CS-205 lecture 11:
Interactive programming and further topics

Cécilia Pradic

6/11/23

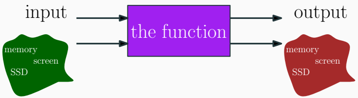
Autograder is live

Link to submit your coursework

`https://csautograder.swansea.ac.uk/web/project/69`

- Detailed submission instructions on canvas
- A bug was reported to me yesterday, should be fixed
- All tests re-ran after final submission
- 37 marks allocated automatically, the rest by handgrading
- The handgrader might compensate for harsh automated grading

Last week: interactive programming



What we have seen

How to

- write types for programs with IO side-effects in types `IO a`

```
print   :: Show a => a -> IO ()
getLine :: IO String
```

- combine them using bind >>= or the **do** notation
- a couple of examples
- compile haskell programs using ghc

Extended example: validating input

```haskell
getYN :: String -> IO Bool
getYN prompt = putStr (prompt ++ "[y/N]:") >>
               hFlush stdout >>
               getLine >>= \s ->
               if s `elem` answers then
                 return (s `elem` yanswers)
               else
                 putStrLn "Wrong input!" >> getYN prompt
      where yanswers = ["y","Y","yes","Yes"]
            answers  = yanswers ++ ["","n", "N","no","No"]
```

Extended example: validating input

```
getYN :: String -> IO Bool
getYN prompt = do {
                  putStr (prompt ++ "[y/N]:");
                  hFlush stdout;
                  s <- getLine;
                  if s `elem` answers then
                     return (s `elem` yanswers)
                  else
                      do {
                           putStrLn "Wrong input!";
                           getYN prompt
                         }
                 }
       where yanswers = ["y","Y","yes","Yes"]
             answers  = yanswers ++ ["","n", "N","no","No"]
```

Some tps/considerations for the lab

- I have not gone over all the `IO` primitives

  → use the online documentation (`hackage/hoogle`)
- You might need some import statement to import functions like `hFlush` or `isDigit` as in e.g.

```
import Data.Char (isDigit) -- imports only isDigit
import System.IO -- imports everything in the module
```

- `hFlush stdout` ≡ `fflush(stdout)`

  flushes the `stdout` buffer → forces printing

Some further topics

Warning

The rest of the lecture will survey some topics you could look into if you want to keep writing Haskell in the future/are curious

Before we move on, questions about Haskell/CW/etc?

Ofc you are free to ask at any later point :)

(more detailed explanation on the material below in `lecture11.hs`)

Further topic 1: monads

Extended do notation?

```haskell
doList :: [(Int,Char)]
doList = do {
              x <- [1..5];
              y <- ['a','z'];
              return (x,y)
            }
```

Extended do notation?

```haskell
divMaybe :: Int -> Int -> Maybe Int
divMaybe x 0 = Nothing
divMaybe x y = Just (x `div` y)

doMaybe :: Int -> Int -> Int -> Maybe Int
doMaybe x y z = do {
                      a <- divMaybe x y;
                      b <- divMaybe z a;
                      return (a + b)
                   }
```

Workhorse behind this: the Monad typeclass

```haskell
(>>=) :: Monad m => m a -> (a -> m b) -> m b
return :: Monad m => a -> m a
```

- `m :: * -> *` is a variable, but not for a type
- Monads consist of a very generic, yet useful abstractions
- Typical instances: `Monad IO`, `Monad []`, `Monad Maybe`, `Monad (Cont r)`, `Monad (State s)`
- `Monad (State s)` = code "as if" we had mutable variables

Maybe the next example to look at if you are interested

The state monad

```
data State s a = Stateful (s -> a * s)

return :: a -> State s a
return x m = (x, m)

(>>=) :: State s a -> (a -> State s b) -> State s b
(Stateful c) >>= f = Stateful
                     \m -> let (x, m') = c m in
                           let Stateful g = f x in
                           g m'

escape :: State s a -> s -> a
escape (Stateful c) m = fst (c m)
```

- Usefuleness wrt IO: one can go back to pure computations via `escape`

(function of type `IO a -> a` named `unsafePerformIO`)

Laziness

Lazy evaluation

With FP languages, there are two popular kind of evaluation strategies

```
(\x -> x + x ) (1+2)
```

- **Eager/CBV**: evaluate arguments first

$$
\begin{aligned}
(\lambda x \to x + x)(1+2) &\to (\lambda x \to x + x)3 \\
&\to (\lambda x \to x + x)3 \\
&\to 3 + 3 \\
&\to 6
\end{aligned}
$$

- **Lazy/CBN**: substitute arguments in the function body first

$$
\begin{aligned}
(\lambda x \to x + x)(1+2) &\to (1+2) + (1+2) \\
&\to 3 + (1+2) \\
&\to 3 + 3 \\
&\to 6
\end{aligned}
$$

Haskell

In pure functional progamming languages, the evaluation strategy mostly does not matter for the result!

