

Fortgeschrittene funktionale Programmierung

LVA 185.A05, VU 2.0, ECTS 3.0
SS 2021

(Stand: 04.03.2021)

Jens Knoop



Technische Universität Wien
Information Systems Engineering
Compilers and Languages



Lecture 1

Part I: Motivation

- Chapter 1: Why Functional Programming Matters
 - + Chap. 1.5: Recommended Reading: Basic, Advanced

Part II: Programming Principles

- Chapter 2: Programming with Streams
 - + Chap. 2.7: Recommended Reading: Basic, Advanced

Outline in more Detail (1)

Part I: Motivation

- Chap. 1: Why Functional Programming Matters
 - 1.1 Reconsidering Folk Knowledge
 - 1.2 Glueing Functions Together: Higher-Order Functions
 - 1.3 Glueing Programs Together: Lazy Evaluation
 - 1.3.1 Square Root Computation
 - 1.3.2 Numerical Integration
 - 1.3.3 Numerical Differentiation
 - 1.4 Summary, Looking ahead
 - 1.5 References, Further Reading

Part II: Programming Principles

- Chap. 2: Programming with Streams
 - 2.1 Streams, Stream Generators
 - 2.2 The Generate-Prune Pattern
 - 2.2.1 The Generate-Select/Filter Pattern
 - 2.2.2 The Generate-Transform Pattern
 - 2.3.3 Pattern Combinations
 - 2.2.4 Summary

Outline in more Detail (2)

Part II: Programming Principles

- Chap. 2: Programming with Streams (cont'd)

- 2.3 Boosting Performance

- 2.3.1 Motivation

- 2.3.2 Stream Programming combined with Münchhausen Principle

- 2.3.3 Stream Programming combined with Memoization

- 2.3.4 Summary

- 2.4 Stream Diagrams

- 2.5 Pitfalls, Remedies

- 2.5.1 Livelocks, Lazy Patterns

- 2.5.2 Lifting, Undecidability

- 2.5.3 Termination, Domain-specific Knowledge

- 2.6 Summary, Looking ahead

- 2.7 References, Further Reading

Sometimes, the elegant implementation is a function.
Not a method. Not a class. Not a framework.
Just a function.
John Carmack

...quoted from: [Yaron Minsky](#). [OCaml for the Masses](#). Communications of the ACM 54(11):53-58, 2011 (...why the next language you learn should be functional.)

Functional Programming

...owes its name to the fact that programs are composed of only **functions**:

- The **main program** is itself a function.
- It accepts the program's input as its arguments and delivers the program's output as its result.
- It is defined in terms of other functions, which themselves are defined in terms of still more functions (eventually by primitive functions).

...why should functional programming matter?

Chapter 1

Why Functional Programming Matters

“Why Functional Programming Matters”

...the title of a now classical [position statement](#) and [plea](#) for functional programming by John Hughes he denoted

- ...an attempt to demonstrate to the “real world” that functional programming is vitally important, and also to help functional programmers exploit its advantages to the full by making it clear what those advantages are.

The statement is based on a 1984 internal memo at Chalmers University, and has slightly revised been published in:

- Computer Journal 32(2):98-107, 1989.
- Research Topics in Functional Programming. David Turner (Ed.), Addison-Wesley, 1990.
- <http://www.cs.chalmers.se/~rjmh/Papers/whyfp.html>

Chapter 1.1

Reconsidering Folk Knowledge

Lecture 1

Detailed
Outline

Chap. 1

1.1

1.2

1.3

1.4

1.5

Chap. 2

Concludin
Note

Assignme

Folk Knowledge

...on the **benefits** of **functional programming**:

- Functional programs are free of assignments & side-effects.
 - Function calls have no effect except of computing their result.
- ⇒ **Functional programs are thus free of a major source of bugs!**
- The evaluation order of expressions is irrelevant, expressions can be evaluated any time.
 - Programmers are free from specifying the control flow explicitly.
 - Expressions can be replaced by their value and vice versa; programs are **referentially transparent**.
- ⇒ **Functional programs are thus easier to cope with mathematically (e.g., for proving them correct)!**

Folk Knowledge (cont'd)

...functional programs are

- a magnitude of order smaller than conventional programs

⇒ Functional programmers are thus much more productive!

Regarding evidence consider e.g.:

“Higher-level languages are more productive, says Sergio Antoy, Textronics Professor of computer science at Oregon’s Portland State University, in the sense that they require fewer lines of code. A program written in machine language, for instance, might require 100 pages of code covering every little detail, whereas the same program might take only 50 pages in C and 25 in Java, as the level of abstraction increases. In a functional language, Antoy says, the same task might be accomplished in only 15 pages.”

quoted from: Neil Savage. Using Functions for Easier Programming.
Communications of the ACM 61(5):29-30, 2018.

Overall, however,

...the features attributed to functional programming by 'folk knowledge' do not really explain the power of functional programming; in particular, they do not provide

- any help in exploiting the power of functional languages. (programs, e.g., cannot be written which are particularly lacking in assignment statements, or which are particularly referentially transparent).
- a yardstick of program quality, nothing a functional programmer should strive for when writing a program.

What we need

...is a positive characterization of what

1. makes the vital nature of functional programming and its strengths.
2. makes a 'good' functional program a functional programmer should strive for.

John Hughes' Thesis

The **expressiveness** of a language

- depends much on the **power** of the **concepts** and **primitives** allowing to **glue** solutions of subproblems to the solution of an overall problem, i.e., its power to support a **modular program design** (as an example, consider the making of a chair).

Functional programming provides two new, especially powerful kinds of **glue**:

- ▶ **Higher-order functions** (\rightsquigarrow **glueing functions** together)
- ▶ **Lazy evaluation** (\rightsquigarrow **glueing programs** together)

John Hughes' Thesis (cont'd)

The **vital nature** of **functional programming** and its **strengths**

- result from the two new kinds of **glue**, which enable **conceptually new opportunities for modularization** and re-use (beyond the more technical ones of lexical scoping, separate compilation, etc.), and making them more easily to achieve.

Striving for **'good' functional programs** means

- functional programmers shall strive for programs which are **smaller, simpler, more general**.

Functional programmers shall assume this can be achieved by **modularization** using as **glue**

- ▶ **higher-order functions**
- ▶ **lazy evaluation**

Chapter 1.2

Glueing Functions Together: Higher-Order Functions

Lecture 1

Detailed
Outline

Chap. 1

1.1

1.2

1.3

1.4

1.5

Chap. 2

Concludin
Note

Assignme

Preparing the Setting

...following the position statement, program examples will be presented in **MirandaTM** syntax:

► Lists

`listof X ::= nil | cons X (listof X)`

► Abbreviations (for convenience)

`[]` **means** `nil`

`[1]` **means** `cons 1 nil`

`[1,2,3]` **means** `cons 1 (cons 2 (cons 3 nil))`

► A simple function: Adding the elements of a list

`sum nil` `= 0`

`sum (cons num list)` `= num + sum list`

Note

...only the **framed parts** are specific to computing a **sum**:

$$\begin{array}{rcl} \text{sum nil} & = & \begin{array}{c} +---+ \\ | \text{ 0 } | \\ +---+ \end{array} \\ \\ \text{sum (cons num list)} & = & \begin{array}{c} +---+ \\ \text{num } | \text{ + } | \text{ sum list} \\ +---+ \end{array} \end{array}$$

This observation suggests that computing a sum of values can be modularly decomposed by properly combining a

- general recursion pattern (called **reduce**)
- set of more specific operations (in the example: **+**, **0**)

Exploiting the Observation

Exam. 1: Adding the elements of a list

```
sum = reduce add 0
      where add x y = x+y
```

The example allows to conclude the definition of the **higher-order function reduce** almost immediately:

```
(reduce f x) nil           = x
(reduce f x) (cons a l) = f a ((reduce f x) l)
```

Recalled for convenience:

```
sum nil           = +----+
                   | 0 |
                   +----+

sum (cons num list) = num +----+ sum list
                   | + |
                   +----+
```

Immediate Benefit: Re-use of the HoF reduce

...without any further programming effort we obtain implementations of many other functions, e.g.:

Exam. 2: Multiplying the elements of a list

```
product = reduce mult 1
           where mult x y = x*y
```

Exam. 3: Test, if *some* element of a list equals 'true'

```
anytrue = reduce or false
```

Exam. 4: Test, if *all* elements of a list equal 'true'

```
alltrue = reduce and true
```

Exam. 5: Concatenating two lists

```
append a b = reduce cons b a
```

Exam. 6: Doubling each element of a list

```
doubleall = reduce doubleandcons nil
           where doubleandcons num list
                 = cons (2*num) list
```

Lecture 1

Detailed
Outline

Chap. 1

1.1

1.2

1.3

1.4

1.5

Chap. 2

Concludin
Note

Assignme

How does it work? (1)

Intuitively, the effect of applying `(reduce f a)` to a list is to replace in the list all occurrences of

- `cons` by `f`
- `nil` by `a`

For illustration reconsider selected examples in more detail:

Exam.1: Adding the elements of a list

```
sum [2,3,5] ->> sum (cons 2 (cons 3 (cons 5 nil)))  
->> reduce add 0 (cons 2 (cons 3 (cons 5 nil)))  
->> (add 2 (add 3 (add 5 0)))  
->> 10
```

Exam. 2: Multiplying the elements of a list

```
product [2,3,5] ->> product (cons 2 (cons 3 (cons 5 nil)))  
->> reduce mult 1 (cons 2 (cons 3 (cons 5 nil)))  
->> (mult 2 (mult 3 (mult 5 1)))  
->> 30
```

How does it work? (2)

Exam. 5: Concatenating two lists

Note: The expression `reduce cons nil` is the identity on lists. Exploiting this fact suggests the implementation of `append` in the form of: `append a b = reduce cons b a`

```
append [1,2] [3,4]
->> {expanding [1,2] }
->> append (cons 1 (cons 2 nil)) [3,4]
->> {expanding append}
->> reduce cons [3,4] (cons 1 (cons 2 nil))
->> {replacing cons by cons and nil by [3,4] }
      (cons 1 (cons 2 [3,4]))
->> {expanding [3,4] }
      (cons 1 (cons 2 (cons 3 (cons 4 nil))))
->> {syntactically sugaring the list expression}
      [1,2,3,4]
```

Lecture 1

Detailed
Outline

Chap. 1

1.1

1.2

1.3

1.4

1.5

Chap. 2

Concluding
Note

Assignment

How does it work? (3)

Exam. 6: Doubling each element of a list

```
doubleall = reduce doubleandcons nil
  where doubleandcons num list
        = cons (2*num) list
```

Note that `doubleandcons` can stepwise be modularized, too:

1. `doubleandcons = fandcons double`
 where `fandcons f el list = cons (f el) list`
 `double n = 2*n`
2. `fandcons f = cons . f`
 with `'.'` sequential composition of functions: $(g \cdot h) k = g (h k)$

How does it work? (4)

...the correctness of the two modularization steps for `doubleandcons` follows from:

```
fandcons f el = (cons . f) el  
              = cons (f el)
```

which yields as desired:

```
fandcons f el list = cons (f el) list
```


How does it work? (5)

Putting the parts together, we obtain the following version of `doubleall` based on `reduce`:

Exam. 6.1: Doubling each element of a list

```
doubleall = reduce (cons . double) nil
```

Introducing the higher-order function `map`, which applies a function `f` to every element of a list:

```
map f = reduce (cons . f) nil
```

we eventually get the final version of `doubleall`, which is indirectly based on `reduce` via `map`:

Exam. 6.2: Doubling each element of a list

```
doubleall = map double  
           where double n = 2*n
```

Summing up

By **decomposing (modularizing)** and representing a simple function (**sum** in the example) as a combination of

- a **higher-order function** and
- some **simple specific functions** as arguments

we obtained a **program frame** (**reduce**) that allows us to implement many functions on lists essentially **without any further programming effort!**

This is especially useful for **complex data structures** as we are going to show next!

