Interaktivní a automatizované dokazování korektnosti programů

Zdeněk Sawa

Katedra informatiky, FEI,
Vysoká škola báňská – Technická univerzita Ostrava
17. listopadu 2172/15, Ostrava-Poruba 708 00
Česká republika

17. března 2023

Vyučující

Jméno: doc. Ing. Zdeněk Sawa, Ph.D.

E-mail: zdenek.sawa@vsb.cz

Místnost: EA413

Web: https://www.cs.vsb.cz/sawa/iadkp

Na těchto stránkách najdete:

- informace o předmětu
- odkazy na studijní materiály
- slidy z přednášek
- aktuální informace

Požadavky

- **Zápočet** (35 bodů):
  - Řešení zadaných problémů (15 bodů)
    - V průběhu semestru budou pravidelně zadávány problémy k řešení jako domácí úlohy.
    - Minimum pro získání zápočtu je 7 bodů.
  - Důkaz korektnosti zadaného programu (20 bodů)
    - Na konci semestru bude zadán jako domácí úkol problém vytvořit v Coqu důkaz korektnosti celého jednoho programu (typicky implementace nějakého známého algoritmu).
    - Minimum pro získání zápočtu je 10 bodů.
  - Celkem je nutné získat minimálně 20 bodů.
- **Zkouška** (65 bodů)
  - Zkouška bude probíhat ústní formou.
  - Celkově je třeba získat minimálně 30 bodů.

Motivace — korektnost počítačových programů

Jednou z požadovaných vlastností počítačových programů je **korektnost** — tj. že program neobsahuje chyby.

Kupříkladu po implementaci nějakého algoritmu typicky požadujeme, aby pro každý možný vstup z množiny všech přípustných vstupů splňoval následující:

- po nějakém konečném počtu kroku **se zastaví**
- **vrátí správný výsledek** (tj. výsledek odpovídající specifikaci problému, který má daný algoritmus řešit)

Poznámka: Předpokládáme, že je nějak přesně specifikováno:

- co jsou přípustné **vstupy**
- co má program počítat (tj. jak mají vypadat **výstupy** a co mají splňovat)

Motivace — korektnost počítačových programů

Příklady toho, co může být na daném programu špatně:

- skončí chybou za běhu (např. neinicializovaný pointer, přístup mimo meze pole, dělení nulou, . . .)
- vrací chybný výsledek
- běží do nekonečna a nikdy nevrátí žádný výsledek

Abychom prokázali, že program obsahuje chybu, stačí najít **alespoň jeden** příklad vstupu, na kterém se program chová „špatně“.

Korektní program je takový, který pracuje správně na **všech** možných vstupech.

Testování a jeho omezení

Testování:

- vyzkoušení chování programu na konkrétních vstupech
- nezbytná součást vývoje software
- slouží k odhalení chyb — **může prokázat přítomnost chyby**
- typicky nemůže prokázat, že se program chová korektně pro všechny vstupy:
  - množina přípustných vstupů bývá většinou **nekonečná** nebo přinejmenším velmi velká
  - není tak možné vyzkoušet chování programu na všech možných vstupech
  - chyba se může projevovat jen na některých specifických vstupech, které nemusí být v testovacích datech zastoupeny

Pojem „důkaz“

Možností, jak ukázat, že daný program se chová správně pro jakýkoli možný vstup a při jakémkoli výpočtu, je podat **matematický důkaz**, že tomu tak je.

Obecně se v matematice či logice pod pojmem **důkaz** rozumí druh zdůvodnění, které:

- se skládá z mnoha menších logických kroků argumentace, kde každý jednotlivý krok je dostatečně jednoduchý na to, aby nebylo pochyb o jeho správnosti
- pracuje s dostatečně přesně a detailně definovanými pojmy

Dokazování korektnosti programů

Pokud chceme vytvářet matematické důkazy korektnosti programů, je třeba:

- formalizovat **sémantiku** programovacího jazyka, ve kterém jsou jsou programy reprezentovány:
  - tj. popsat formálně význam všech jednotlivých konstrukcí jazyka tak, aby se dalo s programy v daném jazyce, stavy systému, běhy algoritmu, atd. pracovat jako s matematickými objekty
- použít nějakou vhodnou logiku, která by ve formulích umožnila hovořit o hodnotách proměnných programu, obsahu paměti, změnách těchto hodnot provedených jednotlivými instrukcemi, apod.

  Často používané logiky pro tento účel jsou:
  - **Hoarova logika**
  - **separační logika**

Formální a neformální důkazy

Důkazy (nejen důkazy korektnosti programů, ale obecně jakékoli matematické důkazy) mohou být popisovány s různou mírou formálnosti a detailnosti:

- **formální důkazy**:
  - veškerá tvrzení jsou formalizována jako formule nějakého druhu (formální) logiky
  - jednotlivé kroky důkazu se řídí přesně danými pravidly daného logického systému
  - tato pravidla popisují (jednoduché) manipulace s formulemi na syntaktické úrovni
  - pro ověření správnosti důkazu není třeba formulím vůbec rozumět; je ovšem třeba spousta trpělivosti a pečlivosti

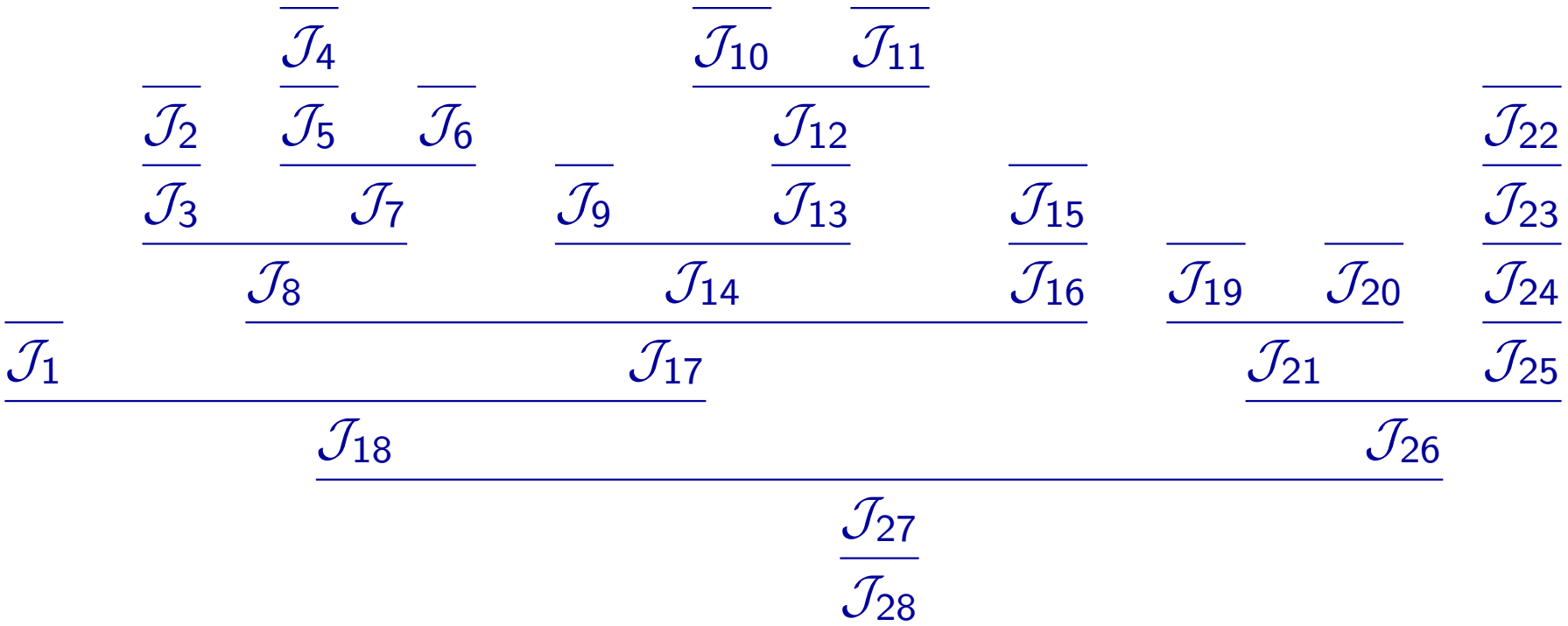
Formální a neformální důkazy

- **běžné matematické důkazy** (např. v odborných matematických knihách či článcích):
  - kombinují použití matematického formalismu (např. různých druhů rovnic a matematických výrazů) s popisem v přirozené řeči
  - daly by se (s trochou úsilí) relativně přímočaře přepsat do podoby formálního důkazu

- **neformální sdělení hlavní myšlenky**:
  - typicky například při vzájemné komunikaci mezi odborníky v dané oblasti
  - obsahuje jen klíčové myšlenky daného důkazu
  - spoustu detailů a rutinních kroků by bylo třeba doplnit
  - spíše nápověda, jak postupovat při vytváření daného důkazu, než důkaz v pravém slova smyslu

Formální důkazy

Formální důkazy mohou být často reprezentovány ve formě **stromu**:



Alternativně mohou být též důkazy reprezentovány jako **sekvence**.

Formální důkazy

Ať už se na důkaz díváme jako na strom nebo jako na sekvenci, v obou případech se důkaz skládá z dílčích kroků, které vznikají aplikací nějakých daných **odvozovacích pravidel** (používá se též pojem **dedukční** či **inferenční pravidla**), která typicky bývají tvaru

$$\frac{\mathcal{J}_1 \quad \mathcal{J}_2 \quad \cdots \quad \mathcal{J}_k}{\mathcal{J}}$$

kde:

- $\mathcal{J}_1, \mathcal{J}_2, \cdots, \mathcal{J}_k$ jsou **premisy**
- $\mathcal{J}$ je **závěr**

Příklad:

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

Rozdíl mezi nalezením důkazu a ověřením důkazu

Obecně pro matematické důkazy platí, že:

- Důkaz daného tvrzení může být velmi těžké najít.

