Haskell Libraries

The GHC Team

Haskell Libraries

by The GHC Team

Table of Contents

Chapter 1. The `concurrent` category: concurrency support

1.1. Concurrent Haskell

GHC and Hugs both provide concurrency extensions, as described in Concurrent Haskell (http://research.microsoft.com/Users/simonpj/Papers/concurrent-haskell.ps.gz).

Concurrency in GHC and Hugs is "lightweight", which means that both thread creation and context switching overheads are extremely low. Scheduling of Haskell threads is done internally in the Haskell runtime system, and doesn't make use of any operating system-supplied thread packages.

Haskell threads can communicate via `MVar`s, a kind of synchronised mutable variable. Several common concurrency abstractions can be built from `MVar`s, and these are provided by the `Concurrent` library, which is described in the later sections. Threads may also communicate via exceptions.

1.2. Concurrency Basics

To gain access to the concurrency primitives, just `import Concurrent` in your Haskell module. In GHC, you also need to add the `-syslib concurrent` option to the command line.

To create a new thread, use `forkIO`:

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

This sparks off a new thread to run the `IO` computation passed as the first argument.

The returned `ThreadId` is an abstract type representing a handle to the newly created thread. The `ThreadId` type is an instance of both `Eq` and `Ord`, where the `Ord` instance implements an arbitrary total ordering over `ThreadId`s.

Threads may also be killed via the `ThreadId`:

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

this terminates the given thread (Note: `killThread` is not implemented in Hugs yet). Any work already done by the thread isn't lost: the computation is suspended until required by another thread. The memory used by the thread will be garbage collected if it isn't referenced from anywhere else.

More generally, an arbitrary exception (see Section 3.7) may be raised in any thread for which we have a `ThreadId`, with `raiseInThread`:

```
raiseInThread :: ThreadId -> Exception -> IO ()
```

Actually killThread just raises the ThreadKilled exception in the target thread, the normal action of which is to just terminate the thread. The target thread will stop whatever it was doing (even if it was blocked on an MVar or other computation) and handle the exception.

One important property of raiseInThread (and therefore killThread) is that they are *synchronous*. This means that after performing a raiseInThread operation, the calling thread can be certain that the target thread has received the exception. In other words, the target thread cannot perform any more processing unless it handles the exception that has just been raised in it. This is a useful property to know when dealing with race conditions: eg. if there are two threads that can kill each other, it is guaranteed that only one of the threads will get to kill the other.

The ThreadId for the current thread can be obtained with myThreadId:

```
myThreadId :: IO ThreadId
```

NOTE: if you have a ThreadId, you essentially have a pointer to the thread itself. This means the thread itself can't be garbage collected until you drop the ThreadId. This misfeature will hopefully be corrected at a later date.

1.3. Scheduling

GHC uses *preemptive multitasking*: context switches can occur at any time. At present, Hugs uses *cooperative multitasking*: context switches only occur when you use one of the primitives defined in this module. This means that programs such as:

```
main = forkIO (write 'a') » write 'b'
 where write c = putChar c » write c
```

will print either aaaaaaaaaaaaaa... or bbbbbbbbbbbb..., instead of some random interleaving of as and bs. In practice, cooperative multitasking is sufficient for writing simple graphical user interfaces.

The yield action forces a context-switch to any other currently runnable threads (if any), and is occasionally useful when implementing concurrency abstractions:

```
yield :: IO ()
```

1.3.1. Thread Waiting

Finally, there are operations to delay a concurrent thread, and to make one wait:

```
threadDelay     :: Int -> IO () - delay rescheduling for N microseconds
threadWaitRead  :: Int -> IO () -
wait for input on specified file descriptor
threadWaitWrite :: Int -> IO () - (read and write, respectively).
```

The `threadDelay` operation will cause the current thread to suspend for a given number of microseconds. Note that the resolution used by the Haskell runtime system's internal timer together with the fact that the thread may take some time to be rescheduled after the time has expired, means that the accuracy is more like 1/50 second.

`threadWaitRead` and `threadWaitWrite` can be used to block a thread until I/O is available on a given file descriptor. These primitives are used by the I/O subsystem to ensure that a thread waiting on I/O doesn't hang the entire system.

1.3.2. Blocking

Calling a foreign C procedure (such as `getchar`) that blocks waiting for input will block *all* threads, in both GHC and Hugs. The GHC I/O system uses non-blocking I/O internally to implement thread-friendly I/O, so calling standard Haskell I/O functions blocks only the thead making the call.

1.4. Concurrency abstractions

1.4.1. M-Vars

The `Concurrent` interface provides access to "M-Vars", which are *synchronising variables*.

`MVar`s are rendezvous points, mostly for concurrent threads. They begin either empty or full, and any attempt to read an empty `MVar` blocks. When an `MVar` is written, a single blocked thread may be freed. Reading an `MVar` toggles its state from full back to empty. Therefore, any value written to an `MVar` may only be read once. Multiple reads and writes are allowed, but there must be at least one read between any two writes. Interface:

```
data MVar a - abstract
instance Eq (MVar a)

newEmptyMVar      :: IO (MVar a)
newMVar           :: a -> IO (MVar a)
takeMVar          :: MVar a -> IO a
putMVar           :: MVar a -> a -> IO ()
readMVar          :: MVar a -> IO a
swapMVar          :: MVar a -> a -> IO a
isEmptyMVar       :: MVar a -> IO Bool
```

The operation `isEmptyMVar` returns a flag indicating whether the `MVar` is currently empty or filled in, i.e., will a thread block when performing a `takeMVar` on that `MVar` or not?

Please notice that the Boolean value returned from `isEmptyMVar` represent just a snapshot of the state of the `MVar`. By the time a thread gets to inspect the result and act upon it, other threads may have accessed the `MVar` and changed the 'filled-in' status of the variable.

The same proviso applies to `isEmptyChan` (next sub-section).

These two predicates are currently only supported by GHC.

1.4.2. Channel Variables

A *channel variable* (`CVar`) is a one-element channel, as described in the paper:

```
data CVar a
newCVar :: IO (CVar a)
putCVar :: CVar a -> a -> IO ()
getCVar :: CVar a -> IO a
```

1.4.3. Channels

A `Channel` is an unbounded channel:

```
data Chan a
newChan         :: IO (Chan a)
putChan         :: Chan a -> a -> IO ()
getChan         :: Chan a -> IO a
dupChan         :: Chan a -> IO (Chan a)
unGetChan       :: Chan a -> a -> IO ()
getChanContents :: Chan a -> IO [a]
```

1.4.4. Semaphores

General and quantity semaphores:

```
data QSem
newQSem     :: Int   -> IO QSem
waitQSem    :: QSem  -> IO ()
signalQSem  :: QSem  -> IO ()

data QSemN
newQSemN    :: Int   -> IO QSemN
```

```
signalQSemN :: QSemN -> Int -> IO ()
waitQSemN   :: QSemN -> Int -> IO ()
```

1.4.5. Merging Streams

Merging streams—binary and n-ary:

```
mergeIO  :: [a]   -> [a] -> IO [a]
nmergeIO :: [[a]] -> IO [a]
```

These actions fork one thread for each input list that concurrently evaluates that list; the results are merged into a single output list.

Note: Hugs does not provide the functions `mergeIO` or `nmergeIO` since these require preemptive multitasking.

1.4.6. Sample Variables

A *Sample variable* (`SampleVar`) is slightly different from a normal `MVar`:

- Reading an empty `SampleVar` causes the reader to block (same as `takeMVar` on empty `MVar`).
- Reading a filled `SampleVar` empties it and returns value. (same as `takeMVar`)
- Writing to an empty `SampleVar` fills it with a value, and potentially, wakes up a blocked reader (same as for `putMVar` on empty `MVar`).
- Writing to a filled `SampleVar` overwrites the current value. (different from `putMVar` on full `MVar`.)

```
type SampleVar a = MVar (Int, MVar a)

emptySampleVar :: SampleVar a -> IO ()
newSampleVar   :: IO (SampleVar a)
readSample     :: SampleVar a -> IO a
writeSample    :: SampleVar a -> a -> IO ()
```

1.5. The Concurrent library interface

The full interface for the `Concurrent` library is given below for reference:

```
module Concurrent where

data ThreadId    - thread identifiers
instance Eq  ThreadId
instance Ord ThreadId

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

data MVar a      - Synchronisation variables
instance Eq (MVar a)
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


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 ()
isEmptyChan      :: Chan a -> IO Bool
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 ()

data QSemN       - General/quantity semaphores
newQSemN         :: Int   -> IO QSemN
waitQSemN        :: QSemN -> Int -> IO ()
signalQSemN      :: QSemN -> Int -> IO ()

type SampleVar a - Sample variables
newEmptySampleVar:: IO (SampleVar a)
```

```
newSampleVar     :: a -> IO (SampleVar a)
emptySampleVar   :: SampleVar a -> IO ()
readSampleVar    :: SampleVar a -> IO a
writeSampleVar   :: SampleVar a -> a -> IO ()

threadDelay      :: Int -> IO ()
threadWaitRead   :: Int -> IO ()
threadWaitWrite  :: Int -> IO ()
```

1.6. GHC-specific concurrency issues

In a standalone GHC program, only the main thread is required to terminate in order for the process to terminate. Thus all other forked threads will simply terminate at the same time as the main thread (the terminology for this kind of behaviour is "daemonic threads").

If you want the program to wait for child threads to finish before exiting, you need to program this yourself. A simple mechanism is to have each child thread write to an `MVar` when it completes, and have the main thread wait on all the `MVar`s before exiting:

```
myForkIO :: IO () -> IO (MVar ())
myForkIO io = do
  mvar <- newEmptyMVar
  forkIO (io `finally` putMVar mvar ())
  return mvar
```

Note that we use `finally` from the `Exception` module to make sure that the `MVar` is written to even if the thread dies or is killed for some reason.

A better method is to keep a global list of all child threads which we should wait for at the end of the program:

```
children :: MVar [MVar ()]
children = unsafePerformIO (newMVar [])

waitForChildren :: IO ()
waitForChildren = do
  (mvar:mvars) <- takeMVar children
  putMVar children mvars
  takeMVar mvar
  waitForChildren

forkChild :: IO () -> IO ()
forkChild io = do
   mvar <- newEmptyMVar
   forkIO (p `finally` putMVar mvar ())
```

```
     childs <- takeMVar children
     putMVar children (mvar:childs)

later = flip finally

main =
  later waitForChildren $
  ...
```

Chapter 2. The `data` category: datatypes

2.1. The `FiniteMap` type

What functional programmers call a *finite map*, everyone else calls a *lookup table*.

