Fortgeschrittene funktionale Programmierung

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

(Stand: 22.04.2020)

Jens Knoop

Technische Universität Wien
Information Systems Engineering
Compilers and Languages

Lecture 3

Part II: Programming Principles

- Chapter 4: Equational Reasoning for Functional Pearls

Part IV: Advanced Language Concepts

- Chapter 9: Monoids
- Chapter 10: Functors
- Chapter 11: Applicative Functors
- Chapter 14: Kinds

Outline in more Detail (1)

Part II: Programming Principles

Outline in more Detail (2)

Outline in more Detail (3)

Outline in more Detail (4)

- Chap. 11: Applicative Functors

  11.1 The Type Constructor Class Applicative

  11.2 Applicative Examples

    11.2.1 The Identity Applicative

    11.2.2 The List Applicative

    11.2.3 The Maybe Applicative

    11.2.4 The Either Applicative

    11.2.5 The Map Applicative

    11.2.6 The Ziplist Applicative

    11.2.7 The Input/Output Applicative

  11.3 References, Further Reading

- Chap. 14: Kinds

  14.1 Kinds of Types

  14.2 Kinds of Type Constructors

  14.3 References, Further Reading

Chapter 4

Equational Reasoning for Functional Pearls

Chapter 4.1

Equational Reasoning

Lecture 3
Detailed Outline
Chap. 4
4.1
4.2
4.3
4.4
4.5
4.6
From Type to Higher-Order Type Classes
Chap. 9
Chap. 10
Chap. 11
Chap. 14
Final Note

Equational Reasoning

...a well-known mathematical means for reasoning about and proving the validity of e.g. arithmetical statements:

Proposition 4.1.1

$$(a+b) \; * \; (a-b) \; = \; a^2 \; - \; b^2$$

Proof. Equational reasoning yields:

$$\begin{array}{rcl} & & (a+b) \; * \; (a-b) \\ \text{(Distributivity of } *, +) & = & a*a \; - \; a*b \; + \; b*a \; - \; b*b \\ \text{(Commutativity of } *) & = & a*a \; - \; a*b \; + \; a*b \; - \; b*b \\ & = & a*a \; - \; b*b \\ & = & a^2 \; - \; b^2 \end{array}$$

□

Equational Reasoning

...carries over to functional programming because in functional programming the equality symbol `'='` means:

- 'equal by definition:'

  The value of the left-hand side expression is defined as the value of the right-hand side expression.

An equation of the form

```
f x y = x+y
```

as (part of the) definition of a function `f` is thus a

- genuine mathematical equation:

  The expression on the left hand side and the right hand side of `=` have the same value.

Illustrating Equational Reasoning

...in a functional programming context:

Proposition 4.1.2

The Haskell functions f and g:

```
f :: Int -> Int -> Int
f a b = (a+b) * (a-b)

g :: Int -> Int -> Int
g a b = a^2 - b^2
```

denote the same function.

Proof. Using Proposition 4.1.1 and equational reasoning we obtain:

$$
\begin{array}{rcl}
 & & \texttt{f a b} \\
\text{(Definition of } \texttt{f}\text{, unfolding } \texttt{f}\text{)} & = & \texttt{(a+b) * (a-b)} \\
\text{(Proposition 4.1.1)} & = & \texttt{a}^2 \texttt{ - } \texttt{b}^2 \\
\text{(Definition of } \texttt{g}\text{, folding } \texttt{g}\text{)} & = & \texttt{g a b}
\end{array}
$$

□

Folding, Unfolding of Functional Definitions

...can be applied from

- left-to-right (called unfolding)
- right-to-left (called folding)

in equational reasoning as shown in the proof of Proposition 4.1.2

Note

...however, that some care on folding/unfolding must be taken because the Haskell semantics implicitly imposes an ordering on the equations.

For illustration consider:

```
isZero :: Int -> Bool
isZero 0 = True
isZero n = False
```

The first equation `isZero 0 = True` can be viewed as a logical property. It can

– freely be applied in both directions.

The second equation `isZero n = False` can not. It can

– only be applied, if `n` is different from `0`.

Towards Functional Pearls (1)

Consider functions reverse, fast_reverse for list reversal:

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]

fast_reverse :: [a] -> [a]
fast_reverse xs = fr xs []
  where fr [] ys     = ys
        fr (x:xs) ys = fr xs (x:ys)
```

Note:

- reverse requires $\frac{n(n+1)}{2}$ calls of the concatenation function (++) with n denoting the length of the argument list.
- fast_reverse does not rely on list concatenation (++) but on list construction (:); it is thus much more efficient.

Towards Functional Pearls (2)

If we could prove Theorem 4.1.5 stating that `reverse` and `fast_reverse` actually denote the same function, replacing `reverse` by `fast_reverse` would yield a significant speed-up of programs:

Theorem 4.1.5 (Equality)

The functions `reverse` and `fast_reverse` denote the same function, i.e.,

$\forall$ `ls` $\in$ `a-List. reverse ls = fast_reverse ls`

Proving Theorem 4.1.5: The Functional Pearl!

Equational reasoning (in concert with other techniques like induction) will be instrumental to conduct this proof showing that `reverse` and `fast_reverse` are equal and hence, the optimization of replacing `reverse` by `fast_reverse` correct!

Proving Theor. 4.1.5: The Functional Pearl (1)

Proof of Theorem 4.1.5 by structural induction on the structure of the list argument and equational reasoning.

Induction base: Let `ls = []`. We obtain:

```
                              reverse ls
(ls = [])                  =  reverse []
(Unfolding reverse)        =  []
(Folding fr)               =  fr [] []
(Folding fast_reverse)     =  fast_reverse []
([] = ls)                  =  fast_reverse ls
```

Proving Theor. 4.1.5: The Functional Pearl (2)

Induction step: Let `ls = (v:ls')`. We obtain:

$$
\begin{array}{rcl}
 & & \texttt{reverse ls} \\
(\texttt{lst = (v:ls')}) & = & \texttt{reverse (v:ls')} \\
(\text{Unfolding } \texttt{reverse}) & = & \texttt{reverse ls' ++ [v]} \\
(\text{IH}) & = & \texttt{fast\_reverse ls' ++ [v]} \\
(\text{Unfolding } \texttt{fast\_reverse}) & = & \texttt{(fr ls' []) ++ [v]} \\
(\text{Lemma 4.1.7}) & = & \texttt{fr ls' [v]} \\
(\text{Folding } \texttt{fr}) & = & \texttt{fr ls' (v:[])} \\
(\text{Folding } \texttt{fr}) & = & \texttt{fr (v:ls') []} \\
(\text{Folding } \texttt{fast\_reverse}) & = & \texttt{fast\_reverse (v:ls')} \\
(\texttt{(v:lst') = ls}) & = & \texttt{fast\_reverse ls} \qquad \square
\end{array}
$$

Proving the Supporting Results (1)

Lemma 4.1.6

$\forall$ `ls1,ls2` $\in$ `a-List` $\forall$ `v` $\in$ `a-Value.`
`(fr ls1 ls2) ++ [v] = fr ls1 (ls2 ++ [v])`

Proof. by structural induction on the structure of the list argument `ls1` and equational reasoning.

Induction base: Let `ls1` = `[]`, let `ls2` $\in$ `a-List`, and let `v` $\in$ `a-Value`. We obtain:

		`(fr ls1 ls2) ++ [v]`
(`ls1=[]`)	=	`(fr [] ls2) ++ [v]`
(Unfolding `fr`)	=	`ls2 ++ [v]`
(Folding `fr`)	=	`fr [] (ls2 ++ [v])`
(`[]=ls1`)	=	`fr ls1 (ls2 ++ [v])`

Proving the Supporting Results (2)

Induction step: Let $\mathtt{ls1} = (\mathtt{v}' \mathtt{:ls1}')$, let $\mathtt{ls2} \in \mathtt{a\text{-}List}$, and let $\mathtt{v} \in \mathtt{a\text{-}Value}$. We obtain:

$$
\begin{array}{rcl}
 & & (\mathtt{fr\ ls1\ ls2})\ \mathtt{++}\ [\mathtt{v}] \\
(\mathtt{ls1} = (\mathtt{v}'\mathtt{:ls1}')) & = & (\mathtt{fr}\ (\mathtt{v}'\mathtt{:ls1}')\ \mathtt{ls2})\ \mathtt{++}\ [\mathtt{v}] \\
(\text{Unfolding } \mathtt{fr}) & = & (\mathtt{fr\ ls1}'\ (\mathtt{v}'\mathtt{:ls2}))\ \mathtt{++}\ [\mathtt{v}] \\
(\mathtt{ls3} =_{df} (\mathtt{v}'\mathtt{:ls2})) & = & (\mathtt{fr\ ls1}'\ \mathtt{ls3})\ \mathtt{++}\ [\mathtt{v}] \\
(\text{IH}) & = & \mathtt{fr\ ls1}'\ (\mathtt{ls3} \mathtt{++} [\mathtt{v}]) \\
((\mathtt{v}'\mathtt{:ls2}) = \mathtt{ls3}) & = & \mathtt{fr\ ls1}'\ ((\mathtt{v}'\mathtt{:ls2}) \mathtt{++} [\mathtt{v}]) \\
(\text{Def. of } (\mathtt{:}) \text{ and } (\mathtt{++})) & = & \mathtt{fr\ ls1}'\ (\mathtt{v}'\mathtt{:}(\mathtt{ls2} \mathtt{++} [\mathtt{v}])) \\
(\text{Folding } \mathtt{fr}) & = & \mathtt{fr}\ (\mathtt{v}'\mathtt{:ls1}')\ (\mathtt{ls2} \mathtt{++} [\mathtt{v}]) \\
((\mathtt{v}'\mathtt{:ls1}') = \mathtt{ls1}) & = & \mathtt{fr\ ls1}\ (\mathtt{ls2} \mathtt{++} [\mathtt{v}]) \qquad \Box
\end{array}
$$

Proving the Supporting Results (3)

Lemma 4.1.7

$\forall\, \texttt{ls}' \in \texttt{a-List}\ \forall\, \texttt{v} \in \texttt{a-Value}.$
$\quad (\texttt{fr ls}'\ \texttt{[]})\ \texttt{++}\ \texttt{[v]} = \texttt{fr ls}'\ \texttt{[v]}$

Proof. Let $\texttt{ls}' \in \texttt{a-List}$ and let $\texttt{v} \in \texttt{a-Value}$. Setting $\texttt{ls1} = \texttt{ls}'$ and $\texttt{ls2} = \texttt{[]}$, we obtain by equational reasoning and Lemma 4.1.6:

$$
\begin{array}{rcl}
 & & (\texttt{fr ls}'\ \texttt{[]})\ \texttt{++}\ \texttt{[v]} \\
(\texttt{ls}'\texttt{=ls1,[]=ls2}) & = & (\texttt{fr ls1 ls2})\ \texttt{++}\ \texttt{[v]} \\
(\text{Lemma 4.1.6}) & = & \texttt{fr ls1 (ls2 ++ [v])} \\
(\texttt{ls1=ls}',\ \texttt{ls2=[]}) & = & \texttt{fr ls}'\ \texttt{([] ++ [v])} \\
(\texttt{[]++[v]=[v]}) & = & \texttt{fr ls}'\ \texttt{[v]}
\end{array}
$$

□

Application: Program Optimization

...equational reasoning together with inductive proof principles (structural induction) allowed us to prove Theorem 4.1.5:

– For all finite lists `xs`, the Haskell expressions `reverse xs`, `fast_reverse xs` are equal, i.e., have the same value:

  $\forall$ `xs` $\in$ `a-List. reverse xs == fast_reverse xs`

Replacing `reverse` by `fast_reverse` is thus safe:

Corollary 4.1.8 (Optimization)

Replacing every call of `reverse` by a call of `fast_reverse` in a program is a safe optimization of the program.

Comparing the Suitability

...of functional and imperative programming for equational reasoning.

Functional definitions are

- genuine mathematical equations.

This enables reasoning about functional programs by means of equational reasoning as is known from mathematics and standard (algebraic) reasoning.

Reasoning about functional programs is thus a lot easier as about imperative programs where equational reasoning does not apply (as easily).

Note

...in imperative programming, the equality symbol '=' means:

- 'equal by assignment:'

  The contents of the memory cell named by the left-hand side variable is replaced by the value of the right-hand side expression.

An 'equation' of the form

