Teorie kategorií pro programátory

LKFP přednáška 9, jaro 2024/25

Jan Laštovička

9. dubna 2025

1 Model jednoduše typovaného lambda kalkulu

Každému atomickému typu $b \in B$ přiřadíme množinu $||X||$. Funkčním typům $\sigma \to \tau$ pak můžeme induktivně přiřadit množinu $||\sigma \to \tau||$ všech funkcí z $||\sigma||$ do $||\tau||$; tedy $||\sigma \to \tau|| = ||\tau||^{||\sigma||}$. Takto jsme každému typu σ přiřadili množinu $||\sigma||$.

Prostředím e myslíme zobrazení, které každé proměnné v_i^σ přiřazuje hodnotu $e(v_i^\sigma) \in ||\sigma||$. *Model* $||M||_e$ výrazu M typovaného lambda kalkulu v prostředí e je prvek množiny $||\sigma||$ definovaný následovně.

1. $||x||_e = e(x)$, kde x je proměnná,
2. $||MN||_e = ||M||_e(||N||_e)$,
3. $||\lambda x.M||_e = \{(d, ||M||_{e'}) \mid d \in ||\sigma|| \text{ a } e[x \to d]\}$, kde σ je typ proměnné x a $e[x \to d]$ značí prostředí shodné s e až na to, že x je přiřazena hodnota d.

Pokud je výraz M uzavřený, pak místo $||M||_e$ můžeme psát jen $||M||$.

Pokud se dva uzavřené výrazy M a N rovanjí v teorii $\lambda\eta$, pak $||M|| = ||N||$.

Pokud je pro výraz M čistého lambda kalkulu σ možný typ, pak *doplněním typu proměnným* v M podle σ myslíme typ N typovaného lambda kalkulu, pro který je $|N| = M$. Například doplněním typu proměnných v $\lambda x.x$ podle $i \to i$ ($i \in B$) získáme výraz $\lambda x^i.x^i$.

2 Model omezeného Haskellu

V této časti si vytvoříme množinový model omezené části Haskellu. Následujícími řádky importujeme jen vybrané symboly ze základní výbavy Haskellu.

```
-# LANGUAGE NoImplicitPrelude #-
import Prelude (Show, Integer, (*), (+), (-))
```

Model získáme vhodným rozšířením jednoduše typovaného lambda kalkulu.

Modely atomických vestavěných typů musíme zadat. Například ||`Integer`|| je množina všech celých čísel $\mathbb{Z}$.

Vezměme definici datového typu:

```
data Type b1 ... bn = constructors
```

U definic datových typů budeme vynechávat část `deriving Show` umožňující snadný tisk hodnot. Nepřipouštíme definici rekurzivního datového typu. Pro každé uzavřené monotypy *type1*,...,*typen* je ||*Type type1 ... typen*|| nejmenší množina, která pro každou instanci konstruktoru v *constructors*:

```
Const type1' ... typem'
```

splňuje:

$$(\mathit{Const}, x_1, \ldots, x_m) \in ||\mathit{Type\ type1\ \ldots\ typen}||,$$

kde $x_i \in ||\mathit{typei'}||$ pro každé $1 \leq i \leq m$.

Například pro:

```
data Boolean = True | False
```

je ||`Boolean`|| = {`True`, `False`} nebo pro:

```
data Maybe a = Nothing | Just a
```

je ||`Maybe Integer`|| = {`Nothing`} ∪ {(`Just`, z) | $z \in \mathbb{Z}$}

Literály modelujeme prvky modelu jejich typu. Například:

$$||\texttt{1}|| = 1 \in \mathbb{Z} = ||\texttt{Integer}||.$$

Modely vestavěných funkcí musíme zadat. Například:

$$||\texttt{(+)}^{\texttt{Integer -> Integer -> Integer}}|| = F^+ = \{(x, F_x^+) \mid x \in \mathbb{Z}\},$$

kde $F_x^+ = \{(y, x+y) \mid y \in \mathbb{Z}\}$. Platí $||\texttt{(+)}|| \in \mathbb{Z}^{\mathbb{Z}^{\mathbb{Z}}}$. Například dostáváme:

$$||\texttt{(+) 2 3}|| = (||\texttt{(+)}||(2))(3) = F^+(2)(3) = F_2^+(3) = 5.$$

Vezměme definici jména:

```
name :: type
name = expr
```

Zavedeme omezení, že výraz ***expr*** může obsahovat jen dříve definovaná jména. Nepřipouštíme tedy rekurzivní definice.

