I/O in Purely Functional Languages

Four centuries ago, Descartes pondered the mind-body problem: how can incorporeal minds interact with physical bodies?

Designers of purely declarative languages (such as Haskell) face an analogous problem: how can virtual software interact with the physical world?

Or using fewer $0.25 words: how can you do input/output in a purely functional language?

If the computation consists of just applying a function to an argument and getting the result this is easy.

But what if you want interaction?

Three main techniques:

- stream-based I/O
- continuation-based I/O
- monads

Stream-Based I/O

The input and output are both (potentially infinite) streams of characters.

This works OK if input and output aren't interleaved ... but due to lazy evaluation for more complex interactive programs it becomes difficult to predict the program's behavior.

Example (from Miranda). In Miranda the value of a command level expression is a list of `system messages'. The system `prints' such a list of messages by reading it in order from left to right, evaluating and obeying each message in turn as it is encountered.

sys_message ::= Stdout [char] | Stderr [char]
| Tofile [char] [char] | Closefile [char] | Appendfile [char]
| System [char] | Exit num

Stdout, for example, causes its string argument to be printed to standard out.

Continuation-Based I/O

Example from the previous version of Haskell: write a file, read it back in, and compare the contents

```
s1 = “this is a test ...”

main = writeFile “ReadMe” s1 exit (
   readfile “ReadMe” exit (\s2->
   appendChan stdout
      (if s1==s2 then “contents match”
      else “something intervened!”) exit
   done))
```

type Name = String
type StrCont = String -> Dialog

writeFile :: Name -> String -> FailCont -> SuccCont -> Dialog
readFile :: Name -> FailCont -> StrCont -> Dialog
appendChan :: Name -> String -> FailCont -> SuccCont -> Dialog

done :: Dialog
exit :: FailCont
(FailCont is the type failure continuation)

Monads

The current preferred solution to Haskell's mind-body problem.
Based on some mathematically intense ideas from category theory.

Exposition mostly stolen from Phil Wadler's paper in Sept 1997 ACM Computing Surveys.

IO ( ): the type of simple commands.
( ) is the unit type, like void in C++

A term of type IO ( ) denotes an action, but does not necessarily perform the action.

Function to print a character:

```
putChar :: Char -> IO ()
```

Thus `putChar '!'` denotes the action that, *if it is ever performed*, will print an exclamation.

done :: IO ( )
Thus `done` denotes the action that, *if it is ever performed*, will do nothing (done is not built in -- we'll define it shortly).

More Monads

>> is a function to combine monads. If m and n are commands, then m>>n is the command that, if it is ever performed, will do m and then n.

```
(>>) :: IO () -> IO () -> IO ()
```

(Actually >> has a more general type in Haskell 98, but let's use this for now.)

We can now define a function `puts` to put a string:

```
puts :: String -> IO ()
puts [] = done
puts (c:s) = putc c >> puts s
```

Therefore `puts ’hi’` is equivalent to

```
putc ’h’ >> (putc ’i’ >> done)
```

`puts ’hi’` is thus the command that, if it is ever performed, will print `’hi’`

Performing Commands

But, you cry, how does anything ever actually happen??

1) We can have a distinguished top-level variable main with the following type

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

When we use “runhugs” (rather than “hugs”) the command ‘main’ will be run.

```
main = puts “haskell lives!”
```

(If you want to try this, it is in the file `~borning/hasell/hello.hs` on ceylon.)

2) We can start Haskell as usual using “hugs”. We define any variable as a command, then invoke it.

In a file:

```
test = puts “haskell lives!”
```

then invoke it using

```
test
```

Monads and Equational Reasoning: ML

Consider the following ML expression. (ML is a mostly-functional programming language, but with call-by-value semantics.)

```
print “hi there”;
```

This has value `() : unit` and as a side effect prints to standard out.

```
print “ha!”; print(“ha!”);
```

prints “ha!ha!”

However, if we try to abstract this:

```
let val
  x =(print “ha!”)
in x ; x end;
```

then the laugh is on us ...

Or consider:

```
let val
  x =(print “ha!”)
in 3+4 end;
```

Monads and Equational Reasoning: ML (2)

This does work in ML if we abstract a function:

```
let fun
  f () =(print “ha!”)
in f (); f () end;
```

this prints “ha!ha!”

and finally:

```
let fun
  f () =(print “ha!”)
in 3+4 end;
```

this evaluates to 7 (and doesn't guffaw)

Monads and Equational Reasoning: Haskell

Now consider the same examples in Haskell.

```
puts “ha!” >> puts (“ha!”);
```

is the command that, *if it is ever executed*, prints “ha!ha!”

We can abstract this:

```
let
  x =(puts “ha!”)
in x >> x
```

to give the command that, *if it is ever executed*, prints “ha!ha!”

Further:

```
let
  x =(puts “ha!”)
in 3+4
```

if evaluated, has value 7 (as expected)

Commands that Yield Values

```
getChar :: IO Char
```

if the input buffer contains ABC and we perform the getChar command then the command yields ‘A’, leaving BC in the buffer.

The return command does nothing and returns a value.

```
return :: a -> IO a
```

(We need this so that we can return values from commands -- if we just tried to include the value in a sequence of commands the type would be wrong.)

The done command isn’t actually built in -- but we can define it as

```
done :: IO ()
done = return ()
```

Primitives for Combining Commands that Yield Values

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

(Again, in Haskell 98 this actually has a more general type.

Thus if m and n are commands, then

```
m>>=n
```

is the command that, if it is performed, first performs m, which should yield a value x. It then performs n, passing x as a parameter. The value returned by n is the value of the whole command `m>>=n`

Example:

```
getChar >>= putChar
```

gets a character and then puts it.