```
x = x+y
```

thus does not represent a mathematical equation meaning that `x` and `x+y` have the same value but a command, an instruction, a destructive assignment statement meaning that

– the sum of the values stored in the memory cells named `x` and `y` is used for overwriting the value stored so far in the memory cell named `x`, destroying thereby this value.

Note: To avoid confusion some imperative languages thus use a different symbol, e.g. `:=` such as in Pascal, to denote the assignment operator (instead of the conceptually misleading symbol `=`).

Illustrating the Difference

...consider the definition-like symbol sequence S:

```
x = 1
y = 2
x = x + y
```

In functional languages like Haskell, S is an

– invalid sequence of definitions raising an error that `x` is defined multiple times. Since `=` means 'equal by definition', redefinition is forbidden. S can not be evaluated.

In imperative languages like C, Java, etc., S is a

– valid sequence of destructive assignment statements meaning that after executing S the memory cells named `x` and `y` store the values 3 and 2, respectively. No error is raised.

Summing up

Functional definitions are

- genuine mathematical equations.

This allows us to prove

- equality and other relations among functional expressions

applying standard mathematical reasoning.

Proven equality of functions can be used e.g. for optimization by replacing a

- less efficient implementation (called initial algorithm, initial program) by a more efficient one (called final algorithm, final program).

  Example:

  - Initial program: `reverse`
  - Final program: `fastReverse`

Next, we are going to consider this approach in the realm of combinatorially complex problems of functional pearls.

Chapter 4.2

Application: Functional Pearls

Chapter 4.2.1

Functional Pearls: The Very Idea

Functional Pearls: The Very Idea

1. Pick a combinatorially (highly) complex problem P.
2. Solve P by a conceptually straightforward, simple, and intuitive algorithm, the so-called initial algorithm (IA) implemented by some initial program IP, which is
   - obviously correct
   - typically (hopelessly) inefficient.
3. The Functional Perl:
   3.1 Transform IP step by step into some final program (FP) which may be
      - conceptually more complex, less intuitive, not at all obviously correct but (much more) efficient than IP (e.g., feasible instead of practically infeasible, logarithmic instead of quadratic, linear instead of quasi linear,...)

   3.2 Prove that every transformation step preserves the semantics of the program it is applied to (ensuring overall equivalence of the initial and the final program and hence the correctness of the latter).

The Beauty of a Functional Pearl

It is important to note: The functional pearl is

- not the finally resulting (efficient) implementation
- but the calculation and proof process leading to it!

The elegance of the calculation and proof process makes the

- beauty of a functional pearl!

The transformation of

– `reverse` into `fast_reverse` together with the proof of the two functions' equality

can be considered a most simple example of a functional pearl.

Chapter 4.2.2

Functional Pearls: Origin, Background

Functional Pearls: Origin, Background

In 1990, in the course of founding the

- *Journal of Functional Programming*

Richard Bird was asked by the then designated editors-in-chief Simon Peyton Jones and Philip Wadler to contribute a regular column to the journal entitled

- Functional Pearls.

In spirit, this column should follow and emulate the successful series of essays written by Jon Bentley in the 1980s under the title

- Programming Pearls

and published in the

- *Communications of the ACM.*

Functional Pearl Examples

From 1990 to (roughly) 2011 some

- 80 functional pearls have been published in the *Journal of Functional Programming* dealing with
  - Divide-and-conquer
  - Greedy
  - Exhaustive search
  - ...

  and other problems.

Some more were published in proceedings of conferences including editions of the series of the

- *International Conference of Functional Programming*
- *Mathematics of Program Construction*

Roughly a quarter of these pearls have been written by Richard Bird.

A Major Resource of Functional Pearls

In 2011, Richard Bird presented a collection of 30 "revised, polished, and re-polished functional pearls" written by him and others in his monograph:

- Richard Bird. *Pearls of Functional Algorithm Design*. Cambridge University Press, 2011

Here, we consider three of them with a particular focus on the use of equational reasoning for proving the transformation steps correct leading from the initial programs being

- obviously correct but (hopelessly) inefficient

into their final versions being

- much more efficient (but possibly less intuitive):

– Pearl 1: The Smallest Free Number Problem
– Pear 2: Not the Maximum Segment Sum Problem
– Pearl 3: A Simple Sudoku Solver

Go for Equational Reasoning!

...the name of the GoFER language, which is both acronym and name of a functional programming language standing for:

Go F(or) E(quational) R(easoning)

might be considered an indication of the relevance and importance of equational reasoning in the realm of functional programming.

Looking ahead

– In spirit, the program transformation processes follow a correctness by construction approach (cf. Chapter 6.7.1), where correctness of a program constructed by a transformation is ensured by equational reasoning (and other techniques especially inductive reasoning).

Chapter 4.3

The Smallest Free Number

The Smallest Free Number (SFN) Problem

The SFN Problem:

- Let X be a finite set of natural numbers.
- Compute the smallest natural number y that is not in X.

Examples:

The smallest free number of set

- $\{0, 1, 5, 9, 2\}$ is 3.
- $\{0, 1, 2, 3, 18, 19, 22, 25, 42, 71\}$ is 4.
- $\{8, 23, 9, 12, 11, 1, 10, 0, 13, 7, 41, 4, 21, 5, 17, 3, 19, 2, 6\}$ is not immediately obvious!

Chapter 4.3.1

The Initial Algorithm

The SFN Problem

...can easily be solved, if

- X is represented as an increasingly ordered list xs of numbers without duplicates.
- If so, it suffices to look for the first gap in xs.

Illustration:

- Let X be set:
  $\{8, 23, 9, 12, 11, 1, 10, 0, 13, 7, 41, 4, 21, 5, 17, 3, 19, 2, 6\}$
- After sorting (and removing duplicates) we obtain list:
  $[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 17, 19, 21, 23, 41]$
- Looking for the first gap yields:
  The smallest free number of X is 14!

IA: The Initial SFNP Algorithm

...based on the previous observation, the initial algorithm *IA* (for 'Initial Algorithm') for the SFNP problem is the following:

IA: Initial SFNP Algorithm

1. Represent X as a list of integers xs.
2. Sort xs increasingly, while removing all duplicates.
3. Compute the first gap in the list obtained from step 2.

IP_1: The 1st Initial SFNP Program

IA can easily be implemented by a system of two functions called:

- `ssfn` (for 'simple sfn')
- `sap` (for 'search and pick').

IP_1: 1st Initial SFNP Program

```
ssfn :: [Integer] -> Integer
ssfn = (sap 0) . removeDuplicates . quickSort

sap :: Integer -> [Integer] -> Integer
sap n []      = n
sap n (x:xs)
 | n /= x     = n
 | otherwise = sap (n+1) xs
```

IP_2: The 2nd Initial SFNP Program

Note, function `minfree` implements IA, too, giving us a second initial program IP_2 solving the SFN problem.

IP_2: 2nd Initial SFNP Program

```
minfree :: [Nat] -> Nat
minfree xs = head $ ([0..]) \\ xs
```

where

```
(\\) :: Eq a => [a] -> [a] -> [a]
xs \\ ys = filter (`notElem` ys) xs
```

denotes difference on sets (i.e., `xs\\ys` is the list of those elements of `xs` that remain after removing any elements in `ys`) and

```
type Nat = Int
```

the type of natural numbers starting from 0.

Looking at IA, IP_1 and IP_2 in More Detail

...the initial algorithm IA and its implementing programs IP_1 and IP_2 for the SFN problem are (obviously) sound but inefficient:

- IA_1, IP_1: Sorting is not of linear time complexity.
- IP_2: Evaluating `minfree` for a list of length n requires $O(n^2)$ steps in the worst case.

  (Note: Evaluating `minfree [n-1,n-2 .. 0]` requires doublechecking that "$i,\ 0 \leq i \leq n,$ is not an element of list `[n-1,n-2 .. 0]`" and thus $n(n+1)/2$ equality tests.)

The SFN Problem as a Functional Pearl

...starting from IP_2

- develop a new SFNP Algorithm LinSFNP which is of linear time complexity (i.e., linear in the number of elements of the inital set X of natural numbers)
- prove that all steps transforming IA_2 into LinSFNP are correct (i.e., preserve the semantics of IA_2).

Outline

Starting from IP_2, i.e., from `minfree`, we will develop:

1. an array based
2. a divide-and-conquer based

linear time algorithm for the SFN problem.

Both algorithms rely on the following Key Fact (KF):

KF: In `[0..length xs]`, there is a number which is not in `xs`

where `xs` denotes the argument list of natural numbers.

KF implies: The smallest number not in `xs` is given by

- the smallest number not in `filter (<=n) xs`, where `n == length xs`!

Chapter 4.4

Not the Maximum Segment Sum

The Maximum Segment Sum (MSS) Problem

A segment of a list

– is a contiguous subsequence.

The MSS Problem:

– Let L be a list of (positive and negative) integers.
– Compute the maximum of the sums of all possible segments of L.

Example:

Let L be the list

– $[-4,-3,-7,\underbrace{2,1}_{\text{segment } [2,1]},-2,-1,-4]$.

The maximum segment sum of L is

– 3, the sum of the elements of the segment `[2,1]`.

The MSS Problem: Background, Motivation

The MSS Problem

- was considered quite often in the late 1980s mostly as a show- case by programmers to illustrate and demonstrate their favorite style of program development or their particular theorem prover.

In this chapter, however, we consider

- the 'Maximum Non-Segment Sum (MNSS) Problem'

in the spirit of a functional pearl problem.

The Max. Non-Segment Sum (MNSS) Problem

A non-segment of a list

- is a subsequence that is not a segment, i.e., a non-segment has one or more 'holes' in it.

The MNSS Problem:

- Let L be a list of (positive and negative) integers.
- Compute the maximum of the sums of all possible non-segments of L.

Example:

Let L be the list

- $[-4,-3,-7,\overbrace{\underbrace{2,1},-2,\underbrace{-1}}^{\text{segment } [2,1,-2,-1]},-4]$.

non-segment $[2,1]{+}{+}[-1]$

The maximum non-segment sum of L is

- 2, the sum of the elements from the non-segment `[2,1,-1]`.

What does MNSS qualify a Pearl Problem?

...let L be a list of length n.

- There are $O(n^2)$ segments of L.
- There are $O(2^n)$ subsequences of L.

This means there are

- many more non-segments of a list than segments.

This raises the problem:

- Can the maximum non-segment sum be computed in linear time?

This (pearl) problem will be tackled in this chapter.

Chapter 4.4.1

The Initial Algorithm

IA: The Initial MNSS Algorithm

...the MNSS problem can easily be solved by a three-stage process matching the generate/transform/select pattern:

IA: Initial MNSS Algorithm

1. Generate: Compute a list of all non-segments of the argument list.
2. Transform: Compute the sum of all these non-segments.
3. Select: Pick a non-segment whose sum is maximum.

IP: The Initial MNSS Program

IA can straightforwardly be implemented in Haskell as composition of three functions.

IP: Initial MNSS Program