Out code is derived from that in this paper: “S Adams "Efficient sets: a balancing act" Journal of functional programming 3(4) Oct 1993, pages 553-562” Guess what? The implementation uses balanced trees.

```
data FiniteMap key elt  - abstract

-      BUILDING
emptyFM         :: FiniteMap key elt
unitFM          :: key -> elt -> FiniteMap key elt
listToFM        :: Ord key => [(key,elt)] -> FiniteMap key elt
                        - In the case of duplicates, the last is taken

-      ADDING AND DELETING
                   - Throws away any previous binding
                   -
In the list case, the items are added starting with the
                   - first one in the list
addToFM         :: Ord key => FiniteMap key elt -> key -> elt  -
> FiniteMap key elt
addListToFM     :: Ord key => FiniteMap key elt -> [(key,elt)] -
> FiniteMap key elt

                 - Combines with previous binding
                 - In the combining function, the first argument is
                 - the "old" element, while the second is the "new" one.
addToFM_C       :: Ord key => (elt -> elt -> elt)
                           -> FiniteMap key elt -> key -> elt
                           -> FiniteMap key elt
addListToFM_C   :: Ord key => (elt -> elt -> elt)
                           -> FiniteMap key elt -> [(key,elt)]
                           -> FiniteMap key elt

                 -
Deletion doesn't complain if you try to delete something
                 - which isn't there
delFromFM       :: Ord key => FiniteMap key elt -> key   -
> FiniteMap key elt
delListFromFM   :: Ord key => FiniteMap key elt -> [key] -
> FiniteMap key elt
```

```
-      COMBINING
                 - Bindings in right argument shadow those in the left
plusFM          :: Ord key => FiniteMap key elt -> FiniteMap key elt
                           -> FiniteMap key elt

                   -
Combines bindings for the same thing with the given function
plusFM_C        :: Ord key => (elt -> elt -> elt)
                           -> FiniteMap key elt -> FiniteMap key elt -
> FiniteMap key elt

minusFM         :: Ord key => FiniteMap key elt -> FiniteMap key elt -
> FiniteMap key elt
                   -
(minusFM a1 a2) deletes from a1 any bindings which are bound in a2

intersectFM     :: Ord key => FiniteMap key elt -> FiniteMap key elt -
> FiniteMap key elt
intersectFM_C   :: Ord key => (elt -> elt -> elt)
                           -> FiniteMap key elt -> FiniteMap key elt -
> FiniteMap key elt

-      MAPPING, FOLDING, FILTERING
foldFM          :: (key -> elt -> a -> a) -> a -> FiniteMap key elt -> a
mapFM           :: (key -> elt1 -> elt2) -> FiniteMap key elt1 -
> FiniteMap key elt2
filterFM        :: Ord key => (key -> elt -> Bool)
                           -> FiniteMap key elt -> FiniteMap key elt

-      INTERROGATING
sizeFM          :: FiniteMap key elt -> Int
isEmptyFM       :: FiniteMap key elt -> Bool

elemFM          :: Ord key => key -> FiniteMap key elt -> Bool
lookupFM        :: Ord key => FiniteMap key elt -> key -> Maybe elt
lookupWithDefaultFM
                :: Ord key => FiniteMap key elt -> elt -> key -> elt
                - lookupWithDefaultFM supplies a "default" elt
                - to return for an unmapped key

-      LISTIFYING
fmToList        :: FiniteMap key elt -> [(key,elt)]
keysFM          :: FiniteMap key elt -> [key]
eltsFM          :: FiniteMap key elt -> [elt]
```

2.2. The `Set` type

Our implementation of *sets* (key property: no duplicates) is just a variant of the `FiniteMap` module.

```
data Set        -- abstract
                -- instance of: Eq

emptySet        :: Set a
mkSet           :: Ord a => [a]  -> Set a
setToList       :: Set a -> [a]
unitSet         :: a -> Set a
singletonSet    :: a -> Set a  -- deprecated, use unitSet.

union           :: Ord a => Set a -> Set a -> Set a
unionManySets   :: Ord a => [Set a] -> Set a
minusSet        :: Ord a => Set a -> Set a -> Set a
mapSet          :: Ord a => (b -> a) -> Set b -> Set a
intersect       :: Ord a => Set a -> Set a -> Set a

elementOf       :: Ord a => a -> Set a -> Bool
isEmptySet      :: Set a -> Bool

cardinality     :: Set a -> Int
```

Chapter 3. The `lang` category: language support

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

3.1. 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.)

3.2. `Addr`

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

```
module Addr where

data Addr  – Address type
instance Eq Addr
instance Ord Addr
instance Show Addr
```

```
addrToInt              :: Addr -> Int
intToAddr              :: Int  -> 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
indexStablePtrOffAddr :: Addr -> Int -> StablePtr a

- 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
readStablePtrOffAddr   :: Addr -> Int -> IO (StablePtr a)

- write value into mutable memory
writeCharOffAddr       :: Addr -> Int -> Char   -> IO ()
writeIntOffAddr        :: Addr -> Int -> Int    -> IO ()
writeAddrOffAddr       :: Addr -> Int -> Addr   -> 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 ()
writeInt8OffAddr       :: Addr -> Int -> Int8   -> IO ()
```

```
writeInt16OffAddr     :: Addr -> Int -> Int16  -> IO ()
writeInt32OffAddr     :: Addr -> Int -> Int32  -> IO ()
writeInt64OffAddr     :: Addr -> Int -> Int64  -> IO ()
writeForeignObjOffAddr :: Addr -> Int -> ForeignObj -> IO ()
writeStablePtrOffAddr  :: Addr -> Int -> StablePtr a -> IO ()
```

Hugs provides `Addr` and `nullAddr` but does not provide any of the index, read or write functions. They can be implemented using GreenCard if required.

3.3. Bits

This library defines bitwise operations for signed and unsigned ints. Instances of `class Bits` can be obtained from the `Int` (Section 3.11) and `Word` (Section 3.19) modules.

```
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.
- `shift x i` and `rotate x i` shift to the left if `i` is positive and to the right otherwise.
- `bit i` is the value with the i'th bit set.

3.4. ByteArray

`ByteArray`s are chunks of immutable Haskell heap:

```
data ByteArray ix - abstract

instance Eq (ByteArray ix)

newByteArray        :: Ix ix => (ix,ix) -> ST s (ByteArray ix)

indexCharArray      :: Ix ix => ByteArray ix -> ix -> Char
indexIntArray       :: Ix ix => ByteArray ix -> ix -> Int
indexWordArray      :: Ix ix => ByteArray ix -> ix -> Word
indexAddrArray      :: Ix ix => ByteArray ix -> ix -> Addr
indexFloatArray     :: Ix ix => ByteArray ix -> ix -> Float
indexDoubleArray    :: Ix ix => ByteArray ix -> ix -> Double
indexStablePtrArray :: Ix ix => ByteArray ix -> ix -> (StablePtr a)

sizeofByteArray     :: Ix ix => ByteArray ix -> Int
boundsOfByteArray   :: Ix ix => ByteArray ix -> (ix, ix)
```

Remarks:

- The operation `newByteArray` creates a byte array of length equal to the range of its indices *in bytes*.
- `sizeofByteArray` returns the size of the byte array, *in bytes*.
- Equality on byte arrays is value equality, not pointer equality (as is the case for its mutable variant.) Two byte arrays are equal if they're of the same length and they're pairwise equal.

3.5. The CCall module

This module is deprecated. Use the Foreign Function Interface instead.

The `CCall` module defines the classes `CCallable` and `CReturnable`, along with instances for the primitive types (`Int`, `Int#`, `Float`, `Float#` etc.) GHC knows to import this module if you use `_ccall_`, but if you need to define your own instances of these classes, you will need to import `CCall` explicitly.

More information on how to use `_ccall_` can be found in .

3.6. 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:

```
data Dynamic - abstract
instance Show Dynamic

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).

3.6.1. Representing types

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

```
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, `TypeRep`s 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 `TypeRep`s to be implemented efficiently, the *abstract* `TyCon` type is used, with the constructor function `mkTyCon` provided:

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

An implementation of the `Dynamic` interface guarantees the following,

```
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.

3.6.2. The Typeable class

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

```
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
```

3.6.3. Utility functions

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

```
dynApply   :: Dynamic -> Dynamic -> Dynamic - unsafe.
dynApplyMb :: Dynamic -> Dynamic -> Maybe Dynamic
```

3.7. 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:

```
data Exception
  = IOException  IOError - IO exceptions (from 'fail')
  | ArithException   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 AsyncException - Externally generated errors
  | NonTermination         - Infinite Loop

instance Eq   Exception
instance Ord  Exception
instance Show Exception

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

instance Eq   ArithError
instance Ord  ArithError
instance Show ArithError

data AsyncException
  = StackOverflow
  | HeapOverflow
  | ThreadKilled

instance Eq   AsyncException
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:

```
throw :: Exception -> a
```

3.7.1. The `try` functions

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

```
tryAll    :: a    -> IO (Either Exception a)
tryAllIO  :: IO a -> IO (Either Exception a)
try   :: (Exception -> Maybe b) -> a    -> IO (Either b a)
tryIO   :: (Exception -> Maybe b) -> IO a -> IO (Either b a)
```

The simplest version is `tryAll`. It takes a single argument, evaluates it (as if you'd applied `seq` to it), and returns either `Right a` if the evaluation succeeded with result `a`, or `Left e` if an exception was raised, where `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 `error "urk" + 1 'div' 0`. Does `tryAll` return `Just (ErrorCall "urk")` or `Just (ArithError DivideByZero)`? The answer is "either": `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 `tryAll` is an IO computation.

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

The functions `try` and `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:

```
justIoErrors :: Exception -> Maybe IOError
justArithExceptions  :: Exception -> Maybe ArithException
justErrors :: Exception -> Maybe String
justDynExceptions :: Exception -> Maybe Dynamic
justAssertions :: Exception -> Maybe String
justAsyncExceptions  :: Exception -> Maybe AsyncException
```

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

```
    result <- 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 `try` or `catch`.

3.7.2. The `catch` functions

The `catch` family is similar to the `try` family:

```
catchAll   :: a    -> (Exception -> IO a) -> IO a
catchAllIO :: IO a -> (Exception -> IO a) -> IO a
catch      :: (Exception -> Maybe b) -> a    -> (b -> IO a) -> IO a
catchIO    :: (Exception -> Maybe b) -> IO a -> (b -> IO a) -> IO a
```

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.

`catch` and `catchIO` take exception predicate arguments in the same way as `try` and `tryIO`.

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

Also, don't forget to `import Prelude hiding (catch)` when using this library, to avoid the name clash between `Exception.catch` and `IO.catch`.

3.7.3. Dynamic Exceptions

Because the `Exception` datatype isn't extensible, we added an interface for throwing and catching exceptions of type `Dynamic` (see Section 3.6), which allows exception values of any type in the `Typeable` class to be thrown and caught.

```
throwDyn :: Typeable exception => exception -> b
catchDyn :: Typeable exception => IO a -> (exception -> IO a) -> IO a
```

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

3.7.4. Other Utilities

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

```
bracket   :: IO a -> (a -> IO b) -> (a -> IO c) -> IO c
bracket_  :: IO a -> IO b -> IO c -> IO c
finally :: IO a -> IO b -> IO b
```

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

```
process_file = bracket (openFile "filename") closeFile
               (do
         ...
                )
```

`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.

3.8. 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.

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

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

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

```
indexCharOffForeignObj   :: ForeignObj -> Int -> Char
indexIntOffForeignObj    :: ForeignObj -> Int -> Int
indexWordOffForeignObj   :: ForeignObj -> Int -> Word
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
readWordOffForeignObj    :: ForeignObj -> Int -> IO Word
readAddrOffForeignObj    :: ForeignObj -> Int -> IO Addr
```

```
readFloatOffForeignObj   :: ForeignObj -> Int -> IO Float
readDoubleOffForeignObj  :: ForeignObj -> Int -> IO Double
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 ()
writeWordOffForeignObj   :: ForeignObj -> Int -> Word   -> 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 ()
```

3.9. The `GlaExts` interface

The `GlaExts` interface provides access to extensions that only GHC implements. These currently are: unboxed types, including the representations of the primitive types (Int, Float, etc.), and the GHC primitive operations (`+#`, `==#`, etc.).

