Contents


```
-- |
-- Module      :  Cryptol.Eval.Reference
-- Description :  The reference implementation of the Cryptol evaluation semantics.
-- Copyright   :  (c) 2013-2020 Galois, Inc.
-- License     :  BSD3
-- Maintainer  :  cryptol@galois.com
-- Stability   :  provisional
-- Portability :  portable

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

module Cryptol.Eval.Reference
  ( Value(..)
  , E(..)
```

```
  , evaluate
  , evalExpr
  , evalDeclGroup
  , ppValue
  , ppEValue
  ) where

import Data.Bits
import Data.Ratio((%))
import Data.List
  (genericIndex, genericLength, genericReplicate, 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 qualified GHC.Integer.GMP.Internals as Integer

import Cryptol.ModuleSystem.Name (asPrim)
import Cryptol.TypeCheck.Solver.InfNat (Nat'(..), nAdd, nMin, nMul)
import Cryptol.TypeCheck.AST
import Cryptol.Backend.FloatHelpers (BF(..))
import qualified Cryptol.Backend.FloatHelpers as FP
import Cryptol.Backend.Monad (EvalError(..))
import Cryptol.Eval.Type
  (TValue(..), isTBit, evalValType, evalNumType, TypeEnv, bindTypeVar)
import Cryptol.Eval.Concrete (mkBv, ppBV, lg2)
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,loadedNewtypes)
```

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`
`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 (closed) Cryptol value type `t` as a complete partial order (cpo) $M(\texttt{t})$. The values of $M(\texttt{t})$ represent the *values* present in the type `t`; we distinguish these from the *computations* at type `t`. Operationally, the difference is that computations may raise errors or cause nontermination when evaluated; however, values are already evaluated, and will not cause errors or nontermination. Denotationally, we represent this difference via a monad (in the style of Moggi) called E. As an operation on CPOs, E adds a new bottom element representing nontermination, and a collection of erroneous values representing various runtime error conditions.

To each Cryptol expression `e : t` we assign a meaning $M(\texttt{e})$ in $E(M(\texttt{t}))$; in particular, recursive Cryptol programs of type `t` are modeled as least fixed points in $E(M(\texttt{t}))$. In other words, this is a domain-theoretic denotational semantics. Note, we do not requre CPOs defined via $M(\texttt{t})$ to have bottom elements, which is why we must take fixpoints in E. We cannot directly represent values without bottom in Haskell, so instead we are careful in this document only to write clearly-terminating functions, unless they represent computations under E.

$M(\texttt{Bit})$ is a discrete cpo with values for `True`, `False`, which we simply represent in Haskell as `Bool`. Similarly, $M(\texttt{Integer})$ is a discrete cpo with values for integers, which we model as Haskell's `Integer`. Likewise with the other base types.

The value cpos for lists, tuples, and records are cartesian products of *computations*. For example $M(\texttt{(a,b)}) = E(M(\texttt{a})) \times E(M(\texttt{b}))$. The cpo ordering is pointwise. The trivial types `[0]t`, `()` and `{}` denote single-element cpos. $M(\texttt{a -> b})$ is the continuous function space $E(M(\texttt{a})) \to E(M(\texttt{b}))$.

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 $E(M(\texttt{t[t1/a1 ... tn/an]}))$.

Computation Monad

This monad represents either an evaluated thing of type `a` or an evaluation error. In the reference interpreter, only things under this monad should potentially result in errors or fail to terminate.

```
-- | Computation monad for the reference evaluator.
data E a = Value !a | Err EvalError

instance Functor E where
  fmap f (Value x) = Value (f x)
  fmap _ (Err e)   = Err e
```

```
instance Applicative E where
  pure x = Value x
  Err e   <*> _       = Err e
  Value _ <*> Err e   = Err e
  Value f <*> Value x = Value (f x)

instance Monad E where
  m >>= f =
    case m of
      Value x -> f x
      Err r   -> Err r

eitherToE :: Either EvalError a -> E a
eitherToE (Left e)  = Err e
eitherToE (Right x) = pure x
```

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 !Bool                 -- ^ @ Bit    @ booleans
  | VInteger !Integer          -- ^ @ Integer @  or @Z n@ integers
  | VRational !Rational        -- ^ @ Rational @ rationals
  | VFloat !BF                 -- ^ Floating point numbers
  | VList Nat' [E Value]       -- ^ @ [n]a   @ finite or infinite lists
  | VTuple [E Value]           -- ^ @ ( .. ) @ tuples
  | VRecord [(Ident, E Value)] -- ^ @ { .. } @ records
  | VFun (E Value -> E Value)  -- ^ functions
  | VPoly (TValue -> E Value)  -- ^ polymorphic values (kind *)
  | VNumPoly (Nat' -> E Value) -- ^ polymorphic values (kind #)
```

Operations on Values

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

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

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

fromVFloat :: Value -> BigFloat
```

```
fromVFloat = bfValue . fromVFloat'

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

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

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

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

-- | Destructor for @VFun@.
fromVFun :: Value -> (E Value -> E Value)
fromVFun (VFun f) = f
fromVFun _        = evalPanic "fromVFun" ["Expected a function"]

-- | Look up a field in a record.
lookupRecord :: Ident -> Value -> E 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 -> E 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.