```
mnss :: [Int] -> [Int]
mnss = maximum . map sum . nonsegs
```

where

- `nonsegs` computes a list of all non-segments of the argument list,
- `map sum` computes the sum of all these non-segments,
- `maximum` picks those whose sum is maximum.

Chapter 4.4.2

The Linear Time Algorithm

Work Plan to Derive the Linear Time Alg.

Recall the initial algorithm for the MNSS problem with `nonsegs` replaced by its supporting functions:

```
mnss    = maximum . map sum .
                  extract . filter nonseg . markings
extract = map (map fst . filter snd)
nonseg  = (== N) . foldl step E . map snd
```

Work plan:

- Express `extract . filter nonseg . markings` as an instance of `foldl`.
- Apply then the fusion law of `foldl` to arrive at a better algorithm.

Transforming, transforming, transforming

...and proving semantics preservation of every transformation step.

The Linear Time Algorithm

...for the MNSS Problem:

```
mnss xs
 = fourth (foldl h (start (take 3 xs)) (drop 3 xs))

start [x,y,z]
 = (0, max [x+y+z,y+z,z], max [x,x+y,y], x+z)
```

...less obviously sound for itself compared to the initial algorithm for the MNSS Problem:

```
mnss :: [Int] -> [Int]
mnss = maximum . map sum . nonsegs
```

but efficient and proven correct on the fly of its construction.

Chapter 4.4.3

In Closing

Background

The MSS Problem goes back to Jon R. Bentley:

- Jon R. Bentley. Programming Pearls. Addison-Wesley, 1987.

David Gries and Richard Bird later on presented an invariant assertions and algebraic approach, respectively.

- David Gries. The Maximum Segment Sum Problem. In *Formal Development of Programs and Proofs*. Edsger W. Dijkstra (Ed.), Addison-Wesley, 43-45, 1990.
- Richard Bird. Algebraic Identities for Program Calculation. Computer Journal 32(2):122-126, 1989.

Recent Results

...on the MSS Problem have been presented in:

– Shin-Cheng Mu. The Maximum Segment Sum is Back. In Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Program Manipulation (PEPM 2008), 31-39, 2008.

Chapter 4.5

A Simple Sudoku Solver

Sudoku Puzzles

	3	7	8		6			5
		5	2	7			3	
				3	5		6	8
		1					9	3
		2		5		4		
5	7					8		
2	1		5	6				
	4			2	1	5		
6			3		7	2	4	

Fill in the grid so that every row, every column, and every 3×3 box contains the digits $1 - 9$. There's no maths involved. You solve the puzzle with reasoning and logic.

The Independent Newspaper

Chapter 4.5.1

Two Initial Algorithms

IA_1, IA_2: Two Initial Soduko Algorithms

There are two straightforward (brute force) approaches to solving a Sudoku puzzle:

IA_1: 1st Initial Soduko Algorithm:

- Construct a list of all correctly completed grids.
- Subsequently, test the input grid against them to identify those whose non-blank entries match the given ones.

IA_2: 2nd Initial Sodudo Algorithm:

- Start with the input grid and construct all possible choices for the blank entries.
- Then compute all grids that arise from making every possible choice and filter the result for the valid ones.

In the following we proceed with IA_2 for solving the Sudoku problem.

Preliminaries

...data types for modelling Soduko puzzles:

- $m \times n$-matrix: A list of m rows of the same length n.

```
type Matrix a = [Row a]
type Row a    = [a]
```

- Grid: A 9×9-matrix of digits.

```
type Grid  = Matrix Digit
type Digit = Char
```

- Valid digits: '1' to '9'; '0' stands for a blank.

```
digits = ['1'..'9']
blank  = (== '0')
```

In the following, we assume that the input grid is valid, i.e.,

- it contains only digits and blanks
- no digit is repeated in any row, column or box.

IP: The Initial Soduko Program

...IA_2 can straightforwardly be implemented in Haskell as a composition of three functions matching the generate/filter pattern:

IP: Initial Sudoku Program

```
solve = filter valid . expand . choices
```

```
choices :: Grid -> Matrix Choices
expand  :: Matrix Choices -> [Grid]
valid   :: Grid -> Bool
```

where

- Generate:
  - `choices` constructs all choices for the blank entries of the input grid,
  - `expand` computes all grids that arise from making every possible choice,
- Filter: `filter valid` selects all the valid grids.

Completing the Initial Program (1)

...we start with introducing the type synonym

```
type Choices = [Digit]
```

whose values will represent the set of choices.

Based on this, we next define the subsidiary functions of `solve`, i.e., the functions

- `choices`
- `expand`
- `valid`

Completing the Initial Program (2)

Implementing `choices`:

```
choices :: Grid -> Matrix Choices
choices  = map (map choice)
choice d = if blank d then digits else [d]
```

Intuitively

- If the cell is blank, then all digits are installed as possible choices.
- Otherwise there is no choice and a singleton is returned.

Completing the Initial Program (3)

Implementing expand:

```
expand :: Matrix Choices -> [Grid]
expand :: cp . map cp

cp :: [[a]] -> [[a]]     (cp ≙ cartesian_product)
cp [] = [[]]
cp (xs:xss) = [x:ys | x <- xs, ys <- cp xss]
```

Intuitively

- Expansion is a Cartesian product, i.e., a list of lists yielded by the function cp, e.g., cp[[1,2] ,[3] ,[4,5]] ->> [[1,3,4] ,[1,3,5] ,[2,3,4] ,[2,3,5]]
- map cp returns a list of all possible choices for each row.
- cp . map cp, finally, installs each choice for the rows in all possible ways.

Completing the Initial Program (4)

Implementing `valid`:

```
valid :: Grid -> Bool
valid g = all nodups (rows g) &&
          all nodups (cols g) &&
          all nodups (boxs g)

nodups :: Eq a => [a] -> Bool
nodups [] = True
nodups (x:xs) = all (x/=) xs && nodups xs
```

(nodups $\widehat{=}$ no_duplicates)

Intuitively

– A grid is valid, if no row, column or box contains duplicates.

Completing the Initial Program (5)

Implementing `rows` and `columns`:

```
rows :: Matrix a -> Matrix a
rows = id

cols :: Matrix a -> Matrix a
cols [xs]       = [ [x] | x <- xs]
cols (xs:xss) = zipWith (:) xs (cols xss)
```

Intuitively

- `rows` is the identity function, since the grid is already given as a list of rows.
- `columns` computes the transpose of a matrix.

Completing the Initial Program (6)

Implementing `boxs`:

```
boxs :: Matrix a -> Matrix a
boxs = map ungroup . ungroup . map cols .
       group . map group

group :: [a] -> [[a]]
group [] = []
group xs = take 3 xs : group (drop 3 xs)

ungroup :: [[a]] -> [a]
ungroup = concat
```

Intuitively

- `group` splits a list into groups of three.
- `ungroup` takes a grouped list and ungroups it.
- `group . map group` produces a list of matrices; transposing each matrix and ungrouping them yields the boxes.

Completing the Initial Program (7)

...illustrating the effect of `boxs` for the (4×4)-case, when `group` splits a list into groups of two:

$$\begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{pmatrix} \rightarrow \begin{pmatrix} \begin{pmatrix} ab & cd \\ ef & gh \end{pmatrix} \\ \begin{pmatrix} ij & kl \\ mn & op \end{pmatrix} \end{pmatrix} \rightarrow \begin{pmatrix} \begin{pmatrix} ab & ef \\ cd & gh \end{pmatrix} \\ \begin{pmatrix} ij & mn \\ kl & op \end{pmatrix} \end{pmatrix}$$

Note: Eventually, the elements of the 4 boxes show up as the elements of the 4 rows, where they can easily be accessed.

Wholemeal Programming

Instead of

- thinking about matrices in terms of indices, and
- doing arithmetic on indices to identify rows, columns, and boxes

the preceding approach has gone for functions which

- treat a matrix as a complete entity in itself.

Geraint Jones coined the notion

- wholemeal programming

for this style of programming.

Wholemeal programming

- helps avoiding indexitis and
- encourages lawful program construction.

Lawful Programming

Lemma 4.5.1.1

The laws (A), (B), and (C) hold on arbitrary $(N \times N)$-matrices, in particular on (9×9)-grids:

```
rows . rows = id                    (A)
cols . cols = id                    (B)
boxs . boxs = id                    (C)
```

This means, all 3 functions are involutions.

Lemma 4.5.1.2

The laws (D), (E), and (F) hold on $(N^2 \times N^2)$-matrices:

```
map rows . expand = expand . rows   (D)
map cols . expand = expand . cols   (E)
map boxs . expand = expand . boxs   (F)
```

A Quick Analysis of the Initial Program

...suppose that half of the entries (cells) of the input grid are fixed.

Then there are about 9^{40}, or

147.808.829.414.345.923.316.083.210.206.383.297.601

grids to be constructed and checked for validity!

This is hopeless!

Chapter 4.5.2

Pruning the Initial Algorithm

Optimizing the Initial Algorithm

1st Optimization: Pruning the matrix of choices:

Idea

- Remove any choices from a cell c that occurs as a singleton entry in the row, column or box containing c.

Hence, we are seeking for a function

```
prune :: Matrix Choices -> Matrix Choices
```

which satisfies

```
filter valid . expand
  = filter valid . expand . prune
```

and implements the above idea.

Pruning a Row

Pruning a row

```
pruneRow :: Row Choices -> Row Choices
pruneRow row = map (remove fixed) row
               where fixed = [d | [d] <- row]

remove xs ds
  = if singleton ds then ds else ds \\ xs
```

Intuitively

– `remove` removes choices from any choice that is not fixed.

Laws for pruneRow, nodups, and cp

- The function pruneRow satisfies law (G):

```
filter nodups . cp
  = filter nodups . cp . pruneRow             (G)
```

- The functions nodups and cp satisfy laws (H) and (I):

  If f is an involution, i.e., `f . f = id`, then

```
filter (p.f) = map f . filter p . map f       (H)
filter (all p) . cp = cp . map (filter p)     (I)
```

Rewriting `filter valid . expand`

...using `nodups`, `boxs`, `cols`, and `rows`.

We can prove:

Lemma 4.5.2.1

```
filter valid . expand
  = filter (all nodups . boxs) .
    filter (all nodups . cols) .
    filter (all nodups . rows) . expand
```

(Note: The order of the 3 filters on the right hand side above is not relevant.)

Work plan: Apply each of the filters to `expand`.

...doing this requires some reasoning which we exemplify for the `boxs` case.

Proof Sketch of Lemma 4.5.2.1: boxs Case (1)

```
  filter (all nodups . boxs) . expand
= {(H), since boxs . boxs = id}
  map boxs . filter (all nodups) . map boxs . expand
= {(F)}
  map boxs . filter (all nodups) . expand boxs
= {definition of expand}
  map boxs . filter (all nodups) . cp . map cp . boxs
= {(I), and map f . map g = map (f . g)}
  map boxs . cp . map (filter nodups . cp) . boxs
= {(G)}
  map boxs . cp . map (filter nodups . cp . pruneRow) . boxs
```

Proof Sketch of Lemma 4.5.2.1: boxs Case (2)