This module used to provide access to all the Glasgow extensions, but these have since been moved into separate libraries for compatibility with Hugs (version 2.09: in fact, you can still get at this stuff via `GlaExts` for compatibility, but this facility will likely be removed in the future).

```
- the representation of some basic types:
data Char    = C# Char#
data Int     = I# Int#
data Addr    = A# Addr#
data Word    = W# Word#
data Float   = F# Float#
data Double  = D# Double#
data Integer = S# Int#              - small integers
             | J# Int# ByteArray#   - large integers
```

```
module GHC  - all primops and primitive types.
```

3.10. IOExts

(Documetation for this library is sadly inadequate.)

This library provides the following extensions to the IO monad:

- The operations `fixIO`, `unsafePerformIO` and `unsafeInterleaveIO` described in [ImperativeFP].

  Everyone knows that if the I/O computation wrapped in `unsafePerformIO` performs side effects, then the relative order in which those side effects take place (relative to the main I/O trunk, or other calls to `unsafePerformIO`) is indeterminate. It is less well known that `unsafePerformIO` is not type safe. For example:

  ```
  test :: IORef [a]
  test = unsafePerformIO $ newIORef []

  main = do
          writeIORef test [42]
          bang <- readIORef test
          print (bang :: [Char])
  ```

  This program will core dump. This problem with polymorphic references is well known in the ML community, and does not arise with normal monadic use of references. There is no easy way to make it impossible once you use `unsafePerformIO`. Indeed, it is possible to write `coerce :: a -> b` with the help of `unsafePerformIO`. So be careful!

- 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:

  ```
  > 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

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)

data IOArray ix elt - mutable arrays indexed by values of type ix
                    - containing values of type a.
newIOArray          :: Ix ix => (ix,ix) -> elt -> IO (IOArray ix elt)
boundsIOArray       :: Ix ix => IOArray ix elt -> (ix, ix)
readIOArray         :: Ix ix => IOArray ix elt -> ix -> IO elt
writeIOArray        :: Ix ix => IOArray ix elt -> ix -> elt -> IO ()
freezeIOArray       :: Ix ix => IOArray ix elt -> IO (Array ix elt)
instance Eq (IOArray ix elt)

openFileEx          :: FilePath -> IOModeEx -> IO Handle
data IOModeEx = BinaryMode IO.IOMode | TextMode IO.IOMode
instance Eq IOModeEx
instance Read IOModeEx
instance Show IOModeEx

performGC           :: IO ()
trace               :: String -> a -> a
unsafePtrEq         :: a -> a -> Bool
```

3.11. Int

This interface provides a collection of sized, signed integers. The types supported are as follows:

type	number of bits

Int8	8
Int16	16
Int32	32
Int64	64

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

```
int8ToInt16 :: Int8   -> Int16
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
```

```
intToInt8  :: Int   -> Int8
intToInt16 :: Int   -> Int16
intToInt32 :: Int   -> Int32
intToInt64 :: Int   -> Int64

integerToInt8  :: Integer -> Int8
integerToInt16 :: Integer -> Int16
integerToInt32 :: Integer -> Int32
integerToInt64 :: Integer -> Int64

int64ToInteger :: Int64 -> Integer
int32ToInteger :: Int32 -> Integer
int16ToInteger :: Int16 -> Integer
int8ToInteger  :: Int8  -> Integer
```

- 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.
- Right and left shifts by amounts greater than or equal to the width of the type result in either zero or -1, depending on the sign of the value being shifted. This is contrary to the behaviour in C, which is undefined; a common interpretation is to truncate the shift count to the width of the type, for example `1 « 32 == 1` in some C implementations.
- Hugs does not provide `Int64` at the moment.

The `Int` module also exports the overloaded operations for converting to and from Haskell `Int`s:

```
toInt   :: (Integral a) => a -> Int
fromInt :: (Num a) => Int -> a
```

Portability note: both Hugs98 and all releases of GHC prior to ghc-4.05 also exports these two via the Prelude. So, to have code that uses `toInt` and `fromInt` be maximally portable, make sure you add an import on `Int` (even if the version of Hugs or GHC you're currently using may not export these two from there.)

3.12. 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:

```
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.

3.13. MutableArray

The `MutableArray` interface provide operations for reading and writing values to mutable arrays. There's two kinds of mutable arrays, the mutatable version of Haskell `Array`s and *mutable byte arrays*, chunks of memory containing values of some basic type.

3.13.1. Mutable arrays

The mutable array section of the API provides the following operations:

```
- mutable arrays:
newArray      :: Ix ix -> (ix,ix) -> elt -> ST s (MutableArray s ix elt)
boundsOfArray :: Ix ix => MutableArray s ix elt -> (ix, ix)
readArray     :: Ix ix => MutableArray s ix elt -> ix -> ST s elt
writeArray    :: Ix ix => MutableArray s ix elt -> ix -> elt -> ST s ()
freezeArray   :: Ix ix => MutableArray s ix elt -> ST s (Array ix elt)
thawArray     :: Ix ix => Array ix elt -> ST s (MutableArray s ix elt)

unsafeFreezeArray   :: Ix ix => MutableArray s ix elt -
> ST s (Array ix elt)
unsafeThawArray     :: Ix ix => Array ix elt -
> ST s (MutableArray s ix elt)
```

Remarks:

- The `freezeArray` action converts a mutable array into an immutable one by copying, whereas `unsafeFreezeArray` returns an immutable array that is effectively just the type cast version of the mutable array. Should you write to the mutable array after it has been (unsafely) frozen, you'll side-effect the immutable array in the process. Please don't :-)
- The operation `thawArray` goes the other way, converting an immutable `Array` into a mutable one. This is done by copying. The operation `unsafeThawArray` is also provided, which places the same kind of proof obligation on the programmer as `unsafeFreezeArray` does.

3.13.2. Mutable byte arrays

```
- creators:
newCharArray      :: Ix ix => (ix,ix) -> ST s (MutableByteArray s ix)
newAddrArray      :: Ix ix => (ix,ix) -> ST s (MutableByteArray s ix)
newIntArray       :: Ix ix => (ix,ix) -> ST s (MutableByteArray s ix)
newWordArray      :: Ix ix => (ix,ix) -> ST s (MutableByteArray s ix)
newFloatArray     :: Ix ix => (ix,ix) -> ST s (MutableByteArray s ix)
newDoubleArray    :: Ix ix => (ix,ix) -> ST s (MutableByteArray s ix)
newStablePtrArray :: Ix ix => (ix,ix) -> ST s (MutableByteArray s ix)

boundsOfMutableByteArray
                   :: Ix ix => MutableByteArray s ix -> (ix, ix)

readCharArray      :: Ix ix => MutableByteArray s ix -> ix -> ST s Char
readIntArray       :: Ix ix => MutableByteArray s ix -> ix -> ST s Int
readAddrArray      :: Ix ix => MutableByteArray s ix -> ix -> ST s Addr
readFloatArray     :: Ix ix => MutableByteArray s ix -> ix -> ST s Float
readDoubleArray    :: Ix ix => MutableByteArray s ix -> ix -> ST s Double
readStablePtrArray :: Ix ix => MutableByteArray s ix -> ix -
> ST s (StablePtr a)
readWord8Array    :: Ix ix => MutableByteArray s ix -> ix -> ST s Word8
readWord16Array    :: Ix ix => MutableByteArray s ix -> ix -> ST s Word16
readWord32Array    :: Ix ix => MutableByteArray s ix -> ix -> ST s Word32
readWord64Array    :: Ix ix => MutableByteArray s ix -> ix -> ST s Word64
readInt8Array    :: Ix ix => MutableByteArray s ix -> ix -> ST s Int8
readInt16Array    :: Ix ix => MutableByteArray s ix -> ix -> ST s Int16
readInt32Array    :: Ix ix => MutableByteArray s ix -> ix -> ST s Int32
readInt64Array    :: Ix ix => MutableByteArray s ix -> ix -> ST s Int64

writeCharArray        :: Ix ix => MutableByteArray s ix -> ix -> Char -
> ST s ()
writeIntArray         :: Ix ix => MutableByteArray s ix -> ix -> Int  -
> ST s ()
writeAddrArray        :: Ix ix => MutableByteArray s ix -> ix -> Addr -
> ST s ()
writeFloatArray       :: Ix ix => MutableByteArray s ix -> ix -> Float -
> ST s ()
writeDoubleArray      :: Ix ix => MutableByteArray s ix -> ix -> Double -
> ST s ()
writeStablePtrArray   :: Ix ix => MutableByteArray s ix -> ix -
> StablePtr a -> ST s ()
writeWord8Array
:: Ix ix => MutableByteArray s ix -> ix -> Word8  -> ST s ()
writeWord16Array      :: Ix ix => MutableByteArray s ix -> ix -> Word16 -
> ST s ()
writeWord32Array      :: Ix ix => MutableByteArray s ix -> ix -> Word32 -
> ST s ()
```

```
writeWord64Array      :: Ix ix => MutableByteArray s ix -> ix -> Word64 -
> ST s ()
writeInt8Array
:: Ix ix => MutableByteArray s ix -> ix -> Int8  -> ST s ()
writeInt16Array       :: Ix ix => MutableByteArray s ix -> ix -> Int16 -
> ST s ()
writeInt32Array
:: Ix ix => MutableByteArray s ix -> ix -> Int32 -> ST s ()
writeInt64Array
:: Ix ix => MutableByteArray s ix -> ix -> Int64 -> ST s ()

freezeCharArray       :: Ix ix => MutableByteArray s ix -
> ST s (ByteArray ix)
freezeIntArray        :: Ix ix => MutableByteArray s ix -
> ST s (ByteArray ix)
freezeAddrArray       :: Ix ix => MutableByteArray s ix -
> ST s (ByteArray ix)
freezeFloatArray      :: Ix ix => MutableByteArray s ix -
> ST s (ByteArray ix)
freezeDoubleArray     :: Ix ix => MutableByteArray s ix -
> ST s (ByteArray ix)
freezeStablePtrArray  :: Ix ix => MutableByteArray s ix -
> ST s (ByteArray ix)

unsafeFreezeByteArray  :: Ix ix => MutableByteArray s ix -
> ST s (ByteArray ix)

sizeofMutableByteArray :: Ix ix => MutableByteArray s ix -> Int

thawByteArray       :: Ix ix => ByteArray ixt -
> ST s (MutableByteArray s ix)
unsafeThawByteArray :: Ix ix => ByteArray ixt -
> ST s (MutableByteArray s ix)
```

Remarks:

- A Mutable byte array is created by specifying its size in units of some basic type. For example,

```
mkPair :: ST s (MutableByteArray s Int)
mkPair = newIntArray (0,1)
```

creates a mutable array capable of storing two `Int`s. Notice that the range size *is not in bytes*, but in units of the basic type.

- A mutable byte array is not parameterised over the kind of values it contains. A consequence of this is that it is possible to have byte arrays containing a mix of basic types, or even read a value from the array at a different type from which it was written, e.g.,

```
isLitteEndian :: IO Bool
isLitteEndian = stToIO $ do
```

```
x <- newIntArray (0,1)
writeIntArray x 1
v <- readCharArray x 0
return (v == chr 1)
```

It's left as an exercise for the reader to determine whether having byte arrays not be parameterised over the type of values they contain is a bug or a feature..

- As for mutable arrays, operations for turning mutable byte arrays into immutable byte arrays are also provided by the `freeze*` class of actions. There's also the non-copying `unsafeFreezeByteArray`.
- Operations for going the other way, where an immutable byte array is 'thawed' are also provided. `thawByteArray` does this by copying, whereas `unsafeThawByteArray` does not
- The operation `sizeofMutableByteArray` returns the size of the array, *in bytes.*

3.14. NumExts

The `NumExts` interface collect together various numeric operations that have proven to be commonly useful

```
 - Going between Doubles and Floats:
doubleToFloat :: Double -> Float
floatToDouble :: Float  -> Double

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

showListWith  :: (a -> ShowS) -> [a] -> 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

  ```
  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))
  ```

- `showListWith` is strictly speaking not a 'NumExts' kind of function, but it's sometimes useful in conjunction with the other `show*` functions that `NumExts` exports. It is the non-overloaded version of `showList`, allowing you to supply the `shows` function to use per list element. For instance,

  ```
  putStrLn (NumExts.showListWith NumExts.showHex [0..16])
  ```

will print out the elements of `[1..16]` in hexadecimal form.