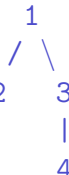
Generalizing the Approach

...to (more) **complex data structures** using **trees** as example:

`treeof X ::= node X (listof (treeof X))`

A value of type `(treeof X)`:

```
node 1
  (cons (node 2 nil)
        (cons (node 3 (cons (node 4 nil) nil))
              nil))
```



```
      1
     / \
    2   3
      |
      4
```

The Higher-order Function `redtree`

...following the spirit of `reduce` on lists we introduce a **higher-order function** `redtree` (short for 'reduce tree') on trees:

```
redtree f g a (node label subtrees)
= f label (redtree' f g a subtrees)
where
  redtree' f g a (cons subtree rest)
    = g (redtree f g a subtree) (redtree' f g a rest)
  redtree' f g a nil = a
```

Note: `redtree` takes 3 arguments `f`, `g`, `a` (and a tree value):

- `f` to replace occurrences of `node` with
- `g` to replace occurrences of `cons` with
- `a` to replace occurrences of `nil` with

in tree values.

Applications of redtree (1)

Like `reduce` allows to implement many functions on list without any effort, `redtree` allows this on trees as we demonstrate by three examples:

Exam. 7: Adding the labels of the leaves of a tree.

Exam. 8: Generating the list of labels occurring in a tree.

Exam. 9: A function `maptree` on trees which applies a function `f` to every label of a tree, i.e., `maptree` is the analogue of the function `map` on lists.

As a `running example`, we consider the tree value below:

```
node 1
  (cons (node 2 nil)
        (cons (node 3 (cons (node 4 nil) nil))
              nil))
```

```
      1
     / \
    2   3
       |
       4
```

Applications of redtree (2)

Exam. 7: Adding the labels of the leaves of a tree

```
sumtree = redtree add add 0
```

```
sumtree (node 1  
        (cons (node 2 nil)  
              (cons (node 3 (cons (node 4 nil) nil))  
                    nil))))
```

```
->> redtree add add 0  
      (node 1  
        (cons (node 2 nil)  
              (cons (node 3 (cons (node 4 nil) nil))  
                    nil))))
```

```
->> (add 1  
      (add (add 2 0 )  
            (add (add 3 (add (add 4 0 ) 0 )  
                  0 ))))
```

```
->> 10
```

Applications of redtree (3)

Exam. 8: Generating the list of labels occurring in a tree

```
labels = redtree cons append nil
```

```
labels (node 1
        (cons (node 2 nil)
              (cons (node 3 (cons (node 4 nil) nil))
                    nil))))
```

```
->> redtree cons append nil
      (node 1
        (cons (node 2 nil)
              (cons (node 3 (cons (node 4 nil) nil))
                    nil))))
```

```
->> (cons 1
         (app'd (cons 2 nil)
               (app'd (cons 3 (app'd (cons 4 nil) nil))
                     nil))))
```

```
->> [1,2,3,4]
```

Applications of redtree (4)

Exam. 9: A function `maptree` which applies a function `f` to every label of a tree

```
maptree f = redtree (node . f) cons nil
```

```
maptree double (node 1
                  (cons (node 2 nil)
                        (cons (node 3 (cons (node 4 nil) nil))
                              nil))))
```

```
->> redtree (node . double) cons nil
      (node 1
        (cons (node 2 nil)
              (cons (node 3 (cons (node 4 nil) nil))
                    nil))))
```

```
->> ...
```

```
->> (node 2
      (cons (node 4 nil)
            (cons (node 6 (cons (node 8 nil) nil))
                  nil))))
```

Lecture 1

Detailed
Outline

Chap. 1

1.1

1.2

1.3

1.4

1.5

Chap. 2

Concludin
Note

Assignme

Summing up (1)

The **simplicity** and **elegance** of the preceding examples materializes from combining

- a **higher-order function** and
- a **specific specializing function**

Once the **higher-order function** is implemented, lots of

- functions can be implemented essentially effort-less!

Summing up (2)

Lesson learnt:

- Whenever a new data type is defined (like lists, trees,...), implement first a **higher-order function** allowing to process values of this type (e.g., visiting each component of a structured data value such as nodes in a graph or tree).

Benefits:

- Manipulating elements of this data type becomes easy; knowledge about this data type is locally concentrated and encapsulated.

Look & feel:

- Whenever a new data structure demands a new control structure, then this control structure can easily be added following the methodology used above (note that this resembles to some extent the concepts known from conventional extensible languages).

Chapter 1.3

Glueing Programs Together: Lazy Evaluation

Lecture 1

Detailed
Outline

Chap. 1

1.1

1.2

1.3

1.3.1

1.3.2

1.3.3

1.4

1.5

Chap. 2

Concludin
Note

Assignme

Preparing the Setting

- We consider a **function** from its **input** to its **output** a **complete functional program**.
- If **f** and **g** are complete functional programs, then also their composition **(g . f)** is a complete functional program.

Applied to input **in**, **(g . f)** yields the output **out**:

$$\text{out} = (\text{g} . \text{f}) \text{ in} = \text{g} (\text{f in})$$

Task: Implementing the **communication** between **f** and **g**:
E.g., using **temporary files** as **conventional glue**.

Possible problems:

1. Temporary files could get too large and exceed the available storage capacity.
2. **f** might not terminate.

Lazy Evaluation

...as **functional glue** allows a more elegant approach by **decomposing** a program into a

- generator
- selector

component/module **glued** together by **functional composition** and synchronized by

- **lazy evaluation**

ensuring:

- The **generator** ‘runs as little as possible’ till it is terminated by the **selector**.

In the following

...three examples for illustrating this **modularization strategy**:

1. Square root computation
2. Numerical integration
3. Numerical differentiation

Note, only the first example will be considered in full detail here (see complete course notes for the other two examples).

Chapter 1.3.1

Square Root Computation

Lecture 1

Detailed
Outline

Chap. 1

1.1

1.2

1.3

1.3.1

1.3.2

1.3.3

1.4

1.5

Chap. 2

Concluding
Note

Assignme

The Newton-Raphson Approach

...for square root computation.

Given: N , a positive number

Sought: $\text{squareRoot}(N)$, the square root of N

Iteration formula: $a(n+1) = (a(n) + N/a(n)) / 2$

Justification: If for some initial approximation $a(0)$, the sequence of approximations converges to some limit a , $a \neq 0$, a equals the square root of N . Consider:

$$\begin{array}{llll} (a + N/a) / 2 & = & a & | \quad *2 \\ \Leftrightarrow a + N/a & = & 2a & | \quad -a \\ \Leftrightarrow N/a & = & a & | \quad *a \\ \Leftrightarrow N & = & a*a & | \quad \text{sqr} \\ \Leftrightarrow \text{squareRoot}(N) & = & a & \end{array}$$

A Typical Imperative Implementation

...realizing this approach (here in [Fortran](#)):

```
C      N is called ZN here so that it has
C      the right type
      X = A0
      Y = A0 + 2.*EPS
C      The value of Y does not matter so long
C      as ABS(X-Y).GT. EPS
100    IF (ABS(X-Y).LE. EPS) GOTO 200
      Y = X
      X = (X + ZN/X) / 2.
      GOTO 100
200    CONTINUE
C      The square root of ZN is now in X
```

⇒ this is essentially a [monolithic](#), not decomposable program.

Developing now a Modular Functional Version

First, we define function `next`, which computes the `next approximation` from the previous one:

$$\text{next } N \ x = (x + N/x) / 2$$

Second, we define function `g`:

$$g = \text{next } N$$

This leaves us with computing the (possibly infinite) sequence of approximations:

$$[a_0, g \ a_0, g \ (g \ a_0), g \ (g \ (g \ a_0)), \dots]$$

which is equivalent to:

$$[a_0, \text{next } N \ a_0, \text{next } N \ (\text{next } N \ a_0), \\ \text{next } N \ (\text{next } N \ (\text{next } N \ a_0)), \dots]$$

Writing a Generator

...applied to some function `f` and some initial value `a`, function `repeat` computes the (possibly infinite) sequence of values resulting from repeatedly applying `f` to `a`; `repeat` will be the `generator` component in this example:

Generator A:

```
repeat f a = cons a (repeat f (f a))
```

Note:

- Applying `repeat` to the arguments `g` and `a0` yields the desired sequence of approximations:

```
repeat g a0
```

```
->> repeat (next N) a0
```

```
->> [a0, next N a0, next N (next N a0),  
      next N (next N (next N a0)), ...]
```

- Evaluating `repeat g a0` does not terminate!

Writing a Selector

...applied to some value $\text{eps} > 0$ and some list xs , function `within` picks the first element of xs , which differs at most by eps from its preceding element; `within` will be the `selector` in this example allowing to tame the `looping evaluation` of the `generator`:

Selector A:

```
within eps (cons a (cons b rest))  
  = b,                                if  $\text{abs}(a-b) \leq \text{eps}$   
  = within eps (cons b rest), otherwise
```

Glueing together Generator and Selector

...to obtain the final program.

Glueing together Generator A and Selector A:

$$\text{sqrt } N \text{ eps } a0 = \underbrace{\text{within eps}}_{\text{Selector A}} \underbrace{(\text{repeat (next N) } a0)}_{\text{Generator A}}$$

Effect: The composition of Generator A and Selector A stops approximating the value of the square root of N once the latest two approximations of this value differ at most by $\text{eps} > 0$, used here as indication of sufficient precision of the currently reached approximation.

Summing up

The **functional version** of the program approximating the square root of a number is unlike the imperative one **not monolithic** but composed of two **modules** running in perfect **synchronization**.

Modules:

- **Generator program/module**: **repeat**
[a0, g a0, g(g a0), g(g(g a0)), ...]
...potentially infinite, no pre-defined limit of length.
- **Selector program/module**: **within**
 $g^i \text{ a0 with } \text{abs}(g^i \text{ a0} - g^{i+1} \text{ a0}) \leq \text{eps}$
...**lazy evaluation** ensures that the selector function is applied eventually \Rightarrow **termination!**

Synchronized by:

- **Lazy evaluation**

...**overcoming** the problem of the **looping** generator for free.

Immediate Benefit: Modules are Re-usable

...we will demonstrate that

- Generator A
- Selector A

can indeed easily be re-used, and therefore be considered modules.

We are going to start re-using Generator A with a new selector.

Re-using Generator A with a new Selector

Consider a new criterion for termination:

- Instead of awaiting the difference of successive approximations to approach zero (i.e., $\leq \text{eps}$), await their ratio to approach one (i.e., $\leq 1+\text{eps}$).

