Fortgeschrittene funktionale Programmierung
LVA 185.A05, VU 2.0, ECTS 3.0
SS 2019
(Stand: 26.06.2019)

Jens Knoop

Technische Universität Wien
Information Systems Engineering
Compilers and Languages
Table of Contents (1)

Part I: Motivation

► Chap. 1: Why Functional Programming Matters
  1.1 Setting the Stage
  1.2 Glueing Functions Together
  1.3 Glueing Programs Together
  1.4 Summary
  1.5 References, Further Reading

Part II: Programming Principles

► Chap. 2: Programming with Streams
  2.1 Streams
  2.2 Stream Diagrams
  2.3 Memoization
  2.4 Boosting Performance
  2.5 References, Further Reading
Table of Contents (2)

► 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 References, Further Reading

► Chap. 4: Equational Reasoning
  4.1 Motivation
  4.2 Functional Pearls
  4.3 The Smallest Free Number
  4.4 Not the Maximum Segment Sum
  4.5 A Simple Sudoku Solver
  4.6 References, Further Reading
Table of Contents (3)

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
  5.6 Test Data Generators at Work: An Example
  5.7 Monitoring, Reporting, and Coverage
  5.8 Implementation of QuickCheck
  5.9 Summary
  5.10 References, Further Reading
Table of Contents (4)

► Chap. 6: Verification
  6.1 Inductive Proof Principles on Natural Numbers
    6.1.1 Natural Induction
    6.1.2 Strong Induction
  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
    6.3.4 Inductive Proofs on Haskell Stream Approximants
  6.4 Approximation
  6.5 Coinduction
  6.6 Fixed Point Induction
  6.7 Other Approaches, VerificationTools
    6.7.1 Correctness by Construction
    6.7.2 Selected other Approaches and Tools
  6.8 References, Further Reading
Table of Contents (5)

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
  ► 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
Table of Contents (6)

- Chap. 9: Monoids
  9.1 Motivation
  9.2 The Type Class Monoid
  9.3 Monoid Examples
    9.3.1 Lists as Monoid
    9.3.2 Numerical Types as Monoids
    9.3.3 Bool as Monoid
    9.3.4 Ordering as Monoid
  9.4 Summary and Looking ahead
  9.5 References, Further Reading

- Chap. 10: Functors
  10.1 Motivation
  10.2 The Type Constructor Class Functor
Table of Contents (7)

► Chap. 10: Functors (cont’d)

10.3 Predefined Functors
  10.3.1 The Identity Functor
  10.3.2 The Maybe Functor
  10.3.3 The List Functor
  10.3.4 The Input/Output Functor
  10.3.5 The Either Functor
  10.3.6 The Map Functor

10.4 The Type Constructor Class Applicative

10.5 Predefined Applicatives
  10.5.1 The Identity Applicative
  10.5.2 The Maybe Applicative
  10.5.3 The List Applicative
  10.5.4 The Input/Output Applicative
  10.5.5 The Either Applicative
  10.5.6 The Map Applicative
  10.5.7 The Ziplist Applicative

10.5 Kinds of Types and Type Constructors

10.6 References, Further Reading
Table of Contents (8)

Chap. 11: Monads

11.1 Motivation
   11.1.1 Functional Composition Reconsidered
   11.1.2 Example: Debug Information
   11.1.3 Example: Random Numbers
   11.1.4 Findings, Looking ahead
   11.1.5 Excursus on Functional Composition

11.2 The Type Constructor Class Monad

11.3 Syntactic Sugar: The do-Notation

11.4 Predefined Monads
   11.4.1 The Identity Monad
   11.4.2 The List Monad
   11.4.3 The Maybe Monad
   11.4.4 The Either Monad
   11.4.5 The Map Monad
   11.4.6 The State Monad
   11.4.7 The Input/Output Monad
Table of Contents (9)

► Chap. 11: Monads (cont’d)
  11.5 Monadic Programming
    11.5.1 Folding Trees
    11.5.2 Numbering Tree Labels
    11.5.3 Renaming Tree Labels
  11.6 MonadPlus
    11.6.1 The Type Constructor Class MonadPlus
    11.6.2 The Maybe MonadPlus
    11.6.3 The List MonadPlus
  11.7 Summary
  11.8 References, Further Reading

► Chap. 12: Arrows
  12.1 Motivation
  12.2 The Type Constructor Class Arrow
  12.3 A Fresh Look at the Haskell Class Hierarchy
  12.4 References, Further Reading
Table of Contents (10)

Part V: Applications

► Chap. 13: Parsing

13.1 Motivation
13.2 Combinator Parsing
  13.2.1 Primitive Parsers
  13.2.2 Parser Combinators
  13.2.3 Universal Combinator Parser Basis
  13.2.4 Structure of Combinator Parsers
  13.2.5 Writing Combinator Parsers: Examples

13.3 Monadic Parsing
  13.3.1 Parsers as Monads
  13.3.2 Parsers by Type Class Instantiations
  13.3.3 Universal Monadic Parser Basis
  13.3.4 Utility Parsers
  13.3.5 Structure of a Monadic Parser
  13.3.6 Writing Monadic Parsers: Examples

13.4 Summary
13.5 References, Further Reading
Chap. 14: Logic Programming Functionally

14.1 Motivation

14.1.1 On the Evolution of Programming Languages
14.1.2 Functional vs. Logic Languages
14.1.3 A Curry Appetizer
14.1.4 Outline

14.2 The Combinator Approach

14.2.1 Introduction
14.2.2 Diagonalization
14.2.3 Diagonalization with Monads
14.2.4 Filtering with Conditions
14.2.5 Indicating Search Progress
14.2.6 Selecting a Search Strategy
14.2.7 Terms, Substitutions, Unification, and Predicates
14.2.8 Combinators for Logic Programs
14.2.9 Writing Logic Programs: Two Examples

14.3 Summary

14.4 References, Further Reading
Chap. 15: Pretty Printing

15.1 Motivation

15.2 The Simple Pretty Printer
   15.2.1 Basic Document Operators
   15.2.2 Normal Forms of String Documents
   15.2.3 Printing Trees

15.3 The Prettier Printer
   15.3.1 Algebraic Documents
   15.3.2 Algebraic Representations of Document Operators
   15.3.3 Multiple Layouts of Algebraic Documents
   15.3.4 Normal Forms of Algebraic Documents
   15.3.5 Improving Performance
   15.3.6 Utility Functions
   15.3.7 Printing XML-like Documents

15.4 The Prettier Printer Code Library
   15.4.1 The Prettier Printer
   15.4.2 The Tree Example
   15.4.3 The XML Example

15.5 Summary

15.6 References, Further Reading
Chap. 16: Functional Reactive Programming

16.1 Motivation
16.2 An Imperative Robot Language
   16.2.1 The Robot’s World
   16.2.2 Modelling the Robot’s World
   16.2.3 Modelling Robots
   16.2.4 Modelling Robot Commands as State Monad
   16.2.5 The Imperative Robot Language
   16.2.6 Defining a Robot’s World
   16.2.7 Robot Graphics: Animation in Action

16.3 Robots on Wheels
   16.3.1 The Setting
   16.3.2 Modelling the Robots’ World
   16.3.3 Classes of Robots
   16.3.4 Robot Simulation in Action
   16.3.5 Examples

16.4 Summary
16.5 References, Further Reading
Table of Contents (14)

Part VI: Extensions, Perspectives

➤ Chap. 17: Extensions to Parallel and ‘Real World’ Functional Programming
  17.1 Parallelism in Functional Languages
  17.2 Haskell for ‘Real World’ Programming
  17.3 References, Further Reading

➤ Chap. 18: Conclusions and Perspectives
  18.1 Research Venues, Research Topics, and More
  18.2 Programming Contest
  18.3 In Conclusion
  18.4 References, Further Reading

➤ References

Appendix

➤ A Mathematical Foundations
Table of Contents (15)

- 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 (16)

▷ 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 (17)

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

John Carmack
Motivation

The preceding, a quote from a recent article by Yaron Minsky:

- OCaml for the Masses
  ...why the next language you learn should be functional.

The next, a quote from a classical article by John Hughes:

- Why Functional Programming Matters
  ...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.
Chapter 1

Why Functional Programming Matters
Why Functional Programming Matters

...considering a position statement by John Hughes which is based on a 1984 internal memo at Chalmers University, and has slightly revised been published in:


“...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 ma- king it clear what those advantages are.”
Chapter 1.1
Setting the Stage
Introductory Statement

A matter of **fact**:

- Software is becoming more and more complex.
- Hence: Structuring software well becomes paramount.
- Well-structured software is more easily to read, to write, to debug, and to be re-used.

**Claim:**

- Conventional languages place **conceptual limits on the way problems can be modularized**.
- Functional languages push these limits back.
- **Fundamental**: Higher-order functions and lazy evaluation.

**Purpose of the position statement:**

- Providing evidence for this claim.
A First Observation

...functional programming owes its name 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).
Evidence by Folk Knowledge: Soft Facts

...characteristics & advantages of functional programming:

Functional programs are

- free of assignments and 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 their correctness)
...this ‘folk knowledge’ list 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.”
Evidence by Folk Knowledge: Hard(er) Facts

Aren’t there any hard(er) facts providing evidence for substantial and “real” advantages? Yes, there are, e.g.:

Functional programs are

▶ a magnitude of order smaller than conventional programs
⇒ functional programmers are thus much more productive

Issue left open, however:

▶ Why? Can the productivity gain be concluded from the list of advantages of the ‘standard catalogue,’ i.e., from dropping features?

Hardly. Dropping features reminds more to a medieval monk denying himself the pleasures of life in the hope of getting virtuous.
The ‘folk knowledge’ catalogue is not satisfying; in particular, it does not provide

- any help in exploiting the power of functional languages
  - Programs cannot be written which are particularly lacking in assignment statements, or which are particularly referentially transparent
- a yardstick of program quality, thus no model to strive for

We need a positive characterization of the vital nature of

- functional programming, of its strengths
- what makes a ‘good’ functional program, of what a functional programmer should strive for
A Striking Analogue

...structured vs. non-structured programming.

Structured programs are

- free of goto-statements (‘goto considered harmful’)
- blocks in structured programs are free of multiple entries and exits

⇒ easier to mathematically cope with than unstructured programs

...this is also essentially a negative ‘is-not’ characterization.
Conceptually more Important

Structured programs are

- designed modularly

in contrast to non-structured ones.

It is for this reason that structured programming is more efficient/productive:

- Small modules are easier and faster to read, to write, and to 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’
Key Thesis of John Hughes

The expressiveness

- of a language which supports modular design depends much on the power of the concepts and primitives allowing to combine solutions of subproblems to the solution of the overall problem (keyword: glue; example: making of a chair).

Functional programming

- provides two new, especially powerful glues:
  1. Higher-order functions
  2. Lazy evaluation

...offering 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.

Modularization

- ‘smaller, simpler, more general’ is the guideline, which should be followed by a functional programmer when programming.
In the following...

...we will reconsider higher-order functions and lazy evaluation from the perspective of their ‘glueing capability’ enabling to compose functions and programs modularly.

Utilizing

- higher-order functions to glueing functions together
- lazy evaluation to glueing programs together
Chapter 1.2
Glueing Functions Together
Glueing Functions Together

Syntax (in the flavour of Miranda™):

► Lists

\[
\text{listof } X ::= \text{nil} \mid \text{cons } X \text{ (listof } X) \]

► Abbreviations (for convenience)

\[
\begin{align*}
[] & \quad \text{means nil} \\
[1] & \quad \text{means cons } 1 \text{ nil} \\
[1,2,3] & \quad \text{means cons } 1 \text{ (cons } 2 \text{ (cons } 3 \text{ nil)))}
\end{align*}
\]

Example: Adding the elements of a list

\[
\begin{align*}
\text{sum } \text{nil} & = 0 \\
\text{sum (cons num list)} & = \text{num} + \text{sum list}
\end{align*}
\]
Observation

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

\[
\begin{align*}
\text{sum } \text{nil} & \quad = \quad \mid 0 \mid \\
\text{sum (cons num list)} & \quad = \quad \text{num} \quad \mid + \mid \quad \text{sum list}
\end{align*}
\]

...i.e., computing a sum of values can be modularly decomposed by properly combining

- a general recursion pattern and
- a set of more specific operations

(see framed parts above).
Exploiting the Observation

1. Adding the elements of a list

   \[ \text{sum} = \text{reduce add 0} \]
   \[ \text{where } \text{add} \ x \ y = x+y \]

   This reveals the definition of the higher-order function \text{reduce} almost immediately:

   \[
   (\text{reduce } f \ x) \ \text{nil} = x \\
   (\text{reduce } f \ x) \ (\text{cons } a \ l) = f \ a \ ((\text{reduce } f \ x) \ l)
   \]

Recall

   \[
   \text{sum nil} = |0| \\
   \text{sum (cons num list)} = \text{num} \ | \text{+} | \text{sum list}
   \]
Immediate Benefit: Re-use of the HoF reduce

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

2. **Multiplying** the elements of a list
   
   \[
   \text{product} = \text{reduce} \ \text{mult} \ 1 \\
   \text{where} \ \text{mult} \ x \ y = x \times y
   \]

3. **Test**, if *some* element of a list equals “true”
   
   \[
   \text{anytrue} = \text{reduce} \ \text{or} \ \text{false}
   \]

4. **Test**, if *all* elements of a list equal “true”
   
   \[
   \text{alltrue} = \text{reduce} \ \text{and} \ \text{true}
   \]

5. **Concatenating** two lists
   
   \[
   \text{append} \ a \ b = \text{reduce} \ \text{cons} \ b \ a
   \]

6. **Doubling** each element of a list
   
   \[
   \text{doubleall} = \text{reduce} \ \text{doubleandcons} \ nil \\
   \text{where} \ \text{doubleandcons} \ \text{num} \ \text{list} \\
   = \text{cons} \ (2 \times \text{num}) \ \text{list}
   \]
How does it work? (1)

Intuitively, the call \((\text{reduce } f \ a)\) can be understood such that in a list of elements all occurrences of

- \(\text{cons}\) are replaced by \(f\)
- \(\text{nil}\) by \(a\)

in list values.

Examples:

1) Addition:

\[
\text{reduce } \text{add } 0 \ (\text{cons }2 \ (\text{cons }3 \ (\text{cons }5 \ \text{nil})))
\]
\[
\rightarrow (\text{add }2 \ (\text{add }3 \ (\text{add }5 \ 0 )))
\]
\[
\rightarrow 10
\]

2) Multiplication:

\[
\text{reduce } \text{mult } 1 \ (\text{cons }2 \ (\text{cons }3 \ (\text{cons }5 \ \text{nil})))
\]
\[
\rightarrow (\text{mult }2 \ (\text{mult }3 \ (\text{mult }5 \ 1 )))
\]
\[
\rightarrow 30
\]
How does it work? (2)

Examples (cont’d):

5) Concatenating two lists

Key: Observing that reduce cons nil copies a list of elements, leads to:

\[
\text{append } a \ b = \text{reduce cons } b \ a
\]

append \([1,2]\) \([3,4]\)

\[\to \quad \text{reduce cons } [3,4] \ [1,2]\]

\[\to \quad (\text{reduce cons } [3,4]) \ (\text{cons } 1 \ (\text{cons } 2 \ \text{nil}))\]

\[\to \quad \{\text{replacing cons by cons and nil by } [3,4]\} \quad (\text{cons } 1 \ (\text{cons } 2 \ [3,4]))\]

\[\to \quad \{\text{expanding } [3,4]\} \quad (\text{cons } 1 \ (\text{cons } 2 \ (\text{cons } 3 \ (\text{cons } 4 \ \text{nil}))))\]

\[\to \quad \{\text{compressing the list expression}\} \quad [1,2,3,4]\]
How does it work? (3)

Examples (cont’d):

6) **Doubling each element of a list**

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

Note that **doubleandcons** can be modularized further:

▶ **First step**

   doubleandcons = fandcons double
   where fandcons f el list = cons (f el) list
double n = 2*n

▶ **Second step**

   fandcons f = cons . f
   where “.” denotes sequential composition of functions:
   (f . g) h = f (g h)
How does it work? (4)

...correctness of the two modularization steps follows from

\[
\text{fandcons } f \text{ el } = (\text{cons } . f) \text{ el} \\
= \text{cons } (f \text{ el})
\]

which yields as desired:

\[
\text{fandcons f el list } = \text{cons } (f \text{ el}) \text{ list}
\]
How does it work? (5)

Putting things together, we obtain:

6a) **Doubling** each element of a list
\[
\text{doubleall} = \text{reduce} \ (\text{cons} \ . \ \text{double}) \ \text{nil}
\]

Another step of modularization using **map** leads us to:

6b) **Doubling** each element of a list
\[
\text{doubleall} = \text{map} \ \text{double}
\]
where
\[
\text{map} \ f = \text{reduce} \ (\text{cons} \ . \ f) \ \text{nil}
\]
i.e., **map** applies a function \( f \) to every element of a list.
Homework

Using the functions introduced so far, we can define:

- Adding the elements of a matrix

\[
\text{summatrix} = \text{sum} \ . \ \text{map} \ \text{sum}
\]

Think about how summatrix works.
Summing up

By decomposing (modularizing) and representing a simple function (\texttt{sum} in the example) as a combination of

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

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

This is useful for more complex data structures, too, as is shown next...
Generalization

...to more complex data structures:

Example: Trees

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))
The Higher-order Function `redtree`

Analogously to `reduce` on lists we introduce a higher-order function `redtree` 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 (1)

1. Adding the labels of the leaves of a tree
2. Generating a list of all labels occurring in a tree
3. A function `maptree` on trees replicating the function `map` on lists

Running Example:

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

1. Adding the labels of the leaves of a tree

\[
\text{sumtree} = \text{redtree} \ add \ add \ 0
\]

Example:

Performing the replacements in the tree of the running example, we get:

\[
\text{sumtree} (\text{node} \ 1 \\
\phantom{\text{sumtree}} (\text{cons} \ (\text{node} \ 2 \ \text{nil}) \\
\phantom{\text{sumtree}} (\text{cons} \ (\text{node} \ 3 \ (\text{cons} \ (\text{node} \ 4 \ \text{nil}) \ \text{nil})) \\
\phantom{\text{sumtree}} \text{nil}))))
\rightarrow (\text{add} \ 1 \\
\phantom{\text{sumtree}} (\text{add} \ (\text{add} \ 2 \ 0) \\
\phantom{\text{sumtree}} (\text{add} \ (\text{add} \ 3 \ (\text{add} \ (\text{add} \ 4 \ 0) \ 0 \ 0)))
\rightarrow 10
Applications (3)

2. Generating a list of all labels occurring in a tree

\[
\text{labels} = \text{redtree cons append nil}
\]

Example:
Performing the replacements in the tree of the running example, we get:

\[
\text{sumtree (node 1}
\text{ (cons (node 2 nil)}
\text{ (cons (node 3 (cons (node 4 nil) nil)) nil)))}
\rightarrow (\text{cons 1}
\text{ (app'd (cons 2 nil)}
\text{ (app'd (cons 3 (app'd (cons 4 nil) nil)) nil)))}
\rightarrow [1,2,3,4]
\]
3. A function `maptree` applying a function `f` to every label of a tree

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

Example: Homework.
Summing up (1)

The simplicity and elegance of the preceding examples is a consequence of combining

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

Once the higher-order function is implemented, lots of

- further functions can be implemented essentially without any further effort!
Summing up (2)

Lesson learnt:

- Whenever a new data type is introduced, 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 (to some extent this resembles the concepts known from conventional extensible languages).
Chapter 1.3

Glueing Programs Together
Glueing Programs Together

**Recall:** A complete functional program is a function from its input to its output.

- If $f$ and $g$ are complete functional programs, then also their composition
  
  $$(g \cdot f)$$

  is a program. Applied to $\text{in}$ as input, it yields the output
  
  $$\text{out} = (g \cdot f) \text{in} = g(f \text{in})$$

- A possible implementation using **conventional glue** is:
  
  Communication via files

Possible problems:

- Temporary files used for communication can be too large
- $f$ might not terminate
...lazy evaluation allows a more elegant approach.

This is to decompose a program into a
  ➤ generator
  ➤ selector

component/module, which are then glued together.

Intuitively:
  ➤ The generator “runs as little as possible” until it is terminated by the selector.
Three Examples

...for illustrating this modularization strategy:

1. Square root computation
2. Numerical integration
3. Numerical differentiation
Example 1: Square Root Computation

...according to Newton-Raphson.

**Given:** \( N \)

**Sought:** \( \text{squareRoot}(N) \)

**Iteration formula:**

\[
\text{a}(n+1) = \frac{(\text{a}(n) + N/\text{a}(n))}{2}
\]

**Justification:** If the sequence of approximations converges to some limit \( a, a \neq 0 \), for some initial approximation \( a(0) \), we have:

\[
\frac{(a + N/a)}{2} = a \quad \mid \times 2
\]

\[
\Leftrightarrow a + N/a = 2a \quad \mid -a
\]

\[
\Leftrightarrow N/a = a \quad \mid \times a
\]

\[
\Leftrightarrow N = a*a \quad \mid \text{sqr}
\]

\[
\Leftrightarrow \text{squareRoot}(N) = a
\]

I.e., \( a \) stores the value of the square root of \( N \).
We consider first

...a typical imperative (Fortran) implementation:

```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
```

→ essentially a **monolithic**, not decomposable program.
The Functional Version: New Approximations

Computing the next approximation from the previous one:

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

Defining \( g = \text{next } N \), we are interested in computing the (possibly infinite) sequence of approximations:

\[
[a_0, g \ a_0, g (g \ a_0), g (g (g \ a_0)), \ldots]
\]

\[\rightarrow [a_0, \text{next } N \ a_0, \text{next } N (\text{next } N \ a_0), \text{next } N (\text{next } N (\text{next } N \ a_0)), \ldots]\]
The Functional Version: Writing a Generator

The function \texttt{repeat} computes this (possibly infinite) sequence of approximations. It is the \textit{generator} component in this example:

**Generator A:**

\begin{align*}
\text{repeat } f \ a &= \text{cons } a \ (\text{repeat } f \ (f \ a))
\end{align*}

Applying \texttt{repeat} to the arguments \texttt{next } \texttt{N} and \texttt{a0} yields the desired sequence of approximations:

\begin{align*}
\text{repeat } (\text{next } \texttt{N}) \ \texttt{a0} & \rightarrow [\texttt{a0}, g \ \texttt{a0}, g \ (g \ \texttt{a0}), g \ (g \ (g \ \texttt{a0})), \ldots] \\
& \rightarrow [\texttt{a0}, \texttt{next N a0}, \texttt{next N (next N a0)}, \\
& \quad \texttt{next N (next N (next N a0))}, \ldots]
\end{align*}
The Functional Version: Writing a Selector

**Note:** Evaluating the *generator* term `repeat (next N) a0` does not terminate!

**Remedy:** Taming the *generator* by a *selector* to compute \( \text{square root} \ N \) only up to a given tolerance \( \epsilon > 0 \): 

**Selector A:**

\[
\begin{align*}
\text{within} \ \epsilon & \ (\text{cons} \ a \ (\text{cons} \ b \ \text{rest})) \\
= b, & \quad \text{if} \ \text{abs}(a-b) \leq \epsilon \\
= \text{within} \ \epsilon & \ (\text{cons} \ b \ \text{rest}), \ \text{otherwise}
\end{align*}
\]
The Functional Version: Combining Gen./Sel.

...to obtain the final program.

**Composition:** Glueing generator and selector together:

\[ \text{sqrt } N \text{ eps } a0 = \text{within eps } (\text{repeat}(\text{next } N) \text{ a0}) \]

\[ \text{Selector A} \quad \text{Generator A} \]

\[ \leadsto \text{We are done!} \]
The Functional Version: Summing up

➤ **repeat**: generator program/module:

\[ [a_0, \ g \ a_0, \ g(g \ a_0), \ g(g(g \ a_0)), ...] \]

...potentially infinite, no pre-defined limit of length.

➤ **within**: selector program/module:

\[ g^i \ a_0 \] 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!

**Note**: Lazy evaluation ensures that both programs/modules (generator and selector) run strictly synchronized.
Re-using Modules

Next, we will show that

- generators
- selectors

can indeed be considered modules which can easily be re-used.

We are going to start with re-using the generator module.
Re-using a Generator w/ new Selectors

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:

\[
\text{relative eps} \ (\text{cons a} \ (\text{cons b rest})) \\
= b, \quad \text{if abs}(a-b) \leq \text{eps} \times \text{abs} \ b \\
= \text{relative eps} \ (\text{cons b rest}), \text{otherwise}
\]

Composition: **Glueing** old generator and new selector together:

\[
\text{relativesqrt} \ N \ \text{eps} \ a0 \\
= \text{relative eps} \ (\text{repeat} \ (\text{next} \ N) \ a0) \\
\underline{\text{Selector B}} \quad \underline{\text{Generator A}}
\]

\[\implies \text{We are done!}\]
Next: Re-using a Selector w/ new Generators

Note that the module \textit{generator} in the previous example, i.e.

- the component computing the sequence of approximations has been re-used unchanged.

Next, we will re-use the two \textit{selector} modules considering two examples:

- Numerical integration
- Numerical differentiation
Example 2: Numerical Integration

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

**Sought:** The area under $f$ between $a$ and $b$

**Naive Implementation:**
...supposed that the function $f$ is roughly linear between $a$ und $b$.

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

This is sufficiently precise, however, at most for very small intervals.
\[ \int_{a}^{b} f(x) \, dx = A + B = (f(a) + f(b)) \frac{(b-a)}{2} \]
Writing a Generator

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:

```lisp
(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

```lisp
zip (cons a s) (cons b t) = cons (pair a b) (zip s t)
```
Re-using Selectors w/ the new Generator

Note, evaluating the new generator term \( \text{integrate } f \ a \ b \) does not terminate!

Remedy: Taming the new generator by the previously defined two selectors to compute \( \text{integrate } f \ a \ b \) only up to some given limit \( \epsilon > 0 \).

Composition: Re-using selectors for new generator-selector combinations:

1) within \( \epsilon \) (integrate \( f \ a \ b \))

\[
\underbrace{\text{Selector } A}_{\text{Generator } B}
\]

2) relative \( \epsilon \) (integrate \( f \ a \ b \))

\[
\underbrace{\text{Selector } B}_{\text{Generator } B}
\]
Summing up

- One new generator module: **integrate**
  ...potentially infinite, no pre-defined limit of length.
- Two old selector modules: **within**, **relative**
  ...lazy evaluation ensures that the selector function is applied eventually ⇒ termination!

Note, the two selector modules

- picking the solution from the stream of approximate solutions

have been re-used unchanged from the square root example.

- **Lazy evaluation** is the key to synchronize the generator and selector modules!
A Note on Efficiency

The generator \texttt{integrate} as defined previously is

\begin{itemize}
  \item sound but inefficient (many re-computations of \(f\ a\), \(f\ b\),
  and \(f\ \text{mid}\), which are redundant and hence superfluous).
\end{itemize}

Introducing \texttt{locally defined values} as shown below removes this
deficiency:

\begin{verbatim}
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
\end{verbatim}
Example 3: Numerical Differentiation

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

**Sought:** The slope of $f$ at point $x$

**Naive Implementation:**
...supposed that the function $f$ between $x$ and $x+h$ does not “curve much”

```haskell
easydiff f x h = (f (x+h) - f x) / h
```

This is sufficiently precise, however, at most for very small values of $h$. 
Writing a Generator/Selector Combination

Implement a generator computing a sequence of approximations getting successively more accurate:

**Generator C:**

\[
\text{differentiate } h_0 \ f \ x = \text{map (easydiff } f \ x) (\text{repeat halve } h_0) \\
\text{halve } x = x/2
\]

...and combine it with a selector picking a sufficiently accurate approximation:

**Selector A:**

\[
\text{within } \epsilon (\text{differentiate } h_0 \ f \ x)
\]

**Selector A**  **Generator C**

**Homework:** Combine Generator C with Selector B, too.
Summing up

Obviously, all three examples (square root computation, numerical integration, numerical differentiation) enjoy a common composition pattern using and combining a

▶ generator (usually looping!) and
▶ selector (ensuring termination thanks to lazy evaluation!)

This composition/modularization principle can be further generalized to combining

▶ generators, selectors, filters, and transformers

as illustrated in Chapter 2.
Chapter 1.4

Summary
Findings (1)

Central Thesis of John Hughes

- Modularity is the key to programming in the large.

Observation

- 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 the modules together.
- The availability of proper glue is essential!
Findings (2)

Facts

- **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 decompositions of problems (by offering elegant composition means) as here given evidence by an array of simple, yet impressive examples

- In essence, it is the **superior glue**, which allows functional programs to be written so concisely and elegantly (rather than the freedom of assignments, etc.)
Findings (3)

Guidelines

- A functional programmers shall strive for adequate modularization and generalization
  - Especially, if a portion of a program looks ugly or appears to be too complex.
- A functional programmer shall expect that
  - higher-order functions and
  - lazy evaluation

are the tools for achieving this!
The Question of Lazy or Eager Evaluation

...reconsidered. The final conclusion of John Hughes 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**: thanks to lazy evaluation.
- **Algorithm patterns**: thanks to higher-order functions.
Chapter 1.5

References, Further Reading
Chapter 1: Further Reading (1)


Chapter 1: Further Reading (2)


  http://research.microsoft.com/users/simonpj/papers/haskell-retrospective/
Chapter 1: Further Reading (3)


Part II
Programming Principles
Chapter 2
Programming with Streams
Motivation

Streams \equiv Infinite\ Lists \equiv Lazy\ Lists: \ldots used synonymously.

Programming with streams

- Applications
  - Streams plus lazy evaluation yield new modularization principles
    - Generator/selector
    - Generator/filter
    - Generator/transformer
  as instances of the Generator/Prune Paradigm
- Pitfalls and remedies

- Foundations
  - Well-definedness of functions on streams
    (cf. Appendix A.7.5)
  - Proving properties of programs with streams
    (cf. Chapter 6.3.4, 6.4, 6.5, 6.6)
Chapter 2.1
Streams
Streams

...combined with lazy evaluation often

- allow to solve problems elegantly, concisely, and efficiently
- can be a source of hassle if applied inappropriately

More on this next.
Streams

...could be introduced in terms of a new polymorphic data type `Stream` such as:

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

For pragmatic reasons (i.e., convenience/adequacy)

...we prefer modelling streams as ordinary lists waiving the usage of the empty list `[]` in this chapter.

This way

- all pre-defined (polymorphic) functions on lists can directly be used, which otherwise would have to be defined from scratch on the new data type `Stream`. 
Simple Examples of Streams

► **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

Illustration

ones →→ 1 : ones
    →→ 1 : (1 : ones)
    →→ 1 : (1 : (1 : ones))
    →→ ...

ones represents an infinite list (or a stream).
Corecursive Definitions

Definitions of the form

ones = 1 : ones
twos = 2 : twos
threes = 3 : threes

defining the streams of “ones,” “twos,” and “threes”

▶ are called corecursive.

Corecursive definitions

▶ look like recursive definitions but lack a base case.
▶ always yield infinite objects.
▶ remind to Münchhausen’s famous trick of “sich am eigenen Schopfe aus dem Sumpf zu ziehen”!
More Streams defined corecursively

- The stream of natural numbers \texttt{nats}
  \[
  \texttt{nats} = 0 : \text{map (+1) nats} \\
  \rightarrow [0, 1, 2, 3, \ldots]
  \]

- The stream of even natural numbers \texttt{evens}
  \[
  \texttt{evens} = 0 : \text{map (+2) evens} \\
  \rightarrow [0, 2, 4, 6, \ldots]
  \]

- The stream of odd natural numbers \texttt{odds}
  \[
  \texttt{odds} = 1 : \text{map (+2) odds} \\
  \rightarrow [1, 3, 5, 7, \ldots]
  \]

- The stream of natural numbers \texttt{theNats}
  \[
  \texttt{theNats} = 0 : \text{zipWith (+) ones theNats} \\
  \rightarrow [0, 1, 2, 3, \ldots]
  \]
Streams by List Comprehension and Recursion

- The stream of powers of an integer
  
  \[
  \text{powers} :: \text{Int} \rightarrow [\text{Int}]
  \]
  
  \[
  \text{powers} \ n = [n^x | x \leftarrow [0..]]
  \]
  
  \[\Rightarrow [1, n, n*n, n*n*n, \ldots]\]

- The stream of ‘function applications,’ the prelude function \text{iterate}
  
  \[
  \text{iterate} :: (a \rightarrow a) \rightarrow a \rightarrow [a]
  \]
  
  \[
  \text{iterate} \ f \ x = x : \text{iterate} \ f \ (f \ x)
  \]
  
  \[\Rightarrow [x, f \ x, f \ (f \ x), f \ (f \ (f \ x)), \ldots]\]

- Application: Redefining \text{powers} in terms of \text{iterate}
  
  \[
  \text{powers} \ n = \text{iterate} \ (*n) \ 1
  \]
More Applications of iterate

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

powers = iterate (*n) 1
Functions on Streams

head :: [a] -> a
head (x:_ ) = x

Application: Generator/Selector pattern

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

Selector  Generator

Note: Normal order reduction (resp. its efficient implementation variant lazy evaluation) ensures termination. It avoids the infinite sequence of reductions of applicative order reduction:

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

...normal order reduction can be implemented as leftmost-outermost evaluation.

Example: Let \textit{ignore} be defined by

\begin{verbatim}
ignore :: a -> b -> b
ignore a b = b
\end{verbatim}

The leftmost-outermost operation of the term(s)

\begin{verbatim}
ignore twos 42 ≜ twos 'ignore' 42
\end{verbatim}

is given by \textit{ignore} (rather than by \textit{twos}).

“...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)
More Functions on Streams

addFirstTwo :: [Integer] -> Integer
addFirstTwo (x:y:zs) = x+y

Application: Generator/Selector pattern

\[
\begin{align*}
\text{addFirstTwo } & \text{ twos} \quad \Rightarrow \quad \text{addFirstTwo } (2: \text{twos}) \\
& \quad \Rightarrow \quad \text{addFirstTwo } (2:2: \text{twos}) \\
& \quad \Rightarrow \quad 2+2 \\
& \quad \Rightarrow \quad 4
\end{align*}
\]
Functions yielding Streams

> **User-defined stream-yielding functions**