```
= {(I)}
  map boxs . filter (all nodups) . cp .
                map cp . map pruneRow . boxs
= {definition of expand}
  map boxs . filter (all nodups) . expand .
                map pruneRow . boxs
= {(H) in the form map f . filter p =
                      filter (p . f) . map f}
  filter (all nodups . boxs) . map boxs . expand .
                map pruneRow . boxs
= {(F)}
  filter (all nodups . boxs) .  expand . boxs .
                map pruneRow . boxs
```

Summing up

Overall, we have shown:

Lemma 4.5.2.2

```
filter (all nodups . boxs) . expand
  = filter (all nodups . boxs) .
                    expand . pruneBy boxs
```
, where

```
pruneBy f = f . map pruneRow . f
```

Repeating the same calculation for `rows` and `cols` we get:

Lemma 4.5.2.3

```
filter valid . expand
  = filter valid . expand . prune
```
, where

```
prune
  = pruneBy boxs . pruneBy cols . pruneBy rows
```

Implementation of `solve` after the 1st Opt.

Implementation of `solve` after the 1st Optimization (pruning-improved):

```
solve = filter valid . expand . prune . choices
```

Note: Pruning can be done more than once.

- After each round of pruning some choices might be resolved into singletons allowing the next round of pruning to remove even more impossible choices.
- For simple Sudoku problems repeated rounds of pruning will eventually yield the solution of the input Sudoku problem.

Tuning the Solver Further

...based on the following idea:

– Combine pruning with expanding the choices for a single cell only at a time, called single-cell expansion.

Which cell to expand?

– Any cell with the smallest number of choices for which there are at least 2 choices.

Note: If there is a cell with no choices then the Sudoku problem is unsolvable (from a pragmatic point of view, such cells should be identified quickly).

Empowering the Function expand

...we replace the function `expand` by a new version

```
expand = concat . map expand . expand1     (J)
```

where `expand1` expands the choices of a single cell only, which is defined next.

Defining expand1

Think of a cell containing cs choices as sitting in the middle of a row row, i.e., row = row1 ++ [cs] ++ row2, in the matrix of choices, with rows rows1 above it and rows rows2 below it:

```
expand1 :: Matrix Choices -> [Matrix Choices]
expand1 rows
 = [rows1 ++ [row1 ++ [c] : row2] ++ rows2 | c<-cs]
where
 (rows1,row:rows2) = break (any smallest) rows
 (row1, cs:row2)   = break smallest row
 smallest cs       = length cs == n
 n                 = minimum (counts rows)
 counts = filter (/=1) . map length . concat

break p xs
 = (takeWhile (not . p) xs, dropWhile (not . p) xs)
```

Remarks on expand1

- The value `n` is the smallest number of choices, not equal to `1` in any cell of the matrix of choices.
- If the matrix contains only singleton choices, then `n` is the minimum of the empty list, which is not defined.
- The standard function `break p` splits a list into two.
- `break (any smallest) rows` thus breaks the matrix into two lists of rows with the head of the second list being some row that contains a cell with the smallest number of choices.
- Another application of `break` then breaks this row into two sub-rows, with the head of the second being the element `cs` with the smallest number of choices.
- Each possible choice is installed and the matrix reconstructed.
- If there are no choices, `expand1` returns an empty list.

Completeness and Safety of a Matrix

The definition of `n` implies that (J) only holds when

- applied to matrices with at least one non-singleton choice.

This suggests: A matrix is

- complete, if all choices are singletons,
- unsafe, if the singleton choices in any row, column or box contain duplicates.

Note:

- Incomplete and unsafe matrices can never lead to valid grids.
- A complete and safe matrix of choices determines a unique valid grid.

Testing Completeness and Safety

Completeness and safety can be tested as follows:

- Completeness Test:

  ```
  complete = all (all single)
  ```

  where `single` is the test for a singleton list.

- Safety Test:

  ```
  safe m = all ok (rows m) &&
           all ok (cols m) &&
           all ok (boxs m)

  ok row = nodups [d | [d] <- row]
  ```

Equational Reasoning

...allows us to show: If a matrix is safe but incomplete, we have:

```
    filter valid . expand
  = {since expand = concat . map expand . expand1
     on incomplete matrices}
    filter valid . concat . map expand . expand1
  = {since filter p . concat = concat . map (filter p)}
    concat . map (filter valid . expand) . expand1
  = {since filter valid . expand =
                    filter valid . expand . prune}
    concat . map (filter valid . expand . prune) .
                                              expand1
```

Implementation of solve after the 2nd Opt.

Defining search by

```
search = filter valid . expand . prune
```

we have for safe but incomplete matrices the equality

```
search . prune = concat . map search . expand1
```

This leads us to the final

Implementation of solve, after the 2nd Optimization (single cell-improved):

```
solve = search . choices
search m
  | not (safe m) = []
  | complete m'  = [map (map head) m']
  | otherwise    = concat (map search (expand1 m'))
    where m' = prune m
```

Chapter 4.5.3

In Closing

Quality and Performance Assessment

The final version of the Sudoku solver has been tested on various Sudoku puzzles available at

- haskell.org/haskellwiki/Sudoku

It is reported that the solver

- turned out to be most useful, and
- competitive to (many) of the about a dozen different Haskell Sudoku solvers available at this site.

While many of the other solvers use arrays and monads, and reduce or transform the problem to

- Boolean satisfiability, constraint satisfaction, model-checking, etc.

the solver presented here seems unique in terms of length, conceptual simplicity and that it has been derived in part by

- equational reasoning!

Chapter 4.6

References, Further Reading

Chapter 4: Basic Reading

- Richard Bird. *Fifteen Years of Functional Pearls*. In Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming (ICFP 2006), 215, 2006.
- Richard Bird. *How to Write a Functional Pearl*. Invited presentation at the 11th ACM SIGPLAN International Conference on Functional Programming (ICFP 2006), 2006. http://icfp06.cs.uchicago.edu/bird-talk.pdf
- Richard Bird. *Pearls of Functional Algorithm Design*. Cambridge University Press, 2011. (Chapter 1, The smallest free number; Chapter 11, Not the maximum segment sum; Chapter 19, A simple Sudoku solver)
- Jeremy Gibbons. *Functional Pearls – An Editor's Perspective.* www.cs.ox.ac.uk/people/jeremy.gibbons/pearls/

Chapter 4: Selected Further Reading (1)

- Jon R. Bentley. *Programming Pearls*. Addison-Wesley, 1987.
- Jon R. Bentley. *Programming Pearls*. Addison-Wesley, 2nd edition, 2000. (Excerpt of the book online available from `www.cs.bell-labs.com/cm/cs/pearls`)
- Richard Bird. *Algebraic Identities for Program Calculation*. Computer Journal 32(2):122-126, 1989.
- Richard Bird. *Thinking Functionally with Haskell*. Cambridge University Press, 2015. (Chapter 5, A simple Sudoku solver; Chapter 6.6, The maximum segment sum)

Chapter 4: Selected Further Reading (2)

- Antonie J.T. Davie. *An Introduction to Functional Programming Systems using Haskell*. Cambridge University Press, 1992. (Chapter 10, Applicative Program Transformations)
- Kees Doets, Jan van Eijck. *The Haskell Road to Logic, Maths and Programming*. Texts in Computing, Vol. 4, King's College, UK, 2004. (Chapter 1.9, Haskell Equations and Equational Reasoning)
- Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2007. (Chapter 13, Reasoning about programs)

From Type to Higher-Order Type Classes

- Type Classes like
  - `Eq`, `Ord`, `Num`, `Enum`, `Show`, `Monoid`,...

  have types as instances, e.g.,
  - `String`, `Int`, `[Int]`, `Maybe Int`, `Either Int Bool`,...

  which must satisfy a set of laws.
- Higher-Order Type Classes like
  - `Functor`, `Applicative`, `Monad`, `Arrows`,...

  have type constructors as instances, e.g.,
  - `[]`, `(->)`, `((->) Int)`, `Maybe`, `Either`, `Either Int`, `(,)`, `(,,)`, `(,,,)`,...

  which must satisfy a set of laws.

Example

Compare:

- Type class Monoid:

```
class Monoid m where
  mempty  :: m
  mappend :: m -> m -> m
  mconcat :: [m] -> m
  -- Default implementation
  mconcat  = foldr mappend mempty
```

  plus monoid laws.

  Note: Usage of m implies: m must be a type!

- Type constructor class Functor:

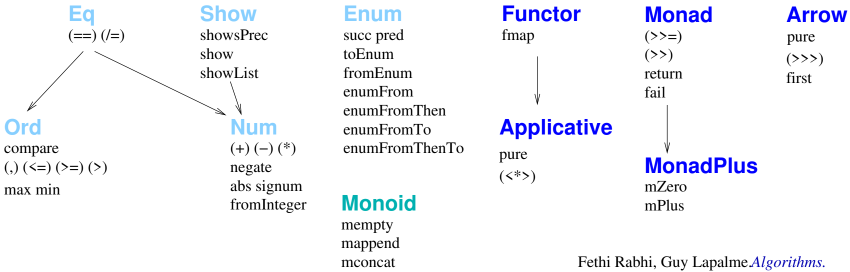
```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

  plus functor laws.

  Note: Usage of f implies: f must be a type constructor!

Type Classes, Type Constructor Classes

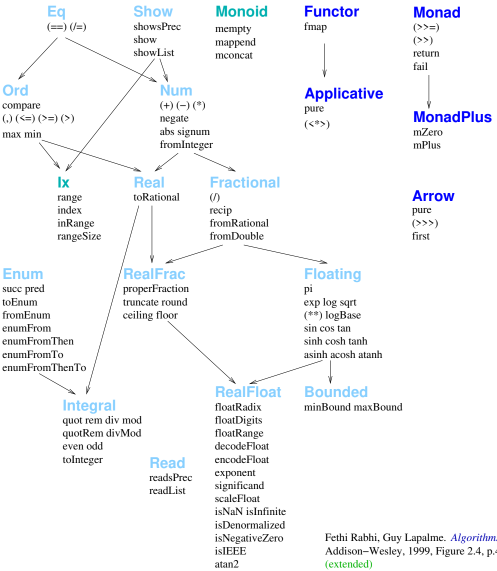
...as part of the Haskell'98 type class hierarchy:



Fethi Rabhi, Guy Lapalme. *Algorithms.* Addison–Wesley, 1999, Figure 2.4, p.46 (extended)

Type Classes, Type Constructor Classes

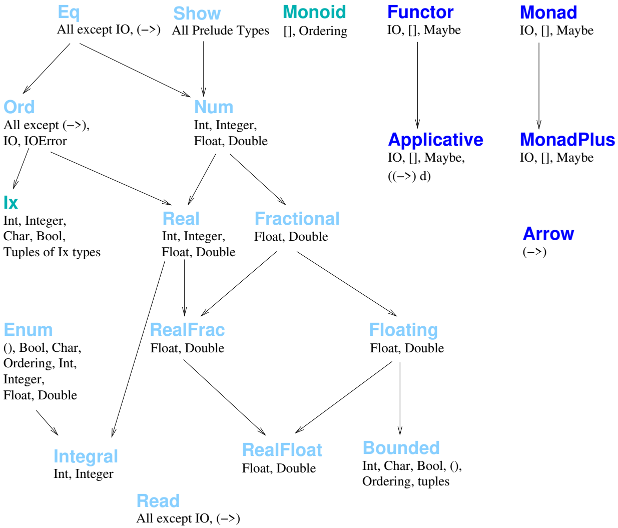
...a larger section of the Haskell'98 type class hierarchy:



Fethi Rabhi, Guy Lapalme. *Algorithms.*
Addison−Wesley, 1999, Figure 2.4, p.46
(extended)

Type (Constr.) Classes w/ Predef. Instances

...of a section of the Haskell'98 type class hierarchy:



Paul Hudak. *The Haskell School of Expression.* Cambridge University Press, 2000, p.156 (extended)

Haskell: A Research Vehicle & Moving Target

...therefore, an update on the Haskell'98 Type Class Hierarchy:

Haskell'98

Functor
fmap

↓

Applicative
pure
(<*>)

Monad
(>>=)
(>>)
return
fail

↓

MonadPlus
mZero
mPlus

Arrow
pure
(>>>)
first

Haskell'98 Onwards

Functor
fmap
(<$) :: a –> f b –> f a
(<$) = fmap . const

↓

Applicative
pure
(<*>)
(*>) :: f a –> f b –> f b
a1 *> a2 = (id <$ a1) <*> a2
(<*) :: f a –> f b –> f a
(<*) = liftA2 const

↓

Monad
(>>=)
(>>)
return
fail

↓

MonadPlus
mZero
mPlus

Category
id :: cat a a
(.) :: cat b c –> cat a b –> cat a c

↓

Arrow
arr :: (b –> c) –> (b 'arr' c)
first :: (b 'arr' c) –> ((b,d) 'arr' ((c,d))
second :: (b 'arr' c) –> ((d,b) 'arr' (d,c))
(***) :: (b 'arr' c) –> (b' 'arr' c') –> ((b,b') 'arr' (c,c'))
(&&&) :: (b 'arr' c) –> (b 'arr' c') –> (c,c'))

where 'arr' is a two–ary type variable

...for more information, check out:

https://wiki.haskell.org/Typeclassopedia

Chapter 9

Monoids

...in medias res.

Chapter 9.2

The Type Class Monoid

The Type Class Monoid

...monoids are instances of type class Monoid obeying the monoid laws.

Type Class Monoid

```
class Monoid m where
  mempty  :: m
  mappend :: m -> m -> m
  mconcat :: [m] -> m
  -- Default implementation
  mconcat  = foldr mappend mempty
```

Monoid Laws

```
mempty `mappend` x           = x             (MonoL1)
x `mappend` mempty           = x             (MonoL2)
(x `mappend` y) `mappend` z  =
  x `mappend` (y `mappend` z)                (MonoL3)
```

Informally

Monoids are types with

- a binary operation `mappend`.
- a value `mempty`.
- a unary operation `mconcat` reducing a list of monoid values to a single monoid value using `mappend`.

The monoid laws

- MonoL1 and MonoL2 require that `mempty` is a left-unit and a right-unit of `mappend`, hence a unit.
- MonoL3 requires that `mappend` is associative.

Programmer obligation:

- Programmers must prove that their instances of `Monoid` satisfy the monoid laws.

Note

- The value `mempty` can be considered a nullary function or a polymorphic constant.
- The name `mappend` is often misleading; for most monoids the effect of `mappend` cannot be thought in terms of "appending" values.
- Usually, it is wise to think of `mappend` in terms of a function that takes two `m` values and maps them to another `m` value.
- Commutativity of `mappend` is not required by the monoid laws.

Chapter 9.3

Monoid Examples

Chapter 9.3.1

The List Monoid

The List Monoid

...making [a] an instance of type class Monoid:

```
instance Monoid [a] where
  mempty  = []
  mappend = (++)
```

Proof obligation: The monoid laws

Lemma 9.3.1.1 (Soundness of List Monoid)

For every instance of type variable a, the [a] instance of Monoid satisfies the three monoid laws MonoL1, MonoL2, and MonoL3.

...[a] is thus a proper instance of Monoid, the so-called list monoid.

Example: Applying the List Monoid Operations

```
mempty ->> []

[1,2,3] `mappend` [4,5,6] ->> [1,2,3,4,5,6]

[1,2,3] `mappend` mempty ->> [1,2,3] ++ [] ->> [1,2,3]

"Advanced " `mappend` "Functional " `mappend`
   "Programming"
      ->> "Advanced Functional Programming"

"Advanced " `mappend` ("Functional " `mappend`
   "Programming"
      ->> "Advanced Functional Programming")

("Advanced " `mappend` "Functional ") `mappend`
   "Programming"
      ->> "Advanced Functional Programming"
```

Chapter 9.3.2/9.3.3

Numerical/Boolean Monoids

Numerical/Boolean Monoids

Numerical types and the Boolean type `Bool` are equipped with more than one associative operation and corresponding unit. E.g.:

Associative operations:

- Addition `(+)`, multiplication `(*)` for numerical types
- Disjunction `(||)`, conjunction `(&&)` for `Bool`

with units:

- `0` for `(+)`, `1` for `(*)`
- `False` for `(||)`, `True` for `(&&)`

Hence, these types allow different instances; check-out full course notes for details.

Chapter 9.3.4

The Ordering Monoid

The Ordering Monoid

...making type `Ordering` an instance of type class `Monoid`:

```
instance Monoid Ordering where
  mempty          = EQ
  LT `mappend` _ = LT
  EQ `mappend` x = x
  GT `mappend` _ = GT
```

Proof obligation: The monoid laws

Lemma 9.3.4.1 (Soundness of Ordering Monoid)

The `Ordering` instance of `Monoid` satisfies the three monoid laws MonoL1, MonoL2, and MonoL3.

...`Ordering` is thus a proper instance of `Monoid`, the so-called ordering monoid.

Note

The `mappend` operation of the `Ordering` instance of `Monoid`:

- is not commutative:

```
LT `mappend` GT ->> LT
GT `mappend` LT ->> GT
```

- induces a 'lexicographical' comparison of two list arguments.

...we will make use of the latter observation in the following example.

Example: Applying the Monoid Operations (1)

The two definitions of `lengthCompare` without and with `mappend`:

```
lengthCompare :: String -> String -> Ordering
lengthCompare x y
  = let a = length x `compare` length y -- 1st priority
        b = x `compare` y               -- 2nd priority
    in if a == EQ then b else a

lengthCompare :: String -> String -> Ordering
lengthCompare x y = (length x `compare` length y)
                         `mappend` (x `compare` y)
```

...are equivalent what can be proved using the properties of `mappend`.

Example: Applying the Monoid Operations (2)

...as suggested both versions of `lengthCompare` yield:

```
lengthCompare "his" "ants" ->> LT
```

(since string "his" is shorter than string "ants") and

```
lengthCompare "his" "ant"  ->> GT
```

(since string "his" is lexicographically larger than "ant").

Example: Applying the Monoid Operations (3)

...additional comparison criteria can easily be added and prioritirized.

The below extension of `lengthCompare`, e.g., takes the number of vowels as second most important comparison criterion:

```
lengthCompareExt :: String -> String -> Ordering
lengthCompareExt x y
  = (length x `compare` length y)  -- 1st priority
    `mappend` (vowels x `compare` vowels y)
                                   -- 2nd priority
    `mappend` (x `compare` y)      -- 3rd priority
 where vowels = length . filter (`elem` "aeiou")
```

As suggested we get:

```
lengthCompareExt "songs" "abba" ->> GT
lengthCompareExt "song" "abba"  ->> LT
lengthCompareExt "sono" "abba"  ->> GT
lengthCompareExt "sono" "sono"  ->> EQ
```

Chapter 9.4

Summary, Looking ahead

Summary: Commutativity of `mappend`

...unlike associativity, commutativity of the `mappend` operation is not required by the monoid laws for monoids.

For some monoids, commutativity of `mappend` holds, e.g., the:

– sum, product, any, all monoids.

For other instances it does not hold, e.g., the:

– list, ordering monoids.

Summary: Using Monoids

Monoids are most useful for defining

- folds over values of structured data

since folding requires an associative operation.

Folding seems obviousand natural for

- lists

but is possible, too, for the values of many other structured data, e.g.:

- trees

This motivates the introduction of the type (constructor) class `Foldable` as collection of all type constructors whose values can be folded (cf. module `Data.Foldable`; qualified import because of name clashes with the standard prelude).

Looking ahead: Type Constructor Classes

...type classes of a new kind:

```
class Foldable f where
 foldr   :: (a -> b -> b) -> b -> f a -> b
 foldl   :: (a -> b -> a) -> a -> f b -> a
 foldMap :: (Monoid m, Foldable t) =>
                               (a -> m) -> t a -> m
 ...
```

Note:

- `f` and `t` are applied to type variables, here `a` and `b`. This means, `f` and `t` are (1-ary) type constructors, not types.
- `Foldable` is thus a type constructor class, a special type class.
- The `foldl`, `foldr` operations of `Foldable` extend folding of lists to folding of values of other 'foldable' structured data while allowing to reuse the operation names.

Looking ahead: The List Type Constructor []

...is one important instance of Foldable:

```
foldr :: (a -> b -> b) -> b -> [] a -> b
foldl :: (a -> b -> a) -> a -> [] b -> a
```

where `Data.Foldable.foldl` and `Data.Foldable.foldr` are defined in terms of their counterparts `foldl` and `foldr` introduced in Chapter 10.5, LVA 185.A03 Funktionale Programmierung.

`Foldable` is the first example of this new kind of higher-order type classes called type constructor classes of which we consider more examples next: `Functor`, `Applicative`, `Monad`, and `Arrow` (cf. Chapters 10, 11, 12, and 13).

Chapter 9.5

References, Further Reading

Chapter 9: Basic Reading

- Miran Lipovača. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, 2011. (Chapter 12, Monoids)
- Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Chapter 13, Data Structures – Monoids)

Chapter 9: Selected Further Reading

Paul Hudak. *The Haskell School of Expression – Learning Functional Programming through Multimedia*. Cambridge University Press, 2000. (Chapter 13.4.3, Defining New Type Classes for Behaviors)

Chapter 10

Functors

Chapter 10.1

Motivation

Mapping

...over values is a typical and recurring task, e.g., over:

- Lists

```
mapL :: (a -> b) -> ([] a) -> ([] b)
mapL g []     = []
mapL g (l:ls) = g l : mapL g ls
```

- Trees

```
data Tree a = Leaf a | Node a (Tree a) (Tree a)
```

```
mapT :: (a -> b) -> Tree a -> Tree b
mapT g (Leaf v) = Leaf (g v)
mapT g (Node v l r)
  = Node (g v) (mapT g l) (mapT g r)
```

Higher-Order Type (Constructor) Classes

..the conceptual similarity of tasks performed by functions like

- `mapL`, `mapT`

suggests bundling all types whose values can be mapped over in a unique type class:

- `Functor`

offering an (over-loaded) function:

- fmap

having `mapL`, `mapT`, and many more as specific instance implementations.