Selector B:

```
relative eps (cons a (cons b rest))  
  = b,           if abs(a-b) <= eps * abs b  
  = relative eps (cons b rest), otherwise
```

Glueing together (old) Generator A and (new) Selector B:

```
relativesqrt N eps a0  
  = relative eps (repeat (next N) a0)  
    Selector B      Generator A
```


Dually: Re-using Selectors A and B

...with new **generators**.

Dually to re-using a **generator** module as in the previous example, also the **selector** modules can be re-used. To this end we consider two further examples requiring new generators:

- Numerical integration
- Numerical differentiation

Chapter 1.3.2

Numerical Integration

Lecture 1

Detailed
Outline

Chap. 1

1.1

1.2

1.3

1.3.1

1.3.2

1.3.3

1.4

1.5

Chap. 2

Concluding
Note

Assignme

Numerical Integration

Given: A real valued function f of one real argument; two end-points a and b of an interval

Sought: The area under f between a and b

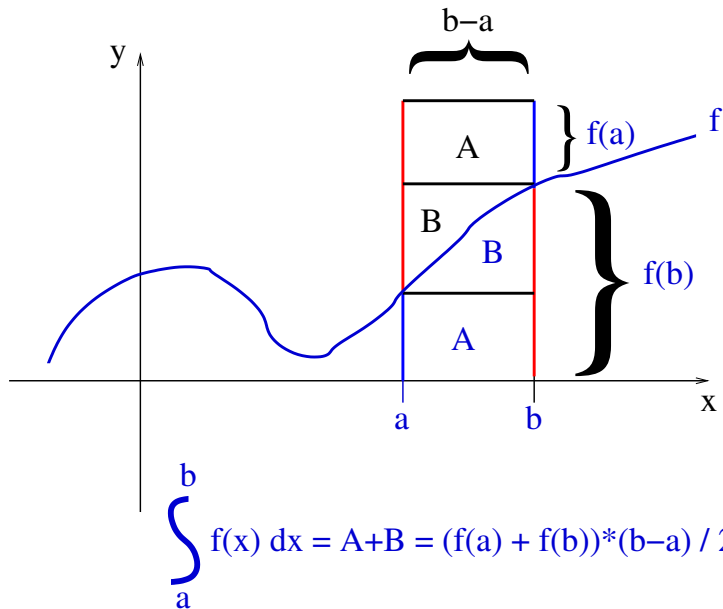
Simple Solution:

...assuming that the function f is roughly linear between a and b .

$$\text{easyintegrate } f \ a \ b = (f \ a + f \ b) * (b-a) / 2$$

Note: The results of `easyintegrate` will be precise enough for practical usages at most for very small intervals. Therefore, we will develop an iterative approximation strategy based on the idea underlying the simple solution.

Illustrating the Essence of easyintegrate



Writing a Generator

Iterative Approximation Strategy

- Halve the interval, compute the areas for both sub-intervals according to the previous formula, and add the two results.
- Continue the previous step repeatedly.

The function `integrate` realizes this strategy:

Generator B:

```
integrate f a b
= cons (easyintegrate f a b)
      map addpair (zip (integrate f a mid)
                      (integrate f mid b)))
  where mid = (a+b)/2
```

where

```
zip (cons a s) (cons b t) = cons (pair a b) (zip s t)
```

Lecture 1

Detailed
Outline

Chap. 1

1.1

1.2

1.3

1.3.1

1.3.2

1.3.3

1.4

1.5

Chap. 2

Concludin
Note

Assignme

Re-using Selectors A, B with Generator B

Note, evaluating the new **generator** term `integrate f a b` does not terminate!

However, the evaluation can be tamed by **glueing** it together with any of the previously defined two **selectors** thereby re-using these selectors and computing `integrate f a b` up to some accuracy.

Re-using **Selectors A, B** for new **generator/selector** combinations:

- * $\underbrace{\text{within eps}}_{\text{Selector A}} \underbrace{(\text{integrate f a b})}_{\text{Generator B}}$
- * $\underbrace{\text{relative eps}}_{\text{Selector B}} \underbrace{(\text{integrate f a b})}_{\text{Generator B}}$

Summing up

- New **generator** module: **integrate**
...looping, no limit for the length of the generated list
- Two old **selector** modules: **within**, **relative**
...picking a particular element of a list.
- Their combination **synchronized** by **lazy evaluation**
...ensuring the selector function is eventually successfully applied \Rightarrow **termination!**

Note, the two **selector** modules **A** and **B** picking the solution

- from the stream of approximate solutions could be re-used from the square root example w/out any change.

In total, we now have **2 generators** and **2 selectors**, which can be **glued** together in any combination. For any combination, their proper **synchronization** (and termination) is ensured by

- ▶ **lazy evaluation!**

Chapter 1.3.3

Numerical Differentiation

Lecture 1

Detailed
Outline

Chap. 1

1.1

1.2

1.3

1.3.1

1.3.2

1.3.3

1.4

1.5

Chap. 2

Concludin
Note

Assignme

Numerical Differentiation

Given: A real valued function f of one real argument; a point x

Sought: The slope of f at point x

Simple Solution:

...assuming that the function f does not 'curve much' between x and $x+h$.

$$\text{easydiff } f \ x \ h = (f \ (x+h) - f \ x) / h$$

Note: The results of `easydiff` will be precise enough for practical usages at most for very small values of h . Therefore, we will develop an iterative approximation strategy based on the idea underlying the simple solution.

Lecture 1

Detailed
Outline

Chap. 1

1.1

1.2

1.3

1.3.1

1.3.2

1.3.3

1.4

1.5

Chap. 2

Concludin
Note

Assignme

Writing a Generator/Selector Combination

Along the lines of the example on numerical integration, we implement a new **generator** computing a sequence of approximations getting more and more accurate by interval halving:

Generator C:

```
differentiate h0 f x
    = map (easydiff f x) (repeat halve h0)
halve x = x/2
```

As before, the new **generator** can now be **glued** together with any of the **selectors** we defined so far picking a sufficiently accurate approximation, e.g.:

Glueing together **Generator C** and **Selector A**:

```
within eps (differentiate h0 f x)
  Selector A    Generator C
```

Summing up

All three examples (square root computation, numerical integration, numerical differentiation) enjoy a **common composition pattern**, namely using and combining a

- **generator** (looping!)
- **selector**

synchronized by

- ▶ **lazy evaluation**

ensuring termination for free.

This **composition/modularization** principle can be further generalized to combining

- **generators** with **selectors**, **filters**, and **transformers**

as illustrated in more detail in **Chapter 2**.

Chapter 1.4

Summary, Looking ahead

Lecture 1

Detailed
Outline

Chap. 1

1.1

1.2

1.3

1.4

1.5

Chap. 2

Concludin
Note

Assignme

Starting Point

...of John Hughes:

- ▶ **Modularity** is the key to programming in the large.

Findings from reconsidering folk knowledge:

- Just modules (i.e., the capability of decomposing a problem) do not suffice.
- The benefit of modularly decomposing a problem into subproblems depends much on the capabilities for **glueing** together the modules to larger programs.

Hence

- ▶ The **availability** of **proper glue** is **essential**!

Finding

Functional programming offers two new kinds of glue:

1. Higher-order functions (glueing functions)
2. Lazy evaluation (glueing programs)

Higher-order functions and lazy evaluation allow substantially

- new exciting modular compositions of programs (by offering elegant and powerful kinds of glue for composing moduls) as given evidence in this chapter by an array of simple, yet striking examples.

Overall, it is the superiority of these 2 kinds of glue allowing

- functional programs to be written so concisely and elegantly (rather than their freedom of assignments, etc.).

Recommendation

...when writing a program, a **functional programmer** shall

- strive for adequate **modularization** and **generalization** (especially, if a portion of a program looks ugly or appears to be too complex).
- expect that **higher-order functions** and **lazy evaluation** are the tools for achieving adequate modularization and generalization.

Lazy or Eager Evaluation?

...the final conclusion of [John Hughes](#) reconsidering this [recurring question](#) is:

- ▶ The benefits of lazy evaluation as a [glue](#) are so evident that [lazy evaluation](#) is too important to make it a [second-class citizen](#).
- ▶ [Lazy evaluation](#) is possibly [the most powerful glue](#) functional programming has to offer.
- ▶ Access to such a powerful means [should not airily be dropped](#).

Lasst uns faul in allen Sachen,
[...]

nur nicht faul zur Faulheit sein.

Gotthold Ephraim Lessing (1729-1781)
dt. Dichter und Dramatiker

Looking ahead

...in [Chapter 2](#) and [Chapter 3](#) we will discuss the power **higher-order functions** and **lazy evaluation** provide the programmer with in further detail:

- [Stream programming](#): exploiting **lazy evaluation** (cf. [Chapter 2](#)).
- [Algorithm patterns](#): exploiting **higher-order functions** (cf. [Chapter 3](#)).

Chapter 1.5

References, Further Reading

Lecture 1

Detailed
Outline

Chap. 1

1.1

1.2

1.3

1.4

1.5

Chap. 2

Concludin
Note

Assignme

Chapter 1: Basic Reading (and Viewing)



John Hughes. *Why Functional Programming Matters*.
Computer Journal 32(2):98-107, 1989.



John Hughes. *Why Functional Programming Matters*.
Invited Keynote, Bangalore, 2016.

<https://www.youtube.com/watch?v=XrNdvWqxBvA>

Lecture 1

Detailed
Outline

Chap. 1

1.1

1.2

1.3

1.4

1.5

Chap. 2

Concludin
Note

Assignme

Chapter 1: Selected Advanced Reading (1)



Neal Ford. *Functional Thinking: Why Functional Programming is on the Rise*. IBM developerWorks, 10 pages, 2013.

<https://www.ibm.com/developerworks/java/library/j-ft20/j-ft20-pdf.pdf>

...why you should care about functional programming even if you don't plan to change languages any time soon.



Neil Savage. *Using Functions for Easier Programming*. Communications of the ACM 61(5):29-30, 2018.

...when the limestone of imperative programming has worn away, the granite of functional programming will be revealed underneath ⟨quote of Simon Peyton Jones⟩.

Chapter 1: Selected Advanced Reading (2)



Greg Michaelson. *Programming Paradigms, Turing Completeness and Computational Thinking*. The Art, Science, and Engineering of Programming 4(3), Article 4, 21 pages, 2020.



Philip Wadler. *The Essence of Functional Programming*. In Conference Record of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'92), 1-14, 1992.



Edsger W. Dijkstra. *Go To Statement Considered Harmful*. Letter to the Editor. Communications of the ACM 11(3):147-148, 1968.

Lecture 1

Detailed
Outline

Chap. 1

1.1

1.2

1.3

1.4

1.5

Chap. 2

Concludin
Note

Assignme

Sometimes, the elegant implementation is a function.
Not a method. Not a class. Not a framework.
Just a function.

John Carmack

Chapter 2

Programming with Streams

Lecture 1

Detailed
Outline

Chap. 1

Chap. 2

2.1

2.2

2.3

2.4

2.5

2.6

2.7

Concludin
Note

Assignme

Streams, Stream Programming

...a powerful means which – thanks to **lazy evaluation** – often allows

- to solve problems **elegantly, concisely, efficiently**
- to **gain/improve performance**

but also a

- a **source of hassle** if applied inappropriately.

