Monády ve funkcionálním programování

LKFP přednáška 10, jaro 2024/25

Jan Laštovička

16. dubna 2025

1 Rozšíření omezeného modelu Haskellu

Mírně rozšíříme omezený model Haskellu z předchozí přednášky.

Model rekurzivních datových struktur musíme definovat speciálně. Například model pro seznamy:

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

definujeme:

$$||\texttt{List } \textit{type}|| = \bigcup_{i=0}^{\infty} L_i$$

kde

$$L_0 = \{\texttt{Nil}\},$$
$$L_i = \{(\texttt{Cons}, x, l) \mid x \in ||\textit{type}|| \text{ a } l \in L_{i-1}\}, \text{ pro } i > 0.$$

Například:

$$||\texttt{List Boolean}|| = \{\texttt{Nil}, (\texttt{Cons}, \texttt{True}, \texttt{Nil}), (\texttt{Cons}, \texttt{False}, \texttt{Nil}), (\texttt{Cons}, \texttt{True}, (\texttt{Cons}, \texttt{True}, \texttt{Nil})) \ldots\}$$

Rekurzivní jména budeme modelovat zvlášť. Například `folder`:

```
foldr :: (a -> b -> b) -> b -> List a -> b
foldr f z Nil = z
foldr f z (Cons x xs) = f x (foldr f z xs)
```

Vezměme uzavřené monotypy σ a τ. Pak

$$||\texttt{folder}^{(\sigma\to\tau\to\tau)\to\tau\to\texttt{List } \sigma\to\tau}|| = F^l = \{(f, F^l_f) \mid f \in ||\sigma||^{||\tau||^{||\tau||}}\},$$

kde

$$F^l_f = \{(x, F^l_{fx}) \mid x \in ||\tau||\},$$
$$F^l_{fx} = \{(\texttt{Nil}, x)\} \cup \{((\texttt{Cons}, a, b), f(a)(F^l_{fx}(b))) \mid a \in ||\sigma|| \text{ a } b \in ||\texttt{List } \sigma||\}$$

Například pro:

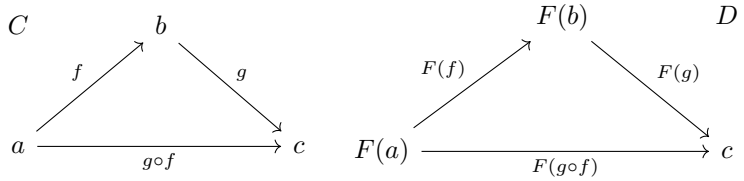
```
sum :: List Integer -> Integer
sum = foldr (+) 0
```

je $||sum|| = \{(\texttt{Nil}, 0)\} \cup \{((\texttt{List}, a, b), a + F^l_{+0}(b))\}$

2 Funktory

Funktor $F : C \to D$ z kategorie C do kategorie D je operace, která každému objektu a z kategorie C přiřadí objekt $F(a)$ z kategorie D a každé šipce f z kategorie C přiřadí šipku $F(f)$ z kategorie D, tak že

1. pro $f : a \to b$ z C je $F(f) : F(a) \to F(b)$,
2. pro objekt a z C je $F(1_a) = 1_{F(a)}$,
3. pro $a \xrightarrow{f} b \xrightarrow{g} c$ z C je $F(g \circ f) = F(g) \circ F(f)$.



Funktory lze přirozeně skládat. Přesněji, pokud $F : C \to D$ a $G : D \to E$ jsou funktory, pak $G \circ F : C \to E$ je funktor, který objektu a kategorie C přiřadí objekt $G(F(a))$ kategorie E a šipce $f : a \to b$ kategorie C přiřadí šipku $G(F(f))$ kategorie E. Na každé kategorii máme jednotkový funktor $1_C : C \to C$, který objekty i šipky zobrazuje na je samé. Tedy máme kategorii, kde objekty jsou kategorie a funktory šipky mezi nimi.

