Input / Output

Fast Track to Haskell

Andres Löh, Edsko de Vries

8–9 April 2013 —

Haskell functions

Haskell functions are functions in the mathematical sense:

- if functions are applied (in possibly different contexts) to the same arguments, they yield the same result;
- in particular, function results cannot depend implicitly on the context;
- this property validates a large number of program transformations.

Sharing and inlining

$$\textbf{let}\ x = f\ y$$
$$\textbf{in}\ \ldots\ x\ \ldots\ x\ \ldots$$

and

$$\ldots\ f\ y\ \ldots\ f\ y\ \ldots$$

always yield the same result (although one may be more efficient than the other).

Transformations such as these are thus available as manual as well as compiler optimizations.

Operations with side effects

What about functions that need access to the "outside world"?

```
getSystemTime   :: ...
getRandomNumber :: ...
readLineFromFile :: ...
printToScreen   :: ...
acceptOnSocket  :: ...
```

The result of these functions depends on the state of the system at the time of execution.

In some cases (`printToScreen`), we are not even interested in the result – just in an effect to the system.

Operations with side effects

What about functions that need access to the "outside world"?

```
getSystemTime    :: Time
getRandomNumber  :: Int
readLineFromFile :: File → String
printToScreen    :: String → ...     -- nothing, void?
acceptOnSocket   :: Socket → ConnectionData
```

Taking our considerations about Haskell functions into account, the above types can't be correct in Haskell – even though this is the approach most other programming languages would take.

Operations with side effects

What about functions that need access to the “outside world”?

```
getSystemTime   :: IO Time
getRandomNumber :: IO Int
readLineFromFile :: File → IO String
printToScreen   :: String → IO ()
acceptOnSocket  :: Socket → IO ConnectionData
```

Instead, as we will see, all of these yield an `IO` result type, thereby specifically indicating that the function may depend on and affect the outside world.

What would happen?

For the moment let's assume we have

$$\text{printToScreen} :: \text{String} \rightarrow \text{Int}$$

which prints the text to the screen as a side effect and returns the number of characters printed.

```
x :: Int
x = printToScreen "Hello"
```

When would this print something? When the module is loaded? When `x` is evaluated the first time? Every time `x` is evaluated?

What would happen?

For the moment let's assume we have

$$\text{printToScreen} :: \text{String} \to \text{Int}$$

which prints the text to the screen as a side effect and returns the number of characters printed.

```
test :: Int
test = let x = printToScreen "Hello" in 20
```

Should this print anything?

What would happen?

For the moment let's assume we have

$$\text{printToScreen} :: \text{String} \to \text{Int}$$

which prints the text to the screen as a side effect and returns the number of characters printed.

$$\text{test} :: \text{Int}$$
$$\text{test} = \textbf{let}\ x = \text{printToScreen}\ \texttt{"Hello"}\ \textbf{in}\ x + x$$

Should this print once or twice?

What would happen?

For the moment let's assume we have

$$\text{printToScreen} :: \text{String} \rightarrow \text{Int}$$

which prints the text to the screen as a side effect and returns the number of characters printed.

$$\begin{aligned}&\text{test} :: \text{Int}\\&\text{test} = \text{printToScreen}\ \texttt{"Hello"} + \text{printToScreen}\ \texttt{"Hello"}\end{aligned}$$

And this, once or twice?

What would happen?

For the moment let's assume we have

$$\text{printToScreen} :: \text{String} \rightarrow \text{Int}$$

which prints the text to the screen as a side effect and returns the number of characters printed.

$$\text{test} :: \text{Int}$$
$$\text{test} = \text{snd}\ (\text{printToScreen}\ \texttt{"Hello"}, \text{printToScreen}\ \texttt{"world"})$$

And this? And potentially in what order?

While it's certainly possible to answer all these questions in a consistent way, the resulting system is certainly subtle to use – in particular if we want to keep Haskell's lazy evaluation strategy.

The IO type

The type IO a describes a value of type a that can be obtained by executing an action that will potentially depend on or affect the state of the outside world.

The IO type

The type IO a describes a value of type a that can be obtained by executing an action that will potentially depend on or affect the state of the outside world.

