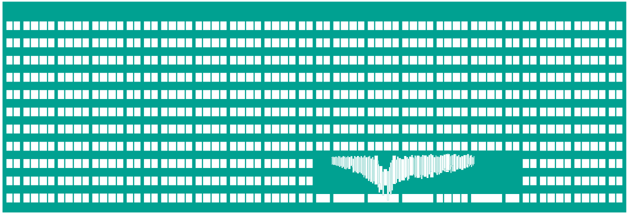
VŠB TECHNICKÁ UNIVERZITA OSTRAVA

VSB TECHNICAL UNIVERSITY OF OSTRAVA



www.vsb.cz

Conclusion - topics that were not properly addressed, yet.

behalek.cs.vsb.cz/wiki/Functional_Programming

Marek Běhálek

VSB – Technical University of Ostrava

marek.behalek@vsb.cz

December 5, 2022

VSB TECHNICAL UNIVERSITY OF OSTRAVA | FACULTY OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE | DEPARTMENT OF COMPUTER SCIENCE

Reasoning about programs

- OK, functional languages have mathematical background, but is this any good for me (I am a programmer, not mathematician;-)?
- Formal definition of language semantic allows to prove program's properties → more trustworthy then just some *tests*.
    - Emended systems, automotive, ...
    - Tools: Formal proof management system Coq `https://coq.inria.fr/` → based on richly-typed functional programming language *Gallina*
        - CompCert - verification of C programs
        - Extract certified programs to Haskell
- Mathematical induction (informally)
    - Prove for $\mathbf{n} = \mathbf{0}$ (base case)
    - On assumption that it holds for **n**, prove that it holds for **n+1**
- Principle of structural induction for lists – we want to prove property **P**
    - Base case – prove **P** for [] outright.
    - Prove **P** for `(x:xs)` on assumption that **P** holds for `xs`.

Reasoning about programs - Example (1)

- We want to prove: `(xs ++ ys) ++ zs = xs ++ (ys ++ zs)`
- We start with *equations* from the source code.

```
[] ++ ys     = ys                -- ++.1
(x:xs) ++ ys = x: (xs ++ ys)     -- ++.2
```

- Now we can start proving (using mathematical induction).

```
-- a) [] => xs
([] ++ ys) ++ zs
= ys ++ zs          -- ++.1
= [] ++ (ys ++ zs) -- ++.1
-- b) (x:xs) => xs
((x:xs)++ys)++zs
= x:(xs++ys)++zs    -- ++.2
= x:((xs++ys)++zs)  -- ++.2
= x:(xs++(ys++zs))  -- assumption
= (x:xs)++(ys++zs)  -- ++.2
```

Reasoning about programs - Example (2)

- Better example: `(length (xs++ys) = length xs + length ys`
- We start with *equations* from the source code.

```
length []     = 0               --len.1
length (_:xs) = 1 + length xs   --len.2
```

- Now we can start proving (using mathematical induction).

```
-- a) [] => xs
length ([] ++ ys)
= length ys               -- ++.1
= 0 + length ys           -- + zero element
= length [] + length ys -- len.1
-- b) (x:xs) => xs
length ((x:xs) ++ ys)
= length (x:(xs++ys)       -- ++.2
= 1 + length (xs++ys)      -- len.2
= 1 + (length xs + length ys)  -- assumption)
= (1 + length xs) + length ys  -- associativity of +
= length (x:xs) + length ys    -- len.2
```

Lambda calculus I

- $\lambda - calculus$ is a formal system in mathematical logic for expressing computation based on function abstraction and application using variable binding and substitution *(wiki)*.
- It was invented in 1930s by Alonzo Church.
- Universal model of computation, *as good as* Turing machine $\rightarrow$ all that can be compute by Turing machine can be expressed in $\lambda - calculus$ $\rightarrow$ roughly, this corresponds to problems that can be solved by a computer.
- Omitting many details, theoretical background for all functional programming languages.
  - Originally $\lambda - calculus$ is *untyped* $\rightarrow$ in programming we need types $\rightarrow$ not that easy to add them.