Note: Streams are also called **infinite lists** or **lazy lists**.

We will focus on

...applications of streams and stream programming with the

1. **Generate-prune** pattern as a powerful modularization principle with instances like:
 - 1.1 **Generate-select**
 - 1.2 **Generate-filter**
 - 1.3 **Generate-transform**
2. Opportunities for performance improvement.
3. Pitfalls and remedies.

In later chapters, we consider the theoretical foundations underlying and justifying stream programming:

4. Well-definedness of functions on streams
(cf. [Appendix A.7.5](#))
5. Proving properties of functions on streams
(cf. [Chapter 6.3.4](#), [6.4](#), [6.5](#), [6.6](#))

Implementing Streams

...could be done by a new polymorphic data type like:

```
data Stream a = a :* Stream a
```

to emphasize the conceptual difference of **streams** (**infinite** by definition) and **lists** (**finite** by definition).

Pragmatically, however, it is advantageous to model **streams** (and **lists**) by ordinary

- list types `[a]` (omitting for streams the empty list `[]`)

since this way we can take advantage of the huge array of pre-defined

- (polymorphic) functions on lists

which otherwise would have to be (re-) defined from scratch.

Chapter 2.1

Streams, Stream Generators

Lecture 1

Detailed
Outline

Chap. 1

Chap. 2

2.1

2.2

2.3

2.4

2.5

2.6

2.7

Concludin
Note

Assignme

Simple Stream Generators

► Built-in streams in Haskell

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

```
[0,2..] ->> [0,2,4,6,8,10,...
```

```
[1,3..] ->> [1,3,5,7,9,11,...
```

```
[1,1..] ->> [1,1,1,1,1,1,...
```

► User-defined streams in Haskell

```
ones = 1 : ones
```

```
ones ->> 1 : ones
```

```
->> 1 : (1 : ones)
```

```
->> 1 : (1 : (1 : ones))
```

```
->> ...
```

Note: The expressions `ones` and `[1,1..]` represent the same infinite lists (or streams), the stream of ‘ones.’

Stream Generators: Corecursive Definitions

Definitions like

`ones = 1 : ones`

`twos = 2 : twos`

`threes = 3 : threes`

defining the streams of 'ones,' 'twos,' 'threes' are called

► **corecursive**.

Corecursive definitions

- are **recursive** definitions but lack a base case.
- always yield **infinite** objects.
- remind to Münchhausen's famous trick of "sich am eigenen Schopfe aus dem Sumpf zu ziehen!"

More Corecursively Defined Stream Generators

The `stream`

- ▶ `nats` of natural numbers:

```
nats = 0 : map (+1) nats  
->> [0,1,2,3,...
```

- ▶ `evens` of even natural numbers :

```
evens = 0 : map (+2) evens  
->> [0,2,4,6,...
```

- ▶ `odds` of odd natural numbers:

```
odds = 1 : map (+2) odds  
->> [1,3,5,7,...
```

- ▶ `theNats` of natural numbers:

```
theNats = 0 : zipWith (+) ones theNats  
->> [0,1,2,3,...
```

Lecture 1

Detailed
Outline

Chap. 1

Chap. 2

2.1

2.2

2.3

2.4

2.5

2.6

2.7

Concludin
Note

Assignme

Stream Generators

...defined in terms of [list comprehension](#) and [recursion](#).

The [stream](#) of

- [powers](#) of some integer:

```
powers :: Int -> [Int]
powers n = [n^x | x <- [0..]]

~> [1, n, n*n, n*n*n, ...]
```

- 'function applications,' the prelude function [iterate](#):

```
iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)

~> [x, f x, f (f x), f (f (f x)), ...]
```

Stream Generators

...defined with `iterate` yielding alternative definitions of some of the stream generators defined so far:

```
powers n = iterate (*n) 1
ones     = iterate id 1
twos     = iterate id 2
threes   = iterate id 3
nats     = iterate (+1) 0
theNats  = iterate (+1) 0
evens    = iterate (+2) 0
odds     = iterate (+2) 1
```

where

```
id = \x -> x
```

Lecture 1

Detailed
Outline

Chap. 1

Chap. 2

2.1

2.2

2.3

2.4

2.5

2.6

2.7

Concluding
Note

Assignme

Streams as Results of Functions

...user-defined stream-yielding functions.

► Streams of integers

```
from :: Int -> [Int]
```

```
from n = n : from (n+1)
```

```
fromStep :: Int -> Int -> [Int]
```

```
fromStep n m = n : fromStep (n+m) m
```

Examples:

```
from 42      ->> [42,43,44,...
```

```
fromStep 3 2 ->> 3 : fromStep 5 2
```

```
->> 3 : 5 : fromStep 7 2
```

```
->> 3 : 5 : 7 : fromStep 9 2
```

```
->> ...
```

```
->> [3,5,7,9,11,13,15,...
```

► Streams of (pseudo) random numbers...

► The stream of prime numbers...

Streams of (Pseudo) Random Numbers (1)

...a **generator** for (periodic) streams of (pseudo) random numbers:

```
randomSequence :: Int -> [Int]           -- Periodic
randomSequence = iterate nextRandNum     -- Generator

nextRandNum :: Int -> Int
nextRandNum n =
    (multiplier * n + increment) 'mod' modulus
```

Lecture 1

Detailed
Outline

Chap. 1

Chap. 2

2.1

2.2

2.3

2.4

2.5

2.6

2.7

Concludin
Note

Assignme

Streams of (Pseudo) Random Numbers (2)

Example: Choosing

<code>seed</code>	<code>= 17489</code>	<code>increment</code>	<code>= 13849</code>
<code>multiplier</code>	<code>= 25173</code>	<code>modulus</code>	<code>= 65536</code>

the evaluation of `randomSequence` with argument `seed` yields a periodic stream of (pseudo) random numbers, where all numbers are in the range of 0 to 65536 and occur with the same frequency:

```
randomSequence seed  
->> [17489, 59134, 9327, 52468, 43805, 8378, ...]
```

The Stream of Primes (1)

...along the idea of Eratosthenes of a Sieve of Primes:

1. Write down the natural numbers from 2 onwards.
2. The smallest number not cancelled is a prime number; cancel all multiples of this number.
3. Repeat step 2 with the then smallest number not cancelled.

Illustrating the algorithmic idea of sieving:

Step 1:

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17...

Step 2 (with '2' as smallest not cancelled number):

2 3 5 7 9 11 13 15 17...

Step 2 (with '3' as smallest not cancelled number):

2 3 5 7 11 13 17...

Step 2 (with '5' as smallest not cancelled number):

2 3 5 7 11 13 17...

...

The Stream of Primes (2)

Exploiting the idea of sieving for implementation.

The `stream primes` of prime numbers as result of applying the `filter` function `sieve` to the `generator [2..]`:

```
primes :: [Int]
primes = sieve [2..]
```

Generator *Filter* *Generator*

```
sieve :: [Int] -> [Int]
```

```
sieve (x:xs) = x : sieve [ y | y <- xs, mod y x > 0]
```

The Stream of Primes (3)

Illustrating the **filtering** property of **sieve** by stepwise evaluation:

```
primes
```

```
->> sieve [2..]
```

```
->> 2 : sieve [ y | y <- [3..], mod y 2 > 0]
```

```
->> 2 : sieve (3 : [ y | y <- [4..], mod y 2 > 0])
```

```
->> 2 : 3 : sieve [ z | z <- [ y | y <- [4..],  
                        mod y 2 > 0 ],  
                        mod z 3 > 0]
```

```
->> ...
```

```
->> 2 : 3 : sieve [ z | z <- [5, 7, 9..],  
                        mod z 3 > 0]
```

```
->> ...
```

```
->> 2 : 3 : sieve [5,7,11,...
```

```
->> ...
```

```
->> [2,3,5,7,11,13,17,19,...
```

Lecture 1

Detailed
Outline

Chap. 1

Chap. 2

2.1

2.2

2.3

2.4

2.5

2.6

2.7

Concludin
Note

Assignme

Note

...evaluating **stream generating terms** does not terminate and yields (at least conceptually) **infinitely long** lists.

Fortunately, the non-terminating evaluation of **stream generating terms** can be tamed using the

- ▶ **Generate-Prune Pattern**

which allows conceptually new ways of

- ▶ **modularizing**

lazily evaluated functional programs.

Chapter 2.2

The Generate-Prune Pattern

Lecture 1

Detailed
Outline

Chap. 1

Chap. 2

2.1

2.2

2.2.1

2.2.2

2.2.3

2.2.4

2.3

2.4

2.5

2.6

2.7

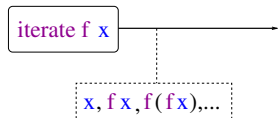
Concluding
Note

Assignme

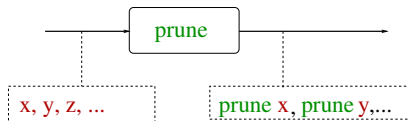
The Generate-Prune Pattern

...a means of **modularly** composing **lazily evaluated** functional programs:

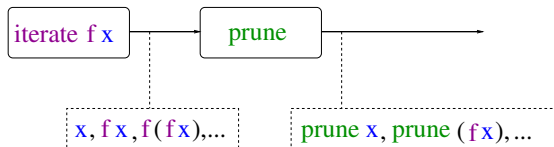
Generator module:



Pruning module:



Linking Generator and Pruning modules together:



Lecture 1

Detailed
Outline

Chap. 1

Chap. 2

2.1

2.2

2.2.1

2.2.2

2.2.3

2.2.4

2.3

2.4

2.5

2.6

2.7

Concluding
Note

Assignme

Basic Instances of the Generate-Prune Pattern

...are: The

1. Generate-select
2. Generate-filter
3. Generate-transform

instances and combinations thereof, which themselves can be considered **patterns**.

Lecture 1

Detailed
Outline

Chap. 1

Chap. 2

2.1

2.2

2.2.1

2.2.2

2.2.3

2.2.4

2.3

2.4

2.5

2.6

2.7

Concludin
Note

Assignme

Chapter 2.2.1

The Generate-Select/Filter Pattern

Lecture 1

Detailed
Outline

Chap. 1

Chap. 2

2.1

2.2

2.2.1

2.2.2

2.2.3

2.2.4

2.3

2.4

2.5

2.6

2.7

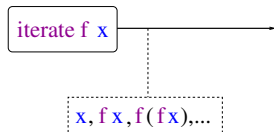
Concluding
Note

Assignme

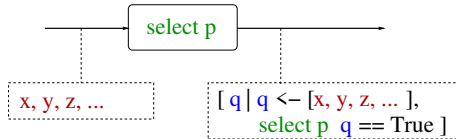
The Generate-Select/Filter Pattern

...at a glance:

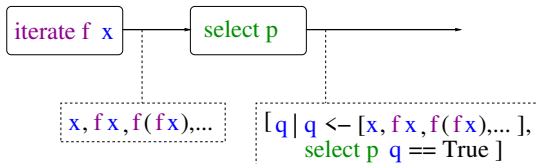
Generator module:



Selector/Filter module:



Linking Generator and Selector/Filter modules together:



Lecture 1

Detailed
Outline

Chap. 1

Chap. 2

2.1

2.2

2.2.1

2.2.2

2.2.3

2.2.4

2.3

2.4

2.5

2.6

2.7

Concluding
Note

Assignme

93/170

```
((take 5) . (drop 5)) primes ->> [13,17,19,23,29]
```

Selector *Generator*

Examples: Generate-Filter Pattern

...applications of the Generate-Filter Pattern:

- The tail of a stream:

```
tail      nats      ->> tail (0 : map (+1) nats)
└──┬──┬──┘
Filter Generator ->> map (+1) nats
                        ->> [1,2,3,...
```

- The prime numbers, which are a palindrome:

```
filter is_palindrome primes ->> [2,3,5,7,11,101,131,...]
└──┬──┬──┘
Filter Generator
```

- Even pseudo random numbers:

```
filter is_even (randomSequence 17489) ->> [69134,
└──┬──┬──┘
Filter Generator                          52468,
                                           8378,...
```

Is it a Selector or a Filter?