- You can think of IO a as a type of actions or abstract descriptions.

The IO type

The type IO a describes a value of type a that can be obtained by executing an action that will potentially depend on or affect the state of the outside world.

- You can think of IO a as a type of actions or abstract descriptions.
- Constructing a value of type IO a means talking about IO, but not (yet) doing IO.

Examples

```
getLine :: IO String
```

is an action that, when executed, reads a line from the user, and returns the string that has been typed in.

Examples

getLine :: IO String

is an action that, when executed, reads a line from the user, and returns the string that has been typed in.

putStrLn :: String $\to$ IO ()

is a function that takes a string and returns an action. The action, when executed, prints the string and returns a value of type () ...

The () type

The type `()` is called unit. You can see it as the 0-tuple.

The datatype has just one element, also written `()`.

Apart from special built-in syntax, you can imagine `()` to be defined as

```
data () = ()
```

The `()` type

The type `()` is called unit. You can see it as the 0-tuple.

The datatype has just one element, also written `()`.

Apart from special built-in syntax, you can imagine `()` to be defined as

```
data () = ()
```

- The `()` type is useful as an argument to `IO` for actions that produce no interesting result. Because `IO` is a parameterized type, some argument type is required, and using `()` is better than, say, using `Bool` or `Int` but always returning the same value.
- There are other, similar, uses of `()` as an argument to other parameterized types.

IO actions are first-class citizens

We can do whatever we like with terms of type `IO a`. Simply mentioning them or passing them around does not cause any effects.

Example:

```
fst (length [getLine, getLine], putStrLn "Hello")
```

yields `2` and neither reads nor prints anything.

The main program

How do we execute an IO action, then?

The main program

How do we execute an IO action, then?

Every full Haskell program must define an entry point

```
main :: IO ()
```

So IO () is the type of the entire Haskell program. This IO action (it might be a rather complicated one) is executed by Haskell's run-time system.

The main program

How do we execute an IO action, then?

Every full Haskell program must define an entry point

```
main :: IO ()
```

So IO () is the type of the entire Haskell program. This IO action (it might be a rather complicated one) is executed by Haskell's run-time system.

We can also enter IO actions on the GHCi prompt. GHCi will treat these in a special way and execute them before printing their result.

The main program

How do we execute an IO action, then?

Every full Haskell program must define an entry point

```
main :: IO ()
```

So IO () is the type of the entire Haskell program. This IO action (it might be a rather complicated one) is executed by Haskell's run-time system.

We can also enter IO actions on the GHCi prompt. GHCi will treat these in a special way and execute them before printing their result.

Try this with getLine . . .

Composing IO actions

It is rather obvious that we need a way to compose actions into larger actions.

One such way is:

$$(\gg) :: \mathrm{IO}\ a \to \mathrm{IO}\ b \to \mathrm{IO}\ b$$

Given two actions, this produces an action that, when executed, runs both actions in sequence, igores the result of the first, and returns the result of the second.

Composing IO actions

It is rather obvious that we need a way to compose actions into larger actions.

One such way is:

$$(\gg) :: \mathrm{IO}\ a \rightarrow \mathrm{IO}\ b \rightarrow \mathrm{IO}\ b$$

Given two actions, this produces an action that, when executed, runs both actions in sequence, igores the result of the first, and returns the result of the second.

Example: Compare

```
putStr "Hello " >> putStr "world\n"
putStr "Hello world\n"
putStrLn "Hello world"
```

(The \n is an escaped newline character.)

How to (not) use results of IO actions

We cannot directly feed an IO a to a function expecting an a.

```
map toUpper getLine   -- type error
```

- This is not surprising: map toUpper expects a String, but getLine is an IO String.
- We cannot expect the whole expression to have type String, as it still performs IO.

How to (not) use results of `IO` actions

We cannot directly feed an `IO a` to a function expecting an `a`.

```
map toUpper getLine    -- type error
```

- This is not surprising: `map toUpper` expects a `String`, but `getLine` is an `IO String`.
- We cannot expect the whole expression to have type `String`, as it still performs IO.

Neither does

```
putStrLn getLine    -- type error
```

work.

