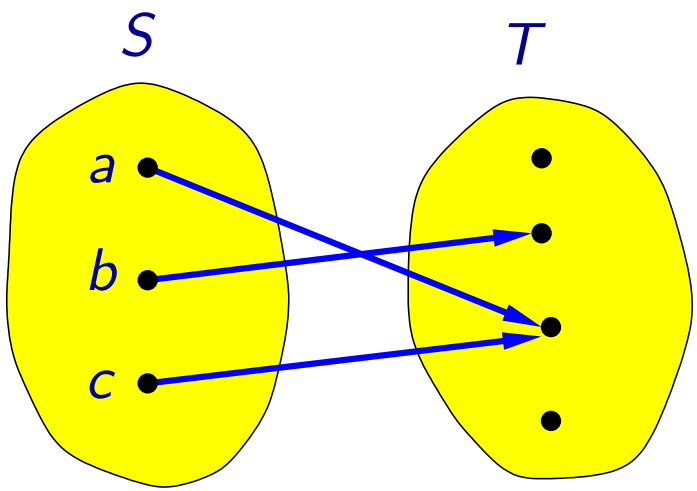
```
Inductive tree (A B : Type) : Type :=
  | leaf : B → tree A B
  | node : A → forest A B → tree A B

with forest (A B : Type) : Type :=
  | one_tree : tree A B → forest A B
  | cons : tree A B → forest A B → forest A B.
```

Dependentní typy

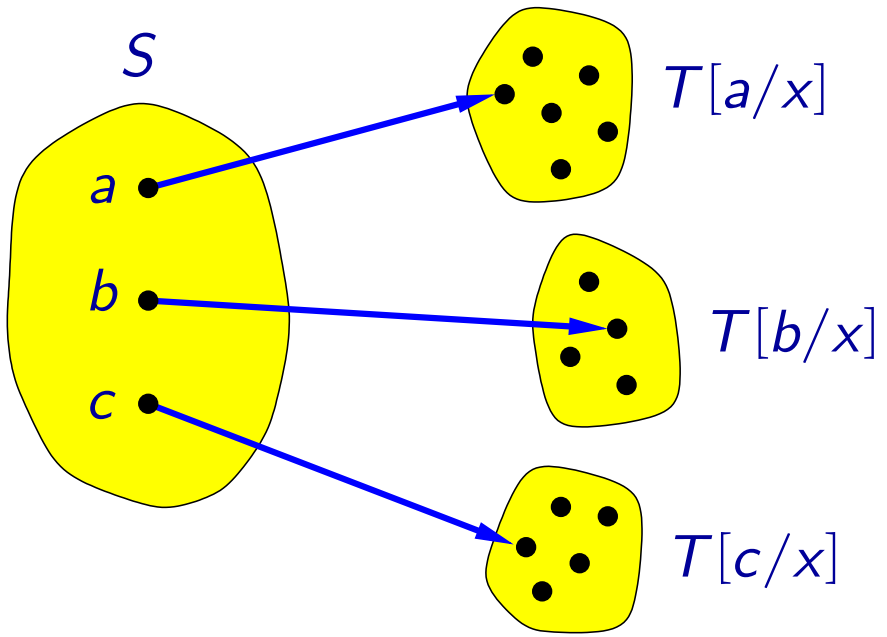
Standardní funkce:

$$f : S \to T$$



Dependentní funkce:

$$f : \forall x : S, T$$



výraz T může obsahovat volnou proměnnou x

Dependentní typy

Poznámka: V literatuře se většinou místo zápisu $f : \forall x : S, T$ používá zápis $f : \prod x : S, T$.

Poznámka: Název produkt pochází pravděpodobně z následující analogie:

- Na množinu všech funkcí typu $S \to T$, kde např. $S = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}\}$, bychom se alternativně mohli dívat jako na kartézský součin:
$$T \times T \times T \times T \times T$$
- Na množinu všech funkcí typu $\mathbb{N} \to T$ bychom se alternativně mohli dívat jako na „nekonečný kartézský součin“:
$$T \times T \times T \times T \times T \times \cdots$$
- Na množinu všech funkcí typu $\prod x : S, T$, kde např. $S = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}\}$, bychom se alternativně mohli dívat jako na kartézský součin:
$$T[\mathtt{a}/x] \times T[\mathtt{b}/x] \times T[\mathtt{c}/x] \times T[\mathtt{d}/x] \times T[\mathtt{e}/x]$$

Dependentní typy

„Obyčejné“ funkční typy jsou speciálním případem dependentních produktů:

Funkční typ $S \to T$ je to samé jako dependentní produkt

$$\forall x : S, T$$

kde se x nevyskytuje jako volná proměnná v T.