Taking a pragmatic point of view, if applied to a **stream**, termination is

- **ensured**, then it is a **selector**:

$\underbrace{((\text{take } 5) . (\text{drop } 5))}_{\text{Selector}} \underbrace{\text{primes}}_{\text{Generator}} \rightarrow [13, 17, 19, 23, 29]$

- **not ensured**, then it is a **filter**:

$\underbrace{\text{filter is_palindrome}}_{\text{Filter}} \underbrace{\text{primes}}_{\text{Generator}} \rightarrow [2, 3, 5, 7, 11, 101, \dots]$

A Note on Termination

...termination of a **generate-select** program depends crucially on evaluating the program in **normal order reduction** (typically implemented in terms of the efficient **lazy order reduction**) to avoid the non-terminating infinite sequence of reductions of evaluating the program in **applicative order reduction**:

- ▶ **Applicative order** reduction:

```
head twos
->> head (2 : twos)
->> head (2 : 2 : twos)
->> head (2 : 2 : 2 : twos)
->> ...
```

- ▶ **Normal/lazy order** reduction:

```
head twos
->> head (2 : twos)
->> 2
```


Reminder

...whenever there is a terminating reduction sequence of an expression, then normal order reduction will terminate.

Church/Rosser Theorem 12.3.2 (LVA 185.A03 FP)

Normal order reduction is typically implemented in terms of its efficient variant of lazy order reduction based on leftmost-outermost evaluation.

Lecture 1

Detailed
Outline

Chap. 1

Chap. 2

2.1

2.2

2.2.1

2.2.2

2.2.3

2.2.4

2.3

2.4

2.5

2.6

2.7

Concludin
Note

Assignme

Chapter 2.2.2

The Generate-Transform Pattern

Lecture 1

Detailed
Outline

Chap. 1

Chap. 2

2.1

2.2

2.2.1

2.2.2

2.2.3

2.2.4

2.3

2.4

2.5

2.6

2.7

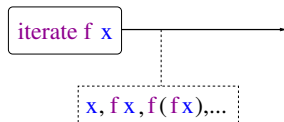
Concluding
Note

Assignme

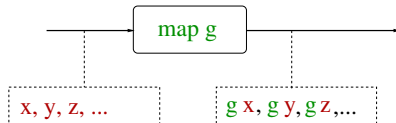
The Generate-Transform Pattern

...at a glance:

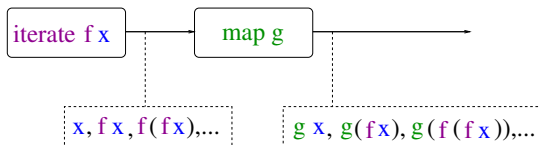
Generator module:



Transformer module:



Linking Generator and Transformer modules together:



Lecture 1

Detailed
Outline

Chap. 1

Chap. 2

2.1

2.2

2.2.1

2.2.2

2.2.3

2.2.4

2.3

2.4

2.5

2.6

2.7

Concludin
Note

Assignme

Examples: Generate-Transform Pattern (1)

- The stream of predecessors of prime numbers:

```
map (\x -> x-1) primes ->> [1,2,4,6,10,12,16,...]
```

Transformer *Generator*

- The stream of truth values indicating which prime numbers are a palindrome:

```
map is_palindrome primes ->> [True,True,True,True,  
True,False,False,...]
```

Transformer *Generator*

- The stream of truth values indicating which values of a stream of pseudo random numbers are even:

```
map is_even (randomSequence 17489) ->> [False,  
True,  
False,...]
```

Transformer *Generator*

Examples: Generate-Transform Pattern (2)

...often random numbers r within a range from p to q :

$$p \leq r \leq q$$

are required.

This also can be achieved using the generate/transform pattern by properly scaling (i.e., transforming) the values of a sequence of pseudo random numbers:

scale 42.0 51.0 randomSequence
Transformer Generator

```
scale :: Float -> Float -> [Int] -> [Float]
scale p q randSeq = map (f p q) randSeq
  where f :: Float -> Float -> Int -> Float
        f p q n = p + ((n * (q-p)) / (modulus-1))
```

Chapter 2.2.3

Pattern Combinations

Lecture 1

Detailed
Outline

Chap. 1

Chap. 2

2.1

2.2

2.2.1

2.2.2

2.2.3

2.2.4

2.3

2.4

2.5

2.6

2.7

Concluding
Note

Assignme

Examples: Pattern Combinations

...applications of pattern combinations:

- ▶ The stream of prime numbers:

```
primes = sieve (tail (map (\n -> n+1) nats))
```

Generator *Filter* *Filter* *Transformer* *Generator*

- ▶ The 6th to the 10th prime number:

```
((take 5) . (drop 5)) primes ->> [13,17,19,23,29]
```

Selector *Filter* *Generator*

- ▶ Selecting and adding the first two elements of a stream:

```
addFirstTwo   twos
```

Selector + Transformer *Generator*

```
->> addFirstTwo (2:twos)
->> addFirstTwo (2:2:twos)
->> 2+2
->> 4
```

where `addFirstTwo (x:y:zs) = x+y`

Chapter 2.2.4

Summary

Lecture 1

Detailed
Outline

Chap. 1

Chap. 2

2.1

2.2

2.2.1

2.2.2

2.2.3

2.2.4

2.3

2.4

2.5

2.6

2.7

Concludin
Note

Assignme

Principles of Modularization

...enabled by **stream programming** and **lazy evaluation**:

- ▶ The **Generate-Select Principle**
...e.g., computing the square root, the n -th Fibonacci number.
- ▶ The **Generate-Filter Principle**
...e.g., computing all even Fibonacci numbers.
- ▶ The **Generate-Transform Principle**
...e.g., 'scaling' random numbers.
- ▶ (Complex) combinations of **generators**, **transformers**, **filters**, and **selectors**.

Lecture 1

Detailed
Outline

Chap. 1

Chap. 2

2.1

2.2

2.2.1

2.2.2

2.2.3

2.2.4

2.3

2.4

2.5

2.6

2.7

Concludin
Note

Assignme

Chapter 2.3

Boosting Performance

Lecture 1

Detailed
Outline

Chap. 1

Chap. 2

2.1

2.2

2.3

2.3.1

2.3.2

2.3.3

2.3.4

2.4

2.5

2.6

2.7

Concluding
Note

Assignme

Chapter 2.3.1

Motivation

Lecture 1

Detailed
Outline

Chap. 1

Chap. 2

2.1

2.2

2.3

2.3.1

2.3.2

2.3.3

2.3.4

2.4

2.5

2.6

2.7

Concluding
Note

Assignme

Recall

...the **straightforward** implementation:

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

of the **Fibonacci function**:

$$\begin{aligned} fib &: \mathbb{N}_0 \rightarrow \mathbb{N}_0 \\ fib(n) &=_{df} \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ fib(n-1) + fib(n-2) & \text{if } n \geq 2 \end{cases} \end{aligned}$$

has **exponential time complexity** and is thus **inacceptably inefficient** and **slow** for all but the smallest arguments (cp. **LVA 185.A03 FP**).

Fortunately

...stream programming can (often) help

- conquering complexity
- gaining/improving performance!

Lecture 1

Detailed
Outline

Chap. 1

Chap. 2

2.1

2.2

2.3

2.3.1

2.3.2

2.3.3

2.3.4

2.4

2.5

2.6

2.7

Concluding
Note

Assignme

Chapter 2.3.2

Stream Programming combined with Münchhausen Principle

Lecture 1

Detailed
Outline

Chap. 1

Chap. 2

2.1

2.2

2.3

2.3.1

2.3.2

2.3.3

2.3.4

2.4

2.5

2.6

2.7

Concluding
Note

Assignme

Computing the Fibonacci Numbers Stream Eff.

0 1 1 2 3 5 8 13.. The stream of Fibonacci numbers

1 1 2 3 5 8 13 21.. The tail of the stream of Fib. numb.

+ + + + + + + +.. ++++++ add columnwise ++++++

1 2 3 5 8 13 21 34.. The tail of the tail of the
stream of Fibonacci numbers

This can easily be implemented as a (corecursive) stream:

```
fibs :: [Int] -- Generator
```

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)  
      'Tuft'      'Swamp'
```

The tail of the tail of the stream of Fib. numb.

The stream of Fibonacci numbers

...using Münchhausen's trick of "sich am eigenen Schopfe aus dem Sumpf zu ziehen!"

The Münchhausen Principle in Detail

'Sw 0 1 1 2 3 5 8 13.. Stream of fibs

am 1 1 2 3 5 8 13 21.. Tail of stream of fibs

p' + + + + + + + +.. +++ add columnwise +++

0 1 1 2 3 5 8 13 21 34.. Stream of fibs

'Tuft' Tail of tail of stream of fibs

...and the implementation as (corecursive) stream:

fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
 'Tuft' adding 'Swamp' columnwise
 Tail of tail of stream of fibs
 Stream of Fibonacci numbers

Note: This way the stream of Fibonacci numbers is computed w/out referring to the recursive default definition of the Fibonacci function.

Application: Generate/Select Principle

Generator:

```
fibs ->> 0 : 1 : 1 : 2 : 3 : 5 : 8 : 13 : 21 : 34 : 55 : 89...
```

Generate-Select applications:

```
fibs!!7 ->> 13
```

```
take 8 fibs ->> [0,1,1,2,3,5,8,13]
```

```
(head . (drop 7)) fibs ->> 13
```

where

```
take :: Int -> [a] -> [a]
```

```
take 0 _ = []
```

```
take _ [] = []
```

```
take n (x:xs) | n>0 = x : take (n-1) xs
```

```
take _ _ = error "Negative argument"
```

Computing Fibonacci Numbers Efficiently

...the **corecursive** definition of the stream **fibs** suggests a **conceptually new** efficient implementation of the **Fibonacci function fib**:

```
fib :: Int -> Int
fib n =   head   (drop (n-1)  fibs)
           $\underbrace{\hspace{1.5cm}}_{\text{Selector 2}} \underbrace{\hspace{1.5cm}}_{\text{Selector 1}} \underbrace{\hspace{1.5cm}}_{\text{Generator}}$ 
          [ =   last   (take n  fibs) ]
               $\underbrace{\hspace{1.5cm}}_{\text{Selector 2}} \underbrace{\hspace{1.5cm}}_{\text{Selector 1}} \underbrace{\hspace{1.5cm}}_{\text{Generator}}$ 
```

And even shorter with only one selector:

