Contents


```
-- |
-- Module      :  Cryptol.Eval.Reference
-- Description :  The reference implementation of the Cryptol evaluation semantics.
-- Copyright   :  (c) 2013-2016 Galois, Inc.
-- License     :  BSD3
-- Maintainer  :  cryptol@galois.com
-- Stability   :  provisional
-- Portability :  portable

{-# LANGUAGE PatternGuards #-}
{-# LANGUAGE BlockArguments #-}

module Cryptol.Eval.Reference
  ( Value(..)
  , evaluate
  , evalExpr
  , evalDeclGroup
```

```haskell
  , ppValue
  ) where

import Control.Applicative (liftA2)
import Data.Bits
import Data.Ratio((%))
import Data.List
  (genericDrop, genericIndex, genericLength, genericReplicate, genericSplitAt,
   genericTake, sortBy)
import Data.Ord (comparing)
import Data.Map (Map)
import qualified Data.Map as Map
import qualified Data.Text as T (pack)
import LibBF (BigFloat)
import qualified LibBF as FP

import Cryptol.ModuleSystem.Name (asPrim)
import Cryptol.TypeCheck.Solver.InfNat (Nat'(..), nAdd, nMin, nMul)
import Cryptol.TypeCheck.AST
import Cryptol.Eval.Monad (EvalError(..), PPOpts(..))
import Cryptol.Eval.Type (TValue(..), isTBit, evalValType, evalNumType, tvSeq)
import Cryptol.Eval.Concrete (mkBv, ppBV, lg2)
import Cryptol.Eval.Concrete.FloatHelpers(fpPP,fpToI,BF(..),fpRound)
import Cryptol.Eval.Concrete.Float(floatFromBits,floatToBits)
import Cryptol.Utils.Ident (Ident,PrimIdent, prelPrim, floatPrim)
import Cryptol.Utils.Panic (panic)
import Cryptol.Utils.PP
import Cryptol.Utils.RecordMap

import qualified Cryptol.ModuleSystem as M
import qualified Cryptol.ModuleSystem.Env as M (loadedModules)
```

Overview

This file describes the semantics of the explicitly-typed Cryptol language (i.e., terms after type checking). Issues related to type inference, type functions, and type constraints are beyond the scope of this document.

Cryptol Types

Cryptol types come in two kinds: numeric types (kind `#`) and value types (kind `*`). While value types are inhabited by well-typed Cryptol expressions, numeric types are only used as parameters to other types; they have no inhabitants. In this implementation we represent numeric types as values of the Haskell type `Nat'` of natural numbers with infinity; value types are represented as values of type `TValue`.

The value types of Cryptol, along with their Haskell representations, are as follows:

Cryptol type	Description	`TValue` representation
`Bit`	booleans	`TVBit`
`Integer`	integers	`TVInteger`

Cryptol type	Description	TValue representation
Z n	integers modulo n	TVIntMod n
Rational	rationals	TVRational
Float e p	floating point	TVFloat
Array	arrays	TVArray
[n]a	finite lists	TVSeq n a
[inf]a	infinite lists	TVStream a
(a, b, c)	tuples	TVTuple [a,b,c]
{x:a, y:b, z:c}	records	TVRec [(x,a),(y,b),(z,c)]
a -> b	functions	TVFun a b

We model each Cryptol value type `t` as a complete partial order (cpo) M(`t`). To each Cryptol expression `e : t` we assign a meaning M(`e`) in M(`t`); in particular, recursive Cryptol programs of type `t` are modeled as least fixed points in M(`t`). In other words, this is a domain-theoretic denotational semantics.

Evaluating a Cryptol expression of base type (one of: `Bit`, `Integer`, `Z n`, or `Rational`) may result in:

- a defined value (e.g., `True` or `False`)
- a run-time error, or
- non-termination.

Accordingly, M(`Bit`) is a flat cpo with values for `True`, `False`, run-time errors of type `EvalError`, and $\bot$; we represent it with the Haskell type `Either EvalError Bool`.

Similarly, M(`Integer`) is a flat cpo with values for integers, run-time errors, and $\bot$; we represent it with the Haskell type `Either EvalError Integer`.

The cpos for lists, tuples, and records are cartesian products. The cpo ordering is pointwise, and the bottom element $\bot$ is the list/tuple/record whose elements are all $\bot$. Trivial types `[0]t`, `()` and `{}` denote single-element cpos where the unique value `[]`/`()`/`{}` *is* the bottom element $\bot$. M(`a -> b`) is the continuous function space M(`a`) $\to$ M(`b`).

Type schemas of the form `{a1 ... an} (p1 ... pk) => t` classify polymorphic values in Cryptol. These are represented with the Haskell type `Schema`. The meaning of a schema is cpo whose elements are functions: For each valid instantiation `t1 ... tn` of the type parameters `a1 ... an` that satisfies the constraints `p1 ... pk`, the function returns a value in M(`t[t1/a1 ... tn/an]`).

Values

The Haskell code in this module defines the semantics of typed Cryptol terms by providing an evaluator to an appropriate `Value` type.

```
-- | Value type for the reference evaluator.
data Value
  = VBit (Either EvalError Bool) -- ^ @ Bit    @ booleans
  | VInteger (Either EvalError Integer) -- ^ @ Integer @   or @Z n@ integers
  | VRational (Either EvalError Rational) -- ^ @ Rational @ rationals
  | VFloat (Either EvalError BF) -- ^ Floating point numbers
  | VList Nat' [Value]           -- ^ @ [n]a   @ finite or infinite lists
  | VTuple [Value]               -- ^ @ ( .. ) @ tuples
```

```
  | VRecord [(Ident, Value)]      -- ^ @ { .. } @ records
  | VFun (Value -> Value)         -- ^ functions
  | VPoly (TValue -> Value)       -- ^ polymorphic values (kind *)
  | VNumPoly (Nat' -> Value)      -- ^ polymorphic values (kind #)
```

Invariant: Undefinedness and run-time exceptions are only allowed inside the argument of a `VBit`, `VInteger` or `VRational` constructor. All other `Value` and list constructors should evaluate without error. For example, a non-terminating computation at type `(Bit,Bit)` must be represented as `VTuple [VBit undefined, VBit undefined]`, and not simply as `undefined`. Similarly, an expression like `1/0:[2]` that raises a run-time error must be encoded as `VList (Nat 2) [VBit (Left e), VBit (Left e)]` where `e = DivideByZero`.

Copy Functions

Functions `copyBySchema` and `copyByTValue` make a copy of the given value, building the spine based only on the type without forcing the value argument. This ensures that undefinedness appears inside `VBit` and `VInteger` values only, and never on any constructors of the `Value` type. In turn, this gives the appropriate semantics to recursive definitions: The bottom value for a compound type is equal to a value of the same type where every individual bit is bottom.

For each Cryptol type `t`, the cpo $M($`t`$)$ can be represented as a subset of values of type `Value` that satisfy the datatype invariant. This subset consists precisely of the output range of `copyByTValue t`. Similarly, the range of output values of `copyBySchema` yields the cpo that represents any given schema.

```
copyBySchema :: Env -> Schema -> Value -> Value
copyBySchema env0 (Forall params _props ty) = go params env0
  where
    go :: [TParam] -> Env -> Value -> Value
    go []       env v = copyByTValue (evalValType (envTypes env) ty) v
    go (p : ps) env v =
      case v of
        VPoly    f -> VPoly    $ \t -> go ps (bindType (tpVar p) (Right t) env) (f t)
        VNumPoly f -> VNumPoly $ \n -> go ps (bindType (tpVar p) (Left n)  env) (f n)
        _          -> evalPanic "copyBySchema" ["Expected polymorphic value"]

copyByTValue :: TValue -> Value -> Value
copyByTValue = go
  where
    go :: TValue -> Value -> Value
    go ty val =
      case ty of
        TVBit        -> VBit (fromVBit val)
        TVInteger    -> VInteger (fromVInteger val)
        TVIntMod _   -> VInteger (fromVInteger val)
        TVRational   -> VRational (fromVRational val)
        TVFloat _ _  -> VFloat (fromVFloat' val)
        TVArray{}    -> evalPanic "copyByTValue" ["Unsupported Array type"]
        TVSeq w ety  -> VList (Nat w) (map (go ety) (copyList w (fromVList val)))
        TVStream ety -> VList Inf (map (go ety) (copyStream (fromVList val)))
        TVTuple etys -> VTuple (zipWith go etys (copyList (genericLength etys) (fromVTuple val)))
        TVRec fields -> VRecord [ (f, go fty (lookupRecord f val)) | (f, fty) <- canonicalFields fie
        TVFun _ bty  -> VFun (\v -> go bty (fromVFun val v))
```

```haskell
        TVAbstract {} -> val

copyStream :: [a] -> [a]
copyStream xs = head xs : copyStream (tail xs)

copyList :: Integer -> [a] -> [a]
copyList 0 _ = []
copyList n xs = head xs : copyList (n - 1) (tail xs)
```

