Contents


```
-- |
-- Module      :  $Header$
-- 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 #-}

module Cryptol.Eval.Reference
  ( Value(..)
  , evaluate
  , ppValue
  ) where

import Control.Applicative (liftA2)
import Data.Bits
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 Cryptol.ModuleSystem.Name (asPrim)
import Cryptol.TypeCheck.Solver.InfNat (Nat'(..))
import Cryptol.TypeCheck.AST
import Cryptol.Eval.Monad (EvalError(..))
import Cryptol.Eval.Type (TValue(..), isTBit, evalValType, evalNumType, tvSeq)
import Cryptol.Prims.Eval (lg2, divModPoly)
import Cryptol.Utils.Ident (Ident, mkIdent)
import Cryptol.Utils.Panic (panic)
import Cryptol.Utils.PP

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`
`[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) $M(\texttt{t})$. To each Cryptol expression `e : t` we assign a meaning $M(\texttt{e})$ in $M(\texttt{t})$; in particular, recursive Cryptol programs of type `t` are modeled as least fixed points in $M(\texttt{t})$. In other words, this is a domain-theoretic denotational semantics.

Evaluating a Cryptol expression of type `Bit` may result in:

- a defined value `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 $\perp$; we represent it with the Haskell type `Either EvalError Bool`.

The cpos for lists, tuples, and records are cartesian products. The cpo ordering is pointwise, and the bottom element $\perp$ is the list/tuple/record whose elements are all $\perp$. Trivial types `[0]t`, `()` and `{}` denote single-element cpos where the unique value `[]`/`()`/`{}` *is* the bottom element $\perp$. 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
  | VList [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` 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 [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` 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)
        TVSeq w ety  -> VList (map (go ety) (copyList w (fromVList val)))
        TVStream ety -> VList (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) <- fields ]
        TVFun _ bty  -> VFun (\v -> go bty (fromVFun val v))

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

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

-- | 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@.
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.

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

instance Monoid Env where
  mempty = Env
    { envVars  = Map.empty
    , envTypes = Map.empty
    }
  mappend l r = Env
    { envVars  = Map.union (envVars  l) (envVars  r)
    , envTypes = Map.union (envTypes l) (envTypes r)
    }

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

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

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

    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. This function pushes tuple and record selections pointwise down into sequences and functions.

```
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
        VList vs    -> VList (map (tupleSel n) vs)
        VFun f      -> VFun (\x -> tupleSel n (f x))
        _           -> evalPanic "evalSel"
                       ["Unexpected value in tuple selection."]
    recordSel n v =
      case v of
        VRecord _   -> lookupRecord n v
        VList vs    -> VList (map (recordSel n) vs)
        VFun f      -> VFun (\x -> recordSel n (f x))
        _           -> evalPanic "evalSel"
                       ["Unexpected value in record selection."]
    listSel n v =
      case v of
        VList vs    -> vs !! n
        _           -> evalPanic "evalSel"
                       ["Unexpected value in list selection."]
```

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))
    VList vs   -> VList (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 Bool -> Either e Bool -> Either e Bool
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) ]
```

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 ]
```

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 mininum 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 [ 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
```

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.

```
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.

```
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.

```
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:

- Logic: `&&`, `||`, `^`, `complement`, `zero`, `True`, `False`
- Arithmetic: `+`, `-`, `*`, `/`, `%`, `^^`, `lg2`, `negate`, `demote`
- Comparison: `<`, `>`, `<=`, `>=`, `==`, `!=`
- Sequences: `#`, `join`, `split`, `splitAt`, `reverse`, `transpose`
- Shifting: `<<`, `>>`, `<<<`, `>>>`
- Indexing: `@`, `@@`, `!`, `!!`, `update`, `updateEnd`

- Enumerations: `fromThen`, `fromTo`, `fromThenTo`, `infFrom`, `infFromThen`
- Polynomials: `pmult`, `pdiv`, `pmod`
- Miscellaneous: `error`, `random`, `trace`

