Haskell and Domain-Specific Languages

Haskell nejen pro informatiky

Otakar Smrž

Institute of Formal and Applied Linguistics
Faculty of Mathematics and Physics
Charles University in Prague

otakar.smrz@mff.cuni.cz

https://wiki.ufal.ms.mff.cuni.cz/courses:pfl080

Part I

Cabal

Cabal

http://www.haskell.org/cabal/

Cabal

http://www.haskell.org/cabal/

```
name:               Encode-Exec
version:            0.9
license:            GPL
license-file:       LICENSE
extra-source-files: Encode/Main.hs, Encode/Setup.hs,
                    Encode/Encode-Exec-Encode.cabal,
                    Decode/Main.hs, Decode/Setup.hs,
                    Decode/Encode-Exec-Decode.cabal,
                    INSTALL, Setup.PL
copyright:          2007
author:             Otakar Smrz
maintainer:         otakar.smrz mff.cuni.cz
homepage:           http://ufal.mff.cuni.cz/~smrz/
category:           Various
build-depends:      Cabal, base, mtl
synopsis:           Executable 'methods' of Encode
```

```
executable:         encode
main-is:            Encode/Main.hs
hs-source-dirs:     ., ..
ghc-options:        -fglasgow-exts

executable:         decode
main-is:            Decode/Main.hs
hs-source-dirs:     ., ..
ghc-options:        -fglasgow-exts
```

Part II

Haddock

Haddock

http://www.haskell.org/haddock/

Haddock

http://www.haskell.org/haddock/

```
description:    The "Encode" library provides a unified interface for converting
                strings from different encodings into a common representation,
                and vice versa. It defines the 'Encode.Encoding' type class,
                whose methods include 'encode' and 'decode'.
                .
                The "Encode\/Main.hs" and "Decode\/Main.hs" programs mimick the
                fuction calls to 'encode' and 'decode', respectively, with the
                following command-line synopsis:
                .
                >    decode ArabTeX < decode.d | encode Buckwalter > encode.d
                >
                >    decode MacArabic < data.MacArabic > data.UTF8
                >
                >    encode WinArabic < data.UTF8 > data.WinArabic
                .
                This software is published under the /GNU General Public License/.
                Note the copyright and license details in the headers of the files,
                and see "LICENSE" distributed with this package.
                .
                "Encode" <http://ufal.mff.cuni.cz/~smrz/Encode/doc/html/>
```

Haddock

```
-- |
-- Module      :  Control.Monad
-- Copyright   :  (c) The University of Glasgow 2001
-- License     :  BSD-style (see the file libraries/base/LICENSE)
--
-- Maintainer  :  libraries@haskell.org
-- Stability   :  provisional
-- Portability :  portable
--
-- The 'Functor', 'Monad' and 'MonadPlus' classes,
-- with some useful operations on monads.

module Control.Monad
    (
    -- * Functor and monad classes

      Functor(fmap)
    , Monad((>>=), (>>), return, fail)

    , MonadPlus (   -- class context: Monad
          mzero     -- :: (MonadPlus m) => m a
        , mplus     -- :: (MonadPlus m) => m a -> m a -> m a
      )
```

Haddock

```
  -- * Functions

  -- ** Naming conventions
  -- $naming

  -- ** Basic functions from the "Prelude"

  , sequence      -- :: (Monad m) => [m a] -> m [a]
  , sequence_     -- :: (Monad m) => [m a] -> m ()
  , (=<<)         -- :: (Monad m) => (a -> m b) -> m a -> m b

  -- ** Generalisations of list functions

  ) where

-- -----------------------------------------------------------------------------
-- Other monad functions

-- | The 'join' function is the conventional monad join operator. It is used to
-- remove one level of monadic structure, projecting its bound argument into the
-- outer level.
join              :: (Monad m) => m (m a) -> m a
join x            =  x >>= id
```

