Kombinátorová logika a bezpředmětové programování

LKFP přednáška 8, jaro 2024/25

Jan Laštovička

31. března 2025

1 Kombinátorová logika

Abeceda kombinátorové logiky *CL* je tvořena

1. konstantami **S** a **K**,
2. proměnnými $x, y, z, \ldots$,
3. symboly levé a pravé závorky.

Množina výrazů $\mathcal{C}$ kombinátorové logiky je nejmenší množina, která splňuje, že

1. pokud x je proměnná, pak $x \in \mathcal{C}$,
2. $\mathbf{S} \in \mathcal{C}$ a $\mathbf{K} \in \mathcal{C}$,
3. pokud $P \in \mathcal{C}$ a $Q \in \mathcal{C}$, pak $(PQ) \in \mathcal{C}$.

Nejvíce vnější závorky vynecháváme a $PQ_1Q_2 \ldots Q_k \equiv \cdots ((PQ_1)Q_2) \cdots Q_k$. Množina *volných proměnných* $\mathrm{FV}(M)$ výrazu M je rovna množině všech proměnných vyskytujících se ve výrazu. Nahrazení všech výskytů proměnné x ve výrazu P za výraz Q značíme $P[x := Q]$.

Teorie kombinátorové logiky je dána axiomy:

1. $\mathbf{K}PQ = P$,
2. $\mathbf{S}PQR = PR(QR)$,
3. $P = P$

a odvozovacími pravidly:

$$\frac{P = Q}{Q = P}, \qquad \frac{P = Q, Q = R}{P = R},$$
$$\frac{P = Q}{PR = QR}, \qquad \frac{P = Q}{RP = RQ}.$$

Definujeme

1. $\mathbf{I} \equiv \mathbf{SKK}$,
2. $\mathbf{B} \equiv \mathbf{S}(\mathbf{KS})\mathbf{K}$,
3. $\mathbf{C} \equiv \mathbf{S}(\mathbf{BBS})(\mathbf{KK})$,
4. $\mathbf{T} \equiv \mathbf{CI}$
5. $\mathbf{M} \equiv \mathbf{SII}$,
6. $\mathbf{L} \equiv \mathbf{CBM}$,
7. $\mathbf{Y} \equiv \mathbf{SLL}$.

Pro každé výrazy P, Q, R platí:

1. $\mathbf{I}P = P$,
2. $\mathbf{B}PQR = P(QR)$,
3. $\mathbf{C}PQR = PRQ$,
4. $\mathbf{T}PQ = QP$
5. $\mathbf{M}P = PP$,
6. $\mathbf{L}PQ = P(QQ)$,
7. $\mathbf{Y}P = P(\mathbf{Y}P)$.

Množinu

$$\{(\mathbf{K}PQ, P) \mid P, Q \in \mathcal{C}\} \cup \{(\mathbf{S}PQR, PR(QR)) \mid P, Q, R \in \mathcal{C}\}$$

si označíme w a nazýváme *osnovou slabé redukce*. Označíme $\rightarrow_w$ nejmenší relaci obsahující w, která je kompatibilní s aplikací, $\twoheadrightarrow_w$ reflexivní a tranzitivní uzávěr $\rightarrow_w$ a $=_w$ nejmenší ekvivalenci obsahující $\twoheadrightarrow_w$. Redukce w má Churchovu-Rosserovu vlastnost. Normální formu redukce w nezýváme *slabou normální formou*. Redukci w nazýváme slabou, protože $\mathbf{SK} \neq_w \mathbf{KI}$, ale $\mathbf{SK} \neq_\beta \mathbf{KI}$, kde $\mathbf{S}, \mathbf{K}, \mathbf{I}$ označují příslušné kombinátory z teorie λ.

Platí $P =_w Q$, právě když $P = Q$.

Churchovo-Rosserova věta pro kombinátorovou logiku:

1. Pokud $P = Q$ a P je v slabé normální formě, pak $P \twoheadrightarrow_w Q$.
2. Pokud P a Q jsou dvě různé slabé normální formy, pak $P \neq Q$. Z čehož vyplývá, že kombinátorová logika je konzistentní.

Indukcí na struktuře výrazu P definujeme $\lambda^* x.P$ následovně.

1. $\lambda^* x.x \equiv \mathbf{I}$,
2. $\lambda^* x.P \equiv \mathbf{K}P$, jestliže $x \notin \mathrm{FV}(P)$,

3. $\lambda^* x.PQ \equiv \mathbf{S}(\lambda^* x.P)(\lambda^* x.Q)$.

Platí:

1. $\mathrm{FV}(\lambda^* x.P) = \mathrm{FV}(P) - \{x\}$,
2. $(\lambda^* x.P)Q = P[x := Q]$.
3. jestliže $y \notin \mathrm{FV}(P)$, pak $\lambda^* x.P \equiv \lambda^* y.P[x := y]$.