```haskell
from :: Int -> [Int]
from n = n : from (n+1)
```

```haskell
fromStep :: Int -> Int -> [Int]
fromStep n m = n : fromStep (n+m) m
```

**Applications**

```haskell
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,...
```

> The stream primes of prime numbers...
Primes: The Sieve of Eratosthenes (1)

Intuition

1. Write down the natural numbers starting at 2.
2. The smallest number not yet cancelled is a prime number. Cancel all multiples of this number.
3. Repeat Step 2 with the smallest number not yet cancelled.

Illustration

Step 1:

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

Step 2 ("with 2"):

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

Step 2 ("with 3"):

2 3 5 7 11 13 17...

Step 2 ("with 5"):

2 3 5 7 11 13 17...

...
The stream of prime numbers \texttt{primes} (generator pattern):

\begin{Verbatim}
primes :: [Int]
primes = sieve [2..]
\end{Verbatim}

\texttt{sieve} :: [Int] \to [Int]
\begin{Verbatim}
sieve (x:xs) = x : sieve [ y | y <- xs, mod y x > 0]
\end{Verbatim}
Primes: The Sieve of Eratosthenes (3)

Illustrating the generator property by stepwise evaluation:

```plaintext
primes
-\rightarrow \text{sieve } [2..]
-\rightarrow 2 : \text{sieve } [ y \mid y \leftarrow [3..], \text{mod} \ y \ 2 > 0]
-\rightarrow 2 : \text{sieve } (3 : [ y \mid y \leftarrow [4..], \text{mod} \ y \ 2 > 0]
-\rightarrow 2 : 3 : \text{sieve } [ z \mid z \leftarrow [ y \mid y \leftarrow [4..],
\quad \text{mod} \ y \ 2 > 0],
\quad \text{mod} \ z \ 3 > 0]
-\rightarrow \ldots
-\rightarrow 2 : 3 : \text{sieve } [ z \mid z \leftarrow [5, 7, 9..],
\quad \text{mod} \ z \ 3 > 0]
-\rightarrow \ldots
-\rightarrow 2 : 3 : \text{sieve } [5, 7, 11, \ldots
-\rightarrow \ldots
-\rightarrow [2, 3, 5, 7, 11, 13, 17, 19, \ldots
```
On Pitfalls in Applications with Streams

Implementing a prime number test (naively):

Consider

\[
\text{member} :: [a] \to a \to \text{Bool} \\
\text{member} [] y = \text{False} \\
\text{member} (x:xs) y = (x==y) || \text{member} xs y
\]
as a transforming selector (\text{a-value} to \text{Bool-value}).

Then

\[
\begin{align*}
\text{member\ primes\ 7} & \to \text{True} \quad \text{...as expected!} \\
t.\ \text{Selector: ...working properly!} \\
\text{member\ primes\ 8} & \to \ldots \quad \text{...does not terminate!} \\
t.\ \text{Selector: ...failing!}
\end{align*}
\]

Homework: Why does the generator/transf. selector implementation of \text{member} and \text{primes} fail? How can the transf. selector \text{member} be modified to work properly as a transf. selector?
Generating (Pseudo) Random Numbers

Generating a sequence of (pseudo) random numbers:

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

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

Choosing

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

we get a sequence of (pseudo-) random numbers beginning w/

\[ [17489, 59134, 9327, 52468, 43805, 8378, \ldots] \]

ranging from 0 to 65536, where all numbers of this interval occur with the same frequency.
Generator/Transformer Modularization

Often one needs to have random numbers within a range from \( p \) to \( q \) inclusive, \( p < q \).

This can be achieved by scaling the values of the sequence.

\[
\text{scale} :: \text{Float} \to \text{Float} \to \text{[Int]} \to \text{[Float]}
\]

\[
\text{scale} \ p \ q \ \text{randSeq} = \text{map} \ (f \ p \ q) \ \text{randSeq}
\]

where \( f :: \text{Float} \to \text{Float} \to \text{Int} \to \text{Float} \)

\[
f \ p \ q \ n = p + ((n \times (q-p)) / (\text{modulus}-1))
\]

Application: Generator/Transformer pattern

\[
\text{scale} \ 42.0 \ 51.0 \ \text{randomSequence}
\]

Transformer \hspace{1cm} Generator
Principles of Modularization

...related to streams:

- The Generator/Selector Principle
  ...e.g., computing the square root, the $n$-th Fibonacci number

- The Generator/Filter Principle
  ...e.g., computing all even Fibonacci numbers

- The Generator/Transformer Principle
  ...e.g., “scaling” random numbers

- Further combinations of generators, filters, and selectors
The Generator/Sel./Filt. Modulariz. Principle

...at a glance:

**Generator**
- \( \text{iterate } f \ x \)
- \( x, f \ x, f(f \ x),... \)

**Selector/Filter**
- \( \text{select } p \)
- \( x, y, z, ... \)
- \([ q \mid q \leftarrow [x, y, z, ...] \),
  \( \text{select } p \ q == \text{True} \)]

**Linking Generator and Selector/Filter together**
- \( \text{iterate } f \ x \)
- \( x, f \ x, f(f \ x),... \)
- \([ q \mid q \leftarrow [x, f \ x, f(f \ x),...] \),
  \( \text{select } p \ q == \text{True} \)]
The Generator/Transf. Modulariz. Principle

...at a glance:

**Generator**

iterate $f(x)$

$x, f(x), f(f(x)), ...$

**Transformer**

map $g$

$x, y, z, ...$

g$x, g(y), g(z), ...$

**Linking Generator and Transformer together**

iterate $f(x)$

map $g$

$x, f(x), f(f(x)), ...$

g$x, g(f(x)), g(f(f(x))), ...$
The Fibonacci Numbers (1)

Recall: The stream of Fibonacci Numbers

\[ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots \]

relying on the function

\( \text{fib} : \mathbb{N}_0 \rightarrow \mathbb{N}_0 \)

\[
\text{fib}(n) = \begin{cases} 
0 & \text{if } n = 0 \\
1 & \text{if } n = 1 \\
fib(n - 1) + fib(n - 2) & \text{if } n \geq 2 
\end{cases}
\]
The Fibonacci Numbers (2)

We learned (LVA 185.A03 FP) that a naive implementation like

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

...which directly exploits the recursive pattern of the underlying mathematical function is

- inacceptably inefficient and slow!
The Fibonacci Numbers (3)

Illustration: By stepwise evaluation

\[ \text{fib} \ 0 \rightarrow \ 0 \quad \text{-- 1 call of fib} \]

\[ \text{fib} \ 1 \rightarrow \ 1 \quad \text{-- 1 call of fib} \]

\[ \text{fib} \ 2 \rightarrow \ \text{fib} \ 1 \ + \ \text{fib} \ 0 \]
\[ \quad \rightarrow \ 1 \ + \ 0 \]
\[ \quad \rightarrow \ 1 \quad \text{-- 3 calls of fib} \]

\[ \text{fib} \ 3 \rightarrow \ \text{fib} \ 2 \ + \ \text{fib} \ 1 \]
\[ \quad \rightarrow \ (\text{fib} \ 1 \ + \ \text{fib} \ 0) \ + \ 1 \]
\[ \quad \rightarrow \ (1 \ + \ 0) \ + \ 1 \]
\[ \quad \rightarrow \ 2 \quad \text{-- 5 calls of fib} \]
The Fibonacci Numbers (4)

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

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
The Fibonacci Numbers (5)

\[
\begin{align*}
\text{fib 8} & \rightarrow\rightarrow \text{fib 7} + \text{fib 6} \\
& \rightarrow\rightarrow (\text{fib 6} + \text{fib 5}) + (\text{fib 5} + \text{fib 4}) \\
& \rightarrow\rightarrow (((\text{fib 5} + \text{fib 4}) + (\text{fib 4} + \text{fib 3})) \\
& \quad + ((\text{fib 4} + \text{fib 3}) + (\text{fib 3} + \text{fib 2}))) \\
& \rightarrow\rightarrow (((\text{fib 4} + \text{fib 3}) + (\text{fib 3} + \text{fib 2})) \\
& \quad + (\text{fib 3} + \text{fib 2}) + (\text{fib 2} + \text{fib 1}))) \\
& \quad + (((\text{fib 3} + \text{fib 2}) + (\text{fib 2} + \text{fib 1})) \\
& \quad \quad + ((\text{fib 2} + \text{fib 1}) + (\text{fib 1} + \text{fib 0})))
\end{align*}
\]

\[
\rightarrow\rightarrow \ldots
\]

\[
\rightarrow\rightarrow 21 \quad -- \quad 60 \text{ calls of fib}
\]

\ldots tree-like recursion (with \textit{exponential growth!}).
Recall (LVA 185.A03 FP): Complexity (1)

For further details, refer to:


**O Notation**

Let $f : \alpha \to IR^+$ be a function with some data type $\alpha$ as domain and the set of positive real numbers as range. Then the class $O(f)$ denotes the set of all functions which “grow slower” than $f$:

\[
O(f) = \{ h \mid h(n) \leq c \ast f(n) \text{ for some positive constant } c \text{ and all } n \geq N_0 \}
\]
...important cost functions:

<table>
<thead>
<tr>
<th>Class</th>
<th>Costs</th>
<th>Intuition: input a thousandfold as large means:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(O(c))</td>
<td>constant</td>
<td>...equal effort</td>
</tr>
<tr>
<td>(O(\log n))</td>
<td>logarithmic</td>
<td>...only tenfold effort</td>
</tr>
<tr>
<td>(O(n))</td>
<td>linear</td>
<td>...also a thousandfold effort</td>
</tr>
<tr>
<td>(O(n \log n))</td>
<td>quasi-linear</td>
<td>...tenthousandfold effort</td>
</tr>
<tr>
<td>(O(n^2))</td>
<td>quadratic</td>
<td>...millionfold effort</td>
</tr>
<tr>
<td>(O(n^3))</td>
<td>cubic</td>
<td>...billionfold effort</td>
</tr>
<tr>
<td>(O(n^c))</td>
<td>polynomial</td>
<td>...gigantic much effort (for big (c))</td>
</tr>
<tr>
<td>(O(2^n))</td>
<td>exponential</td>
<td>...hopeless</td>
</tr>
</tbody>
</table>
Complexity (3)

...the impact of growing inputs in practice:

<table>
<thead>
<tr>
<th>n</th>
<th>Linear</th>
<th>Quadratic</th>
<th>Cubic</th>
<th>Exponential</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 µs</td>
<td>1 µs</td>
<td>1 µs</td>
<td>2 µs</td>
</tr>
<tr>
<td>10</td>
<td>10 µs</td>
<td>100 µs</td>
<td>1 ms</td>
<td>1 ms</td>
</tr>
<tr>
<td>20</td>
<td>20 µs</td>
<td>400 µs</td>
<td>8 ms</td>
<td>1 s</td>
</tr>
<tr>
<td>30</td>
<td>30 µs</td>
<td>900 µs</td>
<td>27 ms</td>
<td>18 min</td>
</tr>
<tr>
<td>40</td>
<td>40 µs</td>
<td>2 ms</td>
<td>64 ms</td>
<td>13 days</td>
</tr>
<tr>
<td>50</td>
<td>50 µs</td>
<td>3 ms</td>
<td>125 ms</td>
<td>36 years</td>
</tr>
<tr>
<td>60</td>
<td>60 µs</td>
<td>4 ms</td>
<td>216 ms</td>
<td>36 560 years</td>
</tr>
<tr>
<td>100</td>
<td>100 µs</td>
<td>10 ms</td>
<td>1 sec</td>
<td>4 * 10^{16} years</td>
</tr>
<tr>
<td>1000</td>
<td>1 ms</td>
<td>1 sec</td>
<td>17 min</td>
<td>very, very long...</td>
</tr>
</tbody>
</table>
Streams to the Rescue

Stream programming

▶ can (often) help to conquer complexity!
The Stream of Fibonacci Numbers: Efficiently

Idea

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 efficiently 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
Applications: Generator/Selector Pattern

Generator

\[ \text{fibs} \rightarrow 0 : 1 : 1 : 2 : 3 : 5 : 8 : 13 : 21 : 34 : 55 : 89 \ldots \]

Generator/Selector

\[ \text{take 5 fibs} \rightarrow [0, 1, 1, 2, 3] \]

where

\[
\begin{align*}
\text{take} & \quad :: \quad \text{Int} \rightarrow \ [a] \rightarrow \ [a] \\
\text{take 0 } _ & \quad = \quad [] \\
\text{take } _ \ [ ] & \quad = \quad [] \\
\text{take } n \ (x:xs) \mid n>0 & \quad = \quad x : \text{take} \ (n-1) \ xs \\
\text{take } _ \ _ & \quad = \quad \text{error} \ "\text{Negative argument}" 
\end{align*}
\]
From Stream \texttt{fibs} to Function \texttt{fib}

...the \textit{corecursive} definition of the stream \texttt{fibs} suggests a conceptually new implementation of the \texttt{Fibonacci} function \texttt{fibs}:

\begin{verbatim}
fib :: Int -> Int
fib n = last (take n fibs)
\end{verbatim}

\begin{itemize}
\item \texttt{Selector 2}
\item \texttt{Selector 1}
\item \texttt{Generator}
\end{itemize}

Even shorter with only one selector:

\begin{verbatim}
fib :: Int -> Int
fib n = fibs !! (n-1)
\end{verbatim}

\begin{itemize}
\item \texttt{Generator}
\item \texttt{Selector}
\end{itemize}

\textbf{Note} the application of the generator(selector) modularization in these two examples.
Lazy Evaluation is Essential for Performance

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

fibs

\[
\begin{align*}
\text{->>} & \quad \{ \text{Replace the call of fibs by the body of fibs} \} \\
0 : 1 : \text{add fibs (tail fibs)}
\end{align*}
\]

\[
\text{->>} \quad \{ \text{Replace both calls of fibs by the body of fibs} \}
\]

\[
\begin{align*}
0 : 1 & : \text{add} (0 : 1 : \text{add fibs (tail fibs)}) \\
(\text{tail} (0 : 1 : \text{add fibs (tail fibs)}))
\end{align*}
\]

\[
\text{->>} \quad \{ \text{Application of tail} \}
\]

\[
\begin{align*}
0 : 1 & : \text{add} (0 : 1 : \text{add fibs (tail fibs)}) \\
(1 : \text{add fibs (tail fibs)})
\end{align*}
\]

\[
\text{->>} \quad \ldots \text{ exponential effort!}
\]

...lazy evaluation ensures that common subexpressions (here, tail and fibs) are not computed multiple times!
Sharing: The Benefit of Lazy Evaluation (1)

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

   ->> \{ Introd. abbrev. allows sharing of results \}
    0 : tf  \hspace{1em} -- \hspace{1em} 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 \( tf = 1 : tf2 \) \hspace{1em} -- \hspace{1em} tf2 reminds to "tail of tail of fibs"
    \hspace{1em} where \( tf2 = add \) fibs tf

   ->> \{ Unfolding of add \}
    0 : tf
    where \( tf = 1 : tf2 \)
    \hspace{1em} where \( tf2 = 1 : add \hspace{1em} tf \hspace{1em} tf2 \)
Sharing: The Benefit of Lazy Evaluation (2)

-\to \{\text{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

-\to 0 : tf
  where tf = 1 : tf2
    where tf2 = 1 : tf3
      where tf3 = 2 : add tf2 tf3

-\to \{tf is only used once and can thus be eliminated\}
  0 : 1 : tf2
  where tf2 = 1 : tf3
    where tf3 = 2 : add tf2 tf3
Sharing: The Benefit of Lazy Evaluation (3)

$$\rightarrow\{\text{Finally, we obtain successsively longer pre-fixes of the stream of Fibonacci numbers}\}$$

$$0 : 1 : \text{tf2}$$

where $\text{tf2} = 1 : \text{tf3}$

where $\text{tf3} = 2 : \text{tf4}$

where $\text{tf4} = \text{add tf2 tf3}$

$$\rightarrow\ 0 : 1 : \text{tf2}$$

where $\text{tf2} = 1 : \text{tf3}$

where $\text{tf3} = 2 : \text{tf4}$

where $\text{tf4} = 3 : \text{add tf3 tf4}$

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

$$\rightarrow\ 0 : 1 : 1 : \text{tf3}$$

where $\text{tf3} = 2 : \text{tf4}$

where $\text{tf4} = 3 : \text{add tf3 tf4}$
Note

...in practice, the ability of recognizing common structures is limited.

For illustration, consider the below variant `FibsFn` of the Fibonacci function that artificially lifts `fibs` to a functional level:

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

Evaluating `FibsFn` shows

- exponential run-time and storage usage!

Memory leak:

- The memory space is consumed so fast that the performance of the program is significantly impacted.
Illustration

\[
\text{fibsFn ()} \\

\rightarrow \rightarrow 0 : 1 : \text{add (fibsFn ()) (tail (fibsFn ()))} \\

\rightarrow \rightarrow 0 : \text{tf} \\
\hspace{1cm} \text{where} \\
\hspace{2cm} \text{tf} = 1 : \text{add (fibsFn ()) (tail (fibsFn ()))}
\]

The equality of \(\text{tf}\) and \(\text{tail(fibsFn())}\) remains undetected. Hence, the following simplification is not done:

\[
\rightarrow \rightarrow 0 : \text{tf} \\
\hspace{1cm} \text{where} \text{ tf} = 1 : \text{add (fibsFn ()) tf}
\]

**Note:** While for a special case like here, this might be possible, there is no general means for detecting such equalities!
Chapter 2.2
Stream Diagrams
Stream Diagrams

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

- processes.

In this chapter, we consider two examples for illustration: The stream of

- Fibonacci numbers
- communications of some client/server application
Example 1: Fibonacci Numbers

...as a stream diagram:
Example 2: A Client/Server Application (1)

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

type Request = 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 Generators and their Interaction

reqs = client resps -- Generator
resps = server reqs -- Generator
Example 2: A Client/Server Application (2)

\[
\text{reqs } \rightarrow\rightarrow \text{ client resps} \\
\rightarrow\rightarrow 1 : \text{ resps} \\
\rightarrow\rightarrow 1 : \text{ server reqs} \\
\rightarrow\rightarrow \{\text{Introducing abbreviations}\} \\
\quad 1 : \text{ tr} \\
\quad \text{ where } \text{ tr} = \text{ server reqs} \\
\rightarrow\rightarrow 1 : \text{ tr} \\
\quad \text{ where } \text{ tr} = 2 : \text{ server tr} \\
\rightarrow\rightarrow 1 : \text{ tr} \\
\quad \text{ where } \text{ tr} = 2 : \text{ tr2} \\
\quad \quad \quad \text{ where } \text{ tr2} = \text{ server tr} \\
\rightarrow\rightarrow 1 : \text{ tr} \\
\quad \text{ where } \text{ tr} = 2 : \text{ tr2} \\
\quad \quad \quad \text{ where } \text{ tr2} = 3 : \text{ server tr2} \\
\rightarrow\rightarrow 1 : 2 : \text{ tr2} \\
\quad \text{ where } \text{ tr2} = 3 : \text{ server tr2} \\
\rightarrow\rightarrow \ldots
\]
Example 2: A Client/Server Application (3)

Application: Generator/Selector pattern

\[
\text{take 10 reqs } \rightarrow [1,2,3,4,5,6,7,8,9,10]
\]

Selector \quad Generator
Example 2: The Client/Server Application

...as a stream diagram:
**Excursus**

Suppose, the client wants to check the first response:

\[
\text{client } (y:ys) = \text{if ok } y \text{ then } 1 : (y:ys) \\
\text{else error "Faulty Server"}
\]

\[
\text{where ok } y = \text{True} \quad \text{-- Trivial check: ‘Always} \\
\text{-- succeeding’}
\]

**Note:** Evaluating

\[
\text{reqs } \rightarrow \rightarrow \text{ client resps} \\
\quad \rightarrow \rightarrow \text{ client (server reqs)} \\
\quad \rightarrow \rightarrow \text{ client (server (client resps))} \\
\quad \rightarrow \rightarrow \text{ client (server (client (server reqs)))} \\
\quad \rightarrow \rightarrow \ldots
\]

...does not terminate!

**The problem:** Livelock! Neither the client nor the server can be unfolded! Pattern matching is “too eager.”
Remedies: Selector Functions, Lazy Patterns

A): Selector Functions
Replacing pattern matching by selector function access (here `head`), and moving the conditional inside the list:

```haskell
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.

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

Note: The conditional must still be moved inside the list but the selector function is not needed. In practice, this can be very many calls of selector functions which are saved by lazy patterns making programs “more” declarative and readable.
Illustrating

...the effect of lazy patterns by stepwise evaluation:

\[
\text{client } \sim (y:ys) = 1 : \text{if ok } y \text{ then } y:ys \\
\quad \text{else error } "\text{Faulty Server}" \\
\]

\[
\text{reqs } \rightarrow\rightarrow \text{ client resps} \\
\quad \rightarrow\rightarrow 1 : \text{if ok } y \text{ then } y:ys \\
\quad \quad \text{else error } "\text{Faulty Server}" \\
\quad \text{where } y:ys = \text{resps} \\
\quad \rightarrow\rightarrow 1 : (y:ys) \\
\quad \text{where } y:ys = \text{resps} \\
\quad \rightarrow\rightarrow 1 : \text{resps}
\]
Chapter 2.3

Memoization
Memoization is

- a means for improving the performance of (functional) programs by avoiding costly recomputations

which benefits from

- stream programming.
Memoization

The concept of memoization goes back to Donald Michie:


Idea

- 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 (1)

A memo function is

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

A memo table allows

- to replace recomputation by table look-up.

Soundness of the overall approach:

- Referential transparency of functional programming languages (especially, absence of side effects!).
Memo Functions, Memo Tables (2)

Requirement

Let $f : a \to b$ be a function. A memo function $\text{memo}$

$$\text{memo} :: (a \to b) \to (a \to b)$$

for replacing $f$ must be defined such that the following equality holds:

$$\text{memo } f \ x = f \ x$$
Making it Concrete: Memo Lists

...as memo tables.

Let \( f : \text{Nat} \to b \) be a (costly to compute) function on natural numbers.

Replace every call of \( f \) by a look-up in \( f\text{\_memolist} \), which can be considered a \( \text{generic} \) memo list, defined by

\[
\text{f\_memolist} = \left[ f \, x \mid x \leftarrow [0..] \right] \quad -- \text{Generator}
\]
Example 1: Computing Fibonacci Numbers

Computing Fibonacci numbers with memoization/memo lists:

\[
\text{fib\_memolist} = [\text{fib } x \mid x \leftarrow [0..]] \\
\text{fib 0 } = 0 \\
\text{fib 1 } = 1 \\
\text{fib } n = \text{fib\_memolist}!!(n-1) + \text{fib\_memolist}!!(n-2)
\]

Compare this with the naive implementation of fib:

\[
\text{fib\_naive } 0 = 0 \\
\text{fib\_naive } 1 = 1 \\
\text{fib\_naive } n = \text{fib\_naive } (n-1) + \text{fib\_naive } (n-2)
\]

Lemma 2.3.1

\[
\forall n \in \mathbb{N}. \text{fib } n = \text{fib\_naive } n
\]
Example 2: Computing Powers

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

\[
\text{pow_memolist} = [\text{power } x \mid x \leftarrow [0..]]
\]

\[
\text{power } 0 = 1
\]

\[
\text{power } i = \text{pow_memolist}!!(i-1) + \text{pow_memolist}!!(i-1)
\]

Compare this with the naive implementation of \text{power}:

\[
\text{power_naive } 0 = 1
\]

\[
\text{power_naive } i = \text{power_naive } (i-1) + \text{power_naive } (i-1)
\]

Lemma 2.3.2

\[
\forall n \in \mathbb{N}. \text{ power } n = \text{power_naive } n
\]

Note: Looking-up the result of the second call instead of re-computing it requires only \(1 + n\) calls of \text{power} instead of \(1 + 2^n\). This results in a significant performance gain!
Summing up (1)

A memo function \texttt{memo} :: (a \rightarrow b) \rightarrow (a \rightarrow b)

\begin{itemize}
  \item is essentially the identity on functions but
  \item keeps track on the arguments it has been applied to and their corresponding result values
\end{itemize}

\textbf{Motto}: Look-up a result which has been computed before instead of recomputing it!

\textbf{Memo functions} are

\begin{itemize}
  \item not a part of the Haskell standard but
  \item are supported by some non-standard libraries.
\end{itemize}
Summing up (2)

**Important design decision**

- when implementing **memo functions**: how many argument/result pairs shall be traced (e.g., a memo function `memo1` for one argument/result pair)?

**Example:**

```haskell
memo_fibsFn :: () -> [Integer]
memo_fibsFn x
    = let mfibs = memo1 memo_fibsFn in
        0 : 1 : zipWith (+) (mfibs ()) (tail (mfibs ()))
```

`memo1`
Summing up (3)

More on memoization, its very idea and application, e.g., in:

► Chapter 19, Memoization

► Chapter 12.3, Memoization
Summing up (4)

  ...introduced streams without memoization.

  ...extended Landin’s streams with memoization.

  ...extended Landin’s streams with memoization.
Chapter 2.4

Boosting Performance
Motivation

Recomputing values unnecessarily is a major source of inefficiency:

▶ Avoiding recomputations of values is a major source of improving the performance of a program.

Techniques which can (often) help achieving this are:

▶ Stream programming
▶ Memoization
Avoiding Recomputations using Stream Prog.

- **Computing Fibonacci numbers using stream programming:**
  
  ```haskell
  fibs :: [Integer] -- Generator
  fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
  
  Applications: Generator/Selector pattern
  take 10 fibs ->> [0,1,1,2,3,5,8,13,21,34]
  fibs!!5 ->> 5
  ```

- **Computing powers using stream programming:**
  
  ```haskell
  powers :: [Integer] -- Generator
  powers = 1 : 2 : zipWith (+) (tail powers) (tail powers)
  
  Applications: Generator/Selector pattern
  take 9 powers ->> [1,2,4,8,16,32,64,128,256]
  powers!!5 ->> 32
  ```

- ...
Avoiding Recomputations using Memoization

- Computing Fibonacci numbers using memoization:
  
  ```haskell
  fib_list = [ fib x | x <- [0..] ] -- Generator
  fib 0 = 0
  fib 1 = 1
  fib n = fib_list!!(n-1) + fib_list!!(n-2)
  
  Applications: Generator/Selector pattern
  take 10 fib_list ->> [0,1,1,2,3,5,8,13,21,34]
  fiblist!!5 ->> 5
  ```

- Computing powers using memoization:
  
  ```haskell
  power_list = [ power x | x <- [0..] ] -- Generator
  power 0 = 1
  power i = power_list!!(i-1) + power_list!!(i-1)
  
  Applications: Generator/Selector pattern
  take 9 power_list ->> [1,2,4,8,16,32,64,128,256]
  power_list!!5 ->> 32
  ```

- ...
Summing up

Stream programming and memoization are important though
► no silver bullets
for improving performance by avoiding recomputations.

If, however, they hit they can significantly
► boost performance: from taking too long to be feasible to
be completed in an instant!

Obvious candidates
► problems that naturally wind up repeatedly computing the
the solution to identical subproblems, e.g. tree-recursive
processes.

Homework: Compare the run-time performance of the
straightforward implementations of fib and power with the
one of their “boosted” versions using stream programming and
memoization.
Sometimes a Silver Bullet exists

Though not in general, sometimes a silver bullet solving a problem exists.

Computing Fibonacci numbers provides (again) a striking example.