3.15. The PackedString type

You need to `import PackedString` and heave in your `-syslib ghc` to use `PackedString`s.

The basic type and functions available are:

```
data PackedString - abstract

packString          :: [Char] -> PackedString
packStringST        :: [Char] -> ST s PackedString
packCBytesST        :: Int -> Addr -> ST s PackedString
packBytesForCST     :: [Char] -> ST s (ByteArray Int)
byteArrayToPS       :: ByteArray Int -> PackedString
unsafeByteArrayToPS :: ByteArray a   -> Int -> PackedString
psToByteArray       :: PackedString -> ByteArray Int
psToByteArrayST     :: PackedString -> ST s (ByteArray Int)

unpackPS        :: PackedString -> [Char]
```

We also provide a wad of list-manipulation-like functions:

```
nilPS       :: PackedString
consPS      :: Char -> PackedString -> PackedString
```

```
headPS      :: PackedString -> Char
tailPS      :: PackedString -> PackedString
nullPS      :: PackedString -> Bool
appendPS    :: PackedString -> PackedString -> PackedString
lengthPS    :: PackedString -> Int
indexPS     :: PackedString -> Int -> Char
            - 0-origin indexing into the string
mapPS       :: (Char -> Char) -> PackedString -> PackedString
filterPS    :: (Char -> Bool) -> PackedString -> PackedString
foldlPS     :: (a -> Char -> a) -> a -> PackedString -> a
foldrPS     :: (Char -> a -> a) -> a -> PackedString -> a
takePS      :: Int -> PackedString -> PackedString
dropPS      :: Int -> PackedString -> PackedString
splitAtPS   :: Int -> PackedString -> (PackedString, PackedString)
takeWhilePS :: (Char -> Bool) -> PackedString -> PackedString
dropWhilePS :: (Char -> Bool) -> PackedString -> PackedString
spanPS      :: (Char -> Bool) -> PackedString -
> (PackedString, PackedString)
breakPS     :: (Char -> Bool) -> PackedString -
> (PackedString, PackedString)
linesPS     :: PackedString -> [PackedString]
wordsPS     :: PackedString -> [PackedString]
reversePS   :: PackedString -> PackedString
concatPS    :: [PackedString] -> PackedString
elemPS      :: Char -> PackedString -> Bool
  - Perl-style split&join
splitPS     :: Char -> PackedString -> [PackedString]
splitWithPS :: (Char -> Bool) -> PackedString -> [PackedString]
joinPS      :: PackedString -> [PackedString] -> PackedString

substrPS   :: PackedString -> Int -> Int -> PackedString
           - pluck out a piece of a PackedString
           - start and end chars you want; both 0-origin-specified
```

3.16. ST

This library provides support for *strict* state threads, as described in the PLDI '94 paper by John Launchbury and Simon Peyton Jones [LazyStateThreads]. 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 -> 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.

3.17. Stable

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

3.17.1. 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:

```
data StablePtr a  – abstract, instance of: Eq.
makeStablePtr  :: a -> IO (StablePtr a)
deRefStablePtr :: 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:

```
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.

3.17.2. 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:

```
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.

3.18. ShowFunctions

The `ShowFunctions` module contains the following instance of `Show` for function types

```
instance Show (a -> b) where
showsPrec _ _ = showString "<function>"
```

3.19. Word

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

type	number of bits
Word8	8
Word16	16
Word32	32
Word64	64

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.

```
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

```
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.

- Right and left shifts by amounts greater than or equal to the width of the type result in a zero result. This is contrary to the behaviour in C, which is undefined; a common interpretation is to truncate the shift count to the width of the type, for example `1 « 32 == 1` in some C implementations.

Implementation notes:

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

Chapter 4. The `net` category: networking support

4.1. Network-interface toolkit—`Socket` and `SocketPrim`

(Darren Moffat supplied the initial version of this library.)

Your best bet for documentation is to look at the code—really!— normally in `fptools/hslibs/net/{BSD,Socket,SocketPrim.lhs}`.

The `BSD` module provides functions to get at system-database info; pretty straightforward if you're into this sort of thing:

```
getHostName         :: IO String

getServiceByName    :: ServiceName -> IO ServiceEntry
getServicePortNumber:: ServiceName -> IO PortNumber
getServiceEntry     :: IO ServiceEntry
setServiceEntry     :: Bool -> IO ()
endServiceEntry     :: IO ()

getProtocolByName   :: ProtocolName -> IO ProtocolEntry
getProtocolByNumber :: ProtocolNumber -> IO ProtcolEntry
getProtocolNumber   :: ProtocolName -> ProtocolNumber
getProtocolEntry    :: IO ProtocolEntry
setProtocolEntry    :: Bool -> IO ()
endProtocolEntry    :: IO ()

getHostByName       :: HostName -> IO HostEntry
getHostByAddr       :: Family -> HostAddress -> IO HostEntry
getHostEntry        :: IO HostEntry
setHostEntry        :: Bool -> IO ()
endHostEntry        :: IO ()
```

The `SocketPrim` interface provides quite direct access to the socket facilities in a BSD Unix system, including all the complications. We hope you don't need to use it! See the source if needed...

The `Socket` interface is a "higher-level" interface to sockets, and it is what we recommend. Please tell us if the facilities it offers are inadequate to your task!

The interface is relatively modest:

```
connectTo       :: Hostname -> PortID -> IO Handle
listenOn        :: PortID -> IO Socket

accept          :: Socket -> IO (Handle, HostName)
```

```
sendTo          :: Hostname -> PortID -> String -> IO ()

recvFrom        :: Hostname -> PortID -> IO String
socketPort      :: Socket -> IO PortID

data PortID     - PortID is a non-abstract type
  = Service String        - Service Name eg "ftp"
  | PortNumber PortNumber - User defined Port Number
  | UnixSocket String     - Unix family socket in file system

type Hostname = String

 - 16-bit value (stored in network byte order).
data PortNumber
 - instance of: Eq, Num, Show.

mkPortNumber :: Int -> PortNumber
```

4.2. URI

The URI library provides utilities for parsing and manipulating Uniform Resource Identifiers (a more general form of Uniform Resource Locators, or URLs). URIs are described in RFC 2396 (http://www.faqs.org/rfcs/rfc2396.html).

```
module URI where

data URI {
  scheme, :: String,
  authority, :: String,
  path, :: String,
  query, :: String,
  fragment :: String
  }
instance Show URI

parseURI        :: String -> Maybe URI

relativeTo      :: URI -> URI -> URI

- support for putting strings into URI-friendly
- escaped format and getting them back again.
- Can't be done transparently, because certain characters
- have different meanings in different kinds of URI.

reserved  :: Char -> Bool
unreserved  :: Char -> Bool
```

```
isAllowedInURI :: Char -> Bool
escapeString    :: String -> (Char->Bool) -> String
unEscapeString :: String -> String
```

Chapter 5. The `num` category: numeric operations

This category is currently empty.

Chapter 6. The `posix` category: POSIX support

6.1. The Posix library

The `Posix` interface gives you access to the set of OS services standardised by POSIX 1003.1b (or the *IEEE Portable Operating System Interface for Computing Environments* - IEEE Std. 1003.1). The interface is accessed by `import Posix` and adding `-syslib posix` on your command-line.

6.1.1. Posix data types

```
data ByteCount  -- instances of : Eq Ord Num Real Integral Ix Enum Show
```

A `ByteCount` is a primitive of type `unsigned`. At a minimum, an conforming implementation must support values in the range `[0, UINT_MAX]`.

```
data ClockTick  -- instances of : Eq Ord Num Real Integral Ix Enum Show
```

A `ClockTick` is a primitive of type `clock_t`, which is used to measure intervals of time in fractions of a second. The resolution is determined by `getSysVar ClockTick`.

```
data DeviceID  -- instances of : Eq Ord Num Real Integral Ix Enum Show
```

A `DeviceID` is a primitive of type `dev_t`. It must be an arithmetic type.

```
data EpochTime -- instances of : Eq Ord Num Real Integral Ix Enum Show
```

A `EpochTime` is a primitive of type `time_t`, which is used to measure seconds since the Epoch. At a minimum, the implementation must support values in the range `[0, INT_MAX]`.

```
data FileID -- instances of : Eq Ord Num Real Integral Ix Enum Show
```

A `FileID` is a primitive of type `ino_t`. It must be an arithmetic type.

```
data FileMode -- instances of : Eq Ord Num Real Integral Ix Enum Show
```

A `FileMode` is a primitive of type `mode_t`. It must be an arithmetic type.

```
data FileOffset - instances of : Eq Ord Num Real Integral Ix Enum Show
```

A `FileOffset` is a primitive of type `off_t`. It must be an arithmetic type.

```
data GroupID - instances of : Eq Ord Num Real Integral Ix Enum Show
```

A `GroupID` is a primitive of type `gid_t`. It must be an arithmetic type.

```
data Limit - instances of : Eq Ord Num Real Integral Ix Enum Show
```

A `Limit` is a primitive of type `long`. At a minimum, the implementation must support values in the range `[LONG_MIN, LONG_MAX]`.

```
data LinkCount - instances of : Eq Ord Num Real Integral Ix Enum Show
```

A `LinkCount` is a primitive of type `nlink_t`. It must be an arithmetic type.

```
data ProcessID - instances of : Eq Ord Num Real Integral Ix Enum Show
type ProcessGroupID = ProcessID
```

A `ProcessID` is a primitive of type `pid_t`. It must be a signed arithmetic type.

```
data UserID - instances of : Eq Ord Num Real Integral Ix Enum Show
```

A `UserID` is a primitive of type `uid_t`. It must be an arithmetic type.

```
data DirStream
```

A `DirStream` is a primitive of type `DIR *`.

```
data FileStatus
```

A `FileStatus` is a primitive of type `struct stat`.

```
data GroupEntry
```

A `GroupEntry` is a primitive of type `struct group`.

```
data ProcessTimes
```

`ProcessTimes` is a primitive structure containing a `clock_t` and a `struct tms`.

```
data SignalSet
```

An `SignalSet` is a primitive of type `sigset_t`.

```
data SystemID
```

A `SystemID` is a primitive of type `struct utsname`.

```
data TerminalAttributes
```

`TerminalAttributes` is a primitive of type `struct termios`.

```
data UserEntry
```

A `UserEntry` is a primitive of type `struct passwd`.