We can use this to define a command to get n characters from the input:

```
gets 0     = return []
gets (i+1) = getChar >>= \c ->
             gets i >>= \s ->
             return (c:s)

main = (gets 10) >>= puts
```

An Analog of Let

Rather than

```
m>>=n
```

we can write

```
do  x <- m
    n x
```

This is the command, that if it is ever performed, first performs m and binds the resulting value to x. It then performs n, passing x as a parameter. The value returned by n is the value of the whole command.

Caution regarding layout rules: the version above works, but the following doesn’t:

```
do  x <- m
   n x
```

Thus:

```
do  c <- getChar
    putChar c
```

More Examples of “Do”

```
gets 0     = return []
gets (i+1) = do c <- getChar
                s <- gets i
                return (c:s)

-- get 10 characters and print them
main = do str <- (gets 10)
          puts str
```

Some built-in I/O commands in Haskell:

Output Functions

These functions write to the standard output device (this is normally the user’s terminal).

```
putChar   :: Char -> IO ()
putStr    :: String -> IO ()
putStrLn  :: String -> IO () -- adds a newline
print     :: Show a => a -> IO ()
```

The print function outputs a value of any printable type to the standard output device (this is normally the user’s terminal). Printable types are those that are instances of class Show; print converts values to strings for output using the show operation and adds a newline.

For example, a program to print the first 20 integers and their powers of 2:

```
main = print ([(n, 2^n) | n <- [0..19]])
```

Input Functions

These functions read input from the standard input device (normally the user’s terminal).

```
getChar      :: IO Char
getLine      :: IO String
getContents  :: IO String
interact     :: (String -> String) -> IO ()
readIO       :: Read a => String -> IO a
readLine     :: Read a => IO a
```

Both getChar and getLine raise an exception on end-of-file.

The getContents operation returns all user input as a single string, which is read lazily as it is needed.

The interact function takes a function of type String->String as its argument. The entire input from the standard input device (normally the user’s terminal) is passed to this function as its argument, and the resulting string is output on the standard output device.

(From Section 7 of the Haskell Report).

Read and Show

```
show :: (Show a) => a -> String
```

example:

```
show (3.2+4.1) => “7.3”
```

```
read :: (Read a) => String -> a
```

examples:

```
read “3.2” + 10.0 => 13.2
read “3.2” :: Double => 13.2
```

(however, the type system will be unhappy with just

read “3.2”

since it won’t be able to resolve the overloading)

The Monadic Calculator

```
-- Haskell calculator (demonstrates a simple
-- interactive program using monads)
-- see ~borning/haskell/calc.hs

calc = do putStr “first number: “
          a <- readLn
          putStr “second number: “
          b <- readLn
          putStr “sum: “
          putStrLn (show (a+b))
          putStr “again? “
          again <- getLine
          if head again == ‘y’ then calc
             else return ()
```

Other Useful Functions for I/O

lines -- break up a string into substrings at the newline chars

```
lines            :: String -> [String]
```

unlines -- put it back together

```
unlines          :: [String] -> String
```

lex -- gets the first token from a string, returning a tuple with the token and the remaining part of the string

```
lex “3*x+2*y” => (“3”, “*x+2*y”)
```

Showing

Consider the Tree type: (see the file ~borning/haskell/Tree1.hs)

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

We can write a show function as follows:

```
showTree  :: (Show a) => Tree a -> String

showTree (Leaf x) = show x

showTree (Branch l r) =
   “<“ ++ showTree l ++ “|” ++ showTree r ++ “>”

sample  = (Branch (Branch (Leaf 1) (Leaf 2))
                  (Leaf 3))
```

Then `showTree sample => “<<1|2>|3>”`

We can definite this as an instance of Show to use the generic overloaded show functions (so that trees will print using showTree):

```
instance Show a => Show (Tree a) where
   show t = showTree t
```

Showing using Derived Instances

We can generate a show method automatically using derived instances.

This version is on ~borning/haskell/Tree2.hs

```
data Tree a = Leaf a | Branch (Tree a) (Tree a) deriving show

sample  = (Branch (Branch (Leaf 1) (Leaf 2))
                  (Leaf 3))
```

Then `sample prints as`

Branch (Branch (Leaf 1) (Leaf 2)) (Leaf 3)

ReadS and ShowS

showTree ends up doing extra concatenation -- we can avoid this by passing along an accumulator (stored in ~borning/haskell/Tree3.hs)

```
 -- version of showTree that uses an accumulator
 showsTree :: (Show a) => Tree a->String->String
 showsTree (Leaf x) s = shows x s
 showsTree (Branch l r) s = ‘<‘ :
    showsTree l (‘|’ : showsTree r (‘>’ : s))
```

Then `showsTree sample ““ => “<<1|2>|3>”`

```
 -- and a version using functional composition:

 showsTree2 :: (Show a) => Tree a -> ShowS
 showsTree2 (Leaf x) = shows x
 showsTree2 (Branch l r) = (‘<‘:) . showsTree l
   . (‘|’:) . showsTree r . (‘>’:)

showsTree2 sample ““ => “<<1|2>|3>”
```

Object Level to Function Level

from the Gentle Introduction:

> “Something more important than just tidying up the code has come about by this transformation: We have raised the presentation from an object level (in this case, strings) to a function level. We can think of the typing as saying that showsTree maps a tree into a showing function. Functions like (‘<‘ :) or (“a string” ++) are primitive showing functions, and we build up more complex functions by function composition.”