Hugs and GHC provide a common set of libraries to aid portability. This document specifies the interfaces to these libraries and documents known differences.

Contents

1 Naming conventions 2
2 Addr 3
3 Bits 4
4 Concurrent 5
5 Dynamic 6
  5.1 Representing types ........................................ 7
  5.2 The Typeable class ........................................ 8
  5.3 Utility functions ........................................ 9
6 Exception 9
  6.1 The try functions ........................................ 10
  6.2 The catch functions ...................................... 11
  6.3 Dynamic Exceptions ...................................... 11
  6.4 Other Utilities ........................................ 11
7 Foreign 12
8 GetOpt 13
9 IOExts 15
10 Int 16
11 NumExts 17
12 Pretty 18
13 ST 19
1. Naming conventions

The set of interfaces specified in this document try to adhere to the following naming conventions:

- Actions that create a new values have the prefix `new` followed by the name of the type of object they're creating, e.g., `newIORef`, `newChan` etc.

- Operations that read a value from a mutable object are prefixed with `read`, and operations that update the contents have the prefix `write`, e.g., `readChan`, `readIOArray`.

Notes:

- This differs from the convention used to name the operations for reading and writing to a file `Handle`, where `get` and `put` are used instead.

- Operations provided by various concurrency abstractions, e.g., `MVar`, `CVar`, also deviate from this naming scheme. This is perhaps defensible, since the read and write operations have additional behaviour, e.g., `takeMVar` tries to read the current value of an `MVar`, locking it if it succeeds.

- Conversions operators have the form `AToB` where `A` and `B` are the types we’re converting between.

- Operations that lazily read values from a mutable object/handle, have the form `getXContents`, e.g., `Channel.getChanContents` and `IO.hGetContents`. (OK, so the latter isn’t called `getHandleContents`, but you hopefully get the picture.)
2 Addr

This library provides machine addresses and is primarily intended for use in creating foreign function interfaces using GreenCard.

```haskell
module Addr where
data Addr -- Address type
instance Eq Addr
instance Ord Addr

nullAddr :: Addr
plusAddr :: Addr -> Int -> Addr

-- read value out of _immutable_ memory
indexCharOffAddr :: Addr -> Int -> Char
indexIntOffAddr :: Addr -> Int -> Int
indexAddrOffAddr :: Addr -> Int -> Addr
indexFloatOffAddr :: Addr -> Int -> Float
indexDoubleOffAddr :: Addr -> Int -> Double
indexWord8OffAddr :: Addr -> Int -> Word8
indexWord16OffAddr :: Addr -> Int -> Word16
indexWord32OffAddr :: Addr -> Int -> Word32
indexWord64OffAddr :: Addr -> Int -> Word64
indexInt8OffAddr :: Addr -> Int -> Int8
indexInt16OffAddr :: Addr -> Int -> Int16
indexInt32OffAddr :: Addr -> Int -> Int32
indexInt64OffAddr :: Addr -> Int -> Int64

-- read value out of mutable memory
readCharOffAddr :: Addr -> Int -> IO Char
readIntOffAddr :: Addr -> Int -> IO Int
readAddrOffAddr :: Addr -> Int -> IO Addr
readFloatOffAddr :: Addr -> Int -> IO Float
readDoubleOffAddr :: Addr -> Int -> IO Double
readWord8OffAddr :: Addr -> Int -> IO Word8
readWord16OffAddr :: Addr -> Int -> IO Word16
readWord32OffAddr :: Addr -> Int -> IO Word32
readWord64OffAddr :: Addr -> Int -> IO Word64
readInt8OffAddr :: Addr -> Int -> IO Int8
readInt16OffAddr :: Addr -> Int -> IO Int16
readInt32OffAddr :: Addr -> Int -> IO Int32
readInt64OffAddr :: Addr -> Int -> IO Int64

-- write value into mutable memory
writeCharOffAddr :: Addr -> Int -> Char -> IO ()
writeIntOffAddr :: Addr -> Int -> Int -> IO ()
writeAddrOffAddr :: Addr -> Int -> Addr -> IO ()
writeForeignObjOffAddr :: Addr -> Int -> ForeignObj -> IO ()
writeFloatOffAddr :: Addr -> Int -> Float -> IO ()
writeDoubleOffAddr :: Addr -> Int -> Double -> IO ()
writeWord8OffAddr :: Addr -> Int -> Word8 -> IO ()
writeWord16OffAddr :: Addr -> Int -> Word16 -> IO ()
writeWord32OffAddr :: Addr -> Int -> Word32 -> IO ()
writeWord64OffAddr :: Addr -> Int -> Word64 -> IO ()
```
3. Bits

This library defines bitwise operations for signed and unsigned ints.

```haskell
module Bits where

infixl 8 'shift', 'rotate'
infixl 7 .&.
infixl 6 'xor'
infixl 5 .|.

class Bits a where
    (.&.), (.|.), xor :: a -> a -> a
    complement :: a -> a
    shift :: a -> Int -> a
    rotate :: a -> Int -> a
    bit :: Int -> a
    setBit :: a -> Int -> a
    clearBit :: a -> Int -> a
    complementBit :: a -> Int -> a
    testBit :: a -> Int -> Bool
    bitSize :: a -> Int
    isSigned :: a -> Bool

shiftL, shiftR :: Bits a => a -> Int -> a
rotateL, rotateR :: Bits a => a -> Int -> a
shiftL a i = shift a i
shiftR a i = shift a (-i)
rotateL a i = rotate a i
rotateR a i = rotate a (-i)
```

Notes:

- `bitSize` and `isSigned` are like `floatRadix` and `floatDigits` – they return parameters of the type of their argument rather than of the particular argument they are applied to. `bitSize` returns the number of bits in the type; and `isSigned` returns whether the type is signed or not.

- `shift` performs sign extension on signed number types. That is, right shifts fill the top bits with 1 if the number is negative and with 0 otherwise.
• Bits are numbered from 0 with bit 0 being the least significant bit.

• \texttt{shift x i} and \texttt{rotate x i} shift to the left if \(i\) is positive and to the right otherwise.

• \texttt{bit i} is the value with the \(i\)th bit set.

