CS-205 lecture 10:
Laziness & some perspectives on functional programming

Cécilia Pradic

01/11/24

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**)

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

Pro/Cons laziness

Pros:

- Call-by-need can save some shared computation at low intellectual cost
  → 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**

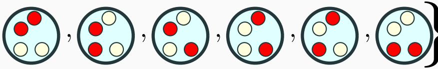
Not possible in eager FP languages

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

Problem

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

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



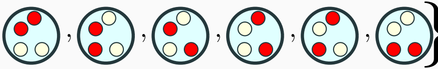
$$\}$$

Extended example: binomial (1/4)

Problem

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

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



$$\}$$

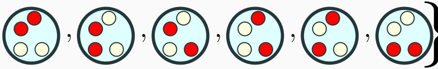
$$\binom{n}{k} \quad = \quad \#\{X \subseteq \{1, \ldots, n\} \mid \#X = k\} \quad = \quad \frac{n!}{k!(n-k)!}$$

Extended example: binomial (1/4)

Problem

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

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



$$\}$$

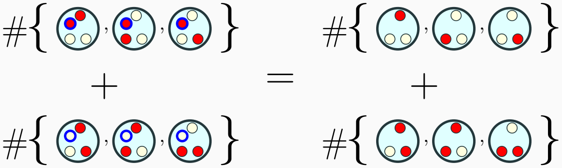
$$\binom{n}{k} = \#\{X \subseteq \{1, \ldots, n\} \mid \#X = k\} = \frac{n!}{k!(n-k)!}$$

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.

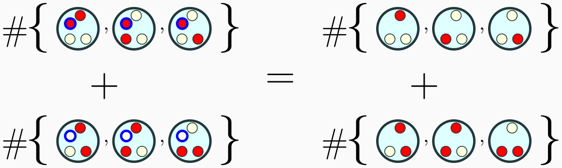
$$\binom{4}{2} =$$



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

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

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



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

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

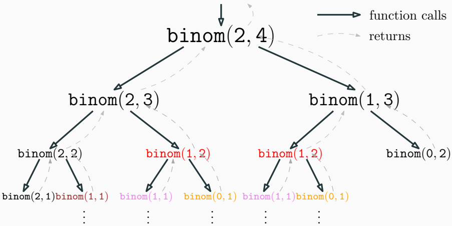
Extended example: binomial (3/4)

```
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.

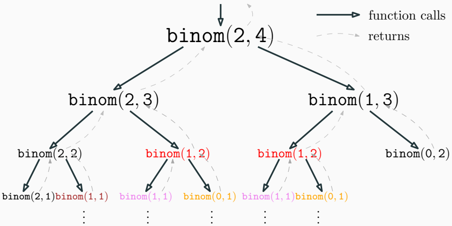
Extended example: binomial (3/5)

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

Simulated in other languages

Requires **state** to simulate call-by-need

```
final int N = 100;
final int K = 20;

final int[][] cache = new Array[K][N];
//assume that main() initializes cache with -1

static int binom(int k, int n)
{
  if (cache[k][n] != -1)
    return cache[k][n];
  if (k > n)
    return cache[k][n] = 0;
  if (k == 0)
    return cache[k][n] = 1;
  else
    return cache[k][n] = binom(k-1,n-1) + binom(k,n-1);
}
```

(Can be done in pure eager languages via a state monad)

My hot takes

Laziness by default

- introduces a lot of complexity for optimizing programs
  (not asymptotically, but up to a constant)
- complexifies the runtime
- (was historically a **strong** reason for haskell existing)
- is **sometimes** nice when prototyping **roughly**
  (CBNeed alone not as good as nice memoization/hash-consing)
  (and benefits don't stack)

List comprehension (switch to different slides)

Some perspectives on functional programming

What have we learned?

Began to program in a very opiniated FP language

- only **pure functions** by default

Some new features we focused on:

- recursive definitions (OK not new, but...)
- parametric polymorphism `fst :: (a, b) -> a`
- algebraic datatypes

  `data AST = Var String | App AST AST | Lambda String AST`
- lambdas (anonymous functions) `\ x -> (x, x^2)`
- higher-order functions (`map`, `filter`)
- type classes (`Show`, `Monad`, . . .)

Transferable skills?

Other functional programming languages

OCaml

- The most mainstream ML dialect (Milner)
- Eager (more performance-oriented)
- No typeclasses, more sophisticated module system
- Similar type system based on HM
- Industry variants: F# (M$), ReasonML (FB), Bucklescript

Other functional programming languages

OCaml

- The most mainstream ML dialect (Milner)
- Eager (more performance-oriented)
- No typeclasses, more sophisticated module system
- Similar type system based on HM
- Industry variants: F# (M$), ReasonML (FB), Bucklescript

LISP:

- Designates a variety of languages (ex: Scheme)
- Typically dynamically typed, based on lists
- Scripting language for emacs among others

Other functional programming languages

OCaml

- The most mainstream ML dialect (Milner)
- Eager (more performance-oriented)
- No typeclasses, more sophisticated module system
- Similar type system based on HM
- Industry variants: F# (M$), ReasonML (FB), Bucklescript

LISP:

- Designates a variety of languages (ex: Scheme)
- Typically dynamically typed, based on lists
- Scripting language for emacs among others

FP design also had significant influence on

Scala, Erlang, Rust, Mathematica, javascript (!)

More hardcore FP stuff?

Advanced topics in Haskell/OCaml:

- Metaprogramming (generics/template/BER)
- GADTs
- higher-kinds

Most hardcore FP languages

Dependently-typed languages: Coq, Lean, Agda, Idris

- Mixes types and values
- Type system rich enough to **do mathematics in**
- Proof assistants/interactive theorem prover

- Expertise on those topics in the theory group in Swansea

(options for projects)

Functional features in more mainstream languages

In **Python**, **Java**, **javascript** and **C++**:

- Historically, objects to simulate higher-order functions

  (cumbersome, requires class definitions)

- Lately: introduction of **lambdas** (anonymous functions)
- Various level of gracefuleness…

  (beware of lexical/dynamic scoping and typing)

For quick reference

`https://learnxinyminutes.com/` and search “lambda”

Lambdas in C++11

Some example from an old student project:

```
auto it = find_if(points.begin(),
                  points.end(),
                  [&f](Vertex * v){
                    return *v == *(f.points[0]); });
```

- The good: static scoping, clear semantics for closures
- The ugly: the type of a λ is compiler/OS-dependent?...
  - Not too much of a hassle when using type inference with `auto`
  - Except for the type errors

Lambdas in Java

Example from some labwork for another module:

```
public static void main(String[] args) throws Exception
{
  Random r = new Random();
  Graph g = new Graph(5, x -> y -> x != y && r.nextInt() % 3 ==
  g.toDotFile("myExample");
}

public Graph(int size,
             Function<Integer,Function<Integer, Boolean>> gen)
```

- The good: static scoping
- The bad: limited support for closures

Lambdas and list comprehension in Python

```
>>> list(map(lambda y: y*y, \
             filter(lambda x: x%5 == 2, range(0,70))) \
        ))
[4, 49, 144, 289, 484, 729, 1024, 1369, 1764, 2209, 2704, 3249,

>>> [ x * x for x in range(0,70) if x%5 == 2]
[4, 49, 144, 289, 484, 729, 1024, 1369, 1764, 2209, 2704, 3249,
```

- The good: reasonable syntax
- The bad: dynamical scoping

Programming with recursion?

Huge issue in “mainstream languages” for complex programs:

- The call stack is of ridiculously small size (4Ko)
- Lots of recursive calls $\Rightarrow$ premature stack overflows

Programming with recursion?

Huge issue in “mainstream languages” for complex programs:

- The call stack is of ridiculously small size (4Ko)
- Lots of recursive calls $\Rightarrow$ premature stack overflows
- (Less of an issue in Haskell due to laziness)

Programming with recursion?

Huge issue in “mainstream languages” for complex programs:

- The call stack is of ridiculously small size (4Ko)
- Lots of recursive calls ⇒ premature stack overflows
- (Less of an issue in Haskell due to laziness)

Solution

Tail-call optimization

Tail-call optimization in an example

The following OCaml code is **tail-recursive**

(value in the recursive call = returned value)

```
let rec findZero f = function
  [] -> None
| head :: _ when f head = 0 -> Some head
| _ :: tail -> findZero f tail
```

morally optimized into a while loop $\Rightarrow$ no stack pointers/overflows

- Common: recursive def $\mapsto$ tail-rec def using an **accumulator**

(you will see that during prolog)

The equivalent while loop if you are curious

Still in OCaml

(one can program in an imperative style there)

(although non-idiomatic)

```
let findZero (f : 'a -> int) (xs : 'a list) : 'a option =
  let r = ref None in let ys = ref xs in
  while !r = None && !ys != [] do
    let head :: tail = !r in
    if f head = 0 then
      r := Some head
    else ys := tail
  done; !r
```

(in truth the compiler does this at a lower level)

But outside of the FP world...

Warning

Some **big** compilers/interpreters **don't** implement TCO optimization!!

- Historical culprits: python or java...
- javascript: browser-dependent

$\Rightarrow$ in those languages, iterative solutions are ultimately going to be more efficient

So in conclusion

- I hope you had fun and retain some things
- From Monday: no more lectures from me (until revisions)
- I should be in labs/office hours

So in conclusion

- I hope you had fun and retain some things
- From Monday: no more lectures from me (until revisions)
- I should be in labs/office hours

Who is immune to propaganda?

`https://github.com/promises-aplus/promises-spec/issues/94`

`https://pages.cpsc.ucalgary.ca/~robin/class/449/Evolution.htm`

So in conclusion

- I hope you had fun and retain some things
- From Monday: no more lectures from me (until revisions)
- I should be in labs/office hours

Who is immune to propaganda?

`https://github.com/promises-aplus/promises-spec/issues/94`

`https://pages.cpsc.ucalgary.ca/~robin/class/449/Evolution.htm`

Thank you for your attention

So in conclusion

- I hope you had fun and retain some things
- From Monday: no more lectures from me (until revisions)
- I should be in labs/office hours

Who is immune to propaganda?

`https://github.com/promises-aplus/promises-spec/issues/94`

`https://pages.cpsc.ucalgary.ca/~robin/class/449/Evolution.htm`

Thank you for your attention

I wish you a nice continuation of your studies!