The equality of Theorem 2.4.1 (cf. Chapter 6) allows a recursion-free direct computation of the Fibonacci numbers, i.e.,

\[(fib_i)_{i \in \mathbb{N}_0} = (0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots)\]

Theorem 2.4.1

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

Using **streams** (together w/ **lazy evaluation**) is advocated by:

- **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
  - **Generator/Prune Paradigm**

  of which the
  - **Generator/selector**
  - **Generator/filter**
  - **Generator/transformer** principle

  and combinations thereof are specific instances of.

- **Boosting performance**: By avoiding recomputations. Most important are:
  - **Stream programming**
  - **Memoization**
Chapter 2.5

References, Further Reading
Chapter 2: Further Reading (1)


Chapter 2: Further Reading (2)


Chapter 2: Further Reading (3)


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)

Chapter 2: Further Reading (4)


Chapter 2: Further Reading (5)


Chapter 2: Further Reading (6)


- Peter Pepper, Petra Hofstedt. Funktionale Programmierung. Springer-V., 2006. (Kapitel 14.2.1, Memoization; Kapitel 15.5, Maps, Funktionen und Memoization)


Chapter 2: Further Reading (7)


Chapter 3

Programming with Higher-Order Functions: Algorithm Patterns
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 emphasises 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 alternatives.
  - Divide-and-conquer (cf. LVA 185.A03 FV, 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
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.
Illustrating the Divide-and-Conquer Principle

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

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:

\[
\text{divide\_and\_conquer} :: (p \to \text{Bool}) \to (p \to s) \to \\
\begin{align*}
\text{Simple enough?} & \quad \text{Solve!} \\
(p \to \text{[p]}) \to (p \to \text{[s]} \to s) \to \\
\text{Divide} & \quad \text{Combine}
\end{align*}
\]

\[
\text{Problem instance} \quad \text{Solution}
\]

\[
\text{divide\_and\_conquer} \; \text{indiv} \; \text{solve} \; \text{divide} \; \text{combine} \; \text{pbi}
\]
\[
= \text{dac} \; \text{pbi}
\]
\[
\text{where}
\]
\[
\text{dac} \; \text{pbi'}
\]
\[
| \text{indiv} \; \text{pbi'} = \text{solve} \; \text{pbi'}
\]
\[
| \text{otherwise} = \text{combine} \; \text{pbi'} \; (\text{map} \; \text{dac} \; (\text{divide} \; \text{pbi'}))
\]
Typical Applications of Divide-and-Conquer

Application fields such as

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

Especially

- Quicksort
- Mergesort
- Binomial coefficients
- ...

177/192
Example 1: Quicksort

\[
\text{quickSort} : \text{Ord a} \Rightarrow [\text{a}] \rightarrow [\text{a}]
\]

\[
\text{quickSort} \; \text{lst} = \text{divide_and_conquer} \; \text{indiv} \; \text{solve} \; \text{divide} \; \text{combine} \; \text{lst}
\]

where

\[
\begin{align*}
\text{indiv} \; \text{ls} & = \text{length} \; \text{ls} \leq 1 \\
\text{solve} & = \text{id} \\
\text{divide} \; (l:ls) & = \left[ \left[ x \mid x \leftarrow ls, x \leq l \right], \right. \\
& \left. \left[ x \mid x \leftarrow ls, x > l \right] \right] \\
\text{combine} \; (l:_) \; [l1,l2] & = l1 \; ++ \; [l] \; ++ \; l2
\end{align*}
\]
Example 2: Fibonacci Numbers (Pitfall!)

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

Consider:

\[
\text{fib} :: \text{Integer} \rightarrow \text{Integer} \\
\text{fib} \ n \\
= \text{divide_and_conquer} \ \text{indiv} \ \text{solve} \ \text{divide} \ \text{combine} \ n \\
\text{where} \\
\text{indiv} \ n = (n == 0) || (n == 1) \\
\text{solve} \ n \\
| n == 0 = 0 \\
| n == 1 = 1 \\
| \text{otherwise} = \text{error} \ "\text{Problem must be divided}" \\
\text{divide} \ n = [n-2,n-1] \\
\text{combine} \ _ \ [l1,l2] = l1 + l2
\]

...shows exponential runtime behaviour due to recomputations!
Illustrating

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

Chapter 3.2
Backtracking Search
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:

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 \texttt{node} representing information of nodes in the search space
- a successor function \texttt{succ} of type \texttt{(node \rightarrow [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]) ->
  (node -> Bool) ->
  node -> [node]
  Initial node  Solution nodes

search_dfs succ goal n  -- n for node
= (search (push n emptyS))
  -- s for stack
where
  search s
    | is_emptyS 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 \( \text{Stack} \) \((\text{user-visible})\), cf. Chapter 8.2:

\[
\text{module Stack \( (\text{Stack,emptyS,is_emptyS,push,pop,top)\) }
\]

\[
\text{where}
\]

\[
-- \quad \text{Interface Spec.: Signatures of stack operations}
\]

\[
\text{emptyS} \quad :: \quad \text{Stack} \ a
\]

\[
\text{is_emptyS} \quad :: \quad \text{Stack} \ a \rightarrow \text{Bool}
\]

\[
\text{push} \quad :: \quad a \rightarrow \text{Stack} \ a \rightarrow \text{Stack} \ a
\]

\[
\text{pop} \quad :: \quad \text{Stack} \ a \rightarrow \text{Stack} \ a
\]

\[
\text{top} \quad :: \quad \text{Stack} \ a \rightarrow \ a
\]

\[
-- \quad \text{Behaviour Spec.: Laws for stack operations}
\]

(1) thru (6) \quad -- \quad \text{cf. Chapter 8.2.}
Implementation A

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

```haskell
data Stack a = Empty | Stk a (Stack a)
emptyS = Empty
is_emptyS Empty = True
is_emptyS _ = 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):

```haskell
newtype Stack a = Stk [a]
emptyS = Stk []
is_emptyS (Stk []) = True
is_emptyS (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
- ...

Example: The Eight-Tile Problem (8TP)

A Backtracking Search Impl. for 8TP (1)

Modeling the board:

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

The initial board (initial configuration):

```haskell
    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):

```haskell
    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):

\[
\text{mandist} \quad : \quad \text{Position} \rightarrow \text{Position} \rightarrow \text{Int} \\
\text{mandist} \ (x1,y1) \ (x2,y2) = \text{abs} \ (x1-x2) + \text{abs} \ (y1-y2)
\]

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

\[
\text{allMoves} \quad : \quad \text{Board} \rightarrow \text{[Board]} \\
\text{allMoves} \ b = [b//[(0,b!i),(i,b!0)]] \\
\quad | \quad i<-\text{[1..8]}, \ \text{mandist} \ (b!0) \ (b!i)==1
\]

...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:

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

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

The successor function:

```haskell
succ8Tile :: Boards -> [BDS]
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:

```haskell
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:

```haskell
dfs8Tile :: [[Position]]
dfs8Tile = map elems ls
  where ((BDS ls):_)
    = search_dfs succ8Tile goal8Tile (BDS [s8T])
```
Chapter 3.3
Priority-first Search
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

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) $\text{search}_\text{pfs}$ solving
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 `<=` cannot 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]) ->
          (node -> Bool) ->
          node -> [node]

search_pfs succ goal n
  = search (enPQ n emptyPQ)
    where
      search pq
        = let m = frontPQ pq
            in search (foldr enPQ (dePQ pq) (succ m))
```

- **Computing successors**
- **Solution?**
- **Initial node**
- **Solution nodes**
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
...
Implementation

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

```haskell
newtype PQQueue 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 (e:r) = e:insert x r' where
                     insert x r@(_:e:r') | x <= e = x:r  -- the smaller the
                     | otherwise = e:insert x r' -- higher the priority

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
- ...

Application fields such as

- Game strategies
- ...

Especially

- The eight-tile problem
- ...
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.

```haskell
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
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:

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:

\[
\begin{align*}
\text{search\_greedy} & : (\text{Ord node}) \to (\text{node} \to \text{[node]}) \to \\
& \quad \textbf{Computing successors} \quad (\text{node} \to \text{Bool}) \to \\
& \quad \textbf{Solution?} \\
& \quad \text{node} \to \text{[node]} \\
& \quad \textbf{Initial node} \quad \textbf{Solution nodes}
\end{align*}
\]

\[
\text{search\_greedy succ goal n} \quad -- \text{n for node}
\]

\[
= \text{search (enPQ n emptyPQ)} 
\]

\[
\text{where}
\]

\[
\text{search pq} \quad -- \text{pq for priority queue}
\]\[
| \text{is\_emptyPQ pq} = [] \\
| \text{goal (frontPQ pq)} = [\text{frontPQ pq}] \\
| \text{otherwise}
\]

\[
= \text{let m = frontPQ pq}
\]

\[
in \text{search (foldr enPQ emptyPQ (succ m))}
\]
...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:

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

Cf. Chapter 3.3 and 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:

```haskell
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):

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

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

```haskell
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:

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

Putting things together:

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

Example: `change 199` $$\rightarrow$$ [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` $$\rightarrow$$ [30,12,6] instead of the optimal solution [24,24].
Chapter 3.5
Dynamic Programming
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 \texttt{fib}

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

Fethi Rabhi, Guy Lapalme. 

\textit{Algorithms: A Functional Programming Approach}. 
Addison-Wesley, 1999, page 179.
Illustrating Divide-and-Conquer for \texttt{fib}

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

\begin{center}
\begin{tikzpicture}
\node {fib 4} [grow=right, sibling distance=3.5cm, level distance=1.5cm] {
  \node {fib 3} [grow=left, sibling distance=3.5cm, level distance=1.5cm] {
    \node {fib 2} [grow=left, sibling distance=3.5cm, level distance=1.5cm] {
      \node {fib 1} [grow=left, sibling distance=3.5cm, level distance=1.5cm] {
        \node {fib 1} [grow=left, sibling distance=3.5cm, level distance=1.5cm] {fib 1}
      }
    }
  }
  \node {fib 2} [grow=right, sibling distance=3.5cm, level distance=1.5cm] {
    \node {fib 1} [grow=right, sibling distance=3.5cm, level distance=1.5cm] {
      \node {fib 0} [grow=right, sibling distance=3.5cm, level distance=1.5cm] {fib 0}
    }
  }
};
\end{tikzpicture}
\end{center}

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.
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`.
Implementing Dynamic Programming as HoF (3)

The HoF Implementation:

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

*Computing the table entry at some coordinates*  
Specifying table bounds  
Result table

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

```haskell
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
- ...


Example: Computing Fibonacci Numbers

Defining the problem-dependent parameters:

```haskell
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:

```haskell
fib :: Int -> Int
fib n = find t n
  where t = dynamic compFib (bndsFibs n)
```
Dynamic Programming vs. Memoization (1)

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).
Dynamic Programming vs. Memoization (2)

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 3.6

References, Further Reading
Chapter 3.1–3.4: Further Reading (1)


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)

Chapter 3.1–3.4: Further Reading (3)


Chapter 3.5: Further Reading (4)


Chapter 3.5: Further Reading (5)


Jon Kleinberg, Éva Tardos. *Algorithm Design*. Addison-Wesley/Pearson, 2006. (Chapter 6, Dynamic Programming)

Chapter 3.5: Further Reading (6)

- Fethi Rabhi, Guy Lapalme. *Algorithms – A Functional Programming Approach*. Addison-Wesley, 1999. (Chapter 5, Abstract data types; Chapter 9, Dynamic programming)


Chapter 3.5: Further Reading (7)


Chapter 4
Equational Reasoning
Chapter 4.1

Motivation
Functional vs. Imperative Programming (1)

In functional programming

- $\equiv$ means ‘equal by definition:’ The value of the left-hand side expression is defined as the value of the right-hand side expression.

- **Functional definitions** of the form

  $$ f \ x \ y = \ldots $$

  in the definition of a function $f$ are thus **genuine mathematical equations**. The expressions on the left hand side and the right hand side of $\equiv$ have the **same value**.
Functional vs. Imperative Programming (2)

In imperative programming

▶ = means ‘equality by assignment:’ The contents of the memory cell denoted by the left-hand side variable is replaced by the value of the right-hand side expression.

▶ A symbol sequence of the form

\[ x \equiv x + y \]

does not represent a mathematical equation meaning that \( x \) and \( x + y \) have the same value but an instruction, a command, a destructive assignment statement meaning that the old value of \( x \) is destroyed and replaced by the value of \( x + y \).

Note: To avoid confusion some imperative languages use thus a different symbol, e.g. := such as in Pascal, to denote the assignment operator (instead of the conceptually misleading symbol =).
Functional vs. Imperative Programming (3)

Example: 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 or Java, $S$ is a

▶ valid sequence of destructive assignment statements meaning that after executing $S$ the memory cells named by $x$ and $y$ store the values 3 and 2, respectively. No error is raised.
Functional vs. Imperative Programming (4)

Summarizing:

For functional definitions
- standard (algebraic) reasoning about mathematical equations applies.

For imperative assignments
- it does not.

Reasoning about functional definitions and programs is thus a lot easier than about imperative assignments and programs.
Illustrating Equational Reasoning

...on expressions.

Proposition 4.1.1

\[(a + b) \ast (a - b) = a^2 - b^2\]

Proof: By equational reasoning we obtain:

\[(a + b) \ast (a - b)\]

(Distributivity of \(\ast, +\))

\[= a \ast a - a \ast b + b \ast a - b \ast b\]

(Commutativity of \(\ast\))

\[= a \ast a - a \ast b + a \ast b - b \ast b\]

\[= a \ast a - b \ast b\]

\[= a^2 - b^2\]

\[\square\]
Illustrating Equational Reasoning

...on functional definitions.

Corollary 4.1.2

The Haskell functions \( f \) and \( g \) defined by

\[
\begin{align*}
f & : \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \\
f \ a \ b &= (a+b) \times (a-b)
\end{align*}
\]

\[
\begin{align*}
g & : \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \\
g \ a \ b &= a^2 - b^2
\end{align*}
\]

denote the same function.

Proof: By equational reasoning and Proposition 4.1.1 we obtain:

\[
\begin{align*}
f \ a \ b &= (a+b) \times (a-b) \\
&= a^2 - b^2 \\
&= g \ a \ b \ 
\end{align*}
\]

\[\square\]
More Examples on Equational Reasoning (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.,
\[ h \ a \ (h \ a \ b) = 634. \]
More Examples on Equational Reasoning (2)

Proof: By equational reasoning using the functional definitions of \( h \), \( a \), and \( b \) we obtain:

\[
\begin{align*}
= & \quad h \ a \ (h \ a \ b) \\
(\text{Def. of } h, \text{ unfolding } h) & = h \ a \ (a^2 + b^2) \\
(\text{Definition of } a, b) & = h \ 3 \ (3^2 + 4^2) \\
& = h \ 3 \ (9 + 16) \\
& = h \ 3 \ 25 \\
(\text{Def. of } h, \text{ unfolding } h) & = 3^2 + 25^2 \\
& = 9 + 625 \\
& = 634 \quad \Box
\end{align*}
\]

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 \( b \).
More Examples on Equational Reasoning (3)

Let

\[
g :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}
g \ x \ y = x^2 - y^2
\]

\[
k :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}
k \ x \ y = x \times y
\]

**Proposition 4.1.4**

The expressions \(k (a+b) (a-b)\) and \(g \ a \ b\) have the same value, i.e., \(k (a+b) (a-b) = g \ a \ b\).
More Examples on Equational Reasoning (4)

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

$$k (a+b) (a-b)$$

(Def. of $k$, unfolding $k$) $=$ $(a+b) \ast (a-b)$

(Distributivity of $\ast$, $+$) $=$ $a*a - a*b + b*a - b*b$

(Commutativity of $\ast$) $=$ $a*a - a*b + a*b - b*b$

$=$ $a*a - b*b$

$=$ $a^2 - b^2$

(Def. of $g$, folding $g$) $=$ $g \, a \, b$  $\square$
Folding, Unfolding of Functional Definitions

...as demonstrated in the proof of Proposition 4.1.4, functional definitions can be applied from

- left-to-right, called unfolding
- right-to-left, called folding

in equational reasoning.
Note

...some care on folding/unfolding needs to be taken though.

Let

\[
\text{isZero} :: \text{Int} \rightarrow \text{Bool} \\
\text{isZero} \ 0 = \text{True} \\
\text{isZero} \ n = \text{False}
\]

While the first equation \(\text{isZero} \ 0 = \text{True}\)

▶ can be viewed as a logical property and freely be applied in both directions

the second equation \(\text{isZero} \ n = \text{False}\)

▶ can not, since Haskell implicitly imposes an ordering on the equations: Applying the second equation is only legal, if \(n\) is different from \(0\).
Equational Reasoning for Optimization (1)

Note, the straightforward implementation of reverse

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

requires $\frac{n(n+1)}{2}$ calls of the concatenation function (++) , where $n$ denotes the length of the argument list.

fast_reverse , which does not depend on list concatenation (++) but on list construction (:) is much more efficient:

fast_reverse :: [a] -> [a]
fast_reverse xs = fr xs []
    where fr [] ys = ys
          fr (x:xs) ys = fr xs (x:ys)
Equational Reasoning for Optimization (2)

Replacing \texttt{reverse} by \texttt{fast\_reverse} would yield a significant speed-up of programs, provided that \texttt{reverse} and \texttt{fast\_reverse} denote actually the same function.

Using equational reasoning we can in fact prove the equality of \texttt{reverse} and \texttt{fast\_reverse}, and hence justify the above sketched optimisation:

\textbf{Theorem 4.1.5}

The functions \texttt{reverse} and \texttt{fast\_reverse} denote the same function, i.e.,

\[
\forall \texttt{ls} \in \texttt{a-List}. \quad \texttt{reverse ls} = \texttt{fast\_reverse ls}
\]
Equational Reasoning for Optimization (3)

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:

\[
\text{reverse } ls \\
(ls = \[]) = \text{reverse } \[] \\
\text{(Unfolding reverse) } = \[] \\
\text{(Folding fr) } = fr \[] \[] \\
\text{(Folding fast_reverse) } = \text{fast_reverse } \[] \\
(\[] = ls) = \text{fast_reverse } ls
\]
Equational Reasoning for Optimization (4)

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

\[
\begin{align*}
\text{reverse } ls \\ 
(lst = (v:ls')) &= \text{reverse } (v:ls') \\
\text{(Unfolding reverse)} &= \text{reverse } ls' ++ [v] \\
\text{(IH)} &= \text{fast_reverse } ls' ++ [v] \\
\text{(Unfolding fast_reverse)} &= (fr \text{ } ls' \text{ } []) ++ [v] \\
\text{(Corollary 4.1.7)} &= fr \text{ } ls' \text{ } [v] \\
\text{(Folding fr)} &= fr \text{ } ls' \text{ } (v:\text{[]}) \\
\text{(Folding fr)} &= fr \text{ } (v:ls') \text{ } [] \\
\text{(Folding fast_reverse)} &= \text{fast_reverse } (v:ls') \\
((v:lst') = ls) &= \text{fast_reverse } ls \quad \square
\end{align*}
\]
Equational Reasoning for Optimization (5)

Lemma 4.1.6

∀ ls1, ls2 ∈ a-List ∀ v ∈ a-Value.
(f r ls1 ls2) ++ [v] = f r ls1 (ls2 ++ [v])

Corollary 4.1.7

∀ ls′ ∈ a-List ∀ v ∈ a-Value.
(f r ls′ []) ++ [v] = f r ls′ [v]

Proof. Let ls′ ∈ a-List and let v ∈ a-Value. Setting ls1 = ls′ and ls2 = [], Lemma 4.1.6 yields:

(f r ls′ []) ++ [v]

(ls′ = ls1, [] = ls2) = (f r ls1 ls2) ++ [v]

(Lemma 4.1.6) = f r ls1 (ls2 ++ [v])

(ls1 = ls′, ls2 = []) = f r ls′ ([] ++ [v])

([] ++ [v] = [v]) = f r ls′ [v] □
Equational Reasoning for Optimization (6)

Proof of Lemma 4.1.6 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:

\[
(f \ r \ ls1 \ ls2) ++ [v] = (f \ r \ [] \ ls2) ++ [v] \\
(ls1=[] = (f \ r \ [] \ ls2) ++ [v] \\
\text{(Unfolding } f \ r) = ls2 ++ [v] \\
(f \ r \ [] \ (ls2 ++ [v])) \\
\text{(Folding } f \ r) = f \ r \ [] \ (ls2 ++ [v]) \\
\text{([]=ls1) = f \ r \ ls1 \ (ls2 ++ [v])}
\]
Equational Reasoning for Optimization (7)

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

\[
\begin{align*}
\text{(fr \ ls1 \ ls2)} +\!+ [v] \\
(\text{ls1} = (v':\text{ls1}')) &= (\text{fr} (v':\text{ls1}') \text{ ls2}) +\!+ [v] \\
(\text{Unfolding fr}) &= (\text{fr} \ \text{ls1}' \ (v':\text{ls2})) +\!+ [v] \\
(\text{ls3} =_{df} (v':\text{ls2})) &= (\text{fr} \ \text{ls1}' \ \text{ls3}) +\!+ [v] \\
(\text{IH}) &= \text{fr} \ \text{ls1}' \ (\text{ls3} +\!+ [v]) \\
((v':\text{ls2}) = \text{ls3}) &= \text{fr} \ \text{ls1}' \ ((v':\text{ls2}) +\!+ [v]) \\
(\text{Def. of} \ (:) \ \text{and} \ (+\!+)) &= \text{fr} \ \text{ls1}' \ (v' :(\text{ls2} +\!+ [v])) \\
(\text{Folding fr}) &= \text{fr} (v' :\text{ls1}') \ (\text{ls2} +\!+ [v]) \\
((v' :\text{ls1}') = \text{ls1}) &= \text{fr} \ \text{ls1} \ (\text{ls2} +\!+ [v]) \quad \square
\end{align*}
\]
Equational Reasoning for Optimization (8)

Equational reasoning together with inductive proof principles, here structural induction, allowed us to prove:

The Haskell expressions `reverse xs` and `fast_reverse xs` are equal for all finite lists `xs` (cf. Theorem 4.1.5):

\[ \forall \, xs \in \text{a-List}. \, \text{reverse } xs = \text{fast_reverse } xs \]

Thus, we have:

**Corollary 4.1.8**

\[ \text{reverse} = \text{fast_reverse} \]

Hence, replacing `reverse` and `fast_reverse` is safe:

**Corollary 4.1.9 (Optimization)**

Programs can safely be optimized by replacing every call of `reverse` by a call of `fast_reverse`.
Conclusion

Functional definitions are genuine mathematical equations. allowing us to prove equality and other relations among functional expressions by means of usual mathematical reasoning.

Proven equality of functions can justify the replacement of a less efficient (called specification) by a more efficient (called implementation) definition of some functionality.

Two examples:

- Specifications: \((x*y)+(x*z)\) // reverse
- Implementations: \(x*(y+z)\) // fastReverse

The development of functional pearls considered next follows this approach in the realm of combinatorial complex problems.
Chapter 4.2

Functional Pearls
Functional Pearls: The Very Idea (1)

The design of functional pearls, i.e., functional programs evolves from calculation!

In more detail:

Starting from a problem with a simple, intuitive but often inefficient specification we shall arrive at an efficient though often more complex and possibly less intuitive implementation by means of mathematical reasoning, i.e., by equational and inductive reasoning, by theorems and laws.

Example: Transforming reverse step by step into fast_reverse.
Functional Pearls: The Very Idea (2)

Note: The functional pearl

- is **not** the finally resulting (efficient) implementation
- but the **calculation and proof process** leading to it!
In the course of founding the
  •  *Journal of Functional Programming*

in 1990, Richard Bird was asked by the designated editors-in-chief Simon Peyton Jones and Philip Wadler to contribute a regular column called
  •  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

in the
  •  *Communications of the ACM*
Functional Pearls: Origin and Background (2)

Since 1990 (till ca. 2011), some

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

and other problems.

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

- *International Conference of Functional Programming*
- *Mathematics of Program Construction*
Roughly a quarter of these pearls have been written by Richard Bird.

In his 2011 monograph


Richard Bird presents a collection of 30 “revised, polished, and re-polished functional pearls” written by him and others.
In this chapter

...we will consider three of these functional pearls focusing especially on the use of equational reasoning for proving the transformation of programs, which are

- obviously correct but (hopelessly) inefficient
- much more efficient (though possibly less intuitive)

correct:

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

Overall, the transformation achieves correctness by construction of the finally resulting program, which is ensured above all by equational reasoning.
Note

...GoFER, acronym and name of a functional programming language stands for

Go F(or) E(quational) R(easoning)!
Chapter 4.3
The Smallest Free Number
The Smallest Free Number (SFN) Problem

The SFN-Problem:

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

Examples:

The smallest free number for

- $\{0, 1, 5, 9, 2\}$ 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!
Analyzing the Problem

Obviously

- The SFN-Problem can easily be solved, if the set $X$ is represented as an increasingly ordered list $xs$ of numbers without duplicates.
- If so, just look for the first gap in $xs$.

Example:

Computing the smallest free number for the set $X$

- $\{8, 23, 9, 12, 11, 1, 10, 0, 13, 7, 41, 4, 21, 5, 17, 3, 19, 2, 6\}$
- After sorting (and removing duplicates):
  $[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 17, 19, 21, 23, 41]$
- Looking for the first gap yields:
  The smallest free number is 14!
SFA: A Straightforward SNFP-Algorithm

The preceding observation suggests the algorithm SFA (reminding to ‘\textit{StraightForward Algorithm}’), which solves the SFN-problem straightforwardly:

The \textit{SFNP-Algorithm} SFA:

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 the previous step.
Implementing the SFNP-Algorithm SFA

...by means of a system of two functions

- \texttt{ssfn} (reminding to ‘simple sfn’) and
- \texttt{sap} (reminding to ‘search and pick’)

Implementation of the SFNP-Algorithm SFA:

\texttt{ssfn :: [Integer] -> Integer}
\texttt{ssfn = (sap 0) . removeDuplicates . quickSort}

\texttt{sap :: Integer -> [Integer] -> Integer}
\texttt{sap n [] = n}
\texttt{sap n (x:xs) |
  | n /= x = n |
  | otherwise = sap (n+1) xs}
The SFNP-Algorithm SFA is sound but inefficient:

- Sorting is not of linear time complexity.

The Functional Pearl View of the SFN-Problem:

Develop an SFNP-Algorithm LinSFNP which is of

- linear time complexity, i.e., which is linear in the number of the elements of the initial set $X$ of natural numbers.
Towards the Linear Time Algorithm

The SFN-Problem can alternatively be solved by the SFNP-Algorithm SFA’ implemented by the function minfree defined by

\[
\text{minfree} :: [\text{Nat}] \rightarrow \text{Nat} \\
\text{minfree} \; \text{xs} = \text{head} \; \left( ([0..]) \setminus \text{xs} \right)
\]

Here

\[
(\setminus) :: \text{Eq} \; a \Rightarrow [a] \rightarrow [a] \rightarrow [a] \\
x \setminus y = \text{filter} \; (\text{\textquoteleft notElem\textquoteright} \; y) \; x
\]

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

\[
\text{type} \; \text{Nat} = \text{Int}
\]

is considered the type of natural numbers starting from 0.
Analysing the SFNP-Algorithm SFA’

...by investigating the function minfree.

Obviously

- \( \text{minfree} \) and hence SFA’ are sound, i.e., SFA’ solves the SFN-Problem.
- But SFA’ is inefficient: Evaluating \( \text{minfree} \) for a list of length \( n \) requires \( O(n^2) \) steps in the worst case.

Note: Evaluating

- \( \text{minfree} \ [n-1,n-2\ldots0] \)

requires evaluating

- \( i \) is not an element in \( [n-1,n-2\ldots0] \) for \( 0 \leq i \leq n \)

and thus \( n(n+1)/2 \) equality tests.
Outline

Starting from SFA' and \texttt{minfree} we will develop

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

linear time algorithms for the SFN-Problem.

Both algorithms rely on the following key fact (KF):

\begin{itemize}
\item In \([0..\text{length }\ xs]\), there is a number which is \textbf{not in} \(xs\)
\end{itemize}

where \(xs\) denotes the argument list of natural numbers.

KF implies: The \textbf{smallest number not in} \(xs\) is given by

\begin{itemize}
\item the \textbf{smallest number not in} \(\text{filter (<=}n\text{) }xs\), where \(n == \text{length }xs\)!
\end{itemize}
The Array-Based SFNP-Algorithm (1)

Based on \texttt{KF}, the array-based \texttt{SFNP-Algorithm LinSFNP} builds a

- checklist of those numbers present in $\text{filter}(\leq n) \ xs$

where checklist is implemented as a

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

Algorithmic idea:

- For each element $x$ in $xs$ with $x \leq n$ the array element at position $x$ is set to \texttt{True}.
- The smallest free number is then found as the position of the first \texttt{False} entry.
The Array-Based SFNP-Algorithm (2)

**Implementation** of the array-based SFNP-Algorithm LinSFNP:

\[
\begin{align*}
\text{minfree} &= \text{search . checklist} \\
\text{search} &:: \text{Array Int Bool} \to \text{Int} \\
\text{search} &= \text{length . takeWhile id . elems} \\
\text{checklist} &:: [\text{Int}] \to \text{Array Int Bool} \\
\text{checklist } xs &= \text{accumArray } (\|\|) \text{ False } (0,n) \\
&\quad \quad \quad \text{(zip (filter } (\leq n) \text{ ) } xs \quad \text{(repeat True))} \\
&\quad \quad \quad \text{where } n = \text{length } xs
\end{align*}
\]

**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.
- If so, `checklist` can 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`)

```haskell
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, Variant B is essentially a procedural program in functional clothing.
The Divide-and-Conquer SFNP-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 4.3.1**

\[
(as ++ bs) \setminus cs = (as \setminus cs) ++ (bs \setminus cs)
\]

\[
as \setminus (bs ++ cs) = (as \setminus bs) \setminus cs
\]

\[
(as \setminus bs) \setminus cs = (as \setminus cs) \setminus bs
\]

**Lemma 4.3.2**

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:

\[
(as ++ bs) \setminus (us ++ vs) = (as \setminus us) ++ (bs \setminus vs)
\]
The Divide-and-Conquer SFNP-Algorithm (2)

Lemma 4.3.3
Let $b$ be a natural number, and let

\begin{itemize}
  \item $\text{as} = [0..b-1]$, $\text{bs} = [b..]$
  \item $\text{us} = \text{filter} (<b) \text{xs}$, $\text{vs} = \text{filter} (\geq b) \text{xs}$
\end{itemize}

Then: $\text{as}$ and $\text{vs}$ are disjoint, and $\text{bs}$ and $\text{us}$ are disjoint.

