Fortgeschrittene funktionale Programmierung

LVA 185.A05, VU 2.0, ECTS 3.0
SS 2020

(Stand: 29.04.2020)

Jens Knoop

Technische Universität Wien
Information Systems Engineering
Compilers and Languages

Lecture 1

Part I: Motivation

- Chapter 1: Why Functional Programming Matters

Part II: Programming Principles

- Chapter 2: Programming with Streams

Outline in more Detail (1)

Outline in more Detail (2)

Part II: Programming Principles

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

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' does not really explain the power of functional programming; in particular, it does 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.

Overall

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

Preparing the Setting

...following the position statement, program examples will be presented in Miranda^{TM} 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:

```
                          +---+
sum nil                 = | 0 |
                          +---+
                               +---+
sum (cons num list) = num      | + |  sum list
                               +---+
```

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
```

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]
```

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 . 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                                                   1
 (cons (node 2 nil)                                     / \
       (cons (node 3 (cons (node 4 nil) nil)) 2   3
       nil))                                                 |
                                                             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)))
```

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

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`:

  ```
  out = (g . f) in = g (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

The Newton-Raphson Approach

...for square root computation.

Given: `N`, a positive number

Sought: `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:

```
     (a + N/a) / 2  = a      | *2
⇔    a + N/a        = 2a     | -a
⇔    N/a            = a      | *a
⇔    N              = a*a    | squr
⇔    squareRoot(N)  = a
```

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:

```
next N x = (x + N/x) / 2
```

Second, we define function `g`:

```
g = next N
```

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

```
[a0, g a0, g (g a0), g (g (g a0)),..
```

which is equivalent to:

```
[a0, next N a0, next N (next N a0),
 next N (next N (next N a0)),..
```

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!

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
Final Note

Writing a Selector

...applied to some value 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 abs(a-b) <= eps
    = within eps (cons b rest), otherwise
```

Glueing together Generator and Selector

...to obtain the final program.

Glueing together Generator A and Selector A:

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

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 $\texttt{eps} > 0$, used here as indication of sufficient precision of the currently reached approximation.

Summing up

The functional version of the program approximating the sqare 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 a0 with abs(g^i a0 - g^{i+1} a0) <= eps
  ...lazy evaluation ensures that the selector function is applied eventually ⇒ 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., <= eps`), await their ratio to approach one (`i.e., <= 1+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:

$$\texttt{relativesqrt N eps a0} \\ \texttt{= } \underbrace{\texttt{relative eps}}_{\textit{Selector B}} \texttt{ (}\underbrace{\texttt{repeat (next N) a0}}_{\textit{Generator A}}\texttt{)}$$

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

Numerical Integration

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

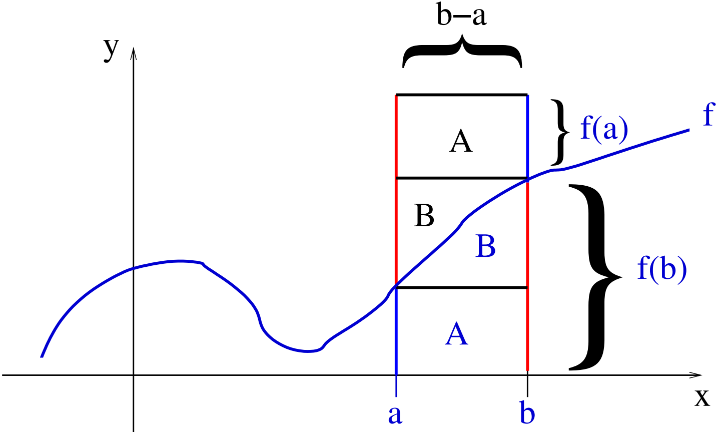
Sought: The area under `f` between `a` and `b`

Simple Solution:
...assuming that the function `f` is roughly linear between `a` und `b`.

