Functional Programming

Vsevolod Nikulin

Okinawa Institute of Science and Technology

vsevolod.nikulin@oist.jp

November 22, 2020

Overview

1. Immutability
2. Laziness
3. Parallelism

Immutability

Immutability

Imperative programming languages:

- Commands executed one after another
- There are variables
- Variables are changed (mutated) by commands

Purely functional programming languages:

- There are terms (expressions)
- Terms are written using other terms
- Terms are evaluated only once
- No ”Variables”, therefore no mutation

Example: Terms

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

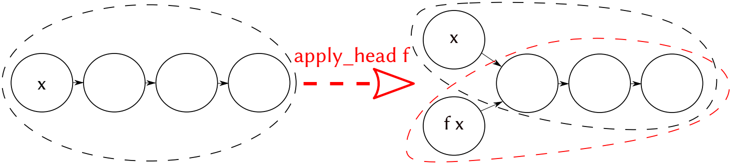
Example: List (1)

When we want to "change" a list, we actually create an entire new list based on the old one.

Example: Apply function to the head of a list

apply_head f x:xs = (f x) : xs

But don't worry! No unnecessary copying under the hood of compiled program:



Example: List (2)

When it useful:

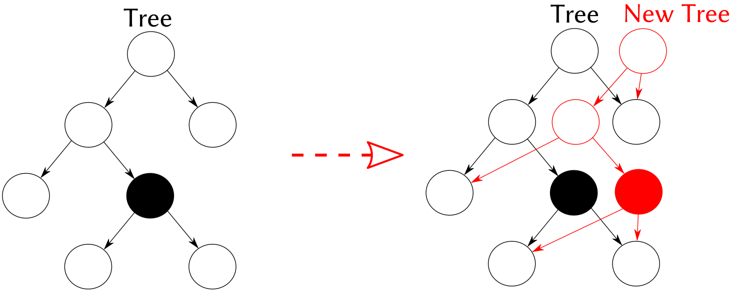
Example: Shared term

```
do_two_lists (x1 : x1s) (x2 : x2s) = ...
do_two_lists xs (apply_head f xs)
```

No copying, if not needed. The same elements (except heads) are used in both lists.

Advanced Example: Tree

Suppose we want to change black element of the illustrated tree



Notice, here we need to copy some elements, but the number of copied elements is proportional to the tree's height. Very small in general!

Laziness

Short-Circuit Evaluation

Liar's paradox

```
bottom :: Bool
bottom = not bottom
```

What will happen here?

Liar's paradox

```
if 2 + 2 == 4 || bottom then "Yes!" else "The Matrix is Broken"
```

This feature is called *short-circuit evaluation*. Can be found in many languages.

Laziness

However, in Haskell this applies to **all** expressions, not just functions on booleans. It's called *Laziness*.

Example: Laziness

```
const42 x = 42
const42 bottom
```

(Almost) Every term is evaluated (if evaluated at all) at the very last moment when its value is actually required.

Laziness: Pros and Cons

Pros:

- If some values are not required, they will not be computed at all!
- Parallel computations on multiple threads are easy to implement (will show later).
- One more cool feature.

Cons:

- Sometimes a long stack of unevaluated terms appears.

However, it is possible to change this behaviour and force computation. Not lazy function is called *strict*. Moreover, Haskell compiler does some optimization by itself.

Strictness

There is one built-in strict function with the following semantics.

Built-in strict function

```
seq :: a -> b -> b
seq a b = b
```

This function forces evaluation of it's first argument.

GHCi session

```
let x = 1 + 1 :: Int
:sprint x
seq x ()
:sprint x
```

When laziness is actually useful

What is happening here?

Example: Laziness in full glory

$$\text{ones} = 1 : \text{ones}$$
$$\text{numbers} = 1 : \mathbf{map}\ (+1)\ \text{numbers}$$

Check your understanding by running the following.

Example: Extracting values

$$\mathbf{take}\ 10\ \text{numbers}$$

Exercises

Parallelism

Multithreading

Three main features of Haskell, namely *immutability*, *purity* and *laziness*, allow us to safely use the following function:

Built-in multithreading function

```
par :: a -> b -> b
par a b = b
```

This function is semantically identical to **seq**. However, it sparks a new thread for evaluation of its first argument.

Example: Parallelism

```
par x (f x)
```

For example, we can evaluate function and it's argument **at the same time**.

The End