Operations on Values

```haskell
-- | Destructor for @VBit@.
fromVBit :: Value -> Either EvalError Bool
fromVBit (VBit b) = b
fromVBit _        = evalPanic "fromVBit" ["Expected a bit"]

-- | Destructor for @VInteger@.
fromVInteger :: Value -> Either EvalError Integer
fromVInteger (VInteger i) = i
fromVInteger _            = evalPanic "fromVInteger" ["Expected an integer"]

-- | Destructor for @VRational@.
fromVRational :: Value -> Either EvalError Rational
fromVRational (VRational i) = i
fromVRational _             = evalPanic "fromVRational" ["Expected a rational"]

fromVFloat :: Value -> Either EvalError BigFloat
fromVFloat = fmap bfValue . fromVFloat'

fromVFloat' :: Value -> Either EvalError BF
fromVFloat' v =
  case v of
    VFloat f -> f
    _ -> evalPanic "fromVFloat" [ "Expected a floating point value." ]


-- | Destructor for @VList@.
fromVList :: Value -> [Value]
fromVList (VList _ vs) = vs
fromVList _            = evalPanic "fromVList" ["Expected a list"]

-- | Destructor for @VTuple@.
fromVTuple :: Value -> [Value]
fromVTuple (VTuple vs) = vs
fromVTuple _           = evalPanic "fromVTuple" ["Expected a tuple"]

-- | Destructor for @VRecord@.
fromVRecord :: Value -> [(Ident, Value)]
fromVRecord (VRecord fs) = fs
fromVRecord _            = evalPanic "fromVRecord" ["Expected a record"]

-- | Destructor for @VFun@.
```

```haskell
fromVFun :: Value -> (Value -> Value)
fromVFun (VFun f) = f
fromVFun _        = evalPanic "fromVFun" ["Expected a function"]

-- | Destructor for @VPoly@.
fromVPoly :: Value -> (TValue -> Value)
fromVPoly (VPoly f) = f
fromVPoly _         = evalPanic "fromVPoly" ["Expected a polymorphic value"]

-- | Destructor for @VNumPoly@.
fromVNumPoly :: Value -> (Nat' -> Value)
fromVNumPoly (VNumPoly f) = f
fromVNumPoly _            = evalPanic "fromVNumPoly" ["Expected a polymorphic value"]

-- | Look up a field in a record.
lookupRecord :: Ident -> Value -> Value
lookupRecord f v =
  case lookup f (fromVRecord v) of
    Just val -> val
    Nothing  -> evalPanic "lookupRecord" ["Malformed record"]

-- | Polymorphic function values that expect a finite numeric type.
vFinPoly :: (Integer -> Value) -> Value
vFinPoly f = VNumPoly g
  where
    g (Nat n) = f n
    g Inf     = evalPanic "vFinPoly" ["Expected finite numeric type"]
```

Environments

An evaluation environment keeps track of the values of term variables and type variables that are in scope at any point.

```haskell
data Env = Env
  { envVars       :: !(Map Name Value)
  , envTypes      :: !(Map TVar (Either Nat' TValue))
  }

instance Semigroup Env where
  l <> r = Env
    { envVars  = Map.union (envVars  l) (envVars  r)
    , envTypes = Map.union (envTypes l) (envTypes r)
    }

instance Monoid Env where
  mempty = Env
    { envVars  = Map.empty
    , envTypes = Map.empty
    }
  mappend l r = l <> r

-- | Bind a variable in the evaluation environment.
```

```haskell
bindVar :: (Name, Value) -> Env -> Env
bindVar (n, val) env = env { envVars = Map.insert n val (envVars env) }

-- | Bind a type variable of kind # or *.
bindType :: TVar -> Either Nat' TValue -> Env -> Env
bindType p ty env = env { envTypes = Map.insert p ty (envTypes env) }
```

Evaluation

The meaning $M(\texttt{expr})$ of a Cryptol expression `expr` is defined by recursion over its structure. For an expression that contains free variables, the meaning also depends on the environment `env`, which assigns values to those variables.

```haskell
evalExpr :: Env     -- ^ Evaluation environment
         -> Expr    -- ^ Expression to evaluate
         -> Value
evalExpr env expr =
  case expr of

    EList es _ty  -> VList (Nat (genericLength es)) [ evalExpr env e | e <- es ]
    ETuple es     -> VTuple [ evalExpr env e | e <- es ]
    ERec fields   -> VRecord [ (f, evalExpr env e) | (f, e) <- canonicalFields fields ]
    ESel e sel    -> evalSel (evalExpr env e) sel
    ESet e sel v  -> evalSet (evalExpr env e) sel (evalExpr env v)

    EIf c t f ->
      condValue (fromVBit (evalExpr env c)) (evalExpr env t) (evalExpr env f)

    EComp _n _ty e branches ->
      evalComp env e branches

    EVar n ->
      case Map.lookup n (envVars env) of
        Just val -> val
        Nothing  -> evalPanic "evalExpr" ["var `" ++ show (pp n) ++ "` is not defined" ]

    ETAbs tv b ->
      case tpKind tv of
        KType -> VPoly    $ \ty -> evalExpr (bindType (tpVar tv) (Right ty) env) b
        KNum  -> VNumPoly $ \n  -> evalExpr (bindType (tpVar tv) (Left n) env) b
        k     -> evalPanic "evalExpr" ["Invalid kind on type abstraction", show k]

    ETApp e ty ->
      case evalExpr env e of
        VPoly f     -> f $! (evalValType (envTypes env) ty)
        VNumPoly f  -> f $! (evalNumType (envTypes env) ty)
        _           -> evalPanic "evalExpr" ["Expected a polymorphic value"]

    EApp e1 e2     -> fromVFun (evalExpr env e1) (evalExpr env e2)
    EAbs n _ty b   -> VFun (\v -> evalExpr (bindVar (n, v) env) b)
    EProofAbs _ e  -> evalExpr env e
```

```
    EProofApp e     -> evalExpr env e
    EWhere e dgs    -> evalExpr (foldl evalDeclGroup env dgs) e
```

Selectors

Apply the the given selector form to the given value.

```
evalSel :: Value -> Selector -> Value
evalSel val sel =
  case sel of
    TupleSel n _  -> tupleSel n val
    RecordSel n _ -> recordSel n val
    ListSel n _  -> listSel n val
  where
    tupleSel n v =
      case v of
        VTuple vs    -> vs !! n
        _            -> evalPanic "evalSel"
                          ["Unexpected value in tuple selection."]
    recordSel n v =
      case v of
        VRecord _    -> lookupRecord n v
        _            -> evalPanic "evalSel"
                          ["Unexpected value in record selection."]
    listSel n v =
      case v of
        VList _ vs   -> vs !! n
        _            -> evalPanic "evalSel"
                          ["Unexpected value in list selection."]
```

Update the given value using the given selector and new value.

```
evalSet :: Value -> Selector -> Value -> Value
evalSet val sel fval =
  case sel of
    TupleSel n _  -> updTupleAt n
    RecordSel n _ -> updRecAt n
    ListSel n _   -> updSeqAt n
  where
    updTupleAt n =
      case val of
        VTuple vs | (as,_:bs) <- splitAt n vs ->
          VTuple (as ++ fval : bs)
        _ -> bad "Invalid tuple upldate."

    updRecAt n =
      case val of
        VRecord vs | (as, (i,_) : bs) <- break ((n==) . fst) vs ->
          VRecord (as ++ (i,fval) : bs)
        _ -> bad "Invalid record update."

    updSeqAt n =
      case val of
```

```
      VList i vs | (as, _ : bs) <- splitAt n vs ->
        VList i (as ++ fval : bs)
      _ -> bad "Invalid sequence update."

  bad msg = evalPanic "evalSet" [msg]
```