```
data Env = Env
  { envVars       :: !(Map Name (E Value))
  , envTypes      :: !TypeEnv
  }
```

```
instance Semigroup Env where
  l <> r = Env
    { envVars  = envVars  l <> envVars  r
    , envTypes = envTypes l <> envTypes r
    }

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

-- | Bind a variable in the evaluation environment.
bindVar :: (Name, E 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 = bindTypeVar 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.

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

    ELocated _ e -> evalExpr env e

    EList es _ty  ->
      pure $ VList (Nat (genericLength es)) [ evalExpr env e | e <- es ]

    ETuple es     ->
      pure $ VTuple [ evalExpr env e | e <- es ]

    ERec fields   ->
      pure $ VRecord [ (f, evalExpr env e) | (f, e) <- canonicalFields fields ]

    ESel e sel    ->
      evalSel sel =<< evalExpr env e

    ESet ty e sel v ->
      evalSet (evalValType (envTypes env) ty)
              (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 -> pure $ VPoly $ \ty ->
                   evalExpr (bindType (tpVar tv) (Right ty) env) b
        KNum  -> pure $ VNumPoly $ \n ->
                   evalExpr (bindType (tpVar tv) (Left n) env) b
        k     -> evalPanic "evalExpr" ["Invalid kind on type abstraction", show k]

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

    EApp e1 e2     -> appFun (evalExpr env e1) (evalExpr env e2)
    EAbs n _ty b   -> pure $ 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

appFun :: E Value -> E Value -> E Value
appFun f v = f >>= \f' -> fromVFun f' v
```

Selectors

Apply the the given selector form to the given value. Note that record selectors work uniformly on both record types and on newtypes.

```
evalSel :: Selector -> Value -> E Value
evalSel sel val =
  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. Note that record selectors work uniformly on both record types and on newtypes.

```
evalSet :: TValue -> E Value -> Selector -> E Value -> E Value
evalSet tyv val sel fval =
  case (tyv, sel) of
    (TVTuple ts, TupleSel n _) -> updTupleAt ts n
    (TVRec fs, RecordSel n _)  -> updRecAt fs n
    (TVNewtype _ _ fs, RecordSel n _) -> updRecAt fs n
    (TVSeq len _, ListSel n _) -> updSeqAt len n
    (_, _) -> evalPanic "evalSet" ["type/selector mismatch", show tyv, show sel]
  where
    updTupleAt ts n =
      pure $ VTuple
          [ if i == n then fval else
              do vs <- fromVTuple <$> val
                 genericIndex vs i
          | (i,_t) <- zip [0 ..] ts
          ]

    updRecAt fs n =
      pure $ VRecord
          [ (f, if f == n then fval else lookupRecord f =<< val)
          | (f, _t) <- canonicalFields fs
          ]

    updSeqAt len n =
      pure $ generateV (Nat len) $ \i ->
        if i == toInteger n then fval else
             do vs <- fromVList <$> val
                indexFront (Nat len) vs i
```

Conditionals

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

```
condValue :: E Bool -> E Value -> E Value -> E Value
condValue c l r = c >>= \b -> if b then l else r
```

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 nm len _ty expr -> [ bindVar (nm, get i) env | i <- idxs ]
     where
      get i =
        do v <- evalExpr env expr
           genericIndex (fromVList v) i

      idxs :: [Integer]
      idxs =
        case evalNumType (envTypes env) len of
        Inf   -> [0 ..]
        Nat n -> [0 .. n-1]

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
         -> E Value
evalComp env expr branches =
    pure $ 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 (evalDecl env') ds
      in env'

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

Newtypes

At runtime, newtypes values are represented in exactly the same way as records. The constructor function for newtypes is thus basically just an identity function that consumes and ignores its type arguments.

```
evalNewtypeDecl :: Env -> Newtype -> Env
evalNewtypeDecl env nt = bindVar (ntName nt, pure val) env
  where
    val = foldr tabs con (ntParams nt)
    con = VFun (\x -> x)
    tabs tp body =
      case tpKind tp of
        KType -> VPoly (\_ -> pure body)
        KNum  -> VNumPoly (\_ -> pure body)
        k -> evalPanic "evalNewtypeDecl" ["illegal newtype parameter kind", show k]
```

Primitives

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

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

Cryptol primitives fall into several groups, mostly delineated by corresponding type classes:

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

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

infixr 0 ~>
(~>) :: String -> a -> (String,a)
nm ~> v = (nm,v)

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

  -- Literals
  [ "True"       ~> VBit True
  , "False"      ~> VBit False
  , "number"     ~> vFinPoly $ \val -> pure $
                    VPoly $ \a ->
                    literal val a
  , "fraction"   ~> vFinPoly \top -> pure $
                    vFinPoly \bot -> pure $
                    vFinPoly \rnd -> pure $
```

```
                  VPoly    \a   -> fraction top bot rnd a
-- Zero
, "zero"        ~> VPoly (pure . zero)