4 Concurrent

This library provides the Concurrent Haskell extensions as described in \textit{Concurrent Haskell} \texttt{<http://research.microsoft.com/Users/simonpj/Papers/concurrent-haskell.ps.gz>.

module Concurrent where

data ThreadId -- thread identifiers
instance Eq ThreadId
instance Ord ThreadId

forkIO :: IO () -> IO ThreadId
killThread :: ThreadId -> IO ()

data MVar a -- Synchronisation variables
newEmptyMVar :: IO (MVar a)
newMVar :: a -> IO (MVar a)
takeMVar :: MVar a -> IO a
putMVar :: MVar a -> a -> IO ()
swapMVar :: MVar a -> a -> IO a
readMVar :: MVar a -> IO a
isEmptyMVar :: MVar a -> IO Bool
instance Eq (MVar a)

data Chan a -- channels
newChan :: IO (Chan a)
writeChan :: Chan a -> a -> IO ()
readChan :: Chan a -> IO a
dupChan :: Chan a -> IO (Chan a)
unreadChan :: Chan a -> a -> IO ()
getChanContents :: Chan a -> IO [a]
writeList2Chan :: Chan a -> [a] -> IO ()

data CVar a -- one element channels
newCVar :: IO (CVar a)
putCVar :: CVar a -> a -> IO ()
getCVar :: CVar a -> IO a

data QSem -- General/quantity semaphores
newQSem :: Int -> IO QSem
waitQSem :: QSem -> IO ()
signalQSem :: QSem -> IO ()
5. Dynamic

The **Dynamic** library provides cheap-and-cheerful dynamic types for Haskell. A dynamically typed value is one which carries type information with it at run-time, and is represented here by the abstract type **Dynamic**. Values can be converted into **Dynamic** ones, which can then be combined and manipulated by the program.
using the operations provided over the abstract, dynamic type. One of these operations allows you to (try to) convert a dynamically-typed value back into a value with the same (monomorphic) type it had before converting it into a dynamically-typed value. If the dynamically-typed value isn’t of the desired type, the coercion will fail.

The Dynamic library is capable of dealing with monomorphic types only; no support for polymorphic dynamic values, but hopefully that will be added at a later stage.

Examples where this library may come in handy (dynamic types, really - hopefully the library provided here will suffice) are: persistent programming, interpreters, distributed programming etc.

The following operations are provided over the Dynamic type:

```hs
data Dynamic -- abstract, instance of: Show --

toDyn        :: Typeable a => a -> Dynamic
fromDyn      :: Typeable a => Dynamic -> a -> a
fromDynamic :: Typeable a => Dynamic -> Maybe a
```

- `toDyn` converts a value into a dynamic one, provided `toDyn` knows the (concrete) type representation of the value. The `Typeable` type class is used to encode this, overloading a function that returns the type representation of a value. More on this below.

- There’s two ways of going from a dynamic value to one with a concrete type: `fromDyn`, tries to convert the dynamic value into a value with the same type as its second argument. If this fails, the default second argument is just returned. `fromDynamic` returns a `Maybe` type instead, `Nothing` coming back if the conversion was not possible.

- The Dynamic type has got a `Show` instance which returns a pretty printed string of the type of the dynamic value. (Useful when debugging).

### 5.1 Representing types

Haskell types are represented as terms using the `TypeRep` abstract type:

```hs
data TypeRep -- abstract, instance of: Eq, Show
data TyCon    -- abstract, instance of: Eq, Show

mkTyCon       :: String    -> TyCon
mkAppTy       :: TyCon    -> [TypeRep] -> TypeRep
mkFunTy       :: TypeRep  -> TypeRep    -> TypeRep
applyTy       :: TypeRep  -> TypeRep    -> Maybe TypeRep
```

- `mkAppTy` applies a type constructor to a sequence of types, returning a type.
- `mkFunTy` is a special case of `mkAppTy`, applying the function type constructor to a pair of types.
- `applyTy` applies a type to a function type. If possible, the result type is returned.
- Type constructors are represented by the abstract type, `TyCon`. 
• Most importantly, TypeReps can be compared for equality. Type equality is used when converting a Dynamic value into a value of some specific type, comparing the type representation that the Dynamic value embeds with equality of the type representation of the type we're trying to convert the dynamically-typed value into.

• To allow comparisons between TypeReps to be implemented efficiently, the abstract TyCon type is used, with the constructor function mkTyCon provided:

```haskell
mkTyCon :: String -> TyCon
```

An implementation of the Dynamic interface guarantees the following,

```haskell
mkTyCon "a" == mkTyCon "a"
```

A really efficient implementation is possible if we guarantee/demand that the strings are unique, and for a particular type constructor, the application mkTyCon to the string that represents the type constructor is never duplicated. **Q: Would this constraint be unworkable in practice?**

• Both TyCon and TypeRep are instances of the Show type classes. To have tuple types be shown in infix form, the Show instance guarantees that type constructors consisting of n-commas, i.e., (mkTyCon ",,,,"), is shown as an (n+1) tuple in infix form.

### 5.2 The Typeable class

To ease the construction of Dynamic values, we introduce the following type class to help working with TypeReps:

```haskell
class Typeable a where
    typeOf :: a -> TypeRep
```

• The typeOf function is overloaded to return the type representation associated with a type.

• **Important:** The argument to typeOf is only used to carry type information around so that overloading can be resolved. Typeable instances should never, ever look at this argument.

• The Dynamic library provide Typeable instances for all Prelude and Hugs/GHC extension library types. They are:

  Prelude types:
  - Int, Char, Bool, Float, Double, Integer, (IO a),
  - [a], (Either a b), (Maybe a), (a->b),
  - (), (,), (,,), (,,,,),
  - Ordering, Complex, Array, Handle

  Hugs/GHC types:
  - Addr, Word8, Word16, Word32, Word64,
  - Int8,Int16,Int32,Int64,
  - ForeignObj, MVar, (ST s a), (StablePtr a)

  GHC types:
  - Word, ByteArray, MutableByteArray
5.3 Utility functions

Operations for applying a dynamic function type to a dynamically typed argument are commonly useful, and also provided:

\[
\text{dynApply} :: \text{Dynamic} \rightarrow \text{Dynamic} \rightarrow \text{Dynamic} \quad --\quad \text{unsafe}.
\]
\[
\text{dynApplyMb} :: \text{Dynamic} \rightarrow \text{Dynamic} \rightarrow \text{Maybe Dynamic}
\]

6 Exception

The Exception library provides an interface for raising and catching both built-in and user defined exceptions. Exceptions are defined by the following (non-abstract) datatype:

```haskell
data Exception = IOException IOError -- IO exceptions (from 'fail')
| ArithException -- Arithmetic exceptions
| ErrorCall String -- Calls to 'error'
| NoMethodError String -- A non-existent method was invoked
| PatternMatchFail String -- A pattern match failed
| NonExhaustiveGuards String -- A guard match failed
| RecSelError String -- Selecting a non-existent field
| RecConError String -- Field missing in record construction
| RecUpdError String -- Record doesn’t contain updated field
| AssertionFailed String -- Assertions
| DynException Dynamic -- Dynamic exceptions
| AsyncException -- Externally generated errors
```

instance Eq Exception
instance Ord Exception
instance Show Exception

```haskell
data ArithException = Overflow
| Underflow
| LossOfPrecision
| DivideByZero
| Denormal
```

instance Eq ArithError
instance Ord ArithError
instance Show ArithError

```haskell
data AsyncException = StackOverflow
| HeapOverflow
| ThreadKilled
deriving (Eq, Ord)
```

instance Eq AsyncException
6. Exception

instance Ord AsyncException
instance Show AsyncException

An implementation should raise the appropriate exception when one of the above conditions arises. Note: GHC currently doesn’t generate the arithmetic or the async exceptions.