Conditionals

We evaluate conditionals on larger types by pushing the conditionals down to the individual bits.

```
condValue :: Either EvalError Bool -> Value -> Value -> Value
condValue c l r =
  case l of
    VBit b     -> VBit (condBit c b (fromVBit r))
    VInteger i -> VInteger (condBit c i (fromVInteger r))
    VRational x -> VRational (condBit c x (fromVRational r))
    VFloat x    -> VFloat (condBit c x (fromVFloat' r))
    VList n vs -> VList n (zipWith (condValue c) vs (fromVList r))
    VTuple vs  -> VTuple (zipWith (condValue c) vs (fromVTuple r))
    VRecord fs -> VRecord [ (f, condValue c v (lookupRecord f r)) | (f, v) <- fs ]
    VFun f     -> VFun (\v -> condValue c (f v) (fromVFun r v))
    VPoly f    -> VPoly (\t -> condValue c (f t) (fromVPoly r t))
    VNumPoly f -> VNumPoly (\n -> condValue c (f n) (fromVNumPoly r n))
```

Conditionals are explicitly lazy: Run-time errors in an untaken branch are ignored.

```
condBit :: Either e Bool -> Either e a -> Either e a -> Either e a
condBit (Left e)  _ _ = Left e
condBit (Right b) x y = if b then x else y
```

List Comprehensions

Cryptol list comprehensions consist of one or more parallel branches; each branch has one or more matches that bind values to variables.

The result of evaluating a match in an initial environment is a list of extended environments. Each new environment binds the same single variable to a different element of the match's list.

```
evalMatch :: Env -> Match -> [Env]
evalMatch env m =
  case m of
    Let d ->
      [ bindVar (evalDecl env d) env ]
    From n _l _ty expr ->
      [ bindVar (n, v) env | v <- fromVList (evalExpr env expr) ]

lenMatch :: Env -> Match -> Nat'
lenMatch env m =
  case m of
    Let _          -> Nat 1
    From _ len _ _ -> evalNumType (envTypes env) len
```

The result of of evaluating a branch in an initial environment is a list of extended environments, each of which extends the initial environment with the same set of new variables. The length of the list is equal to the product of the lengths of the lists in the matches.

```
evalBranch :: Env -> [Match] -> [Env]
evalBranch env [] = [env]
evalBranch env (match : matches) =
  [ env'' | env' <- evalMatch env match
          , env'' <- evalBranch env' matches ]
```

```
lenBranch :: Env -> [Match] -> Nat'
lenBranch _env [] = Nat 1
lenBranch env (match : matches) =
  nMul (lenMatch env match) (lenBranch env matches)
```

The head expression of the comprehension can refer to any variable bound in any of the parallel branches. So to evaluate the comprehension, we zip and merge together the lists of extended environments from each branch. The head expression is then evaluated separately in each merged environment. The length of the resulting list is equal to the minimum length over all parallel branches.

```
evalComp :: Env         -- ^ Starting evaluation environment
         -> Expr        -- ^ Head expression of the comprehension
         -> [[Match]]   -- ^ List of parallel comprehension branches
         -> Value
evalComp env expr branches = VList len [ evalExpr e expr | e <- envs ]
  where
    -- Generate a new environment for each iteration of each
    -- parallel branch.
    benvs :: [[Env]]
    benvs = map (evalBranch env) branches

    -- Zip together the lists of environments from each branch,
    -- producing a list of merged environments. Longer branches get
    -- truncated to the length of the shortest branch.
    envs :: [Env]
    envs = foldr1 (zipWith mappend) benvs

    len :: Nat'
    len = foldr1 nMin (map (lenBranch env) branches)
```

Declarations

Function `evalDeclGroup` extends the given evaluation environment with the result of evaluating the given declaration group. In the case of a recursive declaration group, we tie the recursive knot by evaluating each declaration in the extended environment `env'` that includes all the new bindings.

```
evalDeclGroup :: Env -> DeclGroup -> Env
evalDeclGroup env dg = do
  case dg of
    NonRecursive d ->
      bindVar (evalDecl env d) env
    Recursive ds ->
      let env' = foldr bindVar env bindings
          bindings = map (evalDeclRecursive env') ds
      in env'
```

To evaluate a declaration in a non-recursive context, we need only evaluate the expression on the right-hand side or look up the appropriate primitive.

```haskell
evalDecl :: Env -> Decl -> (Name, Value)
evalDecl env d =
  case dDefinition d of
    DPrim   -> (dName d, evalPrim (dName d))
    DExpr e -> (dName d, evalExpr env e)
```

To evaluate a declaration in a recursive context, we must perform a type-directed copy to build the spine of the value. This ensures that the definedness invariant for type `Value` will be maintained.

```haskell
evalDeclRecursive :: Env -> Decl -> (Name, Value)
evalDeclRecursive env d =
  case dDefinition d of
    DPrim   -> (dName d, evalPrim (dName d))
    DExpr e -> (dName d, copyBySchema env (dSignature d) (evalExpr env e))
```

Primitives

To evaluate a primitive, we look up its implementation by name in a table.

```haskell
evalPrim :: Name -> Value
evalPrim n
  | Just i <- asPrim n, Just v <- Map.lookup i primTable = v
  | otherwise = evalPanic "evalPrim" ["Unimplemented primitive", show n]
```

Cryptol primitives fall into several groups, mostly delenieated by corresponding typeclasses