Pravidlo extenzionality *ext*:

$$\frac{Px = Qx \quad x \notin \mathrm{FV}(PQ)}{P = Q}$$

Induktivně definujeme zobrazení $(\cdot)_\lambda \colon \mathcal{C} \to \Lambda$:

1. $(x)_\lambda \equiv x$,
2. $(\mathbf{S})_\lambda \equiv \mathbf{S} \equiv \lambda xyz.xz(yz)$,
3. $(\mathbf{K})_\lambda \equiv \mathbf{K} \equiv \lambda xy.x$,
4. $(PQ)_\lambda \equiv (P)_\lambda (Q)_\lambda$.

a zobrazení $(\cdot)_{CL} \colon \Lambda \to \mathcal{C}$:

1. $(x)_{CL} \equiv x$,
2. $(MN)_{CL} \equiv (M)_{CL}(N)_{CL}$,
3. $(\lambda x M)_{CL} \equiv \lambda^* x.(M)_{CL}$.

Jestliže $P = Q$, pak $(P)_\lambda = (Q)_\lambda$, kde P, Q jsou výrazy kombinátorové logiky. Obrácená implikace neplatí, protože $\lambda \vdash \mathbf{SK} = \mathbf{KI}$, ale $CL \nvdash \mathbf{SK} = \mathbf{KI}$.

Teorie $\lambda + ext$ a $CL + ext$ jsou ekvivalentní. Přesněji platí:

1. $\lambda + ext \vdash ((M)_{CL})_\lambda = M$,
2. $CL + ext \vdash ((P)_\lambda)_{CL} = P$,
3. $\lambda + ext \vdash M = N$, právě když $CL + ext \vdash (M)_{CL} = (N)_{CL}$,
4. $CL + ext \vdash P = Q$, právě když $\lambda + ext \vdash (P)_\lambda = (Q)_\lambda$.

Je možné rozšířit CL o několik axiomů A tak, že $CL + A$ bude ekvivalentní s λ v právě uvedeném smyslu. Axiomy najdete v knize *The Lambda Calculus, Its Syntax and Semantics* od Henka Barendregta na straně 158.

Převod výrazů lambda kalkulu do výrazů kombinátorové logiky lze vylepšit použitím následujících rovností dokázaných v $CL + ext$:

1. $\mathbf{S}(\mathbf{K}P)\mathbf{I} = P$
2. $\mathbf{S}(\mathbf{K}P)Q = \mathbf{B}PQ$

3. $\mathbf{S}P(\mathbf{K}Q) = \mathbf{C}PQ$

Kompilátor programovacího jazyka Miranda (předchůdce Haskellu) kompiluje programový kód do kombinátorové logiky, který je pak zpracováván rychlým interpretem.

Pro množinu $X \subseteq \Lambda^0$ kombinátorů označme X^+ nejmenší množinu, která splňuje:

1. $X \subseteq X^+$,
2. pokud $M, N \in X^+$, pak $(MN) \in X^+$.

Množina X je *báze*, jestliže pro každý kombinátor $M \in \Lambda^0$ existuje $N \in X^+$ tak, že $N = M$. Platí, že $\{\mathbf{S}, \mathbf{K}\}$ je báze. (Musíme použít extenzionalitu nebo rozšíření kombinátorové logiky o axiomy A.) Definujeme $\mathbf{X} \equiv \lambda x.x\mathbf{KSK}$. Pak $\{\mathbf{X}\}$ je báze. Stačí ověřit, že $\mathbf{XXX} = \mathbf{K}$ a $\mathbf{X}(\mathbf{XX}) = \mathbf{S}$.

2 Hindleyho-Milnerův typový systém

Abeceda typů je tvořena

1. *typovými konstruktory* $C_1, C_2, \ldots$
2. *typovými proměnnými* $\alpha_1, \alpha_2, \ldots$,
3. znaky tečka, $\forall$ a pravá a levá závorka.

Každému typovému konstruktoru C je přiřazeno celé nezáporné číslo $\delta(C)$ nazývané *arita* C. Musí existovat typový konstruktor $\to$ s aritou 2. Budeme uvažovat typové konstruktory `String` a `Integer` s aritou nula a typový konstruktor `List` s aritou jedna.

Označme $V = \{\alpha_1, \alpha_2, \ldots, \}$ množinu všech typových proměnných.

Množina *monotypů* $Monotyp$ je nejmenší množina, pro kterou platí:

1. $V \subseteq Monotyp$,
2. pokud C je typový konstruktor s aritou n a $\tau_1, \ldots, \tau_n \in Monotyp$, pak $(C\tau_1 \ldots \tau_n) \in Monotyp$.