The "bind" operator

We need a more sophisticated variant of $(\gg)$:

$$
\begin{array}{lllll}
(\gg) & :: \mathrm{IO}\ a \to & \mathrm{IO}\ b & \to \mathrm{IO}\ b \\
(\gg\!=) & :: \mathrm{IO}\ a \to & (a \to \mathrm{IO}\ b) & \to \mathrm{IO}\ b
\end{array}
$$

While $(\gg)$ drops the result of the first action, in $(\gg\!=)$ the second argument is a function that is passed the result of the first action.

The "bind" operator

We need a more sophisticated variant of $(\gg)$:

$$
\begin{array}{llcl}
(\gg) & :: \mathrm{IO}\ a \to & \mathrm{IO}\ b & \to \mathrm{IO}\ b \\
(\gg\!=) & :: \mathrm{IO}\ a \to & (a \to \mathrm{IO}\ b) & \to \mathrm{IO}\ b
\end{array}
$$

While $(\gg)$ drops the result of the first action, in $(\gg\!=)$ the second argument is a function that is passed the result of the first action.

Now we can write:

$$
\begin{array}{l}
\mathrm{echo} :: \mathrm{IO}\ () \\
\mathrm{echo} = \mathrm{getLine} \gg\!= \mathrm{putStrLn}
\end{array}
$$

Examples

Often, we put a lambda expression on the right hand side:

$$\mathsf{echoTwice} :: \mathsf{IO}\ ()$$
$$\mathsf{echoTwice} = \mathsf{getLine} \ggg\!= \lambda \mathsf{xs} \rightarrow \mathsf{putStrLn}\ \mathsf{xs} \gg \mathsf{putStrLn}\ \mathsf{xs}$$

Examples

Often, we put a lambda expression on the right hand side:

```
echoTwice :: IO ()
echoTwice = getLine >>= \xs -> putStrLn xs >> putStrLn xs
```

```
greeting :: IO ()
greeting =
  putStrLn "Who are you?" >>
  getLine                 >>= \name ->
  putStrLn ("Hello, " ++ name ++ "!")
```

Examples

Often, we put a lambda expression on the right hand side:

```
echoTwice :: IO ()
echoTwice = getLine >>= \xs -> putStrLn xs >> putStrLn xs
```

```
greeting :: IO ()
greeting =
  putStrLn "Who are you?" >>
  getLine                 >>= \name ->
  putStrLn ("Hello, " ++ name ++ "!")
```

But:

```
capitalize = getLine >>= \xs -> map toUpper xs   -- type error
```

still fails.

Embedding “pure” computations into IO

We often want to combine normal functions with IO.

The function

```
return :: a → IO a
```

allows us to do so.

Embedding "pure" computations into IO

We often want to combine normal functions with IO.

The function

```
return :: a → IO a
```

allows us to do so.

Remember

In order to obtain the result of an IO a, we are allowed, but not forced to do IO.

On the other hand, if the type of a value does not involve IO, we cannot use IO in order to come up with it.

Examples

$$\text{capitalize} :: \text{IO String}$$
$$\text{capitalize} = \text{getLine} \gg\!= \lambda \text{xs} \to \text{return (map toUpper xs)}$$

Examples

$$\text{capitalize} :: \text{IO String}$$
$$\text{capitalize} = \text{getLine} \ggg= \lambda\text{xs} \rightarrow \text{return (map toUpper xs)}$$

Random numbers in a range:

$$\text{dieRoll} :: \text{IO Int}$$
$$\text{dieRoll} = \text{randomRIO}\ (1, 6)$$

Roll twice, return both results in a pair:

$$\text{rollTwice} :: \text{IO (Int, Int)}$$
$$\text{rollTwice} = \text{dieRoll} \ggg= \lambda\text{x} \rightarrow \text{dieRoll} \ggg= \lambda\text{y} \rightarrow \text{return (x, y)}$$

Sequence in terms of bind

The $(\gg)$ operator can be defined in terms of $(\gg\!=)$, by simply ignoring the function argument:

$$(\gg) :: \mathsf{IO}\ a \rightarrow \mathsf{IO}\ b \rightarrow \mathsf{IO}\ b$$
$$x \gg y = x \gg\!= \lambda\_ \rightarrow y$$