```
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`



$$\int_a^b f(x)\,dx = A+B = (f(a) + f(b))*(b-a) / 2$$

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)
```

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:

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

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

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`.

```
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.

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:

$$\underbrace{\texttt{within eps}}_{\textit{Selector A}}\ (\underbrace{\texttt{differentiate h0 f x}}_{\textit{Generator C}})$$

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

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:

- Higher-order functions (glueing functions)
- 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

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

Chapter 1: Selected Further 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 Further 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.

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

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

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,...
```

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..]]
```

  $\rightsquigarrow$ `[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)
```

  $\rightsquigarrow$ `[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
```

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
```

Streams of (Pseudo) Random Numbers (2)

Example: Choosing

```
seed        = 17489        increment = 13849
multiplier  = 25173        modulus   = 65536
```

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]
```

$$\underbrace{\texttt{primes}}_{\textit{Generator}} \ = \ \underbrace{\texttt{sieve}}_{\textit{Filter}}\ \underbrace{\texttt{[2..]}}_{\textit{Generator}}$$

```
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,...
```

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.

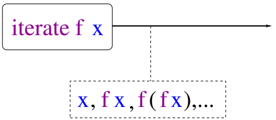
Chapter 2.2

The Generate-Prune Pattern

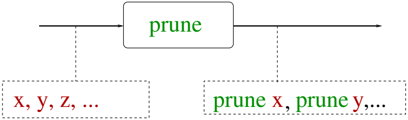
The Generate-Prune Pattern

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

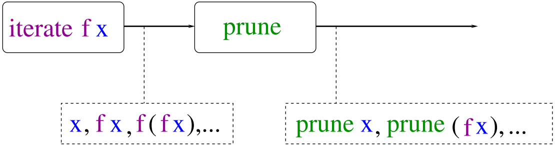
Generator module:



Pruning module:



Linking Generator and Pruning modules together:



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.

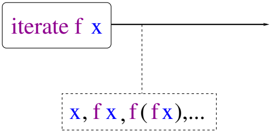
Chapter 2.2.1

The Generate-Select/Filter Pattern

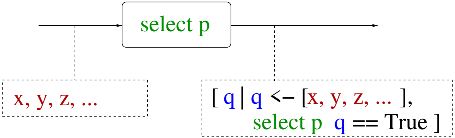
The Generate-Select/Filter Pattern

...at a glance:

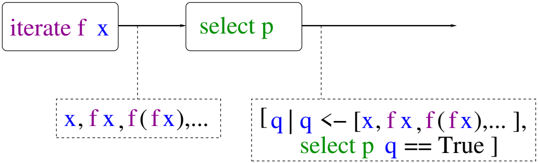
Generator module:



Selector/Filter module:



Linking Generator and Selector/Filter modules together:



Examples: Generate-Select Pattern

...applications of the Generate-Select Pattern:

- The head element of a stream:

  $\underbrace{\texttt{head}}_{\textit{Selector}}\ \underbrace{\texttt{nats}}_{\textit{Generator}}$ `->> head (0 : map (+1) nats)`

  `->> 0`

- Three pseudo random numbers:

  $\underbrace{\texttt{take 3}}_{\textit{Selector}}\ \underbrace{\texttt{(randomSequence 17489)}}_{\textit{Generator}}$ `->> [17489,69134,9327]`

- The 6th to the 10th prime number:

  $\underbrace{\texttt{((take 5) . (drop 5))}}_{\textit{Selector}}\ \underbrace{\texttt{primes}}_{\textit{Generator}}$ `->> [13,17,19,23,29]`

Examples: Generate-Filter Pattern

...applications of the Generate-Filter Pattern:

- The tail of a stream:

$$\underbrace{\texttt{tail}}_{\textit{Filter}}\ \underbrace{\texttt{nats}}_{\textit{Generator}}\ \texttt{->> tail (0 : map (+1) nats)}$$

$$\texttt{->> map (+1) nats}$$

$$\texttt{->> [1,2,3,...}$$

- The prime numbers, which are a palindrome:

$$\underbrace{\texttt{filter is\_palindrome}}_{\textit{Filter}}\ \underbrace{\texttt{primes}}_{\textit{Generator}}\ \texttt{->> [2,3,5,7,11,101,131,..}$$

- Even pseudo random numbers:

$$\underbrace{\texttt{filter is\_even}}_{\textit{Filter}}\ \underbrace{\texttt{(randomSequence 17489)}}_{\textit{Generator}}\ \texttt{->> [69134, 52468, 8378,...}$$

Is it a Selector or a Filter?

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

– ensured, then it is a selector:

$$\underbrace{\texttt{((take 5) . (drop 5))}}_{\textit{Selector}}\ \underbrace{\texttt{primes}}_{\textit{Generator}}\ \texttt{->> [13,17,19,23,29]}$$

– not ensured, then it is a filter:

$$\underbrace{\texttt{filter is\_palindrome}}_{\textit{Filter}}\ \underbrace{\texttt{primes}}_{\textit{Generator}}\ \texttt{->> [2,3,5,7,11,101,...}$$

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.

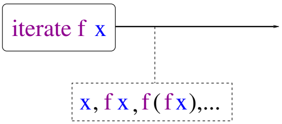
Chapter 2.2.2

The Generate-Transform Pattern

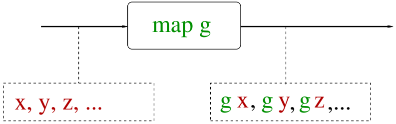
The Generate-Transform Pattern

...at a glance:

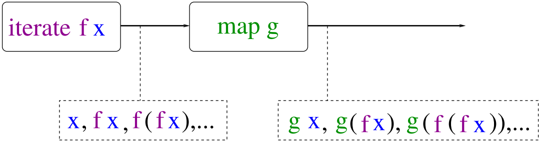
Generator module:



Transformer module:



Linking Generator and Transformer modules together:



Examples: Generate-Transform Pattern (1)

- The stream of predecessors of prime numbers:

  $\underbrace{\texttt{map (\textbackslash x -> x-1)}}_{\textit{Transformer}}\ \underbrace{\texttt{primes}}_{\textit{Generator}}$ ->> [1,2,4,6,10,12,16,...

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

  $\underbrace{\texttt{map is\_palindrome}}_{\textit{Transformer}}\ \underbrace{\texttt{primes}}_{\textit{Generator}}$ ->> [True,True,True,True, True,False,False,...

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

  $\underbrace{\texttt{map is\_even}}_{\textit{Transformer}}\ \underbrace{\texttt{(randomSequence 17489)}}_{\textit{Generator}}$ ->> [False, True, False,...

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:

$$\underbrace{\texttt{scale 42.0 51.0}}_{\textit{Transformer}}\ \underbrace{\texttt{randomSequence}}_{\textit{Generator}}$$

```
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

Examples: Pattern Combinations

...applications of pattern combinations:

- The stream of prime numbers:

$$\underbrace{\texttt{primes}}_{\textit{Generator}} = \underbrace{\texttt{sieve}}_{\textit{Filter}}\ \texttt{(}\underbrace{\texttt{tail}}_{\textit{Filter}}\ \texttt{(}\underbrace{\texttt{map (\textbackslash n -> n+1)}}_{\textit{Transformer}}\ \underbrace{\texttt{nats}}_{\textit{Generator}}\texttt{))}$$

- The 6th to the 10th prime number:

$$\texttt{(}\underbrace{\texttt{(take 5)}}_{\textit{Selector}}\ \texttt{.}\ \underbrace{\texttt{(drop 5)}}_{\textit{Filter}}\texttt{)}\ \underbrace{\texttt{primes}}_{\textit{Generator}}\ \texttt{->> [13,17,19,23,29]}$$

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

$$\underbrace{\texttt{addFirstTwo}}_{\textit{Selector + Transformer}}\ \underbrace{\texttt{twos}}_{\textit{Generator}}$$

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

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

Chapter 2.2.4

Summary

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-Tansform Principle
  ...e.g., 'scaling' random numbers.
- (Complex) combinations of generators, transformers, filters, and selectors.

Chapter 2.3

Boosting Performance

Chapter 2.3.1

Motivation

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:

$$fib : \mathbb{N}_0 \to \mathbb{N}_0$$

$$fib(n) =_{df} \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ fib(n-1) + fib(n-2) & \text{if } n \geq 2 \end{cases}$$

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!

Chapter 2.3.2

Stream Programming combined with Münchhausen Principle

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)
```