Lemma 4.3.3 implies:

Corollary 4.3.4

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

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

where $\text{partition}$ is a Haskell library function which partitions a list into those elements satisfying some property and those that do not.
The Divide-and-Conquer SFNP-Algorithm (3)

Together with

\[
\text{head}(\text{xs}++\text{ys}) = \text{if null xs then head ys else head xs}
\]

we get the basic version of the divide-and-conquer SFNP-Algorithm LinSFNP':

\[
\text{minfree } \text{xs} = \begin{cases} 
\text{if } (\text{null } ([0..b-1]) \setminus \text{us}) \\
\text{then } (\text{head } ([b..]) \setminus \text{vs}) \\\n\text{else } (\text{head } ([0..]) \setminus \text{us}) \\\n\text{where } (\text{us},\text{vs}) = \text{partition } (<b) \text{xs}
\end{cases}
\]

...for any natural number \( b \).
Optimizing DaC-Algorithm LinSFNP’ (1)

Note, evaluating the test

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

Note, too, the lists \([0..b-1]\) and \text{us} are lists of

- distinct natural numbers
- every element of \text{us} is less than \text{b}.

Together, this allows us to replace the test by a test on the length of \text{us}:

\[
\text{null } ([0..b-1] \setminus \text{us}) = \text{length } \text{us} == \text{b}
\]

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.
...inspecting \texttt{minfree} in more detail reveals that it can be generalized to a function \texttt{minfrom}:

\[
\text{minfrom} :: \text{Nat} \rightarrow [	ext{Nat}] \rightarrow \text{Nat} \\
\text{minfrom} \ a \ \text{xs} = \text{head} ([\ a\ ..\ ] \ \setminus \ \text{xs})
\]

where every element of \texttt{xs} is assumed to be greater than or equal to \texttt{a}.
Optimizing DaC-Algorithm LinSFNP’ (3)

...provided that $b$ is chosen such that both

\[
\mathbf{length \ us \ and \ length \ vs \ are \ less \ than \ length \ xs}
\]

the following recursive definition of $\text{minfree}$ 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 DaC-Algorithm LinSFNP' (4)

...we are left with picking \( b \) appropriately.

The value of \( b \) must satisfy:

- \( b > a \)
- The maximum of the lengths of \( u_s \) and \( v_s \) is minimum.

This is ensured, if the value of \( b \) is chosen as

\[
b = a + 1 + n \div 2 \quad \text{with} \quad n = \text{length} \; x_s.
\]
Optimizing DaC-Algorithm LinSFNP’ (5)

Note that

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

2. $\text{length } us = b-a$ implies
   $$(\text{length } vs) = (n - (n \div 2) - 1) \leq n \div 2$$

With this choice, the number of steps for evaluating

$$\text{minfrom 0 xs}$$

is linear in the number of elements of $xs$. 
The Optimized DaC-Algorithm LinSFNP”

As a final optimization, we represent \( xs \) by a pair \((\text{length } xs, xs)\) in order to avoid to repeatedly compute \text{length}.

The Optimized Divide&Conquer SFNP-Algorithm LinSFNP”:

\[
\begin{align*}
\text{minfree } & \text{xs } = \text{minfrom } 0 \ (\text{length } \text{xs}, \text{xs}) \\
\text{minfrom } & \ a \ (n,\text{xs}) \\
| \quad n & \ == \ 0 \quad = \ a \\
| \quad m & \ == \ b-a \quad = \ \text{minfrom } \ b \ (n-m,\text{vs}) \\
| \quad \text{otherwise} & \quad = \ \text{minfrom } \ a \ (m,\text{us}) \\
\text{where } & \quad (\text{us},\text{vs}) = \text{partition } (<b) \ \text{xs} \\
& \quad b \quad = \ a + 1 + n \ \text{div } 2 \\
& \quad m \quad = \ \text{length } \text{us}
\end{align*}
\]
Conclusion

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**.
Last but not least

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
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

- $[-4, -3, -7, 2, 1, -2, -1, -4]$.

The maximum segment sum of $L$ is

- $3$, the sum of the elements of the segment $[2, 1]$. 
The MSS-Problem: Background, Motivation

The MSS-Problem

- was considered quite often in the late 1980s mostly as a showcase for 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 functional pearl problem.
The Max. Non-Segment Sum (MNSS) Problem

A non-segment of a list

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

The MNSS-Problem:

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

Example:

Let $L$ be the list

- $[-4, -3, -7, 2, 1, -2, -1, -4]$.

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 that there are

- many more non-segments of a list than segments

which raises the problem:

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

This (pearl) problem will be tackled in this chapter.
Specifying Solution of the MNSS-Problem

The Specifying Solution of the MNSS-Problem:

\[
\text{mnss} :: [\text{Int}] \rightarrow [\text{Int}]
\]

\[
\text{mnss} = \max \ . \ \text{map} \ . \ \text{sum} \ . \ \text{nonsegs}
\]

Intuitively

- First, \text{nonsegs} computes a list of all non-segments of the argument list,
- \text{map sum} then computes the sum of all these non-segments, and
- \text{maximum}, finally, picks those whose sum is maximum.
The implementation of the function \texttt{nonsegs}:

\[
\text{nonsegs} :: [a] \rightarrow [[a]] \\
\text{nonsegs} = \text{extract} \ . \ \text{filter} \ \text{nonseg} \ . \ \text{markings}
\]

relies on the supporting functions

- \texttt{extract}
- \texttt{nonseg}
- \texttt{markings}

which itself relies on the supporting function

- \texttt{booleans}
Implementing the Supporting Functions

...markings, booleans, and extract:

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)
Notes on the Supporting Functions

...markings, booleans, and extract, i.e., 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 the Supporting Function

...nonseg:

▶ 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).

Last but not least:

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.

Note: The regular expression identifies the leftmost gap \( T^+ F^+ T \) that makes the segment a non-segment.
The 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.
The Implementation of Function nonseg

Implementing nonseg:

\[
\text{nonseg} = (== \text{N}) \cdot \text{foldl} \text{ step} \ E \cdot \text{map} \ \text{snd}
\]

where the middle term \text{foldl} \text{ step} \ E \text{ executes the step of}
the finite automaton:

\[
\begin{align*}
\text{step } E \ \text{False} & = E \\
\text{step } E \ \text{True} & = S \\
\text{step } S \ \text{False} & = M \\
\text{step } S \ \text{True} & = S \\
\text{step } M \ \text{False} & = M \\
\text{step } M \ \text{True} & = N \\
\text{step } N \ \text{False} & = N \\
\text{step } N \ \text{True} & = N
\end{align*}
\]

Note:

- Finite automata process their input from left to right. This leads to the use of \text{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.
Work Plan to Derive the Linear Time Alg.

Recall the specifying solution of the MNSS-Problem with nonsegs replaced by its supporting functions:

\[
\text{mnss} = \text{maximum . map sum . extract . filter nonseg . markings}
\]
\[
\text{extract} = \text{map (map fst . filter snd)}
\]
\[
\text{nonseg} = (== \text{N}) \cdot \text{foldl step E . map snd}
\]

Work plan:

- Express \text{extract . filter nonseg . markings} as an instance of \text{foldl}.
- Apply then the fusion law of \text{foldl} to arrive at a better algorithm.
Towards the Linear Time Algorithm (1)

First, we introduce the function \texttt{pick}:

\begin{verbatim}
pick :: State -> [a] -> [[a]]
pick q
  = extract .
    filter (== q) . foldl step E . map snd) .
    markings
\end{verbatim}

\textbf{Note:}

- \texttt{nonsegs = pick N}
Towards the Linear Time Algorithm (2)

...properties of function \texttt{pick}: By (1) calculation from the definition of \texttt{pick q} (which is tedious!) or by (2) referring to the definition of \texttt{step} we can prove \textbf{Lemma 4.4.1}:

\textbf{Lemma 4.4.1}

\begin{align*}
\text{pick } N &= \text{ nonseqs} \\
\text{pick } E \; xs &= \text{ [[]]} \\
\text{pick } S \; [] &= \text{ []} \\
\text{pick } S \; (xs++[x]) &= \text{ map (++)[x]} \\
&\quad (\text{pick } S \; xs) ++ \text{ pick } E \; xs) \\
\text{pick } M \; [] &= \text{ []} \\
\text{pick } M \; (xs++[x]) &= \text{ pick } M \; xs ++ \text{ pick } S \; xs \\
\text{pick } N \; [] &= \text{ []} \\
\text{pick } N \; (xs++ys) &= \text{ pick } N \; xs ++ \\
&\quad \text{map (++)[x]} \\
&\quad (\text{pick } N \; xs) ++ \text{ pick } M \; xs)
\end{align*}
Towards the Linear Time Algorithm (3)

...next, we recast the definition of \texttt{pick} as an instance of \texttt{foldl}.

To this end, let \texttt{pickall} be specified by:

\[
pickall \; xs = (\text{pick E} \; xs, \text{pick S} \; xs, \\
\text{pick M} \; xs, \text{pick N} \; xs)
\]

This allows us to express \texttt{pickall} as an instance of \texttt{foldl}:

\[
pickall = \text{foldl} \; \text{step} \; ([[]], [], [], [], [] ) \\
\text{step} \; (\text{ess}, \text{nss}, \text{mss}, \text{sss}) \; x \\
= (\text{ess}, \\
\text{map} \; (++[x]) \; (\text{sss}++\text{ess}), \\
\text{mss} +\text{ssss}, \\
\text{nss} +\text{ssss} \; \text{map} \; (++[x]) \; (\text{nss}++\text{mss}))
\]
Two new Solutions of the MNSS-Problem

The 1st new Solution of the MNSS-Problem:

\[ \text{mnss} = \text{maximum} \ . \ \text{map sum} \ . \ \text{fourth} \ . \ \text{pickall} \]

where \text{fourth} returns the fourth element of a quadruple.

Using function \text{tuple}

\[ \text{tuple } f (w,x,y,z) = (f \ w, f \ x, f \ y, f \ z) \]

\text{fourth} can be moved to the front of the defining expression of \text{mnss}:

\[ \text{maximum} \ . \ \text{map sum} \ . \ \text{fourth} \]
\[ = \text{fourth} \ . \ \text{tuple} (\text{maximum} \ . \ \text{map sum}) \]

This allows the 2nd new Solution of the MNSS-Problem:

\[ \text{mnss} = \text{fourth} \ . \ \text{tuple} (\text{maximum} \ . \ \text{map sum}) \ . \ \text{pickall} \]
The Fusion Law of \textit{foldl}

\textbf{Lemma 4.4.2 (Fusion Law of \textit{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:

\[ f = \text{tuple} (\text{maximum} \ . \ \text{map} \ \text{sum}) \]
\[ g = \text{step} \]
\[ a = ([[]], [], [], []) \]

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:

\[ \text{tuple} (\text{maximum} \ . \ \text{map} \ \text{sum}) ([[]], [], [], []) \]
\[ = (0, -\infty, -\infty, -\infty) \]

...which gives the definition of \( b \).
Towards Applying the Fusion Law (2)

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

$$\text{tuple (maximum . map sum) (step (ess, sss, mss, nss) x)} = h (\text{tuple (maximum . map sum) (ess, sss, mss, nss)}) x$$

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 $maximum$:

$$
\text{max (map sum (nss ++ map (\(\text{\([x]\) \text{\(+\) nss ++ mss]\}))\) )}
= (\text{definition of map})
\text{max (map sum nss ++ map (\((\text{\((x) \text{. sum}\) nss ++ mss]\}))\)}}
= (\text{since sum . (x)} = (\text{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 sum (nss ++ mss)) + x)}
= (\text{since max . map (x) = (x) . max})
\text{max (map sum nss) max (max (map sum (nss ++ mss)) + x})
= (\text{introducing n = max (map sum nss) and})
\text{m = max (map sum mss))}
\text{n max ((n max m) + x)}
$$
Towards Applying the Fusion Law (4)

Finally, we arrive at the implementation of \( h \):

\[
h(e, s, m, n) \times = (e, (s \text{ max } e)+x, m \text{ max } s, n \text{ max } ((n \text{ max } m) + x))
\]

This allows the 3rd new Solution of the MNSS-Problem:

\[
mnss = \text{fourth . foldl } h (0, -\infty, -\infty, -\infty)
\]
The Linear Time Algorithm

We are left with dealing with the fictitious $\infty$ 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**:

```haskell
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)
```

319/192
Conclusions (1)

The **MSS-Problem** goes back to Jon R. Bentley:


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


Conclusions (2)

Recent results on the MSS-Problem can be found in:

Chapter 4.5
A Simple Sudoku Solver
Sudoku Puzzles

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

The Independent Newspaper
Preliminaries

Preliminary definitions:

- **$m \times n$-matrix**: A list of $m$ rows of the same length $n$.
  
  ```haskell```
  type Matrix a = [Row a]
  type Row a = [a]
  ```

- **Grid**: A $9 \times 9$-matrix of digits.
  
  ```haskell```
  type Grid = Matrix Digit
  type Digit = Char
  ```

- **Valid digits**: ‘1’ to ‘9’; ‘0’ stands for a blank.
  
  ```haskell```
  digits = ['1'..'9']
  blank = (== '0')
  ```
Assumptions on the Setting

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.
Towards the Specifying Solution

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

1st Approach:
  ▶ 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.

2nd Approach:
  ▶ 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 take the 2nd approach to define the specifying initial solution of the Sudoku-problem.
The Specifying Sudoku-Solution

The Specifying Solution of the Sudoku-Problem:

\[
solve = \text{filter valid} \cdot \text{expand} \cdot \text{choices}
\]

\[
\text{choices} :: \text{Grid} \to \text{Matrix Choices}
\]

\[
\text{expand} :: \text{Matrix Choices} \to \text{[Grid]}
\]

\[
\text{valid} :: \text{Grid} \to \text{Bool}
\]

Intuitively:

- **choices** constructs all choices for the blank entries of the input grid,
- **expand** then computes all grids that arise from making every possible choice,
- **filter valid** finally selects all the valid grids.
Completing the Specifying Sudoku-Solution (1)

First, we introduce the data type

\[
\text{type Choices} = [\text{Digit}]
\]

for representing the set of choices.

Based on this, we define next the subsidiary functions of \text{solve}, i.e., the functions

- \text{choices}
- \text{expand}
- \text{valid}
Completing the Specifying Sudoku-Solution (2)

Implementing `choices`:

```haskell
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 Specifying Sudoku-Solution (3)

Implementing \texttt{expand}:

\begin{verbatim}
expand :: Matrix Choices \to [Grid]
expand :: cp . map cp

cp :: [[a]] \to [[a]] \quad (cp \equiv \text{cartesian product})
cp [] = [[]]
cp (xs:xss) = [x:ys \mid x \gets xs, ys \gets cp xss]
\end{verbatim}

\textbf{Intuitively}

\begin{itemize}
  \item Expansion is a Cartesian product, i.e., a list of lists yielded by the function \texttt{cp}, e.g., \texttt{cp [ [1,2], [3], [4,5] ] \to [ [1,3,4], [1,3,5], [2,3,4], [2,3,5] ]}
  \item \texttt{map cp} returns a list of all possible choices for each row.
  \item \texttt{cp . map cp}, finally, installs each choice for the rows in all possible ways.
\end{itemize}
Completing the Specifying Sudoku-Solution (4)

Implementing `valid`:

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

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

Intuitively

- A grid is valid, if no row, column or box contains duplicates.
Completing the Specifying Sudoku-Solution (5)

Implementing `rows` and `columns`:

```haskell
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 Specifying Sudoku-Solution (6)

Implementing \texttt{boxs}:

\begin{verbatim}
boxs :: Matrix a -> Matrix a
boxs = map ungroup . ungroup . map cols .
       group . map group

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

ungroup :: [[a]] -> [a]
ungroup = concat
\end{verbatim}

Intuitively

- \texttt{group} splits a list into groups of three.
- \texttt{ungroup} takes a grouped list and ungroups it.
- \texttt{group \ . \ map \ group} produces a list of matrices; transposing each matrix and ungrouping them yields the boxes.
Completing the Specifying Sudoku-Solution (7)

...illustrating the effect of boxes for the \((4 \times 4)\)-case, when group splits a list into groups of two:

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

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

Instead of

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

the preceding approach has gone for functions which

- treat a matrix as a complete entity in itself.

Geraint Jones coined the notion

- wholemeal programming

for this style of programming.

Wholemeal programming

- helps avoiding indexitis and
- encourages lawful program construction.
Lawful Programming

Lemma 4.5.1

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

\[
\begin{align*}
\text{rows} \cdot \text{rows} &= \text{id} & (A) \\
\text{cols} \cdot \text{cols} &= \text{id} & (B) \\
\text{boxs} \cdot \text{boxs} &= \text{id} & (C)
\end{align*}
\]

This means, all 3 functions are involutions.

Lemma 4.5.2

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

\[
\begin{align*}
\text{map rows} \cdot \text{expand} &= \text{expand} \cdot \text{rows} & (D) \\
\text{map cols} \cdot \text{expand} &= \text{expand} \cdot \text{cols} & (E) \\
\text{map boxs} \cdot \text{expand} &= \text{expand} \cdot \text{boxs} & (F)
\end{align*}
\]
A Quick Analysis of the Specifying Solution

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

Then there are about $9^{40}$, or


grids to be constructed and checked for validity!

This is hopeless!
Optimizing the Specifying 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

\[
\text{prune} :: \text{Matrix Choices} \rightarrow \text{Matrix Choices}
\]

which satisfies

\[
\text{filter valid} \ . \ \text{expand} \ = \ \text{filter valid} \ . \ \text{expand} \ . \ \text{prune}
\]

and realizes the above idea.
Pruning a Row

Pruning a row

\[
\text{pruneRow} :: \text{Row Choices} \rightarrow \text{Row Choices} \\
\text{pruneRow} \ \text{row} = \text{map} (\text{remove} \ \text{fixed}) \ \text{row} \\
\text{where} \ \text{fixed} = [d \mid [d] \leftarrow \text{row}]
\]

\[
\text{remove} \ \text{xs} \ \text{ds} \\
= \text{if singleton} \ \text{ds} \ \text{then} \ \text{ds} \ \text{else} \ \text{ds} \ \setminus \ \text{xs}
\]

Intuitively

\[\text{remove}\] removes choices from any choice that is not fixed.
Laws for pruneRow, nodeups, and cp

- The function `pruneRow` satisfies law (G):

  \[
  \text{filter nodups . cp} = \text{filter nodups . cp . pruneRow} \quad \text{(G)}
  \]

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

  If \( f \) is an involution, i.e., \( f \cdot f = \text{id} \), then

  \[
  \text{filter (p.f) = map f . filter p . map f} \quad \text{(H)}
  \]

  \[
  \text{filter (all p) . cp = cp . map (filter p)} \quad \text{(I)}
  \]
Rewriting filter valid . expand

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

We can prove:

**Lemma 4.5.3**

\[
\text{filter valid . expand} = \text{filter (all nodups . boxs)}. \\
\text{filter (all nodups . cols)}. \\
\text{filter (all nodups . rows). expand}
\]

(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 4.5.3: boxes Case (1)

\[ \text{filter (all nodups . boxes) . expand} \]
\[ = \{(H), \text{since boxes . boxes = id}\} \]
\[ \text{map boxes . filter (all nodups) . map boxes . expand} \]
\[ = \{(F)\} \]
\[ \text{map boxes . filter (all nodups) . expand boxes} \]
\[ = \{\text{definition of expand}\} \]
\[ \text{map boxes . filter (all nodups) . cp . map cp . boxes} \]
\[ = \{(I), \text{and map f . map g = map (f . g)}\} \]
\[ \text{map boxes . cp . map (filter nodups . cp) . boxes} \]
\[ = \{(G)\} \]
\[ \text{map boxes . cp . map (filter nodups . cp . pruneRow) . boxes} \]
Proof Sketch of Lemma 4.5.3: boxes Case (2)

\[
= \{ (I) \}
\]

\[
\text{map } \text{boxes} \ . \ \text{filter} \ (\text{all nodups}) \ . \ \text{cp} \ .
\]

\[
\text{map } \text{cp} \ . \ \text{map } \text{pruneRow} \ . \ \text{boxes}
\]

\[
= \{ \text{definition of expand} \}
\]

\[
\text{map } \text{boxes} \ . \ \text{filter} \ (\text{all nodups}) \ . \ \text{expand} \ .
\]

\[
\text{map } \text{pruneRow} \ . \ \text{boxes}
\]

\[
= \{ (H) \text{ in the form } \text{map } f \ . \ \text{filter } p =
\]

\[
\text{filter } (p \ . f) \ . \ \text{map } f \}
\]

\[
\text{filter} \ (\text{all nodups} \ . \ \text{boxes}) \ . \ \text{map } \text{boxes} \ . \ \text{expand} \ .
\]

\[
\text{map } \text{pruneRow} \ . \ \text{boxes}
\]

\[
= \{ (F) \}
\]

\[
\text{filter} \ (\text{all nodups} \ . \ \text{boxes}) \ . \ \text{expand} \ . \ \text{boxes} \ .
\]

\[
\text{map } \text{pruneRow} \ . \ \text{boxes}
\]
Summing up

Overall, we have shown:

**Lemma 4.5.4**

\[
\text{filter (all nodups . boxs) . expand} \\
= \text{filter (all nodups . boxs)} . \text{expand . pruneBy boxs}, \text{where}
\]

\[
\text{pruneBy f} = f \cdot \text{map pruneRow} \cdot f
\]

Repeating the same calculation for rows and cols we get:

**Lemma 4.5.5**

\[
\text{filter valid . expand} \\
= \text{filter valid . expand . prune}, \text{where}
\]

\[
\text{prune} \\
= \text{pruneBy boxs . pruneBy cols . pruneBy rows}
\]
Implementation of `solve` after the 1st Opt.

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

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

Note: Pruning can be done more than once.

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

...based on the following idea:

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

Which cell to expand?

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

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

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

\[
\text{expand} = \text{concat} \ . \ \text{map} \ \text{expand} \ . \ \text{expand1} \quad (J)
\]

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

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

\[
\text{expand1} :: \text{Matrix Choices} \rightarrow [\text{Matrix Choices}]
\]
\[
\text{expand1 rows} = [\text{rows1} ++ [\text{row1} ++ [c] : \text{row2}] ++ \text{rows2} | c<-cs]
\]

where
\[
(\text{rows1},\text{row}::\text{rows2}) = \text{break (any smallest) rows}
\]
\[
(\text{row1}, \text{cs}::\text{row2}) = \text{break smallest row}
\]
\[
\text{smallest cs} = \text{length \ cs} == n
\]
\[
n = \text{minimum (counts rows)}
\]
\[
\text{counts} = \text{filter (/=}1) \cdot \text{map length} \cdot \text{concat}
\]

\[
\text{break p xs} = (\text{takeWhile (not \ . \ p) xs}, \text{dropWhile (not \ . \ p) xs})
\]
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:**
  \[
  \text{complete} = \text{all} \ (\text{all} \ \text{single})
  \]
  where \text{single} is the test for a singleton list.

- **Safety Test:**
  \[
  \text{safe} \ m = \text{all ok} \ (\text{rows} \ m) \ &&
  \text{all ok} \ (\text{cols} \ m) \ &&
  \text{all ok} \ (\text{boxs} \ m)
  \]
  \[
  \text{ok row} = \text{nodups} \ [d \mid [d] \leftarrow \text{row}]
  \]
Equational Reasoning

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

\[
\text{filter valid \ . \ expand} \\
= \{\text{since expand} = \text{concat \ . \ map expand \ . \ expand1} \\
\text{on incomplete matrices}\} \\
\text{filter valid \ . \ concat \ . \ map expand \ . \ expand1} \\
= \{\text{since filter p \ . \ concat} = \text{concat \ . \ map (filter p)}\} \\
\text{concat \ . \ map (filter valid \ . \ expand)} \ . \ \text{expand1} \\
= \{\text{since filter valid \ . \ expand} = \text{filter valid \ . \ expand \ . \ prune}\} \\
\text{concat \ . \ map (filter valid \ . \ expand \ . \ prune)} \ . \ \text{expand1}
\]
Implementation of \texttt{solve} after the 2nd Opt.

Defining \texttt{search} by

\[
\text{search} = \text{filter valid . expand . prune}
\]

we have for \texttt{safe} but \texttt{incomplete} matrices the equality

\[
\text{search . prune} = \text{concat . map search . expand1}
\]

This leads us to the final

\textbf{Implementation of \texttt{solve}, after the 2nd Optimization (single cell-improved)}:

\[
\text{solve} = \text{search . choices}
\]

\[
\text{search m}
\]

\[
\mid \not \text{(safe m)} = []
\]

\[
\mid \text{complete } m' = [\text{map (map head) } m']
\]

\[
\mid \text{otherwise} = \text{concat (map search (expand1 } m'))
\]

where \[m' = \text{prune m}\]
Quality and Performance Assessment

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

▶ haskell.org/haskellwiki/Sudoku

It is reported that the solver

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

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

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

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

▶ equational reasoning!
Chapter 4.6
References, Further Reading
Chapter 4: Further Reading (1)


Chapter 4: Further Reading (2)


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, Philip Wadler. *An Introduction to Functional Programming*. Prentice Hall, 1988. (Chapter 4.3.1, Texts as lines)
Chapter 4: Further Reading (3)


Chapter 4: Further Reading (4)

Jeremy Gibbons. *Functional Pearls – An Editor’s Perspective*. www.cs.ox.ac.uk/people/jeremy.gibbons/pearls/


Chapter 4: Further Reading (5)


Part III

Quality Assurance
Chapter 5

Testing
Chapter 5.1
Motivation
Confidence in Correctness

...how can we gain (sufficiently much) confidence that

▶ ours and
▶ other people’s programs

are correct?
Means for Gaining Confidence in Correctness

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

**Correctness by Construction** (*a priori*, Chapter 4)
- Exemplified by the development of *functional pearls*.

**Verification** (*a posteriori*, Chapter 6)
- Rigoros, formal correctness proofs (soundness of the specification, soundness of the implementation).
- High confidence, high effort (typically).

**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.
Minimum Requirements for Systematic Testing

Systematic testing should be

- Specification-based
- Tool-supported
- Automatic

Additional ‘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/testing after program modifications
Specifications

...describing and fixing the meaning of programs can be done:

▶ Informally, e.g., as commentary in the program, in a separate documentation
  ⇝ 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
In this chapter, we consider systematic testing...

...using **QuickCheck**, a combinator library, which supports specification-based, tool-supported testing 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.
Note

...QuickCheck and its property specification and test data generator languages are

▶ examples of so-called 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
Defining Simple Properties w/ QuickCheck (1)

...simple properties can be defined in terms of predicates, i.e., as Boolean valued functions.

Example:

Define inside of a Haskell program the property:

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

Double-checking the property with Hugs yields:

```haskell
Main> quickCheck prop_PlusAssociative
OK, passed 100 tests
```
Defining Simple Properties w/ QuickCheck (2)

...varying the introductory example slightly.

Replace property definition prop_PlusAssociative by:

\[
\text{prop\\_PlusAssociative}' :: \text{Float} \to \text{Float} \to \text{Float} \to \text{Bool} \\
\text{prop\\_PlusAssociative}' \ x \ y \ z = (x+y)+z == x+(y+z)
\]

Double-checking the property with Hugs might yield:

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

- The type specifications for `prop_PlusAssociative` and `prop_PlusAssociative'` are required because of the overloading of `(+)`.
- If the type specifications were missing, error messages on ambiguous overloading would be issued; intuitively, QuickCheck needs to know which test data to generate.
- Type specifications in predicates allow the type-specific generation of test data.
- The associativity property for 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 simple property definitions.

Given:

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

To be tested:

- Correctness of the insertion operation: A list after inserting an element shall be sorted.

Property definition:

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

Actually, this property is falsifiable. It is naive, since the argument list `xs` is not supposed to be sorted itself, and hence 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)

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 the specification of more sophisticated properties like e.g.

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

as well as the testing of

▶ more than one property at the same time.

