Funkcionální programování

Ad-hoc polymorfismus

Petr Pudlák

21. prosince 2011

Osnova

Motivace

- Chceme definovat specifické implementace polymorfních funkcí různě pro různé typy.
- Například na různých typech bude různě definována rovnost či uspořádání.
- Příklad: $\mathit{sort} :: (\mathit{Ord}\ \alpha) \Rightarrow [\alpha] \rightarrow [\alpha]$
  Funkce *sort* tak není úplně polymorfní v proměnné α.
  Můžeme dosadit pouze takový typ, pro který umíme porovnávat hodnoty.

Typové třídy

- *Typová třída* je unární relace[1] na ADT.
- ADT, které patří do relace dané určitou typovou třídou, nazýváme její *instance*.
- Typová třída definuje, jaké funkce musí být definovány pro její instance.
- Typové třídy se v Haskellu dají aplikovat na typové výrazy různých druhů (nejen $*$). Ke každé typové třídě přísluší druh odpovídající odvozenému druhu její proměnné.

[1] Rozšíření Haskellu pak dovolují i n-ární relace, a další modifikace.

Příklady I

```
class Eq a where
  (≡) :: a → a → Bool
  (≢) :: a → a → Bool
class (Eq a) ⇒ Ord a where
  compare :: a → a → Ordering
  (<)     :: a → a → Bool
  (⩾)     :: a → a → Bool
  (>)     :: a → a → Bool
  (⩽)     :: a → a → Bool
  max     :: a → a → a
  min     :: a → a → a
class Bounded a where
  minBound :: a
  maxBound :: a
```

Příklady II

Netriviální druhy

```
class Functor f where
  fmap  :: (a → b) → f a → f b
class Monad m where
  (>>=) :: m a → (a → m b) → m b
  (>>)  :: m a → m b → m b
  return :: a → m a
  fail   :: String → m a
class Monad m ⇒ MonadPlus m where
  mzero :: m a
  mplus :: m a → m a → m a
class Category cat where
  id    :: cat a a
  (∘)   :: cat b c → cat a b → cat a c
```

Výchozí výrazy pro definované funkce

- V definici typových tříd mohou mít jednotlivé funkce výchozí definice.
- Ty mohou být vzájemně rekurzivní.
- Například:

$$
\begin{aligned}
&\textbf{class } \mathit{Eq}\ a \textbf{ where}\\
&\quad (\equiv) :: a \to a \to \mathit{Bool}\\
&\quad x \equiv y = \neg\,(x \not\equiv y)\\
&\quad (\not\equiv) :: a \to a \to \mathit{Bool}\\
&\quad x \not\equiv y = \neg\,(x \equiv y)
\end{aligned}
$$

Implementátor si pak může vybrat pro implementaci pouze jednu z těchto funkcí.
Pro zajímavost: Výše uvedené implementace lze zapsat i bez proměnných:

$$
\begin{aligned}
(\equiv) &= (\neg\circ) \circ (\not\equiv)\\
(\not\equiv) &= (\neg\circ) \circ (\equiv)
\end{aligned}
$$

Instance typových tříd

- Pro zvolený ADT pak definujeme, jak jsou funkce typové třídy implementovány pro tento ADT.
- Implementace může být opět vázána podmínkami na jiné typové třídy. Vznikají pak „implikace" tvaru: *Pokud je pro typ a definována rovnost, je definována rovnost i pro seznamy typu a.*
- Například:

```
instance Functor [] where
  f `fmap` []     = []
  f `fmap` (x:xs) = (f x) : (f `fmap` xs)
instance Eq a => Eq ([] a) where
  []     == []     = True
  (x:xs) == (y:ys) = (x == y) && (xs == ys)
  _      == _      = False
  -- funkce /= je implementovana pomoci == v definice tridy
```

Implementace slovníkem

- Poprvé tento způsob popsal [1].
- Při odvozování typů si v kontextu type checker udržuje informaci, které výrazy musí implementovat které třídy, aby bylo možno korektně použít jednotlivé funkce z typových tříd.
- Každá funkce deklarovaná v programu, jejíž typ nese podmínky typových tříd, bude překladačem transformována na funkci, která má pro každou takovou podmínku skrytý parametr, který v sobě nese implementace funkcí dané typové třídy.
- V místě, kde je funkce použita na konkrétní datové typy, se do parametrů doplní odpovídající implementace.