$$\texttt{fibs} = \underbrace{\underbrace{\texttt{0 : 1}}_{\texttt{'Tuft'}} \texttt{ : } \underbrace{\underbrace{\texttt{zipWith (+) fibs (tail fibs)}}_{\texttt{'Swamp'}}}_{\texttt{The tail of the tail of the stream of Fib. numb.}}}_{\texttt{The stream of Fibonacci numbers}}$$

...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`:

$$\texttt{fib :: Int -> Int}$$

$$\texttt{fib n = } \underbrace{\texttt{head}}_{\textit{Selector 2}} \; \underbrace{\texttt{(drop (n-1)}}_{\textit{Selector 1}} \; \underbrace{\texttt{fibs)}}_{\textit{Generator}}$$

$$\texttt{[= } \underbrace{\texttt{last}}_{\textit{Selector 2}} \; \underbrace{\texttt{(take n}}_{\textit{Selector 1}} \; \underbrace{\texttt{fibs)}}_{\textit{Generator}} \texttt{]}$$

And even shorter with only one selector:

$$\texttt{fib :: Int -> Int}$$

$$\texttt{fib n = } \underbrace{\texttt{fibs}}_{\textit{Generator}} \; \underbrace{\texttt{!! n}}_{\textit{Selector}}$$

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
```

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
```

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

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 $f : \mathrm{ID} \to \mathrm{ID}'$ is a (costly to compute) function with domain ID and range ID'.

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

```
memolist = [ memo f d'' | d'' <- [d'..] ]     -- Generator
memo :: (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)
f :: d -> r
f d' = r'
f x = exp        -- exp involving recursive calls of f
```

Memo Functions, Memo Tables

'Generic' Pattern:

```
memolist = [ memo f d'' | d'' <- [d'..] ]        -- Generator
memo :: (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)
f :: d -> r
f d' = r'
f x  = exp          -- exp involving recursive calls of f
```

Concrete instance for the Fibonacci function:

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

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)
```

$\texttt{fib\_ml n} = \underbrace{\texttt{fib\_memolist}}_{\textit{Generator}}\underbrace{\texttt{!!(n-1)}}_{\textit{Selector}} + \underbrace{\texttt{fib\_memolist}}_{\textit{Generator}}\underbrace{\texttt{!!(n-2)}}_{\textit{Selector}}$

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

$\forall\, n \in \mathbb{N}.\ \texttt{fib\_ml n = fib n}$

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

Example 2: Computing Powers

Computing powers $(2^0, 2^1, \ldots)$ 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)
```

$$\texttt{power\_ml i = }\underbrace{\texttt{pow\_memolist}}_{\textit{Generator}}\underbrace{\texttt{!!(i-1)}}_{\textit{Selector}}\texttt{ + }\underbrace{\texttt{pow\_memolist}}_{\textit{Generator}}\underbrace{\texttt{!!(i-1)}}_{\textit{Selector}}$$

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$\in$ IN. power_ml n = 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

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.