The latter can be achieved by running a (small) program (also called quickCheck) from the command line

▶ Main>quickCheck Module.hs

which checks all properties defined in Module.hs at the same time.
Chapter 5.3

Testing against Abstract Models
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 the reference implementation of an abstract model of queues.
The Abstract Model of Queues

...defined in terms of an:

(Executable) Specification:

```haskell
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 reference implementation for queues; an implementation, which is simple but inefficient.
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: $[7,2,9,4,1,6,8,3] \equiv ([7,2,9,4],[3,8,6,1]), ([7,2],[3,8,6,1,4,9]), ([7,2,9,4,1],[3,8,6]),...$
- Abstract model enqueuing, $(++)$: $[7,2,9,4,1,6,8,3]++[5]$
- Implementation enqueuing, $(:)$: $( [7,2,9,4], 5:[3,8,6,1] ), ( [7,2], 5:[3,8,6,1,4,9] ), ( [7,2,9,4,1], 5:[3,8,6] ),...$
Implementing the Abstract Model of Queues

Implementation:

```haskell
type QueueI a = ([a],[a])
emptyQI = ([],[])  -- (:) instead of (++)!
   -- Therefore, more
   -- efficient!
enQI x (f,b) = (f,x:b)
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-
  -- comes ‘front’
  flipQI q = q  -- when ‘front’
  -- gets empty.
```

384/1927
Relating Implementation and Abstract Model

...by means of the function \texttt{retrieve}:

\[
\text{retrieve} :: \text{QueueI} \ a \rightarrow \text{Queue} \ a
\]
\[
\text{retrieve} \ (f, b) = f ++ \text{reverse} \ b
\]

Note, \texttt{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 \((\text{QueueI} \ a)\) into their unique matching value of \((\text{Queue} \ a)\).

Example:

\[
\text{retrieve} \ ([7,2,9,4],[5,3,8,6,1]) \rightarrow [7,2,9,4,1,6,8,3,5]
\]
\[
\text{retrieve} \ ([7,2],[5,3,8,6,1,4,9]) \rightarrow [7,2,9,4,1,6,8,3,5]
\]
\[
\text{retrieve} \ ([7,2,9,4,1],[5,3,8,6]) \rightarrow [7,2,9,4,1,6,8,3,5]
\]

...
In the following

...we want to test whether operations defined on \((\text{QueueI } a)\) behave in the same way as their specifying counterparts defined on \((\text{Queue } a)\).

For convenience, we will focus on queues of integer values (i.e., \((\text{QueueI } \text{Int})\) and \((\text{Queue } \text{Int})\)) allowing us to omit

- type specifications in property definitions.

Using \texttt{retrieve :: QueueI Int -> Queue Int} we can check, whether the results of applying

- the efficient operations on \((\text{QueueI } \text{Int})\) match the ones of their abstract counterparts on \((\text{Queue } \text{Int})\).
Soundness Properties: Initial Definitions

Defining five soundness properties:

\[
\begin{align*}
\text{prop\_emptyQ} & = \text{retrieve emptyQI} == \text{emptyQ} \\
\text{prop\_enQ} x q & = \text{retrieve (enQI x q)} \\
& = \text{enQ x (retrieve q)} \\
\text{prop\_isemptyQ} q & = \text{is\_isemptyQI q} \\
& = \text{is\_isemptyQ (retrieve q)} \\
\text{prop\_frontQ} q & = \text{frontQI q} == \text{frontQ (retrieve q)} \\
\text{prop\_deQ} q & = \text{retrieve (deQI q)} \\
& = \text{deQ (retrieve q)}
\end{align*}
\]

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

Actually, 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 empty only, if its back is empty, too:
  
  \[ \text{is\_emptyQI} (f,b) \Rightarrow \text{null } b \]

  since \( \text{is\_emptyQI} (f,b) = \text{null } f, \text{emptyQI} = ([],[]) \).

This invariant, however, is not enforced by the implementation!
Falsifiability of frontQI and deQI

...the definitions of is_emptyQI, frontQI, and deQI all rely on the very same assumption that the front of a queue will only be empty if the back also is.

Therefore, in addition to prop_isemptyQ the properties

- prop_frontQ
- prop_deQ

are falsifiable, too!

...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:

\[
\text{invariant} :: \text{QueueI Int} \rightarrow \text{Bool} \\
i\text{invariant}\ (f,b) = (\text{not (null } f)) \text{ || null } b
\]

...and adjust the property definitions accordingly:

\[
\begin{align*}
\text{prop_emptyQ} & = \text{retrieve emptyQI == emptyQ} \\
\text{prop_enQ } x \ q & = \text{invariant } q => \text{retrieve (enQI } x \ q) == \text{enQ } x (\text{retrieve } q) \\
\text{prop_isemptyQ } q & = \text{invariant } q => \text{is_emptyQI } q == \text{is_emptyQ (retrieve } q) \\
\text{prop_frontQ } q & = \text{invariant } q => \text{frontQI } q == \text{frontQ (retrieve } q) \\
\text{prop_deQ } q & = \text{invariant } q => \text{retrieve (deQI } q) == \text{deQ (retrieve } q)
\end{align*}
\]
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 \texttt{not (is\_emptyQI \ q)} to the precondition of the challenged properties.

This leads to:

\[
\begin{align*}
\text{prop\_emptyQ} & \quad = \text{retrieve emptyQI} == \text{emptyQ} \\
\text{prop\_enQ x q} & \quad = \text{invariant q} \implies \\
& \quad \quad \quad \text{retrieve (enQI x q)} == \text{enQ x (retrieve q)} \\
\text{prop\_isemptyQ q} & \quad = \text{invariant q} \implies \\
& \quad \quad \quad \text{is\_isemptyQI q} == \text{is\_isemptyQ (retrieve q)}
\end{align*}
\]

\[
\begin{align*}
\text{prop\_frontQ q} & \quad = \text{invariant q} \&\& \text{not (is\_isemptyQI q)} \implies \\
& \quad \quad \quad \text{frontQI q} == \text{frontQ (retrieve q)} \\
\text{prop\_deQ q} & \quad = \text{invariant q} \&\& \text{not (is\_isemptyQI q)} \implies \\
& \quad \quad \quad \text{retrieve (deQI q)} == \text{deQ (retrieve q)}
\end{align*}
\]
Soundness Issues Reconsidered

Now, all five properties (2nd refinement!) pass 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

\[
\text{invariant} :: \text{QueueI Int} \rightarrow \text{Bool} \\
\text{invariant} (f,b) = (\not\text{null } f) \text{ || null } b
\]

Additionally, we need to check that

- operations producing a queue do only produce queues which satisfy the invariant.
Additional Soundness Properties

...defining soundness properties for operations producing queues:

\[
\begin{align*}
\text{prop_inv_emptyQ} &= \text{invariant } \text{emptyQI} \\
\text{prop_inv_enQ} x q &= \text{invariant } q \implies \text{invariant } (\text{enQI} x q) \\
\text{prop_inv_deQ} q &= \text{invariant } q \land \neg (\text{is_emptyQI } q) \implies \text{invariant } (\text{deQI } q)
\end{align*}
\]
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 \texttt{enQI}

...by replacing the faulty implementation of \texttt{enQI}

\[
\text{enQI } x (f,b) = (f,x:b)
\]

by the sound one:

\[
\text{enQI } x (f,b) = \text{flipQ} (f,x:b)
\]

where

\[
\text{flipQI} ([],b) = (\text{reverse } b,[]) \]
\[
\text{flipQI} q \quad = q
\]

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 maintenance!).
Chapter 5.4

Testing against Algebraic Specifications
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:

\[
\text{prop\_isemptyQ } q = \ \\
\quad \text{invariant } q \implies \text{isEmptyQI } q = (q = \text{emptyQI})
\]

\[
\text{prop\_front\_emptyQ } x = \text{frontQI (enQI } x \text{ emptyQI)} = x
\]

\[
\text{prop\_front\_enQ } x \ q = \ \\
\quad \text{invariant } q \land \neg (\text{is\_emptyQI } q) \implies \ \\
\quad \text{frontQI (enQI } x \ q) = \text{frontQI } q
\]

\[
\text{prop\_deQ\_emptyQ } x = \text{deQI (enQI } x \text{ emptyQI)} = \text{emptyQI}
\]

\[
\text{prop\_deQ\_enQ } x \ q = \ \\
\quad \text{invariant } q \land \neg (\text{is\_emptyQI } q) \implies \ \\
\quad \text{deQI (enQI } x \ q) = \text{enQI } x \ (\text{deQI } q)
\]

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 \texttt{prop\_deQ\_enQ} using \texttt{QuickCheck} yields:

\begin{verbatim}
Main>quickCheck prop\_deQ\_enQ
Falsifiable, after 1 tests:
0
([1],[0])
\end{verbatim}

\textbf{Cause of failure:} Evaluating

\begin{itemize}
\item the left hand side expression yields:
  \texttt{deQI (enQI 0 ([1],[0]))} \rightarrow \texttt{deQI ([1],[0,0])} \\
  \rightarrow \texttt{flipQI ([],[0,0])} \rightarrow \texttt{([0,0],[0])}
\item the right hand side expression yields:
  \texttt{enQI 0 (deQI ([1],[0]))} \rightarrow \texttt{enQI 0 (flipQI ([],[0]))} \\
  \rightarrow \texttt{enQI 0 ([0],[])} \rightarrow \texttt{([0],[0])}
\end{itemize}

\begin{itemize}
\item ([0,0],[0]) and ([0],[0]) are equivalent (they represent the abstract queue \([0,0]\)) but are \textbf{not exactly equal!}
\end{itemize}
Refining the Algebraic Specification

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

\[ q \, \text{'equiv'} \, q' = \text{invariant } q \land \text{invariant } q' \land \text{retrieve } q = \text{retrieve } q' \]

Replacing the initial formulation of

\[
\text{prop\_deQ\_enQ } x \ q = \\
\text{invariant } q \land \text{not (is\_emptyQI } q) \implies \\
\text{deQI (enQI } x \ q) = \text{enQI } x \ (\text{deQI } q)
\]

by the new one

\[
\text{prop\_deQ\_enQ } x \ q = \\
\text{invariant } q \land \text{not (is\_emptyQI } q) \implies \\
\text{deQI (enQI } x \ q) \,'\text{equiv'} \text{enQI } x \ (\text{deQI } q)
\]

the QuickCheck test of \text{prop\_deQ\_enQ} passes successfully!
Testing further Equational Constraints

Analogously to the testing approach in 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 need to introduce additional soundness properties for the operations `enQI` and `deQI`:

\[
\text{prop\_enQ\_equivQ} \quad q \quad q' \quad x = \\
q '\text{equiv'} q' \implies \text{enQI} x q '\text{equiv'} \text{enQI} x q' \\
\]

\[
\text{prop\_deQ\_equivQ} \quad q \quad q' = \\
q '\text{equiv'} q' \land \text{not (null } q) \land \text{not (null } q') \implies \text{deQI } q '\text{equiv'} \text{deQI } q' \\
\]
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 und 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
Motivation

...by default, the parameters of QuickCheck properties are quantified over all values of the underlying data type (e.g., over all integers, over all lists of integers).

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

QuickCheck offers several means for controlling quantification over subsets of values.
Dealing with Subsets of Value Domains (1)

Discussed so far: How to deal with subsets of values using

1. Boolean functions: Used as preconditions in property definitions acting 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: How to deal with subsets of values using

2. Generators: Used for targeted generation of test data of the subset of interest:
   - Generators of the monadic type (Gen a) can generate random values of type a; conceptually, they 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 elements of set set whose elements are randomly generated.
Dealing with Subsets of Value Domains (2)

...both means differ in their strengths and limitations for particular tasks when chosen for representing relations of values such as being equivalent. Representing equivalence by a

- **Boolean function**, it is easy to check whether two values are equivalent, but difficult to generate values which are equivalent.

- **Generator**, i.e., a function from a value to a set of related (here, equivalent) values, it is easy to generate equivalent values, but difficult to check if two given values are equivalent.

While the usage of Boolean functions for representing subsets of values has been illustrated in Chapter 5.3 and Chapter 5.4, the usage of generators will be discussed next.
Type Constructor Gen

...defining generators is eased because Gen is a monadic type constructor (cf. Chapter 11); for Gen as an instance of type constructor class Monad holds:

The generator expression

- return a generates value a, and represents the singleton set \{a\}.
- do \{x <- s; e\} can be thought of to represent the set \{e | x \in s\}.
Random Element Generation

...by means of function `choose`, the most basic function of `QuickCheck`, which makes 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.
- **choose (1,n)**, e.g., represents the set `{1,...,n}`.
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`:

```haskell
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 a random queue `q'`, which contains 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 the generator `equivQ`, we define soundness property:

\[
\text{prop\_equivQ} \ q = \text{invariant} \ q \implies \\
\text{forAll} (\text{equivQ} \ q) \ \lambda q' \to q \ '\text{equiv}' \ q'
\]

...allowing us 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 \(\lambda\)-expression in the definition of `prop\_equivQ`).

- The dual property 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 the generator \texttt{equivQ}, we can define counterparts of the properties \texttt{prop\_enQ\_equivQ} and \texttt{prop\_deQ\_equivQ} allowing us to test, whether \texttt{enQ} and \texttt{deQ} map equivalent queues to equivalent queues:

\begin{align*}
\text{prop\_enQ\_equivQ} \ q \ x &= \text{invariant} \ q \implies \\
&\forall q' \ (\text{equivQ} \ q) \Rightarrow \text{enQI} \ x \ q \ '\text{equiv'} \ \text{enQI} \ x \ q'
\end{align*}

\begin{align*}
\text{prop\_deQ\_equivQ} \ q &= \text{invariant} \ q \ \&\ \& \ \text{not (null q)} \implies \\
&\forall q' \ (\text{equivQ} \ q) \Rightarrow \text{deQI} \ q \ '\text{equiv'} \ \text{deQI} \ q'
\end{align*}

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

\begin{align*}
\text{prop\_enQ\_equivQ} \ q \ q' \ x &= \\
q \ '\text{equiv'} \ q' \implies \text{enQI} \ x \ q \ '\text{equiv'} \ \text{enQI} \ x \ q'
\end{align*}

\begin{align*}
\text{prop\_deQ\_equivQ} \ q \ q' &= \\
q \ '\text{equiv'} \ q' \ \&\ \& \ \text{not (null q)} \ \&\ \& \ \text{not (null q')} \implies \\
&\text{deQI} \ q \ '\text{equiv'} \ \text{deQI} \ q'
\end{align*}
Type-based Generation of Value Sets

...are possible using the overloaded generator arbitrary, e.g., for generating the argument values of properties:

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

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

which is equivalent to the short-hand form:

```haskell
prop_max_le x y = x <= x 'max' y
```
Type-based Generation of Subsets of Values

...by means of arbitrary and a subsequent value modification as required:

Example: The set of numerical values \( \{ y \mid y \geq x \} \) is generated by the generator \( \text{atLeast} \) defined by:

\[
\text{atLeast } x = \begin{cases} 
\text{do } \text{diff} <- \text{arbitrary} \\
\text{return } (x + \text{abs} \text{ diff})
\end{cases}
\]

whose definition is based on the set equality:

\[
\{ y \mid y \geq x \} = \{ x + \text{abs} \ d \mid d \in \mathbb{Z} \}
\]

which holds for numerical values,

Note: The idea underlying the definition of \( \text{atLeast} \) can be adapted to other types than numerical ones.
Selection of Generators

...is enabled by the generator `oneof` which can be thought of as `set union` operator.

**Example:** The generator `orderedLists` (cf. Chapter 5.2) generating sorted lists based on the idea that a sorted list is either empty or the result of attaching a new head element to a sorted list of larger elements:

```haskell
orderedLists = do x <- arbitrary
                 listsFrom x

where
  listsFrom x = oneof [return [],
                       do y <- atLeast x
                          liftM (x:) (listsFrom y)]
```

```haskell
-- either: empty
-- or: a list of elems > x
-- extended
-- by x as new head element
```
Note

...the `oneof` generator picks alternatives with

- equal probability.

This impacts the generation of test data often unduly. E.g.,
the generator `orderedLists` will produce

- the empty list far too often

questioning its adequacy and hence usability as a test data
generator for ordered lists.

`QuickCheck` offers thus means supporting a `weighted selection`
of generators.
Weighted Selection of Generators

...using the generator frequency weights can be assigned to alternatives impacting the relative likelihood they are selected:

```haskell
frequency :: [(Int,Gen a)] -> Gen a
```

Example:

```haskell
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.
Making a Generator Default-Gen. for a Type

...if a non-default generator such as orderedLists is used frequently, it is advisable to define a new type for the value type it generates 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.
Controlling the Size of Generated 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 value 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 interpretes size as an upper bound on the length).

Generators depending on size can be defined using function

sized :: (Int -> Gen a) -> Gen a
Example

...the default generator \texttt{vector} for list values:

\begin{verbatim}
vector n = sequence [arbitrary | i <- [1..n]]
\end{verbatim}

...calling \texttt{vector} with argument \texttt{length} generates lists of random values of length \texttt{length}.

\texttt{vector} in concert with function \texttt{sized}:

\begin{verbatim}
sized $ \n -> do length <- choose (0,n) 
  vector length
\end{verbatim}
The Function resize

...allows to supply an explicit size parameter to a generator:

\[
\text{resize} :: \text{Int} \to \text{Gen}\ a \to \text{Gen}\ a
\]

**Example:** Generating a list of lists while bounding the total number of elements by the size parameter:

\[
\text{sized} \; \lambda n \to \text{resize} (\text{round} \ (\text{sqrt} \ (\text{fromInt}\ n))) \text{arbitrary}
\]

**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 \text{arbitrary} but with a smaller size parameter.
A Note on Lift Functions

...used throughout Chapter 5.5, which are provided by the library module Monad (cf. Chapter 11):

\[
\begin{align*}
\text{liftM} & \quad : \quad \text{Monad } m \Rightarrow (a \rightarrow b) \rightarrow (m a \rightarrow m b) \\
\text{liftM2} & \quad : \quad \text{Monad } m \Rightarrow (a \rightarrow b \rightarrow c) \rightarrow (m a \rightarrow m b \rightarrow m c) \\
\text{liftM3} & \quad : \quad \text{Monad } m \Rightarrow (a \rightarrow b \rightarrow c \rightarrow d) \rightarrow (m a \rightarrow m b \rightarrow m c \rightarrow m d) \\
\text{liftM4} & \quad : \quad \text{Monad } m \Rightarrow (a \rightarrow b \rightarrow c \rightarrow d \rightarrow e) \rightarrow (m a \rightarrow m b \rightarrow m c \rightarrow m d \rightarrow m e) \\
\text{liftM5} & \quad : \quad \text{Monad } m \Rightarrow (a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow f) \rightarrow (m a \rightarrow m b \rightarrow m c \rightarrow m d \rightarrow m e \rightarrow m f)
\end{align*}
\]
Chapter 5.6

Test Data Generators at Work: An Example
Generators for Built-in and User-defined Types

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 inductively defined types.

...this is illustrated next considering a binary tree type.
A User-defined Generator for Binary Trees (1)

...we consider the following binary tree type:

```haskell
data Tree a = Leaf | Branch (Tree a) a (Tree a)
```

(Tree a) can straightforwardly be made an instance of type class Arbitrary:

```haskell
instance Arbitrary a => Arbitrary (Tree a) where
    arbitrary =
        frequency [(1,return Leaf),
                  (3,liftM3 Branch arbitrary arbitrary arbitrary arbitrary)]
```
A User-defined Generator for Binary Trees (2)

Note:

- The assignment of weights (1 vs. 3) shall ensure that not too many trivial trees of size 1 are generated.
- **Problem:** 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. With increasing breadth of the generated trees, the requirement of always selecting the ‘terminating’ branch must be satisfied at ever more places simultaneously...

**Remedy:** Using the **size** parameter in order to ensure

- termination and
- generation of trees of ‘reasonable’ size.
A User-defined Generator for Binary Trees (3)

...replace the initial instance-declaration for (Tree a) by:

```haskell
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, the total size is bounded by the parameter size.
- Generators for recursive types must usually be handled like in this example.
Chapter 5.7
Monitoring, Reporting, and Coverage
Test-Data Monitoring

In practice, it is useful to monitor the generated test cases in order to obtain a hint on the quality and the coverage of test cases of a **QuickCheck** run.

For this purpose **QuickCheck** provides a bunch of monitoring and reporting possibilities.
Why is Test-Data Monitoring Required?

...reconsider the example of inserting into a sorted list:

```haskell
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 every one, whether it is sorted (used as test case) or not (discarded).

Obviously, the likelihood that a randomly generated list

- is sorted is the higher the shorter the list is.

This introduces the danger that

- property `prop_InsertOrdered` is mostly tested with lists of length one or two.
- even a successful test is not meaningful.
QuickCheck Combinators

...allowing to control test-data monitoring:

- trivial
- classify
- collect
Test-Data Monitoring using trivial (1)

The combinator trivial is useful for monitoring purposes, where

▶ the meaning of ‘trivial’ is user-definable, e.g., that lists up to a length of 2 are considered trivial.

Example:

```haskell
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 yields:

```
Main>quickCheck prop_InsertOrdered
OK, passed 100 tests (91% trivial).
```
Test-Data Monitoring using trivial (2)

Observation regarding the example:

- 91% are too many trivial test cases in order to ensure that the total test is meaningful.
- The operator $\Rightarrow$ should thus be used with care in test-case generators.

Remedy:

- User-defined generators, e.g., by using quantification as sketched in Chapter 5.2.

Note:

- The combinator trivial is defined in terms of the more general combinator classify:

  $$\text{trivial } p = \text{classify } p \ "\text{trivial}"$$
Test-Data Monitoring using classify

The combinator classify allows a more refined test-case monitoring than the combinator trivial.

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 yields:

```
Main>quickCheck prop_InsertOrdered
OK, passed 100 tests.
42% unit lists.
40% empty lists.
```
Test-Data Monitoring using collect

Going beyond, the combinator \texttt{collect} allows to keep track on all test cases.

Example:

\[
\text{prop\_InsertOrdered } x \ x s = \text{is\_ordered } x s \Rightarrow
\text{collect (length } x s)} $ \text{is\_ordered (insert } x \ x s)
\]

Double-checking this property yields a histogram of values:

Main\textgreater \texttt{quickCheck prop\_InsertOrdered}
OK, passed 100 tests.
46\% 0.
34\% 1.
15\% 2.
5\% 3.
Chapter 5.8
Implementation of QuickCheck
QuickCheck: Facts and Figures

QuickCheck

- consists in total of about 300 lines of code.

- has been developed by Koen Claessen and John Hughes and initially presented in:

QuickCheck: A Glimpse of the Code

```haskell
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 ()
```
Background Material

For further details, also on applications, refer e.g., to:


as well as to:


...on which the presentation of this chapter is closely based on.
Chapter 5.9

Summary
Relevance and Value of Specifications

Experience shows:

- Formalizing specifications is meaningful (even without a subsequent formal proof of soundness).
- Specifications provided are (initially) often faulty themselves.
Relevance and Value of Testing

Investigations of Richard Hamlet indicate that

▶ a high number of test cases yields meaningful results even in the case of random testing.
▶ the generation of random test cases is often ‘cheap.’

Hence, there are many good reasons advising

▶ the routine usage of tools like QuickCheck!

For further details, refer to:

Relevance and Value of QuickCheck

Experience shows that QuickCheck is an effective tool for

- disclosing bugs in programs and specifications with little effort.
- reducing test costs while at the same time testing more thoroughly.
Related Approaches

...besides QuickCheck there are various other combinator libraries supporting the lightweight testing of Haskell programs, e.g.:

- EasyCheck
- SmallCheck
- Lazy SmallCheck
- Hat (for tracing Haskell programs)
A Confirmation of the Relevance of Testing

...also by:

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.

An influential work, advocating rigor and correctness:


40 years later, a retrospective:

A 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.”
Chapter 5.10

References, Further Reading
Chapter 5: Further Reading (1)

- Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 18.2, QuickCheck)


Chapter 5: Further Reading (2)


Chapter 5: Further Reading (3)


Chapter 5: Further Reading (4)


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
Motivation

...testing and verification aim both at

▶ ensuring the correctness of a program or system

but are of different rigor.

Though testing can be amazingly effective, it is limited to

▶ showing the presence of errors. It can not show their absence (except of the most simple scenarios).

By contrast, verification can

▶ prove the absence of errors!
In this chapter

...we will consider important inductive proof principles for proving properties of functional programs (though 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)
  - ...
Inductive Proof Principles: Outline

...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
Chapter 6.1.1

Natural Induction
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

**Inductive Case**

\[
P(1) \land \left[ \forall n \in \mathbb{IN}. \ P(n) \Rightarrow P(n + 1) \right] \Rightarrow \forall n \in \mathbb{IN}. \ P(n)
\]

Base Case  \quad \text{Induction Hypothesis}  \quad \text{Induction Step}  \quad \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:

$$
\sum_{k=1}^{n} (2k - 1) = \sum_{k=1}^{1} (2k - 1) \\
= 2 \cdot 1 - 1 \\
= 2 - 1 \\
= 1 \\
= 1^2 \\
= n^2
$$
Proof of Lemma 6.1.1.1 (2)

Inductive case: Let $n \in \mathbb{IN}$. 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 as follows:

$$\sum_{k=1}^{n+1} (2k - 1) = 2(n + 1) - 1 + \sum_{k=1}^{n} (2k - 1)$$

(IH)

$$= 2(n + 1) - 1 + n^2$$

$$= 2n + 2 - 1 + n^2$$

$$= 2n + 1 + n^2$$

$$= n^2 + 2n + 1$$

$$= n^2 + n + n + 1$$

$$= (n + 1)(n + 1)$$

$$= (n + 1)^2$$

\[\square\]
Homework

Prove by means of natural (mathematical) induction:

Lemma 6.1.1.2

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
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

\[
\forall n \in \mathbb{IN}. \left[ \left( \forall m < n. P(m) \right) \Rightarrow P(n) \right] \Rightarrow \forall n \in \mathbb{IN}. P(n)
\]

Induction Hypothesis Induction Step 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 is defined by:

\[ \text{fib} : \mathbb{N}_0 \rightarrow \mathbb{N}_0 \]

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

Lemma 6.1.2.1

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

Proof by means of strong induction.
The Key for Proving Lemma 6.1.2.1

...for the case $n \in \mathbb{IN}_0$, $n \geq 2$, is to use the equality

$$fib(m) = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^m - \left(\frac{1-\sqrt{5}}{2}\right)^m}{\sqrt{5}}$$

given by the induction hypothesis (IH) for $m = n - 1$ and $m = n - 2$.

(Note: In the case of $n \geq 2$, we could use this equality even for all $m < n$ by means of the induction hypothesis (instead of only for $m = n - 1$ and $m = n - 2$). This, however, is not required to complete the proof.)
Proof of Lemma 6.1.2.1 (1)

**Case 1:** Let \( n = 0 \). Equational reasoning yields straightforwardly the desired equality:

\[
\text{fib}(0) = 0 = \frac{0}{\sqrt{5}} = \frac{1 - 1}{\sqrt{5}} = \frac{(\frac{1 + \sqrt{5}}{2})^0 - (\frac{1 - \sqrt{5}}{2})^0}{\sqrt{5}}
\]

(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:

\[
\text{fib}(1) = 1 = \frac{\sqrt{5}}{\sqrt{5}} = \frac{\frac{1}{2} + \frac{\sqrt{5}}{2} - (\frac{1}{2} - \frac{\sqrt{5}}{2})}{\sqrt{5}} = \frac{(\frac{1 + \sqrt{5}}{2})^1 - (\frac{1 - \sqrt{5}}{2})^1}{\sqrt{5}}
\]

(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:

$$\begin{align*}
\text{fib}(n) &= \text{fib}(n-2) + \text{fib}(n-1) \\
(\text{2x IH}) &= \frac{(1+\sqrt{5})^{n-2} - (1-\sqrt{5})^{n-2}}{\sqrt{5}} + \frac{(1+\sqrt{5})^{n-1} - (1-\sqrt{5})^{n-1}}{\sqrt{5}} \\
&= \left[ \frac{(1+\sqrt{5})^{n-2} + (1+\sqrt{5})^{n-1}}{\sqrt{5}} \right] - \left[ \frac{(1-\sqrt{5})^{n-2} + (1-\sqrt{5})^{n-1}}{\sqrt{5}} \right] \\
&= \frac{(1+\sqrt{5})^{n-2} \left[ 1 + \frac{1+\sqrt{5}}{2} \right] - (1-\sqrt{5})^{n-2} \left[ 1 + \frac{1-\sqrt{5}}{2} \right]}{\sqrt{5}} \\
&= \frac{(1+\sqrt{5})^{n-2} \left( 1 + \frac{1+\sqrt{5}}{2} \right)^2 - (1-\sqrt{5})^{n-2} \left( 1 - \frac{1-\sqrt{5}}{2} \right)^2}{\sqrt{5}} \\
&= \frac{(1+\sqrt{5})^n - (1-\sqrt{5})^n}{\sqrt{5}}.
\end{align*}$$

\[\square\]
Proof of Equality (*)

The equality marked by (*) holds because of the two equalities shown below which are proved by equational reasoning using the binomial formulae:

\[
\left(\frac{1 + \sqrt{5}}{2}\right)^2 = \frac{1 + 2\sqrt{5} + 5}{4} = \frac{6 + 2\sqrt{5}}{4} = \frac{3 + \sqrt{5}}{2} = 1 + \frac{1 + \sqrt{5}}{2}
\]

\[
\left(\frac{1 - \sqrt{5}}{2}\right)^2 = \frac{1 - 2\sqrt{5} + 5}{4} = \frac{6 - 2\sqrt{5}}{4} = \frac{3 - \sqrt{5}}{2} = 1 + \frac{1 - \sqrt{5}}{2}
\]
Homework

Let function $f$ be defined by:

$$f : \mathbb{IN}_0 \to \mathbb{IN}_0$$

$$f(n) =
\begin{cases}
0 & \text{if } n = 0 \\
1 & \text{if } n = 1 \\
\sum_{k=0}^{n-1} f(k) & \text{if } n \geq 2
\end{cases}$$

Prove by means of (Lemma 6.1.2.3 and) strong induction:

**Lemma 6.1.2.2**

$$(\forall n \in \mathbb{IN}. \ n \geq 2). \ f(n) = 2^{n-2}$$

Prove by natural (mathematical) induction:

**Lemma 6.1.2.3**

$$(\forall n \in \mathbb{IN}. \ n \geq 3). \ \sum_{k=0}^{n-3} 2^k = 2^{n-2} - 1$$
Excursus: Which Rectangle

...is the ‘most’ typical, the ‘nicest’ rectangle?

Rectangle 1

Rectangle 2

Rectangle 3

....most people say ‘Rectangle 3’!
Why?

\[ \frac{8 \text{ UoL}}{5 \text{ UoL}} = 1.6 \quad (\text{UoL} \equiv \text{Unit of Length}) \]
The value 1.6 comes close to...

...the Golden Ratio:

\[ \phi = \frac{1 + \sqrt{5}}{2} = 1.61803398874989... \]

Note: The ratio of section A and section B, denoted by \( \phi \) and called the Golden Ratio, is the same as the ratio of section B and section C:

\[ \phi = \text{df } A/B = B/C \]
The Golden Ratio

...is perceived by most people as very harmonious:

\[
\frac{8 \text{ UoL}}{5 \text{ UoL}} = 1.6
\]
Computing $\phi$

\[
\begin{array}{c|c|c}
\hline
x & \frac{x+1}{x} & \phi \\
\hline
1 & df & \frac{x}{1} \\
\hline
\end{array}
\]

$1 + \frac{1}{x} \iff \phi \iff \phi = x$

Using $\phi$ for $x$, we get:

\[
1 + \frac{1}{\phi} = \phi \iff \phi + 1 = \phi^2 \iff 0 = \phi^2 - \phi - 1
\]

\[
\phi = \frac{1 + \sqrt{5}}{2} = 1.618... \\
\phi' = \frac{1 - \sqrt{5}}{2} = -0.618...
\]

Note: $\phi'$ lacks a geometrical interpretation.
The Golden Ratio

...shows up not only as the ratio of sections but also as the ratio of areas, e.g., rectangles:

<table>
<thead>
<tr>
<th>1 UoL</th>
<th>( (\phi - 1) \text{ UoL} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1^{2} \text{ UoL}^{2} = 1 \text{ UoL}^{2}</td>
<td>1*(\phi - 1) \text{ UoL}^{2} = (\phi - 1) \text{ UoL}^{2}</td>
</tr>
</tbody>
</table>
The Golden Ratio

...related also to (the ratio of subsequent) Fibonacci numbers:

\[
\begin{array}{c|c|c}
\hline
21 & 13 & \\
\hline
8 & 3 & \\
\hline
\end{array}
\]
Illustration

The sequence of Fibonacci numbers:

\[ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, \ldots \]

The sequence of the ratios of the Fibonacci numbers:

\[
\begin{align*}
1/1 & = 1 \\
2/1 & = 2 \\
3/2 & = 1.5 \\
5/3 & = 1.6 \\
8/5 & = 1.6 \\
13/8 & = 1.625 \\
21/13 & = 1.615384615384615 \\
34/21 & = 1.619047619047619 \\
& \ldots \\
1,346,269/832,040 & = 1.618033988750541 \approx \phi
\end{align*}
\]
The Golden Ratio

...as the limit of the ratios of the Fibonacci numbers:

\[ \lim_{n \to \infty} \frac{\text{fib}(n+1)}{\text{fib}(n)} = \frac{1 + \sqrt{5}}{2} = \phi \]

...letting Lemma 6.1.2.1 perhaps less arbitrarily looking than it might do at first sight.
Chapter 6.2

Inductive Proof Principles on Structured Data
Chapter 6.2.1
Induction and Recursion
Induction and Recursion

...are closely related.

**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.

Hence:

- **Induction** (bottom-up) and **recursion** (top-down) can be considered the two sides of the same coin.
In fact

...the preferred usage of induction over recursion in some contexts resp. vice versa

▶ e.g., defining data structures (induction)
▶ e.g., defining algorithms (recursion)

is often mostly due to historical reasons.

Data type (inductively defined)

data Tree = Leaf Int | Node Tree Int Tree

Algorithm (recursively defined)

fac :: Int -> Int
fac n = if n == 0 then 1 else n * fac (n-1)
Illustration

▶ Inductive definition of (simple) arithmetic expressions:

(r1) Each numeral $n$ and variable $v$ is an (elementary) arithmetic expression.

