

Fortgeschrittene funktionale Programmierung

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

(Stand: 18.05.2020)

Jens Knoop



Technische Universität Wien
Information Systems Engineering
Compilers and Languages



Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1/1969

Table of Contents

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

Table of Contents (1)

Part I: Motivation

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

Part II: Programming Principles

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

Table of Contents (2)

Part II: Programming Principles

- Chap. 2: Programming with Streams (cont'd)
 - 2.3 Boosting Performance
 - 2.3.1 Motivation
 - 2.3.2 Stream Programming combined with Münchhausen Principle
 - 2.3.3 Stream Programming combined with Memoization
 - 2.3.4 Summary
 - 2.4 Stream Diagrams
 - 2.5 Pitfalls, Remedies
 - 2.5.1 Termination, Domain-specific Knowledge
 - 2.5.2 Lifting, Undecidability
 - 2.5.3 Livelocks, Lazy Patterns
 - 2.6 Summary, Looking ahead
 - 2.7 References, Further Reading

Table of Contents (3)

- ▶ Chap. 3: Programming with Higher-Order Functions: Algorithm Patterns
 - 3.1 Divide-and-Conquer
 - 3.2 Backtracking Search
 - 3.3 Priority-first Search
 - 3.4 Greedy Search
 - 3.5 Dynamic Programming
 - 3.6 Dynamic Programming vs. Memoization
 - 3.7 References, Further Reading
- ▶ Chap. 4: Equational Reasoning for Functional Pearls
 - 4.1 Equational Reasoning
 - 4.2 Application: Functional Pearls
 - 4.2.1 Functional Pearls: The Very Idea
 - 4.2.2 Functional Pearls: Origin, Background

Table of Contents (4)

- ▶ Chap. 4: Equational Reasoning for Funct. Pearls (cont'd)
 - 4.3 The Smallest Free Number
 - 4.3.1 The Initial Algorithm
 - 4.3.2 An Array-based Algorithm and Two Variants
 - 4.3.3 A Divide-and-Conquer Algorithm
 - 4.3.4 In Closing
 - 4.4 Not the Maximum Segment Sum
 - 4.4.1 The Initial Algorithm
 - 4.4.2 The Linear Time Algorithm
 - 4.4.3 In Closing
 - 4.5 A Simple Sudoku Solver
 - 4.5.1 Two Initial Algorithms
 - 4.5.2 Pruning the Initial Algorithm
 - 4.5.3 In Closing
 - 4.6 References, Further Reading

Table of Contents (5)

Part III: Quality Assurance

► Chap. 5: Testing

5.1 Motivation

5.2 Defining Properties

5.3 Testing against Abstract Models

5.4 Testing against Algebraic Specifications

5.5 Controlling Test Data Generation

5.5.1 Controlling Quantification over Value Domains

5.5.2 Controlling the Size of Test Data

5.5.3 Example: Test Data Generators at Work

5.6 Monitoring, Reporting, and Coverage

5.7 Implementation of QuickCheck

5.8 Summary

5.9 References, Further Reading

Table of Contents (6)

- ▶ Chap. 6: Verification
 - 6.1 Inductive Proof Principles on Natural Numbers
 - 6.1.1 Natural Induction
 - 6.1.2 Strong Induction
 - 6.1.3 Excursus: Fibonacci and The Golden Ratio
 - 6.2 Inductive Proof Principles on Structured Data
 - 6.2.1 Induction and Recursion
 - 6.2.2 Structural Induction
 - 6.3 Inductive Proofs on Algebraic Data Types
 - 6.3.1 Inductive Proofs on Haskell Trees
 - 6.3.2 Inductive Proofs on Haskell Lists
 - 6.3.3 Inductive Proofs on Partial Haskell Lists
 - 6.4 Proving Properties of Streams
 - 6.4.1 Inductive Proofs on Haskell Stream Approximants
 - 6.4.2 Inductive Proofs on Haskell List and Stream Approximants
 - 6.5 Proving Equality of Streams
 - 6.5.1 Approximation
 - 6.5.2 Coinduction

Table of Contents (7)

- ▶ Chap. 6: Verification (cont'd)
 - 6.6 Fixed Point Induction
 - 6.7 Verified Programming, Verification Tools
 - 6.7.1 Correctness by Construction
 - 6.7.2 Provers, Proof-Assistents, Verified Programming
 - 6.8 References, Further Reading

Part IV: Advanced Language Concepts

- ▶ Chap. 7: Functional Arrays
 - 7.1 Motivation
 - 7.2 Functional Arrays
 - 7.2.1 Static Arrays
 - 7.2.2 Dynamic Arrays
 - 7.3 Summary
 - 7.4 References, Further Reading

Table of Contents (8)

- ▶ Chap. 8: Abstract Data Types
 - 8.1 Motivation
 - 8.2 Stacks
 - 8.3 Queues
 - 8.4 Priority Queues
 - 8.5 Tables
 - 8.6 Displaying ADT Values in Haskell
 - 8.7 Summary
 - 8.8 References, Further Reading
- ▶ Chap. 9: Monoids
 - 9.1 Motivation
 - 9.2 The Type Class Monoid
 - 9.3 Monoid Examples
 - 9.3.1 The List Monoid
 - 9.3.2 Numerical Monoids
 - 9.3.3 Boolean Monoids
 - 9.3.4 The Ordering Monoid

Table of Contents (9)

- ▶ Chap. 9: Monoids (cont'd)
 - 9.4 Summary, Looking ahead
 - 9.5 References, Further Reading
- ▶ Chap. 10: Functors
 - 10.1 Motivation
 - 10.2 The Type Constructor Class Functor
 - 10.3 Functor Examples
 - 10.3.1 The Identity Functor
 - 10.3.2 The List Functor
 - 10.3.3 The Maybe Functor
 - 10.3.4 The Either Functor
 - 10.3.5 The Map Functor
 - 10.3.6 The Input/Output Functor
 - 10.4 References, Further Reading

Table of Contents (10)

- ▶ 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. 12: Monads
 - 12.1 Motivation
 - 12.2 The Type Constructor Class Monad
 - 12.3 Syntactic Sugar: The do-Notation

Table of Contents (11)

► Chap. 12: Monads (cont'd)

12.4 Monad Examples

12.4.1 The Identity Monad

12.4.2 The List Monad

12.4.3 The Maybe Monad

12.4.4 The Either Monad

12.4.5 The Map Monad

12.4.6 The State Monad

12.4.7 The Input/Output Monad

12.5 Monadic Programming

12.5.1 Folding Trees

12.5.2 Numbering Tree Labels

12.5.3 Renaming Tree Labels

12.6 Monad-Plusses

12.6.1 The Type Constructor Class MonadPlus

12.6.2 The List Monad-Plus

12.6.3 The Maybe Monad-Plus

12.7 Summary

12.8 References, Further Reading

Table of Contents (12)

▶ Chap. 13: Arrows

13.1 Motivation

13.2 The Type Constructor Class Arrow

13.3 The Map Arrow

13.4 Application: Modelling Electronic Circuits

13.5 An Update on the Haskell Type Class Hierarchy

13.6 Summary

13.7 References, Further Reading

▶ Chap. 14: Kinds

14.1 Kinds of Types

14.2 Kinds of Type Constructors

14.3 References, Further Reading

Table of Contents (13)

Part V: Applications

► Chap. 15: Parsing

15.1 Motivation

15.2 Combinator Parsing

15.2.1 Primitive Parsers

15.2.2 Parser Combinators

15.2.3 Universal Combinator Parser Basis

15.2.4 Structure of Combinator Parsers

15.2.5 Writing Combinator Parsers: Examples

15.3 Monadic Parsing

15.3.1 The Parser Monad

15.3.2 Parsers as Monadic Operations

15.3.3 Universal Monadic Parser Basis

15.3.4 Utility Parsers

15.3.5 Structure of a Monadic Parser

15.3.6 Writing Monadic Parsers: Examples

15.4 Summary

15.5 References, Further Reading

Table of Contents (14)

► Chap. 16: Logic Programming Functionally

16.1 Motivation

16.1.1 On the Evolution of Programming Languages

16.1.2 Functional vs. Logic Languages

16.1.3 A Curry Appetizer

16.1.4 Outline

16.2 The Combinator Approach

16.2.1 Introduction

16.2.2 Diagonalization

16.2.3 Diagonalization with Monads

16.2.4 Filtering with Conditions

16.2.5 Indicating Search Progress

16.2.6 Selecting a Search Strategy

16.2.7 Terms, Substitutions, Unification, and Predicates

16.2.8 Combinators for Logic Programs

16.2.9 Writing Logic Programs: Two Examples

16.3 In Closing

16.4 References, Further Reading

Table of Contents (15)

► Chap. 17: Pretty Printing

17.1 Motivation

17.2 The Simple Pretty Printer

17.2.1 Basic Document Operators

17.2.2 Normal Forms of String Documents

17.2.3 Printing Trees

17.3 The Prettier Printer

17.3.1 Algebraic Documents

17.3.2 Algebraic Representations of Document Operators

17.3.3 Multiple Layouts of Algebraic Documents

17.3.4 Normal Forms of Algebraic Documents

17.3.5 Improving Performance

17.3.6 Utility Functions

17.3.7 Printing XML-like Documents

17.4 The Prettier Printer Code Library

17.4.1 The Prettier Printer

17.4.2 The Tree Example

17.4.3 The XML Example

17.5 Summary

17.6 References, Further Reading

Table of Contents (16)

► Chap. 18: Functional Reactive Programming

18.1 Motivation

18.2 An Imperative Robot Language

18.2.1 The Robot's World

18.2.2 Modelling the Robot's World

18.2.3 Modelling Robots

18.2.4 Modelling Robot Commands as State Monad

18.2.5 The Imperative Robot Language

18.2.6 Defining a Robot's World

18.2.7 Robot Graphics: Animation in Action

18.3 Robots on Wheels

18.3.1 The Setting

18.3.2 Modelling the Robots' World

18.3.3 Classes of Robots

18.3.4 Robot Simulation in Action

18.3.5 Examples

18.4 In Conclusion

18.5 References, Further Reading

Table of Contents (17)

Part VI: Extensions, Perspectives

- ▶ Chap. 19: Extensions: Parallel and 'Real World' Functional Programming

 - 19.1 Parallelism in Functional Languages

 - 19.2 Haskell for 'Real World' Programming

 - 19.3 References, Further Reading

- ▶ Chap. 20: Conclusions, Perspectives

 - 20.1 Research Venues, Research Topics, and More

 - 20.2 Programming Contest

 - 20.3 In Conclusion

 - 20.4 References, Further Reading

- ▶ References

Appendix

- ▶ A Mathematical Foundations

Table of Contents (18)

► A Mathematical Foundations

A.1 Relations

A.2 Ordered Sets

A.2.1 Pre-Orders, Partial Orders, and More

A.2.2 Hasse Diagrams

A.2.3 Bounds and Extremal Elements

A.2.4 Noetherian and Artinian Orders

A.2.5 Chains

A.2.6 Directed Sets

A.2.7 Maps on Partial Orders

A.2.8 Order Homomorphisms, Order Isomorphisms

A.3 Complete Partially Ordered Sets

A.3.1 Chain and Directly Complete Partial Orders

A.3.2 Maps on Complete Partial Orders

A.3.3 Mechanisms for Constructing Complete Partial Orders

Table of Contents (19)

- ▶ A Mathematical Foundations (cont'd)
 - A.4 Lattices
 - A.4.1 Lattices, Complete Lattices
 - A.4.2 Distributive, Additive Maps on Lattices
 - A.4.3 Lattice Homomorphisms, Lattice Isomorphisms
 - A.4.4 Modular, Distributive, and Boolean Lattices
 - A.4.5 Mechanisms for Constructing Lattices
 - A.4.6 Order-theoretic and Algebraic View of Lattices
 - A.5 Fixed Point Theorems
 - A.5.1 Fixed Points, Towers
 - A.5.2 Fixed Point Theorems for Complete Partial Orders
 - A.5.3 Fixed Point Theorems for Lattices
 - A.6 Fixed Point Induction

Table of Contents (20)

- ▶ A Mathematical Foundations (cont'd)
 - A.7 Completions, Embeddings
 - A.7.1 Downsets
 - A.7.2 Ideal Completion: Embedding of Lattices
 - A.7.3 Cut Completion: Embedding of Partial Orders&Lattices
 - A.7.4 Downset Completion: Embedding of Partial Orders
 - A.7.5 Application: Lists and Streams
 - A.8 References, Further Reading

Part I

Motivation

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

23/1060

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>

Objective of John Hughes' Position Statement

...starting from the obvious fact that

- software is becoming more and more complex

and the conclusion that

- the ability of structuring software well (**modularization!**) becomes thus paramount since well-structured software is more easily to read, write, debug, and be re-used

...to provide evidence for the **claim**:

- Conventional languages place **conceptual limits on the way problems can be modularized.**
- **Functional languages** push back these limits.

Fundamental

- ▶ **Higher-order functions** (\rightsquigarrow composing **functions**)
- ▶ **lazy evaluation** (\rightsquigarrow composing **programs**)

Chapter 1.1

Reconsidering Folk Knowledge

Contents

Part I

Chap. 1

1.1

1.2

1.3

1.4

1.5

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

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)!**

Note

...this set of characteristics and advantages of functional programming is essentially a **negative 'is-not'** characterization:

- “It says a lot about what functional programming is **not** (it has no assignments, no side effects, no explicit specification of flow of control) but not much about what it is.”

It is thus **inappropriate** to explain any superiority of the functional programming style over (more) conventional ones.

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.

Note

...even if there is overwhelming empirical evidence underpinning the productivity claim, the set of characteristics of functional programming and the advantages they imply according to folk knowledge does not contribute to answering: Why?

- Can the productivity claim be concluded from the set of characteristics and advantages referred to by 'folk knowledge,' i.e., does dropping features (like assignments, control-flow specification, etc.) explain the productivity gain?

Hardly!

- In the words of [John Hughes](#), dropping features reminds more to a medieval monk denying himself the pleasures of life in the hope of getting virtuous.

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**

Structured vs. Non-structured Programming

...a striking analogue.

Structured programs are

- free of goto-statements ('goto considered harmful'*)
 - blocks in structured programs are free of multiple entries and exits
- ⇒ Hence, structured programs are easier to cope with mathematically than unstructured programs!
- ⇒ Structured programming is more efficient/productive!

Note, this is essentially a negative 'is-not' characterization, too!

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

Conceptually more Important

...in contrast to non-structured programs, **structured programs** are designed

- ▶ **modularly!**

This is the reason, why **structured programming** is more **efficient** and **productive** than unstructured programming:

- Small modules are easier and faster to read, write, and maintain.
- Re-use becomes easier.
- Modules can be tested independently.

Note: Dropping goto-statements is not an essential source of productivity gain:

- **Absence of gotos** supports 'programming in the small.'
- **Modularity** supports 'programming in the large.'

Next

...we follow [John Hughes](#) reconsidering **higher-order functions** and **lazy evaluation** from the perspective of their 'glueing' capabilities enabling to construct functions and programs **modularly**:

- ▶ **Higher-order functions** for **glueing functions** (cf. [Chap. 1.2](#))
- ▶ **Lazy evaluation** for **glueing programs** (cf. [Chap. 1.3](#))

Chapter 1.2

Glueing Functions Together: Higher-Order Functions

Contents

Part I

Chap. 1

1.1

1.2

1.3

1.4

1.5

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

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
```

Exercise 1.2.1

Using the functions introduced so far, we can define:

`Adding the elements of a matrix`

```
summatrix = sum . map sum
```

1. Think about how `summatrix` works.
2. Stepwise evaluate `summatrix` for some arguments.

Summing up

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

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

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

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

Generalizing the Approach

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

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

A value of type **(treeof X)**:

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

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

The Higher-order Function `redtree`

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

```
redtree f g a (node label subtrees)
= f label (redtree' f g a subtrees)
```

where

```
redtree' f g a (cons subtree rest)
= g (redtree f g a subtree) (redtree' f g a rest)
redtree' f g a nil = a
```

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

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

in tree values.

Applications of redtree (1)

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


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

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

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

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

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



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

Applications of redtree (2)

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

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

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

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

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

```
->> 10
```

Contents

Part I

Chap. 1

1.1

1.2

1.3

1.4

1.5

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

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))))
```

Exercise 1.2.2

Complete the [stepwise evaluation](#) of the term:

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

in [Example 9](#).

Exercise 1.2.3

1. Repeat [Examples 1 to 9](#) in [Haskell](#).
2. Experiment with the resulting implementations.

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

Contents

Part I

Chap. 1

1.1

1.2

1.3

1.3.1

1.3.2

1.3.3

1.4

1.5

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Preparing the Setting

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

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

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

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

Possible problems:

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

Lazy Evaluation

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

- generator
- selector

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

- **lazy evaluation**

ensuring:

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

In the following

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

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

Chapter 1.3.1

Square Root Computation

Contents

Part I

Chap. 1

1.1

1.2

1.3

1.3.1

1.3.2

1.3.3

1.4

1.5

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

The Newton-Raphson Approach

...for square root computation.

Given: N , a positive number

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

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

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

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

Contents

Part I

Chap. 1

1.1

1.2

1.3

1.3.1

1.3.2

1.3.3

1.4

1.5

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

68/1969

A Typical Imperative Implementation

...realizing this approach (here in Fortran):

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

⇒ this is essentially a **monolithic**, not decomposable program.

Developing now a Modular Functional Version

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

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

Second, we define function `g`:

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

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

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

which is equivalent to:

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

Writing a Generator

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

Generator A:

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

Note:

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

```
repeat g a0
```

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

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

- Evaluating `repeat g a0` does not terminate!

Writing a Selector

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

Selector A:

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


Glueing together Generator and Selector

...to obtain the final program.

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

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

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

Looking ahead: As we are going to show soon, **Generator A** and **Selector A** can easily be combined with other generators and selectors, respectively, giving them indeed the flavour of **modules**.

Summing up

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

Modules:

- **Generator program/module**: **repeat**
[$a_0, g\ a_0, g(g\ a_0), g(g(g\ a_0)), \dots$]
...potentially infinite, no pre-defined limit of length.
- **Selector program/module**: **within**
 $g^i\ a_0$ with $\text{abs}(g^i\ a_0 - g^{i+1}\ a_0) \leq \text{eps}$
...**lazy evaluation** ensures that the selector function is applied eventually \Rightarrow **termination!**

Synchronized by:

- ▶ **Lazy evaluation**

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

Immediate Benefit: Modules are Re-usable

...we will demonstrate that

- Generator A
- Selector A

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

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

Re-using Generator A with a new Selector

Consider a new criterion for termination:

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

Selector B:

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

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

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

Dually: Re-using Selectors A and B

...with new **generators**.

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

- Numerical integration
- Numerical differentiation

Chapter 1.3.2

Numerical Integration

Contents

Part I

Chap. 1

1.1

1.2

1.3

1.3.1

1.3.2

1.3.3

1.4

1.5

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Numerical Integration

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

Sought: The area under f between a and b

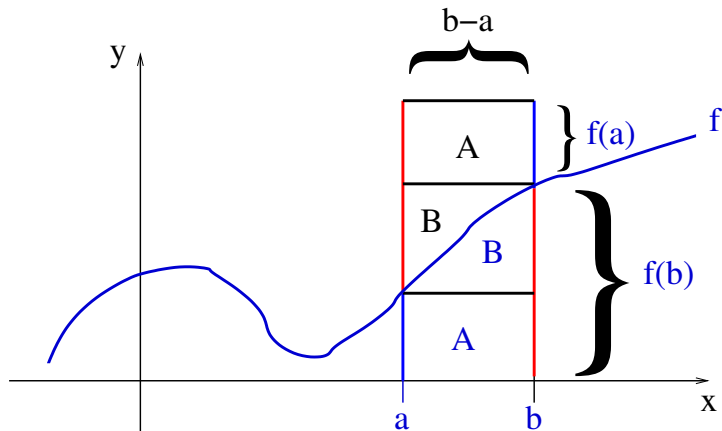
Simple Solution:

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

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

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

Illustrating the Essence of easyintegrate



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

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:

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

Summing up

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

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

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

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

- ▶ **lazy evaluation!**

A Note on Performance

The generator `integrate` as defined before is

- sound but inefficient (many re-computations of `f a`, `f b`, and `f mid`, which are redundant and hence superfluous).

Using `locally defined values` as shown below removes this deficiency:

```
integrate f a b = integ f a b (f a) (f b)
integ f a b fa fb
  = cons ((fa+fb)*(b-a)/2)
        (map addpair (zip (integ f a m fa fm)
                          (integ f m b fm fb)))
      where m = (a+b)/2
            fm = f m
```

Chapter 1.3.3

Numerical Differentiation

Contents

Part I

Chap. 1

1.1

1.2

1.3

1.3.1

1.3.2

1.3.3

1.4

1.5

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Numerical Differentiation

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

Sought: The slope of f at point x

Simple Solution:

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

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

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

Writing a Generator/Selector Combination

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

Generator C:

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

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

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

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

Exercise 1.3.3.1

1. **Glue** together **Generator C** and **Selector B**.

2. Repeat the examples for

2.1 square root computation

2.2 numerical integration

2.3 numerical differentiation

in **Haskell**, and experiment with the resulting implementations.

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

Contents

Part I

Chap. 1

1.1

1.2

1.3

1.4

1.5

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

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

Contents

Part I

Chap. 1

1.1

1.2

1.3

1.4

1.5

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11



Chapter 1: Further Reading (1)

-  Stephen Chang, Matthias Felleisen. *The Call-by-Need Lambda Calculus, Revisited*. In Proceedings of the 21st European Symposium on Programming (ESOP 2012), Springer-V., LNCS 7211, 128-147, 2012.
-  Edsger W. Dijkstra. *Go To Statement Considered Harmful*. Letter to the Editor. Communications of the ACM 11(3):147-148, 1968.
-  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>

Chapter 1: Further Reading (2)

-  Paul Hudak. *Conception, Evolution and Applications of Functional Programming Languages*. Communications of the ACM 21(3):359-411, 1989.
-  John Hughes. *Why Functional Programming Matters*. Computer Journal 32(2):98-107, 1989.
-  Mark P. Jones. *Functional Thinking*. Lecture at the 6th International Summer School on Advanced Functional Programming, Boxmeer, The Netherlands, 2008.
-  Greg Michaelson. *Programming Paradigms, Turing Completeness and Computational Thinking*. The Art, Science, and Engineering of Programming 4(3), Article 4, 21 pages, 2020.

Chapter 1: Further Reading (3)

-  Yaron Minsky. *OCaml for the Masses*. Communications of the ACM 54(11):53-58, 2011.
-  Simon Peyton Jones. *16 Years of Haskell: A Retrospective on the Occasion of its 15th Anniversary – Wearing the Hair Shirt: A Retrospective on Haskell*. Invited Keynote at the 30th ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages (POPL 2003), 2003. <http://research.microsoft.com/users/simonpj/papers/haskell-retrospective/>
-  Chris Sadler, Susan Eisenbach. *Why Functional Programming?* In *Functional Programming: Languages, Tools and Architectures*. Susan Eisenbach (Ed.), Ellis Horwood, 7-8, 1987.
-  Neil Savage. *Using Functions for Easier Programming*. Communications of the ACM 61(5):29-30, 2018.

Part II

Programming Principles

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

100/196

Chapter 2

Programming with Streams

Contents

Part I

Chap. 1

Part II

Chap. 2

2.1

2.2

2.3

2.4

2.5

2.6

2.7

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

101/196

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

Contents

Part I

Chap. 1

Part II

Chap. 2

2.1

2.2

2.3

2.4

2.5

2.6

2.7

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

105/196

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!"

Contents

Part I

Chap. 1

Part II

Chap. 2

2.1

2.2

2.3

2.4

2.5

2.6

2.7

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

107/196

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..]]
```

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

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

```
iterate :: (a -> a) -> a -> [a]
```

```
iterate f x = x : iterate f (f x)
```

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

Stream Generators

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

```
powers n = iterate (*n) 1
```

```
ones     = iterate id 1
```

```
twos     = iterate id 2
```

```
threes   = iterate id 3
```

```
nats     = iterate (+1) 0
```

```
theNats  = iterate (+1) 0
```

```
evens    = iterate (+2) 0
```

```
odds     = iterate (+2) 1
```

where

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

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

Generator *Filter* *Generator*

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

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

The Stream of Primes (3)

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

```
primes
```

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

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

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

Contents

Part I

Chap. 1

Part II

Chap. 2

2.1

2.2

2.3

2.4

2.5

2.6

2.7

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

116/196

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

Contents

Part I

Chap. 1

Part II

Chap. 2

2.1

2.2

2.2.1

2.2.2

2.2.3

2.2.4

2.3

2.4

2.5

2.6

2.7

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

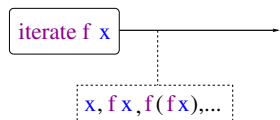
Part IV

Chap. 7

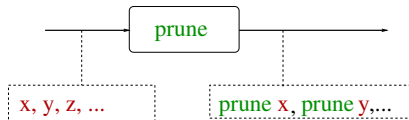
The Generate-Prune Pattern

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

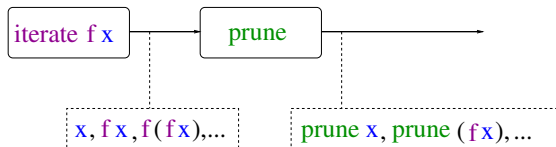
Generator module:



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](#).

Contents

Part I

Chap. 1

Part II

Chap. 2

2.1

2.2

2.2.1

2.2.2

2.2.3

2.2.4

2.3

2.4

2.5

2.6

2.7

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chapter 2.2.1

The Generate-Select/Filter Pattern

Contents

Part I

Chap. 1

Part II

Chap. 2

2.1

2.2

2.2.1

2.2.2

2.2.3

2.2.4

2.3

2.4

2.5

2.6

2.7

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

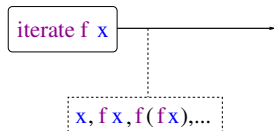
Part IV

Chap. 7

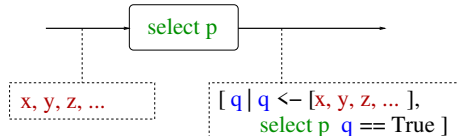
The Generate-Select/Filter Pattern

...at a glance:

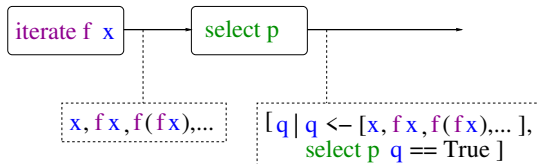
Generator module:



Selector/Filter module:



Linking Generator and Selector/Filter modules together:



Contents

Part I

Chap. 1

Part II

Chap. 2

2.1

2.2

2.2.1

2.2.2

2.2.3

2.2.4

2.3

2.4

2.5

2.6

2.7

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

122/196

Examples: Generate-Select Pattern

...applications of the **Generate-Select** Pattern:

- ▶ The head element of a stream:

$\underbrace{\text{head}}_{\text{Selector}} \quad \underbrace{\text{nats}}_{\text{Generator}} \quad \rightarrow \text{head } (0 : \text{map } (+1) \text{ nats})$
 $\rightarrow 0$

- ▶ Three pseudo random numbers:

$\underbrace{\text{take } 3}_{\text{Selector}} \quad \underbrace{(\text{randomSequence } 17489)}_{\text{Generator}} \rightarrow [17489, 69134, 9327]$

- ▶ The 6th to the 10th prime number:

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

Examples: Generate-Filter Pattern

...applications of the Generate-Filter Pattern:

- ▶ The tail of a stream:

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

- ▶ The prime numbers, which are a palindrome:

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

- ▶ Even pseudo random numbers:

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

Is it a Selector or a Filter?

Taking a pragmatic point of view, if applied to a [stream](#), termination is

- [ensured](#), then it is a [selector](#):

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

- [not ensured](#), then it is a [filter](#):

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

Exercise 2.2.1.1: Corner Cases

Should `tail` and `take_until` defined by:

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

take_until :: (Int -> Bool) -> [Int] -> [Int]
take_until p [] = []
take_until p (n:ns)
  | p n = [n]
  | True = n : take_until p ns
```

be better considered/called `filters` or `selectors`? Does it depend on the context? Consider:

- `tail nats ->> [1,2,3,...`
- `take_until is_even primes ->> [2]`
- `take_until is_odd primes ->> [2,3]`
- `take_until is_even (tail primes) ->> [3,5,7,...`
- `take_until is_odd (tail primes) ->> [3]`

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.

Recall: If **ignore** is defined by:

ignore :: a -> b -> b

ignore a b = b

the **leftmost-outermost operation** of the term(s)

ignore **twos** 42 $\hat{=}$ **twos** 'ignore' 42

is given by **ignore** (rather than by **twos**).

Chapter 2.2.2

The Generate-Transform Pattern

Contents

Part I

Chap. 1

Part II

Chap. 2

2.1

2.2

2.2.1

2.2.2

2.2.3

2.2.4

2.3

2.4

2.5

2.6

2.7

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

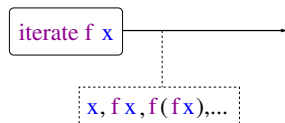
Part IV

Chap. 7

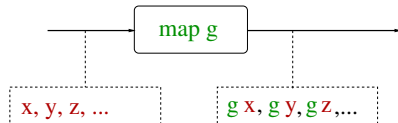
The Generate-Transform Pattern

...at a glance:

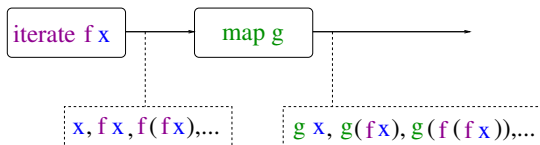
Generator module:



Transformer module:



Linking Generator and Transformer modules together:



Examples: Generate-Transform Pattern (1)

- ▶ The stream of predecessors of prime numbers:

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

Transformer *Generator*

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

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

Transformer *Generator*

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

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

Transformer *Generator*

Examples: Generate-Transform Pattern (2)

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

$$p \leq r \leq q$$

are required.

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

scale 42.0 51.0 *randomSequence*
Transformer *Generator*

```
scale :: Float -> Float -> [Int] -> [Float]
```

```
scale p q randSeq = map (f p q) randSeq
```

```
where f :: Float -> Float -> Int -> Float
```

```
      f p q n = p + ((n * (q-p)) / (modulus-1))
```

Chapter 2.2.3

Pattern Combinations

Contents

Part I

Chap. 1

Part II

Chap. 2

2.1

2.2

2.2.1

2.2.2

2.2.3

2.2.4

2.3

2.4

2.5

2.6

2.7

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

133/196

Examples: Pattern Combinations

...applications of pattern combinations:

- ▶ The stream of prime numbers:

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

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

- ▶ The 6th to the 10th prime number:

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

Selector *Filter* *Generator*

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

```
addFirstTwo   twos
```

Selector + Transformer *Generator*

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

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

```
->> 2+2
```

```
->> 4
```

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

Chapter 2.2.4

Summary

Contents

Part I

Chap. 1

Part II

Chap. 2

2.1

2.2

2.2.1

2.2.2

2.2.3

2.2.4

2.3

2.4

2.5

2.6

2.7

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

135/196

Principles of Modularization

...enabled by [stream programming](#) and [lazy evaluation](#):

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

Chapter 2.3

Boosting Performance

Contents

Part I

Chap. 1

Part II

Chap. 2

2.1

2.2

2.3

2.3.1

2.3.2

2.3.3

2.3.4

2.4

2.5

2.6

2.7

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

137/196

Chapter 2.3.1

Motivation

Contents

Part I

Chap. 1

Part II

Chap. 2

2.1

2.2

2.3

2.3.1

2.3.2

2.3.3

2.3.4

2.4

2.5

2.6

2.7

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

138/196

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**:

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

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

Illustrating the Source of Inefficiency (1)

...calling `fib` again and again for the same arguments:

```
fib 0 ->> 0 -- 1 call of fib
fib 1 ->> 1 -- 1 call of fib
fib 2 ->> fib 1 + fib 0
      ->> 1 + 0
      ->> 1 -- 3 calls of fib
fib 3 ->> fib 2 + fib 1
      ->> (fib 1 + fib 0) + 1
      ->> (1 + 0) + 1
      ->> 2 -- 5 calls of fib
fib 4 ->> fib 3 + fib 2
      ->> (fib 2 + fib 1) + (fib 1 + fib 0)
      ->> ((fib 1 + fib 0) + 1) + (1 + 0)
      ->> ((1 + 0) + 1) + (1 + 0)
      ->> 3 -- 9 calls of fib
```

Contents

Part I

Chap. 1

Part II

Chap. 2

2.1

2.2

2.3

2.3.1

2.3.2

2.3.3

2.3.4

2.4

2.5

2.6

2.7

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

140/196

Illustrating the Source of Inefficiency (2)

```
fib 5 ->> fib 4 + fib 3
->> (fib 3 + fib 2) + (fib 2 + fib 1)
->> ((fib 2 + fib 1) + (fib 1 + fib 0))
      + ((fib 1 + fib 0) + 1)
->> (((fib 1 + fib 0) + 1)
      + (1 + 0)) + ((1 + 0) + 1)
->> (((1 + 0) + 1) + (1 + 0)) + ((1 + 0) + 1)
->> 5 -- 15 calls of fib

fib 8 ->> fib 7 + fib 6
->> (fib 6 + fib 5) + (fib 5 + fib 4)
->> ((fib 5 + fib 4) + (fib 4 + fib 3))
      + ((fib 4 + fib 3) + (fib 3 + fib 2))
->> (((fib 4 + fib 3) + (fib 3 + fib 2))
      + (fib 3 + fib 2) + (fib 2 + fib 1)))
      + (((fib 3 + fib 2) + (fib 2 + fib 1))
      + ((fib 2 + fib 1) + (fib 1 + fib 0)))
->> ...
->> 21 -- 60 calls of fib
```

Contents

Part I

Chap. 1

Part II

Chap. 2

2.1

2.2

2.3

2.3.1

2.3.2

2.3.3

2.3.4

2.4

2.5

2.6

2.7

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

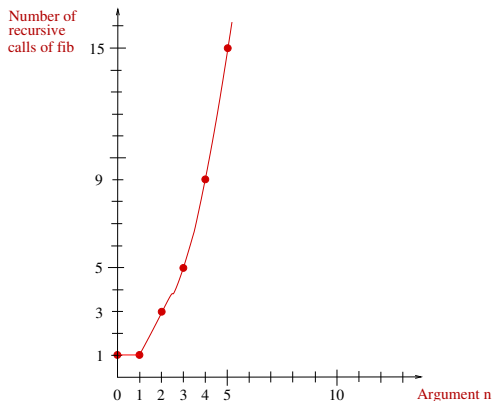
Chap. 7

141/196

Finding

..the naive tree-like recursive computation of the Fibonacci numbers causes very many applications of `fib` (and thus its computation) to the very same arguments.

In effect, the computational effort grows **exponentially!**



Reminder: Asymptotic Upper Bounds

Let $f, g, h : \mathbb{N}_0 \rightarrow \mathbb{R}_0^+$ be functions from the set of natural numbers to the set of positive real numbers (both incl. 0).

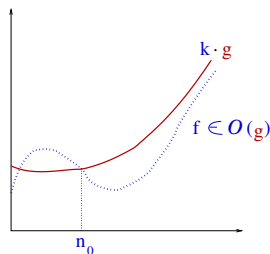
Definition 2.3.1.1 (Upper Bound)

g is called an **asymptotic upper bound** of f iff

$$\exists k \in \mathbb{R}^+. \exists n_0 \in \mathbb{N}_0. \forall n \in \mathbb{N}_0. n \geq n_0 \Rightarrow f(n) \leq k * g(n)$$

In this case, we write: $f \in \mathcal{O}(g)$ (or: $f = \mathcal{O}(g)$) with

$$\mathcal{O}(g) =_{df} \{h \mid g \text{ is an asymptotic upper bound of } h\}$$



The Impact of Important Cost Functions

Class	Costs	Intuition: <i>input a thousandfold as large means</i>
$\mathcal{O}(c)$	constant	...equal effort
$\mathcal{O}(\log n)$	logarithmic	...only tenfold effort
$\mathcal{O}(n)$	linear	...also a thousandfold effort
$\mathcal{O}(n \log n)$	quasi-linear	...tenthousandfold effort
$\mathcal{O}(n^2)$	quadratic	...millionfold effort
$\mathcal{O}(n^3)$	cubic	...billionfold effort
$\mathcal{O}(n^c)$	polynomial	...gigantic big effort (for big c)
$\mathcal{O}(2^n)$	exponential	...hopeless

n	Linear	Quadratic	Cubic	Exponential
1	1 μ s	1 μ s	1 μ s	2 μ s
10	10 μ s	100 μ s	1 ms	1 ms
20	20 μ s	400 μ s	8 ms	1 s
30	30 μ s	900 μ s	27 ms	18 min
40	40 μ s	2 ms	64 ms	13 days
50	50 μ s	3 ms	125 ms	36 years
60	60 μ s	4 ms	216 ms	36 560 years
100	100 μ s	10 ms	1 sec	$4 * 10^{16}$ years
1000	1 ms	1 sec	17 min	very, very long...

Contents

Part I

Chap. 1

Part II

Chap. 2

2.1

2.2

2.3

2.3.1

2.3.2

2.3.3

2.3.4

2.4

2.5

2.6

2.7

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

144/196

Fortunately

...stream programming can (often) help

- conquering complexity
- gaining/improving performance!

Contents

Part I

Chap. 1

Part II

Chap. 2

2.1

2.2

2.3

2.3.1

2.3.2

2.3.3

2.3.4

2.4

2.5

2.6

2.7

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chapter 2.3.2

Stream Programming combined with Münchhausen Principle

Contents

Part I

Chap. 1

Part II

Chap. 2

2.1

2.2

2.3

2.3.1

2.3.2

2.3.3

2.3.4

2.4

2.5

2.6

2.7

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Computing the Fibonacci Numbers Stream Eff.

```
0  1  1  2  3  5  8  13.. The stream of Fibonacci numbers
1  1  2  3  5  8  13 21.. The tail of the stream of Fib. numb
+  +  +  +  +  +  +  +.. ++++++ add columnwise ++++++
1  2  3  5  8  13 21 34.. The tail of the tail of the
                        stream of Fibonacci numbers
```

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

```
fibs :: [Int] -- Generator
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
      'Tuft'   'Swamp'
```

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

The stream of Fibonacci numbers

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

The Münchhausen Principle in Detail

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

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

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

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

'Tuft' Tail of tail of stream of fibs

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

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

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

Application: Generate/Select Principle

Generator:

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

Generate-Select applications:

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

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

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

where

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

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

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

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

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

Contents

Part I

Chap. 1

Part II

Chap. 2

2.1

2.2

2.3

2.3.1

2.3.2

2.3.3

2.3.4

2.4

2.5

2.6

2.7

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

149/196

Computing Fibonacci Numbers Efficiently

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

```
fib :: Int -> Int
fib n =   head   (drop (n-1)  fibs)
         Selector 2 Selector 1 Generator
      [ =   last   (take n    fibs) ]
         Selector 2 Selector 1 Generator
```

And even shorter with only one selector:

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

Contents

Part I

Chap. 1

Part II

Chap. 2

2.1

2.2

2.3

2.3.1

2.3.2

2.3.3

2.3.4

2.4

2.5

2.6

2.7

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

152/196

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 prefixes 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

Contents

Part I

Chap. 1

Part II

Chap. 2

2.1

2.2

2.3

2.3.1

2.3.2

2.3.3

2.3.4

2.4

2.5

2.6

2.7

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Memoization

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

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

Essence

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

Means

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

Memo Functions, Memo Tables

A **memo function** is

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

A **memo table** allows

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

Requirement: A memo function `memo`

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

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

`memo f x = f x`

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

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

Illustrating the Essence of Memo Functions

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

Assume $f : ID \rightarrow ID'$ is a (costly to compute) function with 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 =  $\underbrace{\text{fib\_memolist}!!(n-1)}_{\text{Generator}}$  +  $\underbrace{\text{fib\_memolist}!!(n-2)}_{\text{Selector}}$ 
```

Compare this w/ the straightforward implementation of fib:

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

Lemma 2.3.3.1

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

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

Example 2: Computing Powers

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

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

Compare this w/ the straightforward implement. of power:

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

Lemma 2.3.3.2

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

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

Summing up

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

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

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

Memo functions are

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

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

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

Chapter 2.3.4

Summary

Contents

Part I

Chap. 1

Part II

Chap. 2

2.1

2.2

2.3

2.3.1

2.3.2

2.3.3

2.3.4

2.4

2.5

2.6

2.7

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

163/196

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**.

Contents

Part I

Chap. 1

Part II

Chap. 2

2.1

2.2

2.3

2.3.1

2.3.2

2.3.3

2.3.4

2.4

2.5

2.6

2.7

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

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 =  $\underbrace{\text{fibs}}_{\text{Generator}} \underbrace{!! n}_{\text{Selector}}$ 
```

- ▶ Computing powers:

```
powers :: [Int]
```

```
powers = 1 : 2 : zipWith (+) (tail powers) (tail powers)
```

```
power :: Int -> Int
```

```
power n =  $\underbrace{\text{powers}}_{\text{Generator}} \underbrace{!! n}_{\text{Selector}}$ 
```

- ▶ ...

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)
```

▶ ...

Contents

Part I

Chap. 1

Part II

Chap. 2

2.1

2.2

2.3

2.3.1

2.3.2

2.3.3

2.3.4

2.4

2.5

2.6

2.7

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

166/196

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).

Exercise 2.3.4.1

Compare the **run-time performance** of the straightforward implementations of **fib** and **power** with their **'boosted'** versions using **stream programming** combined with

1. Münchhausen principle
2. memoization

Exercise 2.3.4.2

Evaluate step by step the term `fib 3` for both implementations of `fib`:

- ```
1. fibs :: [Int]
 fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
 fib :: Int -> Int
 fib n = fibs!!n
```
- ```
2. fib_ml :: [Int]
   fib_ml = [ fib n | n <- [0..] ]
   fib :: Int -> Int
   fib 0 = 0
   fib 1 = 1
   fib n = fib_ml!!(n-1) + fib_ml!!(n-2)
```

In closing

Stream programming combined w/ the Münchhausen principle and memoization are important though

- no silver bullets

for improving performance by avoiding recomputations and recursion.

If, however, they hit they can significantly

- **boost performance**: from taking too long to be feasible to be completed in an instant!

Natural candidates are problems that

- naturally wind up repeatedly computing the solution to identical subproblems, e.g., **tree-recursive processes**.

Sometimes

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

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

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

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

Theorem 2.3.4.3

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

Chapter 2.4

Stream Diagrams

Contents

Part I

Chap. 1

Part II

Chap. 2

2.1

2.2

2.3

2.4

2.5

2.6

2.7

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

172/196

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

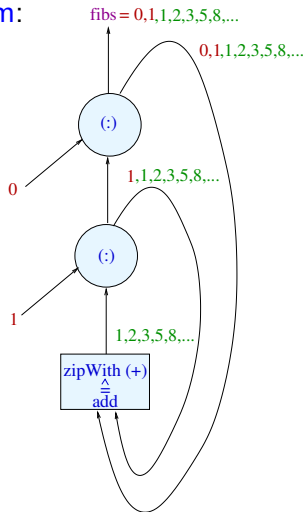
as examples.

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

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

Illustration by Stepwise Evaluation

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

Contents

Part I

Chap. 1

Part II

Chap. 2

2.1

2.2

2.3

2.4

2.5

2.6

2.7

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

177/196

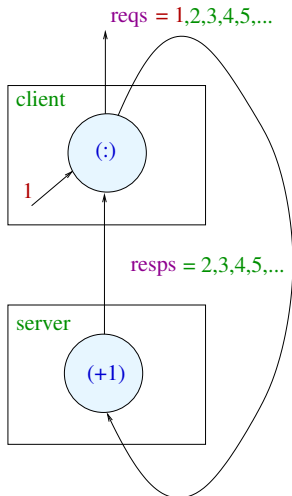
The Stream Diagram

...representing the stream of client/server interactions

reqs = client resps

resps = server reqs

as a stream diagram:



Chapter 2.5

Pitfalls, Remedies

Contents

Part I

Chap. 1

Part II

Chap. 2

2.1

2.2

2.3

2.4

2.5

2.5.1

2.5.2

2.5.3

2.6

2.7

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chapter 2.5.1

Termination, Domain-specific Knowledge

Contents

Part I

Chap. 1

Part II

Chap. 2

2.1

2.2

2.3

2.4

2.5

2.5.1

2.5.2

2.5.3

2.6

2.7

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Note

...lazy evaluation is

- necessary

to ensure termination of generate-select programs but

- not sufficient!

Contents

Part I

Chap. 1

Part II

Chap. 2

2.1

2.2

2.3

2.4

2.5

2.5.1

2.5.2

2.5.3

2.6

2.7

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

For Illustration

...consider the below naive prime number test:

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

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

Then:

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

Exercise 2.5.1.1

1. Why does the `generate-transform/select` implementation of `member` and `primes` fail in case b)?
2. How can the implementation of the `transformer-selector` function `member` be modified to work properly as expected?
3. What `domain-specific knowledge` is exploited for this modification?

Chapter 2.5.2

Lifting, Undecidability

Contents

Part I

Chap. 1

Part II

Chap. 2

2.1

2.2

2.3

2.4

2.5

2.5.1

2.5.2

2.5.3

2.6

2.7

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Functional Lifting

...compare the definition of the stream `fibs` (cp. Chapter 2.3.2):

```
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

Contents

Part I

Chap. 1

Part II

Chap. 2

2.1

2.2

2.3

2.4

2.5

2.5.1

2.5.2

2.5.3

2.6

2.7

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

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 modification 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
```

Exercise 2.5.3.1

1. Implement the various versions of the [client/server application](#) of [Chapter 2.4](#) and [Chapter 2.5.3](#).
2. Experimentally validate their claimed behaviours.

Chapter 2.6

Summary, Looking ahead

Contents

Part I

Chap. 1

Part II

Chap. 2

2.1

2.2

2.3

2.4

2.5

2.6

2.7

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

193/196

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

Contents

Part I

Chap. 1

Part II

Chap. 2

2.1

2.2

2.3

2.4

2.5

2.6

2.7

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV




Chap. 7

Chap. 8




Chap. 9

196/196




Chapter 2: Further Reading (1)

-  Umut A. Acar, Guy E. Blelloch, Robert Harper. *Selective Memoization*. In Conference Record of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2003), 14-25, 2003.
-  Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice-Hall, 2nd edition, 1998. (Chapter 9, Infinite Lists)
-  Richard Bird, Philip Wadler. *An Introduction to Functional Programming*. Prentice Hall, 1988. (Chapter 7, Infinite Lists)




Chapter 2: Further Reading (2)

-  Byron Cook, John Launchbury. *Disposable Memo Functions*. Extended Abstract. In Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming (ICFP'97), 310, 1997 (full paper in Proceedings Haskell'97 workshop).
-  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)
-  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)




Chapter 2: Further Reading (3)

-  Kento Emoto, Sebastian Fischer, Zhenjiang Hu. *Generate, Test, and Aggregate: A Calculation-based Framework for Systematic Parallel Programming with MapReduce*. In Proceedings of the 21st European Symposium on Programming (ESOP 2012), Springer-V., LNCS 7211, 254-273, 2012.
-  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)
-  Daniel P. Friedman, David S. Wise. *CONS should not Evaluate its Arguments*. In Proceedings of the 3rd International Conference on Automata, Languages and Programming, 257-284, 1976.





Chapter 2: Further Reading (4)

-  Max Hailperin, Barbara Kaiser, Karl Knight. *Concrete Abstractions – An Introduction to Computer Science using Scheme*. Brooks/Cole Publishing Company, 1999. (Chapter 12.3, Memoization; Chapter 12.5, Comparing Memoization and Dynamic Programming)
-  Peter Henderson, James H. Morris. *A Lazy Evaluator*. In Conference Record of the 3rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'76), 95-103, 1976.
-  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)




Chapter 2: Further Reading (5)

-  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.
-  Jon Kleinberg, Éva Tardos. *Algorithm Design*. Addison-Wesley/Pearson, 2006. (Chapter 6.2, Principles of Dynamic Programming: Memoization or Iteration over Sub-problems)
-  Peter J. Landin. *A Correspondence between ALGOL60 and Church's Lambda-Notation: Part I*. Communications of the ACM 8(2):89-101, 1965.

Chapter 2: Further Reading (6)

-  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)
-  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)
-  Fethi Rabhi, Guy Lapalme. *Algorithms – A Functional Programming Approach*. Addison-Wesley, 1999. (Chapter 10.1, Process networks)

Chapter 2: Further Reading (7)

-  Jay M. Spitzen, Karl M. Levitt, Lawrence Robinson. *An Example of Hierarchical Design and Proof*. Communications of the ACM 21(12):1064-1075, 1978.
-  Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley/Pearson, 2nd edition, 1999. (Chapter 17, Lazy programming; Chapter 17.6, Infinite lists; Chapter 17.7, Why infinite lists? Chapter 19.6, Avoiding recomputation: 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 3

Programming with Higher-Order Functions: Algorithm Patterns

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Motivation

Programming with higher-order functions

- ▶ Many powerful and general **algorithmic principles** can be encapsulated in a suitable **higher-order function (HoF)**.
- ▶ This allows to **design** a **collection** or a **class of algorithms** (instead of designing an algorithm for only a particular application).

Conceptually

- ▶ this emphasizes the essence of the **underlying algorithmic principle**.

Pragmatically

- ▶ this makes these algorithmic principles **easily re-usable**.

Outline

In this chapter, we demonstrate this reconsidering an array of well-known **top-down** and **bottom-up design principles** of algorithms.

- ▶ **Top-down**: Starting from the initial problem, the algorithm works down to the solution by considering sub-problems or alternatives.
 - **Divide-and-conquer** (cf. LVA 185.A03 FP, Chap. 18.1)
 - **Backtracking search**
 - **Priority-first search**
 - **Greedy search**
- ▶ **Bottom-up**: Starting from small problem instances, the algorithm works up to the solution of the initial problem by combining solutions of smaller problem instances to solutions of larger ones.
 - **Dynamic programming**

Chapter 3.1

Divide-and-Conquer

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Divide and Conquer

Given: A problem instance P .

Sought: A solution S of P .

Algorithmic Idea:

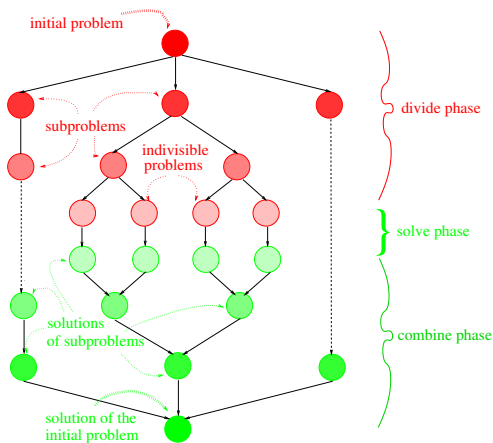
- If a problem instance is **simple/small** enough, solve it: directly or by means of some basic algorithm.
- Otherwise, **divide** the problem instance into smaller subproblem instances by applying the **division** strategy **recursively** until all subproblem instances are simple enough to be solved directly.
- **Combine** the solutions of the subproblem instances to the solution of the initial problem instance.

Applicability Requirement:

- No generation of identical subproblem instances during problem division (otherwise, a performance issue!)

Illustrating the Divide-and-Conquer Principle

...successive stages of a divide-and-conquer algorithm:



Fethi Rabhi, Guy Lapalme.

Algorithms: A Functional Programming Approach.

Addison-Wesley, 1999, page 156.

Implementing Divide-and-Conquer as HoF (1)

Setting:

A `problem` with

- problem instances of `kind p`
- solution instances of `kind s`

Objective:

A higher-order function (HoF) `divide_and_conquer` solving

- suitably parameterized `problem` instances of `kind p` using the ‘`divide and conquer`’ principle.

Implementing Divide-and-Conquer as HoF (2)

The `arguments` of `divide_and_conquer`:

- `indiv :: p -> Bool`: ...yields `True`, if the problem instance can/need not be divided further (e.g., it can *easily* be solved by some *basic* algorithm).
- `solve :: p -> s`: ...yields the solution of a problem instance that can/need not be divided further.
- `divide :: p -> [p]`: ...divides a problem instance into a list of subproblem instances.
- `combine :: p -> [s] -> s`: Given a problem instance and the list of solutions of the subproblem instances derived from it, `combine` yields the solution of the problem instance.

Implementing Divide-and-Conquer as HoF (3)

The HoF Implementation:

$$\begin{array}{c} \text{divide_and_conquer} :: (\underbrace{p \rightarrow \text{Bool}}_{\text{Simple enough?}}) \rightarrow (\underbrace{p \rightarrow s}_{\text{Solve!}}) \rightarrow \\ (\underbrace{p \rightarrow [p]}_{\text{Divide}}) \rightarrow (\underbrace{p \rightarrow [s] \rightarrow s}_{\text{Combine}}) \rightarrow \\ \underbrace{p}_{\text{Problem instance}} \rightarrow \underbrace{s}_{\text{Solution}} \end{array}$$

```
divide_and_conquer indiv solve divide combine pi
= dac pi
  where
    dac pi'
      | indiv pi' = solve pi'
      | otherwise = combine pi' (map dac (divide pi'))
```

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

212/196

Typical Applications of Divide-and-Conquer

Application fields such as

- Numerical analysis
- Cryptography
- Image processing
- Sorting
- ...

Especially

- Quicksort
- Mergesort
- Binomial coefficients
- ...

Example: Quicksort

```
quickSort :: Ord a => [a] -> [a]
quickSort ls
= divide_and_conquer indiv solve divide combine ls
where
  indiv ls           = length ls <= 1
  solve             = id
  divide (l:ls)     = [[ x | x <- ls, x <= l ],
                      [ x | x <- ls, x > l ]]
  combine (l:_) [l1,l2] = l1 ++ [l] ++ l2
```

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

214/196

Counterexample: Fibonacci Numbers (Pitfall!)

...not every problem that can be modeled as a 'divide and conquer' problem is also (directly) suitable for it.

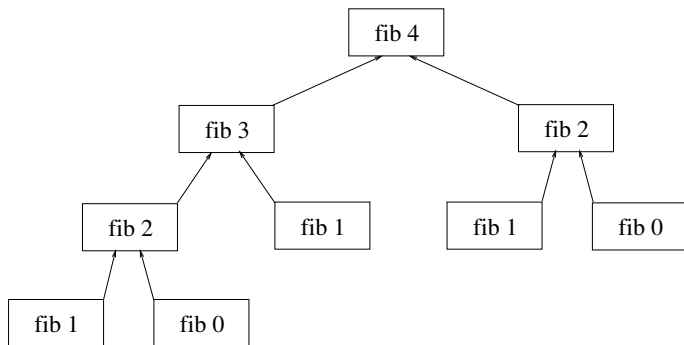
Consider:

```
fib :: Integer -> Integer
fib n
  = divide_and_conquer indiv solve divide combine n
  where
    indiv n      = (n == 0) || (n == 1)
    solve n
      | n == 0   = 0
      | n == 1   = 1
      | otherwise = error "Problem must be divided"
    divide n     = [n-2,n-1]
    combine _ [11,12] = 11 + 12
```

...shows exponential runtime behaviour due to recomputations!

Illustrating

...the [divide-and-conquer computation](#) of the Fibonacci numbers (recomputing the solution to many subproblems!):



Fethi Rabhi, Guy Lapalme.

Algorithms: A Functional Programming Approach.

Addison-Wesley, 1999, page 179.

Chapter 3.2

Backtracking Search

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

217/196

Backtracking Search

Given: A problem instance P .

Sought: A solution S of P .

Algorithmic Idea:

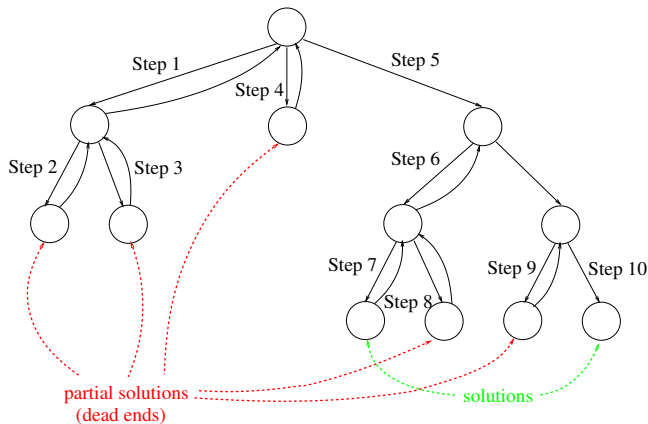
- Search for a particular **solution** of the problem by a **systematic trial-and-error** exploration of the solution space.

Applicability Requirements:

- A set of all possible situations or nodes constituting the **search (node) space**; these are the potential solutions that need to be explored.
- A set of legal moves from a node to other nodes, called the **successors** of that node.
- An **initial node**.
- A **goal node**, i.e., the solution.

Illustrating the Backtracking Search Principle

...general stages of a **backtracking** algorithm:



Fethi Rabhi, Guy Lapalme.

Algorithms: A Functional Programming Approach.

Addison-Wesley, 1999, page 162.

Illustrating Backtracking Search (Cont'd)

Underlying assumptions

- When exploring the graph, each visited path can lead to the goal node with an equal chance.
- Sometimes, however, it might be known that the current path will not lead to the solution.
- In such cases, one **backtracks** to the next level up the tree and tries a different alternative.

Note

- The above process is similar to a **depth-first** graph traversal; this is illustrated in the preceding figure.
- Not all backtracking algorithms stop when the first goal node is reached.
- Some backtracking algorithms work by selecting all valid solutions in the search space.

Implementing Backtracking Search as HoF (1)

Setting:

A `problem` with

- problem instances of `kind p`
- solution instances of `kind s`

Objective:

A higher-order function (HoF) `search_dfs` solving

- suitably parameterized `problem` instances of `kind p` using the ‘backtracking’ principle.

Implementing Backtracking Search as HoF (2)

Note

- Often, the search space is large.

In such cases, the **graph** forming the **search space**

- should not be stored explicitly, i.e., in its entirety, in memory (using **explicitly** represented graphs) but
- be generated on-the-fly as computation proceeds (using **implicitly** represented graphs).

This requires

- a problem-dependent instance of type variable **node** representing information of nodes in the search space
- a **successor** function **succ** of type **(node -> [node])**, which generates the list of successors of a node, i.e., the nodes of its **local environment**.

Implementing Backtracking Search as HoF (3)

Implementation [assumptions](#):

- The search space graph is acyclic and implicitly stored.
- All solutions shall be computed (Note: The HoF can be adjusted to terminate after finding the first solution.)

The [arguments](#) of `search_dfs`:

- `node`: A type representing node information.
- `succ :: node -> [node]`: A function yielding the list of successors of a node (its local environment).
- `goal :: node -> Bool`: A function checking whether a node is a solution.

Implementing Backtracking Search as HoF (4)

The HoF Implementation:

```
search_dfs :: (Eq node) => (node -> [node]) ->
```

Computing successors

```
(node -> Bool) ->
```

Solution?

```
node -> [node]
```

Initial node *Solution nodes*

```
search_dfs succ goal n -- n for node
```

```
= (search (push n empty))
```

```
where
```

```
search s -- s for stack
```

```
| is_empty s = []
```

```
| goal (top s) = top s : search (pop s)
```

```
| otherwise
```

```
  = let m = top s
```

```
    in search (foldr push (pop s) (succ m))
```


Interface and Behaviour Specification

...of the abstract data type (ADT) stack, named `Stack` (user-visible), cf. Chapter 8.2:

```
module Stack (Stack,empty,is_empty,push,pop,top)
                                where
-- Interface Spec.: Signatures of stack operations
empty    :: Stack a
is_empty :: Stack a -> Bool
push     :: a -> Stack a -> Stack a
pop      :: Stack a -> Stack a
top      :: Stack a -> a

-- Behaviour Spec.: Laws for stack operations
Laws (1) thru (6) -- cf. Chapter 8.2.
```

Implementation A

... of the ADT stack as an algebraic data type (user-invisible):

```
data Stack a    = Empty | Stk a (Stack a)
empty           = Empty
is_empty Empty = True
is_empty _     = False
push x s       = Stk x s
pop Empty      = error "Stack is empty"
pop (Stk _ s)  = s
top Empty      = error "Stack is empty"
top (Stk x _)  = x
```

Implementation B

... of the ADT stack as a **new type (user-invisible)**:

```
newtype Stack a    = Stk [a]
empty              = Stk []
is_empty (Stk []) = True
is_empty (Stk _)  = False
push x (Stk xs)   = Stk (x:xs)
pop (Stk [])      = error "Stack is empty"
pop (Stk (_:xs))  = Stk xs
top (Stk [])      = error "Stack is empty"
top (Stk (x:_))   = x
```

Typical Applications of Backtracking Search

Application fields such as

- Knapsack problems
- Game strategies
- ...

Especially

- The eight-tile problem
- The n -queens problem
- Towers of Hanoi
- ...

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

Chap. 4

Part III

Chap. 5

Chap. 6

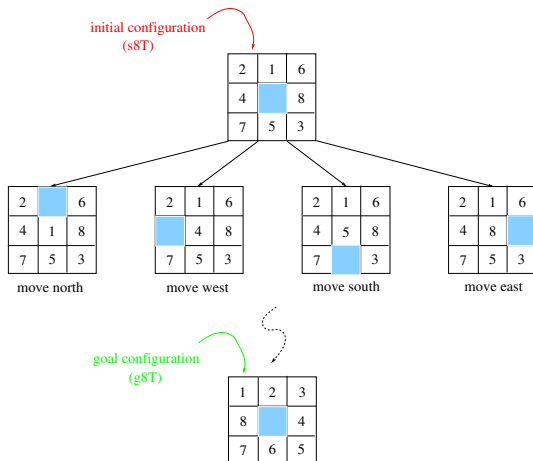
Part IV

Chap. 7

Chap. 8

Chap. 9

Example: The Eight-Tile Problem (8TP)



Fethi Rabhi, Guy Lapalme.
Algorithms: A Functional Programming Approach.
Addison-Wesley, 1999, page 160.

A Backtracking Search Impl. for 8TP (1)

Modeling the board:

```
type Position = (Int,Int)
type Board    = Array Int Position
```

The initial board (initial configuration):

```
s8T :: Board
s8T = array (0,8) [(0,(2,2)),(1,(1,2)),(2,(1,1)),
                  (3,(3,3)),(4,(2,1)),(5,(3,2)),
                  (6,(1,3)),(7,(3,1)),(8,(2,3))]
```

The final board (goal configuration):

```
g8T :: Board
g8T = array (0,8) [(0,(2,2)),(1,(1,1)),(2,(1,2)),
                  (3,(1,3)),(4,(2,3)),(5,(3,3)),
                  (6,(3,2)),(7,(3,1)),(8,(2,1))]
```

A Backtracking Search Impl. for 8TP (2)

Computing the distance of board fields (Manhattan distance = horizontal plus vertical distance):

```
mandist :: Position -> Position -> Int
mandist (x1,y1) (x2,y2) = abs (x1-x2) + abs (y1-y2)
```

Computing all moves (board fields are adjacent iff their Manhattan distance equals 1):

```
allMoves :: Board -> [Board]
allMoves b = [b//[0,b!i),(i,b!0)]
              | i<-[1..8], mandist (b!0) (b!i)==1]
```

...the list of configurations reachable in one move is obtained by placing the space at position *i* and indicating that tile *i* is now where the space was.

A Backtracking Search Impl. for 8TP (3)

Modeling nodes in the search graph:

```
data Boards = BDS [Board]
```

...corresponds to the intermediate configurations from the initial configuration to the current configuration in reverse order.

The **successor** function:

```
succ8Tile :: Boards -> [Boards]
succ8Tile (BDS (n@(b:bs)))
  = filter (notIn bs) [BDS (b':n) | b' <- allMoves b]
where
  notIn bs (BDS (b:_))
    = not (elem (elems b) (map elems bs))
```

...computes all successors that have not been encountered before; the `notIn`-test ensures that only nodes are considered that have not been encountered before.

A Backtracking Search Impl. for 8TP (4)

The `goal` function:

```
goal8Tile :: Boards -> Bool
goal8Tile (BDS (n:_)) = elems n == elems g8T
```

Putting things together:

A `depth-first search` producing the first sequence of moves (in reverse order), which lead to the goal configuration:

```
dfs8Tile :: [[Position]]
dfs8Tile = map elems ls
  where ((BDS ls):_)
        = search_dfs succ8Tile goal8Tile (BDS [s8T])
```

Chapter 3.3

Priority-first Search

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Priority-first Search (1)

Given: A problem instance P .

Sought: A solution S of P .

Algorithmic Idea

- Similar to **backtracking search**, i.e., searching for a particular **solution** of the problem by a **systematic trial-and-error** exploration of the search space **but** the candidate nodes are ordered such that always **the most promising node is first** (**priority-first search/best-first search**).

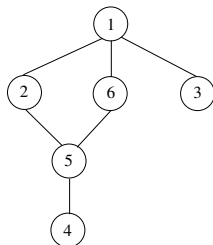
Note: While plain backtracking search proceeds **unguidedly** and can thus be considered **blind**, priority-first search/best-first search benefits from (hopefully accurate) information pointing it towards the ‘most promising’ node.

Priority-first Search (2)

Applicability Requirements

- A set of all possible situations or nodes constituting the **search (node) space**; these are the potential solutions that need to be explored.
- A **comparison criterion** for comparing and ordering candidate nodes wrt their (expected) 'quality' to investigate 'more promising' nodes before 'less promising' nodes.
- A set of legal moves from a node to other nodes, called the **successors** of that node.
- An **initial node**.
- A **goal node**, i.e., a solution.

Illustrating Different Search Strategies



Fethi Rabhi, Guy Lapalme.

Algorithms: A Functional Programming Approach.

Addison-Wesley, 1999, page 167.

Nodes above are ordered according to their identifier value ('smaller' means 'more promising'):

- **Depth-first search** proceeds using ord.: [1,2,5,4,6,3]
- **Breadth-first search** proceeds using ord.: [1,2,6,3,5,4]
- **Priority-first search** can use the most promising ordering, i.e.: [1,2,3,5,4,6].

Implementing Priority-first Search as HoF (1)

Setting:

A `problem` with

- problem instances of `kind p`
- solution instances of `kind s`

Objective:

A higher-order function (HoF) `search_pfs` solving

- suitably parameterized `problem instances of kind p` using the ‘`priority-first/best-first`’ principle.

Implementing Priority-first Search as HoF (2)

Implementation [assumptions](#):

- The search space graph is acyclic and implicitly stored.
- All solutions shall be computed (Note: The HoF can be adjusted to terminate after finding the first solution.)

The [arguments](#) of [search_pfs](#):

- [node](#): A type representing node information.
- [<=](#): A comparison criterion for nodes; usually, this is the relator [<=](#) of the type class [Ord](#). Often, the relator [<=](#) can not exactly be defined but only in terms of a plausible heuristics.
- [succ](#) :: [node](#) -> [[node](#)]: A function yielding the list of successors of a node (its local environment).
- [goal](#) :: [node](#) -> [Bool](#): A function checking whether a node is a solution.

Implementing Priority-first Search as HoF (3)

The HoF Implementation:

```
search_pfs :: (Ord node) => (node -> [node]) ->
              Computing successors
              (node -> Bool) ->
              Solution?
              node -> [node]
              Initial node Solution nodes
```

```
search_pfs succ goal n -- n for node
= search (enPQ n emptyPQ)
  where
    search pq -- pq for priority queue
    | is_emptyPQ pq = []
    | goal (frontPQ pq) = frontPQ pq : search (dePQ pq)
    | otherwise
      = let m = frontPQ pq
          in search (foldr enPQ (dePQ pq) (succ m))
```


Interface and Behaviour Specification

...of the abstract data type (ADT) priority queue, named `PQueue` (user-visible), cf. Chapter 8.3:

```
module PQueue (PQueue,emptyPQ,is_emptyPQ,
               enPQ,dePQ,frontPQ) where

-- Interface Spec.: Signatures of priority queue
--                   operations
emptyPQ      :: PQueue a
is_emptyPQ   :: PQueue a -> Bool
enPQ         :: (Ord a) => a -> PQueue a -> PQueue a
dePQ         :: (Ord a) => PQueue a -> PQueue a
frontPQ      :: (Ord a) => PQueue a -> a

-- Behaviour Spec.: Laws for priority queue operations
...

```

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

241/196

Implementation

...of the ADT priority queue as a new type (user-invisible):

```
newtype PQueue a    = PQ [a]
emptyPQ             = PQ []
is_emptyPQ (PQ []) = True
is_emptyPQ _       = False
enPQ x (PQ pq)     = PQ (insert x pq)
  where
    insert x []           = [x]
    insert x r@(e:r') | x <= e = x:r' -- the smaller the
                                     -- higher the priority
                          | otherwise = e:insert x r'

dePQ (PQ [])          = error "Priority queue is empty"
dePQ (PQ (_:xs))     = PQ xs

frontPQ (PQ [])      = error "Priority queue is empty"
frontPQ (PQ (x:_))  = x
```

Typical Applications of Priority-first Search

Application fields such as

- Game strategies
- ...

Especially

- The eight-tile problem
- ...

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

243/196

Example: A Priority-first Search for 8TP

Comparing nodes heuristically: ...by summing the distance of each square from its home position to its destination as an estimate of the number of moves that will be required to transform the current node into the goal node.

```
heur :: Board -> Int
heur b = sum [mandist (b!i) (g8T!i) | i<-[0..8]]

instance Eq Boards
  where BDS (b1:_) == BDS (b2:_) = heur b1 == heur b2

instance Ord Boards
  where BDS (b1:_) <= BDS (b2:_) = heur b1 <= heur b2

pfs8Tile :: [[Position]]
pfs8Tile = map elems ls
  where ((BDS ls):_)
    = search_pfs succ8Tile goal8Tile (BDS [s8T])
```

Chapter 3.4

Greedy Search

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Greedy Search (1)

Given: A problem instance P .

Sought: A solution S of P .

Algorithmic Idea

- Similar to priority-first/best-first search but limiting the search to immediate successors of a node (greedy search/hill climbing search).

Note: Maintaining the priority queue in priority-first search may be costly in terms of time and memory. Greedy search avoids this time and memory penalty by maintaining a much smaller priority queue considering immediate successors only (the search commits itself to each step taken during the search). Hence, only a single path of the search space is explored instead of its entirety what ensures efficiency. Optimality, however, requires the absence of local minimums.

Greedy Search (2)

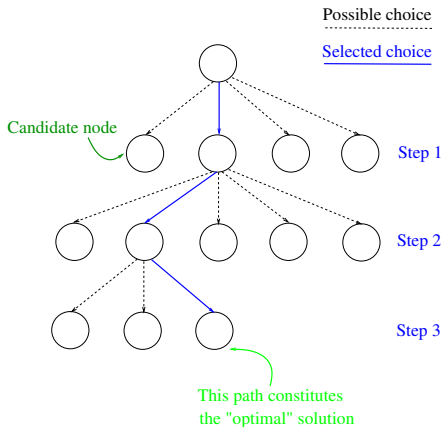
Applicability Requirements

- A set of all possible situations or nodes constituting the **search (node) space**; these are the potential solutions that need to be explored.
- A set of legal moves from a node to other nodes, called the **successors** of that node.
- An **initial node**.
- A **goal node**, i.e., a solution.
- There shall be **no local minimums**, i.e., **no locally best solutions**.

Note: If local minimums exist but are known to be 'close' (enough) to the optimal solution, a greedy search might still be giving a reasonably 'good,' not necessarily optimal solution. Greedy search then becomes a heuristic algorithm.

Illustrating the Greedy Search Principle

...successive stages of a greedy algorithm:



Fethi Rabhi, Guy Lapalme.
Algorithms: A Functional Programming Approach.
Addison-Wesley, 1999, page 171.

Implementing Greedy Search as HoF (1)

Setting:

A `problem` with

- problem instances of `kind p`
- solution instances of `kind s`

Objective:

A higher-order function (HoF) `search_greedy` solving

- suitably parameterized `problem instances of kind p` using the ‘greedy/hill climbing’ principle.

Implementing Greedy Search as HoF (2)

Implementation [assumptions](#):

- The search space graph is acyclic and implicitly stored.
- There are no local minimums, i.e., no locally best solutions.

The [arguments](#) of `search_greedy`:

- `node`: A type representing node information.
- `<=`: A comparison criterion for nodes; usually, this is the relator `<=` of the type class `Ord`.
- `succ :: node -> [node]`: A function yielding the list of successors of a node (its local environment).
- `goal :: node -> Bool`: A function checking whether a node is a solution.

Implementing Greedy Search as HoF (3)

The HoF Implementation:

```
search_greedy :: (Ord node) => (node -> [node]) ->
               Computing successors
               (node -> Bool) ->
               Solution?
               node -> [node]
               Initial node Solution nodes
```

```
search_greedy succ goal n -- n for node
= search (enPQ n emptyPQ)
  where
    search pq -- pq for priority queue
    | is_emptyPQ pq = []
    | goal (frontPQ pq) = [frontPQ pq]
    | otherwise
      = let m = frontPQ pq
        in search (foldr enPQ emptyPQ (succ m))
```

Note

...the essential difference of `search_greedy` compared to `search_pfs` is the replacement of `(dePQ pq)` by `emptyPQ` in the recursive call to `search` to remove old candidate nodes from the `priority` queue:

```
search_pfs: ...search (foldr enPQ (dePQ pq) (succ m))
```

```
search_greedy: ...search (foldr enPQ emptyPQ (succ m))
```

Refer to [Chapter 8.4](#) for details on priority queues as abstract data type (ADT).

Typical Applications of Greedy Search

Application fields such as

- Graph algorithms
- ...

Especially

- Prim's minimum spanning tree algorithm
- The money change problem (MCP)
- ...

Example: A Greedy Search for MCP (1)

Problem statement: Give money change with the least number of coins.

Modeling coins:

```
coins :: [Int]
coins = [1,2,5,10,20,50,100]
```

Modeling nodes (remaining amount of money and change used so far, i.e., the coins that have been returned so far):

```
type NodeChange = (Int,SolChange)
type SolChange  = [Int]
```

Computing successor nodes (by removing every possible coin from the remaining amount):

```
succCoins :: NodeChange -> [NodeChange]
succCoins (r,p) = [(r-c,c:p) | c <- coins, r-c >= 0]
```

Example: A Greedy Search for MCP (2)

The `goal` function:

```
goalCoins :: NodeChange -> Bool
goalCoins (v,_) = v == 0
```

Putting things together:

```
change :: Int -> SolChange
change amount
  = snd (head (search_greedy succCoins goalCoins
                        (amount, [])))
```

Example: `change 199 ->> [2,2,5,20,20,50,100]`

Note: For `coins = [1,3,6,12,24,30]` the above algorithm can yield suboptimal solutions: E.g., `change 48 ->> [30, 12,6]` instead of the optimal solution `[24,24]`.

Chapter 3.5

Dynamic Programming

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Dynamic Programming

Given: A problem instance P .

Sought: A solution S of P .

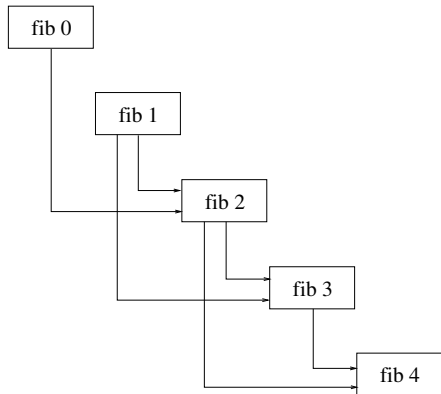
Algorithmic Idea

- Solve (the) smaller instances of the problem first
- Save the solutions of these smaller problem instances
- Use these results to solve larger problem instances

Note: Top-down algorithms as in the previous chapters might suffer from generating a large number of identical subproblems. This replication of work can severely impair performance. Dynamic programming aims at overcoming this shortcoming by systematically precomputing and reusing results in a bottom-up fashion, i.e., from smaller to larger problem instances.

Illustrating Dynamic Programming for fib

...the **dynamic programming computation** of the Fibonacci numbers (**no recomputation** of solutions of subproblems!):



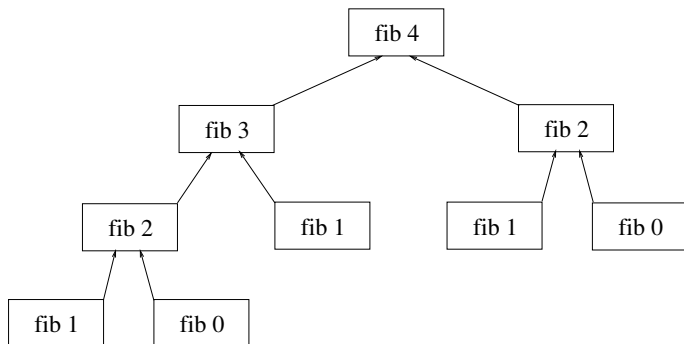
Fethi Rabhi, Guy Lapalme.

Algorithms: A Functional Programming Approach.

Addison-Wesley, 1999, page 179.

Illustrating Divide-and-Conquer for fib

...the **divide-and-conquer computation** of the Fibonacci numbers (**numerous recomputations** of solutions of subproblems!):



Fethi Rabhi, Guy Lapalme.

Algorithms: A Functional Programming Approach.

Addison-Wesley, 1999, page 179.

Implementing Dynamic Programming as HoF (1)

Setting:

A **problem** with

- problem instances of **kind p**
- solution instances of **kind s**

Objective:

A higher-order function (HoF) **dynamic** solving

- suitably parameterized **problem instances of kind p** using the ‘**dynamic programming**’ principle.

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

260/196

Implementing Dynamic Programming as HoF (2)

The `arguments` of `dynamic`:

- `compute :: (Ix coord) => Table entry coord -> coord -> entry`: Given a table and an index, `compute` computes the corresponding entry in the table (possibly using other entries in the table).
- `bnds :: (Ix coord) => (coord, coord)`: The argument `bnds` specifies the boundaries of the table. Since the type of the index is in the class `Ix`, all indices in the table can be generated from these boundaries using the function `range`.

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

261/196

Implementing Dynamic Programming as HoF (3)

The HoF Implementation:

```
dynamic :: (Ix coord) =>
    (Table entry coord -> coord -> entry) ->
    (coord,coord) -> (Table entry coord)
```

Computing the table entry at some coordinates

Specifying table bounds *Result table*

```
dynamic compute bnds = t
  where
    t = newTable (map (\coord -> (coord, compute t coord))
                   (range bnds))
```

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

262/196

Interface/Behaviour Specification

...of the abstract data type (ADT) table, named `Table` (user-visible), cf. Chapter 8.5.2:

```
module Tab (Table', new_T', find_T', upd_T') where

-- Interface Spec.: Signatures of table operations
new_T'  :: (Ix b) => [(b,a)] -> Table' a b
find_T' :: (Ix b) => Table' a b -> b -> a
upd_T'  :: (Ix b) => (b,a) -> Table' a b -> Table' a b

-- Behaviour Spec.: Laws for table operations
...
```

Implementation

...of the ADT table as a new type using array (user-invisible):

```
newtype Table' a b = Tbl' (Array b a)
new_T' assoc_list = Tbl' (array (low,high) assoc_list)
  where indices = map fst assoc_list
        low     = minimum indices
        high    = maximum indices

find (Tbl' a) index = a ! index
upd_T' p@(index,value) (Tbl' a) = Tbl' (a // [p])
```

Note:

- `new_T'` takes an association list of index/value pairs and returns the corresponding table; the boundaries of the new table are determined by computing the maximum and the minimum key in the argument association list.
- `find_T'` and `upd_T'` allow to retrieve and update values in the table. `find_T'` returns a system error, not a user error, when applied to an invalid key.

Typical Applications of Dynamic Programming

Application fields such as

- Graph algorithms
- Search algorithms
- ...

Especially

- Shortest paths for all pairs of nodes of a graph
- Fibonacci numbers
- Chained matrix multiplication
- Optimal binary search (in trees)
- The travelling salesman problem
- ...

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Example: Computing Fibonacci Numbers

Defining the problem-dependent parameters:

```
bndsFibs :: Int -> (Int,Int)
```

```
bndsFibs n = (0,n)
```

```
compFib :: Table Int Int -> Int -> Int
```

```
compFib t i
```

```
  | i <= 1    = i
```

```
  | otherwise = find t (i-1) + find t (i-2)
```

Putting things together:

```
fib :: Int -> Int
```

```
fib n = find t n
```

```
  where t = dynamic compFib (bndsFib n)
```

Chapter 3.6

Dynamic Programming vs. Memoization

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Dynamic Programming vs. Memoization

Overall

- ▶ **Dynamic programming** and **memoization** enjoy very much the same characteristics and offer the programmer quite similar benefits.
- ▶ In practice, differences in behaviour are **minor** and strongly **problem-dependent**.
- ▶ In general, both techniques are **similarly powerful**.

Conceptual difference

- ▶ **Memoization** opportunistically computes and stores argument/result pairs on a by-need basis ('**lazy**' approach).
- ▶ **Dynamic programming** systematically precomputes and stores argument/result pairs before they are needed ('**eager**' approach).

Minor Benefits of Dynamic Programming

- ▶ **Memory efficiency:** For some problems the dynamic programming solution can be adjusted to use asymptotically less memory: **Limited history recurrence**, i.e., only a limited number of preceding values need to be remembered (e.g., two for the computation of Fibonacci numbers) which allows to reuse memory during computation.
- ▶ **Run-time performance:** The systematic programmer-controlled filling of the argument/result pairs table allows sometimes slightly more efficient (by a constant factor) implementations.

Minor Benefits of Memoization

- ▶ **Freedom of conceptual overhead:** The programmer does not need to think about in what order argument/result pairs need to be computed and how to be stored in the memo table. In dynamic programming all table entries are computed systematically when needed.
- ▶ **Freedom of computational overhead:** Only argument/result pairs are computed and stored when needed. In dynamic programming they are systematically precomputed when and before they are needed.

Chapter 3.7

References, Further Reading

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

3.1

3.2

3.3

3.4

3.5

3.6

3.7

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV





Chap. 7

Chap. 8




Chap. 9

271/196




Chapter 3.1–3.4: Further Reading (1)

-  Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974. (Chapter 2.6, Divide-and-conquer)
-  Richard Bird, Philip Wadler. *An Introduction to Functional Programming*. Prentice Hall, 1988. (Chapter 6.4, Divide and Conquer; Chapter 6.5, Search and Enumeration)
-  James R. Bitner, Edward M. Reingold. *Backtrack Programming Techniques*. Communications of the ACM 18(11):651-656, 1975.
-  Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001. (Chapter 16, Greedy Algorithms)





Chapter 3.1–3.4: Further Reading (2)

-  Jon Kleinberg, Éva Tardos. *Algorithm Design*. Addison-Wesley/Pearson, 2006. (Chapter 4, Greedy Algorithms; Chapter 5, Divide and Conquer)
-  Fethi Rabhi, Guy Lapalme. *Algorithms – A Functional Programming Approach*. Addison-Wesley, 1999. (Chapter 5, Abstract data types; Chapter 8, Top-down design techniques)
-  Gunter Saake, Kai-Uwe Sattler. *Algorithmen und Datenstrukturen – Eine Einführung mit Java*. dpunkt.verlag, 4. überarbeitete Auflage, 2010. (Kapitel 8.2, Algorithmenmuster: Greedy; Kapitel 8.3, Rekursion: Divide-and-conquer; Kapitel 8.4, Rekursion: Backtracking)

Chapter 3.1–3.4: Further Reading (3)

-  Robert Sedgewick. *Algorithmen*. Addison-Wesley/Pearson, 2. Auflage, 2002. (Kapitel 5, Rekursion - Teile und Herrsche; Kapitel 44, Erschöpfendes Durchsuchen - Backtracking)
-  Steven S. Skiena. *The Algorithm Design Manual*. Springer-V, 1998. (Chapter 3.6, Divide and Conquer)
-  Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley/Pearson, 2nd edition, 1999. (Chapter 19.6, Avoiding recomputation: memoization – Greedy algorithms)




Chapter 3.5–3.6: Further Reading (4)

-  Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974. (Chapter 2.8, Dynamic programming)
-  Richard E. Bellman. *Dynamic Programming*. Princeton University Press, 1957.
-  Richard E. Bellman, Stuart E. Dreyfus. *Applied Dynamic Programming*. Princeton University Press, 1957.
-  Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001. (Chapter 15, Dynamic Programming)


Chapter 3.5–3.6: Further Reading (5)

-  Max Hailperin, Barbara Kaiser, Karl Knight. *Concrete Abstractions – An Introduction to Computer Science using Scheme*. Brooks/Cole Publishing Company, 1999. (Chapter 12, Dynamic Programming; Chapter 12.5, Comparing Memoization and Dynamic Programming)
-  Jon Kleinberg, Éva Tardos. *Algorithm Design*. Addison-Wesley/Pearson, 2006. (Chapter 6, Dynamic Programming)
-  Peter Pepper, Petra Hofstedt. *Funktionale Programmierung*. Springer-V., 2006. (Kapitel 16.3.2, Ein allgemeines Schema für die globale Suche)

Chapter 3.5–3.6: Further Reading (6)

-  Fethi Rabhi, Guy Lapalme. *Algorithms – A Functional Programming Approach*. Addison-Wesley, 1999. (Chapter 5, Abstract data types; Chapter 9, Dynamic programming)
-  Gunter Saake, Kai-Uwe Sattler. *Algorithmen und Datenstrukturen – Eine Einführung mit Java*. dpunkt.verlag, 4. überarbeitete Auflage, 2010. (Kapitel 8.5, Dynamische Programmierung)
-  Robert Sedgewick. *Algorithmen*. Addison-Wesley/Pearson, 2. Auflage, 2002. (Kapitel 42, Dynamische Programmierung)

Chapter 3.5–3.6: Further Reading (7)

-  Steven S. Skiena. *The Algorithm Design Manual*. Springer-V., 1998. (Chapter 3.1, Dynamic Programming; Chapter 3.2, Limitations of Dynamic Programming)
-  Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley/Pearson, 2nd edition, 1999. (Chapter 19.6, Avoiding recomputation: memoization – dynamic programming)
-  Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley/Pearson, 3rd edition, 2011. (Chapter 20.6, Avoiding recomputation: memoization – dynamic programming)

Chapter 4

Equational Reasoning for Functional Pearls

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

4.1

4.2

4.3

4.4

4.5

4.6

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

279/196

Chapter 4.1

Equational Reasoning

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

4.1

4.2

4.3

4.4

4.5

4.6

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

280/196

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:

$$\begin{aligned} & (a + b) * (a - b) \\ \text{(Distributivity of } *, +) & = a * a - a * b + b * a - b * b \\ \text{(Commutativity of } *) & = a * a - a * b + a * b - b * b \\ & = a * a - b * b \\ & = a^2 - b^2 \quad \square \end{aligned}$$

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 = a² - b²

denote the **same** function.

Proof. Using **Proposition 4.1.1** and **equational reasoning** we obtain:

$$\begin{aligned} & \mathbf{f} \ a \ b \\ \text{(Definition of } \mathbf{f}, \text{ unfolding } \mathbf{f}) & = (a+b) * (a-b) \\ \text{(Proposition 4.1.1)} & = a^2 - b^2 \\ \text{(Definition of } \mathbf{g}, \text{ folding } \mathbf{g}) & = \mathbf{g} \ a \ b \quad \square \end{aligned}$$

Equational Reasoning: More Examples (1)

Let

$$a = 3$$

$$b = 4$$

$$h :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$$

$$h\ x\ y = x^2 + y^2$$

Proposition 4.1.3

The value of the expression `h a (h a b)` is 634, i.e.:

$$\begin{aligned} h\ a\ (h\ a\ b) &= 634 \quad (\text{in mathematical terms}) \\ (h\ a\ (h\ a\ b)) &== 634 \quad (\text{in Haskell terms}) \end{aligned}$$

Equational Reasoning: More Examples (2)

Proof. By [equational reasoning](#) and the functional definitions of `h`, `a`, and `b` we obtain:

$$\begin{aligned} &= \text{h a (h a b)} \\ \text{(Def. of h, unfolding h)} &= \text{h a (a}^2 + \text{b}^2\text{)} \\ \text{(Definition of a, b)} &= \text{h 3 (3}^2 + \text{4}^2\text{)} \\ &= \text{h 3 (9 + 16)} \\ &= \text{h 3 25} \\ \text{(Def. of h, unfolding h)} &= \text{3}^2 + \text{25}^2 \\ &= \text{9 + 625} \\ &= \text{634} \quad \square \end{aligned}$$

Note that the (Haskell) expression `h a (h a b)` is solely evaluated by [equational reasoning](#) applying [standard algebraic mathematical laws](#) and the Haskell definitions of `h`, `a`, and `b`.

Equational Reasoning: More Examples (3)

Let

```
g :: Int -> Int -> Int
g x y = x^2 - y^2
```

```
k :: Int -> Int -> Int
k x y = x * y
```

Proposition 4.1.4

The expressions $k (a+b) (a-b)$ and $g a b$ have the same value, i.e., $k (a+b) (a-b) = g a b$

(resp. $k (a+b) (a-b) == g a b$ in Haskell terms).

Equational Reasoning: More Examples (4)

Proof. By **equational reasoning** using the functional definitions of k and g we obtain:

$$\begin{aligned} & k \ (a+b) \ (a-b) \\ \text{(Def. of } k, \text{ unfolding } k) &= (a+b) * (a-b) \\ \text{(Distributivity of } *, +) &= a*a - a*b + b*a - b*b \\ \text{(Commutativity of } *) &= a*a - a*b + a*b - b*b \\ &= a*a - b*b \\ &= a^2 - b^2 \\ \text{(Def. of } g, \text{ folding } g) &= g \ a \ b \quad \square \end{aligned}$$

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 proofs of **Proposition 4.1.2** through **4.1.4**.

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.,

$$\forall ls \in \text{a-List}. \text{reverse } ls = \text{fast_reverse } ls$$

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:

$$\begin{aligned} & \text{reverse } ls \\ (ls = []) &= \text{reverse } [] \\ (\text{Unfolding reverse}) &= [] \\ (\text{Folding fr}) &= \text{fr } [] \ [] \\ (\text{Folding fast_reverse}) &= \text{fast_reverse } [] \\ ([] = ls) &= \text{fast_reverse } ls \end{aligned}$$

Proving Theor. 4.1.5: The Functional Pearl (2)

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

$$\begin{aligned} & \text{reverse } ls \\ (lst = (v:ls')) &= \text{reverse } (v:ls') \\ \text{(Unfolding reverse)} &= \text{reverse } ls' ++ [v] \\ \text{(IH)} &= \text{fast_reverse } ls' ++ [v] \\ \text{(Unfolding fast_reverse)} &= (\text{fr } ls' []) ++ [v] \\ \text{(Lemma 4.1.7)} &= \text{fr } ls' [v] \\ \text{(Folding fr)} &= \text{fr } ls' (v:[]) \\ \text{(Folding fr)} &= \text{fr } (v:ls') [] \\ \text{(Folding fast_reverse)} &= \text{fast_reverse } (v:ls') \\ ((v:lst') = ls) &= \text{fast_reverse } ls \quad \square \end{aligned}$$

Proving the Supporting Results (1)

Lemma 4.1.6

$\forall ls1, ls2 \in \text{a-List} \quad \forall v \in \text{a-Value}.$

$$(\text{fr } ls1 \text{ } ls2) ++ [v] = \text{fr } ls1 (ls2 ++ [v])$$

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

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

$$\begin{aligned} & (\text{fr } ls1 \text{ } ls2) ++ [v] \\ (ls1 = []) &= (\text{fr } [] \text{ } ls2) ++ [v] \\ (\text{Unfolding fr}) &= ls2 ++ [v] \\ (\text{Folding fr}) &= \text{fr } [] (ls2 ++ [v]) \\ ([] = ls1) &= \text{fr } ls1 (ls2 ++ [v]) \end{aligned}$$

Proving the Supporting Results (2)

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

$$\begin{aligned} & (fr\ ls1\ ls2) ++ [v] \\ (ls1 = (v':ls1')) &= (fr\ (v':ls1')\ ls2) ++ [v] \\ \text{(Unfolding fr)} &= (fr\ ls1'\ (v':ls2)) ++ [v] \\ (ls3 =_{df}\ (v':ls2)) &= (fr\ ls1'\ ls3) ++ [v] \\ \text{(IH)} &= fr\ ls1'\ (ls3 ++ [v]) \\ ((v':ls2) = ls3) &= fr\ ls1'\ ((v':ls2) ++ [v]) \\ \text{(Def. of } (:)\ \text{and } (++)\text{)} &= fr\ ls1'\ (v':(ls2 ++ [v])) \\ \text{(Folding fr)} &= fr\ (v':ls1')\ (ls2 ++ [v]) \\ ((v':ls1') = ls1) &= fr\ ls1\ (ls2 ++ [v]) \quad \square \end{aligned}$$

Proving the Supporting Results (3)

Lemma 4.1.7

$\forall ls' \in \text{a-List} \quad \forall v \in \text{a-Value}.$

$$(\text{fr } ls' \ []) ++ [v] = \text{fr } ls' [v]$$

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

$$\begin{aligned} & (\text{fr } ls' \ []) ++ [v] \\ (ls'=ls1, []=ls2) &= (\text{fr } ls1 \ ls2) ++ [v] \\ (\text{Lemma 4.1.6}) &= \text{fr } ls1 \ (ls2 ++ [v]) \\ (ls1=ls', ls2=[]) &= \text{fr } ls' \ ([] ++ [v]) \\ ([] ++ [v]=[v]) &= \text{fr } ls' \ [v] \quad \square \end{aligned}$$

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:

$\forall xs \in \text{a-List}. \text{reverse } xs == \text{fast_reverse } xs$

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

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

4.1

4.2

4.2.1

4.2.2

4.3

4.4

4.5

4.6

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chapter 4.2.1

Functional Pearls: The Very Idea

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

4.1

4.2

4.2.1

4.2.2

4.3

4.4

4.5

4.6

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

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 Pearl:
 - 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

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

4.1

4.2

4.3

4.3.1

4.3.2

4.3.3

4.3.4

4.4

4.5

4.6

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

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

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

4.1

4.2

4.3

4.3.1

4.3.2

4.3.3

4.3.4

4.4

4.5

4.6

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

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 initial 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 4.3.2

An Array-based Algorithm and Two Variants

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

4.1

4.2

4.3

4.3.1

4.3.2

4.3.3

4.3.4

4.4

4.5

4.6

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

The Array-Based Algorithm: The Idea

Exploiting **KF**, the array-based **SFNP Algorithm** **LinSFNP** builds a

- **checklist** of those numbers present in **filter** ($\leq n$) **xs**

where **checklist** is implemented as a

- **Boolean array** with $n + 1$ slots, numbered from **0** to **n**, whose entries are initially set to **False**.

Algorithmic idea:

- For each element **x** in **xs** with $x \leq n$ the array element at position **x** is set to **True**.
- The **smallest free number** is then found as the position of the first **False** entry.

The Array-Based Algorithm: Implementation

Implementation of the array-based SFNP Algorithm LinSFNP:

```
minfree = search . checklist
search  :: Array Int Bool -> Int
search = length . takeWhile id . elems
checklist :: [Int] -> Array Int Bool
checklist xs = accumArray (||) False (0,n)
               (zip (filter (<=n) xs) (repeat True))
               where n = length xs
```

Note: The array-based SFNP Algorithm LinSFNP

- does not require the elements of `xs` to be distinct
- but does require them to be natural numbers

Variant A of the Array-Based Algorithm

...the function `accumArray` can be used to

- sort a list of numbers in linear time, provided the elements of the list **all lie in some known range**.
- `checklist` can then be replaced by `countlist`.

```
countlist :: [Int] -> Array Int Int
countlist xs =
  accumArray (+) 0 (0,n) (zip xs (repeat 1))

sort xs =
  concat [replicate k x | (x,k) <- countlist xs]
```

Replacing `checklist` by `countlist` and `sort`, the implementation of `minfree`

- boils down to finding the first 0 entry.

Variant B of the Array-Based Algorithm

...instead of using a smart library function like `accumArray` as in [Variant A](#), `checklist` can be implemented

- using a [constant-time array update operation](#).

In Haskell, this can be done using a [monad](#), the

- [state monad](#) (cf. `Data.Array.ST`)

```
checklist xs =  
  runSTArray (do  
    {a <- newArray (0,n) False;  
    sequence [writeArray a x True | x<-xs, x<=n];  
    return a})  
  where n = length xs
```

[Note](#), however, that [Variant B](#) is essentially a [procedural program](#) in [functional clothing](#).

Chapter 4.3.3

A Divide-and-Conquer Algorithm

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

4.1

4.2

4.3

4.3.1

4.3.2

4.3.3

4.3.4

4.4

4.5

4.6

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Towards the Divide-and-Conquer Algorithm (1)

Algorithmic idea:

- Express $\text{minfree}(xs ++ ys)$ in terms of $\text{minfree}(xs)$ and $\text{minfree}(ys)$.

To this end, we first collect some properties of [set differences](#):

Lemma 4.3.3.1

$$(as ++ bs) \setminus cs = (as \setminus cs) ++ (bs \setminus cs)$$

$$as \setminus (bs ++ cs) = (as \setminus bs) \setminus cs$$

$$(as \setminus bs) \setminus cs = (as \setminus cs) \setminus bs$$

Lemma 4.3.3.2

If as and vs are disjoint (i.e., $as \setminus vs = as$), and bs and us are disjoint (i.e., $bs \setminus us = bs$), we have:

$$(as ++ bs) \setminus (us ++ vs) = (as \setminus us) ++ (bs \setminus vs)$$

Towards the Divide-and-Conquer Algorithm (2)

Lemma 4.3.3.3

Let b be a natural number, and let

- $as = [0..b-1]$, $bs = [b..]$
- $us = \text{filter } (<b) \text{ } xs$, $vs = \text{filter } (>=b) \text{ } xs$

Then: as and vs are disjoint, and bs and us are disjoint.

Lemma 4.3.3.3 implies:

Corollary 4.3.3.4

$$[0..] \setminus xs = ([0..b-1] \setminus us) ++ ([b..] \setminus vs)$$

where $(us, vs) = \text{partition } (<b) \text{ } xs$

where `partition` is a Haskell library function which partitions a list into those elements satisfying some property and those that do not.

The Divide-and-Conquer Algorithm LinSFNP'

Together with

```
head (xs++ys) = if null xs then head ys else head xs
```

we get:

The **Basic Divide&Conquer SNFP Algorithm LinSFNP'**:

```
minfree xs = if (null ([0..b-1]) \\ us)
              then (head ([b..]) \\ vs)
              else (head ([0..]) \\ us)
              where (us,vs) = partition (<b) xs
```

...for any natural number **b**.

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

4.1

4.2

4.3

4.3.1

4.3.2

4.3.3

4.3.4

4.4

4.5

4.6

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Optimizing LinSFNP' (1)

Note, evaluating the test

- `(null ([0..b-1]) \\ us)` straightforwardly takes quadratic time in the length of `us`.

Note, too, the lists `[0..b-1]` and `us` are lists of

- `distinct` natural numbers
- every element of `us` is less than `b`.

Together, this allows us to replace the test by a test on the length of `us`:

```
null ([0..b-1] \ us) = length us == b
```

Note, unlike for the array-based algorithm, it is crucial that the argument list `does not contain duplicates` to obtain an

- `efficient` divide-and-conquer algorithm.

Optimizing LinSFNP' (2)

...inspecting `minfree` in more detail reveals that it can be generalized to a function `minfrom`:

```
minfrom :: Nat -> [Nat] -> Nat
minfrom a xs = head ([a..] \\ xs)
```

where every element of `xs` is assumed to be greater than or equal to `a`.

Optimizing LinSFNP' (3)

...provided that b is chosen such that both

– $\text{length } us$ and $\text{length } vs$ are less than $\text{length } xs$

the following recursive definition of minfree is well-defined:

$\text{minfree } xs = \text{minfrom } 0 \ xs$

$\text{minfrom } a \ xs \mid \text{null } xs = a$
 $\mid \text{length } us == b-a = \text{minfrom } b \ vs$
 $\mid \text{otherwise} = \text{minfrom } a \ us$
where $(us,vs) = \text{partition } (<b) \ xs$

Optimizing LinSFNP' (4)

...we are left with picking b appropriately.

The value of b must satisfy:

- $b > a$
- The maximum of the lengths of us and vs is minimum.

This is ensured, if the value of b is chosen as

$$b = a + 1 + n \text{ 'div' } 2 \quad \text{with} \quad n = \text{length } xs.$$

Optimizing LinSFNP' (5)

Note that

- $n \neq 0$ and $\text{length } us < b-a$ implies
 $(\text{length } us) \leq (n \text{ 'div' } 2) < n$
- $\text{length } us = b-a$ implies
 $(\text{length } vs) = (n - (n \text{ 'div' } 2) - 1) \leq n \text{ 'div' } 2$

With this choice, the number of steps for evaluating

`minfrom 0 xs`

is *linear* in the number of elements of `xs`.

The Final Div&Conqu. Algorithm LinSFNP''

As the final optimization, we represent `xs` by a pair `(length xs, xs)` in order to avoid to repeatedly compute `length`.

The Optimized Divide&Conquer SFNP Algorithm LinSFNP'':

```
minfree xs = minfrom 0 (length xs, xs)
minfrom a (n,xs)
  | n == 0      = a
  | m == b-a    = minfrom b (n-m,vs)
  | otherwise   = minfrom a (m,us)
  where (us,vs) = partition (<b) xs
        b       = a + 1 + n div 2
        m       = length us
```

Chapter 4.3.4

In Closing

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

4.1

4.2

4.3

4.3.1

4.3.2

4.3.3

4.3.4

4.4

4.5

4.6

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

In Closing (1)

The **SFN Problem** is not artificial: It can be considered

- a simplified version of the common programming task to find some object which is not in use: **Numbers** then name objects, and **X** the set of objects which are currently in use.

The optimized divide-and-conquer **SFNP Algorithm LinSFNP** is about

- **twice as fast** as the incremental **array-based SFNP Algorithm LinSFNP**
- **20% faster** than **Variant A** of **LinSFNP** using the library function **accumArray**.

In Closing (2)

For a 'procedural' programmer

- an array-update operation takes **constant** time in the size of the array.

For a 'pure functional' programmer

- an array-update operation takes **logarithmic** time in the size of the array.

This different perception explains why there sometimes

- seems to be a **logarithmic gap** between the **best functional** and the **best procedural** solution to a problem.

Sometimes, however, this gap

- **vanishes** as for the **SFN Problem**.

Chapter 4.4

Not the Maximum Segment Sum

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

4.1

4.2

4.3

4.4

4.4.1

4.4.2

4.4.3

4.5

4.6

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

339/196

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

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

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 as a showcase 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 $[2, 1, -2, -1, -4, -3, -7, -4]$.
The subsequence $[2, 1, -2, -1]$ is a **segment**.
The subsequence $[2, 1, -1]$ is a **non-segment** because it has a ‘hole’ at the position of -2 .
The sum of the elements from the non-segment $[2, 1, -1]$ is $2 + 1 + (-1) = 2$.

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

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

4.1

4.2

4.3

4.4

4.4.1

4.4.2

4.4.3

4.5

4.6

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

344/196

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.

Implementing nonsegs

The implementation of function `nonsegs`:

```
nonsegs :: [a] -> [[a]]
nonsegs = extract . filter nonseg . markings
```

relies on the supporting functions:

- `extract`
- `nonseg`
- `markings`

the latter, `markings`, relying on another supporting function:

- `booleans`

Implementing the Supporting Functions

...of `nonseqs`:

```
markings :: [a] -> [[(a,Bool)]]
markings xs = [zip xs bs |
                bs <- booleans (length xs)]
```

```
booleans 0 = [[]]
booleans (n+1) = [b:bs | b <- [True,False],
                      bs <- booleans n]
```

```
extract :: [[(a,Bool)]] -> [[a]]
extract = map (map fst . filter snd)
```

```
nonseq :: [(a.Bool)] -> Bool
nonseq... (completed soon)
```

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

4.1

4.2

4.3

4.4

4.4.1

4.4.2

4.4.3

4.5

4.6

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

348/196

Notes on markings, booleans, and extract

...the intuition underlying their definitions.

To define the function `nonsegs`

- each element of the argument list is `marked` with a Boolean value: `True` indicates that the element is included in the non-segment; `False` indicates that it is not.

This `marking`

- takes place in all possible ways, done by the function `marking` (Note: Markings are in one-to-one correspondence with subsequences.)

Then

- the function `extract` filters for those markings that correspond to a non-segment, and then extracts those whose elements are marked `True`.

Notes on nonseg

The function

- `nonseg :: [(a,Bool)] -> Bool` returns `True` on a list `xms` iff `map snd xsm` describes a non-segment marking (the implementation of `nonseg` is given later).

Note:

The Boolean list `ms` is a non-segment marking iff it is an element of the set represented by the regular expression

$$F^* T^+ F^+ T (T + F)^*$$

where `True` and `False` are abbreviated by `T` and `F`, respectively.

The regular expression identifies the leftmost gap `T+F+T` that makes the segment a non-segment.

A Finite State Automaton

...for recognizing members of the corresponding regular set:

data State = E | S | M | N

Note, the 4 states of the above automaton are used as follows:

- **E** (for **Empty**), starting state: if in **E**, markings only in the set F^* have been recognized.
- **S** (for **Suffix**): if in state **S**, one or more T s have been processed; hence, this indicates markings in the set F^*T^+ , i.e., a non-empty suffix of T s.
- **M** (for **Middle**): if in state **M**, this indicates the processing of markings in the set $F^*T^+F^+$, i.e., a middle segment.
- **N** (for **Non-segment**): if in state **N**, this indicates the processing of non-segments markings.

Implementing nonseg

The Implementation of function `nonseg`:

```
nonseg = (== N) . foldl step E . map snd
```

where the middle term `foldl step E` executes the step of the finite automaton:

```
step E False = E      step M False = M
step E True  = S      step M True  = N
step S False = M      step N False = N
step S True  = S      step N True  = N
```

Note:

- Finite automata process their input from left to right. This leads to the use of `foldl`.
- The input could have been processed from right to left as well, looking for the rightmost gap. This, however, would be less conventional without any benefit from breaking the left to right processing convention.

Chapter 4.4.2

The Linear Time Algorithm

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

4.1

4.2

4.3

4.4

4.4.1

4.4.2

4.4.3

4.5

4.6

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

353/196

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.

Towards the Linear Time Algorithm (1)

First, we introduce the function `pick`:

```
pick :: State -> [a] -> [[a]]
pick q
  = extract .
      filter ((== q) . foldl step E . map snd) .
      markings
```

Note:

- `nonsegs = pick N` (cf. Lemma 4.4.2.1(1))

Towards the Linear Time Algorithm (2)

...properties of function `pick`: By (1) calculation from the definition of `pick q` (which is tedious!) or by (2) referring to the definition of `step` we can prove [Lemma 4.4.2.1](#):

Lemma 4.4.2.1

```
pick N                = nonseqs
pick E xs             = [[]]
pick S []             = []
pick S (xs++[x])     = map (++[x])
                      (pick S xs) ++ pick E xs)
pick M []            = []
pick M (xs++[x])    = pick M xs ++ pick S xs
pick N []            = []
pick N (xs++ys)     = pick N xs ++
                      map (++[x])
                      (pick N xs) ++ pick M xs)
```

Towards the Linear Time Algorithm (3)

...next, we recast the definition of `pick` as an instance of `foldl`.

To this end, let `pickall` be specified by:

```
pickall xs = (pick E xs, pick S xs,  
             pick M xs, pick N xs)
```

This allows us to express `pickall` as an instance of `foldl`:

```
pickall = foldl step ([[]], [], [], [])  
step (ess, nss, mss, sss) x  
  = (ess,  
     map (++[x]) (sss++ess),  
     mss ++ sss,  
     nss ++ map (++[x]) (nss++mss))
```

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

4.1

4.2

4.3

4.4

4.4.1

4.4.2

4.4.3

4.5

4.6

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

357/196

Two new Algorithms for the MNSS Problem

The 1st new Algorithm for the MNSS Problem:

```
mnss = maximum . map sum . fourth . pickall
```

where `fourth` returns the fourth element of a quadruple.

Using function `tuple`

```
tuple f (w,x,y,z) = (f w, f x, f y, f z)
```

`fourth` can be moved to the front of the defining expression of `mnss`:

```
maximum . map sum . fourth  
  = fourth . tuple (maximum . map sum)
```

This allows the 2nd new Algorithm for the MNSS Problem:

```
mnss = fourth . tuple (maximum . map sum) . pickall
```

The Fusion Law of foldl

Lemma 4.4.2.2 (Fusion Law of foldl)

$$f (\text{foldl } g \ a \ xs) = \text{foldl } h \ b \ xs$$

for all finite lists xs provided that for all x and y holds:

$$f \ a = b$$

$$f \ (g \ x \ y) = h \ (f \ x) \ y$$

Towards Applying the Fusion Law (1)

...in our scenario this means application to the instantiations:

```
f = tuple (maximum . map sum)
g = step
a = ([[]], [], [], [])
```

We are now left with finding **h** and **b** to satisfy the conditions of the fusion law.

Because the maximum of an empty set of numbers is $-\infty$, we have:

```
tuple (maximum . map sum) ([[]], [], [], [])
  = (0,  $-\infty$ ,  $-\infty$ ,  $-\infty$ )
```

...which gives the definition of **b**.

Towards Applying the Fusion Law (2)

The definition of `h` needs to satisfy the equation:

```
tuple (maximum . map sum) (step (ess,sss,mss,nss) x)
  = h (tuple (maximum . map sum) (ess,sss,mss,nss)) x
```

Next, we derive `h` by investigating each component in turn. This is demonstrated for the fourth component in detail (the reasoning for the first three components is similar).

Towards Applying the Fusion Law (3)

`max` is used below as an abbreviation for `maximum`:

$$\begin{aligned} & \text{max (map sum (nss ++ map (++ [x]) (nss ++ mss)))} \\ = & \text{ (definition of map)} \\ & \text{max (map sum nss ++ map (sum . (++ [x])) (nss ++ mss))} \\ = & \text{ (since sum . (++ [x]) = (+ x) . sum)} \\ & \text{max (map sum nss ++ map ((+ x) . sum) nss ++ mss)} \\ = & \text{ (since max (xs ++ ys) = (max xs) max (max ys))} \\ & \text{max (map sum nss) max max (map ((+ x) . sum) (nss ++ mss))} \\ = & \text{ (since max . map (+ x) = (+ x) . max)} \\ & \text{max (map sum nss) max (max (map sum (nss ++ mss)) + x)} \\ = & \text{ (introducing } n = \text{max (map sum nss) and} \\ & \quad m = \text{max (map sum mss))} \\ & n \text{ max ((n max m) + x)} \end{aligned}$$

Towards Applying the Fusion Law (4)

Finally, we arrive at the implementation of `h`:

```
h (e, s, m, n) x
  = (e, (s max e)+x, m max s, n max ((n max m) + x))
```

This allows the [3rd new Algorithm for the MNSS Problem](#):

```
mnss = fourth . foldl h (0,-∞,-∞,-∞)
```

The Linear Time Algorithm

We are left with dealing with the fictitious ∞ values.

Here, we eliminate them entirely by considering the first three elements of the list separately, which gives us:

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)
```

Chapter 4.4.3

In Closing

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

4.1

4.2

4.3

4.4

4.4.1

4.4.2

4.4.3

4.5

4.6

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

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.