Například pokud vykládáme předuspořádané množiny (X, R) a (Y, S) jako kategorie, pak každé zobrazení $F : X \to Y$ zachovávající předuspořádání můžeme chápat jako funktor z X do Y. Objektu $x \in X$ přiřadíme objekt $F(x)$ a šipce $(x, y) \in R$ funktor přiřadí šipku $(F(x), F(y)) \in S$. Zobrazení F zachovává předuspořádání, jestliže pro každé $(x, y) \in X$ platí, že

$$\text{pokud } (x, y) \in R, \text{ pak } (F(x), F(y)) \in S.$$

Pro množinu atomických typů B a zobrazení přiřazující atomickému typu b množinu $||b||$ máme funktor z $\mathbf{Lam}(B)$ do $\mathbf{Set}$, který typu τ přiřadí množinu $||\tau||$ a šipce $M : \sigma \to \tau$ přiřadí zobrazení $||M|| : ||\sigma|| \to ||\tau||$.

Podobně pro program P máme funktor z $\mathbf{Hask}(P)$ do $\mathbf{Set}$, který uzavřenému monotypu `type` přiřadí množinu $||$`type`$||$ a výrazu Haskellu `expr :: type1 -> type2` funkci $||$`expr`$|| : ||$`type1`$|| \to ||$`type2`$||$.

Pokud atomickým typům B přiřadíme uzavřené monotypy, obdržíme funktor z kategorie $\mathbf{Lam}(B)$ do $\mathbf{Hask}(P)$, kde lambda výrazu $M : \sigma \to \tau$ přiřadíme odpovídající uzavřený výraz v Haskellu odpovídajícího typu. Například pokud

$B = \{0\}$ a typu `0` přiřadíme typ `Integer`, pak $\lambda x.x \colon 0 \to 0$ zobrazíme na výraz `\ x -> x :: Integer -> Integer`.

Funktor z kategorie do sebe samé se nazývá *endofunktor*. Například na kategorii množin máme endofunktor F, který množině přiřadí potenční množinu, tedy $F(X) = \mathcal{P}(X)$ a zobrazení $f : X \to Y$ zobrazení $F(f) : \mathcal{P}(X) \to \mathcal{P}(Y)$ danou $F(f)(A) = \{f(x) \mid x \in A\} \subseteq Y$ pro $A \subseteq X$. Další endofunktor F na kategorii množin je dán tak, že množině X přiřadí druhou kartézskou mocninu $F(X) = X \times X$ a zobrazení $f \colon X \to Y$ přiřadí zobrazení $F(f) = f \times f \colon X \times X \to Y \times Y$, které je dáno $f \times f(x_1, x_2) = (f(x_1), f(x_2))$.

Definujme třídu:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Každý unární typový konstruktor C (například `Maybe`) zobrazuje uzavřený monotyp τ na $C\ \tau$. Například `Integer` zobrazíme na `Maybe Integer`. Pro definování endofunktoru kategorie **Hask**(P) stačí doplnit definici `fmap` určující, jak šipce $f \colon \tau_1 \to \tau_2$ přiřadit šipku $C\tau_1 \to C\tau_2$. Například:

```
instance Functor Maybe where
  fmap :: (a -> b) -> Maybe a -> Maybe b
  fmap _ Nothing = Nothing
  fmap f (Just x) = Just (f x)
```

nebo:

```
instance Functor List where
  fmap :: (a -> b) -> List a -> List b
  fmap _ Nil = Nil
  fmap f (Cons x xs) = Cons (f x) (fmap f xs)
```

Podle typu argumentů se vybere vhodná definice jména `fmap`. Například:

```
>>> fmap succ (Just 1)
Just 2
>>> fmap succ Nothing
Nothing
>>> fmap succ (Cons 1 (Cons 2 Nil))
Cons 2 (Cons 3 Nil)
```

nebo dokonce:

```
>>> fmap (fmap succ) (Cons (Just 1) Nil)
Cons (Just 2) Nil
```

Můžeme napsat funkci, která vždy vybere vhodný funktor:

```
fSucc :: Functor f => f Integer -> f Integer
fSucc = fmap succ
```

například:

```
>>> fSucc (Just 1)
Just 2
```

3 Přirozené transformace

Pro dva funktory $F, G \colon C \to D$ je *přirozená transformace* $\vartheta \colon F \to G$ funktoru F do funktoru G systém šipek přiřazující každému objektu a kategorie C šipku

$$\vartheta_a \colon F(a) \to G(a)$$

kategorie D, pokud pro každou šipku $f \colon a \to b$ z kategorie C následující diagram komutuje v kategorii B.

$$\begin{array}{ccc} F(a) & \xrightarrow{\vartheta_a} & G(a) \\ {\scriptstyle F(f)}\downarrow & & \downarrow{\scriptstyle G(f)} \\ F(b) & \xrightarrow[\vartheta_b]{} & G(b) \end{array}$$

Pro objekt a kategorie C se šipka $\vartheta_a : F(a) \to G(a)$ nazývá *komponenta* a přirozené transformace ϑ.