```
data BaudRate = B0 | B50 | B75 | B110 | B134 | B150 | B200 | B300 | B600
              | B1200 | B1800 | B2400 | B4800 | B9600 | B19200 | B38400
              deriving (Eq, Show)

data Fd

intToFd :: Int -> Fd - use with care.

data FdOption = AppendOnWrite
              | CloseOnExec
              | NonBlockingRead

data ControlCharacter = EndOfFile
                      | EndOfLine
                      | Erase
                      | Interrupt
                      | Kill
                      | Quit
                      | Suspend
                      | Start
                      | Stop

type ErrorCode = Int

type FileLock = (LockRequest, SeekMode, FileOffset, FileOffset)
-                            whence    start       length

data FlowAction = SuspendOutput | RestartOutput | Transmit-
Stop | TransmitStart

data Handler = Default | Ignore | Catch (IO ())

data LockRequest = ReadLock | WriteLock | Unlock
```

```
                 deriving (Eq, Show)

data OpenMode = ReadOnly | WriteOnly | ReadWrite

data PathVar = LinkLimit
             | InputLineLimit
             | InputQueueLimit
             | FileNameLimit
             | PathNameLimit
             | PipeBufferLimit
             | SetOwnerAndGroupIsRestricted
             | FileNamesAreNotTruncated

data QueueSelector = InputQueue | OutputQueue | BothQueues

type Signal = Int

data SysVar = ArgumentLimit
            | ChildLimit
            | ClockTick
            | GroupLimit
            | OpenFileLimit
            | PosixVersion
            | HasSavedIDs
            | HasJobControl

data TerminalMode = InterruptOnBreak       - BRKINT
                  | MapCRtoLF              - ICRNL
                  | IgnoreBreak            - IGNBRK
                  | IgnoreCR               - IGNCR
                  | IgnoreParityErrors     - IGNPAR
                  | MapLFtoCR              - INLCR
                  | CheckParity            - INPCK
                  | StripHighBit           - ISTRIP
                  | StartStopInput         - IXOFF
                  | StartStopOutput        - IXON
                  | MarkParityErrors       - PARMRK
                  | ProcessOutput          - OPOST
                  | LocalMode              - CLOCAL
                  | ReadEnable             - CREAD
                  | TwoStopBits            - CSTOPB
                  | HangupOnClose          - HUPCL
                  | EnableParity           - PARENB
                  | OddParity              - PARODD
                  | EnableEcho             - ECHO
                  | EchoErase              - ECHOE
                  | EchoKill               - ECHOK
                  | EchoLF                 - ECHONL
                  | ProcessInput           - ICANON
```

```
                 | ExtendedFunctions        - IEXTEN
                 | KeyboardInterrupts       - ISIG
                 | NoFlushOnInterrupt       - NOFLSH
                 | BackgroundWriteInterrupt - TOSTOP

data TerminalState = Immediately | WhenDrained | WhenFlushed

data ProcessStatus = Exited ExitCode
                   | Terminated Signal
                   | Stopped Signal
                   deriving (Eq, Show)
```

6.1.2. Posix Process Primitives

```
forkProcess :: IO (Maybe ProcessID)
```

`forkProcess` calls `fork`, returning `Just pid` to the parent, where `pid` is the ProcessID of the child, and returning `Nothing` to the child.

```
executeFile :: FilePath                     - Command
            -> Bool                         - Search PATH?
            -> [String]                     - Arguments
            -> Maybe [(String, String)]     - Environment
            -> IO ()
```

`executeFile cmd args env` calls one of the `execv*` family, depending on whether or not the current PATH is to be searched for the command, and whether or not an environment is provided to supersede the process's current environment. The basename (leading directory names suppressed) of the command is passed to `execv*` as `arg[0]`; the argument list passed to `executeFile` therefore begins with `arg[1]`.

```
Search PATH?    Supersede environ?      Call
~~~~~~~~~~~~    ~~~~~~~~~~~~~~~~~~      ~~~~~~~
False           False                   execv
False           True                    execve
True            False                   execvp
True            True                    execvpe*
```

Note that `execvpe` is not provided by the POSIX standard, and must be written by hand. Care must be taken to ensure that the search path is extracted from the original environment, and not from the environment to be passed on to the new image.

NOTE: In general, sharing open files between parent and child processes is potential bug farm, and should be avoided unless you really depend on this 'feature' of POSIX' `fork()` semantics. Using

Haskell, there's the extra complication that arguments to `executeFile` might come from files that are read lazily (using `hGetContents`, or some such.) If this is the case, then for your own sanity, please ensure that the arguments to `executeFile` have been fully evaluated before calling `forkProcess` (followed by `executeFile`.) Consider yourself warned :-)

A successful `executeFile` overlays the current process image with a new one, so it only returns on failure.

```
runProcess :: FilePath                    - Command
           -> [String]                    - Arguments
           -> Maybe [(String, String)]    - Environment (Nothing -
> Inherited)
           -> Maybe FilePath              - Working directory (Nothing -
> inherited)
           -> Maybe Handle                - stdin  (Nothing -> inherited)
           -> Maybe Handle                - stdout (Nothing -> inherited)
           -> Maybe Handle                - stderr (Nothing -> inherited)
           -> IO ()
```

`runProcess` is our candidate for the high-level OS-independent primitive.

`runProcess cmd args env wd inhdl outhdl errhdl` runs **cmd** (searching the current `PATH`) with arguments `args`. If `env` is `Just pairs`, the command is executed with the environment specified by `pairs` of variables and values; otherwise, the command is executed with the current environment. If `wd` is `Just dir`, the command is executed with working directory `dir`; otherwise, the command is executed in the current working directory. If `{in,out,err`hdl} is `Just handle`, the command is executed with the `Fd` for `std{in,out,err`} attached to the specified `handle`; otherwise, the `Fd` for `std{in,out,err`} is left unchanged.

```
getProcessStatus :: Bool              - Block?
                 -> Bool              - Stopped processes?
                 -> ProcessID
                 -> IO (Maybe ProcessStatus)
```

`getProcessStatus blk stopped pid` calls `waitpid`, returning `Just tc`, the `ProcessStatus` for process `pid` if it is available, `Nothing` otherwise. If `blk` is `False`, then `WNOHANG` is set in the options for `waitpid`, otherwise not. If `stopped` is `True`, then `WUNTRACED` is set in the options for `waitpid`, otherwise not.

```
getGroupProcessStatus :: Bool         - Block?
                      -> Bool         - Stopped processes?
                      -> ProcessGroupID
                      -> IO (Maybe (ProcessID, ProcessStatus))
```

`getGroupProcessStatus blk stopped pgid` calls `waitpid`, returning `Just (pid, tc)`, the `ProcessID` and `ProcessStatus` for any process in group `pgid` if one is available, `Nothing` otherwise. If `blk` is `False`, then `WNOHANG` is set in the options for `waitpid`, otherwise not. If `stopped` is `True`, then `WUNTRACED` is set in the options for `waitpid`, otherwise not.

```
getAnyProcessStatus :: Bool           - Block?
                    -> Bool           - Stopped processes?
                    -> IO (Maybe (ProcessID, ProcessStatus))
```

`getAnyProcessStatus blk stopped` calls `waitpid`, returning `Just (pid, tc)`, the `ProcessID` and `ProcessStatus` for any child process if one is available, `Nothing` otherwise. If `blk` is `False`, then `WNOHANG` is set in the options for `waitpid`, otherwise not. If `stopped` is `True`, then `WUNTRACED` is set in the options for `waitpid`, otherwise not.

```
exitImmediately :: ExitCode -> IO ()
```

`exitImmediately status` calls `_exit` to terminate the process with the indicated exit `status`. The operation never returns.

```
getEnvironment :: IO [(String, String)]
```

`getEnvironment` parses the environment variable mapping provided by `environ`, returning `(variable, value)` pairs. The operation never fails.

```
setEnvironment :: [(String, String)] -> IO ()
```

`setEnvironment` replaces the process environment with the provided mapping of `(variable, value)` pairs.

```
getEnvVar :: String -> IO String
```

`getEnvVar var` returns the value associated with variable `var` in the current environment (identical functionality provided through standard Haskell library function `System.getEnv`).

The operation may fail with:

`NoSuchThing`

The variable has no mapping in the current environment.

```
setEnvVar :: String -> String -> IO ()
```

`setEnvVar var val` sets the value associated with variable `var` in the current environment to be `val`. Any previous mapping is superseded.

```
removeEnvVar :: String -> IO ()
```

`removeEnvVar var` removes any value associated with variable `var` in the current environment. Deleting a variable for which there is no mapping does not generate an error.

```
nullSignal :: Signal
nullSignal = 0

backgroundRead, sigTTIN         :: Signal
backgroundWrite, sigTTOU        :: Signal
continueProcess, sigCONT        :: Signal
floatingPointException, sigFPE  :: Signal
illegalInstruction, sigILL      :: Signal
internalAbort, sigABRT          :: Signal
keyboardSignal, sigINT          :: Signal
keyboardStop, sigTSTP           :: Signal
keyboardTermination, sigQUIT    :: Signal
killProcess, sigKILL            :: Signal
lostConnection, sigHUP          :: Signal
openEndedPipe, sigPIPE          :: Signal
processStatusChanged, sigCHLD   :: Signal
realTimeAlarm, sigALRM          :: Signal
segmentationViolation, sigSEGV  :: Signal
softwareStop, sigSTOP           :: Signal
softwareTermination, sigTERM    :: Signal
userDefinedSignal1, sigUSR1     :: Signal
userDefinedSignal2, sigUSR2     :: Signal

signalProcess :: Signal -> ProcessID -> IO ()
```

`signalProcess int pid` calls `kill` to signal process `pid` with interrupt signal `int`.

```
raiseSignal :: Signal -> IO ()
```

`raiseSignal int` calls `kill` to signal the current process with interrupt signal `int`.

```
signalProcessGroup :: Signal -> ProcessGroupID -> IO ()
```

`signalProcessGroup int pgid` calls `kill` to signal all processes in group `pgid` with interrupt signal `int`.

```
setStoppedChildFlag :: Bool -> IO Bool
```

`setStoppedChildFlag bool` sets a flag which controls whether or not the `NOCLDSTOP` option will be used the next time a signal handler is installed for `SIGCHLD`. If `bool` is `True` (the default), `NOCLDSTOP` will not be used; otherwise it will be. The operation never fails.

```
queryStoppedChildFlag :: IO Bool
```

`queryStoppedChildFlag` queries the flag which controls whether or not the `NOCLDSTOP` option will be used the next time a signal handler is installed for `SIGCHLD`. If `NOCLDSTOP` will be used, it returns `False`; otherwise (the default) it returns `True`. The operation never fails.

```
emptySignalSet :: SignalSet
fullSignalSet  :: SignalSet
addSignal      :: Signal -> SignalSet -> SignalSet
deleteSignal   :: Signal -> SignalSet -> SignalSet
inSignalSet    :: Signal -> SignalSet -> Bool

installHandler :: Signal
               -> Handler
               -> Maybe SignalSet       - other signals to block
               -> IO Handler            - old handler
```

`installHandler int handler iset` calls `sigaction` to install an interrupt handler for signal `int`. If `handler` is `Default`, `SIG_DFL` is installed; if `handler` is `Ignore`, `SIG_IGN` is installed; if `handler` is `Catch action`, a handler is installed which will invoke `action` in a new thread when (or shortly after) the signal is received. See Section 1.1 for details on how to communicate between threads.

