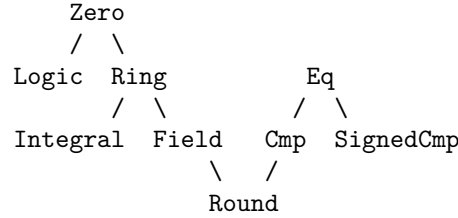


## Contents

Typeclass Hierarchy . . . . .	1
Literals . . . . .	1
Fractional Literals . . . . .	2
Zero . . . . .	2
Boolean . . . . .	2
Arithmetic . . . . .	3
Equality Comparisons . . . . .	4
Comparisons and Ordering . . . . .	5
Signed Comparisons . . . . .	5
Bitvectors . . . . .	5
Rationals . . . . .	6
Z(n) . . . . .	6
Sequences . . . . .	6
Shift And Rotate . . . . .	7
GF(2) polynomials . . . . .	7
Random Values . . . . .	7
Errors and Assertions . . . . .	7
Debugging . . . . .	7
Utility operations . . . . .	7

## Typeclass Hierarchy



This diagram describes how the various typeclasses in Cryptol are related. A type which is an instance of a subclass is also always a member of all of its superclasses. For example, any type which is a member of `Field` is also a member of `Ring` and `Zero`.

## Literals

```

type Literal : # -> * -> Prop

number : {val, rep} Literal val rep => rep
length : {n, a, b} (fin n, Literal n b) => [n]a -> b

// '[a..b]' is syntactic sugar for 'fromTo`{first=a,last=b}'
fromTo : {first, last, a}
         (fin last, last >= first,

```

```

Literal first a, Literal last a) =>
[1 + (last - first)]a

// '[a,b..c]' is syntactic sugar for 'fromThenTo`{first=a,next=b,last=c}'
fromThenTo : {first, next, last, a, len}
  ( fin first, fin next, fin last
  , Literal first a, Literal next a, Literal last a
  , first != next
  , lengthFromThenTo first next last == len) =>
[len]a

```

## Fractional Literals

The predicate `FLiteral m n r a` asserts that the type `a` contains the fraction `m/n`. The flag `r` indicates if we should round (`r >= 1`) or report an error if the number can't be represented exactly.

```
type FLiteral : # -> # -> # -> * -> Prop
```

Fractional literals are desugared into calls to the primitive `fraction`:

```
fraction : { m, n, r, a } FLiteral m n r a => a
```

## Zero

```
type Zero : * -> Prop
```

```
zero : {a} (Zero a) => a
```

Every base and structured type in Cryptol is a member of class `Zero`.

## Boolean

```
type Logic : * -> Prop
```

```
False : Bit
```

```
True : Bit
```

```
(&&) : {a} (Logic a) => a -> a -> a
```

```
(||) : {a} (Logic a) => a -> a -> a
```

```
(^) : {a} (Logic a) => a -> a -> a
```

```
complement : {a} (Logic a) => a -> a
```

```
// The prefix notation '~ x' is syntactic
```

```
// sugar for 'complement x'.
```

```
(==>) : Bit -> Bit -> Bit
```

```
(/\) : Bit -> Bit -> Bit
```

```
(\/) : Bit -> Bit -> Bit
```

```

instance                               Logic Bit
instance (Logic a)                    => Logic ([n]a)
instance (Logic b)                    => Logic (a -> b)
instance (Logic a, Logic b) => Logic (a, b)
instance (Logic a, Logic b) => Logic { x : a, y : b }
// No instance for `Logic Integer`.
// No instance for `Logic (Z n)`.
// No instance for `Logic Rational`.

```

## Arithmetic

```

type Ring    : * -> Prop

```

```

fromInteger : {a} (Ring a) => Integer -> a
(+) : {a} (Ring a) => a -> a -> a
(-) : {a} (Ring a) => a -> a -> a
(*) : {a} (Ring a) => a -> a -> a
negate : {a} (Ring a) => a -> a
  // The prefix notation `- x` is syntactic
  // sugar for `negate x`.

```

```

type Integral : * -> Prop

```

```

(/) : {a} (Integral a) => a -> a -> a
(%) : {a} (Integral a) => a -> a -> a
toInteger : {a} (Integral a) => a -> Integer
infFrom : {a} (Integral a) => a -> [inf]a
  // '[x...]' is syntactic sugar for 'infFrom x'
infFromThen : {a} (Integral a) => a -> a -> [inf]a
  // '[x,y...]' is syntactic sugar for 'infFromThen x y'

```

```

type Field : * -> Prop

```

```

(/.) : {a} (Field a) => a -> a -> a
recip : {a} (Field a) => a -> a

```

```

type Round : * -> Prop

```

```

floor      : {a} (Round a) => a -> Integer
ceiling    : {a} (Round a) => a -> Integer
trunc      : {a} (Round a) => a -> Integer
roundAway  : {a} (Round a) => a -> Integer