Například pro funktory $1_{\mathbf{Set}}, F : \mathbf{Set} \to \mathbf{Set}$, kde F je funktor kartézské druhé mocniny definovaný výše, máme přirozenou transformaci $\vartheta \colon 1_{\mathbf{Set}} \to F$ s komponentami

$$\vartheta_X \colon 1_{\mathbf{Set}}(X) \to F(X) = X \to X \times X$$

určenými $\vartheta_X(x) = (x, x)$ pro $x \in X$. Pro ověření musíme dokázat, že pro funkci $f \colon X \to Y$ diagram

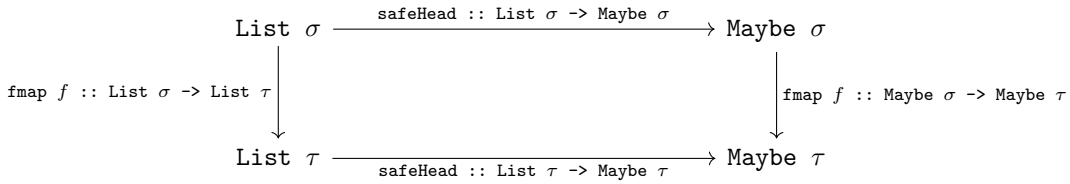
$$\begin{array}{ccc} X & \xrightarrow{\vartheta_X} & X \times X \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle f \times f} \\ Y & \xrightarrow[\vartheta_Y]{} & Y \times Y \end{array}$$

komutuje. To je pravda, protože $f \times f(\vartheta_X(x)) = f \times f(x \times x) = (f(x), f(x)) = \vartheta_Y(f(x))$ pro každé $x \in X$.

V kategorii Haskellu odpovídá přirozená transformace z C_1 do C_2 polymorfním funkcím typu C_1 `a -> `C_2 `a`. Například:

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

je přirozená transformace z `List` do `Maybe`. Musíme dokázat komutativitu diagramu:



Dokážeme si to rozborem případů. Pro `Nil` máme:

`safeHead (fmap` f `Nil)` $=$ `safeHead Nil` $=$ `Nothing`

a

`fmap` f `(safeHead Nil)` $=$ `fmap` f `Nothing` $=$ `Nothing`

Pro `Cons` a b máme:

`safeHead (fmap` f `(Cons` a b`))` $=$ `safeHead (Cons (`f a`) (map` f b`))`
$=$ `Just (`f a`)`

a

`fmap` f `(safeHead (Cons` a b`))` $=$ `fmap` f `(Just` a`)` $=$ `Just (`f a`)`

Tedy pro libovolné x `::` `List` σ platí:

`safeHead (fmap` f x`)` $=$ `fmap` f `(safeHead` x`)`

Dostáváme:

`\ x -> safeHead (fmap` f `x)` $=$ `\ x -> fmap` f `(safeHead x)`

Tedy:

`safeHead . (fmap` f`)` $=$ `fmap` f `. safeHead`

Kategorie funktorů z kategorie C do kategorie D má za objekty funktory z C do D a šipky z funktoru F do funktoru G jsou přirozené transformace z F do G. Jednotka funktoru F je přirozená transformace $1_F \colon F \to F$ s komponentami

$$(1_F)_a = 1_{F(a)} \colon F(a) \to F(a).$$

Složení přirozených transformací $F \xrightarrow{\vartheta} G \xrightarrow{\phi} H$ je přirozená transformace s komponentami