- Literals: `True`, `False`, `number`, `ratio`
- Zero: zero
- Logic: `&&`, `||`, `^`, `complement`
- Ring: `+`, `-`, `*`, `negate`, `fromInteger`
- Integral: `/`, `%`, `^^`, `toInteger`
- Bitvector: `/$ %$`, `lg2`, `<=$`
- Comparison: `<`, `>`, `<=`, `>=`, `==`, `!=`
- Sequences: `#`, `join`, `split`, `splitAt`, `reverse`, `transpose`
- Shifting: `<<`, `>>`, `<<<`, `>>>`
- Indexing: `@`, `@@`, `!`, `!!`, `update`, `updateEnd`
- Enumerations: `fromTo`, `fromThenTo`, `infFrom`, `infFromThen`
- Polynomials: `pmult`, `pdiv`, `pmod`
- Miscellaneous: `error`, `random`, `trace`

```haskell
primTable :: Map PrimIdent Value
primTable = Map.unions
              [ cryptolPrimTable
              , floatPrimTable
              ]
```

```haskell
cryptolPrimTable :: Map PrimIdent Value
cryptolPrimTable = Map.fromList $ map (\(n, v) -> (prelPrim (T.pack n), v))

  -- Literals
  [ ("True"       , VBit (Right True))
  , ("False"      , VBit (Right False))
  , ("number"     , vFinPoly $ \val ->
                    VPoly $ \a ->
                    literal val a)
  , ("fraction"   , vFinPoly \top ->
                    vFinPoly \bot ->
                    vFinPoly \rnd ->
                    VPoly    \a   -> fraction top bot rnd a)
  -- Zero
  , ("zero"       , VPoly zero)

  -- Logic (bitwise)
  , ("&&"         , binary (logicBinary (&&)))
  , ("||"         , binary (logicBinary (||)))
  , ("^"          , binary (logicBinary (/=)))
  , ("complement" , unary  (logicUnary not))

  -- Ring
  , ("+"          , binary (ringBinary
                             (\x y -> Right (x + y))
                             (\x y -> Right (x + y))
                             (fpBin FP.bfAdd fpImplicitRound)
                           ))
  , ("-"          , binary (ringBinary
                              (\x y -> Right (x - y))
                              (\x y -> Right (x - y))
                              (fpBin FP.bfSub fpImplicitRound)
                            ))
  , ("*"          , binary ringMul)
  , ("negate"     , unary  (ringUnary (\x -> Right (- x))
                                      (\x -> Right (- x))
                                      (\_ _ x -> Right (FP.bfNeg x))))
  , ("fromInteger", VPoly $ \a ->
                    VFun $ \x ->
                    ringNullary (fromVInteger x)
                                (fromInteger <$> fromVInteger x)
                                (\e p -> fpFromInteger e p <$> fromVInteger x)
                                 a)

  -- Integral
  , ("toInteger"  , VPoly $ \a ->
                    VFun $ \x ->
                    VInteger $ cryToInteger a x)
  , ("/"          , binary (integralBinary divWrap))
  , ("%"          , binary (integralBinary modWrap))
  , ("^^"         , VPoly $ \aty ->
```

```
                  VPoly $ \ety ->
                  VFun $ \a ->
                  VFun $ \e ->
                  ringExp aty a (cryToInteger ety e))

-- Field
, ("/."          , binary (fieldBinary ratDiv
                                        (fpBin FP.bfDiv fpImplicitRound)
                          ))
, ("recip"       , unary (fieldUnary ratRecip fpRecip))

-- Round
, ("floor"       , unary (roundUnary floor
                                      (fpToInteger "floor" FP.ToNegInf)
                        ))
, ("ceiling"     , unary (roundUnary ceiling
                                      (fpToInteger "ceiling" FP.ToPosInf)
                        ))
, ("trunc"       , unary (roundUnary truncate
                                      (fpToInteger "trunc" FP.ToZero)
                        ))
, ("roundAway",    unary (roundUnary roundAwayRat
                                      (fpToInteger "roundAway" FP.Away)
                        ))
, ("roundToEven", unary (roundUnary round
                                      (fpToInteger "roundToEven" FP.NearEven)
                        ))

-- Comparison
, ("<"           , binary (cmpOrder (\o -> o == LT)))
, (">"           , binary (cmpOrder (\o -> o == GT)))
, ("<="          , binary (cmpOrder (\o -> o /= GT)))
, (">="          , binary (cmpOrder (\o -> o /= LT)))
, ("=="          , binary (cmpOrder (\o -> o == EQ)))
, ("!="          , binary (cmpOrder (\o -> o /= EQ)))
, ("<$"          , binary signedLessThan)

-- Bitvector
, ("/$"          , vFinPoly $ \n ->
                  VFun $ \l ->
                  VFun $ \r ->
                  vWord n $ appOp2 divWrap (fromSignedVWord l) (fromSignedVWord r))
, ("%$"          , vFinPoly $ \n ->
                  VFun $ \l ->
                  VFun $ \r ->
                  vWord n $ appOp2 modWrap (fromSignedVWord l) (fromSignedVWord r))
, (">>$"         , signedShiftRV)
, ("lg2"         , vFinPoly $ \n ->
                  VFun $ \v ->
                  vWord n $ appOp1 lg2Wrap (fromVWord v))
```

```haskell
-- Rational
, ("ratio"       , VFun $ \l ->
                   VFun $ \r ->
                   VRational (appOp2 ratioOp (fromVInteger l) (fromVInteger r)))

-- Z n
, ("fromZ"       , vFinPoly $ \n ->
                   VFun $ \x ->
                   VInteger (flip mod n <$> fromVInteger x))

-- Sequences
, ("#"           , VNumPoly $ \front ->
                   VNumPoly $ \back  ->
                   VPoly $ \_elty  ->
                   VFun $ \l ->
                   VFun $ \r ->
                   VList (nAdd front back) (fromVList l ++ fromVList r))

, ("join"        , VNumPoly $ \parts ->
                   VNumPoly $ \each  ->
                   VPoly $ \_a ->
                   VFun $ \xss ->
                     case each of
                       -- special case when the inner sequences are of length 0
                       Nat 0 -> VList (Nat 0) []
                       _ -> VList (nMul parts each)
                            (concat (map fromVList (fromVList xss))))

, ("split"       , VNumPoly $ \parts ->
                   vFinPoly $ \each  ->
                   VPoly $ \_a ->
                   VFun $ \val ->
                   VList parts (splitV parts each (fromVList val)))

, ("splitAt"     , vFinPoly $ \front ->
                   VNumPoly $ \back ->
                   VPoly $ \_a ->
                   VFun $ \v ->
                   let (xs, ys) = genericSplitAt front (fromVList v)
                   in VTuple [VList (Nat front) xs, VList back ys])

, ("reverse"     , VNumPoly $ \n ->
                   VPoly $ \_a ->
                   VFun $ \v ->
                   VList n (reverse (fromVList v)))

, ("transpose"   , VNumPoly $ \rows ->
                   VNumPoly $ \cols ->
                   VPoly $ \_a ->
                   VFun $ \v ->
                   VList cols
```

```haskell
                   (map (VList rows) (transposeV cols (map fromVList (fromVList v)))))

-- Shifting:
, ("<<"          , shiftV shiftLV)
, (">>"          , shiftV shiftRV)
, ("<<<"         , rotateV rotateLV)
, (">>>"         , rotateV rotateRV)

-- Indexing:
, ("@"           , indexPrimOne  indexFront)
, ("!"           , indexPrimOne  indexBack)
, ("update"      , updatePrim updateFront)
, ("updateEnd"   , updatePrim updateBack)

-- Enumerations
, ("fromTo"      , vFinPoly $ \first ->
                   vFinPoly $ \lst   ->
                   VPoly    $ \ty  ->
                   let f i = literal i ty
                   in VList (Nat (1 + lst - first)) (map f [first .. lst]))

, ("fromThenTo"  , vFinPoly $ \first ->
                   vFinPoly $ \next  ->
                   vFinPoly $ \_lst  ->
                   VPoly    $ \ty    ->
                   vFinPoly $ \len   ->
                   let f i = literal i ty
                   in VList (Nat len) (map f (genericTake len [first, next ..])))

, ("infFrom"     , VPoly $ \ty ->
                   VFun $ \first ->
                   case cryToInteger ty first of
                     Left e -> cryError e (TVStream ty)
                     Right x -> VList Inf (map f [0 ..])
                        where f i = literal (x + i) ty)

, ("infFromThen", VPoly $ \ty ->
                   VFun $ \first ->
                   VFun $ \next ->
                   case cryToInteger ty first of
                     Left e -> cryError e (TVStream ty)
                     Right x ->
                       case cryToInteger ty next of
                         Left e -> cryError e (TVStream ty)
                         Right y -> VList Inf (map f [0 ..])
                           where f i = literal (x + diff * i) ty
                                 diff = y - x)

-- Miscellaneous:
, ("parmap"      , VPoly $ \_a ->
                   VPoly $ \_b ->
```

```haskell
                    VNumPoly $ \n ->
                    VFun $ \f ->
                    VFun $ \xs ->
                      -- Note: the reference implementation simply
                      -- executes parmap sequentially
                      let xs' = map (fromVFun f) (fromVList xs) in
                      VList n xs')

  , ("error"      , VPoly $ \a ->
                    VNumPoly $ \_ ->
                    VFun $ \_s -> cryError (UserError "error") a)
                    -- TODO: obtain error string from argument s

  , ("random"     , VPoly $ \a ->
                    VFun $ \_seed -> cryError (UserError "random: unimplemented") a)

  , ("trace"      , VNumPoly $ \_n ->
                    VPoly $ \_a ->
                    VPoly $ \_b ->
                    VFun $ \_s ->
                    VFun $ \_x ->
                    VFun $ \y -> y)
  ]

unary :: (TValue -> Value -> Value) -> Value
unary f = VPoly $ \ty -> VFun $ \x -> f ty x

binary :: (TValue -> Value -> Value -> Value) -> Value
binary f = VPoly $ \ty -> VFun $ \x -> VFun $ \y -> f ty x y

appOp1 :: (a -> Either EvalError b) -> Either EvalError a -> Either EvalError b
appOp1 _f (Left e) = Left e
appOp1 f (Right x) = f x

appOp2 :: (a -> b -> Either EvalError c) -> Either EvalError a -> Either EvalError b -> Either EvalE
appOp2 _f (Left e) _y = Left e
appOp2 _f _x (Left e) = Left e
appOp2 f (Right x) (Right y) = f x y
```

Word operations

Many Cryptol primitives take numeric arguments in the form of bitvectors. For such operations, any output bit that depends on the numeric value is strict in *all* bits of the numeric argument. This is implemented in function `fromVWord`, which converts a value from a big-endian binary format to an integer. The result is an evaluation error if any of the input bits contain an evaluation error.

```haskell
fromVWord :: Value -> Either EvalError Integer
fromVWord v = fmap bitsToInteger (mapM fromVBit (fromVList v))

-- | Convert a list of booleans in big-endian format to an integer.
bitsToInteger :: [Bool] -> Integer
```

```
bitsToInteger bs = foldl f 0 bs
  where f x b = if b then 2 * x + 1 else 2 * x

fromSignedVWord :: Value -> Either EvalError Integer
fromSignedVWord v = fmap signedBitsToInteger (mapM fromVBit (fromVList v))

-- | Convert a list of booleans in signed big-endian format to an integer.
signedBitsToInteger :: [Bool] -> Integer
signedBitsToInteger [] = evalPanic "signedBitsToInteger" ["Bitvector has zero length"]
signedBitsToInteger (b0 : bs) = foldl f (if b0 then -1 else 0) bs
  where f x b = if b then 2 * x + 1 else 2 * x
```