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

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

-- Integral
, "toInteger"   ~> VPoly $ \a -> pure $
                  VFun $ \x ->
                  VInteger <$> cryToInteger a x
, "/"           ~> binary (integralBinary divWrap)
, "%"           ~> binary (integralBinary modWrap)
, "^^"          ~> VPoly $ \aty -> pure $
                  VPoly $ \ety -> pure $
                  VFun $ \a -> pure $
                  VFun $ \e ->
                    ringExp aty a =<< cryToInteger ety e

-- Field
, "/."          ~> binary (fieldBinary ratDiv zDiv
                                       (fpBin FP.bfDiv fpImplicitRound)
                           )

, "recip"       ~> unary (fieldUnary ratRecip zRecip fpRecip)
```

```
-- Round
, "floor"       ~> unary (roundUnary floor
                    (eitherToE . FP.floatToInteger "floor" FP.ToNegInf))

, "ceiling"     ~> unary (roundUnary ceiling
                    (eitherToE . FP.floatToInteger "ceiling" FP.ToPosInf))

, "trunc"       ~> unary (roundUnary truncate
                    (eitherToE . FP.floatToInteger "trunc" FP.ToZero))

, "roundAway"   ~> unary (roundUnary roundAwayRat
                    (eitherToE . FP.floatToInteger "roundAway" FP.Away))

, "roundToEven"~> unary (roundUnary round
                    (eitherToE . FP.floatToInteger "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 -> pure $
                   VFun $ \l -> pure $
                   VFun $ \r ->
                     vWord n <$> appOp2 divWrap
                                   (fromSignedVWord =<< l)
                                   (fromSignedVWord =<< r)
, "%$"          ~> vFinPoly $ \n -> pure $
                   VFun $ \l -> pure $
                   VFun $ \r ->
                     vWord n <$> appOp2 modWrap
                                   (fromSignedVWord =<< l)
                                   (fromSignedVWord =<< r)
, ">>$"         ~> signedShiftRV
, "lg2"         ~> vFinPoly $ \n -> pure $
                   VFun $ \v ->
                     vWord n <$> appOp1 lg2Wrap (fromVWord =<< v)
-- Rational
, "ratio"       ~> VFun $ \l -> pure $
                   VFun $ \r ->
                   VRational <$> appOp2 ratioOp
                                   (fromVInteger <$> l)
                                   (fromVInteger <$> r)

-- Z n
```

```
  , "fromZ"       ~> vFinPoly $ \n -> pure $
                     VFun $ \x ->
                     VInteger . flip mod n . fromVInteger <$> x

  -- Sequences
  , "#"           ~> vFinPoly $ \front -> pure $
                     VNumPoly $ \back  -> pure $
                     VPoly $ \_elty  -> pure $
                     VFun $ \l -> pure $
                     VFun $ \r ->
                       pure $ generateV (nAdd (Nat front) back) $ \i ->
                         if i < front then
                           do l' <- fromVList <$> l
                              indexFront (Nat front) l' i
                          else
                           do r' <- fromVList <$> r
                              indexFront back r' (i - front)

  , "join"        ~> VNumPoly $ \parts -> pure $
                     vFinPoly $ \each  -> pure $
                     VPoly $ \_a -> pure $
                     VFun $ \v ->
                       pure $ generateV (nMul parts (Nat each)) $ \i ->
                         do let (q,r) = divMod i each
                            xss <- fromVList <$> v
                            xs  <- fromVList <$> indexFront parts xss q
                            indexFront (Nat each) xs r

  , "split"       ~> VNumPoly $ \parts -> pure $
                     vFinPoly $ \each  -> pure $
                     VPoly $ \_a -> pure $
                     VFun $ \val ->
                       pure $ generateV parts $ \i ->
                         pure $ generateV (Nat each) $ \j ->
                           do vs <- fromVList <$> val
                              indexFront (nMul parts (Nat each)) vs (i * each + j)

  , "take"        ~> VNumPoly $ \front -> pure $
                     VNumPoly $ \back  -> pure $
                     VPoly    $ \_a    -> pure $
                     VFun     $ \v ->
                       pure $ generateV front $ \i ->
                                do vs <- fromVList <$> v
                                   indexFront (nAdd front back) vs i

  , "drop"        ~> vFinPoly $ \front -> pure $
                     VNumPoly $ \back -> pure $
                     VPoly $ \_a -> pure $
                     VFun $ \v ->
                       pure $ generateV back $ \i ->
                                do vs <- fromVList <$> v
```

```
                                indexFront (nAdd (Nat front) back) vs (front+i)

  , "reverse"    ~> vFinPoly $ \n -> pure $
                    VPoly $ \_a -> pure $
                    VFun $ \v ->
                      pure $ generateV (Nat n) $ \i ->
                        do vs <- fromVList <$> v
                           indexBack (Nat n) vs i

  , "transpose"  ~> VNumPoly $ \rows -> pure $
                    VNumPoly $ \cols -> pure $
                    VPoly $ \_a -> pure $
                    VFun $ \val ->
                      pure $ generateV cols $ \c ->
                        pure $ generateV rows $ \r ->
                        do xss <- fromVList <$> val
                           xs <- fromVList <$> indexFront rows xss r
                           indexFront cols xs c