(r2) If $e_1$ and $e_2$ are arithmetic expressions, then also $(e_1 + e_2)$, $(e_1 - e_2)$, $(e_1 \cdot e_2)$, 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 the sorted sublists $sl_1$ and $sl_2$.

(ms3) Merge the sorted sublists $sl_1$ and $sl_2$ into the sorted list $sl$ of $l$. 
Summary

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 consists of a node and a left and a right subtree.
  ▶ An **arithmetic expression** is either a numeral or a variable, or 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
The Principle of Structural Induction

Let $S$ be a set of elements inductively constructed from finitely many simpler/simplest elements of $S$, let $\text{sub}(s) \subseteq S$, $s \in S$, denote the finite set of elements $s$ is constructed from, and let $P$ be a property of the elements of $S$.

The Principle of Structural Induction

\begin{align*}
\forall s \in S. \left[ (\forall s' \in \text{sub}(s). \ P(s')) \Rightarrow P(s) \right] & \Rightarrow \forall s \in S. \ P(s) \\
\end{align*}

(Inductive) Case

Induction Hypothesis

Induction Step

Conclusion

(dtsch. Prinzip der strukturellen Induktion)

Note: For the ‘simplest’ elements (or atoms or building blocks) $\hat{s}$ of $S$ we have $\text{sub}(\hat{s}) = \emptyset$. 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

Let the set of (simple) arithmetic expressions $\mathcal{AE}$ be defined by the BNF rule:

$$e ::= n \mid v \mid (e_1 + e_2) \mid (e_1 - e_2) \mid (e_1 \ast e_2) \mid (e_1/e_2)$$

where $n$ and $v$ stand for (integer) numerals and variables, respectively.

Lemma 6.2.2.1

Let $p_e$ and $op_e$ denote the number of parentheses and operators of any expression $e$, $e \in \mathcal{AE}$, respectively. Then:

$$\forall e \in \mathcal{AE}. \quad p_e = 2 \ast op_e$$

Proof by means of structural induction.
(Base) case: Let $e \equiv n$, $n$ a numeral, or $e \equiv v$, $v$ a variable. In both cases $e$ does not contain any parentheses or operators. This means $p_e = 0 = op_e$. This yields directly the desired equality:

\[
\begin{align*}
    p_e & \\
    = 0 & \\
    = 2 \times 0 & \\
    = 2 \times op_e
\end{align*}
\]
Proof of Lemma 6.2.2.1 (2)

(Inductive) case: Let $e \equiv (e_1 \circ e_2)$, $\circ \in \{+, -, \times, /\}$, and $e_1, e_2 \in \mathcal{AE}$. By means of the induction hypothesis (IH), we can assume $p_{e_1} = 2 \times op_{e_1}$ and $p_{e_2} = 2 \times op_{e_2}$. The equality of $p_e$ and $2 \times op_e$ follows then by equational reasoning:

$$
\begin{align*}
(p_e & \quad \equiv \quad (e_1 \circ e_2))
\quad = \quad p_{(e_1 \circ e_2)} \\
\quad = \quad 1 + p_{e_1} + p_{e_2} + 1 \\
\text{(2x IH)} & \quad = \quad 2 \times op_{e_1} + 2 + 2 \times op_{e_2} \\
\quad = \quad 2 \times op_{e_1} + 2 \times 1 + 2 \times op_{e_2} \\
\quad = \quad 2 \times (op_{e_1} + 1 + op_{e_2}) \\
\quad = \quad 2 \times op_{(e_1 \circ e_2)} \\
((e_1 \circ e_2) & \equiv e) \quad = \quad 2 \times op_e
\end{align*}
$$
Homework (1)

Prove by means of structural induction:

**Lemma 6.2.2.2**
Let $lp_e$ and $rp_e$ denote the number of left and right parentheses of any expression $e \in \mathcal{AE}$, respectively. Then:

$$\forall e \in \mathcal{AE}. \; lp_e = rp_e$$

**Lemma 6.2.2.3**
Let $d_e$ and $opd_e$ denote the depth and the number of operands of any expression $e \in \mathcal{AE}$, respectively. Then:

$$\forall e \in \mathcal{AE}. \; opd_e \leq 2^{d_e}$$
Homework (2)

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.

**Lemma 6.2.2.4**

Let \( d_e \) and \( opd_e \) denote the depth and the number of operands of any expression \( e \in \mathcal{AE} \), respectively. Then:

\[
(\forall e \in \mathcal{AE}. \ e \text{ complete}). \ opd_e = 2^{d_e}
\]

Proof by means of structural induction.
Note

...the principles of

- natural (math.) induction (dtsch. vollständige Induktion)
  \[ P(1) \land [\forall n \in \mathbb{N}. P(n) \Rightarrow P(n+1)] \Rightarrow \forall n \in \mathbb{N}. P(n) \]

- strong induction (dtsch. verallgemeinerte Induktion)
  \[ \forall n \in \mathbb{N}. [(\forall m < n. P(m)) \Rightarrow P(n)] \Rightarrow \forall n \in \mathbb{N}. P(n) \]

- 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
Chapter 6.3.1
Inductive Proofs on Haskell Trees
Inductive Proofs on Finite Trees

A tree is called

▶ **finite**, if the length of all paths originating at its root is finite.

▶ **maximum**, if it is finite and all paths from a leaf to its root are of the same length.

Let

\[
\text{data Tree} = \text{Leaf Int} \mid \text{Node Tree Tree}
\]

**Lemma 6.3.1.1**

Let \( \text{depth}(t) \) and \( \text{leaves}(t) \) denote the depth and the number of leaves of any finite tree value \( t :: \text{Tree} \), respectively. Then:

\[
(\forall t :: \text{Tree}. \; t \text{ maximum}). \; \text{leaves}(t) = 2^{\text{depth}(t)}
\]

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 \( 2^{\text{depths}(t)} \):

\[
\begin{align*}
\text{leaves}(t) \\
(t \equiv (\text{Leaf } k)) &= \text{leaves}((\text{Leaf } k)) \\
&= 1 \\
&= 2^0 \\
&= 2^{\text{depths}(t)}
\end{align*}
\]
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:

\[
\begin{align*}
(t \equiv (\text{Node } t_1 \ t_2)) &= \text{leaves}(\text{Node } t_1 \ t_2) \\
(2\times \text{IH}) &= 2^{\text{depth}(t_1)} + 2^{\text{depth}(t_2)} \\
(\text{depth}(t_1) = \text{depth}(t_2)) &= 2^{\text{depth}(t_1)} + 2^{\text{depth}(t_1)} \\
&= 2 \times 2^{\text{depth}(t_1)} \\
&= 2^{\text{depth}(t_1+1)} \\
&= 2^{\text{depth}(t)}
\end{align*}
\]

\( \square \)
Homework

Prove by means of structural induction:

Lemma 6.3.1.2

Let $\text{depth}(t)$ and $\text{leaves}(t)$ denote the depth and the number of leaves of any finite tree value $t :: \text{Tree}$, respectively. Then:

$$(\forall t :: \text{Tree}. \ t \text{ finite}). \ \text{leaves}(t) \leq 2^{\text{depth}(t)}$$
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 :: 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 6.3.1.3
(∀ x :: Maybe Int. x defined). maybe 2 abs x ≥ 0
Proof of Lemma 6.3.1.3

Case 1: Let $x \equiv \text{Nothing}$. We obtain:

\[
\text{maybe 2 abs } x \\
= \text{maybe 2 abs Nothing} \\
= 2 \\
\geq 0
\]

Case 2: Let $x \equiv \text{Just } m$, $m$ defined. We obtain:

\[
\text{maybe 2 abs } x \\
= \text{maybe 2 abs (Just } m) \\
= \text{abs } m \\
\geq 0
\]

$\square$
Chapter 6.3.2

Inductive Proofs on Haskell Lists
Defined and Undefined Values

A computation which
  ▶ is faulty, i.e., produces an error or
  ▶ fails to (regularly) terminate
does not yield a proper value.

The value of such a computation is called
  ▶ undefined, or the undefined value
which is usually denoted by the symbol
  ▶ \( \perp \) (read ‘bottom’).

Conversely, a properly terminating computation yields a value
different from \( \perp \), which is called
  ▶ defined or a defined value.
Example

The function

```haskell
buggy_div :: Int -> Int
buggy_div n = div n 0
```

...produces an error for every argument called with.

The function

```haskell
buggy_fac :: Int -> Int
buggy_fac n = (n-1) * buggy_fac n
buggy_fac 0 = 1
```

...fails to (regularly) terminate for every argument called with.
Very Simple Haskell Terms

...with a value equal to \( \perp \):

- **Error:** The Prelude definition
  
  \[
  \text{undefined} :: \text{a} \quad \text{-- polymorphic} \\
  \text{undefined} \mid \text{False} = \text{undefined} \\
  \text{undefined} \rightarrow> \text{‘error’} \equiv \perp
  \]

  is a very simple expression (of arbitrary type) whose evaluation always leads to an error due to case exhaustion.

- **Non-termination:** The co-recursive definition
  
  \[
  \text{loop} :: \text{a} \quad \text{-- polymorphic} \\
  \text{loop} = \text{loop} \\
  \text{loop} \rightarrow> \text{loop} \rightarrow> \text{loop} \rightarrow> \ldots \equiv \perp
  \]

  is a very simple expression (of arbitrary type) whose evaluation leads to a non-terminating computation.
The Undefined Value ⊥

...is a value of every Haskell data type representing the value of faulty or non-terminating computations.

Intuitively, ⊥ can be considered the ‘least accurate’ approximation of (ordinary) values of the corresponding data type.
Informally

Lists are

- possibly empty finite sequences of values of the same type.

Haskell lists are

- 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 called

- defined, if all values are defined, i.e., different from ⊥.
  
  Examples: [], (1:[]), (1:2:3:[]),...

  Counter-examples: (⊥:[]), (1:⊥:[]), (⊥:2:⊥:[]),...

- lists with undefined values, if at least one value is equal to the undefined value.
  
  Examples: (⊥:[]), (1:⊥:[]), (⊥:2:⊥:[]),...
Defined Lists, Lists with Undefined Values

Definition 6.3.2.1 (Defined, Undefined Values)
A value of a data type is called defined, if it is not equal to \( \bot \); it is called undefined otherwise.

Definition 6.3.2.2 (List)
A list is a possibly empty finite sequence of (defined or undefined) values of the same type built from the empty list \([\,]\).

Definition 6.3.2.3 (Def. List, List w/ Undef. Values)
A list is called

- defined, if all its values are defined.
- a list with possibly undefined values, if some of its values can be equal to \( \bot \).
Structural Induction for Defined Lists

Let $P$ be a property on defined lists.

Proof pattern of structural induction for 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: The above pattern is an instance of the more general pattern of structural induction, specialized for defined lists.
Example 1: Induction over Defined Lists

Let

\[
\text{length} :: [a] \to \text{Int} \\
\text{length} [] = 0 \\
\text{length} (_:\!xs) = 1 + \text{length} \; xs
\]

\textbf{Lemma 6.3.2.4}

We have:

\[(\forall xs, ys :: [a]. \; xs, ys \; \text{defined}). \; \text{length} \; (xs ++ ys) = \text{length} \; xs + \text{length} \; ys\]

Proof by induction on the structure of \(xs\).
Proof of Lemma 6.3.2.4 (1)

Let \( ys :: [a] \) be a defined list.

**Base case:** Let \( xs \equiv [] \). As desired, we obtain by means of equational reasoning:

\[
\begin{align*}
\text{length} (xs ++ ys) & = \text{length} ([] ++ ys) \\
& = \text{length} ys \\
& = 0 + \text{length} ys \\
& = \text{length} [] + \text{length} ys \\
& = \text{length} xs + \text{length} ys
\end{align*}
\]
Proof of Lemma 6.3.2.4 (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:

\[
\begin{align*}
\text{length} (xs ++ ys) &= \text{length} ((x:xs') ++ ys) \\
&= \text{length} (x:(xs' ++ ys)) \\
&= 1 + \text{length} (xs' ++ ys) \\
(\text{IH}) &= 1 + (\text{length} xs' + \text{length} ys) \\
&= (1 + \text{length} xs') + \text{length} ys \\
&= \text{length} (x:xs') + \text{length} ys \\
&= \text{length} xs + \text{length} ys
\end{align*}
\]

\( \square \)
Example 2: Induction over Defined Lists

Let

\[
\text{listSum} :: \text{Num } a \Rightarrow [a] \rightarrow a \\
\text{listSum} [] = 0 \\
\text{listSum} (x:xs) = x + \text{listSum} \hspace{0.5pt} xs
\]

Lemma 6.3.2.5
We have:

\[
(\forall \hspace{0.5pt} xs :: [a]. \hspace{0.5pt} xs \hspace{0.5pt} \text{defined}). \hspace{0.5pt} \text{listSum} \hspace{0.5pt} xs = \text{foldr} (+) 0 \hspace{0.5pt} xs
\]

Proof by induction on the structure of \hspace{0.5pt} xs.
Proof of Lemma 6.3.2.5 (1)

Base case: Let \( \text{xs} \equiv [] \). Equational reasoning yields the desired equality:

\[
\begin{align*}
\text{listSum} \ \text{xs} \\
= \ & \text{listSum} \ [] \\
= \ & 0 \\
= \ & \text{foldr} (+) \ 0 \ [] \\
= \ & \text{foldr} (+) \ 0 \ \text{xs}
\end{align*}
\]
Proof of Lemma 6.3.2.5 (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 \( \text{listSum} \; xs' = \text{foldr} \; (+) \; 0 \; xs' \). This allows us to complete the proof as follows:

\[
\begin{align*}
\text{listSum} \; xs &= \text{listSum} \; (x:xs') \\
&= x + \text{listSum} \; xs' \\
(IH) &= x + \text{foldr} \; (+) \; 0 \; xs' \\
&= \text{foldr} \; (+) \; 0 \; (x:xs') \\
&= \text{foldr} \; (+) \; 0 \; xs
\end{align*}
\] □
Example 3

Lemma 6.3.2.6
For all defined lists \( \textit{xs :: [a]} \), we have:

\[
\text{reverse (reverse \, \textit{xs})} = \textit{xs}
\]

Proof by induction on the structure of \( \textit{xs} \).
Proof of Lemma 6.3.2.6 (1)

Base case: Let $xs \equiv \[]$. Equational reasoning yields the desired equality:

$$\begin{align*}
\text{reverse (reverse } xs) &= \text{reverse (reverse } \[]) \\
(\text{Def. reverse}) &= \text{reverse } \[] \\
(\text{Def. reverse}) &= \[] \\
&= xs
\end{align*}$$
Proof of Lemma 6.3.2.6 (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 $\text{reverse (reverse } xs') = xs'$. This allows us to complete the proof as follows:

\[
\begin{align*}
\text{reverse (reverse } xs) & = \text{ reverse (reverse } (x:xs')) \\
& = \text{ reverse ((reverse } xs') ++ [x]) \\
& = \text{ reverse } [x] ++ \text{ reverse (reverse } xs') \\
([x] = x:[] , \text{ IH}) & = \text{ reverse } (x:[]) ++ xs' \\
& = \text{ (reverse } [] ++ [x]) ++ xs' \\
& = ([] ++ [x]) ++ xs' \\
& = [x] ++ xs' \\
([x] = x:[]) & = (x : []) ++ xs' \\
& = x : ([] ++ xs') \\
& = x:xs' \\
& = xs
\end{align*}
\]

$\square$
Example 4

...sometimes, a truly inductive argument is not required.

Lemma 6.3.2.7
Let $f$ be a strict map. Then:

$$(\forall \mathbf{x} :: [\mathbf{a}]. \mathbf{x} \text{ defined}). (f \cdot \text{head}) \mathbf{x} = \text{head} \cdot (\text{map} \ f \ \mathbf{x})$$

Proof by cases.
Proof of Lemma 6.3.2.7 (1)

Case 1: Let \( \mathbf{x} \equiv [] \). We get:

\[
(f \ . \ \text{head}) \ \mathbf{x}
\]

\[
= (f \ . \ \text{head}) \ [\]
\]

(Def. of \( . \))

\[
= f (\text{head} \ [\])
\]

(Def. of head)

\[
= f \ \bot
\]

(f strict)

\[
= \bot
\]

(Def. of head)

\[
= \text{head} \ [\]
\]

(Def. of map)

\[
= \text{head} \ (\text{map} \ f \ [\])
\]

(Def. of \( . \))

\[
= (\text{head} \ . \ \text{map} \ f) \ [\]
\]

\[
= (\text{head} \ . \ \text{map} \ f) \ \mathbf{x}
\]
Proof of Lemma 6.3.2.7 (2)

Case 2: Let \( xs \equiv (x:xs') \), \( xs \) defined. This implies \( xs' \) and \( x \) are defined, too. We get:

\[
(f \cdot \text{head}) \ xs = (f \cdot \text{head}) (x:xs')
\]

(Def. of \( . \)) \[= f (\text{head} (x:xs'))
\]

(Def. of head) \[= f \ x
\]

(Def. of head, lazy eval.) \[= \text{head} (f \ x : \text{map} \ f \ xs')
\]

(Def. of map) \[= \text{head} (\text{map} \ f (x:xs'))
\]

(Def. of \( . \)) \[= (\text{head} \cdot \text{map} \ f) (x:xs')
\]

\[= (\text{head} \cdot \text{map} \ f) \ xs
\]

Note: The induction hypothesis \((f \cdot \text{head}) \ xs' = (\text{head} \cdot \text{map} \ f) \ xs'\) is not required to complete the proof of case 2; the inductive proof boils down to a proof by cases.
Homework (1)

...examples involving list reversions and concatenations.

Prove by means of structural induction on defined lists:

**Lemma 6.3.2.8**

For all defined lists \(xs, ys, zs :: [a]\), 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 6.3.2.9**

For all defined lists \(xs :: [a]\), we have:

\(xs ++ [] = xs = [] ++ xs\)
Homework (2)

...examples involving list `take` and `drop` operations.

Prove by means of structural induction on defined lists:

**Lemma 6.3.2.10**

For all defined lists `xs :: [a]`, for all `m, n ∈ \mathbb{N}`, `m, n ≥ 0`, we have:

1. $\text{take } n \, \text{drop } n \, \text{xs} = \text{xs}$
   
   $\text{take } m \cdot \text{take } n = \text{take } (\min m \, n)$
   
   $\text{drop } m \cdot \text{drop } n = \text{drop } (m+n)$
   
   $\text{take } m \cdot \text{drop } n = \text{drop } n \cdot \text{take } (m+n)$

2. If (additionally) $n ≥ m$, we have:
   
   $\text{drop } m \cdot \text{take } n = \text{take } (n-m) \cdot \text{drop } m$
Homework (3)

...examples involving list foldings.

Prove by means of structural induction over defined lists:

**Lemma 6.3.2.11**

Let \( \text{op} :: (a \rightarrow a \rightarrow a) \) be associative with unit \( e :: a \), i.e.,
\[
\forall x :: a. \text{e} \; \text{op} \; x = x \land x \; \text{op} \; \text{e} = x.
\]
Then:
\[
(\forall xs :: [a]. \text{xs defined}). \text{foldr} \; \text{op} \; \text{e} \; \text{xs} = \text{foldl} \; \text{op} \; \text{e} \; \text{xs}
\]

**Lemma 6.3.2.12**

Let \( \text{op} :: (a \rightarrow b \rightarrow b) \) be an operator, \( e :: b \) a value. Then:
\[
(\forall xs :: [a]. \text{xs defined}). \text{foldr} \; \text{op} \; \text{e} \; \text{xs} = \text{foldl} \; (\text{flip} \; \text{op}) \; \text{e} \; (\text{reverse} \; \text{xs})
\]
Homework (4)

...examples involving list foldings.

Prove by means of structural induction on defined lists:

Lemma 6.3.2.13
Let \( \text{op1}, \text{op2} :: (a \to a \to a) \) be two operators, \( e :: b \) a value such that

\[
\forall x, y, z :: a. \ x \ op1 \ (y \ op2 \ z) = (x \ op1 \ y) \ op2 \ z \land x \ op1 \ e = e \ op2 \ x
\]

Then:

\[
(\forall xs :: [a]. \ xs \ defined). \ \text{foldr} \ \text{op1} \ e \ xs = \ \text{foldl} \ \text{op2} \ e \ xs
\]
Homework (5)

...examples involving sequential composition and mappings.

Prove by means of structural induction on defined lists:

Lemma 6.3.2.14

1. \[ \text{map} (f \cdot g) = \text{map} f \cdot \text{map} g \]
2. \[ (\text{map} f) \cdot \text{tail} = \text{tail} \cdot \text{map} f \]
3. \[ (\text{map} f) \cdot \text{reverse} = \text{reverse} \cdot \text{map} f \]
4. \[ (\text{map} f) \cdot \text{concat} = \text{concat} \cdot \text{map} (\text{map} f) \]
5. \[ \text{map} f (xs ++ ys) = \text{map} f xs ++ \text{map} f ys \]
6. \[ \text{map} (\lambda x \rightarrow x) = \lambda y \rightarrow y \]

What are the types of the two anonymous \( \lambda \)-abstractions in Lemma 6.3.2.14(6)? Do they have the same type or different ones?
Structural Induction for Lists w/ Undefined Values

Let $P$ be a property on lists with possibly undefined values.

Proof pattern of structural induction for 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 $P(\bot:xs)$ and $P(x:xs)$, $x$ a defined value, are true (induction step).

Note: The above pattern is an instance of the more general pattern of structural induction, specialized for lists with possibly undefined values.
Example: Induct. over Lists w/ Undef. Values

Let

\[
\begin{align*}
\text{length} & : \ [a] \rightarrow \text{Int} \\
\text{length} \ [\] & = 0 \\
\text{length} \ (_{::}\ x) & = 1 + \text{length} \ x
\end{align*}
\]

Lemma 6.3.2.15

We have:

\[
(\forall \ x, y :: [a]. \ x, y \ \text{lists w/ possibly undefined values}). \\
\text{length} \ (x \ + \ y) = \text{length} \ x \ + \ \text{length} \ y
\]

Proof by induction on the structure of \( x \).
Proof of Lemma 6.3.2.15 (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:

\[
\begin{align*}
\text{length} (xs ++ ys) &= \text{length} ([] ++ ys) \\
&= \text{length} ys \\
&= 0 + \text{length} ys \\
&= \text{length} [] + \text{length} ys \\
&= \text{length} xs + \text{length} ys
\end{align*}
\]
Proof of Lemma 6.3.2.15 (2)

Inductive case 1: Let \( xs \equiv (\bot:xs') \). 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:

\[
\begin{align*}
\text{length} \ (xs ++ ys) & = \text{length} \ ((\bot:xs') ++ ys) \\
& = \text{length} \ (\bot:(xs' ++ ys)) \\
& = 1 + \text{length} \ (xs' ++ ys) \\
&(\text{IH}) & = 1 + (\text{length} \ xs' + \text{length} \ ys) \\
& = (1 + \text{length} \ xs') + \text{length} \ ys \\
& = \text{length} \ (\bot:xs') + \text{length} \ ys \\
& = \text{length} \ xs + \text{length} \ ys
\end{align*}
\]
Proof of Lemma 6.3.2.15 (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:

\[
\begin{align*}
\text{length} \ (xs ++ ys) \\
= \text{length} \ ((x:xs') ++ ys) \\
= \text{length} \ (x:(xs' ++ ys)) \\
= 1 + \text{length} \ (xs' ++ ys) \\
\text{(IH)} = 1 + (\text{length} \ xs' + \text{length} \ ys) \\
= (1 + \text{length} \ xs') + \text{length} \ ys \\
= \text{length} \ (x:xs') + \text{length} \ ys \\
= \text{length} \ xs + \text{length} \ ys
\end{align*}
\]
Homework

Which of the statements of the lemmas of Chapter 6.3.2 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
Informally

Haskell lists are called

- **partial**, if they are built from the **undefined list**.
  Examples: \( \bot \), \((1: \bot)\), \((1:2:3: \bot)\), \((1:2:3: \bot)\),...

- **partial with possibly undefined values**, if they are partial and at least one of their values is equal to the **undefined value**.
  Examples: \((1: \bot :3: \bot)\), \((1: \bot :3: \bot)\). \(( \bot : \bot :3: \bot)\),...

**Note** the different types of \( \bot \) and \( \bot \) in the above **examples**:

\( \bot :: \text{Int} \)

\( \bot :: [\text{Int}] \)
Partial Lists

Definition 6.3.3.1 (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 \(\bot\).

Definition 6.3.3.2 (Defined Part. List, Part. List w/ Undef. Values)

A partial list is called

- defined, if all its values are defined.
- a partial list with possibly undefined values, if some of its values can be equal to \(\bot\).
Examples

...of lists and partial lists w/ and w/out undefined values:

\[
\text{empty} = [] \\
\text{ns} = 2 : 3 : 5 : 7 : [] \\
\text{ms} = 2 : \text{loop} : 5 : 7 : []
\]

-- Empty list
-- Defined list
-- List w/ undefined values

\[
\text{pempty} = \text{loop} :: [\text{Int}] \\
\text{xs} = 2 : 3 : 5 : 7 : \text{loop} \\
\text{ys} = 2 : \text{loop} : 5 : 7 : \text{loop}
\]

-- Empty partial list
-- Def. partial list
-- Partial list w/ undefined values

Note: The value of all occurrences of \text{loop} in \text{ns}, \text{ms}, \text{xs}, \text{ys}, and \text{pempty} is equal to \bot but of different type:

\begin{itemize}
\item \text{loop} :: \text{Int} in \text{ms} and \text{ys}.
\item \text{loop} :: [\text{Int}] in \text{pempty}, \text{xs}, and \text{ys}.
\end{itemize}
Evaluating Terms w/, w/out Undef. Values (1)

...using reverse, head, tail, and last as examples:

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
Evaluating Terms w/, w/out Undef. Values (2)

...using `length` and `take` as examples:

```
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
```
Note (1)

...the different behaviour is due to making or not-making a pattern match on the values of the argument list by \texttt{length}, \texttt{reverse}, \texttt{take}, \texttt{drop}, \texttt{head}, \texttt{tail}, and \texttt{last}, respectively:

\begin{verbatim}
length :: [a] \to \text{Int}
length [] = 0
length (a:xs) = 1 + length xs  -- No pattern match
                     -- on the head of the
                     -- argument list!

reverse :: [a] \to [a]
reverse [] = []
reverse (a:xs) = reverse xs ++ [a]  -- Pattern match on
                     -- the head of the
                     -- argument list!

reverse :: [a] \to [a]
reverse = foldl (flip (:)) []  -- Same here, even if
                               -- pointfree defined!
\end{verbatim}
Note (2)

...the definitions of `take` and `drop` recalled:

\[
\begin{align*}
take & \:: \text{Int} \rightarrow \text{[a]} \rightarrow \text{[a]} \\
take\ n\ _\ &|\ n\ \leq\ 0\ \ =\ \ [] \\
take\ _\ \ [\ ]\ &\ =\ \ [] \\
take\ n\ (x:xs)\ &\ =\ x\ :\ take\ (n-1)\ xs\ --\ Pattern \\
&\hspace{1cm} --\ match\ on\ the\ head\ of \\
&\hspace{1cm} --\ of\ the\ argument\ list! \\
drop & \:: \text{Int} \rightarrow \text{[a]} \rightarrow \text{[a]} \\
drop\ n\ xs\ |\ n\ \leq\ 0\ &\ =\ xs \\
drop\ _\ \ [\ ]\ &\ =\ []\ --\ No\ pattern \\
drop\ n\ (_:xs)\ &\ =\ drop\ (n-1)\ xs\ --\ match\ on\ the \\
&\hspace{1cm} --\ head\ of\ the \\
&\hspace{1cm} --\ argument\ list! 
\end{align*}
\]
Note (3)

...the definitions of head and last recalled:

\[
\text{head} :: [a] \rightarrow a
\]
\[
\text{head} \ (x:_) \ = \ x
\]

-- Pattern match on the head of the argument list!

\[
\text{tail} :: [a] \rightarrow [a]
\]
\[
\text{tail} \ (_:xs) \ = \ xs
\]

-- No pattern match on the head of the argument list!

\[
\text{last} :: [a] \rightarrow a
\]
\[
\text{last} \ [x] \ = \ x
\]
\[
\text{last} \ (_:xs) \ = \ \text{last} \ xs
\]

-- Pattern match on the argument list
...data and newtype declarations behave differently regarding pattern matching. Consider:

```haskell
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.
```
Note (5)