The IO type is a functor

instance Functor IO **where**
  $\text{fmap f x} = \text{x} \gg\!= \lambda\text{r} \rightarrow \text{return (f r)}$

Type of fmap specialized to IO:

$\text{fmap} :: (\text{a} \rightarrow \text{b}) \rightarrow \text{IO a} \rightarrow \text{IO b}$

The IO type is a functor

```
instance Functor IO where
  fmap f x = x >>= \r -> return (f r)
```

Type of fmap specialized to IO:

```
fmap :: (a -> b) -> IO a -> IO b
```

Example:

```
capitalize :: IO String
capitalize = getLine >>= \xs -> return (map toUpper xs)
```

This is how we have defined it before.

The IO type is a functor

```
instance Functor IO where
  fmap f x = x >>= λr → return (f r)
```

Type of fmap specialized to IO:

```
fmap :: (a → b) → IO a → IO b
```

Example:

```
capitalize :: IO String
capitalize = fmap (map toUpper) getLine
```

This is how to use fmap.

The IO type is a functor

```
instance Functor IO where
  fmap f x = x >>= λr → return (f r)
```

Type of fmap specialized to IO:

```
fmap :: (a → b) → IO a → IO b
```

Example:

```
capitalize :: IO String
capitalize = fmap (map toUpper) getLine
```

The outer call to fmap operates on the IO String produced by getLine.

The IO type is a functor

```
instance Functor IO where
  fmap f x = x >>= λr → return (f r)
```

Type of fmap specialized to IO:

```
fmap :: (a → b) → IO a → IO b
```

Example:

```
capitalize :: IO String
capitalize = fmap (map toUpper) getLine
```

The argument to fmap is of type String → String, which is a synonym for [Char] → [Char].

The IO type is a functor

```
instance Functor IO where
  fmap f x = x >>= λr → return (f r)
```

Type of fmap specialized to IO:

```
fmap :: (a → b) → IO a → IO b
```

Example:

```
capitalize :: IO String
capitalize = fmap (map toUpper) getLine
```

The inner map operates on the [Char] produced by getLine.

The toUpper function has type Char → Char.

Note partial parameterization in action!

No escape from IO

There actually is a function

```
unsafePerformIO :: IO a → a    -- stay away from this for now
```

However, it's use is mainly for interfacing to such C functions that actually are pure while the type system pessimistically assumes they aren't.

- The function is, as the name says, quite unsafe, and it should not be used.
- It is a deliberate design decision that once you're depending on IO, you cannot write code based on that that doesn't.
- All of $(\gg)$, $(\gg=)$, fmap produce results of IO type.

It’s not that bad

Keep in mind:

- $(\gg=)$ allows you to locally use the result of a preceding IO action and feed it to other functions;
- both $(\gg=)$ with return and fmap can be used to combine entirely IO-free functions with IO-based code;
- in fact, it is good style to try to separate the IO part of a program as much as possible from the core logic of the program, and this is encouraged by the type system.

A simple example – overly IO-centric

Counting from 0 up to a given number, printing each even number.

```haskell
count :: Int → IO ()
count n = go 0
  where
    go :: Int → IO ()
    go i | i > n     = return ()
         | even i    = print i ≫ go (i + 1)
         | otherwise = go (i + 1)
```

Note that print combines putStrLn and show:

```haskell
print :: (Show a) ⇒ a → IO ()
print = putStrLn ∘ show
```

A simple example – keeping IO separated

$$\mathsf{evens} :: \mathsf{Int} \to [\mathsf{Int}]$$
$$\mathsf{evens}\ \mathsf{n} = \mathsf{filter}\ \mathsf{even}\ [0\mathinner{\ldotp\ldotp}\mathsf{n}]$$
$$\mathsf{arrangeInLines} :: \mathsf{Show}\ \mathsf{a} \Rightarrow [\mathsf{a}] \to \mathsf{String}$$
$$\mathsf{arrangeInLines} = \mathsf{unlines} \circ \mathsf{map}\ \mathsf{show}$$
$$\mathsf{count} :: \mathsf{Int} \to \mathsf{IO}\ ()$$
$$\mathsf{count} = \mathsf{putStr} \circ \mathsf{arrangeInLines} \circ \mathsf{evens}$$

Clearly separates the computation, layout, and actual IO.