  Nalezení důkazu může někdy vyžadovat nějaký chytrý nápad, který nemusí být zřejmý.
- Pokud už ale důkaz máme, mělo by být mnohem jednodušší ověřit jeho správnost, než na něj přijít.

Poznámka: Čím je důkaz formálnější a detailnější, tím víc by mělo být jeho ověření jednodušší a mechaničtější (zároveň ale i pracnější).

Použití počítače pro vytváření a kontrolu důkazů

Podobně jako počítačové programy, i důkazy mohou obsahovat **chyby**, například:

- použití nějakého nekorektního logického kroku v argumentaci
- opomenutí nějakého případu, který může nastat
- přehlédnutí nějakého symbolu, přepsání se, apod.

Důležitá myšlenka:

Použít počítačový program, který:

- umožní interaktivně vytvářet zápis důkazů
- kontroluje správnost jednotlivých kroků důkazu
- některé rutinní kroky umožní provést automaticky

Interactive Theorem Proving

Druh počítačového programu, který slouží k interaktivnímu vytváření plně formálních důkazů uživatelem s tím, že

- se program stará o kontrolu veškerých formalit ohledně vytvářeného důkazu
- částečně pomáhá uživateli s některými kroky důkazu, které je možné automatizovat

se označuje jako **proof assistant** nebo též **interactive theorem prover**.

Vytváření formálního důkazu nějakého tvrzení v takovém nástroji se označuje jako **interactive theorem proving**.

Automated Theorem Proving

Oproti tomu pojem **automated theorem proving** označuje použití nástroje, který:

- dostane jako vstup jen tvrzení, které chceme dokázat
- plně automaticky najde buď důkaz daného tvrzení nebo zdůvodnění toho, že dané tvrzení neplatí

Pro algoritmy, které něco takového umožňují, se používá pojem **decision procedures**.

Kombinace Interactive a Automated Theorem Proving

Automated theorem proving naráží na limity toho, co je algoritmicky rozhodnutelné, případně na příliš vysokou výpočetní složitost.

- pokud je jazyk použité logiky dostatečně bohatý na to, aby v něm bylo možné vyjadřovat obecná matematická tvrzení, nemůžeme očekávat, že by hledání důkazů mohlo probíhat plně automaticky
- bude fungovat jen pro určité speciální případy

Pro vytváření složitých důkazů je vhodné oba přístupy, tj. interactive theorem proving a automated theorem proving, kombinovat.

Nástroj Coq

Typickým příkladem nástroje typu **proof assistant**, který v sobě kombinuje interaktivní a automatizované dokazování, je nástroj **Coq**:

- umožňuje interaktivní vytváření formálně ověřených důkazů libovolných matematických tvrzení
- definice, věty, důkazy, atd. se zapisují v podobě zdrojového kódu (trochu připomíná programování v programovacím jazyce)
- Coq kontroluje správnost jednotlivých kroků důkazu, do určité míry umožňuje některé kroky automatizovat
- obsahuje v sobě řadu decision procedures pro dokazování různých specifických druhů formulí
- zároveň je Coq i programovacím jazykem — vysokoúrovňový staticky typovaný funkcionální programovací jazyk

Nástroj Coq

Stránky projektu: http://coq.inria.fr/

- vyvíjeno jako open source software
- aktuálně ve verzi 8.16.1
- možno stáhnout zdrojové kódy i přeložené verze pro řadu platforem (Linux, Win, MacOS, . . .)
- rozsáhlá dokumentace
  http://coq.inria.fr/documentation
- možnost stažení zdrojových kódů mnoha projektů vyvinutých s použitím Coqu

Příklady jiných nástrojů typu proof assistant

Příklady jiných podobných nástrojů — historických i současných:

Automath	(60. léta)
LCF	(70. léta)
Mizar	(70. léta)
Nqthm	
ACL2	http://www.cs.utexas.edu/users/moore/acl2/
Isabelle/HOL	http://isabelle.in.tum.de/
PVS	http://pvs.csl.sri.com/
Twelf	http://www.twelf.org/

Coq — příklady projektů

Příklady velkých projektů vytvořených v Coqu:

- důkaz věty o 4 barvách (four color theorem)

  https://github.com/coq-community/fourcolor

- Feit-Thompson theorem (věta z teorie grup)

  https://github.com/math-comp/odd-order

- CompCert — formálně ověřený kompilátor jazyka C

  https://compcert.org/

- VST (Verified Software Toolchain) — sada knihoven pro práci se separační logikou a verifikaci programů;
  součástí je Verifiable C — sada knihoven pro verifikaci programů v jazyce C

  https://vst.cs.princeton.edu/

Coq — zdroje

Knihy věnované Coqu:

- *Software Foundations*, by Benjamin C. Pierce et al. (nejnovější verze z roku 2022)
  https://softwarefoundations.cis.upenn.edu/
- *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*, by Yves Bertot and Pierre Casteran. Springer-Verlag, 2004.
- *Certified Programming with Dependent Types*, by Adam Chlipala. MIT Press, 2013.
  http://adam.chlipala.net/cpdt/

Série knih Software Foundations

Software Foundations — volně dostupné na adrese
https://softwarefoundations.cis.upenn.edu/

- Volume 1: Logical Foundations
- Volume 2: Programming Language Foundations
- Volume 3: Verified Functional Algorithms
- Volume 4: QuickChick: Property-Based Testing in Coq
- Volume 5: Verifiable C
- Volume 6: Separation Logic Foundations

Poznámka: Knihy z této série jsou zajímavé tím, že jejich text je napsán v podobě zdrojových kódů pro Coq, ze kterého jsou vygenerovány HTML stránky.
Je to zamýšleno tak, že čtenář si čte text z HTML stránek a zároveň provádí příkazy v příslušném zdrojovém souboru přímo v Coqu.

Série knih Software Foundations

V rámci předmětu *Interaktivní a automatizované dokazování korektnosti programů* budeme vycházet z této série knih (především z Volume 1 a 2).

Budeme se primárně zabývat následujícími tématy:

- interaktivní a automatizované dokazování — použití nástroje Coq
- sémantika programovacích jazyků a teorie typů
- dokazování korektnosti programů za použití Hoarovy logiky a separační logiky

Probíraná témata budou částečně souviset s následujícími oblastmi:

- funkcionální programování
- logika

Základy použití nástroje Coq

Coq — spuštění programu

Coq je tvořen několika programy:

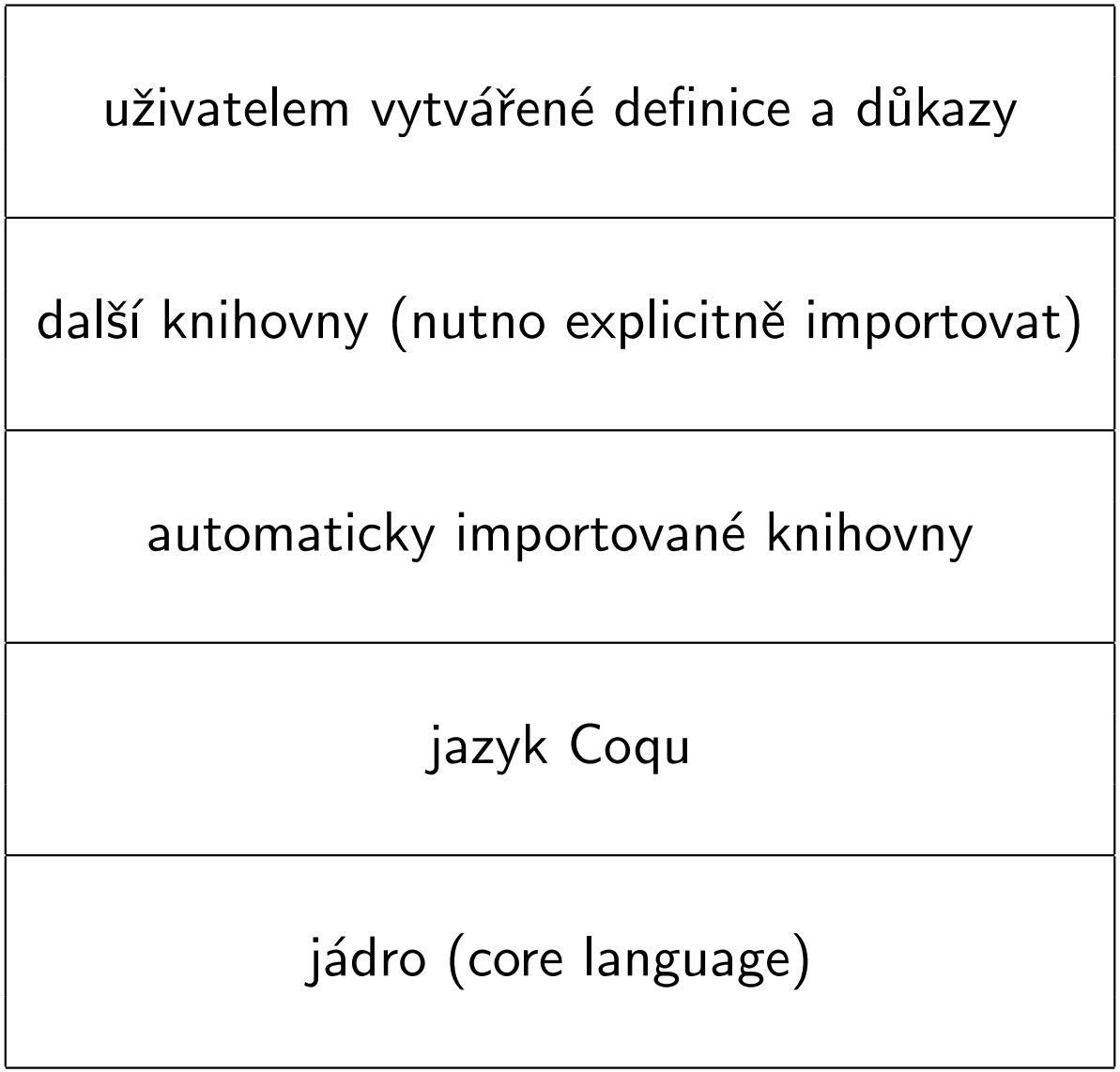
- **coqide** — interaktivní práce s Coqem s použitím vývojového prostředí s GUI ⟵ asi nejběžnější způsob práce
- **coqtop** — interaktivní práce s Coqem prostřednictvím příkazové řádky
- **coqc** — překladač, který zdrojové kódy v jazyce Coqu (s příponou `.v`) kompiluje do binární podoby (soubory s příponou `.vo`)
  — tyto soubory je pak možné importovat jako knihovny
- **coqchk** — minimalistický systém pro kontrolu korektnosti důkazů v binárních souborech (s příponou `.vo`)
- některé další pomocné nástroje (**coqdep**, **coqdoc**, . . .)