Místo $(\to\ \tau_1\tau_2)$ píšeme $(\tau_1 \to \tau_2)$. Typy s aritou nula (C) zapisujeme C. Přijímáme konvence o vynechávání závorek z typovaného lambda kalkulu. Například (`Integer` $\to$ `String`) $\to$ (`List Integer`) $\to$ (`List String`) nebo (`List` α) $\to$ `Integer` jsou monotypy.

Množina volných proměnných $\mathrm{FV}(\tau)$ monotypu τ je dána pravidly:

1. $\mathrm{FV}(\alpha) = \{\alpha\}$,
2. $\mathrm{FV}(C\tau_1 \ldots \tau_n) = \mathrm{FV}(\tau_1) \cup \cdots \cup \mathrm{FV}(\tau_n)$.

Zobrazení $S : V \to Monotyp$, které typovým proměnným přiřazuje monotypy, se nazývá *typová substituce*. Zápis $\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}$ zapisuje substituci S takovou, že $S(\alpha_i) = \tau_i$ pro všechna $1 \le i \le n$ a $S(\alpha) = \alpha$ pro každou typovou proměnnou α různou od $\alpha_1, \ldots, \alpha_n$. *Aplikace* $S\tau$ substituce S na monotyp τ je dána pravidly:

1. $S\alpha = S(\alpha)$,
2. $S(C\tau_1 \ldots \tau_n) = C(S\tau_1 \ldots S\tau_n)$.

Například pro $S = \{\alpha \mapsto \text{Integer}\}$ obdržíme $S(\alpha \to \alpha) = \text{Integer} \to \text{Integer}$.

Množina *polytypů* $Polytyp$ je nejmenší množina, pro kterou platí:

1. $Monotyp \subseteq Polytyp$,
2. pokud $\sigma \in Polytyp$ a α je proměnná, pak $\forall\alpha.\sigma \in Polytyp$.

Například $\forall\alpha.\forall\beta.\alpha \to \beta \to \alpha$ je polytyp.

Pro polytyp $\forall\alpha.\sigma$ je množina jeho volných proměnných dána rovností $\mathrm{FV}(\forall\alpha.\sigma) = \mathrm{FV}(\sigma) - \{\alpha\}$.

Polytyp $\forall\alpha_1 \cdots \forall\alpha_n.\tau$ je *obecnější než* polytyp $\forall\beta_1 \cdots \forall\beta_m.\tau'$, jestliže existuje substituce $S = \{\alpha_1 \mapsto \tau_1, \ldots, \alpha_k \mapsto \tau_k\}$ taková, že $\tau' = S\tau$ a množina $\mathrm{FV}(\forall\alpha_1 \cdots \forall\alpha_n.\tau)$ je disjunktní s $\{\beta_1, \ldots, \beta_m\}$, zapisujeme

$$\forall\alpha_1 \cdots \forall\alpha_n.\tau \sqsubseteq \forall\beta_1 \cdots \forall\beta_m.\tau'.$$

Například $\forall\alpha.\alpha \to \beta \to \alpha$ je obecnější než $\forall\gamma.(\gamma \to \gamma) \to \beta \to (\gamma \to \gamma)$ a ten je obecnější než $(\texttt{Integer} \to \texttt{Integer}) \to \beta \to (\texttt{Integer} \to \texttt{Integer})$.

Formule

$$M : \sigma$$

zapisuje, že výraz M je typu σ. Posloupnost formulí $x_1 : \sigma_1, \ldots, x_n : \sigma_n$, kde proměnné $x_1, \ldots, x_n$ jsou po dvou různé, nazýváme *kontextem* a značíme Γ. Množinu $\mathrm{FV}(\Gamma)$ volných typových proměnných v kontextu Γ definujeme $\mathrm{FV}(\Gamma) = \mathrm{FV}(\sigma_1) \cup \cdots \cup \mathrm{FV}(\sigma_n)$. Výraz $\alpha : \sigma \in \Gamma$ značí, že se formule $\alpha : \sigma$ nalézá v posloupnosti Γ. Prázdný kontext značíme $\emptyset$ nebo jej úplně vynecháváme.

Relace odvoditelnosti $\alpha : \sigma$ z kontextu Γ, kterou symboliky zapíšeme $\Gamma \vdash \alpha : \sigma$, je dána pravidly:

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}\ (var), \qquad \frac{\Gamma \vdash M : (\tau \to \tau') \quad \Gamma \vdash N : \tau}{\Gamma \vdash (MN) : \tau'}\ (app),$$

$$\frac{\Gamma, x : \tau \vdash M : \tau'}{\Gamma \vdash (\lambda x M) : (\tau \to \tau')}\ (abs), \qquad \frac{\Gamma \vdash M : \sigma \quad \sigma \sqsubseteq \sigma'}{\Gamma \vdash M : \sigma'}\ (inst),$$

$$\frac{\Gamma \vdash M : \sigma \quad \alpha \notin free(\Gamma)}{\Gamma \vdash M : \forall\alpha.\sigma}\ (gen).$$