Reading from and writing to files

```
type FilePath = String
  -- operations on entire files
readFile :: FilePath → IO String
writeFile :: FilePath → String → IO ()
```

Reading from and writing to files

```
type FilePath = String
  -- operations on entire files
readFile :: FilePath → IO String
writeFile :: FilePath → String → IO ()
```

```
data IOMode = ReadMode | WriteMode
            | AppendMode | ReadWriteMode
data Handle   -- abstract
withFile    :: FilePath → IOMode → (Handle → IO r) → IO r
hGetLine    :: Handle → IO String
hPutStr     :: Handle → String → IO ()
hPutStrLn   :: Handle → String → IO ()
...
```

Allocating a resource

$$\text{withFile} :: \text{FilePath} \rightarrow \text{IOMode} \rightarrow (\text{Handle} \rightarrow \text{IO r}) \rightarrow \text{IO r}$$

The withFile function is an instance of a common pattern:

- a resource (in this case, a file handle) is allocated;
- the resource is passed to an argument function;
- after the function has been executed, the resource is automatically freed.

Copying the first line of a file

```
copyFileLine :: FilePath → FilePath → IO ()
copyFileLine src tgt =
  withFile src ReadMode  ( λsrcH →
  withFile tgt  WriteMode  ( λtgtH  →
    hGetLine srcH ≫= hPutStrLn tgtH))
```

Copying the first line of a file

```haskell
copyFileLine :: FilePath → FilePath → IO ()
copyFileLine src tgt =
  withFile src ReadMode  $ λsrcH →
  withFile tgt  WriteMode  $ λtgtH  →
    hGetLine srcH >>= hPutStrLn tgtH
```

It is common to use `($)`, a synonym for function application with low operator priority, to save parentheses:

```haskell
($) :: (a → b) → a → b
f $ x = f x
infixr 0  $
```

Copying the first line of a file

```
copyFileLine :: FilePath → FilePath → IO ()
copyFileLine src tgt =
  withFile src ReadMode  $ λsrcH →
  withFile tgt  WriteMode  $ λtgtH  →
    hGetLine srcH >>= hPutStrLn tgtH
```

(Try what happens if the source file does not exist or is empty.)

Buffering

Handles can be buffered:

$$\mathrm{hSetBuffering} :: \mathrm{Handle} \to \mathrm{BufferMode} \to \mathrm{IO}\ ()$$

$$\begin{aligned}\textbf{data}\ \mathrm{BufferMode} &= \mathrm{NoBuffering} \mid \mathrm{LineBuffering} \\ &\mid \mathrm{BlockBuffering}\ (\mathrm{Maybe\ Int})\end{aligned}$$

Buffering

Handles can be buffered:

$$\mathit{hSetBuffering} :: \mathit{Handle} \rightarrow \mathit{BufferMode} \rightarrow \mathit{IO}\ ()$$

$$\begin{aligned}\textbf{data}\ \mathit{BufferMode} &= \mathit{NoBuffering} \mid \mathit{LineBuffering} \\ &\mid \mathit{BlockBuffering}\ (\mathit{Maybe\ Int})\end{aligned}$$

Common pitfall: putStr and putStrLn write to

$$\mathit{stdout} :: \mathit{Handle}$$

By default, stdout is line-buffered. In GHCi, stdout is not buffered. This can lead to unwanted effects when writing interactive programs.

do notation

There's a certain similarity between sequences of binds and imperative programming with assignments:

```
example :: IO ()
example =
  hSetBuffering stdout NoBuffering >>
  putStr "What is your name? " >>
  getLine                          >>= \name ->
  putStr "Where do you live? " >>
  getLine                          >>= \loc ->
  let answer | loc == "London" = "That's wonderful!"
             | otherwise       = "Sorry, " ++ name
                               ++ ", where is " ++ loc ++ "?"
  in putStrLn answer
```