Coq — architektura

Základem Coqu je funkcionální programovací jazyk:

- staticky typovaný
- syntaxí se podobá jiným staticky typovaným funkcionálním jazykům (ML, Standard ML, OCaml, Haskell, . . .)
- velmi mocný typový systém — obecnější než např. typový systém Haskellu
- funkce napsané v tomto jazyce je možné přímo volat a spouštět
- všechny konstrukce, se kterými Coq pracuje (např. formule reprezentující dokazovaná tvrzení, definice pojmů, důkazy, . . .) jsou reprezentovány jako **termy** tohoto jazyka

Coq — architektura



Jádro

Jádro (core language):

- minimalistická podmnožina jazyka
- velmi přesně a formálně definovaná syntaxe, typový systém a sémantika přepisování termů
- žádný syntactic sugar
- vše musí být explicitně uvedeno
- vše se překládá do tohoto minimálního jádra
- způsob, jak jsou vnitřně uloženy termy
- poměrně jednoduchý a přímočarý algoritmus typové kontroly
- vše se kontroluje prostřednictvím tohoto jádra

Poznámka: Případná chyba v implementaci tohoto jádra by mohla ohrozit korektnost vytvářených důkazů.

Jazyk Coqu

Jazyk Coqu:

- složitý a bohatý jazyk
- spousta různých druhů „syntactic sugar“
- mnohé věci umí doplňovat a generovat automaticky
- složitý algoritmus **typové inference**
- snaha ulehčit uživateli práci
- snaha o stručný a přehledný zápis
- často složité a „chytré“ algoritmy — ne vždy je zaručeno úspěšné nalezení řešení

Poznámka: Případná chyba v implementaci těchto dalších komplikovaných částí neohrozí korektnost vytvářených důkazů, protože se vše překládá do minimalistického jazyka jádra a kontroluje prostřednictvím jádra systému.

Coq — knihovny

Věci jako:

- logika (logické spojky, kvantifikátory, . . .)
- základní matematické pojmy (jako třeba kartézský součin, relace, . . .)
- různé druhy čísel (přirozená, celá, racionální, reálná, . . .)
- různé druhy matematických struktur (např. různé druhy uspořádání)
- různé základní datové typy hodící se programování (např. seznamy, různé druhy stromů, . . .)
- věci týkající se teorie množin
- . . .

nejsou zabudovány v jazyce Coqu napevno, ale jsou definovány v **knihovnách**, které jsou implementovány v jazyce Coqu.

Coq — knihovny

Určitá minimální část těchto knihoven je defaultně zavedena automaticky při spuštění Coqu.

Zahrnuje to zejména:

- základní věci týkající se logiky (logické spojky, kvantifikátory, . . .)
- základní datové typy (uspořádané dvojice, seznamy, . . .)
- přirozená čísla

Ostatní knihovny je třeba explicitně importovat (příkazem **Require**), pokud je chceme používat.

Extrahování kódu

Coq umožňuje **extrahovat** (tj. automaticky přeložit) kód napsaný v jazyce Coqu do několika programovacích jazyků.

V současné době je podporována extrakce do následujících jazyků:

- OCaml
- Haskell
- Scheme

Poznámka: Součástí extrahovaného kódu jsou jen ty části kódu, které odpovídají popisu algoritmů a dají se tedy přeložit do běžných funkcionálních programovacích jazyků.

Části kódu týkající se logiky a dokazování extrahovány nejsou.

Syntaxe Coqu — mezery, komentáře, identifikátory

- **white-space znaky** — mezery, tabelátory, konce řádků

  jsou ignorovány, slouží k oddělení jednotlivých tokenů

- **komentáře** — uzavřeny mezi znaky `(*` a `*)`

  Příklad: `(* toto je komentář *)`

  Komentáře mohou být víceřádkové a mohou být v sobě zanořeny:

  `(*` ... `(*` ... `*)` ... `*)`

  Komentáře se chovají stejně jako white-space znaky.

- **identifikátory** — libovolná neprázdná posloupnost tvořená písmeny, číslicemi a znaky `_` (podtržítko) a `'` (apostrof), nezačínající číslicí ani apostrofem

  Příklady: `x1` `my_func` `f'` `TransitiveRelation`

Termy

Většina objektů, se kterými Coq pracuje, je reprezentována pomocí **termů**.

Zahrnuje to např. následující druhy objektů:

- matematické objekty — funkce, relace, čísla, atd.
- datové typy
- funkce realizující určitý algoritmus
- výroky (propositions) — reprezentují např. dokazovaná tvrzení
- důkazy
- matematické struktury
- ...

Termy

Syntaxe termů (resp. její část) — vychází z λ-kalkulu:

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

Poznámka: Symboly jako '$\Rightarrow$' jsou ve zdrojovém kódu ve skutečnosti zapisovány jako '`=>`'.

Typy

Každý term je určitého **typu**.

Formálně můžeme zapsat, že term t je typu A, zápisem:

$$t : A$$

U každého termu t, se kterým se pracuje, je vždy kontrolován jeho typ pomocí **typové kontroly**.

Příklady: Ve standardních knihovnách jsou například definovány typy:

- nat — přirozená čísla (tj. hodnoty $0, 1, 2, \ldots$)
- bool — booleovské hodnoty (tj. true a false)

$$5 : \text{nat} \qquad \text{false} : \text{bool} \qquad 8 + 3 : \text{nat}$$

Environment

Environment si můžeme představit jako databázi či datovou strukturu, která obsahuje informace o všech aktuálně definovaných objektech:

- definice konstant (zahrnuje jako spec. případ i definice funkcí)
- definice typů
- dokázaná tvrzení
- definované notace
- ...

Environment

Uživatel komunikuje s Coqem sekvencí příkazů, kdy jedním příkazem může například:

- definovat novou konstantu (tj. rozšířit aktuální environment o definici nové konstanty)
- definovat nový typ
- zjistit hodnotu dané konstanty (tj. jaký term je jí přiřazen v aktuálním environmentu a jaký je jeho typ)
- zjistit jaký je typ daného termu v aktuálním environmentu
- ...

Při interaktivní práci se tyto příkazy typicky provádí po jednom.

V IDE je barevně odlišena část kódu tvořená příkazy, které již byly provedeny.

Coq jako programovací jazyk

Pokud bychom chtěli používat Coq jako funkcionální programovací jazyk, bude program vypadat jako posloupnost definic konstant (což bude typicky zahrnovat především definice funkcí) a induktivně definovaných typů:

Inductive $T_1 := \cdots$.

Definition $c_1 := t_1$.

Definition $c_2 := t_2$.

$\vdots$

Definice konstant

Příkaz

Definition $c := t$.

provede následující:

- naparsuje term t
- provede typovou kontrolu termu t v aktuálním environmentu E
- přiřadí tento term t (spolu s informací o jeho typu) jako hodnotu nové konstanty c, kterou přidá do environmentu E

Příklad:

Definition my_const := 5.

Definice konstant

Při použití definice je možné explicitně specifikovat typ:

Definition $c : A := t$.

Pokud term t není typu A, systém ohlásí chybu.

Příklad:

Definition my_const : bool := 5. $\longleftarrow$ chyba!

Příkaz se neprovede a dostaneme následující chybové hlášení:

The term "5" has type "nat" while it is expected to have type "bool".

Dotazy na environment

- **Print** c.
  Vypíše definici konstanty c v aktuálním environmentu (tj. term, který je jí přiřazen) a odpovídající typ.

- **Check** t.
  Vypíše typ termu t v aktuálním environmentu.

Poznámka: V IDE není třeba psát tyto dotazovací příkazy do zdrojového kódu.
Je možné také použít následující dvě možnosti:

- Použít speciální panel na dotazy (zobrazí se klávesou F1).
- Umístit kurzor na daný symbol ve zdrojovém kódu a použít příkaz z menu nebo klávesovou zkratku.

Dotazy na environment

Příklad:

Definition c := 5.
Definition d := c + 3.