```
fib :: Int -> Int
fib n =  $\underbrace{\text{fibs}}_{\text{Generator}} \underbrace{!! n}_{\text{Selector}}$ 
```

Note the **generate-select** modularization in the two implementations of **fib**.

Note: Lazy Evaluation is Crucial for Performance

...naive evaluation w/out sharing of common subexpression causes exponential computational effort (using `add` instead of `zipWith (+)`):

`fibs`

```
->> {Replace the call of fibs by the body of fibs}
      0 : 1 : add fibs (tail fibs)
->> {Replace both calls of fibs by the body of fibs}
      0 : 1 : add (0 : 1 : add fibs (tail fibs))
              (tail (0 : 1 : add fibs (tail fibs)))
->> {Application of tail}
      0 : 1 : add (0 : 1 : add fibs (tail fibs))
              (1 : add fibs (tail fibs))
->> ... exponential effort!
```

...**lazy evaluation** ensures that common subexpressions (here, `tail` and `fibs`) are not computed multiple times!

The Benefit of Lazy Evaluation: Sharing (1)

```
fibs ->> 0 : 1 : add fibs (tail fibs)

->> {Introd. abbrev. allows sharing of results}
    0 : tf      -- tf reminds to "tail of fibs"
    where tf = 1 : dd fibs (tail fibs)

->> 0 : tf
    where tf = 1 : add fibs tf

->> {Introducing abbreviations allows sharing}
    0 : tf
    where tf = 1 : tf2 -- tf2 reminds to "tail
                        -- of tail of fibs"
                        where tf2 = add fibs tf

->> {Unfolding of add}
    0 : tf
    where tf = 1 : tf2
            where tf2 = 1 : add tf tf2
```

Lecture 1

Detailed
Outline

Chap. 1

Chap. 2

2.1

2.2

2.3

2.3.1

2.3.2

2.3.3

2.3.4

2.4

2.5

2.6

2.7

Concludin
Note

Assignme

The Benefit of Lazy Evaluation: Sharing (2)

->> {Repeating the above steps}

```
0 : tf
```

```
where tf = 1 : tf2
```

```
      where tf2 = 1 : tf3  (tf3 reminds to  
                           "tail of tail  of tail of fibs")  
            where tf3 = add tf tf2
```

->> 0 : tf

```
where tf = 1 : tf2
```

```
      where tf2 = 1 : tf3
```

```
            where tf3 = 2 : add tf2 tf3
```

->> {tf is only used once and can thus be eliminated}

```
0 : 1 : tf2
```

```
where tf2 = 1 : tf3
```

```
      where tf3 = 2 : add tf2 tf3
```

Lecture 1

Detailed
Outline

Chap. 1

Chap. 2

2.1

2.2

2.3

2.3.1

2.3.2

2.3.3

2.3.4

2.4

2.5

2.6

2.7

Concludin
Note

Assignme

The Benefit of Lazy Evaluation: Sharing (3)

->> {Finally, we obtain successsively longer pre-
fixes of the stream of Fibonacci numbers}

```
0 : 1 : tf2
```

```
where tf2 = 1 : tf3
```

```
      where tf3 = 2 : tf4
```

```
            where tf4 = add tf2 tf3
```

->> 0 : 1 : tf2

```
where tf2 = 1 : tf3
```

```
      where tf3 = 2 : tf4
```

```
            where tf4 = 3 : add tf3 tf4
```

{ Note: Eliminating where-clauses corresponds
to garbage collection of unused memory by an
implementation. }

->> 0 : 1 : 1 : tf3

```
      where tf3 = 2 : tf4
```

```
            where tf4 = 3 : add tf3 tf4
```

Chapter 2.3.3

Stream Programming combined with Memoization

Lecture 1

Detailed
Outline

Chap. 1

Chap. 2

2.1

2.2

2.3

2.3.1

2.3.2

2.3.3

2.3.4

2.4

2.5

2.6

2.7

Concludin
Note

Assignme

Memoization

...goes back to [Donald Michie](#):

- Donald Michie. 'Memo' Functions and Machine Learning. Nature, 218:19-22, 1968.

Essence

- Replace, where possible, the (costly) computation of a function according to its body by looking up its value in a table, a so-called [memo table](#).

Means

- A costly to compute function is replaced by an equivalent [memo function](#) using [\(memo\) table look-ups](#). Intuitively, the original function is augmented by a cache storing argument/result pairs.

Memo Functions, Memo Tables

A **memo function** is

- an **ordinary function**, but stores for some or all arguments it has been applied to the results in a **memo table**.

A **memo table** allows

- to replace recomputation by **table look-up**.

Requirement: A **memo function** `memo`

`memo :: (a -> b) -> (a -> b)`

for replacing some function `f : a -> b` must satisfy:

`memo f x = f x`

Referential transparency of pure functional programming languages (especially, **absence of side effects!**) greatly simplifies

- **Soundness** proofs involving **memoization**.

Illustrating the Essence of Memo Functions

...and **memo tables**, sometimes simpler **memo lists** (i.e., one-dimensional **memo tables**).

Assume $f : ID \rightarrow ID'$ is a (costly to compute) function with (enumerable) **domain** ID and **range** ID' :

```
f :: Enum d => d -> r
f d' = r'                                -- basic case
f x = exp                               -- exp involving recursive calls of f
```

Then: Replace calls of **f** by implementing f (except of a few calls for basic cases) by a look-up in a **memo list**:

```
memo_list = [ memo f d'' | d'' <- [d'..] ]    -- Generator
memo :: Enum d => (d -> r) -> d -> r
memo f d' = f d'                            -- Basis ('tuft')
memo f x = exp'                             -- Trigger, (expr' is exp with calls of
-- f replaced by memo list look-ups)
memo_f d = memo f d                        -- memo_f replacing f
```

Memo Functions, Memo Tables: Schematically

'Generic' Pattern, schematically:

```
f :: Enum d => d -> r
f d' = r'                                -- basic case
f x = exp                                -- exp involving recursive calls of f

memo_list = [memo f d'' | d'' <- [d'..]] ]    -- Generator

memo :: Enum d => (d -> r) -> d -> r
memo f d' = f d'                          -- Basis ('tuft')
memo f x = exp'                            -- Trigger, (expr' is exp with calls of
                                           -- f replaced by memo list look-ups)

memo_f d = memo f d                      -- memo_f replacing f
```

Lecture 1

Detailed
Outline

Chap. 1

Chap. 2

2.1

2.2

2.3

2.3.1

2.3.2

2.3.3

2.3.4

2.4

2.5

2.6

2.7

Concludin
Note

Assignme

Memo Functions, Memo Tables: Example

Computing the Fibonacci function using memoization:

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)      -- Not reached by memo!

memo_list = [memo fib n | n <- [0..]]      -- Generator

memo :: (Int -> Int) -> Int -> Int
memo fib 0 = fib 0                    -- Basis ('tuft')
memo fib 1 = fib 1                    -- Basis ('tuft')
memo fib n = memolist !! (n-1) + memolist !! (n-2) -- Trigger

memo_fib n = memo fib n              -- memo_fib replacing fib
```

Lecture 1

Detailed
Outline

Chap. 1

Chap. 2

2.1

2.2

2.3

2.3.1

2.3.2

2.3.3

2.3.4

2.4

2.5

2.6

2.7

Concludin
Note

Assignme

Example 1: Computing Fibonacci Numbers

Computing Fibonacci numbers with memoization/memo lists:

```
fib_memolist = [ fib_ml n | n <- [0..] ]  
fib_ml 0 = 0  
fib_ml 1 = 1  
fib_ml n = fib_memolist!!(n-1) + fib_memolist!!(n-2)  
           Generator Selector      Generator Selector
```

Compare this w/ the straightforward implementation of fib:

```
fib 0 = 0  
fib 1 = 1  
fib n = fib (n-1) + fib (n-2)
```

Lemma 2.3.3.1

$\forall n \in \mathbb{N}. \text{fib_ml } n = \text{fib } n$

Note: Looking-up the result of calls instead of recomputing them again, leads to a **substantial performance gain!**

Example 2: Computing Powers

Computing powers ($2^0, 2^1, \dots$) with memoization/memo lists:

```
pow_memolist = [ power_ml x | x <- [0..] ]  
power_ml 0 = 1  
power_ml i = pow_memolist!!(i-1) + pow_memolist!!(i-1)  
             Generator Selector      Generator Selector
```

Compare this w/ the straightforward implement. of power:

```
power 0 = 1  
power i = power (i-1) + power (i-1)
```

Lemma 2.3.3.2

$\forall n \in \mathbb{N}. \text{ power_ml } n = \text{ power } n$

Note: Looking-up the result of the second call instead of re-computing it requires only $1 + n$ calls of `power_ml` instead of $1 + 2^n$. This is a **significant performance gain!**

Summing up

A **memo function** `memo :: (a -> b) -> (a -> b)`

- is essentially the identity on functions.
- (but) keeps track on the arguments it has been applied to and their corresponding result values.

Motto: Looking-up results which have been computed earlier instead of recomputing them!

Memo functions are

- not a part of the Haskell'98 standard.
- supported by some non-standard libraries.

Note: In [Example 1](#) and [2](#), the general **memo list/memo function pattern** is syntactically condensed by squeezing

- `memo/fib`, `memo/power` into `fib_ml`, `power_ml`, resp.

Chapter 2.3.4

Summary

Lecture 1

Detailed
Outline

Chap. 1

Chap. 2

2.1

2.2

2.3

2.3.1

2.3.2

2.3.3

2.3.4

2.4

2.5

2.6

2.7

Concludin
Note

Assignme

Avoiding Recomputations, Avoiding Recursion

...are major sources of **performance improvement**.

Stream programming combined with

- **Münchhausen principle**
- **memoization**

can (often) help **avoiding recomputing values** unnecessarily and **recursively**.

Lecture 1

Detailed
Outline

Chap. 1

Chap. 2

2.1

2.2

2.3

2.3.1

2.3.2

2.3.3

2.3.4

2.4

2.5

2.6

2.7

Concludin
Note

Assignme

Stream Programming w/ Münchhausen Princ.

...avoiding recomputations, avoiding recursion.

► Computing Fibonacci numbers:

```
fibs :: [Int]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)

fib :: Int -> Int
fib n = fibs !! n
        Generator Selector
```

► Computing powers:

```
powers :: [Int]
powers = 1 : 2 : zipWith (+) (tail powers) (tail powers)

power :: Int -> Int
power n = powers !! n
          Generator Selector
```

► ...

Lecture 1

Detailed
Outline

Chap. 1

Chap. 2

2.1

2.2

2.3

2.3.1

2.3.2

2.3.3

2.3.4

2.4

2.5

2.6

2.7

Concluding
Notes

Assignme

Stream Programming w/ Memoization

...avoiding recomps, avoiding rec. (except of 1st call f. an arg.).

► Computing Fibonacci numbers:

```
fib_ml :: [Int]
fib_ml = [ fib n | n <- [0..] ]      -- Memo list

fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib_ml!!(n-1) + fib_ml!!(n-2)
```

► Computing powers:

```
power_ml :: [Int]
power_ml = [ power n | n <- [0..] ]  -- Memo list

power :: Int -> Int
power 0 = 1
power i = power_ml!!(i-1) + power_ml!!(i-1)
```

► ...