Function `vWord` converts an integer back to the big-endian bitvector representation. If an integer-producing function raises a run-time exception, then the output bitvector will contain the exception in all bit positions.

```
vWord :: Integer -> Either EvalError Integer -> Value
vWord w e = VList (Nat w) [ VBit (fmap (test i) e) | i <- [w-1, w-2 .. 0] ]
  where test i x = testBit x (fromInteger i)
```

Errors

The domain semantics indicate that errors can only exist at the base types. This function constructs an error representation at any type where the given error is “pushed down” into the leaf types.

```
cryError :: EvalError -> TValue -> Value
cryError e TVBit           = VBit (Left e)
cryError e TVInteger       = VInteger (Left e)
cryError e TVIntMod{}      = VInteger (Left e)
cryError e TVRational      = VRational (Left e)
cryError e TVFloat{}       = VFloat (Left e)
cryError _ TVArray{}       = evalPanic "error" ["Array type not supported in `error`"]
cryError e (TVSeq n ety)   = VList (Nat n) (genericReplicate n (cryError e ety))
cryError e (TVStream ety) = VList Inf (repeat (cryError e ety))
cryError e (TVTuple tys)   = VTuple (map (cryError e) tys)
cryError e (TVRec fields) = VRecord [ (f, cryError e fty) | (f, fty) <- canonicalFields fields ]
cryError e (TVFun _ bty)   = VFun (\_ -> cryError e bty)
cryError _ (TVAbstract{}) = evalPanic "error" ["Abstract type encountered in `error`"]
```

Zero

The `Zero` class has a single method `zero` which computes a zero value for all the built-in types for Cryptol. For bits, bitvectors and the base numeric types, this returns the obvious 0 representation. For sequences, records, and tuples, the zero method operates pointwise the underlying types. For functions, `zero` returns the constant function that returns `zero` in the codomain.

```
zero :: TValue -> Value
zero TVBit           = VBit (Right False)
zero TVInteger       = VInteger (Right 0)
zero TVIntMod{}      = VInteger (Right 0)
zero TVRational      = VRational (Right 0)
zero (TVFloat e p)   = VFloat (Right (fpToBF e p FP.bfPosZero))
zero TVArray{}       = evalPanic "zero" ["Array type not in `Zero`"]
zero (TVSeq n ety)   = VList (Nat n) (genericReplicate n (zero ety))
```

```
zero (TVStream ety) = VList Inf (repeat (zero ety))
zero (TVTuple tys)  = VTuple (map zero tys)
zero (TVRec fields) = VRecord [ (f, zero fty) | (f, fty) <- canonicalFields fields ]
zero (TVFun _ bty)  = VFun (\_ -> zero bty)
zero (TVAbstract{}) = evalPanic "zero" ["Abstract type not in `Zero`"]
```

Literals

Given a literal integer, construct a value of a type that can represent that literal.

```
literal :: Integer -> TValue -> Value
literal i = go
  where
   go TVInteger = VInteger (Right i)
   go TVRational = VRational (Right (fromInteger i))
   go (TVIntMod n)
        | i < n = VInteger (Right i)
        | otherwise = evalPanic "literal" ["Literal out of range for type Z " ++ show n]
   go (TVSeq w a)
        | isTBit a = vWord w (Right i)
   go ty = evalPanic "literal" [show ty ++ " cannot represent literals"]
```

Given a fraction, construct a value of a type that can represent that literal. The rounding flag determines the behavior if the literal cannot be represented exactly: 0 means report and error, other numbers round to the nearest representable value.

```
fraction :: Integer -> Integer -> Integer -> TValue -> Value
fraction top btm _rnd ty =
  case ty of
    TVRational -> VRational (Right (top % btm))
    TVFloat e p -> VFloat $ Right $ fpToBF e p  $ fpCheckStatus val
      where val  = FP.bfDiv opts (FP.bfFromInteger top) (FP.bfFromInteger btm)
            opts = fpOpts e p fpImplicitRound
    _ -> evalPanic "fraction" [show ty ++ " cannot represent " ++
                               show top ++ "/" ++ show btm]
```

Logic

Bitwise logic primitives are defined by recursion over the type structure. On type `Bit`, we use `fmap` and `liftA2` to make these operations strict in all arguments. For example, `True || error "foo"` does not evaluate to `True`, but yields a run-time exception. On other types, run-time exceptions on input bits only affect the output bits at the same positions.

```
logicUnary :: (Bool -> Bool) -> TValue -> Value -> Value
logicUnary op = go
  where
    go :: TValue -> Value -> Value
    go ty val =
      case ty of
        TVBit        -> VBit (fmap op (fromVBit val))
        TVSeq w ety  -> VList (Nat w) (map (go ety) (fromVList val))
        TVStream ety -> VList Inf (map (go ety) (fromVList val))
        TVTuple etys -> VTuple (zipWith go etys (fromVTuple val))
```

```
        TVRec fields -> VRecord [ (f, go fty (lookupRecord f val)) | (f, fty) <- canonicalFields fie
        TVFun _ bty  -> VFun (\v -> go bty (fromVFun val v))
        TVInteger    -> evalPanic "logicUnary" ["Integer not in class Logic"]
        TVIntMod _   -> evalPanic "logicUnary" ["Z not in class Logic"]
        TVArray{}    -> evalPanic "logicUnary" ["Array not in class Logic"]
        TVRational   -> evalPanic "logicUnary" ["Rational not in class Logic"]
        TVFloat{}    -> evalPanic "logicUnary" ["Float not in class Logic"]
        TVAbstract{} -> evalPanic "logicUnary" ["Abstract type not in `Logic`"]
```

```
logicBinary :: (Bool -> Bool -> Bool) -> TValue -> Value -> Value -> Value
logicBinary op = go
  where
    go :: TValue -> Value -> Value -> Value
    go ty l r =
      case ty of
        TVBit        -> VBit (liftA2 op (fromVBit l) (fromVBit r))
        TVSeq w ety  -> VList (Nat w) (zipWith (go ety) (fromVList l) (fromVList r))
        TVStream ety -> VList Inf (zipWith (go ety) (fromVList l) (fromVList r))
        TVTuple etys -> VTuple (zipWith3 go etys (fromVTuple l) (fromVTuple r))
        TVRec fields -> VRecord [ (f, go fty (lookupRecord f l) (lookupRecord f r))
                                | (f, fty) <- canonicalFields fields ]
        TVFun _ bty  -> VFun (\v -> go bty (fromVFun l v) (fromVFun r v))
        TVInteger    -> evalPanic "logicBinary" ["Integer not in class Logic"]
        TVIntMod _   -> evalPanic "logicBinary" ["Z not in class Logic"]
        TVArray{}    -> evalPanic "logicBinary" ["Array not in class Logic"]
        TVRational   -> evalPanic "logicBinary" ["Rational not in class Logic"]
        TVFloat{}    -> evalPanic "logicBinary" ["Float not in class Logic"]
        TVAbstract{} -> evalPanic "logicBinary" ["Abstract type not in `Logic`"]
```