Stejně jako proměnné, mají i jména přiřazený monotyp. Tedy modelujeme $\mathit{name}^{\mathit{type'}}$, kde $\mathit{type'}$ je specifičtější než type. Jméno modelujeme:

$$||\mathit{name}^{\mathit{type'}}|| = ||\mathit{expr'}|| \in ||\mathit{type'}||,$$

kde $\mathit{expr'}$ je výraz vzniklý z expr doplněním typů proměnných podle $\mathit{type'}$. Například pro:

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

je $||\texttt{id}^{\texttt{Integer -> Integer}}|| = ||\lambda x^{\texttt{Integer}}.x^{\texttt{Integer}}|| = \{(z,z) \mid z \in \mathbb{Z}\} \in \mathbb{Z}^{\mathbb{Z}}$.

Rozbor R:

(case M of
  $c_1\ \overrightarrow{x_1}$ -> N_1
  $\vdots$
  $c_m\ \overrightarrow{x_m}$ -> N_m)

modelujeme

$$||R||_e = f(x_1, \ldots x_n),$$

kde

$$f = ||(\lambda\overrightarrow{x_i}.N_i)||,$$
$$||M||_e = (c_i, x_1, \ldots, x_n).$$

Například model pro:

```
case Just 1 of
  Just x -> x
  Nothing -> 0
```

je číslo 1.

Komplexnější příklad:

```
maybeHead :: List a -> Maybe a
maybeHead =  x -> case x of
  (Cons x xs) -> Just x
  Nil -> Nothing
```

$||\texttt{maybeHead}^{\texttt{List Integer -> Maybe Integer}}||$
$= \{(\texttt{Nil}, \texttt{Nothing})\} \cup \{((\texttt{Cons}, z, l), (\texttt{Just}, z)) \mid z \in \mathbb{Z} \text{ a } l \in ||\texttt{List Integer}||\}.$

Samozřejmě stále můžeme používat rozpoznávání vzoru jako zkratku:

```
maybeHead :: List a -> Maybe a
maybeHead (Cons x xs) = Just x
maybeHead Nil = Nothing
```

Pokud se dva výrazy Haskellu M a N rovnají vzhledem k redukci Haskellu H a η, pak se rovnají i jejich modely, tedy $||M|| = ||N||$. Toho můžeme využít při dokazování rovnosti modelů výrazů. Například protože

$$\texttt{maybeHead (Cons 1 Nil)} =_{H\eta} \texttt{Just 1}$$

pak nutně

$$||\texttt{maybeHead (Cons 1 Nil)}|| = ||\texttt{Just 1}||.$$

Obrácená implikace neplatí. Jako protipříklad vezměmě

$$||\texttt{\textbackslash\ x -> x * 2}|| = ||\texttt{\textbackslash\ x -> x + x}|| = \{(z, 2z) \mid z \in \mathbb{Z}\},$$

ale

$$\texttt{\textbackslash\ x -> x * 2} \neq_{H\eta} \texttt{\textbackslash\ x -> x + x},$$

protože se jedná o dvě různé normální formy.

3 Teorie kategorií

Kategorie se skládá z

1. *objektů* $a, b, c, \ldots$
2. a *šipek* (také *morfismů*) $f, g, h, \ldots$

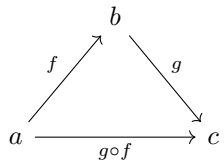
Na objektech a šipkách jsou zavedeny následující čtyři operace.

1. *Doména* přiřazuje každé šipce f objekt $a = \mathrm{dom}(f)$ nazývaný *zdroj* šipky f.
2. *Kodoména* přiřazuje každé šipce f objekt $b = \mathrm{cod}(f)$ nazývaný *cíl* šipky f.

Fakt, že šipka f má zdroj a a cíl b značíme: $f\colon a \to b$ nebo $a \xrightarrow{f} b$.

3. *Jednotka* přiřazuje objektu a šipku $1_a\colon a \to a$.
4. *Skládání* přiřazuje páru šipek (f, g), kde $\mathrm{dom}(g) = \mathrm{cod}(f)$, šipku $g \circ f$ takovou, že $g \circ f\colon \mathrm{dom}(f) \to \mathrm{cod}(g)$.