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

  -- Indexing:
  , "@"          ~> indexPrimOne  indexFront
  , "!"          ~> indexPrimOne  indexBack
  , "update"     ~> updatePrim updateFront
  , "updateEnd"  ~> updatePrim updateBack

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

  , "fromToLessThan" ~>
                    vFinPoly $ \first -> pure $
                    VNumPoly $ \bound -> pure $
                    VPoly    $ \ty    -> pure $
                    let f i = literal i ty in
                    case bound of
                      Inf -> VList Inf (map f [first ..])
                      Nat bound' ->
                        let len = bound' - first in
                        VList (Nat len) (map f (genericTake len [first ..]))

  , "fromToBy"   ~> vFinPoly $ \first  -> pure $
                    vFinPoly $ \lst    -> pure $
```

```
                  vFinPoly $ \stride -> pure $
                  VPoly    $ \ty     -> pure $
                  let f i = literal i ty in
                  let vs  = [ f (first + i*stride) | i <- [0..] ] in
                  let len = 1 + ((lst-first) `div` stride) in
                  VList (Nat len) (genericTake len vs)

, "fromToByLessThan" ~>
                  vFinPoly $ \first  -> pure $
                  VNumPoly $ \bound  -> pure $
                  vFinPoly $ \stride -> pure $
                  VPoly    $ \ty     -> pure $
                  let f i = literal i ty in
                  let vs  = [ f (first + i*stride) | i <- [0..] ] in
                  case bound of
                    Inf -> VList Inf vs
                    Nat bound' ->
                      let len = (bound'-first+stride-1) `div` stride in
                      VList (Nat len) (genericTake len vs)

, "fromToDownBy" ~>
                  vFinPoly $ \first  -> pure $
                  vFinPoly $ \lst    -> pure $
                  vFinPoly $ \stride -> pure $
                  VPoly    $ \ty     -> pure $
                  let f i = literal i ty in
                  let vs  = [ f (first - i*stride) | i <- [0..] ] in
                  let len = 1 + ((first-lst) `div` stride) in
                  VList (Nat len) (genericTake len vs)

, "fromToDownByGreaterThan" ~>
                  vFinPoly $ \first  -> pure $
                  vFinPoly $ \lst    -> pure $
                  vFinPoly $ \stride -> pure $
                  VPoly    $ \ty     -> pure $
                  let f i = literal i ty in
                  let vs  = [ f (first - i*stride) | i <- [0..] ] in
                  let len = (first-lst+stride-1) `div` stride in
                  VList (Nat len) (genericTake len vs)

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

, "infFrom"    ~> VPoly $ \ty -> pure $
                  VFun $ \first ->
```

```
                    do x <- cryToInteger ty first
                       let f i = literal (x + i) ty
                       pure $ VList Inf (map f [0 ..])

  , "infFromThen"~> VPoly $ \ty -> pure $
                    VFun $ \first -> pure $
                    VFun $ \next ->
                    do x <- cryToInteger ty first
                       y <- cryToInteger ty next
                       let diff = y - x
                           f i = literal (x + diff * i) ty
                       pure $ VList Inf (map f [0 ..])

  -- Miscellaneous:
  , "parmap"     ~> VPoly $ \_a -> pure $
                    VPoly $ \_b -> pure $
                    VNumPoly $ \n -> pure $
                    VFun $ \f -> pure $
                    VFun $ \xs ->
                       do f' <- fromVFun <$> f
                          xs' <- fromVList <$> xs
                          -- Note: the reference implementation simply
                          -- executes parmap sequentially
                          pure $ VList n (map f' xs')

  , "error"      ~> VPoly $ \_a -> pure $
                    VNumPoly $ \_ -> pure $
                    VFun $ \s ->
                      do msg <- evalString s
                         cryError (UserError msg)

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

  , "trace"      ~> VNumPoly $ \_n -> pure $
                    VPoly $ \_a -> pure $
                    VPoly $ \_b -> pure $
                    VFun $ \s -> pure $
                    VFun $ \x -> pure $
                    VFun $ \y ->
                       do _ <- evalString s -- evaluate and ignore s
                          _ <- x -- evaluate and ignore x
                          y
  ]


evalString :: E Value -> E String
evalString v =
  do cs <- fromVList <$> v
     ws <- mapM (fromVWord =<<) cs
     pure (map (toEnum . fromInteger) ws)
```

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

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

appOp1 :: (a -> E b) -> E a -> E b
appOp1 f x =
  do x' <- x
     f x'

appOp2 :: (a -> b -> E c) -> E a -> E b -> E c
appOp2 f x y =
  do x' <- x
     y' <- 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.