Skills Matter

Well-Typed

do notation

There's a certain similarity between sequences of binds and imperative programming with assignments:

```haskell
example :: IO ()
example = do
  hSetBuffering stdout NoBuffering
  putStr "What is your name? "
  name <- getLine
  putStr "Where do you live? "
  loc <- getLine
  let answer | loc == "London" = "That's wonderful!"
             | otherwise       = "Sorry, " ++ name
                                 ++ ", where is " ++ loc ++ "?"
  putStrLn answer
```

Every line contains one statement.

do notation

There's a certain similarity between sequences of binds and imperative programming with assignments:

```haskell
example :: IO ()
example = do
  hSetBuffering stdout NoBuffering
  putStr "What is your name? "
  name ← getLine
  putStr "Where do you live? "
  loc ← getLine
  let answer | loc == "London" = "That's wonderful!"
             | otherwise       = "Sorry, " ++ name
                                 ++ ", where is " ++ loc ++ "?"
  putStrLn answer
```

Normal lines must have IO type. Their results are ignored.

do notation

There's a certain similarity between sequences of binds and imperative programming with assignments:

```haskell
example :: IO ()
example = do
  hSetBuffering stdout NoBuffering
  putStr "What is your name? "
  name <- getLine
  putStr "Where do you live? "
  loc <- getLine
  let answer | loc == "London" = "That's wonderful!"
             | otherwise       = "Sorry, " ++ name
                                 ++ ", where is " ++ loc ++ "?"
  putStrLn answer
```

Names can be bound to `IO` -results with `<-`. They scope over the rest of the `do` block.

do notation

There's a certain similarity between sequences of binds and imperative programming with assignments:

```haskell
example :: IO ()
example = do
  hSetBuffering stdout NoBuffering
  putStr "What is your name? "
  name ← getLine
  putStr "Where do you live? "
  loc ← getLine
  let answer | loc == "London" = "That's wonderful!"
             | otherwise       = "Sorry, " ++ name
                                 ++ ", where is " ++ loc ++ "?"
  putStrLn answer
```

A **let** can be used without an **in**. The bound identifiers scope over the rest of the **do** block.

More about do

A corner case is a single monadic expression:

```haskell
helloWorld = do
  putStrLn "Hello world"
```

is the same as writing

```haskell
helloWorld =
  putStrLn "Hello world"
```

More about **do**

A corner case is a single monadic expression:

```
helloWorld = do
  putStrLn "Hello world"
```

is the same as writing

```
helloWorld =
  putStrLn "Hello world"
```

Remember that a **do** is never required. It is just “syntactic sugar” for a chain of $(\gg=)$ and $(\gg)$ applications.

Nested **do**

```
loop :: IO ()
loop = do
  putStrLn "Type 'q' to quit."
  c ← getChar    -- reads a single character
  if c == 'q'
    then putStrLn "Goodbye"
    else do
      putStrLn "Here we go again ..."
      loop
```

If we want to embed a sequence commands into a subexpression and use **do** -notation for that, we need another **do** .

Nested do

```
loop :: IO ()
loop = do
  putStrLn "Type 'q' to quit."
  c ← getChar    -- reads a single character
  if c == 'q'
    then putStrLn "Goodbye"
    else do
      putStrLn "Here we go again ..."
      loop
```

If we want to embed a sequence commands into a subexpression and use **do** -notation for that, we need another **do** .

Note furthermore that there are lots of **do** blocks without return .

About return

- The purpose of return in Haskell is to embed computations into the IO type.
- As such, return can be used in many different places.
- It is fine to use return in the middle of a sequence of commands. It does not jump anywhere.

This is fine:

```
do
  n ← return 2
  print n
```

More things we can do with IO

Mutable variables (module Data.IORef)

In IO, we can actually use mutable variables:

```
data IORef a   -- abstract

newIORef   :: a → IO (IORef a)
readIORef  :: IORef a → IO a
writeIORef :: IORef a → a → IO ()
```

In terms of these, further operations can be defined, such as:

```
modifyIORef :: IORef a → (a → a) → IO ()
modifyIORef ref f = do
  x ← readIORef ref
  writeIORef ref (f x)
```

Mutable variables (module Data.IORef)

In IO, we can actually use mutable variables:

```
data IORef a   -- abstract

newIORef   :: a → IO (IORef a)
readIORef  :: IORef a → IO a
writeIORef :: IORef a → a → IO ()
```

In terms of these, further operations can be defined, such as:

```
modifyIORef :: IORef a → (a → a) → IO ()
modifyIORef ref f = do
  x ← readIORef ref
  writeIORef ref (f x)
```

Using mutable variables, we can build mutable data structures in Haskell – if we really, really want them.