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

4.1

4.2

4.3

4.4

4.4.1

4.4.2

4.4.3

4.5

4.6

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

367/196

Chapter 4.5

A Simple Sudoku Solver

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

4.1

4.2

4.3

4.4

4.5

4.5.1

4.5.2

4.5.3

4.6

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

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

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

4.1

4.2

4.3

4.4

4.5

4.5.1

4.5.2

4.5.3

4.6

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

370/196

IA_1, IA_2 : Two Initial Sudoku Algorithms

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

IA_1 : 1st Initial Sudoku 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 Sudoku 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 [Sudoku 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 Sudoku 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  $\hat{=}$  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.

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

4.1

4.2

4.3

4.4

4.5

4.5.1

4.5.2

4.5.3

4.6

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

376/196

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  $\hat{=}$ 
nodups [] = True                       no_duplicates)
nodups (x:xs) = all (x/=) xs && nodups xs
```

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 `boxes`:

```
boxes :: Matrix a -> Matrix a
boxes = 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:

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

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:

$$\text{rows} \cdot \text{rows} = \text{id} \quad (\text{A})$$

$$\text{cols} \cdot \text{cols} = \text{id} \quad (\text{B})$$

$$\text{boxs} \cdot \text{boxs} = \text{id} \quad (\text{C})$$

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:

$$\text{map rows} \cdot \text{expand} = \text{expand} \cdot \text{rows} \quad (\text{D})$$

$$\text{map cols} \cdot \text{expand} = \text{expand} \cdot \text{cols} \quad (\text{E})$$

$$\text{map boxs} \cdot \text{expand} = \text{expand} \cdot \text{boxs} \quad (\text{F})$$

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

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

4.1

4.2

4.3

4.4

4.5

4.5.1

4.5.2

4.5.3

4.6

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

384/196

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):

$$\begin{aligned} \text{filter nodups} \cdot \text{cp} \\ = \text{filter nodups} \cdot \text{cp} \cdot \text{pruneRow} \end{aligned} \quad (\text{G})$$

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

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

$$\text{filter (p.f)} = \text{map f} \cdot \text{filter p} \cdot \text{map f} \quad (\text{H})$$

$$\text{filter (all p)} \cdot \text{cp} = \text{cp} \cdot \text{map (filter p)} \quad (\text{I})$$

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: boxes Case (1)

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

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

4.1

4.2

4.3

4.4

4.5

4.5.1

4.5.2

4.5.3

4.6

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

389/196

Proof Sketch of Lemma 4.5.2.1: boxes Case (2)

$$\begin{aligned} &= \{(I)\} \\ &\quad \text{map boxes} \cdot \text{filter} \text{ (all nodups)} \cdot \text{cp} \cdot \\ &\quad \quad \text{map cp} \cdot \text{map pruneRow} \cdot \text{boxes} \\ &= \{\text{definition of expand}\} \\ &\quad \text{map boxes} \cdot \text{filter} \text{ (all nodups)} \cdot \text{expand} \cdot \\ &\quad \quad \text{map pruneRow} \cdot \text{boxes} \\ &= \{(H) \text{ in the form } \text{map } f \cdot \text{filter } p = \\ &\quad \quad \quad \text{filter } (p \cdot f) \cdot \text{map } f\} \\ &\quad \text{filter} \text{ (all nodups} \cdot \text{boxes)} \cdot \text{map boxes} \cdot \text{expand} \cdot \\ &\quad \quad \text{map pruneRow} \cdot \text{boxes} \\ &= \{(F)\} \\ &\quad \text{filter} \text{ (all nodups} \cdot \text{boxes)} \cdot \text{expand} \cdot \text{boxes} \cdot \\ &\quad \quad \text{map pruneRow} \cdot \text{boxes} \end{aligned}$$

Summing up

Overall, we have shown:

Lemma 4.5.2.2

```
filter (all nodups . boxes) . expand
  = filter (all nodups . boxes) .
      expand . pruneBy boxes , 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 boxes . 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.

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

4.1

4.2

4.3

4.4

4.5

4.5.1

4.5.2

4.5.3

4.6

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

394/196

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:

$$\begin{aligned} & \text{filter valid} \cdot \text{expand} \\ = & \{ \text{since } \text{expand} = \text{concat} \cdot \text{map } \text{expand} \cdot \text{expand1} \\ & \text{on incomplete matrices} \} \\ & \text{filter valid} \cdot \text{concat} \cdot \text{map } \text{expand} \cdot \text{expand1} \\ = & \{ \text{since } \text{filter } p \cdot \text{concat} = \text{concat} \cdot \text{map } (\text{filter } p) \} \\ & \text{concat} \cdot \text{map } (\text{filter valid} \cdot \text{expand}) \cdot \text{expand1} \\ = & \{ \text{since } \text{filter valid} \cdot \text{expand} = \\ & \text{filter valid} \cdot \text{expand} \cdot \text{prune} \} \\ & \text{concat} \cdot \text{map } (\text{filter valid} \cdot \text{expand} \cdot \text{prune}) \cdot \\ & \text{expand1} \end{aligned}$$

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

4.1

4.2

4.3

4.4

4.5

4.5.1

4.5.2

4.5.3

4.6

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

399/196

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

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

4.1

4.2

4.3

4.4

4.5

4.5.1

4.5.2

4.5.3

4.6

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

401/196

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

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

4.1

4.2

4.3

4.4

4.5

4.6

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7





Chap. 8

Chap. 9





Chap. 10

403/196




Chapter 4: 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. *Fifteen Years of Functional Pearls*. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming (ICFP 2006)*, 215, 2006.





Chapter 4: Further Reading (2)

-  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)
-  Richard Bird. *Thinking Functionally with Haskell*. Cambridge University Press, 2015. (Chapter 5, A simple Sudoku solver; Chapter 6.6, The maximum segment sum)
-  Richard Bird, Philip Wadler. *An Introduction to Functional Programming*. Prentice Hall, 1988. (Chapter 4.3.1, Texts as lines)





Chapter 4: Further Reading (3)

-  Manuel Chakravarty, Gabriele Keller. *Einführung in die Programmierung mit Haskell*. Pearson Studium, 2004. (Kapitel 9, Formale Überlegungen)
-  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)

Chapter 4: Further Reading (4)

-  Jeremy Gibbons. *Functional Pearls – An Editor's Perspective*. www.cs.ox.ac.uk/people/jeremy.gibbons/pearls/
-  David Gries. *The Maximum Segment Sum Problem*. In *Formal Development of Programs and Proofs*. Edsger W. Dijkstra (Ed.), Addison-Wesley (UT Year of Programming Series), 43-45, 1990.
-  Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2007. (Chapter 13, Reasoning about programs)
-  Lambert Meertens. *Functional Pearl: Calculating the Sieve of Eratosthenes*. *Journal of Functional Programming* 14(6):759-763, 2004.

Chapter 4: Further Reading (5)

-  Shin-Cheng Mu. *The Maximum Segment Sum is Back: Deriving Algorithms for two Segment Problems with Bounded Lengths*. In Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Program Manipulation (PEPM 2008), 31-39, 2008.
-  Matti Nykänen. *A Note on the Genuine Sieve of Eratosthenes*. Journal of Functional Programming 21(6):563-572, 2011.
-  Melissa E. O'Neill. *The Genuine Sieve of Eratosthenes*. Journal of Functional Programming 19(1):95-106, 2009.
-  Colin Runciman. *Functional Pearl: Lazy Wheel Sieves and Spirals of Primes*. Journal of Functional Programming 7(2):219-225, 1997.

Part III

Quality Assurance

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

409/196

Chapter 5

Testing

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

Chap. 6

Part IV

Chap. 7

Chap. 8

410/196

Chapter 5.1

Motivation

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

Chap. 6

Part IV

Chap. 7

Chap. 8

411/196

Gaining Confidence in Correctness of Programs

...essentially, three means are at our disposal:

1. **Correctness by Construction** (*a priori*, cf. Chapter 4)
 - Exemplified by the development of **functional pearls**.
2. **Verification** (*a posteriori*, cf. Chapter 6)
 - Rigorous, formal correctness proofs (soundness of the specification, soundness of the implementation).
 - High confidence, high effort (typically).
3. **Testing** (*a posteriori*, Chapter 5)
 - **Ad hoc**: Controllable effort but usually no quantifiable quality statement; hence, a questionable overall value.
 - **Systematically**: Controllable effort, quantifiable quality statement.

...even if conducted systematically, we should keep in mind:

Testing can only show the presence of errors.
Not their absence.

Edsger W. Dijkstra (11.5.1930-6.8.2002)
1972 Recipient of the ACM Turing Award

...nonetheless, testing is often amazingly successful in revealing errors.

Specifications: Basis of any Kind of Testing

...specifications (shall) describe and fix the meaning of programs:

- ▶ **Informally**, e.g., as **commentary** in the program or separately in another document.
 - ↪ **Disadvantage**: often ambiguous, open to interpretation.
- ▶ **Formally**, e.g., in terms of **pre-** and **post-conditions**, in a formal specification language with a precise semantics.
 - ↪ **Advantage**: precise and rigorous, unambiguous.

Requirements for Systematic Testing

'Must' features:

- Specification-based
- Tool-supported
- Automatic

'Nice-to-Have' features:

- Reporting
 - What has been tested?
 - How thoroughly, how comprehensively has been tested?
 - How was success defined?
- Reproducibility, Repeatability
 - Reproducibility of tests
 - Repeatability of tests after program modifications

We consider QuickCheck

...for **systematic testing**, a combinator library enabling tool-supported specification-based in Haskell.

QuickCheck

- ▶ defines a **formal specification language**
...allowing property definitions inside of the Haskell source code.
- ▶ defines a **test data generator language**
...allowing a simple and concise description of a large number of tests.
- ▶ allows **tests** to be **repeated at will**
...ensuring reproducibility.
- ▶ allows **automatic testing** of all properties specified in a module, including the delivery of success/failure reports
...with tests and reports automatically generated.

It is worth noting

...that [QuickCheck](#) and its [property specification](#) and [test data generator languages](#) are

- ▶ examples of [domain-specific embedded languages](#) ...a special strength of functional programming.
- ▶ implemented as a [combinator library](#) in Haskell ...allowing us to make use of the full expressiveness of Haskell when defining properties and test data generators.
- ▶ part of the standard distribution of Haskell (for both [GHC](#) and [Hugs](#); see module [QuickCheck](#)) ...ensuring easy access and immediate usability.

Chapter 5.2

Defining Properties

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

Chap. 6

Part IV

Chap. 7

Chap. 8

418/196

Defining Simple Properties w/ QuickCheck (1)

...simple properties can be defined in terms of Boolean valued functions, so-called predicates.

Example:

Define inside of a Haskell program the (predicate) property:

```
prop_PlusAssociative :: Int -> Int -> Int -> Bool
prop_PlusAssociative x y z = (x+y)+z == x+(y+z)
```

Double-checking `prop_PlusAssociative` with Hugs yields:

```
Main>quickCheck prop_PlusAssociative
OK, passed 100 tests
```

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

Chap. 6

Part IV

Chap. 7

Chap. 8

419/196

Defining Simple Properties w/ QuickCheck (2)

... slightly varying the introductory example.

Replace `Int` by `Float` in the property definition:

```
prop_PlusAssociative' :: Float -> Float -> Float -> Bool
prop_PlusAssociative' x y z = (x+y)+z == x+(y+z)
```

Double-checking `prop_PlusAssociative'` with Hugs might yield:

```
Main>quickCheck prop_PlusAssociative'
Falsifiable, after 13 tests:
1.0
-5.16667
-3.71429
```

Note

- ▶ The type signatures for `prop_PlusAssociative` and `prop_PlusAssociative'` are necessary because of the overloading of `(+)`.
- ▶ If the type signatures were missing, error messages on ambiguous overloading would be issued; intuitively, `QuickCheck` needs to know which test data to generate.
- ▶ Type signatures in predicate definitions allow the `type-specific generation` of test data.
- ▶ Associativity of addition is `falsifiable` for type `Float`; think e.g. of rounding errors.
- ▶ `Success/error reports` are automatically issued and provide information on
 - the number of tests successfully passed
 - a counter example.

A more Advanced Example

...illustrating limitations of property definitions as predicates.

Given:

- A function `insert :: Int -> [Int] -> [Int]`
- A predicate `is_ordered :: [Int] -> Bool`

To be tested:

- Correctness of the `insertion` operation: After inserting an element, the list shall be sorted.

Property definition as a Predicate:

```
prop_InsertOrdered :: Int -> [Int] -> Bool
prop_InsertOrdered x xs = is_ordered (insert x xs)
```

This property, however, is **falsifiable**: It is **naive**, since the argument list `xs` is not required to be sorted itself, and thus **too strong**.

Advanced Features for Property Definitions (1)

...using [new syntactic features](#) for property definitions:

```
prop_InsertOrdered :: Int -> [Int] -> Property
prop_InsertOrdered x xs
  = is_ordered xs ==> is_ordered (insert x xs)
```

Note:

- 'is_ordered xs ==>' adds a [precondition](#) to the property definition; generated [test data](#), which do not match the precondition, are discarded.
- '==>' is thus [not](#) a Boolean operator but [affects the selection](#) of test data; all such operators in [QuickCheck](#) have the result type [Property](#).
- Using ==> amounts to a [trial-and-error](#) approach for test data generation: 'Generate, then check whether the precondition is matched; if not, drop; repeat.'

Advanced Features for Property Definitions (2)

...`QuickCheck` provides further features for property definitions to improve on this:

```
prop_InsertOrdered :: Int -> Property
prop_InsertOrdered x =
  forall orderedLists $ \xs -> is_ordered (insert x xs)
generates randomly a set of sorted lists
tested to satisfy: is_ordered (insert x xs)
```

Note:

- While the preceding definition of `prop_InsertOrdered x xs = is_ordered xs ==> ...` quantifies over all lists, the above property definition **quantifies** explicitly over the subset of ordered lists (cf. [Chapter 5.5](#)).
- Quantifying over subsets of values of a domain avoids test data generation in a trial-and-error fashion. Only ‘meaningful’ test data are generated.

A Quick Reminder to the Operator (\$)

...being defined in the [Standard Prelude](#) of Haskell:

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

The (\$) operator is Haskell's [infix function application](#), and useful for saving parentheses:

```
 f $ g x = f (g x)
```

Looking ahead

...QuickCheck supports also the specification of much more sophisticated properties, e.g.:

- ▶ *The list resulting from insertion coincides with the argument list (except of the inserted element).*

as well as testing of

- ▶ more than one property at the same time.

This is achieved by running a (small) program (also called quickCheck) from the command line. E.g., the call:

```
– Main>quickCheck Module.hs
```

checks all properties defined in `Module.hs` at the same time.

Chapter 5.3

Testing against Abstract Models

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

Chap. 6

Part IV

Chap. 7

Chap. 8

427/196

Objective

Testing the **correctness** (or: **soundness**) of an

- implementation

against a

- reference implementation

of a so-called

- abstract model (or: reference model).

We demonstrate this considering an **extended example**:

- Testing **soundness** of an **efficient implementation** of **queues** against a less efficient **reference implementation** of an **abstract model** of queues.

The Abstract Model of Queues

...defined in terms of an:

(Executable) Specification:

```
type Queue a = [a]
emptyQ       = []
enQ x q      = q ++ [x]  -- Inefficient due to (++)!
is_emptyQ q  = null q   -- Cost of enQ proportional
frontQ (x:q) = x        -- to number of list elements.
deQ (x:q)    = q
```

...in the following, this executable specification of 'first-in-first-out (FIFO)' queues serves as the reference implementation for queues; an implementation, which is simple but inefficient.

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

Chap. 6

Part IV

Chap. 7

Chap. 8

429/196

Implementing Queues more Efficiently

...than by the reference implementation of the abstract model:

Key idea (due to F. Warren Burton, 1982):

- Split a queue into two portions (a queue front and a queue back).
- Store the back of the queue in reverse order.

This queue representation ensures:

- Efficient access to both queue front and queue back: $(++)$ is replaced by $(:)$ (so-called strength reduction).

Example:

- Queue representations: $[7, 2, 9, 4, 1, 6, 8, 3] \cong ([7, 2, 9, 4], [3, 8, 6, 1]), ([7, 2], [3, 8, 6, 1, 4, 9]), ([7, 2, 9, 4, 1], [3, 8, 6]), \dots$
- Abstract model enqueueing, $(++)$: $[7, 2, 9, 4, 1, 6, 8, 3] ++ [5]$
- Implementation enqueueing, $(:)$: $([7, 2, 9, 4], 5: [3, 8, 6, 1]), ([7, 2], 5: [3, 8, 6, 1, 4, 9]), ([7, 2, 9, 4, 1], 5: [3, 8, 6]), \dots$

Implementing the Abstract Model of Queues

Implementation:

```
type QueueI a      = ([a], [a])
emptyQI            = ([], [])
enQI x (f,b)      = (f,x:b)  -- (:) instead of (++)!
                    -- Therefore, more
                    -- efficient!

is_emptyQI (f,b)  = null f
frontQI (x:f,b)   = x
deQI (x:f,b)      = flipQI (f,b)
  where
    flipQI ([],b) = (reverse b, [])  -- 'back' be-
    flipQI q      = q                -- comes 'front'
                                       -- when 'front'
                                       -- gets empty.
```

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

Chap. 6

Part IV

Chap. 7

Chap. 8

431/196

Relating Implementation and Abstract Model

...by means of the function `retrieve`:

```
retrieve :: QueueI a -> Queue a
retrieve (f,b) = f ++ reverse b
```

Note, `retrieve` transforms each of the (usually many)

- 'concrete' representations of an 'abstract' queue into their unique canonical representation as an 'abstract' queue, i.e., it transforms values of `(QueueI a)` into their unique matching value of `(Queue a)`.

Example:

```
retrieve ([7,2,9,4], [5,3,8,6,1]) ->> [7,2,9,4,1,6,8,3,5]
retrieve ([7,2], [5,3,8,6,1,4,9]) ->> [7,2,9,4,1,6,8,3,5]
retrieve ([7,2,9,4,1], [5,3,8,6]) ->> [7,2,9,4,1,6,8,3,5]
...
```


Now

...we want to **test** whether operations defined on `(QueueI a)` behave in the same way as their specifying counterparts defined on `(Queue a)`.

For convenience, we focus on **queues** of **integer values** (i.e., `(QueueI Int)` and `(Queue Int)`). For this reason, we omit giving (the actually required) **type signatures** in **property definitions**.

Using `retrieve :: QueueI Int -> Queue Int` we can check, whether the results of applying

- the **efficient operations** on `(QueueI Int)` match the ones of their abstract counterparts on `(Queue Int)`.

Soundness Properties: Initial Definitions

Defining five soundness properties:

```
prop_emptyQ      = retrieve emptyQI == emptyQ
prop_enQ x q     = retrieve (enQI x q)
                  == enQ x (retrieve q)
prop_isemptyQ q = is_emptyQI q
                  == is_emptyQ (retrieve q)
prop_frontQ q   = frontQI q == frontQ (retrieve q)
prop_deQ q      = retrieve (deQI q)
                  == deQ (retrieve q)
```

...which can reasonably be expected to hold, if the implementation of queues over `(QueueI Int)` is correct wrt their abstract model over `(Queue Int)`.

However, this is not true! Three (out of five) properties can be falsified!

Falsifiability of `prop_isemptyQ`

Testing `prop_isemptyQ` using `QuickCheck`, e.g., yields:

```
Main>quickCheck prop_isemptyQ
Falsifiable, after 4 tests:
([], [-1])
```

Cause of failure: The definition of `is_emptyQI` implicitly assumes that the following `invariant` holds:

- (Silently assumed) `invariant`: The `front` of a list is only empty, if its `back` is empty, too:

$$\text{is_emptyQI } (f, b) \Rightarrow \text{null } b$$

since `is_emptyQI (f, b) = null f`, `emptyQI = ([], [])`.

This `invariant`, however, is not enforced by the implementation!

Falsifiability of frontQI and deQI

...the definitions of `frontQI` and `deQI` rely on the very same assumption as the one of `is_emptyQI` that the front of a queue is only empty, if its back is empty, too.

Thus, in addition to `prop_isemptyQ` the properties

- `prop_frontQ`
- `prop_deQ`

are **falsifiable**, too!

Remedy: The silently made assumption on the invariant, which we took care of when defining `deQI`, must be made explicit in the property definitions.

Soundness Properties: 1st Refinement (1)

We define the **invariant** as follows:

```
invariant :: QueueI Int -> Bool
invariant (f,b) = (not (null f)) || null b
```

...and adjust the **property definitions** accordingly:

```
prop_emptyQ      = retrieve emptyQI == emptyQ
```

```
-----
prop_enQ x q     = invariant q ==>
  retrieve (enQI x q) == enQ x (retrieve q)
```

```
prop_isemptyQ q = invariant q ==>
  is_emptyQI q == is_emptyQ (retrieve q)
```

```
prop_frontQ q   = invariant q ==>
  frontQI q == frontQ (retrieve q)
```

```
prop_deQ q      = invariant q ==>
  retrieve (deQI q) == deQ (retrieve q)
```

Soundness Properties: 1st Refinement (2)

Now, testing `prop_isemptyQ` using `QuickCheck` yields:

```
Main>quickCheck prop_isemptyQ
OK, passed 100 tests
```

However, testing `prop_frontQ` still fails:

```
Main>quickCheck prop_frontQ
Program error: front ([], [])
```

Cause of failure: `frontQI` (as well as `deQI`) may only be applied to non-empty lists.

...so far, we did not take care of this.

Soundness Properties: 2nd Refinement

...to fix this, add `not (is_emptyQI q)` to the precondition of the challenged properties.

This leads to:

```
prop_emptyQ      = retrieve emptyQI == emptyQ
prop_enQ x q     = invariant q ==>
    retrieve (enQI x q) == enQ x (retrieve q)
prop_isemptyQ q = invariant q ==>
    is_emptyQI q == is_emptyQ (retrieve q)
-----
prop_frontQ q = invariant q && not (is_emptyQI q) ==>
    frontQI q == frontQ (retrieve q)
prop_deQ q    = invariant q && not (is_emptyQI q) ==>
    retrieve (deQI q) == deQ (retrieve q)
```

Soundness Issues Reconsidered

After this 2nd refinement, **all five properties** pass now the `QuickCheck` test **successfully!**

However, we are not yet done. So far we only tested that

- ▶ operations **on queues** behave correctly on queues which satisfy the **invariant**:

```
invariant :: QueueI Int -> Bool
invariant (f,b) = (not (null f)) || null b
```

Additionally, we need to check that

- ▶ operations **producing a queue** do only produce queues which satisfy the **invariant**.

Additional Soundness Properties

...for operations producing queues:

```
prop_inv_emptyQ   = invariant emptyQI
prop_inv_enQ x q  = invariant q ==>
                    invariant (enQI x q)
prop_inv_deQ q    = invariant q &&
                    not (is_emptyQI q) ==>
                    invariant (deQI q)
```

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

Chap. 6

Part IV

Chap. 7

Chap. 8

441/196

Testing the Additional Soundness Properties

Testing the additional properties with `QuickCheck` yields:

```
Main>quickCheck prop_inv_enQ
Falsifiable, after 0 tests:
0
([], [])
```

Cause of failure: The implementation of `enQI` does not ensure the validity of the `invariant` when applied to the `empty list`:

- Adding to the back of the empty queue **breaks the invariant!**

This means:

- The implementation of `enQI` by `enQI x (f,b) = (f,x:b)` is faulty and needs to be fixed!

Fixing the Faulty Implementation of enQI

...by replacing the faulty implementation of enQI:

```
enQI x (f,b) = (f,x:b)
```

by the sound one:

```
enQI x (f,b) = flipQ (f,x:b)
```

where

```
flipQI ([],b) = (reverse b, [])
```

```
flipQI q      = q
```

Now, all 8 properties pass the QuickCheck test successfully!

Summary

...reconsidering the development of the example, **testing** revealed

- ▶ (only) **one bug** in the implementation (this was in function **enQI**; for **deQI**, we were keen to get handling empty back queues right from the very beginnings)
- ▶ **several missing preconditions** and **one missing invariant** in the initial property definitions.

This is **typical**, and both revealing flaws in implementations and property definitions is valuable:

- ▶ The initially missing preconditions and the invariant are now explicitly given in the program text as part of the property definitions.
- ▶ They add to understanding the program and are valuable as documentation, both for the program developer and for future users (think of program maintainance!).

Chapter 5.4

Testing against Algebraic Specifications

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

Chap. 6

Part IV

Chap. 7

Chap. 8

445/196

Objective

Testing the **correctness** (or: **soundness**) of an

- implementation

against

- equational constraints

the operations ought to satisfy, a so-called

- algebraic specification.

...testing against an **algebraic specification** is (often) a useful alternative to testing against an **abstract model**. In the following, we demonstrate this considering **queues** as an **example**.

Algebraic Specification of Queue Operations

...any proper definition of **queue operations** can be expected to satisfy the following **equational constraints**:

```
prop_isemptyQ q =
```

```
  invariant q ==> isEmptyQI q == (q == emptyQI)
```

```
prop_front_emptyQ x = frontQI (enQI x emptyQI) == x
```

```
prop_front_enQ x q =
```

```
  invariant q && not (isEmptyQI q) ==>
    frontQI (enQI x q) == frontQI q
```

```
prop_deQ_emptyQ x = deQI (enQI x emptyQI) == emptyQI
```

```
prop_deQ_enQ x q =
```

```
  invariant q && not (isEmptyQI q) ==>
    deQI (enQI x q) == enQI x (deQI q)
```

Compare these property definitions with the **behaviour specification** of the **abstract data type (ADT) queue** in **Chapter 8.3!**

Testing against the Algebraic Specification

...testing the equational constraint `prop_deQ_enQ` using `QuickCheck` yields:

```
Main>quickCheck prop_deQ_enQ
Falsifiable, after 1 tests:
0
([1], [0])
```

Cause of failure: Evaluating

- the left hand side expression yields:

```
deQI (enQI 0 ([1], [0])) ->> deQI ([1], [0,0])
->> flipQI ([], [0,0]) ->> ([0,0], [])
```

- the right hand side expression yields:

```
enQI 0 (deQI ([1], [0])) ->> enQI 0 (flipQI
([], [0])) ->> enQI 0 ([0], []) ->> ([0], [0])
```

- `([0,0], [])` and `([0], [0])` are **equivalent** (they represent the abstract queue `[0,0]`) but are **not exactly equal!**

Refining the Algebraic Specification

...by replacing testing for **equality** by testing for **equivalence**:

```
q 'equiv' q' = invariant q && invariant q' &&
              retrieve q == retrieve q'
```

Replacing the initial formulation of:

```
prop_deQ_enQ x q =
  invariant q && not (is_emptyQI q) ==>
    deQI (enQI x q) == enQI x (deQI q)
```

by the **new one**:

```
prop_deQ_enQ x q =
  invariant q && not (is_emptyQI q) ==>
    deQI (enQI x q) 'equiv' enQI x (deQI q)
```

the **QuickCheck** test of `prop_deQ_enQ` passes **successfully**!

Testing further Equational Constraints

Analogously to [Chapter 5.3](#), we also need to check that

- operations [producing a queue](#) do only produce queues which are [equivalent](#), if the arguments are.

To this end, we introduce additional [soundness properties](#) for the operations [enQI](#) and [deQI](#):

```
prop_enQ_equivQ q q' x =  
  q 'equiv' q' ==> enQI x q 'equiv' enQI x q'
```

```
prop_deQ_equivQ q q' =  
  q 'equiv' q' && not (null q) && not (null q') ==>  
    deQI q 'equiv' deQI q'
```

Note

...though `mathematically sound`, the usability of the property definitions `prop_enQ_equiv` and `prop_deQ_equiv` for testing with `QuickCheck` is limited.

Testing them with `QuickCheck`, we might observe, e.g.:

```
Main>quickCheck prop_enQ_equiv
Arguments exhausted after 58 tests.
```

...which is due to an `implementation feature` of `QuickCheck`:

- `QuickCheck` generates the two lists `q` and `q'` randomly.
- Most of the generated pairs of lists will thus `not be equivalent`, and hence be discarded as test cases.
- `QuickCheck` makes a maximum number of tries of generating test cases (default: 1.000); afterwards, it stops, possibly before the number of 100 test cases is reached.

Looking ahead

...[QuickCheck](#) provides features to cope with such problems of test case generation; providing especially support for

- ▶ [Quantifying over subsets](#) of value domains by means of
 - filters
 - generators ([type-based](#), [weighted](#), [size controlled](#),...)
- ▶ ...
- ▶ [Test case monitoring](#)

...which we are going to illustrate next, mostly driven by examples.

Chapter 5.5

Controlling Test Data Generation

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.5.1

5.5.2

5.5.3

5.6

5.7

5.8

5.9

Chap. 6

Part IV

453/196

Controlling Test Data Domains and Sizes (1)

...or: How to shape the 1) value domains test data are drawn from, and 2) the size of individual test data generated?

1) Value Domains of Test Data and Quantifying over Them

- By default, the parameters of QuickCheck properties are quantified over all values of the underlying data type (e.g., all integers, all lists of integers; **not**: all even integers, all sorted lists of integers, etc.).

As we have seen, however, it is often preferable or even necessary to only quantify over subsets of a value domain (e.g., all sorted lists of integers).

Controlling Test Data Domains and Sizes (2)

2) Size of Individual Test Data

- A set of test data drawn from a value domain should be a 'fair mix of smaller and larger values' avoiding the generation of extremely large values as well as of (too many) duplicates, in particular, of 'trivial' values (e.g., empty list, lists of length 1, empty trees, etc.).

Meeting the 'fairness' requirement is especially challenging for data domains whose values are recursively defined (e.g., trees, lists, etc.).

QuickCheck offers several means for controlling

- ▶ quantification over sets and subsets of sets of value domains (cf. [Chapter 5.5.1](#)).
- ▶ the size of generated values (cf. [Chapter 5.5.2](#)).

Chapter 5.5.1

Controlling Quantification over Value Domains

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.5.1

5.5.2

5.5.3

5.6

5.7

5.8

5.9

Chap. 6

Part IV

456/196

Controlling Quantification over Value Domains

Discussed so far (cf. [Chapter 5.3](#), [5.4](#)):

1. **Boolean functions**: Used as **preconditions** in property definitions act as **test case filters** selecting useful ones:
 - ▶ **Works well**, if most elements of the underlying value domain are members of the relevant subset, too.
 - ▶ **Works poorly**, if only a few elements of the underlying domain are members of the relevant subset.

Discussed next:

2. **Generators**: Used for **targeted generation of test data** of the subset of interest:
 - ▶ Generators of the **monadic type** (**Gen a**) generate random values of type **a**; conceptually, generators can be identified with the **set of values they can generate**.
 - ▶ Generators are used together with the property **forall set p**, which tests property **p** for all randomly generated elements of the set **set**.

Note

...**Boolean functions as test case filters** and **generators** differ in their strengths and limitations for particular tasks, e.g., representing relations of values like **equivalence of values**. Representing **value equivalence** by a

- ▶ **Boolean function** makes it **easy to check** whether two values are equivalent, **but difficult** to generate values which are equivalent.
- ▶ **Generator**, i.e., a function mapping a value to a set of related (e.g., equivalent) values, makes it **easy to generate equivalent** values, **but difficult to check** if two given values are equivalent.

...we now continue with the **generator** approach.

The 1-Ary Type Constructor `Gen`

...values generated by `QuickCheck` are of type `Gen a`.

The type constructor `Gen` is an instance of the type constructor class `Monad` (cf. [Chapter 13](#)), which eases the definition of concrete [data generators](#).

Consider e.g. the two [generator](#) expressions `return a` and `do {x <- s; e}` of type `Gen a`:

- ▶ `return a` can be thought of to represent the singleton set $\{a\}$, and to generate value `a`.
- ▶ `do {x <- s; e}` can be thought of to represent the set $\{e \mid x \in s\}$.

The Function `choose`

...for random element generation.

`choose` is the most basic function of `QuickCheck` supporting to make a choice:

```
choose :: Random a => (a,a) -> Gen a
```

Note:

- ▶ `Random` denotes a type class provided by the library module `Random` of Haskell; its operations support the generation of pseudo-random numbers.
- ▶ `choose` generates a 'random' element of domain `a` of the specified range.
- ▶ Conceptually, `choose (1,n)`, e.g., represents the set $\{1, \dots, n\}$, and randomly selects one element of it.

Defining Generators using choose

...illustrated by defining the generator `equivQ`, which, given a queue value `q`, generates a new queue value `q'` equivalent to `q`:

```
equivQ :: QueueI a -> Gen (QueueI a)
equivQ q =
  do k <- choose (0,0 'max' (n-1))
     return (take (n-k) els,reverse (drop (n-k) els))
  where els = retrieve q
        n   = length els
```

Note:

- ▶ Given a `(QueueI a)`-value `q`, `equivQ` generates randomly a queue `q'` with the same elements as `q`.
- ▶ The number `k` of elements in the back queue of `q'` is chosen properly smaller than the total number of elements of `q'` (supposed this total number is different from 0).

Property Definitions with Generators (1)

...using `equivQ`, we define `soundness` property:

```
prop_equivQ q = invariant q ==>
  forall (equivQ q) $ \q' -> q 'equiv' q'
```

...allowing to test, whether `equivQ` produces in fact queues, which are `equivalent` to the argument it is applied to.

Note:

- ▶ (\$) means function application allowing the omission of parentheses (see the anonymous λ -expression in the definition of `prop_equivQ`).
- ▶ The property dual to `prop_equivQ`, whether all queues equivalent to some queue can be generated by `equivQ`, cannot in general be established by testing.

Property Definitions with Generators (2)

...using `equivQ`, we can define counterparts of the properties `prop_enQ_equivQ` and `prop_deQ_equivQ` allowing to test, whether `enQ` and `deQ` map equivalent queues to equivalent queues:

```
prop_enQ_equivQ q x = invariant q ==>
  forall (equivQ q) $ \q' -> enQI x q 'equiv' enQI x q'
prop_deQ_equivQ q = invariant q && not (null q) ==>
  forall (equivQ q) $ \q' -> deQI q 'equiv' deQI q'
```

For comparison, we recall the initial definitions (cf. Chapter 5.4):

```
prop_enQ_equivQ q q' x =
  q 'equiv' q' ==> enQI x q 'equiv' enQI x q'
prop_deQ_equivQ q q' =
  q 'equiv' q' && not (null q) && not (null q') ==>
  deQI q 'equiv' deQI q'
```

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.5.1

5.5.2

5.5.3

5.6

5.7

5.8

5.9

Chap. 6

Part IV

463/196

Type-based Generation of Value Sets

...is enabled by the overloaded `generator arbitrary`, e.g., for generating the argument values of properties:

Example: Generating (and testing) over unrestricted sets of numerical values:

```
prop_max_le =  
  forAll arbitrary $ \x ->  
    forAll arbitrary $ \y -> x <= x 'max' y
```

This definition is **equivalent** to the **short-hand** form:

```
prop_max_le x y = x <= x 'max' y
```


Type-based Generation of Subsets of Value Sets

...can be achieved by `arbitrary` followed by a suitable value modification:

Example: The generator `atLeast` defined on top of `arbitrary` generates the set of numerical values $\{y \mid y \geq x\}$:

```
atLeast x = do diff <- arbitrary
           return (x + abs diff)
```

Note, the definition of `atLeast` makes use of the equality of the sets:

$$\{y \mid y \geq x\} = \{x + \text{abs } d \mid d \in \mathbb{Z}\}$$

which is valid for numerical values (note, the idea underlying the definition of `atLeast` can be adapted to types other than numerical ones).

Selecting a Generator

...is enabled by the `generator oneof` which can conceptually be thought of as `set union` operator.

Example: The generator `orderedLists` (cf. [Chapter 5.2](#)) for generating sorted lists is based on the idea that a sorted list is either 1) empty or 2) the result of attaching a new head element to a sorted list of larger elements:

```
orderedLists = do x <- arbitrary
                listsFrom x

where
  listsFrom x
    = oneof [return [],           -- either: empty
            do y <- atLeast x    -- or: a list of elems > x
              liftM (x:) (listsFrom y)] -- extended
                                           -- by x as new head element
```

Note

...the `oneof` generator picks alternatives with

- ▶ equal probability.

This can impact the generation of test data unduly. E.g., the generator `orderedLists` will produce

- ▶ the empty list far too often

questioning its usability as an adequate test data generator for ordered lists.

`QuickCheck` offers thus means for a `weighted selection` of generators.

Weighted Selection of a Generator

...is enabled by the `generator frequency`, which allows assigning `weights` to a set of selectable generators controlling their relative likelihood of being actually selected:

```
frequency :: [(Int, Gen a)] -> Gen a
```

Example:

```
listsFrom x
  = frequency [(1, return []),
              (4, do y <- atLeast x
                    liftM (x:) (listsFrom y))]
```

Note:

- ▶ `QuickCheck` generators correspond actually to a probability distribution over a set, rather than just the set itself.
- ▶ The assignment of weights above gives the `cons case` a `weight` of `4`; generated lists will thus have an average length of `4` elements.

Pragmatics: Generators as Default Generators

...if a `generator` like `orderedLists` is used frequently, this generator should be made the `default generator` for values of the generated type. To this end, define a `new type` for the value type generated and make this new type an instance of the type class `Arbitrary` as shown below:

```
newtype OrderedList a = OL [a]
instance (Num a, Arbitrary a) =>
    Arbitrary (OrderedList a) where
    arbitrary = liftM OL orderedLists
```

Example: Redefining `insert` with the new type `OrderedList`

```
insert :: Ord a => a -> OrderedList a
      -> OrderedList a
```

ensures that arguments generated for `insert` will automatically be ordered.

Chapter 5.5.2

Controlling the Size of Test Data

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.5.1

5.5.2

5.5.3

5.6

5.7

5.8

5.9

Chap. 6

Part IV

470/196

Controlling the Size of Test Data

...is usually **necessary** in order to **avoid** the generation of **unreasonably large** test cases; **QuickCheck** provides support for this.

QuickCheck generators are parameterized on an

- ▶ integer valued parameter **size**, which is gradually increased during testing (first tests explore small cases, later tests larger and larger ones).

The **interpretation** of the **size** parameter is up to the

- ▶ implementor of a test case generator (the default generator for lists, e.g., interpretes **size** as an upper bound on the length of lists).

Generators depending on **size** are defined using function:

```
sized :: (Int -> Gen a) -> Gen a
```

Example

...the default generator `vector` for list values:

```
vector n = sequence [arbitrary | i <- [1..n]]
```

...calling `vector` with argument `length` generates lists of random values of length `length`.

`vector` in concert with function `sized`:

```
sized $ \n -> do length <- choose (0,n)
                 vector length
```


The Function `resize`

...allows to supply an explicit `size argument` to a generator:

```
resize :: Int -> Gen a -> Gen a
```

Example: Generating a list of lists while bounding the total number of elements by the size parameter:

```
sized $ \n -> resize (round (sqrt (fromInt n))) arbitrary
```

Note: The definition uses the default generator but replaces the size parameter by its `square root`. The list of lists is generated by the default generator `arbitrary` but with a smaller size parameter.

Chapter 5.5.3

Example: Test Data Generators at Work

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.5.1

5.5.2

5.5.3

5.6

5.7

5.8

5.9

Chap. 6

Part IV

Generators for Built-in and User-defined Types

Note, `test data generators` for

- ▶ predefined ('built-in') types of Haskell
 - are provided by `QuickCheck`.
- ▶ user-defined types
 - must be provided by the user in terms of defining suitable instances of the type class `Arbitrary`.
 - require usually measures to control the size of generated test data, especially for values of inductively defined types.

This is illustrated next considering `binary trees` as example:

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

A User-defined Generator for Binary Trees

...we make the type `(Tree a)` straightforwardly an instance of the type class `Arbitrary`:

```
instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary =
    frequency [(1,return Leaf),
              (3,liftM3 Branch
                arbitrary arbitrary arbitrary)]
```

Note: Assigning the weights (1 resp. 3) to the two subgenerators shall ensure that not too many trivial trees of size 1 are generated.

Analyzing the Arbitrary Instance (Tree a)

Fact:

- ▶ The likelihood that a **finite** tree is generated, is only **one third** because termination is only possible, if all subtrees which are generated are finite.

Problem:

- ▶ With increasing breadth of the tree under generation, the requirement of selecting the 'terminating' branch must be satisfied simultaneously at ever more places pushing the likelihood for this towards 0.

Remedy: Using the **size** parameter in order to ensure

- ▶ **termination**.
- ▶ **generation** of 'reasonably' sized trees.

The Refined Generator for Binary Trees

...replace the initial `instance`-declaration for `(Tree a)` by:

```
instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary = sized arbTree
```

```
arbTree 0 = return Leaf
```

```
arbTree n | n>0 =
```

```
  frequency [(1,return Leaf),
             (3,liftM3 Branch shrub arbitrary shrub)]
```

```
  where shrub = arbTree (n `div` 2)
```

Note:

- ▶ `shrub` is a `generator` for 'small(er)' trees. It is not bound to a special tree; the two occurrences of `shrub` will usually generate different trees.
- ▶ Since the size limit for subtrees is `halved`, their total size is bounded by the argument value of `arbTree`.
- ▶ Generators for values of recursive types must usually be handled like in this example.

A Note on Lift Functions

...lift functions used throughout [Chapter 5.5](#) are provided by the library module `Monad` (cf. [Chapter 13](#)):

```
liftM    :: Monad m => (a -> b) -> (m a -> m b)
liftM2   :: Monad m => (a -> b -> c) -> (m a -> m b -> m c)
liftM3   :: Monad m => (a -> b -> c -> d) ->
              (m a -> m b -> m c -> m d)
liftM4   :: Monad m => (a -> b -> c -> d -> e) ->
              (m a -> m b -> m c -> m d -> m e)
liftM5   :: Monad m => (a -> b -> c -> d -> e -> f) ->
              (m a -> m b -> m c -> m d -> m e -> m f)
```

Chapter 5.6

Monitoring, Reporting, and Coverage

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

Chap. 6

Part IV

Chap. 7

Chap. 8

480/196

Why is Test-Data Monitoring Useful?

...reconsider the example of [inserting](#) into a [sorted list](#):

```
prop_InsertOrdered :: Int -> [Int] -> Property
prop_InsertOrdered x xs
  = is_ordered xs ==> is_ordered (insert x xs)
```

[QuickCheck](#) checks [prop_InsertOrdered](#) by

- ▶ [randomly](#) generating lists

and checking each of them being sorted (used as a test case) or not (discarded).

Analyzing Potential Risks

Fact:

- ▶ The likelihood that a [randomly generated list](#) is sorted decreases with its length.

Conversely: The likelihood of being sorted is the higher the shorter the list is.

Risk:

- ▶ Property [prop_InsertOrdered](#) is likely to be mostly tested with lists of length one or two.
- ▶ Even [QuickCheck](#) runs run to completion are not meaningful.

Test Data Monitoring and Reporting

...can thus provide useful hints on the

▶ **quality** and **coverage** of the test cases
of a **QuickCheck** run.

QuickCheck provides a variety of

▶ **monitoring** and **reporting possibilities**
for this purpose.

Instrumental are the **QuickCheck** combinators:

1. `trivial`
2. `classify`
3. `collect`

The QuickCheck Combinator `trivial`

...allows monitoring and reporting the percentage of test cases which are considered trivial, where the meaning of

- ▶ `'trivial'` is user-definable, e.g., `lists up to a length of 2`.

Example:

```
prop_InsertOrdered :: Int -> [Int] -> Property
prop_InsertOrdered x xs = is_ordered xs ==>
  trivial (length xs <= 2) $ is_ordered (insert x xs)
```

Double-checking the property with Hugs might yield:

```
Main>quickCheck prop_InsertOrdered
OK, passed 100 tests (91% trivial).
```

Analyzing the QuickCheck Run

...reveals:

- ▶ 91% of the test cases were trivial checking lists of length 2 or shorter.
- ▶ These are far too many in order to ensure that the test run is meaningful.
- ▶ This shows again that the operator `==>` must be used with care in test case generators.

Remedy:

- ▶ Replacing the default means of test case generation by a user-defined generator, e.g., by proper quantification as sketched in Chapter 5.2.

Note:

- ▶ The combinator `trivial` is defined in terms of the more general combinator `classify`:
`trivial p = classify p "trivial"`

The QuickCheck Combinator `classify`

...supports a more refined test-case monitoring and reporting than `trivial` by allowing to define sets of interesting test case classes:

Example:

```
prop_InsertOrdered x xs = is_ordered xs ==>
  classify (null xs) "empty lists" $
  classify (length xs == 1) "unit lists" $
  is_ordered (insert x xs)
```

Double-checking this property might yield:

```
Main>quickCheck prop_InsertOrdered
OK, passed 100 tests.
42% unit lists.
40% empty lists.
```

The QuickCheck Combinator `collect`

...goes beyond the monitoring and reporting capabilities of even `classify` by delivering **histograms of test case values**.

Example:

```
prop_InsertOrdered x xs = is_ordered xs ==>
  collect (length xs) $ is_ordered (insert x xs)
```

Double-checking this property might yield:

```
Main>quickCheck prop_InsertOrdered
OK, passed 100 tests.
46% 0.
34% 1.
15% 2.
5% 3.
```

Chapter 5.7

Implementation of QuickCheck

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

Chap. 6

Part IV

Chap. 7

Chap. 8

488/196

QuickCheck: Facts and Figures

QuickCheck

- consists in total of about 300 lines of code.
- has been developed by Koen Claessen and John Hughes.
- was initially presented in:
 - Koen Claessen, John Hughes. [QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs](#). In Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP 2000), 268-279, 2000.
- This chapter is mostly based on:
 - Koen Claessen, John Hughes. [Specification-based Testing with QuickCheck](#). In Jeremy Gibbons, Oege de Moor (Eds.), [The Fun of Programming](#). Palgrave MacMillan, 17-39, 2003.

A Glimpse of the QuickCheck Code

```
newtype Property = Prop (Gen Result)

class Testable a where
  property :: a -> Property

instance Testable Bool where
  property b = Prop (return (resultBool b))

instance Testable Property where
  property p = p

instance (Arbitrary a, Show a, Testable b) =>
         Testable (a -> b) where
  property f = forAll arbitrary f

quickCheck :: Testable a => a -> IO ()
```

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

Chap. 6

Part IV

Chap. 7

Chap. 8

490/196

For further Details

...including [applications](#), refer to e.g.:

- Koen Claessen, John Hughes. [QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs](#). In Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP 2000), 268-279, 2000.
- Koen Claessen, John Hughes. [Testing Monadic Code with QuickCheck](#). In Proceedings of the ACM SIGPLAN 2002 Haskell Workshop (Haskell 2002), 65-77, 2002.

Chapter 5.8

Summary

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

Chap. 6

Part IV

Chap. 7

Chap. 8

On the Relevance and Value

...of [specifications](#), [testing](#), and tools like [QuickCheck](#).

[Specifications](#): Experience shows

- ▶ Formalizing specifications is meaningful (even if they are not used for a formal proof of soundness).
- ▶ Specifications provided are (initially) often faulty themselves.

[Testing](#): Investigations of [Richard Hamlet](#) reported in

- Richard Hamlet. [Random Testing](#). Encyclopedia of Software Engineering, Wiley, 970-978, 1994.

indicate that

- ▶ results from a high number of test cases are meaningful even if [test cases are randomly generated](#).
- ▶ random test case generation is often '[cheap](#).'

On the Value of Tools like QuickCheck

Together, the [findings](#) on [specifications](#) and [testing](#) provide good reasons for using tools like [QuickCheck](#) on a

- ▶ [routine](#) basis.

Experience actually shows that [QuickCheck](#) is effective for

- ▶ disclosing bugs in [programs](#) and [specifications](#) with little effort.
- ▶ reducing [test costs](#) while at the same time testing [more thoroughly](#).

Note that there is a range of other [combinator libraries](#) supporting the lightweight testing of Haskell programs, e.g.:

- [EasyCheck](#)
- [SmallCheck](#)
- [Lazy SmallCheck](#)
- [Hat](#) (for tracing Haskell programs)

In Closing: Another Independent Confirmation

...of the relevance of **testing**:

...the success of tests is that they test
the programmer, not the program.

Rigorous testing regimes rapidly persuade
error-prone programmers (like me) to remove
themselves from the profession.

[...]

...programmers who have survived the rigors of
testing are what make programs of the present day
useful, efficient, and (nearly) correct.

C. Antony Hoare (* 1934)

Recipient of the 1980 ACM A.M. Turing Award:
For his fundamental contributions to the definition and
design of programming languages.

Background: An Influential Work

...of [Tony Hoare](#), advocating [rigor](#) and [correctness](#) from the very beginnings in software development:

- Charles A.R. Hoare. [An Axiomatic Basis for Computer Programming](#). Communications of the ACM 12(10): 576-580, 1969.

and a retrospective written 40 years later:

- Charles A.R. Hoare. [Retrospective: An Axiomatic Basis for Computer Programming](#). Communications of the ACM 52(10):30-32, 2009.

Ext. Quote from Hoare's Retrospective Article

“One thing I got spectacularly wrong. I could see that programs were getting larger, and I thought that testing would be an increasingly ineffective way of removing errors from them. I did not realize that **the success of tests is that they test the programmer, not the program. Rigorous testing regimes rapidly persuade error-prone programmers (like me) to remove themselves from the profession.** Failure in test immediately punishes any lapse in programming concentration, and (just as important) the failure count enables implementers to resist management pressure for premature delivery of unreliable code [...]. The experience, judgment, and intuition of **programmers who have survived the rigors of testing are what make programs of the present day useful, efficient, and (nearly) correct.** Formal methods for achieving correctness must support the intuitive judgment of programmers, not replace it. My basic mistake was to set up **proof in opposition to testing**, where in fact **both of them are valuable and mutually supportive ways of accumulating evidence of the correctness and serviceability of programs.**”

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

Chap. 6

Part IV

Chap. 7

Chap. 8

497/196

Chapter 5.9

References, Further Reading

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

Chap. 6





Part IV

Chap. 7




Chap. 8

498/196





Chapter 5: Further Reading (1)

-  Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 18.2, QuickCheck)
-  F. Warren Burton. *An Efficient Implementation of FIFO Queues*. Information Processing Letters 14(5):205-206, 1982.
-  Koen Claessen, John Hughes. *QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs*. In Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP 2000), 268-279, 2000.
-  Koen Claessen, John Hughes. *Testing Monadic Code with QuickCheck*. In Proceedings of the ACM SIGPLAN 2002 Haskell Workshop (Haskell 2002), 65-77, 2002.



Chapter 5: Further Reading (2)

-  Koen Claessen, John Hughes. *Specification-based Testing with QuickCheck*. In Jeremy Gibbons, Oege de Moor (Eds.), *The Fun of Programming*. Palgrave MacMillan, 17-39, 2003.
-  Koen Claessen, Colin Runciman, Olaf Chitil, John Hughes, Malcolm Wallace. *Testing and Tracing Lazy Functional Programs Using QuickCheck and Hat*. In Johan Jeuring, Simon Peyton Jones (Eds.) *Advanced Functional Programming – Revised Lectures*. Springer-V., LNCS Tutorial 2638, 59-99, 2003.
-  Jan Christiansen, Sebastian Fischer. *Easycheck – Test Data for Free*. In Proceedings of the 9th International Symposium on Functional and Logic Programming (SFLP 2008), Springer-V., LNCS 4989, 322-336, 2008.

Chapter 5: Further Reading (3)

-  Richard Hamlet. *Random Testing*. In J. Marciniak (Ed.), *Encyclopedia of Software Engineering*, Wiley, 970-978, 1994.
-  Charles A.R. Hoare. *An Axiomatic Basis for Computer Programming*. *Communications of the ACM* 12(10):576-580, 1969.
-  Charles A.R. Hoare. *Retrospective: An Axiomatic Basis for Computer Programming*. *Communications of the ACM* 52(10):30-32, 2009.
-  Colin Runciman, Matthew Naylor, Fredrik Lindblad. *Small-Check and Lazy SmallCheck*. In *Proceedings of the ACM SIGPLAN 2008 Haskell Workshop (Haskell 2008)*, 37-48, 2008. (Available from <http://hackage.haskell.org>)

Chapter 5: Further Reading (4)

-  Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Chapter 11, Testing and Quality Assurance; Chapter 26, Advanced Library Design: Building a Bloom Filter – Testing with QuickCheck)
-  Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley/Pearson, 3rd edition, 2011. (Chapter 19.6, DSLs for computation: generating data in QuickCheck)

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8

5.9

Chap. 6

Part IV

Chap. 7

Chap. 8

502/196

This software comes “without warranty of any kind, expressed or implied, including but not limited to, the implied warranties of merchantability and fitness for a particular purpose.”

Chapter 6

Verification

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

6.1

6.2

6.3

6.4

6.5

6.6

6.7

6.8

Part IV

Chap. 7

Chap. 8

Chap. 9

...[both] proof [and] testing [...] are valuable and mutually supportive ways of accumulating evidence of the correctness and serviceability of programs.

C. Antony Hoare (* 1934)

Recipient of the 1980 ACM A.M. Turing Award:

For his fundamental contributions to the definition and design of programming languages.

...while coinciding in their overall goal, testing and verification (proof!) are of different rigor.

Testing, even if it can be amazingly effective, is limited to

- ▶ showing the presence of errors; it can not show their absence (except of the most simple scenarios).

while verification can

- ▶ prove the absence of errors!

Important Proof Techniques

...for proving properties of **functional programs** so far:

- ▶ **Equational reasoning** (cf. **Chapter 4**)

In this chapter, we complement equational reasoning with proof techniques based on important **inductive proof principles** (not limited to functional programs) which may operate on:

- ▶ **Unstructured data**
 - integers
 - chars
 - Booleans
 - ...
- ▶ **Structured data**
 - **lists** (**finite** by definition)
 - **streams** (**infinite** by definition)
 - **trees** (**finite** or **infinite**)
 - ...

Outline of Inductive Proof Principles

...we will consider:

- ▶ Inductive proof principles on natural numbers
 - Natural (or: mathematical) induction (dtsch. **vollständige Induktion**)
 - Strong induction (dtsch. **verallgemeinerte Induktion**)
- ▶ Inductive proof principles on structured data
 - Structural induction (dtsch. **strukturelle Induktion**)

In particular:

 - Structural induction on lists
 - Structural induction on stream approximants
- ▶ Coinduction
- ▶ Fixed point induction

Ohne Mathematik tappt man doch immer im Dunkeln.

Werner von Siemens (1816-1892)
dt. Erfinder und Unternehmer

Chapter 6.1

Inductive Proof Principles on Natural Numbers

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

6.1

6.1.1

6.1.2

6.1.3

6.2

6.3

6.4

6.5

6.6

6.7

6.8

Part IV

Chap. 7

507/196

Chapter 6.1.1

Natural Induction

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

6.1

6.1.1

6.1.2

6.1.3

6.2

6.3

6.4

6.5

6.6

6.7

6.8

Part IV

Chap. 7

The Principle of Natural Induction

Let \mathbb{IN} be the set of natural numbers, and P be a **property** of natural numbers.

The Principle of Natural (or: **Mathematical**) Induction

$$\underbrace{P(1)}_{\substack{\text{Base} \\ \text{Case}}} \wedge \overbrace{[\forall n \in \mathbb{IN}. \underbrace{P(n)}_{\substack{\text{Induction} \\ \text{Hypothesis}}} \Rightarrow \underbrace{P(n+1)}_{\substack{\text{Induction} \\ \text{Step}}}] }^{\text{Inductive Case}} \Rightarrow \underbrace{\forall n \in \mathbb{IN}. P(n)}_{\text{Conclusion}}$$

(dtsch. **Prinzip der vollständigen Induktion**)

Example: Illustrating Natural Induction

Lemma 6.1.1.1

$$\forall n \in \mathbb{N}. \sum_{k=1}^n (2k - 1) = n^2$$

Proof (by means of natural (mathematical) induction).

Proof of Lemma 6.1.1.1 (1)

Base case: Let $n = 1$. In this case we obtain the equality of the left and right hand side expression straightforwardly by equational reasoning:

$$\begin{aligned}\sum_{k=1}^n (2k - 1) &= \sum_{k=1}^1 (2k - 1) \\ &= 2 * 1 - 1 \\ &= 2 - 1 \\ &= 1 \\ &= 1^2 \\ &= n^2\end{aligned}$$

Proof of Lemma 6.1.1.1 (2)

Inductive case: Let $n \in \mathbb{N}$. By means of the **induction hypothesis (IH)** we can assume $\sum_{k=1}^n (2k - 1) = n^2$. This allows us to complete the proof by equational reasoning:

$$\begin{aligned} \sum_{k=1}^{n+1} (2k - 1) &= 2(n + 1) - 1 + \sum_{k=1}^n (2k - 1) \\ \text{(IH)} &= 2(n + 1) - 1 + n^2 \\ &= 2n + 2 - 1 + n^2 \\ &= 2n + 1 + n^2 \\ &= n^2 + 2n + 1 \\ &= n^2 + n + n + 1 \\ &= (n + 1)(n + 1) \\ &= (n + 1)^2 \end{aligned}$$



Exercise 6.1.1.2

Prove by means of **natural (mathematical) induction**:

Lemma 6.1.1.3

1.

$$\forall n \in \mathbb{N}. \sum_{k=1}^n k = \frac{n(n+1)}{2}$$

2.

$$\forall n \in \mathbb{N}. \sum_{k=1}^n k^2 = \frac{n(n+1)(2n+1)}{6}$$

3.

$$\forall n \in \mathbb{N}. \sum_{k=1}^n k^3 = \left(\frac{n(n+1)}{2} \right)^2$$

Chapter 6.1.2

Strong Induction

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

6.1

6.1.1

6.1.2

6.1.3

6.2

6.3

6.4

6.5

6.6

6.7

6.8

Part IV

Chap. 7

The Principle of Strong Induction

Let \mathbb{IN} be the set of natural numbers, and P be a **property** of natural numbers.

The Principle of Strong Induction

(Inductive) Case

$$\forall n \in \mathbb{IN}. \left[\underbrace{(\forall m < n. P(m))}_{\substack{\text{Induction} \\ \text{Hypothesis}}} \Rightarrow \underbrace{P(n)}_{\substack{\text{Induction} \\ \text{Step}}} \right] \Rightarrow \underbrace{\forall n \in \mathbb{IN}. P(n)}_{\text{Conclusion}}$$

(dtsch. Prinzip der verallgemeinerten Induktion)

Note: For the smallest natural number \hat{n} (\mathbb{IN}_0 vs. \mathbb{IN}_1), the induction hypothesis boils down to 'true', i.e., $P(\hat{n})$ has to be proven without relying on anything special.

Example: Illustrating Strong Induction

The Fibonacci function $fib : \mathbb{N}_0 \rightarrow \mathbb{N}_0$ is defined by:

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

Lemma 6.1.2.1

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

Proof (by means of strong induction).

Key for Proving Lemma 6.1.2.1 for $n \geq 2$

...is to assume for $m = n - 1$ and $m = n - 2$ the equality:

$$fib(m) = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^m - \left(\frac{1-\sqrt{5}}{2}\right)^m}{\sqrt{5}}$$

according to the **induction hypothesis (IH)**.

(**Note:** For $n \geq 2$, the induction hypothesis would allow us to use this equality even for all $m < n$ (not just for $m = n - 1$ and $m = n - 2$). Unlike for **Lemma 6.1.2.4**, however, this is not required to complete the proof of **Lemma 6.1.2.1**.)

Proof of Lemma 6.1.2.1 (1)

Case 1: Let $n = 0$. Equational reasoning yields straightforwardly the desired equality:

$$\text{fib}(0) = 0 = \frac{0}{\sqrt{5}} = \frac{1 - 1}{\sqrt{5}} = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^0 - \left(\frac{1-\sqrt{5}}{2}\right)^0}{\sqrt{5}}$$

(**Note:** For proving Case 1, the induction hypothesis allows nothing to assume on the validity of the statement. Fortunately, nothing is required.)

Case 2: Let $n = 1$. Again, equational reasoning yields directly the desired equality:

$$\text{fib}(1) = 1 = \frac{\sqrt{5}}{\sqrt{5}} = \frac{\frac{1}{2} + \frac{\sqrt{5}}{2} - \left(\frac{1}{2} - \frac{\sqrt{5}}{2}\right)}{\sqrt{5}} = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^1 - \left(\frac{1-\sqrt{5}}{2}\right)^1}{\sqrt{5}}$$

(**Note:** For proving Case 2, we could have used the statement for $n = 0$ by means of the induction hypothesis. This, however, is not required.)

Proof of Lemma 6.1.2.1 (2)

Case 3: Let $n \in \mathbb{N}_0$, $n \geq 2$. Using **IH** for $n-2$, $n-1$ we obtain as desired:

$$\begin{aligned} fib(n) &= fib(n-2) + fib(n-1) \\ (2 \times \text{IH}) &= \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{n-2} - \left(\frac{1-\sqrt{5}}{2}\right)^{n-2}}{\sqrt{5}} + \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{n-1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n-1}}{\sqrt{5}} \\ &= \frac{\left[\left(\frac{1+\sqrt{5}}{2}\right)^{n-2} + \left(\frac{1+\sqrt{5}}{2}\right)^{n-1}\right] - \left[\left(\frac{1-\sqrt{5}}{2}\right)^{n-2} + \left(\frac{1-\sqrt{5}}{2}\right)^{n-1}\right]}{\sqrt{5}} \\ &= \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{n-2} \left[1 + \frac{1+\sqrt{5}}{2}\right] - \left(\frac{1-\sqrt{5}}{2}\right)^{n-2} \left[1 + \frac{1-\sqrt{5}}{2}\right]}{\sqrt{5}} \\ (*) &= \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{n-2} \left(\frac{1+\sqrt{5}}{2}\right)^2 - \left(\frac{1-\sqrt{5}}{2}\right)^{n-2} \left(\frac{1-\sqrt{5}}{2}\right)^2}{\sqrt{5}} \\ &= \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}} \end{aligned}$$

□

Proof of (*)

Equality (*) follows from equalities (1) and (2):

$$\left(\frac{1 + \sqrt{5}}{2}\right)^2 = 1 + \frac{1 + \sqrt{5}}{2} \quad (1)$$

$$\left(\frac{1 - \sqrt{5}}{2}\right)^2 = 1 + \frac{1 - \sqrt{5}}{2} \quad (2)$$

...which can be proved by **equational reasoning** and the **binomial formulae (BF)**:

$$\left(\frac{1 + \sqrt{5}}{2}\right)^2 \stackrel{(BF)}{=} \frac{1 + 2\sqrt{5} + 5}{4} = \frac{6 + 2\sqrt{5}}{4} = \frac{3 + \sqrt{5}}{2} = 1 + \frac{1 + \sqrt{5}}{2}$$

$$\left(\frac{1 - \sqrt{5}}{2}\right)^2 \stackrel{(BF)}{=} \frac{1 - 2\sqrt{5} + 5}{4} = \frac{6 - 2\sqrt{5}}{4} = \frac{3 - \sqrt{5}}{2} = 1 + \frac{1 - \sqrt{5}}{2}$$

Exercise 6.1.2.2

Let function $f : \mathbb{IN}_0 \rightarrow \mathbb{IN}_0$ be defined by:

$$\forall n \in \mathbb{IN}_0. f(n) =_{df} \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \sum_{k=0}^{n-1} f(k) & \text{if } n \geq 2 \end{cases}$$

Prove by means of **natural (mathematical) induction**:

Lemma 6.1.2.3

$$(\forall n \in \mathbb{IN}. n \geq 3). \sum_{k=0}^{n-3} 2^k = 2^{n-2} - 1$$

Prove by means of **strong induction** (and **Lemma 6.1.2.3**):

Lemma 6.1.2.4

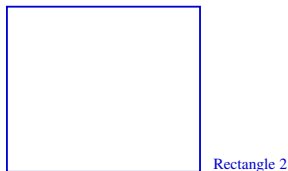
$$(\forall n \in \mathbb{IN}. n \geq 2). f(n) = 2^{n-2}$$

Chapter 6.1.3

Excursus: Fibonacci and the Golden Ratio Or: How Intuitive is Lemma 6.1.2.1?

The Golden Ratio

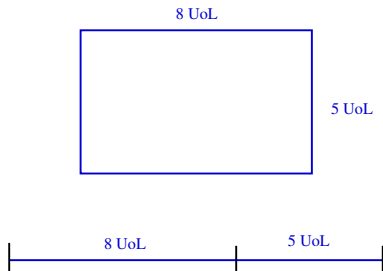
...which rectangle is the 'most' typical, the 'nicest' rectangle?



....most people say 'Rectangle 3!'

Why?

...UoL $\hat{=}$ Unit of Length



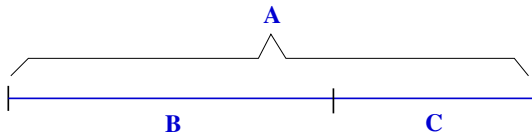
$$8 \text{ UoL} / 5 \text{ UoL} = 1.6$$

The Ratio of 1.6

...matches almost the so-called Golden Ratio:

$$\phi =_{df} \frac{1 + \sqrt{5}}{2} = 1.61803398874989\dots$$

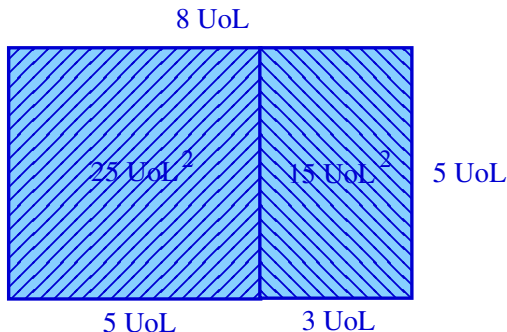
Geometrical Interpretation: Line segments B and C split line segment A in the Golden Ratio, denoted by ϕ if A is to B as B is to C , i.e., $A/B = B/C$:



$$A/B = B/C \Rightarrow \phi =_{df} A/B = \frac{1 + \sqrt{5}}{2} = 1.61803398874989\dots$$

Most People

...perceive the **Golden Ratio** as most harmonious:



$$8 \text{ UoL} / 5 \text{ UoL} = 1.6 \approx \phi = 1.61803\dots$$

$$5 \text{ UoL} / 3 \text{ UoL} = 1.\bar{6} \approx \phi = 1.61803\dots$$

$$25 \text{ UoL}^2 / 15 \text{ UoL}^2 = 1.\bar{6} \approx \phi = 1.61803\dots$$

Computing the Value of ϕ



$$\frac{x}{1} = \frac{x+1}{x} \Rightarrow \phi = x \left(= \frac{x}{1} \right)$$

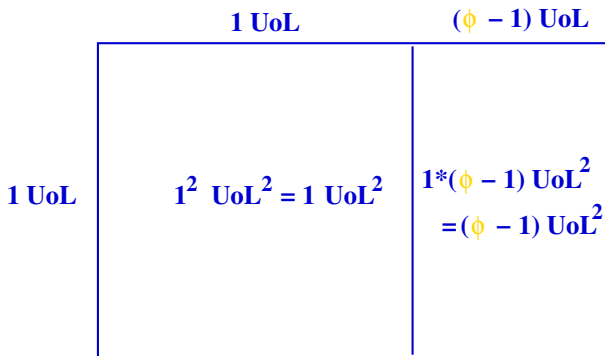
Replacing x by ϕ in the left-hand side equality of ' \Rightarrow ', we get:

$$\begin{aligned} \frac{\phi}{1} &= \frac{\phi+1}{\phi} \\ \Leftrightarrow \phi &= 1 + \frac{1}{\phi} \\ \Leftrightarrow \phi^2 &= \phi + 1 \\ \Leftrightarrow \phi^2 - \phi - 1 &= 0 \\ \Leftrightarrow \phi &= \frac{1+\sqrt{5}}{2} = 1.618\dots \quad (\text{or : } \phi = \frac{1-\sqrt{5}}{2} = -0.618\dots) \end{aligned}$$

Note: The negative ϕ -value lacks a geometric interpretation.

As Suggested Already

...the **Golden Ratio** does not only show up as the **ratio of line segments** but also as the **ratio of areas**, below, **rectangles**:

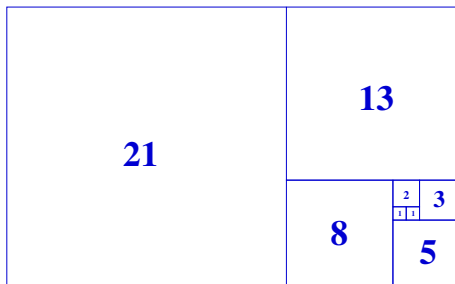


This Way

...also the sequence of **Fibonacci Numbers**:

(0,) 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, ...

is directly linked to the **Golden Ratio**:



Even more, the Sequence of the Ratios

...of successive Fibonacci numbers:

(0,) 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, ...

converges to the Golden Ratio:

$$1 / 1 = 1$$

$$2 / 1 = 2$$

$$3 / 2 = 1.5$$

$$5 / 3 = 1.\bar{6}$$

$$8 / 5 = 1.6$$

$$13 / 8 = 1.625$$

$$21 / 13 = 1.615384615384615$$

$$34 / 21 = 1.619047619047619$$

... ..

$$1, 346, 269 / 832, 040 = 1.618033988750541 \approx \phi$$

The Limit of the Sequence of the Ratios

...of successive **Fibonacci numbers** is the **Golden Ratio** itself:

$$\lim_{n \rightarrow \infty} \frac{\text{fib}(n+1)}{\text{fib}(n)} = \frac{1 + \sqrt{5}}{2} = \phi$$

Overall, this might let looking the statement of **Lemma 6.1.2.1** less arbitrarily than it might appear at first sight.

Chapter 6.2

Inductive Proof Principles on Structured Data

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

6.1

6.2

6.2.1

6.2.2

6.3

6.4

6.5

6.6

6.7

6.8

Part IV

Chap. 7

Chap. 8

Chapter 6.2.1

Induction and Recursion

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

6.1

6.2

6.2.1

6.2.2

6.3

6.4

6.5

6.6

6.7

6.8

Part IV

Chap. 7

Chap. 8

Induction and Recursion

...two closely related notions.

Induction

- ▶ describes things starting from something very simple, and building up from there: A **bottom-up** principle.

Recursion

- ▶ starts from the whole thing, working backward to the simple case(s): A **top-down** principle.

Induction and **recursion** can thus be considered

- ▶ the two sides of the same coin.

The Context-Dependent Preferred Usage

...of **induction** over **recursion** resp. vice versa

- ▶ e.g., defining **data structures** (**induction**)
- ▶ e.g., defining **algorithms** (**recursion**)

is often mostly due to historical reasons.

Data types (**inductive view**):

```
data Tree = Leaf Int | Node Tree Int Tree
```

Algorithms (**recursive view**):

```
fac :: Int -> Int
fac n = if n == 0 then 1 else n * fac (n-1)
```

Illustration

- ▶ **Inductive definition** of **arithmetic expressions**:
 - (r1) Each numeral n and variable v is an (atomic) **arithmetic expression**.
 - (r2) If e_1 and e_2 are arithmetic expressions, then also $(e_1 + e_2)$, $(e_1 - e_2)$, $(e_1 * e_2)$, and (e_1 / e_2) .
 - (r3) Every arithmetic expression is **inductively** constructed by means of rules (r1) and (r2).
- ▶ **Recursive definition** of the **merge sort** algorithm:

A list of integers l is sorted by the following 3 steps:

 - (ms1) Split l into two sublists l_1 and l_2 .
 - (ms2) Sort the sublists l_1 and l_2 **recursively** obtaining their sorted counterparts sl_1 and sl_2 , respectively.
 - (ms3) Merge sl_1 and sl_2 into the sorted list sl of l .

In Closing

Data structures often follow an

- ▶ **inductive** definition pattern, e.g.:
 - A **list** is either empty or a pair consisting of an element and another list.
 - A **(binary) tree** is either a leaf or it is composed of a node and a left and a right subtree.
 - An **arithmetic expression** is either a numeral or a variable, or it is composed of (two) arithmetic expressions by means of a (binary) arithmetic operator.

Algorithms (functions) on data structures often follow a

- ▶ **recursive** definition pattern, e.g.:
 - The function **length** computing the length of a list.
 - The function **depth** computing the depth of a tree.
 - The function **evaluate** computing the value of an arithmetic expression (given a valuation of its variables).

Chapter 6.2.2

Structural Induction

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

6.1

6.2

6.2.1

6.2.2

6.3

6.4

6.5

6.6

6.7

6.8

Part IV

Chap. 7

Chap. 8

The Principle of Structural Induction

Let A and O be a set of atoms and operators, respectively; let S be the set of elements inductively constructed from A and O . Let $sub(s) \subseteq S$, $s \in S$, denote the set of elements s is composed of, and let P be a property of the elements of S .

The Principle of Structural Induction

(Inductive) Case

$$\forall s \in S. \underbrace{[(\forall s' \in sub(s). P(s'))]}_{\text{Induction Hypothesis}} \underbrace{\Rightarrow P(s)}_{\text{Induction Step}} \Rightarrow \underbrace{\forall s \in S. P(s)}_{\text{Conclusion}}$$

(dtsh. Prinzip der strukturellen Induktion)

Note: For the atoms \hat{s} of S , the 'simplest' elements of S , we have $sub(\hat{s}) = \emptyset$. For these elements the induction hypothesis boils down to 'true,' i.e., $P(\hat{s})$ has to be proven without relying on anything special.

Example: Illustrating Structural Induction

...the set of (simple) arithmetic expressions \mathcal{AE} is defined by the BNF rule:

$$e ::= n \mid v \mid (e_1 + e_2) \mid (e_1 - e_2) \mid (e_1 * e_2) \mid (e_1/e_2)$$

where n and v stand for (integer) numerals and variables, respectively.

Lemma 6.2.2.1

Let p_e and op_e , $e \in \mathcal{AE}$, denote the number of parentheses and operators of e , respectively. Then:

$$\forall e \in \mathcal{AE}. p_e = 2 * op_e$$

Proof (by means of structural induction).

Proof of Lemma 6.2.2.1 (1)

(Base) case: Let $e \equiv n$, n a numeral, or $e \equiv v$, v a variable.

In both cases e is free of parentheses and operators, i.e.:

$$p_e = 0 = op_e \quad (*)$$

Using $(*)$, equational reasoning yields directly the desired equality:

$$\begin{aligned} (*) &= p_e \\ &= 0 \\ &= 2 * 0 \\ (*) &= 2 * op_e \end{aligned}$$

Proof of Lemma 6.2.2.1 (2)

(Inductive) case: Let $e \equiv (e_1 \circ e_2)$, $\circ \in \{+, -, *, /\}$, and $e_1, e_2 \in \mathcal{AE}$. By means of the **induction hypothesis (IH)**, we can assume $p_{e_1} = 2 * op_{e_1}$ and $p_{e_2} = 2 * op_{e_2}$. The equality of p_e and $2 * op_e$ follows then by equational reasoning:

$$\begin{aligned} & p_e \\ (e \equiv (e_1 \circ e_2)) &= p_{(e_1 \circ e_2)} \\ &= 1 + p_{e_1} + p_{e_2} + 1 \\ (2x \text{ IH}) &= 2 * op_{e_1} + 2 + 2 * op_{e_2} \\ &= 2 * op_{e_1} + 2 * 1 + 2 * op_{e_2} \\ &= 2 * (op_{e_1} + 1 + op_{e_2}) \\ &= 2 * op_{(e_1 \circ e_2)} \\ ((e_1 \circ e_2) \equiv e) &= 2 * op_e \end{aligned}$$



Exercise 6.2.2.2

Prove by means of structural induction:

Lemma 6.2.2.3

Let lp_e and rp_e , $e \in \mathcal{AE}$, denote the number of left and right parentheses of e , respectively. Then:

$$\forall e \in \mathcal{AE}. lp_e = rp_e$$

Lemma 6.2.2.4

Let d_e and opd_e , $e \in \mathcal{AE}$, denote the depth and the number of operands of e , respectively. Then:

$$\forall e \in \mathcal{AE}. opd_e \leq 2^{d_e}$$

Exercise 6.2.2.5

An **arithmetic expression** is called

- **finite**, if the length of all paths originating at its root operator is finite.
- **complete**, if it is finite and all paths from an operand to the root operator are of the same length.

Prove by means of **structural induction**:

Lemma 6.2.2.6

Let d_e and opd_e , $e \in \mathcal{AE}$, denote the **depth** and the **number of operands** of e , respectively. Then:

$$\forall e \in \mathcal{AE}. e \text{ complete} \Rightarrow opd_e = 2^{d_e}$$

Note

...the principles of

- ▶ natural (math.) induction (dtsch. **vollständige** Induktion)

$$P(1) \wedge [\forall n \in \mathbb{IN}. P(n) \Rightarrow P(n+1)] \Rightarrow \forall n \in \mathbb{IN}. P(n)$$

- ▶ strong induction (dtsch. **verallgemeinerte** Induktion)

$$\forall n \in \mathbb{IN}. [(\forall m < n. P(m)) \Rightarrow P(n)] \Rightarrow \forall n \in \mathbb{IN}. P(n)$$

- ▶ structural induction (dtsch. **strukturelle** Induktion)

$$\forall s \in S. [(\forall s' \in \text{sub}(s). P(s')) \Rightarrow P(s)] \Rightarrow \forall s \in S. P(s)$$

are equally expressive.

Chapter 6.3

Inductive Proofs on Algebraic Data Types

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

6.1

6.2

6.3

6.3.1

6.3.2

6.3.3

6.4

6.5

6.6

6.7

6.8

Part IV

Chap. 7

Chapter 6.3.1

Inductive Proofs on Haskell Trees

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

6.1

6.2

6.3

6.3.1

6.3.2

6.3.3

6.4

6.5

6.6

6.7

6.8

Part IV

Chap. 7

Inductive Proofs on Finite Trees

A tree is called

- *finite*, if every path originating at its root has finite length.
- *maximum*, if it is finite and all paths from a leaf to its root have the same length.

Let

```
data Tree = Leaf Int | Node Tree Tree
```

Lemma 6.3.1.1

Let $depth(t)$ and $leaves(t)$ denote the *depth* and the number of *leaves* of any finite tree value $t :: Tree$, respectively. Then:

$$\forall t :: Tree. t \text{ maximum} \Rightarrow leaves(t) = 2^{depth(t)}$$

Proof (by means of structural induction).

Proof of Lemma 6.3.1.1 (1)

Base case: Let $t \equiv (\text{Leaf } k)$ for some integer value k .

Here, we have $\text{depth}(t) = 0$ and $\text{leaves}(t) = 1$. Equational reasoning yields the desired equality of $\text{leaves}(t)$ and $2^{\text{depth}(t)}$:

$$\begin{aligned} (t \equiv (\text{Leaf } k)) &= \text{leaves}(t) \\ &= \text{leaves}(\text{Leaf } k) \\ &= 1 \\ &= 2^0 \\ &= 2^{\text{depth}(t)} \end{aligned}$$

Proof of Lemma 6.3.1.1 (2)

Inductive case: Let $t \equiv (\text{Node } t_1 \ t_2)$ maximum. This implies t_1, t_2 are maximum themselves, $\text{depth}(t_1) = \text{depth}(t_2)$, and $\text{depth}(t) = \text{depth}(t_1) + 1 = \text{depth}(t_2) + 1$. By means of the **inductive hypothesis (IH)** we can assume $\text{leaves}(t_1) = 2^{\text{depth}(t_1)}$ and $\text{leaves}(t_2) = 2^{\text{depth}(t_2)}$. This allows us to complete the proof as follows:

$$\begin{aligned} & \text{leaves}(t) \\ (\text{t} \equiv (\text{Node } t_1 \ t_2)) &= \text{leaves}(\text{Node } t_1 \ t_2) \\ &= \text{leaves}(t_1) + \text{leaves}(t_2) \\ (2 \times \text{IH}) &= 2^{\text{depth}(t_1)} + 2^{\text{depth}(t_2)} \\ (\text{depth}(t_1) = \text{depth}(t_2)) &= 2^{\text{depth}(t_1)} + 2^{\text{depth}(t_1)} \\ &= 2 * 2^{\text{depth}(t_1)} \\ &= 2^{\text{depth}(t_1+1)} \\ &= 2^{\text{depth}(t)} \end{aligned}$$

□

Exercise 6.3.1.2

Prove by means of [structural induction](#):

Lemma 6.3.1.3

Let $depth(t)$ and $leaves(t)$ denote the [depth](#) and the number of [leaves](#) of any finite tree value $t :: Tree$, respectively. Then:

$$\forall t :: Tree. t \text{ finite} \Rightarrow leaves(t) \leq 2^{depth(t)}$$

Note

...[structural induction](#) boils down to [proof by cases](#) if a data type is non-recursively defined.

```
Maybe a = Nothing | Just a
```

```
maybe :: b -> (a -> b) -> Maybe a -> b
```

```
maybe n f Nothing = n
```

```
maybe n f (Just m) = f m
```

A value $x :: \text{Maybe } a$ is called [defined](#), if x equals [Nothing](#) or x equals [\(Just m\)](#) and $m :: a$ is defined (cp. [Chapter 6.3.2](#), why we are cautious on the value of x).

Lemma 6.3.1.4

$$\forall x :: \text{Maybe } \text{Int}. x \text{ defined} \Rightarrow \text{maybe } 2 \text{ abs } x \geq 0$$

Proof of Lemma 6.3.1.4

Case 1: Let $x \equiv \text{Nothing}$. We obtain:

$$\begin{aligned} & \text{maybe } 2 \text{ abs } x \\ = & \text{ maybe } 2 \text{ abs Nothing} \\ = & 2 \\ \geq & 0 \end{aligned}$$

Case 2: Let $x \equiv \text{Just } m$, m defined. We obtain:

$$\begin{aligned} & \text{maybe } 2 \text{ abs } x \\ = & \text{ maybe } 2 \text{ abs (Just } m) \\ = & \text{ abs } m \\ \geq & 0 \end{aligned}$$



Chapter 6.3.2

Inductive Proofs on Haskell Lists

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

6.1

6.2

6.3

6.3.1

6.3.2

6.3.3

6.4

6.5

6.6

6.7

6.8

Part IV

Chap. 7

554/196

Lists

...can contain

- **defined** values only, e.g.:

```
[], (1 : 2 : 3 : []), (1 : 4 'div' 2 : 3 : []),...
```

- **defined** and **undefined** values, e.g.:

```
(1 : 4 'div' 0 : 3 : fac (-1) : []),...
```

```
head (1 : 4 'div' 0 : 3 : fac (-1) : []) ->> 1
```

```
head (tail (1 : 4 'div' 0 : 3 : fac (-1) : [])) ->> 'error'
```

```
head (tail (tail (tail (1 : 4 'div' 0 : 3 : fac (-1) : []))))  
->> 'non-termination'
```

We thus consider

- **defined** and **undefined** values

in more detail and distinguish **structural induction** on lists with

- (only) **defined** values.
- **defined** and **undefined** values.

Chapter 6.3.2.1

Defined and Undefined Values

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

6.1

6.2

6.3

6.3.1

6.3.2

6.3.3

6.4

6.5

6.6

6.7

6.8

Part IV

Chap. 7

556/196

Defined and Undefined Values

A **computation** is

- ▶ **faulty**, if it
 - produces an **error**.
 - does **not (regularly) terminate**.

The value of a **faulty computation** is called

- **undefined** (or: the **undefined value**)

and usually denoted by the symbol \perp (read: '**bottom**').

- ▶ **non-faulty** if its value
 - is different from \perp .

The value of a **non-faulty computation** is called

- **defined** (or: a **defined value**).

Example

The `function`

```
buggy_div :: Int -> Int
buggy_div n = div n 0
```

...produces an error for every argument called with.

The `function`

```
buggy_fac :: Int -> Int
buggy_fac n = (n-1) * buggy_fac n
buggy_fac 0 = 1
```

...does not (regularly) terminate for any argument called with.

Simple Haskell Terms

...with value \perp :

- ▶ **Error:** The Prelude definition

```
undefined :: a                -- polymorphic
undefined | False = undefined

undefined ->> 'error'  $\hat{=}$   $\perp$ 
```

is an **expression** (of arbitrary type) whose evaluation leads to an **error** due to case exhaustion.

- ▶ **Non-termination:** The co-recursive definition

```
loop :: a                    -- polymorphic
loop = loop

loop ->> loop ->> loop ->> ...  $\hat{=}$   $\perp$ 
```

is an **expression** (of arbitrary type) whose evaluation does not (regularly) terminate.

The Undefined Value \perp

- is an element of every Haskell data type, i.e.: $\perp :: a$.
- is the value of faulty or non-terminating computations.
- can be considered an approximation (the 'least accurate' one) of any ordinary value of a data type.

This gives rise to:

Definition 6.3.2.1.1 (Defined, Undefined Values)

The value of any data type representing the result of a faulty or non-terminating computation is called **undefined** and denoted by \perp ; all other values of a data type are called **defined**.

Lists

...are **finite sequences** of values **built from the empty list**.

Definition 6.3.2.1.2 (List)

A **list** is a **possibly empty finite sequence** of

- (defined or undefined) values of the same type
- built from the empty list `[]`.

It is called

- **defined**, if none of its values equals \perp .
- a **list with possibly undefined values**, if some of its values can equal \perp .

Illustration

Haskell lists are

- possibly empty **finite** sequences of values of the **same type**.

Examples: `[]`, `(1:[])`, `(1:2:3:[])`,...

- built from the **empty list**.

Examples: `[]`, `(1:[])`, `(1:2:3:[])`,...

- composed of **defined** and **undefined** values.

Examples: `[]`, `(1:2:[])`, `(1:⊥:3:[])`, `(⊥:⊥:3:[])`,...

Haskell lists are

- **defined**, if all their values are **defined**.

Examples: `[]`, `(1:[])`, `(1:2:3:[])`,...

- lists with **undefined values**, if some of their values equal the **undefined** value.

Examples: `(⊥:[])`, `(1:⊥:[])`, `(⊥:2:⊥:[])`,...

Chapter 6.3.2.2

Structural Induction over Defined Lists

Structural Induction over Defined Lists

Let P be a [property](#) of defined lists.

[Proof pattern of structural induction over defined lists](#)

1. [Base case](#): Prove that $P([])$ is true.
2. [Inductive case](#): Assuming that $P(xs)$ is true ([induction hypothesis](#)), prove that $P(x:xs)$ is true ([induction step](#)).

[Note](#): This pattern is an instance of the more general pattern of [structural induction](#), specialized here for defined lists.

Example 1: Induction over Defined Lists

Let

```
length :: [a] -> Int
length []      = 0
length (_:xs) = 1 + length xs
```

Lemma 6.3.2.2.1

$\forall xs, ys \text{ defined} :: [a].$

$\text{length } (xs ++ ys) = \text{length } xs + \text{length } ys$

Proof (by induction on the structure of xs).

Proof of Lemma 6.3.2.2.1 (1)

Let $ys :: [a]$ be a defined list.

Base case: Let $xs \equiv []$. As desired, we obtain by means of equational reasoning:

$$\begin{aligned} & \text{length } (xs ++ ys) \\ = & \text{length } ([] ++ ys) \\ = & \text{length } ys \\ = & 0 + \text{length } ys \\ = & \text{length } [] + \text{length } ys \\ = & \text{length } xs + \text{length } ys \end{aligned}$$

Proof of Lemma 6.3.2.2.1 (2)

Inductive case: Let $xs \equiv (x:xs')$, xs defined. This implies xs' (and x) is defined, too. By means of the **induction hypothesis (IH)**, we can thus assume $\text{length } (xs' ++ ys) = (\text{length } xs' + \text{length } ys)$. This allows to complete the proof as follows:

$$\begin{aligned} & \text{length } (xs ++ ys) \\ = & \text{length } ((x:xs') ++ ys) \\ = & \text{length } (x:(xs' ++ ys)) \\ = & 1 + \text{length } (xs' ++ ys) \\ \text{(IH)} \quad = & 1 + (\text{length } xs' + \text{length } ys) \\ = & (1 + \text{length } xs') + \text{length } ys \\ = & \text{length } (x:xs') + \text{length } ys \\ = & \text{length } xs + \text{length } ys \end{aligned}$$



Example 2: Induction over Defined Lists

Let

```
listSum :: Num a => [a] -> a
listSum []      = 0
listSum (x:xs) = x + listSum xs
```

Lemma 6.3.2.2

$\forall xs \text{ defined} :: [a]. \text{listSum } xs = \text{foldr } (+) \ 0 \ xs$

Proof (by induction on the structure of xs).

Proof of Lemma 6.3.2.2.2 (1)

Base case: Let $xs \equiv []$. Equational reasoning yields the desired equality:

$$\begin{aligned} & \text{listSum } xs \\ = & \text{listSum } [] \\ = & 0 \\ = & \text{foldr } (+) \ 0 \ [] \\ = & \text{foldr } (+) \ 0 \ xs \end{aligned}$$

Proof of Lemma 6.3.2.2.2 (2)

Inductive case: Let $xs \equiv (x:xs')$, xs defined. This implies xs' (and x) is defined, too. By means of the **induction hypothesis (IH)**, we can thus assume $listSum\ xs' = foldr\ (+)\ 0\ xs'$. This allows us to complete the proof as follows:

$$\begin{aligned} & listSum\ xs \\ = & listSum\ (x:xs') \\ = & x + listSum\ xs' \\ \text{(IH)} \quad = & x + foldr\ (+)\ 0\ xs' \\ = & foldr\ (+)\ 0\ (x:xs') \\ = & foldr\ (+)\ 0\ xs \end{aligned}$$

□

Example 3: Induction over Defined Lists

Lemma 6.3.2.2.3

$$\forall xs \text{ defined} :: [a]. \text{reverse (reverse xs)} = xs$$

Proof (by induction on the structure of xs).

Proof of Lemma 6.3.2.2.3 (1)

Base case: Let $xs \equiv []$. Equational reasoning yields the desired equality:

$$\begin{aligned} & \text{reverse (reverse xs)} \\ &= \text{reverse (reverse [])} \\ \text{(Def. reverse)} &= \text{reverse []} \\ \text{(Def. reverse)} &= [] \\ &= xs \end{aligned}$$

Proof of Lemma 6.3.2.2.3 (2)

Inductive case: Let $xs \equiv (x:xs')$, xs defined. This implies xs' and x are defined, too. By means of the **induction hypothesis (IH)**, we can thus assume **reverse (reverse xs') = xs'** . This allows us to complete the proof as follows:

$$\begin{aligned} \text{reverse (reverse } xs) &= \text{reverse (reverse (} x:xs')) \\ \text{(Def. reverse)} &= \text{reverse ((reverse } xs') ++ [x]) \\ \text{(L. 6.3.2.8(1))} &= \text{reverse [x] ++ reverse (reverse } xs') \\ \text{([x] = x: [], IH)} &= \text{reverse (x: []) ++ } xs' \\ \text{(Def. reverse)} &= (\text{reverse [] ++ [x]}) ++ xs' \\ \text{(Def. reverse)} &= ([] ++ [x]) ++ xs' \\ \text{(Def. (++))} &= [x] ++ xs' \\ \text{([x] = x: [])} &= (x : []) ++ xs' \\ \text{(Def. (++))} &= x : ([] ++ xs') \\ \text{(Def. (++))} &= x:xs' \\ &= xs \end{aligned}$$

□

Example 4

...sometimes, a truly inductive argument is not required.

Lemma 6.3.2.2.4

Let f be a *strict* map. Then:

$$\forall xs \text{ defined } :: [a]. (f . head) xs = head . (map f xs)$$

Proof (by cases).

Proof of Lemma 6.3.2.2.4 (1)

Case 1: Let $xs \equiv []$. We get:

$$\begin{aligned} & (f \ . \ head) \ xs \\ &= (f \ . \ head) \ [] \\ \text{(Def. of (.))} &= f \ (head \ []) \\ \text{(Def. of head)} &= f \ \perp \\ \text{(f strict)} &= \perp \\ \text{(Def. of head)} &= head \ [] \\ \text{(Def. of map)} &= head \ (map \ f \ []) \\ \text{(Def. of (.))} &= (head \ . \ map \ f) \ [] \\ &= (head \ . \ map \ f) \ xs \end{aligned}$$

Proof of Lemma 6.3.2.2.4 (2)

Case 2: Let $xs \equiv (x:xs')$, xs defined. This implies xs' and x are defined, too. We get:

$$\begin{aligned} & (f \ . \ head) \ xs \\ &= (f \ . \ head) \ (x:xs') \\ \text{(Def. of (.))} &= f \ (head \ (x:xs')) \\ \text{(Def. of head)} &= f \ x \\ \text{(Def. of head, lazy eval.)} &= head \ (f \ x \ : \ map \ f \ xs') \\ \text{(Def. of map)} &= head \ (map \ f \ (x:xs')) \\ \text{(Def. of (.))} &= (head \ . \ map \ f) \ (x:xs') \\ &= (head \ . \ map \ f) \ xs \end{aligned}$$

□

Note: The induction hypothesis $(f \ . \ head) \ xs' = (head \ . \ map \ f) \ xs'$ is not required to complete the proof of case 2; the inductive proof boils down to a proof by cases.

Exercise 6.3.2.2.5

...examples involving list **reversions** and **concatenations**.

Prove by means of **structural induction** over defined lists:

Lemma 6.3.2.2.6

For all xs, ys, zs *defined* $:: [a]$ we have:

1. $\text{reverse } (xs ++ ys) = \text{reverse } ys ++ \text{reverse } xs$
2. $(xs ++ ys) ++ zs = xs ++ (ys ++ zs)$
3. $xs ++ [] = xs$
4. $\text{head } (\text{reverse } xs) = \text{last } xs$
5. $\text{last } (\text{reverse } xs) = \text{head } xs$

Corollary 6.3.2.2.7

For all xs *defined* $:: [a]$ we have:

$$xs ++ [] = xs = [] ++ xs$$

Exercise 6.3.2.2.8

...examples involving list `take` and `drop` operations.

Prove by means of `structural induction on defined lists`:

Lemma 6.3.2.2.9

For all `xs defined :: [a]`, $m, n \in \mathbb{N}$, $m, n \geq 0$, we have:

- $$\begin{aligned} \text{take } n \text{ xs} ++ \text{drop } n \text{ xs} &= \text{xs} \\ \text{take } m . \text{take } n &= \text{take } (\min m \ n) \\ \text{drop } m . \text{drop } n &= \text{drop } (m+n) \\ \text{take } m . \text{drop } n &= \text{drop } n . \text{take } (m+n) \end{aligned}$$
- If (additionally) $n \geq m$, we have:
$$\text{drop } m . \text{take } n = \text{take } (n-m) . \text{drop } m$$

Exercise 6.3.2.2.10

...examples involving list *foldings*.

Prove by means of *structural induction over defined lists*:

Lemma 6.3.2.2.11

Let $op :: (a \rightarrow a \rightarrow a)$ be associative with unit $e :: a$, i.e.,
 $\forall x :: a. e \text{ 'op' } x = x \wedge x \text{ 'op' } e = x$. Then:

$$(\forall xs :: [a]. xs \text{ defined}). \text{ foldr } op \ e \ xs \\ = \text{ foldl } op \ e \ xs$$

Lemma 6.3.2.2.12

Let $op :: (a \rightarrow b \rightarrow b)$ be an operator, $e :: b$ a value. Then:

$$(\forall xs :: [a]. xs \text{ defined}). \text{ foldr } op \ e \ xs \\ = \text{ foldl } (\text{flip } op) \ e \ (\text{reverse } xs)$$

Exercise 6.3.2.2.13

...examples involving list *foldings*.

Prove by means of *structural induction over defined lists*:

Lemma 6.3.2.2.14

Let $op1, op2 :: (a \rightarrow a \rightarrow a)$ be two operators, $e :: b$ a value such that:

$$\forall x,y,z :: a. \quad x \text{ 'op1' } (y \text{ 'op2' } z) = (x \text{ 'op1' } y) \text{ 'op2' } z \wedge \\ x \text{ 'op1' } e = e \text{ 'op2' } x$$

Then:

$$(\forall xs :: [a]. \text{xs defined}). \text{foldr } op1 \ e \ xs \\ = \text{foldl } op2 \ e \ xs$$

Exercise 6.3.2.2.15

...examples involving [sequential composition](#) and [mappings](#).

Prove by means of [structural induction over defined lists](#):

Lemma 6.3.2.2.16

1. $\text{map } (f \ . \ g) \quad = \text{map } f \ . \ \text{map } g$
2. $(\text{map } f) \ . \ \text{tail} \quad = \text{tail} \ . \ \text{map } f$
3. $(\text{map } f) \ . \ \text{reverse} \ = \ \text{reverse} \ . \ \text{map } f$
4. $(\text{map } f) \ . \ \text{concat} \ = \ \text{concat} \ . \ \text{map } (\text{map } f)$
5. $\text{map } f \ (xs \ ++ \ ys) \ = \ \text{map } f \ xs \ ++ \ \text{map } f \ ys$
6. $\text{map } (\backslash x \ -> \ x) \quad = \ \backslash y \ -> \ y$

What are the types of the two anonymous [\$\lambda\$ -abstractions](#) in [Lemma 6.3.2.2.16\(6\)](#)? Do they have the same type or different ones?

Chapter 6.3.2.3

Structural Induction over Lists with Undefined Values

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

6.1

6.2

6.3

6.3.1

6.3.2

6.3.3

6.4

6.5

6.6

6.7

6.8

Part IV

Chap. 7

Structural Induction for Lists w/ Undef. Values

Let P be a **property** of lists with possibly undefined values.

Proof pattern of structural induction over lists with possibly undefined values:

1. **Base case:** Prove that $P([])$ is true.
2. **Inductive case:** Assuming that $P(xs)$ is true (**induction hypothesis**), prove that $P(\perp:xs)$ and $P(x:xs)$, x a defined value, are true (**induction step**).

Note: This pattern is an instance of the more general pattern of **structural induction**, specialized here for lists with possibly undefined values.

Example: Induct. over Lists w/ Undef. Values

Let

```
length :: [a] -> Int
length []      = 0
length (_:xs) = 1 + length xs
```

Lemma 6.3.2.3.1

$\forall xs, ys$ with possibly undefined values $:: [a]$.
 $\text{length } (xs ++ ys) = \text{length } xs + \text{length } ys$

Proof (by induction over the structure of xs).

Proof of Lemma 6.3.2.3.1 (1)

Let $ys :: [a]$ be a list with possibly undefined values.

Base case: Let $xs \equiv []$. As desired, we obtain by means of equational reasoning:

$$\begin{aligned} & \text{length } (xs ++ ys) \\ = & \text{length } ([] ++ ys) \\ = & \text{length } ys \\ = & 0 + \text{length } ys \\ = & \text{length } [] + \text{length } ys \\ = & \text{length } xs + \text{length } ys \end{aligned}$$

Proof of Lemma 6.3.2.3.1 (2)

Inductive case 1: Let $xs \equiv (\perp : xs')$. By means of the **induction hypothesis (IH)**, we can assume $\text{length } (xs' ++ ys) = (\text{length } xs' + \text{length } ys)$. This allows to complete the proof as follows:

$$\begin{aligned} & \text{length } (xs ++ ys) \\ = & \text{length } ((\perp : xs') ++ ys) \\ = & \text{length } (\perp : (xs' ++ ys)) \\ = & 1 + \text{length } (xs' ++ ys) \\ \text{(IH)} \quad = & 1 + (\text{length } xs' + \text{length } ys) \\ = & (1 + \text{length } xs') + \text{length } ys \\ = & \text{length } (\perp : xs') + \text{length } ys \\ = & \text{length } xs + \text{length } ys \end{aligned}$$

Proof of Lemma 6.3.2.3.1 (3)

Inductive case 2: Let $xs \equiv (x:xs')$, x defined. By means of the **induction hypothesis (IH)**, we can assume $\text{length } (xs' ++ ys) = (\text{length } xs' + \text{length } ys)$. This allows to complete the proof as follows:

$$\begin{aligned} & \text{length } (xs ++ ys) \\ = & \text{length } ((x:xs') ++ ys) \\ = & \text{length } (x:(xs' ++ ys)) \\ = & 1 + \text{length } (xs' ++ ys) \\ \text{(IH)} = & 1 + (\text{length } xs' + \text{length } ys) \\ = & (1 + \text{length } xs') + \text{length } ys \\ = & \text{length } (x:xs') + \text{length } ys \\ = & \text{length } xs + \text{length } ys \end{aligned}$$



Exercise 6.3.2.3.2

Which of the statements of the lemmas of [Chapter 6.3.2.2](#) hold for [lists with possibly undefined values](#), too?

Prove your claims or provide counter-examples.

Chapter 6.3.3

Inductive Proofs on Partial Haskell Lists

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

6.1

6.2

6.3

6.3.1

6.3.2

6.3.3

6.4

6.5

6.6

6.7

6.8

Part IV

Chap. 7

Chapter 6.3.3.1

Partial Lists

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

6.1

6.2

6.3

6.3.1

6.3.2

6.3.3

6.4

6.5

6.6

6.7

6.8

Part IV

Chap. 7

Partial Lists

...are finite sequences of values built from the undefined list.

Definition 6.3.3.1.1 (Partial List)

A **partial list** is a possibly empty finite sequence of

- (defined or undefined) values of the same type
- built from the undefined list \perp .

It is called

- **defined**, if none of its values equals \perp (and there is at least one).
- a **partial list with possibly undefined values**, if some of its values can equal \perp .

Illustration

Partial Haskell lists are

- possibly empty **finite** sequences of values built from the **undefined list**.

Examples: \perp , $(1:\perp)$, $(1:2:\perp)$, $(1:2:3:\perp)$,...

- partial lists with **undefined values**, if some of their values equal the **undefined** value.

Examples: $(1:\perp:3:\perp)$, $(1:\perp:\perp:\perp)$. $(\perp:\perp:\perp:\perp)$,...

Note the different types of \perp and \perp in the above **examples**:

$\perp :: \text{Int}$

$\perp :: [\text{Int}]$

Chapter 6.3.3.2

Computing with Lists and Partial Lists

Some Examples of Lists and Partial Lists

...with and without `undefined` values:

```
empty = []                -- Empty list
ns = 2 : 3 : 5 : 7 : []   -- Defined list
ms = 2 : loop : 5 : 7 : [] -- List w/undefined
                               -- values

pempty = loop :: [Int]    -- Empty partial list
xs = 2 : 3 : 5 : 7 : loop -- Def. partial list
ys = 2 : loop : 5 : 7 : loop -- Partial list w/
                               -- undefined values
```

Note: All occurrences of `loop` in `ms`, `ys`, and `pempty` have value \perp but of different type:

- `loop` = \perp :: `Int` in `ms` and `ys`.
- `loop` = \perp :: `[Int]` in `pempty`, `xs`, and `ys`.

Using the Definitions (1)

...introduced before, we get:

```
reverse ns ->> [7,5,3,2]
```

```
reverse ms ->> [7,5 ...followed by an infinite wait
```

```
reverse xs ->> ...infinite wait
```

```
reverse ys ->> ...infinite wait
```

```
head (reverse ms) ->> 7           -- thanks to lazy eval.
```

```
head (tail (reverse ms)) ->> 5   -- thanks to lazy eval.
```

```
head (tail (tail (reverse ms))) ->> ...infinite wait
```

```
head (tail (reverse xs)) ->> ...infinite wait
```

```
last ms ->> 7
```

```
last xs ->> ...infinite wait
```

```
reverse (reverse ms) ->> [2 ...followed by an  
                           infinite wait
```

```
head (reverse (reverse ms)) ->> 2
```

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

6.1

6.2

6.3

6.3.1

6.3.2

6.3.3

6.4

6.5

6.6

6.7

6.8

Part IV

Chap. 7

595/196

Using the Definitions (2)

...introduced before, we also get:

```
length ns ->> 4
length ms ->> 4
length xs ->> ...infinite wait
length ys ->> ...infinite wait

length (take 4 ns) ->> 4
length (take 3 ms) ->> 3
length (take 2 xs) ->> 2
length (take 3 ys) ->> 3

length (take 5 ns) ->> 4
length (take 4 xs) ->> 4
length (take 5 xs) ->> ...infinite wait
```

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

6.1

6.2

6.3

6.3.1

6.3.2

6.3.3

6.4

6.5

6.6

6.7

6.8

Part IV

Chap. 7

596/196

The Different Evaluation Behaviour

...of `length` and `reverse` is due to **requiring** or **not-requiring** a pattern match on the **values** of the argument:

```
length :: [a] -> Int
length []      = 0
length (_:xs) = 1 + length xs      -- No pattern match
                                     -- on the head of the
                                     -- argument list!

reverse :: [a] -> [a]
reverse []      = []
reverse (x:xs) = reverse xs ++ [x] -- Pattern match on
                                     -- the head of the
                                     -- argument list!

reverse :: [a] -> [a]
reverse = foldl (flip (:)) []      -- Same here, even if
                                     -- pointfree defined!
```

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

6.1

6.2

6.3

6.3.1

6.3.2

6.3.3

6.4

6.5

6.6

6.7

6.8

Part IV

Chap. 7

597/196

Similarly

...this holds for `take` and `drop`:

```
take :: Int -> [a] -> [a]
take n _   | n <= 0 = []
take _ []   = []
take n (x:xs) = x : take (n-1) xs -- Pattern
                                     -- match on the head of
                                     -- of the argument list!
```

```
drop :: Int -> [a] -> [a]
drop n xs   | n <= 0 = xs
drop _ []   = []
drop n (_:xs) = drop (n-1) xs -- No pattern
                                     -- match on the
                                     -- head of the
                                     -- argument list!
```

And also

...for `head`, `tail`, and `last`:

```
head :: [a] -> a
```

```
head (x:_) = x
```

```
-- Pattern match on the  
-- head of the argu-  
-- ment list!
```

```
tail :: [a] -> [a]
```

```
tail (_:xs) = xs
```

```
-- No pattern match  
-- on the head of the  
-- argument list!
```

```
last :: [a] -> a
```

```
last [x] = x
```

```
last (_:xs) = last xs
```

```
-- Pattern match on the  
-- argument list
```

Finally

...also `data` and `newtype` declarations behave slightly different regarding `pattern matching`. Compare:

```
data Bool' = B' Bool
```

```
hello' :: Bool' -> String
```

```
hello' (B' _) = "Hello!"
```

```
hello' loop ->> ...infinite wait  -- Pattern match  
                -- required since data declarations might  
                -- have more than one data constructor.
```

```
newtype Bool'' = B'' Bool
```

```
hello'' :: Bool'' -> String
```

```
hello'' (B'' _) = "Hello!"
```

```
hello'' loop ->> "Hello!" -- No Pattern match required  
                    -- since newtype declarations have  
                    -- exactly one data constructor.
```


Last but not least

...`hello'''` behaves also slightly different compared to `hello'` and `hello''`. Pattern matching is not required but for a different reason as for `hello''`:

```
data Bool' = B' Bool

hello''' :: Bool' -> String
hello''' _ = "Hello!"