```
fromVWord :: Value -> E Integer
fromVWord v = bitsToInteger <$> traverse (fmap 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 -> E Integer
fromSignedVWord v = signedBitsToInteger <$> traverse (fmap 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.

```
vWord :: Integer -> Integer -> Value
vWord w e
  | w > toInteger (maxBound :: Int) =
      evalPanic "vWord" ["Word length too large", show w]
  | otherwise =
```

```
      VList (Nat w) [ mkBit i | i <- [w-1, w-2 .. 0 ] ]
 where
   mkBit i = pure (VBit (testBit e (fromInteger i)))
```

Errors

```
cryError :: EvalError -> E a
cryError e = Err e
```

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 False
zero TVInteger      = VInteger 0
zero TVIntMod{}     = VInteger 0
zero TVRational     = VRational 0
zero (TVFloat e p)  = VFloat (fpToBF e p FP.bfPosZero)
zero TVArray{}      = evalPanic "zero" ["Array type not in `Zero`"]
zero (TVSeq n ety)  = VList (Nat n) (genericReplicate n (pure (zero ety)))
zero (TVStream ety) = VList Inf (repeat (pure (zero ety)))
zero (TVTuple tys)  = VTuple (map (pure . zero) tys)
zero (TVRec fields) = VRecord [ (f, pure (zero fty))
                              | (f, fty) <- canonicalFields fields ]
zero (TVFun _ bty)  = VFun (\_ -> pure (zero bty))
zero (TVAbstract{}) = evalPanic "zero" ["Abstract type not in `Zero`"]
zero (TVNewtype{})  = evalPanic "zero" ["Newtype not in `Zero`"]
```

Literals

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

```
literal :: Integer -> TValue -> E Value
literal i = go
  where
   go TVInteger  = pure (VInteger i)
   go TVRational = pure (VRational (fromInteger i))
   go (TVIntMod n)
        | i < n = pure (VInteger i)
        | otherwise = evalPanic "literal"
                                ["Literal out of range for type Z " ++ show n]
   go (TVSeq w a)
        | isTBit a = pure (vWord w 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.

```
-- TODO: we should probably be using the rounding mode here...
fraction :: Integer -> Integer -> Integer -> TValue -> E Value
fraction top btm _rnd ty =
  case ty of
    TVRational -> pure (VRational (top % btm))
    TVFloat e p -> pure $ VFloat $ fpToBF e p  $ FP.fpCheckStatus val
      where val  = FP.bfDiv opts (FP.bfFromInteger top) (FP.bfFromInteger btm)
            opts = FP.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`, the operations are 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 -> E Value -> E Value
logicUnary op = go
  where
    go :: TValue -> E Value -> E Value
    go ty val =
      case ty of
        TVBit        -> VBit . 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 ->
             do val' <- val
                pure $ VRecord [ (f, go fty (lookupRecord f val'))
                               | (f, fty) <- canonicalFields fields ]
        TVFun _ bty  -> pure $ VFun (\v -> go bty (appFun 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`"]
        TVNewtype{}  -> evalPanic "logicUnary" ["Newtype not in `Logic`"]

logicBinary :: (Bool -> Bool -> Bool) -> TValue -> E Value -> E Value -> E Value
logicBinary op = go
  where
    go :: TValue -> E Value -> E Value -> E Value
    go ty l r =
      case ty of
        TVBit ->
          VBit <$> (op <$> (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 ->
         do l' <- l
            r' <- r
            pure $ VRecord
                 [ (f, go fty (lookupRecord f l') (lookupRecord f r'))
                 | (f, fty) <- canonicalFields fields
                 ]
      TVFun _ bty  -> pure $ VFun $ \v ->
                           do l' <- l
                              r' <- r
                              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`"]
      TVNewtype{}  -> evalPanic "logicBinary" ["Newtype 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"`.

```
ringNullary ::
   E Integer ->
   E Rational ->
   (Integer -> Integer -> E BigFloat) ->
   TValue -> E Value
ringNullary i q fl = go
  where
    go :: TValue -> E Value
    go ty =
      case ty of
        TVBit ->
          evalPanic "arithNullary" ["Bit not in class Ring"]
        TVInteger ->
          VInteger <$> i
        TVIntMod n ->
          VInteger . flip mod n <$> i
        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 -> pure $ VList (Nat w) (genericReplicate w (go a))
        TVStream a ->
          pure $ VList Inf (repeat (go a))
        TVFun _ ety ->
          pure $ VFun (const (go ety))
        TVTuple tys ->
          pure $ VTuple (map go tys)
        TVRec fs ->
          pure $ VRecord [ (f, go fty) | (f, fty) <- canonicalFields fs ]
        TVAbstract {} ->
          evalPanic "arithNullary" ["Abstract type not in `Ring`"]
        TVNewtype {} ->
          evalPanic "arithNullary" ["Newtype type not in `Ring`"]

ringUnary ::
  (Integer -> E Integer) ->
  (Rational -> E Rational) ->
  (Integer -> Integer -> BigFloat -> E BigFloat) ->
  TValue -> E Value -> E Value
ringUnary iop qop flop = go
  where
    go :: TValue -> E Value -> E 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 ->
          pure $ VFun (\x -> go ety (appFun val x))
        TVTuple tys ->
```