```

```

roundToEven : {a} (Round a) => a -> Integer

(^^) : {a, e} (Ring a, Integral e) => a -> e -> a

```

```

// No instance for `Bit`.
instance (fin n)          => Ring ([n]Bit)
instance (Ring a)         => Ring ([n]a)
instance (Ring b)         => Ring (a -> b)
instance (Ring a, Ring b) => Ring (a, b)
instance (Ring a, Ring b) => Ring { x : a, y : b }
instance                  Ring Integer
instance (fin n, n>=1)    => Ring (Z n)
instance                  Ring Rational
instance (ValidFloat e p) => Ring (Float e p)

```

Note that because there is no instance for `Ring Bit` the top two instances do not actually overlap.

```

instance                  Integral Integer
instance (fin n)          => Integral ([n]Bit)

instance Field Rational
instance (prime p) => Field (Z p)
instance (ValidFloat e p) => Field (Float e p)

instance Round Rational
instance (ValidFloat e p) => Round (Float e p)

```

## Equality Comparisons

```

type Eq : * -> Prop

(==) : {a} (Eq a) => a -> a -> Bit
(!=) : {a} (Eq a) => a -> a -> Bit
(===) : {a,b} (Eq b) => (a -> b) -> (a -> b) -> a -> Bit
(!==) : {a,b} (Eq b) => (a -> b) -> (a -> b) -> a -> Bit

instance                  Eq Bit
instance (Eq a, fin n) => Eq [n]a
instance (Eq a, Eq b)  => Eq (a, b)
instance (Eq a, Eq b)  => Eq { x : a, y : b }
instance                  Eq Integer
instance                  Eq Rational
instance (fin n, n>=1) => Eq (Z n)
// No instance for functions.

```

## Comparisons and Ordering

```
type Cmp : * -> Prop

(<)  : {a} (Cmp a) => a -> a -> Bit
(>)  : {a} (Cmp a) => a -> a -> Bit
(<=) : {a} (Cmp a) => a -> a -> Bit
(>=) : {a} (Cmp a) => a -> a -> Bit

min  : {a} (Cmp a) => a -> a -> a
max  : {a} (Cmp a) => a -> a -> a

abs  : {a} (Cmp a, Ring a) => a -> a

instance                               Cmp Bit
instance (Cmp a, fin n) => Cmp [n]a
instance (Cmp a, Cmp b) => Cmp (a, b)
instance (Cmp a, Cmp b) => Cmp { x : a, y : b }
instance                               Cmp Integer
instance                               Cmp Rational
// No instance for functions.
```

## Signed Comparisons

```
type SignedCmp : * -> Prop

(<$)  : {a} (SignedCmp a) => a -> a -> Bit
(>$)  : {a} (SignedCmp a) => a -> a -> Bit
(<=$) : {a} (SignedCmp a) => a -> a -> Bit
(>=$) : {a} (SignedCmp a) => a -> a -> Bit

// No instance for Bit
instance (fin n, n >= 1)          => SignedCmp [n]
instance (SignedCmp a, fin n)    => SignedCmp [n]a
      // (for [n]a, where a is other than Bit)
instance (SignedCmp a, SignedCmp b) => SignedCmp (a, b)
instance (SignedCmp a, SignedCmp b) => SignedCmp { x : a, y : b }
// No instance for functions.
```

## Bitvectors

```
(/$)  : {n} (fin n, n >= 1) => [n] -> [n] -> [n]
(%)$) : {n} (fin n, n >= 1) => [n] -> [n] -> [n]

carry  : {n} (fin n) => [n] -> [n] -> Bit
scarry : {n} (fin n, n >= 1) => [n] -> [n] -> Bit
```

```

sborrow : {n} (fin n, n >= 1) => [n] -> [n] -> Bit

zext      : {m, n} (fin m, m >= n) => [n] -> [m]
sext      : {m, n} (fin m, m >= n, n >= 1) => [n] -> [m]

lg2       : {n} (fin n) => [n] -> [n]

// Arithmetic shift only for bitvectors
(>>$)    : {n, ix} (fin n, n >= 1, Integral ix) => [n] -> ix -> [n]

