UNIVERSITY OF CALGARY

Computer Science

Haskell I/O

λ

In- and Output

- Conceptually, functional languages have a problem with input and output, since reading in data is not well modelled using functions and output is usually only a side-effect of functions (and, as such, outside of the usual semantic treatment of function evaluation by reductions)
- Some functional languages do not care about this and simply add to their functional part a rather standard IO part (usually copied from an imperative language)
- Other languages try to stay within the functional ideas as much as possible (which usually can become rather confusing, see Haskell)

Haskell: the monadic classes

- The monad construct was introduced in category theory, a rather theoretic field
- Haskell has three type classes that are based on the monad principle: `Functor`, `Monad` and `MonadPlus`
- The list type class and the `IO` type classes are examples of subclasses of the monadic classes
- `Functor` requires the function `fmap`, `Monad` requires `>>`, `>>=` (bind) and `return`, and `MonadPlus` extends `Monad` by requiring a zero element `mzero` (as a constant function) and `mplus`
- The `IO` type class is not a subclass of `MonadPlus`

Haskell: the monadic classes

- Monads are similar to abstract data types since they require each subclass/instantiation of them to obey certain laws (i.e. there are certain equations between expressions that we expect to be fulfilled)
- The `Monad` class essentially defines around "normal functions" an environment that is (or can be) changed when these functions are performed (i.e. we convert side effects into valid function results in the extended "world")
- This naturally allows for a (theoretically) better treatment of IO (as actions in the outside world).

IO in Haskell

- For the basic data types `Char` and `String` (`[Char]`), Haskell has a corresponding `IO` type that represents values of the basic type with the added "world environment"
- If we are only interested in the effects on the environment (i.e. if we write out data) then we assign to the function as result type `IO ()` (the `IO` data type corresponding to the unit type)
- To produce sequences of actions, we can either use the monad functions `>>` and `>>=`, or we can use the `do` construct

IO in Haskell

- For reading and writing a character, we use the built-in functions `getChar` and `putChar`
- For other data types, the type classes `Show` and `Read` force the existence of functions that convert values of the types into characters or strings, resp. functions that convert characters or strings into values of the other types:

  `show (2+5)` returns `"7"`

  `read "True" ::Bool` returns `True`

In this case, you need to specify the type in order to tell Haskell what to look for.

IO in Haskell

- The following little program reads in one character and then prints it out:

```
main    :: IO ()
main    = do  c <- getChar
              putChar c
```

- Note that do allows for the sequence of actions and that c acts here very much like a variable in an imperative or object-oriented language (but you can't re-assign it)

File handling

- `putChar` and `getChar` write to *stdout* and read from *stdin* (which Haskell calls channels, in modern operating systems we call this streams)
- Other channels and files can be used by creating handles for them. A handle requires a file path and an IOmode and can then be used by several functions to read or write from the file associated with it
- The handle variants of `putChar` and `getChar` are `hPutChar` and `hGetChar` (with a handle as first argument)
- There are quite a few additional functions available (many in the IO library), to read/write lines or whole files

IO exception handling

- While for normal functions it might be acceptable to let the run-time system terminate with an error when they produce an error, there are a lot of "normal" error conditions associated with IO (like end-of-file)
- Therefore Haskell introduced special IO related errors (via a special data type `IOError`) and exception handling via exception handlers (that convert values from `IOError` to the normal `IO a` values)
- Central to this is the function catch:

```
catch :: IO a -> (IOError -> IO a)  -> IO a
```

IO exception handling

- Example (see "Gentle Introduction"):

`catch` recently moved out of Prelude

```
import Haskell.X.Prelude (catch,IOException)
getLineWErr :: IO String
getLineWErr =  catch gL (\err -> return ("Error: " ++
                                  show (err :: IOException)))
               where
                  gL = do c <- getChar
                          if c == '\n'
                          then return ""
                          else do l <- getLineWErr
                                  return (c:l)
```

need to disambiguate the type of `err`

recursion