```
primTable :: Map Ident Value
primTable = Map.fromList $ map (\(n, v) -> (mkIdent (T.pack n), v))

  -- Logic (bitwise):
  [ ("&&"         , binary (logicBinary (&&)))
  , ("||"         , binary (logicBinary (||)))
  , ("^"          , binary (logicBinary (/=)))
  , ("complement" , unary  (logicUnary not))
  , ("zero"       , VPoly (logicNullary (Right False)))
  , ("True"       , VBit (Right True))
  , ("False"      , VBit (Right False))

  -- Arithmetic:
  , ("+"          , binary (arithBinary (\_ x y -> Right (x + y))))
  , ("-"          , binary (arithBinary (\_ x y -> Right (x - y))))
  , ("*"          , binary (arithBinary (\_ x y -> Right (x * y))))
  , ("/"          , binary (arithBinary (const divWrap)))
  , ("%"          , binary (arithBinary (const modWrap)))
  , ("^^"         , binary (arithBinary (\_ x y -> Right (x ^ y))))
  , ("lg2"        , unary  (arithUnary (const lg2)))
  , ("negate"     , unary  (arithUnary (const negate)))
  , ("demote"     , vFinPoly $ \val ->
                    vFinPoly $ \bits ->
                    vWordValue bits val)

  -- 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)))

  -- Sequences:
  , ("#"          , VNumPoly $ \_front ->
                    VNumPoly $ \_back  ->
                    VPoly $ \_elty  ->
                    VFun $ \l ->
                    VFun $ \r ->
                    VList (fromVList l ++ fromVList r))

  , ("join"       , VNumPoly $ \_parts ->
                    VNumPoly $ \_each  ->
                    VPoly $ \_a ->
                    VFun $ \xss ->
                    VList (concat (map fromVList (fromVList xss))))
```

```
  -- FIXME: this doesn't handle the case [inf][0] -> [0]

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

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

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

, ("transpose"  , VNumPoly $ \_a ->
                  VNumPoly $ \b ->
                  VPoly $ \_c ->
                  VFun $ \v ->
                  VList (map VList (transposeV b (map fromVList (fromVList v)))))

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

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

-- Enumerations:
, ("fromThen"   , vFinPoly $ \first ->
                  vFinPoly $ \next  ->
                  vFinPoly $ \bits  ->
                  vFinPoly $ \len   ->
                  VList (map (vWordValue bits) (genericTake len [first, next ..])))

, ("fromTo"     , vFinPoly $ \first ->
                  vFinPoly $ \lst   ->
                  vFinPoly $ \bits  ->
                  VList (map (vWordValue bits) [first .. lst]))
```

```
, ("fromThenTo" , vFinPoly $ \first ->
                  vFinPoly $ \next  ->
                  vFinPoly $ \_lst  ->
                  vFinPoly $ \bits  ->
                  vFinPoly $ \len   ->
                  VList (map (vWordValue bits) (genericTake len [first, next ..])))

, ("infFrom"    , vFinPoly $ \bits ->
                  VFun $ \first ->
                  case fromVWord first of
                    Left e -> VList (repeat (vWordError bits e))
                    Right i -> VList (map (vWordValue bits) [i ..]))

, ("infFromThen", vFinPoly $ \bits ->
                  VFun $ \first ->
                  VFun $ \next ->
                  case fromVWord first of
                    Left e -> VList (repeat (vWordError bits e))
                    Right i ->
                      case fromVWord next of
                        Left e -> VList (repeat (vWordError bits e))
                        Right j -> VList (map (vWordValue bits) [i, j ..]))