Lambda calculus - simplified definition

- Syntax (how it is written) - a lambda term is:
  - $x, y, z...$ - variables, representing a parameter or mathematical/logical value.
  - $(\lambda x.M)$ - abstraction, M is a lambda term, the variable x becomes bound in the expression.
  - (MN) - application, applying a function to an argument. M and N are lambda terms.
- Semantics (how to compute it)
  - $\alpha - conversion : (\lambda x.M[x]) \rightarrow (\lambda y.M[y])$ - renaming the bound variables in the expression. Used to avoid name collisions.
  - $\beta - reduction : ((\lambda x.M)E) \rightarrow (M[x := E])$ -replacing the bound variables with the argument expression in the body of the abstraction *(this really moves forward the computation)*.
  - $\eta - reduction : ((\lambda x.fx) \rightarrow f$ - expresses the idea of extensionality (two functions are the same if and only if they give the same result for all arguments).

Lambda calculus - normal form

- Redex - **Red**ucible **Ex**pression - expression that can be reduced with defined rules.
  - $\alpha - redex, \beta - redex$
- Church–Rosser theorem - when applying reduction rules to terms, the ordering in which the reductions are chosen does not make a difference to the eventual result.
- In other words, if there are two distinct reductions or sequences of reductions that can be applied to the same term, then there exists a term that is reachable from both results.
- **Normal form** - expression that contains no $\beta - redex$.
  - `42, (2, "hello"), \x -> (x + 1)`
- Haskell uses **weak head normal form** - stops when *head* is a lambda abstraction or a data constructor.
  - `(1 + 1, 2 + 2), \x -> 2 + 2,'h' : ("e" ++ "llo").`
- The question that remains is, how do we get the weak head normal form?

Lazy evaluation - what are our option for evaluation strategies?

- When choosing an evaluation strategy for expressions in languages like Haskell, what are key factors?
  - Evaluation order - which reductions are performed first (inner-most, outer-most)
  - How do we pass parameters to a function - by *value*, by name, by reference, by need...
- Function f is strict when and only when: $f \perp = \perp$
- *Strict evaluation* - function's arguments are evaluated completely before the function is applied.
  - innermost reduction, eager evaluation or greedy evaluation
  - Sometime also *Call by value* - it requires strict evaluation, arguments are passed as evaluated values.
  - It is used by most programming languages: Java, C#, F#, OCalm, Scheme...
- Non-strict evaluation - a function may return a result before all of its arguments are fully evaluated.
  - outer-most reduction, normal order evaluation (does not evaluate any of the arguments until they are needed in the body of the function).

Lazy evaluation (1)

- Lazy evaluation - When we are lazy enough, to call our evaluation lazy?
  - Sub-expressions will be evaluated only when they are needed for in evaluation.
  - If they are evaluated, they are evaluated only once.
- In pure functional languages, if we use outer-most reduction, we are doing normal order evaluation $\rightarrow$ only needed sub-expressions are evaluated, only needed arguments are evaluated.
- In pure functional languages, to be lazy enough, all we need is some clever way, how to pass arguments $\rightarrow$ **call by need**.
  - Used in Haskell, option in OCalm, Scheme, some languages simulate lazy behaviour for some sub-systems.
- In pure functional languages, the terms lazy evaluation, call by need, or non-strict evaluation mean the same *thing*.

Lazy evaluation (2)

- Eager evaluation

```
square(1+2)
square(3)
3*3
9
```

- Lazy evaluation

```
square(1+2)
let x = 1+2 in x*x
let x = 3 in x*x
3*3
9
```

Advantages of Lazy evaluation

- If an expression has a normal form, it will be reached by lazy evaluation strategy (theory nonsense:-).
- It allows to use new concepts, like infinite structures or functions → new way how to solve a problem (i still wont use it:-).

  ```
  fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
  ```

- It is useful when processing (large) data (LINQ, Apache Spark,..)
  - Consider following example:

    ```
    map (\x->x^4) (concat (map (\x->[1..x]) [1..10]))
    ```

  - Will be the intermediate results constructed?
  - In fact, we are continually getting items from the final list!
  - How the equivalent in C++ will look like?
    - We need to sacrifice code clarity, or all intermediate results will be computed before we get some result.

Thank you for your attention

Marek Běhálek

VSB – Technical University of Ostrava

marek.behalek@vsb.cz

December 5, 2022

VSB TECHNICAL UNIVERSITY OF OSTRAVA | FACULTY OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE | DEPARTMENT OF COMPUTER SCIENCE