hello''' loop ->> "Hello!" -- No Pattern match required
                             -- since any of possibly
                             -- many data constructors
                             -- matches.
```

In closing: Undefined values cause program failure, whenever they need to be (partially) evaluated for pattern matching or to be displayed as (part of) the result of evaluating a term; the details are subtle as demonstrated by the above examples.

Chapter 6.3.3.3

Inductive Proof Patterns for Partial Lists

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

6.1

6.2

6.3

6.3.1

6.3.2

6.3.3

6.4

6.5

6.6

6.7

6.8

Part IV

Chap. 7

602/196

The Inductive Proof Patterns

...introduced in [Chapter 6.3.2.2](#) and [6.3.2.3](#) apply to [lists](#) (possibly with undefined values), which (by definition) are built from the empty list `[]`.

By contrast, [partial lists](#) (possibly with undefined values, too) are built from the undefined list `⊥`.

We thus need to adapt the inductive proof principles for [lists](#) to work for [partial lists](#) (with possibly undefined values).

Inductive Proofs on Partial Lists

Let P be a [property](#) of partial lists.

A) Proof pattern for defined partial lists:

1. Base case: Prove that $P(\perp)$ is true.
2. Inductive case: Assuming that $P(xs)$ is true (induction hypothesis), prove that $P(x:xs)$ is true (induction step).

B) Proof pattern for partial lists w/ possibly undefined values:

1. Base case: Prove that $P(\perp)$ is true.
2. Inductive case: Assuming that $P(xs)$ is true (induction hypothesis), prove that $P(\perp:xs)$ and $P(x:xs)$, x a defined value, are true (induction step).

Exercise 6.3.3.3.1

Does [Lemma 6.3.2.2.4](#) recalled below hold for [defined partial lists](#), too? Does it make a difference if [partial lists](#) may have values equal to the [undefined value](#) or not?

Lemma 6.3.2.2.4

Let f be a [strict map](#). Then:

$$\forall xs \text{ defined} :: [a]. (f . \text{head}) \text{ xs} = \text{head} . (\text{map } f \text{ xs})$$

Provide a proof or a counter-example to support your claims.

Exercise 6.3.3.3.2

Which of the statements of the lemmas in Chapter 6.3.2.2 and 6.3.2.3 hold for

1. defined partial lists?
2. partial lists with possibly undefined values?

Prove your claims or provide counter-examples.

Inductive Proofs on Lists and Partial Lists

Let P be a *property* defined of lists and partial lists.

C) Proof pattern for lists and partial lists with possibly undefined values:

1. Base case: Prove that $P(\perp)$ and $P([])$ are true.
2. Inductive case: Assuming that $P(xs)$ is true (induction hypothesis), prove that $P(\perp:xs)$ and $P(x:xs)$, x a defined value, are true (induction step).

Exercise 6.3.3.3.3

Which of the statements of the lemmas in Chapter 6.3.2.2, 6.3.2.3, and 6.3.3 hold for:

1. defined lists and defined partial lists?
2. lists and partial lists with possibly undefined values?

Prove your claims or provide counter-examples.

Chapter 6.4

Proving Properties of Streams

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

6.1

6.2

6.3

6.4

6.4.1

6.4.2

6.5

6.6

6.7

6.8

Part IV

Chap. 7

Chap. 8

Chapter 6.4.1

Inductive Proofs on Haskell Stream Approximants

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

6.1

6.2

6.3

6.4

6.4.1

6.4.2

6.5

6.6

6.7

6.8

Part IV

Chap. 7

Chap. 8

Streams

...are **infinite** sequences of values of the same type.

Definition 6.4.1.1 (Stream)

A **stream** is an **infinite sequence** of (defined or undefined) values of the same type.

Definition 6.4.1.2 (Def. Stream, S. w/ Undef. Values)

A **stream** is called

1. **defined**, if all its values are defined.
2. a **stream with possibly undefined values**, if some of its values can be equal to \perp .

Homework: Does it make sense to say, a stream were built from the empty stream or the undefined stream?

Comparing Partial Lists: Approximation Order

...intuitively, a partial list xs approximates a partial list ys , if xs is 'equal to but less defined' than ys , $xs \sqsubseteq ys$:

$$\begin{array}{c} \perp \sqsubseteq 0 : \perp \\ 0 : \perp \sqsubseteq 0 : 1 : \perp \\ 0 : 1 : \perp \sqsubseteq 0 : 1 : 1 : \perp \\ 0 : 1 : 1 : 2 : \perp \sqsubseteq 0 : 1 : 1 : 2 : 3 : \perp \\ \dots \\ \perp \sqsubseteq 0 : 1 : 1 : 2 : 3 : 5 : 8 : \perp \\ 0 : \perp \sqsubseteq 0 : 1 : 1 : 2 : 3 : 5 : 8 : \perp \\ 0 : 1 : 2 : \perp \sqsubseteq 0 : 1 : 1 : 2 : 3 : 5 : 8 : \perp \\ \dots \end{array}$$

Streams can be approximated by infinite sequences of

- increasingly more accurate partial lists, called PL-approximants.

Illustrating Stream Approximation

...the stream of natural numbers

$[1..] = 1 : 2 : 3 : 4 : 5 : 6 : 7 : 8 : 9 : \dots$

is approximated by the infinite sequence of more and more accurate PL-approximants, whose limit is the stream itself:

\perp
 $\sqsubseteq 1 : \perp$
 $\sqsubseteq 1 : 2 : \perp$
 $\sqsubseteq 1 : 2 : 3 : \perp$
 $\sqsubseteq 1 : 2 : 3 : 4 : \perp$
 $\sqsubseteq 1 : 2 : 3 : 4 : 5 : \perp$
 $\sqsubseteq 1 : 2 : 3 : 4 : 5 : 6 : \perp$
 $\sqsubseteq 1 : 2 : 3 : 4 : 5 : 6 : 7 : \perp$
 $\sqsubseteq 1 : 2 : 3 : 4 : 5 : 6 : 7 : 8 : \perp$
 \dots
 $\sqsubseteq 1 : 2 : 3 : 4 : 5 : 6 : 7 : 8 : 9 : \dots = [1..]$

Intuitively

...the **undefined list** \perp is the **'least defined'** partial list; hence, the **'least accurate'** approximant of a stream. Sequences of more and more **'defined'** approximants are getting more and more **'accurate.'**

...considering **partial lists** (which are finite by definition)

- ▶ **approximations** of streams equals in spirit the approach of outputting/printing a **stream prefix** by interrupting the printing of the stream after some period of time by hitting **Ctrl-C**.

Extending this **period of time** further and further yields

- ▶ **more and more accurate approximants** of the **stream**.

Partial Orders, Chains

...towards formalizing the idea of approximation:

Definition 6.4.1.3 (Partially Ordered Set)

A set M with a binary relation R is called a **partially ordered set** iff R is reflexive, transitive, and anti-symmetric; the pair (M, R) is called a **partial order**, and R a **partial order on M** .

Definition 6.4.1.4 (Chain)

A subset $C \subseteq P$ of a partial order (P, \sqsubseteq) is called a **chain**, if C is totally ordered.

Domains

Definition 6.4.1.5 (Domain)

A partial order (D, \sqsubseteq) is called a **domain** (or: **complete partial order (CPO)**), if

1. D has a least element \perp .
2. $\bigsqcup C$ exists for every chain C in D .

The relation \sqsubseteq is then called **approximation order** of (D, \sqsubseteq) .

Example: Let $\mathcal{P}(\mathbb{N})$ be the power set of \mathbb{N} . Then: $(\mathcal{P}(\mathbb{N}), \sqsubseteq)$, $\sqsubseteq =_{df} \subseteq$, is a domain with:

- least element \emptyset
- $\bigsqcup C = \bigcup C$ for every chain $C \subseteq \mathcal{P}(\mathbb{N})$

Approximation Order on Partial Lists, Streams

...let $S_{(PL,St)} =_{df} \{s \mid s \text{ partial list or stream}\}$ be the set of partial lists and streams.

Lemma 6.4.1.6 (Partial Order on $S_{(PL,St)}$)

The relation \sqsubseteq on $S_{(PL,St)}$ defined by:

$$\perp \quad \sqsubseteq \quad xs \\ x : xs \quad \sqsubseteq \quad y : ys \quad \iff_{df} \quad x \equiv y \wedge xs \sqsubseteq ys$$

is a partial order on $S_{(PL,St)}$, where \equiv denotes equality on list resp. stream entries.

Lemma 6.4.1.7 (Domain $((S_{(PL,St)}, \sqsubseteq))$)

$(S_{(PL,St)}, \sqsubseteq)$ with \sqsubseteq as in Lemma 6.4.1.6 is a domain with least element \perp and approximation order \sqsubseteq .

Partial Lists as Stream Approximants

Definition 6.4.1.8 (PL-Approximants)

Let xs be a defined stream. The set of PL-approximants of xs is the set $PL\text{-}Approx(xs) =_{df} \{ take' \ n \ xs \mid n \in \mathbb{N}_0 \}$, where

$take' :: Integer \rightarrow [a] \rightarrow [a]$

$take' \ n \ _ \mid n \leq 0 = \text{undefined}$

$take' \ n \ (x:xs) = x : take' \ (n-1) \ xs$

Note: PL-approximants are built from the undefined list, not the empty list; they all have finite length.

Examples:

- $PL\text{-}Approx([1..]) = \{\perp, 1:\perp, 1:2:\perp, 1:2:3:\perp, \dots\}$
- $PL\text{-}Approx([1,1..]) = \{\perp, 1:\perp, 1:1:\perp, 1:1:1:\perp, \dots\}$

Main Result: Approximation

Lemma 6.4.1.9 (PL-Approximants Chain)

The set $PL\text{-}Approx(xs)$, xs a defined stream, is a chain.

Theorem 6.4.1.10 (Approximation)

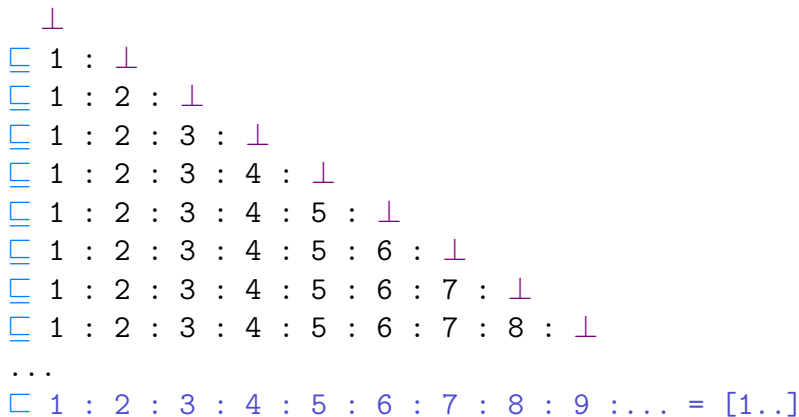
Let xs be a defined stream. Then xs is equal to the least upper bound of its PL-approximants set, its so-called **limit**:

$$xs = \bigsqcup PL\text{-}Approx(xs) = \bigsqcup_{n=0}^{\infty} \text{take}'\ n\ xs$$

Note: Refer to [Appendix A](#) for the definition of technical terms and illustrating examples, if required.

Streams as Limit of their PL-Approximants Sets

...the set of **PL-approximants** of a defined **stream** is a **chain** with the stream itself as its least upper bound (cf. [Approximation Theorem 6.4.1.10](#)) as illustrated below:



Finite and Infinite Sequences of Values

...are quite **diverse objects** enjoying **different properties**.

Properties valid for **lists** (i.e., finite sequences) **might hold** or **might not hold** for **streams** (i.e., infinite sequences) and vice versa, e.g.:

- $\forall z \in \mathbb{Z}. \text{take } n \text{ xs} ++ \text{drop } n \text{ xs} = \text{xs}$
...does hold for defined **lists** and **streams**.
- $\text{reverse} (\text{reverse } \text{xs}) = \text{xs}$
...does hold for defined **lists** but **not** for **streams**.
- $\forall n \in \mathbb{N}. \text{drop } n \text{ xs} \neq []$
...does hold for **streams** but **not** for **lists**.

Finite PL-Approximants and Streams

...are quite **diverse objects**, too.

Properties which are valid for **every partial list** of the **infinite set of finite PL-approximants** of a stream **might hold** or **might not hold** for its **limit**, the stream itself, and vice versa, e.g.:

- $\text{map } (f . g) \text{ } xs = (\text{map } f . \text{map } g) \text{ } xs$
does hold for all **PL-approximants** of a defined stream **and** the **stream** itself.
- *'This sequence is partial'*
...does hold for all **PL-approximants** of a stream but **not** for the **stream** itself.
- $\text{tail } xs$ *'is a stream'*
...does hold for a **stream** but **not** for any of its **PL-approximants**.

Reconsidering the Induction Principles

...considered so far.

The **induction principles** of **Chapter 6.3.2** and **6.3.3** apply to

- ▶ **finite** sequences of (possibly undefined) values

This allows proving properties for **all finite lists** and/or **all finite partial lists** (with possibly undefined values).

Streams, however, are by definition

- ▶ **infinite** sequences of values.

The **induction principles** of **Chapter 6.3.2** and **6.3.3** are thus not directly applicable for proving properties on **streams**, especially in the light of the fact that properties being valid for every PL-approximant of a stream need not hold for the stream itself.

Fortunately

...the [induction principle for partial lists](#) (with and without possibly undefined values) of [Chapter 6.3.3](#) can be used to prove so-called (in analogy to [Definition 6.4.4.1](#))

- [admissible](#) properties of approximant sets

for streams.

A property of a PL-approximants set is [admissible](#) if it holds for its limit, if it holds for each of its elements.

[Equational properties](#) are admissible.

Together with [Approximation Theorem 6.4.1.10](#), this justifies the inductive proof principles considered next.

Inductive Proofs on PL-Approximants Sets

...for proving 'admissible' properties of streams.

Let P be an equational property defined on PL-approximants and streams.

A) Proof pattern for defined PL-approximants:

1. Base case: Prove that $P(\perp)$ is true.
2. Inductive case: Assuming that $P(xs)$ is true (induction hypothesis), prove that $P(x:xs)$ is true (induction step).

B) Proof pattern for PL-approximants w/ possibly undef. values:

1. Base case: Prove that $P(\perp)$ is true.
2. Inductive case: Assuming that $P(xs)$ is true (induction hypothesis), prove that $P(\perp:xs)$ and $P(x:xs)$, x a defined value, are true (induction step).

Example 1: Induction on PL-Approximants

Lemma 6.4.1.11

We have:

$$(\forall \mathbf{xS} \in [\mathbf{a}]. \mathbf{xS} \text{ defined stream}) \forall n \in \mathbb{N}. \\ \text{take } n \mathbf{xS} ++ \text{drop } n \mathbf{xS} = \mathbf{xS}$$

Proof by cases and induction on the structure of \mathbf{xS} .

Proof of Lemma 6.4.1.11 (1)

Case 1: Let $n \in \mathbb{N}$, $n = 0$, and xs be some defined stream. Equational reasoning yields the desired equality:

$$\begin{aligned} & \text{take } n \text{ } xs \text{ } ++ \text{ drop } n \text{ } xs \\ &= \text{take } 0 \text{ } xs \text{ } ++ \text{ drop } 0 \text{ } xs \\ \text{(Def. take)} \quad &= [] \text{ } ++ \text{ } xs \\ &= xs \end{aligned}$$

Case 2: Let $n \in \mathbb{N}$, $n \geq 1$ be some natural number. We now proceed by induction on the structure of xs .

Base case: Let $xs \equiv \perp$. Equational reasoning yields as desired:

$$\begin{aligned} & \text{take } n \text{ } xs \text{ } ++ \text{ drop } n \text{ } xs \\ &= \text{take } n \text{ } \perp \text{ } ++ \text{ drop } n \text{ } \perp \\ \text{(Def. take, case exh.)} \quad &= \perp \text{ } ++ \perp \\ &= \perp \\ &= xs \end{aligned}$$

Proof of Lemma 6.4.1.11 (2)

Inductive case: Let $xs \equiv (x:xs')$ be a defined PL-approximant. Then x is defined and xs' is a defined PL-approximant, too. By means of **Case 1** (if $n=1$) and the **induction hypothesis (IH)** (if $n>1$), we can assume for all $n \in \mathbb{N}$ the equality $(\text{take } (n-1) \text{ } xs' ++ \text{drop } (n-1) \text{ } xs') = xs'$. This allows us to complete the proof as follows:

$$\begin{aligned} & \text{take } n \text{ } xs ++ \text{drop } n \text{ } xs \\ = & \text{take } n \text{ } (x:xs') ++ \text{drop } n \text{ } (x:xs') \\ = & x : (\text{take } (n-1) \text{ } xs' ++ \text{drop } (n-1) \text{ } xs') \\ \text{(Case 1, IH)} = & x : xs' \\ = & (x:xs') \\ = & xs \end{aligned}$$

□

Example 2: Induction on PL-Approximants

Consider the following variant of [Lemma 6.4.1.11](#):

Lemma 6.4.1.12

We have:

$$(\forall xs \in [a]. xs \text{ defined stream}) \forall z \in \mathbb{Z}. \\ \text{take } z \text{ } xs \text{ ++ drop } z \text{ } xs = xs$$

Proof by induction on the structure of xs and cases.

Proof of Lemma 6.4.1.12 (1)

Base case: Let $xs \equiv \perp$.

Case 1: Let $z \in \mathbb{Z}$, $z \leq 0$. Equational reasoning yields the desired equality:

$$\begin{aligned} & \text{take } z \text{ } xs \text{ } ++ \text{ drop } z \text{ } xs \\ = & \text{take } z \text{ } \perp \text{ } ++ \text{ drop } z \text{ } \perp \\ = & [] \text{ } ++ \perp \\ = & \perp \\ = & xs \end{aligned}$$

Case 2: Let $z \in \mathbb{Z}$, $z > 0$. Again, equational reasoning yields as desired:

$$\begin{aligned} & \text{take } z \text{ } xs \text{ } ++ \text{ drop } z \text{ } xs \\ = & \text{take } z \text{ } \perp \text{ } ++ \text{ drop } z \text{ } \perp \\ = & \perp \text{ } ++ \perp \\ = & \perp \\ = & xs \end{aligned}$$

Proof of Lemma 6.4.1.12 (2)

Inductive case: Let $xs \equiv (x:xs')$ be a defined PL-approximant, and $z \in \mathbb{Z}$. xs defined implies that x is defined and that xs' is a defined PL-approximant, too. By means of the **induction hypothesis (IH)**, we can assume for all $z \in \mathbb{Z}$ the equality $(\text{take } (z-1) \text{ } xs' ++ \text{drop } (z-1) \text{ } xs') = xs'$. This allows us to complete the proof as follows:

$$\begin{aligned} & \text{take } z \text{ } xs ++ \text{drop } z \text{ } xs \\ = & \text{take } z \text{ } (x:xs') ++ \text{drop } z \text{ } (x:xs') \\ = & x : (\text{take } (z-1) \text{ } xs' ++ \text{drop } (z-1) \text{ } xs') \\ \text{(IH)} \quad = & x : xs' \\ = & (x:xs') \\ = & xs \end{aligned}$$

□

Example 3: Induction on PL-Approximants

Lemma 6.4.1.13

We have:

$$(\forall \mathbf{xs} \in [\mathbf{a}]. \mathbf{xs} \text{ defined stream}).$$
$$\text{map } (f \ . \ g) \ \mathbf{xs} = (\text{map } f \ . \ \text{map } g) \ \mathbf{xs}$$

Proof by induction on the structure of \mathbf{xs} .

Proof of Lemma 6.4.1.13 (1)

Base case: Let $xs \equiv \perp$. Equational reasoning yields the desired equality:

$$\begin{aligned} & \text{map } (f \ . \ g) \ xs \\ &= \text{map } (f \ . \ g) \ \perp \\ (\text{Def. map, case exh.}) &= \perp \\ (\text{Def. map, case exh.}) &= \text{map } f \ \perp \\ (\text{Def. map, case exh.}) &= \text{map } f \ (\text{map } g \ \perp) \\ (\text{Def. } (.)) &= (\text{map } f \ . \ \text{map } g) \ \perp \\ &= (\text{map } f \ . \ \text{map } g) \ xs \end{aligned}$$

Proof of Lemma 6.4.1.13 (2)

Inductive case: Let $xs \equiv (x:xs')$ be a defined PL-approximant. Then x is defined and xs' is a defined PL-approximant, too. By means of the **induction hypothesis (IH)**, we can assume the equality $\text{map } (f . g) xs' = (\text{map } f . \text{map } g) xs'$. This allows us to complete the proof as follows:

$$\begin{aligned} & \text{map } (f . g) xs \\ &= \text{map } (f . g) (x:xs') \\ \text{(Def. map)} \quad &= ((f . g) x) : \text{map } (f . g) xs' \\ \text{(IH)} \quad &= ((f . g) x) : (\text{map } f . \text{map } g) xs' \\ \text{(2x Def. (.))} \quad &= f (g x) : (\text{map } f (\text{map } g xs')) \\ \text{(Def. map)} \quad &= \text{map } f (g x : \text{map } g xs') \\ \text{(Def. (.))} \quad &= \text{map } f (\text{map } g (x:xs')) \\ &= (\text{map } f . \text{map } g) (x:xs') \\ &= (\text{map } f . \text{map } g) xs \end{aligned}$$

□

Exercise 6.4.1.14

In [Definition 6.4.1.8](#), the set of PL-approximants is defined for defined streams.

1. Extend the notion of PL-approximant sets to streams with possibly undefined values.
2. Adapt the definition of the approximation order \sqsubseteq (cf. [Lemma 6.4.1.6](#)), the [Approximation Theorem 6.4.1.10](#), and the inductive principle for PL-approximant sets accordingly.
3. Do [Lemma 6.4.1.11](#), [6.4.1.12](#), and [6.4.1.13](#) hold for streams with possibly undefined values, too? Prove your claims or provide counter-examples.

Exercise 6.4.1.15

Consider below [Claim 6.3.2.2.3'](#), which extends the statement of [Lemma 6.3.2.2.3](#) to defined streams, and the subsequent attempt to prove it. At first sight, the 'proof' attempt looks quite reasonable. Nonetheless, there must be a flaw. Which one? Where and why?

Claim 6.3.2.2.3'

For all defined streams $xs :: [a]$, we have:

$$\text{reverse (reverse } xs) = xs$$

'Proof' attempt by induction on the structure of xs .

'Proof' Attempt of Claim 6.3.2.2.3' (1)

Base case: Let $xs \equiv \perp$. Equational reasoning yields the desired equality:

$$\begin{aligned} & \text{reverse (reverse xs)} \\ &= \text{reverse (reverse } \perp) \\ \text{(Def. reverse, case ex.)} &= \text{reverse } \perp \\ \text{(Def. reverse, case ex.)} &= \perp \\ &= xs \end{aligned}$$

'Proof' Attempt of Claim 6.3.2.2.3' (2)

Inductive case: Let $xs \equiv (x:xs')$, xs defined. This implies xs' and x are defined, too. By means of the **induction hypothesis (IH)**, we can thus assume **reverse (reverse xs') = xs'** . This allows us to complete the proof as follows:

$$\begin{aligned} \text{reverse (reverse } xs) &= \text{reverse (reverse (} x:xs')) \\ (\text{Def. reverse}) &= \text{reverse ((reverse } xs') ++ [x]) \\ (\text{L. 6.3.2.2.6(1)}) &= \text{reverse [x] ++ reverse (reverse } xs') \\ ([x] = x: [], \text{ IH}) &= \text{reverse (x : []) ++ } xs' \\ (\text{Def. reverse}) &= (\text{reverse [] ++ [x]) ++ } xs' \\ (\text{Def. reverse}) &= ([] ++ [x]) ++ } xs' \\ (\text{Def. (++)}) &= [x] ++ } xs' \\ ([x] = x: []) &= (x : []) ++ } xs' \\ (\text{Def. (++)}) &= x : ([] ++ } xs') \\ (\text{Def. (++)}) &= x:} xs' \\ &= xs \end{aligned}$$



Exercise 6.4.1.16

Recall that properties, which hold for (defined) **lists**

- **can hold**, e.g.,
 $\forall z \in \mathbb{Z}. \text{ take } n \text{ xs } ++ \text{ drop } n \text{ xs} = \text{xs}$
- but **need not hold**, e.g.,
 $\text{reverse} (\text{reverse xs}) = \text{xs}$

for (defined) **streams**.

Which of the **statements** of the **lemmas** in **Chapter 6.3.2**, **6.3.3**, and **6.4.1** hold for

1. **defined streams?**
2. **streams with possibly undefined elements?**

Prove your claims or provide counter-examples.

Chapter 6.4.2

Inductive Proofs on Haskell List and Stream Approximants

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

6.1

6.2

6.3

6.4

6.4.1

6.4.2

6.5

6.6

6.7

6.8

Part IV

Chap. 7

Chap. 8

Approximation Order on Lists, Part. Lists, Streams

Let $S_{(L,PL,St)} =_{df} \{s \mid s \text{ list or partial list or stream}\}$ be the set of lists, partial lists and streams.

Lemma 6.4.2.1 (Partial Order)

The relation \sqsubseteq on $S_{(L,PL,St)}$ defined by:

$$\begin{array}{l} \perp \\ [] \\ x : xs \end{array} \sqsubseteq \begin{array}{l} xs \\ xs \\ y : ys \end{array} \iff_{df} \begin{array}{l} xs = [] \\ x \equiv y \wedge xs \sqsubseteq ys \end{array}$$

is a partial order on $S_{(L,PL,St)}$, where \equiv denotes equality on list resp. stream entries.

Lemma 6.4.2.2 (Domain ($S_{(L,PL,St)}$))

$(S_{(L,PL,St)}, \sqsubseteq)$ with \sqsubseteq as in Lemma 6.4.2.1 is a domain with least element \perp and approximation order \sqsubseteq .

Partial Lists as List and Stream Approximants

Definition 6.4.2.3 (LPL-Approximants)

Let `xs` be a defined list or a defined stream. The **set of LPL-approximants** of `xs` is the set

$$\text{LPL-Approx}(\text{xs}) =_{df} \{ \text{approx } n \text{ xs} \mid n \in \mathbb{N}_0 \}$$

where

```
approx :: Integer -> [a] -> [a]
approx (n+1) []          = []
approx (n+1) (x:xs)     = x : approx n xs
```

Note: There are **LPL-approximants** built from the **undefined list** and others built from the empty list; all of them are of **finite length**.

Notes on approx

```
approx :: Integer -> [a] -> [a]
approx (n+1) []      = []
approx (n+1) (x:xs) = x : approx n xs
```

Pattern `n+1` matches only positive integers ≥ 1 . Thus:

1. `approx m ys ->> ys`,
if `m > len ys`.
2. `approx m ys ->> y0 : y1 : ... : ym-1 : ⊥`,
if `m ≤ len ys`
(i.e., `approx` will cause an error after generating the first `m` elements of `ys`).

Thus, `approx` being similar to `take'` used in [Definition 6.4.1.8](#) behaves differently when applied to lists (which, by definition, are built from the empty list, not the undefined list).

Examples: Applying approx

```
approx 0 [1,2] ->> ⊥
approx 1 [1,2] ->> approx (0+1) [1,2]
                ->> 1 : approx 0 [2]
                ->> 1 : ⊥
approx 2 [1,2] ->> approx (1+1) [1,2]
                ->> 1 : approx 1 [2]
                ->> 1 : approx (0+1) [2]
                ->> 1 : 2 : approx 0 []
                ->> 1 : 2 : ⊥
approx 3 [1,2] ->> approx (2+1) [1,2]
                ->> 1 : approx 2 [2]
                ->> 1 : approx (1+1) [2]
                ->> 1 : 2 : approx 1 []
                ->> 1 : 2 : approx (0+1) []
                ->> 1 : 2 : []
approx 7 [1,2..] ->> 1 : 2 : 3 : 4 : 5 : 6 : 7 : ⊥
```

Examples: PL-Approximants Sets

Lists:

$$LPL\text{-}Approx(\perp) = \{\perp\}$$

$$LPL\text{-}Approx(\square) = \{\perp, \square\}$$

$$LPL\text{-}Approx([1,2]) = \{\perp, 1:\perp, 1:2:\perp, 1:2:\square\}$$

Streams:

$$LPL\text{-}Approx([1..]) = \{\perp, 1:\perp, 1:2:\perp, 1:2:3:\perp, \dots\}$$

$$LPL\text{-}Approx([1,1..]) = \{\perp, 1:\perp, 1:1:\perp, 1:1:1:\perp, \dots\}$$

Main Result: Approximation

Lemma 6.4.2.4 (LPL-Approximants Chain)

The set $LPL-Approx(xs)$, xs a defined list or a defined stream, is a chain.

Theorem 6.4.2.5 (Approximation)

Let xs be a defined list or a defined stream. Then xs is equal to the least upper bound of its LPL-approximants set, its so-called **limit**:

$$xs = \bigsqcup LPL-Approx(xs) = \bigsqcup_{n=0}^{\infty} approx\ n\ xs$$

Proof Sketch of Theorem 6.4.2.5 for Lists

Let $xs \equiv (x_0 : x_1 : x_2 : \dots : x_{\text{len}(xs)-1} : [])$ be a defined list.

$$\begin{aligned} & \bigsqcup_{n=0}^{\infty} \text{approx } n \text{ } xs \\ &= \bigsqcup \left\{ \begin{array}{ll} \perp, & (n = 0) \\ x_0 : \perp, & (n = 1) \\ x_0 : x_1 : \perp, & (n = 2) \\ \dots & \\ x_0 : x_1 : \dots : x_{n-1} : \perp, & (n = \text{len}(xs)) \\ x_0 : x_1 : \dots : x_{n-1} : [], & (n = \text{len}(xs)+1) \\ x_0 : x_1 : \dots : x_{n-1} : [], & (n = \text{len}(xs)+2) \\ \dots & \end{array} \right\} \\ &= x_0 : x_1 : x_2 : \dots : x_{n-1} : [] \\ &= x_0 : x_1 : x_2 : \dots : x_{\text{len}(xs)-1} : [] \\ &\equiv xs \end{aligned}$$

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

6.1

6.2

6.3

6.4

6.4.1

6.4.2

6.5

6.6

6.7

6.8

Part IV

Chap. 7

Chap. 8

Proof Sketch of Theorem 6.4.2.5 for Streams

Let $xs \equiv (x_0 : x_1 : x_2 : \dots : x_n : \dots)$ be a defined stream.

$$\bigsqcup_{n=0}^{\infty} \text{approx } n \text{ } xs$$

$$\begin{aligned} &= \bigsqcup \left\{ \begin{array}{ll} \perp, & (n = 0) \\ x_0 : \perp, & (n = 1) \\ x_0 : x_1 : \perp, & (n = 2) \\ \dots & \\ x_0 : x_1 : \dots : x_{m-1} : \perp, & (n = m) \\ x_0 : x_1 : \dots : x_m : \perp, & (n = m+1) \\ x_0 : x_1 : \dots : x_{m+1} : \perp, & (n = m+2) \\ \dots & \end{array} \right\} \\ &= x_0 : x_1 : x_2 : \dots : x_n : \dots \\ &\equiv xs \end{aligned}$$

Inductive Proofs on LPL-Approximants Sets

...for proving 'admissible' properties of streams.

Let P be an equational property defined on LPL-approximants and streams.

A) Proof pattern for defined LPL-approximants:

1. Base case: Prove that $P(\perp)$ and $P(\square)$ are true.
2. Inductive case: Assuming that $P(xs)$ is true (induction hypothesis), prove that $P(x:xs)$ is true (induction step).

B) Proof pattern for LPL-approximants w/ possibly undefined values:

1. Base case: Prove that $P(\perp)$ and $P(\square)$ are true.
2. Inductive case: Assuming that $P(xs)$ is true (induction hypothesis), prove that $P(\perp:xs)$ and $P(x:xs)$, x a defined value, are true (induction step).

Exercise 6.4.2.6

Which of the statements of the lemmas in Chapter 6.3.2, 6.3.3, and 6.4.1 hold for

1. defined LPL-approximants?
2. LPL-approximants with possibly undefined values,?

Prove your claims or provide counter-examples.

Note

...the careful distinction between **defined** and **undefined values**, between **finite lists** and **finite partial lists**, and **infinite streams** needs to be done analogously for every

- **inductively defined Haskell data type**

such as e.g. trees (cf. **Chapter 6.3.1**). **Lists**, **partial lists**, and **streams** just happen to be three most important representatives of inductively defined data structures.

Doing this results in corresponding **induction principles** for other **inductively defined Haskell data types** tailored for **defined** and **partial**, for **finite** and **infinite values** with and without **possibly undefined** values, etc.

Chapter 6.5

Proving Equality of Streams

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

6.1

6.2

6.3

6.4

6.5

6.5.1

6.5.2

6.6

6.7

6.8

Part IV

Chap. 7

Chap. 8

Chapter 6.5.1

Approximation

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

6.1

6.2

6.3

6.4

6.5

6.5.1

6.5.2

6.6

6.7

6.8

Part IV

Chap. 7

Chap. 8

Approximants, Approximants Sets

...allow to reduce proving the equality of streams to proving

1. the equality of sets (cf. [Approximation Theorem 6.5.1.7](#))
2. equivalent statements amenable to mathematical induction (cf. [Approximation Theorem 6.5.1.8](#))

and provide this way an important [proof principle](#) for proving the [equality of streams](#).

L-Approximants of Defined Lists and Streams

Definition 6.5.1.1 (L-Approximants)

Let xs be a defined list or a defined stream. The set of L-approximants of xs is the set

$$L\text{-Approx}(xs) =_{df} \{ \text{take } n \text{ } xs \mid n \in \mathbb{N}_0 \}, \text{ where}$$

$\text{take} :: \text{Int} \rightarrow [a] \rightarrow [a]$

$\text{take } n _ \mid n \leq 0 = []$

$\text{take } _ [] = []$

$\text{take } n (x:xs) = x : \text{take } (n-1) \text{ } xs$

Note, L-approximants are built from the empty list (not the undefined list); every L-approximant is finite.

Examples:

– $L\text{-Approx}([]) = \{ [] \}$

– $L\text{-Approx}([1,2,3]) = \{ [], 1: [], 1:2: [], 1:2:3: [] \}$

– $L\text{-Approx}([1..]) = \{ [], 1: [], 1:2: [], 1:2:3: [], \dots \}$

Finiteness, Infinity of Sequences

...in terms of L -approximants sets.

Definition 6.5.1.2 (Finite, Infinite Sequences)

A sequence of defined values xs is

1. **finite** (i.e., a *list*), if $L\text{-Approx}(xs)$ is finite.
2. **infinite** (i.e., a *stream*), if $L\text{-Approx}(xs)$ is infinite.

Lemma 6.5.1.3 (Finite, Infinite Sequences)

A sequence of defined values xs is

1. **finite**, if: $\exists m \in \mathbb{N}. (\forall n \in \mathbb{N}. n \geq m). \text{take } n \text{ } xs = \text{take } (n+1) \text{ } xs$
2. **infinite**, if: $\forall n \in \mathbb{N}. \text{take } n \text{ } xs \neq \text{take } (n+1) \text{ } xs$

Corollary 6.5.1.4 (Finite Sequences)

A sequence of defined values xs is **finite**, if:

$$\exists m \in \mathbb{N}. (\forall n \in \mathbb{N}. n \geq m). \text{take } m \text{ } xs = \text{take } (n+1) \text{ } xs$$

Equality of Sequences and Streams

...in terms of L-approximant sets.

Definition 6.5.1.5 (Equality of Sequences)

Let xs and ys be two sequences of defined values. xs and ys are **equal**, if their sets of L-approximants are equal:

$$\begin{aligned} L\text{-Approx}(xs) &= \{ \text{take } n \text{ } xs \mid n \in \mathbb{N} \} \\ &= \{ \text{take } n \text{ } ys \mid n \in \mathbb{N} \} = L\text{-Approx}(ys) \end{aligned}$$

Lemma 6.5.1.6 (Equality of Sequences)

Let xs and ys be two sequences of defined values. xs and ys are **equal**, if for every natural number their L-approximants are equal:

$$\forall n \in \mathbb{N}. \text{take } n \text{ } xs = \text{take } n \text{ } ys$$

Main Results: Stream Equality by Set Equality

...reducing the proof of *stream equality* to proving *set equality*.

Theorem 6.5.1.7 (Stream Equality, Approximation 1)

Let xs, ys be defined streams. Then the following statements are equivalent:

1. $xs = ys$
2. $LPL\text{-}Approx(xs) = LPL\text{-}Approx(ys)$
3. $PL\text{-}Approx(xs) = PL\text{-}Approx(ys)$
4. $L\text{-}Approx(xs) = L\text{-}Approx(ys)$

Main Results: Stream Equality by Nat. Induct.

...reducing the proof of stream equality to an equivalent statement amenable to natural (or: mathematical) induction.

Theorem 6.5.1.8 (Stream Equality, Approximation 2)

Let xs, ys be defined streams. Then the following statements are equivalent:

1. $xs = ys$
2. $\forall n \in \mathbb{N}. \text{approx } n \text{ } xs = \text{approx } n \text{ } ys$
3. $\forall n \in \mathbb{N}. \text{take}' \ n \text{ } xs = \text{take}' \ n \text{ } ys$
4. $\forall n \in \mathbb{N}. \text{take } n \text{ } xs = \text{take } n \text{ } ys$
5. $\forall n \in \mathbb{N}_0. xs!!n = ys!!n$

Note: Proving stream equality is usually technically more convenient using Theorem 6.5.1.8(5) than any of the statements of Theorem 6.5.1.7.

Example: Applying Theorem 6.5.1.8

Consider the `factorial` function:

```
fac :: Int -> Int
fac 0 = 1
fac n = n * fac (n-1)
```

and the two stream functions `facs_mp` and `facs_zw`:

```
facs_mp = map fac [0..]
facs_zw = 1 : zipWith (*) [1..] facs_zw
```

which generate the `stream` of `factorials`: `1,1,2,6,24,...`

According to [Theorem 6.5.1.8\(5\)](#), proving the equality of `facs_mp` and `facs_zw` boils down to proving [Lemma 6.5.1.9](#), which we prove by `natural induction`:

Lemma 6.5.1.9

$$\forall n \in \mathbb{N}_0. \text{facs_mp} !! n = \text{facs_zw} !! n$$

Proof by Lemma 6.5.1.9 (1)

Base case: Let $n=0$. Equational reasoning yields the desired equality in this case:

$$\begin{aligned} & \text{facs_mp!!n} \\ (n = 0) &= \text{facs_mp!!0} \\ (\text{Def. facs_mp}) &= (\text{map fac } [0..])!!0 \\ (\text{L. 6.5.1.10(1)}) &= \text{fac } ([0..]!!0) \\ &= \text{fac } 0 \\ (\text{Def. fac}) &= 1 \\ (\text{Def. (!!)}) &= (1 : \text{zipWith } (*) [1..] \text{facs_zw})!!0 \\ (\text{Def. facs_zw}) &= \text{facs_zw!!0} \\ (n = 0) &= \text{facs_zw!!n} \end{aligned}$$

Proof by Lemma 6.5.1.9 (2)

Inductive case: Let $n \in \mathbb{N}_0$. By means of the **induction hypothesis (IH)**, we can assume $\mathbf{facs_mp}!!n = \mathbf{facs_zw}!!n$. As desired we get:

$\mathbf{facs_mp}!!(n+1)$

(Def. $\mathbf{facs_mp}$) = $(\mathbf{map\ fac\ [0..]})!!(n+1)$

(L. 6.5.1.10(1)) = $\mathbf{fac\ ([0..]!!(n+1))}$

(Def. $[0..]$, $(!!)$) = $\mathbf{fac\ (n+1)}$

(Def. \mathbf{fac}) = $(n+1) * \mathbf{fac\ n}$

(L. 6.5.1.10(3)) = $(n+1) * (\mathbf{facs_mp}!!n)$

(IH) = $(n+1) * (\mathbf{facs_zw}!!n)$

(Def. $(!!)$) = $([1..]!!n) * (\mathbf{facs_zw}!!n)$

(Def. $(*)$) = $(*) ([1..]!!n) (\mathbf{facs_zw}!!n)$

(L. 6.5.1.10(2)) = $(\mathbf{zipWith\ (*)\ [1..]\ facs_zw})!!n$

(Def. $(!!)$) = $(1 : \mathbf{zipWith\ (*)\ [1..]\ facs_zw})!!(n+1)$

(Def. $\mathbf{facs_zw}$) = $\mathbf{facs_zw}!!(n+1)$

Supporting Statement

Lemma 6.5.1.10

For all natural numbers $n \in \mathbb{N}_0$, we have:

1. $(\text{map } f \text{ } xs)!!n = f (xs!!n)$
2. $(\text{zipWith } g \text{ } xs \text{ } ys)!!n = g (xs!!n) (ys!!n)$
3. $\text{fac } n = \text{facs_mp}!!n$

Proof. Homework.

Exercise 6.5.1.11: Applying Theorem 6.5.1.8

Consider the two definitions `fibs_memo` and `fibs_zw`:

```
fibs_memo = [fibm x | x <- [0..]]
fibm 0    = 0
fibm 1    = 1
fibm n    = fibs_memo!!(n-1) + fibs_memo!!(n-2)
fibs_zw   = 0 : 1 : zipWith (+) fibs_zw (tail fibs_zw)
```

which generate the stream of Fibonacci numbers:

0,1,1,2,3,5,8,13,21,34,55,89,...

Prove by means of natural (mathematical) induction:

Lemma 6.5.1.12

$$\forall n \in \mathbb{N}_0. \text{fibs_memo}!!n = \text{fibs_zw}!!n$$

...to obtain by means of Theorem 6.5.1.8(5) as corollary the equality of `fibs_memo` and `fibs_zw`.

Exercise 6.5.1.13: Applying Theorem 6.5.1.7

Repeat [Exercise 6.5.1.11](#) using [Theorem 6.5.1.7](#) to prove the equality of `fibs_memo` and `fibs_zw`.

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

6.1

6.2

6.3

6.4

6.5

6.5.1

6.5.2

6.6

6.7

6.8

Part IV

Chap. 7

Chap. 8

Exercise 6.5.1.14

Why can't we build an inductive proof principle for streams on only **L-approximants** (cf. **Definition 6.5.1.1**)? Compared to the inductive proof principles of **Chapter 6.4**, this would effectively mean to drop or replace proving $P(\perp)$ by proving $P(\square)$ in the base cases of the inductive proof patterns based on **PL-** and **LPL-approximants** of **Chapter 6.4.1** and **Chapter 6.4.2**, respectively. Think e.g. on the consequences of 'proving' a property like 'the reverse of the reverse of a stream is the stream itself.'

Chapter 6.5.2

Coinduction

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

6.1

6.2

6.3

6.4

6.5

6.5.1

6.5.2

6.6

6.7

6.8

Part IV

Chap. 7

Chap. 8

Proof by Coinduction

...another useful principle for proving equality of infinite objects such as streams which

- complements the principle of proof by approximation of Chapter 6.5.1.
- reduces proving equality of two objects to proving they exhibit the same 'observational behaviour.'

For streams, this boils down to proving

- the heads of the streams are the same.
- their tails exhibit the same 'observational behaviour.'

Equality of Streams

...let

- $[A]$ denote the set of streams over a set of elements A .
- $f, g \in [A]$ be written as $f = [f_0, f_1, f_2, f_3, f_4, f_5, \dots]$ and $g = [g_0, g_1, g_2, g_3, g_4, g_5, \dots]$, respectively.

Definition 6.5.2.1 (Equality of Streams)

$f, g \in [A]$ are **equal** iff $\forall i \in \mathbb{N}_0. f_i = g_i$, i.e., f and g have the same 'observational behaviour.'

...in accordance with [Theorem 6.5.1.8](#).

Reducing Equality of Streams

...to their **bisimilarity**.

This requires to introduce:

- **Labelled transition systems (LTS)** for stream representation
- **Stream bisimulation relations** for capturing the notion of 'same' behaviour of streams

and some supporting notions:

- **Expansions** of LTS states
- **Bisimilar** states

Labelled Transition Systems

Definition 6.5.2.2 (Labeled Transition System)

A labelled transition system (LTS) is a triple (Q, A, T) with

- Q a set of states.
- A a set of action labels.
- $T \subseteq Q \times A \times Q$ a ternary transition relation.

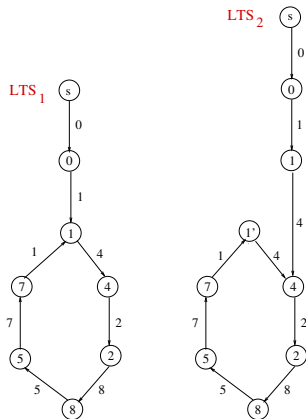
Note: For $(q, a, p) \in T$ we write more conveniently: $q \xrightarrow{a} p$.

Example: Representing Streams as LTSs

The decimal representation of $\frac{1}{7}$ has numerous representations as streams of digits, e.g.:

– $0.\overline{142857}$, $0.\overline{1428571}$, $0.\overline{14285714}$, $0.142857142857142, \dots$

LTS_1 , LTS_2 are LTS representations of the 2nd and 3rd one:



Expansion of LTS States

Let (Q, A, T) be an LTS, and $q \in Q$.

Definition 6.5.2.3 (Expansion of an LTS State)

1. A **finite expansion** of q is a **finite sequence of actions** $[a_0, a_1, a_2, a_3, \dots, a_n]$ such that

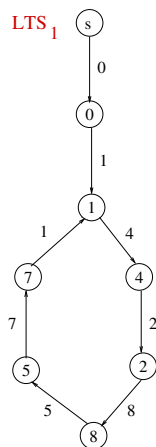
$$(\forall i \in \mathbb{N}_0. i \leq n). \exists q_i, q_{i+1} \in Q. q_0 = q \wedge q_i \xrightarrow{a_i} q_{i+1}.$$

2. An **infinite expansion** of q is an **infinite sequence of actions** $[a_0, a_1, a_2, a_3, \dots]$ such that

$$\forall i \in \mathbb{N}_0. \exists q_i, q_{i+1} \in Q. q_0 = q \wedge q_i \xrightarrow{a_i} q_{i+1}.$$

Example: Expansion of Digit Stream States

Consider LTS_1 representing digit stream $0.1\overline{428571}$:



The (unique) infinite expansion of state (i.e., node)

- s is $01\overline{428571}$, 0 is $1\overline{428571}$, 1 is $4\overline{28571}$, 2 is $8\overline{57142}$,...

Bisimulation Relations, Bisimilar States

Let (Q, A, T) be an LTS, let $p, q \in Q$.

Definition 6.5.2.4 ((Greatest) Bisimulation Relation)

A **bisimulation** on (Q, A, T) is a binary relation R on Q , which satisfies: If $q R p$ and $a \in A$ then:

- $q \xrightarrow{a} q' \Rightarrow \exists p' \in Q. p \xrightarrow{a} p' \wedge q' R p'$
- $p \xrightarrow{a} p' \Rightarrow \exists q' \in Q. q \xrightarrow{a} q' \wedge q' R p'$

The **largest bisimulation** on Q (wrt \subseteq) is denoted by \sim .

Definition 6.5.2.5 (Bisimilar States)

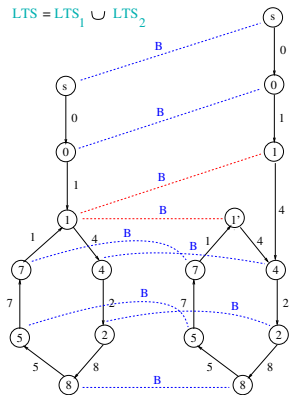
p and q are called **bisimilar**, if there is a bisimulation R on Q with $q R p$.

Example: A Bisimulation for Digit Streams

Consider $LTS = (Q, A, T)$ defined as union of LTS_1, LTS_2 .

We define relation B on Q as follows:

$\forall q, q' \in Q. q B q'$ iff q, q' have the same infinite expansion



Note: B is the largest bisimulation on Q , i.e.: $B = \sim$.

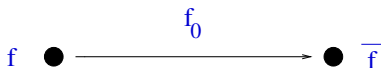
Streams as Labeled Transition Systems

...for $f = [f_0, f_1, f_2, f_3, f_4, \dots] \in [A]$ a stream, let

- f_0 denote the head
- \bar{f} denote the tail

of f , i.e., $f = f_0 : \bar{f}$.

Using this notation, f is represented by the below labelled transition system (which unfolds f partially):



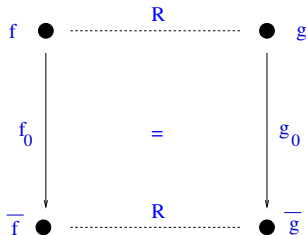
LTS representation of f

Stream Bisimulation

Definition 6.5.2.6 (Stream Bisimulation)

A **stream bisimulation** on $[A]$ is a binary relation R on the set of streams $[A]$, which satisfies:

$$\forall f, g \in [A]. f R g \Rightarrow f_0 = g_0 \wedge \bar{f} R \bar{g}$$



Let \sim denote the **largest stream bisimulation** on $[A]$.

Reducing Stream Equality

...to largest stream bisimulation.

Let $f = [f_0, f_1, f_2, f_3, f_4, \dots]$, $g = [g_0, g_1, g_2, g_3, g_4, \dots] \in [A]$ be two streams with

$$f \xrightarrow{f_0} \bar{f}, \quad g \xrightarrow{g_0} \bar{g}.$$

Then:

Theorem 6.5.2.7 (Stream Equality as Stream Bisim.)

f and g are equal iff $f \sim g$ (i.e., $f_0 = g_0$ and $\bar{f} \sim \bar{g}$).

Reducing Stream Equality

...further to stream bisimulation.

Since \sim is the largest stream bisimulation, we get:

Lemma 6.5.2.8

$$f \sim g \Leftrightarrow \exists B. B \text{ stream bisimulation on } [A] \wedge f B g$$

Together, Theorem 6.4.3.7 and Lemma 6.4.3.8 imply:

Corollary 6.5.2.9

f and g are equal iff

$$\exists B. B \text{ stream bisimulation on } [A] \wedge f B g$$

The Coinductive Proof Pattern

...using [Corollary 6.5.2.9](#), proving the equality of two streams f and g of $[A]$ requires:

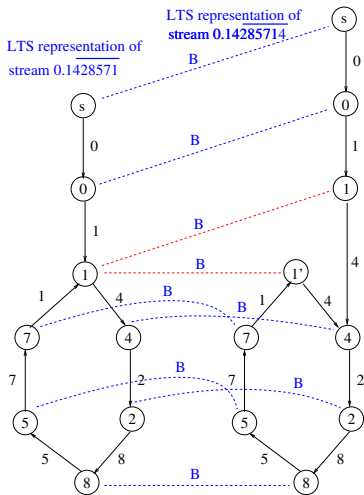
1. Finding a relation B on $[A]$.
2. Proving that B is a [stream bisimulation](#) and $f B g$.

...considering [Haskell streams](#), this means proving the equality of two Haskell streams xs and ys requires:

1. Finding a relation B on the set of [Haskell streams](#).
2. Proving that B is a [stream bisimulation](#) and $xs B ys$.

Example: Stream Bisimulation $B \subseteq \sim$

...for the two streams $0.1\overline{428571}$ and $0.14\overline{285714}$:



... $0.1\overline{428571}$, $0.14\overline{285714}$ are stream bisimilar and hence equal.

Chapter 6.6

Fixed Point Induction

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

6.1

6.2

6.3

6.4

6.5

6.6

6.7

6.8

Part IV

Chap. 7

Chap. 8

Chap. 9

Fixed Point Induction

...a useful **proof principle** allowing us to prove **properties** of the
– **least fixed point** of **continuous functions**

on **complete partial orders** or stronger **complete lattices**, which are both specific **partially ordered sets** (refer to **Appendix A** for definitions of terms, if required).

Admissible Predicates

Let (C, \sqsubseteq) be a complete partial order (CPO) (or: domain), and ψ be a predicate on C , i.e., $\psi : C \rightarrow IB$.

Definition 6.6.1 (Admissible Predicate)

ψ is called **admissible** iff for every chain $D \subseteq C$ holds:

$$(\forall d \in D. \psi(d)) \Rightarrow \psi(\bigsqcup D)$$

Lemma 6.6.2

ψ is admissible, if it is expressible as an equation.

Example: Streams, Sequences of Approximants

Recalling that $(S_{(PL,St)}, \sqsubseteq)$ with

- $S_{(PL,St)}$: Set of partial lists and streams
- \sqsubseteq : Approximation order on $S_{(PL,St)}$ (cf. Lemma 6.4.1.6)

is a domain (or: complete partial order) (cf. Lemma 6.4.1.7), we get:

Corollary 6.6.3

Let ψ be a predicate on the set of partial lists and streams $S_{(PL,St)}$ expressible as an equation, let s be a stream, and $S' \subseteq S$ the infinite chain of its PL-approximants (cf. Definition 6.4.1.8) with $\bigsqcup S' = s$. Then:

$$(\forall s' \in S'. \psi(s')) \Rightarrow \psi(\bigsqcup S') \quad (\Leftrightarrow \psi(s))$$

Monotonic and Continuous Functions on CPOs

Let (C, \sqsubseteq_C) and (D, \sqsubseteq_D) be CPOs, and let $f \in [C \rightarrow D]$ be a map from C to D .

Definition 6.6.4 (Monotonic, Continuous Maps)

f is called

1. **monotonic** (or: **order preserving**) iff

$$\forall c, c' \in C. c \sqsubseteq_C c' \Rightarrow f(c) \sqsubseteq_D f(c')$$

(Preservation of the ordering of elements)

2. **continuous** iff f is monotonic and

$$(\forall C' \subseteq C. C' \neq \emptyset \wedge C' \text{ chain}). f(\bigsqcup_C C') =_D \bigsqcup_D f(C')$$

(Preservation of least upper bounds)

Fixed Points, Least Fixed Points

...of continuous functions on complete partial orders (CPOs).

Definition 6.6.5 (Fixed Point, Least Fixed Point)

Let (C, \sqsubseteq) be a complete partial order, let $f \in [C \xrightarrow{\text{con}} C]$ be a continuous function on C , and let $c \in C$ be an element of C .

Then:

1. c is a **fixed point** of f iff $f(c) = c$.
2. c is the **least fixed point** of f , denoted by μf ,
iff $\forall d \in C. f(d) = d \Rightarrow c \sqsubseteq d$

Note: The Fixed Point Theorem A.5.1.3 of Knaster, Tarski, and Kleene ensures the existence of least fixed points of continuous functions on CPOs.

Fixed Point Induction

...the general pattern of fixed point induction:

Theorem 6.6.6 (Fixed Point Induction)

Let (C, \sqsubseteq) be a complete partial order (CPO), let $f : C \rightarrow C$ be a continuous function on C , and let $\psi : C \rightarrow IB$ be an admissible predicate on C . Then:

$$(\forall c \in C. \psi(c) \Rightarrow \psi(f(c))) \Rightarrow \psi(\mu f)$$

where μf denotes the least fixed point of f .

Proof Sketch of Theorem 6.6.6

- The empty set $\emptyset \subseteq C$ is (trivially) a chain.
- Since C is a CPO, $\bigsqcup \emptyset \text{ exists} = \perp_C$ with \perp_C the least element of C .
- ψ admissible yields $\psi(\perp_C)$.
(Note that $(\forall d \in \emptyset. \psi(d))$ holds trivially; ψ admissible thus implies $\psi(\bigsqcup \emptyset) = \psi(\perp_C) = \text{true}$.)
- Using the assumptions of Theorem 6.4.4.6, we can prove by induction on $n \in \mathbb{N}_0$:
 - $D =_{df} \{f^n(\perp_C) \mid n \in \mathbb{N}_0\} \subseteq C$ is a chain.
 - $\forall n \in \mathbb{N}_0. \psi(f^n(\perp_C))$.
- D chain, $\forall d \in D. \psi(d)$, ψ admissible, yields $\psi(\bigsqcup D)$.
- Last but not least, Fixed Point Theorem A.5.1.3 (Knafter, Tarski, Kleene) yields $\mu f = \bigsqcup D$.
- Thus, we obtain $\psi(\mu f)$ completing the proof.

Chapter 6.7

Verified Programming, Verification Tools

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

6.1

6.2

6.3

6.4

6.5

6.6

6.7

6.7.1

6.7.2

6.8

Part IV

Chap. 7

Chap. 8

Chapter 6.7.1

Correctness by Construction

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

6.1

6.2

6.3

6.4

6.5

6.6

6.7

6.7.1

6.7.2

6.8

Part IV

Chap. 7

Chap. 8

Correctness by Construction

...strives for ensuring correctness of a program on the fly of developing it by proving the result of every step of the development process correct.

Conceptually, [correctness by construction](#) is an

- ▶ *a priori* (or: *on-the-fly*) approach.

This is dual to [testing](#) and [verification](#), which conceptually are

- ▶ *a posteriori* approaches

as they are applied to a program [after](#) its development is [finished](#).

Techniques for Correctness by Construction

...in principle, every proof technique can be made use of by approaches aiming at correctness by construction, among these

- (inductive) proof principles (cf. Chapter 6)
- equational reasoning, sometimes also called proof by program calculation (cf. Chapter 4).

Particularly important, however, are approaches based on

- transformation rules

which are proven correct and ensure equivalence of the program they are applied to and the one resulting from them.

Example: Functional Pearls

...developing **functional pearls** starting with a program being

- **obviously correct** (but usually inefficient)

by a sequence of **transformation steps** into a program being

- still **correct** and (hopefully) more **efficient**

where (ideally) **every** transformation step is **proved correct** (cp. **Chapter 4**), can be considered an approach in the spirit of

- **correctness by construction**.

Chapter 6.7.2

Provers, Proof-Assistents, Verified Programming

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

6.1

6.2

6.3

6.4

6.5

6.6

6.7

6.7.1

6.7.2

6.8

Part IV

Chap. 7

Chap. 8

Provers, Proof-Assistants, Verified Prog. (1)

Provers, proof-assistants for verifying

- ▶ **Equational properties** of functional programs (Sonnex et al., TACAS 2012)
 - Tool **Zeno**: Proof search is based on induction and equality reasoning which are driven by syntactic heuristics.
- ▶ **First-order and call-by-value recursive functional programs** (Suter et al., SAS 2011)
 - Tool **Leon**: Based on extending SMT to recursive programs.
- ▶ **Higher-order functional programs** (Unno et al., POPL 2013)
 - Tool **MoChi-X**: Prototype implementation of a type inference algorithm as extension of the software model checker **MoChi** (Kobayashi et al., PLDI 2011).

Provers, Proof-Assistants, Verified Prog. (2)

- ▶ **Lazy Haskell** (Mitchell et al., Haskell 2008)
 - Tool **Catch**: Based on static analysis; can prove absence of pattern matching failures; evaluated on ‘real’ programs.
- ▶ ...

Language integrated approaches:

- ▶ Programming by **contracts** (Vytiniotis et al., POPL 2013)
- ▶ Verified functional programming in **Agda** (see next slide)
- ▶ ...

Verified Functional Programming in Agda



Aaron Stump. Verified Functional Programming in Agda. ACM Books Series, No. 9, 2016.

...a text snippet from the book:

'Agda is an advanced programming language based on Type Theory. Agda's type system is expressive enough to support full functional verification of programs, in two styles.

In external verification, we write pure functional programs and then write proofs of properties about them. The proofs are separate external artifacts, typically using structural induction.

In internal verification, we specify properties of programs through rich types for the programs themselves. This often necessitates including proofs inside code, to show the type checker that the specified properties hold.

The power to prove properties of programs in these two styles is a profound addition to the practice of programming, giving programmers the power to guarantee the absence of bugs, and thus improve the quality of software more than previously possible.'

Chapter 6.8

References, Further Reading

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

6.1

6.2

6.3

6.4

6.5

6.6

6.7

6.8






Part IV

Chap. 7






Chap. 8

Chap. 9




Chapter 6: Further Reading (1)

-  Martin Aigner, Günter M. Ziegler. *Proofs from the Book*. Springer-V., 4th edition, 2010.
-  André Arnold, Irène Guessarian. *Mathematics for Computer Science*. Prentice Hall, 1996.
-  Roland Backhouse, Roy Crole, Jerimy R. Gibbons (Eds.). *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*. International Summer School and Workshop, Oxford, UK, April 10-14, 2000, Revised Lectures, Springer-V., LNCS 2297, 2002.
-  Falk Bartels. *Generalized Coinduction*. Electronic Notes in Theoretical Computer Science 44:1, 21 pages, 2001.
<http://www.elsevier.nl/locate/entcs/volume44.html>
-  Falk Bartels. *Generalized Coinduction*. Journal of Mathematical Structures in Computer Science 13(2):321-348, 2003.





Chapter 6: Further Reading (2)

-  Richard Bird. *Algebraic Identities for Program Calculation*. Computer Journal 32(2):122-126, 1989.
-  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.
-  Richard Bird. *Thinking Functionally with Haskell*. Cambridge University Press, 2015. (Chapter 6, Proofs)
-  Richard Bird, Philip Wadler. *An Introduction to Functional Programming*. Prentice Hall, 1988. (Chapter 5, Induction and Recursion)




Chapter 6: Further Reading (3)

-  Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 18, Programme verifizieren und testen)
-  Matthias Blume, David McAllester. *Sound and Complete Models of Contracts*. *Journal of Functional Programming* 16(4-5):375-414, 2006.
-  Manuel Chakravarty, Gabriele Keller. *Einführung in die Programmierung mit Haskell*. Pearson Studium, 2004. (Kapitel 9.1.2, Induktion; Kapitel 9.1.3, Strukturelle Induktion; Kapitel 9.2, Mit Lemmata und Generalisierung arbeiten; Kapitel 9.3, Programmherleitung)





Chapter 6: Further Reading (4)

-  Roderick Chapman. *Correctness by Construction: A Manifesto for High Integrity Software*. In Proceedings of the 10th Australian Workshop on Safety Critical Systems and Software, Vol. 55, 43-46, 2006.
-  Werner Damm, Bernhard Josko. *A Sound and Relatively* Complete Hoare-Logic for a Language with Higher Type Procedures*. Acta Informatica 20:59-101, 1983.
-  Antonie J.T. Davie. *An Introduction to Functional Programming Systems using Haskell*. Cambridge University Press, 1992. (Chapter 9, Correctness)
-  Henning Dierks, Michael Schenke. *A Unifying Framework for Correct Program Construction*. In Proceedings of the 4th International Conference on the Mathematics of Program Construction (MPC'98). Springer-V., LNCS 1422, 122-150, 1998.




Chapter 6: Further Reading (5)

-  Kees Doets, Jan van Eijck. *The Haskell Road to Logic, Maths and Programming*. Texts in Computing, Vol. 4, King's College, UK, 2004. (Chapter 3, The Use of Logic: Proof – Proof Style, Proof Recipes, Strategic (Proof) Guidelines; Chapter 7, Induction and Recursion; Chapter 10, Corecursion – Proof by Approximation, Proof by Coinduction; Chapter 11.1, More on Mathematical Induction)
-  Andreas Goerdt. *A Hoare Calculus for Functions defined by Recursion on Higher Types*. In Proceedings of the Conference on Logic of Programs, Springer-V, LNCS 193, 106-117, 1985.
-  Anthony Hall, Roderick Chapman. *Correctness by Construction: Developing a Commercial Secure System*. IEEE Software 19(1):18-25, 2002.




Chapter 6: Further Reading (6)

-  Charles A.R. Hoare. *The Ideal of Program Correctness*. The Computer Journal 50(3):254-260, 2007.
-  Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000. (Chapter 11, Proof by Induction; Chapter 14.6, Inductive Properties of Infinite Lists)
-  Chung-Kil Hur, Georg Neis, Derek Dreyer, Viktor Vafeiadis. *The Power of Parameterization in Coinductive Proofs*. In Conference Record of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2013), 193-205, 2013.
-  Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2nd edition, 2016. (Chapter 16, Reasoning about programs)





Chapter 6: Further Reading (7)

-  Bart Jacobs, Jan Rutten. *A Tutorial on (Co)algebras and (Co)induction*. EATCS Bulletin 62:222-259, 1997.
-  Ranjit Jhala, Rupak Majumdar, Andrey Rybalchenko. *HMC: Verifying Functional Programs using Abstract Interpreters*. In Proceedings of the 23rd International Conference on Computer Aided Verification (CAV 2011), Springer-V., LNCS 6806, 470-485, 2011.
-  Steve King, Jonathan Hammond, Roderick Chapman, Andy Pryor. *Is Proof More Cost-Effective than Testing?* IEEE Transactions on Software Engineering 26(8):675-686, 2000.




Chapter 6: Further Reading (8)

-  Naoki Kobayashi, Ryosuke Sato, Hiroshi Unno. *Predicate Abstraction and CEGAR for Higher-Order Model Checking*. In Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2011), 222-233, 2011.
-  Derrick G. Kourie, Bruce W. Watson. *The Correctness-by-Construction Approach to Programming*. Springer-V., 2012.
-  Marina Lenisa. *From Set-theoretic Coinduction to Coalgebraic Coinduction: Some Results, Some Problems*. Electronic Notes in Theoretical Computer Science 19:2-22, 1999.





Chapter 6: Further Reading (9)

-  David Makinson. *Sets, Logic and Maths for Computing*. Springer-V., 2008. (Chapter 4, Recycling Outputs as Inputs: Induction and Recursion; Chapter 4.1, What are Induction and Recursion? Chapter 4.6, Structural Recursion and Induction; Chapter 4.7, Recursion and Induction on Well-Founded Sets)
-  Robin Milner. *Communications and Concurrency*. Prentice Hall, 1989.
-  Robin Milner. *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge University Press, 1999.
-  Neil Mitchell, Colin Runciman. *Not all Patterns, but enough: An Automated Verifier for Partial but Succifient Pattern Matching*. In Proceedings of the 1st ACM SIG-PLAN Symposium on Haskell (Haskell 2008), 49-60, 2008.





Chapter 6: Further Reading (10)

-  Flemming Nielson, Hanne Riis Nielson, Chris Hankin. *Principles of Program Analysis*. 2nd edition, Springer-V., 2005. (Appendix B, Induction and Coinduction)
-  Hanne Riis Nielson, Flemming Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, 1992. Chapter 1, Introduction – Structural Induction; Chapter 6, Axiomatic Program Verification – Fixed Point Induction)
-  Hanne Riis Nielson, Flemming Nielson. *Semantics with Applications: An Appetizer*. Springer-V., 2007. Chapter 1, Introduction – Structural Induction; Chapter 9, Axiomatic Program Verification – Fixed Point Induction)




Chapter 6: Further Reading (11)

-  Lawrence C. Paulson. *Logic and Computation – Interactive Proof with Cambridge LCF*. Cambridge University Press, 1987. (Chapter 4, Structural Induction; Chapter 10, Sample Proofs (with Cambridge LCF))
-  Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003. (Kapitel 11.4.1, Über wohlfundierte Ordnungen; Kapitel 11.4.2, Wie beweist man Terminierung?)
-  Peter Pepper, Petra Hofstedt. *Funktionale Programmierung*. Springer-V., 2006. (Kapitel 10, Beispiel: Berechnung von Fixpunkten)
-  Jan Rutten. *Behavioural Differential Equations: A Coinductive Calculus of Streams, Automata, and Power Series*. Theoretical Computer Science 308:1-53, 2003.




Chapter 6: Further Reading (12)

-  Davide Sangiorgi. *On the Bisimulation Proof Method*. Journal of Mathematical Structures in Computer Science 8:447-479, 1998.
-  Davide Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2011.
-  Davide Sangiorgi, Jan Rutten (Eds). *Advanced Topics in Bisimulation and Coinduction*. Cambridge Tracts in Theoretical Computer Science, Vol. 52, Cambridge University Press, 2011.
-  William Sonnex, Sophia Drossopoulou, Susan Eisenbach. *Zeno: An Automated Prover for Properties of Recursive Data Structures*. In Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2012), Springer-V., LNCS 7214, 407-421, 2012.




Chapter 6: Further Reading (13)

-  Aaron Stump. *Verified Functional Programming in Agda*. ACM Books Series, No. 9, 2016.
-  Philippe Suter, Ali Sinan Köksal, Viktor Kuncak. *Satisfiability Modulo Recursive Programs*. In Proceedings of the 18th International Conference on Static Analysis (SAS 2011), Springer-V., LNCS 6887, 298-315, 2011.
-  Bernhard Steffen, Oliver Rüthing, Malte Isberner. *Grundlagen der höheren Informatik: Induktives Vorgehen*. Springer-V., 2014. (Chapter 4, Induktives Definieren; Chapter 5, Induktives Beweisen; Chapter 6, Induktives Vorgehen: Potential und Grenzen)




Chapter 6: Further Reading (14)

-  Bernhard Steffen, Oliver Rüthing, Michael Huth. *Mathematical Foundations of Advanced Informatics: Inductive Approaches*. Springer-V., 2018. (Chapter 4, Inductive Definitions; Chapter 5, Inductive Proofs; Chapter 6, Inductive Approach: Potential, Limitations, and Pragmatics)
-  Simon Thompson. *Proof*. In *Research Directions in Parallel Functional Programming*, Kevin Hammond, Greg Michaelson (Eds.), Springer-V., Chapter 4, 93-119, 1999.
-  Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley/Pearson, 2nd edition, 1999. (Chapter 8, Reasoning about programs; Chapter 17.9, Proof revisited)




Chapter 6: Further Reading (15)

-  Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley/Pearson, 3rd edition, 2011. (Chapter 9, Reasoning about programs; Chapter 17.9, Proof revisited)
-  Franklyn Turbak, David Gifford with Mark A. Sheldon. *Design Concepts in Programming Languages*. MIT Press, 2008. (Chapter 105, Software Testing; Chapter 106, Formal Methods; Chapter 107, Verification and Validation)
-  Hiroshi Unno, Tachio Terauchi, Naoki Kobayashi. *Automating Relatively Complete Verification of Higher-Order Functional Programs*. In Conference Record of the 40th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2013), 75-86, 2013.





Chapter 6: Further Reading (16)

-  Daniel J. Velleman. *How to Prove It. A Structured Approach*. Cambridge University Press, 3rd edition, 2019.
-  Dimitrios Vytiniotis, Simon Peyton Jones, Dan Rosén, Koen Claessen. *HALO: Haskell to Logic through Denotational Semantics*. In Conference Record of the 40th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2013), 431-442, 2013.
-  Mitchell Wand. *Induction, Recursion, and Programming*. Elsevier, 1980.
-  Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993. (Chapter 1, Basic set theory; Chapter 3, Some principles of induction; Chapter 4, Inductive definitions; Chapter 8, Introduction to domain theory; Chapter 8.2, Streams – an example; Chapter 10.2, Fixed-point induction)






Special Reading for Chapter 6.5.1

-  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 – Proof by Approximation)
-  Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley/Pearson, 2nd edition, 1999. (Chapter 17.9, Proof revisited)
-  Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley/Pearson, 3rd edition, 2011. (Chapter 17.9, Proof revisited)

Special Reading for Chapter 6.5.2 (1)

-  Falk Bartels. *Generalized Coinduction*. Electronic Notes in Theoretical Computer Science 44:1, 21 pages, 2001.
<http://www.elsevier.nl/locate/entcs/volume44.html>
-  Falk Bartels. *Generalized Coinduction*. Journal of Mathematical Structures in Computer Science 13(2):321-348, 2003.
-  Kees Doets, Jan van Eijck. *The Haskell Road to Logic, Maths and Programming*. Texts in Computing, Vol. 4, King's College, UK, 2004. (Chapter 10.3, Proof by Approximation; Chapter 10.4, Proof by Coinduction)
-  Chung-Kil Hur, Georg Neis, Derek Dreyer, Viktor Vafeiadis. *The Power of Parameterization in Coinductive Proofs*. In Conference Record of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2013), 193-205, 2013.




Special Reading for Chapter 6.5.2 (2)

-  Bart Jacobs, Jan Rutten. *A Tutorial on (Co)algebras and (Co)induction*. EATCS Bulletin 62:222-259, 1997.
-  Marina Lenisa. *From Set-theoretic Coinduction to Coalgebraic Coinduction: Some Results, Some Problems*. Electronic Notes in Theoretical Computer Science 19:2-22, 1999.
-  Robin Milner. *Communications and Concurrency*. Prentice Hall, 1989. (Chapter 4)
-  Robin Milner. *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge University Press, 1999.
-  Flemming Nielson, Hanne Riis Nielson, Chris Hankin. *Principles of Program Analysis*. 2nd edition, Springer-V., 2005. (Appendix B.2, Introducing Coinduction; Appendix B.3, Proof by Coinduction)





Special Reading for Chapter 6.5.2 (3)

-  Jan Rutten. *Behavioural Differential Equations: A Coinductive Calculus of Streams, Automata, and Power Series*. Theoretical Computer Science 308:1-53, 2003.
-  Davide Sangiorgi. *On the Bisimulation Proof Method*. Journal of Mathematical Structures in Computer Science 8:447-479, 1998.
-  Davide Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2011.
-  Davide Sangiorgi, Jan Rutten (Eds). *Advanced Topics in Bisimulation and Coinduction*. Cambridge Tracts in Theoretical Computer Science, Vol. 52, Cambridge University Press, 2011.





Special Reading for Chapter 6.6

-  Hanne Riis Nielson, Flemming Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, 1992. (Chapter 6, Axiomatic Program Verification – Fixed Point Induction)
-  Hanne Riis Nielson, Flemming Nielson. *Semantics with Applications: An Appetizer*. Springer-V., 2007. (Chapter 9, Axiomatic Program Verification – Fixed Point Induction)
-  Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993. (Chapter 10.2, Fixed-point induction)

Special Reading for Chapter 6.7.1 (1)

-  Richard Bird. *Algebraic Identities for Program Calculation*. Computer Journal 32(2):122-126, 1989.
-  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.
-  Roderick Chapman. *Correctness by Construction: A Manifesto for High Integrity Software*. In Proceedings of the 10th Australian Workshop on Safety Critical Systems and Software, Vol. 55, 43-46, 2006.

Special Reading for Chapter 6.7.1 (2)

-  Henning Dierks, Michael Schenke. *A Unifying Framework for Correct Program Construction*. In Proceedings of the 4th International Conference on the Mathematics of Program Construction (MPC'98). Springer-V., LNCS 1422, 122-150, 1998.
-  Anthony Hall, Roderick Chapman. *Correctness by Construction: Developing a Commercial Secure System*. IEEE Software 19(1):18-25, 2002.
-  Charles A.R. Hoare. *The Ideal of Program Correctness*. The Computer Journal 50(3):254-260, 2007.
-  Derrick G. Kourie, Bruce W. Watson. *The Correctness-by-Construction Approach to Programming*. Springer-V., 2012.

Part IV

Advanced Language Concepts

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

724/196

Chapter 7

Functional Arrays

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

7.1

7.2

7.3

7.4

Chap. 8

Chap. 9

Chap. 10

Chap. 11

725/196

Chapter 7.1

Motivation

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

7.1

7.2

7.3

7.4

Chap. 8

Chap. 9

Chap. 10

Chap. 11

726/196

Imperative Arrays

...appealing:

- + Values of an array can be accessed or updated in constant time.
- + The update operation does not need extra space.
- + There is no need for chaining the array elements with pointers as they can be stored in contiguous memory locations.

...distracting:

- The size is fixed (defined and fixed at declaration time).

Functional Lists

...appealing:

- + The **size is not fixed**. Lists can get, can be **arbitrarily long, conceptually even infinitely long**.

...distracting:

- Lists do not enjoy the set of favorable properties of imperative arrays; most disturbing, values of a list **can not be accessed or updated in constant time**:

Accessing the i th element of a list (using `(!!)`) takes a number of steps **proportional** to i .

Functional Arrays

...shall complement **functional lists** and be designed and implemented **to get as close as possible** to the favorable properties of imperative arrays, i.e., **functional arrays** shall be:

...**appealing** because:

- + Accessing the i th element of an array (using **(!)**) shall take a **constant** number of steps, regardless of i .

...while accepting the **distracting** limitation applying to imperative arrays, too:

- The **size is fixed** (defined and fixed at creation time).

Note: Functional Arrays

...are not supported by the [standard prelude](#) of Haskell but by several specialized [libraries](#) like:

- `Data.Array` (\rightsquigarrow `import Data.Array`)
- `Data.Array.IArray` (\rightsquigarrow `import Data.Array.IArray`)
- `Data.Array.Diff` (\rightsquigarrow `import Data.Array.Diff`)

providing different kinds and implementations of [functional arrays](#):

- [Static](#) (or: [immutable](#)) arrays (w/out destructive update)
- [Dynamic](#) (or: [mutable](#)) arrays (w/ destructive update)

Nonetheless, how to implement [functional arrays](#)

- most adequately is a topic of [ongoing research](#).

Consequently, libraries evolve, disappear, are replaced over time requiring to stay tuned for updates on the Haskell homepage...

Chapter 7.2

Functional Arrays

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

7.1

7.2

7.2.1

7.2.2

7.3

7.4

Chap. 8

Chap. 9

Chap. 10

731/196

Chapter 7.2.1

Static Arrays

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

7.1

7.2

7.2.1

7.2.2

7.3

7.4

Chap. 8

Chap. 9

Chap. 10

732/196

The Library Data.Array

► `Data.Array` (\rightsquigarrow `import Data.Array`)

...supports `static` (or: `immutable`) `arrays` and provides three functions for creating static arrays:

1. `array bounds list_of_associations` (1st mechanism)
2. `listArray bounds list_of_values` (2nd mechanism)
3. `accumArray f init bounds list_of_associations` (3rd mechanism)

The Functions

...array, listArray, accumArray in more detail:

1. `array` :: `Ix a => (a,a) -> [(a,b)] -> Array a b`
`array` *bounds list_of_associations*
2. `listArray` :: `(Ix a) => (a,a) -> [b] -> Array a b`
`listArray` *bounds list_of_values*
3. `accumArray` :: `(Ix a) => (b -> c -> b) -> b`
`-> (a,a) -> [(a,c)] -> Array a b`
`accumArray` *f init bounds list_of_associations*

The Index Type Class `Ix`

...extends the type class `Ord` (and indirectly the type class `Eq`):

```
class (Ord a) => Ix a where
  range      :: (a,a) -> [a]
  index      :: (a,a) -> a -> Int
  inRange    :: (a,a) -> a -> Bool
  rangeSize  :: (a,a) -> Int
```

Member types of `Ix`

- must provide implementations of `range`, `index`, `inRange`, and `rangeSize`.
- are (mainly) used as index types of arrays.

Creating Static Arrays: 1st Mechanism

...using the function `array`, the most basic means:

► `array :: Ix a => (a,a) -> [(a,b)] -> Array a b`
`array bounds list_of_associations`

where

- `a`: the index type of the array; `b`: its entry type.
- `bounds`: a pair of expressions specifying the smallest and the largest array index.

Example: The expression pair `bounds`

- a) `(0,4)` and b) `((1,1), (10,10))` specify a
 - a) zero-origin vector of length five
 - b) one-origin 10 by 10 matrix, respectively.

Note: `bounds` can be given by any valid expression.

- `list_of_associations`: a list of pairs `(i,x)`, so-called `associations`, specifying that the array entry at index position `i` has value `x`.

Examples: array at Work

Let a' , f , n , m be the expressions:

```
a' = array (1,4) [(3,'c'),(2,' a'),(1,'f'),(4,'e')]
f n = array (0,n) [(i,i*i) | i<- [0..n]]
m   = array ((1,1),(2,3))
      [((i,j),(i*j)) | i<- [1..2], j<- [1..3]]
```

The types of these expressions are:

```
a' :: Array Int Char
f   :: Int -> Array Int Int
m   :: Array (Int,Int) Int
```

Their values are:

```
a'  ->> array (1,4) [(1,'f'),(2, 'a'),(3,'c'),(4,'e')]
f 3  ->> array (0,3) [(0,0),(1,1),(2,4),(3,9)]
m    ->> array ((1,1),(2,3)) [((1,1),1),((1,2),2),
                              ((1,3),3),((2,1),2),
                              ((2,2),4),((2,3),6)]
```

Note

If any specified index of an array is out of bounds

- the whole array is undefined.

I.e.: Function `array` is **strict** in **bounds**.

If two associations in an association list have the same index

- the array entry at that index is undefined.

I.e.: Function `array` is **non-strict** (or: **lazy**) in **values**.

...arrays can thus contain 'undefined' entries.

Example: Arrays at Work

Computing Fibonacci numbers:

```
fibs n = a
  where a = array (1,n) ([[1,0), (2,1)] ++
                        [(i, a!(i-1) + a!(i-2))
                         | i <- [3..n]])
```

Applications:

```
fibs 3 ->> array (1,3) [(1,0), (2,1), (3,1)]
```

```
fibs 5 ->> array (1,5) [(1,0), (2,1), (3,1),
                        (4,2), (5,3)]
```

```
fibs 12 ->> array (1,12) [(1,0), (2,1), (3,1),
                           (4,2), (5,3), (6,5),
                           (7,8), (8,13), (9,21),
                           (10,34), (11,55), (12,89)]
```

The Array Access Function (!)

...the counterpart of the list access function (!!)

$(!) :: \text{Ix } a \Rightarrow \text{Array } a \ b \rightarrow a \rightarrow b$

(!) returns the value $v :: b$ at index position $i :: a$.

Recall: The index type must be a member of the type class `Ix`, which foresees maps for typical index operations.

The Array Access Function (!) at Work

Computing Fibonacci numbers:

```
fibs n = a where
    a = array (1,n)
          ([[ (1,0), (2,1) ] ++ [(i, a!(i-1) +, a!(i-2))
                                | i <- [3..n]])
```

Applications of (!):

```
fibs 5!5      ->> 3
fibs 10!10    ->> 34
fibs 100!10   ->> 34 -- Thanks to lazy evaluation,
                    -- the computation stops at
                    -- fibs 10!10

fibs 50!50    ->> 7.778.742.049
fibs 100!100 ->> 218.922.995.834.555.169.026
fibs 5!10     ->> Program error: Ix.index: index
                    out of range
```

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

7.1

7.2

7.2.1

7.2.2

7.3

7.4

Chap. 8

Chap. 9

Chap. 10

741/196

Note: Local Declarations for Performance (1)

...the `where`-clause in the definition of `fibs` defining `a` locally is crucial for performance as it

- ▶ avoids the creation of new arrays during computation.

For illustration, compare the definitions of `fibs` and `xfibs`, where `a` (of a slightly different type) is globally defined:

```
fibs n = a where
  a = array (1,n)
        ([[ (1,0), (2,1) ] ++ [(i, a!(i-1) + a!(i-2))
                               | i <- [3..n]])
```

```
xfibs n = a n
a n      = array (1,n) ([[ (1,0), (2,1) ] ++
                        [(i, a n!(i-1) + a n!(i-2))
                         | i <- [3..n]])
```

Note: Local Declarations for Performance (2)

While

```
xfibs 3 ->> array (1,3) [(1,0), (2,1), (3,1)]
```

```
xfibs 5 ->> array (1,5) [(1,0), (2,1), (3,1), (4,2), (5,3)]
```

```
xfibs 12 ->> array (1,12) [(1,0), (2,1), (3,1),  
                          (4,2), (5,3), (6,5),  
                          (7,8), (8,13), (9,21),  
                          (10,34), (11,55), (12,89)]
```

```
xfibs 5!5 ->> 3
```

```
xfibs 10!10 ->> 34
```

```
xfibs 25!20 ->> 4.181 -- thanks to lazy evaluation  
                    -- the computation stops asap
```

works well for small arguments, the call:

```
xfibs 25!25 ->> ...takes too long to be feasible!
```

Overall: Though correct, evaluating `xfibs n` is most inefficient due to creating new arrays during the evaluation.

Creating Static Arrays: 2nd Mechanism

...using the function `listArray`, a more sophisticated means:

▶ `listArray :: (Ix a) => (a, a) -> [b] -> Array a b`
`listArray bounds list_of_values`

where

- *bounds*: specifies the values of the **smallest** and the **largest** index.
- *list_of_values*: specifies the **values** of the array elements in terms of a list.

Note: `listArray` is especially useful for the frequent case where

- where an array is constructed from a list of values given in index order.

Example: `listArray` at Work

```
a'' :: Array Int Char
a'' = listArray (1,8) "fun prog"
```

Evaluating `a''` yields:

```
a'' ->> array (1,8) [(1,'f'),(2,'u'),(3,'n'),(4,' '),
                    (5,'p'),(6,'r'),(7,'o'),(8,'g')]
```

Creating Static Arrays: 3rd Mechanism

...using the function `accumArray`, the most powerful means:

► `accumArray` :: (Ix a) => (b -> c -> b) -> b
-> (a, a) -> [(a, c)] -> Array a b
`accumArray f init bounds list_of_associations`

where

- `f`: specifies an **accumulation function**.
- `init`: specifies the (default) value the entries of the array shall be initialized with.
- `bounds`: specifies the values of the **smallest** and the **largest** index.
- `list_of_associations`: specifies the values of the array in terms of an **association list**.

Note: `accumArray` does not require that the indices occurring in `list_of_associations` are pairwise disjoint: Values of 'conflicting' indices are accumulated via `f`.

Example: accumArray at Work (1)

...a histogram function defined using accumArray:

```
histogram :: (Ix a, Num b) =>  
           (a,a) -> [a] -> Array a b
```

```
histogram bounds vs =  
  accumArray (+) 0 bounds [(i,1) | i <- vs]
```

Applications:

```
histogram (1,5) [4,1,4,3,2,5,5,1,2,1,3,4,2,1,1,3,2,1]  
->> array (1,5) [(1,6), (2,4), (3,3), (4,3), (5,2)]
```

```
histogram (-1,4) [1,3,1,1,3,1,1,3,1]  
->> array (-1,4) [(-1,0), (0,0), (1,6), (2,0), (3,3), (4,0)]
```

```
histogram (1,3) [5,3,1,3,4,2,(-4),1,1,3,2,1,5,(-9)]  
->> array
```

```
  Program error: Ix.index: index out of range
```

Example: accumArray at Work (2)

...a prime number test defined with `accumArray`:

```
primes :: Int -> Array Int Bool
primes n =
    accumArray (\e e' -> False) True (2,n) l
    where l = concat [map (flip (,) ())
                      (takeWhile (<=n) [k*i | k<-[2..]]
                               | i<-[2..n 'div' 2])]
```

Applications:

```
(primes 100)!1 ->> Program error: Ix.index: index
                  out of range
```

```
(primes 100)!2 ->> True
```

```
(primes 100)!4 ->> False
```

```
(primes 100)!71 ->> True
```

```
(primes 100)!100 ->> False
```

```
(primes 100)!101 ->> Program error: Ix.index: index
                       out of range
```

More Pre-Defined Operations on Arrays (1)

..pre-defined array operations:

- (!) :: (Ix a) => Array a b -> a -> b
- bounds :: (Ix a) => Array a b -> (a,a)
- indices :: (Ix a) => Array a b -> [a]
- elems :: (Ix a) => Array a b -> [b]
- assocs :: (Ix a) => Array a b -> [(a,b)]
- (//) :: (Ix a) => Array a b -> [(a,b)]
-> Array a b
- ...

More Pre-Defined Operations on Arrays (2)

Informally:

- (!): array **subscripting**, yields the *i*th element of an array.
- **bounds**: yields the **smallest** and **largest** index of an array.
- **indices**: yields a **list of the indices** of an array.
- **elems**: yields a **list of the elements/values** of an array.
- **assocs**: yields a list of **index/value pairs** of the elements of an array, i.e., the **list of associations** of an array.
- (//): array **updating** – (//) takes an array (left argument) and a list of associations (right argument) and returns a **new** array, which is identical to the argument array except for the values of elements occurring in the argument list of associations.

Note: (//) generates a modified copy of the argument array; it does **not** perform a **destructive** update!

- ...

Example: More Array Operations at Work (1)

Applications (w/ pre-defined functions on arrays):

```
elems (primes 10)  
->> [True,True,False,True,False,True,False,False,False]
```

```
assocs (primes 10)  
->> [(2,True),(3,True),(4,False),(5,True),(6,False),  
      (7,True),(8,False),(9,False),(10,False)]
```

```
yieldPrimes (assocs (primes 100))  
->> [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,  
      59,61,67,71,73,79,83,89,97]
```

where

```
yieldPrimes :: [(a,Bool)] -> [a]  
yieldPrimes [] = []  
yieldPrimes ((v,w):t)  
  | w          = v : yieldPrimes t  
  | otherwise = yieldPrimes t
```

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

7.1

7.2

7.2.1

7.2.2

7.3

7.4

Chap. 8

Chap. 9

Chap. 10

751/196

Example: More Array Operations at Work (2)

Let:

```
m = array ((1,1),(2,3)) [((i,j),i*j) | i <- [1..2],  
                           j <- [1..3]]  
                               :: Array (Int,Int) Int  
m ->> array ((1,1),(2,3)) [((1,1),1),((1,2),2),((1,3),3),  
                             ((2,1),2),((2,2),4),((2,3),6)]  
m!(1,2) ->> 2, m!(2,2) ->> 4, m!(2,3) ->> 6
```

Applications of array operations:

```
bounds m ->> ((1,1),(2,3))  
indices m ->> [(1,1),(1,2),(1,3),(2,1),(2,2),(2,3)]  
elems m ->> [1,2,3,2,4,6]  
assocs m ->> [((1,1),1),((1,2),2),((1,3),3),  
              ((2,1),2), ((2,2),4), ((2,3),6)]  
m // [((1,1),4), ((2,2),8)]  
->> array ((1,1),(2,3)) [((1,1),4),((1,2),2),((1,3),3),  
                          ((2,1),2),((2,2),8),((2,3),6)]
```


Example: More Array Operations at Work (3)

...illustrating the `update` operation (`//`) by means of modifying the `histogram` function:

```
histogram (lower,upper) xs
  = updHist (array (lower,upper)
                  [(i,0) | i <- [lower..upper]])
            xs
```

```
updHist a []      = a
updHist a (x:xs) = updHist (a // [(x, (a!x + 1))]) xs
```

Application:

```
histogram (0,9) [3,1,4,1,5,9,2]
->> array (0,9) [(0,0), (1,2), (2,1), (3,1), (4,1),
                 (5,1), (6,0), (7,0), (8,0), (9,1)]
```

Updating Arrays: accum complementing (//)

...`accum`, another pre-defined operation on arrays:

```
► accum :: (Ix a) => (b -> c -> b) -> Array a b
      -> [(a,c)] -> Array a b
```

`accum f a list_of_associations`

...instead of replacing previously stored values as `(//)` does, `accum` accumulates values referring to the same index using `f`.

Example:

```
accum (+) m [((1,1),4), ((2,2),8)] -- m as before
->> array ((1,1),(2,3))
      [((1,1),5), ((1,2),2), ((1,3),3),
       ((2,1),2), ((2,2),12), ((2,3),6)]
```

Note: The result of `accum` is a `new` matrix, which is identical to `m` except for the entries at positions `(1,1)` and `(2,2)` to whose values `1` and `4`, `4` and `8` have been added, respectively.

Higher-Order Functions on Arrays

...can be defined just as on lists, e.g.:

```
amap :: (b -> c) -> Array a b -> Array a c
```

Example: The call

```
amap (\x -> x*10) a
```

yields a new array where all elements of `a` are multiplied by `10`.

User-defined Higher-Order Array Functions

The functions `row` and `col` return a row and a column of a matrix, respectively:

```
row :: (Ix a, Ix b) =>
      a -> Array (a,b) c -> Array b c
row i m = ixmap (l',u') (\j->(i,j)) m
      where ((l,l'),(u,u')) = bounds m
```

```
col :: (Ix a, Ix b) =>
      a -> Array (b,a) c -> Array b c
col j m = ixmap (l,u) (\i->(i,j)) m
      where ((l,l'),(u,u')) = bounds m
```

where

```
ixmap :: (Ix a, Ix b) => (a,a) -> (a -> b)
      -> Array b c -> Array a c
ixmap b f a = array b [(k,a!f k) | k <- range b]
```

Examples: row, col at Work

...where `m` is assumed to be as before:

```
row 1 m ->> array (1,3) [(1,1), (2,2), (3,3)]
```

```
row 2 m ->> array (1,3) [(1,2), (2,4), (3,6)]
```

```
row 3 m ->> array (1,3) [(1,
```

```
Program error: Ix.index: index out of  
range
```

```
col 1 m ->> array (1,2) [(1,1), (2,2)]
```

```
col 2 m ->> array (1,2) [(1,2), (2,4)]
```

```
col 3 m ->> array (1,2) [(1,3), (2,6)]
```

```
col 4 m ->> array (1,2) [(1,
```

```
Program error: Ix.index: index out of  
range
```

Chapter 7.2.2

Dynamic Arrays

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

7.1

7.2

7.2.1

7.2.2

7.3

7.4

Chap. 8

Chap. 9

Chap. 10

758/196

The Library `Data.Array.Diff`

► `Data.Array.Diff` (\rightsquigarrow `import Data.Array.Diff`)

...supports `dynamic` (or: `mutable`) arrays.

Compared to the library `Data.Array`, the type:

- `DiffArray` (for dynamic arrays)

replaces the type

- `Array` (for static arrays)

...everything else behaves analogously.*)

*) `Data.Array.Diff` is no longer maintained; `Data.Array.IO` can be considered a substitute but offers a different monadic-based interface.

Chapter 7.3

Summary

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

7.1

7.2

7.3

7.4

Chap. 8

Chap. 9

Chap. 10

Chap. 11

760/196

Summing up (1)

Static (Immutable) Arrays

- ▶ **Access operator (!)**: Each array element is accessible in constant time.
- ▶ **Update operator (//)**: Not a destructive update; instead: an identical copy of the argument array is created except of those elements being 'updated.' Updates thus do not take constant time.

Dynamic (Mutable) Arrays

- ▶ **Update operator (//)**: Destructive update; update operations take constant time per index.
- ▶ **Access operator (!)**: Access to array elements may sometimes take longer as for static arrays.

Summing up (2)

Updates

- ▶ can often completely be avoided by smartly written recursive array constructions (cp. the [prime number test](#) in [Chapter 7.2.1](#)).

Dynamic arrays

- ▶ should only be used if constant time updates are crucial for the application.

For an [extended example](#) showing

- [arrays](#) at work

refer to [Chapter 18.2](#) dealing with an [imperative robot language](#) for controlling [robot](#) actions.

Chapter 7.4

References, Further Reading

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

7.1

7.2

7.3

7.4

Chap. 8





Chap. 9

Chap. 10




Chap. 11

763/196



Chapter 7: Further Reading (1)

-  Henry G. Baker. *Shallow Binding Makes Functional Arrays Fast*. ACM SIGPLAN Notices 26(8):145-147, 1991.
-  Richard Bird. *Thinking Functionally with Haskell*. Cambridge University Press, 2015. (Chapter 10.5, Mutable arrays; Chapter 10.6, Immutable arrays)
-  Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Chapter 10.1, Arrays)
-  Manuel M.T. Chakravarty, Gabriele Keller. *An Approach to Fast Arrays in Haskell*. In Johan Jeuring, Simon Peyton Jones (Eds.) *Advanced Functional Programming – Revised Lectures*. Springer-V., LNCS Tutorial 2638, 27-58, 2003.





Chapter 7: Further Reading (2)

-  Antonie J.T. Davie. *An Introduction to Functional Programming Systems using Haskell*. Cambridge University Press, 1992. (Chapter 4.6, Arrays)
-  Klaus E. Grue. *Arrays in Pure Functional Programming Languages*. International Journal on Lisp and Symbolic Computation 2:105-113, Kluwer Academic Publishers, 1989.
-  Paul Hudak. *Arrays, Non-determinism, Side-effects, and Parallelism: A Functional Perspective*. In Proceedings of a Workshop on Graph Reduction (WGR'86), Springer-V., LNCS 279, 312-327, 1986.




Chapter 7: Further Reading (3)

-  Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000. (Chapter 19.4, All the World is a Grid; Chapter 24.6, The Index Class)
-  John Hughes. *An Efficient Implementation of Purely Functional Arrays*. Technical Report, Programming Methodology Group, Chalmers University of Technology, 1985.
-  Melissa E. O'Neill, F. Warren Burton. *A New Method for Functional Arrays*. *Journal of Functional Languages* 7(5):487-513, 1997.
-  Peter Pepper, Petra Hofstedt. *Funktionale Programmierung*. Springer-V., 2006. (Kapitel 14, Funktionale Arrays und numerische Mathematik)

Chapter 7: Further Reading (4)

-  Simon Peyton Jones (Ed.). *Haskell 98: Language and Libraries. The Revised Report*. Cambridge University Press, 2003. www.haskell.org/definitions. (Chapter 16, Arrays)
-  Simon Peyton Jones. *Haskell 98 Libraries: Arrays*. Journal of Functional Programming 13(1):173-178, 2003.
-  Fethi Rabhi, Guy Lapalme. *Algorithms – A Functional Programming Approach*. Addison-Wesley, 1999. (Chapter 2.7, Arrays; Chapter 4.3, Arrays)
-  Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Chapter 12, Barcode Recognition – Introducing Arrays)

Chapter 7: Further Reading (5)

-  Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley/Pearson, 2nd edition, 1999. (Chapter 19, Time and space behaviour – arrays)
-  Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley/Pearson, 3rd edition, 2011. (Chapter 20, Time and space behaviour)
-  Philip Wadler. *A New Array Operation*. In Proceedings of a Workshop on Graph Reduction (WGR'86), Springer-V., LNCS 279, 328-335, 1986.

Chapter 8

Abstract Data Types

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

8.1

8.2

8.3

8.4

8.5

8.6

8.7

8.8

Chap. 9

Chapter 8.1

Motivation

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

8.1

8.2

8.3

8.4

8.5

8.6

8.7

8.8

Chap. 9

770/196

Concrete Data Types (CDTs)

...are specified by **naming their values** (not by naming their operations):

- ▶ With the exception of functions as values of a CDT, every CDT value is uniquely described by an expression composed of **constructors**.
- ▶ Using pattern matching, these expressions can be generated, inspected, and modified in various ways by operations associated with the CDT.
- ▶ There is **no need**, however, **to specify any operation** associated with a CDT at the time of defining it.

...the Haskell means for defining CDTs are **algebraic** (and **new type**) data type definitions.

Illustration: CDTs, CDT Values in Haskell

```
type Forename = String
...
type Publisher = String
type Edition   = Int

data Vehicle = Bicycle | Motorcycle | Car | Bus
data Tree a  = Nil | Leaf a | Root (Tree a) a (Tree a)
data Person  = P Forename Surname Address
newtype Book = B (Author,Title,Publisher,Edition)

v1 = Bicycle :: Vehicle
v2 = Car     :: Vehicle
t1 = Leaf 42 :: Tree Int
t2 = Root Nil True (Leaf False) :: Tree Bool
p  = P "Simon" "Thompson" "unknown" :: Person
b  = B ("Thompson","Haskell","Addison-Wesley",2) :: Book
```

Note: At the time of defining the above CDTs, there is no need to define operations manipulating their values.

Abstract Data Types (ADTs)

...are specified by **naming their operations** (not by naming their values):

- ▶ The **meaning of the operations** is precisely specified by means of **laws**, while the internal structure of the ADT, i.e., the representation of its values and the definition of its associated operations are left open; there is **no need to define the internal structure of an ADT** at the time of defining it.
- ▶ An ADT and its associated operations **are implemented by a CDT and the operations associated with it**, which, however, **are kept invisible** to a user of the ADT.
- ▶ In general, an ADT can be **implemented by various CDTs**, which can be chosen for simplicity, performance, etc.

...the Haskell means of choice for defining and implementing ADTs are **modules** hiding their CDT implementations.

Why Abstract Data Types?

...by introducing a **level of indirection** between specification and implementation of a data type, we achieve:

- ▶ **Separation of concerns:** Separation of **specification** (interface and behaviour specification) and **implementation** of a data type (in terms of a CDT and CDT operations matching the ADT operations).
- ▶ **Information hiding:** No disclosure of the internal structure of the CDT, the representation and implementation of its values and the operations working on them.
- ▶ **Security:** CDT values implementing their (only) implicitly defined ADT counterparts can exclusively be created, accessed, and manipulated using the ADT operations implemented by their CDT counterparts.

Defining and Implementing an ADT

...is technically a three-stage approach of **specification**, **implementation**, and **verification**:

▶ **Specification (user-visible)**

- **Interface Specification**: Signatures of ADT operations
- **Behaviour Specification**: Laws for ADT operations

▶ **Implementation (user-invisible)**

- Implementing the ADT values in terms of a CDT
- Implementing the ADT operations as CDT operations

▶ **Verification**

- **Specification**: Proving that the ADT laws are consistent and complete (**proof obligation of the ADT specifier**)
- **Implementation**: Proving that the implemented CDT operations are sound, i.e., satisfy the ADT laws (**proof obligation of the CDT implementor**)

Benefits of Abstract Data Type Definitions

...supporting **programming-in-the large**:

- ▶ ADTs enable **modular program development** by separating the responsibilities for specifying and implementing a data type and the operations associated with it.

...supporting **reusability** and **maintainability**:

- ▶ If non-functional requirements for an ADT implementation change or evolve over time, a current CDT implementation of the ADT and its operations **can easily be replaced** by a new one fitting better to the new requirements as long as the new CDT implementation satisfies the interface and behaviour specification of the ADT.

In the following

...we demonstrate how ADTs can be defined and implemented in Haskell considering:

- Stacks
- Queues
- Priority Queues
- Tables

The Challenge:

- ADTs are **not** a first-class citizen in Haskell.
- Therefore, we have to pragmatically make use of Haskell features allowing us to achieve the constituting properties of ADTs of **information hiding**, of **separating** their **user-visible specification** from their **user-invisible implementation** as good as possible.

Chapter 8.2

Stacks

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

8.1

8.2

8.3

8.4

8.5

8.6

8.7

8.8

Chap. 9

Interface Specification

...of the ADT stack, named `Stack` (user-visible):

```
module Stack (Stack, empty, is_empty, push, pop, top)
    where

-- Interface Spec.: Signatures of stack operations
empty      :: Stack a
is_empty   :: Stack a -> Bool
push       :: a -> Stack a -> Stack a
pop        :: Stack a -> Stack a
top        :: Stack a -> a

-- Behaviour Spec.: Laws for stack operations
Laws (1) thru (6)
```

Note, the `laws` must be chosen to enforce a `last-in/first-out` (**LIFO**) behaviour of stacks; any implementation of stacks must ensure these laws.

Behaviour Specification

...of the **stack operations** of the **ADT stack (user-visible)**:

Behaviour Spec.: Laws for stack operations

- 1) `is_empty empty` == `True`
- 2) `is_empty (push v s)` == `False`
- 3) `top empty` == `undef`
- 4) `top (push v s)` == `v`
- 5) `pop empty` == `undef`
- 6) `pop (push v s)` == `s`

Homework: Prove that the above laws enforce a **last-in/first-out (LIFO)** behaviour of stacks.

Implementation A

...of the ADT `stack` as an algebraic data type (user-invisible):

```
data Stack a    = Empty | Stk a (Stack a)
empty           = Empty
is_empty Empty = True
is_empty _     = False
push x s       = Stk x s
pop Empty      = error "Stack is empty"
pop (Stk _ s)  = s
top Empty      = error "Stack is empty"
top (Stk x _)  = x
```

Implementation B

...of the ADT stack as a new type (user-invisible):

```
newtype Stack a    = Stk [a]
empty              = Stk []
is_empty (Stk []) = True
is_empty (Stk _) = False
push x (Stk xs)   = Stk (x:xs)
pop (Stk [])      = error "Stack is empty"
pop (Stk (_:xs)) = Stk xs
top (Stk [])      = error "Stack is empty"
top (Stk (x:_))   = x
```

“Implementation” C

...of the ADT stack as an alias type (user-invisible):

```
type Stack a = [a]
empty        = []
is_empty []  = True
is_empty _   = False
push x xs    = (x:xs)
pop []       = error "Stack is empty"
pop (_:xs)   = xs
top []       = error "Stack is empty"
top (x:_)    = x
```

Verification

Specifier and implementer of the ADT stack can prove, respectively:

Lemma 8.2.1 (Consistency, Completeness)

The 6 laws of the behaviour specification of the ADT stack are consistent and complete.

Lemma 8.2.2 (Soundness)

The implementations A and B (and C) satisfy the 6 laws of the behaviour specification of the ADT stack.

A Critical Note on “Implementation” C

...of stacks as an

- alias type of predefined lists: `type Stack a = [a]`

Obvious (but actually only apparent) benefit of implementing stacks as predefined lists:

- Even less conceptual overhead than for stacks implemented as a new type `newtype Stack a = Stk [a]` where the constructor `Stk` needs to be handled by the implementations of the stack operations.

But

Security is broken and lost!

- ▶ All predefined operations on lists are available on stacks (not just the 5 ADT operations of stack).

Worse

- ▶ Many of the predefined operations on lists (reversal, element picking, etc.) are not even meaningful for stacks.
- ▶ Even hiding the implementation in a module can not prevent the application of such meaningless operations to stacks but requires to explicitly abstain from them.

Hence

- ▶ “Implementation” C violates the spirit of an ADT implementation and should not be considered a reasonable and valid implementation of the ADT stack.

Chapter 8.3

Queues

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

8.1

8.2

8.3

8.4

8.5

8.6

8.7

8.8

Chap. 9

787/196

Interface Specification

...of the ADT `queue`, named `Queue` (user-visible):

```
module Queue (Queue,emptyQ,is_EmptyQ,
              enQ,deQ,frontQ) where

-- Interface Spec.: Signatures of queue operations
emptyQ      :: Queue a
is_emptyQ   :: Queue a -> Bool
enQ         :: a -> Queue a -> Queue a
deQ         :: Queue a -> Queue a
frontQ      :: Queue a -> a

-- Behaviour Spec.: Laws for queue operations
Laws (1) thru (6)
```

Note, the `laws` must be chosen to enforce a `first-in/first-out` (`FIFO`) behaviour of queues; any implementation of queues must ensure these laws.

Behaviour Specification

...of the queue operations of the ADT queue (user-visible):

Behaviour Spec.: Laws for queue operations:

- 1) `is_emptyQ emptyQ` == `True`
- 2) `is_emptyQ (enQ v q)` == `False`
- 3) `frontQ emptyQ` == `undef`
- 4) `frontQ (enQ v q)` == `if is_emptyQ q`
 `then v`
 `else frontQ q`
- 5) `deQ emptyQ` == `undef`
- 6) `deQ (enQ v q)` == `if is_emptyQ q`
 `then emptyQ`
 `else enQ ((deQ q) v)`

Homework: Prove that the above laws enforce a **first-in/first-out (FIFO)** behaviour of queues.

Implementation A

...of the ADT queue as a new type (user-invisible):

```
newtype Queue a = Q [a]
emptyQ          = Q []
is_emptyQ (Q []) = True
is_emptyQ _     = False
enQ x (Q q)     = Q (q ++ [x])
deQ (Q [])      = error "Queue is empty"
deQ (Q (_:xs))  = Q xs
frontQ (Q [])   = error "Queue is empty"
frontQ (Q (x:_)) = x
```

Implementation B

...of the ADT queue as a new type (user-invisible):

```
newtype Queue a          = Q ([a], [a])
                           front rear (in reverse order)
                           of the queue

emptyQ                    = Q ([], [])
is_emptyQ (Q ([], []))   = True
is_emptyQ _                = False
enQ x (Q ([], []))       = Q ([x], [])
enQ y (Q (xs, ys))       = Q (xs, y:ys)
deQ (Q ([], []))         = error "Queue is empty"
deQ (Q ([], ys))         = Q (tail(reverse ys), [])
deQ (Q (x:xs, ys))       = Q (xs, ys)
frontQ (Q ([], []))      = error "Queue is empty"
frontQ (Q ([], ys))      = last ys
frontQ (Q (x:xs, ys))    = x
```

Verification

Specifier and implementer of the ADT queue can prove, respectively:

Lemma 8.3.1 (Consistency, Completeness)

The 6 laws of the of the behaviour specification of the ADT queue are consistent and complete.

Lemma 8.3.2 (Soundness)

The implementations A and B satisfy the 6 laws of the behaviour specification of the ADT queue.

Exercise 8.3.3

Implementation B of the ADT queue is more efficient than implementation A. Why?

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

8.1

8.2

8.3

8.4

8.5

8.6

8.7

8.8

Chap. 9

793/196

Chapter 8.4

Priority Queues

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

8.1

8.2

8.3

8.4

8.5

8.6

8.7

8.8

Chap. 9

794/196

Interface/Behaviour Specification

...of the ADT priority queue, named PQueue (user-visible):

```
module PQueue (PQueue,emptyPQ,is_emptyPQ,
               enPQ,dePQ,frontPQ) where
```

```
-- Interface Spec.: Signatures of priority queue ops
```

```
emptyPQ      :: PQueue a
```

```
is_emptyPQ   :: PQueue a -> Bool
```

```
enPQ         :: (Ord a) => a -> PQueue a -> PQueue a
```

```
dePQ         :: (Ord a) => PQueue a -> PQueue a
```

```
frontPQ      :: (Ord a) => PQueue a -> a
```

```
-- Behaviour Spec.: Laws for priority queue operations
```

```
...Homework!
```

Note: Each entry of a priority queue has a priority associated with it. The dequeue operation always removes the entry with the highest (or lowest) priority, which is ensured by the enqueue operation, which places a new element according to its priority in a queue.

Implementation

...of the ADT priority queue as a new type (user-invisible):

```
newtype PQueue a      = PQ [a]
emptyPQ               = PQ []
is_emptyPQ (PQ [])   = True
is_emptyPQ _         = False
enPQ x (PQ pq)       = PQ (insert x pq)
  where
    insert x []           = [x]
    insert x r@(e:r') | x <= e = x:r' -- the smaller the
                                     -- higher the priority
                          | otherwise = e:insert x r'

dePQ (PQ [])          = error "Priority queue is empty"
dePQ (PQ (_:xs))     = PQ xs

frontPQ (PQ [])       = error "Priority queue is empty"
frontPQ (PQ (x:_))   = x
```

Verification

Specifier and implementer of the ADT priority queue need to show, respectively:

- ▶ The laws of the behaviour specification of the ADT priority queues are consistent and complete.
- ▶ The implementation satisfies the laws of the behaviour specification of the ADT priority queue.

...where the specification of the laws was left as homework.

Chapter 8.5

Tables

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

8.1

8.2

8.3

8.4

8.5

8.5.1

8.5.2

8.6

8.7

8.8

Chapter 8.5.1

Tables as Functions and Lists

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

8.1

8.2

8.3

8.4

8.5

8.5.1

8.5.2

8.6

8.7

8.8

Interface/Behaviour Specification

...of the ADT table, named `Table` (user-visible):