```
          VTuple . zipWith go tys . fromVTuple <$> val
        TVRec fs ->
          do val' <- val
             pure $ VRecord [ (f, go fty (lookupRecord f val'))
                            | (f, fty) <- canonicalFields fs ]
        TVAbstract {} ->
          evalPanic "arithUnary" ["Abstract type not in `Ring`"]
        TVNewtype {} ->
          evalPanic "arithUnary" ["Newtype not in `Ring`"]

ringBinary ::
  (Integer -> Integer -> E Integer) ->
  (Rational -> Rational -> E Rational) ->
  (Integer -> Integer -> BigFloat -> BigFloat -> E BigFloat) ->
  TValue -> E Value -> E Value -> E Value
ringBinary iop qop flop = go
  where
    go :: TValue -> E Value -> E Value -> E 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 <$
        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 ->
          pure $ VFun (\x -> go ety (appFun l x) (appFun r x))
        TVTuple tys ->
          VTuple <$> (zipWith3 go tys <$>
                      (fromVTuple <$> l) <*>
                      (fromVTuple <$> r))
        TVRec fs ->
          do l' <- l
```

```
            r' <- r
            pure $ VRecord
              [ (f, go fty (lookupRecord f l') (lookupRecord f r'))
              | (f, fty) <- canonicalFields fs ]
        TVAbstract {} ->
          evalPanic "arithBinary" ["Abstract type not in class `Ring`"]
        TVNewtype {} ->
          evalPanic "arithBinary" ["Newtype not in class `Ring`"]
```

Integral

```
cryToInteger :: TValue -> E Value -> E 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 -> E Integer) ->
    TValue -> E Value -> E Value -> E 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 -> E Value -> Integer -> E Value
ringExp a v i = foldl (ringMul a) (literal 1 a) (genericReplicate i v)

ringMul :: TValue -> E Value -> E Value -> E Value
ringMul = ringBinary (\x y -> pure (x * y))
                     (\x y -> pure (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 -> E Integer
divWrap _ 0 = cryError DivideByZero
divWrap x y = pure (x `quot` y)

modWrap :: Integer -> Integer -> E Integer
modWrap _ 0 = cryError DivideByZero
modWrap x y = pure (x `rem` y)

lg2Wrap :: Integer -> E Integer
lg2Wrap x = if x < 0 then cryError LogNegative else pure (lg2 x)
```

Field

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

```
fieldUnary :: (Rational -> E Rational) ->
              (Integer -> Integer -> E Integer) ->
              (Integer -> Integer -> BigFloat -> E BigFloat) ->
              TValue -> E Value -> E Value
fieldUnary qop zop flop ty v = case ty of
  TVRational  -> VRational <$> appOp1 qop (fromVRational <$> v)
  TVIntMod m  -> VInteger <$> appOp1 (zop m) (fromVInteger <$> 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 -> E Rational) ->
   (Integer -> Integer -> Integer -> E Integer) ->
   (Integer -> Integer -> BigFloat -> BigFloat -> E BigFloat) ->
   TValue -> E Value -> E Value -> E Value
fieldBinary qop zop flop ty l r = case ty of
  TVRational  -> VRational <$>
                   appOp2 qop (fromVRational <$> l) (fromVRational <$> r)
  TVIntMod m  -> VInteger <$>
                   appOp2 (zop m) (fromVInteger <$> l) (fromVInteger <$> r)
  TVFloat e p -> VFloat . fpToBF e p <$>
                      appOp2 (flop e p) (fromVFloat <$> l) (fromVFloat <$> r)
  _ -> evalPanic "fieldBinary" [show ty ++ " is not a Field type"]

ratDiv :: Rational -> Rational -> E Rational
ratDiv _ 0 = cryError DivideByZero
ratDiv x y = pure (x / y)

ratRecip :: Rational -> E  Rational
ratRecip 0 = cryError DivideByZero
ratRecip x = pure (recip x)

zRecip :: Integer -> Integer -> E Integer
zRecip m x = if r == 0 then cryError DivideByZero else pure r
   where r = Integer.recipModInteger x m

zDiv :: Integer -> Integer -> Integer -> E Integer
zDiv m x y = f <$> zRecip m y
  where f yinv = (x * yinv) `mod` m
```

Round

```
roundUnary :: (Rational -> Integer) ->
              (BF -> E Integer) ->
              TValue -> E Value -> E Value
roundUnary op flop ty v = case ty of
  TVRational -> VInteger . op . fromVRational <$> v
```

```
  TVFloat {} -> VInteger <$> (flop . 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.

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

Rational

```
ratioOp :: Integer -> Integer -> E Rational
ratioOp _ 0 = cryError DivideByZero
ratioOp x y = pure (fromInteger x / fromInteger y)
```

Comparison

Comparison primitives may be applied to any type that is constructed of out of base types and tuples, records and finite sequences. All such types are compared using a lexicographic ordering of components. On bits, we have `False` < `True`. Sequences and tuples are compared left-to-right, and record fields are compared in alphabetical order.

Comparisons on base types 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.