Haddock

```
{- $naming

The functions in this library use the following naming conventions:

* A postfix \'@M@\' always stands for a function in the Kleisli category:
  The monad type constructor @m@ is added to function results
  (modulo currying) and nowhere else.  So, for example,

>  filter  ::              (a ->   Bool) -> [a] ->   [a]
>  filterM :: (Monad m) => (a -> m Bool) -> [a] -> m [a]

* A postfix \'@_@\' changes the result type from @(m a)@ to @(m ())@.
  Thus, for example:

>  sequence  :: Monad m => [m a] -> m [a]
>  sequence_ :: Monad m => [m a] -> m ()

* A prefix \'@m@\' generalizes an existing function to a monadic form.
  Thus, for example:

>  sum  :: Num a       => [a]   -> a
>  msum :: MonadPlus m => [m a] -> m a

-}
```

Part III

Functional Dependencies

Multi-Parameter Type Classes

Type classes can be generalized as defining relations, not only sets, over possibly multiple and higher-order types, cf. (2).

Multi-Parameter Type Classes

Type classes can be generalized as defining relations, not only sets, over possibly multiple and higher-order types, cf. (2).

```
class Eq a where (==) :: a -> a -> Bool
                 (/=) :: a -> a -> Bool
```

Multi-Parameter Type Classes

Type classes can be generalized as defining relations, not only sets, over possibly multiple and higher-order types, cf. (2).

```
class Eq a where (==) :: a -> a -> Bool
                 (/=) :: a -> a -> Bool
```

```
class Coerce a b where coerce :: a -> b
```

Multi-Parameter Type Classes

Type classes can be generalized as defining relations, not only sets, over possibly multiple and higher-order types, cf. (2).

```
class Eq a where (==) :: a -> a -> Bool
                 (/=) :: a -> a -> Bool


class Coerce a b where coerce :: a -> b


class FiniteMap i e f

instance Eq i => FiniteMap i e [(i, e)]
instance Eq i => FiniteMap i e (i -> e)
instance Ix i => FiniteMap i e (Array i e)
```

Functional Dependencies

Type inference on multi-parameter classes is often ambiguous.

Functional Dependencies

Type inference on multi-parameter classes is often ambiguous.

Restricting the classes with functional dependencies changes relations on types to functions on types, a powerful thing! (1)

Functional Dependencies

Type inference on multi-parameter classes is often ambiguous.

Restricting the classes with functional dependencies changes relations on types to functions on types, a powerful thing! (1)

```
data Morphs a = Morphs a [Prefix] [Suffix]
```

Functional Dependencies

Type inference on multi-parameter classes is often ambiguous.

Restricting the classes with functional dependencies changes relations on types to functions on types, a powerful thing! (1)

```
data Morphs a = Morphs a [Prefix] [Suffix]

class Morphing a b | a -> b where
    morph :: a -> Morphs b
```

Functional Dependencies

Type inference on multi-parameter classes is often ambiguous.

Restricting the classes with functional dependencies changes relations on types to functions on types, a powerful thing! (1)

```
data Morphs a = Morphs a [Prefix] [Suffix]

class Morphing a b | a -> b where
    morph :: a -> Morphs b

instance Morphing (Morphs a) a where
    morph = id
```

Functional Dependencies

Type inference on multi-parameter classes is often ambiguous.

Restricting the classes with functional dependencies changes relations on types to functions on types, a powerful thing! (1)

```
data Morphs a = Morphs a [Prefix] [Suffix]

class Morphing a b | a -> b where
    morph :: a -> Morphs b

instance Morphing (Morphs a) a where
    morph = id

instance Morphing PatternT PatternT where
    morph x = Morphs x [] []
```

Functional Dependencies

Type inference on multi-parameter classes is often ambiguous.