Exceptions may be thrown explicitly from anywhere:

\[
\text{throw} :: \text{Exception} \to a
\]

6.1 The try functions

There are several functions for catching and examining exceptions; all of them may only be used from within the \text{IO} monad. Firstly the \text{try} family of functions:

\[
\begin{array}{l}
\text{tryAll} :: a \to \text{IO} (\text{Either Exception a}) \\
\text{tryAllIO} :: \text{IO} a \to \text{IO} (\text{Either Exception a}) \\
\text{try} :: (\text{Exception} \to \text{Maybe b}) \to a \to \text{IO} (\text{Either b a}) \\
\text{tryIO} :: (\text{Exception} \to \text{Maybe b}) \to \text{IO} a \to \text{IO} (\text{Either b a})
\end{array}
\]

The simplest version is \text{tryAll}. It takes a single argument, evaluates it (as if you’d applied \text{seq} to it), and returns either \text{Right a} if the evaluation succeeded with result \text{a}, or \text{Left e} if an exception was raised, where \text{e} is the exception. Note that due to Haskell’s unspecified evaluation order, an expression may return one of several possible exceptions: consider the expression \text{error "urk" + 1 \, \div\, 0}. Does \text{tryAll} return \text{Just (ErrorCall "urk")} or \text{Just (ArithError DivideByZero)}? The answer is "either": \text{tryAll} makes a non-deterministic choice about which exception to return. If you call it again, you might get a different exception back. This is ok, because \text{tryAll} is an IO computation.

\text{tryAllIO} is the same as \text{tryAll} except that the argument to evaluate is an IO computation. Don’t try to use \text{tryAll} to catch exceptions in IO computations: in GHC an expression of type \text{IO a} is in fact a function, so evaluating it does nothing at all (and therefore raises no exceptions). Hence the need for \text{tryAllIO}, which runs IO computations properly.

The functions \text{try} and \text{tryIO} take an extra argument which is an exception predicate, a function which selects which type of exceptions we’re interested in. The full set of exception predicates is given below:

\[
\begin{array}{l}
\text{justIoErrors} :: \text{Exception} \to \text{Maybe IOError} \\
\text{justArithExceptions} :: \text{Exception} \to \text{Maybe ArithException} \\
\text{justErrors} :: \text{Exception} \to \text{Maybe String} \\
\text{justDynExceptions} :: \text{Exception} \to \text{Maybe Dynamic} \\
\text{justAssertions} :: \text{Exception} \to \text{Maybe String} \\
\text{justAsyncExceptions} :: \text{Exception} \to \text{Maybe AsyncException}
\end{array}
\]

For example, to catch just calls to ’error’ we could use something like

\[
\text{result} \leftarrow \text{try justErrors thing_to_try}
\]

Any other exceptions which aren’t matched by the predicate are re-raised, and may be caught by an enclosing \text{try} or \text{catch}. 
6. Exception

6.2 The catch functions

The catch family is similar to the try family:

\[
\begin{align*}
\text{catchAll} & : a \rightarrow (\text{Exception} \rightarrow \text{IO} \ a) \rightarrow \text{IO} \ a \\
\text{catchAllIO} & : \text{IO} \ a \rightarrow (\text{Exception} \rightarrow \text{IO} \ a) \rightarrow \text{IO} \ a \\
\text{catch} & : (\text{Exception} \rightarrow \text{Maybe} \ b) \rightarrow a \rightarrow (b \rightarrow \text{IO} \ a) \rightarrow \text{IO} \ a \\
\text{catchIO} & : (\text{Exception} \rightarrow \text{Maybe} \ b) \rightarrow \text{IO} \ a \rightarrow (b \rightarrow \text{IO} \ a) \rightarrow \text{IO} \ a
\end{align*}
\]

The difference is that instead of returning an Either type as the result, the catch functions take a handler argument which is invoked in the case that an exception was raised while evaluating the first argument.

\text{catch} and \text{catchIO} take exception predicate arguments in the same way as \text{try} and \text{tryIO}.

Note that \text{catchIO justIoErrors} is identical to \text{IO.catch}. In fact, the implementation of \text{IO.errors} in GHC uses exceptions "under the hood".

Also, don’t forget to \text{import Prelude hiding (catch)} when using this library, to avoid the name clash between \text{Exception.catch} and \text{IO.catch}.

6.3 Dynamic Exceptions

Because the \text{Exception} datatype isn’t extendible, we added an interface for throwing and catching exceptions of type \text{Dynamic} (see Section 5 (Dynamic)), which allows exception values of any type in the \text{Typeable} class to be thrown and caught.

\[
\begin{align*}
\text{throwDyn} & : \text{Typeable} \ \text{exception} \Rightarrow \text{exception} \rightarrow b \\
\text{catchDyn} & : \text{Typeable} \ \text{exception} \Rightarrow \text{IO} \ a \rightarrow (\text{exception} \rightarrow \text{IO} \ a) \rightarrow \text{IO} \ a
\end{align*}
\]

The \text{catchDyn} function only catches exceptions of the required type; all other exceptions are re-thrown as with \text{catchIO} and friends above.