Dotazy na environment:

Print c.	— vypíše: c = 5 : nat
Print d.	— vypíše: d = c + 3 : nat
Check 5.	— vypíše: 5 : nat
Check c.	— vypíše: c : nat
Check d.	— vypíše: d : nat
Check (c + d) ∗ 12.	— vypíše: c + d : nat

Poznámka: Všimněte si, že příkaz **Definition** přiřadí konstantě d jako její hodnotu term c + 3, nikoli např. hodnotu 8.

Definice funkcí

Definice funkcí jsou speciálním případem výše uvedeného použití příkazu **Definition**.

Příklad:

$$\textbf{Definition}\ \text{my\_func} := \textbf{fun}\ (x : \text{nat})\ (y : \text{nat}) \Rightarrow x * x + y * y.$$

Pro lepší čitelnost a stručnější zápis se ovšem častěji používá následující alternativní způsob zápisu definice funkce (který je ekvivalentní s výše uvedeným):

$$\textbf{Definition}\ \text{my\_func}\ (x : \text{nat})\ (y : \text{nat}) := x * x + y * y.$$

V tomto způsobu zápisu je možné specifikovat i typ návratové hodnoty:

$$\textbf{Definition}\ \text{my\_func}\ (x : \text{nat})\ (y : \text{nat}) : \text{nat} := x * x + y * y.$$

Rekurzivní funkce

Při použití příkazu

Definition $c := t$.

je možné se v termu t odkazovat na libovolné dříve definované konstanty (tj. na libovolné konstanty v aktuálním environmentu E).

Speciálně v případě definice funkce je možné se odkazovat na dříve definované konstanty.

Při použití příkazu **Definition** ovšem není možné se v těle funkce odkazovat na ni samotnou, tj. umožnit rekurzivní volání.

Rekurzivní funkce

Příklad: Pro následující kód dostaneme chybové hlášení, že hodnota factorial nebyla nalezena v aktuálním environmentu:

```
Definition factorial (n : nat) : nat :=
    match n with
    | 0 ⇒ 1
    | S n' ⇒ n ∗ factorial n'        ⟵ chyba!
    end.
```

V případě definic rekurzivních funkcí je třeba použít místo příkazu **Definition** příkaz **Fixpoint**:

```
Fixpoint factorial (n : nat) : nat :=
    match n with
    | 0 ⇒ 1
    | S n' ⇒ n ∗ factorial n'
    end.
```

Rekurzivní funkce

Poznámka: Ve skutečnosti je **Fixpoint** de facto také jen speciálním případem příkazu **Definition**.

- Ve vnitřní reprezentaci Coqu je funkce, kterou definujeme pomocí příkazu **Fixpoint**, reprezentována pomocí termu, který používá tzv. **operátor pevného bodu fix**.

  (Jak se můžeme přesvědčit například pomocí příkazu **Print**.)

Příklad: Výše uvedená definice funkce factorial by mohla být zapsána také takto:

```
Definition factorial :=
    fix f (n : nat) : nat :=
        match n with
        | 0 ⇒ 1
        | S n' ⇒ n * f n'
        end.
```

Vyhodnocení hodnoty termu

Příkaz

Compute t.

provede vyhodnocení hodnoty termu t, tj. spustí výpočet, jehož výsledkem je term reprezentující výslednou hodnotu.

Příklad:

Compute factorial 5. — vypíše: 120 : nat

Vyhodnocení hodnoty termu

Poznámka: Příkaz **Compute** je speciálním případem obecnějšího příkazu **Eval**, který umožňuje různými způsoby vyhodnocovat hodnotu termu.

Následující příkaz vyhodnotí term t použití strategie vyhodnocování str:

> **Eval** str **in** t.

Příklady takových strategií vyhodnocování jsou např.:

- cbv — call by value
- lazy — lazy
- hnf — head normal form
- compute — efektivnější způsob vyhodnocení
- vm_compute — ještě efektivnější způsob vyhodnocení zahrnující překlad do bytecodu vykonávaného virtuálním strojem implementovaným v C

Poznámka: příkaz **Compute** je zkratka pro **Eval** vm_compute **in**

Vyhodnocení hodnoty termu

Termy se vyhodnocují postupným přepisováním podle přepisovacích pravidel (tato pravidla budou podrobněji popsána později):

- β-redukce
- δ-redukce
- ζ-redukce
- ι-redukce
- ...

Poznámka: U strategie vyhodnocení je možné přesněji definovat, která z těchto pravidel se mají či naopak nemají používat.

Toto přepisování je možné používat nejen ke spočítání nějaké výsledné hodnoty (např. volání funkce s nějakou hodnotou argumentu), ale také pro **úpravy** termů — např. zjednodušení nějakého výrazu.

Vyhodnocení hodnoty termu

Efektivnější strategie jako compute a vm_compute ve skutečnosti nepracují přímo pomocí přepisování termů, ale

- vnitřně transformují term nejprve do bytekódu, který je efektivnější na vykonávání
- provádí instrukce tohoto bytekódu
- po skončení výpočtu převedou výslednou hodnotu opět do podoby termu

Rychlost vykonávání kódu pomocí vm_compute je plně srovnatelná s rychlostí běžně používaných funkcionálních jazyků (jako je třeba Haskell).

Strategie, které pracují přímo s přepisováním termu jsou výrazně pomalejší. Jsou určeny spíše na úpravy termů při dokazování tvrzení.
(Umožňují ale zase jemnější kontrolu nad tím, co se má a nemá přepisovat, a v jakém pořadí.)

Vyhodnocení hodnoty termu

Vyhodnocený term je též možné přiřadit jako hodnotu konstanty:

```
Definition c := Eval compute in t.
```

Příklad: Následující příkaz přiřadí konstantě q term 120:

```
Definition q := Eval compute in factorial 5.
```

Provede tedy to samé, co by provedl příkaz

```
Definition q := 120.
```

Důkazy tvrzení

V Coqu jsou i důkazy tvrzení reprezentovány termy a je s nimi možno pracovat podobně jako s ostatními objekty.

Termy reprezentující důkazy jsou však často značně velké a nepřehledné.

Přímá práce s těmito termy je sice v principu možná, ale velmi uživatelsky nepřívětivá.

V tomto uživatelsky nepřívětivém „manuálním“ módu by důkaz tvrzení A explicitním napsáním termu t (který reprezentuje daný důkaz) vypadal takto:

Theorem $thm_1 : A$.
Proof t.

Důkazy tvrzení

Standardně se tvrzení dokazují následujícím způsobem:

Theorem $thm_1 : A$.
Proof.
...
Qed.

Důkaz se vytváří postupně pomocí tzv. **taktik**:

- taktiky jsou speciální příkazy vytvářející postupně term reprezentující daný důkaz
- uživatel tento term nevidí
- uživateli se zobrazuje aktuální stav jako jeden či více **cílů** (**goals**)
- důkaz se buduje odzadu

Důkazy tvrzení

Cíl (**goal**) se skládá z:

- **lokálního kontextu** — což je posloupnost lokálních předpokladů a definic, např:
  - $x : \mathsf{nat}$ — proměnná x je typu nat
  - $h_1 : x > 1$ — platí předpodklad $x > 1$
    h_1 je jméno, kterým se můžeme na tento předpoklad odkazovat
  - $y := x * x : \mathsf{nat}$ — lokální proměnná y má přiřazenu hodnotu danou termem $x * x$ a je typu nat
- **závěru** — typicky je to formule reprezentující tvrzení, které je třeba dokázat

Pro označení cíle, který se skládá z lokálního kontextu Γ a cíle A budeme používat zápis:

$$\Gamma \vdash A$$

Důkazy tvrzení

- Typicky se začíná s cílem, jehož lokální kontext je prázdný a závěr je tvrzení, které chceme dokázat.
- použití taktiky buď cíl vyřeší nebo vytvoří jeden nebo více podcílů, které je třeba dokázat:

$$\frac{\Gamma_1 \vdash B_1 \qquad \Gamma_2 \vdash B_2 \qquad \cdots \qquad \Gamma_k \vdash B_k}{\Gamma \vdash A}$$

- Není to tak, že by taktiky přesně odpovídaly jednotlivým logickým krokům důkazu.

  Taktika obecně může ve stromě důkazu vybudouvat libovolný počet takových kroků.

Důkazy tvrzení

- Důkaz musí být ukončen příkazem

Qed.

Ten zkontroluje, že již nezbývá žádný nevyřešený podcíl, a také provede celkovou kontrolu vytvořeného stromu důkazu (který je reprezentován příslušným termem).

Důkazy tvrzení

Následující příkazy se z hlediska Coqu chovají úplně stejně jako příkaz **Theorem**:

- **Lemma**
- **Fact**
- **Remark**
- **Corollary**
- **Proposition**
- **Property**

Použití různých označení místo **Theorem** slouží čistě pro uživatele, aby měl možnost neformálně vyznačit, jakou roli hraje dané tvrzení v celkovém důkazu, např.:

- **Lemma** — pomocné tvrzení, dílčí krok důkazu
- **Corollary** — bezprostřední jednoduchý důsledek nějaké předtím dokázané věty nebo lemmatu

Axiomy a předpoklady

Kromě definovaných pojmů a dokazovaných tvrzení může zdrojový kód obsahovat také:

- **Axiomy** — nedokazovaná tvrzení, o kterých prostě předpokládáme, že platí:

Axiom $ax_1 : A$.

ax_1 je zda název axiomu a A formule reprezentující znění tohoto axiomu.

