Language.CoreErlang.Pretty

```haskell
-----------------------------------------------------------------------------
-- |
-- Module      :  Language.CoreErlang.Pretty
-- Copyright   :  (c) Henrique Ferreiro García 2008
--                (c) David Castro Pérez 2008
--                (c) Eric Bailey 2016
-- License     :  BSD-style (see the file LICENSE)
--
-- Maintainer  :  Alex Kropivny <alex.kropivny@gmail.com>
-- Stability   :  experimental
-- Portability :  portable
--
-- Pretty printer for CoreErlang.
--
-----------------------------------------------------------------------------
{-# LANGUAGE FlexibleInstances    #-}
{-# LANGUAGE TypeSynonymInstances #-}

module Language.CoreErlang.Pretty (
  -- * Pretty printing
  Pretty,
  prettyPrintStyleMode, prettyPrintWithMode, prettyPrint,
  -- * Pretty-printing styles (from -- "Text.PrettyPrint.HughesPJ")
  P.Style(..), P.style, P.Mode(..),
  -- * CoreErlang formatting modes
  PPMode(..), Indent, PPLayout(..), defaultMode) where

import           Control.Category           ((<<<))
import           Language.CoreErlang.Syntax
import           Prelude                    hiding (exp)

import qualified Text.PrettyPrint           as P

infixl 5 $$$

-------------------------------------------------------------------------------
```

```haskell
-- | Varieties of layout we can use.
data PPLayout = PPDefault   -- ^ classical layout
              | PPNoLayout  -- ^ everything on a single line
  deriving Eq

type Indent = Int

-- | Pretty-printing parameters.
data PPMode = PPMode
  { altIndent    :: Indent   -- ^ indentation of the alternatives
                             -- in a @case@ expression
  , caseIndent   :: Indent   -- ^ indentation of the declarations
                             -- in a @case@ expression
  , fundefIndent :: Indent   -- ^ indentation of the declarations
                             -- in a function definition
  , lambdaIndent :: Indent   -- ^ indentation of the declarations
                             -- in a @lambda@ expression
  , letIndent    :: Indent   -- ^ indentation of the declarations
                             -- in a @let@ expression
  , letrecIndent :: Indent   -- ^ indentation of the declarations
                             -- in a @letrec@ expression
  , onsideIndent :: Indent   -- ^ indentation added for continuation
                             -- lines that would otherwise be offside
  , layout       :: PPLayout -- ^ Pretty-printing style to use
  }

-- | The default mode: pretty-print using sensible defaults.
defaultMode :: PPMode
defaultMode = PPMode { altIndent    = 4
                     , caseIndent   = 4
                     , fundefIndent = 4
                     , lambdaIndent = 4
                     , letIndent    = 4
                     , letrecIndent = 4
                     , onsideIndent = 4
                     , layout       = PPDefault
                     }

-- | Pretty printing monad
newtype DocM s a = DocM (s -> a)

instance Functor (DocM s) where
  fmap f xs = do x <- xs; return (f x)

instance Applicative (DocM s) where
  pure        = return
  (<*>) m1 m2 = do x1 <- m1; x2 <- m2; return (x1 x2)

instance Monad (DocM s) where
  (>>=)  = thenDocM
  (>>)   = then_DocM
  return = retDocM

{-# INLINE thenDocM  #-}
{-# INLINE then_DocM #-}
{-# INLINE retDocM   #-}
{-# INLINE unDocM    #-}
{-# INLINE getPPEnv  #-}

thenDocM :: DocM s a -> (a -> DocM s b) -> DocM s b
thenDocM m k = DocM $ (\s -> case unDocM m $ s of a -> unDocM (k a) $ s)

then_DocM :: DocM s a -> DocM s b -> DocM s b
then_DocM m k = DocM $ (\s -> case unDocM m $ s of _ -> unDocM k $ s)

retDocM :: a -> DocM s a
retDocM a = DocM (const a)

unDocM :: DocM s a -> (s -> a)
unDocM (DocM f) = f

-- all this extra stuff, just for this one function.
getPPEnv :: DocM s s
getPPEnv = DocM id

-- | The document type produced by these pretty printers uses a 'PPMode'
-- environment.
type Doc = DocM PPMode P.Doc

-- | Things that can be pretty-printed, including all the syntactic objects
-- in "Language.CoreErlang.Syntax".
class Pretty a where
        -- | Pretty-print something in isolation.
        pretty :: a -> Doc
        -- | Pretty-print something in a precedence context.
        prettyPrec :: Int -> a -> Doc
        pretty = prettyPrec 0
        prettyPrec _ = pretty
```

The pretty printing combinators

```haskell
empty :: Doc
empty = return P.empty

nest :: Int -> Doc -> Doc
nest i m = m >>= return . P.nest i
```

Literals

```haskell
text :: String -> Doc
text  = return . P.text

char :: Char -> Doc
char = return . P.char

integer :: Integer -> Doc
integer = return . P.integer

double :: Double -> Doc
double = return . P.double

-- Simple Combining Forms
-- parens, brackets, braces, quotes, doubleQuotes :: Doc -> Doc
parens, spacedParens, brackets, braces :: Doc -> Doc
parens       d = d >>= return . P.parens
spacedParens d = d >>= return . ((P.text "( " P.<>) <<< (P.<> P.text " )"))
brackets     d = d >>= return . P.brackets
braces       d = d >>= return . P.braces
```