6.4 Other Utilities

The bracket functions are useful for making sure that resources are released properly by code that may raise exceptions:

\[
\begin{align*}
\text{bracket} & : \text{IO} \ a \rightarrow (a \rightarrow \text{IO} \ b) \rightarrow (a \rightarrow \text{IO} \ c) \rightarrow \text{IO} \ c \\
\text{bracket_} & : \text{IO} \ a \rightarrow \text{IO} \ b \rightarrow \text{IO} \ c \rightarrow \text{IO} \ c \\
\text{finally} & : \text{IO} \ a \rightarrow \text{IO} \ b \rightarrow \text{IO} \ b
\end{align*}
\]

For example, to open a file, do some work on it and then close it again, we might use something like:

\[
\begin{align*}
\text{process_file} = \text{bracket} \ (\text{openFile "filename"}) \ \text{closeFile} \\
& \hspace{1em} \begin{array}{c}
\text{do} \\
\ldots \\
\end{array}
\end{align*}
\]
bracket works as follows: it executes its first argument ("open"), then its third argument, followed finally by its second argument ("close"). If the third argument happened to raise an exception, then the close operation will still be performed, and the exception will be re-raised.

This means that in the example above the file will always be closed, even if an error occurs during processing.

The arguments to bracket are in this order so that we can partially apply it, like:

```
withFile name = bracket (openFile name) closeFile
```

The bracket function is a variant of bracket that throws away the result of the open, and finally is an even simpler version where we just want some closing code.

7 Foreign

This module provides the ForeignObj type, which is a Haskell reference to an object in the outside world. Foreign objects are boxed versions of Addr#, the only reason for their existence is so that they can be used with finalisers (see Section 16.6 (Finalisation for foreign objects)).

```
module Foreign where
  data ForeignObj -- abstract, instance of: Eq

  makeForeignObj :: Addr{-object-} -> IO ForeignObj
  writeForeignObj :: ForeignObj -> Addr{-new value-} -> IO ()
```

In addition to the above, the following operations for indexing via a ForeignObj are also, mirrored on the same operations provided over Addr:

```
indexCharOffForeignObj :: ForeignObj -> Int -> Char
indexIntOffForeignObj :: ForeignObj -> Int -> Int
indexAddrOffForeignObj :: ForeignObj -> Int -> Addr
indexFloatOffForeignObj :: ForeignObj -> Int -> Float
indexDoubleOffForeignObj :: ForeignObj -> Int -> Double
indexWord8OffForeignObj :: ForeignObj -> Int -> Word8
indexWord16OffForeignObj :: ForeignObj -> Int -> Word16
indexWord32OffForeignObj :: ForeignObj -> Int -> Word32
indexWord64OffForeignObj :: ForeignObj -> Int -> Word64
indexInt8OffForeignObj :: ForeignObj -> Int -> Int8
indexInt16OffForeignObj :: ForeignObj -> Int -> Int16
indexInt32OffForeignObj :: ForeignObj -> Int -> Int32
indexInt64OffForeignObj :: ForeignObj -> Int -> Int64

-- read value out of mutable memory
readCharOffForeignObj :: ForeignObj -> Int -> IO Char
readIntOffForeignObj :: ForeignObj -> Int -> IO Int
readAddrOffForeignObj :: ForeignObj -> Int -> IO Addr
readFloatOffForeignObj :: ForeignObj -> Int -> IO Float
readDoubleOffForeignObj :: ForeignObj -> Int -> IO Double
```
8 GetOpt

The GetOpt library contains Sven Panne’s Haskell implementation of getopt, providing features nigh-on identical to GNU getopt:

```haskell
-- representing a single option:
data OptDescr a = Option [Char] -- list of short option characters 
                 [String] -- list of long option strings (without "--") 
                 (ArgDescr a) -- argument descriptor 
                 String -- explanation of option for user

-- argument option:
data ArgDescr a = NoArg a -- no argument expected 
                | ReqArg (String -> a) String -- option requires argument 
                | OptArg (Maybe String -> a) String -- optional argument

usageInfo :: String -- header 
     -> [OptDescr a] -- options recognised 
     -> String -- nicely formatted description of options

getOpt :: ArgOrder a -- non-option handling 
     -> [OptDescr a] -- options recognised
```

```haskell
readWord8OffForeignObj :: ForeignObj -> Int -> IO Word8
readWord16OffForeignObj :: ForeignObj -> Int -> IO Word16
readWord32OffForeignObj :: ForeignObj -> Int -> IO Word32
readWord64OffForeignObj :: ForeignObj -> Int -> IO Word64
readInt8OffForeignObj :: ForeignObj -> Int -> IO Int8
readInt16OffForeignObj :: ForeignObj -> Int -> IO Int16
readInt32OffForeignObj :: ForeignObj -> Int -> IO Int32
readInt64OffForeignObj :: ForeignObj -> Int -> IO Int64
writeCharOffForeignObj :: ForeignObj -> Int -> Char -> IO ()
writeIntOffForeignObj :: ForeignObj -> Int -> Int -> IO ()
writeAddrOffForeignObj :: ForeignObj -> Int -> Addr -> IO ()
writeFloatOffForeignObj :: ForeignObj -> Int -> Float -> IO ()
writeDoubleOffForeignObj :: ForeignObj -> Int -> Double -> IO ()
writeWord8OffForeignObj :: ForeignObj -> Int -> Word8 -> IO ()
writeWord16OffForeignObj :: ForeignObj -> Int -> Word16 -> IO ()
writeWord32OffForeignObj :: ForeignObj -> Int -> Word32 -> IO ()
writeWord64OffForeignObj :: ForeignObj -> Int -> Word64 -> IO ()
writeInt8OffForeignObj :: ForeignObj -> Int -> Int8 -> IO ()
writeInt16OffForeignObj :: ForeignObj -> Int -> Int16 -> IO ()
writeInt32OffForeignObj :: ForeignObj -> Int -> Int32 -> IO ()
writeInt64OffForeignObj :: ForeignObj -> Int -> Int64 -> IO ()
```
The command-line options recognised is described by a list of `OptDescr` values. The `OptDescr` describes the long and short strings that recognise the option, together with a help string and info on whether the option takes extra arguments, if any.

From a list of option values, `usageInfo` returns a nicely formatted string that enumerates the different options supported together with a short message about what.

To decode a command-line with respect to a list of options, `getOpt` is used. It processes the command-line, and returns the list of values that matched (and those that didn’t). The first argument to `getOpt` controls whether the user is to give the options in any old order or not.