```
module Table (Table,new_T,find_T,upd_T) where
-- Interface Spec.: Signatures of table operations
new_T  :: (Eq b) => [(b,a)] -> Table a b
find_T :: (Eq b) => Table a b -> b -> a
upd_T  :: (Eq b) => (b,a) -> Table a b -> Table a b
```

-- Behaviour Spec.: Laws for table operations

Intuitively:

```
-- new_T assoc_list: create a new table and ini-
--   tialize it with the data of assoc_list.
-- find_T tab ind: retrieve information stored in
--   table tab at index ind.
-- upd_T (ind,val) tab: update the entry of table
--   tab stored at index ind with value val.
```

Details: Homework!

Implementation A

...of the ADT table as a function (user-invisible):

```
newtype Table a b = Tbl (b -> a)

new_T assoc_list =
  foldr upd_T
    (Tbl (_ -> error "Item not found"))
    assoc_list

find_T (Tbl f) index = f index

upd_T (index,value) (Tbl f) = Tbl g
  where g j | j==index  = value
           | otherwise = f j
```

Implementation B

...of the ADT table as a new type (user-invisible):

```
newtype Table a b = Tbl [(b,a)]

new_T assoc_list = Tbl assoc_list

find_T (Tbl []) i = error "Item not found"
find_T (Tbl ((j,value):r)) index
  | index==j    = value
  | otherwise   = find_T (Tbl r) index

upd_T e (Tbl []) = Tbl [e]
upd_T e'@(index,_) (Tbl (e@(j,_) : r))
  | index==j    = Tbl (e':r)
  | otherwise   = Tbl (e:r')
where Tbl r' = upd_T e' (Tbl r)
```

Verification

Specifier and implementer of the **ADT table** need to show, respectively:

- ▶ The **laws of the behaviour specification** of the **ADT table** are consistent and complete.
- ▶ The implementation satisfies the **laws of the behaviour specification** of the **ADT table**.

...where the specification of the laws was left for **homework**.

Chapter 8.5.2

Tables as Arrays

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

8.1

8.2

8.3

8.4

8.5

8.5.1

8.5.2

8.6

8.7

8.8

Interface/Behaviour Specification

...of the ADT table, named `Table'` (user-visible):

```
module Tab (Table',new_T',find_T',upd_T') where

  -- Interface Spec.: Signatures of table operations
  new_T'   :: (Ix b) => [(b,a)] -> Table' a b
  find_T'  :: (Ix b) => Table' a b -> b -> a
  upd_T'   :: (Ix b) => (b,a) -> Table' a b
              -> Table' a b

  -- Behaviour Spec.: Laws for table operations
  ...Homework!
```

Note: The signatures of the table operations have been enlarged by the context `(Ix b) =>` in order to be prepared for array manipulations.

Implementation

...of the ADT table as a new type (user-invisible):

```
newtype Table' a b = Tbl' (Array b a)

new_T' assoc_list = Tbl' (array (low,high) assoc_list)
  where indices    = map fst assoc_list
        low        = minimum indices
        high       = maximum indices

find_T' (Tbl' a) index      = a ! index

upd_T' p@(index,value) (Tbl' a) = Tbl' (a // [p])
```

Note

- `new_T'` takes an association list of index/value pairs and returns the corresponding table.

To this end, `new_T'` determines first the list of indices `indices` of association list `assoc_list`, and based on this the boundaries of the new table array by computing the minimum `low` and the maximum `high` index of `assoc_list`; afterwards it constructs the new table array applying the function `array` to the pair of array bounds `(low,high)` and association list `assoc_list`.

- `find_T'` and `upd_T'` are used to retrieve and update values in the table array, respectively. Note that `find_T'` returns a system error, not a user error, when applied to an invalid index.

Verification

Specifier and implementer of the ADT table need to show, respectively:

- ▶ The **laws for table** are consistent and complete.
- ▶ The implementation satisfies the **laws of the ADT operations** of the ADT table.

...whose specification was left as **homework**.

Chapter 8.6

Displaying ADT Values in Haskell

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

8.1

8.2

8.3

8.4

8.5

8.6

8.7

8.8

Chap. 9

809/196

Displaying ADT Values

...is often necessary but **requires some special care**, especially in **Haskell**.

The reasons for this are twofold:

1. ADT values can only be accessed using the ADT operations. Usually, it is crude and cumbersome to display all values of a complex ADT value like a stack or a queue using only the ADT operations, e.g., by completely popping a whole stack.
2. Displaying ADT values straightforwardly in terms of their CDT representations can reveal the internal structure of the CDT breaking the ADT principles of **information hiding** and (possibly) **security**.

In Haskell

...[breaking](#) the principles of [information hiding](#) and (possibly) [security](#) always happens if the CDT implementing an ADT is made an instance of the type class [Show](#) using an automatic

▶ [deriving](#)-clause

which is demonstrated next considering stacks for illustration.

The Problem: Automatic deriving-Clauses

...are unsafe:

```
data Stack a      = Empty
                  | Stk a (Stack a) deriving Show

newtype Stack a  = Stk [a] deriving Show

type Stack a     = [a] -- Lists are instance of Show;
                       -- hence, no deriving clause
                       -- required.
```

because displaying stack values reveals their internal structure:

```
push 3 (push 2 (push 1 emptyS))
->> Stk 3 (Stk 2 (Stk 1 Empty))
```

```
push 3 (push 2 (push 1 emptyS))
->> Stk [3,2,1]
```

```
push 3 (push 2 (push 1 emptyS))
->> [3,2,1] ->> (3:2:1:[])
```

A Note on Information Hiding and Security (1)

Information hiding

- ▶ is **broken** for all three implementation variants as algebraic type, new type, and type alias: Displaying stack values discloses their internal structure and data constructors.

Security

- ▶ is **broken** for the variant as **type alias**: All list operations are immediately available to create, access, and manipulate stack values using arbitrary list operations. Therefore, type aliases of basic types are not considered valid ADT implementations.
- ▶ is **preserved** for the variants as **algebraic type** and **new type**: This is because the data value constructors **Empty** and **Stk** are not exported from the module. A user of the module can thus not use or create a stack value by any other way than the operations exported by the module.

A Note on Information Hiding and Security (2)

This holds analogously for the other ADT implementations:

Stacks

```
data Stack a      = Empty
                  | Stk a (Stack a) deriving Show
newtype Stack a = Stk [a] deriving Show
type Stack a     = [a]
```

Queues and Priority Queues

```
newtype Queue a  = Q [a] deriving Show
newtype PQueue a = PQ [a] deriving Show
```

Tables

```
newtype Table a b = Tbl [(b,a)] deriving Show
newtype Table a b = Tbl (Array b a) deriving Show
```

...straightforward and easy but `unsafe` and (possibly) `insecure`.

Remedy: Explicit instance-Declarations (1)

...the *safe*, *secure*, and thus *recommended* way for displaying ADT values, here *stacks*:

- A) `instance (Show a) => Show (Stack a) where`
 `showsPrec _ Empty str = showChar '-' str`
 `showsPrec _ (Stk x s) str`
 `= shows x (showChar '|' (shows s str))`
- B) `instance (Show a) => Show (Stack a) where`
 `showsPrec _ (Stk []) str = showChar '-' str`
 `showsPrec _ (Stk (x:xs)) str`
 `= shows x (showChar '|' (shows (Stk xs) str))`
- C) `instance (Show a) => Show (Stack a) where`
 `showsPrec _ [] str = showChar '-' str`
 `showsPrec _ (x:xs) str`
 `= shows x (showChar '|' (shows xs str))`

Remedy: Explicit instance-Declarations (2)

This way, the very same output for all 3 implementations:

```
push 3 (push 2 (push 1 emptyS)) ->> 3|2|1|-
```

No implementation details about the internal data structure are disclosed:

- ▶ Independently of the chosen implementation A, B, (or C), the output is the same.
- ▶ Hence, the actually chosen implementation of the ADT `Stack` remains hidden. It is not disclosed to the user (of the module).

Note: The first argument of `showsPrec` is an unused precedence value.

Challenge: Displaying Tables

...represented as `functions` because there is no general meaningful way to display a function. An instance declaration for

```
newtype Table a b = Tbl (b -> a)
```

for the type class `Show` could thus be chosen minimal/trivial:

```
instance Show (Table a b) where
  showsPrec _ _ str = showString "<<A Table>>" str
```

Chapter 8.7

Summary

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

8.1

8.2

8.3

8.4

8.5

8.6

8.7

8.8

Chap. 9

Abstract Data Types

...are not a **first-class citizen** in Haskell.

Nonetheless, specifying and implementing ADTs using modules ensures all three design goals strived for with ADTs:

- ▶ **Separation of concerns:** Separation of **specification** (interface and behaviour specification) and **implementation** of a data type (in terms of a CDT and CDT operations matching the ADT operations).
- ▶ **Information hiding:** No disclosure of the internal structure of the CDT, the representation and implementation of its values and the operations working on them.
- ▶ **Security:** CDT values implementing their (only) implicitly defined ADT counterparts can exclusively be created, accessed, and manipulated by using the ADT operations implemented by their CDT counterparts.

Note

Due to [limitations](#) of the [module concept](#) in [Haskell](#), the

- ▶ [behaviour specification](#) of ADTs can only be given as [comments](#).

If [ADT values](#) need to be [displayed](#), this can be done by

- ▶ by making the underlying CDT a member of the type class [Show](#).

This should always be done by means of an explicit

- ▶ [instance](#)-declaration

since a (more convenient) [deriving](#)-clause, if possible, would reveal the internal representation of the CDT values, especially the data constructors of the CDT breaking the [information hiding principle](#) of ADTs (though the constructors could not be used by a user since they are not exported from the module).

Benefits of Using Abstract Data Types

...evolve directly from the 'by-design built-in' ADT properties:

- ▶ **Separation of concerns**, i.e., the separation of the specification and implementation of a data type

enables

- ▶ **Information hiding**: Only the interface and the behaviour specification of the ADT are publicly known; its implementation as a CDT and operations on it are hidden.

This ensures:

- ▶ **Security** of the data (structure) and its data values from uncontrolled, unintended, or not permitted access.

Altogether, this enables:

- ▶ **Simple exchangeability** of the CDT implementation of an ADT (e.g., **simplicity** vs. **scalability/performance**).
- ▶ **Modularization** and **programming-load sharing** supporting programming-in-the-large.

Relevance of Abstract Data Types

...there are many more examples of **data structures**, which can be specified and implemented in terms of **abstract data types** in order to benefit from the built-in ADT properties such as **separation of concerns**, **information hiding**, **security**, **exchangeability**, **modularity**, etc., including

- Sets
- Heaps
- Trees (binary search trees, balanced trees,...)
- ...

and also

- **Arrays**

as illustrated next.

Arrays as Abstract Data Type in Haskell (1)

```
module Array (  
    module Ix, -- export all of Ix (for convenience)  
    Array, array, listarray (!), bounds, indices,  
    elems, assocs, accumArray, (//),  
    accum, ixmap ) where  
  
import Ix  
infixl 9 !, // ... -- Operator precedence  
data (Ix a) => Array a b = ... -- Abstract  
  
array      :: (Ix a) => (a,a) -> [(a,b)] -> Array a b  
listArray  :: (Ix a) => (a,a) -> [b] -> Array a b  
(!)       :: (Ix a) => Array a b -> a -> b  
bounds     :: (Ix a) => Array a b (a,a)  
indices    :: (Ix a) => Array a b -> [a]  
elems      :: (Ix a) => Array a b -> [b]  
assocs     :: (Ix a) => Array a b -> [(a,b)]
```

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

8.1

8.2

8.3

8.4

8.5

8.6

8.7

8.8

Chap. 9

823/196

Arrays as Abstract Data Type in Haskell (2)

```
accumArray :: (Ix a) => (b -> c -> b) -> b
              -> (a,a) -> [(a,c)] -> Array a b
(//)       :: (Ix a) => Array a b -> [(a,b)]
              -> Array a b
accum      :: (Ix a) => (b -> c -> b) -> Array a b
              -> [(a,c)] -> Array a b
ixmap     :: (Ix a, Ix b) => (a,a) -> (a -> b)
              -> Array b c -> Array a c

instance Functor (Array a) where...
instance (Ix a, Eq b) => Eq (Array a b) where...
instance (Ix a, Ord b) => Ord (Array a b) where...
instance (Ix a, Show a, Show b)
    => Show (Array a b) where...
instance (Ix a, Read a, Read b)
    => Read (Array a b) where...
```


Arrays as Abstract Data Type in Haskell (3)

For the definition of the functions and instance declarations of the module `Array`, see:

- Simon Peyton Jones (Ed.). *Haskell 98: Language and Libraries. The Revised Report*. Cambridge University Press, 173-178, 2003. (Chapter 16, Arrays)

Chapter 8.8

References, Further Reading

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

8.1

8.2

8.3

8.4

8.5

8.6






8.7

8.8





Chap. 9

826/196

Chapter 8: Further Reading (1)

-  Manoochehr Azmoodeh. *Abstract Data Types and Algorithms*. Macmillan Education, 1988.
-  Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice-Hall, 2nd edition, 1998. (Chapter 8, Abstract data types)
-  Richard Bird, Philip Wadler. *An Introduction to Functional Programming*. Prentice Hall, 1988. (Chapter 8.4, Abstract types)
-  Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Chapter 10, Arrays, Listen und Stacks)
-  F. Warren Burton. *An Efficient Implementation of FIFO Queues*. Information Processing Letters 14(5):205-206, 1982.



Chapter 8: Further Reading (2)

-  Antonie J.T. Davie. *An Introduction to Functional Programming Systems using Haskell*. Cambridge University Press, 1992. (Chapter 4.5, Abstract Types and Modules)
-  Gerhard Goos, Wolf Zimmermann. *Programmiersprachen*. In *Informatik-Handbuch*, Peter Rechenberg, Gustav Pomberger (Hrsg.), Carl Hanser Verlag, 4. Auflage, 515-562, 2006. (Kapitel 2.1, Methodische Grundlagen: Abstrakte Datentypen, Grundlegende abstrakte Datentypen)
-  John V. Guttag. *Abstract Data Types and the Development of Data Structures*. *Communications of the ACM* 20(6):396-404, 1977.
-  John V. Guttag, James J. Horning. *The Algebra Specification of Abstract Data Types*. *Acta Informatica* 10(1):27-52, 1978.



Chapter 8: Further Reading (3)

-  John V. Guttag, Ellis Horowitz, David R. Musser. *Abstract Data Types and Software Validation*. Communications of the ACM 21(12):1048-1064, 1978.
-  Rachel Harrison. *Abstract Data Types in Standard ML*. J. Wiley, 1993.
-  Chris Okasaki. *Simple and Efficient Purely Functional Queues and Dequeues*. Journal of Functional Programming 5(4):583-592, 1995.
-  Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

Chapter 8: Further Reading (4)

-  Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003. (Kapitel 14.1, Abstrakte Datentypen; Kapitel 14.3, Generische abstrakte Datentypen; Kapitel 14.4, Abstrakte Datentypen in ML und Gofer; Kapitel 15.3, Ein abstrakter Datentyp für Sequenzen)
-  Simon Peyton Jones (Ed.). *Haskell 98: Language and Libraries. The Revised Report*. Cambridge University Press, 2003. (Chapter 16, Arrays)
-  Fethi Rabhi, Guy Lapalme. *Algorithms – A Functional Programming Approach*. Addison-Wesley, 1999. (Chapter 5, Abstract Data Types)

Chapter 8: Further Reading (5)

-  Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley/Pearson, 2nd edition, 1999.
(Chapter 16, Abstract data types)
-  Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley/Pearson, 3rd edition, 2011.
(Chapter 16, Abstract data types)

Chapter 9

Monoids

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

9.1

9.2

9.3

9.4

9.5

Chap. 10

Chap. 11

Chapter 9.1

Motivation

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

9.1

9.2

9.3

9.4

9.5

Chap. 10

Chap. 11

Types

...equipped with an **associative operation**, a **left-unit**, and a **right-unit** like e.g.:

- **lists** with **concatenation** (**++**) and empty list **[]**

$(xs ++ ys) ++ zs = xs ++ (ys ++ zs)$ (associative)

$[] ++ xs = xs$ (left-unit)

$xs ++ [] = xs$ (right-unit)

- **Bool** with **conjunction** (**&&**) and Boolean constant **True**

$(b1 \&\& b2) \&\& b3 = b1 \&\& (b2 \&\& b3)$ (associative)

$True \&\& b = b$ (left-unit)

$b \&\& True = b$ (right-unit)

should be made **instances** of the **type class Monoid**.

Chapter 9.2

The Type Class Monoid

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

9.1

9.2

9.3

9.4

9.5

Chap. 10

Chap. 11

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

<code>mempty 'mappend' x</code>	<code>= x</code>	<code>(MonoL1)</code>
<code>x 'mappend' mempty</code>	<code>= x</code>	<code>(MonoL2)</code>
<code>(x 'mappend' y) 'mappend' z</code>	<code>=</code>	
<code> x 'mappend' (y 'mappend' z)</code>		<code>(MonoL3)</code>

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

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

9.1

9.2

9.3

9.3.1

9.3.2

9.3.3

9.3.4

9.4

839/196

Chapter 9.3.1

The List Monoid

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

9.1

9.2

9.3

9.3.1

9.3.2

9.3.3

9.3.4

9.4

840/196

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"
```

Note

...the `mappend` operation of the `list monoid` is **not** commutative:

```
" Semester " 'mappend' " Holiday "  
              ->> " Semester Holiday "
```

is **different** from:

```
" Holiday " 'mappend' " Semester "  
            ->> " Holiday Semester "
```

Chapter 9.3.2

Numerical Monoids

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

9.1

9.2

9.3

9.3.1

9.3.2

9.3.3

9.3.4

9.4

Numerical Types

...are equipped with more than one associative operation with corresponding unit. E.g.:

- Addition (+) with unit 0.
- Multiplication (*) with unit 1.

To allow more than one monoid instance implementation for

- numerical types

we use `newtype` declarations, and additionally,

- record syntax

to get `selector functions` for free (cf. [Chapter 5.4](#), LVA 185.A03 [Funktionale Programmierung](#)).

The Sum and Product Monoids

1. The **sum monoid** of **numerical** types:

```
newtype Sum a = Sum { getSum :: a }  
  deriving (Eq, Ord, Read, Show, Bounded)
```

```
instance Num a => Monoid (Sum a) where
```

```
  mempty = Sum 0
```

```
  Sum x 'mappend' Sum y = Sum (x+y)
```

2. The **product monoid** of **numerical** types:

```
newtype Product a = Product { getProduct :: a }  
  deriving (Eq, Ord, Read, Show, Bounded)
```

```
instance Num a => Monoid (Product a) where
```

```
  mempty = Product 1
```

```
  Product x 'mappend' Product y = Product (x*y)
```

Proof Obligation: The Monoid Laws

Lemma 9.3.2.1 (Soundness of Sum, Product Monoid)

For every numerical instance of type variable a , the $(\text{Sum } a)$ and $(\text{Product } a)$ instances of Monoid satisfy the three monoid laws MonoL1 , MonoL2 , and MonoL3 .

... $(\text{Sum } a)$ and $(\text{Product } a)$ are thus proper instances of Monoid , the so-called sum and product monoids .

Note: The mappend operations of the sum and product monoid are commutative.

Example: Applying the Monoid Operations

Sum monoid:

```
Sum 3 'mappend' mempty ->> Sum 3
getSum $ Sum 3 'mappend' mempty ->> 3
getSum $ Sum 17 'mappend' Sum 4 ->> 21
getSum . mconcat . map Sum $ [3,7,11] ->> 21
getSum $ Sum 3 'mappend' Sum 7 'mappend' Sum 11 ->> 21
```

Product monoid:

```
Product 3 'mappend' mempty ->> Product 3
getProduct $ Product 3 'mappend' mempty ->> 3
getProduct $ Product 3 'mappend' Product 7 ->> 21
getProduct . mconcat . map Product $ [3,7,11] ->> 231
getProduct $ Product 3 'mappend' Product 7
                    'mappend' Product 11 ->> 231
```


Chapter 9.3.3

Boolean Monoids

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

9.1

9.2

9.3

9.3.1

9.3.2

9.3.3

9.3.4

9.4

849/196

The Type Bool

...is (like numerical types) equipped with more than one associative operation with corresponding unit. E.g.:

- Conjunction (`&&`) with unit `True`.
- Disjunction (`||`) with unit `False`.

To allow more than one monoid instance implementation for

- `Bool`

we use `newtype` declarations, and additionally,

- `record` syntax

to get `selector functions` for free (cf. [Chapter 5.4](#), [LVA 185.A03 Funktionale Programmierung](#)).

The All and Any Monoids

1. The `all` monoid of `Bool`:

```
newtype All = All {getAll :: Bool}
  deriving (Eq, Ord, Read, Show, Bounded)

instance Monoid All where
  mempty = All True
  All x 'mappend' All y = All (x && y)
  -- 'All' because True if every argument is true.
```

2. The `any` monoid of `Bool`:

```
newtype Any = Any {getAny :: Bool}
  deriving (Eq, Ord, Read, Show, Bounded)

instance Monoid Any where
  mempty = Any False
  Any x 'mappend' Any y = Any (x || y)
  -- 'Any' because True if some argument is true.
```

Proof Obligation: The Monoid Laws

Lemma 9.3.3.1 (Soundness of All, Any Monoid)

The `All` and `Any` instances of `Monoid` satisfy the three monoid laws `MonoL1`, `MonoL2`, and `MonoL3`.

...`All` and `Any` are thus proper instances of `Monoid`, the so-called `all` and `any monoids`.

Note: The `mappend` operations of the `all` and `any monoid` are commutative.

Example: Applying the Monoid Operations

All monoid:

```
All True 'mappend' mempty ->> All True
getAll $ All True 'mappend' mempty ->> True
getAll $ All True 'mappend' All False ->> False
getAll . mconcat . map All $ [False,True,True,False]
                                     ->> False
```

Any monoid:

```
Any True 'mappend' Any False ->> Any True
getAny $ Any True 'mappend' Any False ->> True
getAny $ mempty 'mappend' Any False ->> False
getAny . mconcat . map Any $ [False,True,False,False]
                                     ->> True
```

Chapter 9.3.4

The Ordering Monoid

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

9.1

9.2

9.3

9.3.1

9.3.2

9.3.3

9.3.4

9.4

854/196

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 $\rightarrow\rightarrow$ LT

GT `'mappend'` LT $\rightarrow\rightarrow$ 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 `prioritized`.

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

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

9.1

9.2

9.3

9.4

9.5

Chap. 10

Chap. 11

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 obvious and 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

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

9.1

9.2

9.3




9.4

9.5

Chap. 10

Chap. 11

Chapter 9: 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)
-  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 10

Functors

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

10.1

10.2

10.3

10.4

Chap. 11

Chapter 10.1

Motivation

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

10.1

10.2

10.3

10.4

Chap. 11

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

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

10.1

10.2

10.3

10.4

Chap. 11

872/196

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

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

(id over-loaded!)

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

(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 1 = "January"
...
g 12 = "December"

fmap      g      ->>
  :: Int -> String

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

fmap      g'     ->>
  :: 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: fmap g (M 8) ->> fmap (M (g 8)) ->> M "August"
          :: Month Int      :: Month String
```

Revisiting Type Classes

Recall the definition of the `type class Monoid` for comparison with the `type constructor class Functor`:

```
class Monoid m where
  empty    :: m
  mappend  :: m -> m -> m
  mconcat  :: [m] -> m
  mconcat  = foldr mappend empty
```

Note:

- The argument `m` of `Monoid` is a `type variable`. Functions declared in `Monoid` operate on values of type `m`; `m` itself does not operate on anything.
- This holds for every type class; recall the definitions of type classes we considered so far: `Eq`, `Ord`, `Num`, `Enum`, . . .

Type Classes vs. Type Constructor Classes

Type classes and type constructor classes are conceptually equal but differ in the type of their instances:

- Type constructor classes (`Foldable`, `Functor`, `Monad`, `Arrow`, ...) have type constructors as instances:
 - `Tree`, `[]`, `(,)`, `(->)`, ...
- Type classes (`Eq`, `Ord`, `Num`, `Monoid`, ...) have types as instances:
 - `Tree a`, `[] a`, `(,) a b`, `(->) a b`, ...

Type constructors are maps

- constructing new types from given ones.

Examples: Tuple constructors `(,)`, `(,,)`, `(,,,)`; list constructor `[]`; map constructor `(->)`; input/output constructor `IO`, ...

Example: Two Functor Instances (1)

...making the 1-ary type constructors `[]` and `Tree` for lists and trees, respectively, instances of `Functor`:

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

instance Functor Tree where
  fmap g (Leaf v) = Leaf (g v)
  fmap g (Node v l r)
    = Node (g v) (fmap g l) (fmap g r)
```

Note:

- The symbol `[]` is used above in two rôles (over-loaded), as a
 - type constructor in: `instance Functor [] where...`
 - value of some list type in: `fmap g [] = []`.
- The declarations `instance Functor [a] where...`, `instance Functor (Tree a) where...` would not be correct, since `[a]` and `(Tree a)` denote types, no type constructors.

Example: Proof Obligations f. Inst. [], Tree (2)

We have:

Lemma 10.2.1 (Functor Laws for [] and Tree)

The [] and Tree instances of Functor satisfy the two functor laws FL1 and FL2, respectively.

... [] and Tree are thus proper instances of Functor, the list and tree functors.

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

10.1

10.2

10.3

10.4

Chap. 11

Example: Alternative Inst. Impl. f. [], Tree (3)

The instance declarations for [] and Tree could have been given equivalently and more concisely as follows:

```
instance Functor [] where
  fmap = mapL           -- user-defined mapL

instance Functor [] where
  fmap = map            -- predefined map

instance Functor Tree where
  fmap = mapT           -- user-defined mapT
```

Example: Applying the Functor Operation (4)

List functor:

```
ms = [1..5]
fmap (*2) ms ->> [2,4,6,8,10]
fmap (^3) ms ->> [1,8,27,64,125]
fmap (3^) ms ->> [3,9,27,81,243]
```

Tree functor:

```
t = Node 2 (Node 3 (Leaf 5) (Leaf 7)) (Leaf 11)
fmap (*2) t
->> Node 4 (Node 6 (Leaf 10) (Leaf 14)) (Leaf 22)
fmap (^3) t
->> Node 8 (Node 27 (Leaf 125) (Leaf 343))
      (Leaf 1331)
fmap (3^) t
->> Node 9 (Node 27 (Leaf 243) (Leaf 2187))
      (Leaf 177147)
```

Instances of Functor

...can be pre-defined and user-defined 1-ary type constructors.

Predefined type constructors of different arity are e.g.:

- 1-ary type constructors: `[]`, `Maybe`, `IO`, ...
- 2-ary type constructors: `(,)`, `(->)`, `Either`, ...
- 3-ary type constructors: `(,,)`, ...
- 4-ary type constructors: `(,,,)`, ...
- ...

Note:

- Only 1-ary type constructors are instance candidates of `Functor`. This can also be partially evaluated type constructors of higher arity, e.g., `(Either a)`, `((->) r)`.
- Considering types as 0-ary type constructors shows the conceptual coincidence of type classes and type constructor classes.

Recall

...the following are **equivalent**:

- (a,b) is equivalent to $(,) a b$
 (a,b,c) is equivalent to $(, ,) a b c$, etc.
- $[a]$ is equivalent to $[] a$
- $a \rightarrow b$ is equivalent to $(\rightarrow) a b$
- $T a b$ is equivalent to $((T a) b)$ (i.e., associativity to the left as for function application)

Example

...the signatures of

```
fac :: Int -> Int
```

```
list2pair :: [a] -> (a,a)
```

can thus **equivalently** be written as:

```
fac :: (->) Int Int
```

```
list2pair :: [] a -> (a,a)
```

```
list2pair :: [a] -> (,) a a
```

```
list2pair :: (->) [a] (a,a)
```

```
list2pair :: [] a -> (,) a a
```

```
...
```

```
list2pair :: (->) ([] a) ((,) a a)
```

...we are more familiar, however, with the 'classical' forms, which may thus appear more easily **comprehensible**.

Chapter 10.3

Functor Examples

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

10.1

10.2

10.3

10.3.1

10.3.2

10.3.3

886/196

Chapter 10.3.1

The Identity Functor

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

10.1

10.2

10.3

10.3.1

10.3.2

10.3.3

887/196

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 *identity functor*.

Chapter 10.3.2

The List Functor

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

10.1

10.2

10.3

10.3.1

10.3.2

10.3.3

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

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

10.1

10.2

10.3

10.3.1

10.3.2

10.3.3

891/196

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
```

Anti-Example: Invalid Functor Instance (1)

...consider type `Maybe_with_counter`, which is almost like `Maybe` but whose `Just` values contain an additional `Int` value which shall be used for counting the number of applications of `fmap`:

```
data Maybe_with_counter a
  = Nothing_wc | Just_wc Int a deriving Show
```

...making `Maybe_with_counter` an instance of `Functor`:

```
instance Functor Maybe_with_counter where
  fmap g Nothing_wc = Nothing_wc
  fmap g (Just_wc counter x) = Just_wc (counter+1) (g x)
```

We show: The `Maybe_with_counter` instance of `Functor`
– violates functor law `FL1`.

This means, `Maybe_with_counter` is an invalid instance of `Functor` and thus an anti-example.

Anti-Example: Invalid Functor Instance (2)

```
Nothing_wc      :: Maybe_with_counter a
Just_wc 0 "fun"  :: Maybe_with_counter [Char]
Just_wc 100 [1,2,3] :: Maybe_with_counter [Int]

Nothing_wc      ->> Nothing_wc
Just_wc 0 "fun"  ->> Just_wc 0 "fun"
Just_wc 100 [1,2,3] ->> Just_wc 100 [1,2,3]

fmap (++ "prog") Nothing_wc
  ->> Nothing_wc
fmap (++ "prog") (Just_wc 0 "fun")
  ->> Just_wc 1 "funprog"
fmap (++ "prog") (fmap (++ " ") (Just_wc 0 "fun"))
  ->> Just_wc 2 "fun prog"
```

...while everything is *fine* with these examples...

Anti-Example: Invalid Functor Instance (3)

...evaluating

```
fmap id (Just_wc 0 "fun")
```

and

```
id (Just_wc 0 "fun")
```

yield **different** values:

```
fmap id (Just_wc 0 "fun") ->> Just_wc 1 "fun"  
id (Just_wc 0 "fun")      ->> Just_wc 0 "fun"
```

Hence, functor law **FL1** is violated: Equality `fmap id = id` does not hold for the `Maybe_with_counter` instance. Thus:

Lemma 10.3.3.2 (Invalid Instance)

`Maybe_with_counter` is not a valid instance of `Functor`.

Chapter 10.3.4

The Either Functor

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

10.1

10.2

10.3

10.3.1

10.3.2

10.3.3

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"
```

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

10.1

10.2

10.3

10.3.1

10.3.2

10.3.3

900/196

Exercise 10.3.4.2

Consider the following `instance declaration` for `(Either a)`:

```
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 (g x)
```

Is this instance declaration valid, i.e., does it satisfy the functor laws? Is it meaningful?

Think about the constraints the above instance declaration imposes on the types which are eligible for `a` and `b`.

Chapter 10.3.5

The Map Functor

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

10.1

10.2

10.3

10.3.1

10.3.2

10.3.3

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 $((\rightarrow) \text{ d})$ instance of `Functor` satisfies the two functor laws `FL1` and `FL2`.

... $((\rightarrow) \text{ d})$ 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 = (.)
```

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

10.1

10.2

10.3

10.3.1

10.3.2

10.3.3

906/196

Notes on the Map Functor

...for the map functor $((\rightarrow) d)$ the type of the generic operation `fmap` of the type constructor class `Functor`

$$\text{fmap} :: (\text{Functor } f) \Rightarrow (a \rightarrow b) \rightarrow f a \rightarrow f b$$

specializes to:

$$\text{fmap} :: (a \rightarrow b) \rightarrow (((\rightarrow) d) a) \rightarrow (((\rightarrow) d) b)$$

Using infix notation for (\rightarrow) , this can equivalently be written as:

$$\text{fmap} :: (a \rightarrow b) \rightarrow (d \rightarrow a) \rightarrow (d \rightarrow b)$$

where `fmap` can be implemented by:

$$\underbrace{\text{fmap } g}_{:: a \rightarrow b} \underbrace{h}_{:: d \rightarrow a} = \underbrace{(g \cdot h)}_{:: (a \rightarrow b) \rightarrow (d \rightarrow a) \rightarrow (d \rightarrow b)}$$

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

- $(a \rightarrow b)$

to type

- $(f\ a \rightarrow f\ b)$

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

$$\text{fmap} :: (\text{Functor } f) \Rightarrow (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b$$

can be thought of in two ways. As a map which takes a map $g :: a \rightarrow b$ and

1. lifts g to a new function $h :: f\ a \rightarrow f\ b$ operating on functor values \rightsquigarrow **curried view**.
2. a functor value $v :: f\ a$ and maps g over $v \rightsquigarrow$ **uncurried view**.

Exercise 10.3.5.2

Following the example of the map functor, provide (most general) type information for the below instances of `Functor`:

1. Identity
2. Maybe
3. List
4. Input/Output
5. Either
6. Tree (cf. Chapter 10.2)

Chapter 10.3.6

The Input/Output Functor

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

10.1

10.2

10.3

10.3.1

10.3.2

10.3.3

914/196

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

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

10.1




10.2

10.3

10.4

Chap. 11

Chapter 10: Further Reading (1)

-  Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000. (Chapter 18.1, The Functor Class)
-  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)
-  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)

Chapter 10: Further Reading (2)

-  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)

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

10.1

10.2

10.3

10.4

Chap. 11

920/196

Chapter 11

Applicative Functors

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

11.1

11.2

11.3

Chap. 12

Chapter 11.1

The Type Constructor Class Applicative

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

11.1

11.2

11.3

Chap. 12

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 applicative 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

$$\underbrace{\underbrace{\text{pure id}}_{:: a \rightarrow a}}_{:: f (a \rightarrow a)} \quad \underbrace{\langle * \rangle v}_{:: f a} = \underbrace{v}_{:: f a} \quad (\text{AL1})$$

$$\underbrace{\underbrace{\text{pure g}}_{:: a \rightarrow b}}_{:: f (a \rightarrow b)} \quad \underbrace{\langle * \rangle \underbrace{\text{pure x}}_{:: a}}_{:: f a} = \underbrace{\text{pure (g x)}}_{:: f b} \quad (\text{AL3})$$

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]
```


Exercise 11.1.1: Type Correctness of Laws

Show that the applicative laws **AL2** and **AL4** are type correct. Annotate the laws with the (most general) type information applying:

$$\text{pure } (.) \langle * \rangle u \langle * \rangle v \langle * \rangle w = u \langle * \rangle (v \langle * \rangle w) \quad (\text{AL2})$$

$$u \langle * \rangle \text{pure } y = \text{pure } (\$ y) \langle * \rangle u \quad (\text{AL4})$$

Chapter 11.2

Applicative Examples

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

11.1

11.2

11.2.1

11.2.2

Chapter 11.2.1

The Identity Applicative

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

11.1

11.2

11.2.1

11.2.2

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
```

```
    pure      =      Id
  :: a -> Id a  :: a -> Id a
```

```
    Id g      <*>    Id x      =    Id (g      x)
  :: (a -> b)  :: a          :: a -> b  :: a
  :: Id (a -> b)  :: Id a      :: b
  :: Id b
```

Chapter 11.2.2

The List Applicative

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

11.1

11.2

11.2.1

11.2.2

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

Annotations for the first equation:

- `pure` is annotated with `a -> [] a`
- `x` is annotated with `a`
- `[x]` is annotated with `[] a`

Annotations for the second equation:

- `gs` is annotated with `[] (a -> b)`
- `xs` is annotated with `[] a`
- `g` is annotated with `a -> b`
- `x` is annotated with `a`
- The list comprehension `[g x | g <- gs, x <- xs]` is annotated with `[] 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)

```
(++) <$> ["yes","no","ok"] <*> ["?",".", "!"]
->> (fmap (++) ["yes","no","ok"]) <*> ["?",".", "!"]
->> [("yes"++), ("no"++), ("ok"++)] <*> ["?",".", "!"]
->> [f x | f <- [("yes"++), ("no"++), ("ok"++)],
      x <- ["?",".", "!"] ]
->> ["yes?", "yes.", "yes!", "no?", "no.", "no!",
      "ok?", "ok.", "ok!"]

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 preceding 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

The Maybe Applicative

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

11.1

11.2

11.2.1

11.2.2

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

```
    pure           = Just
  :: a -> Maybe a  :: a -> Maybe a
```

```
    Nothing <*> _ = Nothing
  :: Maybe (a -> b)
   :: Maybe a
   :: Maybe b
```

```
    (Just g) <*> something = fmap g something
  :: Maybe (a -> b)
   :: Maybe a
   :: a -> b
   :: Maybe a
   :: 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
```

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

11.1

11.2

11.2.1

11.2.2

Example: Applying the Applicative Operations (3)

```
pure (+) <*> Nothing <*> Just 5
->> Just (+) <*> Nothing <*> Just 5
->> (fmap (+) Nothing) <*> Just 5
->> Nothing <*> Just 5
->> Nothing
```

Note: The operator (`<*>`) is left-associative, i.e.:

```
pure (+) <*> Just 3 <*> Just 5 =
      (pure (+) <*> Just 3) <*> Just 5
```

Chapter 11.2.4

The Either Applicative

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

11.1

11.2

11.2.1

11.2.2

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

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

11.1

11.2

11.2.1

11.2.2

The Map Applicative

...making the 1-ary type constructor $((\rightarrow) \text{ d})$ 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 $((\rightarrow) \text{ d})$ instance of `Applicative` satisfies the four applicative laws `AL1`, `AL2`, `AL3`, and `AL4`.

... $((\rightarrow) \text{ d})$ 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)

$\underbrace{\text{:: } a}$ $\underbrace{\text{:: } d}$ $\underbrace{\text{:: } a}$

$\underbrace{\text{:: } ((->) d) a}$

$\underbrace{\text{:: } ((->) d) (a \rightarrow b)}^g$ `<*>` $\underbrace{\text{:: } ((->) d) a}^h$ = $\backslash x \rightarrow g\ x\ (h\ x)$

$\underbrace{\text{:: } d \rightarrow (a \rightarrow b)}$ $\underbrace{\text{:: } d \rightarrow a}$

$\underbrace{\text{:: } d}$ $\underbrace{\text{:: } d}$ $\underbrace{\text{:: } d}$

$\underbrace{\text{:: } a}$

$\underbrace{\text{:: } b}$

$\underbrace{\text{:: } d \rightarrow b}$

$\underbrace{\text{:: } ((->) d) b}$

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
```

Exercise 11.2.5.2

...computing with the `map` applicative.

Complete the stepwise evaluation of the below example:

```
(\x y z -> [x,y,z]) <$> (+3) <*> (*2) <*> (/2) $ 5
->> (fmap (\x y z -> [x,y,z]) (+3)) <*> (*2) <*> (/2) $ 5
->> ((\x y z -> [x,y,z]) . (+3)) <*> (*2) <*> (/2) $ 5
->> ...
->> [8.0,10.0,2.5]
```


Chapter 11.2.6

The Ziplist Applicative

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

11.1

11.2

11.2.1

11.2.2

The Ziplist Applicative

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

```
newtype ZipList a = ZL [a]
  -- the newtype declaration is required since [] can
  -- not be made a 2nd time an instance of Applicative

instance Applicative ZipList where
  pure x          = ZL (repeat x)
  ZL gs <*> ZL xs = ZL (zipWith (\g x -> g x) gs xs)
```

Proof obligation: The applicative laws

Lemma 11.2.6.1 (Soundness of Ziplist Applicative)

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

...`ZipList` is thus a proper instance of `Applicative`, the so-called `ziplist applicative`.

Intuitively

...<*> applies the first function to the first value, the second function to the second value, and so on.

As a reminder:

```
repeat :: a -> [a]
```

```
repeat x = x : repeat x -- generates stream [x,.x,..
```

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

```
zipWith _ [] _ = []
```

```
zipWith _ _ [] = []
```

```
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
```

The ZipList 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 ZipList where
```

```
pure x = ZL (repeat x)
```

```
  :: a
  :: [a]
  :: ZipList a
```

```
  ZL gs    <*>    ZL xs
  :: ZipList (a -> b) :: ZipList a
```

```
  = ZL (zipWith (\g x -> g x)
             :: (a->b,a) -> b
             :: ((a->b) -> a -> b)
             :: [b]
             :: ZipList b
             gs
             xs)
  :: [(a->b)]
  :: [a]
```

Example: Applying the Applicative Operations

```
getZipList $ (+) <$> ZL [1,2,3] <*> ZL [100,100,100]
->> getZipList $ (fmap (+) ZL [1,2,3]) <*> ZL [100,100,100]
->> getZipList $ ZL [(1+),(2+),(3+)] <*> ZL [100,100,100]
->> getZipList $ ZL [1+100,2+100,3+100]
->> getZipList $ ZL [101,102,103]
->> [101,102,103]
```

```
getZipList $ (+) <$> ZL [1,2,3] <*> ZL [100,100..]
->> getZipList $ (fmap (+) ZL [1,2,3]) <*> ZL [100,100,..]
->> getZipList $ ZL [(1+),(2+),(3+)] <*> ZL [100,100,..]
->> getZipList $ ZL [1+100,2+100,3+100]
->> [101,102,103]
```

```
getZipList $ max <$> ZL [1,2,3,4,5,3] <*> ZL [5,3,1,2]
->> ... ->> [5,3,3,4]
```

```
getZipList $ (,) <$> ZL "dog" <*> ZL "cat" <*> ZL "rat"
->> ... ->> [( 'd','c','r'), ('o','a','a'), ('g','t','t')]
```

Chapter 11.2.7

The Input/Output Applicative

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

11.1

11.2

11.2.1

11.2.2

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)

The **IO** 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

```
    pure       = return
    (<*>) a b = do
      g <- a
      x <- b
      return (g x)
```

The diagram illustrates the type signatures for the IO Applicative instance. Brackets are used to group the types in the definitions of `pure` and `<*>` to show how they relate to the `do` notation and the `return` function. In the `do` block, `g` is of type `a -> b` and `x` is of type `a`. The `return` function takes these two arguments and returns a value of type `a`. The final result of the `do` block is of type `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

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

11.1

11.2

11.3

Chap. 12

Chapter 11: Further Reading



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

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

11.1

11.2

11.3

Chap. 12

Chapter 12

Monads

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

12.1

12.2

12.3

Chapter 12.1

Motivation

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

12.1

12.2

12.3

965/196

Monad: The Mystic Type Constructor Class

```
class Monad m where
  (>=>) :: m a -> (a -> m b) -> m b
  ...
```

...is there any reason for the [mystic aura](#) around [monads](#)?

Compare [monad](#) with other type constructor classes:

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

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

Monad: The Mystic Type Constructor Class

```
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  return :: a -> m a
  (>>)   :: m a -> m b -> m b
  fail   :: String -> m a
  c >> k = c >>= \_ -> k
  fail s = error s
```

For comparison repeated:

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

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

Does the Name Itself

...give reason for a **kind of mysticism?**

Monad, derived from Greek *monas*, means:

- **unit, unity** (in German: **Eins, Einheit**).

Does the Usage of Monads

...(in other fields) give reason for a kind of mysticism?

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

12.1

12.2

12.3

969/196

Monads in Philosophy

Gottfried Wilhelm Leibniz (* 1646 in Leipzig; † 1716 in Hannover) used the **monad** notion as a counterpart of

- ‘atom’ denoting like atom ‘something indivisible’

to ‘solve’ (more accurate possibly: tackle) the so-called

- **body-soul problem** (in German: **Leib-Seele-Problem**)

evolving from the **body-soul dualism** in the the classical formulation of **René Descartes** (* 1596 in La Haye 50 km south of Tours, today Descartes; † 1650 in Stockholm).

Monads in Category Theory

Eugenio Moggi introduced the monad notion to

- category theory

and used it for describing the

- semantics of programming languages.

in the realm of

- programming languages theory.



Eugenio Moggi. Computational Lambda Calculus and Monads. In Proceedings of the 4th Annual IEEE Symposium on Logic in Computer Science (LICS'89), 14-23, 1989.

Monads in Philosophy, Category Theory

Monads in Leibniz' Philosophy:

Definition (Gottfried Wilhelm Leibniz, 1714)

[Monadology, Paragraph 1]: The **monad** we want to talk about here is nothing else as a simple substance (German: Substanz), which is contained in the composite matter (German: Zusammengesetztes); simple means as much as: to be without parts.

Monads in Category Theory (cf. Saunders Mac Lane, 1971):

Definition (Eugenio Moggi, 1989)

[LICS'89]: A **monad** over a category \mathcal{C} is a triple (T, η, μ) , where $T : \mathcal{C} \rightarrow \mathcal{C}$ is a functor, $\eta : Id_{\mathcal{C}} \rightarrow T$ and $\mu : T^2 \rightarrow T$ are natural transformations and the following equations hold:

$$\begin{aligned}\mu_{TA}; \mu_A &= T(\mu_a); \mu_A \\ \eta_{TA}; \mu_A &= id_{TA} = T(\eta_A); \mu_A\end{aligned}$$

... "a monad is a monoid in the category of endofunctors."

Monads in Functional Programming

...the **monad** notion became particularly popular in the field of **functional programming** (Philip Wadler, 1992) because (**Has-kell-style**) **monads**

- allow to introduce some useful **aspects of imperative programming** such as sequencing into functional programming
- are well suited to smoothly integrate **input/output** into functional programming, as well as many other programming tasks and domains
- provide a suitable **interface** between **functional programming** and **programming paradigms with side effects**, in particular, imperative and object-oriented programming

...**without breaking** the **functional paradigm!**

These Capabilities let Monads

...appear to be a **Suisse Knife** of **Functional Programming!**

Monadic programming seems/is perfect for problems involving:

- **Global state**
 - Updating data during computation is often simpler than making all data dependencies explicit (the **state monad**).
- **Huge data structures**
 - No need for replicating a data structure that is not needed otherwise.
- **Exception and error handling**
 - The **Maybe monad**.
- ...
- **Side-effects, explicit sequencing, evaluation orders**
 - Canonical scenario: **Input/output operations** (the **IO monad**).

Good to Know

...the **monad** notion in **functional programming** lost its links to those in **philosophy** and **category theory** (almost) completely if there have been ever any tied ones, and hence, everything which might or might be considered a mystery or a miracle.

Rather than introducing a mystery, **monads** and **monadic programming** close a 'functional gap' between

- **function application**
- **sequential function composition**
- **functorial mapping**

Comparing Functorial and Monadic Mapping

► Functorial mapping:

```
fmap :: (Functor f) => (a -> b) -> f a -> f b  
fmap k c = ... "(unpack, map, pack)"
```

```
(<*>) :: (Applicative f) => f (a -> b) -> f a -> f b  
(<*>) k c = ... "(unpack, unpack, map, pack)"
```

► Monadic mapping and sequencing:

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b  
(>>=) c k = ... "(unpack, map, repeat >>=)"
```


Why and How Monadic Sequencing? (1)

The associativity of ($\gg=$) allows writing

$$((((c \gg= k) \gg= k1) \gg= k2) \gg= k3) \gg= k4)$$

more concisely:

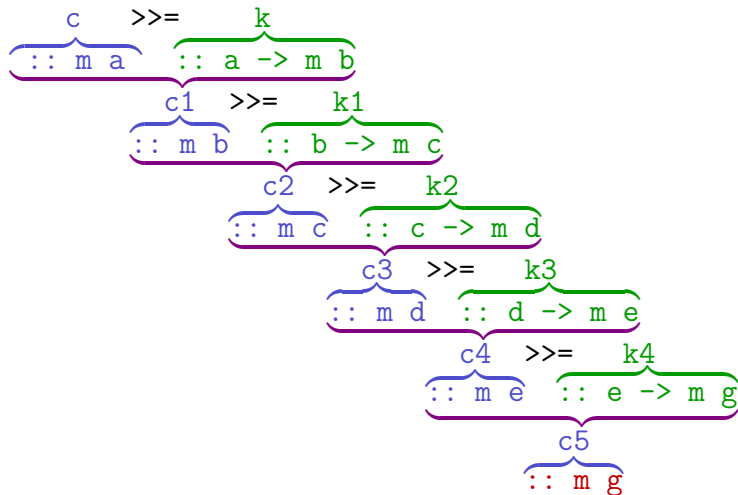
$$c \gg= k \gg= k1 \gg= k2 \gg= k3 \gg= k4$$

Double-checking types yields:

$$\underbrace{c}_{:: m a} \gg= \underbrace{k}_{:: a \rightarrow m b} \gg= \underbrace{k1}_{:: b \rightarrow m c} \gg= \underbrace{k2}_{:: c \rightarrow m d} \gg= \underbrace{k3}_{:: d \rightarrow m e} \gg= \underbrace{k4}_{:: e \rightarrow m g}$$

$$\underbrace{\quad}_{:: m b}$$
$$\underbrace{\quad}_{:: m c}$$
$$\underbrace{\quad}_{:: m d}$$
$$\underbrace{\quad}_{:: m e}$$
$$\underbrace{\quad}_{:: m g}$$

Why and How Monadic Sequencing? (2)

$$\underbrace{c}_{:: m a} \gg= \underbrace{k}_{:: a \rightarrow m b} \gg= \underbrace{k1}_{:: b \rightarrow m c} \gg= \underbrace{k2}_{:: c \rightarrow m d} \gg= \underbrace{k3}_{:: d \rightarrow m e} \gg= \underbrace{k4}_{:: e \rightarrow m g} :: m g$$


Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

12.1

12.2

12.3

978/196

Why and How Monadic Sequencing? (3)

$c \gg= k \gg= k1 \gg= k2 \gg= k3 \gg= k4 :: m g$

$\underbrace{c}_{:: m a} \quad \underbrace{\gg= k}_{:: a \rightarrow m b} \quad \underbrace{\gg= k1}_{:: b \rightarrow m c} \quad \underbrace{\gg= k2}_{:: c \rightarrow m d} \quad \underbrace{\gg= k3}_{:: d \rightarrow m e} \quad \underbrace{\gg= k4}_{:: e \rightarrow m g}$

$c \gg= k \gg= k1 \gg= k2 \gg= k3 \gg= k4$
 $->> c1 \gg= k1 \gg= k2 \gg= k3 \gg= k4$
 $\quad ->> c2 \gg= k2 \gg= k3 \gg= k4$
 $\quad \quad ->> c3 \gg= k3 \gg= k4$
 $\quad \quad \quad ->> c4 \gg= k4$
 $\quad \quad \quad \quad ->> c5 :: m g$

Why so Differently?

...why do functional composition and monadic sequencing look so differently?

Functional Composition:

$$\begin{aligned} (\cdot) &:: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c) \\ (g \cdot f) \ x &= g (f \ x) \quad \text{-- } (g \cdot f) = \lambda y \rightarrow g (f \ y) \end{aligned}$$

Monadic Sequencing:

$$\begin{aligned} (>>=) &:: (\text{Monad } m) \Rightarrow m \ a \rightarrow (a \rightarrow m \ b) \rightarrow m \ b \\ (>>=) \ c \ k &= k \ \text{"unpack } c\text{"} \end{aligned}$$

Or (using infix notation):

$$\begin{aligned} (>>=) &:: (\text{Monad } m) \Rightarrow m \ a \rightarrow (a \rightarrow m \ b) \rightarrow m \ b \\ c \ >>= \ k &= k \ \text{"unpack } c\text{"} \end{aligned}$$

The different Appearance is an Artifact!

The standard operator $(.)$ for **functional composition**:

$(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$

$(g . f) x = g (f x)$

...enables sequences of function applications applied **R2L**:

$(k . (\dots . (h . (g . f)) \dots)) x$
 $\rightarrow\rightarrow k (\dots (h (g (f x)))) \dots)$

We can define a dual operator $(;)$ for **function composition**:

$(;) :: (a \rightarrow b) \rightarrow (b \rightarrow c) \rightarrow (a \rightarrow c)$

$(f ; g) = (g . f)$

...enabling sequences of function applications applied **L2R**:

$((\dots((f ; g) ; h) ; \dots) ; k) x$
 $\rightarrow\rightarrow k (\dots (h (g (f x)))) \dots)$

The Operator (;)

...suggests introducing another operator ($\gg;$):

$(\gg;) :: a \rightarrow (a \rightarrow b) \rightarrow b$

$x \gg; f = f\ x$

enabling also sequences of function applications applied [L2R](#):

$(\dots(((x \gg; f) \gg; f_1) \gg; f_2) \gg; \dots \gg; f_n)$

$\hat{=} x \gg; f \gg; f_1 \gg; f_2 \gg; \dots \gg; f_n$

...where a value x is fed to the sequence of functions which are then applied one after the other to x (resp. its resulting images).

Opposing and Comparing

...non-monadic ($>>;$) and monadic ($>>=$) sequencing:

1. Ordinary Functional Sequencing from left to right:

$(>>;) :: a \rightarrow (a \rightarrow b) \rightarrow b$

$x >>; f = f\ x$

...enables L2R application sequences of the form:

$x >>; f >>; f1 >>; f2 >>; f3 >>; \dots >>; fn$

2. Monadic Functional Sequencing from left to right:

$(>>=) :: (\text{Monad } m) \Rightarrow m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$

$c >>= k = k$ “unpack c ”

...enables L2R application sequences of the form:

$c >>= k >>= k1 >>= k2 >>= k3 >>= \dots >>= kn$

...reveals: There is **no mystery at all!**

Summing up

...the difference between $(\gg;)$ and $(\gg=)$ is a technical one:

$(\gg;) :: a \rightarrow (a \rightarrow b) \rightarrow b$

$x \gg; f = f x$

- The second argument f of $(\gg;)$ can directly be applied to its first argument x .
- This means, $(\gg;)$ is *parametric polymorphic*.

$(\gg=) :: (\text{Monad } m) \Rightarrow m a \rightarrow (a \rightarrow m b) \rightarrow m b$

$c \gg= k = k \text{ "unpack } c"$

- The first argument c of $(\gg=)$ needs to be *unpacked* before its second argument k can be applied to it.
- The *unpacking* of the first argument is type specific.
- Hence, $(\gg=)$ can only be *ad hoc polymorphic*, and must be a *member function* of some *type (constructor) class*.
- This *type constructor class* is (called) *Monad*.

...again, except of this difference, *no mystery!*

Chapter 12.2

The Type Constructor Class Monad

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

12.1

12.2

12.3

985/196

The Type Constructor Class Monad

...monads are instances of the type constructor class `Monad` obeying the monad laws:

Type Constructor Class Monad

```
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return :: a -> m a
  (>>)  :: m a -> m b -> m b
  fail  :: String -> m a
  c >> k = c >>= \_ -> k
  fail s = error s
```

Monad Laws

`return x >>= f` = `f x` (ML1)

`c >>= return` = `c` (ML2)

`c >>= (\x -> (f x) >>= g)` = `(c >>= f) >>= g` (ML3)

Note

...monads must be 1-ary type constructors (like functors).

Intuitively, the monad laws require from (proper) monad instances:

- return is unit of ($\gg=$), i.e., it must pass its argument without any other effect (just as function pure of type constructor class `Applicative`) (ML1, ML2).
- ($\gg=$) is associative, i.e., sequencings given by ($\gg=$) must not depend on how they are bracketed (ML3).

Programmer obligation

- Programmers must prove that their instances of `Monad` satisfy the monad laws.

Note: Sequence operator ($\gg=$): Read as `bind` (Paul Hudak) or `then` (Simon Thompson). Sequence operator (\gg): Derived from ($\gg=$), read as `sequence` (Paul Hudak).

Type Constructor Class Monad in more Detail

class Monad m where

-- 'Primary' functions (relevant for every monad)

return :: a -> m a -- Value 'lifting:' Ma-

-- king a monadic value

(>>=) :: m a -> (a -> m b) -> m b -- Sequencing

-- 'Secondary' functions (relevant for some monads)

fail :: String -> m a -- Error handling

(>>) :: m a -> m b -> m b -- Simplified sequencing

-- Default implementations

fail s = error s -- Failing computation:

$\underbrace{\text{:: String}}_{\text{:: m a}} = \underbrace{\text{error s}}_{\text{:: String}}_{\text{:: m a}}$ -- Outputting s as error

$\underbrace{\text{:: m a}}_{\text{:: m a}}$ -- error message

$\underbrace{\text{c}}_{\text{:: m a}} \underbrace{\text{>> k}}_{\text{:: m b}} = \underbrace{\text{c}}_{\text{:: m a}} \underbrace{\text{>>= _ -> k}}_{\text{:: a -> m b}}_{\text{:: m b}}$

The Monad Laws in more Detail

...with added type information:

$$\underbrace{\underbrace{\text{return } x}_{:: a \rightarrow m a} \gg=}_{:: m a} \underbrace{f}_{:: a \rightarrow m b} = \underbrace{f}_{:: a \rightarrow m b} \gg= \underbrace{x}_{:: a} \quad (\text{ML1})$$

The diagram shows the first monad law (ML1) with detailed type annotations. On the left, the expression `return x >>= f` is annotated with types: `return x` has type `a -> m a`, `x` has type `a`, and `f` has type `a -> m b`. Brackets group these into a larger expression with type `m a`. On the right, the expression `f >>= x` is annotated with types: `f` has type `a -> m b` and `x` has type `a`. Brackets group these into a larger expression with type `m b`. The two sides are shown to be equal, labeled (ML1).

$$\underbrace{c}_{:: m a} \gg= \underbrace{\text{return } c}_{:: a \rightarrow m a} = c \quad (\text{ML2})$$

The diagram shows the second monad law (ML2) with detailed type annotations. On the left, the expression `c >>= return c` is annotated with types: `c` has type `m a` and `return c` has type `a -> m a`. Brackets group these into a larger expression with type `m a`. On the right, the expression `c` is annotated with type `m a`. The two sides are shown to be equal, labeled (ML2).

Exercise 12.2.1: Type Correctness of Laws

Show that the monad law **ML3** is type correct. Annotate the law with the (most general) type information applying:

$$c \gg= (\backslash x \rightarrow (f \ x) \gg= g) = (c \gg= f) \gg= g \quad \text{(ML3)}$$

Associativity of (\gg)

Lemma 12.2.2 (Associativity of (\gg))

Monotonicity of ($\gg=$) for some monad m implies that the default implementation of (\gg) is associative, too, i.e.:

$$c1 \gg (c2 \gg c3) = (c1 \gg c2) \gg c3$$

Compared with the associativity statement of [Lemma 12.2.2](#) for (\gg), the left-hand side of (ML3) requiring the associativity of ($\gg=$) looks 'ugly':

$$c \gg= (\lambda x \rightarrow (f x) \gg= g) = (c \gg= f) \gg= g \quad \text{(ML3)}$$

To improve on this, we introduce a new operator ($\gg@$):

$$\begin{aligned} (\gg@) &:: \text{Monad } m \Rightarrow (a \rightarrow m b) \rightarrow (b \rightarrow m c) \\ &\quad \rightarrow (a \rightarrow m c) \end{aligned}$$

$$f \gg@ g = \lambda x \rightarrow (f x) \gg= g$$

The Monad Laws in Terms of ($>@>$)

...using ($>@>$), the monad laws, especially the **associativity requirement**, look as natural and obvious as for ($>>$).

Lemma 12.2.3

If ($>>=$) and **return** of some monad **m** are associative and unit of ($>>=$), respectively, then we have:

$$\text{return } >@> f = f \quad (\text{ML1}')$$

$$f >@> \text{return} = f \quad (\text{ML2}')$$

$$(f >@> g) >@> h = f >@> (g >@> h) \quad (\text{ML3}')$$

Intuitively

- **return** is unit of ($>@>$) (**ML1'**, **ML2'**).
- ($>@>$) is **associative** (**ML3'**).

A Law linking Classes Monad and Functor

...type constructors, which shall be proper instances of both `Monad` and `Functor` must satisfy law `MFL`:

```
fmap g xs = xs >>= return . g           (MFL)
           ( = do x <- xs; return (g x) )
```

(regarding the do-notation, refer to [Chapter 12.3.](#))

Selected Utility Functions for Monads (1)

```
(=<<)      :: Monad m => (a -> m b) -> m a -> m b
f =<< x    = x >>= f

sequence  :: Monad m => [m a] -> m [a]
sequence = foldr mcons (return [])
          where mcons p q = do l <- p
                               ls <- q
                               return (l:ls)

sequence_ :: Monad m => [m a] -> m ()
sequence_ = foldr (>>) (return ())

mapM      :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f as = sequence (map f as)

mapM_     :: Monad m => (a -> m b) -> [a] -> m ()
mapM_ f as = sequence_ (map f as)
```

Selected Utility Functions for Monads (2)

```
mapF      :: Monad m => (a -> b) -> m [a] -> m [b]
mapF f x  = do v <- x; return (f v)
  -- equals map on lists, i.e., for picking [] as m

joinM     :: Monad m => m (m a) -> m a
joinM x   = do v <- x; v
  -- equals concat on lists, i.e., for picking [] as m
```

...and many more (see e.g., library [Monad](#)).

Lemma 12.2.4

1. `mapF (f . g) = mapF . mapF g`
2. `joinM return = joinM . mapF return`
3. `joinM return = id`

Exercise 12.2.5

1. Prove [Lemma 12.2.4](#).
2. Do the functor and monad laws imply law **FML**? Prove or provide a counter-example.
3. Show that the below equalities are type correct. Annotate them with the (most general) type information applying:

3.1 the defining equation of $(>@>)$:

$$\begin{aligned} (>@>) &:: \text{Monad } m \Rightarrow (a \rightarrow m \ b) \rightarrow (b \rightarrow m \ c) \\ &\rightarrow (a \rightarrow m \ c) \end{aligned}$$

$$f >@> g = \lambda x \rightarrow (f \ x) >>= g$$

3.2 the statement of [Lemma 12.2.2](#):

$$c1 >> (c2 >> c3) = (c1 >> c2) >> c3$$

3.3 the statements of [Lemma 12.2.3](#):

$$\text{return } >@> f \quad = f \quad \text{(ML1')}$$

$$f >@> \text{return} \quad = f \quad \text{(ML2')}$$

$$(f >@> g) >@> h = f >@> (g >@> h) \quad \text{(ML3')}$$

Chapter 12.3

Syntactic Sugar: The do-Notation

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

12.1

12.2

12.3

99/196

The do-Notation

...the **monadic operations** (`>>=`) and (`>>`) allow very much as functional composition (`.`)

- to explicitly specify the sequencing of (fitting) operations.

Both **functional** and **monadic sequencing** introduce

- an **imperative** flavour into **functional** programming.

The **syntactic sugar** of the so-called

- **do-notation**

replacing (`>>=`) and (`>>`) allows to express this imperative flavour of **monadic sequencing** syntactically even more **compelling** and **concise**.

Relating Monadic Operations and do-Notation

...four **conversion rules** allow converting sequences of monadic operations composed of

- $(\gg=)$ and (\gg)

into **equivalent** ($\langle \Leftarrow \Rightarrow \rangle$) sequences of

- **do**-blocks

and vice versa.

Intuitively

Recall:

$(\gg=) :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$

$(\gg) :: m\ a \rightarrow m\ b \rightarrow m\ b$

Then:

$dc\ v \gg= \overbrace{\quad}^f \quad \dashv\!\!\dashv\! \gg \quad \overbrace{\quad}^f\ v$
 $:: m\ a \quad :: (a \rightarrow m\ b) \quad :: m\ b$
" <=> do x <- dc v; y <- f x; return y "

$dc\ v \gg dc'\ v' \dashv\!\!\dashv\! dc\ v \gg= _ \rightarrow dc'\ v'$
 $:: m\ a \quad :: m\ b \quad :: m\ a \quad :: (a \rightarrow m\ b)$
" <=> do _ <- dc v; y <- dc' v'; return y "

with dc, dc' some data constructors of type constructor m .

The Conversion Rules

(R1) `do e <=> e`

(R2) `do e1;e2;...;en <=> e1 >>= _ -> do e2;...;en`
`<=> e1 >> do e2;...;en`

(R3) `do let decl_list;e2;...;en <=> let decl_list`
`in do e2;...;en`

(R4) `do pattern <- e1;e2;...;en <=>`
`let ok pattern = do e2;...;en`
`ok _ = fail "..."`
`in e1 >>= ok`

...and as a special case of the 'pattern' rule (R4):

(R4') `do x <- e1;e2;...;en <=>`
`e1 >>= \x -> do e2;...;en`

Notes on the Conversion Rules

Intuitively

- (R2): If the return value of an operation is not needed, it can be moved to the front.
- (R3): A `let`-expression storing a value can be placed in front of the `do`-block.
- (R4): Return values bound to a pattern require a supporting function that handles the pattern matching and the execution of the remaining operations, or that calls `fail`, if the pattern matching fails.

Note: It is rule (R4) which necessitates `fail` as a monadic operation in `Monad`. Overwriting this operation allows a monad-specific exception and error handling.

Illustrating the do-Notation

...using the **monad laws** as example.

A) The **monad laws** using `(>>=)` and `(>>)`:

`return a >>= f` = `f a` (ML1)

`c >>= return` = `c` (ML2)

`c >>= (\x -> (f x) >>= g)` = `(c >>= f) >>= g` (ML3)

B) The **monad laws** using **do**-notation:

`do x <- return a; f x` = `f a` (ML1)

`do x <- c; return x` = `c` (ML2)

`do x <- c; y <- f x; g y` =
`do y <- (do x <- c; f x); g y` (ML3)

Semicolons vs. Linebreaks in do-Notation

B) do-notation in 'one' line (w/ ';', no linebreaks):

do x <- return a; f x = f a (ML1)

do x <- c; return x = c (ML2)

do x <- c; y <- f x; g y =
do y <- (do x <- c; f x); g y (ML3)

C) do-notation in 'several' lines (w/ linebreaks, no ';'):

do x <- return a
f x = f a (ML1)

do x <- c
return x = c (ML2)

do x <- c
y <- f x
g y = do y <- (do x <- c
f x)
g y (ML3)

Chapter 12.4

Monad Examples

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

12.1

12.2

12.3

1005/19

Predefined Monads in Haskell

We consider a selection of [predefined monads](#):

- [Identity](#) monad
- [List](#) monad
- [Maybe](#) monad
- [Map](#) monad
- [State](#) monad
- [Input/Output](#) monad

...but there are many more of them predefined in [Haskell](#):

- [Writer](#) monad
- [Reader](#) monad
- [Failure](#) monad
- ...

As a Rule of Thumb

...when making a 1-ary type constructor a monad, then:

- ($\gg=$) will be defined to unpack the value of the first argument, map the second argument over it, and return the packed result this yields.
- `return` will be defined in the most straightforward way to lift the argument value to its monadic counterpart.
- (\gg) and `fail` are usually not to be implemented afresh. Usually, their default implementations provided in type constructor class `Monad` are just fine.

If the default implementations of (\gg) and `fail` are used, this means for

- (\gg): the first argument is evaluated and dropped, the second argument is evaluated and returned as result (makes sense for some monads like the IO-monad).
- `fail`: the computation stops by calling `error` with some appropriate error message.

Chapter 12.4.1

The Identity Monad

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

12.1

12.2

12.3

1008/19

The Identity Monad

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

```
newtype Id a = Id a
instance Monad Id where
  (Id x) >>= f = f x
  return      = Id
```

Note:

- `Id`: 1-ary **type** constructor, i.e., if `a` is a type variable, then `Id a` denotes a type.
- `Id`: 1-ary **data** (or **value**) constructor, i.e., if `x :: a`, then `Id x` is a value of type `Id a`: `Id x :: Id a`.
- `(>>)`, `fail` implicitly defined by default implementations.
- `(>>=)` :: `Id a -> (a -> Id b) -> Id b`
- `return` :: `a -> Id a`
- `(>>)` :: `Id a -> Id b -> Id b`

Proof Obligation: The Monad Laws

Lemma 12.4.1.1 (Soundness of Identity Monad)

The `Id` instance of `Monad` satisfies the three monad laws `ML1`, `ML2`, and `ML3`.

...`Id` is thus a proper instance of `Monad`, the so-called *identity monad*.

The Identity Monad Operations in more Detail

The monad operations recalled:

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
v >>= k = ... :: m b
return :: (Monad m) => a -> m a
return v = ... :: m a
```

The instance declaration for `Id` with added type information:

```
instance Monad Id where
  Id x >>= f = f x -- yields an (Id b)-value
  :: Id a    :: a -> Id b  :: Id b
  return x = Id x -- yields an (Id a)-value
  :: a      :: Id a
```

Recall the overloading of `Id` (newtype `Id a = Id a`):

- `Id` followed by `x`: `Id` is **data** (or **value**) constructor ($\text{Id} \hat{=} \text{Id}$).
- `Id` followed by `a` or `b`: `Id` is **type** constructor ($\text{Id} \hat{=} \text{Id}$).

Note

Intuitively

- The identity monad maps a type to itself.
- It represents the trivial state, in which no actions are performed, and values are returned immediately.
- It is useful because it allows to specify computation sequences on values of its type (cf. [Chapter 12.5.1](#))

Moreover

- The operation $(>@>)$ boils down to [forward composition](#) of functions $(>.>)$ ($\hat{=}$ $(>>;)$) for the identity monad:
$$(>.>) :: (a \rightarrow b) \rightarrow (b \rightarrow c) \rightarrow (a \rightarrow c)$$
$$g >.> f = f . g = g ; f$$
- Forward composition of functions $(>.>)$ is [associative](#) with [unit](#) element [id](#).

Chapter 12.4.2

The List Monad

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

12.1

12.2

12.3

1013/19

The List Monad

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

```
instance Monad [] where
```

```
  xs >>= f = concat (map f xs)  -- concat, map:
  return x  = [x]                -- Standard Prelude
  fail s    = []
```

Note:

- `[]`: 1-ary **type** constructor, i.e., if `a` is a type variable, then `[a]` ($\hat{=}$ `[] a`) denotes a type.
- `[]`: 1-ary **data** (or **value**) constructor, i.e., if `x :: a`, then `[x]` is a value of type `[a]`: `[x] :: [a]`; in particular, `[]` is a value, the empty list, i.e., `[] :: [a]`
- `(>>)` is implicitly defined by its default implementation; the default implementation of `fail` is overwritten.
- `(>>=)` `:: [] a -> (a -> [] b) -> [] b`
- `return` `:: a -> [] a`
- `(>>)` `:: [] a -> [] b -> [] b`

Proof Obligation: The Monad Laws

Lemma 12.4.2.1 (Soundness of List Monad)

The `[]` instance of `Monad` satisfies the three monad laws `ML1`, `ML2`, and `ML3`.

... `[]` is thus a proper instance of `Monad`, the so-called `identity monad`.

For convenience, we `recall` from the `Standard Prelude`:

```
concat :: [[a]] -> [a]
concat lss = foldr (++) [] lss
concat [[1,2,3], [4], [5,6]] ->> [1,2,3,4,5,6]

map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x : xs) = f x : map f xs
map (*2) [1,2,3] ->> [2,4,6]
```

The List Monad Operations in more Detail

The monad operations recalled:

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
v >>= k = ... :: m b
return :: (Monad m) => a -> m a
return v = ... :: m a
fail :: (Monad m) => String -> m a
fail s = ... :: m a
```

The instance declaration for `[]` with added type information:

```
instance Monad [] where
  xs >>= f      = concat (map f xs)      -- yields a [b]-list
  :: [] a      :: a -> [] b              :: [] ([] b)
  :: [] b

  return x      = [x]                  -- yields the singleton list [x]
  :: a          :: [] a

  fail s        = []                    -- yields the empty list []
  :: String     :: [] a
```


Example: Applying the Monad Operations

```
ls = [1,2,3] :: [] Int
f = \n -> [(n,odd(n))] :: Int -> [] (Int,Bool)
g = \n -> [x*n | x <- [1.5,2.5,3.5]] :: Int -> [] Float
h = \n -> [1..n] :: Int -> [] Int
```

```
h 3 >>= f
->> ls >>= f
->> concat [ [(1,True)], [(2,False)], [(3,True)] ]
->> [(1,True),(2,False),(3,True)] :: [] (Int,Bool)

h 3 >>= g
->> ls >>= g
->> concat [ [ x*n | x <- [1.5,2.5,3.5] ] | n <- [1,2,3] ]
->> concat [ [1.5*1,2.5*1,3.5*1], [1.5*2,2.5*2,3.5*2],
            [1.5*3,2.5*3,3.5*3] ]
->> concat [ [1.5,2.5,3.5], [3.0,5.0,7.0], [4.5,7.5,10.5] ]
->> [1.5,2.5,3.5,3.0,5.0,7.0,4.5,7.5,10.5] :: [] Float
```

The Example in More Detail

The monad operations recalled:

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
v >>= k = ... :: m b
return :: (Monad m) => a -> m a
return v = ... :: m a
fail :: (Monad m) => String -> m a
fail s = ... :: m a
```

The instance declaration for `[]` with added type information:

```
instance Monad [] where
  xs >>= f = concat (map f xs)  -- yields a [b]-list
  :: [] a  :: a -> [] b          :: [] ([] b)
  :: [] b

  return x = [x]  -- yields the singleton list [x]
  :: a       :: [] a
  fail s     = []  -- yields the empty list []
  :: String  :: [] a
```

Examples:

```
ls = [1,2,3] :: [] Int
f = \n -> [(n,odd(n))] :: Int -> [] (Int,Bool)
g = \n -> [x*n | x <- [1.5,2.5,3.5]] :: Int -> [] Float
h = \n -> [1..n] :: Int -> [] Int

h 3 >>= f ->> ls >>= f ->> concat [ [(1,True)], [(2,False)], [(3,True)] ]
->> [(1,True),(2,False),(3,True)] :: [] (Int,Bool)

h 3 >>= g ->> ls >>= g ->> concat [ [ x*n | x <- [1.5,2.5,3.5] ] | n <- [1,2,3] ]
->> concat [ [1.5*1,2.5*1,3.5*1], [1.5*2,2.5*2,3.5*2], [1.5*3,2.5*3,3.5*3] ]
->> concat [ [1.5,2.5,3.5], [3.0,5.0,7.0], [4.5,7.5,10.5] ]
->> [1.5,2.5,3.5,3.0,5.0,7.0,4.5,7.5,10.5] :: [] Float
```

Reconsidering the List Monad Implementation

...the `list monad` could have `equivalently` been implemented by:

```
instance Monad [] where
  (x:xs) >>= f = f x ++ (xs >>= f)
  [] >>= f = []
  return x = [x]
  fail s = []
```

Recall: The operations `(>>=)` and `return` of the `list monad` have types:

```
(>>=)  :: [a] -> (a -> [b]) -> [b]
return :: a -> [a]
```

List Monad and List Comprehension

...the **list monad** and **list comprehension** are closely related:

```
do x <- [1,2,3]
   y <- [4,5,6]
   return (x,y)
->> [(1,4), (1,5), (1,6),
      (2,4), (2,5), (2,6),
      (3,4), (3,5), (3,6)]
```

In fact, the following expressions are **equivalent**:

Proposition 12.4.2.2

$$[(x,y) \mid x \leftarrow [1,2,3], y \leftarrow [4,5,6]] \Leftrightarrow$$

```
do x <- [1,2,3]
   y <- [4,5,6]
   return (x,y)
```

...**list comprehension** is **syntactic sugar** for **monadic syntax**!

List comprehension: Syntactic Sugar

...for monadic syntax.

We have:

Lemma 12.4.2.3

$$[f\ x \mid x \leftarrow xs] \iff \text{do } x \leftarrow xs; \text{return } (f\ x)$$

Lemma 12.4.2.4

$$[a \mid a \leftarrow as, p\ a] \iff \text{do } a \leftarrow as; \text{if } (p\ a) \text{ then return } a \text{ else fail ""}$$

Exercise 12.4.2.5

Prove by stepwise evaluation the equivalences stated in:

1. Proposition 12.4.2.2
2. Lemma 12.4.2.3
3. Lemma 12.4.2.4

Chapter 12.4.3

The Maybe Monad

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

12.1

12.2

12.3

The Maybe Monad

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

```
data Maybe a = Nothing | Just a

instance Monad Maybe where
  (Just x) >>= k = k x
  Nothing  >>= k = Nothing
  return   = Just
  fail s   = Nothing
```

Note:

- `(>>=)` :: `Maybe a -> (a -> Maybe b) -> Maybe b`
- `return` :: `a -> Maybe a`
- `(>>>)` :: `Maybe a -> Maybe b -> Maybe b`
- The `Maybe monad` is useful for computation sequences that can produce a result, but might also produce an error.

Proof Obligation: The Monad Laws

Lemma 12.4.3.1 (Soundness of Maybe Monad)

The `Maybe` instance of `Monad` satisfies the three monad laws `ML1`, `ML2`, and `ML3`.

...`Maybe` is thus a proper instance of `Monad`, the so-called `maybe monad`.

Recall that `Maybe` is also an instance of `Functor`:

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

Lemma 12.4.3.2 (MFL Soundness of Maybe Mo/Fu)

The `Maybe` instances of `Monad` and `Functor` satisfy law `MFL` (of [Chap. 12.2](#)).

The Maybe Monad Operations in More Detail

The monad operations recalled:

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
v >>= k = ... :: m b
return :: (Monad m) => a -> m a
return v = ... :: m a
fail :: (Monad m) => String -> m a
fail s = ... :: m a
```

The instance declaration for `Maybe` with added type information:

```
instance Monad Maybe where
  Just x >>= k = k x           -- yields a Just-value
  :: Maybe a  :: a -> Maybe b  :: Maybe b
  Nothing >>= k = Nothing     -- yields the Nothing-value
  :: Maybe a  :: a -> Maybe b  :: Maybe b
  return x = Just x          -- yields the Just-value
  :: a
  fail s = Nothing          -- yields the empty list
  :: String
  :: Maybe a
```

Example: Error Handling: (1)

...or: How to compose **functions** with **monadic value ranges**.

Let f' , g' be two functions of type:

$$f' :: a \rightarrow b$$

$$g' :: b \rightarrow c$$

Obviously, composing f' and g' sequentially is straightforward:

$$h' :: a \rightarrow c$$

$$h' = (g' . f')$$

$$h' \ x \ \rightarrow\! \rightarrow \ (g' . f') \ x \ \rightarrow\! \rightarrow \ g' \ (f' \ x)$$

Example: Error Handling (2)

If the computations of f' and g' can fail, this can be taken care of by replacing f' and g' by two new functions f and g embedding the computation into the `Maybe` type:

```
f :: a -> Maybe b           -- f replaces f'
g :: b -> Maybe c           -- g replaces g'
```

Unlike f' and g' , however, f and g can not straightforwardly be sequentially composed:

```
h :: a -> Maybe c           -- "h = (g . f)":
h x = case (f x) of         -- Composing f and g
    Nothing -> Nothing      -- requires nested
    Just y   -> case (g y) of -- case clauses
        Nothing -> Nothing
        Just z   -> Just z
```

Though possible, the explicit nesting of cases to sequentially compose f and g is inconvenient and tedious.

Example: Error Handling (3)

Step 1: Hiding nestings.

...embedding f' and g' into the `Maybe` type gets a lot easier by exploiting the monad property of `Maybe`: Using the `monadic sequencing operations` for composing `f` and `g` allows:

```
h :: a -> Maybe c           -- "h = (g . f)"
h x = f x >>= \y -> g y >>= \z -> return z
```

or, `equivalently`, using the `do` notation:

```
h :: a -> Maybe c           -- "h = (g . f)"
h x = do y <- f x
         z <- g y
         return z
```

...the 'nasty' error checks are now hidden in the implementation of the bind operation (`>>=`) of the `maybe monad`.

Example: Error Handling (4)

Step 2: Hiding the bind operation ($\gg=$).

Note that the sequence of monad operations:

$$f\ x \gg= \backslash y \rightarrow g\ y \gg= \backslash z \rightarrow \text{return}\ z$$

can be [simplified](#) to:

$$f\ x \gg= \backslash y \rightarrow g\ y \gg= \backslash z \rightarrow \text{return}\ z$$

\Leftrightarrow (simplification by currying)

$$f\ x \gg= \backslash y \rightarrow g\ y \gg= \text{return}$$

\Leftrightarrow (monad law for return)

$$f\ x \gg= \backslash y \rightarrow g\ y$$

\Leftrightarrow (simplification by currying)

$$f\ x \gg= g$$

Hence, $h\ x$ (“ $= g\ (f\ x)$ ”) is equivalent to $f\ x \gg= g$.

Example: Error Handling (5)

...making use of this observation and introducing function:

```
composeM :: Monad m => (b -> m c) ->
              (a -> m b) -> (a -> m c)
(g 'composeM' f) x = f x >>= g
```

allows an even more pleasing notation for composing f and g :

```
h :: a -> Maybe c           -- "h = (g . f)"
h = (g 'composeM' f)
```

Hence, we get:

```
(g 'composeM' f)
```

as the monadic notational counterpart of sequentially composing f' and g' :

```
(g' . f')
```

Example: Error Handling (6)

Overall: Using monadic sequencing

$f\ x \gg= g$ (or equivalently: $(g\ \text{'composeM'}\ f)\ x$)

for embedding the composition of f' and g' into the **Maybe** type preserves the original syntactical form of composing f' and g' :

$$(g' . f')\ x = g'\ (f'\ x)$$

in almost a 1-to-1 kind:

$$(g\ \text{'composeM'}\ f)\ x = f\ x \gg= g$$

Chapter 12.4.4

The Either Monad

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

12.1

12.2

12.3

1033/19

Exercise 12.4.4.1. The Either Monad

1. Make the type constructor `(Either a)` an instance of `Monad`.
2. Provide (most general) type information for the defining equations of the monad operations `(>>=)`, `(>>)`, `return`, and `fail` of `(Either a)`.
3. Prove that `(Either a)` satisfies the monad laws.
4. Does your implementation of the `(Either a)` monad instance and the implementation of the `(Either a)` functor instance of [Chapter 10.3.4](#) satisfy the law `FML` (of [Chap. 12.2](#))? Prove or provide a counter-example.

Chapter 12.4.5

The Map Monad

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

12.1

12.2

12.3

1035/19

The Map Monad

...making the 1-ary type constructor $((\rightarrow) d)$ an instance of `Monad`:

```
instance Monad ((->) d) where
  h >>= f = \x -> f (h x) x
  return x = \_ -> x
```

Note: (`d` for domain, `r` for range)

```
(>>=) :: ((->) d) r -> (r -> ((->) d) r') -> ((->) d) r'
return :: r -> ((->) d) r
(>>)   :: ((->) d) r -> ((->) d) r' -> ((->) d) r'
```

Proof obligation: The monad laws

Lemma 12.4.5.1 (Soundness of Map Monad)

The $((\rightarrow) d)$ instance of `Monad` satisfies the three monad laws `ML1`, `ML2`, and `ML3`.

... $((\rightarrow) d)$ is thus a proper instance of `Monad`, the so-called `map monad`.

Example (w/ String, Int, (Bool,String) for d, r, r', resp.) (1)

```
(>>=) :: ((->) d) r -> (r -> ((->) d) r') -> ((->) d) r'
(≡ (>>=) :: (d -> r) -> (r -> (d -> r')) -> (d -> r') )
h >>= f = \x -> f (h x) x

h_length :: ((->) String) Int
(≡ h_length :: String -> Int )
h_length = length

f_cp_p :: Int -> ((->) String) ((,) Bool String)
(≡ f_cp_p :: Int -> (String -> (Bool,String) ) )
f_cp_p n s = (,) (mod n 2 == 1) (copy n s)
  where copy n s = if n > 0 then s++" "++copy (n-1) s else ""

g :: ((->) String) ((,) Bool String)
(≡ g :: String -> (Bool,String) )
g = \s -> f_cp_p (h_length s) s
(≡ g s = (mod (length s) 2 == 1, copy (length s) s) )

h_length >>= f_cp_p
->> (\x -> f_cp_p (h_length x) x)      ( = g )

(h_length >>= f_cp_p) "Fun"
->> ... ->> (True, "Fun Fun Fun")
```

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

12.1

12.2

12.3

1037/19

Example (w/ String, Int, (Bool,String) for d, r, r', resp.) (2)

...in more detail:

```
h_length >>= f_cp_p
->> (\x -> f_cp_p (h_length x) x)
    = g      ( :: String -> (Bool,String) )

(h_length >>= f_cp_p) "Fun"
->> (\x -> f_cp_p (h_length x) x) "Fun"
    = g "Fun"
->> (mod (length "Fun") 2 == 1, copy (length "Fun") "Fun")
->> (mod 3 2 == 1, copy 3 "Fun")
->> (True, "Fun Fun Fun")      ( :: (Bool,String) )
```

Example (w/ String, Int, (Bool,String) for d, r, r', resp.) (3)

```
(>>=)  :: ((->) d) r -> (r -> ((->) d) r') -> ((->) d) r'
h >>= f  = \x -> f (h x) x

return  :: r -> ((->) d) r  (≡ return :: Int -> ((->) String) Int)
return x = \_ -> x          ≡ return :: Int -> (String -> Int)

return 0 = \_ -> 0      ( :: String -> Int )

return 0 >>= f_cp_p
->> \x -> f_cp_p ((return 0) x ) x
->> \x -> f_cp_p (\_ -> 0) x) x ( :: String -> (Bool,String) )

(return 0 >>= f_cp_p) "Fun"
->> (\x -> f_cp_p ((return 0) x ) x) "Fun"
->> f_cp_p ((return 0) "Fun" ) "Fun"
->> f_cp_p ((\_ -> 0) "Fun") "Fun"
->> f_cp_p 0 "Fun"
->> (mod 0 2 == 1, copy 0 "Fun")
->> (False, "")      ( :: (Bool,String) )

(return 1 >>= f_cp_p) "Fun" ->> ... ->> (True, "Fun")
(return 2 >>= f_cp_p) "Fun" ->> ... ->> (False, "Fun Fun")
(return 3 >>= f_cp_p) "Fun" ->> ... ->> (True, "Fun Fun Fun")
```

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

12.1

12.2

12.3
1039/10

Example (w/ String, Int for d, r, resp.) (4)

```
(>>=)  :: ((->) d) r -> (r -> ((->) d) r') -> ((->) d) r'  
h >>= f  = \x -> f (h x) x
```

```
return :: r -> ((->) d) r (≡ return :: Int -> ((->) String) Int)  
return x = \_ -> x          ≡ return :: Int -> (String -> Int)
```

```
return 3 = \_ -> 3    ( :: String -> Int )
```

```
h_length >>= return
```

```
->> \x -> return (h_length x) x
```

```
->> \x -> return (length x) x
```

```
->> \x -> (\_ -> length x) x    ( :: String -> Int )
```

```
(h_length >>= return) "Fun"
```

```
->> (\x -> (return (h_length x) x)) "Fun"
```

```
->> return (h_length "Fun") "Fun"
```

```
->> return (length "Fun") "Fun"
```

```
->> return 3 "Fun"
```

```
->> (\_ -> 3) "Fun"
```

```
->> 3    ( :: Int )
```


Exercise 12.4.5.2

1. Recall the monad operations:

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
```

```
v >>= k = ... :: m b
```

```
return :: (Monad m) => a -> m a
```

```
return v = ... :: m a
```

Show that the defining equations of (`>>=`) and `return` of the `map` instance are type correct:

```
instance Monad ((->) d) where
```

```
h >>= f = \x -> f (h x) x
```

```
return x = \_ -> x
```

2. Evaluate stepwise:

```
2.1 (return 2 >>= f_cp_p) "Fun"
```

```
2.2 (h_length >>= return) "Fun Prog"
```

```
2.3 (h_length >>= return >>= f_cp_p) "Fun"
```

Chapter 12.4.6

The State Monad

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

12.1

12.2

12.3

Objective: Modelling Global State, Side-Effects

...by means of functions, which, applied to some

- initial `state s`

yield `a`

- new `state s'`

as part of the overall result of the computation.

Key: The `State Monad` based on some appropriate `state type`:

```
newtype State st a = St (st -> (st, a))
```

Note:

- `State` : 2-ary type constructor (linking `st` and `a`).
- `st, a`: Type variables (concrete types inserted for `st` and `a` are the actual `state type` of interest and the type of an additional result of state transformations, respectively).
- `(st -> (st, a))`: The `state transformer` map.

State Transformers

...map (or: transform) global (internal program) states of a type `st` into (possibly modified) new states of type `st` while additionally computing a result of some type `a`.

In more detail:

State transformers are mappings `m`:

$$m :: st \rightarrow (st, a)$$

mapping states `s :: st` to pairs of (possibly modified result) states `s' :: st` and values `x :: a`:

$$\underbrace{m\ s}_{::\ st} \rightarrow \underbrace{(s')}_{::\ st}, \underbrace{x}_{::\ a}$$

The State Monad

...making the 1-ary type constructor `(State st)` an instance of `Monad`:

instance `Monad (State st)` where

```
(St h) >>= f = St (\s -> let (s',x) = h s
                          :: st          St f' = f x
                          in f' s')
                          :: (st,b)
```

```
-- Applying map h :: (st -> (st,a)) to state s :: st
-- yields a pair (s',x) :: (st,a) onto whose 2nd compo-
-- nent x :: a map f :: a -> (State st) b is applied.
-- This yields a state value St f' :: (State st) b,
-- whose map value f' :: st -> (st,b) is applied to
-- s' :: st yielding a pair f' s' :: (st,b) as required.
```

```
return x = St (\s -> (s,x))
           :: a      :: st      :: (st,a)
```

```
-- x :: a and every state s :: st are identically mapped.
```

Note

...for the `state monad (State st)` the monad operations `(>>=)`, `return`, and `(>>)` have the types:

$$\begin{aligned} (>>=) &:: (\text{State } st) \ a \ \rightarrow \ (a \ \rightarrow \ (\text{State } st) \ b) \\ &\qquad\qquad\qquad \rightarrow \ (\text{State } st) \ b \end{aligned}$$
$$\text{return} :: a \ \rightarrow \ (\text{State } st) \ a$$
$$(>>) :: (\text{State } st) \ a \ \rightarrow \ (\text{State } st) \ b \ \rightarrow \ (\text{State } st) \ b$$

The State Monad in more Detail

The monad operations recalled:

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
c >>= k = ... :: m b
return :: (Monad m) => a -> m a
return x = ... :: m a
```

The instance declaration for `(State st)` with added type information:

```
instance Monad (State st) where
  St h >>= f
  :: (State st) a
  = St (\s -> let ... in f' s') -- constructing
      st :: (st,b) -- a proper state
      :: st -> (st,b) -- value using m
      :: (State st) b -- and f.

  return x = St (\s -> (s,x)) -- constructing a proper
      st :: (State st) a -- state value using x
  :: (State st) a -- in the simplest way.
```

Proof Obligation: The Monad Laws

Lemma 12.4.6.1 (Soundness of the State Monad)

The `(State st)` instance of `Monad` satisfies the three monad laws `ML1`, `ML2`, and `ML3`.

... `(State st)` is thus a proper instance of `Monad`, the so-called `state monad`.

State': The Specialized State Monad

...specialized for a concrete state type `CStT` ('Concrete State Type') (e.g., `Int`, `[String]`, ...):

```
newtype State' a = St' (CStT -> (CStT, a))
```

```
instance Monad State' where
```

```
St' m >>= f = St' (\cs -> let (cs', x) = m cs
                             :: CStT      St' f' = f x
                             in f' cs')
                             ::] (CStT, b)
```

```
return x = St' (\cs -> (cs, x))
           :: a      :: CStT :: (CStT, a)
```

Note: `State'` is a 1-ary type constructor whereas `State` is a 2-ary type constructor.

Proof Obligation: The Monad Laws (`State'`)

Lemma 12.4.6.2 (Soundness of Spec. State Monad)

The `State'` instance of `Monad` satisfies the three monad laws `ML1`, `ML2`, and `ML3`.

...(`State'`) is thus a proper instance of `Monad`, the so-called specialized state monad.

Note: For `State'` the types of the monad operations (`>>=`), `return`, and (`>>`) boil down to:

`>>=` :: `State' a -> (a -> State' b) -> State' b`

`return` :: `a -> State' a`

`>>` :: `State' a -> State' b -> State' b`

The State Monad Reconsidered (1)

...sometimes **renaming objects** helps getting things clear(er).

Think about `st_otw` as a type variable where the values of appropriate concrete types for `st_otw` describe or model the

- state of the world (`st_otw`).

The sequencing operation (`>>=`) of the state monad (`State st_otw`) allows then to **transform current states of the world** into **new states of the world**, i.e., to

- **transform** (the description of) the **state of the world it is currently in** into (the description of) the world it is in after the transformation, i.e., (the description of) the **new state the world is in** afterwards.

This suggests that **state transformer** are of the type:

```
state_transformer :: st_otw -> st_otw
```

...class `Monad` makes this a bit more complex as shown next.

The State Monad Reconsidered (2)

```
newtype (State st_otw) a = St (st_otw -> (st_otw,a))
```

```
instance Monad (State st_otw) where
```

```
  St h >>= f
```

```
  = St (\current_state ->
```

```
      let (intermediate_state,x) = h current_state
```

```
          St g = f x
```

```
          (new_state,z) = g intermediate_state
```

```
      in (new_state,z))
```

```
return x = St (\current_state -> (current_state,x))
```

Note (compare especially the similarity of the definitions of (>>=), (;)):

```
- (>>=) :: (State st_otw) a -> (a -> (State st_otw) b) ->  
                                                (State st_otw) b
```

```
return :: a -> (State st_otw) a
```

```
- (g . f) = (f; g) = \x -> let intermediate = f x  
                             y = g intermediate  
                             in y           -- note: y = g (f x)
```

Chapter 12.4.7

The Input/Output Monad

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

12.1

12.2

12.3

1053/19

The Input/Output Monad

```
instance Monad IO where      (Impl. intern. hidden)
  (>>=)  :: IO a -> (a -> IO b) -> IO b
  return :: a -> IO a
  (>>)   :: IO a -> IO b -> IO b
  fail   :: String -> IO a
```

Note:

- IO-values are so-called IO-commands (or commands).
- Commands have a procedural effect (i.e., reading or writing) and a functional effect (i.e., computing a value).
- (>>=): With p , q commands, $p \gg= q$ is a composed command that first executes p , thereby performing a read or write operation and yielding an a -value x as result; subsequently q is applied to x , thereby performing a read or write operation and yielding a b -value y as result.
- return: Lifts an a -value to an IO a -value w/out performing any input or output operation.

Proof Obligation: The Monad Laws

Lemma 12.4.7.1 (Soundness of I/O Monad)

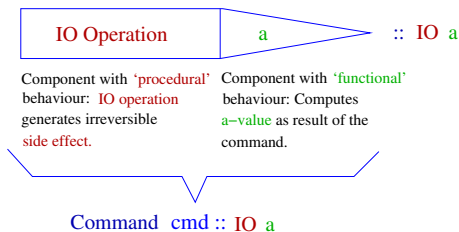
The **IO** instance of **Monad** satisfies the three monad laws **ML1**, **ML2**, and **ML3**.

...**IO** is thus a proper instance of **Monad**, the so-called **input/output (I/O) monad**.

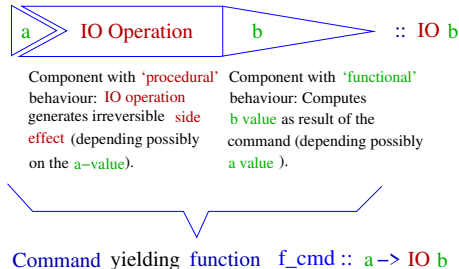
Note: The implementation of the input/output monad is internally hidden; it is thus the compiler writer who is in charge for proving **Lemma 12.4.7.1**.

Illustrating the Nature of Commands

Command $\text{cmd} :: \text{IO } a$

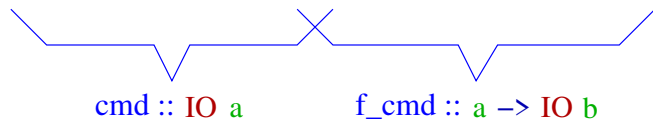


Command yielding function $\text{f_cmd} :: a \rightarrow \text{IO } b$



Illustrating

...the operational meaning of $(\text{cmd} \gg= \text{f_cmd})$:



$$\text{cmd} \gg= \text{f_cmd} \hat{=} \text{cmd} \gg= \backslash x \rightarrow \text{f_cmd } x$$

Illustrating

...the operational meaning of $(\text{cmd} \gg \text{cmd}')$:



$\text{cmd} :: \text{IO } a$

$\text{cmd}' :: \text{IO } b$

$\text{cmd} \gg \text{cmd}' \hat{=} \text{cmd} \gg \backslash_ \rightarrow \text{cmd}'$

Illustrating

...the operational meaning of `return`:



Component with **'procedural'** behaviour: `'empty'`; no IO operation, no side effect.

Component with **'functional'** behaviour: Forwards the `a-value` as the result of the command.

Command `return` `:: a -> IO a`

The Type

...of all **read commands** is

- **(IO a)** (for type instances **a** whose values can be read).

The **a**-value into which the read value is transformed serves as the (formally required and actually wanted) result of read operations.

...of all **write commands** is

- **(IO ())**, where **()** is the singleton **null tuple type** with the single unique element **()**.

() as (the one and only) value of the null tuple type **()** serves as the **formally required** result of write operations.

The I/O Monad viewed as a State Monad

...the `input/output monad` is similar in spirit to the `state monad`: It passes around the “`state of the world!`”

For a suitable type `World` whose values represent the

- `states of the world`

`interactive programs` (or `IO-programs`) can informally be considered functions of a type `IO` with:

- “`type IO = (World -> World)`”

In order to reflect that `interactive programs` do not only modify the state of the world but may also `return` a `result`, e.g., the `Int`-value of a sequence of characters that has been read from the keyboard and interpreted as an integer, this leads to changing the informal type of `IO-programs` from `IO` to `(IO a)`:

- “`type IO a = (World -> (World, a))`”

The Input/Output Monad (1)

...allows switching from a **batch**-like handling of **input/output**:



Peter Pepper. *Funktionale Programmierung*. Springer-Verlag, 2003, p. 245.

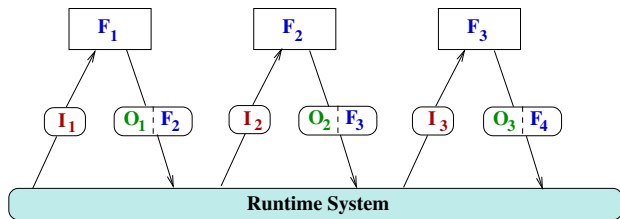
where

- all input data must be provided at the very beginning
- there is **no interaction** between a **program** and a **user** (i.e., once called there is no opportunity for the user to react on a program's response and behaviour)

to a...

The Input/Output Monad (2)

...truly interactive handling of **input/output** in terms of sequentially composed **dialogue components**, while preserving **referential transparency** as far as possible:



Peter Pepper. *Funktionale Programmierung*.
Springer-Verlag, 2003, p. 253.

Note that **input/output** operations are a **major source** for **side effects**: read statements e.g. will yield different values for every call causing unavoidably the loss of **referential transparency**.

Examples: Simple IO Programs (1)

...a [question/response interaction](#) with a user:

```
ask :: String -> IO String
ask question = do putStrLn question
                  getLine

interAct :: IO ()
interAct =
    do name <- ask "May I ask your name?"
       putStrLine ("Welcome " ++ name ++ "!!")
```


Examples: Simple IO Programs (2)

...input/output from and to files:

```
type FilePath = String    -- file names according
                           -- to the conventions of
                           -- the operating system

writeFile  :: FilePath -> String -> IO ()
appendFile :: FilePath -> String -> IO ()
readFile   :: FilePath -> IO String
isEOF      :: FilePath -> IO Bool

interAct :: IO ()
interAct = do putStr "Please input a file name: "
              fname <- getLine
              contents <- readFile fname
              putStr contents
```

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

12.1

12.2

12.3

1065/10

Examples: Simple IO Programs (3)

...the sequence of [input/output commands](#) with [local declarations](#) within a `do`-construct

```
reverse2lines :: IO ()
reverse2lines = do line1 <- getLine
                   line2 <- getLine
                   let rev1 = reverse line1
                       rev2 = reverse line2
                   putStrLn rev2
                   putStrLn rev1
```

is [equivalent](#) to the following one without:

```
reverse2lines :: IO ()
reverse2lines = do line1 <- getLine
                   line2 <- getLine
                   putStrLn (reverse line2)
                   putStrLn (reverse line1)
```

Examples: Simple IO Programs (4)

...sequences of (canonic) **monadic operations**:

```
writeFile "testFile.txt" "Hello File System!"  
>> putStr "Hello World!" >> putStr "Oh, yeah."
```

can be replaced by their equivalent **do**-expressions:

```
do writeFile "testFile.txt" "Hello File System!"  
  putStr "Hello World!"  
  putStr "Oh, yeah."
```

Examples: Simple IO Programs (5)

...note the sometimes subtle differences in the representation of values of **output** and **non-output** types.

Output types:

```
Main>putStr ('a':('b':('c':[])))    Main>putChar (head ['x','y','z'])
->> abc :: IO ()                  ->> x :: IO ()
```

Non-output types:

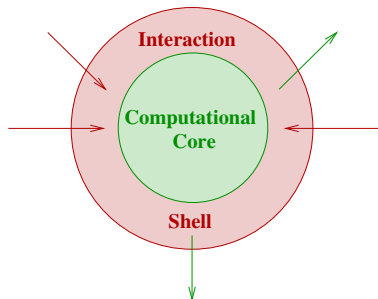
```
Main>('a':('b':('c':[])))          Main>head ['x','y','z']
->> "abc" :: [Char]                ->> 'x' :: Char

Main>print "abc"                    Main>print 'x'
->> "abc" :: IO ()                 ->> 'a' :: IO ()
```

Monadic Input/Output in Haskell

...allows us to conceptually think of a Haskell program as being composed of a

- purely functional computational core
- procedural-like interaction shell.



Manuel Chakravarty, Gabriele Keller. *Einführung in die Programmierung mit Haskell*. Pearson, 2004, p. 89.

The Conceptual Separation

...of functions belonging to the

- **computational core** (pure functions)
- **interaction shell** (impure functions, i.e., performing input/output operations causing side effects).

is achieved by assigning different **types** to them:

- **Int**, **Real**, **String**,... vs. **IO Int**, **IO Real**, **IO String**,...

with the type constructor **IO** a pre-defined **monad**.

The **monadic implementation** of **input/output** allows us

- precisely specify the evaluation order of functions of the interaction shell (i.e., basic **input/output** primitives provided by Haskell) by using the **monadic sequencing** operations (**>>=**) and (**>>**).

...see e.g. lecture notes of **LVA 185.A03 Funktionale Programmierung** for further details and examples.

Chapter 12.5

Monadic Programming

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

12.1

12.2

12.3

1071/19

Monadic Programming

...we consider [three examples](#) for [illustration](#):

1. [Folding trees](#) by adding the values of their numerical labels.
2. [Numbering tree labels](#) (and overwriting the original labels).
3. [Renaming tree labels](#) by the number of their occurrences.

The first two examples are handled

- without
- with

[monads](#) in order to [oppose](#) and [illustrate](#) the [relative merits](#) of the [two programming styles](#).

Chapter 12.5.1

Folding Trees

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

12.1

12.2

12.3

1073/19

The Setting

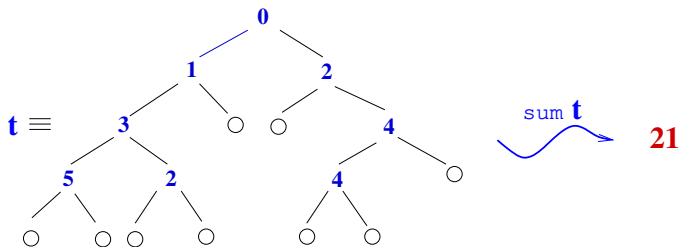
Given:

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

Objective:

- Write a function that computes the sum of the values of all labels of a tree of type `Tree Int`.

Illustration:



For Comparison

...we consider three [approaches](#):

1. w/out monads
2. w/ monads
3. w/ monads followed by [unpacking](#) the monadic result.

1st Approach: Straightforward w/out Monads

...using a [recursive](#) function:

```
sum :: Tree Int -> Int
sum Nil           = 0
sum (Node n t1 t2) = n + sum t1 + sum t2
```

Note:

- The [evaluation order](#) of the right-hand term of the (non-trivial) defining equation of `sTree` is [not fixed](#); only [data dependencies](#) need to be respected.
- This leaves interpreter and compiler a [degree of freedom](#) in picking an evaluation order.
- This freedom can not be broken by a programmer by using a specific right-hand side term:

```
sum (Node n t1 t2) = n + sum t1 + sum t2
sum (Node n t1 t2) = sum t2 + n + sum t1
...
sum (Node n t1 t2) = sum t2 + sum t1 + n
```

2nd Approach: Using the Identity Monad

...using the `identity monad` `Id`:

```
sum' :: Tree Int -> Id Int
sum' Nil = return 0
sum' (Node n t1 t2) =
  do s2 <- sum' t2      -- Evaluating right subtree
     num <- return n    -- Bounding n :: Int to num
     s1 <- sum' t1      -- Evaluating left subtree
     return (s2+num+s1) -- Yielding Id (num+s1+s2) ::
                        -- Id Int as result
```

Note:

- The evaluation order of the defining 'equations' for `s2`, `n`, and `s1` is **explicitly fixed**; there is no degree of freedom for the sequence in which values are bound to them.
- Changing their order allows the programmer to enforce a different evaluation order.
- Note, this does not apply to evaluating `s2+num+s1`.

Recall

...the definition of the **identity monad** `Id`:

```
newtype Id a = Id a

instance Monad Id where
  (Id x) >>= f = f x
  return      = Id
```

...and the overloading of `Id`:

- `Id`: 1-ary **type** constructor, i.e., if `a` is a type variable, then `Id a` denotes a type.
- `Id`: 1-ary **data** (or **value**) constructor, i.e., if `x :: a`, then `Id x` is a value of type `Id a`: `Id x :: Id a`.

Illustrating the Imperative Flavour of `sum'`

...unlike `sum`, `sum'` enjoys an 'imperative' flavour quite similar to sequentially sequencing assignment statements of some imperative programming language:

Imperative

```
s2 := sumTree t2;  
s1 := sumTree t1;  
num := n;  
return (s2+s1+num);
```

Monadic

```
do s2 <- sumTree t2  
   s1 <- sumTree t1  
   num <- return n  
   return (s2+s1+num)
```

3rd Approach: Unpacking the Monadic Result

...to this end we introduce an **extraction function** unpacking a monadic value:

```
extract :: Id a -> a
extract (Id x) = x
```

This allows function `sum''` yielding again an `Int`-value (instead of a monadic one):

```
sum'' :: Tree Int -> Int
sum'' = extract . sum'
```

Example:

```
t = (Node 5 (Node 3 Nil Nil) (Node 7 Nil Nil))
sum'' t ->> (extract . sum') t
         ->> extract (sum' t)
         ->> extract (Id 15)
         ->> 15
```


Chapter 12.5.2

Numbering Tree Labels

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

12.1

12.2

12.3

The Setting

Given:

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

Objective:

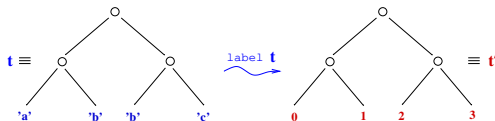
- Replace the labels of leafs by continuous natural numbers.

Illustration: The tree value $t :: \text{Tree Char}$:

```
t = Branch (Branch (Leaf 'a') (Leaf 'b'))  
          (Branch (Leaf 'b') (Leaf 'c'))
```

shall be transformed into the tree value $t' :: \text{Tree Int}$:

```
t' = Branch (Branch (Leaf 0) (Leaf 1))  
          (Branch (Leaf 2) (Leaf 3))
```



For Comparison

...we consider two approaches:

1. w/out monads
2. w/ monads

1st Approach: Straightforward w/out Monads

...using a pair of functions, one of which a `recursive` supporting function:

```
label :: Tree a -> Tree Int
label t = snd (lab t 0)

lab :: Tree a -> Int -> (Int, Tree Int)
lab (Leaf a) n = (n+1, Leaf n)
lab (Branch t1 t2) n
  = let (n1,t1') = lab t1 n
        (n2,t2') = lab t2 n1
      in (n2, Branch t1' t2')
```

Note: The solution is simple and straightforward but passing the counter value `n` through the incarnations of `lab` is `tedious` and `intricate`.

2nd Approach: Using the Spec. State Monad (1)

...using the pattern of the specialized state monad `State'`:

```
newtype Label a = Lab (Int -> (Int, a))
```

```
instance Monad Label where
```

```
Lab lt >>= flt = Lab $ \n -> let (n', x) = lt n
                                Lab lt' = flt x
                                in lt' n'
```

```
return x      = Lab (\n -> (n, x))
```

Note:

- The `$`-operator in the defining equation of `(>>=)` can be replaced by bracketing: `(\n -> let ... in lt' n')`.
- For the state monad `Label` the monad operations `(>>=)` and `return` have the types:

```
(>>=) :: Label a -> (a -> Label b) -> Label b
return :: a -> Label a
```

2nd Approach: Using the Spec. State Monad (2)

...the **renaming of labels** is now achieved by using:

```
label' :: Tree a -> Tree Int
label' t = let Lab lt = lab' t
           in snd (lt 0)
```