Ring Arithmetic

Ring primitives may be applied to any type that is made up of finite bitvectors or one of the numeric base types. On type `[n]`, arithmetic operators are strict in all input bits, as indicated by the definition of `fromVWord`. For example, `[error "foo", True] * 2` does not evaluate to `[True, False]`, but to `[error "foo", error "foo"]`.

```
ringNullary ::
   Either EvalError Integer ->
   Either EvalError Rational ->
   (Integer -> Integer -> Either EvalError BigFloat) ->
   TValue -> Value
ringNullary i q fl = go
  where
    go :: TValue -> Value
    go ty =
      case ty of
        TVBit ->
          evalPanic "arithNullary" ["Bit not in class Ring"]
        TVInteger ->
          VInteger i
        TVIntMod n ->
          VInteger (flip mod n <$> i)
```

```haskell
        TVRational ->
          VRational q
        TVFloat e p ->
          VFloat (fpToBF e p <$> fl e p)
        TVArray{} ->
          evalPanic "arithNullary" ["Array not in class Ring"]
        TVSeq w a
          | isTBit a  -> vWord w i
          | otherwise -> VList (Nat w) (genericReplicate w (go a))
        TVStream a ->
          VList Inf (repeat (go a))
        TVFun _ ety ->
          VFun (const (go ety))
        TVTuple tys ->
          VTuple (map go tys)
        TVRec fs ->
          VRecord [ (f, go fty) | (f, fty) <- canonicalFields fs ]
        TVAbstract {} ->
          evalPanic "arithNullary" ["Abstract type not in `Ring`"]

ringUnary ::
  (Integer -> Either EvalError Integer) ->
  (Rational -> Either EvalError Rational) ->
  (Integer -> Integer -> BigFloat -> Either EvalError BigFloat) ->
  TValue -> Value -> Value
ringUnary iop qop flop = go
  where
    go :: TValue -> Value -> Value
    go ty val =
      case ty of
        TVBit ->
          evalPanic "arithUnary" ["Bit not in class Ring"]
        TVInteger ->
          VInteger $ appOp1 iop (fromVInteger val)
        TVArray{} ->
          evalPanic "arithUnary" ["Array not in class Ring"]
        TVIntMod n ->
          VInteger $ appOp1 (\i -> flip mod n <$> iop i) (fromVInteger val)
        TVRational ->
          VRational $ appOp1 qop (fromVRational val)
        TVFloat e p ->
          VFloat (fpToBF e p <$> appOp1 (flop e p) (fromVFloat val))
        TVSeq w a
          | isTBit a  -> vWord w (iop =<< fromVWord val)
          | otherwise -> VList (Nat w) (map (go a) (fromVList val))
        TVStream a ->
          VList Inf (map (go a) (fromVList val))
        TVFun _ ety ->
          VFun (\x -> go ety (fromVFun val x))
        TVTuple tys ->
          VTuple (zipWith go tys (fromVTuple val))
```

```
        TVRec fs ->
          VRecord [ (f, go fty (lookupRecord f val)) | (f, fty) <- canonicalFields fs ]
        TVAbstract {} ->
          evalPanic "arithUnary" ["Abstract type not in `Ring`"]

ringBinary ::
  (Integer -> Integer -> Either EvalError Integer) ->
  (Rational -> Rational -> Either EvalError Rational) ->
  (Integer -> Integer -> BigFloat -> BigFloat -> Either EvalError BigFloat) ->
  TValue -> Value -> Value -> Value
ringBinary iop qop flop = go
  where
    go :: TValue -> Value -> Value -> Value
    go ty l r =
      case ty of
        TVBit ->
          evalPanic "arithBinary" ["Bit not in class Ring"]
        TVInteger ->
          VInteger $ appOp2 iop (fromVInteger l) (fromVInteger r)
        TVIntMod n ->
          VInteger $ appOp2 (\i j -> flip mod n <$> iop i j) (fromVInteger l) (fromVInteger r)
        TVRational ->
          VRational $ appOp2 qop (fromVRational l) (fromVRational r)
        TVFloat e p ->
          VFloat $ fpToBF e p <$> appOp2 (flop e p) (fromVFloat l) (fromVFloat r)
        TVArray{} ->
          evalPanic "arithBinary" ["Array not in class Ring"]
        TVSeq w a
          | isTBit a  -> vWord w $ appOp2 iop (fromVWord l) (fromVWord r)
          | otherwise -> VList (Nat w) (zipWith (go a) (fromVList l) (fromVList r))
        TVStream a ->
          VList Inf (zipWith (go a) (fromVList l) (fromVList r))
        TVFun _ ety ->
          VFun (\x -> go ety (fromVFun l x) (fromVFun r x))
        TVTuple tys ->
          VTuple (zipWith3 go tys (fromVTuple l) (fromVTuple r))
        TVRec fs ->
          VRecord [ (f, go fty (lookupRecord f l) (lookupRecord f r)) | (f, fty) <- canonicalFields
        TVAbstract {} ->
          evalPanic "arithBinary" ["Abstract type not in class `Ring`"]
```

Integral

```
cryToInteger :: TValue -> Value -> Either EvalError Integer
cryToInteger ty v = case ty of
  TVInteger -> fromVInteger v
  TVSeq _ a | isTBit a -> fromVWord v
  _ -> evalPanic "toInteger" [show ty ++ " is not an integral type"]

integralBinary ::
    (Integer -> Integer -> Either EvalError Integer) ->
    TValue -> Value -> Value -> Value
```

```
integralBinary op ty x y = case ty of
  TVInteger ->
      VInteger $ appOp2 op (fromVInteger x) (fromVInteger y)
  TVSeq w a | isTBit a ->
      vWord w $ appOp2 op (fromVWord x) (fromVWord y)

  _ -> evalPanic "integralBinary" [show ty ++ " is not an integral type"]

ringExp :: TValue -> Value -> Either EvalError Integer -> Value
ringExp a _ (Left e)  = cryError e a
ringExp a v (Right i) = foldl (ringMul a) (literal 1 a) (genericReplicate i v)

ringMul :: TValue -> Value -> Value -> Value
ringMul = ringBinary (\x y -> Right (x * y))
                     (\x y -> Right (x * y))
                     (fpBin FP.bfMul fpImplicitRound)
```

Signed bitvector division (`/$`) and remainder (`%$`) are defined so that division rounds toward zero, and the remainder `x %$ y` has the same sign as `x`. Accordingly, they are implemented with Haskell's `quot` and `rem` operations.

```
divWrap :: Integer -> Integer -> Either EvalError Integer
divWrap _ 0 = Left DivideByZero
divWrap x y = Right (x `quot` y)

modWrap :: Integer -> Integer -> Either EvalError Integer
modWrap _ 0 = Left DivideByZero
modWrap x y = Right (x `rem` y)

lg2Wrap :: Integer -> Either EvalError Integer
lg2Wrap x = if x < 0 then Left LogNegative else Right (lg2 x)
```

Field

Types that represent fields are have, in addition to the ring operations a recipricol operator and a field division operator (not to be confused with integral division).

```
fieldUnary :: (Rational -> Either EvalError Rational) ->
              (Integer -> Integer -> BigFloat -> Either EvalError BigFloat) ->
              TValue -> Value -> Value
fieldUnary qop flop ty v = case ty of
  TVRational  -> VRational $ appOp1 qop (fromVRational v)
  TVFloat e p -> VFloat $ fpToBF e p <$> appOp1 (flop e p) (fromVFloat v)
  _ -> evalPanic "fieldUnary" [show ty ++ " is not a Field type"]

fieldBinary ::
   (Rational -> Rational -> Either EvalError Rational) ->
   (Integer -> Integer -> BigFloat -> BigFloat -> Either EvalError BigFloat) ->
   TValue -> Value -> Value -> Value
fieldBinary qop flop ty l r = case ty of
  TVRational -> VRational $ appOp2 qop (fromVRational l) (fromVRational r)
  TVFloat e p -> VFloat $ fpToBF e p <$> appOp2 (flop e p)
                                             (fromVFloat l) (fromVFloat r)
```

```haskell
  _ -> evalPanic "fieldBinary" [show ty ++ " is not a Field type"]

ratDiv :: Rational -> Rational -> Either EvalError Rational
ratDiv _ 0 = Left DivideByZero
ratDiv x y = Right (x / y)

ratRecip :: Rational -> Either EvalError Rational
ratRecip 0 = Left DivideByZero
ratRecip x = Right (recip x)
```

Round

```haskell
roundUnary :: (Rational -> Integer) ->
              (Integer -> Integer -> BigFloat -> Either EvalError Integer) ->
              TValue -> Value -> Value
roundUnary op flop ty v = case ty of
  TVRational -> VInteger (op <$> fromVRational v)
  TVFloat e p -> VInteger (flop e p =<< fromVFloat v)
  _ -> evalPanic "roundUnary" [show ty ++ " is not a Round type"]
```

Haskell's definition of "round" is slightly different, as it does "round to even" on ties.

```haskell
roundAwayRat :: Rational -> Integer
roundAwayRat x
  | x >= 0    = floor (x + 0.5)
  | otherwise = ceiling (x - 0.5)
```

Rational

```haskell
ratioOp :: Integer -> Integer -> Either EvalError Rational
ratioOp _ 0 = Left DivideByZero
ratioOp x y = Right (fromInteger x / fromInteger y)
```

Comparison

Comparison primitives may be applied to any type that contains a finite number of bits. All such types are compared using a lexicographic ordering on bits, where `False` < `True`. Lists and tuples are compared left-to-right, and record fields are compared in alphabetical order.