- **Nedefinované konstanty** — konstanty, u kterých je deklarován pouze typ, ale není jim přiřazen žádný term

  Reprezentuje tedy nějakou blíže nespesifikovanou hodnotu daného typu.

Parameter $c : A$.

Axiomy a předpoklady

- Axiomy a nedefinované konstanty se v typických důkazech nevyskytují buď vůbec nebo jen vzácně.
- Mohou být ale užitečné při vytváření dosud nehotových důkazů, kdy mohou zastupovat některé části, které zatím nejsou hotové a budou případně doplněny později.

Axiomy a předpoklady

Nehotový důkaz je také možné místo **Qed** ukončit pomocí příkazu **Admitted**:

Theorem $thm_1 : A$.
Proof.
...
Admitted.

Výše uvedené použití příkazu **Admitted** zahodí dosud vytvořenou část stromu důkazu a udělá to samé, co by udělal příkaz

Axiom $thm_1 : A$.

tj. přidá do environmentu axiom thm_1.

Příkaz Abort

Další možností ukončení důkazu je příkaz **Abort**.

Theorem $thm_1 : A$.
Proof.
...
Abort.

Stejně jako **Admitted** zahodí dosud vytvořenou část stromu důkazu, ale na rozdíl od něj do environmentu nic nepřidá.

Ukončení důkazu příkazem **Abort** tedy označuje neúspěšný pokus o důkaz, který nebude nadále k ničemu používán.

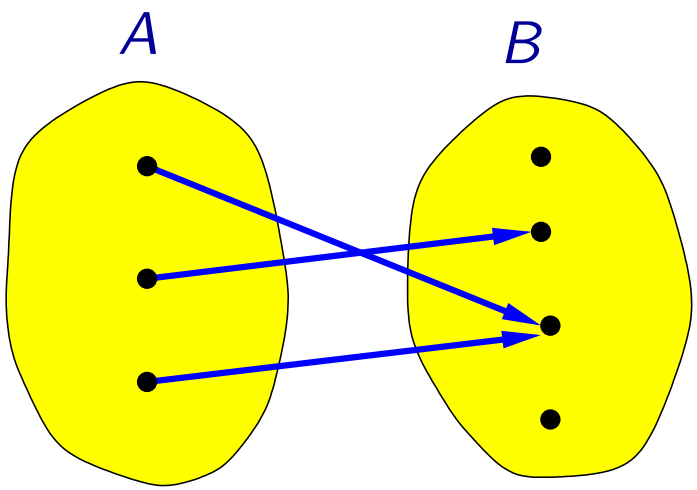
V hotových důkazech by se tedy typicky neměl vyskytovat.

Základní druhy termů

Funkční typy

Pokud A a B jsou typy, tak zápis $A \to B$ označuje typ všech funkcí z A do B.

$$f : A \to B$$



Příklad: $\text{is\_prime} : \text{nat} \to \text{bool}$

Poznámka: Místo symbolu '$\to$' se ve zdrojovém kódu píše '`->`'.

Aplikace funkce

Předpokládejme, že A a B jsou nějaké libovolné typy.

Předpokládejme dále, že

- f je funkce z A do B, tj. $f : A \to B$
- a je hodnota (např. konstanta nebo proměnná) typu A, tj. $a : A$

Následující zápis označuje **aplikaci** funkce f na argument a:

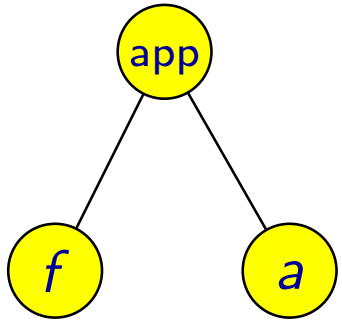
$$f\ a$$

Term $f\ a$ je typu B, tj. $f\ a : B$

Poznámka: V programovacích jazycích jako je C nebo Java by odpovídající konstrukce (tj. volání funkce f s hodnotou argumentu a) byla zapsána jako výraz $f(a)$.

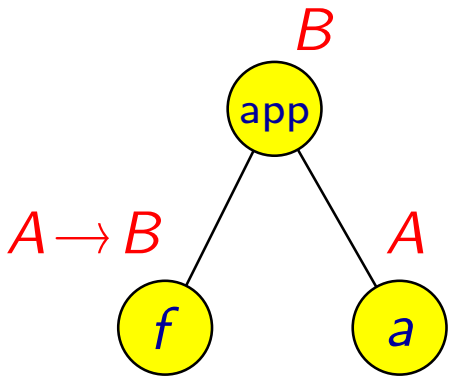
Aplikace funkce

Term $f\ a$ může být reprezentován následujícím **abstraktním syntaktickým stromem**:



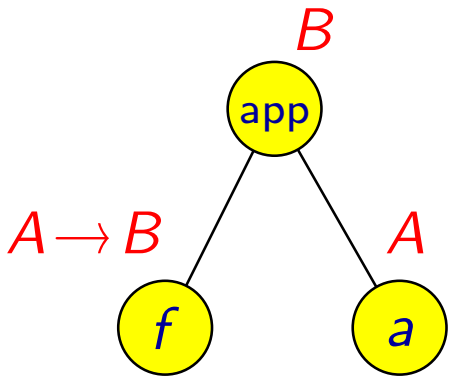
Aplikace funkce

Term $f\ a$ může být reprezentován následujícím **abstraktním syntaktickým stromem**:



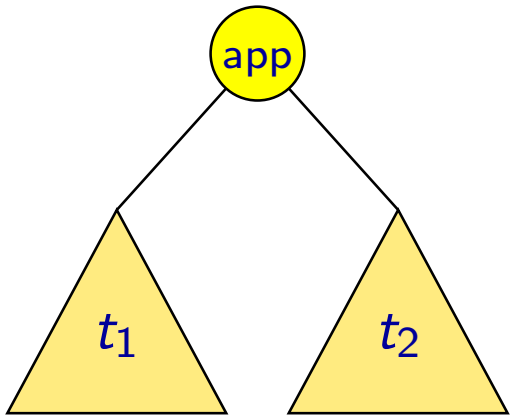
Aplikace funkce

Term $f\ a$ může být reprezentován následujícím **abstraktním syntaktickým stromem**:



Aplikace funkce je operace, která může být aplikována na libovolné dva termy t_1 a t_2 odpovídajících typů:

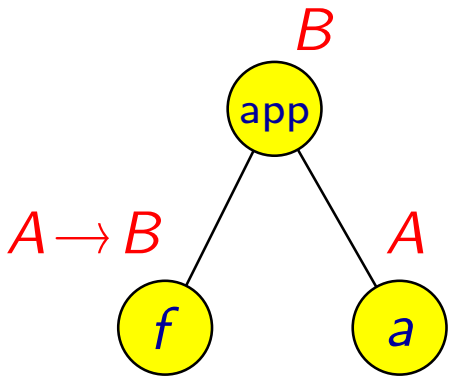
Term $t_1\ t_2$:



$t_1 : S \to T$
$t_2 : S$
$t_1\ t_2 : T$

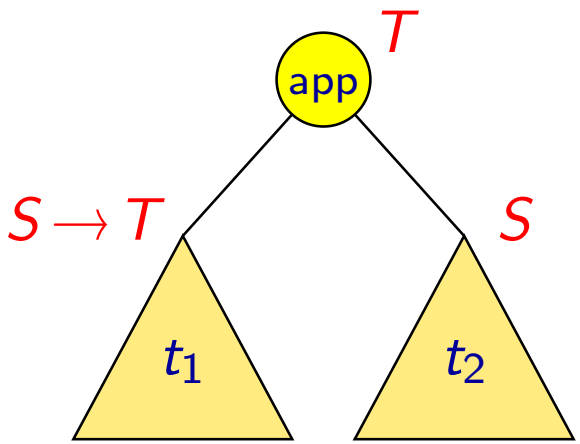
Aplikace funkce

Term $f\ a$ může být reprezentován následujícím **abstraktním syntaktickým stromem**:



Aplikace funkce je operace, která může být aplikována na libovolné dva termy t_1 a t_2 odpovídajících typů:

Term $t_1\ t_2$:

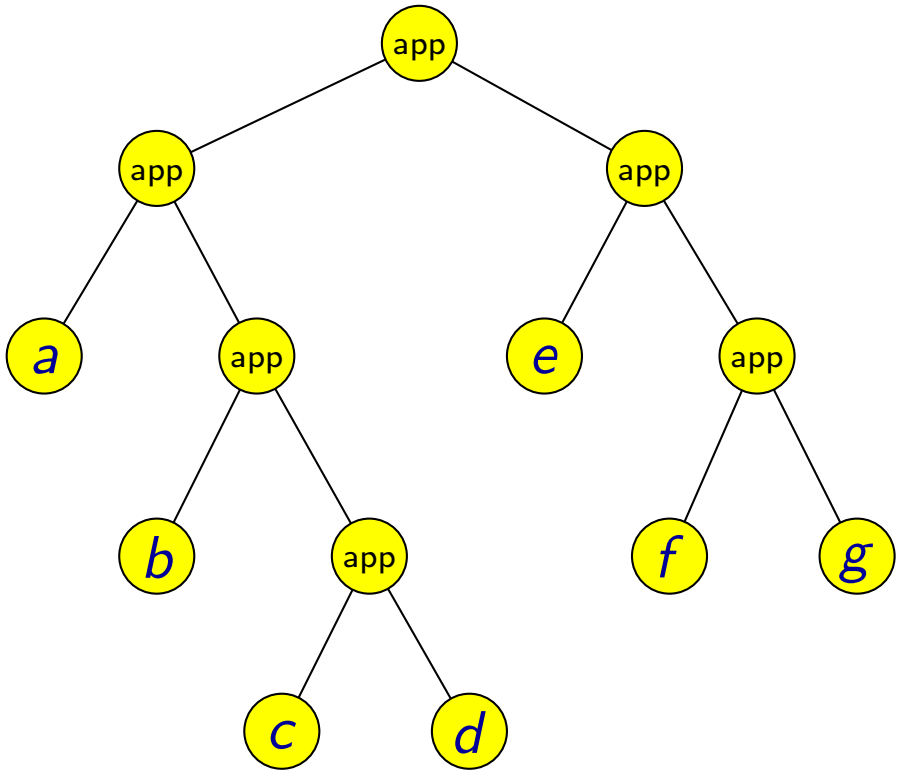


$t_1 : S \to T$
$t_2 : S$
$t_1\ t_2 : T$

Závorky

Strukturu termu je možné vyznačit pomocí závorek (znaky '(' a ')').