Restricting the classes with functional dependencies changes relations on types to functions on types, a powerful thing! (1)

```
data Morphs a = Morphs a [Prefix] [Suffix]

class Morphing a b | a -> b where
    morph :: a -> Morphs b

instance Morphing (Morphs a) a where
    morph = id

instance Morphing PatternQ PatternQ where
    morph x = Morphs x [] []
```

Functional Dependencies

Type inference on multi-parameter classes is often ambiguous.

Restricting the classes with functional dependencies changes relations on types to functions on types, a powerful thing! (1)

```
data Morphs a = Morphs a [Prefix] [Suffix]

class Morphing a b | a -> b where
    morph :: a -> Morphs b

instance Morphing (Morphs a) a where
    morph = id

instance Morphing PatternL PatternL where
    morph x = Morphs x [] []
```

Functional Dependencies

Type inference on multi-parameter classes is often ambiguous.

Restricting the classes with functional dependencies changes relations on types to functions on types, a powerful thing! (1)

```
data Morphs a = Morphs a [Prefix] [Suffix]

class Morphing a b | a -> b where
    morph :: a -> Morphs b

instance Morphing (Morphs a) a where
    morph = id

instance Morphing String  String  where
    morph x = Morphs x [] []
```

```haskell
data PatternT = FaCaL | FuCuL | FiCAL | MaFCUL {- ... -}
    deriving (Enum, Show, Eq)
```

```haskell
data PatternT = FaCaL | FuCuL | FiCAL | MaFCUL {- ... -}
    deriving (Enum, Show, Eq)

data Prefix = Al | LA | Prefix String
data Suffix = Iy | AT | At {- ... -} | Suffix String
```

```haskell
data PatternT = FaCaL | FuCuL | FiCAL | MaFCUL {- ... -}
    deriving (Enum, Show, Eq)

data Prefix = Al | LA | Prefix String
data Suffix = Iy | AT | At {- ... -} | Suffix String


(|<) :: Morphing a b => a -> Suffix -> Morphs b
(>|) :: Morphing a b => Prefix -> a -> Morphs b

y |< x = Morphs t p (x : s) where Morphs t p s = morph y
x >| y = Morphs t (x : p) s where Morphs t p s = morph y
```

```haskell
data PatternT = FaCaL | FuCuL | FiCAL | MaFCUL {- ... -}
    deriving (Enum, Show, Eq)

data Prefix = Al | LA | Prefix String
data Suffix = Iy | AT | At {- ... -} | Suffix String

(|<) :: Morphing a b => a -> Suffix -> Morphs b
(>|) :: Morphing a b => Prefix -> a -> Morphs b

y |< x = Morphs t p (x : s) where Morphs t p s = morph y
x >| y = Morphs t (x : p) s where Morphs t p s = morph y
```

```haskell
lA >| FiCL |< Iy      inflect (FiCL `noun` []) "N---------I"
```

```haskell
data PatternT = FaCaL | FuCuL | FiCAL | MaFCUL {- ... -}
    deriving (Enum, Show, Eq)

data Prefix = Al | LA | Prefix String
data Suffix = Iy | AT | At {- ... -} | Suffix String


(|<) :: Morphing a b => a -> Suffix -> Morphs b
(>|) :: Morphing a b => Prefix -> a -> Morphs b

y |< x = Morphs t p (x : s) where Morphs t p s = morph y
x >| y = Morphs t (x : p) s where Morphs t p s = morph y
```

```haskell
lA >| FiCL |< Iy     inflect (FiCL `noun` []) "N---------I"
```

Explore ElixirFM (3) and discuss the class-related design decisions.

References

Thomas Hallgren.
Fun with Functional Dependencies.
In *Proceedings of the Joint CS/CE Winter Meeting*, pages 135–145, Göteborg, Sweden, January 2001. Department of Computing Science, Chalmers University of Technology.

Mark P. Jones.
Type Classes with Functional Dependencies.
In *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 230–244, London, UK, 2000. Springer.

Otakar Smrž.
Functional Arabic Morphology. Formal System and Implementation.
PhD thesis, Charles University in Prague, 2007.