Comparisons on type `Bit` are strict in both arguments. Comparisons on larger types have short-circuiting behavior: A comparison involving an error/undefined element will only yield an error if all corresponding bits to the *left* of that position are equal.

```haskell
-- | Process two elements based on their lexicographic ordering.
cmpOrder :: (Ordering -> Bool) -> TValue -> Value -> Value -> Value
cmpOrder p ty l r = VBit (fmap p (lexCompare ty l r))

-- | Lexicographic ordering on two values.
lexCompare :: TValue -> Value -> Value -> Either EvalError Ordering
lexCompare ty l r =
  case ty of
    TVBit ->
      compare <$> fromVBit l <*> fromVBit r
```

```
    TVInteger ->
      compare <$> fromVInteger l <*> fromVInteger r
    TVIntMod _ ->
      compare <$> fromVInteger l <*> fromVInteger r
    TVRational ->
      compare <$> fromVRational l <*> fromVRational r
    TVFloat{} ->
      compare <$> fromVFloat l <*> fromVFloat r
    TVArray{} ->
      evalPanic "lexCompare" ["invalid type"]
    TVSeq _w ety ->
      lexList (zipWith (lexCompare ety) (fromVList l) (fromVList r))
    TVStream _ ->
      evalPanic "lexCompare" ["invalid type"]
    TVFun _ _ ->
      evalPanic "lexCompare" ["invalid type"]
    TVTuple etys ->
      lexList (zipWith3 lexCompare etys (fromVTuple l) (fromVTuple r))
    TVRec fields ->
      let tys    = map snd (canonicalFields fields)
          ls     = map snd (sortBy (comparing fst) (fromVRecord l))
          rs     = map snd (sortBy (comparing fst) (fromVRecord r))
       in lexList (zipWith3 lexCompare tys ls rs)
    TVAbstract {} ->
      evalPanic "lexCompare" ["Abstract type not in `Cmp`"]

lexList :: [Either EvalError Ordering] -> Either EvalError Ordering
lexList [] = Right EQ
lexList (e : es) =
  case e of
    Left err -> Left err
    Right LT -> Right LT
    Right EQ -> lexList es
    Right GT -> Right GT
```

Signed comparisons may be applied to any type made up of non-empty bitvectors. All such types are compared using a lexicographic ordering: Lists and tuples are compared left-to-right, and record fields are compared in alphabetical order.

```
signedLessThan :: TValue -> Value -> Value -> Value
signedLessThan ty l r = VBit (fmap (== LT) (lexSignedCompare ty l r))

-- | Lexicographic ordering on two signed values.
lexSignedCompare :: TValue -> Value -> Value -> Either EvalError Ordering
lexSignedCompare ty l r =
  case ty of
    TVBit ->
      evalPanic "lexSignedCompare" ["invalid type"]
    TVInteger ->
      evalPanic "lexSignedCompare" ["invalid type"]
    TVIntMod _ ->
      evalPanic "lexSignedCompare" ["invalid type"]
```

```
    TVRational ->
      evalPanic "lexSignedCompare" ["invalid type"]
    TVFloat{} ->
      evalPanic "lexSignedCompare" ["invalid type"]
    TVArray{} ->
      evalPanic "lexSignedCompare" ["invalid type"]
    TVSeq _w ety
      | isTBit ety -> compare <$> fromSignedVWord l <*> fromSignedVWord r
      | otherwise ->
        lexList (zipWith (lexSignedCompare ety) (fromVList l) (fromVList r))
    TVStream _ ->
      evalPanic "lexSignedCompare" ["invalid type"]
    TVFun _ _ ->
      evalPanic "lexSignedCompare" ["invalid type"]
    TVTuple etys ->
      lexList (zipWith3 lexSignedCompare etys (fromVTuple l) (fromVTuple r))
    TVRec fields ->
      let tys    = map snd (canonicalFields fields)
          ls     = map snd (sortBy (comparing fst) (fromVRecord l))
          rs     = map snd (sortBy (comparing fst) (fromVRecord r))
       in lexList (zipWith3 lexSignedCompare tys ls rs)
    TVAbstract {} ->
      evalPanic "lexSignedCompare" ["Abstract type not in `Cmp`"]
```

Sequences

```
-- | Split a list into 'w' pieces, each of length 'k'.
splitV :: Nat' -> Integer -> [Value] -> [Value]
splitV w k xs =
  case w of
    Nat 0 -> []
    Nat n -> VList (Nat k) ys : splitV (Nat (n - 1)) k zs
    Inf   -> VList (Nat k) ys : splitV Inf k zs
  where
    (ys, zs) = genericSplitAt k xs

-- | Transpose a list of length-'w' lists into 'w' lists.
transposeV :: Nat' -> [[Value]] -> [[Value]]
transposeV w xss =
  case w of
    Nat 0 -> []
    Nat n -> heads : transposeV (Nat (n - 1)) tails
    Inf   -> heads : transposeV Inf tails
  where
    (heads, tails) = dest xss

    -- Split a list of non-empty lists into
    -- a list of heads and a list of tails
    dest :: [[Value]] -> ([Value], [[Value]])
    dest [] = ([], [])
    dest ([] : _) = evalPanic "transposeV" ["Expected non-empty list"]
```

```haskell
    dest ((y : ys) : yss) = (y : zs, ys : zss)
      where (zs, zss) = dest yss
```

Shifting

Shift and rotate operations are strict in all bits of the shift/rotate amount, but as lazy as possible in the list values.

```haskell
shiftV :: (Nat' -> Value -> [Value] -> Integer -> [Value]) -> Value
shiftV op =
  VNumPoly $ \n ->
  VPoly $ \ix ->
  VPoly $ \a ->
  VFun $ \v ->
  VFun $ \x ->
  copyByTValue (tvSeq n a) $
  case cryToInteger ix x of
    Left e  -> cryError e (tvSeq n a)
    Right i -> VList n (op n (zero a) (fromVList v) i)

shiftLV :: Nat' -> Value -> [Value] -> Integer -> [Value]
shiftLV w z vs i =
  case w of
    Nat n -> genericDrop (min n i) vs ++ genericReplicate (min n i) z
    Inf   -> genericDrop i vs

shiftRV :: Nat' -> Value -> [Value] -> Integer -> [Value]
shiftRV w z vs i =
  case w of
    Nat n -> genericReplicate (min n i) z ++ genericTake (n - min n i) vs
    Inf   -> genericReplicate i z ++ vs

rotateV :: (Integer -> [Value] -> Integer -> [Value]) -> Value
rotateV op =
  vFinPoly $ \n ->
  VPoly $ \ix ->
  VPoly $ \a ->
  VFun $ \v ->
  VFun $ \x ->
  copyByTValue (TVSeq n a) $
  case cryToInteger ix x of
    Left e  -> cryError e (tvSeq (Nat n) a)
    Right i -> VList (Nat n) (op n (fromVList v) i)

rotateLV :: Integer -> [Value] -> Integer -> [Value]
rotateLV 0 vs _ = vs
rotateLV w vs i = ys ++ xs
  where (xs, ys) = genericSplitAt (i `mod` w) vs

rotateRV :: Integer -> [Value] -> Integer -> [Value]
rotateRV 0 vs _ = vs
rotateRV w vs i = ys ++ xs
```

```haskell
  where (xs, ys) = genericSplitAt ((w - i) `mod` w) vs

signedShiftRV :: Value
signedShiftRV =
  VNumPoly $ \(Nat n) ->
  VPoly $ \ix ->
  VFun $ \v ->
  VFun $ \x ->
  copyByTValue (tvSeq (Nat n) TVBit) $
  case cryToInteger ix x of
    Left e -> cryError e (tvSeq (Nat n) TVBit)
    Right i -> VList (Nat n) $
      let vs = fromVList v
          z = head vs in
      genericReplicate (min n i) z ++ genericTake (n - min n i) vs
```