```
lab' :: Tree a -> Label (Tree Int)
```

```
lab' (Leaf a) = do n <- get_label
                 return (Leaf n)
```

```
lab' (Branch t1 t2) = do t1' <- lab' t1
                        t2' <- lab' t2
                        return (Branch t1' t2')
```

```
get_label :: Label Int
```

```
get_label = Lab (\n -> (n+1,n))
```

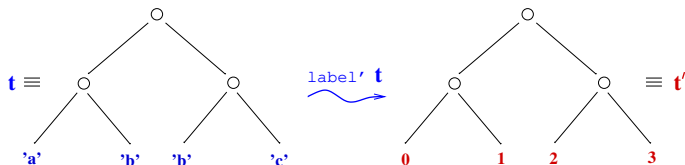
2nd Approach: Using the Spec. State Monad (3)

Example: Applying `label'` to tree value `t`:

```
t = Branch (Branch (Leaf 'a') (Leaf 'b'))  
          (Branch (Leaf 'b') (Leaf 'c'))
```

...we get as desired:

```
label' t ->> Branch (Branch (Leaf 0) (Leaf 1))  
                  (Branch (Leaf 2) (Leaf 3))  
                ≡ t'
```



Exercise 12.5.2.1

Provide (most general) type information for the **defining equations** of

1. the operations

1.1 `(>>=)`

1.2 `return`

of the instance declaration of `orangeLabel` for class `Monad`.

2. the functions

2.1 `label'`

2.2 `lab'`

2.3 `get_label`

of the monadic solution of the numbering problem.

Chapter 12.5.3

Renaming Tree Labels

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

12.1

12.2

12.3

1089/19

The Setting

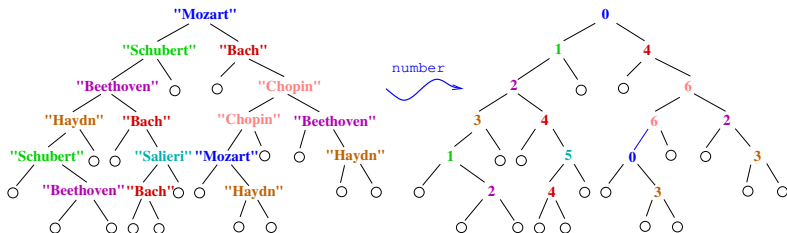
Given:

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

Objective:

- Rename labels of equal `a`-value by the same natural number.

Illustration:



Ultimate Goal

...a function `number` of type

```
number :: Eq a => Tree a -> Tree Int
```

solving this task using the `state monad State`.

Towards the Monadic Approach (1)

We start [defining](#):

```
number_tree :: Eq a => Tree a -> State a (Tree Int)
number_tree Nil = return Nil
number_tree (Node x t1 t2) =
    = do num <- number_node x
         nt1 <- number_tree t1
         nt2 <- number_tree t2
         return (Node num nt1 nt2)
```

...post-poning the implementation of [number_node](#).

Towards the Monadic Approach (2)

Additionally, we introduce a `table` type

```
type Table a = [a]
```

for storing `pairs` of the form

```
(<string>, <number of occurrences>)
```

In particular, the list (or table) value

```
[True, False]
```

encodes that `True` represents (or is associated with) `0` and `False` with `1`.

Mon. Approach: Using the State Monad (1)

...using the pattern of the `state monad` `State st`:

```
newtype State a b = St (Table a -> (Table a, b))
```

```
instance Monad (State a) where
```

```
  (St st) >>= f
```

```
    = St (\tab -> let (tab', y)      = st tab
```

```
                  (St transf) = f y
```

```
                  in transf tab')
```

```
  return x = St (\tab -> (tab, x))
```

Intuitively:

- Computing `b`-values: The (functional) `result`
- Updating tables: The `side effect`

...of the monadic operations.

Mon. Approach: Using the State Monad (2)

...providing the post-poned implementation of `number_node`:

```
number_node :: Eq a => a -> (State a) Int
number_node x = St (num_node x)

num_node :: Eq a => a -> (Table a -> (Table a, Int))
num_node x table
  | elem x table = (table, lookup x table)
  | otherwise    = (table ++ [x], length table)
-- num_node yields the position of x in the table:
-- if x is stored in the table, using lookup; if
-- not, after adding x to the table using length.

lookup :: Eq a => a -> Table a -> Int
lookup x table = ... -- Homework: Completing the
                    -- implementation of lookup.
```

Mon. Approach: Using the State Monad (3)

Putting the pieces together, `number_tree` is fully defined:

```
number_tree :: Eq a => Tree a -> State a (Tree Int)
number_tree Nil = return Nil
number_tree (Node x t1 t2)
    = do num <- number_node x
         nt1 <- number_tree t1
         nt2 <- number_tree t2
         return (Node num nt1 nt2)
```

Note, for every value `t :: Eq a => Tree a`, e.g., the tree of the illustrating example, we can conclude (functional and hence) type correctness:

```
number_tree t :: State a (Tree Int)
               ≡ (State a) (Tree Int)
               ≡ ((State a) (Tree Int))
```

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

12.1

12.2

12.3

1096/19

Mon. Approach: Using the State Monad (4)

...introducing and using the `extract` function:

```
extract :: State a b -> b
extract (St st) = snd (st [])
```

we get the implementation of the initially envisioned function `number`:

```
number :: Eq a => Tree a -> Tree Int
number = extract . number_tree
```

Example 12.5.3.1

Provide (most general) type information for the **defining equations** of

1. the operations

- 1.1 ($\gg=$)

- 1.2 `return`

of the state instance declaration of **(State a)**.

2. the functions

- 2.1 `number`

- 2.2 `number_tree`

- 2.3 `number_node`

- 2.4 `num_node`

- 2.5 `lookup`

of the monadic solution of the renaming problem.

Example 12.5.3.2

1. Implement a solver for the **tree label renaming** problem **without** using **monadic programming**.
2. Compare your **non-monadic implementation** with the **monadic** one of **Chapter 12.5.3**. Do you have a preference for one of the two? Why? Or, why not?

Chapter 12.6

Monad-Plusses

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

12.1

12.2

12.3

1100/19

Chapter 12.6.1

The Type Constructor Class MonadPlus

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

12.1

12.2

12.3

1101/19

The Type Constructor Class MonadPlus

...monads with a 'plus' operation and a 'zero' element, which is a unit for 'plus' and a zero for ($\gg=$), can be instances of the type constructor class `MonadPlus` obeying the monad-plus laws:

Type Constructor Class MonadPlus

```
class Monad m => MonadPlus m where
  mzero  :: m a
  mplus  :: m a -> m a -> m a
```

Monad-Plus Laws

<code>m >>= (_ -> mzero)</code>	<code>= mzero</code>	(MPL1)
<code>mzero >>= m</code>	<code>= mzero</code>	(MPL2)
<code>m 'mplus' mzero</code>	<code>= m</code>	(MPL3)
<code>mzero 'mplus' m</code>	<code>= m</code>	(MPL4)

Note

...`MonadPlus` instances are `monads` and thus must satisfy in addition to the `monad-plus laws` also all `monad laws`.

Intuitively, the `monad-plus` laws require from (proper) `monad-plus` instances:

- `mzero` is `left-zero` and `right-zero` for `(>>=)`.
- `mzero` is `left-unit` and `right-unit` for `plusplus`.

Programmer obligation:

- Programmers **must prove** that their instances of `MonadPlus` satisfy the `monad` and `monad-plus` laws.

Note: The `IO` monad can not be made an instance of `MonadPlus` because it is lacking an appropriate `'zero'` element.

Chapter 12.6.2

The List Monad-Plus

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

12.1

12.2

12.3

1104/19

The List Monad-Plus

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

```
instance MonadPlus [] where           -- note the over-
    mzero = []                       -- loading of Id
    mplus = (++)
```

Proof obligation: The Monad-Plus Laws

Lemma 12.6.2.1 (Soundness of List Monad-Plus)

The `[]` instance of `MonadPlus` satisfies all `monad` and `monad-plus` laws.

... `[]` is thus a proper instance of `MonadPlus`, the so-called `list monad-plus`.

Chapter 12.6.3

The Maybe Monad-Plus

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

12.1

12.2

12.3

1106/19

The Maybe Monad-Plus

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

```
instance MonadPlus Maybe where
  mzero          = Nothing
  Nothing 'mplus' ys = ys
  xs 'mplus' ys   = xs
```

Proof obligation: The Monad-Plus Laws

Lemma 12.6.3.1 (Soundness of Maybe Monad-Plus)

The `Maybe` instance of `MonadPlus` satisfies all `monad` and `monad-plus` laws.

...`Maybe` is thus a proper instance of `MonadPlus`, the so-called `maybe monad-plus`.

Exercise 12.6.3.2

1. Provide (most general) type information for
 - 1.1 the **monad-plus** laws **MPL1**, **MPL2**, **MPL3**, and **MPL4**.
 - 1.2 the defining equations of 'zero' element and 'plus' operation of the
 - 1.2.1 **Maybe** instance
 - 1.2.2 **[]** instanceof **MonadPlus**.
2. Which of the other monads considered in **Chapter 12.4** (**Identity**, **Either**, **Map**, **State**, **Input/Output**) could be reasonable instances of **MonadPlus**? Which of them are pre-defined instances?
 - 2.1 Provide instance declarations, where possible, together with (most general) type information for the defining equations of the **monad-plus** operations.
 - 2.2 Prove that all instances satisfy the **monad-plus** laws.

Chapter 12.7

Summary

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

12.1

12.2

12.3

1109/19

Summary

Monads (i.e., instances of the type constructor class **Monad**) combine features of

- **functors** and **functional composition/sequencing**:

$$(>>=) :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$$
$$c >>= k >>= k' >>= k'' >>= \dots$$

Monads are thus well-suited for

- **structuring** and **ordering** the steps of a computation

because the monadic sequencing operations **(>>=)** and **(>>)**

- allow **specifying** the order of computations explicitly.
- offer an adequately **high abstraction** by decoupling the data type forming a monad (instance) from the structure of computation.
- support equational reasoning, e.g., in terms of the **monad laws**.

Monads

...are often considered of being fanned by an aura of something

- mystic, wondrous that is difficult to grasp and lets monads appear the Holy Grail of functional programming (*'once I will have understood monads, I will have understood functional programming'*).

This (slightly odd) image of monads might be due to the origin and ties of the monad notion to (possibly often difficult considered) fields like

- philosophy, category theory, programming languages theory and semantics.

Recall

Monads in Leibniz' Philosophy:

Definition (Gottfried Wilhelm Leibniz, 1714)

[Monadology, Paragraph 1]: The **monad** we want to talk about here is nothing else as a simple substance (German: Substanz), which is contained in the composite matter (German: Zusammengesetztes); simple means as much as: to be without parts.

Monads in Category Theory (cf. Saunders Mac Lane, 1971):

Definition (Eugenio Moggi, 1989)

[LICS'89]: A **monad over a category \mathcal{C}** is a triple (T, η, μ) , where $T : \mathcal{C} \rightarrow \mathcal{C}$ is a functor, $\eta : Id_{\mathcal{C}} \rightarrow T$ and $\mu : T^2 \rightarrow T$ are natural transformations and the following equations hold:

$$\begin{aligned}\mu_{TA}; \mu_A &= T(\mu_a); \mu_A \\ \eta_{TA}; \mu_A &= id_{TA} = T(\eta_A); \mu_A\end{aligned}$$

... "a monad is a monoid in the category of endofunctors."

But Remember

...the **monad** notion in **functional programming** (in **Haskell**, too) lost its connection to the **monad** notion in **philosophy** and **category theory** (almost) completely, and hence, everything which might or might be considered a mystery or miracle.

Rather than introducing a mystery, **monads** and **monadic sequencing** in **functional programming** close a 'functional gap' between **function application**, **sequential function composition**, and **functorial mapping**.

On the Closing of a 'Functional Gap' (1)

...smashing the myth behind functional programming monads.

► Function application ('mapping over'):

$(\$)$:: $(a \rightarrow b) \rightarrow a \rightarrow b$

$g \$ x = g x$

– Special case (m a for a , m b for b):

$(\$)$:: $(m a \rightarrow m b) \rightarrow m a \rightarrow m b$

$g \$ x = g x$

► Sequential function composition ('sequencing'):

$(.)$:: $(b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$

$(f . g) x = f (g x)$

– Special case (m a for a , m b for b , m c for c):

$(.)$:: $(m b \rightarrow m c) \rightarrow (m a \rightarrow m b) \rightarrow (m a \rightarrow m c)$

$(f . g) x = f (g x)$

...one implementation fits all types: Parametric polymorphism

On the Closing of a 'Functional Gap' (2)

- ▶ Functorial mapping ('mapping over'):

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

`fmap g c = ... '(unpack, map, pack)'`

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

`(<*>)` `k c = ... '(unpack, unpack, map, pack)'`

- ▶ (Monadic) mapping plus sequencing:

`(>>=)` :: (Monad `m`) => `m` a -> (a -> `m` b) -> `m` b

`(>>=)` `c k = k 'unpack c'`

`'(unpack, map, repeat >>=)'`

...type-specific instance implementations required for 1-ary type constructors: *Ad hoc* polymorphism

Commonalities of Functions at a Glimpse

...compare (same color means 'correspond to each other'):

(.) :: (b -> c) -> (a -> b) -> (a -> c)

(f . g) x = f (g x)

(;) :: (a -> b) -> (b -> c) -> (a -> c)

(f ; g) = g . f -- pointfree

(>>.) :: a -> (a -> b) -> b

x >>. f = f x

(;<<) :: (a -> b) -> a -> b

f ;<< x = x >>. f -- Non-monadic operations

(=<<) :: Monad m => (a -> m b) -> m a -> m b -- Monadic op.

f =<< x = x >>= f

(>>=) :: Monad m => m a -> (a -> m b) -> m b

m >>= k = k 'unpack m'

(>@>) :: Monad m => (a -> m b) -> (b -> m c) -> (a -> m c)

f >@> g = \x -> (f x) >>= g

(<@<) :: Monad m => (b -> m c) -> (a -> m b) -> (a -> m c)

f <@< g = g >@> f -- pointfree

Chapter 12.8

References, Further Reading

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

12.1

12.2





12.3

1117/19





Chapter 12: Further Reading (1)

-  Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 17, Monaden)
-  Manuel Chakravarty, Gabriele Keller. *Einführung in die Programmierung mit Haskell*. Pearson Studium, 2004. (Kapitel 7, Eingabe und Ausgabe)
-  Ernst-Erich Doberkat. *Haskell: Eine Einführung für Objektorientierte*. Oldenbourg Verlag, 2012. (Kapitel 5, Ein-/Ausgabe; Kapitel 7, Monaden)
-  Andrew D. Gordon. *Functional Programming and Input/Output*. PhD thesis. University of Cambridge, British Computer Society Distinguished Dissertations in Computer Science, Cambridge University Press, 1992.




Chapter 12: Further Reading (2)

-  Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000. (Chapter 18.2, The Monad Class; Chapter 18.3, The MonadPlus Class; Chapter 18.4, State Monads)
-  Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2007. (Chapter 10.6, Class and Instance Declarations – Monadic Types)
-  John Launchbury, Simon Peyton Jones. *State in Haskell*. *Lisp and Symbolic Computation* 8(4):293-341, 1995.
-  Miran Lipovača. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, 2011. (Chapter 13, A Fistful of Monads; Chapter 14, For a Few Monads More)

Chapter 12: Further Reading (3)

-  Saunders Mac Lane. *Categories for the Working Mathematician*. Springer-V., 1971 (2nd edition, 1998).
-  Eugenio Moggi. *Computational Lambda Calculus and Monads*. In Proceedings of the 4th Annual IEEE Symposium on Logic in Computer Science (LICS'89), 14-23, 1989.
-  Eugenio Moggi. *Notions of Computation and Monads*. Information and Computation 93(1):55-92, 1991.
-  Martin Odersky. *Funktionale Programmierung*. In Informatik-Handbuch, Peter Rechenberg, Gustav Pomberger (Hrsg.), Carl Hanser Verlag, 4. Auflage, 599-612, 2006. (Kapitel 5.3, Funktionale Komposition: Monaden, Beispiele für Monaden)





Chapter 12: Further Reading (4)

-  Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Chapter 7, I/O – The I/O Monad; Chapter 14, Monads; Chapter 15, Programming with Monads; Chapter 16, Using Parsec – Applicative Functors for Parsing; Chapter 18, Monad Transformers; Chapter 19, Error Handling – Error Handling in Monads)
-  Peter Pepper, Petra Hofstedt. *Funktionale Programmierung*. Springer-V., 2006. (Kapitel 11, Beispiel: Monaden; Kapitel 17, Zeit und Zustand in der funktionalen Welt)
-  Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003. (Kapitel 21.2, Ein kommandobasiertes Ein-/Ausgabemodell; Kapitel 22.2, Kommandos; Kapitel 22.6.4, Anmerkungen zu Monaden)





Chapter 12: Further Reading (5)

-  Simon Peyton Jones, Philip Wadler. *Imperative Functional Programming*. In Conference Record of the 20 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'93), 71-84, 1993.
-  Simon Peyton Jones. *Tackling the Awkward Squad: Monadic Input/Output, Concurrency, Exceptions, and Foreign-language Calls in Haskell*. In Tony Hoare, Manfred Broy, Ralf Steinbruggen (Eds.), *Engineering Theories of Software Construction*, IOS Press, 47-96, 2001 (Presented at the 2000 Marktoberdorf Summer School).





Chapter 12: Further Reading (6)

-  Fethi Rabhi, Guy Lapalme. *Algorithms – A Functional Programming Approach*. Addison-Wesley, 1999. (Chapter 10.2, Monads)
-  Tom Schrijvers, Peter J. Stuckey, Philip Wadler. *Monadic Constraint Programming*. *Journal of Functional Programming* 19(6):663-697, 2009.
-  Michael Spivey. *A Functional Theory of Exceptions*. *Science of Computer Programming* 14(1):25-42, 1990.
-  Wouter S. Swierstra, Thorsten Altenkirch. *Beauty in the Beast: A Functional Semantics for the Awkward Squad*. In *Proceedings of the ACM SIGPLAN Workshop on Haskell (Haskell 2007)*, 25-36, 2007.






Chapter 12: Further Reading (7)

-  Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 2nd edition, 1999. (Chapter 18, Programming with actions; Chapter 18.8, Monads for functional programming)
-  Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3rd edition, 2011. (Chapter 18, Programming with monads)
-  Philip Wadler. *The Essence of Functional Programming*. In Conference Record of the 19th Annual Symposium on Principles of Programming Languages (POPL'92), 1-14, 1992.
-  Philip Wadler. *Comprehending Monads*. *Mathematical Structures in Computer Science* 2:461-493, 1992.

Chapter 12: Further Reading (8)

-  Philip Wadler. *Monads for Functional Programming*. In Johan Jeuring, Erik Meijer (Eds.), *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques*. Springer-V., LNCS 925, 24-52, 1995.
-  Philip Wadler. *How to Declare an Imperative*. In Proceedings of the 1995 International Symposium on Logic Programming (ILPS'95), Invited Presentation, MIT Press, 18-32, 1995.
-  Philip Wadler. *How to Declare an Imperative*. *ACM Computing Surveys* 29(3):240-263, 1997.
-  A (Reasonably) Comprehensive List of Tutorials on Monads: haskell.org/haskellwiki/Monad_tutorials.

Chapter 12: Monads - Background Reading (9)

-  René Descartes. *Meditationes de prima philosophia*. 1641.
-  Gottfried Wilhelm Leibniz. *Monadology* (Original in French). 90 Paragraphen, 1714.
-  Eugenio Moggi. *Computational Lambda Calculus and Monads*. In Proceedings of the 4th Annual IEEE Symposium on Logic in Computer Science (LICS'89), 14-23, 1989.
-  Eugenio Moggi. *Notions of Computation and Monads*. Information and Computation 93(1):55-92, 1991.
-  Thomas Petricek. *What We Talk about when We Talk about Monads*. The Art, Science, and Engineering of Programming 2(3), Article 12, 1-27, 2018.

Chapter 13

Arrows

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

13.1

1127/19

Chapter 13.1

Motivation

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

Motivation

...monads do not always suffice.

The higher-order type constructor class `Arrow`

- complements the type class `Monad`

with a complementary mechanism for

- composing and sequencing functions

which support 2-ary type constructors and is useful e.g. for:

- electronic circuits modelling (this chapter)
- functional reactive programming (cf. Chapter 18).

Chapter 13.2

The Type Constructor Class Arrow

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

13.1

1130/19

The Type Constructor Class Arrow

Arrows are instances of the type constructor class `Arrows` obeying the arrow laws:

```
class Arrow a where
  pure  :: (b -> c) -> a b c
      -- equivalently: pure :: ((->) b c) -> a b c
  (>>>) :: a b c -> a c d -> a b d
  first :: a b c -> a (b,d) (c,d)
```

Note:

- `pure` allows embedding of ordinary maps into the constructor class `Arrow` (the role of `pure` for maps is similar to the role of `return` in class `Monad` for values of type `a`).
- `(>>>)` serves the composition of computations.
- `first` has as an analogue on the level of ordinary functions: The function `firstfun` with
`firstfun f = \ (x,y) -> (f x, y)`

The Arrow Laws

...proper instances of `Arrow` must satisfy the following nine arrow laws:

Arrow Laws

<code>pure id >>> f = f</code>	<code>(ArrL1): identity</code>
<code>f >>> pure id = f</code>	<code>(ArrL2): identity</code>
<code>(f >>> g) >>> h = f >>> (g >>> h)</code>	<code>(ArrL3): associativity</code>
<code>pure (g . f) = pure f >>> pure g</code>	<code>(ArrL4): functor composition</code>
<code>first (pure f) = pure (f × id)</code>	<code>(ArrL5): extension</code>
<code>first (f >>> g) = first f >>> first g</code>	<code>(ArrL6): functor</code>
<code>first f >>> pure (id × g) = pure (id × g) >>> first f</code>	<code>(ArrL7): exchange</code>
<code>first f >>> pure fst = pure fst >>> f</code>	<code>(ArrL8): unit</code>
<code>first (first f) >>> pure assoc = pure assoc >>> first f</code>	<code>(ArrL9): association</code>

Utility Functions for Arrows (1)

The product map \times :

$$(\times) :: (a \rightarrow a') \rightarrow (b \rightarrow b') \rightarrow (a,b) \rightarrow (a',b')$$
$$(f \times g) \sim (a,b) = (f a, g b)$$

Regrouping arguments via `assoc`, `unassoc`, and `swap`:

$$\text{assoc} :: ((a,b),c) \rightarrow (a,(b,c))$$
$$\text{assoc} \sim (\sim(x,y),z) = (x,(y,z))$$
$$\text{unassoc} :: (a,(b,c)) \rightarrow ((a,b),c)$$
$$\text{unassoc} \sim (x,\sim(y,z)) = ((x,y),z)$$
$$\text{swap} :: (a,b) \rightarrow (b,a)$$
$$\text{swap} \sim (x,y) = (y,x)$$

The dual analogue of `first`, `map second`:

$$\text{second} :: \text{Arrow } a \Rightarrow a \ b \ c \rightarrow a \ (d,b) \ (d,c)$$
$$\text{second } f = \text{pure } \text{swap} \gg \gg \text{first } f \gg \gg \text{pure } \text{swap}$$

Utility Functions for Arrows (2)

Derived operators for arrows:

```
(***) :: Arrow a => a b c -> a b' c' ->  
                                             a (b,b') (c,c')
```

```
f *** g = first f >>> second g
```

```
(&&&) :: Arrow a => a b c -> a b c' -> a b (c,c')
```

```
f &&& g = pure (_-> (b,b)) >>> (f *** g)
```

```
idA :: Arrow a => a b b
```

```
idA = pure id
```

Chapter 13.3

The Map Arrow

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

13.1

1135/19

The Map Arrow

...making the 2-ary type constructor `(->)` an instance of `Arrow`:

```
instance Arrow (->) where
```

```
  pure f = f
```

```
  f >>> g = g . f
```

```
  first f = f × id
```

where

```
(×) :: (b -> c) -> (d -> e) -> (b,d) -> (c,e)
```

```
(f × g) ~ (bv,dv) = (f bv, g dv) :: (c,e)
```

Note: Defining `first f = \ (b,d) -> (f b, d)` is equivalent.

Proof obligation: The arrow laws

Lemma 13.3.1 (Arrow Laws for `(->)`)

The `(->)` instance of `Arrows` satisfies the 9 arrow laws.

...`(->)` is thus a proper instance of `Arrow`, the so-called `map arrow`.

The Map Arrow in More Detail

...with added type information:

```
class Arrow a where
  pure  :: ((->) b c) -> a b c
  (>>>) :: a b c -> a c d -> a b d
  first :: a b c -> a (b,d) (c,d)
```

...making `(->)` an instance of `Arrow` means constructor `a` equals `(->)`:

```
instance Arrow (->) where
  pure f      = f
  :: (->) b c  :: (->) b c

  f >>> g    = g . f
  :: (->) b c :: (->) c d :: (->) b d

  first f     = f × id
  :: (->) b c  :: (->) (b,d) (c,d)
```

Recall: Defining `first` by `first f = \ (b,d) -> (f b, d)` is equivalent.

Note

`(>>>)` :: Arrow a => a b c -> a c d -> a b d

...introduces [composition](#) for 2-ary type constructors.

This means, for the `map` instance of class `Arrow`:

```
instance Arrow (->) where
  pure f  = f
  f >>> g = g . f
  first f = f × id
```

[arrow composition](#) boils down to:

- ordinary functional composition, i.e.: `(>>>) = (.)`

Chapter 13.4

Application: Modelling Electronic Circuits

A Notion of Computation

The map `add` introduces a notion of computation:

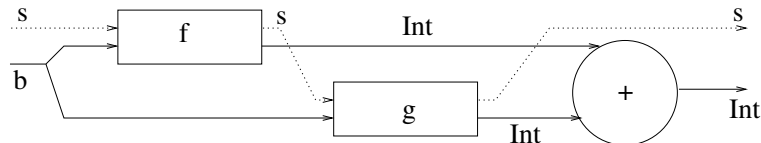
```
add :: (b -> Int) -> (b -> Int) -> (b -> Int)
add f g z = f z + g z
```

...which can be **generalized** in various ways, e.g., to

- state transformers
- non-determinism
- map transformers
- simple automata

for modelling electronic circuits.

Illustration:



Towards Modelling Electronic Circuits (1)

...generalizing `add` to `state transformers`:

```
type State s i o = (s,i) -> (s,o)
```

```
addST :: State s b Int -> State s b Int ->  
                                             State s b Int
```

```
addST f g (s,z) = let (s',x) = f (s,z)  
                    (s'',y) = g (s',z)  
                    in (s'',x+y)
```

Towards Modelling Electronic Circuits (2)

...generalizing `add` to non-determinism:

```
type NonDet i o = i -> [o]
```

```
addND :: NonDet b Int -> NonDet b Int ->
```

```
      NonDet b Int  
addND f g z = [ x+y | x <- f z, y <- g z ]
```

Towards Modelling Electronic Circuits (3)

...generalizing `add` to `map transformers`:

```
type MapTrans s i o = (s -> i) -> (s -> o)
```

```
addMT :: MapTrans s b Int -> MapTrans s b Int ->  
                                             MapTrans s b Int
```

```
addMT f g m z = f m z + g m z
```

Towards Modelling Electronic Circuits (4)

...generalizing `add` to simple automata:

```
newtype Auto i o = A (i -> (o, Auto i o))
```

```
addAuto :: Auto b Int -> Auto b Int -> Auto b Int
```

```
addAuto (A f) (A g)
```

```
  = A (\z -> let (x,f') = f z
```

```
              (y,g') = g z
```

```
              in (x+y), addAuto f' g'))
```


Putting all this together

...allows us

- modelling of synchronous circuits (with feedback loops).

Note:

- The preceding examples have in common that there is a type $A \rightsquigarrow B$ of **computations**, where inputs of type A are transformed into outputs of type B .
- The type class **Arrow** yields a sufficiently general interface to describe these commonalities uniformly and to encapsulate them in a class.

Returning to the Application

...we are now going to make the previously introduced types instances of the [type constructor class Arrow](#). To this end, we reintroduce them as new types (using [newtype](#)):

```
newtype State s i o = ST ((s,i) -> (s,o))
```

```
newtype NonDet i o = ND (i -> [o])
```

```
newtype MapTrans s i o = MT ((s -> i) -> (s -> o))
```

```
newtype Auto i o = A (i -> (o, Auto i o))
```

The State Transformer Arrow

...making `(State s)` an instance of `Arrow`:

```
newtype State s i o = ST ((s,i) -> (s,o))
```

```
instance Arrow (State s) where
```

```
  pure f          = ST (id × f)
```

```
  ST f >>> ST g = ST (g . f)
```

```
  first (ST f)   = ST (assoc . (f × id) . unassoc)
```

The State Transformer Arrow in more Detail

...with added type information:

```
class Arrow a where
  pure  :: ((->) b c) -> a b c
  (>>>) :: a b c -> a c d -> a b d
  first :: a b c -> a (b,d) (c,d)
```

...making (State s) an instance of Arrow means type constructor variable a is set to (State s):

```
newtype State s i o = ST ((s,i) -> (s,o))
```

```
instance Arrow (State s) where
  pure f           = ST (id × f)
  :: (->) b c     >>> ST g           = ST (g . f)
  :: (State s) b c >>> ST g           = ST (g . f)
  first (ST f)    = ST (assoc . (f × id) . unassoc)
  :: (State s) b c >>> ST g           = ST (g . f)
```

The Non-Determinism Arrow

...making `NonDet` an instance of `Arrow`:

```
newtype NonDet i o = ND (i -> [o])
```

```
instance Arrow NonDet where
```

```
  pure f      = ND (\b -> [f b])
```

```
  ND f >>> ND g = ND (\b -> [d | c <- f b, d <- g c])
```

```
  first (ND f) = ND (\(b,d) -> [(c,d) | c <- f b])
```

The Non-Determinism Arrow in more Detail

...with added type information:

```
class Arrow a where
  pure   :: ((->) b c) -> a b c
  (>>>) :: a b c -> a c d -> a b d
  first :: a b c -> a (b,d) (c,d)
```

...making `NonDet` an instance of `Arrow` means type constructor variable `a` is set to `NonDet`:

```
NonDet i o = ND (i -> [o])
```

```
instance Arrow NonDet where
```

```
  pure f           = ND (\b -> [f b])
  :: ((->) b c)    = ND (\b -> [f b])
  :: NonDet b c
  ND f >>> ND g    = ND (\b -> [d | c <- f b, d <- g c])
  :: NonDet b c   = ND (\b -> [d | c <- f b, d <- g c])
  :: NonDet c d   = ND (\b -> [d | c <- f b, d <- g c])
  :: NonDet b d
  first (ND f)     = ND (\(b,d) -> [(c,d) | c <- f b])
  :: NonDet b c   = ND (\(b,d) -> [(c,d) | c <- f b])
  :: NonDet (b,d) (c,d)
```

The Map Transformer Arrow

...making `(MapTrans s)` an instance of `Arrow`:

```
newtype MapTrans s i o = MT ((s -> i) -> (s -> o))
```

```
instance Arrow (MapTrans s) where
```

```
  pure f      = MT (f .)
```

```
  MT f >>> MT g = MT (g . f)
```

```
  first (MT f) = MT (zipMap . (f x id) . unzipMap)
```

where

```
zipMap      :: (s -> a, s -> b) -> (s -> (a,b))
```

```
zipMap h s = (fst h s, snd h s)
```

```
unzipMap    :: (s -> (a,b)) -> (s -> a, s -> b)
```

```
unzipMap h = (fst . h, snd . h)
```

The Map Transformer Arrow in more Detail

...with added type information:

```
class Arrow a where
  pure   :: ((->) b c) -> a b c
  (>>>) :: a b c -> a c d -> a b d
  first  :: a b c -> a (b,d) (c,d)
```

...making `(MapTrans s)` an instance of `Arrow` means type constructor variable `a` is set to `(MapTrans s)`:

```
MapTrans s i o = MT ((s -> i) -> (s -> o))
```

```
instance Arrow (MapTrans s) where
```

```
  pure f      =      MT (f .)
```

```
  :: (->) b c  :: (MapTrans s) b c
```

MT f

>>>

MT g

=

MT (g . f)

```
  :: (MapTrans s) b c  :: (MapTrans s) c d  :: (MapTrans s) b d
  first (MT f)         =      MT (zipMap . (f x id) . unzipMap)
```

```
  :: (MapTrans s) b c
```

```
  :: (MapTrans s) (b,d) (c,d)
```


The Automata Arrow

...making `Auto` an instance of `Arrow`:

```
newtype Auto i o = A (i -> (o, Auto i o))
```

```
instance Arrow Auto where
```

```
  pure f      = A (\b -> (f b, pure f))
```

```
  A f >>> A g = A (\b -> let (c,f') = f b
                           (d,g') = g c
                           in (d, f' >>> g'))
```

```
  first (A f) = A (\(b,d) -> let (c,f') = f b
                              in ((c,d),first f'))
```

The Automata Arrow in more Detail

...with added type information:

```
class Arrow a where
  pure  :: ((->) b c) -> a b c
  (>>>) :: a b c -> a c d -> a b d
  first :: a b c -> a (b,d) (c,d)
```

...making `Auto` an instance of `Arrow` means type constructor variable `a` is set to `Auto`:

```
Auto i o = A (i -> (o, Auto i o))
```

```
instance Arrow Auto where
```

```
  pure f      = A (\b -> (f b, pure f))
```

$\underbrace{\text{pure } f}_{:: \text{Auto } b \text{ } c}$

$\underbrace{= A (\backslash b \rightarrow (f \text{ } b, \text{pure } f))}_{:: \text{Auto } b \text{ } c}$

`A f >>>`

`A g`

$\underbrace{= A (\backslash b \rightarrow \text{let } (c,f') = f \text{ } b \text{ } (d,g') = g \text{ } c \text{ } \text{in } (d, f' \text{ } \gg\gg \text{ } g'))}_{:: \text{Auto } b \text{ } d}$

$\underbrace{:: \text{Auto } b \text{ } c}_{\text{first } (A \text{ } f)}$

$\underbrace{:: \text{Auto } c \text{ } d}_{= A (\backslash (b,d) \rightarrow \text{let } (c,f') = f \text{ } b \text{ } \text{in } ((c,d), \text{first } f'))}$

$\underbrace{:: \text{Auto } b \text{ } d}$

`first (A f)`

`=`

$\underbrace{A (\backslash (b,d) \rightarrow \text{let } (c,f') = f \text{ } b \text{ } \text{in } ((c,d), \text{first } f'))}_{:: \text{Auto } (b,d) \text{ } (c,d)}$

$\underbrace{:: \text{Auto } b \text{ } c}$

$\underbrace{:: \text{Auto } (b,d) \text{ } (c,d)}$

Proof Obligation: The Arrow Laws

Lemma 13.4.1 (Soundness: Arrow Laws)

The `state transformer`, `non-determinism`, `map transformer`, and `automata` instances of `Arrow` satisfy the arrow laws and are thus proper arrows.

Last but not least, it is worth noting

....that each of the considered variants of `add` results as a specialization of general combinator `addA` with the corresponding `arrow`-type:

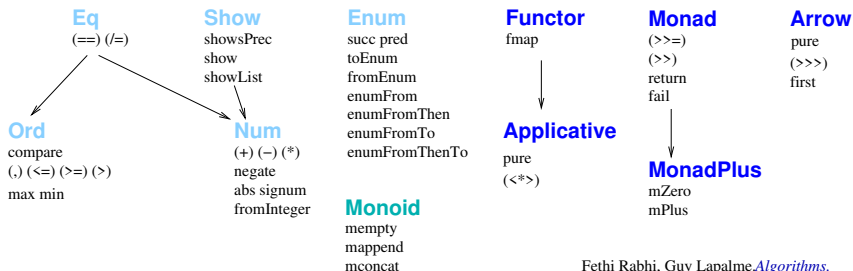
```
addA :: Arrow a => a b Int -> a b Int -> a b Int
addA f g = f &&& g >>> pure (uncurry (+))
```

Chapter 13.5

An Update on the Haskell Type Class Hierarchy

Monoids, Monads, Functors, Arrows,...

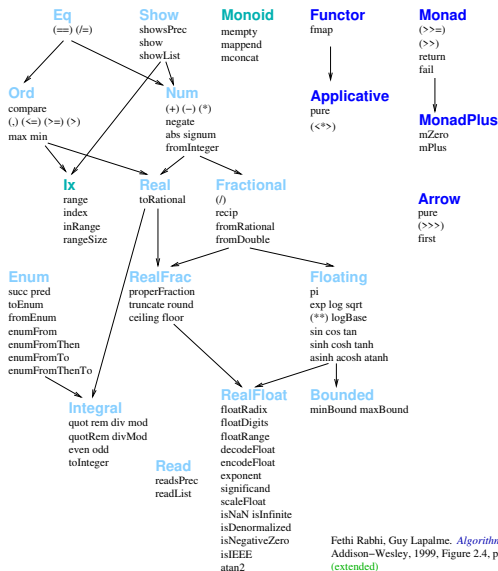
...as elements of the Haskell'98 type class hierarchy:



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

Type Classes and Type Class Functions

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



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

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

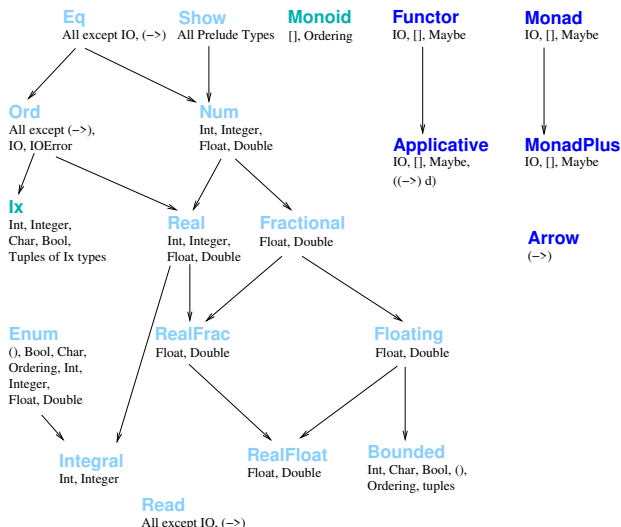
Chap. 13

13.1

1159/10

Type Classes and Type Class 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)

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

13.1

1160/19

'98 Type Class Memberships of Selected Types

Type	Instance of	Derivation
()	Read	Eq Ord Enum Bounded
[a]	Read	Eq Ord
[]	Functor Applicative Monad MonadPlus	
(a,b)	Read	Eq Ord Bounded
((->) d)	Functor Applicative	
(->)	Arrow	
Array	Functor Eq Ord Read	
Bool		Eq Ord Enum Read Bounded
Char	Eq Ord Enum Read	
Complex	Floating Read	
Double	RealFloat Read	
Either		Eq Ord Read
Float	RealFloat Read	
Int	Integral Bounded Ix Read	
Integer	Integral Ix Read	
IO	Functor Applicative Monad MonadPlus	
IOError	Eq	
Maybe	Functor Applicative Monad MonadPlus	Eq Ord Read
Ordering	Monoid	
Ratio	RealFrac Read	Eq Ord Enum Read Bounded

Fethi Rabhi, Guy Lapalme. *Algorithms*.
Addison-Wesley, 1999, Table 2.4, p. 47
(extended)

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

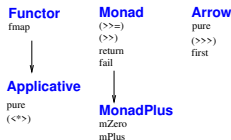
13.1

1161/19

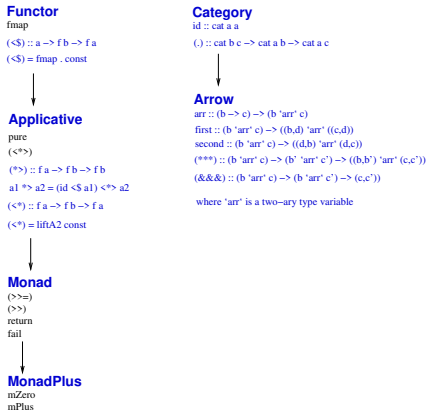
An Update on the Haskell Type Class Hierarchy

...Haskell is a research vehicle and, hence, a moving target:

Haskell'98



Haskell'98 Onwards



...for more information, check out:

<https://wiki.haskell.org/Typeclassopedia>

Chapter 13.6

Summary

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

Summing up

- Functions and programs often contain components that are ‘function-like’ ‘w/out being just functions.’
- **Arrows** define a common interface for coping w/ the “no-tion of computation” of such function-like components.
- **Monads** are a special case of **arrows**.
- Like **monads**, **arrows** allow to meaningfully structure the computation process of programs.
- **Arrow** combinators operate on ‘computations’, not on values. They are **point-free** in distinction to the ‘common case’ of functional programming.
- Analogous to the monadic case a **do-like** notational variant makes programming with **arrow** operations often easier and more suggestive (cf. literature hint at the end of the chapter), whereas the pointfree variant is more useful and advantageous for proof-theoretic reasoning.

Chapter 13.7

References, Further Reading

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11




Chap. 12

Chap. 13



13.1

1165/19

Chapter 13: Further Reading (1)

-  Martin Braun, Oleg Lobachev, Philip W. Trinder: *Arrows for Parallel Computation*. CoRR, <http://arxiv.org/abs/1801.02216>, 2018.
-  Paul Hudak, Antony Courtney, Henrik Nilsson, John Peterson. *Arrows, Robots, and Functional Reactive Programming*. In Johan Jeuring, Simon Peyton Jones (Eds.) *Advanced Functional Programming – Revised Lectures*. Springer-V., LNCS Tutorial 2638, 159-187, 2003.
-  John Hughes. *Generalising Monads to Arrows*. *Science of Computer Programming* 37:67-111, 2000.

Chapter 13: Further Reading (2)

-  Ross Paterson. *A New Notation for Arrows*. In Proceedings of the 6th ACM SIGPLAN Conference on Functional Programming (ICFP 2001), 229-240, 2001.
-  Ross Paterson. *Arrows and Computation*. In Jeremy Gibbons, Oege de Moor (Eds.), *The Fun of Programming*. Palgrave MacMillan, 201-222, 2003.

Chapter 14

Kinds

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

Chap. 14

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

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1170/19

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

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1172/19

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.
```

Exercise 14.2.1

Type Constructors as Functors and Applicatives:

Recalling the definition of the type constructor class `Functor`:

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

it becomes obvious that only `type constructors` of kind

`(* -> *)`

are eligible as possible instances of `Functor`.

What is the kind of type constructors eligible for the type constructor class `Applicative`?

Chapter 14.3

References, Further Reading

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10



Chap. 11

Chap. 12

Chap. 13

1177/19

Chapter 14: Further 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)

Es ist nicht genug zu wissen,
man muss auch anwenden.

Johann Wolfgang von Goethe (1749-1832)
dt. Dichter und Naturforscher

Part V

Applications

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1179/10

Chapter 15

Parsing

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

Chap. 14

1180/19

Parsing: Lexical and Syntactical Analysis

Parsing

- basic task of a compiler.
- umbrella term for the **lexical and syntactical analysis** of the structure of text, e.g., **source code text of programs**.
- enjoys a long history, see e.g.
 - William H. Burge. **Recursive Programming Techniques**. Addison-Wesley, 1975.

as an example of an early text book concerned with parsing.

Last but not least

- an application often used for demonstrating the power and elegance of functional programming.

Functional Approaches for Parsing

...two different but conceptually related approaches are:

1. Combinator parsing

- Graham Hutton. [Higher-Order Functions for Parsing](#). Journal of Functional Programming 2(3):323-343, 1992.

2. Monadic parsing

- Graham Hutton, Erik Meijer. [Monadic Parser Combinators](#). Technical Report NOTTCS-TR-96-4, Dept. of Computer Science, University of Nottingham, 1996.
- Graham Hutton, Erik Meijer. [Monadic Parsing in Haskell](#). Journal of Functional Programming 8(4):437-444, 1998.

which are both well-suited for building recursive descent parsers.

Chapter 15.1

Motivation

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1183/10

Informally

...the parsing problem is the following:

1. Read a sequence of objects/values of a type **a**.
2. Yield an object/value or a sequence of objects/values of a type **b**.

Illustration:

1. Read a sequence of values of type **Char**:

```
⟨ if n mod = 0 then 2*n else 2*n+1 fi ⟩
```

2. Yield a sequence of pairs of tokens and strings:

```
⟨ (if_token, ""), (var_token, "n"), (op_token, "mod"),  
  (rel_token, "="), (cst_token, "0"), (then_token, ""),  
  (cst_token, "2"), (op_token, "*"), (var_token, "n"),  
  (else_token, ""), ..., (fi_token, "") ⟩
```


Parsing Arithmetic Expressions

...a parser `p` for arithmetic expressions could be assumed to

1. read strings representing well-formed arithmetic expression
2. yield the `Exp` values matching the strings read with:

```
data Exp = Lit Int | Var Char | Op Ops Exp Exp
data Ops = Add | Sub | Mul | Div | Mod
```

Example:

```
p "((2+b)*5)"
->> Op Mul (Op Add (Lit 2) (Var 'b')) (Lit 5)
```

Note

...such a parser `p` for arithmetic expressions were

- ▶ the reverse of the `show` function:

```
show Op Mul (Op Add (Lit 2) (Var 'b')) (Lit 5)
->> "((2+b)*5)"
```

```
p "((2+b)*5)"
->> Op Mul (Op Add (Lit 2) (Var 'b')) (Lit 5)
```

- ▶ similar to the automatically derived `read` function for `Exp` values, differing, however, in the kind of arguments they accept

- `p`: Strings of the form `"((2+b)*5)"`:

```
p "((2+b)*5)"
->> Op Mul (Op Add (Lit 2) (Var 'b')) (Lit 5)
```

- `read`: Strings of the form `"Op Mul (Add (Lit ...)"`:

```
read "Op Mul (Add (Lit 2) (Var 'b')) (Lit 5)"
->> Op Mul (Op Add (Lit 2) (Var 'b')) (Lit 5)
```

Towards the Type of Parser Functions (1)

...considering `parsing` as

1. `reading` of sequences of objects of some type `a`
2. `yielding` objects or sequences of objects of some type `b`

suggests naively the `type` of `parser functions` should be:

```
type Parse_naive a b = [a] -> b
```

This, however, raises some questions. Assume, `bracket` and `number` are `parser functions` recognizing `brackets` and `numbers`, respectively:

Parser	Input	What shall be the output?
<code>bracket</code>	<code>"(xyz"</code>	<code>->> '('?</code> If so, what to do w/ <code>"xyz"</code> ?
<code>number</code>	<code>"234"</code>	<code>->> 2?</code> Or: <code>23?</code> Or: <code>234?</code>
<code>bracket</code>	<code>"234"</code>	<code>->> No result? Failure?</code>

Towards the Type of a Parser Function (2)

...this means, we have to answer:

How shall a `parser function` behave if

- (i) the input is `not completely read`?
- (ii) there are `multiple results`?
- (iii) there is a `failure`?

The latter two questions suggest the following type refinement:

```
type Parse_refined a b = [a] -> [b]
```

which allows for the previous example the following `output`:

Parser	Input	Output
<code>bracket</code>	<code>"(xyz"</code>	<code>[(']</code>
<code>number</code>	<code>"234"</code>	<code>[2,23,234]</code>
<code>bracket</code>	<code>"234"</code>	<code>[]</code>

Towards the Type of a Parser Function (3)

...we are left with answering:

- (i) What a parser function shall do with the part of the input that is not read?

Answering this question leads finally to the definite definition of the type of parser functions:

```
type Parse  $\underbrace{a \ b}_{\text{input type}} = \underbrace{[a]}_{\text{output type}} \rightarrow [(b, [a])]$ 
```

...which enables as output lists of pairs of recognized objects and left-over inputs:

Parser	Input	Output
bracket	"(xyz"	[('(', "xyz")]
number	"234"	[(2, "34"), (23, "4"), (234, "")]
bracket	"234"	[]

Informally

...if a **parser function** delivers

- the **empty list**, this signals **failure** of the analysis.
- a **non-empty list**, this signals **success** of the analysis: Every list element represents the result of a successful parse.

In the **success** case, every list element is a **pair**, whose

- **first component** is the **identified object (token)**
- **second component** is the remaining **input** which must still be analyzed.

Note, delivering **multiple results** by means of **lists**

- is known as the so-called **list of successes** technique (Philip Wadler, 1985).
- enables parsers to also analyze **ambiguous** grammars.

Reference

...the following presentation is based on:

- Simon Thompson. [Haskell – The Craft of Functional Programming](#), Addison-Wesley/Pearson, 2nd edition, 1999, Chapter 17.
- Graham Hutton, Erik Meijer. [Monadic Parsing in Haskell](#). Journal of Functional Programming 8(4):437-444, 1998.

Chapter 15.2

Combinator Parsing

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1192/10

Objective

...developing a **combinator library** for **parsing** composed of

- Four **primitive parser** functions
 - 1.&2. Two **input-independent** ones (**none**, **succeed**)
 - 3.&4. Two **input-dependent** ones (**token**, **spot**)
- Three **parser combinators** for
 1. **Alternatives** (**alt**)
 2. **Sequencing** (**(>*>)**)
 3. **Transforming** (**build**)

...forming a **universal parser basis**, which allows to construct **parser functions** at will, i.e., according to what is required by a **parsing problem**.

Chapter 15.2.1

Primitive Parsers

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1194/10

The two Input-independent Primitive Parsers

Recall:

```
type Parse a b = [a] -> [(b, [a])]
```

1. `none`, the always failing parser

```
none :: Parse a b
```

```
none _ = []
```

2. `succeed`, the always succeeding parser

```
succeed :: b -> Parse a b
```

```
succeed val inp = [(val,inp)]
```

Note:

- Parser `none` always fails. It does not accept anything.
- Parser `succeed` always succeeds without consuming its input or parts of it. In BNF-notation this corresponds to the symbol ϵ representing the empty word.

The two Input-dependent Primitive Parsers

3. `token`, the parser recognizing single objects (so-called tokens):

```
token :: Eq a => a -> Parse a a
```

```
token t (x:xs)
```

```
  | t == x      = [(t,x)]
```

```
  | otherwise   = []
```

```
token t []      = []
```

4. `spot`, the parser recognizing single objects enjoying some property:

```
spot :: (a -> Bool) -> Parse a a
```

```
spot p (x:xs)
```

```
  | p x          = [(x,xs)]
```

```
  | otherwise    = []
```

```
spot p []       = []
```

Example: Using the Primitive Parsers

...for constructing `parsers` for `simple parsing problems`:

```
bracket = token '('  
dig     = spot isDigit  
  
isDigit :: Char -> Bool  
isDigit ch = ('0' <= ch) && (ch <= '9')
```

Note: The parser functions `token` and `bracket` could also be defined using `spot`:

```
token :: Eq a => a -> Parse a a  
token t = spot (== t)  
  
bracket :: Char -> Parse Char Char  
bracket = spot (== '(')
```

Chapter 15.2.2

Parser Combinators

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1198/10

Parser Combinators

...to write more complex and powerful **parser functions**, we need in addition to **primitive parsers**

- **parser-combining functions** (or **parser combinators**)

which are re-usable **higher-order polymorphic functions**.

The Parser Combinator for Alternatives

Combining parsers as alternatives:

1. `alt`, the parser combining parsers as alternatives:

```
alt :: Parse a b -> Parse a b -> Parse a b
alt p1 p2 input = p1 input ++ p2 input
```

Intuitively: `alt` combines the results of the parses of `p1` and `p2`. The success of either of them is a success of their combination.

Example: Alternatively Combining Parsers

```
(bracket 'alt' dig) "234" ->> [] ++ [(2,"34")]  
                    ->> [(2,"34")]
```

...reflecting that numbers might start with a bracket or a digit.

```
(lit 'alt' var 'alt' opexp) "(234+7)" ->> ...
```

...reflecting that expressions are either literals, or variables or complex expressions starting with an operator.

The Parser Combinator for Sequential Comp.

Combining parsers sequentially:

2. ($>*>$), the parser combining parsers sequentially:

```
infixr 5 >*>
```

```
(>*>) :: Parse a b -> Parse a c -> Parse a (b,c)
```

```
(>*>) p1 p2 input
```

```
  = [(y,z),rem2) | (y,rem1) <- p1 input,  
                  (z,rem2) <- p2 rem1]
```

Note:

- The values $(y,rem1)$ run through the results of parser $p1$ applied to $input$. Parser $p2$ is applied to the part $rem1$ of the input that is unconsumed by $p1$ in every particular case. The results of the successful parses of $p1$ and $p2$, y and z , are returned as a pair.

Example: Sequentially Composing Parsers

...evaluating `number "24("` yields a list of two parse results `[(2, "4("), (24, "(")]`. We thus get for the composition of the parsers `number` and `bracket` applied to input `"24("`:

```
(number >*> bracket) "24("
->> [((y,z),rem2) | (y,rem1) <- [(2,"4("), (24,"(")],
      (z,rem2) <- bracket rem1 ]

->> [((2,z),rem2) | (z,rem2) <- bracket "4(" ] ++
     [((24,z),rem2) | (z,rem2) <- bracket "(" ]
->> [] ++ [((24,z),rem2) | (z,rem2) <- bracket "(" ]
->> [((24,z),rem2) | (z,rem2) <- bracket "(" ]
->> [((24,z),rem2) | (z,rem2) <- [('(',"")] ]
->> [((24,'('),"")] ]
```

The Parser Combinator for Transformations

Combining a parser with a `map` transforming the parse results:

3. `build`, the parser transforming obtained parse results:

```
build :: Parse a b -> (b -> c) -> Parse a c
build p f input = [(f x,rem) | (x,rem) <- p input]
```

Intuitively: The map argument `f` of `build` transforms the items returned by its parser argument: It `builds` something from it.

Example: Transforming Parse Results

...the parser `digList` is assumed to return a list of digit lists, whose elements are transformed by `digsToNum` into the numbers whose values they represent:

```
(digList 'build' digsToNum) "21a3"  
->> [(digsToNum x,rem) | (x,rem) <- digList "21a3"]  
->> [(digsToNum x,rem) | (x,rem) <-  
      [("2", "1a3"), ("21", "a3")]]  
->> [(digsToNum "2", "1a3"), (digsToNum "21", "a3")]  
->> [(2, "1a3"), (21, "a3")]
```

Chapter 15.2.3

Universal Combinator Parser Basis

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

Chap. 14

1206/10

Universal Combinator Parser Basis

...together, the four **primitive parsers**

1.,2.,3.,4.: **none**, **succeed**, **token**, **spot**

and the three **parser combinators**

1.,2.,3.: **alt**, **(>*>)**, **build**

form a **universal combinator parser basis**, i.e., they allow us to build any parser we might be in need of.

The Universal Parser Basis at a Glance (1)

The **priority** of the **sequencing operator**:

```
infixr 5 >*>
```

The **type** of **parser functions**:

```
type Parse a b = [a] -> [(b, [a])]
```

Two **input-independent primitive parser** functions:

1. The **always failing parser** function:

```
none :: Parse a b  
none _ = []
```

2. The **always succeeding parser** function:

```
succeed :: b -> Parse a b  
succeed val input = [(val, input)]
```


The Universal Parser Basis at a Glance (2)

Two input-dependent primitive parser functions:

3. The parser for recognizing single objects:

```
token :: Eq a => a -> Parse a a
token t = spot (==t)
```

4. The parser for recognizing single objects satisfying some property:

```
spot :: (a -> Bool) -> Parse a a
spot p (x:xs)
  | p x      = [(x,xs)]
  | otherwise = []
spot p []   = []
```

The Universal Parser Basis at a Glance (3)

Three parser combinators:

5. Alternatives

```
alt :: Parse a b -> Parse a b -> Parse a b
alt p1 p2 input = p1 input ++ p2 input
```

6. Sequencing

```
(>*>) :: Parse a b -> Parse a c -> Parse a (b,c)
(>*>) p1 p2 input
  = [(y,z),rem2) | (y,rem1) <- p1 input,
                  (z,rem2) <- p2 rem1]
```

7. Transformation

```
build :: Parse a b -> (b -> c) -> Parse a c
build p f input = [(fx,rem) | (x,rem) <- p input]
```

Chapter 15.2.4

Structure of Combinator Parsers

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

Chap. 14

The Structure of Combinator Parsers

...is usually as follows:

```
type Parse a b = [a] -> [(b, [a])]
```

```
none      :: Parse a b
```

```
succeed  :: b -> Parse a b
```

```
token    :: Eq a => a -> Parse a a
```

```
spot     :: (a -> Bool) -> Parse a a
```

```
alt      :: Parse a b -> Parse a b -> Parse a b
```

```
(>*>)    :: Parse a b -> Parse a c -> Parse a (b,c)
```

```
build    :: Parse a b -> (b -> c) -> Parse a c
```

```
list     :: Parse a b -> Parse a [b]
```

```
topLevel :: Parse a b -> [a] -> b      -- see Exam. 2,  
                                         -- Chap. 15.2.5
```

Combinator Parsers

...are well-suited for writing so-called [recursive descent parsers](#).

This is because the [parser functions](#) (summarized on the previous slide)

- are structurally similar to grammars in [BNF-form](#).
- provide for every operator of the [BNF-grammar](#) a corresponding ([higher-order](#)) [parser function](#).

These ([higher-order](#)) [parser functions](#) allow

- [combining](#) simple(r) parsers to (more) complex ones.
- are therefore called [combining forms](#), or, as a short hand, [combinators](#) (cf. Graham Hutton. [Higher-order Functions for Parsing](#). Journal of Functional Programming 2(3), 323-343, 1992).

Chapter 15.2.5

Writing Combinator Parsers: Examples

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

Chap. 14

Using the Parser Basis

...for constructing (more) complex **parser functions**.

A **parser**

1. recognizing a **list of objects** (**example 1**).
2. transforming a **string expression** into a **value** of a suitable **algebraic data type** for expressions (**example 2**).

Example 1: Parsing a List of Objects

...let `p` be a `parser` recognizing single objects. Then `list` applied to `p` is a `parser` recognizing `lists of objects`:

```
list :: Parse a b -> Parse a [b]
list p = (succeed []) 'alt'
         ((p >*> list p) 'build' (uncurry ()))
```

Intuitively

- A list of objects can be `empty`: This is recognized by the parser `succeed` called with `[]`.
- A list of objects can be `non-empty`, i.e., it consists of an object followed by a list of objects: This is recognized by the sequentially composed parsers `p` and `(list p)`:
`(p >*> list p)`.
- The parser `build`, finally, is used to turn a pair `(x,xs)` into the list `(x:xs)`.

Example 2: Parsing Arithm. Expressions (1)

...parsing arithmetic expressions like "(234+~42)*b", we shall construct the corresponding value of the algebraic data type:

```
data Expr = Lit Int | Var Char | Op Ops Expr Expr
data Ops  = Add | Sub | Mul | Div | Mod
```

Parsing "(234+~42)*b", e.g., shall yield the Exp-value:

```
Op Mul (Op Add (Lit 234) (Lit -42)) (Var 'b')
```

...according to the below assumptions for string expressions:

- Variables are the lower case characters from 'a' to 'z'.
- Literals are of the form 67, ~89, etc., where ~ is used for unary minus.
- Binary operators are +, *, -, /, %, where / and % represent integer division and modulo operation, respectively.
- Expressions are fully bracketed.
- White space is not permitted.

Example 2: Parsing Arithm. Expressions (2)

The `parser` for `string` expressions:

```
parser :: Parse Char Expr
parser = nameParse 'alt' litParse 'alt' opExpParse
```

...is composed of `three parsers` reflecting the three kinds of expressions:

- `variables` (or `variable names`)
- `literals` (or `numerals`)
- `fully bracketed operator expressions`.

Example 2: Parsing Arithm. Expressions (3)

Parsing variable names:

```
nameParse :: Parse Char Expr
nameParse = spot isName 'build' Var

isName :: Char -> Bool           -- A variable name
isName x = ('a' <= x && x <= 'z') -- must be a lower
                                   -- case character
```

Parsing literals (numerals):

```
litParse :: Parse Char Expr
litParse                                     -- A literal starts
= ((optional (token '~')) >*>              -- optionally with '~'
   (neList (spot isDigit))                 -- followed by a non-
   'build' (charlistToExpr . uncurry (++))) -- empty
                                             -- list of digits
```

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1219/10

Example 2: Parsing Arithm. Expressions (4)

Parsing fully bracketed operator expressions:

```
optExpParse :: Parse Char Expr
opExpParse      -- A non-trivial expression
= (token '('   >*> -- must start with an opening bracket,
   parser      >*> -- must be followed by an expression,
   spot isOp   >*> -- must be followed by an operator,
   parser      >*> -- must be followed by an expression,
   token ')')   -- must end with a closing bracket.
  'build' makeExpr
```

Example 2: Parsing Arithm. Expressions (5)

...required supporting `parser functions`:

```
neList    :: Parse a b -> Parse a [b]
```

```
optional :: Parse a b -> Parse a [b]
```

where

- `neList p` recognizes a non-empty list of the objects recognized by `p`.
- `optional p` recognizes an object recognized by `p` or succeeds immediately.

Note: `neList`, `optional`, and some other supporting functions including

- `isOp`
- `charlistToExpr`

are still be defined, left here as an `exercise`.

Example 2: Parsing Arithm. Expressions (6)

...we are left with defining a **top-level parser** function, which converts a string into an expression when called with **parser**:

Converting a string into the expression it represents:

```
topLevel :: Parse a b -> [a] -> b
topLevel p input
  = case results of
      [] -> error "parse unsuccessful"
      _  -> head results
  where
      results = [found | (found, []) <- p input]
```

Note:

- The parse of an input is successful, if the result contains at least one parse, in which all the input has been read.
- `topLevel parser "(234+~42)*b)" ->>`
`Op Mul (Op Add (Lit 234) (Lit -42)) (Var 'b')`

Chapter 15.3

Monadic Parsing

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1223/10

Monadic Parsing

...complements the concept of **combining forms** underlying **combinator parsing** with the one of **monads**.

Since monads are 1-ary type constructors, the **type of parser functions** must be adjusted accordingly:

```
newtype Parser a = Parse (String -> [(a,String)])
```

output type input type

At the same time, we re-use the **convention** of **Chapter 13.2** that **delivery** of the

- **empty list** signals **failure** of a parsing analysis.
- **non-empty list** signals **success** of a parsing analysis: each element of the **list** is a pair, whose **first component** is the **identified object (token)** and whose **second component** the **input** which is still to be parsed.

Chapter 15.3.1

The Parser Monad

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1225/10

The Parser Monad

Recalling the definition of type class `Monad`:

```
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b -- (>>), failure are
  return :: a -> m a                  -- not needed: Their de-
                                      -- fault implement. apply.
```

...making `Parser`, a 1-ary type constructor, an instance of `Monad`:

```
instance Monad Parser where
  p >>= f = Parse (\cs -> concat [(parse (f a)) cs' |
                                (a,cs') <- (parse p) cs])
  return a = Parse (\cs -> [(a,cs)])
```

where

```
parse :: (Parser a) -> (String -> [(a,String)])
parse (Parse p) = p
```

Remarks on the Parser Monad

```
instance Monad Parser where
```

```
  p >>= f = Parse (\cs -> concat [(parse (f a)) cs' |  
                                   (a,cs') <- (parse p) cs])
```

```
  return a = Parse (\cs -> [(a,cs)])
```

Intuitively:

- The parser `(return a)` succeeds without consuming any of the argument string, and returns the single value `a`.
- `parse` denotes a deconstructor function for parsers defined by `parse (Parse p) = p`.
- The parser sequence `p >>= f` applies first parser `(parse p)` to the argument string `cs` yielding a list of results of the form `(a,cs')`, where `a` is a value and `cs'` is a string. For each such pair the parser `(parse (f a))` is applied to the unconsumed input string `cs'`. The result is a list of lists which is concatenated to give the final list of results.

Proof Obligation: The Monad Laws

...`Parser` satisfies the `monad laws` and is thus a valid instance of `Monad`. We have:

Lemma 15.3.1.1 (Soundness of Parser Monad)

1. `return a >>= f = f a`
2. `p >>= return = p`
3. `p >>= (\a -> (f a >>= g)) = (p >>= (\a -> f a)) >>= g`

Note:

- `(>>=)` being `associative` allows suppression of parentheses when parsers are applied sequentially.
- `return` being `left-unit` and `right-unit` for `(>>=)` allows some parser definitions to be simplified.

Chapter 15.3.2

Parsers as Monadic Operations

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1229/10

Monadic Operations as Parsers

...`Parser` as an instance of `Monad` provides us already with two important parser functions, a `primitive parser` and a (`monadic`) `parser combinator`:

1. `return`, the `always succeeding parser`
6. `(>>=)`, a `combinator for sequentially combining parsers`

which are the `monadic` counterparts of the `combinator parsers`

1. `succeed`
6. `(>*>)`

of `Chapter 15.2.1` and `15.2.2`, respectively.

The `MonadPlus` instance of `Parser` will give us two more `parser functions`...

In more Detail

...the `MonadPlus` (cf. [Chapter 12.6](#)) instance of `Parser`:

```
class Monad m => MonadPlus m where
  mzero  :: m a
  mplus  :: m a -> m a -> m a
```

will provide us with the [parser functions](#):

2. `mzero`, the [always failing parser](#)
5. `mplus` (via `(++)`), the [parser for alternatives](#) (or [non-deterministic choice](#))

which are the [monadic counterparts](#) of the [parser combinators](#)

2. `none`
5. `alt`

of [Chapter 15.2.1](#) and [15.2.2](#), respectively.

The Parser Monad-Plus

...yields the new parser functions `mzero` and `mplus`:

```
instance MonadPlus Parser where
  -- The always failing parser
  mzero = Parse (\cs -> [])

  -- The parser combinator for alternatives:
  p 'mplus' q = Parse (\cs -> parse p cs ++ parse q cs)
```

Note: `mplus` can yield more than one result; the value of `(parse p cs ++ parse q cs)` can be a list of any length. In this sense `mplus` is considered to explore parsers *alternatively* (or, in this sense, *non-deterministically*).

Proof Obligations: The Monad-Plus Laws

...we can prove that `Parser` satisfies the `Monad-Plus` laws:

Lemma 15.3.2.1 (Soundness of Parser Monad-Plus)

1. `p >>= (_ -> mzero) = mzero`
2. `mzero >>= p = mzero`
3. `mzero 'mplus' p = p`
4. `p 'mplus' mzero = p`

This means:

- `mzero` is `left-zero` and `right-zero` for `(>>=)`.
- `mzero` is `left-unit` and `right-unit` for `mplus`.

Moreover

...we can prove the following laws:

Lemma 15.3.2.2

1. $p \text{ 'mplus' } (q \text{ 'mplus' } r) = (p \text{ 'mplus' } q) \text{ 'mplus' } r$
2. $(p \text{ 'mplus' } q) \gg= f = (p \gg= f) \text{ 'mplus' } (q \gg= f)$
3. $p \gg= (\lambda a \rightarrow f a \text{ 'mplus' } g a) = (p \gg= f) \text{ 'mplus' } (p \gg= g)$

This means:

- `mplus` is associative.
- `(>>=)` distributes through `mplus`.

Chapter 15.3.3

Universal Monadic Parser Basis

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1235/10

Towards a Universal Monadic Parser Basis

...in order to arrive at a [universal monadic parser basis](#) as in [Chapter 15.2.3](#) we are left with defining [monadic](#) counterparts of the

- 3.,4. [primitive](#) parsers [token](#) and [spot](#).
- 6. parser [combinator](#) [build](#).

The Monadic Counterpart of Parser `spot`

...parser `sat` recognizes `single characters` satisfying a given property:

```
sat  :: (Char -> Bool) -> Parser Char
sat  p =
  do {c <- item; if p c then return c else zero}
```

`sat` is the `monadic` counterpart of the parser function `spot` of Chapter 15.2.1.

The Monadic Counterpart of Parser token

...parser `char` recognizes `single characters`; it is defined in terms of parser `sat`:

```
char  :: Char -> Parser Char
char c = sat (== c)
```

`char` is the `monadic` counterpart of the parser function `token` of [Chapter 15.2.1](#).

The Universal Monadic Parser Basis (1)

The `type` of parser functions:

```
newtype Parser a = Parse (String -> [(a,String)])
```

Two `input-independent` primitive parser functions:

1. The `always succeeding` parser function:

```
return :: a -> Parser a  
return a = Parse (\cs -> [(a,cs)])
```

2. The `always failing` parser function:

```
mzero :: Parser a  
mzero = Parse (\cs -> [])
```

The Universal Monadic Parser Basis (2)

Two **input-dependent primitive parser** functions:

3. The **parser** for recognizing **single objects**:

```
char :: Char -> Parser Char
char c = sat (== c)
```

4. The **parser** for recognizing **single objects satisfying some property**:

```
sat :: (Char -> Bool) -> Parser Char
sat p =
  do {c <- item; if p c then return c else zero}
```


The Universal Monadic Parser Basis (3)

Three parser combinators:

5. Alternatives

```
mplus :: Parser a -> Parser a -> Parser a
p 'mplus' q =
  Parse (\cs -> parse p cs ++ parse q cs)
```

6. Sequencing

```
(>>=) :: Parser a -> (a -> Parser b) -> Parser b
p >>= f =
  Parse (\cs -> concat [(parse (f a)) cs' |
                        (a,cs') <- (parse p) cs])
```

7. Transformation

```
mbuild :: Parser a -> (a -> b) -> Parser b
mbuild p f inp = ... (completion left as homework)
```

Chapter 15.3.4

Utility Parsers

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1242/19

Utility Parsers (1)

Consuming the first character of an input string, if it is non-empty, and **failing** otherwise:

```
item :: Parser Char
item = Parse (\cs -> case cs of
                    ""      -> []
                    (c:cs) -> [(c,cs)])
```

Parsing a specific **string**:

```
string :: String -> Parser String
string ""      = return ""
string (c:cs) = do char c; string cs; return (c:cs)
```

Utility Parsers (2)

The **deterministically** selecting parser:

```
(+++) :: Parser a -> Parser a -> Parser a
p +++ q
= Parse (\cs -> case parse (p 'mplus' q) cs of
             []      -> []
             (x:xs) -> [x])
```

Note:

- **(+++)** shows the same behavior as **mplus**, but yields at most one result (in this sense 'deterministically'), whereas **mplus** can yield several ones (in this sense 'non-deterministically')
- **(+++)** satisfies all of the previously listed properties of **mplus**.

Utility Parsers (3)

Applying a `parser p` repeatedly:

```
-- zero or more applications of p
many :: Parser a -> Parser [a]
many p = many1 p +++ return []

-- one or more applications of p
many1 :: Parser a -> Parser [a]
many1 p = do a <- p; as <- many p; return (a:as)
```

Note: As above, useful parsers are often `recursively defined`.

Utility Parsers (4)

A **variant** of the parser `many` with **interspersed applications** of parser `sep`, whose result values are thrown away:

```
sepby :: Parser a -> Parser b -> Parser [a]
p 'sepby' sep = (p 'sepby1' sep) +++ return []

sepby1 :: Parser a -> Parser b -> Parser [a]
p 'sepby1' sep = do a <- p
                    as <- many (do sep; p)
                    return (a:as)
```

Utility Parsers (5)

Repeated applications of a parser `p` separated by applications of a parser `op`, whose result value is an operator which is assumed to associate to the left, and used to combine the results from the `p` parsers in `chainl` and `chainl1`:

```
chainl :: Parser a -> Parser (a -> a -> a)
                                             -> a -> Parser a
```

```
chainl p op a = (p 'chainl1' op) +++ return a
```

```
chainl1 :: Parser a -> Parser (a -> a -> a)
                                             -> Parser a
```

```
p 'chainl1' op = do a <- p; rest a
                  where rest a = (do f <- op
                                     b <- p
                                     rest (f a b))
                               +++ return a
```

Utility Parsers (6)

Handling `white space`, `tabs`, `newlines`, etc.

- Parsing a string with blanks, tabs, and newlines:

```
space :: Parser String
space = many (sat isSpace)
```

- Parsing a token by means of a parser `p` skipping any 'trailing' space:

```
token :: Parser a -> Parser a
token p = do {a <- p; space; return a}
```

- Parsing a symbolic token:

```
symb :: String -> Parser String
symb cs = token (string cs)
```

- Applying a parser `p` and throwing away any leading space:

```
apply :: Parser a -> String -> [(a,String)]
apply p = parse (do {space; p})
```


Note

...[parsers](#) handling [comments](#) or [keywords](#) can be defined in a similar fashion allowing together avoidance of a dedicated [lexical analysis](#) (for token recognition), which typically precedes parsing.

Chapter 15.3.5

Structure of a Monadic Parser

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1250/10

The Typical Structure of a Monadic Parser

...using the sequencing operator ($\gg=$) or the syntactically sugared `do`-notation:

<code>p1 >>= \a1 -></code>	<code>do a1 <- p1</code>
<code>p2 >>= \a2 -></code>	<code> a2 <- p2</code>
<code>...</code>	<code>...</code>
<code>pn >>= \an -></code>	<code> an <- pn</code>
<code>f a1 a2 ... an</code>	<code>f a1 a2 ... an</code>

...the latter one **equivalently expressed** in just **one line**, if so desired:

```
do {a1 <- p1; a2 <- p2;...; an <- pn; f a1 a2...an}
```

Recall: The expressions `ai <- pi` are called **generators** (since they generate values for the variables `ai`). Generators of the form `ai <- pi` can be replaced by `pi`, if the generated value will not be used afterwards.

Note

...the intuitive, natural **operational reading** of such a **monadic parser**:

- Apply parser **p1** and call its result value **a1**.
- Apply subsequently parser **p2** and call its result value **a2**.
- ...
- Apply subsequently parser **pn** and call its result value **an**.
- Combine finally the intermediate results by applying an appropriate function **f**.

Note, most typically **f = return (g a1 a2 ... an)**; for an exception see parser **chain1** in **Chapter 15.3.4**, which needs to parse 'more of the argument string' before it can return a result.

Chapter 15.3.6

Writing Monadic Parsers: Examples

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1253 / 10

Example 1: A Simple Parser

...writing a parser `p` which

- `reads` three characters,
- `drops` the second character of these, and
- `returns` the first and the third character as a pair.

Implementation:

```
p :: Parser (Char,Char)
p = do c <- item; item; d <- item; return (c,d)
```

Example 2: Parsing Arithm. Expressions (1)

...built up from single **digits**, the operators **+**, **-**, *****, **/**, and **parentheses**, respecting the usual precedence rules for additive and multiplicative operators.

Grammar for arithmetic expressions:

```
expr ::= expr addop term | term
term  ::= term mulop factor | factor
factor ::= digit | (expr)
digit ::= 0 | 1 | ... | 9

addop ::= + | -
mulop ::= * | /
```

Example 2: Parsing Arithm. Expressions (2)

The Parsing Problem:

Parsing expressions and evaluating them on-the-fly (yielding their integer values) using the `chain1` combinator of [Chapter 15.3.4](#) to implement the left-recursive production rules for `expr` and `term`.

Example 2: Parsing Arithm. Expressions (3)

The implementation of the parser `expr`:

```
expr  :: Parser Int
addop :: Parser (Int -> Int -> Int)
mulop :: Parser (Int -> Int -> Int)

expr = term 'chainl1' addop
term = factor 'chainl1' mulop

factor =
  digit +++ do {symb "("; n <- expr; symb ")"; return n}
digit =
  do {x <- token (sat isDigit); return (ord x - ord '0')}

addop = do {symb "+"; return (+)}
      +++ do {symb "-"; return (-)}

mulop = do {symb "*"; return (*)}
      +++ do {symb "/"; return (div)}
```

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1257/10

Example 2: Parsing Arithm. Expressions (4)

...using the parser.

Parsing and evaluating the string " 1 - 2 * 3 + 4 " on-the-fly by calling:

```
apply expr " 1 - 2 * 3 + 4 "
```

yields the singleton list:

```
[(-1, "")]
```

which is the desired result.

Chapter 15.4

Summary

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1259/10

In Conclusion

...combinator and monadic parsing rely (in part) on different language features but are quite similar in spirit as becomes obvious when opposing their primitives and combinators:

	Combinator Parsing	Monadic Parsing
Primitive Parsers	none succeed token spot	mzero return char sat
Parser Combinators	alt (>*>) build	mplus (>>=) mbuild

Invaluable

...for `combinator` (as well as `monadic`) `parsing` are:

- ▶ **Higher-order functions:** `Parse a b` (like `Parser a`) is of a functional type; all parser combinators are thus **higher-order functions**.
- ▶ **Polymorphism:** The type `Parse a b` is polymorphic: We do need to be specific about either the input or the output type of the parsers we build. Hence, the parser combinators mentioned above can immediately be reused for tokens of any other data type (in the examples, these were lists and pairs, characters, and expressions).
- ▶ **Lazy evaluation:** 'On demand' generation of the possible parses, automatical backtracking (the parsers will backtrack through the different options until a successful one is found).

Chapter 15.5

References, Further Reading

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10





Chap. 11

Chap. 12





Chap. 13

Chap. 14




Chapter 15.2: Further Reading (1)

-  Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice-Hall, 2nd edition, 1998. (Chapter 11, Parsing)
-  Manuel Chakravarty, Gabriele Keller. *Einführung in die Programmierung mit Haskell*. Pearson Studium, 2004. (Kapitel 13.1.3, Ausdrücke parsen; Kapitel B.1, Der Quellcode für den Ausdrucksparser)
-  Jan van Eijck, Christina Unger. *Computational Semantics with Functional Programming*. Cambridge University Press, 2010. (Chapter 9, Parsing)
-  Jeroen Fokker. *Functional Parsers*. In Johan Jeuring, Erik Meijer (Eds.), *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques*. Springer-V., LNCS 925, 1-23, 1995.




Chapter 15.2: Further Reading (2)

-  Steve Hill. *Combinators for Parsing Expressions*. Journal of Functional Programming 6(3):445-464, 1996.
-  Graham Hutton. *Higher-Order Functions for Parsing*. Journal of Functional Programming 2(3):323-343, 1992.
-  Pieter W.M. Koopman, Marinus J. Plasmeijer. *Efficient Combinator Parsers*. In Proceedings of the 10th International Workshop on the Implementation of Functional Languages (IFL'98), Selected Papers, Springer-V., LNCS 1595, 120-136, 1999.
-  Matthew Might, David Darais, Daniel Spiewak. *Parsing with Derivatives – A Functional Pearl*. In Proceedings of the 16th ACM International Conference on Functional Programming (ICFP 2011), 189-195, 2011.



Chapter 15.2: Further Reading (3)

-  Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003. (Kapitel 18.6.2, Ausdrücke als Bäume)
-  Peter Pepper, Petra Hofstedt. *Funktionale Programmierung*. Springer-V., 2006. (Kapitel 3, Parser als Funktionen höherer Ordnung)
-  S. Doaitse Swierstra. *Combinator Parsing: A Short Tutorial*. In Language Engineering and Rigorous Software Development, International LerNet ALFA Summer School 2008, Revised Tutorial Lectures. Springer-V., LNCS 5520, 252-300, 2009.






Chapter 15.2: Further Reading (4)

-  S. Doaitse Swierstra, P. Azero Alcocer. *Fast, Error Correcting Parser Combinators: A Short Tutorial*. In Proceedings SOFSEM'99, Theory and Practice of Informatics, 26th Seminar on Current Trends in Theory and Practice of Informatics, Springer-V., LNCS 1725, 111-129, 1999.
-  S. Doaitse Swierstra, Luc Duponcheel. *Deterministic, Error Correcting Combinator Parsers*. In: *Advanced Functional Programming, Second International Spring School*, Springer-V., LNCS 1129, 184-207, 1996.
-  Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley/Pearson, 2nd edition, 1999. (Chapter 17.5, Case study: parsing expressions)






Chapter 15.2: Further Reading (5)

-  Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley/Pearson, 3rd edition, 2011. (Chapter 17.5, Case study: parsing expressions)
-  Philip Wadler. *How to Replace Failure with a List of Successes*. In Proceedings of the 4th International Conference on Functional Programming and Computer Architecture (FPCA'85), Springer-V., LNCS 201, 113-128, 1985.

Chapter 15.3: Further Reading (6)

-  Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer Verlag, 2011. (Kapitel 19.10.5, λ -Parser)
-  William H. Burge. *Recursive Programming Techniques*. Addison- Wesley, 1975.
-  Andy Gill, Simon Marlow. *Happy – The Parser Generator for Haskell*. University of Glasgow, 1995.
www.haskell.org/happy
-  Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2007. (Chapter 8, Functional parsers)
-  Graham Hutton, Erik Meijer. *Monadic Parsing in Haskell*. Journal of Functional Programming 8(4):437-444, 1998.

Chapter 15.3: Further Reading (7)

-  Graham Hutton, Erik Meijer. *Monadic Parser Combinators*. Technical Report NOTTCS-TR-96-4, Dept. of Computer Science, University of Nottingham, 1996.
-  Daan Leijen. *Parsec, a free Monadic Parser Combinator Library for Haskell*, 2003.
legacy.cs.uu.nl/daan/parsec.html
-  Daan Leijen, Erik Meijer. *Parsec: A Practical Parser Library*. Electronic Notes in Theoretical Computer Science 41(1), 20 pages, 2001.
-  Simon Peyton Jones, David Lester. *Implementing Functional Languages: A Tutorial*. Prentice Hall, 1992.
-  Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Chapter 10, Code Case Study: Parsing a Binary Data Format)

Chapter 16

Logic Programming Functionally

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

Chap. 14

Logic Programming Functionally

Declarative programming

- **Characterizing:** Programs are declarative assertions about a problem rather than imperative solution procedures.
- **Hence:** Emphasizes the 'what,' rather than the 'how.'
- **Important styles:** Functional and logic programming.

If each of these two styles is appealing for itself

- (features of) functional and logic programming

uniformly combined in just one language should be even more appealing.

Question

- Can and shall (features of) functional and logic programming be uniformly combined?

Yes, they can and should

...a recent article highlights important [benefits](#) of [combining](#) the paradigm features of [functional](#) and [logic programming](#)

- Sergio Antoy, Michael Hanus. [Functional Logic Programming](#). Communications of the ACM 53(4):74-85, 2010.

shedding thereby some light on this question.

...part of it is summarized in [Chapter 16.1](#).

Chapter 16.1

Motivation

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1273/10

Chapter 16.1.1

On the Evolution of Programming Languages

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1274 / 19

The Evolution of Programming Languages (1)

...a continuous and ongoing process of **hiding** the **computer hardware** and the **details of program execution** by the stepwise **introduction of abstractions**.

Assembly languages

- introduce **mnemonic instructions** and **symbolic labels** for **hiding machine codes** and **addresses**.

FORTRAN

- introduces **arrays** and **expressions in standard mathematical notation** for **hiding registers**.

ALGOL-like languages

- introduce **structured statements** for **hiding gotos** and **jump labels**.

Object-oriented languages

- introduce **visibility levels** and **encapsulation** for **hiding the representation of data** and the **management of memory**.

Evolution of Programming Languages (2)

Declarative languages, most prominently functional and logic languages

- remove assignment and other control statements for hiding the order of evaluation.
 - A declarative program is a set of logic statements describing properties of the application domain.
 - The execution of a declarative program is the computation of the value(s) of an expression wrt these properties.

This way:

- The programming effort in a declarative language shifts from encoding the steps for computing a result to structuring the application data and the relationships between application components.
- Declarative languages are similar to formal specification languages but executable.

Chapter 16.1.2

Functional vs. Logic Languages

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1277 / 19

Functional vs. Logic Languages

Functional languages

- are based on the notion of **mathematical function**.
- **programs** are **sets of functions** that operate on data structures and are defined by equations using case distinction and recursion.
- provide **efficient, demand-driven evaluation strategies** that support infinite structures.

Logic languages

- are based on **predicate logic**.
- **programs** are **sets of predicates** defined by restricted forms of logic formulas, such as Horn clauses (implications).
- provide **non-determinism** and **predicates** with **multiple input/output modes** that offer code reuse.

Functional Logic Languages (1)

...there are many: Curry, TOY, Mercury, Escher, Oz, HAL,...

Some of them in [more detail](#):

- [Curry](#)

Michael Hanus, Herbert Kuchen, Juan Jose Moreno-Navarro. [Curry: A Truly Functional Logic Language](#). In Proceedings of the ILPS'95 Workshop on Visions for the Future of Logic Programming, 95-107, 1995.

See also: Michael Hanus (Ed.). [Curry: An Integrated Functional Logic Language \(vers. 0.8.2, 2006\)](#).

<http://www.curry-language.org/>

Functional Logic Languages (2)

- TOY

Francisco J. López-Fraguas, Jaime Sánchez-Hernández.
[TOY: A Multi-paradigm Declarative System](#). In Proceedings of the 10th International Conference on Rewriting Techniques and Applications (RTA'99), Springer-V., LNCS 1631, 244-247, 1999.

- Mercury

Zoltan Somogyi, Fergus Henderson, Thomas Conway.
[The Execution Algorithm of Mercury: An Efficient Purely Declarative Logic Programming Language](#). Journal of Logic Programming 29(1-3):17-64, 1996.

See also: [The Mercury Programming Language](#)

<http://www.mercurylang.org>

Chapter 16.1.3

A Curry Appetizer

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

Chap. 14

A Curry Appetizer (1)

Two important Curry operators:

- `?`, denoting *nondeterministic choice*.
- `:=`, indicating that an *equation is to be solved* rather than an operation to be defined.

Example: Regular expressions and their semantics

```
data RE a = Lit a
          | Alt (RE a) (RE a)
          | Conc (RE a) (RE a)
          | Star (RE a)
```

```
sem :: RE a -> [a]
sem (Lit c)      = [c]
sem (Alt r s)    = sem r ? sem s
sem (Conc r s)   = sem r ++ sem s
sem (Star r)     = [] ? sem (Conc r (Star r))
```

A Curry Appetizer (2)

- Evaluating the semantics of the regular expression

`abstar`:

non-deterministically
`sem abstar ->> ["a", "ab", "abb"]`
`where abstar = Conc (Lit 'a') (Star (Lit 'b'))`

- Checking whether some word `w` is in the language of a regular expression `re`:

`sem re ::= w`

- Checking whether some string `s` contains a word generated by a regular expression `re` (similar to Unix's `grep` utility):

`xs ++ sem re ++ ys ::= s`

Note: `xs` and `ys` are free!

Chapter 16.1.4

Outline

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1284/10

Combining Functional and Logic Programming

...some principal approaches for **combining** their **features**:

- **Ambitious**: Designing a new programming language enjoying features of both programming styles (e.g., **Curry**, **Mercury**, etc.).
- **Less ambitious**: Implementing an interpreter for one style using the other style.
- **Even less ambitious**: Developing a **combinator library** allowing us to write **logic programs** in **Haskell**.

Here

...we follow the [last approach](#) as proposed by [Michael Spivey](#) and [Silvija Seres](#) in:

- Michael Spivey, Silvija Seres. [Combinators for Logic Programming](#). In Jeremy Gibbons, Oege de Moor (Eds.), [The Fun of Programming](#). Palgrave MacMillan, 177-199, 2003.

Central are:

- [Combinators](#)
- [Monads](#)
- [Combinator and monadic programming](#).

Benefits and Limitations

...of this **combinator** approach compared to approaches striving for fully **functional/logic programming languages**:

- Less costly

but also

- less expressive and (likely) less performant.

Chapter 16.2

The Combinator Approach

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1288 / 10

Chapter 16.2.1

Introduction

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1289/10

Three Key Problems

...are to be solved in the course of **developing** this **approach**:

Modelling

1. logic programs yielding (possibly) **multiple answers**
 \rightsquigarrow using the **lists of successes** technique
2. the **evaluation/search strategy** inherent to logic programs
 \rightsquigarrow encapsulating the search strategy in '**search monads**'
3. **logical variables** (no distinction between input and output variables)
 \rightsquigarrow realizing **unification**

Key Problem 1: Multiple Answers

...can easily be handled (re-) using the technique of

- lists of successes (lazy lists) (Philip Wadler, 1985)

Intuitively

- Any function of type $(a \rightarrow b)$ can be replaced by a function of type $(a \rightarrow [b])$.
- Lazy evaluation ensures that the elements of the result list (i.e., the list of successes) are provided as they are found, rather than as a complete list after termination of the computation.

Key Problem 2: Evaluation/Search Strategies

...dealt with investigating an illustrating [running example](#).

This is [factoring](#) of [natural numbers](#): Decomposing a positive integer into the set of pairs of its factors, e.g.:

Integer	Factor pairs
24	(1,24), (2,12), (3,8), (4,6), ..., (24,1)

Obviously, this [problem](#) (instance) is [solved](#) by:

```
factor :: Int -> [(Int,Int)]  
factor n = [ (r,s) | r<-[1..n], s<-[1..n], r*s == n ]
```

In fact, we get:

```
factor 24 ->>  
[(1,24), (2,12), (3,8), (4,6), (6,4), (8,3), (12,2), (24,1)]
```

Note

When implementing the 'obvious' solution we exploit explicit domain knowledge:

- ▶ Most importantly the domain fact:

$$- r * s = n \Rightarrow r \leq n \wedge s \leq n$$

which allows us to restrict our search to a finite space:

$$[1..24] \times [1..24]$$

Often, however, such knowledge is not available:

- ▶ Generally, the search space cannot be restricted a priori!

In the following, we thus consider the factoring problem as a

- ▶ search problem over the infinite 2-dimensional search space:

$$[1..] \times [1..]$$

Back to the Running Example

...adapting function `factor` straightforward to the **infinite search space** `[1..] × [1..]` yields:

```
factor :: Int -> [(Int,Int)]  
factor n = [(r,s) | r<-[1..], s<-[1..], r*s == n]  
                infinite infinite
```

Applying `factor` to the argument `24` yields:

```
factor 24  
->> [(1,24)]
```

...followed by an **infinite wait**.

This is **useless** and of **no practical value!**

The Problem: Unfair Depth-first Search

...the two-dimensional space is searched in a **depth-first order**:

	1	2	3	4	5	6	7	8	9	...
1	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)	(1,8)	(1,9)	...
2	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)	(2,8)	(2,9)	...
3	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)	(3,8)	(3,9)	...
4	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)	(4,8)	(4,9)	...
5	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)	(5,8)	(5,9)	...
6	(6,1)	(6,2)	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)	(6,8)	(6,9)	...
7	(7,1)	(7,2)	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)	(7,8)	(7,9)	...
8	(8,1)	(8,2)	(8,3)	(8,4)	(8,5)	(8,6)	(8,7)	(8,8)	(8,9)	...
9	(9,1)	(9,2)	(9,3)	(9,4)	(9,5)	(9,6)	(9,7)	(9,8)	(9,9)	...
...

This **search order** is **unfair**: Pairs in **rows 2 onwards** will **never** be reached and considered for being a factor pair.

Chapter 16.2.2

Diagonalization

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1297 / 10

Diagonalization to the Rescue (1)

...searching the infinite number of finite diagonals ensures fairness, i.e., every pair will deterministically be visited after a finite number of steps:

	1	2	3	4	5	6	7	8	9	...
1	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)	(1,8)	(1,9)	...
2	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)	(2,8)	(2,9)	...
3	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)	(3,8)	(3,9)	...
4	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)	(4,8)	(4,9)	...
5	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)	(5,8)	(5,9)	...
6	(6,1)	(6,2)	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)	(6,8)	(6,9)	...
7	(7,1)	(7,2)	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)	(7,8)	(7,9)	...
8	(8,1)	(8,2)	(8,3)	(8,4)	(8,5)	(8,6)	(8,7)	(8,8)	(8,9)	...
9	(9,1)	(9,2)	(9,3)	(9,4)	(9,5)	(9,6)	(9,7)	(9,8)	(9,9)	...
...

- Diagonal 1: [(1,1)]
- Diagonal 2: [(1,2), (2,1)]
- Diagonal 3: [(1,3), (2,2), (3,1)]
- Diagonal 4: [(1,4), (2,3), (3,2), (4,1)]
- Diagonal 5: [(1,5), (2,4), (3,3), (4,2), (5,1)]
- ...

Diagonalization to the Rescue (2)

In fact, on visiting the infinite number of finite diagonals, every pair (i, j) of the infinite 2-dimensional search space $[1..] \times [1..]$ is deterministically reached after a finite number of steps as illustrated below:

	1	2	3	4	5	6	7	...
1	$(1,1)_1$	$(1,2)_2$	$(1,3)_4$	$(1,4)_7$	$(1,5)_{11}$	$(1,6)_{16}$	$(1,7)_{22}$...
2	$(2,1)_3$	$(2,2)_5$	$(2,3)_8$	$(2,4)_{12}$	$(2,5)_{17}$	$(2,6)_{23}$	$(2,7)_{30}$...
3	$(3,1)_6$	$(3,2)_9$	$(3,3)_{13}$	$(3,4)_{18}$	$(3,5)_{24}$	$(3,6)_{31}$	$(3,7)_{39}$...
4	$(4,1)_{10}$	$(4,2)_{14}$	$(4,3)_{19}$	$(4,4)_{25}$	$(4,5)_{32}$	$(4,6)_{40}$	$(4,7)_{49}$...
5	$(5,1)_{15}$	$(5,2)_{20}$	$(5,3)_{26}$	$(5,4)_{33}$	$(5,5)_{41}$	$(5,6)_{50}$	$(5,7)_{60}$...
6	$(6,1)_{21}$	$(6,2)_{27}$	$(6,3)_{34}$	$(6,4)_{42}$	$(6,5)_{51}$	$(6,6)_{61}$	$(6,7)_{72}$...
7	$(7,1)_{28}$	$(7,2)_{35}$	$(7,3)_{43}$	$(7,4)_{52}$	$(7,5)_{62}$	$(7,6)_{73}$	$(7,7)_{85}$...
8	$(8,1)_{36}$	$(8,2)_{44}$	$(8,3)_{53}$	$(8,4)_{63}$	$(8,5)_{74}$	$(8,6)_{86}$	$(8,7)_{99}$...
9	$(9,1)_{45}$	$(9,2)_{54}$	$(9,3)_{64}$	$(9,4)_{75}$	$(9,5)_{87}$	$(9,6)_{100}$	$(9,7)_{114}$...
...

Exercise 16.2.2.1

The previous figure illustrates that there is a **bijective** map

$$m : \mathbb{N} \rightarrow \mathbb{N} \times \mathbb{N}$$

How can m formally be defined?

Implementing Diagonalization (1)

...function `diagprod` realizes the **diagonalization strategy**: It enumerates the cartesian product of its argument lists in a **fair order**, i.e., every element is enumerated after some **finite amount of time**:

```
diagprod :: [a] -> [b] -> [(a,b)]
diagprod xs ys =
  [ (xs!!i, ys!!(n-i)) | n<-[0..], i<-[0..n] ]
                        infinite      finite
```

E.g., applied to the **infinite** 2-dimensional space $[1..] \times [1..]$, `diagprod` ejects every pair (x,y) of $[1..] \times [1..]$ in **finite time**:

```
(1,1), (1,2), (2,1), (1,3), (2,2), (3,1), (1,4), (2,3),
(3,2), (4,1), (1,5), (2,4), (3,3), (4,2), (5,1), (1,6),
(2,5), ..., (6,1), (1,7), (2,6), ..., (7,1), ...
```

Implementing Diagonalization (2)

```
diagprod :: [a] -> [b] -> [(a,b)]
```

```
diagprod xs ys = [(xs!!i, ys!!(n-i)) | n<-[0..], i<-[0..n]]
```

n	i	n-i	(xs!!i, ys!!(n-i))	([1..]!!i, [1..]!!(n-i))	#	Diag. #
0	0	0	(xs!!0,ys!!0)	(1,1)	1	1
1	0	1	(xs!!0,ys!!1)	(1,2)	2	2
1	1	0	(xs!!1,ys!!0)	(2,1)	3	
2	0	2	(xs!!0,ys!!2)	(1,3)	4	3
2	1	1	(xs!!1,ys!!1)	(2,2)	5	
2	2	0	(xs!!2,ys!!0)	(3,1)	6	
3	0	3	(xs!!0,ys!!3)	(1,4)	7	4
3	1	2	(xs!!1,ys!!2)	(2,3)	8	
3	2	1	(xs!!2,ys!!1)	(3,2)	9	
3	3	0	(xs!!3,ys!!0)	(4,1)	10	
4	0	4	(xs!!0,ys!!4)	(1,5)	11	5
4	1	3	(xs!!1,ys!!3)	(2,4)	12	
4	2	2	(xs!!2,ys!!2)	(3,3)	13	
4	3	1	(xs!!3,ys!!1)	(4,2)	14	
4	4	0	(xs!!4,ys!!0)	(5,1)	15	
...	

Back to the Running Example

...let's adjust `factor` in a way such that it explores the search space of pairs in a **fair order** using **diagonalization**:

```
factor :: Int -> [(Int,Int)]
factor n =
  [(r,s) | (r,s) <- diagprod  $\underbrace{[1..]}_{\text{infinite}}$   $\overbrace{[1..]}^{\text{infinite}}$ , r*s == n]
```

Applying now `factor` to the argument `24`, we obtain:

```
factor 24 ->>
[(4,6), (6,4), (3,8), (8,3), (2,12), (12,2), (1,24), (24,1)]
```

...i.e., we obtain **all results**, followed by an **infinite wait**.

Of course, this is not surprising, since the search space is **infinite**.

Chapter 16.2.3

Diagonalization with Monads

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1304/19

Finite Lists, Infinite Streams, Monads

...in the following we **conceptually** distinguish between:

- `[a]`: **Finite** lists.
- `Stream a`: **Infinite** lists defined as type alias by:
`type Stream a = [a]`

Note: Distinguishing between `(Stream a)` for **infinite lists** and `[a]` for **finite lists** is conceptually and notationally only as is made explicit by defining `(Stream a)` as a type alias of `[a]`.

Like `[]`, `Stream` is a **1-ary type constructor** and can thus be made an **instance** of type class `Monad`:

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

The Stream Monad

...since `(Stream a)` is a type alias of `[a]`, the `stream` and the `list monad` coincide; the `bind (>>=)` and `return` operation of the `stream monad`

- `(>>=) :: Stream a -> (a -> Stream b) -> Stream b`
- `return :: a -> Stream a`

are thus defined as in [Chapter 12.4.2](#):

```
instance Monad Stream where
  xs >>= f = concat (map f xs)
  return x = [x]           -- yields the singleton list
```

Note: The monad operations `(>>)` and `fail` are not relevant in the following, and thus omitted.

Notational Benefit (1)

...the **monad** operations **return** and **(>>=)** for **lists** and **streams** allow us to avoid or replace **list comprehension**:

E.g., the **expression**

```
[(x,y) | x <- [1..], y <- [10..]]
```

using **list comprehension** is equivalent to the expression

```
[1..] >>= (\x -> [10..] >>= (\y -> return (x,y)))
```

using **monad operations**; this is made explicit by stepwise unfolding the monadic expression yielding first the **equivalent expression**:

```
concat (map (\x -> [(x,y) | y <- [10..]]) [1..])
```

and second the **equivalent expression**:

```
concat  
(map (\x -> concat (map (\y -> [(x,y)]) [10..])) [1..])
```

Notational Benefit (2)

By exploiting the **general rule** that

```
do x1 <- e1; x2 <- e2; ... ; xn <- en; e
```

is a **shorthand** for

```
e1 >>= (\x1 -> e2 >>= (\x2 -> ... >>= (\xn -> e)...))
```

...Haskell's **do**-notation allows an **even more compact equivalent** representation:

```
do x <- [1..]; y <- [10..]; return (x,y)
```

Note

...exploring the pairs of the **search space** using the **stream monad** is **not yet fair**.

E.g., the expression:

```
do x <- [1..]; y <- [10..]; return (x,y)
```

yields the **infinite list** (i.e., **stream**):

```
[(1,10), (1,11), (1,12), (1,13), (1,14), ...]
```

..the **fairness** issue is only handled by defining **another monad**.

Towards a Fair Binding Operation ($\gg=$)

...idea: Embedding [diagonalization](#) into ($\gg=$).

To this end, we introduce a new polymorphic type [Diag](#):

```
newtype Diag a = MkDiag (Stream a) deriving Show
```

together with a utility function for [stripping off](#) the data constructor [MkDiag](#):

```
unDiag :: Diag a -> a  
unDiag (MkDiag xs) = xs
```

The Diag(onalization) Monad

...making `Diag` an instance of the type constructor class `Monad`:

```
instance Monad Diag where
  return x          = MkDiag [x]
  MkDiag xs >>= f =
    MkDiag (concat (diag (map (unDiag . f) xs)))
```

where `diag` rearranges the values into a *fair order*:

```
diag :: Stream (Stream a) -> Stream [a]
diag []          = []
diag (xs:xss) =
  lzw (++) [ [x] | x <- xs] ([ ] : diag xss)
```

Utility Function `lzw`

...using itself the utility function `lzw` ('like `zipWith`.')

```
lzw :: (a -> a -> a) -> Stream a ->
      Stream a -> Stream a
```

```
lzw f [] ys          = ys
```

```
lzw f xs []         = xs
```

```
lzw f (x:xs) (y:ys) = (f x y) : (lzw f xs ys)
```

Note: `lzw` equals `zipWith` except that the non-empty remainder of a non-empty argument list is attached, if one of the argument lists gets empty.

Note

...for monad `Diag` holds:

- `return` yields the singleton list.
- `undiag` strips off the constructor added by the function `f :: a -> Diag b`.
- `diag` arranges the elements of the list into a `fair order` (and works equally well for finite and infinite lists).

Illustrating

...the idea underlying the map `diag`:

Transform an infinite list of infinite lists:

```
[[x11, x12, x13, x14, ..], [x21, x22, x23, ..], [x31, x32, ..], ..]
```

into an infinite list of finite diagonals:

```
[[x11], [x12, x21], [x13, x22, x31], [x14, x23, x32, ..], ..]
```

This way, we get:

```
do x<-MkDiag [1..]; y<-MkDiag [10..]; return (x,y)
->> MkDiag [(1,10), (1,11), (2,10), (1,12), (2,11),
            (3,10), (1,13), ..
```

which means, we are done:

- The pairs are delivered in a fair order!

Back to the Factoring Problem

...the **current status** of our approach:

- ▶ Generating pairs (in a fair order): **Done**.
- ▶ Selecting the pairs being part of the solution: **Still open**.

Next, we are going to tackle the **selection problem**, i.e., filtering out the pairs (r, s) satisfying the equality $r \times s = n$, by:

- ▶ **Filtering with conditions!**

To this end, we introduce a new type constructor class **Bunch**.

Chapter 16.2.4

Filtering with Conditions

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1316/19

The Type Constructor Class Bunch

...is defined by:

```
class Monad m => Bunch m where
  -- Empty result (or no answer)
  zero :: m a
  -- All answers in xm or ym
  alt :: m a -> m a -> m a
  -- Answers yielded by 'auxiliary calculations'
  -- (for now, think of wrap in terms of the
  -- identity, i.e., wrap = id)
  wrap :: m a -> m a
```

Note: `zero` allows to express that a set of answers is empty;
`alt` allows to join two sets of answers.

Making [] and Diag Instances of Bunch

...making (lazy) lists and Diag instances of Bunch:

```
instance Bunch [] where
```

```
  zero      = []
```

```
  alt xs ys = xs ++ ys
```

```
  wrap xs   = xs
```

```
instance Bunch Diag where
```

```
  zero      = MkDiag []
```

```
  alt (MkDiag xs) (MkDiag ys)      -- shuffle in the  
    = MkDiag (shuffle xs ys)      -- interest of
```

```
  wrap xm = xm                    -- fairness
```

```
shuffle :: [a] -> [a] -> [a]
```

```
shuffle [] ys      = ys
```

```
shuffle (x:xs) ys = x : shuffle ys xs
```

Note: wrap will only be used be used in Chapter 16.2.5 onwards.

Filtering with Conditions using test

Using `zero`, the function `test`, which might not look useful at first sight, yields the `key` for `filtering`:

```
test :: Bunch m => Bool -> m ()           -- () type idf.  
test b = if b then return () else zero -- () value idf.
```

In fact, all `do`-expressions `filter` as desired, i.e., the multiples of 3 from the streams `[1..]` and `MkDiag [1..]`, respectively:

```
do x <- [1..]; () <- test (x `mod` 3 == 0); return x  
->> [3,6,9,12,15,18,21,24,27,30,33,..]
```

```
do x <- [1..]; test (x `mod` 3 == 0); return x  
->> [3,6,9,12,15,18,21,24,27,30,33,..]
```

```
do x <- MkDiag [1..]; test (x `mod` 3 == 0); return x  
->> MkDiag [3,6,9,12,15,18,21,24,27,30,33,..]
```

A note on test

In more [detail](#):

```
do x <- [1..];  
  :: Int  :: [] Int  
  () <- test (x 'mod' 3 == 0);  
  :: ()  [()] :: [] (), if true  
         [] :: [] (), if false  
  return x  
  :: [] Int
```

...if `test` evaluates to `true`, it returns the value `()`, and the rest of the program is evaluated. If it evaluates to `false`, it returns `zero`, and the rest of the program is skipped for this value of `x`. This means, `return x` is only reached and evaluated for those values of `x` with `x 'mod' 3` equals `0`.

Nonetheless

...we are **not** yet **done** as the below example shows:

```
do r <- MkDiag [1..]; s <- MkDiag [1..];  
  test (r*s==24); return (r,s)  
->> MkDiag [(1,24)
```

...followed again by an **infinite wait**.

Why is that?

The above **expression** is **equivalent** to:

```
do x <- MkDiag [1..]  
  (do y <- MkDiag [1..]; test(x*y==24);  
   return (x,y))
```

Why is that? (1)

...this means the **generator** for y is merged with the subsequent **test** to the (sub-) expression:

```
do y <- MkDiag [1..]; test(x*y==24); return (x,y)
```

Intuitively

- This expression yields for a given value of x all values of y with $x * y = 24$.
- For $x = 1$ the answer $(1, 24)$ will be found, in order to then search in vain for further fitting values of y .
- For $x = 5$ we thus would not observe any output, since an infinite search would be initiated for values of y satisfying $5 * y = 24$.

Why is that? (2)

...the deeper reason for this (undesired) behaviour:

The `bind` operation (`>>=`) of `Diag` is **not** associative, i.e.,

$$xm \gg= (\backslash x \rightarrow f \ x \gg= g) \neq (xm \gg= f) \gg= g$$

...does **not** hold! Or, equivalently expressed using `do`:

$$\begin{aligned} & \text{do } x \leftarrow xm; y \leftarrow f \ x; g \ y \\ &= xm \gg= (\backslash x \rightarrow f \ x \gg= (\backslash y \rightarrow g \ y)) \\ &= xm \gg= (\backslash x \rightarrow f \ x \gg= g) \\ &= (xm \gg= f) \gg= g \\ &= (xm \gg= (\backslash x \rightarrow f \ x)) \gg= (\backslash y \rightarrow g \ y) \\ &= \text{do } y \leftarrow (\text{do } x \leftarrow xm; f \ x); g \ y \end{aligned}$$

...does **not** hold.

Overcoming the Problem

...frankly, `Diag` is not a valid instance of `Monad`, since it fails the monad law of associativity for `(>>=)`. The order of applying generators is thus essential.

For taking this into account, the generators are explicitly pairwise grouped together to ensure they are treated fairly by diagonalization:

```
do (x,y) <- (do u <- MkDiag [1..];
            v <- MkDiag [1..]; return (u,v))
    test (x*y==24); return (x,y)
->> MkDiag [(4,6), (6,4), (3,8), (8,3), (2,12), (12,2),
            (1,24), (24,1)]
```

...yields now all results, followed, of course, by an infinite wait (due to an infinite search space).

This means, the problem is fixed. We are done.

Note

Getting all results followed by an infinite wait is

- ▶ the best we can hope for if the search space is infinite.

Explicit grouping is

- ▶ only required because `Diag` is not a valid instance of `Monad` since its bind operation (`>>=`) fails to be *associative*. If it were, both expressions would be equivalent and explicit grouping unnecessary.

Next, we will strive for

- ▶ avoiding/replacing *infinite waiting* by indicating *search progress*, i.e., by indicating from time to time that a(nother) result *has not (yet) been found*.

Chapter 16.2.5

Indicating Search Progress

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1326/10

Indicating Search Progress

...to this end, we introduce a new type `Matrix` together with a `cost-guided diagonalization search`, a true `breadth search`.

Intuitively

- Values of type `Matrix`: `Infinite lists of finite lists`.
- `Goal`: A program which yields a matrix of answers, where row i contains all answers which can be computed with costs $c(i)$ specific for row i .
- `Indicating progress`: If the list returned as row k is the empty list, this means '`nothing found`,' i.e., the set of solutions which can be computed with costs $c(k)$ is empty.