Skládání lze vyjádřit diagramem:

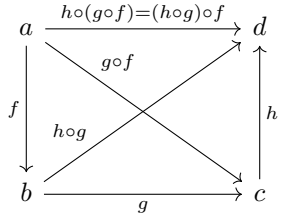


Operace musí splňovat následující dva aiomy.

Asociativita. Pro libovolné objekty a šipky v postavení $a \xrightarrow{f} b \xrightarrow{g} c \xrightarrow{h} d$ musí platit

$$h \circ (g \circ f) = (h \circ g) \circ f.$$

Předchozí rovnost můžeme graficky vyjádřit následujícím diagramem. Vrcholy označují objekty a hrany šipky.

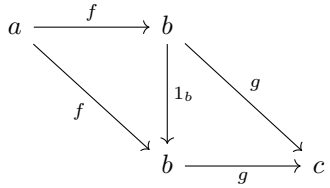


Řekneme, že diagram *komutuje*, jestliže pro každé dva vrcholy a a b a každé dva sledy P_1 a P_2 z vrcholu a do vrcholu b platí, že složení šipek na sledu P_1 je ta samá šipka jako složení šipek na sledu P_2. Předchozí diagram komutuje.

Jednotkový zákon. Pro každé dvě šipky $f\colon a \to b$ a $g\colon b \to c$ platí

$$1_b \circ f = f \text{ a } g \circ 1_b = g.$$

Neboli následující diagram komutuje.



Příkladem je kategorie množin **Set**, kde objekty jsou množiny a šipky jsou zobrazení. Zobrazení $f : X \to Y$ uvažované dohromady s doménou X a kodoménou Y je šipka se zdrojem X a cílem Y. Jednotka 1_X na množině X jeidentické zobrazení na X. Operace skládání šipek $\circ$ je klasické skládání zobrazení, tedy pro šipky $f : X \to Y$ a $g : Y \to Z$ je $g \circ f(x) = g(f(x))$ pro $x \in X$.

Pro množinu B atomických typů označíme $\mathbf{Lam}(B)$ kategorii, kde objekty jsou typy jednoduše typovaného lambda kalkulu vytvořené z B. Šipky z σ do τ odpovídají výrazům jednoduše typovaného lambda kalkulu typu $\sigma \to \tau$. Šipku můžeme zadat výrazem čistého lambda kalkulu společně s jeho možným typem. Například $\lambda x.x : b \to b$ je šipka $\lambda x^b.x^b$. Mezi výrazy shodného typu, které se rovnají v teorii $\lambda\eta$, nerozlišujeme. Všechny odpovídají stejné šipce. Jednotka objektu σ je výraz $\lambda x.x : \sigma \to \sigma$. Složení šipek $f : \sigma_1 \to \sigma_2$ a $g : \sigma_2 \to \sigma_3$ definujeme jako výraz $g \circ f = \lambda x.g(fx) \colon \sigma_1 \to \sigma_3$. Dokážeme jednotkový zákon. Pro $f \colon \sigma \to \tau$ máme

$$1_\tau \circ f = \lambda x.1_\tau(fx) = \lambda x.(\lambda y.y)(fx) = \lambda x.fx = f,$$
$$f \circ 1_\sigma = \lambda x.f(1_\sigma(x)) = \lambda x.f((\lambda y.y)x) = \lambda x.fx = f.$$

Dokážeme asociativitu. Pro $f \colon \sigma_1 \to \sigma_2$, $g \colon \sigma_2 \to \sigma_3$ a $h \colon \sigma_3 \to \sigma_4$ máme

$$h \circ (g \circ f) = h \circ (\lambda x.g(fx)) = \lambda y.h((\lambda x.g(fx))y) = \lambda y.h(g(fy)),$$
$$(h \circ g) \circ f = \lambda x.(h \circ g)(fx) = \lambda x.(\lambda y.h(gy))(fx) = \lambda x.h(g(fx)).$$

Množina P je *předuspořádána*, jestliže je na ní dána reflexivní a tranzitivní relace R. Každou předuspořádanou množinu můžeme chápat jako kategorii, kde objekty jsou prvky množiny P a každému páru $(p, q) \in R$ odpovídá šipka $p \to q$.