If `iset` is `Just s`, then the `sa_mask` of the `sigaction` structure is set to `s`; otherwise it is cleared. The previously installed signal handler for `int` is returned.

```
getSignalMask :: IO SignalSet
```

`getSignalMask` calls `sigprocmask` to determine the set of interrupts which are currently being blocked.

```
setSignalMask :: SignalSet -> IO SignalSet
```

`setSignalMask mask` calls `sigprocmask` with `SIG_SETMASK` to block all interrupts in `mask`. The previous set of blocked interrupts is returned.

```
blockSignals :: SignalSet -> IO SignalSet
```

`setSignalMask mask` calls `sigprocmask` with `SIG_BLOCK` to add all interrupts in `mask` to the set of blocked interrupts. The previous set of blocked interrupts is returned.

```
unBlockSignals :: SignalSet -> IO SignalSet
```

`setSignalMask mask` calls `sigprocmask` with `SIG_UNBLOCK` to remove all interrupts in `mask` from the set of blocked interrupts. The previous set of blocked interrupts is returned.

```
getPendingSignals :: IO SignalSet
```

`getPendingSignals` calls `sigpending` to obtain the set of interrupts which have been received but are currently blocked.

```
awaitSignal :: Maybe SignalSet -> IO ()
```

`awaitSignal iset` suspends execution until an interrupt is received. If `iset` is `Just s`, `awaitSignal` calls `sigsuspend`, installing `s` as the new signal mask before suspending execution; otherwise, it calls `pause`. `awaitSignal` returns on receipt of a signal. If you have installed any signal handlers with `installHandler`, it may be wise to call `yield` directly after `awaitSignal` to ensure that the signal handler runs as promptly.

```
scheduleAlarm :: Int -> IO Int
```

`scheduleAlarm i` calls `alarm` to schedule a real time alarm at least `i` seconds in the future.

```
sleep :: Int -> IO ()
```

`sleep i` calls `sleep` to suspend execution of the program until at least `i` seconds have elapsed or a signal is received.

6.1.3. Posix Process Environment

```
getProcessID :: IO ProcessID
```

`getProcessID` calls `getpid` to obtain the `ProcessID` for the current process.

```
getParentProcessID :: IO ProcessID
```

`getProcessID` calls `getppid` to obtain the `ProcessID` for the parent of the current process.

```
getRealUserID :: IO UserID
```

`getRealUserID` calls `getuid` to obtain the real `UserID` associated with the current process.

`getEffectiveUserID :: IO UserID`

`getRealUserID` calls `geteuid` to obtain the effective `UserID` associated with the current process.

`setUserID :: UserID -> IO ()`

`setUserID uid` calls `setuid` to set the real, effective, and saved set-user-id associated with the current process to `uid`.

`getLoginName :: IO String`

`getLoginName` calls `getlogin` to obtain the login name associated with the current process.

`getRealGroupID :: IO GroupID`

`getRealGroupID` calls `getgid` to obtain the real `GroupID` associated with the current process.

`getEffectiveGroupID :: IO GroupID`

`getEffectiveGroupID` calls `getegid` to obtain the effective `GroupID` associated with the current process.

`setGroupID :: GroupID -> IO ()`

`setGroupID gid` calls `setgid` to set the real, effective, and saved set-group-id associated with the current process to `gid`.

`getGroups :: IO [GroupID]`

`getGroups` calls `getgroups` to obtain the list of supplementary `GroupID`s associated with the current process.

`getEffectiveUserName :: IO String`

`getEffectiveUserName` calls `cuserid` to obtain a name associated with the effective `UserID` of the process.

`getProcessGroupID :: IO ProcessGroupID`

`getProcessGroupID` calls `getpgrp` to obtain the `ProcessGroupID` for the current process.

`createProcessGroup :: ProcessID -> IO ProcessGroupID` `createProcessGroup pid` calls `setpgid` to make process `pid` a new process group leader.

```
joinProcessGroup :: ProcessGroupID -> IO ProcessGroupID
```

`joinProcessGroup pgid` calls `setpgid` to set the `ProcessGroupID` of the current process to `pgid`.

```
setProcessGroupID :: ProcessID -> ProcessGroupID -> IO ()
```

`setProcessGroupID pid pgid` calls `setpgid` to set the `ProcessGroupID` for process `pid` to `pgid`.

```
createSession :: IO ProcessGroupID
```

`createSession` calls `setsid` to create a new session with the current process as session leader.

```
systemName :: SystemID -> String
nodeName :: SystemID -> String
release :: SystemID -> String
version :: SystemID -> String
machine :: SystemID -> String
```

```
getSystemID :: IO SystemID
```

`getSystemID` calls `uname` to obtain information about the current operating system.

```
> epochTime :: IO EpochTime
```

`epochTime` calls `time` to obtain the number of seconds that have elapsed since the epoch (Jan 01 00:00:00 GMT 1970).

```
elapsedTime     :: ProcessTimes -> ClockTick
userTime        :: ProcessTimes -> ClockTick
systemTime      :: ProcessTimes -> ClockTick
childUserTime   :: ProcessTimes -> ClockTick
childSystemTime :: ProcessTimes -> ClockTick
```

```
getProcessTimes :: IO ProcessTimes
```

`getProcessTimes` calls `times` to obtain time-accounting information for the current process and its children.

```
getControllingTerminalName :: IO FilePath
```

`getControllingTerminalName` calls `ctermid` to obtain a name associated with the controlling terminal for the process. If a controlling terminal exists, `getControllingTerminalName` returns the name of the controlling terminal.

The operation may fail with:

`NoSuchThing`

There is no controlling terminal, or its name cannot be determined.

`SystemError`

Various other causes.

```
getTerminalName :: Fd -> IO FilePath
```

`getTerminalName fd` calls `ttyname` to obtain a name associated with the terminal for `Fd fd`. If `fd` is associated with a terminal, `getTerminalName` returns the name of the terminal.

The operation may fail with:

`InappropriateType`

The channel is not associated with a terminal.

`NoSuchThing`

The channel is associated with a terminal, but it has no name.

`SystemError`

Various other causes.

```
queryTerminal :: Fd -> IO Bool
```

`queryTerminal fd` calls `isatty` to determine whether or not `Fd fd` is associated with a terminal.

```
getSysVar :: SysVar -> IO Limit
```

`getSysVar var` calls `sysconf` to obtain the dynamic value of the requested configurable system limit or option. For defined system limits, `getSysVar` returns the associated value. For defined system options, the result of `getSysVar` is undefined, but not failure.

The operation may fail with:

`NoSuchThing`

The requested system limit or option is undefined.

6.1.4. Posix operations on files and directories

```
openDirStream :: FilePath -> IO DirStream
```

`openDirStream dir` calls `opendir` to obtain a directory stream for `dir`.

```
readDirStream :: DirStream -> IO String
```

`readDirStream dp` calls `readdir` to obtain the next directory entry (`struct dirent`) for the open directory stream `dp`, and returns the `d_name` member of that structure.

The operation may fail with:

`EOF`

End of file has been reached.

`SystemError`

Various other causes.

```
rewindDirStream :: DirStream -> IO ()
```

`rewindDirStream dp` calls `rewinddir` to reposition the directory stream `dp` at the beginning of the directory.

```
closeDirStream :: DirStream -> IO ()
```

`closeDirStream dp` calls `closedir` to close the directory stream `dp`.

```
getWorkingDirectory :: IO FilePath
```

`getWorkingDirectory` calls `getcwd` to obtain the name of the current working directory.

```
changeWorkingDirectory :: FilePath -> IO ()
```

`changeWorkingDirectory dir` calls `chdir` to change the current working directory to `dir`.

```
nullFileMode       :: FileMode       - ------
ownerReadMode      :: FileMode       - r-----
ownerWriteMode     :: FileMode       - -w-----
ownerExecuteMode   :: FileMode       - -x----
groupReadMode      :: FileMode       - --r---
groupWriteMode     :: FileMode       - ---w---
groupExecuteMode   :: FileMode       - ---x--
otherReadMode      :: FileMode       - ----r-
otherWriteMode     :: FileMode       - -----w-
otherExecuteMode   :: FileMode       - -----x
setUserIDMode      :: FileMode       - -S----
setGroupIDMode     :: FileMode       - ---S--

stdFileMode        :: FileMode       - rw-rw-rw-

ownerModes         :: FileMode       - rwx----
groupModes         :: FileMode       - --rwx--
otherModes         :: FileMode       - ----rwx
accessModes        :: FileMode       - rwxrwxrwx

unionFileModes     :: FileMode -> FileMode -> FileMode
intersectFileModes :: FileMode -> FileMode -> FileMode

stdInput  :: Fd
stdInput  = intToFd 0

stdOutput :: Fd
stdOutput = intToFd 1

stdError  :: Fd
stdError  = intToFd 2

data OpenFileFlags =
 OpenFileFlags {
    append    :: Bool,
    exclusive :: Bool,
    noctty    :: Bool,
    nonBlock  :: Bool,
    trunc     :: Bool
 }

openFd :: FilePath
       -> OpenMode
       -> Maybe FileMode  - Just x => O_CREAT, Nothing => must exist
       -> OpenFileFlags
       -> IO Fd
```

`openFd path acc mode (OpenFileFlags app excl noctty nonblock trunc)` calls `open` to obtain a `Fd` for the file `path` with access mode `acc`. If `mode` is `Just m`, the `O_CREAT` flag is set and the file's permissions will be based on `m` if it does not already exist; otherwise, the `O_CREAT` flag is not set. The arguments `app`, `excl`, `noctty`, `nonblock`, and `trunc` control whether or not the flags `O_APPEND`, `O_EXCL`, `O_NOCTTY`, `O_NONBLOCK`, and `O_TRUNC` are set, respectively.

```
createFile :: FilePath -> FileMode -> IO Fd
```

`createFile path mode` calls `creat` to obtain a `Fd` for file `path`, which will be created with permissions based on `mode` if it does not already exist.

```
setFileCreationMask :: FileMode -> IO FileMode
```

`setFileCreationMask mode` calls `umask` to set the process's file creation mask to `mode`. The previous file creation mask is returned.

```
createLink :: FilePath -> FilePath -> IO ()
```

`createLink old new` calls `link` to create a new path, `new`, linked to an existing file, `old`.

```
createDirectory :: FilePath -> FileMode -> IO ()
```

`createDirectory dir mode` calls `mkdir` to create a new directory, `dir`, with permissions based on `mode`.

```
createNamedPipe :: FilePath -> FileMode -> IO ()
```

`createNamedPipe fifo mode` calls `mkfifo` to create a new named pipe, `fifo`, with permissions based on `mode`.

```
removeLink :: FilePath -> IO ()
```

`removeLink path` calls `unlink` to remove the link named `path`.

```
removeDirectory :: FilePath -> IO ()
```

`removeDirectory dir` calls `rmdir` to remove the directory named `dir`.

```
rename :: FilePath -> FilePath -> IO ()
```

`rename old new` calls `rename` to rename a file or directory from `old` to `new`.

```
fileMode          :: FileStatus -> FileMode

fileID            :: FileStatus -> FileID
```

```
deviceID          :: FileStatus -> DeviceID

linkCount         :: FileStatus -> LinkCount

fileOwner         :: FileStatus -> UserID
fileGroup         :: FileStatus -> GroupID
fileSize          :: FileStatus -> FileOffset

accessTime        :: FileStatus -> EpochTime
modificationTime  :: FileStatus -> EpochTime
statusChangeTime  :: FileStatus -> EpochTime

isDirectory       :: FileStatus -> Bool
isCharacterDevice :: FileStatus -> Bool
isBlockDevice     :: FileStatus -> Bool
isRegularFile     :: FileStatus -> Bool
isNamedPipe       :: FileStatus -> Bool

getFileStatus     :: FilePath -> IO FileStatus
```

`getFileStatus path` calls `stat` to get the `FileStatus` information for the file `path`.

```
getFdStatus :: Fd -> IO FileStatus
```

`getFdStatus fd` calls `fstat` to get the `FileStatus` information for the file associated with `Fd` `fd`.

```
queryAccess :: FilePath -> Bool -> Bool -> Bool -> IO Bool
```

`queryAccess path r w x` calls `access` to test the access permissions for file `path`. The three arguments, `r`, `w`, and `x` control whether or not `access` is called with `R_OK`, `W_OK`, and `X_OK` respectively.

```
queryFile :: FilePath -> IO Bool
```

`queryFile path` calls `access` with `F_OK` to test for the existence for file `path`.

```
setFileMode :: FilePath -> FileMode -> IO ()
```

`setFileMode path mode` calls `chmod` to set the permission bits associated with file `path` to `mode`.

```
setOwnerAndGroup :: FilePath -> UserID -> GroupID -> IO ()
```

`setOwnerAndGroup path uid gid` calls `chown` to set the `UserID` and `GroupID` associated with file `path` to `uid` and `gid`, respectively.

```
setFileTimes :: FilePath -> EpochTime -> EpochTime -> IO ()
```

`setFileTimes path atime mtime` calls `utime` to set the access and modification times associated with file `path` to `atime` and `mtime`, respectively.

```
touchFile :: FilePath -> IO ()
```

`touchFile path` calls `utime` to set the access and modification times associated with file `path` to the current time.

```
getPathVar :: PathVar -> FilePath -> IO Limit
```

`getPathVar var path` calls `pathconf` to obtain the dynamic value of the requested configurable file limit or option associated with file or directory `path`. For defined file limits, `getPathVar` returns the associated value. For defined file options, the result of `getPathVar` is undefined, but not failure. The operation may fail with:

`NoSuchThing`

The requested file limit or option is undefined.