$$(\phi \circ \vartheta)_a = \phi_x \circ \vartheta_a.$$

Přirozené transformace lze skládat s funktory následujícím způsobem. Vezměme přirozenou transformaci $\vartheta \colon F \to G$ mezi funktory $F, G \colon C \to D$. Složením funktoru $H \colon D \to E$ s přirozenou transformací ϑ obdržíme přirozenou transformaci $H\vartheta \colon H \circ F \to H \circ G$ s komponentami

$$(H\vartheta)_a = H(\vartheta_a) : H(F(a)) \to H(G(a)),$$

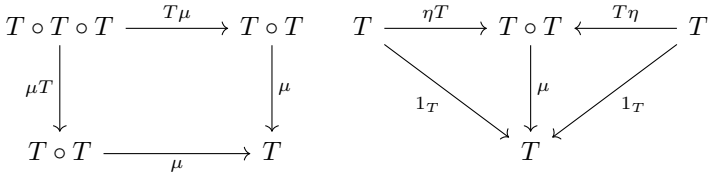
kde a je objekt kategorie C. Složením přirozené transformace ϑ s funktorem $K \colon B \to C$ obdržíme přirozenou transformaci $\vartheta K \colon F \circ K \to G \circ K$ s komponentami

$$(\vartheta K)_a = \vartheta_{K(a)} \colon F(K(a)) \to G(K(a)),$$

kde a je objekt kategorie B.

4 Monády

Monáda v kategorii C je endofunktor $T : C \to C$, pro který jsou dány dvě přirozené transformace $\eta \colon 1_C \to T$ a $\mu \colon T \circ T \to T$, pro které následující dva diagramy komutují.



Například v kategorii množin máme monádu danou endofunktorem potenční množiny $T : \mathbf{Set} \to \mathbf{Set}$ (je definován výše), kde přirozená transformace $\eta : 1_{\mathbf{Set}} \to T$ má komponenty zobrazení

$$\eta_X \colon X \to \mathcal{P}(X)$$

definované $\eta_X(x) = \{x\}$ pro $x \in X$ a přirozená transformace $\mu \colon T \circ T \to T$ má komponenty zobrazení

$$\mu_X \colon \mathcal{P}(\mathcal{P}(X)) \to \mathcal{P}(X)$$

definované $\mu_X(\alpha) = \bigcup \alpha$ pro $\alpha \in \mathcal{P}(\mathcal{P}(X))$. Dokážeme $\mu \circ (\mu T) = \mu \circ (T\mu)$. Pro množinu X chceme dokázat rovnost

$$(\mu \circ (\mu T))_X = (\mu \circ (T\mu))_X \colon \mathcal{P}(\mathcal{P}(\mathcal{P}(X))) \to X.$$

Pro $\alpha \in \mathcal{P}(\mathcal{P}(\mathcal{P}(X)))$ máme

$$\begin{aligned}(\mu \circ (\mu T))_X(\alpha) &= (\mu_X \circ (\mu T)_X)(\alpha) = \mu_X((\mu T)_X(\alpha)) = \mu_X(\mu_{T(X)}(\alpha)) \\ &= \mu_X(\mu_{\mathcal{P}(X)}(\alpha)) = \mu_X\left(\bigcup \alpha\right) = \bigcup\left(\bigcup \alpha\right)\end{aligned}$$

a

$$\begin{aligned}(\mu \circ (T\mu))_X(\alpha) &= (\mu_X \circ (T\mu)_X)(\alpha) = \mu_X((T\mu)_X(\alpha)) = \mu_X(T(\mu_X)(\alpha)) \\ &= \mu_X\left(\left\{\bigcup \beta \mid \beta \in \alpha\right\}\right) = \bigcup\left\{\bigcup \beta \mid \beta \in \alpha\right\} \\ &= \bigcup\left(\bigcup \alpha\right).\end{aligned}$$

Ukázali jsme, že první diagram komutuje. Dokažme $\mu \circ (\eta T) = 1_T \colon T \to T$. Pro množinu X chceme ukázat, že platí

$$(\mu \circ (\eta T))_X = (1_T)_X = (\mu \circ (T\eta))_X \colon \mathcal{P}(X) \to \mathcal{P}(X).$$