Constants

```haskell
comma :: Doc
comma = return P.comma
```

Combinators

```haskell
-- (<>),(<+>),($$),($+$) :: Doc -> Doc -> Doc
(<>),(<+>),($$) :: Doc -> Doc -> Doc
aM <>  bM = do { a <- aM; b <- bM; return $ a P.<>  b }
aM <+> bM = do { a <- aM; b <- bM; return $ a P.<+> b }
aM $$  bM = do { a <- aM; b <- bM; return $ a P.$$  b }
```

```haskell
-- aM $+$ bM = do { a <- aM; b <- bM; return $ a P.$+$ b}

-- hcat, hsep, vcat, sep, cat, fsep, fcat :: [Doc] -> Doc
hcat, hsep, vcat, sep, fsep :: [Doc] -> Doc
hcat dl = sequence dl >>= return . P.hcat
hsep dl = sequence dl >>= return . P.hsep
vcat dl = sequence dl >>= return . P.vcat
sep  dl = sequence dl >>= return . P.sep
-- cat dl = sequence dl >>= return . P.cat
fsep dl = sequence dl >>= return . P.fsep
-- fcat dl = sequence dl >>= return . P.fcat
```

Some More

```haskell
-- Yuk, had to cut-n-paste this one from Pretty.hs
punctuate :: Doc -> [Doc] -> [Doc]
punctuate _ []      = []
punctuate p (d1:ds) = go d1 ds
  where
    go d []     = [d]
    go d (e:es) = (d <> p) : go e es

-- | render the document with a given style and mode.
renderStyleMode :: P.Style -> PPMode -> Doc -> String
renderStyleMode ppStyle ppMode d = P.renderStyle ppStyle . unDocM d $ ppMode

-- | pretty-print with a given style and mode.
prettyPrintStyleMode :: Pretty a => P.Style -> PPMode -> a -> String
prettyPrintStyleMode ppStyle ppMode = renderStyleMode ppStyle ppMode . pretty

-- | pretty-print with the default style and a given mode.
prettyPrintWithMode :: Pretty a => PPMode -> a -> String
prettyPrintWithMode = prettyPrintStyleMode P.style

-- | pretty-print with the default style and 'defaultMode'.
prettyPrint :: Pretty a => a -> String
prettyPrint = prettyPrintWithMode defaultMode
```

Pretty-Print a Module

```haskell
instance Pretty Module where
  pretty (Module m (ModHeader exports attrs) fundefs) =
    topLevel (ppModuleHeader m exports attrs) (map pretty fundefs)
```

Module Header

```
ppModuleHeader :: Atom -> Exports -> [(Atom,Const)] -> Doc
ppModuleHeader m exports attrs = myFsep
  [ text "module"     <+> pretty m <+> (bracketList $ map pretty exports)
  , text "attributes" <+> bracketList (map ppAssign attrs)
  ]

instance Pretty FunName where
  pretty (name,arity) = pretty name <> char '/' <> integer arity

instance Pretty Const where
  pretty (CLit   l) = pretty  l
  pretty (CTuple l) = ppTuple l
  pretty (CList  l) = pretty  l
```

Declarations

```
instance Pretty FunDef where
  pretty (FunDef function exp) =
    (pretty function <+> char '=') $$$ ppBody fundefIndent [pretty exp]
```