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

-- | Lexicographic ordering on two values.
lexCompare :: TValue -> E Value -> E Value -> E 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 <$> r))
    TVRec fields ->
      do let tys = map snd (canonicalFields fields)
         ls <- map snd . sortBy (comparing fst) . fromVRecord <$> l
         rs <- map snd . sortBy (comparing fst) . fromVRecord <$> r
         lexList (zipWith3 lexCompare tys ls rs)
    TVAbstract {} ->
      evalPanic "lexCompare" ["Abstract type not in `Cmp`"]
    TVNewtype {} ->
      evalPanic "lexCompare" ["Newtype not in `Cmp`"]

lexList :: [E Ordering] -> E Ordering
lexList [] = pure EQ
lexList (e : es) =
  e >>= \case
    LT -> pure LT
    EQ -> lexList es
    GT -> pure GT
```

Signed comparisons may be applied to any type made up of non-empty bitvectors using finite sequences, tuples and records. 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 -> E Value -> E Value -> E Value
signedLessThan ty l r = VBit . (== LT) <$> (lexSignedCompare ty l r)

-- | Lexicographic ordering on two signed values.
lexSignedCompare :: TValue -> E Value -> E Value -> E 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 <$> r))
    TVRec fields ->
      do let tys    = map snd (canonicalFields fields)
         ls <- map snd . sortBy (comparing fst) . fromVRecord <$> l
         rs <- map snd . sortBy (comparing fst) . fromVRecord <$> r
         lexList (zipWith3 lexSignedCompare tys ls rs)
    TVAbstract {} ->
      evalPanic "lexSignedCompare" ["Abstract type not in `Cmp`"]
    TVNewtype {} ->
      evalPanic "lexSignedCompare" ["Newtype type not in `Cmp`"]
```

Sequences

```
generateV :: Nat' -> (Integer -> E Value) -> Value
generateV len f = VList len [ f i | i <- idxs ]
  where
   idxs = case len of
            Inf   -> [ 0 .. ]
            Nat n -> [ 0 .. n-1 ]
```

Shifting

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

```
shiftV :: (Nat' -> TValue -> E Value -> Integer -> Value) -> Value
shiftV op =
  VNumPoly $ \n -> pure $
  VPoly $ \ix -> pure $
  VPoly $ \a -> pure $
  VFun $ \v -> pure $
  VFun $ \x ->
  do i <- cryToInteger ix x
     pure $ op n a v i

shiftLV :: Nat' -> TValue -> E Value -> Integer -> Value
shiftLV w a v amt =
  case w of
    Inf   -> generateV Inf $ \i ->
               do vs <- fromVList <$> v
                  indexFront Inf vs (i + amt)
    Nat n -> generateV (Nat n) $ \i ->
               if i + amt < n then
                 do vs <- fromVList <$> v
                    indexFront (Nat n) vs (i + amt)
               else
                 pure (zero a)
```

```
shiftRV :: Nat' -> TValue -> E Value -> Integer -> Value
shiftRV w a v amt =
  generateV w $ \i ->
    if i < amt then
      pure (zero a)
    else
      do vs <- fromVList <$> v
         indexFront w vs (i - amt)

rotateV :: (Integer -> E Value -> Integer -> E Value) -> Value
rotateV op =
  vFinPoly $ \n -> pure $
  VPoly $ \ix -> pure $
  VPoly $ \_a -> pure $
  VFun $ \v -> pure $
  VFun $ \x ->
  do i <- cryToInteger ix x
     op n v i

rotateLV :: Integer -> E Value -> Integer -> E Value
rotateLV 0 v _ = v
rotateLV w v amt =
  pure $ generateV (Nat w) $ \i ->
    do vs <- fromVList <$> v
       indexFront (Nat w) vs ((i + amt) `mod` w)

rotateRV :: Integer -> E Value -> Integer -> E Value
rotateRV 0 v _ = v
rotateRV w v amt =
  pure $ generateV (Nat w) $ \i ->
    do vs <- fromVList <$> v
       indexFront (Nat w) vs ((i - amt) `mod` w)

signedShiftRV :: Value
signedShiftRV =
  VNumPoly $ \(Nat n) -> pure $
  VPoly $ \ix -> pure $
  VFun $ \v -> pure $
  VFun $ \x ->
  do amt <- cryToInteger ix x
     pure $ generateV (Nat n) $ \i ->
       do vs <- fromVList <$> v
          if i < amt then
            indexFront (Nat n) vs 0
          else
            indexFront (Nat n) vs (i - amt)
```