Příklad: term $(a\ (b\ (c\ d)))\ (e\ (f\ g)))$



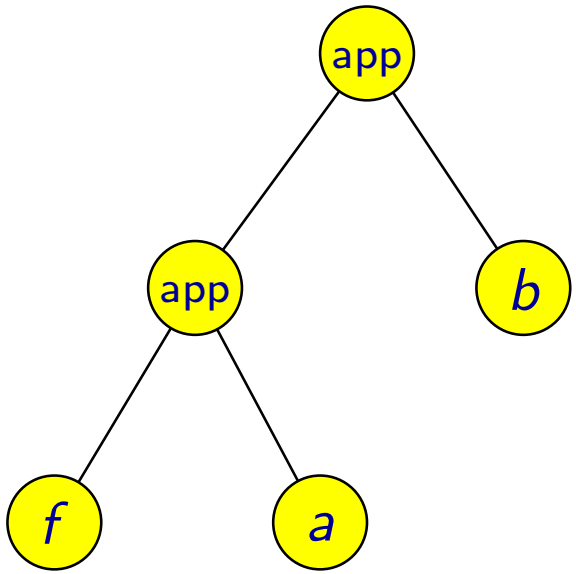
Currying

Veškeré funkce jsou **unární**, tj. očekávají jen jeden argument.

Funkce s vyšší aritou (binární, ternární, ...) je možné reprezentovat pomocí techniky zvané **currying**:

$f : A \to (B \to C)$
$a : A$
$b : B$

term $(f\ a)\ b$:



Na funkci f se můžeme dívat jako na funkci dvou argumentů (typů A a B), která vrací hodnotu typu C, a kterou voláme s hodnotami a a b.

Currying

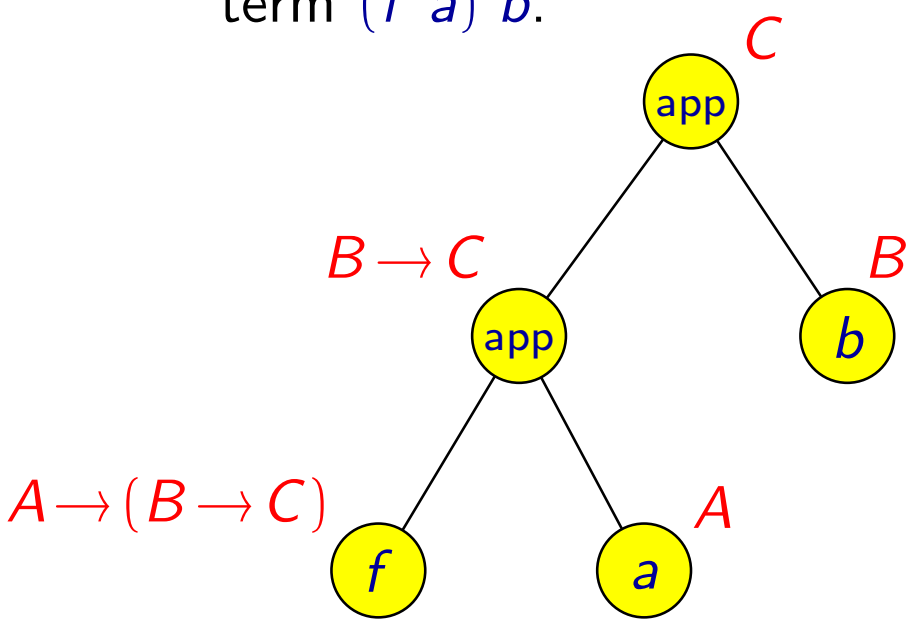
Veškeré funkce jsou **unární**, tj. očekávají jen jeden argument.

Funkce s vyšší aritou (binární, ternární, ...) je možné reprezentovat pomocí techniky zvané **currying**:

$f : A \to (B \to C)$
$a : A$
$b : B$

$f\ a : B \to C$
$(f\ a)\ b : C$

term $(f\ a)\ b$:



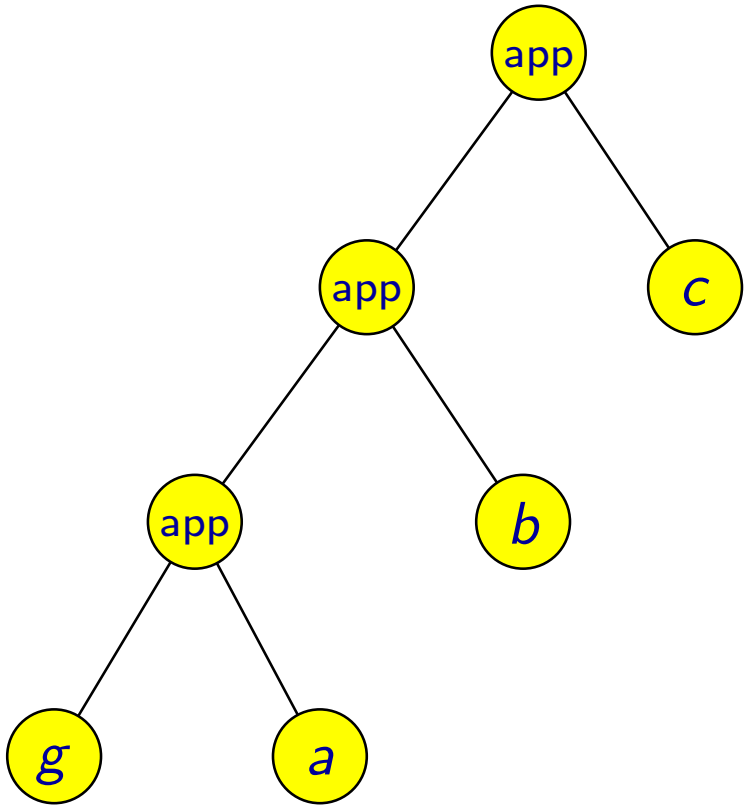
Na funkci f se můžeme dívat jako na funkci dvou argumentů (typů A a B), která vrací hodnotu typu C, a kterou voláme s hodnotami a a b.

Currying

Podobně můžeme reprezentovat ternární funkci g, která:

- očekává tři agrumenty typů A, B a C
- vrací návratovou hodnotu typu D

$g : A \to (B \to (C \to D))$
$a : A$
$b : B$
$c : C$



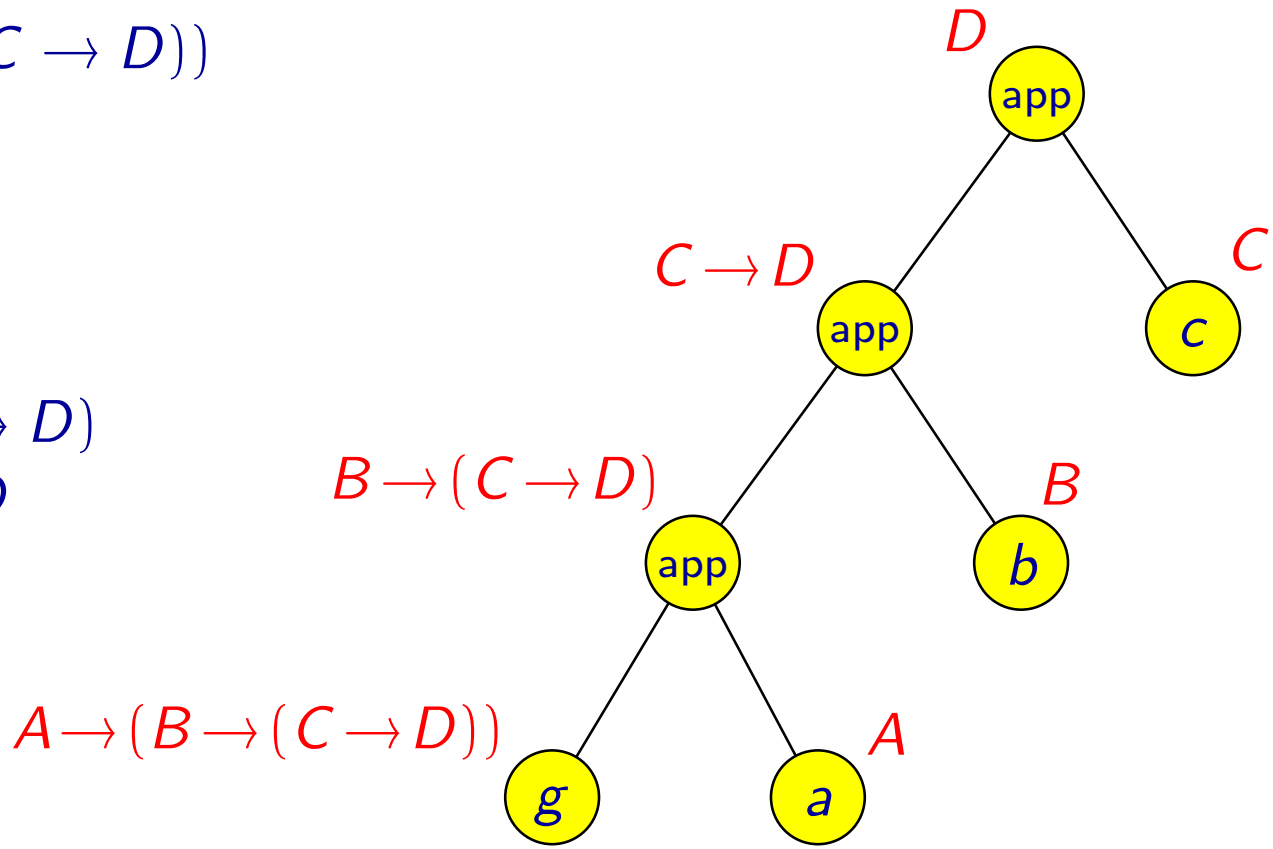
Currying

Podobně můžeme reprezentovat ternární funkci g, která:

- očekává tři agrumenty typů A, B a C
- vrací návratovou hodnotu typu D

$g : A \to (B \to (C \to D))$
$a : A$
$b : B$
$c : C$

$g\ a : B \to (C \to D)$
$(g\ a)\ b : C \to D$
$((g\ a)\ b)\ c : D$



Konvence pro vypouštění závorek

Protože je currying v zápisech termů velmi často používán a použití velkého množství závorek není příliš přehledné, používají se následující dvě konvence pro **vypouštění závorek**:

- aplikace funkce je **asociativní doleva**, tj. zápis

  $$f\ a_1\ a_2\ a_3\ a_4\ a_5\ a_6$$

  je interpretován jako

  $$((((((f\ a_1)\ a_2)\ a_3)\ a_4)\ a_5)\ a_6$$

- operace $\to$ vytvářející funkční typ je **asociativní doprava**, tj. zápis

  $$A_1 \to A_2 \to A_3 \to A_4 \to A_5 \to A_6 \to B$$

  je interpretován jako

  $$A_1 \to (A_2 \to (A_3 \to (A_4 \to (A_5 \to (A_6 \to B)))))$$

Currying

Příklad: Řekněme, že funkce add reprezentuje operaci sčítání na přirozených číslech:

$$\text{add} : \text{nat} \to \text{nat} \to \text{nat}$$

- add 5 3 — reprezentuje hodnotu $5 + 3$ (typu nat)
- add 5 — reprezentuje funkci typu $\text{nat} \to \text{nat}$, která pro hodnotu argumentu x vrací hodnotu $5 + x$

Currying

Příklad: Řekněme, že funkce add reprezentuje operaci sčítání na přirozených číslech:

$$\text{add} : \text{nat} \to \text{nat} \to \text{nat}$$

- add 5 3 — reprezentuje hodnotu $5 + 3$ (typu nat)
- add 5 — reprezentuje funkci typu $\text{nat} \to \text{nat}$, která pro hodnotu argumentu x vrací hodnotu $5 + x$

Příklad: Předpokládejme, že máme také funkci mul, která reprezentuje operaci násobení na přirozených číslech:

$$\text{mul} : \text{nat} \to \text{nat} \to \text{nat}$$

Běžný matematický zápis $x \cdot x + y \cdot y$ bychom mohli reprezentovat jako term

$$\text{add}\ (\text{mul}\ x\ x)\ (\text{mul}\ y\ y)$$

Abstrakce

Další důležitou konstrukcí, která můžeme v termech používat je **abstrakce**, která slouží k vytváření funkcí.

Term

$$\textbf{fun}\ x : A \Rightarrow t_1$$

reprezentuje anonymní funkci:

- Tato funkce očekává jako argument hodnotu typu A, která při volání této funkce bude přiřazena do proměnné x.

  Proměnná x tedy představuje **parametr** této funkce.

- Term t_1 představuje **tělo** této funkce.
  Proměnná x se může vyskytovat v tomto termu t_1 jako lokální proměnná.

- **Návratovou hodnotou** funkce je hodnota, na kterou se vyhotnotí term t_1, pokud se do parametru x dosadí příslušná hodnota argumentu, se kterou byla funkce volána.

Abstrakce

Pro konstrukci

$$\textbf{fun}\ x : A \Rightarrow t_1$$

se v matematických textech většinou používá zápis

$$\lambda x{:}A\,.\,t_1$$

proto se pro její označení kromě pojmu „abstrakce“ někdy též používá pojem **lambda funkce**.

Poznámka: Pro zvýraznění je též možné uzavřít parametr x a jeho typ A do závorek:

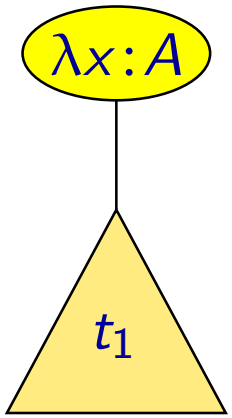
$$\textbf{fun}\ (x : A) \Rightarrow t_1$$

Abstrakce

Pokud za předpokladu, že proměnná x je typu A, platí, že term t_1 je typu B, tak typ termu

$$\textbf{fun}\ x : A \Rightarrow t_1$$

je $A \rightarrow B$ (tj. funkce z A do B).

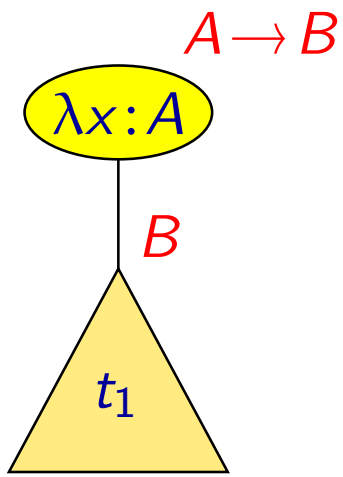


Abstrakce

Pokud za předpokladu, že proměnná x je typu A, platí, že term t_1 je typu B, tak typ termu

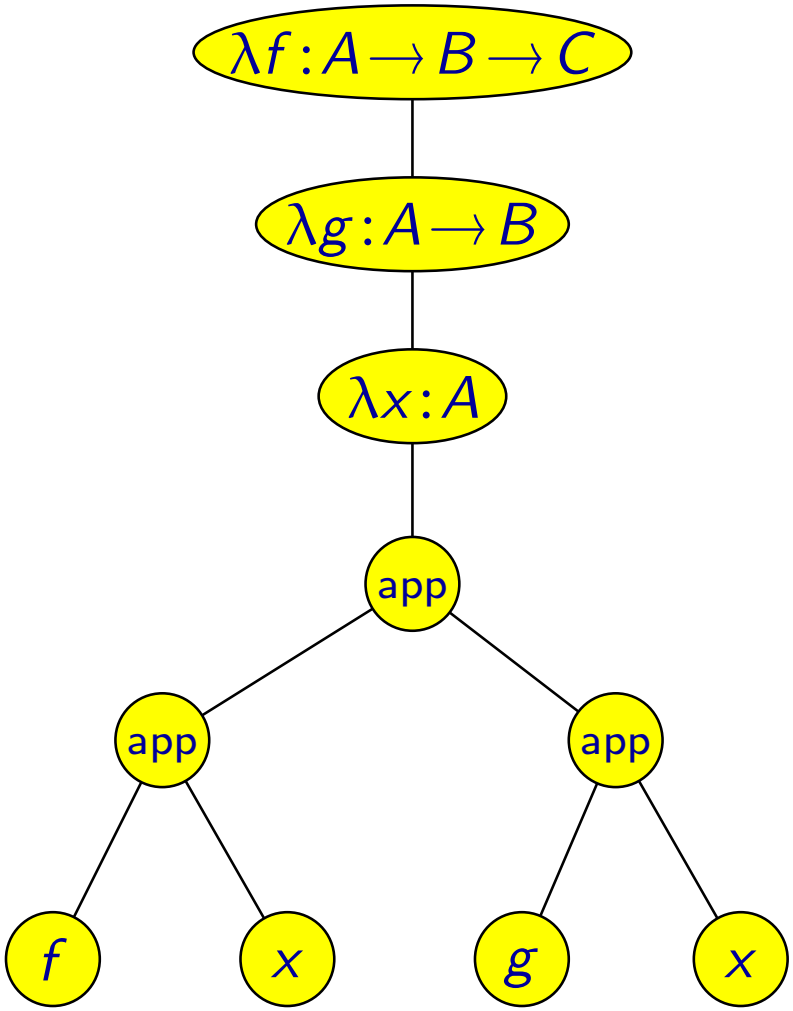
$$\textbf{fun}\ x : A \Rightarrow t_1$$

je $A \to B$ (tj. funkce z A do B).