Note: `Functor` is a representative of a new kind of type classes, a higher-order type class, a so-called:

- type constructor class

This means

...types, whose values can be mapped over compositionally, with a neutral element, like e.g.:

- Lists with mapL and id

```
g :: a -> b, h :: b -> c
mapL g []       = []
mapL g (x:xs)   = (g x) : mapL g xs
mapL (h . g) xs = mapL h (mapL g xs)  (compositional)
mapL id xs      = xs                (neutral element)
```

- Trees with mapT and id

```
g :: a -> b, h :: b -> c
data Tree a = Leaf a | Node a (Tree a) (Tree a)
mapT g (Leaf v)     = Leaf (g v)
mapT g (Node v l r) = Node (g v) (mapT g l) (mapT g r)
mapT (h . g) t = mapT h (mapT g t)    (compositional)
mapT id t      = t                  (neutral element)
```

should be made instances of type constructor class Functor.

Chapter 10.2

The Type Constructor Class Functor

The Type Constructor Class Functor

...functors are instances of the type constructor class Functor obeying the functor laws.

Type Constructor Class Functor

```
class Functor f where
 fmap :: (a -> b) -> f a -> f b
```

Functor Laws

```
fmap id      = id                    (FL1)
fmap (h . g) = fmap h . fmap g       (FL2)
```

Programmer obligation

– Programmers must prove that their instances of Functor satisfy the functor laws.

Note

...argument `f` of `Functor` is applied to type variables, i.e.:

- `f` is a 1-ary type constructor variable (that is applied to type variables `a` and `b`), not a type variable.

...instances of `Functor` (like of other type constructor classes) are thus type constructors, not types.

The functor laws ensure:

- `fmap` preserves the "shape of the container type."
- `fmap` does not regroup the contents of the container.

The Functor Laws in more Detail

...with added type information:

Type Constructor Class Functor

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Functor Laws

$$\underbrace{\texttt{fmap}\ \underbrace{\texttt{id}}_{::\ a\ \to\ a}}_{::\ f\ a\ \to\ f\ a} \quad = \quad \underbrace{\texttt{id}}_{::\ f\ a\ \to\ f\ a} \qquad \text{(FL1)}$$

(id over-loaded!)

$$\underbrace{\texttt{fmap}\ \underbrace{(\underbrace{\texttt{h}}_{::\ c\ \to\ b}\ .\ \underbrace{\texttt{g}}_{::\ a\ \to\ c})}_{::\ a\ \to\ b}}_{::\ f\ a\ \to\ f\ b} \quad = \quad \underbrace{\underbrace{\texttt{fmap}\ \underbrace{\texttt{h}}_{::\ c\ \to\ b}}_{::\ f\ c\ \to\ f\ b}\ .\ \underbrace{\texttt{fmap}\ \underbrace{\texttt{g}}_{::\ a\ \to\ c}}_{::\ f\ a\ \to\ f\ c}}_{::\ f\ a\ \to\ f\ b} \qquad \text{(FL2)}$$

The Curried and Uncurried View of fmap

Curried view: fmap takes

- a polymorphic function `g :: a -> b` and yields a polymorphic function `g' :: f a -> f b`.

Example:

```
newtype Month a = M a

instance Functor Month where
 fmap g (M v) = M (g v)
```

```
g :: Int -> String                g' :: Month Int -> Month String
g 1 = "January"                   g' (M 1) = M "January"
...                               ...
g 12 = "December"                 g' (M 12) = M "December"
```

$$\texttt{fmap}\ \underbrace{\texttt{g}}_{\texttt{:: Int -> String}} \ \texttt{->>}\ \underbrace{\texttt{g'}}_{\texttt{:: Month Int -> Month String}}$$

Uncurried view: fmap takes

- a polymorphic function `g :: a -> b` and a functor value `va :: f a` and yields a new functor value `vb :: f b`.

Example: $\texttt{fmap g}\ \underbrace{\texttt{(M 8)}}_{\texttt{:: Month Int}}\ \texttt{->> fmap (M (g 8)) ->>}\ \underbrace{\texttt{M "August"}}_{\texttt{:: Month String}}$

Chapter 10.3

Functor Examples

Chapter 10.3.1

The Identity Functor

The Identity Functor

...making the 1-ary type constructor Id an instance of Functor (conceptually the simplest functor):

```
newtype Id a = Id a

instance Functor Id where
  fmap g (Id x) = Id g x
```

Proof obligation: The functor laws

Lemma 10.3.1.1 (Soundness of Identity Functor)

The Id instance of Functor satisfies the two functor laws FL1 and FL2.

...Id is thus a proper instance of Functor, the so-called identitiy functor.

Chapter 10.3.2

The List Functor

The List Functor

...making the 1-ary type constructor [] an instance of Functor:

```
instance Functor [] where
  fmap g []     = []
  fmap g (l:ls) = g l : fmap g ls
```

Proof obligation: The functor laws

Lemma 10.3.2.1 (Soundness of List Functor)

The [] instance of Functor satisfies the two functor laws FL1 and FL2.

...[] is thus a proper instance of Functor, the so-called list functor.

Chapter 10.3.3

The Maybe Functor

The Maybe Functor

...making the 1-ary type constructor Maybe an instance of Functor:

```
data Maybe a = Nothing | Just a

instance Functor Maybe where
 fmap g (Just x) = Just (g x)
 fmap g Nothing  = Nothing
```

Proof obligation: The functor laws

Lemma 10.3.3.1 (Soundness of Maybe Functor)

The Maybe instance of Functor satisfies the two functor laws FL1 and FL2.

...Maybe is thus a proper instance of Functor, the so-called maybe functor.

Example: Applying the Functor Operation

```
fmap (++ "Programming") (Just "Functional")
  ->> Just "Functional Programming"

fmap (++ "Programming") Nothing
  ->> Nothing
```

Chapter 10.3.4

The Either Functor

The Either Functor

...making the 1-ary type constructor (Either a) an instance of Functor:

```
data Either a b = Left a | Right b

instance Functor (Either a) where
  fmap g (Right x) = Right (g x)
  fmap g (Left x)  = Left x
```

Note: The type constructor Either has two arguments, i.e., is a 2-ary type constructor. Hence, only the partially evaluated 1-ary type constructor (Either a) can be made an instance of Functor.

Proof Obligation: The Functor Laws

Lemma 10.3.4.1 (Soundness of Either Functor)

The `(Either a)` instance of `Functor` satisfies the two functor laws FL1 and FL2.

...`(Either a)` is thus a proper instance of `Functor`, the so-called either functor.

Example: Applying the Functor Operation

```
fmap length (Right "Programming")
  ->> Right 11

fmap length (Left "Programming")
  ->> Left "Programming"
```

Chapter 10.3.5

The Map Functor

The Map Functor

...making the 1-ary type constructor `((->) d)` an instance of `Functor`:

```
instance Functor ((->) d) where    -- d reminding
  fmap g h = (\x -> g (h x))       -- to domain
```

Note: Like `Either`, also `(->)` is a 2-ary type constructor, i.e., has two arguments. Hence, only the partially evaluated type constructor `((->) d)` can be made an instance of `Functor`, since it is a 1-ary type constructor.

Proof Obligation: The Functor Laws

Lemma 10.3.5.1 (Soundness of Map Functor)

The `((->) d)` instance of `Functor` satisfies the two functor laws FL1 and FL2.

...`((->) d)` is thus a proper instance of `Functor`, the so-called map functor.

The Map Functor in more Detail

...with added type information:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

instance Functor ((->) d) where
  fmap   g              h        =  (\x -> g (h x))
     :: (a -> b)   :: ((->) d) a    :: d         :: d
                                                 :: a
                                               :: b
                                    :: ((->) d) b
```

Note: `fmap` defined (as above) by

```
fmap g h = (\x -> g (h x))
```

means just function composition: `fmap g h = (g . h)`

The Instance Declaration of the Map Functor

...reconsidered.

The observation on the meaning of `fmap` allows us to define the instance declaration of `((->) d)` directly as ordinary functional composition:

```
instance Functor ((->) d) where
  fmap = (.)
```

Notes on the Map Functor

...for the map functor `((->) d)` the type of the generic operation `fmap` of the type constructor class `Functor`

```
fmap :: (Functor f) => (a -> b) -> f a -> f b
```

specializes to:

```
fmap :: (a -> b) -> (((->) d) a) -> (((->) d) b)
```

Using infix notation for `(->)`, this can equivalently be written as:

```
fmap :: (a -> b) -> (d -> a) -> (d -> b)
```

where `fmap` can be implemented by:

$$\texttt{fmap}\ \underbrace{\texttt{g}}_{::\ \texttt{a -> b}}\ \underbrace{\underbrace{\texttt{h}}_{::\ \texttt{d -> a}} = (\underbrace{\texttt{g}}_{::\ \texttt{a -> b}}\ .\ \underbrace{\texttt{h})}_{::\ \texttt{d -> a}}}_{::\ \texttt{(a -> b) -> }\underbrace{\texttt{(d -> a) ->}}_{::\ \texttt{d -> b}}\texttt{ (d -> b)}}$$

Example: Applying the Functor Operation (1)

```
Main>:t fmap (*3) (+100)
fmap (*3) (+100) :: (Num a) => a -> a

fmap (*3) (+100) 1          ->>  303

(*3) `fmap` (+100) $ 1      ->>  303
(*3)    .   (+100) $ 1      ->>  303

fmap (show . (*3)) (+100) 1 ->> "303"
```

Note: Using `fmap` as an infix operator emphasizes the equality of `fmap` and functional composition `(.)` for the map functor `((->) d)`.

Example: Applying the Functor Operation (2)

...recalling the generic type of `fmap`:

```
fmap :: (Functor f) => (a -> b) -> f a -> f b
```

we get:

```
Main>:t fmap (*2)
fmap (*2) :: (Num a, Functor f) => f a -> f a
```

```
Main>:t fmap (replicate 3)
fmap (replicate 3) :: (Functor f) => f a -> f [a]
```

where

```
replicate :: Int -> a -> [a]
replicate n x
 | n <= 0      = []
 | otherwise = x : replicate (n-1) x
```

Example: Applying the Functor Operation (3)

```
fmap (replicate 3) [1,2,3,4]
 ->> [[1,1,1],[2,2,2],[3,3,3],[4,4,4]]

fmap (replicate 3) (Just 4)
 ->> Just [4,4,4]

fmap (replicate 3) (Right "fun")
 ->> Right ["fun","fun","fun"]

fmap (replicate 3) Nothing
 ->> Nothing

fmap (replicate 3) (Left "fun")
 ->> Left "fun"
```

Example: Applying the Functor Operation (4)

Applying `fmap` to n-ary maps (e.g., `(*)`, `(++)`, `\x y z -> ...`, ...) instead of 1-ary maps (e.g., `replicate 3`, `(*3)`, `(+100)`, ...) as so far, we get:

```
fmap (*) (Just 3) ->> Just ((*) 3)

fmap (++) (Just "fun") :: Maybe ([Char] -> [Char])
fmap compare (Just 'a') :: Maybe (Char -> Ordering)
fmap compare "A list of chars" :: [Char -> Ordering]
fmap (\x y z -> x + y / z) [3,4,5,6]
                  :: (Fractional a) => [a -> a -> a]

a = fmap (*) [1,2,3,4] :: [Int -> Int]

fmap (\f -> f 9) a ->> [9,18,27,36]
```

Note

...some of the previous examples showed

- lifting

of a map of type

- `(a -> b)`

to type

- `(f a -> f b)`

by `fmap`. This again shows that `fmap`