The Type Matrix

The new type `Matrix`:

```
newtype Matrix a =  
  MkMatrix (Stream [a]) deriving Show
```

...and a utility function for `stripping off` the data constructor:

```
unMatrix :: Matrix a -> a  
unMatrix (MkMatrix xm) = xm
```


Towards Matrix an Instance of Bunch (1)

...preliminary reasoning about the required operations and their properties:

```
-- Matrix with a single row
```

```
return x = MkMatrix [[x]]
```

```
-- Matrix without rows
```

```
zero = MkMatrix []
```

```
-- Concatenating corresponding rows
```

```
alt (MkMatrix xm) (MkMatrix ym) =  
  MkMatrix (lzw (++) xm ym)
```

```
-- Taking care of the cost management!
```

```
wrap (MkMatrix xm) = MkMatrix ([]:xm)
```

Towards Matrix an Instance of Bunch (2)

```
{- (>>=) is essentially defined in terms of bindm; it handles the data constructor MkMatrix which is not done by bindm. -}
```

```
(>>=) :: Matrix a -> (a -> Matrix b) -> Matrix b  
(MkMatrix xm) >>= f = MkMatrix (bindm xm (unMatrix . f))
```

```
{- bindm is almost the same as (>>=) but without bothering about MkMatrix; it applies f to all the values in xm and collects together the results in a matrix according to their total cost: these are the costs of the argument of f given by xm plus the cost of computing its result. -}
```

```
bindm :: Stream[a] -> (a -> Stream[b]) -> Stream [b]  
bindm xm f = map concat (diag (map (concatAll . map f) xm))
```

```
{- A variant of the concat function using lzw. -}
```

```
concatAll :: [Stream [b]] -> Stream [b]  
concatAll = foldr (lzw (++)) []
```

Making Matrix an Instance of Bunch

...now we are ready to make `Matrix` an instance of the type constructor classes `Monad` and `Bunch`:

```
instance Monad Matrix where
  return x          = MkMatrix [[x]]
  (MkMatrix xm) >>= f = MkMatrix (bindm xm (unMatrix . f))
```

```
instance Bunch Matrix where
  zero                = MkMatrix []
  alt( MkMatrix xm) (MkMatrix ym) =
    MkMatrix (lzw (++) xm ym)
  wrap (MkMatrix xm) =    -- 'wrap xm' yields a matrix w/
    MkMatrix ([]:xm)     -- the same answers but each
                          -- with a cost one higher than
                          -- its cost in 'xm'

intMat = MkMatrix [[n] | n <- [1..]] -- intMat replaces
                                      -- stream [1..]
```

Using intMat and Matrix

...consider the expression:

```
do r <- intMat; s <- intMat; test(r*s==24); return (r,s)
->> MkMatrix [ [], [], [], [], [], [], [], [], [(4,6), (6,4)],
              [(3,8), (8,3)], [], [], [(2,12), (12,2)], [], [], [],
              [], [], [], [], [], [], [], [(1,24), (24,1)], [], [], [], ...
```

Intuitively

- **Diagonals 1 to 8:** No factor pairs of 24 were found (indicated by []).
- **Diagonal 9:** The factor pairs (4,6) and (6,4) were found.
- **Diagonal 10:** The factor pairs (3,8) and (8,3) were found.
- **Diagonals 11 to 12:** No factor pairs of 24 were found (ind'd by []).
- **Diagonal 13:** The factor pairs (2,12) and (12,2) were found.
- ...

...if a diagonal d does not contain a valid factor pair, we get []; otherwise we get the list of valid factor pairs located in d .

I.e., we are done: Infinite waiting is replaced by progress indication!

Illustrating the Location

...of the factor pairs of 24 in the diagonals of the search space by $!(\cdot, \cdot)!$:

	1	2	3	4	5	6	7	8	9	
1	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)	(1,8)	(1,9)	Chap..3
2	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)	(2,8)	(2,9)	Chap..4
3	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)	!(3,8)!	(3,9)	Part III
4	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	!(4,6)!	(4,7)	(4,8)	(4,9)	Chap..5
5	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)	(5,8)	(5,9)	Chap..6
6	(6,1)	(6,2)	(6,3)	!(6,4)!	(6,5)	(6,6)	(6,7)	(6,8)	(6,9)	Part IV
7	(7,1)	(7,2)	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)	(7,8)	(7,9)	Chap..7
8	(8,1)	(8,2)	!(8,3)!	(8,4)	(8,5)	(8,6)	(8,7)	(8,8)	(8,9)	Chap..8
9	(9,1)	(9,2)	(9,3)	(9,4)	(9,5)	(9,6)	(9,7)	(9,8)	(9,9)	Chap..9
...	Chap..10

Contents

Part I

Chap. 1

Part II

Chap..2

Chap..3

Chap..4

Part III

Chap..5

Chap..6

Part IV

Chap..7

Chap..8

Chap..9

Chap..10

Chap. 11

Chap. 12

Chap. 13

1333/10

Chapter 16.2.6

Selecting a Search Strategy

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1334/10

An Array of Search Strategies

...is now at our disposal, namely

1. Depth search ([1..])
2. Diagonalization (MkDiag [[n] | n<-[1..]])
3. Breadth search (MkMatrix [[n] | n<-[1..]])

...and we can choose each of them at the very last moment, just by picking the right monad when calling a function:

```
-- Picking the desired search strategy by choos-
-- ing m accordingly at the time of calling factor
factor :: Bunch m => Int -> m (Int, Int)
factor n = do r <- choose [1..]; s <- choose [1..];
            test (r*s==n); return (r,s)

choose :: Bunch m => Stream a -> m a
choose (x:xs) = wrap (return x 'alt' choose xs)
```

Picking a Search Strategy at Call Time

...specifying the result type of `factor` when calling it selects the `search monad` and thus the `search strategy` applied.

Illustrated in terms of our running example:

```
-- Depth Search: Picking Stream
factor 24 :: Stream (Int,Int)
->> [(1,24)

-- Diagonalization Search: Picking Diag
factor 24 :: Diag (Int, Int)
->> MkDiag [(4,6),(6,4),(3,8),(8,3),(2,12),(12,2),
           (1,24),(24,1)

-- Breadth Search w/ Progress Indication: Picking Matrix
factor 24 :: Matrix (Int, Int)
->> MkMatrix [[], [], [], [], [], [], [], [], [(4,6),(6,4)],
             [(3,8),(8,3)], [], [], [(2,12),(12,2)], [], [], [],
             [], [], [], [], [], [], [], [(1,24),(24,1)], [], [], [], ...
```


Summarizing our Progress so Far

...recall the 3 key problems we have or had to deal with.

Modelling

1. logic programs yielding (possibly) multiple answers: Done (using lazy lists).
2. the evaluation strategy inherent to logic programs: Done.
 - The search strategy implicit to logic programming languages has been made explicit. The type constructors and type classes of Haskell allow even different search strategies and to pick one conveniently at call time.
3. logical variables (i.e., no distinction between input and output variables): Still open!

Next

...we tackle this [third problem](#), i.e.:

Modelling

- ▶ [logical variables](#) (i.e., no distinction between input and output variables).

Common for evaluating logic programs

- ▶ ...not a pure simplification of an initially completely given expression but a simplification of an expression containing variables, for which appropriate values have to be determined. In the course of the computation, variables can be replaced by other subexpressions containing variables themselves, for which then appropriate values have to be found.

Fundamental: [Substitution](#), [unification](#).

Chapter 16.2.7

Terms, Substitutions, Unification, and Predicates

Terms (1)

...towards [logical variables](#) — we introduce a type for [terms](#):

Terms

```
data Term = Int Int
          | Nil
          | Cons Term Term
          | Var Variable deriving Eq
```

...will describe values of [logic variables](#).

Named variables and [generated variables](#)

```
data Variable = Named String
              | Generated Int deriving (Show, Eq)
```

...will be used for [formulating queries](#), respectively, [evolve in the course of the computation](#).

Terms (2)

Utility functions for transforming

- ▶ a **string** into a **named variable**:

```
var :: String -> Term
var s = Var (Named s)
```

- ▶ a **list of integers** into a **term**:

```
list :: [Int] -> Term
list xs = foldr Cons Nil (map Int xs)
```

Substitutions (1)

Substitutions

```
newtype Subst = MkSubst [(Var,Term)]
```

...essentially [mappings from variables to terms](#).

Support functions for substitutions:

```
unSubst :: Subst -> [(Var,Term)]
```

```
unSubst (MkSubst s) = s
```

```
idsubst :: Subst
```

```
idsubst = MkSubst []
```

```
extend :: Var -> Term -> Subst -> Subst
```

```
extend x t (MkSubst s) = MkSubst ((x:t):s)
```

Substitutions (2)

Applying a substitution:

```
apply :: Subst -> Term -> Term
apply s t =          -- Replace each variable
  case deref s t of  -- in t by its image under s
    Cons x xs -> Cons (apply s x) (apply s xs)
    t'         -> t'
```

where

```
deref :: Subst -> Term -> Term
deref s (Var v) =
  case lookup v (unSubst s) of
    Just t   -> deref s t
    Nothing  -> Var v
deref s t = t
```

Term Unification (1)

...unifying terms:

```
unify :: (Term, Term) -> Subst -> Maybe Subst
```

```
unify (t,u) s =
```

```
  case (deref s t, deref s u) of
```

```
    (Nil, Nil) -> Just s
```

```
    (Cons x xs, Cons y ys) ->
```

```
      unify (x,y) s >>= unify (xs, ys)
```

```
    (Int n, Int m) | (n==m) -> Just s
```

```
    (Var x, Var y) | (x==y) -> Just s
```

```
    (Var x, t) -> if occurs x t s
```

```
      then Nothing
```

```
      else Just (extend x t s)
```

```
    (t, Var x) -> if occurs x t s
```

```
      then Nothing
```

```
      else Just (extend x t s)
```

```
    (_,_) -> Nothing
```


Term Unification (2)

where

```
occurs :: Variable -> Term -> Subst -> Bool
occurs x t s =
  case deref s t of
    Var y      -> x == y
    Cons y ys  -> occurs x y s || occurs x ys s
    _         -> False
```

Predicates: Modelling Logic Programs (1)

...in our scenario `m` is of type `bunch`.

`Logic programs` are of type:

```
type Pred m = Answer -> m Answer
```

...intuitively, applied to an 'input' `answer` which provides the information that is already decided about the values of variables, an array of new answers is computed, each of them satisfying the constraints expressed in the program.

`Answers` are of type:

```
newtype Answer = MkAnswer (Subst, Int)
```

...intuitively, the `substitution` carries the information about the values of variables; the `integer value` counts how many variables have been generated so far allowing to generate fresh variables when needed.

Predicates: Modelling Logic Programs (2)

Initial 'input' answer:

```
initial :: Answer
initial = MkAnswer (idsubst, 0)
```

Logical program run: Predicate `p` as query is applied to the initial 'input' answer:

```
run :: Bunch m => Pred m -> m Answer
run p = p initial
```

Example: Choosing `Stream` for `m` allows evaluating the predicate `append` (defined later):

```
run (append (list [1,2],list [3,4],var "z"))
                                     :: Stream Answer
->> [{z=[1,2,3,4]}]                  -- an appropriate show
                                     -- function is assumed
```

Chapter 16.2.8

Combinators for Logic Programs

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1348/10

Combinator (`==`): Equality

...combinator (`==`) ('equality' of terms) allows us to build simple predicates, e.g.:

```
run (var "x" == Int 3) :: Stream Answer
  ->> [{x=3}]
```

Implementation of (`==`) by means of `unify`:

```
(==) :: Bunch m => Term -> Term -> Pred m
(t == u) (MkAnswer (s,n)) =      -- Pred m = (Answer -> m Answer)
  case unify (t,u) s of
    Just s' -> return (MkAnswer (s',n))
    Nothing -> zero
```

Intuitively: If the argument terms `t` and `u` can be unified wrt the input answer `MkAnswer (s,n)`, the most general unifier is returned as the output answer; otherwise there is no answer.

Combinator (&&&): Conjunction

...combinator (&&&) allows us to connect predicates **conjunctively**, e.g.:

```
run (var "x" ::= Int 3 &&& var "y" ::= Int 4)
    :: Stream Answer
->> [{x=3,y=4}]
run (var "x" ::= Int 3 &&& var "x" ::= Int 4)
    :: Stream Answer
->> []
```

Implementation of (&&&) by means of the **bind operation** (>>=) of monad **bunch**:

```
(&&&) :: Bunch m => Pred m -> Pred m -> Pred m
(p &&& q) s = p s >>= q
-- or equivalently using the do-notation:
do t <- p s; u <- q t; return u
```

Combinator (|||): Disjunction

...combinator (|||) allows us to connect predicates **disjunctively**, e.g.:

```
run (var "x" ::= Int 3 ||| var "x" ::= Int 4)
    :: Stream Answer
->> [{x=3,x=4}]
```

Implementation of (|||) by means of the **alt operation** of monad **bunch**:

```
(|||) :: Bunch m => Pred m -> Pred m -> Pred m
(p ||| q) s = alt (p s) (q s)
```

Assigning Priorities to `(::=)`, `(&&&)`, `(|||)`

...is done as follows:

```
infixr 4 ::=
infixr 3 &&&
infixr 2 |||
```


Combinator exists: Existential Quantificat.

...a combinator allowing the introduction of new variables in predicates (exploiting the `Int` component of answers).

Existential quantification: Introducing **local variables** in recursive predicates

```
exists :: Bunch m => (Term -> Pred m) -> Pred m
exists p (MkAnswer (s,n)) =
  p (Var (Generated n)) (MkAnswer (s,n+1))
```

Note:

- The term `exists (\x -> ...x...)` has the same meaning as the predicate `...x...` but with `x` denoting a fresh variable which is different from all the other variables used by the program; `n+1` in `MkAnswer (s,n+1)` ensures that never the same variable is introduced by nested calls of `exists`.
- The function `exists` thus takes as its argument a function, which expects a term and produces a predicate; it invents a fresh variable and applies the given function to that variable.

Named vs. Generated Variables

...illustrating the difference:

- 1)

```
run (var "x" ::= list [1,2,3]
    &&& exists (\t -> var "x" ::= Cons (var "y") t))
    :: Stream Answer
->> [{x=[1,2,3], y=1}]
```
- 2)

```
run (var "x" ::= list[1,2,3]
    &&& var "x" ::= Cons (var "y") (var "t"))
    :: Stream Answer
->> [{t=[2,3], x=[1,2,3], y=1}]
```

Note

- Example 1): The **named variable** `y` is set to the head of the list, which is the value of `x`. The value of the **generated variable** `t` is not output.
- Example 2): The same as above but now `t` denotes a **named variable**, whose value is output.

Cost Management of Recursive Predicates

...ensuring that in connection with the `bunch` type `Matrix` the costs per unfolding of the recursive predicate increase by 1 using `wrap`:

```
step :: Bunch m => Pred m -> Pred m
step p s = wrap (p s)
```

Illustrating the usage and effect of `step`:

```
run (var "x" ::= Int 0) :: Matrix Answer
->> MkMatrix [[{x=0}]]      -- Without step: Just
                             -- the result.

run (step (var "x" ::= Int 0)) :: Matrix Answer
->> MkMatrix [[], [{x=0}]] -- With step: The result
                             -- plus the notification that
                             -- there are no answers of cost 0.
```

Chapter 16.2.9

Writing Logic Programs: Two Examples

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1356 / 10

Writing Logic Programs: Two Examples

We consider two examples:

1. Concatenating lists: The predicate `append`.
2. Testing and constructing 'good' sequences: The predicate `good`.

Example 1: List Concatenation (1)

...implementing a predicate `append (a,b,c)`, where `a`, `b` denote lists and `c` the concatenation of `a` and `b`.

The implementation of predicate `append`:

```
append :: Bunch m => (Term, Term, Term) -> Pred m
append (p,q,r) =
  step (p ::= Nil &&& q ::= r
       ||| exists (\x -> exists (\a -> exists (\b ->
           p ::= Cons x a
           &&& r ::= Cons x b
           &&& append (a,q,b))))))
```

Example 1: List Concatenation (2)

...in more [detail](#):

```
append :: Bunch m => (Term, Term, Term) -> Pred m
append (p,q,r) =
  -- Case 1
  step (p ::= Nil &&& q ::= r
       |||
       -- Case 2
       exists (\x -> exists (\a -> exists (\b ->
         p ::= Cons x a &&& r ::= Cons x b &&& append (a,q,b))))))
```

Intuitively

- **Case 1:** If `p` is `Nil`, then `r` must be the same as `q`.
- **Case 2:** If `p` has the form `Cons x a`, then `r` must have the form `Cons x b`, where `b` is obtained by recursively concatenating `a` with the unchanged `q`.
- **Termination:** Is ensured since the third argument is getting smaller in each recursive call of `append`.

Example 1: List Concatenation (3)

...as common for logic programs, there is no difference between input and output variables. Hence, multiple usages of append are possible, e.g.:

a) Using append for concatenating two lists:

```
run (append (list [1,2], list [3,4], var "z"))
                                     :: Stream Answer
->> [{z=[1,2,3,4]}]
    -- An appropriate implementation of show
    -- generating the above output is assumed.
    -- More closely related to the internal structure
    -- of the value of z would be an output like:
    ->> Cons 1 (Cons 2 (Cons 3 (Cons 4 Nil)))
```


Example 1: List Concatenation (4)

Using `append` for computing the set of lists which equal a given list

b) ...when concatenated:

```
run (append (var "x", var "y", list [1,2,3]))  
                                     :: Stream Answer  
->> [{x = Nil, y = [1,2,3]},  
     {x = [1], y = [2,3]},  
     {x = [1,2], y = [3]},  
     {x = [1,2,3], y = Nil}]
```

c) ...when concatenated with another given list:

```
run (append (var "x", list [2,3], list [1,2,3]))  
                                     :: Stream Answer  
->> [{x = [1]}]
```

Example 2: 'Good' Sequences (1)

...implementing a predicate `good` allowing to

- `generate` sequences of 0s and 1s, which are considered 'good.'
- `check`, if a sequence of 0s and 1s is 'good.'

We define:

1. The sequence `[0]` is `good`.
2. If the sequences `s1` and `s2` are `good`, then also the sequence `[1] ++ s1 ++ s2`.
3. There is no other `good` sequence except of those formed in accordance to the above two rules.

Example 2: 'Good' Sequences (2)

Examples:

- ▶ 'Good' sequences

[0]

[1]++[0]++[0] = [100]

[1]++[0]++[100] = [10100]

[1]++[100]++[0] = [11000]

[1]++[100]++[10100] = [110010100]

...

- ▶ 'Bad' sequences

[1], [11], [110], [000], [010100], [1010101], ...

Example 2: 'Good' Sequences (3)

Lemma 16.2.9.1 (Properties of 'Good' Sequences)

If a sequence s is **good**, then

1. the length of s is odd
2. $s = [0]$ or there is a sequence t with $s = [1] ++ t ++ [00]$

Note: The converse implication of Lemma 16.2.9.1(2) does not hold: the sequence $[11100] = [1] ++ [11] ++ [00]$, e.g., is **bad**.

Example 2: 'Good' Sequences (4)

The implementation of predicate good:

```
good :: Bunch m => Term -> Pred m
good (s) =
  step (s ::= Cons (Int 0) Nil
        ||| exists (\t -> exists (\q -> exists (\r ->
          s ::= Cons (Int 1) t
            &&& append (q,r,t)
            &&& good (q)
            &&& good (r))))))
```

Example 2: 'Good' Sequences (5)

...in more [detail](#):

```
good :: Bunch m => Term -> Pred m
good (s) =
  step (
    -- Case 1
    s ::= Cons (Int 0) Nil
    |||
    -- Case 2
    exist (\t -> exists (\q -> exists (\r ->
      s ::= Cons (Int 1) t
      &&& append (q,r,t) &&& good (q) &&& good (r))))))
```

[Intuitively](#)

- [Case 1](#): Checks if `s` is `[0]`.
- [Case 2](#): If `s` has the form `[1]++t` for some sequence `t`, all ways are checked of splitting `t` into two sequences `q` and `r` with `q++r==t` and `q` and `r` are good sequences themselves.
- [Termination](#): Is ensured, since `t` gets smaller in every recursive call and the number of its splittings is finite.

Example 2: 'Good' Sequences (6)

Using predicate `good`.

1) Checking if a sequence is `good` using `Stream`:

```
run (good (list [1,0,1,1,0,0,1,0,0]))  
                                     :: Stream Answer
```

```
->> [{}] -- Returning the empty set as answer,  
        -- if the argument list is good.
```

```
run (good (list [1,0,1,1,0,0,1,0,1]))  
                                     :: Stream Answer
```

```
->> [] -- Returning no answer, if the argument  
        -- list is bad.
```

Note: The “empty answer” and the “no answer” correspond to the answers “yes” and “no” of a Prolog system.

Example 2: 'Good' Sequences (7)

2a) Constructing **good** sequences using **Stream**:

```
run (good (var "s")) :: Stream Answer
->> [{s=[0]},
     {s=[1,0,0]},
     {s=[1,0,1,0,0]},
     {s=[1,0,1,0,1,0,0]},
     {s=[1,0,1,0,1,0,1,0,0]}, ..
```

...some answers will not be generated, since the **depth search** induced by **Stream** is not fair. The computation is thus likely to **get stuck** at some point.

Example 2: 'Good' Sequences (8)

2b) Constructing **good** sequences using **Diag**:

```
run (good (var "s")) :: Diag Answer
->> Diag [{s=[0]},
          {s=[1,0,0]},
          {s=[1,0,1,0,0]},
          {s=[1,0,1,0,1,0,0]},
          {s=[1,1,0,0,0]},
          {s=[1,0,1,0,1,0,1,0,0]},
          {s=[1,1,0,0,1,0,0]},
          {s=[1,0,1,1,0,0,0]},
          {s=[1,1,0,0,1,0,1,0,0]}, ..
```

...eventually **all answers** will be generated, since the **diagonalization search** induced by **Diag** is fair. However, the output order can hardly be predicted due to the **interaction** of **diagonalization** and **shuffling**.

Example 2: 'Good' Sequences (9)

2c) Constructing **good** sequences using **Matrix**:

```
run (good (var "s")) :: Matrix Answer
->> MkMatrix [ [],
               [{s=[0]}], [], [], [],
               [{s=[1,0,0]}], [], [], [],
               [{s=[1,0,1,0,0]}], [],
               [{s=[1,1,0,0,0]}], [],
               [{s=[1,0,1,0,1,0,0]}], [],
               [{s=[1,0,1,1,0,0,0]}], {s=[1,1,0,0,1,0,0]}], [],
               ..
```

...using the cost-guided 'true' breadth search induced by **Matrix**, the output order of results seems more 'predictable' than for the search induced by **Diag**. Additionally, we get 'progress notifications.'

Exercise 16.2.9.2: Adding Missing Code

Note, code for

1. pretty printing terms and answers
2. making the types `Term`, `Subst`, and `Answer` instances of the type class `Show`

is missing and must be provided before using the approach.

Add the missing code.

Chapter 16.3

In Closing

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1372/19

In Closing

Current **functional logic languages** aim at balancing

- **generality** (in terms of paradigm integration).
- **efficiency** of implementations.

Functional logic programming offers

- support of specification, prototyping, and application programming within a single language.
- terse, yet clear, support for rapid development by avoiding some tedious tasks, and allowance of incremental refinements to improve efficiency.

Overall: Functional logic programming is

- an **emerging paradigm** with **appealing features**.

Chapter 16.4

References, Further Reading

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10





Chap. 11

Chap. 12



Chap. 13

1374/19




Chapter 16: Further Reading (1)

-  Hassan Ait-Kaci, Roger Nasr. *Integrating Logic and Functional Programming*. Lisp and Symbolic Computation 2(1):51-89, 1989.
-  Sergio Antoy, Michael Hanus. *Compiling Multi-Paradigm Declarative Languages into Prolog*. In Proceedings of the International Workshop on Frontiers of Combining Systems (FroCoS 2000), Springer-V., LNCS 1794, 171-185, 2000.
-  Sergio Antoy, Michael Hanus. *Functional Logic Programming*. Communications of the ACM 53(4):74-85, 2010.
-  Sergio Antoy, Michael Hanus. *New Functional Logic Design Patterns*. In Proceedings of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011), Springer-V., LNCS 6816, 19-34, 2011.




Chapter 16: Further Reading (2)

-  Bernd Braßel, Michael Hanus, Björn Peemöller, Fabian Reck. *KiCS2: A New Compiler from Curry to Haskell*. In Proceedings of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011), Springer-V., LNCS 6816, 1-18, 2011.
-  Norbert Eisinger, Tim Geisler, Sven Panne. *Logic Implemented Functionally*. In Proceedings of the 9th International Symposium on Programming Languages: Implementations, Logics, and Programs (PLILP'97), Springer-V., LNCS 1292, 351-368, 1997.




Chapter 16: Further Reading (3)

-  Michael Hanus (Ed.). *Curry: An Integrated Functional Logic Language*. Vers. 0.8.2, 2006.
www.curry-language.org/
Vers. 0.8.3, September 11, 2012:
<http://www.informatik.uni-kiel.de/~curry/report.html>⁴
-  Michael Hanus. *The Integration of Functions into Logic Programming: From Theory to Practice*. Journal of Functional Programming 19&20:583-628, 1994.
-  Michael Hanus. *Multi-paradigm Declarative Languages*. In Proceedings of the 23rd International Conference on Logic Programming (ICLP 2007), Springer-V., LNCS 4670, 45-75, 2007.

Chapter 16: Further Reading (4)

-  Michael Hanus. *Functional Logic Programming: From Theory to Curry*. In *Programming Logics – Essays in Memory of Harald Ganzinger*. Springer-V., LNCS 7797, 123-168, 2013.
-  Michael Hanus, Sergio Antoy, Bernd Braßel, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, F. Steiner. *PAKCS: The Portland Aachen Kiel Curry System*. 2013. Available at www.informatik.uni-kiel.de/~pakcs
-  Michael Hanus, Herbert Kuchen, Juan Jose Moreno-Navarro. *Curry: A Truly Functional Logic Language*. In *Proceedings of the ILPS'95 Workshop on Visions for the Future of Logic Programming*, 95-107, 1995.



Chapter 16: Further Reading (5)

-  Joxan Jaffar, Jean-Louis Lassez. *Constraint Logic Programming*. In Conference Record of the 14th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'87), 111-119, 1987.
-  John W. Lloyd. *Programming in an Integrated Functional and Logic Language*. Journal of Functional and Logic Programming 1999(3), 49 pages, MIT Press, 1999.
-  Francisco J. López-Fraguas, Jaime Sánchez-Hernández. *TOY: A Multi-paradigm Declarative System*. In Proceedings of the 10th International Conference on Rewriting Techniques and Applications (RTA'99), Springer-V., LNCS 1631, 244-247, 1999.



Chapter 16: Further Reading (6)

-  Kimbal Marriott, Peter J. Stuckey. *Programming with Constraints*. MIT Press, 1998.
-  Peter Pepper, Petra Hofstedt. *Funktionale Programmierung*. Springer-V., 2006. (Kapitel 22, Integration von Konzepten anderer Programmiersprachen)
-  Uday S. Reddy. *Narrowing as the Operational Semantics of Functional Languages*. In Proceedings of the IEEE International Symposium on Logic Programming, 138-151, 1985.
-  Tom Schrijvers, Peter J. Stuckey, Philip Wadler. *Monadic Constraint Programming*. Journal of Functional Programming 19(6):663-697, 2009.

Chapter 16: Further Reading (7)

-  Zoltan Somogyi, Fergus Henderson, Thomas Conway. *The Execution Algorithm of Mercury: An Efficient Purely Declarative Logic Programming Language*. *Journal of Logic Programming* 29(1-3):17-64, 1996.
-  Zoltan Somogyi, Fergus J. Henderson, Thomas C. Conway. *Mercury: An Efficient Purely Declarative Logic Programming Language*. In *Proceedings of the 18th Australasian Computer Science Conference*, 499-512, 1995.
-  Silvija Seres, Michael Spivey. *Embedding Prolog in Haskell*. In *Proceedings of the 1999 Haskell Workshop (Haskell'99)*, Technical Report UU-CS-1999-28, Department of Computer Science, University of Utrecht, 25-38, 1999.

Chapter 16: Further Reading (8)

-  Michael Spivey, Silvija Seres. *Combinators for Logic Programming*. In Jeremy Gibbons, Oege de Moor (Eds.), *The Fun of Programming*. Palgrave MacMillan, 177-199, 2003.
-  Philip Wadler. *How to Replace Failure with a List of Successes*. In *Proceedings of the 4th International Conference on Functional Programming and Computer Architecture (FPCA'85)*, Springer-V., LNCS 201, 113-128, 1985.

Chapter 17

Pretty Printing

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1383/10

Chapter 17.1

Motivation

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1384/19

Pretty Printing

...is about

- 'beautifully' printing values of tree-like structures as plain text.

A pretty printer is a

- tool (often a library of routines) designed for converting a tree value into plain text

such that the

- tree structure is preserved and reflected by indentation while utilizing a minimum number of lines to display the tree value.

Pretty printing can thus be considered

- dual to parsing.

Pretty Printing

...is just as **parsing** often used for demonstrating the **power** and **elegance** of **functional programming**, where not just the

- **printed result** of a **pretty printer** shall be **'pretty'**
- but also the **pretty-printer** itself including that its code is **short** and **fast**, and its operators enjoy properties which are appealing from a **mathematical point of view**.

Overall, a **'good'** **pretty printer** must properly balance:

- **Ease** of use
- **Flexibility** of layout
- **'Beauty'** of output

...while being itself **'pretty.'**

The Prettier Printer

...presented in this [chapter](#) has been proposed by [Philip Wadler](#) in:

- Philip Wadler. [A Prettier Printer](#). In Jeremy Gibbons, Oege de Moor (Eds.), [The Fun of Programming](#). Palgrave MacMillan, 2003.

which has been designed to improve (cf. [Chapter 17.5](#)) on a [pretty printer](#) proposed by [John Hughes](#) which is widely recognized as a [standard](#):

- John Hughes. [The Design of a Pretty-Printer Library](#). In Johan Jeuring, Erik Meijer (Eds.), [Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques](#). Springer-V., LNCS 925, 53-96, 1995.

Outline and Assumptions

...the implementation of the [simple pretty printer](#) and the [prettier printer](#) of [Philip Wadler](#) assumes some implementation of a type of documents [Doc](#).

The

1. [simple pretty printer](#) (cf. [Chapter 17.2](#))

- implements [Doc](#) as [strings](#).
- supports for every document only [one possible layout](#), in particular, no attempt is made to compress structure on to a single line.

2. [prettier printer](#) (cf. [Chapter 17.3](#))

- implements [Doc](#) in terms of suitable [algebraic sum data types](#).
- allows [multiple layouts](#) of a document and to pick a best one out of them for printing a document.

Chapter 17.2

The Simple Pretty Printer

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1389/10

Chapter 17.2.1

Basic Document Operators

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1390/10

The Simple Pretty Printer

...(as well as the `prettier printer` later on) relies on six basic document operators:

Associative operator for concatenating documents:

```
(<>) :: Doc -> Doc -> Doc
```

The empty document being a right and left unit for (<>):

```
nil :: Doc
```

Converting a string into a document (arguments of function `text` shall not contain newline characters):

```
text :: String -> Doc
```

The document representing a line break:

```
line :: Doc
```

Adding indentation to a document:

```
nest :: Int -> Doc -> Doc
```

Layouting a document as a string:

```
layout :: Doc -> String
```

String Documents

...choosing for the `simple pretty printer strings` for implementing `documents`, i.e.:

- type `Doc = String`

the implementation of the `basic operators` boils down to:

- `(<>)`: String `concatenation ++`.
- `nil`: The `empty` string `[]`.
- `text`: The `identity` on strings.
- `line`: The string formed by the `newline` character `'\n'`.
- `nest i: indentation`, adding `i` spaces (only used after line breaks by means of `line`).
- `layout`: The `identity` on strings.

Note

...the coupling of `line` and `nest` is an essential difference to the `pretty printer` of `John Hughes`, where insertion of spaces is also allowed in `front of strings`.

This difference is key for succeeding with only `one concatenation operator` for documents instead of the `two` in the `pretty printer` of `John Hughes` (cf. `Chapter 17.5`).

Chapter 17.2.2

Normal Forms of String Documents

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1394 / 10

String Documents

...can always be reduced to a **normal form** representation alternating applications of function

- text with **line breaks** nested to a given **indentation**:

```
text s_0 <> nest i_1 line <> text s_1 <> ...  
                                <> nest i_k line <> text s_k
```

where every

- s_j is a **string** (possibly empty).
- i_j is a **natural number** (possibly zero).

Example: Normal Form Representation

The document (i.e., a `Doc`-value):

```
text "bbbbbb" <> text "[" <>
nest 2 (
  line <> text "ccc" <> text ", " <>
  line <> text "dd"
) <>
line <> text "]" :: Doc
```

which prints as:

```
bbbbbb[
      ccc,
      dd
    ]
```

has the normal form (representation):

```
text "bbbbbb[" <>
nest 2 line <> text "ccc," <>
nest 2 line <> text "dd" <>
nest 0 line <> text "]" :: Doc
```

Normal Form Representations

...of [string documents](#) exist because of a variety of [laws](#) the basic operators of the [simple pretty printer](#) enjoy. In particular:

Lemma 17.2.2.1 (Associativity of Doc. Concatenat.)
[\(<>\)](#) is [associative](#) with unit [nil](#).

...as well as the collection of [basic operator laws](#) compiled in [Lemma 17.2.2.2](#).

Basic Operator Laws

Lemma 17.2.2.2 (Basic Operator Laws)

1. Operator `text` is a homomorphism from string to document concatenation:

```
text (s ++ t)   = text s <> text t
text ""         = nil
```

2. Opr. `nest` is a homomorph. from addition to composition:

```
nest (i+j) x    = nest i (nest j x)
nest 0 x        = x
```

3. Opr. `nest` distributes through document concatenation:

```
nest i (x <> y) = nest i x <> nest i y
nest i nil      = nil
```

4. Nesting is absorbed by `text` (differently to the pretty printer of Hughes):

```
nest i (text s) = text s
```

Note

...the laws compiled in Lemma 17.2.2.1 and 17.2.2.2

- ▶ come, except of the last one, in pairs with a corresponding law for the unit of the respective operator.
- ▶ are sufficient to ensure that every document can be transformed into normal form, where the
 - laws of part 1) and 2) are applied from left to right.
 - last of part 3) and 4) are applied from right to left.

Laws

...relating string documents with their layouts:

Lemma 17.2.2.3 (Layout Operator Laws)

1. Operator `layout` is a homomorphism from document to string concatenation:

$$\begin{aligned} \text{layout } (x \langle \rangle y) &= \text{layout } x ++ \text{layout } y \\ \text{layout } \text{nil} &= "" \end{aligned}$$

2. Operator `layout` is the inverse of function `text`:

$$\text{layout } (\text{text } s) = s$$

3. The result of `layout` applied to a nested line is a newline followed by one space for each level of indentation:

$$\text{layout } (\text{nest } i \text{ line}) = '\n' : \text{copy } i \text{ ' '}$$

Chapter 17.2.3

Printing Trees

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1401/19

Using the Simple Pretty Printer

...for [prettily printing](#) values of the data type `Tree` defined by:

```
data Tree = Node String [Tree]
```

For illustration, consider `Tree`-value `t`:

```
t = Node "aaa"  
  [Node "bbbb" [Node "ccc" [], Node "dd" []],  
   Node "eee" [],  
   Node "ffff"  
     [Node "gg" [], Node "hhh" [], Node "ii" []]]
```

Two different Layouts of *t* as Strings

```
aaa[bbbb[ccc,  
        dd],  
    eee,  
    ffff[gg,  
         hhh,  
         ii]]
```

```
aaa[  
    bbbbb[  
        ccc,  
        dd  
    ],  
    eee,  
    ffff[  
        gg,  
        hhh,  
        ii  
    ]  
]
```

where `t = Node "aaa"`

```
[Node "bbbb" [Node "ccc" [],Node "dd" []],  
 Node "eee" [],  
 Node "ffff"  
  [Node "gg" [],Node "hhh" [],Node "ii" []]]
```

The Layout Strategies

...used for **layouting** and **printing** tree **t**:

- **Left**: Tree **siblings** start on a new line, properly indented.
- **Right**: Every **subtree** starts on a new line, properly indented by two spaces.

```
aaa[bbbb[ccc,
        dd],
    eee,
    ffff[gg,
        hhh,
        ii]]
```

```
aaa[
    bbbb[
        ccc,
        dd
    ],
    eee,
    ffff[
        gg,
        hhh,
        ii
    ]
]
```

Implementing the 'Left' Layout Strategy

...by means of a utility function `showTree` converting a tree into a string document according to the 'left' layout strategy:

```
type Doc = String
data Tree = Node String [Tree]

showTree :: Tree -> Doc
showTree (Node s ts) =
  text s <> nest (length s) (showBracket ts)

showBracket :: [Tree] -> Doc
showBracket [] = nil
showBracket ts =
  text "[" <> nest 1 (showTrees ts) <> text "]"

showTrees :: [Tree] -> Doc
showTrees [t] = showTree t
showTrees (t:ts) =
  showTree t <> text "," <> line <> showTrees ts
```

Implementing the 'Right' Layout Strategy

...by means of a utility function `showTree'` converting a tree into a string document according to the 'right' layout strategy:

```
type Doc = String
data Tree = Node String [Tree]

showTree' :: Tree -> Doc
showTree' (Node s ts) = text s <> showBracket' ts

showBracket' :: [Tree] -> Doc
showBracket' [] = nil
showBracket' ts =
  text "[" <> nest 2 (line <> showTrees' ts) <> line
                                     <> text "]"

showTrees' :: [Tree] -> Doc
showTrees' [t] = showTree t
showTrees' (t:ts) =
  showTree t <> text "," <> line <> showTrees ts
```

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1406/10

Chapter 17.3

The Prettier Printer

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1407/19

Chapter 17.3.1

Algebraic Documents

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1408/10

Algebraic Documents

...for the `prettier printer` we consider a `document` a

- concatenation of `items`, where each `item` is a `text` or a `line break` indented a given amount.

`Documents` are thus implemented as an `algebraic sum data type`:

```
data Doc = Nil
         | String 'Text' Doc
         | Int 'Line' Doc
```

Note, the `data constructors` `Nil`, `Text`, and `Line` of `Doc` relate to the `basic document operators` `nil`, `text`, and `line` of the `simple pretty printer` as follows:

- (1) `Nil` $\hat{=}$ `nil`
- (2) `s 'Text' x` $\hat{=}$ `text s <> x`
- (3) `i 'Line' x` $\hat{=}$ `nest i line <> x`

Example: String vs. Algebraic Document Rep.

...the **normal form** representation of the **string document** considered in **Chapter 17.2.2**:

```
text "bbbbbb[" <>
nest 2 line <> text "ccc," <>
nest 2 line <> text "dd" <>
nest 0 line <> text "]"
```

...is represented by the algebraic **Doc**-value:

```
"bbbbbb[" 'Text' (
2 'Line' ("ccc," 'Text' (
2 'Line' ("dd," 'Text' (
0 'Line' ("]," 'Text' Nil))))))
```

Chapter 17.3.2

Implementing Document Operators on Algebraic Documents

Implementations

...of the basic document operators on algebraic documents can easily be derived from 'equations' (1) - (3) of Chapter 17.3.1:

```
nil           = Nil
text s       = s 'Text' Nil
line         = 0 'Line' Nil

Nil <> y      = y
(s 'Text' x) <> y = s 'Text' (x <> y)
(i 'Line' x) <> y = i 'Line' (x <> y)

nest i Nil   = Nil
nest i (s 'Text' x) = s 'Text' nest i x
nest i (j 'Line' x) = (i+j) 'Line' nest i x

layout Nil   = ""
layout (s 'Text' x) = s ++ layout x
layout (i 'Line' x) = '\n' : copy i ' ' ++ layout x
```

Justification

...for the derived definitions can be given using **equational reasoning**, e.g.:

Proposition 17.3.2.1

$$(s \text{ 'Text' } x) \langle \rangle y = s \text{ 'Text' } (x \langle \rangle y)$$

Proof by equational reasoning.

$$\begin{aligned} & (s \text{ 'Text' } x) \langle \rangle y \\ = & \{ \text{Definition of Text, equ. (2)} \} \\ & (\text{text } s \langle \rangle x) \langle \rangle y \\ = & \{ \text{Associativity of } \langle \rangle \} \\ & \text{text } s \langle \rangle (x \langle \rangle y) \\ = & \{ \text{Definition of Text, equ. (2)} \} \\ & s \text{ 'Text' } (x \langle \rangle y) \end{aligned}$$

□

...similarly, **correctness** of the other equations from the previous slide can be shown.

Chapter 17.3.3

Multiple Layouts of Algebraic Documents

Single vs. Multiple Layouts of Documents

...so far, a **document** d could essentially be considered **equivalent** to a

- ▶ **single string** defining a **unique single layout** for d .

Next, a **document** shall be considered **equivalent** to a

- ▶ **set of strings**, each of them defining a **layout** for d , together thus **multiple layouts**.

To achieve this, only one **new document operator** must be added:

```
group :: Doc -> Doc  
group x = flatten x <|> x
```

with **flatten** and (**<|>**) to be defined soon.

The Meaning of group

...applied to a **document** representing a **set of layouts**, **group**

- ▶ returns the set **with one new element added** representing the **layout**, in which everything is compressed on one line.

This is **achieved** by

- ▶ **replacing** each **newline** (and the corresponding indentation) with **text** consisting of a **single space**.

Note: Variants where

- ▶ each **newline** carries with it the alternate text it should be replaced with

are possible, e.g. some **newlines** might be replaced by the **empty text**, others by a **single space** (but are **not considered** here).

The relative 'Beauty' of a Layout

...depends much on the preferred maximum line width considered eligible for a layout.

Therefore, the document operator `layout` used so far is replaced by a new operator `pretty`:

```
pretty :: Int -> Doc -> String
```

which picks the 'prettiest' among a set of layouts depending on the `Int`-value of the preferred maximum line width argument.

Example

...replacing `showTree` of the 'left' layout strategy for trees of Chapter 17.2.3:

```
data Tree = Node String [Tree]

showTree :: Tree -> Doc
showTree (Node s ts) =
  text s <> nest (length s) (showBracket ts)
```

by a refined version with an additional call of `group`:

```
showTree (Node s ts) =
  group (text s <> nest (length s) (showBracket ts))
```

will ensure that

- ▶ trees are fit onto one line where possible (\leq *max* width).
- ▶ sufficiently many line breaks are inserted in order to avoid exceeding the preferred maximum line width.

Example (cont'd)

...calling, e.g., `pretty 30` will (when completely specified!) yield the output:

```
aaa[bbbb[ccc, dd],  
    eee,  
    ffff[gg, hhh, ii]]
```

Defining the new Operators (`<|>`), `flatten`

...for completing the implementation of the operators `group` and `pretty`.

`Union` operator, forming the `union` of `two sets of layouts`:

```
<|> :: Doc -> Doc -> Doc
```

`Flattening` operator, replacing each `line break` (and its associated `indentation`) by a `single space`:

```
flatten :: Doc -> Doc
```

Note: The operators `<|>` and `flatten` will not directly be exposed to the user but only via `group` and the operators `fillwords` and `fill` defined in [Chapter 17.3.6](#).

Required Invariant for ($\langle | \rangle$)

...assuming that a **document** always represents a **non-empty set of layouts**, which all **flatten** to the **same layout**, the following **invariant** for the **union** operator ($\langle | \rangle$) is required:

- ▶ **Invariant:** In ($x \langle | \rangle y$) all layouts of x and y flatten to the same layout.

...this **invariant** must be ensured when creating a union ($\langle | \rangle$).

Distribution Laws

...required for the implementations of (`<|>`) and `flatten`.

Each operator on simple documents extends pointwise through union:

Distributive Laws for (`<|>`)

1. $(x \langle | \rangle y) \langle \rangle z = (x \langle \rangle z) \langle | \rangle (y \langle \rangle z)$
2. $x \langle \rangle (y \langle | \rangle z) = (x \langle \rangle y) \langle | \rangle (x \langle \rangle z)$
3. $\text{nest } i (x \langle | \rangle y) = \text{nest } i x \langle | \rangle \text{nest } i y$

Since flattening gives the same result for each element of a set, the distribution law for `flatten` is simpler:

Distributive Law for `flatten`

$$\text{flatten } (x \langle | \rangle y) = \text{flatten } x$$

Interaction Laws

...required for the implementation of `flatten`.

Concerning the *interaction* of `flatten` with other *document operators*:

Interaction Laws for `flatten`

1. `flatten (x <> y)` = `flatten x <> flatten y`
2. `flatten nil` = `nil`
3. `flatten (text s)` = `text s`
4. `flatten line` = `text " "`
5. `flatten (nest i x)` = `flatten x`

Note, laws (4) and (5) are the most *interesting* ones:

- (4): *linebreaks* are replaced by a *single space*.
- (5): *indentations* are removed.

Recalling the Implementation

...of `group` in terms of `flatten` and `<|>`:

```
group :: Doc -> Doc  
group x = flatten x <|> x
```

Recall, too:

- Documents always represent a non-empty set of layouts whose elements all flatten to the same layout.
- `group` adds the `flattened layout` to a set of layouts.

Chapter 17.3.4

Normal Forms of Algebraic Documents

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1425/10

Normal Form Representations

...due to the laws for flattening (`flatten`) and union (`<<|>>`) every document can be reduced to a representation in normal form of the form:

$$x_1 \langle | \rangle \dots \langle | \rangle x_n$$

where every x_j is in the normal form of simple documents (cf. Chapter 17.2.2).

Picking a 'prettiest' Layout

...out of a **set of layouts** is done by means of an **ordering relation on lines** depending on the preferred **maximum line width**, and extended lexically to an ordering between **documents**.

Out of **two lines**

- ▶ which **both do not exceed** the maximum width, pick the **longer** one.
- ▶ of which **at least one exceeds** the maximum width, pick the **shorter** one.

Note: These rules require to pick sometimes a layout where some lines exceed the limit. This is an important difference to the approach of **John Hughes**, done only, however, if unavoidable.

Adapting the Algebraic Definition of Doc

...the algebraic definition of `Doc` of Chapter 17.3.1 is extended by a new data value constructor `Union` representing the union of two documents:

```
data Doc = Nil
         | String 'Text' Doc
         | Int 'Line' Doc
         | Doc 'Union' Doc    -- Union, the new
                             -- data constructor!
```

Note, these data value constructors relate to the basic document operators as follows:

- (1) `Nil` $\hat{=}$ `nil`
- (2) `s 'Text' x` $\hat{=}$ `text s <> x`
- (3) `i 'Line' x` $\hat{=}$ `nest i line <> x`
- (4) `x 'Union' y` $\hat{=}$ `x <|> y`

Required Invariants for Union

...assuming again that a **document** always represents a **non-empty set of layouts flattening all to the same layout**, two **invariants** are required for **Union**:

- ▶ **Invariant 1**: In $(x \text{ 'Union' } y)$ all layouts of x and y flatten to the same layout.
- ▶ **Invariant 2**: Every first line of a document in x is at least as long as every first line of a document in y .

...these **invariants** must be ensured when creating a **Union**.

Performance

...of pretty printing is improved by applying the distributive law for `Union` giving

```
(s 'Text' (x 'Union' y))
```

preference to the equivalent

```
((s 'Text' x) 'Union' (s 'Text' y))
```

Illustrating the Performance Impact (1)

...of [distributivity](#) considering the [document](#):

```
group(  
  group(  
    group(  
      group(text "hello" <> line <> text "a")  
      <> line <> text "b")  
    <> line <> text "c")  
  <> line <> text "d")
```

...and its possible [layouts](#):

hello a b c d	hello a b c	hello a b	hello a	hello
	d	c	b	a
		d	c	b
			d	c
				d

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1431 / 10

Illustrating the Performance Impact (2)

...printing the previous document within a maximum line width of 5, its

▶ `right-most layout` must be picked

...ideally, while the other ones are eliminated in one fell swoop.

Intuitively, this is achieved by picking a representation, which brings to the front any common string, e.g.:

```
"hello" 'Text' ((" ") 'Text' x) 'Union' (0 'Line' y))
```

for suitable documents `x` and `y`, where `"hello"` has been factored out of all the layouts in `x` and `y`, and `" "` of all the layouts in `x`.

Since `"hello"` followed by `" "` is of length 6 exceeding the limit 5, the right operand of `Union` can immediately be chosen without further examination of `x`, as desired.

Fixing the Performance Issue

...to realize this, $\langle \rangle$ and `nest` must be extended to specify how they interact with `Union`:

$$(x \text{ 'Union' } y) \langle \rangle z = (x \langle \rangle z) \text{ 'Union' } (y \langle \rangle z) \quad (1)$$

$$\text{nest } k (x \text{ 'Union' } y) = \text{nest } k x \text{ 'Union' } \text{nest } k y \quad (2)$$

while the definitions of `nil`, `text`, `line`, $\langle \rangle$, and `nest` remain unchanged.

Note, (1) and (2) follow from the distributive laws. In particular, they preserve `Invariant 2` required by `Union`.

Algebraic Definitions

...of `group` and `flatten` are then easily derived:

```
group Nil = Nil
group (i 'Line' x) = (" " 'Text' flatten x)
                  'Union' (i 'Line' x)

group (s 'Text' x) = s 'Text' group x
group (x 'Union' y) = group x 'Union' y

flatten Nil = Nil
flatten (i 'Line' x) = " " 'Text' flatten x
flatten (s 'Text' x) = s 'Text' flatten x
flatten (x 'Union' y) = flatten x
```

Justification (1)

...for the derived definitions can be given using [equational reasoning](#), e.g.:

Proposition 17.3.4.1

$$\text{group } (i \text{ 'Line' } x) =$$
$$(" \text{ " 'Text' flatten } x) \text{ 'Union' } (i \text{ 'Line' } x)$$

[Proof](#) by equational reasoning.

$$\begin{aligned} & \text{group } (i \text{ 'Line' } x) \\ = & \{ \text{Definition of Line, equ. (3)} \} \\ & \text{group } (\text{nest } i \text{ line } \langle \rangle x) \\ = & \{ \text{Definition of group} \} \\ & \text{flatten } (\text{nest } i \text{ line } \langle \rangle x) \langle | \rangle (\text{nest } i \text{ line } s \langle \rangle x) \\ = & \{ \text{Definition of flatten} \} \\ & (\text{text } " \text{ " } \langle \rangle \text{ flatten } x) \langle | \rangle (\text{nest } i \text{ line } \langle \rangle x) \\ = & \{ \text{Definition of Text, Union, Line, equ. (2), (4), (3)} \} \\ & (" \text{ " 'Text' flatten } x) \text{ 'Union' } (i \text{ 'Line' } x) \quad \square \end{aligned}$$

Justification (2)

Proposition 17.3.4.2

group (s 'Text' x) = s 'Text' group x

Proof by equational reasoning.

```
group (s 'Text' x)
= {Definition of Text, equ. (2)}
group (text s <> x)
= {Definition of group}
flatten (text s <> x) <|> (text s <> x)
= {Definition of flatten}
(text s <> flatten x) <|> (text s <> x)
= {( <> ) distributes through ( <|> )}
text s <> (flatten x <|> x)
= {Definition of group}
text s <> group x
= {Definition of Text, equ. (2)}
s 'Text' group x
```

□

Picking the 'best' Layout (1)

...among a set of layouts using functions `best` and `better`:

```
best w k Nil = Nil
best w k (i 'Line' x) = i 'Line' best w i x
best w k (s 'Text' x)
    = s 'Text' best w (k + length s) x
best w k (x 'Union' y)
    = better w k (best w k x) (best w k y)
better w k x y
    = if fits (w-k) x then x else y
```

Note:

- `best`: Converts a 'union'-afflicted document into a 'union'-free document.
- Argument `w`: Maximum line width.
- Argument `k`: Already consumed letters (including indentation) on current line.

Picking the 'best' Layout (2)

Check, if the first document line stays within the maximum line length `w`:

```
fits w x | w < 0      = False -- cannot fit
fits w Nil           = True  -- fits trivially
fits w (s 'Text' x)
  = fits (w - length s) x  -- fits if x fits into
                           -- the remaining space
                           -- after placing s
fits w (i 'Line' x) = True  -- yes, it fits
```

Last but not least, the `output routine`: Pick the best layout and `convert` it to a string:

```
pretty w x = layout (best w 0 x)
```

Chapter 17.3.5

Improving Performance

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1439/10

Intuitively

...pretty printing a document should be doable in time $\mathcal{O}(s)$, where s is the size of the document, i.e., a count of

- ▶ the number of `(<>)`, `nil`, `text`, `nest`, and `group` operations
- ▶ plus the length of all string arguments to `text`.

and in space proportional to $\mathcal{O}(w \max d)$, where

- ▶ w is the width available for printing
- ▶ d is the depth of the document, the depth of calls to `nest` or `group`.

Sources of Inefficiency

...of the `prettier printer` implementation so far:

1. `Document concatenation` might pile up to the left:

```
(...((text s_0 <> text s_1) <> ...) <> text s_n
```

...assuming each string has length one, this may require time $\mathcal{O}(n^2)$ to process (instead of $\mathcal{O}(n)$ as hoped for).

2. `Nesting of documents` adds a layer of processing to increment the indentation of the inner document:

```
nest i_o (text s_0 <> nest i_1 (text s_1 <>  
    ... <> nest i_n (text s_n)...))
```

...even if we assume document concatenation associates to the right.

...assuming again each string has length one, this may also require time $\mathcal{O}(n^2)$ to process (instead of $\mathcal{O}(n)$ as hoped for).

Performance Fixes

...for [inefficiency source 1](#)):

- ▶ Adding an explicit representation for [concatenation](#), and generalizing each operation to act on a list of concatenated documents.

...for [inefficiency source 2](#)):

- ▶ Adding an explicit representation for [nesting](#), and maintaining a current indentation that is incremented as nesting operators are processed.

Combining [both fixes](#) suggests

- ▶ generalizing each operation to work on a list of [indentation-document](#) pairs.

Implementing the Fixes

...by switching to a **new representation** for **documents** such that there is **one data constructor** for **every operator** building a **document**:

```
data DOC = NIL
        | DOC :<> DOC
        | NEST Int DOC
        | TEXT String
        | LINE
        | DOC :<|> DOC
```

Note: To avoid name clashes with the previous definitions, capital letters are used.

Implementing the Document Operators

...building a document of the new algebraic type is straightforward:

```
nil      = NIL
x <> y   = x :<> y
nest i x = NEST i x
text s   = TEXT s
line     = LINE
```

As before, also the **invariants** on the equality of flattened layouts and on the relative lengths of first lines are required:

- In $(x :<|> y)$ all layouts in x and y flatten to the same layout.
- No first line in x is shorter than any first line in y .

Implementing group and flatten

...for the [new algebraic type](#) is straightforward, too:

```
group x          = flatten x :<|> x
flatten NIL      = NIL
flatten (x :<> y) = flatten x:<> flatten y
flatten (NEST i x) = NEST i (flatten x)
flatten (TEXT s)  = TEXT s
flatten LINE      = TEXT " "
flatten (x :<|> y) = flatten x
```

...the definitions follow immediately from the equations given before.

The Representation Function `rep`

...maps a list of **indentation-document pairs** into the corresponding **document**:

```
rep z = fold (<>) nil [nest i x | (i,x) <- z]
```

Finding the 'best' Layout

...the operation `best` of Chapter 17.3.4 to find the 'best' layout of a document is generalized to act on a list of `indentation-document pairs` by combining it with the new representation function `rep`:

```
be w k z = best w k (rep z)      (hypothesis)
```

The `new definition` is directly derived from the old one:

```
best w k x                = be w k [(0,x)]
be w k []                 = Nil
be w k ((i,NIL):z)       = be w k z
be w k ((i,x :<> y) : z) = be w k ((i,x) : (i,y) : z)
be w k ((i,NEST j x) : z) = be w k ((i+j),x) : z)
be w k ((i,TEXT s) : z)  = s 'Text' be w (k,+length s) z
be w k ((i,LINE) : z)    = i 'Line' be w i z
be w k ((i.x :<|> y) : z) =
  better w k (be w k ((i.x) : z)) (be w k (i,y) : z))
```

Correctness

...of the equations of the previous slide can be shown by [equational reasoning](#), e.g.:

Proposition 17.3.5.1

$$\text{best } w \text{ k } x = \text{be } w \text{ k } [(0,x)]$$

[Proof](#) by equational reasoning.

$$\begin{aligned} & \text{best } w \text{ k } x \\ = & \{0 \text{ is unit for nest}\} \\ & \text{best } w \text{ k } (\text{nest } 0 \text{ } x) \\ = & \{\text{nil is unit for } \langle \rangle\} \\ & \text{best } w \text{ k } (\text{nest } 0 \text{ } x \langle \rangle \text{ nil}) \\ = & \{\text{Definition of rep, hypothesis}\} \\ & \text{be } w \text{ k } [(0,x)] \end{aligned}$$

□

Last but not least

...while the argument to `best` is represented using

- ▶ `DOC`

its result is represented using the formerly introduced type

- ▶ `Doc`

Hence, `pretty` can be defined as in [Chapter 17.3.4](#):

```
pretty w x = layout (best w 0 x)
```

The functions `layout`, `better`, and `fits`, finally, remain unchanged.

Chapter 17.3.6

Utility Functions

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1450/19

Utility Functions (1)

...for **recurringly** occurring **tasks**, e.g.:

- ▶ **Separating** two documents by inserting a **space**:

```
x <+> y           = x <> text " " <> y
```

- ▶ **Separating** two documents by inserting a **line break**:

```
x </> y           = x <> line <> y
```

- ▶ **Folding** a document:

```
folddoc f []      = nil
```

```
folddoc f [x]     = x
```

```
folddoc f (x:xs) = f x (folddoc f xs)
```

- ▶ **Advanced** document **folding**:

```
spread           = folddoc (<+>)
```

```
stack            = folddoc (</>)
```

Utility Functions (2)

...as [abbreviations](#) of frequently occurring tasks, e.g.:

- ▶ An opening bracket, followed by an indented portion, followed by a closing bracket, abbreviated by [bracket](#):

```
bracket l x r = group (text l <>
                        nest 2 (line <> x) <>
                        line <> text r)
```

- ▶ The 'right' layout strategy for trees of [Chapter 17.2.3](#), abbreviated by [showBracket'](#):

```
showBracket' ts = bracket "[" (showTrees' ts) "]"
```

- ▶ Taking a string, returning a document, where every line is filled with as many words as will fit (note: [words](#) is from the [Haskell Standard Library](#)), abbreviated by [fillwords](#):

```
x <+> y = x <> (text " " :<|> line) <> y
fillwords = folddoc (<+>) . map text . words
```

Utility Functions (3)

...abbreviations (cont'd):

- ▶ A variant of `fillwords` collapsing a list of documents to a single document by putting a space between two documents when this leads to a reasonable layout, and a newline otherwise, abbreviated by `fill`:

```
fill []          = nil
fill [x]         = x
fill (x:y:zs) =
  (flatten x <+> fill (flatten y : zs)) :<|>
  (x </> fill (y : zs))
```

Note: `fill` is copied from `pretty printer library` of [Simon Peyton Jones](#), which extends the one of [John Hughes](#).

Chapter 17.3.7

Printing XML-like Documents

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1454 / 10

Printing XML Documents

...enjoying a simplified XML syntax with `elements`, `attributes`, and `text` defined by:

```
data XML = Elt String [Att] [XML]
          | Txt String

data Att = Att String String
```

Utility Functions (1)

...for [printing XML documents](#):

- ▶ Showing [documents](#):

```
showXML x = folddoc (<>) (showXMLs x)
```

- ▶ Showing [elements](#):

```
showXMLs (Elt n a []) =  
  [text "<" <> showTag n a <> text ">"]  
showXMLs (Elt n a c) =  
  [text "<" <> showTag n a <> text ">" <>  
    showFill showXMLs c <>  
    text "</" <> text n <> text ">"]
```

- ▶ Showing [text](#):

```
showXMLs (Txt s) = map text (words s)
```

- ▶ Showing [attributes](#):

```
showAtts (Att n v) =  
  [text n <> text "=" <> text (quoted v)]
```


Utility Functions (2)

...for [printing XML documents](#) (cont'd):

- ▶ Adding [quotes](#):

```
quoted s = "\"" ++ s ++ "\""
```

- ▶ Showing [tags](#):

```
showTag n a = text n <> showFill showAtts a
```

- ▶ Filling [lines](#):

```
showFill f [] = nil
```

```
showFill f xs =
```

```
  bracket "" (fill (concat (map f xs))) ""
```

Example: 1st Layout of an XML Document

...for a **maximum** line width of **30** characters:

```
<p
  color="red" font="Times"
  size="10"
>
  Here is some
  <em> emphasized </em> text.
  Here is a
  <a
    href="http://www.eg.com/"
  > link </a>
  elsewhere.
</p>
```

Example: 2nd Layout of an XML Document

...for a **maximum** line width of **60** characters:

```
<p color="red" font="Times" size="10" >  
  Here is some <em> emphasized </em> text. Here is a  
  <a href="http://www.eg.com/" > link </a> elsewhere.  
</p>
```

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1459/10

Example: 3rd Layout of an XML Document

...dropping the two occurrences of `flatten` in `fill` (cf. [Chapter 17.3.6](#)) leads to the following output:

```
<p color="red" font="Times" size="10" >  
  Here is some <em>  
    emphasized  
  </em> text. Here is a <a  
    href="http://www.eg.com/"  
  > link </a> elsewhere.  
</p>
```

...in the above layout `start` and `close tags` of the emphasis and anchor elements are crammed together with other text, rather than getting lines to themselves; it thus looks less 'beautiful.'

Chapter 17.4

The Prettier Printer Code Library

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

Chap. 14

A Summary

...of the [code](#) of the

- performance-improved fully-fledged [prettier printer](#).
- [tree](#) example.
- [XML-documents](#) example.

according to:

- Philip Wadler. [A Prettier Printer](#). In Jeremy Gibbons, Oege de Moor (Eds.), [The Fun of Programming](#). Palgrave MacMillan, 2003.

Chapter 17.4.1

The Prettier Printer

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1463 / 10

The Prettier Printer (1)

Defining operator priorities

```
infixr 5:<|>  
infixr 6:<>  
infixr 6 <>
```

Defining algebraic document types

```
data DOC = NIL  
        | DOC :<> DOC  
        | NEST Int DOC  
        | TEXT String  
        | LINE  
        | DOC :<|> DOC  
  
data Doc = Nil  
        | String 'Text' Doc  
        | Int 'Line' Doc
```


The Prettier Printer (2)

Defining basic operators algebraically

```
nil      = NIL
x <> y   = x :<> y
nest i x = NEST i x
text s   = TEXT s
line     = LINE
```

Layouting normal form documents

```
layout Nil          = ""
layout (s 'Text' x) = s ++ layout x
layout (i 'Line' x) = '\n': copy i ' ' ++ layout x
copy i x            = [x | _ <- [1..i]]
```

The Prettier Printer (3)

Generating multiple layouts

```
group x = flatten x :<|> x
```

Flattening layouts

```
flatten NIL           = NIL
flatten (x :<> y)     = flatten x:<> flatten y
flatten (NEST i x)    = NEST i (flatten x)
flatten (TEXT s)      = TEXT s
flatten LINE          = TEXT " "
flatten (x :<|> y)    = flatten x
```

The Prettier Printer (4)

Ordering and comparing layouts

```
best w k x = be w k [(0,x)]
be w k [] = Nil
be w k ((i,NIL):z) = be w k z
be w k ((i,x :<> y) : z) = be w k ((i,x) : (i,y): z)
be w k ((i,NEST j x) : z) = be w k ((i+j),x) : z)
be w k ((i,TEXT s) : z) = s 'Text' be w (k+length s) z
be w k ((i,LINE) : z) = i 'Line' be w i z
be w k ((i.x :<|> y) : z) =
  better w k (be w k ((i.x) : z)) (be w k (i,y) : z))
better w k x y = if fits (w-k) x then x else y
fits w x | w<0      = False
fits w Nil          = True
fits w (s 'Text' x) = fits (w - length s) x
fits w (i 'Line' x) = True
```

The Prettier Printer (5)

Printing documents prettily

```
pretty w x = layout (best w 0 x)
```

Defining utility functions

```
x <+> y           = x <> text " " <> y
```

```
x </> y           = x <> line <> y
```

```
x <+/> y          = x <> (text " " :<|> line) <> y
```

```
folddoc f []      = nil
```

```
folddoc f [x]     = x
```

```
folddoc f (x:xs) = f x (folddoc f xs)
```

```
spread           = folddoc (<+>)
```

```
stack            = folddoc (</>)
```

```
bracket l x r    =
```

```
  group (text l <> nest 2 (line <> x) <>
```

```
    line <> text r)
```

The Prettier Printer (6)

Defining utility functions (cont'd)

```
fillwords      = folddoc (<+/>) . map text . words
fill []        = nil
fill [x]       = x
fill (x:y:zs) =
  (flatten x <+> fill (flatten y : zs))
                :<|> (x </> fill (y : zs))
```

Chapter 17.4.2

The Tree Example

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1470/19

The Tree Example (1)

Defining trees

```
data Tree = Node String [Tree]
```

Defining utility functions

```
showTree (Node s ts) =  
  group (text s <> nest (length s) (showBracket ts))  
showBracket [] = nil  
showBracket ts =  
  text "[" <> nest 1 (showTrees ts) <> text "]"  
showTrees [t]      = showTree t  
showTrees (t:ts) =  
  showTree t <> text "," <> line <> showTrees ts
```

The Tree Example (2)

Defining utility functions (cont'd)

```
showTree' (Node s ts) = text s <> showBracket' ts
showBracket' [] = nil
showBracket' ts = bracket "[" (showTrees' ts) "]"
showTrees' [t] = showTree t
showTrees' (t:ts) =
  showTree t <> text "," <> line <> showTrees ts
```


The Tree Example (3)

Defining a tree value for illustration

```
tree = Node "aaa" [ Node "bbbb" [ Node "ccc" [],  
                                  Node "dd" []  
                                ],  
                  Node "eee" [],  
                  Node "ffff" [ Node "gg" [],  
                                 Node "hhh" [],  
                                 Node "ii" []  
                              ]  
                          ]
```

Defining two testing environments

```
testtree w = putStr(pretty w (showTree tree))  
testtree' w = putStr(pretty w (showTree' tree))
```

Chapter 17.4.3

The XML Example

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1474/19

The XML Example (1)

Defining the XML-like document format

```
data XML = Elt String [Att] [XML]
         | Txt String

data Att = Att String String
```

Defining utility functions

```
showXML x = folddoc (<>) (showXMLs x)

showXMLs (Elt n a []) =
  [text "<" <> showTag n a <> text ">"]
showXMLs (Elt n a c) =
  [text "<" <> showTag n a <> text ">" <>
   showFill showXMLs c <>
   text "</" <> text n <> text ">"]
showXMLs (Txt s) = map text (words s)
```

The XML Example (2)

Defining utility functions (cont'd)

```
showAtts (Att n v) =  
  [text n <> text "=" <> text (quoted v)]  
quoted s = "\"" ++ s ++ "\""  
showTag n a = text n <> showFill showAtts a  
showFill f [] = nil  
showFill f xs =  
  bracket "" (fill (concat (map f xs))) ""
```

The XML Example (3)

Defining an XML-document value for illustration

```
xml =  
  Elt "p" [Att "color" "red",  
           Att "font" "Times",  
           Att "size" "10"  
        ] [Txt "Here is some",  
           Elt "em" [] [Txt "emphasized"],  
           Txt "text.",  
           Txt "Here is a",  
           Elt "a" [Att "href" "http://www.eg.com/"]  
                 [Txt "link" ],  
           Txt "elsewhere."  
        ]
```

Defining a testing environment

```
testXML w = putStr (pretty w (showXML xml))
```

Chapter 17.5

Summary

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1478/10

Summary

...the [pretty printer library](#) proposed by [John Hughes](#) is widely recognized as a [standard](#):

- John Hughes. [The Design of a Pretty-Printer Library](#). In Johan Jeuring, Erik Meijer (Eds.), [Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques](#). Springer-V., LNCS 925, 53-96, 1995.

...a variant of it is implemented in the [Glasgow Haskell Compiler](#):

- Simon Peyton Jones. [Haskell pretty-printer library](#). 1997. www.haskell.org/libraries/#prettyprinting

Why 'prettier' than 'pretty'?

...the **pretty printer** of John Hughes

- ▶ uses **two operators** for the **horizontal** and **vertical** concatenation of documents
 - one without a unit (**vertical**)
 - one with a right-unit but no left-unit (**horizontal**).

...the **prettier printer** of Philip Wadler can be considered an improvement of the **pretty printer** of John Hughes because it

- ▶ uses only **one operator** for document **concatenation** which
 - is **associative**.
 - has a **left-unit** and a **right-unit**.
- ▶ consists of about **30% less code**.
- ▶ is about **30% faster**.

In Closing

...a hint to an early work on an [imperative pretty printer](#) by:

- Derek Oppen. [Pretty-printing](#). ACM Transactions on Programming Languages and Systems 2(4):465-483, 1980.

and a [functional](#) realization of it by:

- Olaf Chitil. [Pretty Printing with Lazy Dequeues](#). In Proceedings of the ACM SIGPLAN Haskell Workshop (Haskell 2001), Universiteit Utrecht UU-CS-2001-23, 183-201, 2001.

Chapter 17.6

References, Further Reading

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10


Chap. 11

Chap. 12




Chap. 13

1482/10



Chapter 17: Further Reading (1)

-  Manuel M.T. Chakravarty, Gabriele Keller. *Einführung in die Programmierung mit Haskell*. Pearson Studium, 2004. (Kapitel 13.1.2, Ausdrücke formatieren; Kapitel 13.2.1, Formatieren und Auswerten in erweiterter Version)
-  Olaf Chitil. *Pretty Printing with Lazy Dequeues*. In Proceedings of the ACM SIGPLAN 2001 Haskell Workshop (Haskell 2001), Universiteit Utrecht UU-CS-2001-23, 183-201, 2001.
-  John Hughes. *The Design of a Pretty-Printer Library*. In Johan Jeuring, Erik Meijer (Eds.), *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques*. Springer-V., LNCS 925, 53-96, 1995.

Chapter 17: Further Reading (2)

-  Derek Oppen. *Pretty-printing*. ACM Transactions on Programming Languages and Systems 2(4):465-483, 1980.
-  Tillmann Rendel, Klaus Ostermann. *Invertible Syntax Descriptions: Unifying Parsing and Pretty Printing*. In Proceedings of the 3rd ACM Haskell Symposium on Haskell (Haskell 2010), 1-12, 2010.
-  Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Chapter 5, Writing a Library: Working with JSON Data – Pretty Printing a String, Fleshing Out the Pretty-Printing Library)

Chapter 17: Further Reading (3)

-  Simon Peyton Jones. *Haskell pretty-printer library*. 1997.
www.haskell.org/libraries/#prettyprinting
-  Philip Wadler. *A Prettier Printer*. In Jeremy Gibbons, Oege de Moor (Eds.), *The Fun of Programming*. Palgrave MacMillan, 223-243, 2003.

Chapter 18

Functional Reactive Programming

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

Chap. 14

Chapter 18.1

Motivation

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1487/19

Hybrid Systems

...are **systems** composed of

- ▶ continuous
- ▶ discrete

components.

Mobile Robots

...are special **hybrid systems** (or **cyber-physical systems**) from both a **physical** and **logical** perspective:

► Physically

- **Continuous** components: Voltage-controlled motors, batteries, range finders,...
- **Discrete** components: Microprocessors, bumper switches, digital communication,...

► Logically

- **Continuous** notions: Wheel speed, orientation, distance from a wall,...
- **Discrete** notions: Running into another object, receiving a message, achieving a goal,...

In this chapter

...designing and implementing two

- ▶ imperative-style languages for controlling robots

Beyond the concrete application, this provides two examples of

- ▶ domain specific language (DSL)

and an application of the type constructor classes

- ▶ Monad
- ▶ Arrow
- ▶ Functor

Note, the languages aim at **simulating** robots in order to allow running simulations at home without having to buy (possibly expensive) robots first.

Reading

...for [Chapter 18.2](#) (using [monads](#)):

- Paul Hudak. [The Haskell School of Expression – Learning Functional Programming through Multimedia](#). Cambridge University Press, 2000. (Chapter 19, An Imperative Robot Language)

...for [Chapter 18.3](#) (using [arrows](#)):

- Paul Hudak, Antony Courtney, Herik Nilsson, John Peterson. [Arrows, Robots, and Functional Reactive Programming](#). Summer School on Advanced Functional Programming 2002, Springer-V., LNCS 2638, 159-187, 2003.

Note: [Chapter 18.2](#) and [18.3](#) are independent and do not build upon each other.

Chapter 18.2

An Imperative Robot Language

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1492/10

Chapter 18.2.1

The Robot's World

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

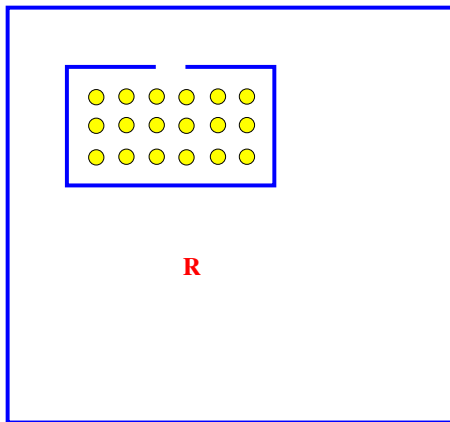
Chap. 12

Chap. 13

1493/10

The Robot's World

...a two-dimensional grid surrounded by walls, with rooms having doors, and gold coins as treasures!



In more detail

...the robot's world is

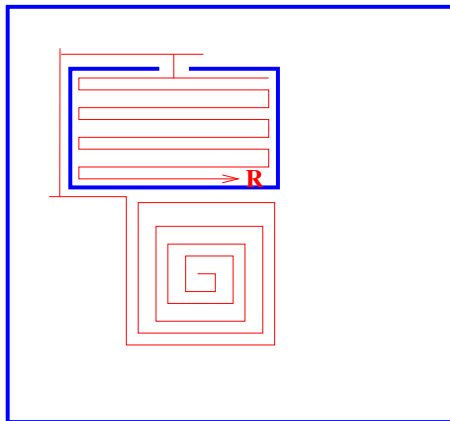
- ▶ a finite two-dimensional grid of square form
 - equipped with walls
 - possibly forming rooms, possibly having doors
 - with gold coins placed on some grid points

The preceding example shows

- ▶ a robot's world with one room, an open door, full of gold: Eldorado!
- ▶ a robot sitting in the centre of the world ready for exploring it!

The Robot's Mission

...exploring the world, collecting treasures, leaving footprints!



In more detail

...the robot's mission is

- ▶ to explore its world, to collect the treasures in it, and to leave footprints of its exploration, i.e., to
 - strolling and searching through its world, e.g., following the path way of an outward-oriented spiral.
 - picking up the gold coins it finds on its way and saving them in its pocket.
 - dropping gold coins at some (other) grid points.
 - marking its way with differently colored pens.

Objective

...developing an imperative-like robot language allowing to write programs, which advise a robot how to explore and shape its world!

E.g., programs such as:

```
(1) drawSquare =  
    do penDown  
    move  
    turnRight  
    move  
    turnRight  
    move  
    turnRight  
    move
```

```
(2) moveToWall =  
    while (isnt blocked)  
    do move
```

```
(3) getCoinsToWall =  
    while (isnt blocked) $  
    do move  
    checkAndPickCoin
```

In more detail

...assuming that `Robot` is a `Monad`:

```
newtype Robot a = Rob...
```

```
instance Monad Robot where...
```

```
drawSquare =
```

```
do penDown      (penDown :: Robot () / pen ready to write)
   move         (move :: Robot () / moving one space for-
   turnRight    (turnRight :: Robot () / turn 90 degrees
   move         (move :: Robot () / moving one space for-
   turnRight    (turnRight :: Robot () / turn 90 degrees
   move         (move :: Robot () / moving one space for-
   turnRight    (turnRight :: Robot () / turn 90 degrees
   move         (move :: Robot () / moving one space for-
```

Note, for the `robot monad`, operation `(>>)` is relevant!

The Implementation Environment

...required **modules**:

```
module Robot where
  import Array
  import List
  import Monad
  import SOEGraphics
  import Win32Misc (timeGetTime)
  import qualified GraphicsWindows as GW (getEvent)
```

Note:

- **Graphics**, **SOEGraphics** are two commonly used graphics libraries being **Windows** compatible.
- Double-check the **SOE homepage** at haskell.org/soe regarding the availability of the modules **SOEGraphics** and **GraphicsWindows**.

Chapter 18.2.2

Modelling the Robot's World

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1501/19

Modelling the World

...the robots live and act in a 2-dimensional grid.

Positions are given by their `x` and `y` coordinates:

```
type Position = (Int,Int)
```

Directions a robot can face or head to:

```
data Direction = North | East | South | West
                deriving (Eq, Show, Enum)
```

World, a two-dimensional grid as `Array`-type:

```
type Grid = Array Position [Direction]
```

Chapter 18.2.3

Modelling Robots

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1503/10

Modelling Robots

...by their **internal states**, which are **characterized** by **6 values**:

1. Robot position
2. Robot orientation
3. Pen status (up or down)
4. Pen color
5. Treasure map
6. Number of coins in the robot's pocket

Note, the **grid** does not change and is thus not part of a **robot (state)**.

Modelling Internal Robot States

...as an algebraic product type:

```
data RobotState = RState { position :: Position
                          , facing :: Direction
                          , pen :: Bool
                          , color :: Color
                          , treasure :: [Position]
                          , pocket :: Int
                          } deriving Show
```

where the number of coins at a position is given by the number of its occurrences in `treasure`, and `Color` defines the set of possible pen colors:

```
data Color = Black | Blue | Green | Cyan
           | Red | Magenta | Yellow | White
           deriving (Eq, Ord, Bounded, Enum,
                    Ix, Show, Read)
```

Note

...the definition of `RobotState` takes advantage of Haskell's `field-label` (or `record`) syntax: The `field labels` (`position`, `facing`, `pen`, `color`, `treasure`, `pocket`) offer

- access to state components by names instead of position without requiring `specific selector functions`.

This advantage would have been lost defining robot states equivalently but without `field-label` syntax as in:

```
data RobotState = RState
    Position
    Direction
    Bool
    Color
    [Position]
    Int deriving Show
```

Illustrating Field-label Syntax Usage (1)

...generating, modifying, and accessing values of robot-state components.

Example 1: Generating field values

The definition

```
s1 = RState { position = (0,0)
             , facing   = East
             , pen      = True
             , color    = Green
             , treasure = [(2,3), (7,9), (12,42)]
             , pocket   = 2
             } :: RobotState
```

is *equivalent* to:

```
s2 = RState (0,0) East True Green
          [(2,3), (7,9), (12,42)] 2 :: RobotState
```

Illustrating Field-label Syntax Usage (2)

Example 2: Modifying field values

```
s3 = s2 { position = (22,43), pen = False }  
->> RState { position = (22,43)  
           , facing = East  
           , pen = False  
           , color = Green  
           , treasure = [(2,3), (7,9), (12,42)]  
           , pocket = 2  
           } :: RobotState
```

Example 3: Accessing field values

```
position s1 ->> (0,0)  
treasure s3 ->> [(2,3), (7,9), (12,42)]  
color     s3 ->> Green
```

Example 4: Using field names in patterns

```
jump (RState { position = (x,y) }) = (x+2,y+1)
```

Benefits and Advantages

...of using `field-label` syntax:

- It is more 'informative' (due to `field` names).
- The order of `fields` gets irrelevant, e.g., the definition of:

```
s4 = RState { position = (0,0)
             , pocket   = 2
             , pen      = True
             , color    = Green
             , treasure = [(2,3), (7,9), (12,42)]
             , facing   = East
           } :: RobotState
```

is equivalent to the robot state defined by `s1`.

Chapter 18.2.4

Modelling Robot Commands as State Monad

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1510/19

Modelling Robot Commands

...by `Robot`, a 1-ary type constructor, defined by:

```
newtype Robot a =  
  Rob (RobotState -> Grid -> Window  
      -> IO (RobotState,a))
```

allows making `Robot` an instance of type class `Monad` (matching the pattern of a `state monad` by conceptually considering the `Grid` argument part of the state):

```
instance Monad Robot where  
  Rob sf0 >>= f = Rob $ \s0 g w ->  
    do (s1,a1) <- sf0 s0 g w  
       let Rob sf1 = f a1  
           (s2,a2) <- sf1 s1 g w  
       return (s2,a2)  
  return a      = Rob (\s _ _ -> return (s,a))
```

Note

- \$ can be replaced by parentheses:

```
instance Monad Robot where
```

```
  Rob sf0 >>= f = Rob (\s0 g w ->
                        do (s1,a1) <- sf0 s0 g w
                           let Rob sf1 = f a1
                               (s2,a2) <- sf1 s1 g w
                           return (s2,a2))
  return a      = Rob (\s _ _ -> return (s,a))
```

- the Grid argument in

```
newtype Robot a =
```

```
  Rob (RobotState -> Grid -> Window
       -> IO (RobotState,a))
```

can conceptually be considered a 'read-only' part of a robot state; the Window argument allows specifying the window, in which the graphics is displayed.

Chapter 18.2.5

The Imperative Robot Language

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1513/10

IRL: The Imperative Robot Language

Key insight:

- ▶ Taking state as input
- ▶ Possibly querying the state in some way
- ▶ Returning a possibly modified state

...reveals the **imperative nature** of IRL commands.

Utility Functions

...not intended (except of `at`) for direct usage by an IRL programmer.

► **Direction commands:**

```
right, left :: Direction -> Direction
right d = toEnum (succ (mod (fromEnum d) 4))
left d   = toEnum (pred (mod (fromEnum d) 4))

at :: Grid -> Position -> [Direction]
at = (!)
```

► **Supporting functions for updating and querying states:**

```
updateState :: (RobotState -> RobotState)
              -> Robot ()
updateState u = Rob (\s _ _ -> return (u s, ()))

queryState :: (RobotState -> a) -> Robot a
queryState q = Rob (\s _ _ -> return (s, q s))
```

Recalling the Definition of Type Class Enum

...of the [Standard Prelude](#):

```
class Enum a where
  succ, pred      :: a -> a
  toEnum         :: Int -> a
  fromEnum       :: a -> Int
  enumFrom       :: a -> [a]           -- [n..]
  enumFromThen   :: a -> a -> [a]     -- [n,n'..]
  enumFromTo     :: a -> a -> [a]     -- [n..m]
  enumFromThenTo :: a -> a -> a -> [a] -- [n,n'..m]

  succ           = toEnum . (+1) . fromEnum
  pred          = toEnum . (subtract 1) . fromEnum
  enumFrom x     = map toEnum [fromEnum x..]
  enumFromThen x y = map toEnum [fromEnum x, fromEnum y..]
  enumFromTo x y = map toEnum [fromEnum x..fromEnum y]
  enumFromThenTo x y z = map toEnum [fromEnum x,
                                     fromEnum y..fromEnum z]

  toEnum, fromEnum = ...implementation is type-dependent
```

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1516/10

Recalling the Usage of Type Class Enum

The following 'equalities' hold:

```
enumFrom n           ≐ [n..]
enumFromThen n n'    ≐ [n,n'..]
enumFromTo n m       ≐ [n..m]
enumFromThenTo n n' m ≐ [n,n'..m]
```

Example:

```
data Color = Red | Orange | Yellow | Green
           | Blue | Indigo | Violet deriving Enum

[Red..Green]    ->> [Red, Orange, Yellow, Green]
[Red, Yellow..] ->> [Red, Yellow, Blue, Violet]
fromEnum Blue   ->> 4
toEnum 3        ->> Green
```

IRL Commands for Robot Orientation

...by updating the internal robot state.

▶ Turn right:

```
turnLeft :: Robot ()
```

```
turnLeft =
```

```
  updateState (\s -> s {facing = left (facing s)})
```

▶ Turn left:

```
turnRight :: Robot ()
```

```
turnRight =
```

```
  updateState (\s -> s {facing = right (facing s)})
```

▶ Turn to:

```
turnTo :: Direction -> Robot ()
```

```
turnTo d = updateState (\s -> s {facing = d})
```

▶ Facing what direction?

```
direction :: Robot Direction
```

```
direction = queryState facing
```

IRL Command for Blockade Checking

- ▶ Motion blocked in direction currently facing?

```
blocked :: Robot Bool
```

```
blocked =
```

```
  Rob $ \s g _ ->
```

```
    return (s, facing s 'notElem' (g 'at' position s))
```

with `notElem` from the [Standard Prelude](#).

IRL Commands for Motion

- ▶ Moving forward one space if not blocked:

```
move :: Robot ()
move =
  cond1 (isnt blocked)
    (Rob $ \s _ w -> do
      let newPos = movePos (position s) (facing s)
      graphicsMove w s newPos
      return (s {position = newPos}, ()))
    )
```

- ▶ Moving forward one space in direction of:

```
movePos :: Position -> Direction -> Position
movePos (x,y) d = case d of North -> (x,y+1)
                        South -> (x,y-1)
                        East  -> (x+1,y)
                        West  -> (x-1,y)
```


IRL Commands for Pen Usage

- ▶ Choose pen color for writing:

```
setPenColor :: Color -> Robot ()  
setPenColor c = updateState (\s -> s {color = c})
```

- ▶ Pen down to start writing:

```
penDown :: Robot ()  
penDown = updateState (\s -> s {pen = True})
```

- ▶ Pen up to stop writing:

```
penUp :: Robot ()  
penUp = updateState (\s -> s {pen = False})
```

IRL Commands for Coin Handling (1)

- ▶ At position with coin according to treasure map?

```
onCoin :: Robot Bool
onCoin = queryState (\s ->
                    position s 'elem' treasure s)
```

- ▶ Pick coin:

```
pickCoin :: Robot ()
pickCoin =
  cond1 onCoin
  (Robot $ \s _ w ->
    do eraseCoin w (position s)
       return (s {treasure =
                  position s 'delete' treasure s,
                  pocket = pocket s+1}, ()))
)
```

IRL Commands for Coin Handling (2)

- ▶ How many coins currently in pocket?

```
coins :: Robot Int
coins = queryState pocket
```

- ▶ Drop coin, if there is at least one in the pocket:

```
dropCoin :: Robot ()
dropCoin =
  cond1 (coins >* return 0)
  (Robot $ \s _ w ->
    do drawCoin w (position s)
      return (s {treasure =
                  position s : treasure s,
                  pocket = pocket s-1}, ()))
  )
```

Utility Functions for Logic and Control (1)

- ▶ Conditionally performing commands:

```
cond :: Robot Bool -> Robot a
      -> Robot a -> Robot a
cond p c a = do pred <- p
              if pred then c else a
cond1 p c = cond p c (return ())
```

- ▶ Performing commands while some condition is met:

```
while :: Robot Bool -> Robot () -> Robot ()
while p b = cond1 p (b >> while p b)
```

- ▶ Connecting commands 'disjunctively:'

```
(||*) :: Robot Bool -> Robot Bool -> Robot Bool
b1 ||* b2 = do p <- b1
              if p then return True
              else b2
```

Utility Functions for Logic and Control (2)

- ▶ Connecting commands 'conjunctively:'

```
(&&*) :: Robot Bool -> Robot Bool -> Robot Bool
b1 &&* b2 = do p <- b1
            if p then b2
            else return False
```

- ▶ Lifting negation to commands:

```
isnt :: Robot Bool -> Robot Bool
isnt = liftM not
```

- ▶ Lifting comparisons to commands:

```
(>*) :: Robot Int -> Robot Int -> Robot Bool
(>*) = liftM2 (>)

(<*) :: Robot Int -> Robot Int -> Robot Bool
(<*) = liftM2 (<)
```

Recalling the Definitions of the Lift Operators

...the higher-order lift operations `liftM` and `liftM2` are defined in the library `Monad` (as well as `liftM3`, `liftM4`, and `liftM5`):

```
liftM :: (Monad m) => (a -> b) -> (m a -> m b)
liftM f = \a -> do a' <- a
           return (f a')
```

```
liftM2 :: (Monad m) => (a -> b -> c)
           -> (m a -> m b -> m c)
liftM2 f = \a b -> do a' <- a
                     b' <- b
                     return (f a' b')
```

Note

The implementations of

- `isnt`, `(>*)`, and `(<*)` are based on `liftM` and `liftM2`, thereby avoiding the usage of special `lift` functions.
- `(||*)` and `(&&*)` are not based on `liftM2`, thereby avoiding (unnecessary) strictness in their second arguments.

Illustrating the Usage of `cond` and `cond1`

...moving the robot one space forward if it is not blocked; moving it one space to the right if it is.

An [implementation](#) using

▶ `cond`:

```
evade :: Robot ()
evade = cond blocked
      (do turnRight
         move)
      move
```

▶ `cond1`:

```
evade' :: Robot ()
evade' = do cond1 blocked turnRight
           move
```


Moving in a Spiral

...an [example](#) of an [advanced IRL program](#):

```
spiral :: Robot ()
spiral = penDown >> loop 1
  where loop n =
      let twice = do turnRight
                    moven n
                    turnRight
                    moven n
      in con blocked
         (twice >> turnRight >> moven n)
         (twice >> loop (n+1))

moven :: Int -> Robot ()
moven n = mapM . (const move) [1..]
```

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1529/10

Chapter 18.2.6

Defining a Robot's World

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1530/19

The Robot's World: Preliminary Definitions

The **robots' world** is a grid of type **Array**:

```
type Grid = Array Position [Direction]
```

Grid points can be **accesssed** using:

```
at :: Grid -> Position -> [Direction]  
at = (!)
```

Defining the Initial World g_0 (1)

The `size` of the initial grid world g_0 is given by:

```
size :: Int
size = 20
```

with the grid world's

- ▶ centre at: $(0,0)$
- ▶ corners at: $(-size, size)$ $(size, size)$
 $((-size), (-size))$ $(size, (-size))$

Defining the Initial World g0 (2)

..inner, border, and corner points of world g0 are characterized by the directions of motion they allow:

- ▶ Inner points of g0 allow moving toward:
interior = [North, South, East, West]
- ▶ Border points at the north, east, south, and west border allow moving toward:
nb = [South, East, West] (nb: north border)
eb = [North, South, West]
sb = [North, East, West]
wb = [North, South, East] (wb: west border)
- ▶ Corner points at the northwest, northeast, southeast, and southwest corner allow moving toward:
nwc = [South, East] (nwc: northwest corner)
nec = [South, West]
sec = [North, West]
swc = [North, East] (swc: southwest corner)

Defining the Initial World g0 (3)

...all **grid points**, i.e., **inner** and **border grid points** can thus be **enumerated** using **list comprehension**, which allows to define the **initial world grid g0** as follows:

```
g0 :: Grid
g0 = array ((-size, -size), (size, size))
  [((i, size), nb) | i <- r ] ++
  [((i, -size), sb) | i <- r ] ++
  [((size, i), eb) | i <- r ] ++
  [((-size, i), wb) | i <- r ] ++
  [((size, i), eb) | i <- r ] ++
  [((i,j), interior) | i <- r, j <- r ] ++
  [((size, size), nec), ((size, -size), sec),
   ((-size, size), nwc),
   ((-size, -size), swc)]
where r = [1-size..size-1]
```

Building World g1 from World g0

...by erecting a [west/east-oriented wall](#) leading from $(-5,10)$ to $(5,10)$:

```
g1 :: Grid
g1 = g0 // mkHorWall (-5) 5 10
```

where `(//)` is the [Array](#) library function (cf. [Chapter 7.2](#)):

```
(//) :: Ix a => Array a b -> [(a,b)] -> Array a b
```

Recalling the (//) Function

...of the `Array` library:

```
(//) :: Ix a => Array a b -> [(a,b)] -> Array a b
```

and illustrating its usage: To this end, let:

```
colors :: Array Int Color
colors = array (0,7)
           [(0,Black), (1,Blue), (2,Green), (3,Cyan),
            (4,Red), (5,Magenta), (6,Yellow),
            (7,White)]
```

then:

```
colors // [(0,White), (7,Black)]
->> array (0,7) [(0,White), (1,Blue), (2,Green), (3,Cyan),
                 (4,Red), (5,Magenta), (6,Yellow),
                 (7,Black)] :: Array Int Color
```

swaps the 'black' und 'white' entries in `colors`.

Note

Type `Color` is defined as in the

- ▶ `Graphics` library:

```
data Color = Black | Blue | Green | Cyan
           | Red | Magenta | Yellow | White
           deriving (Eq, Ord, Bounded, Enum,
                   Ix, Show, Read)
```

Equivalently but *more concisely* we could have defined

- ▶ `colors` by:

```
colors :: Array Int Color
colors = array (0,7) (zip [0..7] [Black..White])
```

Utility Functions for Building Walls

Building walls horizontally (west/east-oriented, leading from (x_1, y) to (x_2, y)):

```
mkHorWall :: Int -> Int -> Int -> [(Position, [Direction])]
mkHorWall x1 x2 y =
  [((x,y), nb) | x <- [x1..x2]] ++
  [((x,y+1), sb) | x <- [x1..x2]]
```

Building walls vertically (north/south-oriented, leading from (x, y_1) to (x, y_2)):

```
mkVerWall :: Int -> Int -> Int -> [(Position, [Direction])]
mkVerWall y1 y2 x =
  [((x,y), eb) | y <- [y1..y2]] ++
  [((x+1,y), wb) | y <- [y1..y2]]
```

Utility Functions for Building Rooms

...naively, `rooms` could be built using `mkHorWall` and `mkVerWall` straightforwardly:

```
mkBox :: Position -> Position
      -> [(Position, [Direction])]
mkBox (x1, y1) (x2, y2) =
  mkHorWall (x1+1) x2 y1 ++ mkHorWall (x1+1) x2 y2 ++
  mkVerWall (y1+1) y2 x1 ++ mkVerWall (y1+1) y2 x2
```

This, however, creates two field entries for each of the four inner corners causing their values undefined after the call is finished (cf. [Chapter 7.2](#)).

This problem can elegantly be overcome by using the `Array` library operation `accum` (cf. [Chapter 7.2](#)) in combination with `mkBox`.

Recalling the accum Function

...of the `Array` library:

```
accum :: (Ix a) => (b -> c -> b)
      -> Array a b -> [(a,c)] -> Array a b
```

As discussed in [Chapter 7.2](#), `accum`

- ▶ is quite similar to `(//)`.
- ▶ in case of replicated entries the function of the first argument is applied for resolving conflicts.
- ▶ the `intersect` function of the `List` library is appropriate for this in the case of our example, e.g.:

```
[South, East, West] 'intersect'
  [North, South, West] ->> [South, West]
```

represents the [northeast corner](#).

Building World g2 from World g0

...by **building** a **room** with its **lower left** and **upper right corner** at positions **(-10,5)** and **(-5,10)**, respectively:

```
g2 :: Grid
```

```
g2 = accum intersect g0 (mkBox (-15,8) (2,17))
```

using **accum**, **intersect**, and **mkBox**.

Building World g3 from World g2

...by adding a `door` (to the middle of the top wall of the room)

```
g3 :: Grid
g3 = accum union g2 [((-7,17), interior),
                    ((-7,18), interior)]
```

using `accum`, `union`, and `interior`.

Chapter 18.2.7

Robot Graphics: Animation in Action

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1543/10

Objective of Animation

...drawing the world the robot lives in and then showing the robot running around (at some predetermined rate) accomplishing its mission:

- ▶ Drawing lines if the pen is down.
- ▶ Picking up coins.
- ▶ Dropping coins, letting them thereby appear in possibly other locations.

This requires to incrementally update the drawn and displayed graphics, which will be achieved by means of the operations of the Graphics library.

Updating the Graphics Incrementally

...key for incrementally updating the displayed world the Graphics library operation `drawInWindowNow`:

```
drawInWindowNow :: Window -> Color  
                  -> Point -> Point -> IO ()
```

which **draws** the **updated graphics immediately** after any changes, and can be used, e.g., for **drawing lines**:

```
drawLine :: Window -> Color  
           -> Point -> Point -> IO ()  
drawLine w c p1 p2 =  
  drawInWindowNow w (withColor c (line p1 p2))
```

Note

...in order to work properly, the incremental update of the world must be organized such that the

- ▶ absence of interferences of graphics actions

is ensured.

This is achieved by assuming:

1. Grid points are 10 pixels apart.
2. Walls are drawn halfway between grid points.
3. The robot pen draws lines directly from one grid point to the next.
4. Coins are drawn as yellow circles just above and to the left of each grid point.
5. Coins are erased by drawing black circles over the yellow ones which are already there.

Defining Top-level Constants

...for dealing with the preceding assumptions.

Half the distance between grid points:

```
d :: Int
d = 5
```

Color of walls and coins:

```
wc, cc :: Color
wc = Blue
cc = Yellow
```

Window size:

```
xWin, yWin :: Int
xWin = 600
yWin = 500
```

Defining Utility Functions (1)

Drawing grids:

```
drawGrid :: Window -> Grid -> IO ()
drawGrid w wld =
  let (low@(xMin,yMin),hi@(xMax,yMax)) = bounds wld
      (x1,y1)                          = trans low
      (x2,y2)                          = trans hi
  in
  do drawLine w wc (x1-d,y1+d) (x1-d,y2-d)
     drawLine w wc (x1-d,y1+d) (x1+d,y2+d)
     sequence_ [drawPos w (trans (x,y)) (wld 'at' (x,y))
                | x <- [xMin..xMax], y <- [yMin..yMax]]
```

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1548 / 10

Defining Utility Functions (2)

Used by drawGrid:

```
drawPos :: Window -> Point -> [Direction] -> IO ()
drawPos x (x,y) ds =
  do if North `notElem` ds
      then drawLine w wc (x-d,y-d) (x+d,y-d)
      else return ()
  if East `notElem` ds
  then drawLine w wc (x+d,y-d) (x+d,y+d)
  else return ()
```

Used by drawGrid, from the Array library:

```
bounds :: Ix a => Array a b -> (a,a)
-- yields the bounds of its array argument
```

Defining Utility Functions (3)

Dropping and drawing coins:

```
drawCoins :: Window -> RobotState -> IO ()
drawCoins w s = mapM_ (drawCoin w) (treasure s)

drawCoin :: Window -> Position -> IO ()
drawCoin w p =
  let (x,y) = trans p
  in drawInWindowNow w
     (withColor cc (ellipse (x-5,y-1) (x-1,y-5)))
```

Erasing coins:

```
eraseCoin :: Window -> Position -> IO ()
eraseCoin w p =
  let (x,y) = trans p
  in drawInWindowNow w
     (withColor Black (ellipse (x-5,y-1) (x-1,y-5)))
```

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1550/10

Defining Utility Functions (4)

Drawing robot moves:

```
graphicsMove :: Window -> RobotState
                                     -> Position -> IO ()
graphicsMove w s newPos =
  do if pen s
      then drawLine w (color s) (trans (position s))
                                     (trans newPos)
      else return ()
  getWindowTick w

trans      :: Position -> Point
trans (x,y) = (div xWin 2+2*d*x, div yWin 2-2*d*y)
```

Causing a short delay after each robot move

```
getWindowTick :: Window -> IO ()
```

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1551/10

Running IRL Programs: The Top-level Prg. (1)

...putting it all together.

Running an IRL program:

```
runRobot :: Robot () -> RobotState -> Grid -> IO ()
runRobot (Robot sf) s g =
  runGraphics $
  do w <- openWindowEx "Robot World" (Just (0,0))
      (Just (xWin, yWin)) drawGraphic (Just 10)
  drawGrid w g
  drawCoins w s
  spaceWait w
  sf s g w
  spaceClose w
```

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1552/10

Running IRL Programs: The Top-level Prg. (2)

Intuitively, `runRobot`

- opens a window
- draws a grid
- draws the coins
- waits for the user to hit the spacebar
- continues running the program with starting state `s` and grid `g`
- closes the window when execution is complete and the spacebar is pressed again.

where `spaceWait` provides the user with progress control by awaiting the user's `pressing the spacebar`:

```
spaceWait    :: Window -> IO ()
spaceWait w = do k <- getKey w
                if k == ' ' then return ()
                           else spaceWait w
```

Animation in Action (1)

...the grids `g0` through `g3` can now be used to run IRL programs with.

1) Fixing `s0` as a suitable starting state:

```
s0 :: RobotState
s0 = RobotState { position = (0,0)
                 , pen = False
                 , color = Red
                 , facing = North
                 , treasure = tr
                 , pocket = 0
                 }
```

2) Placing 'treasure' (all coins are placed inside the room in grid `g3`):

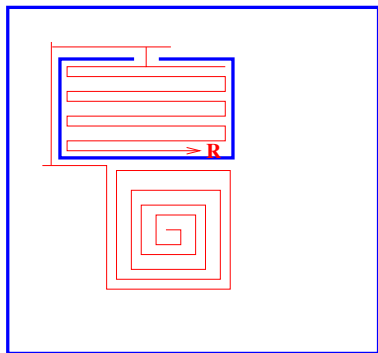
```
tr :: [Position]
tr = [(x,y) | x <- [-13,-11..1], y <- [9,11..15]]
```

Animation in Action (2)

3) Running the 'spiral' program with s0, g0:

```
main = runRobot spiral s0 g0
```

...leads to the 'spiral' example shown for illustration at the beginning of this chapter:



Chapter 18.3

Robots on Wheels

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1556/10

Outline

...we consider and define a [simulation](#) of

- ▶ [mobile robots](#) (called [Simbots](#))

using [functional reactive programming](#).

The implementation will make use of the type class

- ▶ [Arrow](#)

which is another example of a [type constructor class](#) generalizing the concept of a [monad](#).

Chapter 18.3.1

The Setting

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1558/10

The Configuration of Mobile Robots (1)

...is **assumed** to be as follows:

*“Robots are differential drive robots having **two wheels** that are each driven by an independent motor. The relative velocity of these two wheels governs the turning rate of the robot. If the velocities are identical, the robot will go straight.*

*A robot has several kinds of sensors. Among these, (1) a **bumper switch** to detect when the robot gets ‘stuck’ because of being blocked by something, (2) a **range finder** to determine the nearest object in any given direction (in the following it is assumed that there are four independent range finders that only look forward, backward, left and right; the range finder will thus only be queried at these four angles), (4) an **animate object tracker** that gives the current position of all other robots and possibly those of some free-moving balls that are within a certain distance from the robot.*

The Configuration of Mobile Robots (2)

This object tracker can be thought of as *modelling either a visual subsystem that can 'see' these objects, or a communication subsystem through which the robots and balls share each other's coordinates. Some further capabilities will be introduced as need occurs.*

Last but not least, each robot has a unique ID."

The Application Scenario: Robot Soccer

...the overall task:

“Write a program to play ‘robocup soccer’ as follows:

Use wall segments to create two goals at either end of the field.

Decide on a number of players on each team and write generic controllers, such as one for a goalkeeper, one for attack, and one for defense.

Create an initial world where the ball is at the center mark, and each of the players is positioned strategically while being on-side (with the defensive players also outside of the center circle. Each team may use the same controller, or different ones.”

Code for 'Robots on Wheels'

...can be down-loaded at the [Yampa homepage](http://www.haskell.org/yampa) at

<http://www.haskell.org/yampa>

In the following we highlight essential [code snippets](#).

Chapter 18.3.2

Modelling the Robots' World

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1563/10

Signal Functions, Signals, and Simbots

Signal functions are

- ▶ [signal transformers](#), i.e., functions mapping signals to signals,
- ▶ of type [SF](#), a 2-ary type constructor defined in [Yampa](#), which is an instance of type constructor class [Arrow](#).

[Yampa](#) provides

- ▶ a number of [primitive signal functions](#) and a set of special [composition operators](#) (or [combinators](#)) for constructing (more) complex signal functions from simpler ones.

[Signals](#) are no

- ▶ first-class values in [Yampa](#) but can only be manipulated by means of signal functions to avoid time- and space-leaks (abstract data type).

[Simbot](#) is a short hand for [simulated robot](#).

Modelling Time, Signals, and Signal Functions

SF is an instance of class `Arrow`:

```
type Time      = Double
```

```
type Signal a~ = Time -> a
```

```
type SF a b    = Signal a -> Signal b
```

Intuitively: SF-values are **signal transformers** resp. **signal functions** (thus the type name `SF`).

Modelling Simbots

```
type RobotType = String
type RobotId   = Int

type SimbotController =
    SimbotProperties -> SF SimbotInput SimbotOutput

Class HasRobotProperties i where
  rpType      :: i -> RobotType      -- Type of robot
  rpId        :: i -> RobotId        -- Identity of robot
  rpDiameter  :: i -> Length         -- Distance between wheels
  rpAccMax    :: i -> Acceleration   -- Max translational acc
  rpWSMax     :: i -> Speed          -- Max wheel speed
```

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1566/10

Modelling the World

```
type WorldTemplate = [ObjectTemplate]

data ObjectTemplate =
  OTBlock      otPos  :: Position2  -- Square obstacle
| OTVWall     otPos  :: Position2  -- Vertical wall
| OTHWall     otPos  :: Position2  -- Horizontal wall
| OTBall      otPos  :: Position2  -- Ball
| OTSimbotA   otRId  :: RobotId,   -- Simbot A robot
              otPos  :: Position2,
              otHdng :: Heading
| OTSimbotB   otRId  :: RobotId,   -- Simbot B robot
              otPos  :: Position2,
              otHdng :: Heading
```

Chapter 18.3.3

Classes of Robots

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1568 / 10

Types of Robots

...usually, there are **different types of robots**

- ▶ differing in their features (2 wheels, 3 wheels, camera, sonar, speaker, blinker, etc.)

The **type of a robot** is fixed by its

- ▶ **input** and **output** types

which are encoded in **input** and **output classes** together with the functions operating on the class elements.

Input Classes (1)

...and functions operating on their elements:

```
data BatteryStatus = BSHigh | BSLow | BSCritical
                    deriving (Eq, Show)
```

```
class HasRobotStatus i where
  -- Current battery status
  rsBattStat :: i -> BatteryStatus
  -- Currently stuck or not stuck
  rsIsStuck  :: i -> Bool
```

```
-- Derived event sources:
```

```
rsBattStatChanged  :: HasRobotStatus i =>
                    SF i (Event BatteryStatus)
rsBattStatLow      :: HasRobotStatus i => SF i (Event ())
rsBattStatCritical :: HasRobotStatus i => SF i (Event ())
rsStuck            :: HasRobotStatus i => SF i (Event ())
```

Input Classes (2)

```
class HasOdometry where
  -- Current position
  odometryPosition :: i -> Position2
  -- Current heading
  odometryHeading  :: i -> Heading

class HasRangeFinder i where
  rfRange      :: i -> Angle -> Distance
  rfMaxRange   :: i -> Distance

-- Derived range finders:
rfFront :: HasRangeFinder i => i -> Distance
rfBack  :: HasRangeFinder i => i -> Distance
rfLeft  :: HasRangeFinder i => i -> Distance
rfRight :: HasRangeFinder i => i -> Distance
```

Input Classes (3)

```
class HasAnimateObjectTracker i where
  aotOtherRobots :: i -> [(RobotType, Angle, Distance)]
  aotBalls       :: i -> [(Angle, Distance)]

class HasTextualConsoleInput i where
  tciKey :: i -> Maybe Char

tciNewKeyDown :: HasTextualConsoleInput i =>
                Maybe Char -> SF i (Event Char)

tciKeyDown    :: HasTextualConsoleInput i =>
                SF i (Event Char)
```

Output Classes

...and functions operating on their elements:

```
class MergeableRecord o => HasDiffDrive o where
  -- Brake both wheels
  ddBrake :: MR o
  -- Set wheel velocities
  ddVelDiff :: Velocity -> Velocity -> MR o
  -- Set velocities and rotation
  ddVelTR :: Velocity -> RotVel -> MR o

class MergeableRecord o =>
  HasTextConsoleOutput o where
  tcoPrintMessage :: Event String -> MR o
```

Chapter 18.3.4

Robot Simulation in Action

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1574/19

Typical Structure of a Robot Control Program

```
module MyRobotShow where

import AFrob
import AFrobRobotSim

main :: IO ()
main = runSim (Just world) rcA rcB

world :: WorldTemplate
world = ...

-- controller for simbot A
rcA :: SimbotController
rcA = ...

-- controller for simbot B
rcB :: SimbotController
rcB = ...
```

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1575/10

Robot Simulation in Action

Running a robot simulation:

```
runSim :: Maybe WorldTemplate
        -> SimbotController
        -> SimbotController -> IO ()
```

Simbot controllers:

```
rcA :: SimbotController
rcA rProps =
  case rrpId rProps of
    1 -> rcA1 rProps
    2 -> rcA2 rProps
    3 -> rcA3 rProps

rcA1, rcA2, rcA3 :: SimbotController
rcA1 = ...
rcA2 = ...
rcA3 = ...
```


Chapter 18.3.5

Examples

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1577/19

Robot Actions: Control Programs (1)

A stationary robot:

```
rcStop :: SimbotController
rcStop _ = constant (mrFinalize ddBrake)
```

A blind robot moving at constant speed:

```
rcBlind1 _ =
  constant (mrFinalize $ ddVelDiff 10 10)
```

A blind robot moving at half the maximum speed:

```
rcBlind2 rps =
  let max = rpWSMax rps
  in constant (mrFinalize $
                ddVelDiff (max/2) (max/2))
```

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1578/10

Robot Actions: Control Programs (2)

A robot rotating at a pre-given speed:

```
rcTurn :: Velocity -> SimbotController
rcTurn vel rps =
  let vMax = rpWSMax rps
      rMax = 2 * (vMax - vel) / rpDiameter rps
  in constant (mrFinalize $ ddVelTR vel rMax)
```

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1579/10

Chapter 18.4

In Conclusion

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1580/19

The Origins

...of [functional reactive programming \(FRP\)](#) can be traced back to [functional reactive animation \(FRAn\)](#):

- ▶ Conal Elliot, Paul Hudak. [Functional Reactive Animation](#). In Proceedings of the 2nd ACM SIGPLAN 1997 International Conference on Functional Programming (ICFP'97), 263-273, 1997.
- ▶ Conal Elliot. [Functional Implementations of Continuous Modeled Animation](#). In Proceedings of the 10th International Symposium on Principles of Declarative Programming, held jointly with the International Conference on Algebraic and Logic Programming (PLILP/ALP'98), Springer-V., LNCS 1490, 284-299, 1998.