Indexing

Indexing operations are strict in all index bits, but as lazy as possible in the list values. An index greater than or equal to the length of the list produces a run-time error.

```haskell
-- | Indexing operations that return one element.
indexPrimOne :: (Nat' -> TValue -> [Value] -> Integer -> Value) -> Value
indexPrimOne op =
  VNumPoly $ \n ->
  VPoly $ \a ->
  VPoly $ \ix ->
  VFun $ \l ->
  VFun $ \r ->
  copyByTValue a $
  case cryToInteger ix r of
    Left e -> cryError e a
    Right i -> op n a (fromVList l) i

indexFront :: Nat' -> TValue -> [Value] -> Integer -> Value
indexFront w a vs ix =
  case w of
    Nat n | n <= ix -> cryError (InvalidIndex (Just ix)) a
    _               -> genericIndex vs ix

indexBack :: Nat' -> TValue -> [Value] -> Integer -> Value
indexBack w a vs ix =
  case w of
    Nat n | n > ix    -> genericIndex vs (n - ix - 1)
          | otherwise -> cryError (InvalidIndex (Just ix)) a
    Inf               -> evalPanic "indexBack" ["unexpected infinite sequence"]

updatePrim :: (Nat' -> [Value] -> Integer -> Value -> [Value]) -> Value
updatePrim op =
  VNumPoly $ \len ->
  VPoly $ \eltTy ->
  VPoly $ \ix ->
```

```haskell
  VFun $ \xs ->
  VFun $ \idx ->
  VFun $ \val ->
  copyByTValue (tvSeq len eltTy) $
  case cryToInteger ix idx of
    Left e -> cryError e (tvSeq len eltTy)
    Right i
      | Nat i < len -> VList len (op len (fromVList xs) i val)
      | otherwise   -> cryError (InvalidIndex (Just i)) (tvSeq len eltTy)

updateFront :: Nat' -> [Value] -> Integer -> Value -> [Value]
updateFront _ vs i x = updateAt vs i x

updateBack :: Nat' -> [Value] -> Integer -> Value -> [Value]
updateBack Inf _vs _i _x = evalPanic "Unexpected infinite sequence in updateEnd" []
updateBack (Nat n) vs i x = updateAt vs (n - i - 1) x

updateAt :: [a] -> Integer -> a -> [a]
updateAt []       _ _ = []
updateAt (_ : xs) 0 y = y : xs
updateAt (x : xs) i y = x : updateAt xs (i - 1) y
```

Floating Point Numbers

Whenever we do operations that do not have an explicit rounding mode, we round towards the closest number, with ties resolved to the even one.

```haskell
fpImplicitRound :: FP.RoundMode
fpImplicitRound = FP.NearEven
```

The function `fpOpts` deifnes an object specifying the precision and rounding mode to be used during a floating point computation. We assume that its aruments are known to be within the limits imposed by the concrete library used to perform the computation.

```haskell
fpOpts :: Integer -> Integer -> FP.RoundMode -> FP.BFOpts
fpOpts e p r = FP.expBits (fromInteger e)  <>
               FP.precBits (fromInteger p) <>
               FP.rnd r
```

Most floating point computations produce a result and some additional information about what happened while computing. The function `fpCheckStatus` examines this status and reports erorrs in unusual situations.

```haskell
fpCheckStatus :: (FP.BigFloat, FP.Status) -> FP.BigFloat
fpCheckStatus (r,s) =
  case s of
    FP.MemError -> panic "fpCheckStatus" [ "Out of memory" ]
    _           -> r
```

We annotate floating point values with their precision. This is only used when pretty printing values.

```haskell
fpToBF :: Integer -> Integer -> BigFloat -> BF
fpToBF e p x = BF { bfValue = x, bfExpWidth = e, bfPrecWidth = p }
```

The following two functions convert between floaitng point numbers and integers.

```
fpFromInteger :: Integer -> Integer -> Integer -> BigFloat
fpFromInteger e p = fpCheckStatus . FP.bfRoundFloat opts . FP.bfFromInteger
  where opts = fpOpts e p fpImplicitRound
```

```
fpToInteger ::
  String -> FP.RoundMode ->
  Integer -> Integer ->
  BigFloat -> Either EvalError Integer
fpToInteger fun r _ _ f =
  case fpToI r f of
    Nothing -> Left (BadValue fun)
    Just i  -> Right i
```

This just captures a common pattern for binary floating point primitives.

```
fpBin :: (FP.BFOpts -> BigFloat -> BigFloat -> (BigFloat,FP.Status)) ->
         FP.RoundMode -> Integer -> Integer ->
         BigFloat -> BigFloat -> Either EvalError BigFloat
fpBin f r e p x y = Right (fpCheckStatus (f (fpOpts e p r) x y))
```

Computes the reciprocal of a floating point number via division. This assumes that 1 can be represented exactly, which should be true for all supported precisions.

```
fpRecip :: Integer -> Integer -> BigFloat -> Either EvalError BigFloat
fpRecip e p x = Right (fpCheckStatus (FP.bfDiv opts (FP.bfFromInteger 1) x))
  where opts = fpOpts e p fpImplicitRound
```

```
floatPrimTable :: Map PrimIdent Value
floatPrimTable = Map.fromList $ map (\(n, v) -> (floatPrim (T.pack n), v))
   [ "fpNaN"       ~> vFinPoly \e -> vFinPoly \p ->
                      VFloat $ Right $ fpToBF e p FP.bfNaN

   , "fpPosInf"    ~> vFinPoly \e -> vFinPoly \p ->
                      VFloat $ Right $ fpToBF e p FP.bfPosInf

   , "fpFromBits"  ~> vFinPoly \e -> vFinPoly \p -> VFun \bvv ->
                      VFloat (floatFromBits e p <$> fromVWord bvv)

   , "fpToBits"    ~> vFinPoly \e -> vFinPoly \p -> VFun \fpv ->
                      vWord (e + p) (floatToBits e p <$> fromVFloat fpv)

   , "=.="         ~> vFinPoly \_ -> vFinPoly \_ -> VFun \xv -> VFun \yv ->
                      VBit do x <- fromVFloat xv
                              y <- fromVFloat yv
                              pure (FP.bfCompare x y == EQ)

   , "fpAdd"      ~> fpArith FP.bfAdd
   , "fpSub"      ~> fpArith FP.bfSub
   , "fpMul"      ~> fpArith FP.bfMul
   , "fpDiv"      ~> fpArith FP.bfDiv
   ]
  where
  (~>) = (,)
  fpArith f = vFinPoly \e -> vFinPoly \p ->
```

```
              VFun \vr -> VFun \xv -> VFun \yv ->
              VFloat do r <- fromVWord vr
                        rnd <- case fpRound r of
                                 Just r' -> pure r'
                                 Nothing -> Left (BadRoundingMode r)
                        x <- fromVFloat xv
                        y <- fromVFloat yv
                        fpToBF e p <$> fpBin f rnd e p x y
```

Error Handling

The `evalPanic` function is only called if an internal data invariant is violated, such as an expression that is not well-typed. Panics should (hopefully) never occur in practice; a panic message indicates a bug in Cryptol.

```
evalPanic :: String -> [String] -> a
evalPanic cxt = panic ("[Reference Evaluator]" ++ cxt)
```

Pretty Printing

```
ppValue :: PPOpts -> Value -> Doc
ppValue opts val =
  case val of
    VBit b     -> text (either show show b)
    VInteger i -> text (either show show i)
    VRational q -> text (either show show q)
    VFloat fl -> text (either show (show . fpPP opts) fl)
    VList l vs ->
      case l of
        Inf -> ppList (map (ppValue opts) (take (useInfLength opts) vs) ++ [text "..."])
        Nat n ->
          -- For lists of defined bits, print the value as a numeral.
          case traverse isBit vs of
            Just bs -> ppBV opts (mkBv n (bitsToInteger bs))
            Nothing -> ppList (map (ppValue opts) vs)
      where ppList docs = brackets (fsep (punctuate comma docs))
            isBit v = case v of VBit (Right b) -> Just b
                                _              -> Nothing
    VTuple vs  -> parens (sep (punctuate comma (map (ppValue opts) vs)))
    VRecord fs -> braces (sep (punctuate comma (map ppField fs)))
      where ppField (f,r) = pp f <+> char '=' <+> ppValue opts r
    VFun _     -> text "<function>"
    VPoly _    -> text "<polymorphic value>"
    VNumPoly _ -> text "<polymorphic value>"
```

Module Command

This module implements the core functionality of the `:eval <expression>` command for the Cryptol REPL, which prints the result of running the reference evaluator on an expression.

```
evaluate :: Expr -> M.ModuleCmd Value
evaluate expr (_,modEnv) = return (Right (evalExpr env expr, modEnv), [])
```

```haskell
where
  extDgs = concatMap mDecls (M.loadedModules modEnv)
  env = foldl evalDeclGroup mempty extDgs
```