-- Polynomials:
, ("pmult"      , let mul res _  _  0 = res
                      mul res bs as n = mul (if even as then res else xor res bs)
                                            (bs `shiftL` 1) (as `shiftR` 1) (n-1)
                  in vFinPoly $ \a ->
                     vFinPoly $ \b ->
                     VFun $ \x ->
                     VFun $ \y ->
                     case fromVWord x of
                       Left e -> vWordError (1 + a + b) e
                       Right i ->
                         case fromVWord y of
                           Left e -> vWordError (1 + a + b) e
                           Right j -> vWordValue (1 + a + b) (mul 0 i j (1+b)))
, ("pdiv"       , vFinPoly $ \a ->
                  vFinPoly $ \b ->
                  VFun $ \x ->
                  VFun $ \y ->
                  case fromVWord x of
                    Left e -> vWordError a e
                    Right i ->
                      case fromVWord y of
                        Left e -> vWordError a e
                        Right j -> vWordValue a
                                   (fst (divModPoly i (fromInteger a) j (fromInteger b))))

, ("pmod"       , vFinPoly $ \a ->
```

```
                    vFinPoly $ \b ->
                    VFun $ \x ->
                    VFun $ \y ->
                    case fromVWord x of
                      Left e -> vWordError b e
                      Right i ->
                        case fromVWord y of
                          Left e -> vWordError b e
                          Right j -> vWordValue b
                                       (snd (divModPoly i (fromInteger a) j (fromInteger b + 1))))

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

  , ("random"     , VPoly $ \a ->
                    VFun $ \_seed ->
                    logicNullary (Left (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
```

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 -> 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
```

Functions `vWord`, `vWordValue`, and `vWordError` convert from integers 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.

```
-- | Convert an integer to big-endian binary value of the specified width.
vWordValue :: Integer -> Integer -> Value
vWordValue w x = VList (map (VBit . Right) (integerToBits w x))

-- | Convert an integer to a big-endian format of the specified width.
integerToBits :: Integer -> Integer -> [Bool]
integerToBits w x = go [] w x
  where go bs 0 _ = bs
        go bs n a = go (odd a : bs) (n - 1) $! (a `div` 2)

-- | Create a run-time error value of bitvector type.
vWordError :: Integer -> EvalError -> Value
vWordError w e = VList (genericReplicate w (VBit (Left e)))

vWord :: Integer -> Either EvalError Integer -> Value
vWord w (Left e) = vWordError w e
vWord w (Right x) = vWordValue w x
```

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.

```
logicNullary :: Either EvalError Bool -> TValue -> Value
logicNullary b = go
  where
    go TVBit          = VBit b
    go (TVSeq n ety)  = VList (genericReplicate n (go ety))
    go (TVStream ety) = VList (repeat (go ety))
    go (TVTuple tys)  = VTuple (map go tys)
    go (TVRec fields) = VRecord [ (f, go fty) | (f, fty) <- fields ]
    go (TVFun _ bty)  = VFun (\_ -> go bty)

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 (map (go ety) (fromVList val))
        TVStream ety -> VList (map (go ety) (fromVList val))
        TVTuple etys -> VTuple (zipWith go etys (fromVTuple val))
        TVRec fields -> VRecord [ (f, go fty (lookupRecord f val)) | (f, fty) <- fields ]
        TVFun _ bty  -> VFun (\v -> go bty (fromVFun val v))
```

```
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 (zipWith (go ety) (fromVList l) (fromVList r))
        TVStream ety -> VList (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) <- fields ]
        TVFun _ bty  -> VFun (\v -> go bty (fromVFun l v) (fromVFun r v))
```

Arithmetic

Arithmetic primitives may be applied to any type that is made up of finite bitvectors. 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"]`.

```
arithUnary :: (Integer -> Integer -> Integer)
           -> TValue -> Value -> Value
arithUnary op = go
  where
    go :: TValue -> Value -> Value
    go ty val =
      case ty of
        TVBit ->
          evalPanic "arithUnary" ["Bit not in class Arith"]
        TVSeq w a
          | isTBit a  -> vWord w (op w <$> fromVWord val)
          | otherwise -> VList (map (go a) (fromVList val))
        TVStream a ->
          VList (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) <- fs ]

arithBinary :: (Integer -> Integer -> Integer -> Either EvalError Integer)
            -> TValue -> Value -> Value -> Value
arithBinary op = go
  where
    go :: TValue -> Value -> Value -> Value
    go ty l r =
      case ty of
        TVBit ->
          evalPanic "arithBinary" ["Bit not in class Arith"]
```

```
        TVSeq w a
          | isTBit a  -> case fromVWord l of
                           Left e -> vWordError w e
                           Right i ->
                             case fromVWord r of
                               Left e -> vWordError w e
                               Right j -> vWord w (op w i j)
          | otherwise -> VList (zipWith (go a) (fromVList l) (fromVList r))
        TVStream a ->
          VList (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) <- fs ]

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

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