Expressions

```
instance Pretty Literal where
  pretty (LChar c)    = char c
  pretty (LString s)  = text (show s)
  pretty (LInt i)     = integer i
  pretty (LFloat f)   = double f
  pretty (LAtom a)    = pretty a
  pretty LNil         = bracketList [empty]

instance Pretty Atom where
  pretty (Atom a) = char '\'' <> text a <> char '\''

instance Pretty Exps where
  pretty (Exp e)           = pretty e
  pretty (Exps (Constr e)) = angleList (map pretty e)
  pretty (Exps (Ann e cs)) = spacedParens $
    angleList (map pretty e) $$$ ppAnn cs

instance Pretty Exp where
  pretty (Var v)      =  text v
  pretty (Lit l)      = pretty l
  pretty (FunName f)  = pretty f
  pretty (App e exps) = text "apply" <+> pretty e <> parenList (map pretty exps)
  pretty (ModCall (e1,e2) exps) =
    sep [ text "call" <+> pretty e1 <> char ':' <> pretty e2
        , parenList (map pretty exps)
        ]
  pretty (Fun vars e) =
    sep [ text "fun " <> parenList (map pretty vars) <+> text "->"
        , ppBody lambdaIndent [pretty e]
        ]
  pretty (Seq e1 e2) = sep [text "do", pretty e1, pretty e2]
  pretty (Let (vars,e1) e2) =
    text "let" <+> angleList (map text vars) <+> char '='
    <+> pretty e1 $$$ text "in" <+> pretty e2
  pretty (Letrec fundefs e) =
    sep [ text "letrec" <+> ppBody letrecIndent (map pretty fundefs)
        , text "in", pretty e
        ]
  pretty (Case e alts) = sep [text "case", pretty e, text "of"]
                         $$$ ppBody caseIndent (map pretty alts)
                         $$$ text "end"
  pretty (Tuple exps)    = braceList $ map pretty exps
  pretty (List l)        = pretty l
  pretty (PrimOp a exps) = text "primop" <+>
                           pretty a <> parenList (map pretty exps)
  pretty (Binary bs)     = char '#' <> braceList (map pretty bs) <> char '#'
  pretty (Try e (vars1,exps1) (vars2,exps2)) =
    text "try"
    $$$ ppBody caseIndent [pretty e]
    $$$ text "of" <+> angleList (map text vars1) <+> text "->"
    $$$ ppBody altIndent [pretty exps1]
    $$$ text "catch" <+> angleList (map text vars2) <+> text "->"
    $$$ ppBody altIndent [pretty exps2]
  pretty (Rec alts tout) =
    text "receive"
    $$$ ppBody caseIndent (map pretty alts)
    $$$ text "after"
    $$$ ppBody caseIndent [pretty tout]
  pretty (Catch e) = sep [text "catch", pretty e]

instance Pretty a => Pretty (List a) where
  pretty (L l)    = bracketList $ map pretty l
  pretty (LL h t) = brackets . hcat $
    punctuate comma (map pretty h) ++ [char '|' <> pretty t]
instance Pretty Clause where
  pretty (Clause pats guard exps) =
    myFsep [pretty pats, pretty guard <+> text "->"]
    $$$ ppBody altIndent [pretty exps]

instance Pretty Pats where
  pretty (Pat p)  = pretty p
  pretty (Pats p) = angleList (map pretty p)

instance Pretty Pat where
  pretty (PVar v)     = text v
  pretty (PLit l)     = pretty l
  pretty (PTuple p)   = braceList $ map pretty p
  pretty (PList l)    = pretty l
  pretty (PBinary bs) = char '#' <> braceList (map pretty bs) <> char '#'
  pretty (PAlias a)   = pretty a

instance Pretty Alias where
 pretty (Alias v p) = ppAssign (Var v,p) -- FIXME: hack!

instance Pretty Guard where
  pretty (Guard e) = text "when" <+> pretty e

instance Pretty Timeout where
  pretty (Timeout e1 e2) =
    pretty e1 <+> text "->" $$$ ppBody altIndent [pretty e2]

instance Pretty a => Pretty (BitString a) where
  pretty (BitString e es) =
    text "#<" <> pretty e <> char '>' <> parenList (map pretty es)
```

Annotations

```
instance Pretty a => Pretty (Ann a) where
  pretty (Constr a) = pretty a
  pretty (Ann a cs) = spacedParens (pretty a $$$ ppAnn cs)

instance Pretty VarName where
  pretty = text
```

Pretty printing utilities

```
angles :: Doc -> Doc
angles p = char '<' <> p <> char '>'

angleList, braceList, bracketList, parenList, commaSep :: [Doc] -> Doc
```

```haskell
angleList   = angles       . commaSep
braceList   = braces       . commaSep
bracketList = brackets     . commaSep
parenList   = parens       . commaSep
commaSep    = myFsepSimple . punctuate comma

-- | Monadic PP Combinators -- these examine the env
topLevel :: Doc -> [Doc] -> Doc
topLevel header dl = do e <- fmap layout getPPEnv
                        let s = case e of
                                  PPDefault  -> header $$ vcat dl
                                  PPNoLayout -> header <+> hsep dl
                        s $$$ text "end"

ppAssign :: (Pretty a,Pretty b) => (a,b) -> Doc
ppAssign (a,b) = pretty a <+> char '=' <+> pretty b

ppTuple :: Pretty a => [a] -> Doc
ppTuple t = braceList (map pretty t)

ppBody :: (PPMode -> Int) -> [Doc] -> Doc
ppBody f dl = do e <- fmap layout getPPEnv
                 i <- fmap f getPPEnv
                 case e of
                   PPDefault -> nest i . vcat $ dl
                   _         -> hsep dl

($$$) :: Doc -> Doc -> Doc
a $$$ b = layoutChoice (a $$) (a <+>) b

myFsepSimple :: [Doc] -> Doc
myFsepSimple = layoutChoice fsep hsep
```

Same, except that continuation lines are indented, which is necessary to avoid triggering the offside rule.

```haskell
myFsep :: [Doc] -> Doc
myFsep = layoutChoice fsep' hsep
  where
    fsep' []     = empty
    fsep' (d:ds) = do e <- getPPEnv
                      let n = onsideIndent e
                      nest n (fsep (nest (-n) d:ds))

layoutChoice :: (a -> Doc) -> (a -> Doc) -> a -> Doc
layoutChoice a b dl = do e <- getPPEnv
                         if layout e == PPDefault
```

```
                              then a dl
                              else b dl

ppAnn :: (Pretty a) => [a] -> Doc
ppAnn cs = text "-|" <+> bracketList (map pretty cs)
```