`SystemError`

Various other causes.

```
getFdVar :: PathVar -> Fd -> IO Limit
```

`getFdVar var fd` calls `fpathconf` to obtain the dynamic value of the requested configurable file limit or option associated with the file or directory attached to the open channel `fd`. For defined file limits, `getFdVar` returns the associated value. For defined file options, the result of `getFdVar` is undefined, but not failure.

The operation may fail with:

`NoSuchThing`

The requested file limit or option is undefined.

`SystemError`

Various other causes.

6.1.5. Posix Input and Output Primitives

```
createPipe :: IO (Fd, Fd)
```

`createPipe` calls `pipe` to create a pipe and returns a pair of `Fd`s, the first for reading and the second for writing.

```
dup :: Fd -> IO Fd
```

`dup fd` calls `dup` to duplicate `Fd fd` to another `Fd`.

```
dupTo :: Fd -> Fd -> IO ()
```

`dupTo src dst` calls `dup2` to duplicate `Fd src` to `Fd dst`.

```
fdClose :: Fd -> IO ()
```

`fdClose fd` calls `close` to close `Fd fd`.

```
fdRead :: Fd -> ByteCount -> IO (String, ByteCount)
```

`fdRead fd nbytes` calls `read` to read at most `nbytes` bytes from `Fd fd`, and returns the result as a string paired with the number of bytes actually read.

The operation may fail with:

`EOF`
: End of file has been reached.

`SystemError`
: Various other causes.

```
fdWrite :: Fd -> String -> IO ByteCount
```

`fdWrite fd s` calls `write` to write the string `s` to `Fd fd` as a contiguous sequence of bytes. It returns the number of bytes successfully written.

```
queryFdOption :: FdOption -> Fd -> IO Bool
```

getFdOption opt fd calls fcntl to determine whether or not the flag associated with FdOption opt is set for Fd fd.

```
setFdOption :: Fd -> FdOption -> Bool -> IO ()
```

setFdOption fd opt val calls fcntl to set the flag associated with FdOption opt on Fd fd to val.

```
getLock :: Fd -> FileLock -> IO (Maybe (ProcessID, FileLock))
```

getLock fd lock calls fcntl to get the first FileLock for Fd fd which blocks the FileLock lock. If no such FileLock exists, getLock returns Nothing. Otherwise, it returns Just (pid, block), where block is the blocking FileLock and pid is the ProcessID of the process holding the blocking FileLock.

```
setLock :: Fd -> FileLock -> IO ()
```

setLock fd lock calls fcntl with F_SETLK to set or clear a lock segment for Fd fd as indicated by the FileLock lock. setLock does not block, but fails with SystemError if the request cannot be satisfied immediately.

```
waitToSetLock :: Fd -> FileLock -> IO ()
```

waitToSetLock fd lock calls fcntl with F_SETLKW to set or clear a lock segment for Fd fd as indicated by the FileLock lock. If the request cannot be satisfied immediately, waitToSetLock blocks until the request can be satisfied.

```
fdSeek :: Fd -> SeekMode -> FileOffset -> IO FileOffset
```

fdSeek fd whence offset calls lseek to position the Fd fd at the given offset from the starting location indicated by whence. It returns the resulting offset from the start of the file in bytes.

6.1.6. Posix, Device- and Class-Specific Functions

```
terminalMode    :: TerminalMode -> TerminalAttributes -> Bool
withMode        :: TerminalAttributes -> TerminalMode -
> TerminalAttributes
withoutMode     :: TerminalAttributes -> TerminalMode -
> TerminalAttributes

bitsPerByte     :: TerminalAttributes -> Int
withBits        :: TerminalAttributes -> Int -> TerminalAttributes
```

```
controlChar     :: TerminalAttributes -> ControlCharacter -> Maybe Char
withCC          :: TerminalAttributes
                -> (ControlCharacter, Char)
                -> TerminalAttributes
withoutCC       :: TerminalAttributes
                -> ControlCharacter
                -> TerminalAttributes

inputTime       :: TerminalAttributes -> Int
withTime        :: TerminalAttributes -> Int -> TerminalAttributes

minInput        :: TerminalAttributes -> Int
withMinInput    :: TerminalAttributes -> Int -> TerminalAttributes

inputSpeed      :: TerminalAttributes -> BaudRate
withInputSpeed  :: TerminalAttributes -> BaudRate -> TerminalAttributes

outputSpeed     :: TerminalAttributes -> BaudRate
withOutputSpeed :: TerminalAttributes -> BaudRate -> TerminalAttributes

getTerminalAttributes :: Fd -> IO TerminalAttributes
```

`getTerminalAttributes fd` calls `tcgetattr` to obtain the `TerminalAttributes` associated with `Fd` `fd`.

```
setTerminalAttributes :: Fd
                      -> TerminalAttributes
                      -> TerminalState
                      -> IO ()
```

`setTerminalAttributes fd attr ts` calls `tcsetattr` to change the `TerminalAttributes` associated with `Fd` `fd` to `attr`, when the terminal is in the state indicated by `ts`.

```
sendBreak :: Fd -> Int -> IO ()
```

`sendBreak fd duration` calls `tcsendbreak` to transmit a continuous stream of zero-valued bits on `Fd` `fd` for the specified implementation-dependent `duration`.

```
drainOutput :: Fd -> IO ()
```

`drainOutput fd` calls `tcdrain` to block until all output written to `Fd` `fd` has been transmitted.

```
discardData :: Fd -> QueueSelector -> IO ()
```

`discardData fd queues` calls `tcflush` to discard pending input and/or output for `Fd` `fd`, as indicated by the `QueueSelector` `queues`.

```
controlFlow :: Fd -> FlowAction -> IO ()
```

`controlFlow fd action` calls `tcflow` to control the flow of data on `Fd` `fd`, as indicated by `action`.

```
getTerminalProcessGroupID :: Fd -> IO ProcessGroupID
```

`getTerminalProcessGroupID fd` calls `tcgetpgrp` to obtain the `ProcessGroupID` of the foreground process group associated with the terminal attached to `Fd` `fd`.

```
setTerminalProcessGroupID :: Fd -> ProcessGroupID -> IO ()
```

`setTerminalProcessGroupID fd pgid` calls `tcsetpgrp` to set the `ProcessGroupID` of the foreground process group associated with the terminal attached to `Fd` `fd` to `pgid`.

6.1.7. Posix System Databases

```
groupName    :: GroupEntry -> String
groupID      :: GroupEntry -> GroupID
groupMembers :: GroupEntry -> [String]

getGroupEntryForID :: GroupID -> IO GroupEntry
```

`getGroupEntryForID gid` calls `getgrgid` to obtain the `GroupEntry` information associated with `GroupID` `gid`.

The operation may fail with:

`NoSuchThing`

There is no group entry for the GroupID.

```
getGroupEntryForName :: String -> IO GroupEntry
```

`getGroupEntryForName name` calls `getgrnam` to obtain the `GroupEntry` information associated with the group called `name`.

The operation may fail with:

`NoSuchThing`

There is no group entry for the name.

```
userName      :: UserEntry -> String
userID        :: UserEntry -> UserID
userGroupID   :: UserEntry -> GroupID
homeDirectory :: UserEntry -> String
userShell     :: UserEntry -> String
```

```
getUserEntryForID :: UserID -> IO UserEntry
```

`getUserEntryForID gid` calls `getpwuid` to obtain the `UserEntry` information associated with `UserID uid`. The operation may fail with:

`NoSuchThing`

There is no user entry for the UserID.

```
getUserEntryForName :: String -> IO UserEntry
```

`getUserEntryForName name` calls `getpwnam` to obtain the `UserEntry` information associated with the user login `name`.

The operation may fail with:

`NoSuchThing`

There is no user entry for the name.

6.1.8. POSIX Errors

```
getErrorCode :: IO ErrorCode
```

`getErrorCode` returns the current value of the external variable `errno`. It never fails.

```
setErrorCode :: ErrorCode -> IO ()
```

`setErrorCode err` sets the external variable `errno` to `err`. It never fails.

```
noError :: ErrorCode
noError = 0

argumentListTooLong, e2BIG              :: ErrorCode
badFd, eBADF                            :: ErrorCode
brokenPipe, ePIPE                       :: ErrorCode
directoryNotEmpty, eNOTEMPTY            :: ErrorCode
execFormatError, eNOEXEC                :: ErrorCode
fileAlreadyExists, eEXIST               :: ErrorCode
fileTooLarge, eFBIG                     :: ErrorCode
filenameTooLong, eNAMETOOLONG           :: ErrorCode
improperLink, eXDEV                     :: ErrorCode
inappropriateIOControlOperation, eNOTTY :: ErrorCode
inputOutputError, eIO                   :: ErrorCode
interruptedOperation, eINTR             :: ErrorCode
invalidArgument, eINVAL                 :: ErrorCode
invalidSeek, eSPIPE                     :: ErrorCode
isADirectory, eISDIR                    :: ErrorCode
noChildProcess, eCHILD                  :: ErrorCode
noLocksAvailable, eNOLCK                :: ErrorCode
noSpaceLeftOnDevice, eNOSPC             :: ErrorCode
noSuchOperationOnDevice, eNODEV         :: ErrorCode
noSuchDeviceOrAddress, eNXIO            :: ErrorCode
noSuchFileOrDirectory, eNOENT           :: ErrorCode
noSuchProcess, eSRCH                    :: ErrorCode
notADirectory, eNOTDIR                  :: ErrorCode
notEnoughMemory, eNOMEM                 :: ErrorCode
operationNotImplemented, eNOSYS         :: ErrorCode
operationNotPermitted, ePERM            :: ErrorCode
permissionDenied, eACCES                :: ErrorCode
readOnlyFileSystem, eROFS               :: ErrorCode
resourceBusy, eBUSY                     :: ErrorCode
resourceDeadlockAvoided, eDEADLK        :: ErrorCode
resourceTemporarilyUnavailable, eAGAIN  :: ErrorCode
tooManyLinks, eMLINK                    :: ErrorCode
tooManyOpenFiles, eMFILE                :: ErrorCode
tooManyOpenFilesInSystem, eNFILE        :: ErrorCode
```