- Haskell is **lazy**. (there are pros/cons with that)
- It tries to avoid to duplicate computations (call by need strategy) (as)

$$\begin{aligned}(\lambda x \to x + x)(1+2) &\to (1+2)+(1+2)\\ &\to 3+3\\ &\to 6\end{aligned}$$

Pro/Cons laziness

Pros:

- Call-by-need can save some shared computation at low intellectual cost
  $\rightarrow$ nice for rapid prototyping of complicated code
- Some nice idiosyncratic applications in the next slide

Cons:

- Harder to reason about complexity
- Counter-intuitive
- More complicated runtime because thunking is necessary
- `unsafePerformIO` has really hard-to-predict behaviours
- laziness can easily be emulated in eager languages

(essentially replace `a` by `() -> a`)

Some applications

- Infinite values can be used seamlessly in the language

```
allNats :: [Int]
allNats = 0 : map (+1) allNats

-- >>> take 5 allNats
-- [0,1,2,3,4]
```

- Nice tricks, like support for memoization/dynamic programming **without side-effects or state monad**

Not possible in eager FP languages

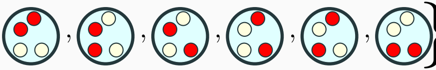
Next slides: explanation of the dynamic programming example in `lecture11.hs`

Extended example: binomial (1/4)

Problem

Compute the number of ways $\binom{n}{k}$ to pick k elements among n.

$$\binom{4}{2} = \#\Big\{ \quad \Big\}$$

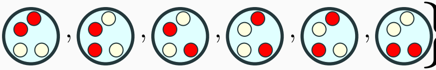


Extended example: binomial (1/4)

Problem

Compute the number of ways $\binom{n}{k}$ to pick k elements among n.

$$\binom{4}{2} = \#\{ \quad \}$$



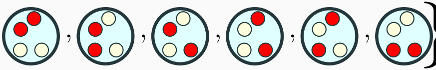
$$\binom{n}{k} = \#\{X \subseteq \{1, \ldots, n\} \mid \#X = k\} = \frac{n!}{k!(n-k)!}$$

Extended example: binomial (1/4)

Problem

Compute the number of ways $\binom{n}{k}$ to pick k elements among n.

$$\binom{4}{2} = \#\{ \quad \}$$



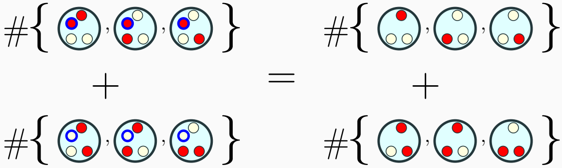
$$\binom{n}{k} = \#\{X \subseteq \{1, \ldots, n\} \mid \#X = k\} = \frac{n!}{k!(n-k)!}$$

Issue with the closed formula: $n!$ overflows fast while $\binom{k}{n}$ is polynomial if $k = O(1)$.
Alternative way of computing?

Extended example: binomial (2/4)

Decomposition by fixing an element and asking whether it is picked or not.

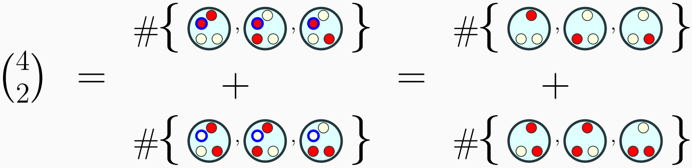
$$\binom{4}{2} =$$



$$= \binom{3}{1} + \binom{3}{2}$$

Extended example: binomial (2/4)

Decomposition by fixing an element and asking whether it is picked or not.



$$= \binom{3}{1} + \binom{3}{2}$$

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

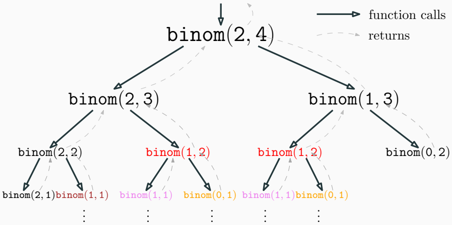
Extended example: binomial (3/4)

```
binom :: Int -> Int -> Int
binom k n | k > n = 0
binom 0 n         = 1
binom k n         = binom (k-1) (n-1) + binom k (n-1)
```

Proof of termination: by induction over *n*.

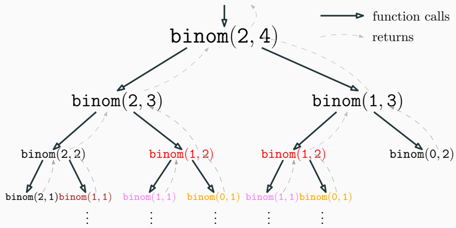
Extended example: binomial (3/5)

Issue: exponential number of calls (inefficient)



Extended example: binomial (3/5)

Issue: exponential number of calls (inefficient)



But there are redundant calls!

- Dynamic programming/memoization: cache the common subcomputations!

Extended example: binomial (4/4)

```
binomial :: Int -> Int -> Int
binomial k n | k > n     = 0
             | otherwise = a ! (k, n)
                 where a = array ((0,0),(k,n))
                                 [ ((i,j), b i j) | i <- [0..k],
                                                    j <- [0..n]]
                       b 0 k = 1
                       b i j | i == j = 1
                       b i j = (a ! (i,j-1)) + (a ! (i-1,j-1))
```

Some caveats:

- The imperative implementation might be more straightforward
- Also does not mesh well with **hash-consing** if the input domain is more complex

Next time

Innovation compared to previous years

We are dropping the mandatory verification part from the exam.

Leaves us one extra session. Rough ideas:

- Set up a "real" project with `cabal`
- Haskell & contemporary FP
  - Other FP languages
  - FP features in traditionally imperative languages
  - Proof assistants
- Some other ideas for further topics you may want to look at on your own that could make use of the module content?
- Q&A, AMA related to the module content

  (in which case, it would be useful to have questions in advance)
- Am also open to suggestions until Thursday!