V Coqu jsou tedy „obyčejné“ funkční typy reprezentovány jako pouhá notační zkratka pro příslušný dependentní produkt:

Notation "$A \to B$" := $(\forall \_ : A, B)$: type_scope

Dependentní typy

Speciálním případem dependentního produktu je **parametrický polymorfismus** — případ, kde typ S je typové universum:

$$\forall x : S, T$$

Například

$$\forall X : \mathbf{Type}, X$$
$$\forall X : \mathbf{Type}, X \to X \to X$$
$$\forall X : \mathbf{Type}, (\mathsf{nat} \to X \to X) \to \mathsf{nat} \to \mathsf{list}\ X$$

Dependentní typy

Dependentní typy je možné používat i jako typy **konstruktorů** v induktivních typech:

Inductive A : **Type** :=
| c : $\forall X$: **Type**, $(X \to X) \to A$.

Dependentní typy

Jako induktivní typ je možné vyjádřit například tzv. **dependent sum**, v literatuře většinou označovaný

$$\sum x : A, B$$

kde se x může vyskytovat jako volná proměnná v B:

Inductive dep_sum : **Type** :=
| dep_pair : $\forall x : A,\ B \rightarrow$ dep_sum.

Reprezentuje typ uspořádaných dvojic $\langle a, b \rangle$, kde a je typu A a b je typu $B[a/x]$.

Logika

Typové universum Prop

Jak už bylo uvedeno dříve, **výroky** jsou Coqu reprezentovány jako prvky speciálního typového universa **Prop**:

- z formálního hlediska jsou výroky speciálním případem typů
- prvky těchto typů jsou **důkazy**

  Tj. pokud platí

  $$E[\Gamma] \vdash A : \mathbf{Prop}$$

  znamená to, že term A reprezentuje výrok.

  Pokud pak navíc platí

  $$E[\Gamma] \vdash t : A$$

  znamená to, že t je **důkazem** tvrzení A (resp. že t je term, který reprezentuje tento důkaz).

- platí **Prop** : **Type**

Typové universum Prop

Předpokládejme, že M je nějaký typ (např. z universa **Set**):

- unární predikát P reprezentující nějakou vlastnost prvků z M bude typu

$$M \to \mathbf{Prop}$$

  Pokud bude $E[\Gamma] \vdash x : M$, bude $E[\Gamma] \vdash P\ x : \mathbf{Prop}$.

- binární predikát R reprezentující nějakou binární relaci na prvcích z M bude typu

$$M \to M \to \mathbf{Prop}$$

Typové universum Prop

- binární logické spojky jako konjunkce, disjunkce, implikace, atd. budou typu

$$\textbf{Prop} \to \textbf{Prop} \to \textbf{Prop}$$

- univerzální a existenční kvantifikátor nad prvky typu M budou typu

$$(M \to \textbf{Prop}) \to \textbf{Prop}$$

Poznámka: Věci jako logické spojky a kvantifikátory nejsou „natvrdo“ zabudovány v jazyce Coqu ani v jeho formálním typovém systému.

Tyto věci jsou nadefinovány v jazyce Coqu v rámci standardních knihoven.

Pokud uživatel chce, může si nadefinovat vlastní alternativy těchto definic.