Indexing

Indexing and update 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.

```
-- | Indexing operations that return one element.
indexPrimOne :: (Nat' -> [E Value] -> Integer -> E Value) -> Value
indexPrimOne op =
  VNumPoly $ \n -> pure $
  VPoly $ \_a -> pure $
  VPoly $ \ix -> pure $
  VFun $ \l -> pure $
  VFun $ \r ->
  do vs <- fromVList <$> l
     i <- cryToInteger ix r
     op n vs i

indexFront :: Nat' -> [E Value] -> Integer -> E Value
indexFront w vs ix =
  case w of
    Nat n | 0 <= ix && ix < n -> genericIndex vs ix
    Inf   | 0 <= ix -> genericIndex vs ix
    _ -> cryError (InvalidIndex (Just ix))

indexBack :: Nat' -> [E Value] -> Integer -> E Value
indexBack w vs ix =
  case w of
    Nat n | 0 <= ix && ix < n -> genericIndex vs (n - ix - 1)
          | otherwise -> cryError (InvalidIndex (Just ix))
    Inf               -> evalPanic "indexBack" ["unexpected infinite sequence"]

updatePrim :: (Nat' -> Integer -> Integer) -> Value
updatePrim op =
  VNumPoly $ \len -> pure $
  VPoly $ \_eltTy -> pure $
  VPoly $ \ix -> pure $
  VFun $ \xs -> pure $
  VFun $ \idx -> pure $
  VFun $ \val ->
  do j <- cryToInteger ix idx
     if Nat j < len then
       pure $ generateV len $ \i ->
         if i == op len j then
           val
         else
           do xs' <- fromVList <$> xs
              indexFront len xs' i
     else
       cryError (InvalidIndex (Just j))

updateFront :: Nat' -> Integer -> Integer
updateFront _ j = j
```

```
updateBack :: Nat' -> Integer -> Integer
updateBack Inf _j = evalPanic "Unexpected infinite sequence in updateEnd" []
updateBack (Nat n) j = n - j - 1
```

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.

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

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

```
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 = FP.fpCheckStatus . FP.bfRoundFloat opts . FP.bfFromInteger
  where opts = FP.fpOpts e p fpImplicitRound
```

These functions capture the interactions with rationals.

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 -> E BigFloat
fpBin f r e p x y = pure (FP.fpCheckStatus (f (FP.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 -> E BigFloat
fpRecip e p x = pure (FP.fpCheckStatus (FP.bfDiv opts (FP.bfFromInteger 1) x))
  where opts = FP.fpOpts e p fpImplicitRound

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

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

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

   , "fpToBits"    ~> vFinPoly \e -> pure $
                      vFinPoly \p -> pure $
```

```
                    VFun \fpv ->
                      vWord (e + p) . FP.floatToBits e p . fromVFloat <$> fpv

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

  , "fpIsFinite" ~> vFinPoly \_ -> pure $
                    vFinPoly \_ -> pure $
                    VFun \xv ->
                      do x <- fromVFloat <$> xv
                         pure (VBit (FP.bfIsFinite x))

  , "fpAdd"       ~> fpArith FP.bfAdd
  , "fpSub"       ~> fpArith FP.bfSub
  , "fpMul"       ~> fpArith FP.bfMul
  , "fpDiv"       ~> fpArith FP.bfDiv

  , "fpToRational" ~>
     vFinPoly \_ -> pure $
     vFinPoly \_ -> pure $
     VFun \fpv ->
       do fp <- fromVFloat' <$> fpv
          VRational <$> (eitherToE (FP.floatToRational "fpToRational" fp))
  , "fpFromRational" ~>
    vFinPoly \e -> pure $
    vFinPoly \p -> pure $
    VFun \rmv -> pure $
    VFun \rv ->
      do rm  <- fromVWord =<< rmv
         rm' <- eitherToE (FP.fpRound rm)
         rat <- fromVRational <$> rv
         pure (VFloat (FP.floatFromRational e p rm' rat))
  ]
 where
 fpArith f = vFinPoly \e -> pure $
             vFinPoly \p -> pure $
             VFun \vr -> pure $
             VFun \xv -> pure $
             VFun \yv ->
               do r <- fromVWord =<< vr
                  rnd <- eitherToE (FP.fpRound r)
                  x <- fromVFloat <$> xv
                  y <- fromVFloat <$> yv
                  VFloat . 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

```
ppEValue :: PPOpts -> E Value -> Doc
ppEValue _opts (Err e) = text (show e)
ppEValue opts (Value v) = ppValue opts v

ppValue :: PPOpts -> Value -> Doc
ppValue opts val =
  case val of
    VBit b     -> text (show b)
    VInteger i -> text (show i)
    VRational q -> text (show q)
    VFloat fl -> text (show (FP.fpPP opts fl))
    VList l vs ->
      case l of
        Inf -> ppList (map (ppEValue 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 (ppEValue opts) vs)
      where ppList docs = brackets (fsep (punctuate comma docs))
            isBit v = case v of Value (VBit b) -> Just b
                                _      -> Nothing
    VTuple vs  -> parens (sep (punctuate comma (map (ppEValue opts) vs)))
    VRecord fs -> braces (sep (punctuate comma (map ppField fs)))
      where ppField (f,r) = pp f <+> char '=' <+> ppEValue 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 (E Value)
evaluate expr minp = return (Right (val, modEnv), [])
  where
    modEnv = M.minpModuleEnv minp
    extDgs = concatMap mDecls (M.loadedModules modEnv) ++ M.deDecls (M.meDynEnv modEnv)
    nts    = Map.elems (M.loadedNewtypes modEnv)
```

```
env     = foldl evalDeclGroup (foldl evalNewtypeDecl mempty nts) extDgs
val     = evalExpr env expr
```