To hopefully illuminate the role of the different `GetOpt` data structures, here’s the command-line options for a (very simple) compiler:

```haskell
module Opts where

import GetOpt
import Maybe ( fromMaybe )

data Flag
    = Verbose | Version
    | Input String | Output String | LibDir String
    deriving Show

options :: [OptDescr Flag]
options =
    [ Option ['v'] ["verbose"] (NoArg Verbose) "chatty output on stderr"
        , Option ['V','?'] ["version"] (NoArg Version) "show version number"
        , Option ['o'] ["output"] (OptArg outp "FILE") "output FILE"
        , Option ['c'] [] (OptArg inp "FILE") "input FILE"
        , Option ['L'] ["libdir"] (ReqArg LibDir "DIR") "library directory"
    ]

inp,outp :: Maybe String -> Flag
outp = Output . fromMaybe "stdout"
inp = Input . fromMaybe "stdout"

compilerOpts :: [String] -> IO ([Flag], [String])
compilerOpts argv =
    case (getOpt Permute options argv) of
```
(o,n,[] ) -> return (o,n)
(_,_,errs) -> fail (userError (concat errs ++ usageInfo header options))
where header = "Usage: ic [OPTION...] files..."

9 IOExts

This library provides the following extensions to the IO monad:

- The operations `fixIO`, `unsafePerformIO` and `unsafeInterleaveIO` described in [?]
- References (aka mutable variables) and mutable arrays (but no form of mutable byte arrays)
- `openFileEx` extends the standard `openFile` action with support for opening binary files.
- `performGC` triggers an immediate garbage collection
- When called, `trace` prints the string in its first argument to standard error, before returning the second argument as its result. The `trace` function is not referentially transparent, and should only be used for debugging, or for monitoring execution. Some implementations of `trace` may decorate the string that’s output to indicate that you’re tracing.
- `unsafePtrEq` compares two values for pointer equality without evaluating them. The results are not referentially transparent and may vary significantly from one compiler to another or in the face of semantics-preserving program changes. However, pointer equality is useful in creating a number of referentially transparent constructs such as this simplified memoisation function:

```haskell
> cache :: (a -> b) -> (a -> b)
> cache f = \x -> unsafePerformIO (check x)
> where
>   ref = unsafePerformIO (newIORef (error "cache", error "cache"))
>   check x = readIORef ref >>= \(x',a) ->
>     if x `unsafePtrEq` x’ then
>       return a
>     else
>       let a = f x in
>       writeIORef ref (x, a) >>
>       return a
```

module IOExts where

```haskell
fixIO :: (a -> IO a) -> IO a
unsafePerformIO :: IO a -> a
unsafeInterleaveIO :: IO a -> IO a

data IORef a -- mutable variables containing values of type a
newIORef :: a -> IO (IORef a)
readIORef :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()
instance Eq (IORef a)
```
10. Int

This library provides signed integers of various sizes. The types supported are as follows:

<table>
<thead>
<tr>
<th>type</th>
<th>number of bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int8</td>
<td>8</td>
</tr>
<tr>
<td>Int16</td>
<td>16</td>
</tr>
<tr>
<td>Int32</td>
<td>32</td>
</tr>
<tr>
<td>Int64</td>
<td>64</td>
</tr>
</tbody>
</table>

For each type \( I \) above, we provide the following instances.

data I -- Signed Ints
iToInt :: I -> Int -- not provided for Int64
intToI :: Int -> I -- not provided for Int64
instance Eq I
instance Ord I
instance Show I
instance Read I
instance Bounded I
instance Num I
instance Real I
instance Integral I
instance Enum I
instance Ix I
instance Bits I

Plus the coercion functions

\[ \text{int8ToInt16 :: Int8} \to \text{Int16} \]
The `NumExts` interface collects various numeric operations that have proven to be commonly useful.

```haskell
int8ToInt32 :: Int8 -> Int32
int8ToInt64 :: Int8 -> Int64
int16ToInt8 :: Int16 -> Int8
int16ToInt32 :: Int16 -> Int32
int16ToInt64 :: Int16 -> Int64
int32ToInt8 :: Int32 -> Int8
int32ToInt16 :: Int32 -> Int16
int32ToInt64 :: Int32 -> Int64
int64ToInt8 :: Int64 -> Int8
int64ToInt16 :: Int64 -> Int16
int64ToInt32 :: Int64 -> Int32
int8ToInt :: Int8 -> Int
int16ToInt :: Int16 -> Int
int32ToInt :: Int32 -> Int
int64ToInt :: Int64 -> Int
integerToInt8 :: Integer -> Int8
integerToInt16 :: Integer -> Int16
integerToInt32 :: Integer -> Int32
integerToInt64 :: Integer -> Int64
int64ToInteger :: Int64 -> Integer
int32ToInteger :: Int32 -> Integer
int16ToInteger :: Int16 -> Integer
int8ToInteger :: Int8 -> Integer

doubleToFloat :: Double -> Float
floatToDouble :: Float -> Double
```

- The rules that hold for `Enum` instances over a bounded type such as `Int` (see the section of the Haskell report dealing with arithmetic sequences) also hold for the `Enum` instances over the various `Int` types defined here.
- Hugs does not provide `Int64` at the moment.
showHex :: Integral a => a -> ShowS
showOct :: Integral a => a -> ShowS
showBin :: Integral a => a -> ShowS

showIntAtBase :: Integral a -> a -- base
-> (a -> Char) -- digit to char
-> a -- number to show.
-> ShowS

Notes:

• If `doubleToFloat` is applied to a `Double` that is within the representable range for `Float`, the result may be the next higher or lower representable `Float` value. If the `Double` is out of range, the result is undefined.

• No loss of precision occurs in the other direction with `floatToDouble`, the floating value remains unchanged.

• `showOct`, `showHex` and `showBin` will prefix 0o, 0x and 0b, respectively. Like `Numeric.showInt`, these show functions work on positive numbers only.

• `showIntAtBase` is the more general function for converting a number at some base into a series of characters. The above `show*` functions use it, for instance, here’s how `showHex` could be defined

```haskell
showHex :: Integral a => a -> ShowS
showHex n r = showString "0x" $ showIntAtBase 16 (toChrHex n r) 
  where toChrHex d | d < 10 = chr (ord '0' + fromIntegral d) 
                     | otherwise = chr (ord 'a' + fromIntegral (d - 10))
```