Logické spojky a kvantifikátory

$\rightarrow$	->		implikace
$\wedge$	/\	and	konjunkce
$\vee$	\/	or	disjunkce
$\leftrightarrow$	<->	iff	ekvivalence
$\neg$	~	neg	negace
$\bot$		False	
$\top$		True	
$\forall$		forall	univerzální kvantifikátor
$\exists$		exists	existenční kvantifikátor

Taktiky

- **Implikace**:

  Taktika intro h.

$$\frac{\cdots, h : A \vdash B}{\cdots \vdash A \to B}$$

  Taktika apply h.

$$\frac{\cdots, h : A \to B \vdash A}{\cdots, h : A \to B \vdash B}$$

$$\frac{\cdots, h : A \to B \to C \vdash A \qquad \cdots, h : A \to B \to C \vdash B}{\cdots, h : A \to B \to C \vdash C}$$

Taktiky

- **Konjunkce**:

  Taktika split.

$$\frac{\cdots \vdash A \qquad \cdots \vdash B}{\cdots \vdash A \wedge B}$$

  Taktika destruct h as $[h_1\ h_2]$.

$$\frac{\cdots, h_1 : A,\ h_2 : B \vdash C}{\cdots, h : A \wedge B \vdash C}$$

Taktiky

- **Disjunkce**:

  Taktika left.

$$\frac{\cdots \vdash A}{\cdots \vdash A \vee B}$$

  Taktika right.

$$\frac{\cdots \vdash B}{\cdots \vdash A \vee B}$$

  Taktika destruct h as $[h_1 \mid h_2]$.

$$\frac{\cdots, h_1 : A \vdash C \qquad \cdots, h_2 : B \vdash C}{\cdots, h : A \vee B \vdash C}$$

Taktiky

- **Ekvivalence**:

  Taktika split.

$$\frac{\cdots \vdash A \to B \qquad \cdots \vdash B \to A}{\cdots \vdash A \leftrightarrow B}$$

  Taktika destruct h as $[h_1\ h_2]$.

$$\frac{\cdots, h_1 : A \to B,\ h_2 : B \to A \vdash C}{\cdots, h : A \leftrightarrow B \vdash C}$$

Taktiky

- **Negace**:

  Taktika intro h.

$$\frac{\cdots, h : A \vdash \bot}{\cdots \vdash \neg A}$$

  Taktika apply h.

$$\frac{\cdots, h : \neg A \vdash A}{\cdots, h : \neg A \vdash \bot}$$

$$\frac{\cdots, h : A \to \neg B \vdash A \qquad \cdots, h : A \to \neg B \vdash B}{\cdots, h : A \to \neg B \vdash \bot}$$

Taktiky

- **False**:

  Taktika exfalso *h*.

  $$\frac{\ldots \vdash \bot}{\ldots \vdash A}$$

  Taktika contradition.
  Taktika contradict *h*.
  Taktika destruct *h*.

  $$\frac{}{\ldots, h : \bot \vdash A}$$

- **True**:

  Taktika apply I.
  Taktika exact I.
  Taktika constructor.

  $$\frac{}{\ldots \vdash \top}$$

Taktiky

- **Univerzální kvantifikátor**:

  Taktika intro x.

$$\frac{\cdots, x : A \vdash B}{\cdots \vdash \forall x : A, B}$$

  Taktika apply h.

$$\frac{}{\cdots, h : \forall x : A, B \vdash B[t/x]}$$

  Taktika generalize t.

$$\frac{\cdots \vdash \forall x : A, B}{\cdots \vdash B[t/x]}$$

- **Existenční kvantifikátor**:

  Taktika exists t.

$$\frac{\cdots \vdash B[t/x]}{\cdots \vdash \exists x : A, B}$$

  Taktika destruct h as $[x\ h_1]$.

$$\frac{\cdots, x : A, h_1 : B \vdash C}{\cdots, h : \exists x : A, B \vdash C}$$

Taktiky

- **Rovnost**:

  Taktika reflexivity.

$$\frac{}{\cdots \vdash t_1 = t_1}$$

  Taktika symmetry.

$$\frac{\cdots \vdash t_2 = t_1}{\cdots \vdash t_1 = t_2}$$

  Taktika transitivity t.

$$\frac{\cdots \vdash t_1 = t \qquad \cdots \vdash t = t_2}{\cdots \vdash t_1 = t_2}$$

Taktiky

- **Důkaz pomocného tvrzení**:

  Taktika assert $(h : A)$.

$$\frac{\cdots \vdash A \qquad \cdots, h : A \vdash B}{\cdots \vdash B}$$

  Taktika enough $(h : A)$.

$$\frac{\cdots, h : A \vdash B \qquad \cdots \vdash A}{\cdots \vdash B}$$

  Taktika cut A.

$$\frac{\cdots \vdash A \to B \qquad \cdots \vdash A}{\cdots \vdash B}$$