```
fmap :: (Functor f) => (a -> b) -> f a -> f b
```

can be thought of in two ways. As a map which takes a map `g :: a -> b` and

1. lifts `g` to a new function `h :: f a -> f b` operating on functor values ⇝ curried view.
2. a functor value `v :: f a` and maps `g` over `v` ⇝ uncurried view.

Chapter 10.3.6

The Input/Output Functor

The Input/Output Functor

...making the 1-ary type constructor IO for input/output an instance of Functor:

```
instance Functor IO where
  fmap g action = do result <- action
                     return (g result)
```

Proof obligation: The functor laws

Lemma 10.3.6.1 (Soundness of IO Functor)

The IO instance of Functor satisfies the two functor laws FL1 and FL2.

...IO is thus a proper instance of Functor, the so-called input/output (IO) functor.

Example: Applying the Functor Operation (1)

...the two versions of program main

```
main =
 do line <- fmap reverse getLine
    putStrLn $ "You said " ++ line ++ " backwards!"
    putStrLn $ "Yes, you said " ++ line ++ " backwards!"

main =
 do line <- getLine
    let line' = reverse line
    putStrLn $ "You said " ++ line' ++ " backwards!"
    putStrLn $ "Yes, you said " ++ line' ++ " backwards!"
```

which differ in using and not using fmap are equivalent.

Example: Applying the Functor Operation (2)

```
import Data.Char
import Data.List
```

The expressions

```
do line <- fmap (intersperse '-' . reverse .
                 map toUpper) getLine
   putStrLn line
```

and

```
(\xs -> intersperse '-' (reverse (map toUpper xs)))
```

have the same input/output effect.

Applied e.g. to the input string `"fun prog"`, the output is in both cases the string `"G-O-R-P- -N-U-F"`.

Chapter 10.4

References, Further Reading

Chapter 10: Basic Reading

- Miran Lipovača. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, 2011. (Chapter 7, Making Our Own Types and Type Classes – The Functor Type Class)
- Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000. (Chapter 18.1, The Functor Class)

Chapter 10: Selected Further Reading

- Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Chapter 10, Code Case Study: Parsing a Binary Data Format – Introducing Functors, Writing a Functor Instance for Parse, Using Functors for Parsing)
- Peter Pepper, Petra Hofstedt. *Funktionale Programmierung*. Springer-V., 2006. (Kapitel 11.1, Kategorien, Funktoren und Monaden)
- Fethi Rabhi, Guy Lapalme. *Algorithms – A Functional Programming Approach*. Addison-Wesley, 1999. (Chapter 2.8.3, Type classes and inheritance)

Chapter 11

Applicative Functors

Chapter 11.1

The Type Constructor Class Applicative

The Type Constructor Class Applicative

...applicatives are instances of the type constructor class Applicative obeying the applicative laws.

Type Constructor Class Applicative

```
class (Functor f) => Applicative f where
 pure  :: a -> f a                    -- Value ‘lifting’:
                                      -- Making an appli-
                                      -- cative value
  (<*>) :: f (a -> b) -> f a -> f b  -- Mapping over
```

Applicative Laws

```
pure id <*> v               = v                  (AL1)
pure (.) <*> u <*> v <*> w = u <*> (v <*> w)    (AL2)
pure g <*> pure x           = pure (g x)         (AL3)
u <*> pure y                = pure ($ y) <*> u   (AL4)
```

Note

...applicatives must be functors and hence 1-ary type constructors.

Intuitively

- `pure` takes a value of any type and returns an applicative value.
- `(<*>)` takes a functor value, which has a function in it, and another functor value, which has a value in it. It extracts the function from the first functor and maps it over the value of the second one.

Programmer obligation

- Programmers must prove that their instances of `Applicative` satisfy the applicative laws.

Selected Applicative Laws in more Detail

...with added type information:

Class Applicative

```
class (Functor f) => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

Applicative Laws

$$\underbrace{\underbrace{\texttt{pure}\ \underbrace{\texttt{id}}_{::\ \texttt{a -> a}}}_{::\ \texttt{f (a -> a)}}\ \texttt{<*>}\ \underbrace{\texttt{v}}_{::\ \texttt{f a}}}_{::\ \texttt{f a}} = \underbrace{\texttt{v}}_{::\ \texttt{f a}} \qquad \text{(AL1)}$$

$$\underbrace{\underbrace{\texttt{pure}\ \underbrace{\texttt{g}}_{::\ \texttt{a -> b}}}_{::\ \texttt{f (a -> b)}}\ \texttt{<*>}\ \underbrace{\texttt{pure}\ \underbrace{\texttt{x}}_{::\ \texttt{a}}}_{::\ \texttt{f a}}}_{::\ \texttt{f b}} = \underbrace{\texttt{pure}\ (\underbrace{\underbrace{\texttt{g}}_{::\ \texttt{a -> b}}\ \underbrace{\texttt{x}}_{::\ \texttt{a}}}_{::\ \texttt{b}})}_{::\ \texttt{f b}} \qquad \text{(AL3)}$$

Syntactic Sugar: Infix Operator <$>

...as alias for fmap for more compelling operation sequences involving both fmap and (<*>).

The infix alias (<$>) of fmap of Functor:

```
(<$>) :: (Functor f) => (a -> b) -> f a -> f b
g <$> x = fmap g x
```

Example: Using (<$>) as infix operator, we can write:

```
(++) <$> Just "Functional " <*> Just "Programming"
  ->> Just "Functional Programming"
```

instead of the less compelling variants using the prefix operator fmap:

```
(fmap (++) Just "Functional ") <*> Just "Programming"
  ->> Just "Functional Programming"
```

...or its infix variant ‘fmap‘:

```
((++) `fmap` Just "Functional ") <*> Just "Programming"
  ->> Just "Functional Programming"
```

Note

...overloading `f` and defining `(<$>)` by:

```
(<$>) :: (Functor f) => (a -> b) -> f a -> f b
f <$> x = fmap f x
```

would be valid, too, since the context allows to decide if `f` is used as type constructor (`f`) or as argument (`f`).

Utility Maps for Applicatives

Utility Maps:

```
liftA2 :: (Applicative f) =>
               (a -> b -> c) -> f a -> f b -> f c
liftA2 g a b = g <$> a <*> b
```

```
sequenceA :: (Applicative f) => [f a] -> f [a]
sequenceA []     = pure []
sequenceA (x:xs) = (:) <$> x <*> sequenceA xs
```

```
sequenceA :: (Applicative f) => [f a] -> f [a]
sequenceA = foldr (liftA2 (:)) (pure [])
```

Examples:

```
fmap (\x -> [x]) (Just 4)        ->> Just [4]
liftA2 (:) (Just 3) (Just [4]) ->> Just [3,4]
(:) <$> Just 3 <*> Just 4        ->> Just [3,4]
```

Chapter 11.2

Applicative Examples

Chapter 11.2.1

The Identity Applicative

The Identity Applicative

...making the 1-ary type constructor `Id` an instance of `Applicative` (conceptually the simplest applicative):

```
newtype Id a = Id a

instance Applicative Id where
  pure               = Id
  Id g <*> (Id x) = Id (g x)
```

Note: `g` plays the rôle of the applicative functor.

Proof obligation: The applicative laws

Lemma 11.2.1.1 (Soundness of Identity Applicative)

The `Id` instance of `Applicative` satisfies the four applicative laws AL1, AL2, AL3, and AL4.

...`Id` is thus a proper instance of `Applicative`, the so-called identity applicative.

The Identity Applicative in more Detail

...with added type information:

```
pure  :: (Applicative f) => a -> f a
(<*>) :: (Applicative f) => f (a -> b) -> f a -> f b
```

```
instance Applicative Id where
```

$$\underbrace{\texttt{pure}}_{\texttt{:: a -> Id a}} \;\texttt{=}\; \underbrace{\texttt{Id}}_{\texttt{:: a -> Id a}}$$

$$\underbrace{\underbrace{\texttt{Id}\ \underbrace{\texttt{g}}_{\texttt{:: (a -> b)}}}_{\texttt{:: Id (a -> b)}} \;\texttt{<*>}\; \underbrace{\texttt{Id}\ \underbrace{\texttt{x}}_{\texttt{:: a}}}_{\texttt{:: Id a}}}_{\texttt{:: Id b}} \;\texttt{=}\; \underbrace{\texttt{Id}\ \underbrace{(\underbrace{\texttt{g}}_{\texttt{:: a -> b}}\ \underbrace{\texttt{x}}_{\texttt{:: a}})}_{\texttt{:: b}}}_{\texttt{:: Id b}}$$

Chapter 11.2.2

The List Applicative

The List Applicative

...making the 1-ary type constructor [] an instance of Applicative:

```
instance Applicative [] where
  pure x     = [x]
  gs <*> xs = [g x | g <- gs, x <- xs]
```

Proof obligation: The applicative laws

Lemma 11.2.2.1 (Soundness of List Applicative)

The [] instance of Applicative satisfies the four applicative laws AL1, AL2, AL3, and AL4.

...[] is thus a proper instance of Applicative, the so-called list applicative.

The List Applicative in more Detail

...with added type information:

```
pure  :: (Applicative f) => a -> f a
(<*>) :: (Applicative f) => f (a -> b) -> f a -> f b

instance Applicative [] where
      pure          x    = [ x ]
 :: a -> [] a    :: a      :: a
                         :: [] a

       gs          <*>   xs    = [  g        x | g <- gs, x <- xs]
:: [] (a -> b)        :: [] a   :: a -> b :: a
                                     :: b
                                           :: [] b
```

Example: Applying the Applicative Operations (1)

```
pure "Hallo" :: String        ->> ["Hallo"]
pure "Hallo" :: Maybe String ->> Just "Hallo"

[(*0),(+100),(^2)] <*> [1,2,3]
 ->> [ f x | f <- [(*0),(+100),(^2)], x <- [1,2,3] ]
 ->> [0,0,0,101,102,103,1,4,9]

[(+),(*)] <*> [1,2] <*> [3,4]
 ->> [ f x | f <- [(+),(*)], x <- [1,2] ] <*> [3,4]
 ->> [(1+),(2+),(1*),(2*)] <*> [3,4]
 ->> [ f x | f <- [(1+),(2+),(1*),(2*)], x <- [3,4] ]
 ->> [4,5,5,6,3,4,6,8]
```

Example: Applying the Applicative Operations (2)

```
filter (>50) $ (*) <$> [2,5,10] <*> [8,10,11]
 ->> filter (>50) $ (fmap (*) [2,5,10]) <*> [8,10,11]
 ->> filter (>50) $ [(2*),(5*),(10*)] <*> [8,10,11]
 ->> filter (>50) $ [ f x | f <- [(2*),(5*),(10*)],
                            x <- [8,10,11] ]
 ->> filter (>50) $ [16,20,22,40,50,55,80,100,110]
 ->> filter (>50) [16,20,22,40,50,55,80,100,110]
 ->> [55,80,100,110]
```

Example: Applying the Applicative Operations (3)

The preceeding example using `filter` shows that expressions using list comprehension:

```
[x*y | x <- [2,5,10], y <- [8,10,11]]
 ->> [16,20,22,40,50,55,80,100,110]
```

...can alternatively be written using `(<$>)` and `<*>` and vice versa:

```
(*) <$> [2,5,10] <*> [8,10,11]
 ->> [16,20,22,40,50,55,80,100,110]
```

Chapter 11.2.3/11.2.4

The Maybe/Either Applicatives

The Maybe Applicative

...making the 1-ary type constructor Maybe an instance of Applicative:

```
instance Applicative Maybe where
  pure                       = Just
  Nothing  <*> _             = Nothing
  (Just g) <*> something = fmap g something
```

Note: g plays the rôle of the applicative functor.