Pro program výše omezeného Haskellu P můžeme uvažovat kategorii $\mathbf{Hask}(P)$. Objekty kategorie budou uzavřené monotypy definované programem. Například objekty v programu:

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

jsou typy `Integer`, `List Integer`, `(List Integer) -> Integer`, ... Šipky odpovídají výrazům Haskellu funkčních typů. Konkrétně výraz *`expr :: type1 -> type2`* je šipkou z *`type1`* do *`type2`*. Například

```
(+) :: Integer -> (Integer -> Integer)
```

je šipkou se zdrojem `Integer` a cílem `Integer -> Integer`. Výraz uvažujeme včetně jeho monotypu. Výrazům shodného typu se stejným modelem odpovídá stejná šipka. Například výrazy

```
(\ x -> x + x) :: Integer -> Integer
```

a

```
(\ x -> 2 * x) :: Integer -> Integer
```

odpovídají stejné šipce. Model obou výrazů je funkce $\{(z, 2z) \mid a \in Z\}$.

Jednotka na typu *`type`* je přímo funkce identity

$1_{\texttt{\textit{type}}} = $ `\ x -> x :: type -> type`.

Například jednotka objektu `Integer` přiřadí šipku

```
\ x -> x :: Integer -> Integer
```

Složení šipek *`expr1 :: type1 -> type2`* a *`expr2 :: type2 -> type3`* je šipka

`\ x -> expr2 (expr1 x) :: type1 -> type3`

Například složení šipky

```
(+) 2 :: Integer -> Integer
```

a šipky

```
(*) 2 :: Integer -> Integer
```

je šipka

```
\ x -> (*) 2 ((+) 2 x) :: Integer -> Integer
```

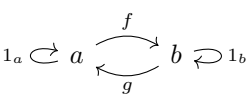
Do programu dodáme definice:

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

(.) :: (b -> c) -> (a -> b) -> (a -> c)
(.) g f x = g (f x)
```

Pak $1_{\texttt{\textit{type}}}$ = `id :: type -> type` a složení šipek *`expr1`* a *`expr2`* je šipka *`expr2 . expr1 :: type1 -> type3`*. V kategorii $\mathbf{Hask}(P)$ budeme program P vynechávat, protože bude zadán kontextem. Program P budeme postupně definovat.

Vezměme šipky $f\colon a \to b$ a $g\colon b \to a$ nějaké kategorie. Pokud $f \circ g$ je jednotka na b a $g \circ f$ je jednotka na a, pak říkáme, že g je *inverzní* k f. Vyjádřeno komutujícím diagramem:



Pokud inverzní šipka existuje, pak je jediná. Dokážeme si to. Nechť $h, g\colon b \to a$ jsou šipky inverzní k $f\colon a \to b$. Pak máme $g = 1_a \circ g = (h \circ f) \circ g = h \circ (f \circ g) =$

$h \circ 1_b = h$. Tedy $h = b$. Proto pokud inverzní šipka existuje, tak ji značíme $g = f^{-1}$.

Například v kategorii množin mají inverzi pouze bijekce. Konkrétně inverzní šipka k bijekci $f\colon X \to Y$ je inverzní zobrazení $f^{-1}\colon Y \to X$.

Definujme:

```
flip :: (a -> b -> c) -> (b -> a -> c)
flip f y x = f x y
```

Pak je šipka

```
(+) 2 :: Integer -> Integer
```

je inverzní k

```
flip (-) 2 :: Integer -> Integer
```

Šipku $f\colon a \to b$ nezveme *izomorfizmem*, jestliže má inverzní šipku. V takovém případě říkáme, že objekty a a b jsou izomorfní. Například v kategorii množin jsou izomorfní každé dvě množiny, které mají stejnou kardinalitu. Speciálně pro konečné množiny dostáváme, že jsou izomorfní každé dvě konečné množiny se stejným počtem prvků.

Definujme:

```
data Two = C1 | C2
data Bool = True | False
```

Typy `Bool` a `Two` jsou izomorfní.

Objekt nazveme *iniciální*, jestliže z něj vede právě jedna šipka do každého objektu.

V kategorii množin je prázdná množina iniciální objekt. Zvolme množinu X, pak existuje právě jedno zobrazení $f\colon \emptyset \to X$ a to $f = \emptyset$.

Definujme:

```
data Void
```

Platí $||$`Void`$|| = \emptyset$. Typ `Void` je iniciální objekt v kategorii **Hask**.

Šipku z `Void` do libovolného typu definujeme funkcí:

```
absurd :: Void -> a
absurd x = case x of {}
```

Poznamenejme, že $||$`case x of {}`$|| = \emptyset$.

Každé dva iniciální prvky jsou izomorfní. Nechť a a b jsou iniciální prvky. Protože a je iniciální prvek, musí existovat šipka $f : a \to b$. Podobně musí existovat šipka $g\colon b \to a$. Složení šipek $g \circ f$ je šipka $a \to a$, ale z toho, že a je iniciální, plyne, že $g \circ f = 1_a$. Podobně ukážeme, že $f \circ g = 1_b$.

Říkáme také, že iniciální objekt je až na izomorfizmus jedinečný.

Pro kategorii $\mathcal{C}$ definujeme *duální* kategorii $\mathcal{C}^{\mathrm{op}}$ následovně.

1. Objekty $\mathcal{C}$ jsou objekty $\mathcal{C}^{\mathrm{op}}$.
2. Pokud $f\colon a \to b$ v kategorii $\mathcal{C}$, pak $f\colon b \to a$ v kategorii $\mathcal{C}^{\mathrm{op}}$.
3. Identity v kategorii $\mathcal{C}$ jsou stejné jako identity v kategorii $\mathcal{C}^{\mathrm{op}}$.
4. Pokud $h = g \circ f$ v $\mathcal{C}$, pak $h = f \circ g$ v $\mathcal{C}^{\mathrm{op}}$.

Objekt v kategorii $\mathcal{C}$ nezveme *terminální*, jestliže je iniciální v kategorii $\mathcal{C}^{\mathrm{op}}$. Tedy objekt je terminální, jestliže do něj vede z každého objektu právě jedna šipka. Pokud terminální objekt existuje, je až na izomorfizumus jedinečný.V kategorii množin je libovolná jednoprvková množina terminální objekt. Vezměme například $\{\emptyset\}$ a libovolnou množinu X, pak existuje právě jedno zobrazení $f\colon X \to \{\emptyset\}$ a to zobrazení $f(x) = \emptyset$.

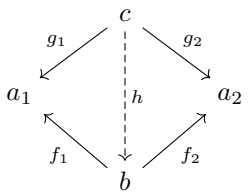
Definujeme terminální objekt v kategorii **Hask**:

```
data Singleton = Star
```

Šipky do terminálního objektu:

```
unit :: a -> Singleton
unit _ = Star
```

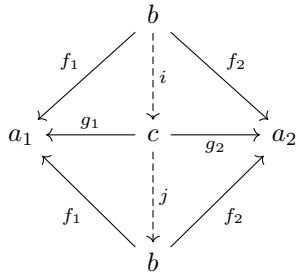
Buďte a_1 a a_2 objekty. Potom objekt b spolu s šipkami $f_1 : b \to a_1$ a $f_2\colon b \to a_2$ nezýváme *součinem* (*produktem*) objektů a_1 a a_2, jestliže pro každý objekt c spolu s šipkami $g_1\colon c \to a_1$ a $g_2\colon c \to a_2$ existuje jediná šipka $h\colon c \to b$ tak, že $g_1 = f_1 \circ h$ a $g_2 = f_2 \circ h$. Šipky f_1 a f_2 se nazývají *projekce*.



Přerušovaná šipka značí, že mezi objekty existuje jediná šipka, pro kterou diagram komutuje.

Pokud součin objektů a_1 a a_2 existuje, pak je až na izomorfizmus jediný. Tvrzení si dokážeme. Nechť objekt b s projekcemi $f_1 : b \to a_1$ a $f_2 : b \to a_2$ a objekt c s projekcemi $g_1\colon c \to a_1$ a $g_2\colon c \to a_2$ jsou dva součiny objektů a_1 a a_2. Protože c je součin, existuje jediná šipka $i\colon b \to c$, tak že $g_1 \circ i = f_1$ a $g_2 \circ i = f_2$. Podobně z toho, že b je součin, plyne, že existuje jediná šipka $j : c \to b$ která

splňuje, že $f_1 \circ j = g_1$ a $f_2 \circ j = g_2$.



Pro složenou šipku $j \circ i$ platí, že $f_1 \circ (j \circ i) = (f_1 \circ j) \circ i = g_1 \circ i = f_1$ a podobně $f_2 \circ (j \circ i) = f_2$. Protože $f_1 \circ 1_b = f_1$ a $f_2 \circ 1_b = f_2$, z jednoznačnosti plyne, že $j \circ i = 1_b$. Podobně můžeme ukázat, že $i \circ j = 1_c$. Tedy $i \colon b \to c$ je izomorfizmus.

Součin objektů a_1 a a_2 značíme $a_1 \times a_2$, příslušné projekce značíme $\pi_1 : a_1 \times a_2 \to a_1$ a $\pi_2 : a_1 \times a_2 \to a_2$.

Součinem množin X a Y v kategorii množin je kartézský součin množin $X \times Y$ spolu s projekcemi $\pi_1(x, y) = x$ a $\pi_2(x, y) = y$. Skutečně, vezměme libovolnou množinu Z a zobrazení $f_1 \colon Z \to X$ a $f_2 \colon Z \to Y$, pak pro zobrazení $g \colon Z \to X \times Y$ definované $g(z) = (f_1(z), f_2(z))$ platí, že $\pi_1(h(z)) = \pi_1(f_1(z), f_2(z)) = f_1(z)$ a podobně $\pi_2(h(z)) = f_2(z)$. Nechť $h \colon Z \to X \times Y$ je zobrazení, které splňuje, že $\pi_1 \circ h = f_1$ a $\pi_2 \circ h = f_2$. Pro každé $z \in Z$ platí, že $h(z) = (\pi_1(h(z)), \pi_2(h(z))) = (f_1(z), f_2(z)) = g(z)$. Tedy $h = g$.

Definujeme:

```
data Product a b = Pair a b

fst :: Product a b -> a
fst (Pair x y) = x

snd :: Product a b -> b
snd (Pair x y) = y
```

Pak pro typy *`type1`* a *`type2`* je typ `(Product` *`type1 type2`*`)` spolu s projekcemi

```
fst :: Product type1 type2 -> type1
```

a

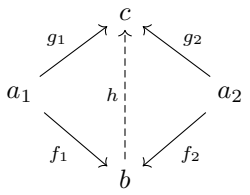
```
snd :: Product type1 type2 -> type2
```

součinem typů *`type1`* a *`type1`*.

Šipku h z definice součinu můžeme definovat:

```
factorizer :: (c -> a) -> (c -> b) -> c -> Product a b
factorizer f g z = Pair (f z) (g z)
```

Duálním pojmem k součinu objektů je *součet* (*koprodukt*) objektů. Přesněji pro objekty a_1 a a_2 je objekt b spolu s šipkami $f_1 : a_1 \to b$ a $f_2 : a_2 \to b$ *součtem* objektů a_1 a a_2, jestliže pro každý objekt c spolu s šipkami $g_1 : a_1 \to c$ a $f_2 : a_2 \to c$ existuje jediná šipka $h : b \to c$ taková, že diagram



komutuje. Šipky $f_1 : a_1 \to b$ a $f_2 : a_2 \to b$ nazýváme *injekce*. Pokud součet objektů a_1 a a_2 existuje, je až na izomorfizmus jediný a značíme jej $a_1 + a_2$. Injekce značíme $i_1 : a_1 \to a_1 + a_2$ a $i_2 : a_2 \to a_1 + a_2$.

V kategorii množin je součtem množin X a Y jejich disjunktní sjednocení spolu s příslušnými injekcemi. Přesněji $X + Y = \{(1, x) \mid x \in X\} \cup \{(2, y) \mid y \in B\}$. Injekce definujeme, tak že $i_1(x) = (1, x)$ a $i_2(y) = (2, y)$ pro $x \in X$ a $y \in Y$.

Definujeme:

```
data Either a b = Left a | Right b
```

Pro typy *`type1`* a *`type2`* je typ `(Either` *`type1 type2`*`)` spolu s injekcemi:

```
Left :: type1 -> Either type1 type2
Right :: type2 -> Either type1 type2
```

součet typů *`type1`* a *`type2`*. Šipka h z definice součtu objektů:

```
factorizer' :: (a -> c) -> (b -> c) -> Either' a b -> c
factorizer' i j (Left' a) = i a
factorizer' i j (Right' b) = j b
```

Součet typu s terminálním typem:

```
data Maybe a = Just a | Nothing
```

Použití:

```
maybeHead :: List a -> Maybe a
maybeHead (Cons x xs) = Just x
maybeHead Nil = Nothing
```

Použité zdroje:

1. S. Mac Lane: Categories for the Working Mathematician
2. Steve Awodey: Category theory
3. Bartosz Milewski: Category Theory for Programmers
4. Jan Starý: Úvod do teorie kategorií