```
-- | 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
    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 (fromVList l) (fromVList r))
    TVRec fields ->
      let tys    = map snd (sortBy (comparing fst) fields)
          ls     = map snd (sortBy (comparing fst) (fromVRecord l))
          rs     = map snd (sortBy (comparing fst) (fromVRecord r))
       in lexList (zipWith3 lexCompare tys ls rs)

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
```

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 ys : splitV (Nat (n - 1)) k zs
    Inf   -> VList 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"]
    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.

```
shiftV :: (Nat' -> Value -> [Value] -> Integer -> [Value]) -> Value
shiftV op =
  VNumPoly $ \a ->
  VNumPoly $ \_b ->
  VPoly $ \c ->
  VFun $ \v ->
  VFun $ \x ->
  case fromVWord x of
    Left e -> logicNullary (Left e) (tvSeq a c)
    Right i -> VList (op a (logicNullary (Right False) c) (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 $ \a ->
  VNumPoly $ \_b ->
  VPoly $ \c ->
  VFun $ \v ->
  VFun $ \x ->
  case fromVWord x of
    Left e -> VList (genericReplicate a (logicNullary (Left e) c))
    Right i -> VList (op a (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
  where (xs, ys) = genericSplitAt ((w - i) `mod` w) vs
```

Indexing

Indexing operations are strict in all index bits, but as lazy as possible in the list values.

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

-- | Indexing operations that return many elements.
indexPrimMany :: (Nat' -> TValue -> [Value] -> Integer -> Value) -> Value
indexPrimMany op =
  VNumPoly $ \n ->
  VPoly    $ \a ->
  VNumPoly $ \_m ->
  VNumPoly $ \_w ->
  VFun $ \l ->
  VFun $ \r -> VList [ case fromVWord y of
                         Left e -> logicNullary (Left e) a
                         Right i -> op n a xs i
                     | let xs = fromVList l, y <- fromVList r ]

indexFront :: Nat' -> TValue -> [Value] -> Integer -> Value
indexFront w a vs ix =
  case w of
    Nat n | n <= ix -> logicNullary (Left (InvalidIndex 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 -> logicNullary (Left (InvalidIndex ix)) a
    Inf               -> evalPanic "indexBack" ["unexpected infinite sequence"]

updatePrim :: (Nat' -> [Value] -> Integer -> Value -> [Value]) -> Value
updatePrim op =
  VNumPoly $ \len ->
  VPoly $ \eltTy ->
  VNumPoly $ \_idxLen ->
  VFun $ \xs ->
  VFun $ \idx ->
  VFun $ \val ->
  case fromVWord idx of
```

```
    Left e -> logicNullary (Left e) (tvSeq len eltTy)
    Right i -> VList (op len (fromVList xs) i val)

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
```

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 :: Value -> Doc
ppValue val =
  case val of
    VBit b     -> text (either show show b)
    VList vs   -> brackets (fsep (punctuate comma (map ppValue vs)))
    VTuple vs  -> parens (sep (punctuate comma (map ppValue vs)))
    VRecord fs -> braces (sep (punctuate comma (map ppField fs)))
      where ppField (f,r) = pp f <+> char '=' <+> ppValue 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), [])
  where
    extDgs = concatMap mDecls (M.loadedModules modEnv)
    env = foldl evalDeclGroup mempty extDgs
```