Lecture 1

Detailed
Outline

Chap. 1

Chap. 2

2.1

2.2

2.3

2.3.1

2.3.2

2.3.3

2.3.4

2.4

2.5

2.6

2.7

Concludin
Note

Assignme

Memoization vs. Münchhausen Approach

Memoization approach:

- The first time `fib_ml` and `power_ml` are evaluated for an argument, the computation proceeds as prescribed by the default recursive definitions of the Fibonacci and the power function.
- Subsequent calls of `fib_ml` and `power_ml` for an argument they have been applied to before, however, benefit from memoization: Recomputation and recursion is replaced by referring to the stored value.

This is different for the Münchhausen approach:

- It does not refer at all to the default recursive definitions of the Fibonacci and the power function.
- Even the very first look-up of the stream functions for an argument benefits and does not rely on a recursive computation process (`zipWith` does not count).

In closing

Stream programming combined w/ the [Münchhausen principle](#) and [memoization](#) are important though

- [no silver bullets](#)

for improving performance by avoiding recomputations and recursion.

If, however, they hit they can significantly

- [boost performance](#): from taking too long to be feasible to be completed in an instant!

[Natural candidates](#) are problems that

- naturally wind up repeatedly computing the solution to identical subproblems, e.g., [tree-recursive processes](#).

Sometimes

...however, a problem-dependent **silver bullet** might exist.

Computing **Fibonacci numbers** is (again) a striking example.

The equality of **Theorem 2.3.4.3** (cf. **Chapter 6**) allows a (recursion-free) **direct computation** of the **Fibonacci numbers**:

$$\begin{aligned} fib : \mathbb{N}_0 &\rightarrow \mathbb{N}_0 \\ fib(n) &=_{df} \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ fib(n-1) + fib(n-2) & \text{if } n \geq 2 \end{cases} \end{aligned}$$

Theorem 2.3.4.3

$$\forall n \in \mathbb{N}_0. fib(n) = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}$$

Chapter 2.4

Stream Diagrams

Lecture 1

Detailed
Outline

Chap. 1

Chap. 2

2.1

2.2

2.3

2.4

2.5

2.6

2.7

Concludin
Note

Assignme

Stream Diagrams

...are a means for considering and visualizing problems on **streams** as

- processes.

We illustrate this considering the **streams** of

1. Fibonacci numbers
2. **communications** of some **client/server** application

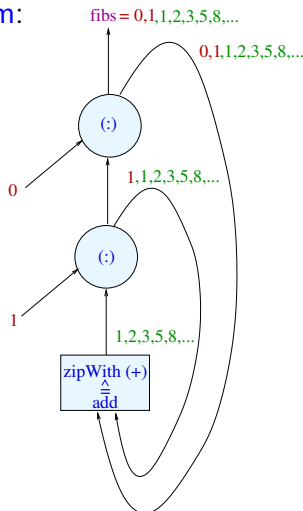
as examples.

Example 1: Fibonacci Numbers

...representing the **stream** of **Fibonacci numbers** defined by

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

as a **stream diagram**:



Example 2: A Client/Server Application

...a **client/server interaction** (e.g., Web server/Web browser):

```
type Request = Integer
```

```
type Response = Integer
```

```
client :: [Response] -> [Request]
```

```
client ys = 1 : ys    -- issues 1 as the 1st request,  
                      -- followed by all responses it  
                      -- receives (from the server).
```

```
server :: [Request] -> [Response]
```

```
server xs = map (+1) xs -- adds 1 to each request it  
                      -- receives (from the client).
```

Two Transformer-Generator Programs and their Interaction

```
reqs = client resps    -- Transformer-Generator
```

```
resps = server reqs    -- Transformer-Generator
```

Lecture 1

Detailed
Outline

Chap. 1

Chap. 2

2.1

2.2

2.3

2.4

2.5

2.6

2.7

Concludin
Note

Assignme

Stepwise Eval. the Client/Server Interactions

```
reqs ->> client resps
->> 1 : resps
->> 1 : server reqs
->> {Introducing abbreviations}
    1 : tr where tr = server reqs
->> 1 : tr where tr = 2 : (server tr)
->> 1 : tr where tr = 2 : tr2
                        tr2 = server tr
->> 1 : tr where tr = 2 : tr2
                        tr2 = 3 : server tr2
->> 1 : (2 : tr2) where tr2 = 3 : (server tr2)
->> ...
->> 1 : (2 : (3 : (4 : (5 : (... )))))
```

Application: Generate-Select pattern

```
take 10    reqs    ->> [1,2,3,4,5,6,7,8,9,10]
  ⏟         ⏟
Selector Generator
```

Lecture 1

Detailed
Outline

Chap. 1

Chap. 2

2.1

2.2

2.3

2.4

2.5

2.6

2.7

Concluding
Note

Assignme

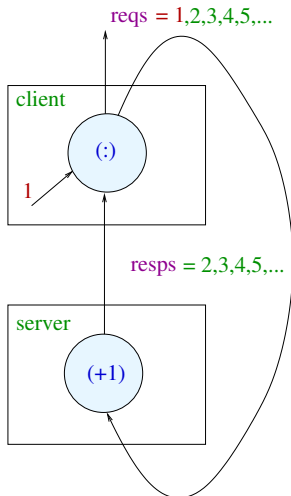
The Stream Diagram

...representing the stream of client/server interactions

reqs = client resps

resps = server reqs

as a stream diagram:



Chapter 2.5

Pitfalls, Remedies

Lecture 1

Detailed
Outline

Chap. 1

Chap. 2

2.1

2.2

2.3

2.4

2.5

2.5.1

2.5.2

2.5.3

2.6

2.7

Concluding
Note

Assignme

Chapter 2.5.1

Livelocks, Lazy Patterns

Lecture 1

Detailed
Outline

Chap. 1

Chap. 2

2.1

2.2

2.3

2.4

2.5

2.5.1

2.5.2

2.5.3

2.6

2.7

Concluding
Note

Assignme

Reconsider

...the [client/server](#) application of [Chapter 2.4](#).

Suppose, the [client](#) wants to [check the first response](#) before a new action. To this end, we replace its implementation:

```
client :: [Response] -> [Request]
client ys = 1 : ys
```

by:

```
client' :: [Response] -> [Request]
client' (y:ys) = if ok y then 1 : (y:ys)
                  else error "Faulty Server"
  where ok y = True
```

introducing a trivial check: a check which [always succeeds!](#)

Technically

...we replace as part of the shift from `client` to `client'` the

- **irrefutable** pattern `ys` in: `client ys = 1 : ys`
by the

- **refutable** pattern `(y:ys)` in:
`client' (y:ys) = if ok y...`

This modification looks harmless but **evaluating**:

```
reqs ->> client' resps
      ->> client' (server reqs)
      ->> client' (server (client' resps))
      ->> client' (server (client' (server reqs)))
      ->> ...
```

...does **not terminate** because of a **livelock!** Neither `client` nor `server` can be **unfolded**: Pattern matching for the **refutable** pattern `(y:ys)` requires verifying that the argument is not empty causing the pattern match being **'too eager!'**

Stepwise evaluating client/server

```
reqs ->> client resps
|      ->> { client ys = 1 : ys
|           (pattern ys is irrefutable, thus expand!) }
---> (*) 1 : resps
      (**) (implying: resps == tail reqs)
      ->> 1 : (server reqs)
      ->> { server xs = map (+1) xs
           (pattern xs is irrefutable, thus expand!) }
           1 : (map (+1) reqs)
      (*)
      ->> 1 : (map (+1) (1 : resps))
      (**)
      ->> 1 : (map (+1) (1 : tail (reqs)))
      ->> 1 : (2 : (map (+1) (tail (reqs))))
      ->> ...
      ->> 1 : (2 : (3 : (4 : (5 : (... )))))
```

Lecture 1

Detailed
Outline

Chap. 1

Chap. 2

2.1

2.2

2.3

2.4

2.5

2.5.1

2.5.2

2.5.3

2.6

2.7

Concluding
Note

Assignme

Stepwise evaluating client'/server (1)

reqs ->> client' resps

pm
->> { client' (y:ys) = if ok y then ...
Pattern (y:ys) is **refutable**; thus, we have
to verify that resps has at least a head
element! This requires to look inside of
resps. Hence, unfold resps! }

pm
->> client' (server reqs)

pm
->> { Now we have to look inside of server reqs
for a head element, where
server xs = map (+1) xs
Pattern xs is **irrefutable**, thus unfold! }
client' (map (+1) reqs)

pm
->> { Now we have to look inside of map (+1) reqs
for a head element; this in turn requires
to look inside of reqs. Hence, unfold reqs! }

pm
->> client' (map (+1) (client' resps))

Stepwise evaluating client'/server (2)

```
reqs ->> client' resps
      pm
      ->> ...
      pm
      ->> client' (map (+1) (client' resps))
      pm
      ->> { Now we have to look inside of
            (map (+1) (client' resps))
            for a head element; this in turn requires
            to look inside of resps. Thus unfold
            resps! }
            client' (map (+1) (client' (server reqs)))
      pm
      ->> ...
```

...and so on. We never see a head element of the argument of `client'`; hence, pattern matching does not terminate...

Livelock!

Remedy A: Moving the Test plus Selector

Replace the **refutable** pattern expression `(y:ys)`, whose check turned out to be **too eager**, by (i) the **irrefutable** pattern expression `ys`, (ii) pushing the test inside of the list, and (iii) using a selector function for accessing the head element of the argument `ys` of `client'` (i.e., `head ys`):

```
client' ys = 1 : if ok (head ys) then ys
               else error "Faulty Server"
```

Disadvantage: The 'fix' works but requires the selector function `head`, and looks less naturally and less functionally than the original implementation using the 'list decomposing' pattern `(y:ys)`:

```
client' (y:ys) = if ok y then 1 : (y:ys)
                  else error "Faulty Server"
```

Remedy B: Moving the Test plus Lazy Pattern

Replace the **refutable pattern** $(y:ys)$ by its **irrefutable lazy counterpart** $\sim(y:ys)$:

```
client'  $\sim(y:ys) = 1$  : if ok  $y$  then  $(y:ys)$   
                        else error "Faulty Server"
```

Since $\sim(y:ys)$ is **irrefutable**, evaluating the right-hand side expression starts w/out that pattern matching must have been completed.

Advantages:

1. The selector function **head** is not required any longer.
2. The 'fix' is more declarative and readable as that of **Remedy A**, even though the test must still be moved inside of the list.

Note, in practice **lazy patterns** allow saving very many calls of selector functions while making programs at the same time **more declarative** and **readable**, and thus **more appealing**.

Illustrating

...the effect of the **lazy pattern** by **stepwise evaluation**:

```
client' ~(y:ys) = 1 : if ok y then (y:ys)
                      else error "Faulty Server"
```

```
reqs ->> client' resps
->> 1 : if ok y then (y:ys)
      else error "Faulty Server"
      where (y:ys) = resps
->> 1 : (y:ys)
      where (y:ys) = resps
->> 1 : resps
->> ...
->> 1 : (2 : (3 : (4 : (5 : (... )))))
```

Excursus: Irrefutable vs. Refutable Patterns (1)

Haskell (like other functional languages) distinguishes between

- **irrefutable** (by default: variable names, wild card `_`)
- **refutable** (by default: all others, e.g. `[]`, `(x:xs)`, ...)

patterns.