Implementace slovníkem – příklad

- Funkce $elem :: (Eq\ a) \Rightarrow a \rightarrow [a] \rightarrow Bool$ bude při překladu nahrazena funkcí:

$$
\begin{aligned}
&elem :: (a \rightarrow a \rightarrow Bool, a \rightarrow a \rightarrow Bool) \rightarrow a \rightarrow [a] \rightarrow Bool \\
&elem\ (eq, notEq) = elem' \\
&\quad \textbf{where}\ elem'\ \_\ [\,] \quad\quad = False \\
&\qquad\qquad elem'\ x\ (y : ys) = (x\ `eq`\ y) \vee (elem'\ x\ ys)
\end{aligned}
$$

Automatické odvození instancí algebraických datových typů

- U algebraických datových typů se v mnoha případech implementují typové třídy stále stejným způsobem.
- Haskell umožňuje automatické odvození instancí při definici těchto typových tříd: *Eq*, *Ord*, *Enum*, *Bounded*, *Show* a *Read*.
- Příklad:

```
data Maybe a = Nothing | Just a
  deriving (Eq, Show, Read)
```

Příklad – typové proměnné netriviálního druhu I

Pomocí typových tříd

```
import Data.Monoid
import Data.Foldable (Foldable, foldMap)
import qualified Data.Foldable as F

data TreeList t a = Nil | a :> (t (TreeList t a))
infixr 5 :>
```

Na instancích *Foldable* lze postupně obsah datové struktury poskládat do

Příklad – typové proměnné netriviálního druhu II

Pomocí typových tříd

jedné výsledné hodnoty. Jedna z (více možných) definic toto skládání definuje pomocí *Monoid*u:

```
instance Foldable t ⇒ Foldable (TreeList t) where
    -- foldMap :: (Monoid t) => (a -> t) -> TreeList t a -> m
  foldMap f = fm
    where
      fm Nil        = mempty
      fm (x :> xs) = (f x) `mappend` (foldMap fm xs)
```

Funkce, která počítá počet prvků seznamu za předpokladu, že větvící typ je *Foldable*:

```
count :: (Foldable t) ⇒ TreeList t a → Int
count = getSum ∘ foldMap (const (Sum 1))
```

Seznamy:

Příklad – typové proměnné netriviálního druhu III

Pomocí typových tříd

```
newtype Id a = Id {unid :: a}
  deriving (Eq, Show)
instance Foldable Id where
  foldMap f (Id x) = f x

type List a = TreeList Id a

exampleList = 1 :> Id (2 :> Id (3 :> Id Nil))
```

Binární stromy:

```
data Pair a = Pair {pairL :: a, pairR :: a}
  deriving (Eq, Show)
instance Foldable Pair where
  foldMap f (Pair x y) = f x `mappend` f y

type BinTree a = TreeList Pair a
```

Příklad – typové proměnné netriviálního druhu IV

Pomocí typových tříd

```
exampleBinTree =
  "a" :> Pair
    ("ab" :> (Pair Nil Nil))
    ("ac" :> (Pair
      Nil
      ("acd" :> Pair Nil Nil)
      )
    )
```

n-ární stromy (instance pro Foldable [] je definována v modulu Foldable):

```
type NTree a = TreeList [] a
exampleNTree =
  0.0 :>
    [-1.0 :> []
```

Příklad – typové proměnné netriviálního druhu V

Pomocí typových tříd

$$
\begin{aligned}
&, -0.5 :> [] \\
&, 0.5 :> [] \\
&, 1.0 :> [3 :> []] \\
&]
\end{aligned}
$$

Osnova

Doporučená četba

- http://www.haskell.org/haskellwiki/The_Monad.Reader/Issue3/Functional_Programming_vs_Object_Oriented_Programming

Literatura I

[1] P. Wadler and S. Blott. ?How to make ad-hoc polymorphism less ad hoc? In: *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1989, pp. 60–76. URL: http://homepages.inf.ed.ac.uk/wadler/topics/type-classes.html.