Seminal Works

...on functional reactive programming (FRP):

- ▶ Zhanyong Wan, Paul Hudak. **Functional Reactive Programming from First Principles**. In Proceedings of the ACM SIGPLAN 2000 Conference on Programming Languages Design and Implementation (PLDI 2000), ACM Press, 2000.
- ▶ John Peterson, Zhanyong Wan, Paul Hudak, Henrik Nilsson. **Yale FRP User's Manual**. Department of Computer Science, Yale University, January 2001.
<http://www.haskell.org/frp/manual.html>
- ▶ Henrik Nilsson, Antony Courtney, John Peterson. **Functional Reactive Programming, Continued**. In Proceedings of the ACM SIGPLAN Workshop on Haskell (Haskell 2002), 51-64, 2002.

Applications of FRP (1)

...on [Functional Reactive Robotics \(FRob\)](#):

- ▶ Izzet Pembeci, Henrik Nilsson, Gregory D. Hager. [Functional Reactive Robotics: An Exercise in Principled Integration of Domain-Specific Languages](#). In Proceedings of the 4th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2002), 168-179, 2002.
- ▶ John Peterson, Gregory Hager, Paul Hudak. [A Language for Declarative Robotic Programming](#). In Proceedings of the IEEE International Conference on Robotics and Automation (ICRA'99), Vol. 2, 1144-1151, 1999.

Applications of FRP (2)

...on [Functional Animation Languages \(FAL\)](#):

- ▶ Paul Hudak. [The Haskell School of Expression – Learning Functional Programming through Multimedia](#). Cambridge University Press, 2000. (Chapter 15, A Module of Reactive Animations)

...on [Functional Vision Systems \(FVision\)](#):

- ▶ Alastair Reid, John Peterson, Gregory D. Hager, Paul Hudak. [Prototyping Real-Time Vision Systems: An Experiment in DSL Design](#). In Proceedings of the 1999 International Conference on Software Engineering (ICSE'99), 484-493, 1999.

...on [Functional Reactive User Interfaces \(FRUIt\)](#):

- ▶ Antony Courtney, Conal Elliot. [Genuinely Functional User Interfaces](#). In Proceedings of the 2001 Haskell Workshop, September 2001.

Applications of FRP (3)

...towards [Real-Time FRP \(RT-FRP\)](#):

- ▶ Zhanyong Wan, Walid Taha, Paul Hudak. [Real-Time FRP](#). In Proceedings of the 6th ACM SIGPLAN International Conference on Functional Programming (ICFP 2001), 146-156, 2001.
- ▶ Zhanyong Wan. [Functional Reactive Programming for Real-Time Embedded Systems](#). PhD thesis. Department of Computer Science, Yale University, December 2002.

...towards [Event-Driven FRP \(ED-FRP\)](#):

- ▶ Zhanyong Wan, Walid Taha, Paul Hudak. [Event-Driven FRP](#). In Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages (PADL 2002), Springer-V., LNCS 2257, 155-172, 2002.

Chapter 18.5

References, Further Reading

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10




Chap. 11

Chap. 12




Chap. 13

1586/10



Chapter 18: Further Reading (1)

-  Ronald C. Arkin. *Behavior-Based Robotics*. MIT Press, 1998.
-  Antony Courtney, Conal Elliot. *Genuinely Functional User Interfaces*. In Proceedings of the 2001 Haskell Workshop (Haskell 2001), September 2001.
-  Conal Elliot. *Functional Implementations of Continuous Modeled Animation*. In Proceedings of the 10th International Symposium on Principles of Declarative Programming, held jointly with the International Conference on Algebraic and Logic Programming (PLILP/ALP'98), Springer-V., LNCS 1490, 284-299, 1998.




Chapter 18: Further Reading (2)

-  Conal Elliot, Paul Hudak. *Functional Reactive Animation*. In Proceedings of the 2nd ACM SIGPLAN 1997 International Conference on Functional Programming (ICFP'97), 263-273, 1997.
-  David Harel, Assaf Marron, Gera Weiss. *Behavioral Programming*. Communications of the ACM 55(7):90-100, 2012.
-  Paul Hudak. *The Haskell School of Expression – Learning Functional Programming through Multimedia*. Cambridge University Press, 2000. (Chapter 15, A Module of Reactive Animations; Chapter 18, Higher-Order Types; Chapter 19, An Imperative Robot Language)

Chapter 18: Further Reading (3)

-  Paul Hudak, Antony Courtney, Henrik Nilsson, John Peterson. *Arrows, Robots, and Functional Reactive Programming*. In Johan Jeuring, Simon Peyton Jones (Eds.), *Advanced Functional Programming – Revised Lectures*. Springer-V., LNCS Tutorial 2638, 159-187, 2003.
-  John Hughes. *Generalising Monads to Arrows*. *Science of Computer Programming* 37:67-111, 2000.
-  Henrik Nilsson, Antony Courtney, John Peterson. *Functional Reactive Programming, Continued*. In Proceedings of the ACM SIGPLAN Workshop on Haskell (Haskell 2002), 51-64, 2002.
-  Johan Nordlander. *Reactive Objects and Functional Programming*. PhD thesis. Chalmers University of Technology, 1999.




Chapter 18: Further Reading (4)

-  Ross Paterson. *A New Notation for Arrows*. In Proceedings of the 6th ACM SIGPLAN Conference on Functional Programming (ICFP 2001), 229-240, 2001.
-  Ross Paterson. *Arrows and Computation*. In Jeremy Gibbons, Oege de Moor (Eds.), *The Fun of Programming*. Palgrave MacMillan, 201-222, 2003.
-  Izzet Pembeci, Henrik Nilsson, Gregory D. Hager. *Functional Reactive Robotics: An Exercise in Principled Integration of Domain-Specific Languages*. In Proceedings of the 4th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2002), 168-179, 2002.

Chapter 18: Further Reading (5)

-  John Peterson, Gregory D. Hager, Paul Hudak. *A Language for Declarative Robotic Programming*. In Proceedings of the IEEE International Conference on Robotics and Automation (ICRA'99), Vol. 2, 1144-1151, 1999.
-  John Peterson, Paul Hudak, Conal Elliot. *Lambda in Motion: Controlling Robots with Haskell*. In Proceedings of the 1st International Workshop on Practical Aspects of Declarative Languages (PADL'99), Springer-V., LNCS 1551, 91-105, 1999.
-  John Peterson, Zhanyong Wan, Paul Hudak, Henrik Nilsson. *Yale FRP User's Manual*. Department of Computer Science, Yale University, January 2001.
www.haskell.org/frp/manual.html

Chapter 18: Further Reading (6)

-  Tomas Petricek, Jon Skeet. *Real World Functional Programming: With Examples in F# and C#*. Manning Publications Co., 2009. (Chapter 16, Developing reactive functional programs)
-  Alastair Reid, John Peterson, Gregory D. Hager, Paul Hudak. *Prototyping Real-Time Vision Systems: An Experiment in DSL Design*. In Proceedings of the 1999 International Conference on Software Engineering (ICSE'99), 484-493, 1999.
-  Zhanyong Wan. *Functional Reactive Programming for Real-Time Embedded Systems*. PhD Thesis, Department of Computer Science, Yale University, December 2002.

Chapter 18: Further Reading (7)

-  Zhanyong Wan, Paul Hudak. *Functional Reactive Programming from First Principles*. In Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI 2000), 242-252, 2000.
-  Zhanyong Wan, Walid Taha, Paul Hudak. *Real-Time FRP*. In Proceedings of the 6th ACM SIGPLAN International Conference on Functional Programming (ICFP 2001), 146-156, 2001.
-  Zhanyong Wan, Walid Taha, Paul Hudak. *Event-Driven FRP*. In Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages (PADL 2002), Springer-V., LNCS 2257, 155-172, 2002.

Part VI

Extensions, Perspectives

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1594 / 19

Chapter 19

Extensions: Parallel and 'Real World' Functional Programming

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1595/10

Chapter 19.1

Parallelism in Functional Languages

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1596 / 10

Motivation, Background

...recall:

- ▶ Konrad Hinsen. [The Promises of Functional Programming](#). *Computing in Science and Engineering* 11(4):86-90, 2009.

...adopting a functional programming style could make your programs more robust, more compact, and **more easily parallelizable**.

Reading for this chapter:

- ▶ Peter Pepper, Petra Hofstedt. [Funktionale Programmierung](#), Springer-V., 2006. (In German). (Kapitel 21, Massiv Parallele Programme)

Parallelism in Programming Languages

Predominant in imperative languages:

- ▶ Libraries (PVM, MPI) \rightsquigarrow Message Passing Model (C++, C, Fortran)
- ▶ Data-parallel Languages (e.g., High Performance Fortran)

Predominant in functional languages:

- ▶ Implicit (expression) parallelism
- ▶ Explicit parallelism
- ▶ Algorithmic skeletons

Implicit Parallelism

...also known as **expression parallelism**.

Idea: If $f(e_1, \dots, e_n)$ is a functional expression, then

- ▶ arguments (and functions) can be evaluated **in parallel**.

Most **important**

- ▶ **advantage:** Parallelism **for free!** No effort for the programmer at all.
- ▶ **disadvantage:** Results often unsatisfying; e.g. granularity, load distribution, etc., is not taken into account.

Overall, **expression parallelism** is

- ▶ **easy to detect** (for the compiler) but **hard to fully exploit**.

Explicit Parallelism

Idea: Introducing and using

- ▶ **meta-statements** (e.g., for controlling the data and load distribution, communication).

Most important

- ▶ **advantage**: Often very good results thanks to explicit hands-on control of the programmer.
- ▶ **disadvantage**: High programming effort and loss of functional elegance.

Algorithmic Skeletons

...a **compromise** between

- ▶ **explicit imperative** parallel programming
- ▶ **implicit functional** expression parallelism

In the following

...we consider a setting with

- ▶ massively parallel systems
- ▶ algorithmic skeletons

Massively Parallel Systems

...are typically characterized by a

- ▶ large number of processors with
 - local memory
 - communication by message exchange
- ▶ MIMD-Parallel Processor Architecture (Multiple Instruction/Multiple Data)

Here we focus and restrict ourselves to

- ▶ SPMD-Programming Style (Single Program/Multiple Data)

Algorithmic Skeletons

- ▶ represent typical patterns for parallelization ([Farm](#), [Map](#), [Reduce](#), [Branch&Bound](#), [Divide&Conquer](#),...).
- ▶ are [easy to instantiate](#) for the programmer.
- ▶ allow parallel programming at a [high level of abstraction](#).

Implementing Algorithmic Skeletons

...in functional languages

- ▶ by special **higher-order functions**.
- ▶ with parallel implementation.
- ▶ embedded in sequential languages.
- ▶ using message passing via skeleton hierarchies.

Advantages:

- ▶ **Hiding** of parallel implementation details in the skeleton.
- ▶ **Elegance and (parallel) efficiency** for special application patterns.

Example: Parallel Map on Distributed List

Consider the higher-order function `map` on lists:

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = (f x) : (map f xs)
```

Observation:

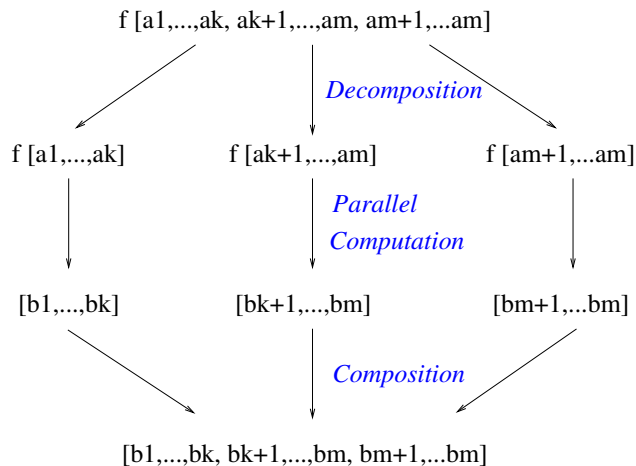
- Applying `f` to a list element does not depend on other list elements.

Parallelization idea:

- Divide the list into sublists followed by `parallel` application of `map` to the sublists:
 \rightsquigarrow parallelization pattern `Farm`.

Parallel Map on Distributed Lists

Illustration:



Peter Pepper, Petra Hofstedt. Funktionale Programmierung.
Springer, 2006, S. 445.

Implementing

...the **parallel map function** requires

- ▶ special data structures, which take into account the aspect of **distribution** (ordinary lists are inefficient for this purpose).

Skeletons on distributed data structures are so-called

- ▶ **data-parallel skeletons**.

Note the **difference** between:

- ▶ **Data-parallelism**: Assumes an **a priori** distribution of data on different processors.
- ▶ **Task-parallelism**: Processes and data to be distributed are not known **a priori** but dynamically generated.

Implementing a Parallel Application

...using [algorithmic skeletons](#) requires:

- ▶ Recognizing problem-inherent parallelism.
- ▶ Selecting an adequate data distribution (granularity).
- ▶ Selecting a suitable skeleton from a library.
- ▶ Instantiating the skeleton problem-specifically.

Remark:

- ▶ Some languages (e.g., [Eden](#)) support the implementation of skeletons (in addition to those which might be provided by a library).

Data Distribution on Processors

...is **crucial** for

- ▶ the structure of the complete algorithm.
- ▶ efficiency.

The **hardness** of the **distribution problems** depends on

- ▶ Independence of all data elements (like in the map-example): Distribution is easy.
- ▶ Independence of subsets of data elements.
- ▶ Complex dependences of data elements: Adequate distribution is challenging.

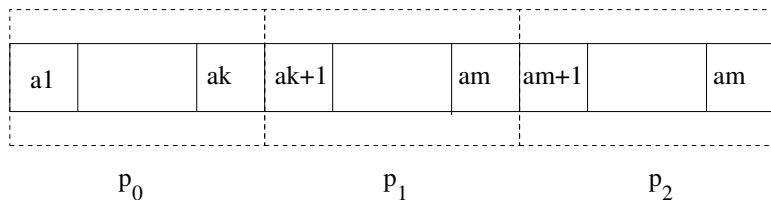
Auxiliary means: So-called **covers** for

- ▶ describing the **decomposition** and **communication pattern** of a data structure (investigated by various researchers).

Example (1)

...illustrating a [simple list cover](#).

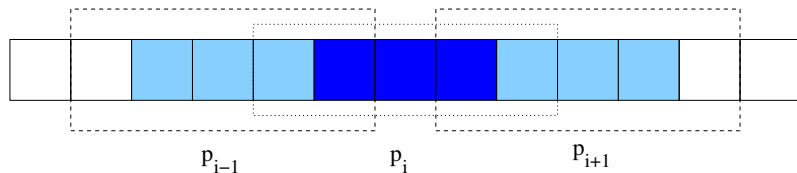
Distributing a list on [three](#) processors p_0 , p_1 , and p_2 :



Peter Pepper, Petra Hofstedt. Funktionale Programmierung.
Springer, 2006, S. 446.

Example (2)

...illustrating a list cover with overlapping elements.



Peter Pepper, Petra Hofstedt. Funktionale Programmierung.
Springer, 2006, S. 446.

General Structure of a Cover

```
Cover = {  
    Type S a          -- Whole object  
        C b          -- Cover  
        U c          -- Local sub-objects  
  
    split :: S a -> C (U a) -- Decomposing the  
                                -- original object  
  
    glue  :: C (U a) -> S a  -- Composing the  
                                -- original object  
  