12 Pretty

This library contains Simon Peyton Jones’ implementation of John Hughes’s pretty printer combinators.

```haskell
module Pretty where

infixl 6 <>
infixl 6 <+>
infixl 5 $$, $$+
data Doc -- the Document datatype

-- The primitive Doc values
empty :: Doc

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values

-- The primitive Doc values
This library provides support for strict state threads, as described in the PLDI ’94 paper by John Launchbury and Simon Peyton Jones [?]. In addition to the monad ST, it also provides mutable variables STRef and mutable arrays STArray.

module ST( module ST, module Monad ) where
import Monad

data ST s a -- abstract type
runST :: forall a. (forall s. ST s a) -> a
fixST :: (a -> ST s a) -> ST s a
unsafeInterleaveST :: ST s a -> ST s a
instance Functor (ST s)
instance Monad (ST s)

data STRef s a -- mutable variables in state thread s
               -- containing values of type a.
newSTRef     :: a -> ST s (STRef s a)
readSTRef    :: STRef s a -> ST s a
writeSTRef   :: STRef s a -> a -> ST s ()
instance Eq (STRef s a)

data STArray s ix elt -- mutable arrays in state thread s
                        -- indexed by values of type ix
                        -- containing values of type a.
newSTArray   ::Ix ix => (ix,ix) -> elt -> ST s (STArray s ix elt)
boundsSTArray::Ix ix => STArray s ix elt -> (ix, ix)
readSTArray  ::Ix ix => STArray s ix elt -> ix -> elt -> ST s elt
writeSTArray ::Ix ix => STArray s ix elt -> ix -> elt -> ST s ()
thawSTArray  ::Ix ix => Array ix elt -> ST s (STArray s ix elt)
freezeSTArray::Ix ix => STArray s ix elt -> ST s (Array ix elt)
unsafeFreezeSTArray::Ix ix => STArray s ix elt -> ST s (Array ix elt)
instance Eq (STArray s ix elt)

unsafeIOToST :: IO a -> ST s a
stToIO      :: ST s a -> IO a

Notes:

- GHC also supports ByteArrays — these aren’t supported by Hugs yet.
- The operations freezeSTArray and thawSTArray convert mutable arrays to and from immutable arrays. Semantically, they are identical to copying the array and they are usually implemented that way. The operation unsafeFreezeSTArray is a faster version of freezeSTArray which omits the copying step. It’s a safe substitute for freezeSTArray if you don’t modify the mutable array after freezing it.
- Hugs provides thenLazyST and thenStrictST so that you can import LazyST (say) and still use the strict instance in those places where it matters. GHC implements LazyST and ST using different types, so this isn’t possible.
- Operations for coercing an ST action into an IO one, and vice versa are also provided. Notice that coercing an IO action into an ST action is ‘lossy’, since any exception raised within the IO action will not be caught within the ST monad, as it doesn’t support (monadic) exceptions.

14 Stable

This module provides two kinds of stable references to Haskell objects, stable names and stable pointers.
14. Stable

14.1 Stable Pointers

A stable pointer is a reference to a Haskell expression that can be passed to foreign functions via the foreign function interface.

Normally a Haskell object will move around from time to time, because of garbage collection, hence we can’t just pass the address of an object to a foreign function and expect it to remain valid. Stable pointers provide a level of indirection so that the foreign code can get the “real address” of the Haskell object by calling `deRefStablePtr` on the stable pointer object it has.

The Haskell interface provided by the `Stable` module is as follows:

```haskell
data StablePtr a -- abstract, instance of: Eq.
makeStablePtr :: a -> IO (StablePtr a)
defRefStablePtr :: StablePtr a -> IO a
freeStablePtr :: StablePtr a -> IO ()
```

Care must be taken to free stable pointers that are no longer required using the `freeStablePtr` function, otherwise two bad things can happen:

- The object referenced by the stable pointer will be retained in the heap.
- The runtime system’s internal stable pointer table will grow, which imposes an overhead on garbage collection.

Notes:

- If `sp1 :: StablePtr` and `sp2 :: StablePtr` and `sp1 == sp2` then `sp1` and `sp2` are either the same stable pointer, or they were created by calls to `makeStablePtr` on the same object. Another way to say this is ”every time you call `makeStablePtr` on an object you get back the same stable pointer”.
- The reverse is not necessarily true: if two stable pointers are not equal, it doesn’t mean that they don’t refer to the same Haskell object (although they probably don’t).
- Calling `deRefStablePtr` on a stable pointer which has previously been freed results in undefined behaviour.

The C interface (which is brought into scope by `#include <Stable.h>`) is as follows:

```c
typedef StablePtr /* abstract, probably an unsigned long */
extern StgPtr deRefStablePtr(StgStablePtr stable_ptr);
static void freeStablePtr(StgStablePtr sp);
static StgStablePtr splitStablePtr(StgStablePtr sp);
```

The functions `deRefStablePtr` and `freeStablePtr` are equivalent to the Haskell functions of the same name above.

The function `splitStablePtr` allows a stable pointer to be duplicated without making a new one with `makeStablePtr`. The stable pointer won’t be removed from the runtime system’s internal table until `freeStablePtr` is called on both pointers.
14.2 Stable Names

A haskell object can be given a *stable name* by calling `makeStableName` on it. Stable names solve the following problem: suppose you want to build a hash table with Haskell objects as keys, but you want to use pointer equality for comparison; maybe because the keys are large and hashing would be slow, or perhaps because the keys are infinite in size. We can’t build a hash table using the address of the object as the key, because objects get moved around by the garbage collector, meaning a re-hash would be necessary after every garbage collection.

Enter stable names. A stable name is an abstract entity that supports equality and hashing, with the following interface:

```haskell
data StableName a -- abstract, instance Eq.
makeStableName :: a -> IO (StableName a)
hashStableName :: StableName a -> Int
```

All these operations run in constant time.

Stable names have the following properties:

1. If `sn1 :: StablePtr` and `sn2 :: StablePtr` and `sn1 == sn2` then `sn1` and `sn2` are either the same stable name, or they were created by calls to `makeStableName` on the same object.

2. The reverse is not necessarily true: if two stable names are not equal, it doesn’t mean that they don’t refer to the same Haskell object (although they probably don’t).

3. There is no `freeStableName` operation. Stable names are reclaimed by the runtime system when they are no longer needed.

4. There is no `deRefStableName` operation. You can’t get back from a stable name to the original Haskell object. The reason for this is that the existence of a stable name for an object doesn’t guarantee the existence of the object itself; it can still be garbage collected.

5. There is a `hashStableName` operation, which converts a stable name to an `Int`. The `Int` returned is not necessarily unique (that is, it doesn’t satisfy property (1) above), but it can be used for building hash tables of stable names.

Properties (1) and (2) are similar to stable pointers, but the key differences are that you can’t get back to the original object from a stable name, and you can convert one to an `Int` for hashing.

15 LazyST

This library is identical to `ST` except that the `ST` monad instance is `lazy`. The lazy ST monad tends to be more prone to space leaks than the strict version, so most programmers will use the former unless laziness is explicitly required. LazyST provides two additional operations:

```haskell
lazyToStrictST :: LazyST.ST s a -> ST.ST s a
strictToLazyST :: ST.ST s a -> LazyST.ST s a
```
These are used to convert between lazy and strict state threads. The semantics with respect to laziness are as you would expect: the strict state thread passed to `strictToLazyST` is not performed until the result of the lazy state thread it returns is demanded.

## 16 Weak

The `Weak` library provides a "weak pointer" abstraction, giving the user some control over the garbage collection of specified objects, and allowing objects to be "finalised" with an arbitrary Haskell IO computation when they die.

Weak pointers partially replace the old foreign object interface, as we will explain later.

### 16.1 Module Signature

```haskell
module Weak (
  Weak, -- abstract
  -- instance Eq (Weak v)
  mkWeak, -- :: k -> v -> IO () -> IO (Weak v)
  deRefWeak, -- :: Weak v -> IO (Maybe v)
  finalise, -- :: Weak v -> IO ()
  -- Not yet implemented
  -- replaceFinaliser -- :: Weak v -> IO () -> IO ()
  mkWeakPtr, -- :: k -> IO () -> IO (Weak k)
  mkWeakPair, -- :: k -> v -> IO () -> IO (Weak (k,v))
  mkWeakNoFinaliser, -- :: k -> v -> IO (Weak v)
  addFinaliser, -- :: k -> IO () -> IO ()
  addForeignFinaliser -- :: ForeignObj -> IO () -> IO ()
) where
```

### 16.2 Weak pointers

In general terms, a weak pointer is a reference to an object that is not followed by the garbage collector — that is, the existence of a weak pointer to an object has no effect on the lifetime of that object. A weak pointer can be de-referenced to find out whether the object it refers to is still alive or not, and if so to return the object itself.

Weak pointers are particularly useful for caches and memo tables. To build a memo table, you build a data structure mapping from the function argument (the key) to its result (the value). When you apply the function to a new argument you first check whether the key/value pair is already in the memo table. The key point is that the memo table itself should not keep the key and value alive. So the table should contain a weak pointer to the key, not an ordinary pointer. The pointer to the value must not be weak, because the only reference to the value might indeed be from the memo table.

So it looks as if the memo table will keep all its values alive for ever. One way to solve this is to purge the table occasionally, by deleting entries whose keys have died.
The weak pointers in this library support another approach, called finalisation. When the key referred to by a weak pointer dies, the storage manager arranges to run a programmer-specified finaliser. In the case of memo tables, for example, the finaliser could remove the key/value pair from the memo table.

Another difficulty with the memo table is that the value of a key/value pair might itself contain a pointer to the key. So the memo table keeps the value alive, which keeps the key alive, even though there may be no other references to the key so both should die. The weak pointers in this library provide a slight generalisation of the basic weak-pointer idea, in which each weak pointer actually contains both a key and a value. We describe this in more detail below.

### 16.3 The simple interface

```haskell
mkWeakPtr :: a -> IO () -> IO (Weak a)
deRefWeak :: Weak a -> IO (Maybe a)
addFinaliser :: a -> IO () -> IO ()
```

`mkWeakPtr` takes a value of any type `a`, and a finaliser of type `IO ()`, and returns a weak pointer object referring to the value, of type `Weak a`. It is in the `IO` monad because it has the side effect of arranging that the finaliser will be run when the object dies. In what follows, a “weak pointer object”, or “weak pointer” for short, means precisely “a Haskell value of type `Weak t`” for some type `t`. A weak pointer (object) is a first-class Haskell value; it can be passed to functions, stored in data structures, and so on.

`deRefWeak` dereferences a weak pointer, returning `Just v` if the value is still alive. If the key has already died, then `deRefWeak` returns `Nothing`; that’s why it’s in the `IO` monad - the return value of `deRefWeak` depends on when the garbage collector runs.

`addFinaliser` is just another name for `mkWeakPtr` except that it throws the weak pointer itself away. (The runtime system will remember that the weak pointer and hence the finaliser exists even if the program has forgotten it.)

```haskell
addFinaliser :: a -> IO () -> IO ()
addFinaliser v f = do { mkWeakPtr v f; return () }
```

The effect of `addFinaliser` is simply that the finaliser runs when the referenced object dies.

The following properties hold:

- `deRefWeak` returns the original object until that object is considered dead; it returns `Nothing` subsequently.

- Every finaliser will eventually be run, exactly once, either soon after the object dies, or at the end of the program. There is no requirement for the programmer to hold onto the weak pointer itself; finalisation is completely unaffected by whether the weak pointer itself is alive.

- There may be multiple weak pointers to a single object. In this case, the finalisers for each of these weak pointers will all be run in some arbitrary order, or perhaps concurrently, when the object dies. If the programmer specifies a finaliser that assumes it has the only reference to an object (for example, a file that it wishes to close), then the programmer must ensure that there is only one such finaliser.

- The storage manager attempts to run the finaliser(s) for an object soon after the object dies, but promptness is not guaranteed. (What is guaranteed is that the finaliser will eventually run, exactly once.)
At the moment when a finaliser is run, a call to \texttt{deRefWeak} will return \texttt{Nothing}.

A finaliser may contain a pointer to the object, but that pointer will not keep the object alive. For example:

```haskell
f :: Show a => a -> IO a
f x = addFinaliser x (print (show x))
```

Here the finaliser \texttt{print (show x)} contains a reference to \texttt{x} itself, but that does not keep \texttt{x} alive. When that is the only reference to \texttt{x}, the finaliser is run; and the message appears on the screen.

A finaliser may even resurrect the object, by (say) storing it in some global data structure.

### 16.4 The general interface

The \texttt{Weak} library offers a slight generalisation of the simple weak pointers described so far:

```haskell
mkWeak :: k -> v -> IO () -> IO (Weak v)
```

\texttt{mkWeak} takes a key of any type \texttt{k} and a value of any type \texttt{v}, as well as a finaliser, and returns a weak pointer of type \texttt{Weak v}.

\texttt{deRefWeak} returns the \texttt{value} only, not the key, as its type (given above) implies:

```haskell
deRefWeak :: Weak a -> IO (Maybe a)
```

However, \texttt{deRefWeak} returns \texttt{Nothing} if the \texttt{key}, not the value, has died. Furthermore, references from the value to the key do not keep the key alive, in the same way that the finaliser does not keep the key alive.

Simple weak pointers are readily defined in terms of these more general weak pointers:

```haskell
mkWeakPtr :: a -> IO () -> IO (Weak a)
mkWeakPtr v f = mkWeak v v f
```

These more general weak pointers are enough to implement memo tables properly.

A weak pointer can be finalised early, using the \texttt{finalise} operation:

```haskell
finalise :: Weak v -> IO ()
```

When you don’t need a finaliser, we provide the following operation:

```haskell
mkWeakNoFinaliser :: k -> v -> IO (Weak v)
mkWeakNoFinaliser k v = mkWeak k v (return ())
```

Which creates a weak pointer with a null finaliser. Lots of null finalisers can be expensive, because each one runs in a separate thread, so the intention is that \texttt{mkWeakNoFinaliser} avoids all the extra costs by generating a special kind of weak pointer without a finaliser. So although the semantics of \texttt{mkWeakNoFinaliser} is as given above, its actual implementation is somewhat different.
16.5 A precise semantics

The above informal specification is fine for simple situations, but matters can get complicated. In particular, it needs to be clear exactly when a key dies, so that any weak pointers that refer to it can be finalised. Suppose, for example, the value of one weak pointer refers to the key of another...does that keep the key alive?

The behaviour is simply this:

- If a weak pointer (object) refers to an unreachable key, it may be finalised.
- Finalisation means (a) arrange that subsequent calls to deRefWeak return Nothing; and (b) run the finaliser.

This behaviour depends on what it means for a key to be reachable. Informally, something is reachable if it can be reached by following ordinary pointers from the root set, but not following weak pointers. We define reachability more precisely as follows A heap object is reachable if:

- It is a member of the root set.
- It is directly pointed to by a reachable object, other than a weak pointer object.
- It is a weak pointer object whose key is reachable.
- It is the value or finaliser of an object whose key is reachable.

The root set consists of all runnable threads, and all stable pointers (see Section 14.2 (Stable Pointers)). NOTE: currently all top-level objects are considered to be reachable, although we hope to remove this restriction in the future. A Char or small Int will also be constantly reachable, since the garbage collector replaces heap-resident Chars and small Ints with pointers to static copies.

Notice that a pointer to the key from its associated value or finaliser does not make the key reachable. However, if the key is reachable some other way, then the value and the finaliser are reachable, and so, therefore, are any other keys they refer to directly or indirectly.

16.6 Finalisation for foreign objects

A foreign object is some data that lives outside the Haskell heap, for example some malloced data in C land. It’s useful to be able to know when the Haskell program no longer needs the malloced data, so it can be freed. We can use weak pointers and finalisers for this, but we have to be careful: the foreign data is usually referenced by an address, i.e. an Addr (see Section 2 (Addr)), and we must retain the invariant that if the Haskell program still needs the foreign object, then it retains the Addr object in the heap. This invariant isn’t guaranteed to hold if we use Addr, because an Addr consists of a box around a raw address Addr#. If the Haskell program can manipulate the Addr# object independently of the heap-resident Addr, then the foreign object could
be inadvertently finalised early, because a weak pointer to the Addr would find no more references to its key and trigger the finaliser despite the fact that the program still holds the Addr# and intends to use it.

To avoid this somewhat subtle race condition, we use another type of foreign address, called ForeignObj (see Section 7 (Foreign)). Historical note: ForeignObj is identical to the old ForeignObj except that it no longer supports finalisation – that’s provided by the weak pointer finalisation mechanism above.

A ForeignObj is basically an address, but the ForeignObj itself is a heap-resident object and can therefore be watched by weak pointers. A ForeignObj can be passed to C functions (in which case the C function gets a straightforward pointer), but it cannot be decomposed into an Addr#.

## 17 Word

This library provides unsigned integers of various sizes. The types supported are as follows:

<table>
<thead>
<tr>
<th>type</th>
<th>number of bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word8</td>
<td>8</td>
</tr>
<tr>
<td>Word16</td>
<td>16</td>
</tr>
<tr>
<td>Word32</td>
<td>32</td>
</tr>
<tr>
<td>Word64</td>
<td>64</td>
</tr>
</tbody>
</table>

For each type \( W \) above, we provide the following functions and instances. The type \( I \) refers to the signed integer type of the same size.

```haskell
data W = -- Unsigned Ints
    instance Eq W
    instance Ord W
    instance Show W
    instance Read W
    instance Bounded W
    instance Num W
    instance Real W
    instance Integral W
    instance Enum W
    instance Ix W
    instance Bits W
```

Plus

```haskell
word8ToWord16 :: Word8 -> Word16
word8ToWord32 :: Word8 -> Word32
word8ToWord64 :: Word8 -> Word64

word16ToWord8 :: Word16 -> Word8
word16ToWord32 :: Word16 -> Word32
```
word16ToWord64 :: Word16 -> Word64

word32ToWord8 :: Word32 -> Word8
word32ToWord16 :: Word32 -> Word16
word32ToWord64 :: Word32 -> Word64

word64ToWord8 :: Word64 -> Word8
word64ToWord16 :: Word64 -> Word16
word64ToWord32 :: Word64 -> Word32

word8ToInt :: Word8 -> Int
word16ToInt :: Word16 -> Int
word32ToInt :: Word32 -> Int
word64ToInt :: Word64 -> Int

intToWord8 :: Int -> Word8
intToWord16 :: Int -> Word16
intToWord32 :: Int -> Word32
intToWord64 :: Int -> Word64

word64ToInteger :: Word64 -> Integer
integerToWord64 :: Integer -> Word64

Notes:

- All arithmetic is performed modulo $2^n$
  One non-obvious consequence of this is that `negate` should *not* raise an error on negative arguments.

- The coercion `wToI` converts an unsigned n-bit value to the signed n-bit value with the same representation. For example, `word8ToInt8 0xff = -1`. Likewise, `iToW` converts signed n-bit values to the corresponding unsigned n-bit value.

- Use `Prelude.fromIntegral :: (Integral a, Num b) => a -> b` to coerce between different sizes or to preserve sign when converting between values of the same size.

- It would be very natural to add a type a type `Natural` providing an unbounded size unsigned integer — just as `Integer` provides unbounded size signed integers. We do not do that yet since there is no demand for it. Doing so would require `Bits.bitSize` to return `Maybe Int`.

- The rules that hold for `Enum` instances over a bounded type such as `Int` (see the section of the Haskell report dealing with arithmetic sequences) also hold for the `Enum` instances over the various `Word` types defined here.

Implementation notes:

- Hugs only provides `Eq`, `Ord`, `Read` and `Show` instances for `Word64` at the moment.