Random numbers (module System.Random)

We can use a standard random number generator, initialized by system time:

$$
\begin{aligned}
&\mathsf{randomIO} &&:: \mathsf{Random}\ \mathsf{a} \Rightarrow \mathsf{IO}\ \mathsf{a} \\
&\mathsf{randomRIO} &&:: \mathsf{Random}\ \mathsf{a} \Rightarrow (\mathsf{a}, \mathsf{a}) \rightarrow \mathsf{IO}\ \mathsf{a} \quad \text{-- takes a range}
\end{aligned}
$$

By default, the Random class is instantiated by integers, floating point numbers, Char and Bool.

Access system properties (Data.Time, System.Environment)

```
getCurrentTime :: IO UTCTime
getProgName    :: IO String
getArgs        :: IO [String]
getEnvironment :: IO ([String, String])
```

It is easy to access the current time, or to get information about the environment the program was started in.

Accessing the file system (System.Directory)

```
getHomeDirectory      :: IO FilePath
setCurrentDirectory   :: FilePath → IO ()

getDirectoryContents  :: FilePath → IO [FilePath]
doesFileExist         :: FilePath → IO Bool
doesDirectoryExist    :: FilePath → IO Bool
```

Example: Obtain all proper files in a directory

```
filesInDirectory :: FilePath → IO [FilePath]
filesInDirectory dir = do
  contents ← getDirectoryContents dir
   . . .
```

Now `contents` contains subdirectories as well as files. We want to filter the list, but . . .

Example: Obtain all proper files in a directory

```
filesInDirectory :: FilePath → IO [FilePath]
filesInDirectory dir = do
  contents ← getDirectoryContents dir
  return (filter doesFileExist contents)    -- type error
```

This is wrong:

```
filter         :: (a → Bool) → [a] → [a]
doesFileExist :: FilePath → IO Bool
```

Example: Obtain all proper files in a directory

```
filesInDirectory :: FilePath → IO [FilePath]
filesInDirectory dir = do
  contents ← getDirectoryContents dir
  filterM doesFileExist contents
```

Fortunately, there's a function called `filterM`:

```
filter        :: (a →    Bool) → [a] →    [a]
filterM       :: (a → IO Bool) → [a] → IO [a]
doesFileExist :: FilePath → IO Bool
```

Defining filterM

There's nothing magic about filterM – it is easy to define it:

$$
\begin{array}{l}
\mathrm{filterM} :: (a \rightarrow \mathrm{IO\ Bool}) \rightarrow [a] \rightarrow \mathrm{IO}\ [a] \\
\mathrm{filterM}\ p\ [\,] \qquad\qquad = \mathrm{return}\ [\,] \\
\mathrm{filterM}\ p\ (x : xs)\ = \mathbf{do} \\
\quad \mathrm{bool} \qquad\quad\ \leftarrow p\ x \\
\quad \mathrm{filteredTail} \leftarrow \mathrm{filterM}\ p\ xs \\
\quad \mathrm{return}\ (\mathbf{if}\ \mathrm{bool}\ \mathbf{then}\ x : \mathrm{filteredTail}\ \mathbf{else}\ \mathrm{filteredTail})
\end{array}
$$

Defining filterM

There's nothing magic about filterM – it is easy to define it:

```
filterM :: (a → IO Bool) → [a] → IO [a]
filterM p []        = return []
filterM p (x : xs)  = do
  bool        ← p x
  filteredTail ← filterM p xs
  return (if bool then x : filteredTail else filteredTail)
```

Note that this function

- is very similar to filter, but more explicit about the order of events,
- still follows the standard design principle for list functions.