Následující strom dokazuje, že $\Gamma \vdash (\texttt{id c}) : \texttt{Integer}$, kde $\Gamma = \texttt{id} : \forall\alpha.\alpha \to \alpha, \texttt{c} : \texttt{Integer}$.

$$\dfrac{\dfrac{\dfrac{\texttt{id} : \forall\alpha.\alpha \to \alpha \in \Gamma}{\Gamma \vdash \texttt{id} : \forall\alpha.\alpha \to \alpha}\,(var) \qquad \forall\alpha.\alpha \to \alpha \sqsubseteq \texttt{Integer} \to \texttt{Integer}}{\Gamma \vdash \texttt{id} : \texttt{Integer} \to \texttt{Integer}}\,(inst) \qquad \dfrac{\texttt{c} : \texttt{Integer} \in \Gamma}{\Gamma \vdash \texttt{c} : \texttt{Integer}}\,(var)}{\Gamma \vdash (\texttt{id}\ \texttt{c}) : \texttt{Integer}}\,(app)$$

Následující strom dokazuje, že $\vdash (\lambda x x) : \forall\alpha.\alpha \to \alpha$.

$$\dfrac{\dfrac{\dfrac{x : \alpha \in x : \alpha}{x : \alpha \vdash x : \alpha}\,(var)}{\vdash (\lambda x x) : \alpha \to \alpha}\,(abs) \qquad \alpha \notin \mathrm{FV}(\emptyset)}{\vdash (\lambda x x) : \forall\alpha.\alpha \to \alpha}\,(gen)$$

3 Bezpředmětové programování

Typový systém Haskellu vychází z Hindleyho-Milnerova typového systému. Polytyp $\forall\alpha_1.\ldots\forall\alpha_n.\tau$ zapíšeme pouze jako typ τ. Například:

```
id' :: a -> a
id' x = x
```

Použití:

```
>>> id' (\ x -> x) (id' 1)
1
```

Typový konstruktor C s n argumenty můžeme v Haskellu definovat konstrukcí `data`:

`data` $C\ \alpha_1\ \ldots\ \alpha_n$ `=` *constructors* `deriving Show`

Konstruktory *constructors* můžou používat typové proměnné $\alpha_1, \ldots, \alpha_n$. Například:

```
data List a = Nil | Cons a (List a) deriving Show
```

Ukázka použití:

```
x :: List Integer
x = Cons 1 (Cons 2 Nil)
```

Základní kombinátory v Haskellu:

1. $\mathbf{I}$ = `id`
2. $\mathbf{B}$ = `(.)`
3. $\mathbf{C}$ = `flip`

4. **K** = `const`

5. $(\lambda fx.fx)$ = `($)`

Technika *bezpředmětového programování* spočívá v tom, že programujeme pouze s použitím kombinátorů. Tedy nepoužíváme proměnné. Například funkci na sečtení tří čísel:

```
sum3 :: Integer -> Integer -> Integer -> Integer
sum3 x y z = (x + y) + z
```

můžeme bez proměnných napsat takto:

```
sum3' :: Integer -> Integer -> Integer -> Integer
sum3' = (.) ((.) (+)) (+)
```

Čitelnější verzi obdržíme použitím infixové notace kombinátoru `(.)`:

```
sum3'' :: Integer -> Integer -> Integer -> Integer
sum3'' = ((+) .) . (+)
```

Poznámka: Pro infixový operátor *op* je `(`*arg op*`)` ekvivalentní s `(`*op*`)` *arg*.

Vezměme jako další příklad funkci počítající počet výskytů čísla v seznamu:

```
count :: Int -> [Int] -> Int
count x [] = 0
count x (y:ys) =
  if x == y then 1 + count x ys
  else count x ys
```

Funkci můžeme přepsat za použití `length` a `filter` (filtrování seznamu):

```
count' :: Int -> [Int] -> Int
count' x y = length (filter ((==) x) y)
```

Nyní už je snadné odstranit proměnné:

```
count'' :: Int -> [Int] -> Int
count'' = (length .) . filter . (==)
```

Pro práci se seznamy je užitečný kombinátor `foldr`, který by mohl být definován:

```
foldr' :: (a -> b -> b) -> b -> [a] -> b
foldr' f z [] = z
foldr' f z (x:xs) = f x (foldr' f z xs)
```

Napište s jeho pomocí bezpředmětově funkce `sum' :: [Integer] -> Integer`

počítající součet prvků seznamu,

`length' :: [a] -> Int`

vracející délku seznamu a

`map' :: (a -> b) -> [a] -> [b]`

mapující funkci přes seznam.