Intuitively, **any value** passed (for pattern matching) to an

- **irrefutable pattern expression** matches it (e.g. `[]`, `[1,2,3]` both match pattern expression **ys** and pattern expression `_`).

whereas **values** passed to a

- **refutable pattern expression** match it **only if they fit** (e.g. `[]` matches pattern expression `[]` but not pattern expression `(y:ys)`, whereas `[1,2,3]` matches pattern expression `(y:ys)` but not pattern expression `[]`).

Excursus: Irrefutable vs. Refutable Patterns (2)

Illustrating the **impact** of **refutability** and **irrefutability** of a pattern (expression):

$f :: [a] \rightarrow \text{Int}$	$g :: [a] \rightarrow \text{Int}$
$f (x:xs) = 99$	$g \text{ } ys = 99$
$f [] = 42$	$g [] = 42$

The **empty list** does not match the refutable pattern $(x:xs)$ but the irrefutable pattern ys . Hence, calling f and g with the empty list, the second defining equation of f and the first one of g determine the result of f and g , respectively:

$f [] \rightarrow 42$	$g [] \rightarrow 99$
-----------------------	-----------------------

Excursus: Lazy Patterns (3)

In *Haskell*, *refutable* pattern expressions can be made *irrefutable* by preceding them with a tilde \sim :

```
h :: [a] -> Int
h ~(x:xs) = 99
h []      = 42
```

Pattern expressions made *irrefutable* this way like $\sim(x:xs)$ are also called *lazy pattern expressions* or *lazy patterns*.

Calling *h* with the empty list yields:

```
h [] ->> 99
```

Excursus: The Effect of Laziness of Patterns (4)

For **lazy patterns**, the check if a value matches the pattern is postponed until it is really required, i.e., until the pattern components are referred to in the course of evaluating the right-hand side term. This is not the case for calling **h** with whatever list value but it is the case for **k**. Hence, calling **h** with the empty list returns **99** as result, whereas calling **k** with the empty list crashes.

h :: [a] -> Int

h ~(x:xs) = 99

h [] = 42

h [] ->> 99

k :: [a] -> Int

k ~(x:xs) = 99 + x * sum xs

k [] = 42

k [] ->> 'run-time error'

While the above example is superficial and pathological, **lazy patterns** are **quite useful** in other ones like the **client/server** example.

Chapter 2.5.2

Lifting, Undecidability

Lecture 1

Detailed
Outline

Chap. 1

Chap. 2

2.1

2.2

2.3

2.4

2.5

2.5.1

2.5.2

2.5.3

2.6

2.7

Concluding
Note

Assignme

Functional Lifting

...compare the definition of the stream `fibs` (cp. [Chapter 2.3.1](#)):

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

with the definition of the stream `FibsFn`:

```
fibsFn :: () -> [Int]
fibsFn x = 0 : 1 : zipWith (+) (fibsFn ()) (tail (fibsFn ()))
```

which, intuitively, [lifts](#) the definition of `fibs` to a functional level.

Note

...evaluating

- `fibs`

is **fast** and **efficient**, whereas evaluating

- `fibsFn`

shows an

- ▶ **exponential** run-time and storage (**memory leak**) usage.

Intuitively, this is because:

- ▶ The ability of recognizing **common structures** is limited.

Memory leak: The memory space is consumed so fast that the performance of a program is severely impacted.

For Illustration

...consider:

```
fibsFn ()  
->> 0 : 1 : add (fibsFn ()) (tail (fibsFn ()))  
->> 0 : tf  
    where  
        tf = 1 : add (fibsFn ()) (tail (fibsFn ()))
```

The [equality](#) of `tf` and `tail(fibsFn())` remains undetected by compilers. Hence, the below simplification remains undone:

```
->> 0 : tf  
    where tf = 1 : add (fibsFn ()) tf
```

Note: While for special cases like the one here, this were possible, there is [no general means](#) for detecting such equalities.

Chapter 2.5.3

Termination, Domain-specific Knowledge

Lecture 1

Detailed
Outline

Chap. 1

Chap. 2

2.1

2.2

2.3

2.4

2.5

2.5.1

2.5.2

2.5.3

2.6

2.7

Concluding
Note

Assignme

Note

...lazy evaluation is

- necessary

to ensure termination of generate-select programs but

- not sufficient!

Lecture 1

Detailed
Outline

Chap. 1

Chap. 2

2.1

2.2

2.3

2.4

2.5

2.5.1

2.5.2

2.5.3

2.6

2.7

Concluding
Note

Assignme

For Illustration

...consider the below **naive prime number test**:

```
member :: Eq a => [a] -> a -> Bool
member []      y = False
member (x:xs) y = (x==y) || member xs y
```

where **member** can be considered a **transformer/selector** (a-value to **Bool**-value).

Then:

- a) $\underbrace{\text{member primes } 7 \rightarrow \text{True}}_{\text{Transformer/Selector: ...works properly!}} \text{ ...does terminate!}$
- b) $\underbrace{\text{member primes } 8 \rightarrow \dots}_{\text{Transformer/Selector: ...fails!}} \text{ ...does not terminate!}$

Chapter 2.6

Summary, Looking ahead

Lecture 1

Detailed
Outline

Chap. 1

Chap. 2

2.1

2.2

2.3

2.4

2.5

2.6

2.7

Concludin
Note

Assignme

Summary

...stream programming together with lazy evaluation enables:

- ▶ **Higher abstraction:** Constraining oneself to finite lists is often more complex, and – at the same time – unnatural.
- ▶ **Modularization:** Streams together with lazy evaluation allow for elegant possibilities of decomposing a computational problem. Most important is the
 - **Generate-Prune Pattern**of which the
 - Generate-select
 - Generate-filter
 - Generate-transform patternand combinations thereof are specific instances.
- ▶ **Boosting performance:** Avoiding recomputations and recursion using stream programming combined with:
 - Münchhausen principle (cf. Chapter 2.3.2)
 - memoization (cf. Chapter 2.3.3)

Looking ahead

We will occasionally return to

- stream programming

in later chapters, e.g., in [Chapter 16](#) on

- ‘Logic Programming Functionally’

in the context of exploring (conceptually) [infinite search spaces](#) in a [fair order](#) ensuring that every item of the search space is visited within a [finite amount of time](#).

Chapter 2.7

References, Further Reading

Lecture 1

Detailed
Outline

Chap. 1

Chap. 2

2.1

2.2

2.3

2.4

2.5




2.6

2.7

Concludin
Note

Assignme

Chapter 2: Basic Reading

-  Kees Doets, Jan van Eijck. *The Haskell Road to Logic, Maths and Programming*. Texts in Computing, Vol. 4, King's College, UK, 2004. (Chapter 10, Corecursion)
-  Paul Hudak. *The Haskell School of Expression – Learning Functional Programming through Multimedia*. Cambridge University Press, 2000. (Chapter 14, Programming with Streams; Chapter 14.3, Stream Diagrams; Chapter 14.4, Lazy Patterns; Chapter 14.5, Memoization)
-  Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley/Pearson, 3rd edition, 2011. (Chapter 17, Lazy programming; Chapter 17.6, Infinite lists; Chapter 17.7, Why infinite lists? Chapter 20.6, Avoiding recomputation: memoization)

Lecture 1

Detailed
Outline

Chap. 1

Chap. 2

2.1

2.2

2.3

2.4

2.5




2.6

2.7





Concludin
Note

Assignme

Chapter 2: Selected Advanced Reading (1)

-  [Antonie J.T. Davie. *An Introduction to Functional Programming Systems using Haskell*. Cambridge University Press, 1992. \(Chapter 7.3, Streams; Chapter 7.6, Irrefutable Patterns; Chapter 7.8, Memo Functions\)](#)
-  [Anthony J. Field, Peter G. Harrison. *Functional Programming*. Addison-Wesley, 1988. \(Chapter 4.2, Processing 'infinite' data structures; Chapter 4.3, Process networks; Chapter 19, Memoization\)](#)
-  [John Hughes. *Lazy Memo Functions*. In Proceedings of the IFIP Symposium on Functional Programming Languages and Computer Architecture \(FPCA'85\), Springer-V., LNCS 201, 129-146, 1985.](#)

Chapter 2: Selected Advanced Reading (2)

-  Donald Michie. *'Memo' Functions and Machine Learning*. Nature, 218:19-22, 1968.
-  Peter Pepper, Petra Hofstedt. *Funktionale Programmierung*. Springer-V., 2006. (Kapitel 14.2.1, Memoization; Kapitel 15.5, Maps, Funktionen und Memoization)
-  Fethi Rabhi, Guy Lapalme. *Algorithms – A Functional Programming Approach*. Addison-Wesley, 1999. (Chapter 10.1, Process networks)
-  Simon Peyton Jones (Ed.). *Haskell 98: Language and Libraries. The Revised Report*. Cambridge University Press, 2003. (Chapter 3.12, Let Expressions – irrefutable patterns; Chapter 3.17.2, Informal Semantics of Pattern Matching – irrefutable, refutable patterns; Chapter 4.4.3.2, Pattern bindings – 'lazily' matching patterns)

Lecture 1

Detailed
Outline

Chap. 1

Chap. 2

2.1

2.2

2.3

2.4

2.5

2.6

2.7

Concludin
Note

Assignme

Concluding Note

...for additional information and details refer to

► full course notes

available in TUWEL and at the homepage of the course at:

[http://www.complang.tuwien.ac.at/knoop/
ffp185A05_ss2021.html](http://www.complang.tuwien.ac.at/knoop/ffp185A05_ss2021.html)

Assignment for Thursday, 11 March 2021

...independent study of [Part I, Chapter 1](#), and [Part II, Chapter 2](#) and of [Central and Control Questions I](#) for self-assessment and as a basis of the flipped classroom session on [03/11/2021](#):

Lecture, Flipped Classroom	Topic Lecture	Topic Flip. Classr.
Thu, 03/04/2021, 4.15-6.00 pm	P. I, Ch. 1 P. II, Ch. 2	n.a. / Prel. Mtg.
Thu, 03/11/2021, 4.15-6.00 pm	P. IV, Ch. 7, 8 P. II, Ch. 3	P. I, Ch. 1 P. II, Ch. 2
Thu, 03/25/2021, 4.15-6.00 pm	P. II, Ch. 4 P. IV, Ch. 9–11, 14	P. IV, Ch. 7, 8 P. II, Ch. 3
Thu, 04/15/2021, 4.15-6.00 pm	P. IV, Ch. 12, 13	P. II, Ch. 4 P. IV, Ch. 9–11, 14
Thu, 04/22/2021, 4.15-6.00 pm	P. III, Ch. 5, 6	P. IV, Ch. 12, 13
Thu, 04/29/2021, 4.15-6.00 pm	P. V, Ch. 15, 16	P. III, Ch. 5, 6
Thu, 05/20/2021, 4.15-6.00 pm	P. V, Ch. 17, 18 P. VI, Ch. 19, 20	P. V, Ch. 15, 16

Lecture 1

Detailed
Outline

Chap. 1

Chap. 2

Concludin
Note

Assignme