Chapter 7. The `text` category: text manipulation

7.1. Pretty

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

```
module Pretty where
infixl 6 <>
infixl 6 <+>
infixl 5 $$, $+$
data Doc  - the Document datatype

- The primitive Doc values
empty                     :: Doc
text                      :: String   -> Doc
char                      :: Char     -> Doc
int                       :: Int      -> Doc
integer                   :: Integer  -> Doc
float                     :: Float    -> Doc
double                    :: Double   -> Doc
rational                  :: Rational -> Doc
semi, comma, colon, space, equals              :: Doc
lparen, rparen, lbrack, rbrack, lbrace, rbrace :: Doc
parens, brackets, braces  :: Doc -> Doc
quotes, doubleQuotes      :: Doc -> Doc

- Combining Doc values
(<>)    :: Doc -> Doc -> Doc     - Beside
hcat    :: [Doc] -> Doc          - List version of <>
(<+>)   :: Doc -> Doc -> Doc     - Beside, separated by space
hsep    :: [Doc] -> Doc          - List version of <+>
($$)    :: Doc -> Doc -> Doc     - Above; if there is no
                                   - overlap it "dovetails" the two
vcat    :: [Doc] -> Doc          - List version of $$
cat     :: [Doc] -> Doc          - Either hcat or vcat
sep     :: [Doc] -> Doc          - Either hsep or vcat
fcat    :: [Doc] -> Doc          - "Paragraph fill" version of cat
fsep    :: [Doc] -> Doc          - "Paragraph fill" version of sep
nest    :: Int -> Doc -> Doc     - Nested
hang    :: Doc -> Int -> Doc -> Doc
punctuate :: Doc -> [Doc] -> [Doc]
- punctuate p [d1, ... dn] = [d1 <> p, d2 <> p, ... dn-1 <> p, dn]
```

```
- Displaying Doc values
instance Show Doc
render     :: Doc -> String             - Uses default style
renderStyle  :: Style -> Doc -> String
data Style = Style { lineLength     :: Int,   - In chars
                       ribbonsPerLine :: Float, - Ratio of ribbon length
                                                - to line length
                       mode :: Mode
               }
data Mode = PageMode            - Normal
             | ZigZagMode          - With zig-zag cuts
             | LeftMode            - No indentation, infinitely long lines
             | OneLineMode         - All on one line
```

7.2. The `Regex` and `MatchPS` interfaces

(Sigbjorn Finne supplied the regular-expressions interface.)

The `Regex` library provides quite direct interface to the GNU regular-expression library, for doing manipulation on `PackedString`s. You probably need to see the GNU documentation if you are operating at this level. Alternatively, you can use the simpler and higher-level `RegexString` interface.

The datatypes and functions that `Regex` provides are:

```
data PatBuffer  # just a bunch of bytes (mutable)

data REmatch
 = REmatch (Array Int GroupBounds)  - for $1, ... $n
           GroupBounds              - for $` (everything before match)
           GroupBounds              - for $& (entire matched string)
           GroupBounds              - for $' (everything after)
           GroupBounds              - for $+ (matched by last bracket)

- GroupBounds hold the interval where a group
- matched inside a string, e.g.
-
- matching "reg(exp)" "a regexp" returns the pair (5,7) for the
- (exp) group. (PackedString indices start from 0)

type GroupBounds = (Int, Int)

re_compile_pattern
        :: PackedString         - pattern to compile
        -> Bool                 - True <=> assume single-line mode
        -> Bool                 - True <=> case-insensitive
```

```
        -> PrimIO PatBuffer

re_match :: PatBuffer           - compiled regexp
         -> PackedString        - string to match
         -> Int                 - start position
         -> Bool                - True <=> record results in registers
         -> PrimIO (Maybe REmatch)

- Matching on 2 strings is useful when you're dealing with multiple
- buffers, which is something that could prove useful for
- PackedStrings, as we don't want to stuff the contents of a file
- into one massive heap chunk, but load (smaller chunks) on demand.

re_match2 :: PatBuffer          - 2-string version
          -> PackedString
          -> PackedString
          -> Int
          -> Int
          -> Bool
          -> PrimIO (Maybe REmatch)

re_search :: PatBuffer          - compiled regexp
          -> PackedString       - string to search
          -> Int                - start index
          -> Int                - stop index
          -> Bool               - True <=> record results in registers
          -> PrimIO (Maybe REmatch)

re_search2 :: PatBuffer         - Double buffer search
           -> PackedString
           -> PackedString
           -> Int               - start index
           -> Int               - range (?)
           -> Int               - stop index
           -> Bool              - True <=> results in registers
           -> PrimIO (Maybe REmatch)
```

The `MatchPS` module provides Perl-like "higher-level" facilities to operate on `PackedStrings`. The regular expressions in question are in Perl syntax. The "flags" on various functions can include: `i` for case-insensitive, `s` for single-line mode, and `g` for global. (It's probably worth your time to peruse the source code. . .)

```
matchPS :: PackedString    - regexp
        -> PackedString    - string to match
        -> [Char]          - flags
        -> Maybe REmatch   - info about what matched and where

searchPS :: PackedString    - regexp
```

```
         -> PackedString    - string to match
         -> [Char]          - flags
         -> Maybe REmatch

- Perl-like match-and-substitute:
substPS :: PackedString     - regexp
        -> PackedString     - replacement
        -> [Char]           - flags
        -> PackedString     - string
        -> PackedString

- same as substPS, but no prefix and suffix:
replacePS :: PackedString  - regexp
          -> PackedString  - replacement
          -> [Char]        - flags
          -> PackedString  - string
          -> PackedString

match2PS :: PackedString   - regexp
         -> PackedString   - string1 to match
         -> PackedString   - string2 to match
         -> [Char]         - flags
         -> Maybe REmatch

search2PS :: PackedString  - regexp
          -> PackedString  - string to match
          -> PackedString  - string to match
          -> [Char]        - flags
          -> Maybe REmatch

- functions to pull the matched pieces out of an REmatch:

getMatchesNo    :: REmatch -> Int
getMatchedGroup :: REmatch -> Int -> PackedString -> PackedString
getWholeMatch   :: REmatch -> PackedString -> PackedString
getLastMatch    :: REmatch -> PackedString -> PackedString
getAfterMatch   :: REmatch -> PackedString -> PackedString

- (reverse) brute-force string matching;
- Perl equivalent is index/rindex:
findPS, rfindPS :: PackedString -> PackedString -> Maybe Int

- Equivalent to Perl "chop" (off the last character, if any):
chopPS :: PackedString -> PackedString

- matchPrefixPS: tries to match as much as possible of strA starting
- from the beginning of strB (handy when matching fancy literals in
- parsers):
matchPrefixPS :: PackedString -> PackedString -> Int
```

7.3. The `RegexString` interface

(Simon Marlow supplied the String Regex wrapper.)

For simple regular expression operations, the `Regex` library is a little heavyweight. `RegexString` permits regex matching on ordinary Haskell `String`s.

The datatypes and functions that `RegexString` provides are:

```
data Regex              - a compiled regular expression

mkRegex
        :: String       - regexp to compile
        -> Regex        - compiled regexp

matchRegex
        :: Regex        - compiled regexp
        -> String       - string to match
        -> Maybe [String] - text of $1, $2, ... (if matched)
```

Chapter 8. The `util` category: miscellaneous utilities

8.1. The `Memo` library

The `Memo` library provides fast polymorphic memo functions using hash tables. The interface is:

```
memo :: (a -> b) -> a -> b
```

So, for example, `memo f` is a version of `f` that caches the results of previous calls.

The searching is very fast, being based on pointer equality. One consequence of this is that the caching will only be effective if *exactly the same argument is passed again to the memoised function*. This means not just a copy of a previous argument, but the same instance. It's not useful to memoise integer functions using this interface, because integers are generally copied a lot and two instances of '27' are unlikely to refer to the same object.

This memoisation library works well when the keys are large (or even infinite).

The memo table implementation uses weak pointers and stable names (see the GHC/Hugs library document) to avoid space leaks and allow hashing for arbitrary Haskell objects. NOTE: while individual memo table entries will be garbage collected if the associated key becomes garbage, the memo table itself will not be collected if the function becomes garbage. We plan to fix this in a future version.

There's another version of `memo` if you want to explicitly give a size for the hash table (the default size is 1001 buckets):

```
memo_sized :: Int -> (a -> b) -> a -> b
```

8.2. The `Select` interface

The `Select` interface provides a Haskell wrapper for the `select()` OS call supplied by many modern UNIX variants. `Select` exports the following:

```
type TimeOut = Maybe Int
  - Nothing => wait indefinitely.
  - Just x | x >= 0    => block waiting for 'x' micro seconds.
  -        | otherwise => block waiting for '-x' micro seconds.

hSelect :: [Handle]
```

```
        -> [Handle]
        -> [Handle]
        -> TimeOut
        -> IO SelectResult

type SelectResult
 = ( [Handle]  - input  handles ready
   , [Handle]  - output handles ready
   , [Handle]  - exc.   handles ready
   )
```

Here's an example of how it could be used:

```
module Main(main) where

import Select
import IO

main :: IO ()
main = do
  hSetBuffering stdin NoBuffering
  putStrLn "waiting for input to appear"
  hSelect [stdin] [] [] Nothing
  putStrLn "input ready, let's try reading"
  x <- getChar
  print x
```

where the call to `hSelect` makes the process go to sleep until there's input available on `stdin`.

Notice that this particular use of `hSelect` is now really a no-op with GHC compiled code, as its implementation of IO will take care to avoid blocking the process (i.e., all running Haskell threads), and call `select()` for you, if needs be. However, `hSelect` exposes functionality that is useful in other contexts (e.g., you want to wait for input on two `Handles` for 3 seconds, but no longer.)

8.3. GetOpt

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

```
module GetOpt where

- 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 decription of options

getOpt :: ArgOrder a    - non-option handling
       -> [OptDescr a]  - options recognised
       -> [String]      - the command-line
       -> ( [a]         - options
          , [String]    - non-options
  ,[String]     - error messages
  )

data ArgOrder a
  = RequireOrder
  | Permute
  | ReturnInOrder (String -> a)
```

- 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:

```
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 out-
put on stderr"
 , Option ['V','?'] ["version"] (NoArg Version)       "show ver-
sion 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..."
```

8.4. The `Readline` interface

(Darren Moffat supplied the `Readline` interface.)

The `Readline` module is a straightforward interface to the GNU Readline library. As such, you will need to look at the GNU documentation (and have a `libreadline.a` file around somewhere. . .)

You'll need to link any Readlining program with `-lreadline -ltermcap`, besides the usual `-syslib ghc` (and `-fhaskell-1.3`).

The main function you'll use is:

```
readline :: String{-the prompt-} -> IO String
```

If you want to mess around with Full Readline G(l)ory, we also provide:

```
rlInitialize, addHistory,

rlBindKey, rlAddDefun, RlCallbackFunction(..),

rlGetLineBuffer, rlSetLineBuffer, rlGetPoint, rlSetPoint, rlGetEnd,
rlSetEnd, rlGetMark, rlSetMark, rlSetDone, rlPendingInput,
```

`rlPrompt, rlTerminalName, rlSetReadlineName, rlGetReadlineName`

(All those names are just Haskellised versions of what you will see in the GNU readline documentation.)

Chapter 9. Win32

To do.