Vezměme $\alpha \in \mathcal{P}(X)$. Dostáváme

$$\begin{aligned}(\mu \circ (\eta T))_X(\alpha) &= (\mu_X \circ (\eta T)_X)(\alpha) = \mu_X((\eta T)_X(\alpha)) = \mu_X(\eta_{T(X)}(\alpha)) \\ &= \mu_X(\{\alpha\}) = \bigcup\{\alpha\} = \alpha\end{aligned}$$

a

$$(1_T)_X(\alpha) = 1_{T(X)}(\alpha) = \alpha$$

a

$$\begin{aligned}(\mu \circ (T\eta))_X(\alpha) &= (\mu_X \circ (T\eta)_X)(\alpha) = \mu_X((T\eta)_X(\alpha)) = \mu_X(T(\eta_X)(\alpha)) \\ &= \mu_X(\{\eta_X(x) \mid x \in \alpha\}) = \mu_X(\{\{x\} \mid x \in \alpha\}) = \bigcup\{\{x\} \mid x \in \alpha\} \\ &= \alpha.\end{aligned}$$

Ukázali jsme, že druhý diagram komutuje.

Definujme v Haskellu třídu:

```
class Functor m => Monad m where
  join :: m (m a) -> m a
  return :: a -> m a
```

Monádu v Haskellu definujeme jako instanci třídy `Monad`, kde přirozené transformace `join` a `return` splňují podmínky monády:

```
join . fmap join = join . join
join . fmap return = id
join . return = id
```

Například:

```
instance Monad Maybe where
  return :: a -> Maybe a
  return = Just

  join :: Maybe (Maybe a) -> Maybe a
  join (Just x) = x
  join Nothing = Nothing
```

Chceme dokázat:

```
join . fmap join = join . join
```

kde oba výrazy jsou typu `Maybe (Maybe (Maybe` τ`)) -> (Maybe` τ`)` pro nějaký uzavřený monotyp τ. Vezměme proměnou x typu `Maybe (Maybe (Maybe` τ`))`. Dostáváme:

`(join . fmap join)` x = `join (fmap join` x`)`

a

`(join . join)` x = `join (join` x`)`

Rozborem případů pro typ `Maybe` dostáváme:

```
join (fmap join Nothing) = join Nothing = Nothing
```

a

```
join (join Nothing) = join Nothing = Nothing
```

a pro y je typu `(Maybe (Maybe` τ`)` dostáváme:

`join (fmap join (Just` y`)) = join (Just (join` y`)) = join` y

a

`join (join (Just` y`)) = join` y

Tím jsme dokázali požadovanou rovnost.

Chceme dokázat:

`join . fmap return` $=$ `id :: Maybe` τ `-> Maybe` τ

Vezměme x `:: Maybe` τ. Dostáváme:

$$\begin{aligned}\texttt{(join . fmap return)}\ x &= \texttt{join (fmap return}\ x\texttt{)} \\ &= \texttt{join (fmap Just}\ x\texttt{)}\end{aligned}$$

a `id` $x = x$. Rozebereme případy. Pro `Nothing` máme:

$$\texttt{join (fmap Just Nothing)} = \texttt{join Nothing} = \texttt{Nothing}$$

Pro `Just` y, kde y `::` τ máme:

$$\texttt{join (fmap Just (Just}\ y\texttt{))} = \texttt{join (Just (Just}\ y\texttt{))} = \texttt{Just}\ y$$

Dokážeme zbývající rovnost:

$$\texttt{join . return} = \texttt{id}$$

Pro x typu `Maybe` τ dostáváme:

$$\texttt{(join . return)}\ x = \texttt{join (return}\ x\texttt{)} = \texttt{join (Just}\ x\texttt{)} = x = \texttt{id}\ x.$$

Pro snadnější použití monády v programování definujeme funkci (někdy nazývanou `bind`):

```
(>>=) :: Monad m => m a1 -> (a1 -> m a2) -> m a2
ma >>= f = join (fmap f ma)
```

Ukázka použití:

```
safeSecondDouble :: List Integer -> Maybe Integer
safeSecondDouble l = safeTail l >>=
  (\ result -> safeHead result >>=
    (\ result -> return (result * 2)))
```

máme:

```
>>> safeSecondDouble (Cons 1 (Cons 2 Nil))
Just 4

>>> safeSecondDouble (Cons 1 Nil)
Nothing

>>> safeSecondDouble Nil
Nothing
```