Proof obligation: The applicative laws

Lemma 11.2.3.1 (Soundness of Maybe Applicative)

The Maybe instance of Applicative satisfies the four applicative laws AL1, AL2, AL3, and AL4.

...Maybe is thus a proper instance of Applicative, the so-called maybe applicative.

The Maybe Applicative in more Detail

...with added type information:

```
pure  :: (Applicative f) => a -> f a
(<*>) :: (Applicative f) => f (a -> b) -> f a -> f b
fmap  :: (Functor f)     => (a -> b) -> f a -> f b
```

```
instance Applicative Maybe where
```

$$\underbrace{\texttt{pure}}_{\texttt{:: a -> Maybe a}} = \underbrace{\texttt{Just}}_{\texttt{:: a -> Maybe a}}$$

$$\underbrace{\underbrace{\texttt{Nothing}}_{\texttt{:: Maybe (a -> b)}} \; \texttt{<*>} \; \underbrace{\texttt{\_}}_{\texttt{:: Maybe a}}}_{\texttt{:: Maybe b}} = \underbrace{\texttt{Nothing}}_{\texttt{:: Maybe b}}$$

$$\underbrace{\underbrace{\texttt{(Just g)}}_{\texttt{:: Maybe (a -> b)}} \; \texttt{<*>} \; \underbrace{\texttt{something}}_{\texttt{:: Maybe a}}}_{\texttt{:: Maybe b}} = \underbrace{\underbrace{\texttt{fmap g}}_{\texttt{:: a -> b}} \; \underbrace{\texttt{something}}_{\texttt{:: Maybe a}}}_{\texttt{:: Maybe b}}$$

Example: Applying the Applicative Operations (1)

```
Just (+3) <*> Just 9
 ->> fmap (+3) (Just 9)
 ->> Just 12

Just (+3) <*> Nothing
 ->> fmap (+3) Nothing
 ->> Nothing

Just (++ "good ") <*> Just "morning"
 ->> fmap (++ "good ") "morning"
 ->> Just "good morning"

Just (++ "good ") <*> Nothing
 ->> fmap (++ "good ") Nothing
 ->> Nothing

Nothing <*> Just "good "
 ->> Nothing
```

Example: Applying the Applicative Operations (2)

```
pure (+) <*> Just 3  <*> Just 5
 ->> Just (+) <*> Just 3  <*> Just 5
 ->> (fmap (+) Just 3) <*> Just 5
 ->> Just (3+) <*> Just 5
 ->> Just 8

pure (+) <*> Just 3  <*> Nothing
 ->> Just (+) <*> Just 3  <*> Nothing
 ->> fmap (+) Just 3 <*> Nothing
 ->> Just (3+) <*> Nothing
 ->> fmap (3+) Nothing
 ->> Nothing
```

Exercise 11.2.4.1: The Either Applicative

1. Make type constructor `(Either a)` an instance of `Applicative`.
2. Show that the defining equations of the applicative operations `pure` and `(<*>)` of `(Either a)` are type correct. Annotate the laws with the (most general) type information applying.
3. Prove that your `(Either a)` instance of `Applicative` satisfies the applicative laws.

Chapter 11.2.5

The Map Applicative

The Map Applicative

...making the 1-ary type constructor ((-> d) an instance of Applicative:

```
instance Applicative ((->) d) where
 pure x  = (\_ -> x)
 g <*> h = \x -> g x (h x)
```

Proof obligation: The applicative laws

Lemma 11.2.5.1 (Soundness of Map Applicative)

The ((->) d) instance of Applicative satisfies the four applicative laws AL1, AL2, AL3, and AL4.

...(->) d) is thus a proper instance of Applicative, the so-called map applicative.

The Map Applicative in more Detail

...with added type information:

```
pure  :: (Applicative f) => a -> f a
(<*>) :: (Applicative f) => f (a -> b) -> f a -> f b

instance Applicative ((->) d) where
 pure x  = (\_ -> x)
```

$$\texttt{pure}\ \underbrace{\texttt{x}}_{::\ \texttt{a}}\ \texttt{=}\ \underbrace{\texttt{(}\underbrace{\texttt{\\_}}_{::\ \texttt{d}}\ \texttt{->}\ \underbrace{\texttt{x}}_{::\ \texttt{a}}\texttt{)}}_{::\ \texttt{((->) d) a}}$$

$$\underbrace{\underbrace{\texttt{g}}_{::\ \texttt{((->) d) (a -> b)}}}_{::\ \texttt{d -> (a -> b)}}\ \texttt{<*>}\ \underbrace{\underbrace{\texttt{h}}_{::\ \texttt{((->) d) a}}}_{::\ \texttt{d -> a}}\ \texttt{=}\ \underbrace{\underbrace{\texttt{\\}\underbrace{\texttt{x}}_{::\ \texttt{d}}\ \texttt{->}\ \underbrace{\texttt{g}\ \underbrace{\texttt{x}}_{::\ \texttt{d}}\ \underbrace{\texttt{(h}\ \underbrace{\texttt{x}}_{::\ \texttt{d}}\texttt{)}}_{::\ \texttt{a}}}_{::\ \texttt{b}}}_{::\ \texttt{d -> b}}}_{::\ \texttt{((->) d) b)}}$$

Example: Applying the Applicative Operations

```
pure 3 "Hello"
 ->> (pure 3) "Hello"          (left-assoc. of expr.)
 ->> (\_ -> 3) "Hello"
 ->> 3
```

```
(+) <$> (+3) <*> (*100) :: (Num a) => a -> a
```

```
(+) <$> (+3) <*> (*100) $ 5 :: Int
 ->> (fmap (+) (+3)) <*> (*100) $ 5
 ->> ((+) . (+3)) <*> (*100) $ 5
 ->> (\x -> ((+) . (+3)) x ((*100) x)) $ 5
 ->> ((+) . (+3)) 5 ((*100) 5)
 ->> (+)((+3) 5) (5*100)
 ->> (+)(5+3) 500
 ->> (+) 8 500
 ->> (8+) 500
 ->> 8+500
 ->> 508 :: Int
```

Chapter 11.2.7

The Input/Output Applicative

The Input/Output Applicative

...making the 1-ary type constructor IO an instance of Applicative:

```
instance Applicative IO where
 pure    = return
 a <*> b = do g <- a
              x <- b
              return (g x)
```

Proof obligation: The applicative laws

Lemma 11.2.7.1 (Soundness of IO Applicative)

TheIO instance of Applicative satisfies the four applicative laws AL1, AL2, AL3, and AL4.

...IO is thus a proper instance of Applicative, the so-called input/output (IO) applicative.

The Input/Output Applicative in more Detail

...with added type information:

```
pure  :: (Applicative f) => a -> f a
(<*>) :: (Applicative f) => f (a -> b) -> f a -> f b
```

```
instance Applicative IO where
```

$$\underbrace{\texttt{pure}}_{\texttt{:: a -> IO a}} \;=\; \underbrace{\texttt{return}}_{\texttt{:: a -> IO a}}$$

$$\underbrace{\texttt{a}}_{\texttt{:: IO (a -> b)}} \;\texttt{<*>}\; \underbrace{\texttt{b}}_{\texttt{:: IO a}} \;=\; \texttt{do}\; \underbrace{\texttt{g}}_{\texttt{:: a -> b}} \;\texttt{<-}\; \underbrace{\texttt{a}}_{\texttt{:: IO (a -> b)}}$$

$$\underbrace{\texttt{x}}_{\texttt{:: a}} \;\texttt{<-}\; \underbrace{\texttt{b}}_{\texttt{:: IO a}}$$

$$\underbrace{\texttt{return}\;\underbrace{(\underbrace{\texttt{g}}_{\texttt{:: a -> b}}\;\underbrace{\texttt{x}}_{\texttt{:: a}})}_{\texttt{:: b}}}_{\texttt{:: IO b}}$$

Example: Applying the Applicative Operations

...the following two versions of `myAction` are equivalent:

```
myAction :: IO String
myAction = do a <- getLine
              b <- getLine
              return $ a++b
```

```
myAction :: IO String
myAction = (++) <$> getLine <*> getLine
```

Type and effect of `myAction'` are similar but slightly different:

```
myAction' :: IO ()
myAction' =
 do a <- (++) <$> getLine <*> getLine
    putStrLn $ "Concatenation yields: " ++ a
```

Chapter 11.3

References, Further Reading

Chapter 11: Basic Reading

Miran Lipovača. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, 2011. (Chapter 11, Applicative Functors)

Chapter 14

Kinds

Kinds

Just as values also

- types
- type constructors

have types themselves, so-called:

- kinds.

Kinds of types and type constructors are represented by expressions over the symbol $*$ (read as "star" or as "type").

Chapter 14.1

Kinds of Types

Types

...i.e., nullary type constructors, type constructors accepting no type arguments, have kind $*$. Intuitively, $*$ indicates that types are 'concrete', 'final'.

In GHCi, kinds of types (and type constructors) can be computed and displayed using the command "`:k`".

Examples:

```
ghci> :k Int
Int :: *
```

```
ghci> :k (Char,String)
(Char,String) :: *
```

```
ghci> :k [Float]
[Float] :: *
```

```
ghci> :k (Int -> Int)
(Int -> Int) :: *
```

Chapter 14.2

Kinds of Type Constructors

Type Constructors

...take types as arguments to produce concrete types.

Examples:

The 1-ary type constructor Maybe, the 2-ary type constructor Either, and the 3-ary type constructor Tree:

```
data Maybe a    = Nothing | Just a
data Either a b = Left a  | Right b
data Tree a b c = Leaf a b
                  | Node a (Tree a b c) (Tree a b c)
```

produce for a, b, and c chosen Int, String, and Bool, respectively, the concrete types:

```
Maybe Int                :: *      -- a concrete type
Either Int String        :: *      -- a concrete type
Tree Int String Bool     :: *      -- a concrete type
```

...of kind $*$.

Kinds of Type Constructors

Like concrete types, type constructors have kinds, too, reflecting the number of their type arguments.

Examples:

```
ghci> :k Maybe
Maybe :: * -> *          -- a type constructor accepting
                         -- a concrete type as argument
                         -- and yielding a concrete type.

ghci> :k Either
Either :: * -> * -> *  -- a type constructor accepting
                       -- two concrete types as arguments
                       -- and yielding a concrete type.

ghci> :k Tree
Tree :: * -> * -> * -> * -- a type constructor accep-
                         -- ting three concrete types...
```

Kinds of Partially Evaluated Type Constructors

Like functions, type constructors can be partially evaluated, too, resulting in different kinds.

Examples:

```
ghci> :k Either
Either :: * -> * -> *  -- a type constructor accepting
                       -- two concrete types as arguments
                       -- and yielding a concrete type.

ghci> :k Either Int
Either Int :: * -> *   -- a type constructor accepting
                       -- one concrete type as argument
                       -- and yielding a concrete type.

ghci> :k Either Int Char
Either Int Char :: *    -- a concrete type.
```

Chapter 14.3

References, Further Reading

Chapter 14: Basic Reading

- Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000. (Chapter 18.5, Type Class Type Errors, Kinds of Types)
- Simon Peyton Jones (Ed.). *Haskell 98: Language and Libraries. The Revised Report*. Cambridge University Press, 2003. (Chapter 4.1.1, Kinds; Chapter 4.6, Kind Inference)

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