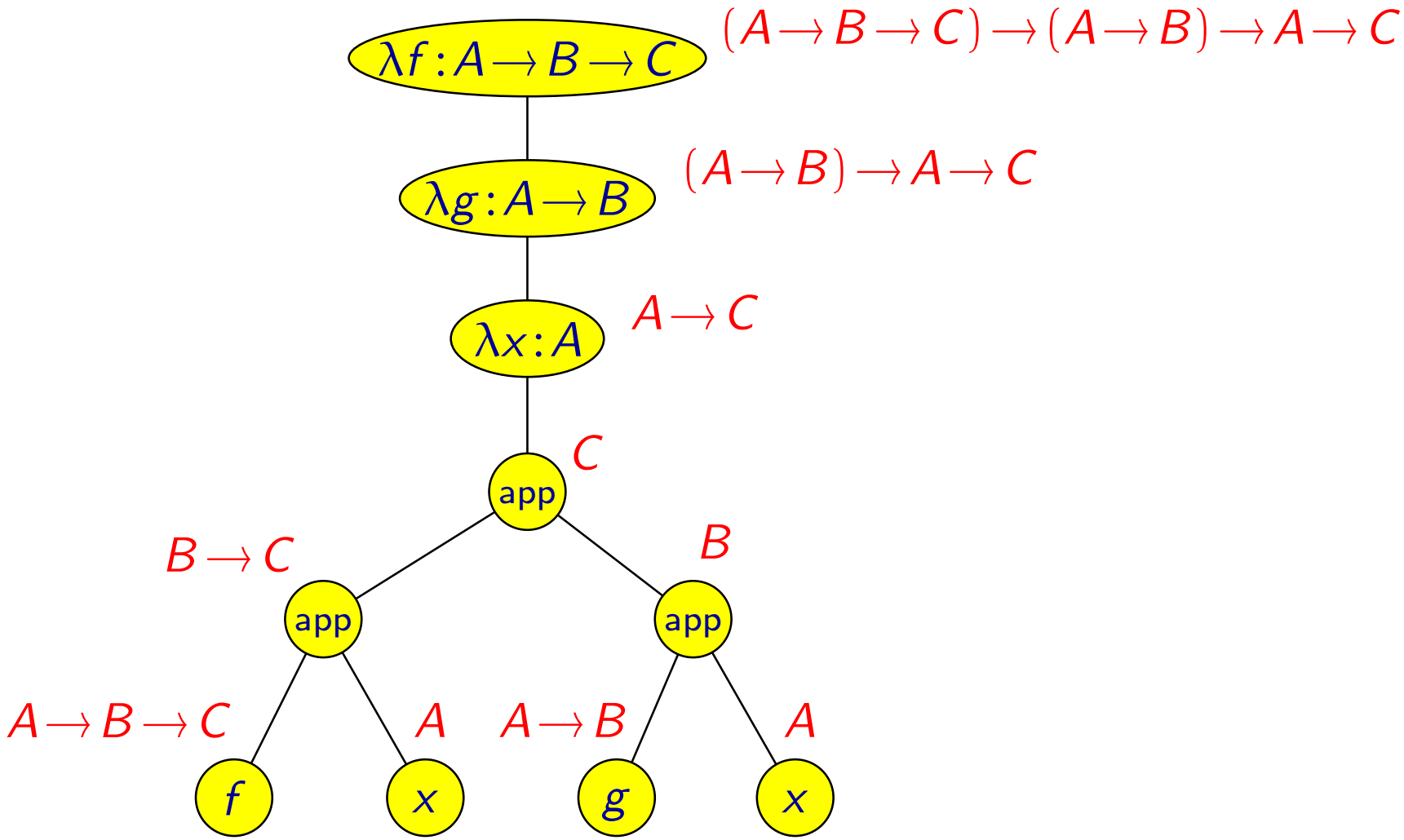
Příklad: Abstraktní syntaktický strom termu

fun $(f : A \to B \to C) \Rightarrow (\textbf{fun}\ (g : A \to B) \Rightarrow (\textbf{fun}\ (x : A) \Rightarrow (f\ x\ (g\ x))))$



Příklad: Abstraktní syntaktický strom termu

fun $(f : A \to B \to C) \Rightarrow$ (**fun** $(g : A \to B) \Rightarrow$ (**fun** $(x : A) \Rightarrow (f\ x\ (g\ x))$))



Abstrakce

Abstrakce je asociativní doprava, tj.

$$\textbf{fun}\ (x_1 : A_1) \Rightarrow \textbf{fun}\ (x_2 : A_2) \Rightarrow \textbf{fun}\ (x_3 : A_3) \Rightarrow \textbf{fun}\ (x_4 : A_4) \Rightarrow t_1$$

je interpretováno jako

$$\textbf{fun}\ (x_1 : A_1) \Rightarrow (\textbf{fun}\ (x_2 : A_2) \Rightarrow (\textbf{fun}\ (x_3 : A_3) \Rightarrow (\textbf{fun}\ (x_4 : A_4) \Rightarrow t_1)))$$

Navíc je v tomto případě možné používat následující zkrácený zápis (který reprezentuje přesně stejný term jako výše uvedený delší zápis):

$$\textbf{fun}\ (x_1 : A_1)\ (x_2 : A_2)\ (x_3 : A_3)\ (x_4 : A_4) \Rightarrow t_1$$

Poznámka: Při tomto zkráceném způsobu zápisu není možné vynechat závorky kolem názvů proměnných a jejich typů: $(x_1 : A_1)$, $(x_2 : A_2)$, ...

Abstrakce

Příklad: Místo termu

$$\textbf{fun}\ (f : A \to B \to C) \Rightarrow (\textbf{fun}\ (g : A \to B) \Rightarrow (\textbf{fun}\ (x : A) \Rightarrow (f\ x\ (g\ x))))$$

je možné psát

$$\textbf{fun}\ (f : A \to B \to C)\ (g : A \to B)\ (x : A) \Rightarrow f\ x\ (g\ x)$$

Abstrakce

Pokud má několik po sobě jdoucích parametrů stejný typ, je možné je spojit do jedné závorky, kde je typ uveden jen jednou.

Příklad: Místo

$$\textbf{fun}\ (x : A)\ (y : A)\ (z : A)\ (u : B)\ (v : B) \Rightarrow t_1$$

je možné psát

$$\textbf{fun}\ (x\ y\ z : A)\ (u\ v : B) \Rightarrow t_1$$

Pokud platí následující:

- pokud proměnné x, y a z jsou typu A a proměnné u a v typu B, tak term t_1 typu C,

bude výše uvedený term typu

$$A \to A \to A \to B \to B \to C$$

Definice konstant

Pro zápis konstant, jejichž hodnota je funkce, je možné používat stručnější a přehlednější způsob zápisu (který funguje jako syntactic sugar pro výše uvedené):

Definition id $(x : \mathsf{nat}) := x$.

Je možné i explicitně specifikovat typ návratové hodnoty:

Definition id $(x : \mathsf{nat}) : \mathsf{nat} := x$.

Definice konstant

Příklad funkce s více argumenty (několik alternativních způsobů zápisu):

```
Definition f := fun x : nat ⇒
                  fun y : nat ⇒
                    add (mul x x) (mul y y).

Definition f := fun (x : nat) (y : nat) ⇒
                    add (mul x x) (mul y y).

Definition f := fun (x y : nat) ⇒
                    add (mul x x) (mul y y).

Definition f (x : nat) (y : nat) :=
                    add (mul x x) (mul y y).

Definition f (x y : nat) :=
                    add (mul x x) (mul y y).
```

Typová inference

V zápise funkcí je možné často typy parametrů vynechat, protože typový systém Coqu je shopen příslušné typy dopočítat pomocí **typové inference**.

Je tak možné psát například

```
Definition f x y := add (mul x x) (mul y y).
```

nebo

```
Definition f := fun x y ⇒ add (mul x x) (mul y y).
```

V obou případech systém automaticky určí, že proměnné x a y mají být typu nat.

Do environmentu se uloží vždy plně otypovaný term, kde jsou doplněné typy proměnných uvedeny.

Typová inference

Ne vždy je možné typ proměnné pomocí typové inference doplnit.

Například u následující definice zahlásí systém chybu, protože z uvedených informací není schopen typ proměnné x určit:

Definition id $x := x$. $\longleftarrow$ Chyba!

Konstrukce **let** .. **in** .. — lokální definice

Term tvaru

$$\textbf{let } x : A := t_1 \textbf{ in } t_2$$

reprezentuje definici **lokální proměnné** x typu A, které je přiřazena hodnota daná termem t_1.

Proměnná x může být libovolně použita v termu t_2.

Celková hodnota termu **let** $x : A := t_1$ **in** t_2 je dána vyhodnocením termu t_2 s tím, že hodnota proměnné x tomto termu je dána hodnotou termu t_1.

Podobně jako u funkcí, je i u výše uvedené konstrukce možné typ proměnné vynechat. Systém se ho pokusí dopočítat pomocí typové inference:

$$\textbf{let } x := t_1 \textbf{ in } t_2$$

Konstrukce **let** .. **in** .. — lokální definice

Příklad:
Funkce

```
Definition h (x : nat) :=
    let y : nat := g (f x x) in
    add (mul y y) x.
```

či

```
Definition h (x : nat) :=
    let y := g (f x x) in
    add (mul y y) x.
```

bude počítat stejné hodnoty jako funkce

```
Definition h (x : nat) :=
    add (mul (g (f x x)) (g (f x x))) x.
```

Konstrukce **let .. in ..** — lokální definice

Pokud lokální proměnné přiřazujeme funkci, můžeme používat podobné konvence jako při definici funkce pomocí příkazu **Definition**.

Příklad: Místo

```
Definition h (x : nat) :=
    let f := (fun y : nat ⇒ g y x) in
    f (f x).
```

můžeme psát

```
Definition h (x : nat) :=
    let f (y : nat) := g y x in
    f (f x).
```

Explicitní stanovení typu termu

Term tvaru

$$t : A$$

se vyhodnotí stejně jako term t.

Při typové kontrole se navíc zkountroluje, že term t je skutečně typu A. (Pokud ne, je ohlášena chyba.)