Listing files recursively

This time, we'd like to have all directories and files (recursively) at a given location:

```
recursiveFiles :: FilePath → IO [FilePath]
recursiveFiles dir = do
  contents ← getDirectoryContents dir
  subdirs  ← filterM doesDirectoryExist contents
  let subdirs′ = filter (not ∘ ("." `isPrefixOf`)) subdirs
  ...
```

We'd like to execute `recursiveFiles` on all the `subdirs′`. How? It looks like a `map` ...

Listing files recursively

```haskell
recursiveFiles :: FilePath → IO [FilePath]
recursiveFiles dir = do
  contents ← getDirectoryContents dir
  subdirs  ← filterM doesDirectoryExist contents
  let subdirs′ = filter (not ∘ ("." `isPrefixOf`)) subdirs
  map recursiveFiles subdirs′   -- type error
```

This does not work:

```haskell
map recursiveFiles subdirs′ :: [IO [FilePath]]
```

but we need `IO [FilePath]`.

Listing files recursively

```
recursiveFiles :: FilePath → IO [FilePath]
recursiveFiles dir = do
   contents ← getDirectoryContents dir
   subdirs  ← filterM doesDirectoryExist contents
   let subdirs' = filter (not ∘ ("." `isPrefixOf`)) subdirs
   mapM recursiveFiles subdirs'    -- type error
```

There's a `mapM` much like `filterM`:

```
map  :: (a →    b) → [a] →    [b]
mapM :: (a → IO b) → [a] → IO [b]
```

But still

```
mapM recursiveFiles subdirs' :: IO [[FilePath]]
```

Listing files recursively

```haskell
recursiveFiles :: FilePath → IO [FilePath]
recursiveFiles dir = do
  contents ← getDirectoryContents dir
  subdirs  ← filterM doesDirectoryExist contents
  let subdirs′ = filter (not ∘ ("." `isPrefixOf`)) subdirs
  recs     ← mapM recursiveFiles subdirs′
  return (contents ++ concat recs)
```

We can now use

```haskell
concat :: [[a]] → [a]
```

to collapse the list of lists.

Defining mapM

We can define mapM using the standard pattern for lists directly, but it's easier to see it as a composition of an ordinary map with a function called sequence:

```
sequence :: [IO a] → IO [a]
sequence []       = return []
sequence (x : xs) = do
  r  ← x
  rs ← sequence xs
  return (r : rs)
```

Defining mapM

We can define mapM using the standard pattern for lists directly, but it's easier to see it as a composition of an ordinary map with a function called sequence:

```
sequence :: [IO a] → IO [a]
sequence []       = return []
sequence (x : xs) = do
  r  ← x
  rs ← sequence xs
  return (r : rs)
```

Now:

```
mapM :: (a → IO b) → [a] → IO [b]
mapM f = sequence ∘ map f
```

Using `sequence`

The function `sequence` can be independently useful.

Question: What does the following do?

```
sequence [randomRIO (1, 10), return 5, fmap read getLine]
```

Useful IO functions (module Control.Monad)

```
liftM       :: (a → b) → IO a → IO b    -- synonym for fmap
mapM        :: (a → IO b) → [a] → IO [b]
mapM_       :: (a → IO b) → [a] → IO ()
forM        :: [a] → (a → IO b) → IO [b]  -- flipped mapM
forM_       :: [a] → (a → IO b) → IO ()
sequence    :: [IO a] → IO [a]
sequence_   :: [IO a] → IO ()
forever     :: IO a → IO b
filterM     :: (a → IO Bool) → [a] → IO [a]
replicateM  :: Int → IO a → IO [a]
replicateM_ :: Int → IO a → IO ()
when        :: Bool → IO () → IO ()
unless      :: Bool → IO () → IO ()
```

The underscored variants are more efficient versions for the case that the results are uninteresting.

Powerful IO operators

- Except for return and $(\gg=)$, all operations on IO can be defined in libraries.
- There is no need for built-in control structures (loops, etc) in the language.
- What makes it possible to define these so easily is the combination of having IO as a first-class type, higher-order functions and on-demand evaluation.
- Simon Peyton Jones therefore says: “Haskell is the world’s finest imperative programming language.”