}
```

where it must **hold**: `glue . split = id`

Note: The above code snippet is not (valid) Haskell.

Implementing Covers

...requires **support** for

- ▶ the specification of covers.
- ▶ the programming of algorithmic skeletons on covers.
- ▶ the provision of often used skeletons in libraries.

which is **currently** a

- ▶ **hot research topic**

in **functional programming**.

Chapter 19.2

Haskell for 'Real World' Programming

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1615/19

'Real World' Haskell (1)

...[Haskell](#) these days provides considerable, mature, and stable support for:

- ▶ Systems Programming
- ▶ (Network) Client and Server Programming
- ▶ Data Base and Web Programming
- ▶ Multicore Programming
- ▶ Foreign Language Interfaces
- ▶ Graphical User Interfaces
- ▶ File I/O and filesystem programming
- ▶ Automated Testing, Error Handling, and Debugging
- ▶ Performance Analysis and Tuning
- ▶ ...

'Real World' Haskell (2)

This support comes mostly in terms of

- ▶ sophisticated libraries

and makes [Haskell](#) a reasonable choice for addressing and solving

- ▶ real world problems

as the choice of a language depends much on the ability and support a [programming language](#) provides for linking and connecting to the 'outer world:' the language's

- ▶ eco-system.

See e.g.:



Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008.

Chapter 19.3

References, Further Reading

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12





Chap. 13

1618/19





Chapter 19.1: Further Reading (1)

-  Joe Armstrong, Robert Virding, Claes Wikstrom, Mike Williams. *Concurrent Programming in Erlang*. Prentice-Hall, 2nd edition, 1996.
-  Martin Braun, Oleg Lobachev, Philip W. Trinder: *Arrows for Parallel Computation*. CoRR, <http://arxiv.org/abs/1801.02216>, 2018.
-  Manuel M.T. Chakravarty, Yike Guo, Martin Köhler, Hendrik C.R. Lock. *GOFIN: Higher-Order Functions meet Concurrency Constraints*. *Science of Computer Programming* 30(1-2):157-199 1998.





Chapter 19.1: Further Reading (2)

-  Manuel M.T. Chakravarty, Roman Leshchinsky, Simon Peyton Jones, Gabriele Keller, Simon Marlow. *Data Parallel Haskell: A Status Report*. In Proceedings on the Workshop on Declarative Aspects of Multicore Programming (DAMP 2007), ACM, New York, 10-18, 2007.
-  Koen Claessen. *A Poor Man's Concurrency Monad*. Journal of Functional Programming 9(3):313-323, 1999.
-  Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. The MIT Press, 1989.
-  Antonie J.T. Davie. *An Introduction to Functional Programming Systems using Haskell*. Cambridge University Press, 1992. (Chapter 11, Parallel Evaluation)




Chapter 19.1: Further Reading (3)

-  Sören Holmström. *PFL: A Functional Language for Parallel Programming*. In *Declarative Programming Workshop*, 114-139, 1983.
-  Peng Li, Simon Marlow, Simon Peyton Jones, Andrew Tolmach. *Lightweight Concurrency Primitives for GHC*. In *Proceedings of the ACM SIGPLAN Workshop on Haskell (Haskell 2007)*, 107-118, 2007.
-  Hans-Werner Loidl et al. *Comparing Parallel Functional Languages: Programming and Performance*. *Higher-Order and Symbolic Computation* 16(3):203-251, 2003.
-  Simon Marlow. *Parallel and Concurrent Programming in Haskell*. O'Reilley, 2013.




Chapter 19.1: Further Reading (4)

-  Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Chapter 24, Concurrent and Multicore Programming)
-  Peter Pepper, Petra Hofstedt. *Funktionale Programmierung*. Springer-V., 2006. (Kapitel 21, Massiv Parallele Programme)
-  Tomas Petricek, Jon Skeet. *Real World Functional Programming: With Examples in F# and C#*. Manning Publications Co., 2009. (Chapter 14, Writing parallel functional programs)
-  Simon Peyton Jones, Andrew Gordon, Sigbjorn Finne. *Concurrent Haskell*. In Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96), 295-308, 1996.



Chapter 19.1: Further Reading (5)

-  Robert F. Pointon, Philip W. Trinder, Hans-Wolfgang Loidl. *The Design and Implementation of Glasgow Distributed Haskell*. In Proceedings of the 12th International Workshop on Implementation of Functional Languages (IFL 2000), LNCS 2011, Springer-V., 53-70, 2000.
-  Fethi A. Rabhi. *Exploiting Parallelism in Functional Languages: A Paradigm Oriented Approach*. In J. R. Davy, P. M. Dew (Eds.), *Abstract Machine Models for Highly Parallel Computers*, Oxford University Press, 118-139, 1995.
-  Fethi A. Rabhi, Guy Lapalme. *Algorithms – A Functional Programming Approach*. Addison-Wesley, 1999. (Chapter 10.3, Parallel Algorithms)

Chapter 19.1: Further Reading (6)

-  Simon Peyton Jones, Satnam Sing. *A Tutorial on Parallel and Concurrent Programming in Haskell. Advanced Functional Programming – Revised Lectures*. Springer-V., LNCS 5832, 267-305, 2008.
-  Philip W. Trinder, Kevin Hammond, Hans-Wolfgang Loidl, Simon Peyton Jones. *Algorithms + Strategy = Parallelism*. Journal of Functional Programming 8(1):23-60, 1998.
-  Philip W. Trinder, Hans-Wolfgang Loidl, Robert F. Poynton. *Parallel and Distributed Haskells*. Journal of Functional Programming 12(4&5):469-510, 2002.





Chapter 19.2: Further Reading (7)

-  Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Chapter 17, Interfacing with C: The FFI; Chapter 19, Error Handling; Chapter 20, Systems Programming in Haskell; Chapter 21, Using Data Bases; Chapter 22, Extended Example: Web Client Programming; Chapter 23, GUI Programming with gtk2hs; Chapter 24, Concurrent and Multicore Programming; Chapter 27, Sockets and Syslog; Chapter 25, Profiling and Optimization; Chapter 28, Software Transactional Memory)
-  Magnus Carlsson, Thomas Hallgren. *Fudgets – A Graphical User Interface in a Lazy Functional Language*. In Proceedings of the 6th ACM International Conference on Functional Programming Languages and Computer Architecture (FPCA'93), 321-330, 1993.

Chapter 19.2: Further Reading (8)

-  Antony Courtney, Conal Elliot. *Genuinely Functional User Interfaces*. In Proceedings of the 2001 Haskell Workshop (Haskell 2001), September 2001.
-  Thomas Hallgren, Magnus Carlsson. *Programming with Fudgets*. In Johan Jeuring, Erik Meijer (Eds.), *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques*. Springer-V., LNCS 925, 137-182, 1995.
-  Nigel W.O. Hutchison, Ute Neuhaus, Manfred Schmidt-Schauß, Cordelia V. Hall. *Natural Expert: A Commercial Functional Programming Environment*. *Journal of Functional Programming* 7(2):163-182, 1997.

Chapter 19.2: Further Reading (9)

-  Peter Pepper, Petra Hofstedt. *Funktionale Programmierung*. Springer-V., 2006. (Kapitel 19, Agenten und Prozesse; Kapitel 20, Graphische Schnittstellen (GUIs))
-  Tomas Petricek, Jon Skeet. *Real World Functional Programming: With Examples in F# and C#*. Manning Publications Co., 2009.
-  Neil Savage. *Using Functions for Easier Programming*. Communications of the ACM 61(5):29-30, 2018.
-  Curt J. Simpson. *Experience Report: Haskell in the “Real World”*: Writing a Commercial Application in a Lazy Functional Language. In Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP 2009), 185-190, 2009.

Chapter 19.2: Further Reading (10)

-  “Haskell community.” *Hackage: A Repository for Open Source Haskell Libraries*. hackage.haskell.org
-  “Haskell community.” *Haskell wiki*.
haskell.org/haskellwiki/Applications_and_libraries
-  “Haskell community.” *Haskell in Industry and Open Source*.
www.haskell.org/haskellwiki/Haskell_in_industry
-  Hoogle, Hayoo. Useful search engines.
www.haskell.org/hoogle,
holumbus.fh-wedel.de/hayoo/hayoo.html

Chapter 20

Conclusions, Perspectives

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1629/19

Chapter 20.1

Research Venues, Research Topics, and More

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1630/19

Research Venues, Research Topics, and More

...for **functional programming** and **functional programming languages**:

- ▶ **Research/publication/dissemination venues**
 - Conference and Workshop Series
 - Archival Journals
 - Summer Schools
- ▶ **Research Topics**
- ▶ **Functional Programming in the Real World**

Relevant Conference and Workshop Series

For [functional programming](#):

- ▶ Annual ACM SIGPLAN International Conference on Functional Programming (ICFP) Series, since 1996.
- ▶ Annual Symposium on Functional and Logic Programming (FLPS) Series, since 2000.
- ▶ Annual ACM SIGPLAN Haskell Workshop Series, since 2002.
- ▶ HAL Workshop Series, since 2006.

For [programming in general](#):

- ▶ Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages and Systems (POPL), since 1973.
- ▶ Annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), since 1988 (resp. 1973).

Relevant Archival Journals

For **functional programming**:

- ▶ **Journal of Functional Programming**, since 1991.

For **programming in general**:

- ▶ **ACM Transactions on Programming Languages and Systems (TOPLAS)**, since 1979.
- ▶ **ACM Computing Surveys**, since 1969.

Summer Schools

Focused on [functional programming](#):

- ▶ Summer School Series on [Advanced Functional Programming](#). Springer-V., LNCS series.

Hot Research Topics – Haskell Symposium (1)

...in [theory and practice of functional programming](#) considering the [2012 Call for Papers of the Haskell Symposium](#):

“The purpose of the [Haskell Symposium](#) is to discuss [experiences with Haskell](#) and future developments for the language.

Topics of interest include, but are not limited to:

- ▶ [Language Design](#), with a focus on possible extensions and modifications of Haskell as well as critical discussions of the status quo;
- ▶ [Theory](#), such as formal treatments of the semantics of the present language or future extensions, type systems, and foundations for program analysis and transformation;
- ▶ [Implementations](#), including program analysis and transformation, static and dynamic compilation for sequential, parallel, and distributed architectures, memory management as well as foreign function and component interfaces;

Hot Research Topics – Haskell Symposium (2)

- ▶ **Tools**, in the form of profilers, tracers, debuggers, pre-processors, testing tools, and suchlike;
- ▶ **Applications**, using Haskell for scientific and symbolic computing, database, multimedia, telecom and web applications, and so forth;
- ▶ **Functional Pearls**, being elegant, instructive examples of using Haskell;
- ▶ **Experience Reports**, general practice and experience with Haskell, e.g., in an education or industry context.”

More on [Haskell 2012](http://www.haskell.org/haskell-symposium/2012/), Copenhagen, DK, 13 Sep 2012:
<http://www.haskell.org/haskell-symposium/2012/>

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1636/10

Hot Research Topics – ICFP (1)

...in [theory and practice of functional programming](#) considering the [2012 Call for Papers of ICFP](#):

“[ICFP 2012](#) seeks original papers on the [art and science of functional programming](#). Submissions are invited on all topics from [principles to practice](#), from [foundations to features](#), and from [abstraction to application](#). The scope includes all languages that encourage functional programming, including both purely applicative and imperative languages, as well as languages with objects, concurrency, or parallelism.

Topics of interest include (but are not limited to):

- ▶ [Language Design](#): concurrency and distribution; modules; components and composition; metaprogramming; interoperability; type systems; relations to imperative, object-oriented, or logic programming

Hot Research Topics – ICFP (2)

- ▶ **Implementation**: abstract machines; virtual machines; interpretation; compilation; compile-time and run-time optimization; memory management; multi-threading; exploiting parallel hardware; interfaces to foreign functions, services, components, or low-level machine resources
- ▶ **Software-Development Techniques**: algorithms and data structures; design patterns; specification; verification; validation; proof assistants; debugging; testing; tracing; profiling
- ▶ **Foundations**: formal semantics; lambda calculus; rewriting; type theory; monads; continuations; control; state; effects; program verification; dependent types
- ▶ **Analysis and Transformation**: control-flow; data-flow; abstract interpretation; partial evaluation; program calculation

Hot Research Topics – ICFP (3)

- ▶ **Applications and Domain-Specific Languages:** symbolic computing; formal-methods tools; artificial intelligence; systems programming; distributed-systems and web programming; hardware design; databases; XML processing; scientific and numerical computing; graphical user interfaces; multimedia programming; scripting; system administration; security
- ▶ **Education:** teaching introductory programming; parallel programming; mathematical proof; algebra
- ▶ **Functional Pearls:** elegant, instructive, and fun essays on functional programming
- ▶ **Experience Reports:** short papers that provide evidence that functional programming really works or describe obstacles that have kept it from working”

Chapter 20.2

Programming Contest

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1640/19

Programming Contest Series: Background (1)

...considering the [2012 contest edition](#) for illustration.

The [ICFP Programming Contest 2012](#) is the 15th instance of the annual programming contest series sponsored by [The ACM SIGPLAN International Conference on Functional Programming](#). This year, [the contest starts at 12:00 July 13 Friday UTC and ends at 12:00 July 16 Monday UTC](#). There will be a lightning division, ending at 12:00 July 14 Saturday UTC.

The task description will be published at icfpcontest2012.wordpress.com/task when the contest starts. Solutions to the task must be submitted online before the contest ends. Details of the submission procedure will be announced along with the contest task.

This is an [open contest](#). [Anybody may participate](#) except for the contest organisers and members of the same group as the contest chairs. [No advance registration or entry fee is required](#).

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1641/10

Programming Contest Series: Background (2)

Any programming language(s) may be used as long as the submitted program can be run by the judges on a standard Linux environment with no network connection. Details of the judges' environment will be announced later.

There will be cash prizes for the first and second place teams, the team winning the lightning division, and a discretionary judges' prize. There may also be travel support for the winning teams to attend the conference. (The prizes and travel support are subject to the budget plan of ICFP 2012 pending approval by ACM.)...

More on [ICFP 2012](http://icfpconference.org/icfp2012/cfp.html), Copenhagen, DK, 10-12 Sep 2012:
<http://icfpconference.org/icfp2012/cfp.html>

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1642/10

The 22nd Programming Contest at ICFP 2019

In 2019, the [programming contest](#) started on

- ▶ Friday 21 June 2019 10:00am UTC. The 24hr lightning division will end at Saturday 22 June 2019 10:00am UTC and the 72hr full contest will end at Monday 24 June 2019 10:00am UTC; full information is available online:

<https://icfpcontest2019.github.io>

- ▶ News were made available at the following sites:
 - Programming contest series at the ICFP conf. series:
<https://www.icfpconference.org/contest.html>
 - 22nd Programming contest edition in 2019:
<https://icfpcontest2019.github.io/>
 - 2019 Host conference:
ICFP 2019, Berlin, Germany, August 18-23, 2019:
<https://icfp19.sigplan.org/home>

Contest Announcement at ICFP 2020

...coming soon!

Key dates can be expected to be similar as in 2019.

ICFP 2020, Jersey City, New Jersey, USA,
Sun 23 - Fri 28 August 2020:

<https://icfp20.sigplan.org/home>

...stay tuned for [conference](#) and [contest news](#) at:

- ▶ Programming contest series at the ICFP conf. series:
<https://www.icfpconference.org/contest.html>
- ▶ 23rd Programming contest edition in 2020:
<https://icfpcontest2020.github.io/>
- ▶ 2020 Host conference:
ICFP 2020, Jersey City, New Jersey, USA, Sun 23 - Fri 28
August 2020: <https://icfp20.sigplan.org/home>

Chapter 20.3

In Conclusion

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1645/19

Functional Programming

...certainly arrived in the [real world](#):

- ▶ Curt J. Simpson. [Experience Report: Haskell in the “Real World”: Writing a Commercial Application in a Lazy Functional Language](#). In Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP 2009), 185-190, 2009.
- ▶ Jerzy Karczmarczuk. [Scientific Computation and Functional Programming](#). Computing in Science and Engineering 1(3):64-72, 1999.
- ▶ Bryan O’Sullivan, John Goerzen, Don Stewart. [Real World Haskell](#). O’Reilly, 2008.
- ▶ [Haskell in Industry and Open Source](#):
www.haskell.org/haskellwiki/Haskell_in_industry

A Plea for Functional Programming

...even though the titles of:

- ▶ Philip Wadler. [Why no one uses Functional Languages](#). ACM SIGPLAN Notices 33(8):23-27, 1998.
- ▶ Philip Wadler. [An angry half-dozen](#). ACM SIGPLAN Notices 33(2):25-30, 1998.

might suggest the opposite, [Philip Wadler's](#) lamentation is only an apparent one and much more an impassioned

- ▶ [plea for functional programming](#)

in the real world summarizing a number of [very general obstacles](#) preventing good or even superior ideas also in the field of programming to make their way into mainstream practices easily and fast.

More Pleas for Functional Programming

...in and for the [real world](#):

- ▶ Konrad Hinsien. [The Promises of Functional Programming](#). Computing in Science and Engineering 11(4): 86-90, 2009.
- ▶ Konstantin Läufer, Geoge K. Thiruvathukal. [The Promises of Typed, Pure, and Lazy Functional Programming: Part II](#). Computing in Science and Engineering 11(5): 68-75, 2009.
- ▶ Yaron Minsky. [OCaml for the Masses](#). Communications of the ACM, 54(11):53-58, 2011.
- ▶ Neal Ford. [Functional Thinking: Why Functional Programming is on the Rise](#). IBM developerWorks, 10 pages, 2013.

and quite recently:

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

Recall Edsger W. Dijkstra's Prediction

The clarity and economy of expression that the language of functional programming permits is often very impressive, and, but for human inertia, functional programming can be expected to have a brilliant future.)*

Edsger W. Dijkstra (11.5.1930-6.8.2002)
1972 Recipient of the ACM Turing Award

*) Quote from: Introducing a course on calculi. Announcement of a lecture course at the University of Texas at Austin, 1995.

In the Words of Simon Peyton Jones

*When the limestone of
imperative programming has worn away,
the granite of functional programming
will be revealed underneath.*

Simon Peyton Jones

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

G 1650/10

In the Words of John Carmack

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

John Carmack

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1651 / 19

Chapter 20.4

References, Further Reading

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10





Chap. 11

Chap. 12





Chap. 13

1652/10





Chapter 20: Further Reading (1)

-  John W. Backus. *Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs*. Communications of the ACM 21(8):613-641, 1978.
-  Urban Boquist. *Code Optimization Techniques for Lazy Functional Languages*. PhD thesis, Chalmers University of Technology, 1999.
-  Atze Dijkstra, Jeroen Fokker, S. Doaitse Swierstra. *The Architecture of the Utrecht Haskell Compiler*. In Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell (Haskell 2009), 93-104, 2009.
-  Atze Dijkstra, Jeroen Fokker, S. Doaitse Swierstra. *UHC Utrecht Haskell Compiler*, 2009. www.cs.uu.nl/wiki/UHC.





Chapter 20: Further Reading (2)

-  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>
-  Konrad Hinsen. *The Promises of Functional Programming*. *Computing in Science and Engineering* 11(4):86-90, 2009.
-  Paul Hudak. *Conception, Evolution and Applications of Functional Programming Languages*. *Communications of the ACM* 21(3):359-411, 1989.
-  John Hughes. *Why Functional Programming Matters*. *Computer Journal* 32(2):98-107, 1989.




Chapter 20: Further Reading (3)

-  John Hughes. *Why Functional Programming Matters*. Invited Keynote, Bangalore, 2016.
<https://www.youtube.com/watch?v=XrNdvWqxBvA>
-  Nigel W.O. Hutchison, Ute Neuhaus, Manfred Schmidt-Schauß, Cordelia V. Hall. *Natural Expert: A Commercial Functional Programming Environment*. *Journal of Functional Programming* 7(2):163-182, 1997.
-  Jerzy Karczmarczuk. *Scientific Computation and Functional Programming*. *Computing in Science and Engineering* 1(3):64-72, 1999.
-  Konstantin Läufer, George K. Thiruvathukal. *The Promises of Typed, Pure, and Lazy Functional Programming: Part II*. *Computing in Science and Engineering* 11(5):68-75, 2009.





Chapter 20: Further Reading (4)

-  Greg Michaelson. *Programming Paradigms, Turing Completeness and Computational Thinking*. The Art, Science, and Engineering of Programming 4(3), Article 4, 21 pages, 2020.
-  Yaron Minsky. *OCaml for the Masses*. Communications of the ACM 54(11):53-58, 2011.
-  Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Chapter 25, Profiling and Optimization)
-  Neil Savage. *Using Functions for Easier Programming*. Communications of the ACM 61(5):29-30, 2018.

Chapter 20: Further Reading (4)

-  Curt J. Simpson. *Experience Report: Haskell in the “Real World”: Writing a Commercial Application in a Lazy Functional Language*. In Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP 2009), 185-190, 2009.
-  David A. Turner. *Total Functional Programming*. Journal of Universal Computer Science 10(7):751-768, 2004.
-  Marcos Viera, S. Doaitse Swierstra, Wouter S. Swierstra. *Attribute Grammars fly First Class: How do we do Aspect Oriented Programming in Haskell*. In Proceedings of the 14th ACM SIGPLAN Conference on Functional Programming (ICFP 2009), 245-256, 2009.

Chapter 20: Further Reading (5)

-  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.
-  Philip Wadler. *An angry half-dozen*. ACM SIGPLAN Notices 33(2):25-30, 1998.
-  Philip Wadler. *Why no one uses Functional Languages*. ACM SIGPLAN Notices 33(8):23-27, 1998.
-  'Haskell community.' *Haskell in Industry and Open Source*. www.haskell.org/haskellwiki/Haskell_in_industry

References

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1659/10

Reading

...for deepened and independent studies.

- ▶ I Textbooks
- ▶ II Monographs
- ▶ III Volumes
- ▶ IV Articles
- ▶ V Haskell 98 – Language Definition
- ▶ V The History of Haskell

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10






Chap. 11

Chap. 12

Chap. 13

G 1660/19

I Textbooks (1)

-  Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
-  Martin Aigner, Günter M. Ziegler. *Proofs from the Book*. Springer-V., 4th edition, 2010.
-  Ronald C. Arkin. *Behavior-Based Robotics*. MIT Press, 1998.
-  Joe Armstrong, Robert Virding, Claes Wikstrom, Mike Williams. *Concurrent Programming in Erlang*. Prentice Hall, 2nd edition, 1996.
-  André Arnold, Irène Guessarian. *Mathematics for Computer Science*. Prentice Hall, 1996.






I Textbooks (2)

-  Manoochehr Azmoodeh. *Abstract Data Types and Algorithms*. Macmillan Education, 1988.
-  Hendrik Pieter Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Revised edition, North Holland, 1984.
-  Richard E. Bellman. *Dynamic Programming*. Princeton University Press, 1957.
-  Richard E. Bellman, Stuart E. Dreyfus. *Applied Dynamic Programming*. Princeton University Press, 1957.
-  Rudolf Berghammer. *Ordnungen, Verbände und Relationen mit Anwendungen*. Springer-V., 2012.





I Textbooks (3)

-  Rudolf Berghammer. *Ordnungen und Verbände: Grundlagen, Vorgehensweisen und Anwendungen*. Springer-V., 2013.
-  Richard Bird. *Introduction to Functional Programming using Haskell*. 2nd edition, Prentice Hall, 1998.
-  Richard Bird. *Thinking Functionally with Haskell*. Cambridge University Press, 2015.
-  Richard Bird, Philip Wadler. *An Introduction to Functional Programming*. Prentice Hall, 1988.
-  William H. Burge. *Recursive Programming Techniques*. Addison-Wesley, 1975.
-  Garret Birkhoff. *Lattice Theory*. American Mathematical Society, 3rd edition, 1967.

I Textbooks (4)

-  Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011.
-  Manuel M.T. Chakravarty, Gabriele Keller. *Einführung in die Programmierung mit Haskell*. Pearson Studium, 2004.
-  Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.
-  Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*. 2nd edition, MIT Press, 2001.
-  Peter Crawley, Robert P. Dilworth. *Algebraic Theory of Lattices*. Prentice Hall, 1973.

I Textbooks (5)

-  H. Conrad Cunningham. *Notes on Functional Programming with Haskell*. Course Notes, University of Mississippi, 2007. citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.114.2822&rep=rep1&type=pdf
-  Brian A. Davey, Hilary A. Priestley. *Introduction to Lattices and Order*. Cambridge Mathematical Textbooks, Cambridge University Press, 2nd edition, 2002.
-  Antonie J.T. Davie. *An Introduction to Functional Programming Systems using Haskell*. Cambridge University Press, 1992.
-  Ernst-Erich Doberkat. *Haskell: Eine Einführung für Objektorientierte*. Oldenbourg Verlag, 2012.






I Textbooks (6)

-  Kees Doets, Jan van Eijck. *The Haskell Road to Logic, Maths and Programming*. Texts in Computing, Vol. 4, King's College, UK, 2004.
-  Jan van Eijck, Christina Unger. *Computational Semantics with Functional Programming*. Cambridge University Press, 2010.
-  Marcel Erné. *Einführung in die Ordnungstheorie*. Bibliographisches Institut, 2. Auflage, 1982.
-  Martin Erwig. *Grundlagen funktionaler Programmierung*. Oldenbourg Verlag, 1999.
-  Anthony J. Field, Peter G. Harrison. *Functional Programming*. Addison-Wesley, 1988.







I Textbooks (7)

-  Helmuth Gericke. *Theorie der Verbände*. Bibliographisches Institut, 2. Auflage, 1967.
-  George Grätzer. *General Lattice Theory*. Birkhäuser, 2nd edition, 1998.
-  George Grätzer. *Lattice Theory: Foundation*. Birkhäuser, 2011.
-  Max Hailperin, Barbara Kaiser, Karl Knight. *Concrete Abstractions – An Introduction to Computer Science using Scheme*. Brooks/Cole Publishing Company, 1999.
-  Paul R. Halmos. *Naive Set Theory*. Springer-V., Reprint, 2001.







I Textbooks (8)

-  Chris Hankin. *An Introduction to Lambda Calculi for Computer Scientists*. King's College London Publications, 2004.
-  Rachel Harrison. *Abstract Data Types in Standard ML*. J. Wiley, 1993.
-  Peter Henderson. *Functional Programming: Application and Implementation*. Prentice Hall, 1980.
-  Hans Hermes. *Einführung in die Verbandstheorie*. Springer-V., 2. Auflage, 1967.
-  Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000.

I Textbooks (9)

-  Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2nd edition, 2016.
-  Richard Johnsonbaugh. *Discrete Mathematics*. Pearson, 7th edition, 2009.
-  Mark P. Jones, Alastair Reid et al. (Eds.). *The Hugs98 User Manual*. www.haskell.org/hugs
-  Stephen C. Kleene. *Introduction to Metamathematics*. North Holland, 1952. (Reprint, North Holland, 1980)
-  Jon Kleinberg, Éva Tardos. *Algorithm Design*. Addison-Wesley/Pearson, 2006.
-  Derrick G. Kourie, Bruce W. Watson. *The Correctness-by-Construction Approach to Programming*. Springer-V., 2012.

I Textbooks (10)

-  Wolfram-Manfred Lippe. *Funktionale und Applikative Programmierung*. eXamen.press, 2009.
-  Miran Lipovača. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, 2011.
-  Seymour Lipschutz. *Set Theory and Related Topics*. McGraw Hill Schaum's Outline Series, 2nd edition, 1998.
-  Bruce J. MacLennan. *Functional Programming: Practice and Theory*. Addison-Wesley, 1990.
-  David Makinson. *Sets, Logic and Maths for Computing*. Springer-V., 2008.
-  Kimbal Marriott, Peter J. Stuckey. *Programming with Constraints*. MIT Press, 1998.

I Textbooks (11)

-  Simon Marlow. *Parallel and Concurrent Programming in Haskell*. O'Reilley, 2013.
-  Greg Michaelson. *An Introduction to Functional Programming through Lambda Calculus*. Dover Publications, 2nd edition, 2011.
-  Robin Milner. *Communications and Concurrency*. Prentice Hall, 1989.
-  Robin Milner. *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge University Press, 1999.
-  Hanne Riis Nielson, Flemming Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, 1992.
-  Hanne Riis Nielson, Flemming Nielson. *Semantics with Applications: An Appetizer*. Springer-V., 2007.





I Textbooks (12)

-  Flemming Nielson, Hanne Riis Nielson, Chris Hankin. *Principles of Program Analysis*. 2nd edition, Springer-V., 2005.
-  Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.
-  John O'Donnell, Cordelia Hall, Rex Page. *Discrete Mathematics Using a Computer*. Springer-V., 2nd edition, 2006.
-  Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008.
-  Lawrence C. Paulson. *Logic and Computation – Interactive Proof with Cambridge LCF*. Cambridge University Press, 1987.
-  Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003.




I Textbooks (13)

-  Peter Pepper, Petra Hofstedt. *Funktionale Programmierung: Sprachdesign und Programmierertechnik*. Springer-V., 2006.
-  Simon Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.
-  Simon Peyton Jones, David Lester. *Implementing Functional Languages: A Tutorial*. Prentice Hall, 1992.
-  Tomas Petricek, Jon Skeet. *Real World Functional Programming: With Examples in F# and C#*. Manning Publications Co., 2009.
-  Fethi A. Rabhi, Guy Lapalme. *Algorithms – A Functional Programming Approach*. Addison-Wesley, 1999.






I Textbooks (14)

-  Chris Reade. *Elements of Functional Programming*. Addison-Wesley, 1989.
-  George E. Revesz. *Lambda-Calculus, Combinators and Functional Programming*. Cambridge University Press, 1988.
-  Steven Roman. *Lattices and Ordered Sets*. Springer-V., 2008.
-  Gunter Saake, Kai-Uwe Sattler. *Algorithmen und Datenstrukturen – Eine Einführung mit Java*. dpunkt.verlag, 4. überarbeitete Auflage, 2010.
-  Davide Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2011.





I Textbooks (15)

-  Robert Sedgewick. *Algorithmen*. Addison-Wesley/Pearson, 2. Auflage, 2002.
-  Steven S. Skiena. *The Algorithm Design Manual*. Springer-V., 1998.
-  Bernhard Steffen, Oliver Rüthing, Malte Isberner. *Grundlagen der höheren Informatik: Induktives Vorgehen*. Springer-V., 2014.
-  Bernhard Steffen, Oliver Rüthing, Michael Huth. *Mathematical Foundations of Advanced Informatics: Inductive Approaches*. Springer-V., 2018.
-  Aaron Stump. *Verified Functional Programming in Agda*. ACM Books Series, No. 9, 2016.

I Textbooks (16)

-  Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley/Pearson, 3rd edition, 2011.
-  Franklyn Turbak, David Gifford with Mark A. Sheldon. *Design Concepts in Programming Languages*. MIT Press, 2008.
-  Daniel J. Velleman. *How to Prove It. A Structured Approach*. Cambridge University Press, 3rd edition, 2019.
-  Mitchell Wand. *Induction, Recursion, and Programming*. Elsevier, 1980.
-  Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.

II Monographs (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. *Pearls of Functional Algorithm Design*. Cambridge University Press, 2011.
-  Urban Boquist. *Code Optimization Techniques for Lazy Functional Languages*. PhD thesis, Chalmers University of Technology, 1999.

II Monographs (2)

-  Andrew D. Gordon. *Functional Programming and Input/Output*. PhD thesis. University of Cambridge, British Computer Society Distinguished Dissertations in Computer Science, Cambridge University Press, 1992.
-  Saunders Mac Lane. *Categories for the Working Mathematician*. Springer-V., 1971 (2nd edition, 1998).
-  Johan Nordlander. *Reactive Objects and Functional Programming*. PhD thesis. Chalmers University of Technology, 1999.
-  Zhanyong Wan. *Functional Reactive Programming for Real-Time Embedded Systems*. PhD thesis. Department of Computer Science, Yale University, December 2002.


III Volumes (1)

-  Roland Backhouse, Roy Crole, Jeremy R. Gibbons (Eds.). *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*. International Summer School and Workshop, Oxford, UK, April 10-14, 2000, Revised Lectures, Springer-V., LNCS 2297, 2002.
-  Jeremy Gibbons, Oege de Moor (Eds.). *The Fun of Programming*. Palgrave MacMillan, 2003.
-  George Grätzer, Friedrich Wehrung (Eds.). *Lattice Theory: Special Topics and Applications, Vol. I*. Birkhäuser, 2014.
-  George Grätzer, Friedrich Wehrung (Eds.). *Lattice Theory: Special Topics and Applications, Vol. II*. Birkhäuser, 2016.





III Volumes (2)

-  Johan Jeuring, Erik Meijer (Eds.). *Advanced Functional Programming*. Springer-V., LNCS 925, 1995.
-  Johan Jeuring, Simon Peyton Jones (Eds.). *Advanced Functional Programming – Revised Lectures*. Springer-V., LNCS Tutorial 2638, 2003.
-  Pieter Koopman, Rinus Plasmeijer, S. Doaitse Swierstra (Eds.). *Advanced Functional Programming – Revised Lectures*. Springer-V., LNCS 5832, 2008.
-  Peter Rechenberg, Gustav Pomberger (Eds.). *Informatik-Handbuch*. Carl Hanser Verlag, 4th edition, 2006.
-  Colin Runciman, David Wakeling (Eds.). *Applications of Functional Programming*. UCL Press, 1995.





III Volumes (3)

-  Davide Sangiorgi, Jan Rutten (Eds.). *Advanced Topics in Bisimulation and Coinduction*. Cambridge Tracts in Theoretical Computer Science, Vol. 52, Cambridge University Press, 2011.
-  S. Doaitse Swierstra, Pedro Rangel Henriques, José Nuno Oliveira (Eds.). *Advanced Functional Programming – Revised Lectures*. Springer-V., LNCS 1608, 1999.





IV Articles (1)

-  Umut A. Acar, Guy E. Blelloch, Robert Harper. *Selective Memoization*. In Conference Record of the 30th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2003), 14-25, 2003.
-  Hassan Ait-Kaci, Roger Nasr. *Integrating Logic and Functional Programming*. *Lisp and Symbolic Computation* 2(1):51-89, 1989.
-  Sergio Antoy, Michael Hanus. *Compiling Multi-Paradigm Declarative Languages into Prolog*. In Proceedings of the International Workshop on Frontiers of Combining Systems (FroCoS 2000), Springer-V., LNCS 1794, 171-185, 2000.
-  Sergio Antoy, Michael Hanus. *Functional Logic Programming*. *Communications of the ACM* 53(4):74-85, 2010.

IV Articles (2)

-  Sergio Antoy, Michael Hanus. *New Functional Logic Design Patterns*. In Proceedings of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011), Springer-V., LNCS 6816, 19-34, 2011.
-  Henry G. Baker. *Shallow Binding Makes Functional Arrays Fast*. ACM SIGPLAN Notices 26(8):145-147, 1991.
-  Falk Bartels. *Generalized Coinduction*. Electronic Notes in Theoretical Computer Science 44:1, 21 pages, 2001.
<http://www.elsevier.nl/locate/entcs/volume44.html>
-  Falk Bartels. *Generalized Coinduction*. Journal of Mathematical Structures in Computer Science 13(2):321-348, 2003.



IV Articles (3)

-  Martin Braun, Oleg Lobachev, Philip W. Trinder: *Arrows for Parallel Computation*. CoRR, <http://arxiv.org/abs/1801.02216>, 2018.
-  Richard Bird. *Algebraic Identities for Program Calculation*. Computer Journal 32(2):122-126, 1989.
-  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>




IV Articles (4)

-  Garret Birkhoff. *Applications of Lattice Algebra*. Mathematical Proceedings of the Cambridge Philosophical Society 30(2):115-122, 1934.
-  James R. Bitner, Edward M. Reingold. *Backtrack Programming Techniques*. Communications of the ACM 18(11):651-656, 1975.
-  Matthias Blume, David McAllester. *Sound and Complete Models of Contracts*. Journal of Functional Programming 16(4-5):375-414, 2006.




IV Articles (5)

-  Ana Bove, Peter Dybjer, Andrés Sicard-Ramírez. *Combining Interactive and Automatic Reasoning in First Order Theories of Functional Programs*. In Proceedings of the 15th International Conference on Foundations of Software Science and Computational Structures (FoSSaCS 2012), Springer-V., LNCS 7213, 104-118, 2012.
-  Bernd Braßel, Michael Hanus, Björn Peemöller, Fabian Reck. *KiCS2: A New Compiler from Curry to Haskell*. In Proceedings of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011), Springer-V., LNCS 6816, 1-18, 2011.

IV Articles (6)

-  F. Warren Burton. *An Efficient Implementation of FIFO Queues*. Information Processing Letters 14(5):205-206, 1982.
-  Magnus Carlsson, Thomas Hallgren. *Fudgets – A Graphical User Interface in a Lazy Functional Language*. In Proceedings of the 6th ACM International Conference on Functional Programming Languages and Computer Architecture (FPCA'93), 321-330, 1993.
-  Manuel M.T. Chakravarty, Yike Guo, Martin Köhler, Hendrik C.R. Lock. *GOFIN: Higher-Order Functions meet Concurrency Constraints*. Science of Computer Programming 30(1-2):157-199 1998.




IV Articles (7)

-  Manuel M.T. Chakravarty, Gabriele Keller. *An Approach to Fast Arrays in Haskell*. In Johan Jeuring, Simon Peyton Jones (Eds.) *Advanced Functional Programming – Revised Lectures*. Springer-V., LNCS Tutorial 2638, 27-58, 2003.
-  Manuel M.T. Chakravarty, Roman Leshchinsky, Simon Peyton Jones, Gabriele Keller, Simon Marlow. *Data Parallel Haskell: A Status Report*. In Proceedings on the Workshop on Declarative Aspects of Multicore Programming (DAMP 2007), ACM, New York, 10-18, 2007.
-  Stephen Chang, Matthias Felleisen. *The Call-by-Need Lambda Calculus, Revisited*. In Proceedings of the 21st European Symposium on Programming (ESOP 2012), Springer-V., LNCS 7211, 128-147, 2012.




IV Articles (8)

-  Roderick Chapman. *Correctness by Construction: A Manifesto for High Integrity Software*. In Proceedings of the 10th Australian Workshop on Safety Critical Systems and Software, Vol. 55, 43-46, 2006.
-  Olaf Chitil. *Pretty Printing with Lazy Dequeues*. In Proceedings of the ACM SIGPLAN 2001 Haskell Workshop (Haskell 2001), Universiteit Utrecht UU-CS-2001-23, 183-201, 2001.
-  Jan Christiansen, Sebastian Fischer. *Easycheck – Test Data for Free*. In Proceedings of the 9th International Symposium on Functional and Logic Programming (FLPS 2008), Springer-V., LNCS 4989, 322-336, 2008.
-  Koen Claessen. *A Poor Man's Concurrency Monad*. Journal of Functional Programming 9(3):313-323, 1999.





IV Articles (9)

-  Koen Claessen, John Hughes. *QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs*. In Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP 2000), 268-279, 2000.
-  Koen Claessen, John Hughes. *Testing Monadic Code with QuickCheck*. In Proceedings ACM of the SIGPLAN 2002 Haskell Workshop (Haskell 2002), 65-77, 2002.
-  Koen Claessen, John Hughes. *Specification-based Testing with QuickCheck*. In Jeremy Gibbons, Oege de Moor (Eds.), *The Fun of Programming*. Palgrave MacMillan, 17-39, 2003.




IV Articles (10)

-  Koen Claessen, Colin Runciman, Olaf Chitil, John Hughes, Malcolm Wallace. *Testing and Tracing Lazy Functional Programs Using Quickcheck and Hat*. In Johan Jeuring, Simon Peyton Jones (Eds.) *Advanced Functional Programming – Revised Lectures*. Springer-V., LNCS Tutorial 2638, 59-99, 2003.
-  Byron Cook, John Launchbury. *Disposable Memo Functions*. In Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming (ICFP'97), 310, 1997 (full paper in Proceedings Haskell'97 workshop).
-  Antony Courtney, Conal Elliot. *Genuinely Functional User Interfaces*. In Proceedings of the 2001 Haskell Workshop (Haskell 2001), September 2001.



IV Articles (11)

-  Werner Damm, Bernhard Josko. *A Sound and Relatively* Complete Hoare-Logic for a Language with Higher Type Procedures*. *Acta Informatica* 20:59-101, 1983.
-  Anne C. Davis. *A Characterization of Complete Lattices*. *Pacific Journal of Mathematics* 5(2):311-319, 1955.
-  Henning Dierks, Michael Schenke. *A Unifying Framework for Correct Program Construction*. In *Proceedings of the 4th International Conference on the Mathematics of Program Construction (MPC'98)*. Springer-V., LNCS 1422, 122-150, 1998.
-  Atze Dijkstra, Jeroen Fokker, S. Doaitse Swierstra. *The Architecture of the Utrecht Haskell Compiler*. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell (Haskell 2009)*, 93-104, 2009.




IV Articles (12)

-  Atze Dijkstra, Jeroen Fokker, S. Doaitse Swierstra. *UHC Utrecht Haskell Compiler*, 2009. www.cs.uu.nl/wiki/UHC
-  Edsger W. Dijkstra. *Go To Statement Considered Harmful*. Letter to the Editor. *Communications of the ACM* 11(3):147-148, 1968.
-  Norbert Eisinger, Tim Geisler, Sven Panne. *Logic Implemented Functionally*. In *Proceedings of the 9th International Symposium on Programming Languages: Implementations, Logics, and Programs (PLILP'97)*, Springer-V., LNCS 1292, 351-368, 1997.




IV Articles (13)

-  Conal Elliot. *Functional Implementations of Continuous Modeled Animation*. In Proceedings of the 10th International Symposium on Principles of Declarative Programming, held jointly with the International Conference on Algebraic and Logic Programming (PLILP/ALP'98), Springer-V., LNCS 1490, 284-299, 1998.
-  Conal Elliot, Paul Hudak. *Functional Reactive Animation*. In Proceedings of the 2nd ACM SIGPLAN 1997 International Conference on Functional Programming (ICFP'97), 263-273, 1997.





IV Articles (14)

-  Kento Emoto, Sebastian Fischer, Zhenjiang Hu. *Generate, Test, and Aggregate: A Calculation-based Framework for Systematic Parallel Programming with MapReduce*. In Proceedings of the 21st European Symposium on Programming (ESOP 2012), Springer-V., LNCS 7211, 254-273, 2012.
-  Jeroen Fokker. *Functional Parsers*. In Johan Jeuring, Erik Meijer (Eds.), *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques*. Springer-V., LNCS 925, 1-23, 1995.
-  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>





IV Articles (15)

-  Daniel P. Friedman, David S. Wise. *CONS should not Evaluate its Arguments*. In Proceedings of the 3rd International Conference on Automata, Languages and Programming, 257-284, 1976.
-  Jeremy Gibbons. *Functional Pearls – An Editor's Perspective*. www.cs.ox.ac.uk/people/jeremy.gibbons/pearls/
-  Andy Gill, Simon Marlow. *Happy – The Parser Generator for Haskell*. University of Glasgow, 1995.
www.haskell.org/happy





IV Articles (16)

-  Andreas Goerdt. *A Hoare Calculus for Functions defined by Recursion on Higher Types*. In *Proceedings of the Conference on Logic of Programs*, Springer-V, LNCS 193, 106-117, 1985.
-  David Gries. *The Maximum Segment Sum Problem*. In *Formal Development of Programs and Proofs*. Edsger W. Dijkstra (Ed.), Addison-Wesley (UT Year of Programming Series), 43-45, 1990.
-  Klaus E. Grue. *Arrays in Pure Functional Programming Languages*. *International Journal on Lisp and Symbolic Computation* 2:105-113, Kluwer Academic Publishers, 1989.
-  John V. Guttag. *Abstract Data Types and the Development of Data Structures*. *Communications of the ACM* 20(6):396-404, 1977.





IV Articles (17)

-  John V. Guttag, James J. Horning. *The Algebra Specification of Abstract Data Types*. *Acta Informatica* 10(1):27-52, 1978.
-  John V. Guttag, Ellis Horowitz, David R. Musser. *Abstract Data Types and Software Validation*. *Communications of the ACM* 21(12):1048-1064, 1978.
-  Anthony Hall, Roderick Chapman. *Correctness by Construction: Developing a Commercial Secure System*. *IEEE Software* 19(1):18-25, 2002.
-  Thomas Hallgren, Magnus Carlsson. *Programming with Fudgets*. In Johan Jeuring, Erik Meijer (Eds.), *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques*. Springer-V., LNCS 925, 137-182, 1995.






IV Articles (18)

-  Richard Hamlet. *Random Testing*. In J. Marciniak (Ed.), *Encyclopedia of Software Engineering*, Wiley, 970-978, 1994.
-  Michael Hanus. *The Integration of Functions into Logic Programming: From Theory to Practice*. *Journal of Functional Programming* 19&20:583-628, 1994.
-  Michael Hanus (Ed.). *Curry: An Integrated Functional Logic Language*. Vers. 0.8.2, 2006.
www.curry-language.org/
-  Michael Hanus. *Multi-paradigm Declarative Languages*. In *Proceedings of the 23rd International Conference on Logic Programming (ICLP 2007)*, Springer-V., LNCS 4670, 45-75, 2007.





IV Articles (19)

-  Michael Hanus. *Functional Logic Programming: From Theory to Curry*. In *Programming Logics – Essays in Memory of Harald Ganzinger*. Springer-V., LNCS 7797, 123-168, 2013.
-  Michael Hanus, Sergio Antoy, Bernd Braßel, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, F. Steiner. *PAKCS: The Portland Aachen Kiel Curry System*. 2013. Available at www.informatik.uni-kiel.de/~pakcs
-  Michael Hanus, Herbert Kuchen, Juan Jose Moreno-Navarro. *Curry: A Truly Functional Logic Language*. In *Proceedings of the ILPS'95 Workshop on Visions for the Future of Logic Programming*, 95-107, 1995.
-  David Harel, Assaf Marron, Gera Weiss. *Behavioral Programming*. *Communications of the ACM* 55(7):90-100, 2012.





IV Articles (20)

-  Peter Henderson, James H. Morris. *A Lazy Evaluator*. In Conference Record of the 3rd Annual ACM Symposium on Principles of Programming Languages (POPL'76), 95-103, 1976.
-  Steve Hill. *Combinators for Parsing Expressions*. Journal of Functional Programming 6(3):445-464, 1996.
-  Konrad Hinsén. *The Promises of Functional Programming*. Computing in Science and Engineering 11(4):86-90, 2009.
-  Charles A.R. Hoare. *An Axiomatic Basis for Computer Programming*. Communications of the ACM 12(10):576-580, 1969.
-  Charles A.R. Hoare. *The Ideal of Program Correctness*. The Computer Journal 50(3):254-260, 2007.






IV Articles (21)

-  Charles A.R. Hoare. *Retrospective: An Axiomatic Basis for Computer Programming*. *Communications of the ACM* 52(10):30-32, 2009.
-  Sören Holmström. *PFL: A Functional Language for Parallel Programming*. In *Declarative Programming Workshop*, 114-139, 1983.
-  Paul Hudak. *Arrays, Non-determinism, Side-effects, and Parallelism: A Functional Perspective*. In *Proceedings of a Workshop on Graph Reduction (WGR'86)*, Springer-V., LNCS 279, 312-327, 1986.
-  Paul Hudak, Antony Courtney, Henrik Nilsson, John Peterson. *Arrows, Robots, and Functional Reactive Programming*. In Johan Jeuring, Simon Peyton Jones (Eds.) *Advanced Functional Programming – Revised Lectures*. Springer-V., LNCS Tutorial 2638, 159-187, 2003.




IV Articles (22)

-  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.
-  John Hughes. *An Efficient Implementation of Purely Functional Arrays*. Technical Report, Programming Methodology Group, Chalmers University of Technology, 1985.
-  John Hughes. *Why Functional Programming Matters*. Computer Journal 32(2):98-107, 1989.
-  John Hughes. *The Design of a Pretty-Printer Library*. In Johan Jeuring, Erik Meijer (Eds.), *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques*. Springer-V., LNCS 925, 53-96, 1995.




IV Articles (23)

-  John Hughes. *Generalising Monads to Arrows*. Science of Computer Programming 37:67-111, 2000.
-  Chung-Kil Hur, Georg Neis, Derek Dreyer, Viktor Vafeiadis. *The Power of Parameterization in Coinductive Proofs*. In Conference Record of the 40th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2013), 193-205, 2013.
-  Graham Hutton. *Higher-Order Functions for Parsing*. Journal of Functional Programming 2(3):323-343, 1992.
-  Graham Hutton, Erik Meijer. *Monadic Parser Combinators*. Technical Report NOTTCS-TR-96-4, Dept. of Computer Science, University of Nottingham, 1996.
-  Graham Hutton, Erik Meijer. *Monadic Parsing in Haskell*. Journal of Functional Programming 8(4):437-444, 1998.



IV Articles (24)

-  Nigel W.O. Hutchison, Ute Neuhaus, Manfred Schmidt-Schauß, Cordelia V. Hall. *Natural Expert: A Commercial Functional Programming Environment*. *Journal of Functional Programming* 7(2):163-182, 1997.
-  Bart Jacobs, Jan Rutten. *A Tutorial on (Co)algebras and (Co)induction*. *EATCS Bulletin* 62:222-259, 1997.
-  Joxan Jaffar, Jean-Louis Lassez. *Constraint Logic Programming*. In *Conference Record of the 14th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'87)*, 111-119, 1987.






IV Articles (25)

-  Ranjit Jhala, Rupak Majumdar, Andrey Rybalchenko. *HMC: Verifying Functional Programs using Abstract Interpreters*. In Proceedings of the 23rd International Conference on Computer Aided Verification (CAV 2011), Springer-V., LNCS 6806, 470-485, 2011.
-  Mark P. Jones. *Functional Thinking*. Lecture at the 6th International Summer School on Advanced Functional Programming, Boxmeer, The Netherlands, 2008.
-  Jerzy Karczmarczuk. *Scientific Computation and Functional Programming*. Computing in Science and Engineering 1(3):64-72, 1999.





IV Articles (26)

-  Naoki Kobayashi, Ryosuke Sato, Hiroshi Unno. *Predicate Abstraction and CEGAR for Higher-Order Model Checking*. In Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2011), 222-233, 2011.
-  Pieter W.M. Koopman, Marinus J. Plasmeijer. *Efficient Combinator Parsers*. In Proceedings of the 10th International Workshop on the Implementation of Functional Languages (IFL'98), Selected Papers, Springer-V., LNCS 1595, 120-136, 1999.
-  Konstantin Läufer, George K. Thiruvathukal. *The Promises of Typed, Pure, and Lazy Functional Programming: Part II*. Computing in Science and Engineering 11(5):68-75, 2009.

IV Articles (27)

-  Peter J. Landin. *A Correspondence between ALGOL60 and Church's Lambda-Notation: Part I*. *Communications of the ACM* 8(2):89-101, 1965.
-  Guy Lapalme, Fabrice Lavier. *Using a Functional Language for Parsing and Semantic Processing*. *Computational Intelligence* 9(2):111-131, 1993.
-  John Launchbury, Simon Peyton Jones. *State in Haskell*. *Lisp and Symbolic Computation* 8(4):293-341, 1995.
-  Daan Leijen. *Parsec, a free Monadic Parser Combinator Library for Haskell*, 2003.
legacy.cs.uu.nl/daan/parsec.html
-  Daan Leijen, Erik Meijer. *Parsec: A Practical Parser Library*. *Electronic Notes in Theoretical Computer Science* 41(1), 20 pages, 2001.

IV Articles (28)

-  Marina Lenisa. *From Set-theoretic Coinduction to Coalgebraic Coinduction: Some Results, Some Problems*. *Electronic Notes in Theoretical Computer Science* 19:2-22, 1999.
-  Peng Li, Simon Marlow, Simon Peyton Jones, Andrew Tolmach. *Lightweight Concurrency Primitives for GHC*. In *Proceedings of the ACM SIGPLAN Workshop on Haskell (Haskell 2007)*, 107-118, 2007.
-  John W. Lloyd. *Programming in an Integrated Functional and Logic Language*. *Journal of Functional and Logic Programming* 1999(3), 49 pages, MIT Press, 1999.
-  Hans-Werner Loidl et al. *Comparing Parallel Functional Languages: Programming and Performance*. *Higher-Order and Symbolic Computation* 16(3):203-251, 2003.





IV Articles (29)

-  Rita Loogen, Yolanda Ortega-Mallén, Ricardo Pena-Mari. *Parallel Functional Programming in Eden*. *Journal of Functional Programming* 15(3):431-475, 2005.
-  Francisco J. López-Fraguas, Jaime Sánchez-Hernández. *TOY: A Multi-paradigm Declarative System*. In *Proceedings of the 10th International Conference on Rewriting Techniques and Applications (RTA'99)*, Springer-V., LNCS 1631, 244-247, 1999.
-  George Markowsky. *Chain-complete Posets and Directed Sets with Applications*. *Algebra Universalis* 6(1):53-68, 1976.
-  Lambert Meertens. *Functional Pearl: Calculating the Sieve of Eratosthenes*. *Journal of Functional Programming* 14(6):759-763, 2004.




IV Articles (30)

-  Greg Michaelson. *Programming Paradigms, Turing Completeness and Computational Thinking*. *The Art, Science, and Engineering of Programming* 4(3), Article 4, 21 pages, 2020.
-  Donald Michie. *'Memo' Functions and Machine Learning*. *Nature*, 218:19-22, 1968.
-  Matthew Might, David Darais, Daniel Spiewak. *Parsing with Derivatives – A Functional Pearl*. In *Proceedings of the 16th ACM International Conference on Functional Programming (ICFP 2011)*, 189-195, 2011.
-  Yaron Minsky. *OCaml for the Masses*. *Communications of the ACM* 54(11):53-58, 2011.





IV Articles (31)

-  Neil Mitchell, Colin Runciman. *Not all Patterns, but enough: An Automated Verifier for Partial but Succifient Pattern Matching*. In Proceedings of the 1st ACM SIG-PLAN Symposium on Haskell (Haskell 2008), 49-60, 2008.
-  Eugenio Moggi. *Computational Lambda Calculus and Monads*. In Proceedings of the 4th Annual IEEE Symposium on Logic in Computer Science (LICS'89), 14-23, 1989.
-  Eugenio Moggi. *Notions of Computation and Monads*. Information and Computation 93(1):55-92, 1991.
-  Juan Jose Moreno-Navarro, Mario Rodriguez-Artalejo. *Logic Programming with Functions and Predicates: The Language BABEL*. Journal of Logic Programming 1(3-4):191-223, 1992.





IV Articles (32)

-  Shin-Cheng Mu. *The Maximum Segment Sum is Back: Deriving Algorithms for two Segment Problems with Bounded Lengths*. In Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Program Manipulation (PEPM 2008), 31-39, 2008.
-  Martin Müller, Tobias Müller, Peter Van Roy. *Multiparadigm Programming in Oz*. In Proceedings of the Workshop on Visions for the Future of Logic Programming (ILPS'95), 1995.
-  Flemming Nielson, Hanne Riis Nielson. *Finiteness Conditions for Fixed Point Iteration*. In Proceedings of the 7th ACM Conference on LISP and Functional Programming (LFP'92), 96-108, 1992.




IV Articles (33)

-  Henrik Nilsson, Antony Courtney, John Peterson. *Functional Reactive Programming, Continued*. In Proceedings of the ACM SIGPLAN Workshop on Haskell (Haskell 2002), 51-64, 2002.
-  Chris Okasaki. *Simple and Efficient Purely Functional Queues and Dequeues*. *Journal of Functional Programming* 5(4):583-592, 1995.
-  Matti Nykänen. *A Note on the Genuine Sieve of Eratosthenes*. *Journal of Functional Programming* 21(6):563-572, 2011.
-  Melissa E. O'Neill. *The Genuine Sieve of Eratosthenes*. *Journal of Functional Programming* 19(1):95-106, 2009.

IV Articles (34)

-  Melissa E. O'Neill, F. Warren Burton. *A New Method for Functional Arrays*. *Journal of Functional Languages* 7(5):487-513, 1997.
-  Derek Oppen. *Pretty-printing*. *ACM Transactions on Programming Languages and Systems* 2(4):465-483, 1980.
-  Ross Paterson. *A New Notation for Arrows*. In *Proceedings of the 6th ACM SIGPLAN Conference on Functional Programming (ICFP 2001)*, 229-240, 2001.
-  Ross Paterson. *Arrows and Computation*. In Jeremy Gibbons, Oege de Moor (Eds.), *The Fun of Programming*. Palgrave MacMillan, 201-222, 2003.




IV Articles (35)

-  Izzet Pembeci, Henrik Nilsson, Gregory D. Hager. *Functional Reactive Robotics: An Exercise in Principled Integration of Domain-Specific Languages*. In Proceedings of the 4th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2002), 168-179, 2002.
-  John Peterson, Gregory D. Hager, Paul Hudak. *A Language for Declarative Robotic Programming*. In Proceedings of the IEEE International Conference on Robotics and Automation (ICRA'99), Vol. 2, 1144-1151, 1999.
-  John Peterson, Paul Hudak, Conal Elliot. *Lambda in Motion: Controlling Robots with Haskell*. In Proceedings of the 1st International Workshop on Practical Aspects of Declarative Languages (PADL'99), Springer-V., LNCS 1551, 91-105, 1999.




IV Articles (36)

-  John Peterson, Zhanyong Wan, Paul Hudak, Henrik Nilsson. *Yale FRP User's Manual*. Department of Computer Science, Yale University, January 2001.
www.haskell.org/frp/manual.html
-  Thomas Petricek. *What We Talk about when We Talk about Monads*. *The Art, Science, and Engineering of Programming* 2(3), Article 12, 1-27, 2018.
-  Simon Peyton Jones. *Haskell pretty-printer library*. 1997.
www.haskell.org/libraries/#prettyprinting.
-  Simon Peyton Jones. *Tackling the Awkward Squad: Monadic Input/Output, Concurrency, Exceptions, and Foreign-language Calls in Haskell*. In Tony Hoare, Manfred Broy, Ralf Steinbruggen (Eds.), *Engineering Theories of Software Construction*, IOS Press, 47-96, 2001 (Presented at the 2000 Marktoberdorf Summer School).




IV Articles (37)

-  Simon Peyton Jones. *Haskell 98 Libraries: Arrays*. *Journal of Functional Programming* 13(1):173-178, 2003.
-  Simon Peyton Jones, Jean-Marc Eber, Julian Seward. *Composing Contracts: An Adventure in Financial Engineering*. In *Proceedings of the 5th ACM SIGPLAN Conference on Functional Programming (ICFP 2000)*, 280-292, 2000.
-  Simon Peyton Jones, Andrew Gordon, Sigbjorn Finne. *Concurrent Haskell*. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*, 295-308, 1996.

IV Articles (38)

-  Simon Peyton Jones, Satnam Sing. *A Tutorial on Parallel and Concurrent Programming in Haskell. Advanced Functional Programming – Revised Lectures*. Springer-V., LNCS 5832, 267-305, 2008.
-  Simon Peyton Jones, Philip Wadler. *Imperative Functional Programming*. In Conference Record of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'93), 71-84, 1993.
-  Robert F. Pointon, Philip W. Trinder, Hans-Wolfgang Loidl. *The Design and Implementation of Glasgow Distributed Haskell*. In Proceedings of the 12th 97International Workshop on Implementation of Functional Languages (IFL 2000), Springer-V., LNCS 2011, 53-70, 2000.




IV Articles (39)

-  Fethi A. Rabhi. *Exploiting Parallelism in Functional Languages: A Paradigm Oriented Approach*. In J. R. Davy, P. M. Dew (Eds.), *Abstract Machine Models for Highly Parallel Computers*, Oxford University Press, 118-139, 1995.
-  Uday S. Reddy. *Narrowing as the Operational Semantics of Functional Languages*. In *Proceedings of the IEEE International Symposium on Logic Programming*, 138-151, 1985.
-  Alastair Reid, John Peterson, Gregory D. Hager, Paul Hudak. *Prototyping Real-Time Vision Systems: An Experiment in DSL Design*. In *Proceedings of the 1999 International Conference on Software Engineering (ICSE'99)*, 484-493, 1999.

IV Articles (40)

-  Tillmann Rendel, Klaus Ostermann. *Invertible Syntax Descriptions: Unifying Parsing and Pretty Printing*. In Proceedings of the 3rd ACM Haskell Symposium on Haskell (Haskell 2010), 1-12, 2010.
-  Colin Runciman. *Lazy Wheel Sieves and Spirals of Primes*. Journal of Functional Programming 7(2):219-225, 1997.
-  Colin Runciman, Matthew Naylor, Fredrik Lindblad. *Small-Check and Lazy SmallCheck*. In Proceedings of the ACM SIGPLAN 2008 Workshop on Haskell (Haskell 2008), 37-48, 2008. <http://hackage.haskell.org>

IV Articles (41)

-  Jan Rutten. *Behavioural Differential Equations: A Coinductive Calculus of Streams, Automata, and Power Series*. *Theoretical Computer Science* 308:1-53, 2003.
-  Chris Sadler, Susan Eisenbach. *Why Functional Programming?* In *Functional Programming: Languages, Tools and Architectures*. Susan Eisenbach (Ed.), Ellis Horwood, 7-8, 1987.
-  Neil Savage. *Using Functions for Easier Programming*. *Communications of the ACM* 61(5):29-30, 2018.




IV Articles (42)

-  Tom Schrijvers, Peter J. Stuckey, Philip Wadler. *Monadic Constraint Programming*. *Journal of Functional Programming* 19(6):663-697, 2009.
-  Silvija Seres, Michael Spivey. *Embedding Prolog in Haskell*. In *Proceedings of the 1999 Haskell Workshop (Haskell'99)*, Technical Report UU-CS-1999-28, Department of Computer Science, University of Utrecht, 25-38, 1999.
-  Curt J. Simpson. *Experience Report: Haskell in the "Real World": Writing a Commercial Application in a Lazy Functional Language*. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP 2009)*, 185-190, 2009.

IV Articles (43)

-  Zoltan Somogyi, Fergus Henderson, Thomas Conway. *The Execution Algorithm of Mercury: An Efficient Purely Declarative Logic Programming Language*. *Journal of Logic Programming* 29(1-3):17-64, 1996.
-  Zoltan Somogyi, Fergus J. Henderson, Thomas C. Conway. *Mercury: An Efficient Purely Declarative Logic Programming Language*. In *Proceedings of the 18th Australasian Computer Science Conference*, 499-512, 1995.
-  William Sonnex, Sophia Drossopoulou, Susan Eisenbach. *Zeno: An Automated Prover for Properties of Recursive Data Structures*. In *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2012)*, Springer-V., LNCS 7214, 407-421, 2012.

IV Articles (44)

-  Jay M. Spitzen, Karl M. Levitt, Lawrence Robinson. *An Example of Hierarchical Design and Proof*. *Communications of the ACM* 21(12):1064-1075, 1978.
-  Michael Spivey. *A Functional Theory of Exceptions*. *Science of Computer Programming* 14(1):25-42, 1990.
-  Michael Spivey, Silvija Seres. *Combinators for Logic Programming*. In Jeremy Gibbons, Oege de Moor (Eds.), *The Fun of Programming*. Palgrave MacMillan, 177-199, 2003.





IV Articles (45)

-  Philippe Suter, Ali Sinan Köksal, Viktor Kuncak. *Satisfiability Modulo Recursive Programs*. In Proceedings of the 18th International Conference on Static Analysis (SAS 2011), Springer-V., LNCS 6887, 298-315, 2011.
-  S. Doaitse Swierstra. *Combinator Parsing: A Short Tutorial*. In Language Engineering and Rigorous Software Development, International LerNet ALFA Summer School 2008, Revised Tutorial Lectures. Springer-V., LNCS 5520, 252-300, 2009.
-  S. Doaitse Swierstra, P. Azero Alcocer. *Fast, Error Correcting Parser Combinators: A Short Tutorial*. In Proceedings SOFSEM'99, Theory and Practice of Informatics, 26th Seminar on Current Trends in Theory and Practice of Informatics, Springer-V., LNCS 1725, 111-129, 1999.




IV Articles (46)

-  S. Doaitse Swierstra, Luc Duponcheel. *Deterministic, Error Correcting Combinator Parsers*. In: *Advanced Functional Programming, Second International Spring School*, Springer-V., LNCS 1129, 184-207, 1996.
-  Wouter S. Swierstra, Thorsten Altenkirch. *Beauty in the Beast: A Functional Semantics for the Awkward Squad*. In *Proceedings of the ACM SIGPLAN Workshop on Haskell (Haskell 2007)*, 25-36, 2007.
-  Alfred Tarski. *A Lattice-theoretical Fixpoint Theorem and its Applications*. *Pacific Journal of Mathematics* 5(2):285-309, 1955.
-  Simon Thompson. *Proof*. In *Research Directions in Parallel Functional Programming*, Kevin Hammond, Greg Michaelson (Eds.), Springer-V., Chapter 4, 93-119, 1999.





IV Articles (47)

-  Philip W. Trinder, Kevin Hammond, Hans-Wolfgang Loidl, Simon Peyton Jones. *Algorithms + Strategy = Parallelism*. Journal of Functional Programming 8(1):23-60, 1998.
-  Philip W. Trinder, Hans-Wolfgang Loidl, Robert F. Pionton. *Parallel and Distributed Haskells*. Journal of Functional Programming 12(4&5):469-510, 2002.
-  David A. Turner. *Total Functional Programming*. Journal of Universal Computer Science 10(7):751-768, 2004.
-  Hiroshi Unno, Tachio Terauchi, Naoki Kobayashi. *Automating Relatively Complete Verification of Higher-Order Functional Programs*. In Conference Record of the 40th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2013), 75-86, 2013.





IV Articles (48)

-  Marcos Viera, S. Doaitse Swierstra, Wouter S. Swierstra. *Attribute Grammars fly First Class: How do we do Aspect Oriented Programming in Haskell*. In Proceedings of the 14th ACM SIGPLAN Conference on Functional Programming (ICFP 2009), 245-256, 2009.
-  Dimitrios Vytiniotis, Simon Peyton Jones, Dan Rosén, Koen Claessen. *HALO: Haskell to Logic through Denotational Semantics*. In Conference Record of the 40th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2013), 431-442, 2013.
-  Philip Wadler. *How to Replace Failure with a List of Successes*. In Proceedings of the 4th International Conference on Functional Programming and Computer Architecture (FPCA'85), Springer-V., LNCS 201, 113-128, 1985.

IV Articles (49)

-  Philip Wadler. *A New Array Operation*. In Proceedings of a Workshop on Graph Reduction (WGR'86), Springer-V., LNCS 279, 328-335, 1986.
-  Philip Wadler. *Comprehending Monads*. Mathematical Structures in Computer Science 2:461-493, 1992.
-  Philip Wadler. *Monads for Functional Programming*. In Johan Jeuring, Erik Meijer (Eds.), *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques*. Springer-V., LNCS 925, 24-52, 1995.
-  Philip Wadler. *How to Declare an Imperative*. In Proceedings of the 1995 International Symposium on Logic Programming (ILPS'95), Invited Presentation, MIT Press, 18-32, 1995.





IV Articles (50)

-  Philip Wadler. *How to Declare an Imperative*. ACM Computing Surveys 29(3):240-263, 1997.
-  Philip Wadler. *An angry half-dozen*. ACM SIGPLAN Notices 33(2): 25-30, 1998.
-  Philip Wadler. *Why no one uses Functional Languages*. ACM SIGPLAN Notices 33(8): 23-27, 1998.
-  Philip Wadler. *A Prettier Printer*. In Jeremy Gibbons, Oege de Moor (Eds.), *The Fun of Programming*. Palgrave MacMillan, 223-243, 2003.


IV Articles (51)

-  Zhanyong Wan, Paul Hudak. *Functional Reactive Programming from First Principles*. In Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI 2000), 242-252, 2000.
-  Zhanyong Wan, Walid Taha, Paul Hudak. *Real-Time FRP*. In Proceedings of the 6th ACM SIGPLAN International Conference on Functional Programming (ICFP 2001), 146-156, 2001.
-  Zhanyong Wan, Walid Taha, Paul Hudak. *Event-Driven FRP*. In Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages (PADL 2002), Springer-V., LNCS 2257, 155-172, 2002.


V Haskell 98 – Language Definition

-  Paul Hudak, Simon Peyton Jones, Philip Wadler (Eds.). *Report on the Programming Language Haskell: Version 1.1*. Technical Report, Yale University and Glasgow University, August 1991.
-  Paul Hudak, Simon Peyton Jones, Philip Wadler (Eds.). *Report on the Programming Language Haskell: A Non-strict Purely Funcional Language (Version 1.2)*. ACM SIGPLAN Notices 27(5):1-164, 1992.
-  Simon Marlow (Ed.). *Haskell 2010 Language Report, 2010*. www.haskell.org/definition/haskell2010.pdf
-  Simon Peyton Jones (Ed.). *Haskell 98: Language and Libraries. The Revised Report*. Cambridge University Press, 2003. www.haskell.org/definitions

VI The History of Haskell

 Simon Peyton Jones. *16 Years of Haskell: A Retrospective on the Occasion of its 15th Anniversary – Wearing the Hair Shirt: A Retrospective on Haskell*. Invited Keynote Presentation at the 30th ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages (POPL 2003), 2003.

`research.microsoft.com/users/simonpj/
papers/haskell-retrospective/`

 Paul Hudak, John Hughes, Simon Peyton Jones, Philip Wadler. *A History of Haskell: Being Lazy with Class*. In Proceedings of the 3rd ACM SIGPLAN 2007 Conference on History of Programming Languages (HOPL III), 12-1 - 12-55, 2007 (ACM Digital Library www.acm.org/dl)

Appendix

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1735/10

Appendix A

Mathematical Foundations

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1736/10

A.1

Relations

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1737/19

Relations

Let M_i , $1 \leq i \leq k$, be sets.

Definition A.1.1 (k -ary Relation)

A (k -ary) relation is a set R of ordered tuples of elements of M_1, \dots, M_k , i.e., $R \subseteq M_1 \times \dots \times M_k$ is a subset of the cartesian product of the sets M_i , $1 \leq i \leq k$.

Examples

- \emptyset is the smallest relation on $M_1 \times \dots \times M_k$.
- $M_1 \times \dots \times M_k$ is the biggest relation on $M_1 \times \dots \times M_k$.

Binary Relations

Let M, N be sets.

Definition A.1.2 (Binary Relation)

A (binary) relation is a set R of ordered pairs of elements of M and N , i.e., R is a subset of the cartesian product of M and N , $R \subseteq M \times N$, called a relation from M to N .

Examples

- \emptyset is the smallest relation from M to N .
- $M \times N$ is the biggest relation from M to N .

Note

- If R is a relation from M to N , it is common to write $m R n$, $R(m, n)$, or $R m n$ instead of $(m, n) \in R$.

Between, On

Definition A.1.3 (Between, On)

A relation R from M to N is called a **relation between M and N** (or a **relation on $M \times N$**).

If M equals N , then R is called a **relation on M** , in symbols: (M, R) .

Domain and Range of a Binary Relation

Definition A.1.4 (Domain and Range)

Let R be a relation from M to N .

The sets

- $dom(R) =_{df} \{m \mid \exists n \in N. (m, n) \in R\}$
- $ran(R) =_{df} \{n \mid \exists m \in M. (m, n) \in R\}$

are called the **domain** and the **range** of R , respectively.

Properties of Relations on a Set M

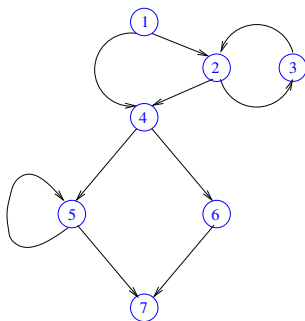
Definition A.1.5 (Properties of Relations on M)

A relation R on a set M is called

1. **reflexive** iff $\forall m \in M. m R m$
2. **irreflexive** iff $\forall m \in M. \neg m R m$
3. **transitive** iff $\forall m, n, p \in M. m R n \wedge n R p \Rightarrow m R p$
4. **intransitive** iff $\forall m, n, p \in M. m R n \wedge n R p \Rightarrow \neg m R p$
5. **symmetric** iff $\forall m, n \in M. m R n \iff n R m$
6. **antisymmetric** iff $\forall m, n \in M. m R n \wedge n R m \Rightarrow m = n$
7. **asymmetric** iff $\forall m, n \in M. m R n \Rightarrow \neg n R m$
8. **linear** iff $\forall m, n \in M. m R n \vee n R m \vee m = n$
9. **total** iff $\forall m, n \in M. m R n \vee n R m$

(Anti-) Example

Let $G = (N, E, s \equiv 1, e \equiv 7)$ be the below (flow) graph, and let R be the relation ' \cdot is linked to \cdot via a (directed) edge' on N of G (e.g., node 4 is linked to node 6 but not vice versa).



The relation R is not reflexive, not irreflexive, not transitive, not intransitive, not symmetric, not antisymmetric, not asymmetric, not linear, and not total.

Equivalence Relation

Let R be a relation on M .

Definition A.1.6 (Equivalence Relation)

R is an **equivalence relation** (or **equivalence**) iff R is reflexive, transitive, and symmetric.

Exercise A.1.7

Let $|$ denote the divisibility relation on the set of natural numbers \mathbb{N}_0 , i.e., the relation ‘ \cdot divides \cdot ’ (w/out remainder), e.g. $5 | 35$.

Prove or disprove: The divisibility relation $|$ on \mathbb{N}_0 is

1. reflexive
2. irreflexive
3. transitive
4. intransitive
5. symmetric
6. antisymmetric
7. asymmetric
8. linear
9. total
10. equivalence (relation)

Proof or counterexample.

A.2

Ordered Sets

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1746/19

A.2.1

Pre-Orders, Partial Orders, and More

Ordered Sets

Let R be a relation on M .

Definition A.2.1.1 (Pre-Order)

R is a **pre-order** (or **quasi-order**) iff R is reflexive and transitive.

Definition A.2.1.2 (Partial Order)

R is a **partial order** (or **poset** or **order**) iff R is reflexive, transitive, and antisymmetric.

Definition A.2.1.3 (Strict Partial Order)

R is a **strict partial order** iff R is asymmetric and transitive.

Examples of Ordered Sets

Pre-order (reflexive, transitive)

- The relation \Rightarrow on logical formulas.

Partial order (reflexive, transitive, antisymmetric)

- The relations $=$, \leq and \geq on \mathbb{N} .
- The relation $m \mid n$ (m is a divisor of n) on \mathbb{N} .

Strict partial order (asymmetric, transitive)

- The relations $<$ and $>$ on \mathbb{N} .
- The relations \subset and \supset on sets.

Equivalence relation (reflexive, transitive, symmetric)

- The relation \iff on logical formulas.
- The relation 'have the same prime number divisors' on \mathbb{N} .
- The relation 'are citizens of the same country' on people.

Note

- An antisymmetric pre-order is a partial order; a symmetric pre-order is an equivalence relation.
- For convenience, also the pair (M, R) is called a pre-order, partial order, and strict partial order, respectively.
- More accurately, we could speak of the pair (M, R) as of a set M which is pre-ordered, partially ordered, and strictly partially ordered by R , respectively.
- Synonymously, we also speak of M as a pre-ordered, partially ordered, and a strictly partially ordered set, respectively, or of M as a set which is equipped with a pre-order, partial order and strict partial order, respectively.
- On any set, the equality relation $=$ is a partial order, called the discrete (partial) order.

The Strict Part of an Order

Let \sqsubseteq be a pre-order (reflexive, transitive) on P .

Definition A.2.1.4 (Strict Part of \sqsubseteq)

The relation \sqsubset on P defined by

$$\forall p, q \in P. p \sqsubset q \iff_{df} p \sqsubseteq q \wedge p \neq q$$

is called the **strict part** of \sqsubseteq .

Corollary A.2.1.5 (Strict Partial Order)

Let (P, \sqsubseteq) be a partial order, let \sqsubset be the strict part of \sqsubseteq .

Then: (P, \sqsubset) is a **strict partial order**.

Useful Results

Let \sqsubset be a strict partial order (asymmetric, transitive) on P .

Lemma A.2.1.6

The relation \sqsubset is irreflexive.

Lemma A.2.1.7

The pair (P, \sqsubseteq) , where \sqsubseteq is defined by

$$\forall p, q \in P. p \sqsubseteq q \iff_{df} p \sqsubset q \vee p = q$$

is a **partial order**.

Induced (or Inherited) Partial Order

Definition A.2.1.8 (Induced Partial Order)

Let (P, \sqsubseteq_P) be a partially ordered set, let $Q \subseteq P$ be a subset of P , and let \sqsubseteq_Q be the relation on Q defined by

$$\forall q, r \in Q. q \sqsubseteq_Q r \iff_{df} q \sqsubseteq_P r$$

Then: \sqsubseteq_Q is called the **induced partial order** on Q (or the **inherited order** from P on Q).

Exercise A.2.1.9

Let $|$ denote the divisibility relation on the set of natural numbers \mathbb{N}_0 , i.e., the relation ' \cdot divides \cdot ' (w/out remainder), e.g. $5 | 35$.

Prove or disprove: The divisibility relation $|$ on \mathbb{N}_0 is a

1. pre-order
2. partial order
3. strict partial order
4. equivalence (relation)

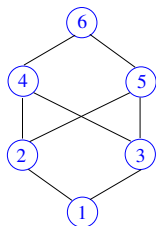
Proof or counterexample.

A.2.2

Hasse Diagrams

Hasse Diagrams

...are a sparse graphical representation of partial orders.



The links of a **Hasse diagram**

- are read from **below** to **above** (lower means smaller).
- represent the relation R of ' \cdot is an immediate predecessor of \cdot ' defined by

$$p R q \iff_{df} p \sqsubset q \wedge \nexists r \in P. p \sqsubset r \sqsubset q$$

of a partial order (P, \sqsubseteq) , where \sqsubset is the strict part of \sqsubseteq .

Reading Hasse Diagrams

The **Hasse diagram** representation of a **partial order**

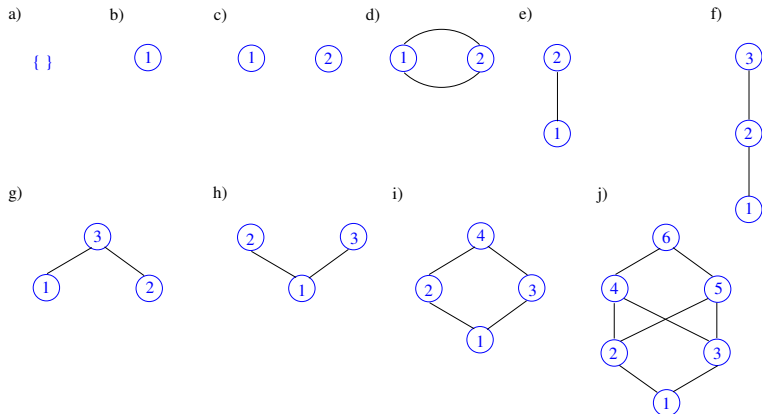
- omits links which express reflexive and transitive relations explicitly
- focuses on the 'immediate predecessor' relation.

The representation of a **partial order** by its **Hasse diagram**

- is sparse and thus economical (in the number of links).
- while preserving all relevant information of the partial order it represents:
 - $p \sqsubseteq q \wedge p = q$ (**reflexivity**): trivially represented (just without an explicit link)
 - $p \sqsubseteq q \wedge p \neq q$ (**transitivity**): represented by ascending paths (with at least one link) from p to q .

Exercise A.2.2.1

Which of the below diagrams are Hasse diagrams representing a partial order?



Exercise A.2.2.2

Let $|$ denote the divisibility relation on the set of natural numbers \mathbb{N}_0 , i.e., the relation ‘ \cdot divides \cdot ’ (w/out remainder), e.g. $5 | 35$.

Draw an expressive section of the [Hasse diagram](#) of the divisibility relation $|$ on \mathbb{N}_0 .

A.2.3

Bounds and Extremal Elements

Bounds in Pre-Orders

Definition A.2.3.1 (Bounds in Pre-Orders)

Let (Q, \sqsubseteq) be a pre-order, let $q \in Q$ and $Q' \subseteq Q$.

q is called a

1. **lower bound** of Q' , in signs: $q \sqsubseteq Q'$, if $\forall q' \in Q'. q \sqsubseteq q'$
2. **upper bound** of Q' , in signs: $Q' \sqsubseteq q$, if $\forall q' \in Q'. q' \sqsubseteq q$
3. **greatest lower bound (glb)** (or **infimum**) of Q' , in signs: $\sqcap Q'$, if q is a lower bound of Q' and for every other lower bound \hat{q} of Q' holds: $\hat{q} \sqsubseteq q$.
4. **least upper bound (lub)** (or **supremum**) of Q' , in signs: $\sqcup Q'$, if q is an upper bound of Q' and for every other upper bound \hat{q} of Q' holds: $q \sqsubseteq \hat{q}$.

Extremal Elements in Pre-Orders

Definition A.2.3.2 (Extremal Elements in Pre-Ord's)

Let (Q, \sqsubseteq) be a pre-order, let \sqsubset be the strict part of \sqsubseteq , and let $Q' \subseteq Q$ and $q \in Q'$.

q is called a

1. **minimal element** of Q' , if there is no $q' \in Q'$ with $q' \sqsubset q$.
2. **maximal element** of Q' , if there is no $q' \in Q'$ with $q \sqsubset q'$.
3. **least (or minimum) element** of Q' , if $q \sqsubseteq Q'$.
4. **greatest (or maximum) element** of Q' , if $Q' \sqsubseteq q$.

Note: Least and greatest elements of Q itself are usually denoted by \perp and \top (**bottom**, **top** (in German: **Tief**, **Hoch**)), respectively, if they exist. **Least (greatest)** elements of Q are always **minimal (maximal)** elements of Q .

Existence and Uniqueness

...of **bounds** and **extremal elements** in **partially ordered sets**.

Let (P, \sqsubseteq) be a partial order, and let $Q \subseteq P$ be a subset of P .

Lemma A.2.3.3 (lub/glb: Unique if Existent)

Least upper bounds, greatest lower bounds, least elements, and greatest elements in Q are **unique**, if they exist.

Lemma A.2.3.4 (Minimal/Maximal El.: Not Unique)

Minimal and maximal elements in Q are usually **not unique**.

Note: Lemma A.2.3.3 suggests considering \sqcup and \sqcap partial maps $\sqcup, \sqcap : \mathcal{P}(P) \rightarrow P$ from the powerset $\mathcal{P}(P)$ of P to P .

Lemma A.2.3.3 does not hold for pre-orders.

Characterization of Least, Greatest Elements

...in terms of **infima** and **suprema** of sets.

Let (P, \sqsubseteq) be a partial order.

Lemma A.2.3.5 (Characterization of \perp and \top)

The **least element** \perp and the **greatest element** \top of P are given by the **supremum** and the **infimum** of the **empty set**, and the **infimum** and the **supremum** of P , respectively, i.e.,

$$\perp = \bigsqcup \emptyset = \bigsqcap P \quad \text{and} \quad \top = \bigsqcap \emptyset = \bigsqcup P$$

if they exist.

Lower and Upper Bound Sets

Considering \sqcup and \sqcap partial functions $\sqcup, \sqcap : \mathcal{P}(P) \rightarrow P$ on the powerset of a partial order (P, \sqsubseteq) suggests introducing two further maps $LB, UB : \mathcal{P}(P) \rightarrow \mathcal{P}(P)$ on $\mathcal{P}(P)$:

Definition A.2.3.6 (Lower and Upper Bound Sets)

Let (P, \sqsubseteq) be a partial order. Then:

$LB, UB : \mathcal{P}(P) \rightarrow \mathcal{P}(P)$ denote two maps, which map a subset $Q \subseteq P$ to the set of its **lower bounds** and **upper bounds**, respectively:

1. $\forall Q \subseteq P. LB(Q) =_{df} \{lb \in P \mid lb \sqsubseteq Q\}$
2. $\forall Q \subseteq P. UB(Q) =_{df} \{ub \in P \mid Q \sqsubseteq ub\}$

Properties of Lower and Upper Bound Sets

Lemma A.2.3.7

Let (P, \sqsubseteq) be a partial order, and let $Q \subseteq P$. Then:

$$\bigsqcup Q = \bigsqcap UB(Q) \quad \text{and} \quad \bigsqcap Q = \bigsqcup LB(Q)$$

if the supremum and the infimum of Q exist.

Lemma A.2.3.8

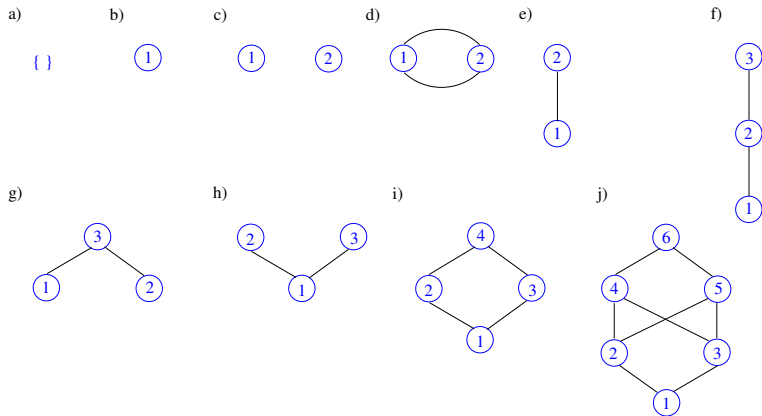
Let (P, \sqsubseteq) be a partial order, and let $Q, Q_1, Q_2 \subseteq P$. Then:

1. $Q_1 \subseteq Q_2 \Rightarrow LB(Q_1) \supseteq LB(Q_2) \wedge UB(Q_1) \supseteq UB(Q_2)$
2. $UB(LB(UB(Q))) = UB(Q)$
3. $LB(UB(LB(Q))) = LB(Q)$

Note: Lemma A.2.3.8(1) shows that LB and UB are antitonic maps (cf. Chapter A.2.7).

Exercise A.2.3.9

Which of the elements of the below diagrams are minimal, maximal, least or greatest?



Exercise A.2.3.10

Let $|$ denote the divisibility relation on the set of natural numbers \mathbb{N}_0 , i.e., the relation ‘ \cdot divides \cdot ’ (w/out remainder), e.g. $5 | 35$.

Write down the sets of elements of \mathbb{N}_0 , which are

1. minimal
2. maximal
3. least
4. greatest

wrt the divisibility relation $|$ on \mathbb{N}_0 .

A.2.4

Noetherian and Artinian Orders

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1769/19

Noetherian and Artinian Orders

Let (P, \sqsubseteq) be a partial order.

Definition A.2.4.1 (Noetherian Order)

(P, \sqsubseteq) is called a **Noetherian order**, if every non-empty subset $\emptyset \neq Q \subseteq P$ contains a minimal element.

Definition A.2.4.2 (Artinian Order)

(P, \sqsubseteq) is called an **Artinian order**, if the dual order (P, \supseteq) of (P, \sqsubseteq) is a Noetherian order.

Lemma A.2.4.3

(P, \sqsubseteq) is an **Artinian order** iff every non-empty subset $\emptyset \neq Q \subseteq P$ contains a maximal element.

Well-founded Orders

Let (P, \sqsubseteq) be a partial order.

Definition A.2.4.4 (Well-founded Order)

(P, \sqsubseteq) is called a **well-founded order**, if (P, \sqsubseteq) is a Noetherian order and totally ordered.

Lemma A.2.4.5

(P, \sqsubseteq) is a **well-founded order** iff every non-empty subset $\emptyset \neq Q \subseteq P$ contains a least element.

Noetherian Induction

Theorem A.2.4.6 (Noetherian Induction)

Let (N, \sqsubseteq) be a Noetherian order, let $N_{min} \subseteq N$ be the set of minimal elements of N , and let $\phi : N \rightarrow \mathbb{B}$ be a predicate on N . Then:

If

1. $\forall n \in N_{min}. \phi(n)$ (Induction base)
2. $\forall n \in N \setminus N_{min}. (\forall m \sqsubset n. \phi(m)) \Rightarrow \phi(n)$ (Induction step)

then:

$$\forall n \in N. \phi(n)$$

A.2.5

Chains

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1773/19

Chains, Antichains

Let (P, \sqsubseteq) be a partial order.

Definition A.2.5.1 (Chain)

A set $C \subseteq P$ is called a **chain**, if the elements of C are totally ordered, i.e., $\forall c_1, c_2 \in C. c_1 \sqsubseteq c_2 \vee c_2 \sqsubseteq c_1$.

Definition A.2.5.2 (Antichain)

A set $C \subseteq P$ is called an **antichain**, if $\forall c_1, c_2 \in C. c_1 \sqsubseteq c_2 \Rightarrow c_1 = c_2$.

Definition A.2.5.3 (Finite, Infinite (Anti-) Chain)

Let $C \subseteq P$ be a chain or an antichain. C is called **finite**, if the number of its elements is finite; C is called **infinite** otherwise.

Note: Any set P may be converted into an antichain by giving it the discrete order: $(P, =)$.

Ascending Chains, Descending Chains

Definition A.2.5.4 (Ascending, Descending Chain)

Let $C \subseteq P$ be a chain. C given in the form of

- $C = \{c_0 \sqsubseteq c_1 \sqsubseteq c_2 \sqsubseteq \dots\}$
- $C = \{c_0 \sqsupseteq c_1 \sqsupseteq c_2 \sqsupseteq \dots\}$

is called an **ascending chain** and **descending chain**, respectively.

Examples of Chains

The set

- $S =_{df} \{n \in \mathbb{N} \mid n \text{ even}\}$ is a chain in \mathbb{N} .
- $S =_{df} \{z \in \mathbb{Z} \mid z \text{ odd}\}$ is a chain in \mathbb{Z} .
- $S =_{df} \{\{k \in \mathbb{N} \mid k < n\} \mid n \in \mathbb{N}\}$ is a chain in the powerset $\mathcal{P}(\mathbb{N})$ of \mathbb{N} .

Note: A chain can always be given in the form of an ascending or descending chain.

- $\{0 \leq 2 \leq 4 \leq 6 \leq \dots\}$: \mathbb{N} as ascending chain.
- $\{\dots \geq 6 \geq 4 \geq 2 \geq 0\}$: \mathbb{N} as descending chain.
- $\{\dots \leq -3 \leq -1 \leq 1 \leq 3 \leq \dots\}$: \mathbb{Z} as ascending chain.
- $\{\dots \geq 3 \geq 1 \geq -1 \geq -3 \geq \dots\}$: \mathbb{Z} as descending chain.
- ...

Eventually Stationary Sequences

Definition A.2.5.5 (Stationary Sequence)

1. An ascending sequence of the form

$$p_0 \sqsubseteq p_1 \sqsubseteq p_2 \sqsubseteq \dots$$

is called **eventually stationary**, if

$$\exists n \in \mathbb{IN}. \forall j \in \mathbb{IN}. p_{n+j} = p_n$$

2. A descending sequence of the form

$$p_0 \sqsupseteq p_1 \sqsupseteq p_2 \sqsupseteq \dots$$

is called **eventually stationary**, if

$$\exists n \in \mathbb{IN}. \forall j \in \mathbb{IN}. p_{n+j} = p_n$$

Chains and Sequences

Lemma A.2.5.6

An ascending or descending sequence of the form

$$p_0 \sqsubseteq p_1 \sqsubseteq p_2 \sqsubseteq \dots \quad \text{or} \quad p_0 \sqsupseteq p_1 \sqsupseteq p_2 \sqsupseteq \dots$$

1. is a finite chain iff it is eventually stationary.
2. is an infinite chain iff it is not eventually stationary.

Note the subtle difference between the notion of **chains** in terms of sets

$$\{p_0 \sqsubseteq p_1 \sqsubseteq p_2 \sqsubseteq \dots\} \quad \text{or} \quad \{p_0 \sqsupseteq p_1 \sqsupseteq p_2 \sqsupseteq \dots\}$$

and in terms of sequences

$$p_0 \sqsubseteq p_1 \sqsubseteq p_2 \sqsubseteq \dots \quad \text{or} \quad p_0 \sqsupseteq p_1 \sqsupseteq p_2 \sqsupseteq \dots$$

Sequences may contain **duplicates**, which would correspond to defining **chains** in terms of **multisets**.

Ascending, Descending Chain Condition

Let (P, \sqsubseteq) be a partial order.

Definition A.2.5.7 (Asc./Desc. Chain Condition)

(P, \sqsubseteq) satisfies the

1. **ascending chain condition** (in German: *aufsteigende Kettenbedingung*), if every ascending chain is eventually stationary, i.e., for every chain $p_1 \sqsubseteq p_2 \sqsubseteq \dots \sqsubseteq p_n \sqsubseteq \dots$ there is an index $m \geq 1$ with $p_m = p_{m+j}$ for all $j \in \mathbb{N}$.
2. **descending chain condition** (in German: *absteigende Kettenbedingung*), if every descending chain is eventually stationary, i.e., for every chain $p_1 \supseteq p_2 \supseteq \dots \supseteq p_n \supseteq \dots$ there is an index $m \geq 1$ with $p_m = p_{m+j}$ for all $j \in \mathbb{N}$.

Chains and Noetherian Orders

Let (P, \sqsubseteq) be a partial order.

Lemma A.2.5.8 (Noetherian Order)

The following propositions are equivalent:

1. (P, \sqsubseteq) is a Noetherian order.
2. (P, \sqsubseteq) satisfies the descending chain condition.
3. Every chain of the form

$$p_0 \sqsupseteq p_1 \sqsupseteq p_2 \sqsupseteq \dots$$

is eventually stationary, i.e.: $\exists n \in \mathbb{N}. \forall j \in \mathbb{N}. p_{n+j} = p_n$.

4. Every chain of the form

$$p_0 \sqsubset p_1 \sqsubset p_2 \sqsubset \dots$$

is finite.

Chains and Artinian Orders

Let (P, \sqsubseteq) be a partial order.

Lemma A.2.5.9 (Artinian Order)

The following propositions are equivalent:

1. (P, \sqsubseteq) is an Artinian order.
2. (P, \sqsubseteq) satisfies the ascending chain condition.
3. Every chain of the form

$$p_0 \sqsubseteq p_1 \sqsubseteq p_2 \sqsubseteq \dots$$

is eventually stationary, i.e.: $\exists n \in \mathbb{N}. \forall j \in \mathbb{N}. p_{n+j} = p_n$.

4. Every chain of the form

$$p_0 \sqsubset p_1 \sqsubset p_2 \sqsubset \dots$$

is finite.

Chains and Noetherian, Artinian Orders

Let (P, \subseteq) be a partial order.

Lemma A.2.5.10 (Noetherian and Artinian Order)

The following propositions are equivalent:

1. (P, \subseteq) is a Noetherian and an Artinian order.
2. (P, \subseteq) satisfies the descending and the ascending chain condition.
3. Every chain $C \subseteq P$ is finite.

A.2.6

Directed Sets

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1783/10

Directed Sets

Let (P, \sqsubseteq) be a partial order, and let $\emptyset \neq D \subseteq P$.

Definition A.2.6.1 (Directed Set)

$D (\neq \emptyset)$ is called a **directed set** (in German: gerichtete Menge), if

$$\forall d, e \in D. \exists f \in D. f \in UB(\{d, e\})$$

i.e., for any two elements d and e there is a common upper bound of d and e in D , i.e., $UB(\{d, e\}) \cap D \neq \emptyset$.

Properties of Directed Sets

Let (P, \sqsubseteq) be a partial order, and let $D \subseteq P$.

Lemma A.2.6.2

D is a **directed set** iff any finite subset $D' \subseteq D$ has an upper bound in D , i.e., $\exists d \in D. d \in UB(D')$, i.e., $UB(D') \cap D \neq \emptyset$.

Lemma A.2.6.3

If D has a greatest element, then D is a directed set.

Properties of Finite Directed Sets

Let (P, \sqsubseteq) be a partial order, and let $D \subseteq P$.

Corollary A.2.6.4

Let D be a *finite directed set*. Then: $\bigsqcup D$ exists $\in D$ and is the greatest element of D .

Proof. Since D a directed set, we have:

$$\exists d \in D. d \in UB(D), \text{ i.e., } UB(D) \cap D \neq \emptyset.$$

This means $D \sqsubseteq d$. The antisymmetry of \sqsubseteq yields that the element enjoying this property is unique. Thus, d is the (unique) greatest element of D given by $\bigsqcup D$, i.e., $d = \bigsqcup D$.

Note: If D is infinite, the proposition of [Corollary A.2.6.4](#) does usually not hold.

Strongly Directed Sets

Let (P, \sqsubseteq) be a partial order with least element \perp , and let $D \subseteq P$.

Definition A.2.6.5 (Strongly Directed Set)

$D \neq \emptyset$ is called a **strongly directed set** (in German: *stark gerichtete Menge*), if

1. $\perp \in D$
2. $\forall d, e \in D. \exists f \in D. f = \bigsqcup\{d, e\}$, i.e., for any two elements d and e the supremum $\bigsqcup\{d, e\}$ of d and e exists in D .

Properties of Strongly Directed Sets

Let (P, \sqsubseteq) be a partial order with least element \perp , and let $D \subseteq P$.

Lemma A.2.6.6

D is a strongly directed set iff every finite subset $D' \subseteq D$ has a supremum in D , i.e., $\exists d \in D. d = \bigsqcup D'$.

Lemma A.2.6.7

Let D be a finite strongly directed set. Then: $\bigsqcup D$ exists $\in D$ and is the greatest element of D .

Note: The proposition of Lemma A.2.6.7 does usually not hold, if D is infinite.

Directed Sets, Strongly Directed Sets, Chains

Let (P, \subseteq) be a partial order with least element \perp .

Lemma A.2.6.8

Let $\emptyset \neq D \subseteq P$ be a non-empty subset of P . Then:

1. D is a directed set, if D is a strongly directed set.
2. D is a strongly directed set, if $\perp \in D$ and D is a chain.

Corollary A.2.6.9

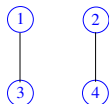
Let $\emptyset \neq D \subseteq P$ be a non-empty subset of P . Then:

$\perp \in D \wedge D$ chain $\Rightarrow D$ strongly directed set $\Rightarrow D$ directed set

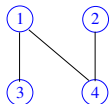
Exercise A.2.6.10

Which of the below **partial orders** are **(strongly) directed sets**?
Which of their **subsets** are **(strongly) directed sets**?

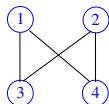
a)



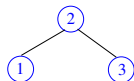
b)



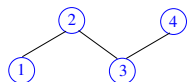
c)



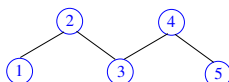
d)



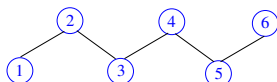
e)



f)



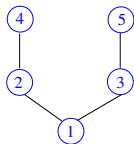
g)



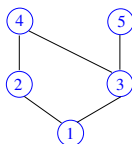
Exercise A.2.6.11

Which of the below **partial orders** are (strongly) directed sets?
Which of their **subsets** are (strongly) directed sets?

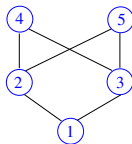
a)



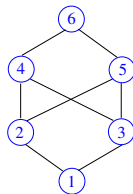
b)



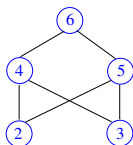
c)



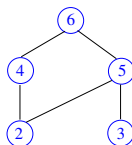
d)



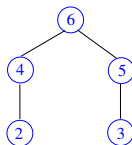
e)



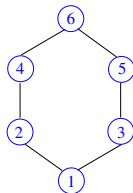
f)



g)



h)



Exercise A.2.6.12

Let $(\mathbb{N}_0, \sqsubseteq)$ be the partial order with $\sqsubseteq =_{df} |$, where $|$ denotes the divisibility relation on the natural numbers \mathbb{N}_0 , i.e., the relation ‘ \cdot divides \cdot ’ (w/out remainder), e.g. $5 | 35$.

Is the set \mathbb{N}_0

1. directed?
2. strongly directed?

What subsets of \mathbb{N}_0 are

1. directed?
2. strongly directed?

Proof or counterexample.

A.2.7

Maps on Partial Orders

Monotonic and Antitonic Maps on POs

Let (C, \sqsubseteq_C) and (D, \sqsubseteq_D) be partial orders, and let $f \in [C \rightarrow D]$ be a map from C to D .

Definition A.2.7.1 (Monotonic Maps on POs)

f is called **monotonic** (or **order preserving**) iff

$$\forall c, c' \in C. c \sqsubseteq_C c' \Rightarrow f(c) \sqsubseteq_D f(c')$$

(Preservation of the ordering of elements)

Definition A.2.7.2 (Antitonic Maps on POs)

f is called **antitonic** (or **order inversing**) iff

$$\forall c, c' \in C. c \sqsubseteq_C c' \Rightarrow f(c') \sqsupseteq_D f(c)$$

(Inversion of the ordering of elements)

Expanding and Contracting Maps on POs

Let (C, \sqsubseteq_C) be a partial order, let $f \in [C \rightarrow C]$ be a map on C , and let $\hat{c} \in C$ be an element of C .

Definition A.2.7.3 (Expanding Maps on POs)

f is called

1. **expanding** (or **inflationary**) for \hat{c} iff $\hat{c} \sqsubseteq f(\hat{c})$
2. **expanding** (or **inflationary**) iff $\forall c \in C. c \sqsubseteq f(c)$

Definition A.2.7.4 (Contracting Maps on POs)

f is called

1. **contracting** (or **deflationary**) for \hat{c} iff $f(\hat{c}) \sqsubseteq \hat{c}$
2. **contracting** (or **deflationary**) iff $\forall c \in C. f(c) \sqsubseteq c$

A.2.8

Order Homomorphisms, Order Isomorphisms

PO Homomorphisms, PO Isomorphisms

Let (P, \sqsubseteq_P) and (R, \sqsubseteq_R) be partial orders, and let $f \in [P \rightarrow R]$ be a map from P to R .

Definition A.2.8.1 (PO Hom. & Isomorphism)

f is called an

1. **order homomorphism** between P and R , if f is monotonic (or order preserving), i.e.,

$$\forall p, q \in P. p \sqsubseteq_P q \Rightarrow f(p) \sqsubseteq_R f(q)$$

2. **order isomorphism** between P and R , if f is a bijective order homomorphism between P and R and the inverse f^{-1} of f is an order homomorphism between R and P .

Definition A.2.8.2 (Order Isomorphic)

(P, \sqsubseteq_P) and (R, \sqsubseteq_R) are called **order isomorphic**, if there is an order isomorphism between P and R .

PO Embeddings

Let (P, \sqsubseteq_P) and (R, \sqsubseteq_R) be partial orders, and let $f \in [P \rightarrow R]$ be a map from P to R .

Definition A.2.8.3 (PO Embedding)

f is called an **order embedding** of P in R iff

$$\forall p, q \in P. p \sqsubseteq_P q \iff f(p) \sqsubseteq_R f(q)$$

Lemma A.2.8.4 (PO Embeddings and Isomorphisms)

f is an order isomorphism between P and R iff f is an order embedding of P in R and f is surjective.

Intuitively: Partial orders, which are order isomorphic, are ‘essentially the same.’

A.3

Complete Partially Ordered Sets

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1799/10

A.3.1

Chain and Directly Complete Partial Orders

Complete Partially Ordered Sets

...or **Complete Partial Orders**:

- ▶ a slightly weaker ordering notion than that of a lattice (cf. [Appendix A.4](#)), which is often more adequate for the modelling of problems in computer science, where full lattice properties are often not required.
- ▶ come in two different flavours as so-called
 - Chain Complete Partial Orders (CCPOs)
 - Directedly Complete Partial Orders (DCPOs)based on the notions of **chains** and **directed sets**, respectively, which, however, are equivalent (cf. [Theorem 3.1.7](#)).

Complete Partial Orders: CCPO View

Definition A.3.1.1 (Chain Complete Partial Order)

A partial order (P, \sqsubseteq) is a

1. **chain complete partial order (pre-CCPO)**, if every non-empty (ascending) chain $\emptyset \neq C \subseteq P$ has a least upper bound $\bigsqcup C$ in P , i.e., $\bigsqcup C \text{ exists} \in P$.
2. **pointed chain complete partial order (CCPO)**, if every (ascending) chain $C \subseteq P$ has a least upper bound $\bigsqcup C$ in P , i.e., $\bigsqcup C \text{ exists} \in P$.

Note: Some authors use **CCPO** and **CCPPO** instead of **pre-CCPO** and **CCPO**, respectively.

Complete Partial Orders: DCPO View

Definition A.3.1.2 (Directedly Complete Partial Ord.)

A partial order (P, \sqsubseteq) is a

1. **directedly complete partial order (pre-DCPO)**, if every directed subset $D \subseteq P$ has a least upper bound $\bigsqcup D$ in P , i.e., $\bigsqcup D \text{ exists } \in P$.
2. **pointed directedly complete partial order (DCPO)**, if it is a pre-DCPO and has a least element \perp .

Note: Some authors use DCPO and DCPPO instead of pre-DCPO and DCPO, respectively.

Remarks on CCPOs and DCPOs

On **CCPOs**:

- A **CCPO** is often called a **domain**.
- 'Ascending chain' and 'chain' can equivalently be used in **Definition A.3.1.1**, since a chain can always be given in ascending order. 'Ascending' chain is just more intuitive.

On **DCPOs**:

- A **directed set** S , in which by definition every finite subset has an upper bound in S , does not need to have a supremum in S , if S is infinite. Therefore, the **DCPO property does not trivially follow** from the directed set property (cf. **Corollary A.2.6.4**).

Existence of Least Elements in CPOs

Lemma A.3.1.3 (Least Elem. Existence in CPOs)

Let (C, \sqsubseteq) be a CPO, i.e., a CCPO or DCPO. Then there is a unique least element in C , denoted by \perp , which is given by the supremum of the empty chain or set, i.e.: $\perp = \bigsqcup \emptyset$.

Corollary A.3.1.4 (Non-Emptiness of CPOs)

Let (C, \sqsubseteq) be a CPO, i.e., a CCPO or DCPO. Then: $C \neq \emptyset$.

Note: Lemma A.3.1.3 does not hold for pre-CPOs, i.e., a pre-CPO (P, \sqsubseteq) does not need to have a least element.

Relating Finite POs, CCPOs and DCPOs

Let P be a finite set, and let \sqsubseteq be a relation on P .

Lemma A.3.1.5 (Fin. POs, pre-CCPOs, pre-DCPOs)

The following propositions are equivalent:

1. (P, \sqsubseteq) is a partial order.
2. (P, \sqsubseteq) is a pre-CCPO.
3. (P, \sqsubseteq) is a pre-DCPO.

Lemma A.3.1.6 (Finite POs, CCPOs, DCPOs)

Let $p \in P$ with $p \sqsubseteq P$. Then the following propositions are equivalent:

1. (P, \sqsubseteq) is a partial order.
2. (P, \sqsubseteq) is a CCPO.
3. (P, \sqsubseteq) is a DCPO.

Equivalence of CCPOs and DCPOs

Theorem A.3.1.7 (Equivalence)

Let (P, \sqsubseteq) be a partial order. Then the following propositions are equivalent:

1. (P, \sqsubseteq) is a CCPO.
2. (P, \sqsubseteq) is a DCPO.

Note: We simply speak of a CPO, if its flavour based on chains (CCPO) or directed sets (DCPO) does not matter; analogously, this applies to pre-CPOs.

Examples of pre-CPOs and CPOs (1)

- ▶ $(\mathcal{P}(\mathbb{N}), \subseteq)$ is a CPO (i.e., a CCPO and a DCPO).

- Least element: \emptyset

- Least upper bound $\bigsqcup C$ of C chain $C \subseteq \mathcal{P}(\mathbb{N})$: $\bigcup_{C' \in C} C'$

- ▶ The set of finite and infinite strings S partially ordered by the prefix relation $\sqsubseteq_{\text{pref}}$ defined by

$$\forall s, s'' \in S. s \sqsubseteq_{\text{pref}} s'' \iff_{df}$$

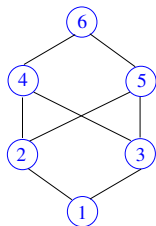
$$s = s'' \vee (s \text{ finite} \wedge \exists s' \in S. s ++ s' = s'')$$

is a CPO.

- ▶ $(\{-n \mid n \in \mathbb{N}\}, \leq)$ is a pre-CPO (i.e., a pre-CCPO and a pre-DCPO) but not a CPO (i.e., not a CCPO and DCPO).

Examples of pre-CPOs and CPOs (2)

- ▶ (\emptyset, \emptyset) is a pre-CPO (i.e., a pre-CCPO and a pre-DCPO) but not a CPO (i.e., not a CCPO and DCPO).
(Both the pre-CCPO (absence of non-empty chains in \emptyset) and the pre-DCPO (\emptyset is the only subset of \emptyset and is not directed by definition) property holds trivially. Note also that $P = \emptyset$ implies $\sqsubseteq = \emptyset \subseteq P \times P$).
- ▶ The partial order (P, \sqsubseteq) given by the below Hasse diagram is a CPO.



Examples of pre-CPOs and CPOs (3)

- ▶ The set of finite and infinite strings S partially ordered by the lexicographical order \sqsubseteq_{lex} defined by

$$\forall s, t \in S. s \sqsubseteq_{lex} t \iff_{df}$$

$$s = t \vee (\exists p \text{ finite}, s', t' \in S. s = p ++ s' \wedge t = p ++ t' \wedge (s' = \varepsilon \vee s'_1 < t'_1))$$

where ε denotes the empty string, $w \downarrow_1$ denotes the first character of a string w , and $<$ the lexicographical ordering on characters, is a CPO (i.e., a CCPO and a DCPO).

(Anti-) Examples of CPOs

▶ (\mathbb{N}, \leq) is not a CPO (i.e., not a CCPO and DCPO).

▶ The set of finite strings S_{fin} partially ordered by the

– prefix relation \sqsubseteq_{pref} defined by

$$\forall s, s' \in S_{fin}. s \sqsubseteq_{pref} s' \iff \exists s'' \in S_{fin}. s ++ s'' = s'$$

is not a CPO (i.e., not a CCPO and DCPO).

– lexicographical order \sqsubseteq_{lex} defined by

$$\begin{aligned} \forall s, t \in S_{fin}. s \sqsubseteq_{lex} t \iff &df \\ &\exists p, s', t' \in S_{fin}. s = p ++ s' \wedge t = p ++ t' \wedge \\ &(s' = \varepsilon \vee s' \downarrow_1 < t' \downarrow_1) \end{aligned}$$

where ε denotes the empty string, $w \downarrow_1$ denotes the first character of a string w , and $<$ the lexicographical ordering on characters, is not a CPO (i.e., not a CCPO and DCPO).

▶ $(\mathcal{P}_{fin}(\mathbb{N}), \subseteq)$ is not a CPO (i.e., not a CCPO and DCPO).

Exercise A.3.1.8

Which of the **partial orders** given by the below **Hasse diagrams** are **(pre-) CCPOs**? Which ones are **(pre-) DCPOs**?

a)

{ }

b)



c)



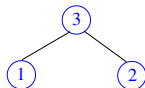
d)



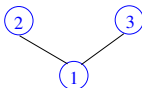
e)



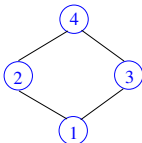
f)



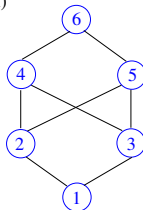
g)



h)



i)



Strongly Directed CPOs: A DCPO Variant

On DCPOs based on Strongly Directed Sets

- Replacing directed sets by strongly directed sets in Definition A.3.1.2 leads to SDCPOs.
- Recalling that strongly directed sets are not empty (cf. Definition A.2.6.5), there is no analogue of pre-DCPOs for strongly directed sets.
- A strongly directed set S , in which by definition every finite subset has a supremum in S , does not need to have a supremum itself in S , if S is infinite. Therefore, the SDCPO property does not trivially follow from the strongly directed property of sets (cf. Corollary A.2.6.4).

Exercise A.3.1.9

Let $(\mathbb{N}_0, \sqsubseteq)$ be the partial order with $\sqsubseteq =_{df} |$, where $|$ denotes the divisibility relation on the natural numbers \mathbb{N}_0 , i.e., the relation ‘ \cdot divides \cdot ’ (w/out remainder), e.g. $5 | 35$.

Prove or disprove: $(\mathbb{N}_0, \sqsubseteq)$ is a

1. pre-CCPO
2. CCPO
3. pre-DCPO
4. DCPO
5. SDCPO

Proof or counterexample.

A.3.2

Maps on Complete Partial Orders

Continuous Maps on CCPOs

Let (C, \sqsubseteq_C) and (D, \sqsubseteq_D) be CCPOs, and let $f \in [C \rightarrow D]$ be a map from C to D .

Definition A.3.2.1 (Continuous Maps on CCPOs)

f is called **continuous** iff f is monotonic and

$$\forall C' \neq \emptyset \text{ chain} \subseteq C. f(\bigsqcup_C C') =_D \bigsqcup_D f(C')$$

(Preservation of least upper bounds)

Note: $\forall S \subseteq C. f(S) =_{df} \{f(s) \mid s \in S\}$

Continuous Maps on DCPOs

Let (D, \sqsubseteq_D) and (E, \sqsubseteq_E) be DCPOs, and let $f \in [D \rightarrow E]$ be a map from D to E .

Definition A.3.2.2 (Continuous Maps on DCPOs)

f is called **continuous** iff

$$\forall D' \neq \emptyset \text{ directed set } \subseteq D. f(D') \text{ directed set } \subseteq E \wedge \\ f(\bigsqcup_D D') =_E \bigsqcup_E f(D')$$

(Preservation of least upper bounds)

Note: $\forall S \subseteq D. f(S) =_{df} \{f(s) \mid s \in S\}$

Characterizing Monotonicity

Let $(C, \sqsubseteq_C), (D, \sqsubseteq_D)$ be CCPOs, let $(E, \sqsubseteq_E), (F, \sqsubseteq_F)$ be DCPOs.

Lemma A.3.2.3 (Characterizing Monotonicity)

1. $f : C \rightarrow D$ is monotonic

iff $\forall C' \neq \emptyset$ *chain* $\subseteq C$.

$$f(C') \text{ chain} \subseteq D \wedge f(\bigsqcup_C C') \sqsupseteq_D \bigsqcup_D f(C')$$

2. $g : E \rightarrow F$ is monotonic

iff $\forall E' \neq \emptyset$ *directed set* $\subseteq E$.

$$g(E') \text{ directed set} \subseteq F \wedge g(\bigsqcup_E E') \sqsupseteq_F \bigsqcup_F g(E')$$

Strict Maps on CCPOs and DCPOs

Let (C, \sqsubseteq_C) , (D, \sqsubseteq_D) be CCPOs with least elements \perp_C and \perp_D , respectively, let (E, \sqsubseteq_E) , (F, \sqsubseteq_F) be DCPOs with least elements \perp_E and \perp_F , respectively, and let $f \in [C \xrightarrow{\text{con}} D]$ and $g \in [E \xrightarrow{\text{con}} F]$ be continuous maps.

Definition A.3.2.4 (Strict Functions on CPOs)

f and g are called **strict**, if the equalities

$$- f(\bigsqcup_C C') =_D \bigsqcup_D f(C'), \quad g(\bigsqcup_E E') =_F \bigsqcup_F g(E')$$

also hold for $C' = \emptyset$ and $E' = \emptyset$, i.e., if the equalities

$$- f(\bigsqcup_C \emptyset) =_C f(\perp_C) =_D \perp_D =_D \bigsqcup \emptyset$$

$$- f(\bigsqcup_E \emptyset) =_E g(\perp_E) =_F \perp_F =_F \bigsqcup \emptyset$$

are valid.

A.3.3

Mechanisms for Constructing Complete Partial Orders

Common CCPO and DCPO Constructions

The following construction principles hold for

- CCPOs
- DCPOs

Therefore, we simply write CPO.

Common CPO Constructions: Flat pre-CPOs

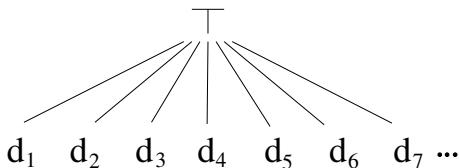
Lemma A.3.3.1 (Flat Pre-CPO Construction)

Let D be a set. Then:

$(D \dot{\cup} \{\top\}, \sqsubseteq_{flat})$ with \sqsubseteq_{flat} defined by

$$\forall d, e \in D \dot{\cup} \{\top\}. d \sqsubseteq_{flat} e \iff_{df} e = \top \vee d = e$$

is a pre-CPO, a so-called flat pre-CPO.



Common CPO Constructions: Flat CPOs

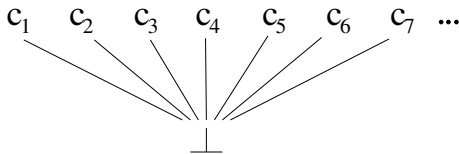
Lemma A.3.3.2 (Flat CPO Construction)

Let C be a set. Then:

$(C \dot{\cup} \{\perp\}, \sqsubseteq_{flat})$ with \sqsubseteq_{flat} defined by

$$\forall c, d \in C \dot{\cup} \{\perp\}. c \sqsubseteq_{flat} d \iff_{df} c = \perp \vee c = d$$

is a CPO, a so-called **flat CPO**.



Common CPO Constructions: Products (1)

Lemma A.3.3.3 (Non-strict Product Construction)

Let $(P_1, \sqsubseteq_1), (P_2, \sqsubseteq_2), \dots, (P_n, \sqsubseteq_n)$ be CPOs. Then:

The **non-strict product** $(\times P_i, \sqsubseteq_{\times})$, where

– $\times P_i =_{df} P_1 \times P_2 \times \dots \times P_n$ is the cartesian product of all P_i , $1 \leq i \leq n$

– \sqsubseteq_{\times} is defined pointwise by

$$\forall (p_1, \dots, p_n), (q_1, \dots, q_n) \in \times P_i.$$

$$(p_1, \dots, p_n) \sqsubseteq_{\times} (q_1, \dots, q_n) \iff_{df}$$

$$\forall i \in \{1, \dots, n\}. p_i \sqsubseteq_i q_i$$

is a CPO.

Common CPO Constructions: Products (2)

Lemma A.3.3.4 (Strict Product Construction)

Let $(P_1, \sqsubseteq_1), (P_2, \sqsubseteq_2), \dots, (P_n, \sqsubseteq_n)$ be CPOs. Then:

The **strict** (or **smash**) **product** $(\bigotimes P_i, \sqsubseteq_{\otimes})$, where

- $\bigotimes P_i =_{df} \times P_i$ is the the cartesian product of all P_i
- $\sqsubseteq_{\otimes} =_{df} \sqsubseteq_{\times}$ defined pointwise with the additional setting
$$(p_1, \dots, p_n) = \perp \iff_{df} \exists i \in \{1, \dots, n\}. p_i = \perp_i$$

is a CPO.

Common CPO Constructions: Sums (1)

Lemma A.3.3.5 (Separated Sum Construction)

Let $(P_1, \sqsubseteq_1), (P_2, \sqsubseteq_2), \dots, (P_n, \sqsubseteq_n)$ be CPOs. Then:

The **separated** (or **direct**) **sum** $(\bigoplus_{\perp} P_i, \sqsubseteq_{\bigoplus_{\perp}})$, where

– $\bigoplus_{\perp} P_i =_{df} P_1 \dot{\cup} P_2 \dot{\cup} \dots \dot{\cup} P_n \dot{\cup} \{\perp\}$ is the disjoint union of all P_i , $1 \leq i \leq n$, and a fresh bottom element \perp

– $\sqsubseteq_{\bigoplus_{\perp}}$ is defined by

$$\forall p, q \in \bigoplus_{\perp} P_i. p \sqsubseteq_{\bigoplus_{\perp}} q \iff_{df}$$

$$p = \perp \vee (\exists i \in \{1, \dots, n\}. p, q \in P_i \wedge p \sqsubseteq_i q)$$

is a CPO.

Common CPO Constructions: Sums (2)

Lemma A.3.3.6 (Coalesced Sum Construction)

Let $(P_1, \sqsubseteq_1), (P_2, \sqsubseteq_2), \dots, (P_n, \sqsubseteq_n)$ be CPOs. Then:

The **coalesced sum** $(\bigoplus_{\vee} P_i, \sqsubseteq_{\bigoplus_{\vee}})$, where

- $\bigoplus_{\vee} P_i =_{df} P_1 \setminus \{\perp_1\} \dot{\cup} P_2 \setminus \{\perp_2\} \dot{\cup} \dots \dot{\cup} P_n \setminus \{\perp_n\} \dot{\cup} \{\perp\}$
is the disjoint union of all P_i , $1 \leq i \leq n$, and a fresh bottom element \perp , which is identified with and replaces the least elements \perp_i of the sets P_i , i.e., $\perp =_{df} \perp_i$, $i \in \{1, \dots, n\}$
- $\sqsubseteq_{\bigoplus_{\vee}}$ is defined by
$$\forall p, q \in \bigoplus_{\vee} P_i. p \sqsubseteq_{\bigoplus_{\vee}} q \iff_{df} p = \perp \vee (\exists i \in \{1, \dots, n\}. p, q \in P_i \wedge p \sqsubseteq_i q)$$

is a CPO.

Common CPO Constructions: Function Space

Lemma A.3.3.7 (Continuous Function Space Con.)

Let (C, \sqsubseteq_C) and (D, \sqsubseteq_D) be pre-CPOs. Then:

The **continuous function space** $([C \xrightarrow{con} D], \sqsubseteq_{cfs})$, where

- $[C \xrightarrow{con} D]$ is the set of continuous maps from C to D
- \sqsubseteq_{cfs} is defined pointwise by
$$\forall f, g \in [C \xrightarrow{con} D]. f \sqsubseteq_{cfs} g \iff \forall c \in C. f(c) \sqsubseteq_D g(c)$$

is a **pre-CPO**. It is a **CPO**, if (D, \sqsubseteq_D) is a **CPO**.

Note: The definition of \sqsubseteq_{cfs} does not make use of C being a pre-CPO. This requirement is only to allow us tailoring the definition to continuous maps.

Applications of CPOs

...in functional programming:

- **Flat CPOs:** Modelling, ordering the values of, e.g., the polymorphic type `Maybe a`.
- **Non-strict Product CPOs:** Modelling, ordering the values of tuple types, approximating the values of streams, modelling non-strict functions.
- **Strict Product CPOs:** Modelling, ordering the values of tuple types, modeling strict functions.
- **Sum CPOs:** Modelling, ordering the values of union types (called `sum types` in `Haskell`).
- **Function-space CPOs:** Defining the (denotational) semantics of programs.

A.4

Lattices

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1830/10

A.4.1

Lattices, Complete Lattices

Lattices and Complete Lattices

Let (P, \sqsubseteq) be a partial order, $P \neq \emptyset$.

Definition A.4.1.1 (Lattice)

(P, \sqsubseteq) is a **lattice** (in German: **Verband**), if every **non-empty finite** subset P' of P has a least upper bound and a greatest lower bound in P .

Definition A.4.1.2 (Complete Lattice)

(P, \sqsubseteq) is a **complete lattice** (in German: **vollständiger Verband**), if **every** subset P' of P has a least upper bound and a greatest lower bound in P .

Note: Lattices and complete lattices are special partial orders.

Properties of Complete Lattices

Lemma A.4.1.3 (Existence of Extremal Elements)

Let (P, \sqsubseteq) be a complete lattice. Then there is

1. a least element in P , denoted by \perp , satisfying:
$$\perp = \bigsqcup \emptyset = \bigsqcap P.$$
2. a greatest element in P , denoted by \top , satisfying:
$$\top = \bigsqcap \emptyset = \bigsqcup P.$$

Lemma A.4.1.4 (Characterization Lemma)

Let (P, \sqsubseteq) be a partial order. Then the following propositions are equivalent:

1. (P, \sqsubseteq) is a complete lattice.
2. Every subset of P has a least upper bound.
3. Every subset of P has a greatest lower bound.

Properties of Finite Lattices

Lemma A.4.1.5 (Finiteness implies Completeness)

If (P, \sqsubseteq) is a finite lattice, then (P, \sqsubseteq) is a complete lattice.

Corollary A.4.1.6 (Finiteness impl. Ex. of ext. Elem.)

If (P, \sqsubseteq) is a finite lattice, then (P, \sqsubseteq) has a least element and a greatest element.

Complete Semi-Lattices

Let (P, \sqsubseteq) be a partial order, $P \neq \emptyset$.

Definition A.4.1.7 (Complete Semi-Lattice)

(P, \sqsubseteq) is a **complete**

1. **join semi-lattice** (in German: **Vereinigungshalbverband**) iff
 $\forall \emptyset \neq S \subseteq P. \bigsqcup S \text{ exists } \in P.$
2. **meet semi-lattice** (in German: **Schnitthalbverband**) iff
 $\forall \emptyset \neq S \subseteq P. \bigsqcap S \text{ exists } \in P.$

Properties of Complete Semi-Lattices (1)

Proposition A.4.1.8 (Extr. Bounds in C. Semi-Lat.)

If (P, \sqsubseteq) is a complete

1. join semi-lattice, then $\bigsqcup P$ exists $\in P$ (whereas $\bigsqcup \emptyset (\hat{=} \perp)$ does usually not exist in P).
2. meet semi-lattice, then $\bigsqcap P$ exists $\in P$ (whereas $\bigsqcap \emptyset (\hat{=} \top)$ does usually not exist in P).

Informally: Least elements need not exist in complete join semi-lattices, greatest elements need not exist in complete meet semi-lattices.

Properties of Complete Semi-Lattices (2)

Lemma A.4.1.9 (Ex. great. El. in C. Join Semi-Lat.)

Let (P, \sqsubseteq) be a complete join semi-lattice. Then:

$\bigsqcup P$ exists $\in P$ and is the (unique) greatest element in P that is usually denoted by \top , i.e., $\top = \bigsqcup P$.

Lemma A.4.1.10 (Ex. least El. in C. Meet Semi-Lat.)

Let (P, \sqsubseteq) be a complete meet semi-lattice. Then:

$\bigsqcap P$ exists $\in P$ and is the (unique) least element in P that is usually denoted by \perp , i.e., $\perp = \bigsqcap P$.

Characterizing Upper and Lower Bounds (1)

...in complete semi-lattices.

Lemma A.4.1.11 (Char. u./l. Bounds in C. Semi-L.)

1. Let (P, \sqsubseteq) be a complete join semi-lattice, and let $Q \subseteq P$ be a subset of P .

If there is a lower bound for Q in P , i.e. if $\{p \in P \mid p \sqsubseteq Q\} \neq \emptyset$, then $\prod Q$ exists $\in P$ satisfying

$$\prod Q = \bigsqcup \{p \in P \mid p \sqsubseteq Q\}$$

2. Let (P, \sqsubseteq) be a complete meet semi-lattice, and let $Q \subseteq P$ be a subset of P .

If there is an upper bound for Q in P , i.e. if $\{p \in P \mid Q \sqsubseteq p\} \neq \emptyset$, then $\bigsqcup Q$ exists $\in P$ satisfying

$$\bigsqcup Q = \prod \{p \in P \mid Q \sqsubseteq p\}$$

Characterizing Upper and Lower Bounds (2)

Lemma A.4.1.12 (L./gr. Elements in C. Semi-L.)

If (P, \sqsubseteq) is a complete

1. join semi-lattice and $\bigsqcup \emptyset$ exists $\in P$, then $\bigsqcup \emptyset$ is the (unique) least element in P , denoted by \perp , i.e., $\perp = \bigsqcup \emptyset$.
2. meet semi-lattice and $\bigsqcap \emptyset$ exists $\in P$, then $\bigsqcap \emptyset$ is the (unique) greatest element in P , denoted by \top , i.e., $\top = \bigsqcap \emptyset$.

Relating Complete Semi-Lattices and Lattices

Lemma A.4.1.13 (Complete Semi-Lattices & Lattices)

If (P, \sqsubseteq) is a complete

1. join semi-lattice and $\bigsqcup \emptyset$ exists $\in P$
2. meet semi-lattice and $\bigsqcap \emptyset$ exists $\in P$

then (P, \sqsubseteq) is a complete lattice.

Exercise A.4.1.14

Prove or refute:

If (P, \sqsubseteq) is a complete lattice, then

1. $(P \setminus \{\perp\}, \sqsubseteq_{\setminus \perp})$ is a complete join semi-lattice.
2. $(P \setminus \{\top\}, \sqsubseteq_{\setminus \top})$ is a complete meet semi-lattice.

where $\sqsubseteq_{\setminus \perp}$ and $\sqsubseteq_{\setminus \top}$ denote the restrictions of \sqsubseteq from P to $P \setminus \{\perp\}$ and $P \setminus \{\top\}$, respectively. Proof or counterexample.

Relating Lattices and Complete Partial Orders

Lemma A.4.1.15 (Complete Lattices and CPOs)

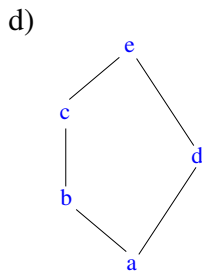
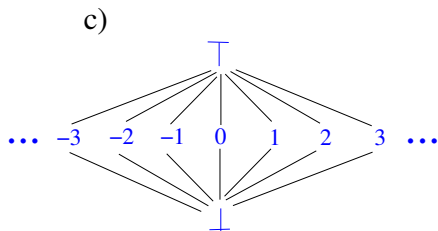
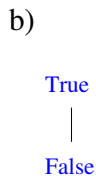
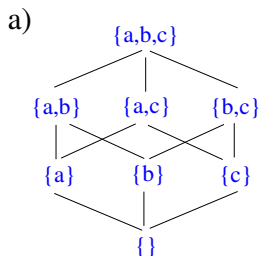
If (P, \sqsubseteq) is a complete lattice, then (P, \sqsubseteq) is a CPO (i.e., a CCPO and DCPO).

Corollary A.4.1.16 (Finite Lattices and CPOs)

If (P, \sqsubseteq) is a finite lattice, then (P, \sqsubseteq) is a CPO (i.e., a CCPO and DCPO).

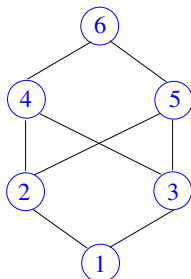
Note: Lemma A.4.1.15 does not hold for lattices.

Examples of Complete Lattices



(Anti-) Examples

- ▶ The partial order (P, \sqsubseteq) given by the below Hasse diagram is not a lattice (whereas it is a CPO).



- ▶ $(\mathcal{P}_{fin}(\mathbb{N}), \subseteq)$ is not a complete lattice (and not a CPO).

Exercise A.4.1.17

Which of the **partial orders** given by the below **Hasse diagrams** are **lattices**? Which ones are **complete lattices**?

a)

{ }

b)



c)



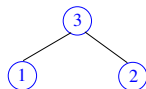
d)



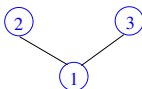
e)



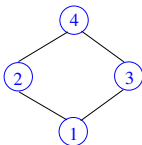
f)



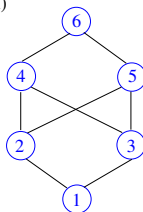
g)



h)



i)



Exercise A.4.1.18

Let $(\mathbb{N}_0, \sqsubseteq)$ be the partial order with $\sqsubseteq =_{df} |$, where $|$ denotes the divisibility relation on the natural numbers \mathbb{N}_0 , i.e., the relation ‘ \cdot divides \cdot ’ (w/out remainder), e.g. $5 | 35$.

Prove or refute: $(\mathbb{N}_0, \sqsubseteq)$ is a

1. lattice
2. complete lattice
3. complete join semi-lattice
4. complete meet semi-lattice

Proof or counterexample.

Summary, Overview

Corollary A.4.1.19

Let $P \neq \emptyset$ be a non-empty set, and \sqsubseteq a relation on P . Then:

(P, \sqsubseteq) finite lattice (L. A.4.1.5) \vee

(P, \sqsubseteq) complete join semi-lattice and

$\bigsqcup \emptyset \text{ exists } \in P$ (L. A.4.1.13(1)) \vee

(P, \sqsubseteq) complete meet semi-lattice and

$\bigsqcap \emptyset \text{ exists } \in P$ (L. A.4.1.13(2))

$\Rightarrow (P, \sqsubseteq)$ complete lattice

(D. A.4.1.2 and

L. A.4.1.14) $\Rightarrow (P, \sqsubseteq)$ lattice and complete partial order

(D. A.4.1.1 and

D. A.3.1.1/2) $\Rightarrow (P, \sqsubseteq)$ partial order

(D. A.2.1.2) $\Rightarrow (P, \sqsubseteq)$ pre-order

Exercise A.4.1.20

Let

$QO, PO, \mathcal{L}, CPO, \mathcal{CL}, \mathcal{FL}, CJSL, CJSL_{\perp}, CMSL, CMSL^{\top}$

denote the sets of all quasi-orders QO , partial orders PO , lattices \mathcal{L} , complete partial orders CPO , complete lattices \mathcal{CL} , finite lattices \mathcal{FL} , complete join semi-lattices without/with least element $CJSL/CJSL_{\perp}$, and meet semi-lattices without/with greatest element $CMSL/CMSL^{\top}$.

1. What further implications or equivalences hold in addition to those listed in [Corollary A.4.1.19](#)? (Proof or counterexample)
2. What inclusions or (set) equalities hold among QO, PO, \mathcal{L} , etc.? (Proof or counterexample)

A.4.2

Distributive, Additive Maps on Lattices

Distributive, Additive Maps on Lattices

Let (P, \sqsubseteq) be a complete lattice, and let $f \in [P \rightarrow P]$ be a map on P .

Definition A.4.2.1 (Distributive, Additive Map)

f is called

1. **distributive** (or \sqcap -continuous) iff

$$\forall \emptyset \neq P' \subseteq P. f(\sqcap P') = \sqcap f(P')$$

(Preservation of greatest lower bounds)

2. **additive** (or \sqcup -continuous) iff

$$\forall \emptyset \neq P' \subseteq P. f(\sqcup P') = \sqcup f(P')$$

(Preservation of least upper bounds)

Note: $\forall S \subseteq P. f(S) =_{df} \{ f(s) \mid s \in S \}$

Characterizing Monotonicity

...in terms of the preservation of greatest lower and least upper bounds:

Lemma A.4.2.2 (Characterizing Monotonicity)

Let (P, \sqsubseteq) be a complete lattice, and let $f \in [P \rightarrow P]$ be a map on P . Then:

$$\begin{aligned} f \text{ is monotonic} &\iff \forall P' \subseteq P. f(\bigsqcap P') \sqsubseteq \bigsqcap f(P') \\ &\iff \forall P' \subseteq P. f(\bigsqcup P') \supseteq \bigsqcup f(P') \end{aligned}$$

Note: $\forall S \subseteq P. f(S) =_{df} \{ f(s) \mid s \in S \}$

Useful Results on Mon., Distr., and Additivity

Let (P, \sqsubseteq) be a complete lattice, and let $f \in [P \rightarrow P]$ be a map on P .

Lemma A.4.2.3

f is monotonic, if f is distributive (or additive).
(i.e., distributivity (or additivity) implies monotonicity)

Note: Distributivity and additivity are independent of each other; none implies the other one.

A.4.3

Lattice Homomorphisms, Lattice Isomorphisms

Lattice Homomorphisms, Lattice Isomorphisms

Let (P, \sqsubseteq_P) and (R, \sqsubseteq_R) be two lattices, and let $f \in [P \rightarrow R]$ be a map from P to R .

Definition A.4.3.1 (Lattice Homomorphism)

f is called a **lattice homomorphism**, if

$$\forall p, q \in P. f(p \sqcup_P q) = f(p) \sqcup_Q f(q) \wedge \\ f(p \sqcap_P q) = f(p) \sqcap_Q f(q)$$

Definition A.4.3.2 (Lattice Isomorphism)

1. f is called a **lattice isomorphism**, if f is a lattice homomorphism and bijective.
2. (P, \sqsubseteq_P) and (R, \sqsubseteq_R) are called **isomorphic**, if there is lattice isomorphism between P and R .

Useful Results (1)

Let (P, \sqsubseteq_P) and (R, \sqsubseteq_R) be two lattices, and let $f \in [P \rightarrow R]$ be a map from P to R .

Lemma A.4.3.3

$$f \in [P \xrightarrow{hom} R] \Rightarrow f \in [P \xrightarrow{mon} R]$$

The reverse implication of [Lemma A.4.3.3](#) does not hold, however, the following weaker relation holds:

Lemma A.4.3.4

$$\begin{aligned} f \in [P \xrightarrow{mon} R] \Rightarrow \\ \forall p, q \in P. f(p \sqcup_P q) \sqsupseteq_Q f(p) \sqcup_Q f(q) \wedge \\ f(p \sqcap_P q) \sqsubseteq_Q f(p) \sqcap_Q f(q) \end{aligned}$$

Useful Results (2)

Let (P, \sqsubseteq_P) and (R, \sqsubseteq_R) be two lattices, and let $f \in [P \rightarrow R]$ be a map from P to R .

Lemma A.4.3.5

$$f \in [P \xrightarrow{iso} R] \Rightarrow f^{-1} \in [R \xrightarrow{iso} P]$$

Lemma A.4.3.6

$$f \in [P \xrightarrow{iso} R] \iff f \in [P \xrightarrow{po-hom} R] \text{ wrt } \sqsubseteq_P \text{ and } \sqsubseteq_Q$$

A.4.4

Modular, Distributive, and Boolean Lattices

Modular Lattices

Let (P, \sqsubseteq) be a lattice with meet operation \sqcap and join operation \sqcup .

Lemma A.4.4.1

$$\forall p, q, r \in P. p \sqsubseteq r \Rightarrow p \sqcup (q \sqcap r) \sqsubseteq (p \sqcup q) \sqcap r$$

Definition A.4.4.2 (Modular Lattice)

(P, \sqsubseteq) is called **modular**, if

$$\forall p, q, r \in P. p \sqsubseteq r \Rightarrow p \sqcup (q \sqcap r) = (p \sqcup q) \sqcap r$$

Characterizing Modular Lattices

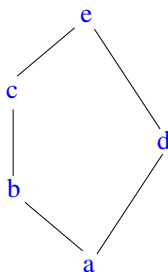
Theorem A.4.4.3 (Characterizing Modular Lattices)

A lattice (P, \sqsubseteq) is

1. **modular** iff

$$\forall p, q, r \in P. p \sqsubseteq q, p \sqcap r = q \sqcap r, p \sqcup r = q \sqcup r \Rightarrow p = q$$

2. **not modular** iff (P, \sqsubseteq) contains a sublattice, which is isomorphic to the lattice:



Distributive Lattices

Let (P, \sqsubseteq) be a lattice with meet operation \sqcap and join operation \sqcup .

Lemma A.4.4.4

1. $\forall p, q, r \in P. p \sqcup (q \sqcap r) \sqsubseteq (p \sqcup q) \sqcap (p \sqcup r)$
2. $\forall p, q, r \in P. p \sqcap (q \sqcup r) \sqsupseteq (p \sqcap q) \sqcup (p \sqcap r)$

Definition A.4.4.5 (Distributive Lattice)

(P, \sqsubseteq) is called **distributive**, if

1. $\forall p, q, r \in P. p \sqcup (q \sqcap r) = (p \sqcup q) \sqcap (p \sqcup r)$
2. $\forall p, q, r \in P. p \sqcap (q \sqcup r) = (p \sqcap q) \sqcup (p \sqcap r)$

Towards Characterizing Distributive Lattices

Lemma A.4.4.6

The following two propositions are equivalent:

1. $\forall p, q, r \in P. p \sqcup (q \sqcap r) = (p \sqcup q) \sqcap (p \sqcup r)$
2. $\forall p, q, r \in P. p \sqcap (q \sqcup r) = (p \sqcap q) \sqcup (p \sqcap r)$

Hence, it is sufficient to require the validity of [property \(1\)](#) or of [property \(2\)](#) in [Definition A.4.4.5](#).

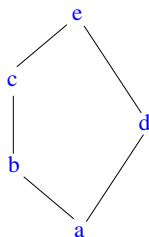
Characterizing Distributive Lattices

Let (P, \sqsubseteq) be a lattice.

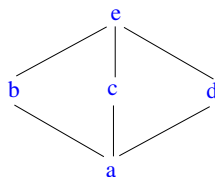
Theorem A.4.4.7 (Characterizing Distributive Lat.)

(P, \sqsubseteq) is not distributive iff (P, \sqsubseteq) contains a sublattice that is isomorphic to one of the below two lattices:

a)



b)



Corollary A.4.4.8

If (P, \sqsubseteq) is distributive, then (P, \sqsubseteq) is modular.

Boolean Lattices

Let (P, \sqsubseteq) be a lattice with meet operation \sqcap , join operation \sqcup , least element \perp , and greatest element \top .

Definition A.4.4.9 (Complement)

Let $p, q \in P$. Then:

1. q is called a **complement** of p , if $p \sqcup q = \top$ and $p \sqcap q = \perp$.
2. P is called **complementary**, if every element in P has a complement.

Definition A.4.4.10 (Boolean Lattice)

(P, \sqsubseteq) is called **Boolean**, if it is complementary, distributive, and $\perp \neq \top$.

Note: If (P, \sqsubseteq) is Boolean, then every element $p \in P$ has an unambiguous unique complement in P , which is denoted by \bar{p} .

Useful Result

Lemma A.4.4.11

Let (P, \sqsubseteq) be a Boolean lattice, and let $p, q, r \in P$. Then:

$$1. \quad \overline{\overline{p}} = p \quad (\text{Involution Law})$$

$$2. \quad \overline{p \sqcup q} = \overline{p} \sqcap \overline{q}, \quad \overline{p \sqcap q} = \overline{p} \sqcup \overline{q} \quad (\text{De Morgan Laws})$$

$$3. \quad p \sqsubseteq q \iff \overline{p} \sqcup q = \top \iff p \sqcap \overline{q} = \perp$$

$$4. \quad p \sqsubseteq q \sqcup r \iff p \sqcap \overline{q} \sqsubseteq r \iff \overline{q} \sqsubseteq \overline{p} \sqcup r$$

Boolean Lat. Homomorphisms/Isomorphisms

Let (P, \sqsubseteq_P) and (Q, \sqsubseteq_Q) be two Boolean lattices, and let $f \in [P \rightarrow Q]$ be a map from P to Q .

Definition A.4.4.12 (Boolean Lattice Homomorphism)

f is called a **Boolean lattice homomorphism**, if f is a lattice homomorphism and

$$\forall p \in P. f(\bar{p}) = \overline{f(p)}$$

Definition A.4.4.13 (Boolean Lattice Isomorphism)

f is called a **Boolean lattice isomorphism**, if f is a Boolean lattice homomorphism and bijective.

Useful Results

Let (P, \sqsubseteq_P) and (Q, \sqsubseteq_Q) be two Boolean lattices, and let $f \in [P \xrightarrow{bhom} Q]$ be a Boolean lattice homomorphism from P to Q .

Lemma A.4.4.14

$$f(\perp) = \perp \wedge f(\top) = \top$$

Lemma A.4.4.15

f is a Boolean lattice isomorphism iff $f(\perp) = \perp \wedge f(\top) = \top$

Summary, Overview

Corollary A.4.4.16

Let $P \neq \emptyset$ be a non-empty set, and \sqsubseteq a relation on P . Then:

- (P, \sqsubseteq) Boolean lattice
- (Def. A.4.4.10) $\Rightarrow (P, \sqsubseteq)$ distributive lattice
- (Cor. A.4.4.8) $\Rightarrow (P, \sqsubseteq)$ modular lattice
- (Def. A.4.4.2) $\Rightarrow (P, \sqsubseteq)$ lattice
- (Def. A.4.1.1) $\Rightarrow (P, \sqsubseteq)$ partial order
- (Def. A.2.1.2) $\Rightarrow (P, \sqsubseteq)$ pre-order

Corollary A.4.4.17

$$QO \supset PO \supset L \supset ML \supset DL \supset BL$$

where all inclusions are proper and QO , PO , L , ML , DL , and BL denote the sets of all quasi- (or pre-) orders, partial orders, lattices, modular, distributive, and Boolean lattices.

Exercise A.4.4.18

Let $(\mathbb{N}_0, \sqsubseteq)$ be the partial order with $\sqsubseteq =_{df} |$, where $|$ denotes the divisibility relation on the natural numbers \mathbb{N}_0 , i.e., the relation ‘ \cdot divides \cdot ’ (w/out remainder), e.g. $5 | 35$.

Prove or refute: $(\mathbb{N}_0, \sqsubseteq)$ is a

1. modular lattice
2. distributive lattice
3. Boolean lattice

Proof or counterexample.

A.4.5

Mechanisms for Constructing Lattices

Common Lattice Constructions: Flat Lattices

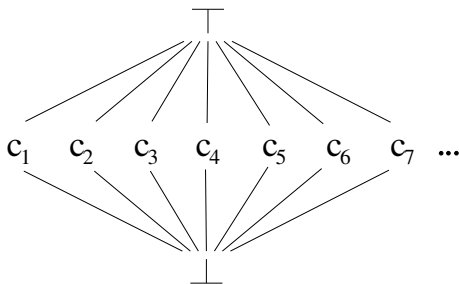
Lemma A.4.5.1 (Flat Lattice Construction)

Let C be a set. Then:

$(C \dot{\cup} \{\perp, \top\}, \sqsubseteq_{flat})$ with \sqsubseteq_{flat} defined by

$$\forall c, d \in C \dot{\cup} \{\perp, \top\}. c \sqsubseteq_{flat} d \iff_{df} c = \perp \vee c = d \vee d = \top$$

is a **complete lattice**, a so-called **flat lattice** (or **diamond lattice**).



Lattice Constructions: Products, Sums,...

Like the principle for constructing flat CPOs also the principles for constructing

- non-strict products
- strict products
- separate sums
- coalesced sums
- continuous (here: additive, distributive) function spaces

carry over from CPOs to (complete) lattices (cf. App. A.3.3).

A.4.6

Order-theoretic and Algebraic View of Lattices

Motivation

In [Definition A.4.1.1](#), we introduced [lattices](#) as special

- [ordered sets](#) (P, \sqsubseteq)

which induces an

- [order-theoretic](#) view of lattices.

Alternatively, [lattices](#) can be introduced as special

- [algebraic structures](#) (P, \sqcap, \sqcup)

which induces an

- [algebraic](#) view of lattices.

Next, we will show that both views are equivalent:

- [Order-theoretically](#) defined lattices can be considered [algebraically](#) and vice versa.

Lattices as Algebraic Structures

Definition A.4.6.1 (Algebraic Lattice)

An **algebraic lattice** is an algebraic structure (P, \sqcap, \sqcup) , where

- $P \neq \emptyset$ is a non-empty set.
- $\sqcap, \sqcup : P \times P \rightarrow P$ are two maps such that for all elements $p, q, r \in P$ the following laws hold (infix notation):
 - **Commutative Laws:** $p \sqcap q = q \sqcap p$
 $p \sqcup q = q \sqcup p$
 - **Associative Laws:** $(p \sqcap q) \sqcap r = p \sqcap (q \sqcap r)$
 $(p \sqcup q) \sqcup r = p \sqcup (q \sqcup r)$
 - **Absorption Laws:** $(p \sqcap q) \sqcup p = p$
 $(p \sqcup q) \sqcap p = p$

Properties of Algebraic Lattices

Let (P, \sqcap, \sqcup) be an algebraic lattice.

Lemma A.4.6.2 (Idempotency Laws)

For all $p \in P$, the maps $\sqcap, \sqcup : P \times P \rightarrow P$ satisfy the following laws:

- Idempotency Laws: $p \sqcap p = p$
 $p \sqcup p = p$

Lemma A.4.6.3

For all $p, q \in P$, the maps $\sqcap, \sqcup : P \times P \rightarrow P$ satisfy:

1. $p \sqcap q = p \iff p \sqcup q = q$
2. $p \sqcap q = p \sqcup q \iff p = q$

Induced (Partial) Order

Let (P, \sqcap, \sqcup) be an algebraic lattice.

Lemma A.4.6.4

The relation $\sqsubseteq \subseteq P \times P$ on P defined by

$$\forall p, q \in P. p \sqsubseteq q \iff_{df} p \sqcap q = p$$

is a partial order relation on P , i.e., \sqsubseteq is reflexive, transitive, and antisymmetric.

Definition A.4.6.5 (Induced Partial Order)

The relation \sqsubseteq defined in Lemma A.4.6.4 is called the **induced (partial) order** of (P, \sqcap, \sqcup) .

Properties of the Induced Partial Order

Let (P, \sqcap, \sqcup) be an algebraic lattice, and let \sqsubseteq be the induced partial order of (P, \sqcap, \sqcup) .

Lemma A.4.6.6

For all $p, q \in P$, the infimum ($\hat{=}$ greatest lower bound) and the supremum ($\hat{=}$ least upper bound) of the set $\{p, q\}$ exist and are given by the images of \sqcap and \sqcup applied to p and q , respectively, i.e.:

$$\forall p, q \in P. \sqcap\{p, q\} = p \sqcap q \wedge \sqcup\{p, q\} = p \sqcup q$$

Lemma A.4.6.6 can inductively be extended yielding:

Lemma A.4.6.7

Let $\emptyset \neq Q \subseteq P$ be a non-empty finite subset of P . Then:

$$\exists glb, lub \in P. glb = \sqcap Q \wedge lub = \sqcup Q$$

Algebraic Lattices Order-theoretically

Corollary A.4.6.8 (From (P, \sqcap, \sqcup) to (P, \sqsubseteq))

Let (P, \sqcap, \sqcup) be an algebraic lattice. Then:

(P, \sqsubseteq) , where \sqsubseteq is the induced partial order of (P, \sqcap, \sqcup) , is an order-theoretic lattice in the sense of [Definition A.4.1.1](#).

Induced Algebraic Maps

Let (P, \sqsubseteq) be an order-theoretic lattice.

Definition A.4.6.9 (Induced Algebraic Maps)

The partial order \sqsubseteq of (P, \sqsubseteq) induces two maps \sqcap and \sqcup from $P \times P$ to P defined by:

1. $\forall p, q \in P. p \sqcap q =_{df} \sqcap\{p, q\}$
2. $\forall p, q \in P. p \sqcup q =_{df} \sqcup\{p, q\}$

Properties of the Induced Algebraic Maps (1)

Let (P, \sqsubseteq) be an order-theoretic lattice, and let \sqcap and \sqcup be the induced algebraic maps of (P, \sqsubseteq) .

Lemma A.4.6.10

Let $p, q \in P$. Then the following propositions are equivalent:

1. $p \sqsubseteq q$
2. $p \sqcap q = p$
3. $p \sqcup q = q$

Properties of the Induced Algebraic Maps (2)

Let (P, \sqsubseteq) be an order-theoretic lattice, and let \sqcap and \sqcup be the induced algebraic maps of (P, \sqsubseteq) .

Lemma A.4.6.11

For all $p, q, r \in P$, the induced maps \sqcap and \sqcup satisfy the following laws:

1. **Commutative Laws:**
 $p \sqcap q = q \sqcap p$
 $p \sqcup q = q \sqcup p$
2. **Associative Laws:**
 $(p \sqcap q) \sqcap r = p \sqcap (q \sqcap r)$
 $(p \sqcup q) \sqcup r = p \sqcup (q \sqcup r)$
3. **Absorption Laws:**
 $(p \sqcap q) \sqcup p = p$
 $(p \sqcup q) \sqcap p = p$
4. **Idempotency Laws:**
 $p \sqcap p = p$
 $p \sqcup p = p$

Order-theoretic Lattices Algebraically

Corollary A.4.6.12 (From (P, \sqsubseteq) to (P, \sqcap, \sqcup))

Let (P, \sqsubseteq) be an order-theoretic lattice. Then:

(P, \sqcap, \sqcup) , where \sqcap and \sqcup are the induced maps of (P, \sqsubseteq) , is an algebraic lattice in the sense of [Definition A.4.6.1](#).

Equivalence (1)

...of the [order-theoretic](#) and the [algebraic view](#) of lattices.

From [order-theoretic](#) to [algebraic lattices](#):

- An order-theoretic lattice (P, \sqsubseteq) can be considered algebraically by switching from (P, \sqsubseteq) to (P, \sqcap, \sqcup) , where \sqcap and \sqcup are the induced maps of (P, \sqsubseteq) .

From [algebraic](#) to [order-theoretic lattices](#):

- An algebraic lattice (P, \sqcap, \sqcup) can be considered order-theoretically by switching from (P, \sqcap, \sqcup) to (P, \sqsubseteq) , where \sqsubseteq is the induced partial order of (P, \sqcap, \sqcup) .

Equivalence (2)

Together, this allows us to simply speak of a lattice P , and to speak only more precisely of P as an

- order-theoretic lattice (P, \sqsubseteq)
- algebraic lattice (P, \sqcap, \sqcup)

if we want to emphasize that we think of P as a special **ordered set** or as a special **algebraic structure**.

Bottom and Top vs. Zero and One (1)

Let P be a lattice with a least and a greatest element.

Considering P

- **order-theoretically** as (P, \sqsubseteq) , it is appropriate to think of its least and greatest element in terms of bottom \perp and top \top with
 - Bottom $\perp \in P$: $\perp = \bigsqcup \emptyset$
 - Top $\top \in P$: $\top = \bigsqcap \emptyset$
- **algebraically** as (P, \sqcap, \sqcup) , it is appropriate to think of its least and greatest element in terms of Zero $\mathbf{0}$ and One $\mathbf{1}$, where (P, \sqcap, \sqcup) is said to have a (if existent, uniquely determined)
 - Zero $\mathbf{0} \in P$: $\forall p \in P. p \sqcup \mathbf{0} = p$
 - One $\mathbf{1} \in P$: $\forall p \in P. p \sqcap \mathbf{1} = p$

Bottom and Top vs. Zero and One (2)

Lemma A.4.6.13

Let P be a lattice. Then:

1. (P, \sqsubseteq) has a bottom element \perp iff (P, \sqcap, \sqcup) has a zero element $\mathbf{0}$, and in that case:

$$(\bigsqcup \emptyset =) \perp = \mathbf{0}$$

2. (P, \sqsubseteq) has a top element \top iff (P, \sqcap, \sqcup) has a one element $\mathbf{1}$, and in that case:

$$(\bigsqcap \emptyset =) \top = \mathbf{1}$$

On the Adequacy of the two Lattice Views

In **mathematics**, usually the

- **algebraic view** of a lattice is more appropriate as it is in line with other algebraic structures ('a set together with some maps satisfying a number of laws'), e.g., **groups, rings, fields, vector spaces, categories**, etc., which are investigated and dealt with in mathematics.

In **computer science**, usually the

- **order-theoretic view** of a lattice is more appropriate, since the order relation can often be interpreted and understood as '**· carries more/less information than ·**,' '**· is more/less defined than ·**,' '**· is stronger/weaker than ·**,' etc., which often fits naturally to problems investigated and dealt with in computer science.

Exercise A.4.6.14

Let $(\mathbb{N}_0, \sqsubseteq)$ be the lattice with $\sqsubseteq =_{df} |$, where $|$ denotes the divisibility relation on the natural numbers \mathbb{N}_0 , i.e., the relation ‘ \cdot divides \cdot ’ (w/out remainder), e.g. $5 | 35$.

Provide the definition of $(\mathbb{N}_0, \wedge, \vee)$, i.e., write down the algebraically defined counterpart of $(\mathbb{N}_0, \sqsubseteq)$. To this end, provide the definition of the meet and join operation on $\mathbb{N}_0 \times \mathbb{N}_0$:

1. $\wedge : \mathbb{N}_0 \times \mathbb{N}_0 \rightarrow \mathbb{N}_0$
2. $\vee : \mathbb{N}_0 \times \mathbb{N}_0 \rightarrow \mathbb{N}_0$

What is the

1. zero element **0**
2. one element **1**

of $(\mathbb{N}_0, \wedge, \vee)$?

A.5

Fixed Point Theorems

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1889/10

A.5.1

Fixed Points, Towers

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

1890/10

Fixed Points of Functions

Definition A.5.1.1 (Fixed Point)

Let M be a set, let $f \in [M \rightarrow M]$ be a function on M , and let $m \in M$ be an element of M . Then:

m is called a **fixed point** of f iff $f(m) = m$.

Least, Greatest Fixed Points in Partial Orders

Definition A.5.1.2 (Least, Greatest Fixed Point)

Let (P, \sqsubseteq) be a partial order, let $f \in [P \rightarrow P]$ be a function on P , and let p be a fixed point of f , i.e., $f(p) = p$. Then:

p is called the

1. **least fixed point** of f , denoted by μf ,
iff $\forall q \in P. f(q) = q \Rightarrow p \sqsubseteq q$
2. **greatest fixed point** of f , denoted by νf ,
iff $\forall q \in P. f(q) = q \Rightarrow q \sqsubseteq p$

Towers in Chain Complete Partial Orders

Definition A.5.1.3 (f -Tower in C)

Let (C, \sqsubseteq) be a CCPO, let $f \in [C \rightarrow C]$ be a function on C , and let $T \subseteq C$ be a subset of C . Then:

T is called an f -tower in C iff

1. $\perp \in T$.
2. If $t \in T$, then also $f(t) \in T$.
3. If $T' \subseteq T$ is a chain in C , then $\bigsqcup T' \in T$.

Least Towers in Chain Complete Partial Orders

Lemma A.5.1.4 (The Least f -Tower in C)

The intersection

$$I =_{df} \bigcap \{T \mid T \text{ } f\text{-tower in } C\}$$

of all f -towers in C is the least f -tower in C , i.e.,

1. I is an f -tower in C .
2. $\forall T$ f -tower in C . $I \subseteq T$.

Lemma A.5.1.5 (Least f -Towers and Chains)

The least f -tower in C is a chain in C , if f is expanding.

A.5.2

Fixed Point Theorems for Complete Partial Orders

Fixed Points of Exp./Monotonic Functions

Fixed Point Theorem A.5.2.1 (Expanding Function)

Let (C, \sqsubseteq) be a CCPO, and let $f \in [C \xrightarrow{\text{exp}} C]$ be an expanding function on C . Then:

The **supremum** of the **least f -tower** in C is a **fixed point** of f .

Fixed Point Theorem A.5.2.2 (Monotonic Function)

Let (C, \sqsubseteq) be a CCPO, and let $f \in [C \xrightarrow{\text{mon}} C]$ be a monotonic function on C . Then:

f has a unique **least fixed point** μf , which is given by the **supremum** of the **least f -tower** in C .

Note

- [Theorem A.5.2.1](#) and [Theorem A.5.2.2](#) ensure the existence of a fixed point for expanding functions and of a unique least fixed point for monotonic functions, respectively, but do not provide constructive procedures for computing or approximating them.
- This is in contrast to [Theorem A.5.2.3](#), which does so for continuous functions. In practice, continuous functions are thus more important and considered where possible.

Least Fixed Points of Continuous Functions

Fixed Point Theorem A.5.2.3 (Knaster, Tarski, Kleene)

Let (C, \sqsubseteq) be a CCPO, and let $f \in [C \xrightarrow{\text{con}} C]$ be a continuous function on C . Then:

f has a unique **least fixed point** $\mu f \in C$, which is given by the **supremum** of the (so-called) **Kleene chain** $\{\perp, f(\perp), f^2(\perp), f^3(\perp), \dots\}$, i.e.:

$$\mu f = \bigsqcup_{i \in \mathbb{N}_0} f^i(\perp) = \bigsqcup \{\perp, f(\perp), f^2(\perp), \dots\}$$

Note: $f^0 =_{df} Id_C$; $f^i =_{df} f \circ f^{i-1}$, $i > 0$.

Proof of Fixed Point Theorem A.5.2.3 (1)

We have to prove:

$$\mu f = \bigsqcup_{i \in \mathbb{N}_0} f^i(\perp) = \bigsqcup \{f^i(\perp) \mid i \geq 0\}$$

1. exists,
2. is a fixed point of f ,
3. is the least fixed point of f .

Proof of Fixed Point Theorem A.5.2.3 (2)

1. Existence

- By definition of \perp as the least element of C and of f^0 as the identity on C we have: $\perp = f^0(\perp) \sqsubseteq f^1(\perp) = f(\perp)$.
- Since f is continuous and hence monotonic, we obtain by means of (natural) induction:
 $\forall i, j \in \mathbb{N}_0. i < j \Rightarrow f^i(\perp) \sqsubseteq f^{i+1}(\perp) \sqsubseteq f^j(\perp)$.
- Hence, the set $\{f^i(\perp) \mid i \geq 0\}$ is a (possibly infinite) chain in C .
- Since (C, \sqsubseteq) is a CCPO and $\{f^i(\perp) \mid i \geq 0\}$ a chain in C , this implies by definition of a CCPO that the least upper bound of the chain $\{f^i(\perp) \mid i \geq 0\}$

$$\bigsqcup \{f^i(\perp) \mid i \geq 0\} = \bigsqcup_{i \in \mathbb{N}_0} f^i(\perp) \text{ exists.}$$

Proof of Fixed Point Theorem A.5.2.3 (3)

2. Fixed point property

$$f\left(\bigsqcup_{i \in \mathbb{N}_0} f^i(\perp)\right)$$

$$(f \text{ continuous}) = \bigsqcup_{i \in \mathbb{N}_0} f(f^i(\perp))$$

$$= \bigsqcup_{i \in \mathbb{N}_1} f^i(\perp)$$

$(C' =_{df} \{f^i \perp \mid i \geq 1\})$ is a chain \Rightarrow

$$\bigsqcup C' \text{ exists} = \perp \sqcup \bigsqcup C' = \perp \sqcup \bigsqcup_{i \in \mathbb{N}_1} f^i(\perp)$$

$$(f^0(\perp) =_{df} \perp) = \bigsqcup_{i \in \mathbb{N}_0} f^i(\perp)$$

Proof of Fixed Point Theorem A.5.2.3 (4)

3. Least fixed point property

- Let c be an arbitrary fixed point of f . Then: $\perp \sqsubseteq c$.
- Since f is continuous and hence monotonic, we obtain by means of (natural) induction:
 $\forall i \in \mathbb{N}_0. f^i(\perp) \sqsubseteq f^i(c) (= c)$.
- Since c is a fixed point of f , this implies:
 $\forall i \in \mathbb{N}_0. f^i(\perp) \sqsubseteq c (= f^i(c))$.
- Thus, c is an upper bound of the set $\{f^i(\perp) \mid i \in \mathbb{N}_0\}$.
- Since $\{f^i(\perp) \mid i \in \mathbb{N}_0\}$ is a chain, and $\bigsqcup_{i \in \mathbb{N}_0} f^i(\perp)$ is by definition the least upper bound of this chain, we obtain the desired inclusion

$$\bigsqcup_{i \in \mathbb{N}_0} f^i(\perp) \sqsubseteq c.$$



Least Conditional Fixed Points

Let (C, \sqsubseteq) be a CCPO, let $f \in [C \rightarrow C]$ be a function on C , and let $d, c_d \in C$ be elements of C .

Definition A.5.2.4 (Least Conditional Fixed Point)

c_d is called the **least conditional fixed point** of f wrt d (in German: *kleinster bedingter Fixpunkt*) iff c_d is the least fixed point of C with $d \sqsubseteq c_d$, i.e.:

$$\forall x \in C. f(x) = x \wedge d \sqsubseteq x \Rightarrow c_d \sqsubseteq x$$

Least Cond. Fixed Points of Cont. Functions

Theorem A.5.2.5 (Conditional Fixed Point Theorem)

Let (C, \sqsubseteq) be a CCPO, let $d \in C$, and let $f \in [C \xrightarrow{\text{con}} C]$ be a continuous function on C which is expanding for d , i.e., $d \sqsubseteq f(d)$. Then:

f has a least conditional fixed point $\mu f_d \in C$, which is given by the supremum of the (generalized) Kleene chain $\{d, f(d), f^2(d), \dots\}$, i.e.:

$$\mu f_d = \bigsqcup_{i \in \mathbb{N}_0} f^i(d) = \bigsqcup \{d, f(d), f^2(d), \dots\}$$

Finite Fixed Points

Let (C, \sqsubseteq) be a CCPO, let $d \in C$, and let $f \in [C \xrightarrow{\text{mon}} C]$ be a monotonic function on C .

Theorem A.5.2.6 (Finite Fixed Point Theorem)

If two succeeding elements in the Kleene chain of f are equal, i.e., if there is some $i \in \mathbb{N}$ with $f^i(\perp) = f^{i+1}(\perp)$, then we have: $\mu f = f^i(\perp)$.

Theorem A.5.2.7 (Finite Conditional FP Theorem)

If f is expanding for d , i.e., $d \sqsubseteq f(d)$, and two succeeding elements in the (generalized) Kleene chain of f wrt d are equal, i.e., if there is some $i \in \mathbb{N}$ with $f^i(d) = f^{i+1}(d)$, then we have: $\mu f_d = f^i(d)$.

Note: Theorems A.5.2.6 and A.5.2.7 do not require continuity of f . Monotonicity (and expandingness) of f suffice(s).

Towards the Existence of Finite Fixed Points

Let (P, \sqsubseteq) be a partial order, and let $p, r \in P$.

Definition A.5.2.8 (Chain-finite Partial Order)

(P, \sqsubseteq) is called **chain-finite** (in German: *kettenendlich*) iff P does not contain an infinite chain.

Definition A.5.2.9 (Finite Element)

p is called

1. **finite** iff the set $Q =_{df} \{q \in P \mid q \sqsubseteq p\}$ does not contain an infinite chain.
2. **finite relative to r** iff the set $Q =_{df} \{q \in P \mid r \sqsubseteq q \sqsubseteq p\}$ does not contain an infinite chain.

Existence of Finite Fixed Points

...there are numerous sufficient conditions ensuring the existence of a least finite fixed point of a function f , which often hold in practice (cf. Nielson/Nielson 1992), e.g.:

- the domain or the range of f are finite or chain-finite,
- the least fixed point of f is finite,
- f is of the form $f(c) = c \sqcup g(c)$ with g a monotonic function on a chain-finite (data) domain.

Fixed Point Theorems, Lattices, and DCPOs

Note: Complete lattices (cf. [Lemma A.4.1.13](#)) and DCPOs with a least element (cf. [Lemma A.3.1.5](#)) are CCPOs, too.

Thus, we can conclude:

Corollary A.5.2.10 (Fixed Points, Lattices, DCPOs)

The fixed point theorems of [Chapter A.5.2](#) hold for functions on complete lattices and on DCPOs with a least element, too.

A.5.3

Fixed Point Theorems for Lattices

Fixed Points of Monotonic Functions

Fixed Point Theorem A.5.3.1 (Knaster, Tarski)

Let (P, \sqsubseteq) be a complete lattice, and let $f \in [P \xrightarrow{mon} P]$ be a monotonic function on P . Then:

1. f has a unique least fixed point $\mu f \in P$, which is given by $\mu f = \bigcap \{p \in P \mid f(p) \sqsubseteq p\}$.
2. f has a unique greatest fixed point $\nu f \in P$, which is given by $\nu f = \bigcup \{p \in P \mid p \sqsubseteq f(p)\}$.

Characterization Theorem A.5.3.2 (Davis)

Let (P, \sqsubseteq) be a lattice. Then:

(P, \sqsubseteq) is complete iff every $f \in [P \xrightarrow{mon} P]$ has a fixed point.

The Fixed Point Lattice of Mon. Functions

Theorem A.5.3.3 (Lattice of Fixed Points)

Let (P, \sqsubseteq) be a complete lattice, let $f \in [P \xrightarrow{\text{mon}} P]$ be a monotonic function on P , and let $\text{Fix}(f) =_{df} \{p \in P \mid f(p) = p\}$ be the set of all fixed points of f . Then:

Every subset $F \subseteq \text{Fix}(f)$ has a supremum and an infimum in $\text{Fix}(f)$, i.e., $(\text{Fix}(f), \sqsubseteq|_{\text{Fix}(f)})$ is a complete lattice.

Theorem A.5.3.4 (Ordering of Fixed Points)

Let (P, \sqsubseteq) be a complete lattice, and let $f \in [P \xrightarrow{\text{mon}} P]$ be a monotonic function on P . Then:

$$\bigsqcup_{i \in \mathbb{N}_0} f^i(\perp) \sqsubseteq \mu f \sqsubseteq \nu f \sqsubseteq \bigsqcap_{i \in \mathbb{N}_0} f^i(\top)$$

Fixed Points of Add./Distributive Functions

For **additive** and **distributive functions**, the leftmost and the rightmost inequality of Theorem A.5.3.4 become equalities:

Fixed Point Theorem A.5.3.5 (Knaster,Tarski,Kleene)

Let (P, \sqsubseteq) be a complete lattice, and let $f \in [P \rightarrow P]$ be a function on P . Then: f has a unique

1. least fixed point $\mu f \in P$ given by $\mu f = \bigsqcup_{i \in \mathbb{N}_0} f^i(\perp)$, if f is additive, i.e., $f \in [P \xrightarrow{add} P]$.
2. greatest fixed point $\nu f \in P$ given by $\nu f = \bigsqcap_{i \in \mathbb{N}_0} f^i(\top)$, if f is distributive, i.e., $f \in [P \xrightarrow{dis} P]$.

Recall: $f^0 =_{df} Id_C$; $f^i =_{df} f \circ f^{i-1}$, $i > 0$.

A.6

Fixed Point Induction

Admissible Predicates

Fixed point induction allows proving properties of fixed points of continuous functions. Essential is the notion of **admissible predicates**:

Definition A.6.1 (Admissible Predicate)

Let (P, \sqsubseteq) be a complete lattice, and let $\phi : P \rightarrow \mathbb{B}$ be a predicate on P . Then:

ϕ is called **admissible** (or **\sqcup -admissible**) iff for every chain $C \subseteq P$ holds:

$$(\forall c \in C. \phi(c)) \Rightarrow \phi(\bigsqcup C)$$

Lemma A.6.2

Let (P, \sqsubseteq) be a complete lattice, and let $\phi : P \rightarrow \mathbb{B}$ be an admissible predicate on P . Then: $\phi(\perp) = \text{true}$.

Proof. The admissibility of ϕ implies $\phi(\bigsqcup \emptyset) = \text{true}$. Moreover, we have $\perp = \bigsqcup \emptyset$, which completes the proof.

Sufficient Conditions for Admissibility

Theorem A.6.3 (Admissibility Condition 1)

Let (P, \sqsubseteq) be a complete lattice, and let $\phi : P \rightarrow \mathbb{B}$ be a predicate on P . Then:

ϕ is admissible, if there is a complete lattice (Q, \sqsubseteq_Q) and two additive functions $f, g \in [P \xrightarrow{add} Q]$, such that

$$\forall p \in P. \phi(p) \iff f(p) \sqsubseteq_Q g(p)$$

Theorem A.6.4 (Admissibility Condition 2)

Let (P, \sqsubseteq) be a complete lattice, and let $\phi, \psi : P \rightarrow \mathbb{B}$ be two admissible predicates on P . Then:

The conjunction of ϕ and ψ , the predicate $\phi \wedge \psi$ defined by

$$\forall p \in P. (\phi \wedge \psi)(p) =_{df} \phi(p) \wedge \psi(p)$$

is admissible.

Fixed Point Induction on Complete Lattices

Theorem A.6.5 (Fixed Point Induction on Com. Lat.)

Let (P, \sqsubseteq) be a complete lattice, let $f \in [P \xrightarrow{add} P]$ be an additive function on P , and let $\phi : P \rightarrow \mathbb{B}$ be an admissible predicate on P . Then:

The validity of

$$- \forall p \in P. \phi(p) \Rightarrow \phi(f(p)) \quad (\text{Induction step})$$

implies the validity of $\phi(\mu f)$.

Note: The **induction base**, i.e., the validity of $\phi(\perp)$, is implied by the admissibility of ϕ (cf. [Lemma A.6.2](#)) and proved when verifying the admissibility of ϕ .

Fixed Point Induction on CCPOs

The notion of admissibility of a predicate carries over from complete lattices to CCPOs.

Theorem A.6.6 (Fixed Point Induction on CCPOs)

Let (C, \sqsubseteq) be a CCPO, let $f \in [C \xrightarrow{mon} C]$ be a monotonic function on C , and let $\phi : C \rightarrow \mathbb{B}$ be an admissible predicate on C . Then:

The validity of

$$- \forall c \in C. \phi(c) \Rightarrow \phi(f(c)) \quad (\text{Induction step})$$

implies the validity of $\phi(\mu f)$.

Note: Theorem A.6.6 holds (of course still), if we replace the CCPO (C, \sqsubseteq) by a complete lattice (P, \sqsubseteq) .

A.7

Completions, Embeddings

A.7.1

Downsets

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10

Chap. 11

Chap. 12

Chap. 13

19/19

Downsets

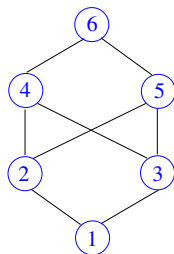
Definition A.7.1.1 (Downset)

Let (P, \sqsubseteq) be a partial order, let $D \subseteq P$ be a subset of P , and let $p, q \in P$ with $p \sqsubseteq q$. Then:

1. D is called a **downset** (or **lower set** or **order ideal**) (in German: *Abwärtsmenge*) of P , if: $q \in D \Rightarrow p \in D$.
2. $\mathcal{D}(P)$ denotes the **set** of all **downsets** of P .

Example

Let (P, \sqsubseteq) be the partial order given by the below Hasse diagram.



Then, e.g.:

- $\emptyset, P \in \mathcal{D}(P), \forall q \in P. \{p \in P \mid p \sqsubseteq q\} \in \mathcal{D}(P)$
- $\{1, 3\}, \{1, 2, 3\}, \{1, 2, 3, 4\} \in \mathcal{D}(P)$
- $\{2, 3\}, \{2, 4, 5\}, \{1, 2, 4, 5\} \notin \mathcal{D}(P)$

Properties of Downsets

Lemma A.7.1.2

Let (P, \sqsubseteq) be a partial order, let $q \in P$, and $Q \subseteq P$. Then:

1. $\emptyset \in \mathcal{D}(P)$, $P \in \mathcal{D}(P)$, are (trivial) downsets of P .
2. $\downarrow q =_{df} \{p \in P \mid p \sqsubseteq q\} \in \mathcal{D}(P)$.
3. $\downarrow Q =_{df} \{p \in P \mid \exists q \in Q. p \sqsubseteq q\} \in \mathcal{D}(P)$.
4. $Q \in \mathcal{D}(P) \iff Q = \downarrow Q$

Lemma A.7.1.3

Let (P, \sqsubseteq) be a partial order, and let $p, q \in P$. Then the following propositions are equivalent:

1. $p \sqsubseteq q$
2. $\downarrow p \subseteq \downarrow q$
3. $\forall D \in \mathcal{D}(P). q \in D \Rightarrow p \in D$.

Characterization of Downsets

Lemma A.7.1.4 (Downsets of a PO)

Let (P, \sqsubseteq) be a partial order. Then:

$$\mathcal{D}(P) = \{\downarrow Q \mid Q \subseteq P\}$$

Corollary A.7.1.5

Let (P, \sqsubseteq) be a partial order, let $D \in \mathcal{D}(P)$, and let $p, q \in P$ with $p \sqsubseteq q$. Then: $q \in D \Rightarrow p \in D$.

The Lattice of Downsets: Complete & Distr.

Let (P, \sqsubseteq) be a partial order, let $\mathcal{D}(P)$ be the set of downsets of P , and let \subseteq denote set inclusion.

Theorem A.7.1.6 (Complete & Distr. L. of Downsets)

$(\mathcal{D}(P), \subseteq)$ is a complete and distributive lattice, the so-called **downset lattice** of P , with set intersection \cap as meet operation, set union \cup as join operation, least element \emptyset , and greatest element P .

Recall: Complete lattices are CCPOs and DCPOs, too (cf. Lemma A.4.1.13). Thus, we have:

Corollary A.7.1.7 (The CCPO/DCPO of Downsets)

$(\mathcal{D}(P), \subseteq)$ is a **CCPO** and a **DCPO** with least element \emptyset .

From POs to Lattices, CCPOs, and DCPOs

Construction Principle:

Theorem A.7.1.6 and Corollary A.7.1.7 yield a construction principle that shows how to construct

- a complete lattice and thus also a CCPO and a DCPO from a given partial order (P, \sqsubseteq) (cf. Appendix A.3.3 and Appendix A.4.5).

Principal Downsets

The downsets of the form $\{p \in P \mid p \sqsubseteq q\}$ of a partial order (P, \sqsubseteq) considered in Lemma A.7.1.2(2) are peculiar, and will reoccur as so-called **principal ideals** (cf. Chapter A.7.2) and **principal cuts** (cf. Chapter A.7.3) of lattices. Therefore, we introduce these distinguished downsets explicitly.

Definition A.7.1.8 (Principal Downsets of a PO)

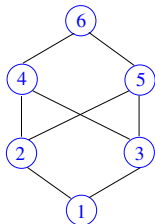
Let (P, \sqsubseteq) be a partial order, and let $q \in P$ be an element of P . Then:

1. $\downarrow q =_{df} \{p \in P \mid p \sqsubseteq q\}$ denotes the **principal downset** (in German: **Hauptabwärtsmenge**) generated by q .
2. $\mathcal{PD}(P) = \{\downarrow q \mid q \in P\}$ denotes the **set** of all **principal downsets** of P .

Downsets, Directed Sets (1)

...principal downsets of partial orders are **directed** but usually not strongly directed.

Example 1: Consider the below partial order (P, \sqsubseteq) :



- $\forall p \in P. \downarrow p =_{df} \{r \mid r \sqsubseteq p\}$ *directed* $\in \mathcal{D}(P)$.
- $\forall p \in P \setminus \{6\}. \downarrow p$ *strongly directed* $\in \mathcal{D}(P)$.
- $\downarrow 6 =_{df} \{r \mid r \sqsubseteq 6\} = \{1, 2, 3, 4, 5, 6\} = P \in \mathcal{D}(P)$ is a downset of P , however, it is not strongly directed, since its subsets $\{2, 3\}, \{1, 2, 3\} \subseteq \downarrow 6$ do not have a least upper bound in $\downarrow 6 = P$ (though upper bounds: $4, 5, 6$).

Downsets, Directed Sets (2)

Example 2: Consider the below lattice (\mathbb{Z}, \leq) :

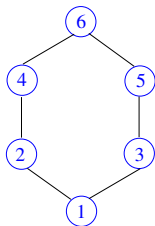


- $\mathcal{D}(\mathbb{Z}) = \emptyset \cup \mathcal{PD}(\mathbb{Z}) \cup \mathbb{Z} = \emptyset \cup \{\downarrow z =_{df} \{r \in \mathbb{Z} \mid r \leq z\} \mid z \in \mathbb{Z}\} \cup \mathbb{Z}$
- $\forall S \in \mathcal{D}(\mathbb{Z})$. S directed but not strongly directed (since it lacks a least element).

Downsets, Directed Sets (3)

...arbitrary downsets even of complete lattices are usually **not strongly directed**, though **directed**.

Example 3: Consider the below complete lattice (P, \sqsubseteq) :



E.g., the downsets

$$- \downarrow\{4, 5\} =_{df} \{r \mid r \sqsubseteq 4 \vee r \sqsubseteq 5\} = \{1, 2, 3, 4, 5\} \in \mathcal{D}(P)$$

$$- \downarrow\{3, 4\} =_{df} \{r \mid r \sqsubseteq 3 \vee r \sqsubseteq 4\} = \{1, 2, 3, 4\} \in \mathcal{D}(P)$$

of P are directed but not strongly directed: The subsets $\{2, 3\} \subseteq \downarrow\{4, 5\}$ and $\{1, 2, 3\} \subseteq \downarrow\{3, 4\}$ do not have a least upper bound in $\downarrow\{4, 5\}$ and $\downarrow\{3, 4\}$, respectively.

A.7.2

Ideal Completion: Embedding of Lattices

Lattice Ideals

Definition A.7.2.1 (Lattice Ideal)

Let (P, \sqsubseteq) be a lattice, let $\emptyset \neq I \subseteq P$ be a non-empty subset of P , and let $p, q \in P$. Then:

1. I is called an **ideal** (or **lattice ideal**) of P , if:

$$1.1 \quad p, q \in I \Rightarrow p \sqcup q \in I.$$

$$1.2 \quad q \in I \Rightarrow p \sqcap q \in I.$$

2. $\mathcal{I}(P)$ denotes the **set** of all **ideals** of P .

Properties of Lattice Ideals

Lemma A.7.2.2 (Ideal Properties 1)

Let (P, \sqsubseteq) be a lattice, let $I \in \mathcal{I}(P)$, and let $q \in I$. Then:

1. $\{p \in P \mid p \sqsubseteq q\} \subseteq I$.
2. $P \in \mathcal{I}(P)$ is a (trivial) ideal of P .

Lemma A.7.2.3 (Ideal Properties 2)

Let (P, \sqsubseteq) be a lattice with least element \perp , and $I \in \mathcal{I}(P)$. Then:

1. $\perp \in I$.
2. $\{\perp\} \in \mathcal{I}(P)$ is a (trivial) ideal of P .

Characterizing Lattice Ideals

Theorem A.7.2.4 (Ideal Characterization)

Let (P, \sqsubseteq) be a lattice, and let $\emptyset \neq I \subseteq P$ be a non-empty subset of P . Then:

$$I \in \mathcal{I}(P) \text{ iff } \forall p, q \in P. p, q \in I \iff p \sqcup q \in I$$

Lattice Ideals and Order Ideals

Lemma A.7.2.5

Let (P, \sqsubseteq) be a lattice, let $I \in \mathcal{I}(P)$, and let $p, q \in P$ with $p \sqsubseteq q$. Then: $q \in I \Rightarrow p \in I$.

Corollary A.7.1.5 – recalled

Let (P, \sqsubseteq) be a partial order, let $D \in \mathcal{D}(P)$, and let $p, q \in P$ with $p \sqsubseteq q$. Then: $q \in D \Rightarrow p \in D$.

Corollary A.7.2.6

Let (P, \sqsubseteq) be a lattice, and let $I \subseteq P$. Then:

$$I \in \mathcal{I}(P) \Rightarrow I \in \mathcal{D}(P) \quad (\text{i.e., } \mathcal{I}(P) \subseteq \mathcal{D}(P)).$$

Note: The reverse implication of [Corollary A.7.2.6](#) does not hold.

The Complete Lattice of Ideals

Theorem A.7.2.7 (The Complete Lattice of Ideals)

Let (P, \sqsubseteq) be a lattice with least element \perp , and let $\sqsubseteq_{\mathcal{I}}$ be the following ordering relation on the set $\mathcal{I}(P)$ of ideals of P :

$$\forall I, J \in \mathcal{I}(P). I \sqsubseteq_{\mathcal{I}} J \text{ iff } I \subseteq J$$

Then: $(\mathcal{I}(P), \sqsubseteq_{\mathcal{I}})$ is a complete lattice, the so-called **lattice of ideals** of P , with join operation $\sqcup_{\mathcal{I}}$ defined by

$$\forall I, J \in \mathcal{I}(P). I \sqcup_{\mathcal{I}} J =_{df} \{p \in P \mid \exists i \in I, j \in J. p \sqsubseteq i \sqcup j\}$$

and meet operation $\sqcap_{\mathcal{I}}$ defined by

$$\forall I, J \in \mathcal{I}(P). I \sqcap_{\mathcal{I}} J =_{df} I \cap J$$

and with least element $\{\perp\}$ and greatest element P .

Principal Ideals

Lemma A.7.2.8

Let (P, \sqsubseteq) be a lattice, and let $q \in P$ be an element of P .
Then:

$$\downarrow q = \{p \in P \mid p \sqsubseteq q\} \text{ ideal} \in \mathcal{I}(P).$$

Definition A.7.2.9 (Principal Ideal)

Let (P, \sqsubseteq) be a lattice, and let $q \in P$ be an element of P .
Then:

1. $\downarrow q$ is called the **principal ideal** of P generated by q .
2. $\mathcal{PI}(P) =_{df} \{\downarrow q \mid q \in P\}$ denotes the **set** of all **principal ideals** of P .

Towards the Sublattice of Principal Ideals

Lemma A.7.2.10

Let (P, \sqsubseteq) be a lattice with least element, and let $(\mathcal{I}(P), \sqsubseteq_{\mathcal{I}})$ be the complete lattice of ideals of P . Then:

$$\forall q, r \in P. \downarrow q \sqcap_{\mathcal{I}} \downarrow r = \downarrow(q \sqcap r) \wedge \downarrow q \sqcup_{\mathcal{I}} \downarrow r = \downarrow(q \sqcup r)$$

The Sublattice of Principal Ideals

Theorem A.7.2.11 (Sublattice of Principal Ideals)

Let (P, \sqsubseteq) be a lattice with least element, let $(\mathcal{I}(P), \sqsubseteq_{\mathcal{I}})$ be the complete lattice of ideals of P , let $\mathcal{PI}(P)$ be the set of the principal ideals of P , and let $\sqsubseteq_{\mathcal{PI}}$ be the restriction of $\sqsubseteq_{\mathcal{I}}$ onto $\mathcal{PI}(P)$. Then:

$(\mathcal{PI}(P), \sqsubseteq_{\mathcal{PI}})$ is a sublattice of $(\mathcal{I}(P), \sqsubseteq_{\mathcal{I}})$.

Note: The sublattice $(\mathcal{PI}(P), \sqsubseteq_{\mathcal{PI}})$ of $(\mathcal{I}(P), \sqsubseteq_{\mathcal{I}})$ is

- usually not complete, not even if (P, \sqsubseteq) is complete.

(The lattice (\mathbb{Z}, \leq) , e.g., enriched with a least element \perp and a greatest element \top is complete, while the lattice of its principal ideals $(\mathcal{PI}(\mathbb{Z}), \sqsubseteq_{\mathcal{PI}})$ is not.)

Ideal Completion and Embedding of a Lattice

Theorem A.7.2.12 (Ideal Completion & Embedding)

Let (P, \sqsubseteq) be a lattice with least element, and let $(\mathcal{I}(P), \sqsubseteq_{\mathcal{I}})$ be the complete lattice of its ideals. Then:

The map

$$e_{\mathcal{I}} : P \rightarrow \mathcal{PI}(P) \text{ defined by } \forall p \in P. e_{\mathcal{I}}(p) =_{df} \downarrow p$$

is a lattice isomorphism between P and the (sub)lattice $\mathcal{PI}(P)$ of its principal ideals.

Intuitively

Theorem A.7.2.12 shows how a lattice (P, \sqsubseteq) with least element

- can be considered a sublattice of the complete lattice of the ideals of P ; in more detail, how it can be considered the sublattice $(\mathcal{PI}(P), \sqsubseteq_{\mathcal{PI}})$ of the complete lattice $(\mathcal{I}(P), \sqsubseteq_{\mathcal{I}})$.

A.7.3

Cut Completion: Embedding of Partial Orders and Lattices

Definition A.7.3.1 (Cut)

Let (P, \sqsubseteq) be a partial order, and let $Q \subseteq P$ be a subset of P .
Then:

1. Q is called a **cut** (in German: *Schnitt*) of P , if $Q = LB(UB(Q))$.
2. $\mathcal{C}(P)$ denotes the **set** of all **cuts** of P .

Properties of Cuts

Lemma A.7.3.2

Let (P, \sqsubseteq) be a partial order, and let $q \in P$ be an element of P . Then:

1. $LB(\{q\}) =_{df} \downarrow q =_{df} \{p \in P \mid p \sqsubseteq q\} \in \mathcal{C}(P)$
2. $LB(UB(\{q\})) = \{p \in P \mid p \sqsubseteq q\} = LB(\{q\})$

Note: If (P, \sqsubseteq) is a lattice,

1. Lemma A.7.3.2(1) yields that **principal ideals** are **cuts** of P :

$$\forall q \in P. \langle q \rangle =_{df} \{p \in P \mid p \sqsubseteq q\} = LB(\{q\}) \in \mathcal{C}(P)$$

$$(\text{or: } \forall Q \subseteq P. Q \in \mathcal{PI}(P) \Rightarrow Q \in \mathcal{C}(P))$$

2. Lemma A.7.3.2(2) characterizes the principal ideals of P in terms of the function composition $LB \circ UB$.

Principal Cuts

Definition A.7.3.3 (Principal Cut)

Let (P, \sqsubseteq) be a partial order, and let $q \in P$ be an element of P . Then:

1. $\downarrow q =_{df} LB(UB(\{q\}))$ is called the **principal cut** of P generated by q .
2. $\mathcal{PC}(P) =_{df} \{\downarrow q \mid q \in P\}$ denotes the **set** of all **principal cuts** of P .

Properties of Cuts and Ideals of Lattices

Lemma A.7.3.4

Let (P, \sqsubseteq) be a lattice with least element, and let $Q \subseteq P$.
Then:

$$Q \in \mathcal{C}(P) \Rightarrow Q \in \mathcal{I}(P)$$

Corollary A.7.3.5

Let (P, \sqsubseteq) be a lattice with least element, and let $Q \subseteq P$.
Then:

$$Q \in \mathcal{C}(P) \Rightarrow Q \neq \emptyset$$

Note: Corollary A.7.3.5 does not hold for partial orders.

The Complete Lattice of Cuts

Theorem A.7.3.6 (The Complete Lattice of Cuts)

Let (P, \sqsubseteq) be a partial order, and let \sqsubseteq_c be the following ordering relation on the set $\mathcal{C}(P)$ of cuts of P :

$$\forall C, D \in \mathcal{C}(P). C \sqsubseteq_c D \text{ iff } C \subseteq D$$

Then: $(\mathcal{C}(P), \sqsubseteq_c)$ is a complete lattice, the so-called **lattice of cuts** of P , with join operation \sqcup_c defined by

$$\forall C, D \in \mathcal{C}(P). C \sqcup_c D =_{df} \bigcap \{E \in \mathcal{C}(P) \mid C \cup D \subseteq E\}$$

and meet operation \sqcap_c defined by

$$\forall C, D \in \mathcal{C}(P). C \sqcap_c D =_{df} C \cap D$$

and with least element $\{\perp\}$ and greatest element P .

Cut Completion and Embedding of a PO

Theorem A.7.3.7 (PO Cut Completion & Embedd'g)

Let (P, \sqsubseteq) be a partial order, and let $(\mathcal{C}(P), \sqsubseteq_c)$ be the complete lattice of its cuts. Then:

The map

$$e_c : P \rightarrow \mathcal{PC}(P) \text{ defined by } \forall p \in P. e_c(p) =_{df} LB(UB(\{p\}))$$

is an **order isomorphism** between P and the partial order $(\mathcal{PC}(P), \sqsubseteq_{\mathcal{PC}})$ of the principal cuts of P .

Cut Completion and Embedding of a Lattice

Theorem A.7.3.8 (Lattice Cut Completion & Emb'g)

Let (P, \sqsubseteq) be a lattice, let $(\mathcal{C}(P), \sqsubseteq_{\mathcal{C}})$ be the complete lattice of its cuts, and let $e_{\mathcal{C}} : P \rightarrow \mathcal{PC}(P)$ be the map of Theorem A.7.3.7. Then:

$(\mathcal{PC}(P), \sqsubseteq_{\mathcal{PC}})$ is a sublattice of $(\mathcal{C}(P), \sqsubseteq_{\mathcal{C}})$ and $e_{\mathcal{C}}$ is a lattice isomorphism between P and the sublattice $\mathcal{PC}(P)$ of the principal cuts of P .

A.7.4

Downset Completion: Embedding of Partial Orders

Downsets, Ideals, and Cuts

Lemma A.7.4.1

We have:

1. $\mathcal{C}(P) \subseteq \mathcal{D}(P)$, if (P, \sqsubseteq) is a partial order.
2. $\mathcal{C}(P) \subseteq \mathcal{I}(P) \subseteq \mathcal{D}(P)$, if (P, \sqsubseteq) is a lattice with least element.

Downset Completion and Embedding of a PO

Theorem A.7.4.2 (Downset Completion and Emb.'g)

Let (P, \subseteq) be a partial order, and let $(\mathcal{D}(P), \subseteq)$ be the complete and distributive lattice of its downsets (cf. Theorem A.7.1.6). Then:

The map $e_c : P \rightarrow \mathcal{PC}(P)$ (of Theorem A.7.3.7) defined by

$$\forall p \in P. e_c(p) =_{df} LB(UB(\{p\}))$$

is an order isomorphism between P and the partial order $(\mathcal{PC}(P), \subseteq)$ of the principal cuts of P , or, equivalently, the map $e_c : P \rightarrow \mathcal{D}(P)$ defined as above is a partial order embedding of $(\mathcal{PC}(P), \subseteq)$ into $(\mathcal{D}(P), \subseteq)$.

Intuitively

Theorem A.7.4.2 shows how a partial order (P, \sqsubseteq)

- can be considered a partial order of the complete and distributive lattice of its downsets; in more detail, how it can be considered the partial order $(\mathcal{PC}(P), \sqsubseteq_{\mathcal{PC}})$ of the complete and distributive lattice $(\mathcal{D}(P), \sqsubseteq_{\mathcal{D}})$.

A.7.5

Application: Lists and Streams

Technically

...the construction of Chapter A.7.4 works by

- switching from the elements p of a set P partially ordered by a relation \sqsubseteq to the principal downsets $\downarrow p \in \mathcal{PD}(P)$ of the set of downsets $\mathcal{D}(P)$ of P ordered by the subset inclusion \subseteq .

Identifying

- every element $p \in P$ with its principal downset

$$\downarrow p =_{df} \{r \mid r \sqsubseteq p\} \in \mathcal{PD}(P)$$

yields an

- embedding of P into $\mathcal{PD}(P) =_{df} \{\downarrow q \mid q \in P\}$, i.e., a function $e : P \rightarrow \mathcal{PD}(P)$ with

$$\forall p, q \in P. p \sqsubseteq q \Leftrightarrow \downarrow p \subseteq \downarrow q$$

From Monotonic to Continuous Functions

...completion is the key to Theorem A.7.5.1:

Let (P, \sqsubseteq_P) be a partial order, let $\downarrow q =_{df} \{p \in P \mid p \sqsubseteq q\}$ for $q \in P$, let $\mathcal{PD}(P) =_{df} \{\downarrow q \mid q \in P\}$, and let (C, \sqsubseteq_C) be a CPO.

Theorem A.7.5.1 (From Monotonicity to Continuity)

A monotonic function $f \in [P \xrightarrow{mon} C]$ can uniquely be extended to a continuous function $\hat{f} \in [\mathcal{PD}(P) \xrightarrow{con} C]$.

Application: Lists and Streams (1)

Lemma A.7.5.2 (The CPO of Lists and Streams)

Let L be the set of all finite and infinite lists, and let \sqsubseteq_{pfx} be the prefix relation ' \cdot is a prefix of \cdot ' on L defined by

$$\forall l, l'' \in L. l \sqsubseteq_{\text{pfx}} l'' \iff_{\text{df}} \\ l = l'' \vee (l \text{ finite} \wedge \exists l' \in L. l ++ l' = l'')$$

Then: $(L, \sqsubseteq_{\text{pfx}})$ is a CPO (i.e., a CCPO and DCPO).

Lemma A.7.5.3 (Downsets of the Set of Lists)

Let L be the set of all finite and infinite lists, and let $\mathcal{PD}(L) = \{\downarrow l \mid l \in L\}$ be the set of principal downsets of L . Then:

1. $\downarrow l =_{\text{df}} \{l' \in L \mid l' \sqsubseteq_{\text{pfx}} l\}$ is a directed set (even a strongly directed set), i.e., a directed downset of lists.
2. $(\mathcal{PD}(L), \subseteq)$ is a CPO (i.e., a CCPO and DCPO).

Application: Lists and Streams (2)

Putting these [findings](#) together, we obtain:

- The set of downsets of lists ordered by set inclusion is a CPO.
- Every (infinite) chain of ever longer finite lists represents the corresponding stream, the supremum of this chain.
- [Theorem A.7.4.3](#) allows the application of a function to a stream to be approximated and computed by applying the function to the finite prefixes of the stream yielding a chain of approximations of the stream that would result from the application of the function to the stream itself.
- Continuity ensures the correctness of this procedure: it yields the equality of the supremum of the computed chain of approximations and the result of applying the continuous function to the argument stream itself.

Application: Lists and Streams (3)

Together, this [implies](#):

- Recursive equations and functions on streams as considered in [Chapter 2](#) are well defined.

A.8

References, Further Reading

Contents

Part I

Chap. 1

Part II

Chap. 2

Chap. 3

Chap. 4

Part III

Chap. 5

Chap. 6

Part IV

Chap. 7

Chap. 8

Chap. 9

Chap. 10




Chap. 11

Chap. 12



Chap. 13

1959/10





Appendix A: Further Reading (1)

-  André Arnold, Irène Guessarian. *Mathematics for Computer Science*. Prentice Hall, 1996.
-  Roland Backhouse, Roy Crole, Jeremy R. Gibbons (Eds.). *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*. International Summer School and Workshop, Oxford, UK, April 10-14, 2000, Revised Lectures, Springer-V., LNCS 2297, 2002. (Chapter 1, Ordered Sets and Complete Lattices by Hilary A. Priestley; Chapter 2, Algebras and Coalgebras by Peter Aczel; Chapter 4, Calculating Functional Programs by Jeremy Gibbons)
-  Rudolf Berghammer. *Ordnungen, Verbände und Relationen mit Anwendungen*. Vieweg+Teubner, 2008.

Appendix A: Further Reading (2)

-  Rudolf Berghammer. *Ordnungen, Verbände und Relationen mit Anwendungen*. Springer-V., 2012. (Kapitel 1, Ordnungen und Verbände; Kapitel 2.4, Vollständige Verbände; Kapitel 3, Fixpunkttheorie mit Anwendungen; Kapitel 4, Vervollständigung und Darstellung mittels Vervollständigung; Kapitel 5, Wohlgeordnete Mengen und das Auswahlaxiom)
-  Rudolf Berghammer. *Ordnungen und Verbände: Grundlagen, Vorgehensweisen und Anwendungen*. Springer-V., 2013. (Kapitel 2, Verbände und Ordnungen; Kapitel 3.4, Vollständige Verbände; Kapitel 4, Fixpunkttheorie mit Anwendungen; Kapitel 5, Vervollständigung und Darstellung mittels Vervollständigung; Kapitel 6, Wohlgeordnete Mengen und das Auswahlaxiom)





Appendix A: Further Reading (3)

-  Garret Birkhoff. *Applications of Lattice Algebra*. Mathematical Proceedings of the Cambridge Philosophical Society 30(2):115-122, 1934.
-  Garret Birkhoff. *Lattice Theory*. American Mathematical Society, 3rd edition, 1967.
-  Peter Crawley, Robert P. Dilworth. *Algebraic Theory of Lattices*. Prentice Hall, 1973.
-  Brian A. Davey, Hilary A. Priestley. *Introduction to Lattices and Order*. Cambridge Mathematical Textbooks, Cambridge University Press, 2nd edition, 2002. (Chapter 1, Ordered Sets; Chapter 2, Lattices and Complete Lattices; Chapter 8, CPOs and Fixpoint Theorems)





Appendix A: Further Reading (4)

-  Anne C. Davis. *A Characterization of Complete Lattices*. Pacific Journal of Mathematics 5(2):311-319, 1955.
-  Marcel Ern . *Einf hrung in die Ordnungstheorie*. Bibliographisches Institut, 2. Auflage, 1982.
-  Helmuth Gericke. *Theorie der Verb nde*. Bibliographisches Institut, 2. Auflage, 1967.
-  George Gr tzer. *General Lattice Theory*. Birkh user, 2nd edition, 1998. (Chapter 1, First Concepts; Chapter 2, Distributive Lattices; Chapter 3, Congruences and Ideals; Chapter 5, Varieties of Lattices)
-  George Gr tzer. *Lattice Theory: Foundation*. Birkh user, 2011.





Appendix A: Further Reading (5)

-  George Grätzer, Friedrich Wehrung (Eds.). *Lattice Theory: Special Topics and Applications, Vol. I*. Birkhäuser, 2014.
-  George Grätzer, Friedrich Wehrung (Eds.). *Lattice Theory: Special Topics and Applications, Vol. II*. Birkhäuser, 2016.
-  Paul R. Halmos. *Naive Set Theory*. Springer-V., Reprint, 2001. (Chapter 6, Ordered Pairs; Chapter 7, Relations; Chapter 8, Functions)
-  Hans Hermes. *Einführung in die Verbandstheorie*. Springer-V., 2. Auflage, 1967.





Appendix A: Further Reading (6)

-  Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000. (Chapter 11, Proof by Induction; Chapter 14.6, Inductive Properties of Infinite Lists)
-  Richard Johnsonbaugh. *Discrete Mathematics*. Pearson, 7th edition, 2009. (Chapter 3, Functions, Sequences, and Relations)
-  Stephen C. Kleene. *Introduction to Metamathematics*. North Holland, 1952. (Reprint, North Holland, 1980)
-  Seymour Lipschutz. *Set Theory and Related Topics*. McGraw Hill Schaum's Outline Series, 2nd edition, 1998. (Chapter 4, Functions; Chapter 6, Relations)




Appendix A: Further Reading (7)

-  David Makinson. *Sets, Logic and Maths for Computing*. Springer-V., 2008. (Chapter 1, Collecting Things Together: Sets; Chapter 2, Comparing Things: Relations)
-  George Markowsky. *Chain-complete Posets and Directed Sets with Applications*. *Algebra Universalis* 6(1):53-68, 1976.
-  Flemming Nielson, Hanne Riis Nielson. *Finiteness Conditions for Fixed Point Iteration*. In *Proceedings of the 7th ACM Conference on LISP and Functional Programming (LFP'92)*, 96-108, 1992.
-  Hanne Riis Nielson, Flemming Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, 1992. (Chapter 4, Denotational Semantics)




Appendix A: Further Reading (8)

-  Hanne Riis Nielson, Flemming Nielson. *Semantics with Applications: An Appetizer*. Springer-V., 2007. (Chapter 5, Denotational Semantics)
-  Flemming Nielson, Hanne Riis Nielson, Chris Hankin. *Principles of Program Analysis*. Springer-V., 2nd edition, 2005. (Appendix A, Partially Ordered Sets)
-  Peter Pepper, Petra Hofstedt. *Funktionale Programmierung: Sprachdesign und Programmieretechnik*. Springer-V., 2006. (Kapitel 10, Beispiel: Berechnung von Fixpunkten; Kapitel 10.2, Ein bisschen Mathematik: CPOs und Fixpunkte)
-  Steven Roman. *Lattices and Ordered Sets*. Springer-V., 2008.

Appendix A: Further Reading (9)

-  Bernhard Steffen, Oliver Rüthing, Malte Isberner. *Grundlagen der höheren Informatik. Induktives Vorgehen*. Springer-V., 2014. (Kapitel 5.1, Ordnungsrelationen; Kapitel 5.2, Ordnungen und Teilstrukturen)
-  Bernhard Steffen, Oliver Rüthing, Michael Huth. *Mathematical Foundations of Advanced Informatics: Inductive Approaches*. Springer-V., 2018. (Chapter 5.1, Order Relations; Chapter 5.2, Orders and Substructures)
-  Alfred Tarski. *A Lattice-theoretical Fixpoint Theorem and its Applications*. Pacific Journal of Mathematics 5(2):285-309, 1955.

Appendix A: Further Reading (10)

-  Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3rd edition, 2011. (Chapter 9, Reasoning about Programs; Chapter 17.9, Proof revisited)
-  Franklyn Turbak, David Gifford with Mark A. Sheldon. *Design Concepts in Programming Languages*. MIT Press, 2008. (Chapter 5, Fixed Points; Chapter 105, Software Testing; Chapter 106, Formal Methods; Chapter 107, Verification and Validation)
-  Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993. (Chapter 1, Basic set theory; Chapter 8, Introduction to domain theory; Chapter 8.2, Streams – an example; Chapter 8.3, Construction on cpo's; Chapter 10, Recursion techniques; Chapter 10.2, Fixed-point induction)