```

## Rationals

```
ratio : Integer -> Integer -> Rational
```

## Z(n)

```
fromZ : {n} (fin n, n >= 1) => Z n -> Integer
```

## Sequences

```

join      : {parts,each,a} (fin each) => [parts][each]a -> [parts * each]a
split     : {parts,each,a} (fin each) => [parts * each]a -> [parts][each]a

(#)       : {front,back,a} (fin front) => [front]a -> [back]a -> [front + back]a
splitAt   : {front,back,a} (fin front) => [front]a -> ([back] a, [front] a)

reverse   : {n,a} (fin n) => [n]a -> [n]a
transpose : {n,m,a} [n][m]a -> [m][n]a

(@)       : {n,a,ix} (Integral ix) => [n]a -> ix -> a
(@@)      : {n,k,ix,a} (Integral ix) => [n]a -> [k]ix -> [k]a
(!)       : {n,a,ix} (fin n, Integral ix) => [n]a -> ix -> a
(!!)      : {n,k,ix,a} (fin n, Integral ix) => [n]a -> [k]ix -> [k]a
update    : {n,a,ix} (Integral ix) => [n]a -> ix -> a -> [n]a
updateEnd : {n,a,ix} (fin n, Integral ix) => [n]a -> ix -> a -> [n]a
updates   : {n,k,ix,a} (Integral ix, fin k) => [n]a -> [k]ix -> [k]a -> [n]a
updatesEnd : {n,k,ix,d} (fin n, Integral ix, fin k) => [n]a -> [k]ix -> [k]a -> [n]a

take      : {front,back,elem} (fin front) => [front + back]elem -> [front]elem
drop      : {front,back,elem} (fin front) => [front + back]elem -> [back]elem
head      : {a, b} [1 + a]b -> b
tail      : {a, b} [1 + a]b -> [a]b
last      : {a, b} [1 + a]b -> b

```

```

// Declarations of the form 'x @ i = e' are syntactic
// sugar for 'x = generate (\i -> e)'.

```

`generate` : {n, a} (fin n, n >= 1) => (Integer -> a) -> [n]a

`groupBy` : {each,parts,elem} (fin each) => [parts \* each]elem -> [parts][each]elem

Function `groupBy` is the same as `split` but with its type arguments in a different order.

## Shift And Rotate

`(<<)` : {n,ix,a} (Integral ix, Zero a) => [n]a -> ix -> [n]a

`(>>)` : {n,ix,a} (Integral ix, Zero a) => [n]a -> ix -> [n]a

`(<<<)` : {n,ix,a} (fin n, Integral ix) => [n]a -> ix -> [n]a

`(>>>)` : {n,ix,a} (fin n, Integral ix) => [n]a -> ix -> [n]a

## GF(2) polynomials

`pmult` : {u, v} (fin u, fin v) => [1 + u] -> [1 + v] -> [1 + u + v]

`pdiv` : {u, v} (fin u, fin v) => [u] -> [v] -> [u]

`pmod` : {u, v} (fin u, fin v) => [u] -> [1 + v] -> [v]

## Random Values

`random` : {a} => [256] -> a

## Errors and Assertions

`undefined` : {a} a

`error` : {a,n} (fin n) => String n -> a

`assert` : {a,n} (fin n) => Bit -> String n -> a -> a

## Debugging

`trace` : {n, a, b} (fin n) => String n -> a -> b -> b

`traceVal` : {n, a} (fin n) => String n -> a -> a

## Utility operations

`and` : {n} (fin n) => [n]Bit -> Bit

`or` : {n} (fin n) => [n]Bit -> Bit

`all` : {n, a} (fin n) => (a -> Bit) -> [n]a -> Bit

`any` : {n, a} (fin n) => (a -> Bit) -> [n]a -> Bit

`elem` : {n, a} (fin n, Eq a) => a -> [n]a -> Bit

`deepseq` : {a, b} Eq a => a -> b -> b

`rnf` : {a} Eq a => a -> a

`map` : {n, a, b} (a -> b) -> [n]a -> [n]b

```

foldl  : {n, a, b} (fin n) => (a -> b -> a) -> a -> [n]b -> a
foldl' : {n, a, b} (fin n, Eq a) => (a -> b -> a) -> a -> [n]b -> a
foldr  : {n, a, b} (fin n) => (a -> b -> b) -> b -> [n]a -> b
foldl' : {n, a, b} (fin n, Eq a) => (a -> b -> a) -> a -> [n]b -> a
scanl  : {n, b, a} (b -> a -> b) -> b -> [n]a -> [n+1]b
scanr  : {n, a, b} (fin n) => (a -> b -> b) -> b -> [n]a -> [n+1]b
sum     : {n, a} (fin n, Eq a, Ring a) => [n]a -> a
product : {n, a} (fin n, Eq a, Ring a) => [n]a -> a

iterate : {a} (a -> a) -> a -> [inf]a
repeat  : {n, a} a -> [n]a
zip     : {n, a, b} [n]a -> [n]b -> [n](a, b)
zipWith : {n, a, b, c} (a -> b -> c) -> [n]a -> [n]b -> [n]c
uncurry : {a, b, c} (a -> b -> c) -> (a, b) -> c
curry   : {a, b, c} ((a, b) -> c) -> a -> b -> c

```