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
```

$\underbrace{\texttt{fibs}}_{\textit{Generator}}\ \underbrace{\texttt{!! n}}_{\textit{Selector}}$

- Computing powers:

```
powers :: [Int]
powers = 1 : 2 : zipWith (+) (tail powers) (tail powers)
power :: Int -> Int
power n = powers !! n
```

$\underbrace{\texttt{powers}}_{\textit{Generator}}\ \underbrace{\texttt{!! n}}_{\textit{Selector}}$

- ...

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)
```

- ...

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:

$$fib : \mathbb{N}_0 \to \mathbb{N}_0$$

$$fib(n) =_{df} \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ fib(n-1) + fib(n-2) & \text{if } n \geq 2 \end{cases}$$

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

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

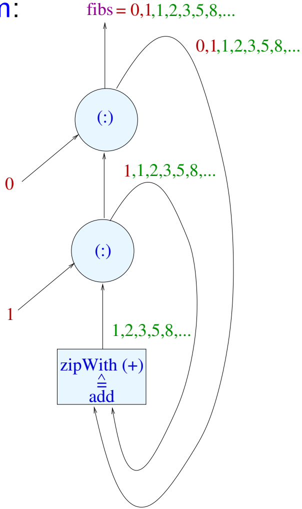
as examples.

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
                     -- received (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
```

Illustrating the Client/Server Interactions

Application: Generate-Select pattern

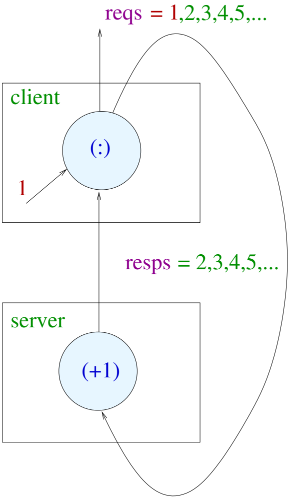
$$\underbrace{\texttt{take 10}}_{\textit{Selector}}\ \underbrace{\texttt{reqs}}_{\textit{Generator}}\ \texttt{->>}\ \texttt{[1,2,3,4,5,6,7,8,9,10]}$$

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

Chapter 2.5.1

Termination, Domain-specific Knowledge

Note

...lazy evaluation is

- necessary

to ensure termination of generate-select programs but

- not sufficient!

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) $\underbrace{\texttt{member primes}}_{\text{Transformer/Selector}}$ `7 ->> True` ...does terminate!

Transformer/Selector: ...works properly!

b) $\underbrace{\texttt{member primes}}_{\text{Transformer/Selector}}$ `8 ->> ...` ...does not terminate!

Transformer/Selector: ...fails!

Chapter 2.5.2

Lifting, Undecidability

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

Livelocks, Lazy Patterns

Reconsider

...the client/server application of Chapter 2.4.

Suppose, the client wants to check the first response:

```
client (y:ys) = if ok y then 1 : (y:ys)
                        else error "Faulty Server"
  where ok y = True     -- The check is trivial:
                        -- ‘always succeeding’
```

Though this modifaction looks harmless, 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 is ‘too eager!’

Remedies: Selector Functions, Lazy Patterns

A): Selector Functions

Replacing pattern matching by selector function access (using `head`), and pushing the conditional inside of the list:

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

B): Lazy patterns (preceding tilde ~)

Defering pattern matching (no selector function required).

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

Note: The conditional must still be moved inside of the list but the call of the selector function is no longer required. In practice, very many calls of selector functions can be saved this way by using lazy patterns making programs at the same time 'more' declarative and readable.

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
```

Chapter 2.6

Summary, Looking ahead

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 pattern

  and 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

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)

Chapter 2: Selected Further Reading (1)

- Antonie J.T. Davie. *An Introduction to Functional Programming Systems using Haskell*. Cambridge University Press, 1992. (Chapter 7.3, Streams; 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 Further 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)

Final Note

...for additional information and details refer to

- full course notes

available at the homepage of the course at:

http:://www.complang.tuwien.ac.at/knoop/ffp185A05_ss2020.html