...the following variant of hello' behaves differently, since pattern matching is no longer required:

```haskell
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 summary: 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 examples.
...introduced in Chapter 6.3.2 apply to

► defined lists
► lists with possibly undefined values

which are built by definition from the empty list `[]`.

By contrast, partial lists are built from the undefined list `⊥` (such as `xs`) and may contain values equal to the undefined value (such as `ys`).

We thus need a new inductive proof principle tailored for partial lists (with possibly undefined values).
Inductive Proofs on Partial Lists

Let $P$ be a property defined on partial lists.

Proof pattern for defined partial lists:

1. Base case: Prove that $P(\bot)$ is true.
2. Inductive case: Assuming that $P(xs)$ is true (induction hypothesis), prove that $P(x:xs)$ is true (induction step).

Proof pattern for partial lists with possibly undefined values:

1. Base case: Prove that $P(\bot)$ is true.
2. Inductive case: Assuming that $P(xs)$ is true (induction hypothesis), prove that $P(\bot:xs)$ and $P(x:xs)$, $x$ a defined value, are true (induction step).
Homework (1)

Does Lemma 6.3.2.7 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 6.3.2.7
Let $f$ be a strict map. Then:

$$(\forall xs :: [a]. \text{xs defined}) \cdot (f \cdot \text{head}) \cdot \text{x}s = \text{head} \cdot (\text{map} \ f \ \text{x}s)$$

Provide a proof or a counter-example to support your claims.
Homework (2)

Which of the statements of the lemmas in Chapter 6.3.2 hold for

- defined partial lists?
- 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 on lists and partial lists.

Proof pattern for lists and partial lists with possibly undefined values:

1. **Base case:** Prove that $P(\bot)$ and $P([\ ])$ are true.

2. **Inductive case:** Assuming that $P(xs)$ is true (induction hypothesis), prove that $P(\bot:xs)$ and $P(x:xs)$, $x$ a defined value, are true (induction step).
Homework

Which of the statements of the lemmas in Chapter 6.3.2 and 6.3.3 hold for

- defined lists and defined partial lists?
- lists and partial lists with possibly undefined values?

Prove your claims or provide counter-examples.
Chapter 6.3.4

Inductive Proofs on Haskell Stream Approximants
Streams

...are infinite sequences of values of the same type.

**Definition 6.3.4.1 (Stream)**

A stream is an infinite sequence of (defined or undefined) values of the same type.

**Definition 6.3.4.2 (Def. Stream, S. w/ Undef. Values)**

A stream is called

- **defined**, if all its values are defined.
- a stream with possibly undefined values, if some of its values can be equal to $\bot$.

**Homework:** Is it meaningful to say, a stream were built from the empty 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 $ys$, $xs \sqsubseteq ys$:

$$
\bot \sqsubseteq 0 : \bot
$$

$$
0 : \bot \sqsubseteq 0 : 1 : \bot
$$

$$
0 : 1 : \bot \sqsubseteq 0 : 1 : 1 : \bot
$$

$$
0 : 1 : 1 : 2 : \bot \sqsubseteq 0 : 1 : 1 : 2 : 3 : \bot
$$

$$
\ldots
$$

$$
\bot \sqsubseteq 0 : 1 : 1 : 2 : 3 : 5 : 8 : \bot
$$

$$
0 : \bot \sqsubseteq 0 : 1 : 1 : 2 : 3 : 5 : 8 : \bot
$$

$$
0 : 1 : 2 : \bot \sqsubseteq 0 : 1 : 1 : 2 : 3 : 5 : 8 : \bot
$$

$$
\ldots
$$

Streams can be approximated by infinite sequences of

- increasingly more accurate partial lists, called PL-approximants.
Illustrating Stream Approximation

...the stream of natural numbers

\[ [1...] = 1 : 2 : 3 : 4 : 5 : 6 : 7 : 8 : 9 : \ldots \]

is approximated by the infinite sequence of more and more accurate PL-approximants, whose limit is the stream itself:

\[
\begin{align*}
\perp \\
\perp 1 : \perp \\
\perp 1 : 2 : \perp \\
\perp 1 : 2 : 3 : \perp \\
\perp 1 : 2 : 3 : 4 : \perp \\
\perp 1 : 2 : 3 : 4 : 5 : \perp \\
\perp 1 : 2 : 3 : 4 : 5 : 6 : \perp \\
\perp 1 : 2 : 3 : 4 : 5 : 6 : 7 : \perp \\
\perp 1 : 2 : 3 : 4 : 5 : 6 : 7 : 8 : \perp \\
\ldots \\
\perp 1 : 2 : 3 : 4 : 5 : 6 : 7 : 8 : 9 : \ldots = [1...] 
\end{align*}
\]
Intuitively

...the undefined list ⊥ is the ‘least defined,’ hence the ‘least accurate’ partial list (or approximant). Sequences of more and more ‘defined’ approximants are getting more and more accurate.

...considering (finite) partial lists

▶ approximations, called approximants, 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

▶ successively more accurate approximants of the stream.
Approximation Order on Partial Lists, Streams

...formalizing the idea of approximation:

Definition 6.3.4.3 (Partially Ordered Set)

A binary relation $R$ on $M$ is called a partially ordered set (or partial order) iff $R$ is reflexive, transitive, and anti-symmetric; the pair $(M, R)$ is called a partial order.

Let $S_{(PL, St)} = \{ s \mid s \text{ partial list or stream} \}$ be the set of partial lists and streams.

Lemma 6.3.4.4 (Approximation Order)

The relation $\sqsubseteq$ on $S_{(PL, St)}$ defined by:

\[
\begin{align*}
\bot & \sqsubseteq xs \\
x : xs & \sqsubseteq y : ys \iff_{df} x \equiv y \land xs \sqsubseteq ys 
\end{align*}
\]

is a partial order, called approximation order, where $\equiv$ denotes equality on partial list/stream elements.
Partial Lists as Stream Approximants

Definition 6.3.4.5 (PL-Approximants)

The set of PL-approximants of a defined stream \( xs \) is defined by \( PL-\text{Approx}(xs) = df \{ \text{take}' \ n \ xs \mid n \in \mathbb{N}_0 \} \), where

\[
\begin{align*}
\text{take}' & : \text{Integer} \to [a] \to [a] \\
\text{take}' \ n \ _ & \mid n \leq 0 \quad = \text{undefined} \\
\text{take}' \ n \ (x:xs) & \quad = x : \text{take}'(n-1)\ xs
\end{align*}
\]

Note: PL-approximants are built from the undefined list, not the empty list; they all have finite length.

Examples:

\[
\begin{align*}
\text{PL-Approx}([1 \ldots]) & = \{\bot, 1: \bot, 1:2: \bot, 1:2:3: \bot, \ldots\} \\
\text{PL-Approx}([1,1 \ldots]) & = \{\bot, 1: \bot, 1:1: \bot, 1:1:1: \bot, \ldots\}
\end{align*}
\]
Chains, Domains

**Definition 6.3.4.6 (Chain)**

A subset $C \subseteq P$ of a partial order $(P, \sqsubseteq)$ is called a **chain**, if the elements of $C$ are totally ordered.

**Definition 6.3.4.7 (Domain)**

A partial order $(D, \sqsubseteq)$ is called a **domain** (or complete partial order (CPO)), if

1. $D$ has a least element $\bot$.
2. $\bigsqcup C$ exists for every chain $C$ in $D$.

$\sqsubseteq$ 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 $\emptyset$ and $\bigsqcup C = \bigcup C$ for every chain $C \subseteq \mathcal{P}(\mathbb{N})$. 

---

559/192
Main Results

**Lemma 6.3.4.8 (Partial Lists and Streams Domain)**

\((S_{(PL,St)}, \sqsubseteq)\) is a domain with the undefined list \(\bot\) as least element, and the order \(\sqsubseteq\) defined in Lemma 6.3.4.4 as approximation order.

**Lemma 6.3.4.9 (PL-Approximants Chain)**

The set \(PL-\text{Approx}(xs)\) of a defined stream \(xs\) is a chain.

**Theorem 6.3.4.10 (Approximation)**

A defined stream \(xs\) is equal to the least upper bound of its PL-approximants set, also called its limit:

\[
\bigsqcup \text{PL-Approx}(xs) = \bigsqcup_{n=0}^{\infty} \text{take}' n \; xs = xs
\]

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 6.3.4.10) as illustrated below:

\[
\begin{align*}
\bot \\
\leq 1 : \bot \\
\leq 1 : 2 : \bot \\
\leq 1 : 2 : 3 : \bot \\
\leq 1 : 2 : 3 : 4 : \bot \\
\leq 1 : 2 : 3 : 4 : 5 : \bot \\
\leq 1 : 2 : 3 : 4 : 5 : 6 : \bot \\
\leq 1 : 2 : 3 : 4 : 5 : 6 : 7 : \bot \\
\leq 1 : 2 : 3 : 4 : 5 : 6 : 7 : 8 : \bot \\
\ldots \\
\leq 1 : 2 : 3 : 4 : 5 : 6 : 7 : 8 : 9 : \ldots = [1..]
\end{align*}
\]
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.

- reverse (reverse xs)) = 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.: 

- \( \text{map (f . g) xs = (map f . map g) xs} \)
  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

and thus allow to prove properties for all finite lists and/or all finite partial lists (with possibly undefined values).

Streams, however, are by definition

▶ infinite sequences of values.

Thus, the induction principles of Chapter 6.3.2 and 6.3.3 are not applicable for free for proving properties on streams, especially in the light of the fact that properties being valid for all PL-approximants 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 6.6.1) admissible properties for streams.

Intuitively, a property is admissible, if it holds for the limit of a PL-approximants set, if it holds for each of its elements.

Equational properties are admissible.

Together with Approximation Theorem 6.3.4.10, this justifies the proceeding 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.

**Proof pattern for defined PL-approximants:**

- **Base case:** Prove that $P(\bot)$ is true.
- **Inductive case:** Assuming that $P(xs)$ is true (induction hypothesis), prove that $P(x:xs)$ is true (induction step).

**Proof pattern for PL-approximants w/ possibly undef. values:**

- **Base case:** Prove that $P(\bot)$ is true.
- **Inductive case:** Assuming that $P(xs)$ is true (induction hypothesis), prove that $P(\bot:xs)$ and $P(x:xs)$, $x$ a defined value, are true (induction step).
Example 1: Induction on PL-Approximants

Lemma 6.3.4.11
We have:

\((\forall \, xs \in [a] \cdot xs \text{ defined stream}) \forall n \in \mathbb{N}.\)

\[\text{take } n \, xs \, ++ \, \text{drop } n \, xs = xs\]

Proof by cases and induction on the structure of \(xs\).
Proof of Lemma 6.3.4.11 (1)

Case 1: Let $n \in \mathbb{IN}$, $n = 0$, and $xs$ be some defined stream. Equational reasoning yields the desired equality:

\[
\text{take } n \ xs \ ++ \ \text{drop } n \ xs = \text{take } 0 \ xs \ ++ \ \text{drop } 0 \ xs \\
= \text{take } \left[ \right] \ ++ \ xs \\
= xs
\]

Case 2: Let $n \in \mathbb{IN}$, $n \geq 1$ be some natural number. We now proceed by induction on the structure of $xs$.

Base case: Let $xs \equiv \bot$. Equational reasoning yields as desired:

\[
\text{take } n \ xs \ ++ \ \text{drop } n \ xs = \text{take } n \ \bot \ ++ \ \text{drop } n \ \bot \\
(\text{Def. take, case exh.}) = \bot \ ++ \ \bot \\
= \bot \\
= \bot
\]

= $xs$
Proof of Lemma 6.3.4.11 (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 $n \in \mathbb{IN}$ the equality $(\text{take} \ (n-1) \ xs' \ ++ \ \text{drop} \ (n-1) \ xs') = xs'$. This allows us to complete the proof as follows:

\[
\begin{align*}
\text{take} \ n \ xs \ ++ \ \text{drop} \ n \ xs &= \text{take} \ n \ (x:xs') \ ++ \ \text{drop} \ n \ (x:xs') \\
&= x : (\text{take} \ (n-1) \ xs' \ ++ \ \text{drop} \ (n-1) \ xs') \\
&= x : xs' \\
&= (x:xs') \\
&= xs \ \\
\end{align*}
\]

(Case 1, IH)
Example 2: Induction on PL-Approximants

Consider the following variant of Lemma 6.3.4.11:

Lemma 6.3.4.12
We have:

\( \forall xs \in [a]. \, \text{xs defined stream} \) \( \forall z \in \mathbb{Z} \).
\[
\text{take } z \, \text{xs} ++ \text{drop } z \, \text{xs} = \text{xs}
\]

Proof by induction on the structure of xs and cases.
Proof of Lemma 6.3.4.12 (1)

Base case: Let $xs \equiv \bot$.

Case 1: Let $z \in \mathbb{Z}$, $z \leq 0$. Equational reasoning yields the desired equality:

$$
\text{take } z \; xs \; ++ \; \text{drop } z \; xs \\
= \; \text{take } z \; \bot \; ++ \; \text{drop } z \; \bot \\
= \; [] \; ++ \; \bot \\
= \; \bot \\
= \; xs
$$

Case 2: Let $z \in \mathbb{Z}$, $z > 0$. Again, equational reasoning yields as desired:

$$
\text{take } z \; xs \; ++ \; \text{drop } z \; xs \\
= \; \text{take } z \; \bot \; ++ \; \text{drop } z \; \bot \\
= \; \bot \; ++ \; \bot \\
= \; \bot \\
= \; xs
$$
Proof of Lemma 6.3.4.12 (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 $(\text{take } (z-1) xs' +\text{ drop } (z-1) xs') = xs'$. This allows us to complete the proof as follows:

\[
\begin{align*}
\text{take } z \ x s \ +\text{ drop } z \ x s \\
&= \text{take } z \ (x:xs') \ +\text{ drop } z \ (x:xs') \\
&= x : (\text{take } (z-1) xs' +\text{ drop } (z-1) xs') \\
(\text{IH}) &= x : xs' \\
&= (x:xs') \\
&= xs
\end{align*}
\]
Example 3: Induction on PL-Approximants

Lemma 6.3.4.13
We have:

\[(\forall xs \in [a]. \; xs \text{ defined stream}). \]
\[
\text{map} \; (f \; \cdot \; g) \; xs = (\text{map} \; f \; \cdot \; \text{map} \; g) \; xs
\]

Proof by induction on the structure of \(xs\).
Proof of Lemma 6.3.4.13 (1)

Base case: Let $xs \equiv \bot$. Equational reasoning yields the desired equality:

$$
\begin{align*}
\text{map} (f \cdot g) xs &= \text{map} (f \cdot g) \bot \\
(\text{Def. map, case exh.}) &= \bot \\
(\text{Def. map, case exh.}) &= \text{map} f \bot \\
(\text{Def. map, case exh.}) &= \text{map} f (\text{map} g \bot) \\
(\text{Def. (.)}) &= (\text{map} f \cdot \text{map} g) \bot \\
&= (\text{map} f \cdot \text{map} g) xs
\end{align*}
$$
Proof of Lemma 6.3.4.13 (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 \( \text{map} \ (f \cdot g) \ xs' = (\text{map} \ f \cdot \text{map} \ g) \ xs' \). This allows us to complete the proof as follows:

\[
\begin{align*}
\text{map} \ (f \cdot g) \ xs &= \text{map} \ (f \cdot g) \ (x:xs') \\
\text{(Def. map)} &= ((f \cdot g) \ x) : \text{map} \ (f \cdot g) \ xs' \\
\text{(IH)} &= ((f \cdot g) \ x) : (\text{map} \ f \cdot \text{map} \ g) \ xs' \\
\text{(2x Def. (\cdot))} &= f \ (g \ x) : (\text{map} \ f \ (\text{map} \ g \ xs')) \\
\text{(Def. map)} &= \text{map} \ f \ (g \ x : \text{map} \ g \ xs') \\
\text{(Def. (\cdot))} &= \text{map} \ f \ (\text{map} \ g \ (x:xs')) \\
&= (\text{map} \ f \cdot \text{map} \ g) \ (x:xs') \\
&= (\text{map} \ f \cdot \text{map} \ g) \ xs \\
\end{align*}
\]

\( \square \)
Homework (1)

In Definition 6.3.4.5, 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 6.3.4.4), the Approximation Theorem 6.3.4.10, and the inductive principle for PL-approximants sets accordingly.

3. Do Lemma 6.3.4.11, 6.3.4.12, and 6.3.4.13 hold for streams with possibly undefined values, too? Prove your claims or provide counter-examples.
Consider Claim 6.3.2.6′, which extends the statement of Lemma 6.3.2.6 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 6.3.2.6′
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 6.3.2.6’ (1)

Base case: Let $xs \equiv \bot$. Equational reasoning yields the desired equality:

\[
\begin{align*}
\text{reverse (reverse } xs) &= \text{reverse (reverse } \bot) \\
\text{(Def. reverse, case exh.)} &= \text{reverse } \bot \\
\text{(Def. reverse, case exh.)} &= \bot \\
\end{align*}
\]
‘Proof’ Attempt of Claim 6.3.2.6’ (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 \( \text{reverse} \ (\text{reverse} \ xs') = xs' \). This allows us to complete the proof as follows:

\[
\begin{align*}
\text{reverse} \ (\text{reverse} \ xs) &= \text{reverse} \ (\text{reverse} \ (x:xs')) \\
&= \text{reverse} \ ((\text{reverse} \ xs') ++ \ [x]) \\
&= \text{reverse} \ [x] ++ \text{reverse} \ (\text{reverse} \ xs') \\
&= \text{reverse} \ (x : []) ++ xs' \\
&= (x : []) ++ xs' \\
&= x : ([] ++ xs') \\
&= x:xs' \\
&= xs
\end{align*}
\]

\( \square \)
Homework (3)

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.,
  \[ \text{reverse} \left( \text{reverse } \text{xs} \right) = \text{xs} \]

for (defined) streams.

Which of the statements of the lemmas in Chapter 6.3.2, 6.3.3, and 6.3.4 hold for

- defined streams?
- streams with possibly undefined elements?

Prove your claims or provide counter-examples.
Approximation Order on Lists, Part. Lists, Streams

Let \( S_{(L,PL,St)} = \{ s \mid s \text{ list or partial list or stream} \} \) be the set of lists, partial lists and streams.

**Lemma 6.3.4.14 (Approximation Order)**

The relation \( \sqsubseteq \) on \( S_{(L,PL,St)} \) defined by:

- \( \bot \sqsubseteq xs \)
- \([ ] \sqsubseteq xs \iff df xs = [ ] \)
- \( x : xs \sqsubseteq y : ys \iff df x \equiv y \land xs \sqsubseteq ys \)

is a partial order, called approximation order.
Partial Lists as List and Stream Approximants

Definition 6.3.4.15 (LPL-Approximants)

The set of LPL-approximants of a defined stream $xs$ is defined by $LPL-\text{Approx}(xs) \overset{df}{=} \{ \text{approx } n \ xs \mid n \in \IN_0 \}$, where

\[
\text{approx} :: \text{Integer} \rightarrow [a] \rightarrow [a]
\]

\[
\text{approx} \ (n+1) \ [] = []
\]

\[
\text{approx} \ (n+1) \ (x:xs) = x : \text{approx} \ n \ xs
\]

Note: There are LPL-approximants built from the undefined list and others built from empty list; they all have finite length.

Examples:

- $LPL-\text{Approx}(\bot) = \{ \bot \}$
- $LPL-\text{Approx}([]) = \{ \bot, [] \}$
- $LPL-\text{Approx}([1,2,3]) = \{ \bot, 1: \bot, 1:2: \bot, 1:2:3: [] \}$
- $LPL-\text{Approx}([1..]) = \{ \bot, 1: \bot, 1:2: \bot, 1:2:3: \bot, \ldots \}$
- $LPL-\text{Approx}([1,1..]) = \{ \bot, 1: \bot, 1:1: \bot, 1:1:1: \bot, \ldots \}$
...approx is similar to take’ used in Definition 6.3.4.5, however, behaves differently when applied to lists (which, by definition, are built from the empty list, not the undefined list):

\[
\text{approx} :: \text{Integer} \rightarrow [a] \rightarrow [a] \\
\text{approx} (n+1) [] = [] \\
\text{approx} (n+1) (x:xs) = x : \text{approx} n xs
\]

Note: Pattern \( n+1 \) matches only positive integers \( \geq 1 \). Thus:

1. \( \text{approx} m \ ys \rightarrow \triangleright \triangleright \ ys \), if \( m > \text{len} \ ys \).

2. \( \text{approx} m \ ys \rightarrow \triangleright \triangleright y_0 : y_1 : \ldots : y_{m-1} : \bot \), if \( m \leq \text{len} \ ys \), i.e., \( \text{approx} \) will cause an error after generating the first \( m \) elements of \( ys \).
Examples

approx 0 [1,2]  \[\rightarrow \bot\]
approx 1 [1,2]  \[\rightarrow \approx (0+1) [1,2]\]
\[\rightarrow 1 : \approx 0 [2]\]
\[\rightarrow 1 : \bot\]
approx 2 [1,2]  \[\rightarrow \approx (1+1) [1,2]\]
\[\rightarrow 1 : \approx 1 [2]\]
\[\rightarrow 1 : \approx (0+1) [2]\]
\[\rightarrow 1 : 2 : \approx 0 []\]
\[\rightarrow 1 : 2 : \bot\]
approx 3 [1,2]  \[\rightarrow \approx (2+1) [1,2]\]
\[\rightarrow 1 : \approx 2 [2]\]
\[\rightarrow 1 : \approx (1+1) [2]\]
\[\rightarrow 1 : 2 : \approx 1 []\]
\[\rightarrow 1 : 2 : \approx (0+1) []\]
\[\rightarrow 1 : 2 : []\]
approx 7 [1,2..]  \[\rightarrow 1 : 2 : 3 : 4 : 5 : 6 : 7 : \bot\]
Intermediate Results

Lemma 6.3.4.16 (Lists, Part. Lists, Streams Domain)

\((S_{(L,PL,St)}, \sqsubseteq)\) is a domain with the undefined list \(\perp\) as its least element.

Lemma 6.3.4.17 (LPL-Approximants Chain)

The set \(LPL\-Approx(xs)\) of a defined list or a defined stream \(xs\) is a chain.

Theorem 6.3.4.18 (Approximation)

A defined list or a defined stream \(xs\) is equal to the least upper bound of its LPL-approximants set, also called its limit:

\[
\bigsqcup_{n=0}^{\infty} LPL\-Approx(xs) = \bigsqcup_{n=0}^{\infty} approx\ n\ xs = xs
\]
Proof Sketch of Theorem 6.3.4.18 for Lists

Let \( xs \equiv (x_0 : x_1 : x_2 : \ldots : x_{\text{len}(xs)-1} : []) \) be a defined list.

\[
\begin{align*}
\biguplus_{n=0}^{\infty} \text{approx } n \; xs &= \biguplus \{ \bot, \\
&\quad x_0 : \bot, \\
&\quad x_0 : x_1 : \bot, \\
&\quad \ldots \\
&\quad x_0 : x_1 : \ldots : x_{n-1} : \bot, \\
&\quad x_0 : x_1 : \ldots : x_{n-1} : [], \\
&\quad \ldots \\
&\quad \} \\
&= x_0 : x_1 : x_2 : \ldots : x_{\text{len}(xs)-1} : [] \\
&= x_0 : x_1 : x_2 : \ldots : x_{\text{len}(xs)-1} : [] \\
&= xs
\end{align*}
\]
Proof Sketch of Theorem 6.3.4.18 for Streams

Let \( xs \equiv (x_0 : x_1 : x_2 : \ldots : x_n : \ldots) \) be a defined stream.

\[
\bigcup_{n=0}^{\infty} \text{approx } n \ \ xs
\]

\[
= \bigcup \left\{ \perp, \quad (n = 0) \\
\quad x_0 : \perp, \quad (n = 1) \\
\quad x_0 : x_1 : \perp, \quad (n = 2) \\
\quad \ldots \\
\quad x_0 : x_1 : \ldots : x_{m-1} : \perp, \quad (n = m) \\
\quad x_0 : x_1 : \ldots : x_m : \perp, \quad (n = m+1) \\
\quad x_0 : x_1 : \ldots : x_{m+1} : \perp, \quad (n = m+2) \\
\quad \ldots \\
\right\}
\equiv \ x_0 : x_1 : x_2 : \ldots : x_n : \ldots
\equiv \ \ xs
Inductive Proofs on LPL-Approximants Sets

...for proving ‘admissible’ properties of streams.

Let $P$ be an equational property defined on LPL-approximants and streams.

Proof pattern for defined LPL-approximants:

- **Base case:** Prove that $P(\bot)$ and $P([\ ])$ are true.
- **Inductive case:** Assuming that $P(xs)$ is true (induction hypothesis), prove that $P(x:xs)$ is true (induction step).

Proof pattern for LPL-approximants w/ possibly undef. values:

- **Base case:** Prove that $P(\bot)$ and $P([\ ])$ are true.
- **Inductive case:** Assuming that $P(xs)$ is true (induction hypothesis), prove that $P(\bot:xs)$ and $P(x:xs)$, $x$ a defined value, are true (induction step).
Homework

Which of the statements of the lemmas in Chapter 6.3.2, 6.3.3, and 6.3.4 hold for

- defined LPL-approximants?
- 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 trees e.g. (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 6.4
Proof by Approximation
Approximation

...a useful principle for proving equality of infinite objects such as streams, which exploits a conclusion on

- Approximation Theorem 6.3.4.10 and 6.3.4.18

making thereby the proof of equality amenable to

- natural (or mathematical) induction.
L-Approximants

Definition 6.4.1 (L-Approximants)

The set of L-approximants of a defined list or a defined stream \(xs\) is defined by \(\text{Approx}(xs) \overset{df}{=} \{ \text{take } n \ xs \mid n \in \mathbb{IN}_0 \}\)

Note, L-approximants are built from the empty list, not the undefined list; they all have finite length.

Examples:

- \(L-\text{Approx}([]) = \{ [] \}\)
- \(L-\text{Approx}([1,2,3]) = \{ [], 1: [], 1:2: [], 1:2:3: [] \}\)
- \(L-\text{Approx}([1..]) = \{ [], 1: [], 1:2: [], 1:2:3: [], \ldots \}\)
Finiteness, Infinity of Sequences

...in terms of $L$-approximant sets.

**Definition 6.4.2 (Finite, Infinite Sequences)**

A sequence of values $xs$ is

1. **finite**, if its $L$-approximants set $L$-$\text{Approx}(xs)$ is finite.
2. **infinite**, if its $L$-approximant sets $L$-$\text{Approx}(xs)$ is infinite.

**Lemma 6.4.3 (Finite, Infinite Sequences)**

A sequence of values $xs$ is

1. **finite**, i.e., a list, if
   \[ \exists m \in \mathbb{IN}. \ (\forall n \in \mathbb{IN}. \ n \geq m). \ \text{take} \ n \ xs = \text{take} \ (n+1) \ xs \]
2. **infinite**, i.e., a stream, if
   \[ \forall n \in \mathbb{IN}. \ \text{take} \ n \ xs \neq \text{take} \ (n+1) \ xs \]
Equality of Sequences

...in terms of approximant sets.

Definition 6.4.4 (Equality of Sequences)

Two sequences of values $xs$ and $ys$ are equal, if their L-approximant sets are equal, i.e.,

$$ L-\text{Approx}(xs) = \{ \text{take } n \text{ } xs \mid n \in \mathbb{IN} \} = \{ \text{take } n \text{ } ys \mid n \in \mathbb{IN} \} = L-\text{Approx}(ys) $$

Lemma 6.4.5 (Equality of Sequences)

Two sequences of values $xs$ and $ys$ are equal, if all their L-approximants are equal, i.e.,

$$ \forall n \in \mathbb{IN}. \text{take } n \text{ } xs = \text{take } n \text{ } ys $$
Equality of Sequences, Lists and Streams

Corollary 6.4.6 (Finite Sequences)
A sequence of values \( xs \) is finite, i.e., a list, if
\[
\exists m \in \mathbb{N}. \ (\forall n \in \mathbb{N}. \ n \geq m). \ \text{take} \ m \ xs = \text{take} \ (n+1) \ xs
\]

Corollary 6.4.7 (Equality of Lists, Streams)
Two lists or two streams \( xs \) and \( ys \) are equal, if
\[
\forall n \in \mathbb{N}. \ \text{take} \ n \ xs = \text{take} \ n \ ys
\]

Corollary 6.4.8 (Equality of Streams)
Two streams \( xs \) and \( ys \) are equal, if
\[
\forall n \in \mathbb{N}_0. \ xs!!n = ys!!n
\]
Main Results (1)

...reducing the proof of stream equality to a proof of set equality.

Theorem 6.4.9 (Approximation, Stream Equality)

For defined streams $xs, ys$ the following claims are equivalent:

1. $xs = ys$
2. $LPL$-Approx$(xs) = LPL$-Approx$(ys)$
3. $PL$-Approx$(xs) = PL$-Approx$(ys)$
4. $L$-Approx$(xs) = L$-Approx$(ys)$
Main Results (2)

...reducing the proof of stream equality to a proof of an equivalent statement accessible to a proof by natural (or mathematical) induction.

Corollary 6.4.10 (Approximation, Stream Equality)

For defined streams $xs, ys$ the following claims are equivalent:

1. $xs = ys$

2. $\forall n \in \mathbb{IN}. \text{approx } n \ xs = \text{approx } n \ ys$

3. $\forall n \in \mathbb{IN}. \text{take}' \ n \ xs = \text{take}' \ n \ ys$

4. $\forall n \in \mathbb{IN}. \text{take} \ n \ xs = \text{take} \ n \ ys$

5. $\forall n \in \mathbb{IN}_0. \ xs!!n = ys!!n$

Note: Proving along the lines of Corollary 6.4.10(5) is usually more convenient than along the lines of Theorem 6.4.9.
Example: Proof by Approximation

Let

\[
\begin{align*}
\text{fac} & : \text{Int} \rightarrow \text{Int} \\
\text{fac} \ 0 & = 1 \\
\text{fac} \ n & = n \times \text{fac} \ (n-1)
\end{align*}
\]

Consider the two definitions \texttt{facs\_mp} and \texttt{facs\_zw}:

\[
\begin{align*}
\text{facs\_mp} & = \text{map} \ \text{fac} \ \texttt{[0..]} \\
\text{facs\_zw} & = 1 : \text{zipWith} \ (*) \ \texttt{[1..]} \ \texttt{facs\_zw}
\end{align*}
\]

generating the stream of factorials 1, 1, 2, 6, 24, 120, 720, ...

We have:

**Lemma 6.4.11**

\[
\forall \ n \in \mathbb{N}_0. \ \text{facs\_mp}!!n = \text{facs\_zw}!!n
\]
Proof by Lemma 6.4.11 (1)

Base case: Let $n=0$. Equational reasoning yields the desired equality:

\[
\text{facs}_\text{mp}!!n = \text{facs}_\text{mp}!!0 \\
= (\text{map fac [0..])!!0} \\
(\text{Def. facs}_\text{mp}) = \text{fac ([0..])!!0} \\
(\text{L. 6.4.12(1)}) = \text{fac 0} \\
(\text{Def. fac}) = 1 \\
(\text{Def. (!!)}) = (1:\text{zipWith (*) [1..] facs}_\text{zw})!!0 \\
(\text{Def. facs}_\text{zw}) = \text{facs}_\text{zw}!!0 \\
= \text{facs}_\text{zw}!!n
\]
Proof by Lemma 6.4.11 (2)

Inductive case: Let \( n \in \mathbb{IN}_0 \). By means of the induction hypothesis (IH), we can assume \( \text{facs\_mp}!!n = \text{facs\_zw}!!n \). As desired we get:

\[
\text{facs\_mp}!!(n+1)
\]

(Def. \( \text{facs\_mp} \)) \[= (\text{map fac} \ [0..])!!(n+1) \]

(L. 6.4.12(1)) \[= \text{fac} \ ([0..])!!(n+1) \]

(Def. \([0..], (!!)\)) \[= \text{fac} (n+1) \]

(Def. \( \text{fac} \)) \[= (n+1) * \text{fac} n \]

(L. 6.4.12(3)) \[= (n+1) * (\text{facs\_mp}!!n) \]

(IH) \[= (n+1) * (\text{facs\_zw}!!n) \]

(Def. \((!!)\)) \[= ([1..])!!n) * (\text{facs\_zw}!!n) \]

(Def. \((*)\)) \[= (*) ([1..])!!n) (\text{facs\_zw}!!n) \]

(L. 6.4.12(2)) \[= (\text{zipWith} \ (* \ [1..]) \text{facs\_zw})!!n \]

(Def. \((!!)\)) \[= (1: \text{zipWith} \ (* \ [1..]) \text{facs\_zw})!!(n+1) \]

(Def. \( \text{facs\_zw} \)) \[= \text{facs\_zw}!!(n+1) \]
Supporting Statement

**Lemma 6.4.12**

For all natural numbers $n \in \mathbb{N}_0$, we have:

1. $(\text{map } f \; \text{xs})!!n = f(\text{xs}!!n)$
2. $(\text{zipWith } g \; \text{xs} \; \text{ys})!!n = g(\text{xs}!!n)(\text{ys}!!n)
3. $\text{fac } n = \text{facs_mp}!!n$

**Homework:** Prove Lemma 6.4.12.
Homework (1)

Consider the two definitions \texttt{fibs\_memo} and \texttt{fibs\_zw}:

\begin{verbatim}
fibs\_memo = [fibs\_x | x <- [0..]]
fibs\_x 0 = 0
fibs\_x 1 = 1
fibs\_x n = fibs\_memo!(n-1) + fibs\_memo!(n-2)
fibs\_zw = 0 : 1 : zipWith (+) fibs\_zw (tail fibs\_zw)
\end{verbatim}

generating the stream of Fibonacci numbers
\[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots\]

Prove by means of natural (mathematical) induction:

\textbf{Lemma 6.4.13}

\[\forall n \in \mathbb{N}_0. \; \texttt{fibs\_memo}!!n = \texttt{fibs\_zw}!!n\]
Homework (2)

Why can’t we build an inductive proof principle for streams on only \textit{L-approximants} (cf. Definition 6.4.1)? Compared to the inductive proof principles of Chapter 6.3.4, this would effectively mean to drop or replace the proof of $P(\bot)$ by a proof of $P(\emptyset)$ in the base case of the inductive proof patterns based on PL- and LPL-approximants of Chapter 6.3.4. Think e.g. on the consequences of ‘proving’ a property like ‘the reverse of the reverse of a stream is the stream itself.’
Chapter 6.4: Further Reading


Chapter 6.5

Coinduction
...another useful principle for proving equality of infinite objects such as streams which complements the principle of proof by approximation.
Equality as Same Observational Behaviour

...informally, equality of two infinite objects such as streams means that the two objects have

► the same ‘observational behaviour.’

For streams, this informally boils down to

► the heads of the streams are the same.
► their tails have the same ‘observational behaviour.’
Equality of Streams

...formally, let \([A]\) denote the set of streams over a set of elements \(A\), and let streams \(f, g \in [A]\) be written as 
\[ f = [f_0, f_1, f_2, f_3, f_4, f_5, \ldots] \] and 
\[ g = [g_0, g_1, g_2, g_3, g_4, g_5, \ldots] \].

Definition 6.5.1 (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 Corollary 6.4.8.
In the following

...we will show how to reduce equality of streams to bisimilarity of streams.

This requires the notions of

- **Labelled transition systems (LTS)** representing streams.
- **Stream bisimulation relations** capturing the notion of ‘same’ behaviour of streams.

and some related supporting notions such as

- **Expansions** of LTS states.
- **Bisimilar** states.
Labeled Transition Systems

Definition 6.5.2 (Labeled Transition System)

A labeled transition system (LTS) is a triple \((Q, A, T)\) where

- \(Q\) is a set of states.
- \(A\) is a set of action labels.
- \(T \subseteq Q \times A \times Q\) is a ternary relation, the so-called transition relation.

Note: If \((q, a, p) \in T\), 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.:

$0.\overline{142857}$, $0.\overline{1428571}$, $0.\overline{14285714}$, $0.142857142857142,...$

$LTS_1, LTS_2$ are LTS representations of the snd and thd one:
Expansion of LTS States

Let \((Q, A, T)\) be an LTS, and \(q \in Q\).

Definition 6.5.3 (Expansion of an LTS State)

1. A finite expansion of \(q\) is a finite sequence of actions \([a_0, a_1, a_2, a_3, \ldots, a_n]\) such that

\[(\forall i \in \mathbb{N}_0. \ i \leq n). \ \exists q_i, q_{i+1} \in Q. \ q_0 = q \land q_i \xrightarrow{a_i} q_{i+1}.
\]

2. An infinite expansion of \(q\) is an infinite sequence of actions \([a_0, a_1, a_2, a_3, \ldots]\) such that

\[\forall i \in \mathbb{N}_0. \ \exists q_i, q_{i+1} \in Q. \ q_0 = q \land q_i \xrightarrow{a_i} q_{i+1}.
\]
Example: Expansion of Digit Stream States

Consider $LTS_1$ representing digit stream $0.\overline{1428571}$:

The unique infinite expansion of state (i.e., node)

- $s$ is $0\overline{1428571}$,
- 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 6.5.4 ((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:

- \( q \xrightarrow{a} q' \implies \exists p' \in Q.\ p \xrightarrow{a} p' \land q' R p' \)
- \( p \xrightarrow{a} p' \implies \exists q' \in Q.\ q \xrightarrow{a} q' \land q' R p' \)

The largest bisimulation on \(Q\) (wrt \(\subseteq\)) is denoted by \(\sim\).

Definition 6.5.5 (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 \( \text{LTS} = (Q, A, T) \) defined as union of \( \text{LTS}_1, \text{LTS}_2 \).

We define relation \( B \) on \( Q \) as follows:

\[ \forall q, q' \in Q. \ q \bowtie q' \text{ iff } q, q' \text{ have the same 'infinite expansion'} \]

\[ \text{LTS} = \text{LTS}_1 \cup \text{LTS}_2 \]

\[ B = \sim \] is the largest bisimulation on the state set \( Q \) of \( \text{LTS} \).
Streams as Labeled Transition Systems

We introduce the following notation:

If \( f = [f_0, f_1, f_2, f_3, f_4, \ldots] \in [A] \) is a stream, then

- \( f_0 \) denotes the head
- \( \overline{f} \) denotes the tail

of \( f \), i.e., \( f = f_0 : \overline{f} \).

Using this notation, \( f \) is represented by the below labeled transition system (which unfolds \( f \) partially):

\[
\begin{array}{c}
\text{LTS representation of } f \\
\begin{array}{c}
\bullet \quad f_0 \\
\bullet \quad \overline{f}
\end{array}
\end{array}
\]
Stream Bisimulation

Definition 6.5.6 (Stream Bisimulation)

A stream bisimulation on $[A]$ is a binary relation $R$ on the set of streams $[A]$, which satisfies:

$$\forall f, g \in [A]. \ f \ R \ g \ \Rightarrow \ f_0 = g_0 \land \ \overline{f} \ R \overline{g}$$

Let $\sim$ denote the largest stream bisimulation on $[A]$. 
Reducing Stream Equality

...to largest stream bisimulation.

Let \( f = [f_0, f_1, f_2, f_3, f_4, \ldots] \), \( g = [g_0, g_1, g_2, g_3, g_4, \ldots] \) be two streams with

\[
\begin{align*}
f & \xrightarrow{f_0} \bar{f}, \\
g & \xrightarrow{g_0} \bar{g}.
\end{align*}
\]

Then:

**Theorem 6.5.7 (Stream Equality as Stream Bisimul.)**

\( f \) 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.

By definition, \(~\) is the largest stream bisimulation. This yields:

Lemma 6.5.8

\[ f \sim g \iff \exists B. \ B \text{ stream bisimulation on } [A] \land f B g \]

Together, Theorem 6.5.7 and Lemma 6.5.8 imply:

Corollary 6.5.9

\( f \) and \( g \) are equal iff

\[ \exists B. \ B \text{ stream bisimulation on } [A] \land f B g \]
Coinductive Proof Pattern

...using Corollary 6.5.9, proving the equality of two streams $f$ and $g$ of $[A]$ requires:

2. Proving that $B$ is a stream bisimulation with $f B g$.

...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 with $xs B ys$. 

Example: Stream Bisimulation $B \subseteq \sim$

...for streams $0.1428571$ and $0.14285714$:

...$0.1428571$, $0.14285714$ are stream bisimilar and hence equal.
Chapter 6.5: Further Reading (1)


Chapter 6.5: Further Reading (2)


Chapter 6.5: Further Reading (3)


Chapter 6.6

Fixed Point Induction
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 more specifically 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 \(\psi\) be a predicate on \(C\), i.e., \(\psi : C \rightarrow IB\).

**Definition 6.6.1 (Admissible Predicate)**

\(\psi\) 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**

\(\psi\) is admissible, if it is expressible as an equation.
Example: Streams, Sequences of Approximants

Recalling that \((S_{(PL,St)}, \sqsubseteq)\) with \(S_{(PL,St)}\) the set of streams and partial lists (cf. Definition 6.3.4.3), and \(\sqsubseteq\) the approximation order defined in Lemma 6.3.4.4, is a CPO (or domain) (cf. Lemma 6.3.4.8), we get as corollary:

**Corollary 6.6.3**

Let \(\psi\) 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 6.3.4.5) with \(\bigsqcup S' = s\). Then:

\[
(\forall s' \in S'. \psi(s')) \Rightarrow \psi(\bigsqcup S') \quad (\iff \psi(s))
\]
Monotonic and Continuous Functions on CPOs

Let \((C, \sqsubseteq_C)\) and \((D, \sqsubseteq_D)\) be CPOs, and let \(f \in [C \to D]\) 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
\[
(\forall C' \subseteq C. \ C' \neq \emptyset \land C' \text{ chain}). \ f(\bigsqcup_C C') = D \ \bigsqcup_D f(C')
\]
(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 called a **fixed point of** \(f\) iff \(f(c) = c\).
2. \(c\) is called the **least fixed point of** \(f\), denoted by \(\mu f\), iff \(\forall d \in C. \ f(d) = d \Rightarrow c \sqsubseteq d\)

Note: 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 \(f : C \to C\) be a continuous function on \(C\), and let \(\psi : C \to IB\) be an admissible predicate on \(C\). Then:

\[
(\forall c \in C. \ \psi(c) \ \Rightarrow \ \psi(f(c))) \ \Rightarrow \ \psi(\mu f)
\]

where \(\mu f\) denotes the least fixed point of \(f\).
The empty set $\emptyset \subseteq C$ is (trivially) a chain.

Since $C$ is a CPO, $\bigsqcup \emptyset$ exists $= \bot_C$ with $\bot_C$ the least element of $C$.

$\psi$ admissible yields $\psi(\bot_C)$.

(Note that $(\forall d \in \emptyset. \psi(d))$ holds trivially; $\psi$ admissible thus implies $\psi(\bigsqcup \emptyset) = \psi(\bot_C) = \text{true}$.)

Using the assumptions of Theorem 6.6.6, we can prove by induction on $n \in \mathbb{IN}_0$:

$\begin{itemize}
  \item $D =_{df} \{f^n(\bot_C) \mid n \in \mathbb{IN}_0\} \subseteq C$ is a chain.
  \item $\forall n \in \mathbb{IN}_0. \psi(f^n(\bot_C))$.
\end{itemize}$

$D$ chain, $\forall d \in D. \psi(d)$, $\psi$ admissible, yields $\psi(\bigsqcup D)$.

Last but not least, Fixed Point Theorem A.5.1.3 (Knaster, Tarski, Kleene) yields $\mu f = \bigsqcup D$.

Thus, we obtain $\psi(\mu f)$, which completes the proof.
Chapter 6.6: Further Reading


Chapter 6.7

Other Approaches, Verification Tools
Chapter 6.7.1
Correctness by Construction
Correctness by Construction

...conceptually, testing and verification are

- *a posteriori* approaches

for proving correctness of a program as they are applied after the program development is finished.

Conceptually dual to testing and verification is the approach of

- correctness by construction

which strives to prove correctness of a program on the fly of its development by proving correctness of every step of the development.

Hence, correctness by construction is conceptually an

- *a priori* (or *on-the-fly*) approach.
Techniques for Correctness by Correctness

...in principle, every proof technique can be made use of by approaches aiming at correctness by construction.

This includes the inductive proof principles discussed in Chapter 6 as well as equational reasoning discussed in Chapter 4, which sometimes is also called proof by program calculation.

Approaches for proven correct rule-based program transformations, however, are prevailing and thus of particular importance.
...the development of a functional pearl starting with a program being

- obviously correct (but inefficient)

by a sequence of transformation steps into a program being (more)

- efficient and still correct

since (ideally) every transformation step is proved correct (cp. Chapter 4), can be considered an approach in the spirit of ensuring correctness by construction.
Chapter 6.7.1: Further Reading (1)


Chapter 6.7.1: Further Reading (2)


Chapter 6.7.2
Selected other Approaches and Tools
Other Approaches and Tools: A Selection (1)

- Programming by contracts (Vytiniotis et al., POPL 2013)
- 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.
- Verifying first-order and call-by-value recursive functional programs (Suter et al., SAS 2011)
  - Tool Leon: Based on extending SMT to recursive programs.
Other Approaches and Tools: A Selection (2)

- Verifying 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).

- Verifying lazy Haskell (Mitchell et al., Haskell 2008)
  - Tool Catch: Based on static analysis; can prove absence of pattern matching failures; evaluated on ‘real’ programs.

- ...
Verified Functional Programming in Agda


...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
Chapter 6: Further Reading (1)


Chapter 6: Further Reading (2)


Chapter 6: Further Reading (3)

Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 18, Programme verifizieren und testen)


Chapter 6: Further Reading (4)


Chapter 6: Further Reading (5)


Chapter 6: Further Reading (6)


Chapter 6: Further Reading (7)


Chapter 6: Further Reading (8)


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)


Chapter 6: Further Reading (10)


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))


Chapter 6: Further Reading (12)


Chapter 6: Further Reading (13)


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)


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)


Chapter 6: Further Reading (16)


Part IV

Advanced Language Concepts
Chapter 7
Functional Arrays
Chapter 7.1
Motivation
Distinctive

...properties of imperative arrays:

+ 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.
  – Their size is fixed (defined at the time of declaration).
Functional Lists and Arrays

Functional lists

- do not enjoy the set of favorable properties of imperative arrays; most importantly, 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 $i$.
- can be arbitrarily long, potentially even infinite.

Functional arrays

- are designed and implemented to get as close as possible to the properties of imperative arrays.
  - Accessing the $i$th element of an array (using `(!)`) takes a constant number of steps, regardless of $i$.
- are of fixed size (defined at the time they are created).
Functional Arrays

...are not supported by the standard prelude of Haskell but by various libraries

- import Array
- import Data.Array.IArray
- import Data.Array.Diff

providing different kinds and implementations of functional arrays:

- Static arrays (w/out destructive update)
- Dynamic arrays (w/ destructive update)
Chapter 7.2

Functional Arrays
Chapter 7.2.1

Static Arrays
Static Arrays

...are supported by the library `Array`:

- `import Array`

which provides three functions for creating static arrays:

- `array bounds list_ofAssociations`
- `listArray bounds list_of_values`
- `accumArray f init bounds list_ofAssociations`
In more detail

...the three functions for creating static arrays:

- \textbf{array} :: Ix a \Rightarrow (a,a) \rightarrow [(a,b)] \rightarrow Array a b
  \textbf{array} \ bounds \ list\_of\_associations

- \textbf{listArray} :: (Ix a) \Rightarrow (a,a) \rightarrow [b] \rightarrow Array a b
  \textbf{listArray} \ bounds \ list\_of\_values

- \textbf{accumArray} :: (Ix a) \Rightarrow (b \rightarrow c \rightarrow b) \rightarrow b
  \rightarrow (a,a) \rightarrow [(a,c)] \rightarrow Array a b
  \textbf{accumArray} \ f \ init \ bounds \ list\_of\_associations
The Index Type Class \texttt{Ix}

...extends the type class \texttt{Ord} (and indirectly type class \texttt{Eq}):

\[
\text{class (Ord a) => Ix a where}
\]

\[
\text{range} \quad :: \quad (a,a) \rightarrow [a] \\
\text{index} \quad :: \quad (a,a) \rightarrow a \rightarrow \text{Int} \\
\text{inRange} \quad :: \quad (a,a) \rightarrow a \rightarrow \text{Bool} \\
\text{rangeSize} \quad :: \quad (a,a) \rightarrow \text{Int}
\]

Members of \texttt{Ix}

- must provide implementations of \texttt{range}, \texttt{index}, \\
  \texttt{inRange}, and \texttt{rangeSize}.
- are (mainly) used for indices of arrays.
Creating Static Arrays: 1st Mechanism

...using the function \texttt{array}, the most fundamental means:

\begin{itemize}
  \item \texttt{array} :: \texttt{Ix a} \Rightarrow (\texttt{a,a}) \rightarrow \left[(\texttt{a,b})\right] \rightarrow \texttt{Array a b}
\end{itemize}

\begin{itemize}
  \item \texttt{array \ bounds list_of_associations}
\end{itemize}

where

\begin{itemize}
  \item \texttt{bounds} specifies the values of the smallest and array largest index.
  \item \texttt{Example:} The bound values \((0,4)\) and \(((1,1),(10,10))\) specify a
    \begin{itemize}
      \item zero-origin vector of length five
      \item one-origin 10 by 10 matrix, respectively.
    \end{itemize}
  \item Note: The components of \texttt{bounds} can be given by arbitrary expressions.
  \item \texttt{list_of_associations} is a list of \texttt{associations} of the form \((i,x)\) specifying that the value of the array element at index position \(i\) is \(x\).\end{itemize}
Examples

Let \( a', f \ n, \) and \( m \) be the following expressions:

\[
\begin{align*}
a' &= \text{array } (1,4) \left[ (3, 'c'), (2, 'a'), (1, 'f'), (4, 'e') \right] \\
f \ n &= \text{array } (0,n) \left[ (i,i*i) \mid i <- [0..n] \right] \\
m &= \text{array } ((1,1),(2,3)) \\
& \quad \left[ ((i,j),(i*j)) \mid i <- [1..2], j <- [1..3] \right]
\end{align*}
\]

These expressions have type

\[
\begin{align*}
a' &: \text{Array Int Char} \\
f &: \text{Int -> Array Int Int} \\
m &: \text{Array (Int,Int) Int}
\end{align*}
\]

and value

\[
\begin{align*}
a' &\rightarrow> \text{array } (1,4) \left[ (1, 'f'), (2, 'a'), (3, 'c'), (4, 'e') \right] \\
f \ 3 &\rightarrow> \text{array } (0,3) \left[ (0,0), (1,1), (2,4), (3,9) \right] \\
m &\rightarrow> \text{array } ((1,1),(2,3)) \left[ \left( ((1,1),1), ((1,2),2), \\
&\quad (1,3),3), (2,1),2), \\
&\quad (2,2),4), (2,3),6) \right]
\end{align*}
\]
Remarks

...arrays have type \texttt{Array a b} where

- \texttt{a} represents the type of the \texttt{index}
- \texttt{b} represents the type of the \texttt{values} of array elements.

Note:

- An array is undefined if any specified index is out of bounds.
- If two associations in the association list have the same index, the value at that index is undefined.

This means: The function \texttt{array} is \texttt{strict} in bounds but \texttt{non-strict (lazy)} in values. Arrays can thus contain ‘undefined’ elements.
Examples

Computing Fibonacci numbers:

```haskell
fibs n = a
    where a = array (1,n) ([ (1,0), (2,1) ] ++
                          [(i, a!(i-1) + a!(i-2)) | i <- [3..n]])
```

Applications:

```haskell
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 array access function (!)

(!) :: Ix a => Array a b -> a -> b

returns the value v :: b at index position i :: a.

Recall: The index type must be a member of type class Ix, which provides maps specifically needed for index operations.
Examples

Computing Fibonacci numbers:

```haskell
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
- `fibs 100!10` --> 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
A Note on Performance

Declaring \texttt{a} locally in a \texttt{where}-clause in the definition of \texttt{fibs}

\begin{itemize}
  \item avoids creating new arrays during computation
  \item is crucial for performance.
\end{itemize}

For comparison consider the definition of \texttt{xfibs}, where \texttt{a} (of a slightly different type) is globally defined:

\begin{verbatim}
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]])
\end{verbatim}
Examples

Applications:

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
xfibs  25!25 ->> ...takes too long to be feasible!

Note: Though correct, evaluating xfibs n is most inefficient
due to the creation of new arrays during the evaluation.
Creating Static Arrays: 2nd Mechanism

...using the function `listArray`, a more sophisticated means:

\[
\text{listArray} :: (Ix \ a) => (\ a, \ a) \to [b] \to \text{Array} \ a \ b
\]

where

\[
\text{bounds} \text{ specifies the values of the smallest and the largest index.}
\]

\[
\text{list\_of\_values} \text{ specifies the values of the array elements in terms of a list.}
\]

**Note:** The function `listArray` is especially useful

\[
\text{for the frequently occurring case where an array is constructed from a list of values given in index order.}
\]
Example

\[ a'' :: \text{Array} \ \text{Int} \ \text{Char} \]
\[ a'' = \text{listArray} \ (1,8) \ "\text{fun prog}\" \]

\[ a'' \rightarrow\rightarrow \text{array} \ (1,8) \ [(1,'f'), (2,'u'), (3,'n'), (4,' '), (5,'p'), (6,'r'), (7,'o'), (8,'g')] \]
Creating Static Arrays: 3rd Mechanism

...using the function `accumArray`, the most powerful means:

\[
\text{accumArray} :: (\text{Ix } a) \Rightarrow (b \rightarrow c \rightarrow b) \rightarrow b \\
\rightarrow (a,a) \rightarrow [(a,c)] \rightarrow \text{Array } a \ b
\]

\[
\text{accumArray } f \text{ init } \text{ bounds list_of_associations}
\]

where

- \( f \) specifies an accumulation function.
- \( \text{init} \) specifies the (default) value the elements of the array shall be initialized with.
- \( \text{bounds} \) specifies the values of the smallest and the largest index.
- \( \text{list_of_associations} \) 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. Instead, values of ‘conflicting’ indices are accumulated via \( f \).
Example 1: A Histogram Function

...using the function `accumArray`:

```haskell
histogram :: (Ix a, Num b) => (a,a) -> [a] -> Array a b

histogram bounds vs =
    accumArray (+) 0 bounds [(i,1) | i <- vs]
```

Applications:

```haskell
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 2: A Prime Number Test

...using the function `accumArray`:

```haskell
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`
Array Operators (1)

...pre-defined array operators:

- (!): 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!

- ...
Array Operators (2)

...the syntactic signatures of the array operators:

- (!) :: (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
- ...

...
Example: The Prime Number Test

Applications (w/ pre-defined functions on arrays):

```
elems (primes 10)
  --> [True,True,False,True,False,True,False,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
```
Example: More Uses of the Array Operators

The setting:

\[
m = \text{array}\ ((1,1),(2,3))\ [((i,j),i\times j) \mid i \leftarrow [1..2], j \leftarrow [1..3]]
:: \text{Array}\ (\text{Int},\text{Int})\ \text{Int}
\]

\[
m \rightarrow array\ ((1,1),(2,3))\ [((1,1),1),((1,2),2),((1,3),3),
((2,1),2),((2,2),4),((2,3),6)]
\]

\[
m!(1,2) \rightarrow 2, m!(2,2) \rightarrow 4, m!(2,3) \rightarrow 6
\]

Applications of array operators:

\[
\text{bounds}\ m \rightarrow ((1,1),(2,3))
\]

\[
\text{indices}\ m \rightarrow [(1,1),(1,2),(1,3),(2,1),(2,2),(2,3)]
\]

\[
\text{elems}\ m \rightarrow [1,2,3,2,4,6]
\]

\[
\text{assocs}\ m \rightarrow [((1,1),1),((1,2),2),((1,3),3),
((2,1),2),((2,2),4),((2,3),6)]
\]

\[
m // [((1,1),4), ((2,2),8)]
\rightarrow array\ ((1,1),(2,3))\ [((1,1),4),((1,2),2),((1,3),3),
((2,1),2),((2,2),8),((2,3),6)]
\]
Updating Arrays: (//) vs. accum

...`accum`, another pre-defined function 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`.

Application:

```
  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 elements at positions `(1,1)` and `(2,2)` to whose values 1 and 4, 4 and 8 have been added, respectively.
Example: A Modified Histogram Function

...illustrating the update operator (//):

```haskell
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:

```haskell
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)]
```
Pre-defined Higher-Order Array Functions

...higher-order functions can be defined on arrays just as on lists.

Examples:

\[
\text{amap} :: (b \rightarrow c) \rightarrow \text{Array} \ a \ b \rightarrow \text{Array} \ a \ c
\]

\[
\text{amap} \ (\backslash x \rightarrow x \times 10) \ a
\]

...yields an array where all elements of \( a \) are multiplied by 10.

\[
\text{ixmap} :: (\text{Ix} \ a, \text{Ix} \ b) \Rightarrow (a,a) \rightarrow (a \rightarrow b) \rightarrow \text{Array} \ b \ c \rightarrow \text{Array} \ a \ c
\]

\[
\text{ixmap} \ b \ f \ a = \text{array} \ b \ [(k,a!f k) \mid k \leftarrow \text{range} \ b]
\]
User-defined Higher-Order Array Functions

The functions `row` and `col` return a row and a column of a matrix, respectively:

```haskell
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
```
Example: Uses of row and col

Applications (with m 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
Dynamic Arrays

...are supported by the library `Data.Array.Diff`:

▶ `import Data.Array.Diff`

The type

▶ `DiffArray` (for dynamic arrays)

replaces the type

▶ `Array` (for static arrays)

...everything else behaves analogously.
Chapter 7.3

Summary
Summing up

Static Arrays

- **Access operator (!):** Access to each array element in constant time.
- **Update operator (//):** No 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 Arrays

- **Update operator (//):** Destructive update; updates take constant time per index.
- **Access operator (!):** Access to array elements may sometimes take longer as for static arrays.
Note

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 16.2 dealing with an imperative robot language for controlling robot actions.
Chapter 7.4

References, Further Reading
Chapter 7: Further Reading (1)


Chapter 7: Further Reading (2)


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)
Chapter 7: Further Reading (3)


Chapter 7: Further Reading (4)


Chapter 7: Further Reading (5)


Chapter 8

Abstract Data Types
Chapter 8.1
Motivation
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 specifikator)
  - 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**: 

▶ Enabling **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 will demonstrate this considering ADT definitions and implementations for

- Stacks
- Queues
- Priority Queues
- Tables
Chapter 8.2

Stacks
Interface Specification

...of the ADT stack, named `Stack` (user-visible):

```haskell
module Stack (Stack,emptyS,is_emptyS,push,pop,top)
where

-- Interface Spec.: Signatures of stack operations
emptyS   :: Stack a
is_emptyS :: Stack a -> Bool
push     :: a -> Stack a -> Stack a
pop      :: Stack a -> Stack a
top      :: Stack a -> a

-- Behaviour Spec.: Laws for stack operations
(1) thru (6) -- cf. next slide; laws
   -- must be ensured by
   -- any implementation.
```
Behaviour Specification

...of the stack operations of the ADT stack (user-visible):

Behaviour Spec.: Laws for stack operations

1) is_emptyS emptyS == True
2) is_emptyS (push v s) == False
3) top emptyS == undef
4) top (push v s) == v
5) pop emptyS == undef
6) pop (push v s) == s

Note: 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):

```haskell
data Stack a = Empty | Stk a (Stack a)
emptyS = Empty
is_emptyS Empty = True
is_emptyS _ = 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):

```haskell
newtype Stack a = Stk [a]
emptyS  = Stk []
is_emptyS (Stk []) = True
is_emptyS (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):

```haskell
    type Stack a = [a]
    emptyS       = []
    is_emptyS [] = True
    is_emptyS _  = 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 implementor of the ADT stack can prove, respectively:

**Lemma 8.2.1 (Consistency and Completeness)**

The 6 laws of the behaviour specification of the ADT stack are consistent and complete.

**Lemma 8.2.2 (Soundness)**

Implementations A and B (and C) satisfy the 6 laws of the behaviour specification of the ADT stack.
Critical Remark

...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
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
(1) thru (6) -- cf. next slide; laws
            -- must be ensured by
            -- any implementation.
Behaviour Specification

...of the queue operations of the ADT queue (user-visible):

**Behaviour Spec.: Laws for queue operations:**

1) \( \text{is_emptyQ emptyQ} \) \( \equiv \) True
2) \( \text{is_emptyQ (enQ v q)} \) \( \equiv \) False
3) \( \text{frontQ emptyQ} \) \( \equiv \) undefined
4) \( \text{frontQ (enQ v q)} \) \( \equiv \) if is_emptyQ q

then \( v \)

else \( \text{frontQ q} \)

5) \( \text{deQ emptyQ} \) \( \equiv \) undefined
6) \( \text{deQ (enQ v q)} \) \( \equiv \) if is_emptyQ q

then emptyQ

else \( \text{enQ ((deQ q) v)} \)

**Note:** 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):

```haskell
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):

```haskell
newtype Queue a = Q ([a],[a])
  where
    front = reverse (fst (Q ([],[])))
    rear = reverse (snd (Q ([],[])))
    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
```

729/192
Verification

Specificator and implementor of the ADT queue can prove, respectively:

**Lemma 8.3.1 (Consistency and Completeness)**

The 6 laws of the behaviour specification of the ADT queue are consistent and complete.

**Lemma 8.3.2 (Soundness)**

Implementations A and B satisfy the 6 laws of the behaviour specification of the ADT queue.
Homework 8.3.3

Implementation B of the ADT queue is more efficient than implementation A. Why?
Chapter 8.4
Priority Queues
Interface/Behaviour Specification

...of the ADT priority queue, named **PQueue** (user-visible):

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

-- Interface Spec.: Signatures of priority queue op's
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):

```haskell
newtype PQQueue 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

Specificator and implementor 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 for homework.
Chapter 8.5

Tables
Chapter 8.5.1

Tables as Functions and Lists
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 initialize 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):

```haskell
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):**

```haskell
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

Specifícator and implementor 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
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):

```haskell
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

Specificator and implementor 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 for homework here.
Chapter 8.6
Displaying ADT Values in Haskell
Displaying ADT Values

...is often necessary but requires some special care, especially in Haskell.

The reasons for this are twofold:

- 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.

- 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.
Displaying Stacks using deriving-Clauses

...is unsafe:

```haskell
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:

```haskell
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:[])
```
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.
Note on Information Hiding and Security (2)

This holds analogously for other ADT implementations:

**Stacks**

```hs
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**

```hs
newtype Queue a = Q [a] deriving Show
newtype PQueue a = PQ [a] deriving Show
```

**Tables**

```hs
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.
Displaying Stacks using instance-Decl.'s (1)

...the safe and 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))`
Displaying Stacks using instance-Decl.’s (2)

This way, the very same output for all 3 implementations:

\[
push\ 3\ (push\ 2\ (push\ 1\ emptyS)) \rightarrow \rightarrow\ 3|2|1|-
\]

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 `showPrec` is an unused precedence value.
Displaying Tables Represented as Functions

...note that there is no general meaningful way to display a function. An instance declaration for

```haskell
define Table a b = Tbl (b -> a)
```

for the type class `Show` could thus be chosen minimal/trivial:

```haskell
instance Show (Table a b) where
  showsPrec _ _ str = showString "<<A Table>>" str
```
Chapter 8.7

Summary
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 the limitation of the module concept in Haskell, the behaviour specification of an ADT can only be provided in terms of 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 and only be done by means of an explicit `instance`-declaration

since a (more convenient) `deriving`-clause 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)]
Arrays as Abstract Data Type in Haskell (2)

```haskell
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
```

```haskell
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:

Chapter 8.8

References, Further Reading
Chapter 8: Further Reading (1)


Chapter 8: Further Reading (2)


Chapter 8: Further Reading (3)


Chapter 8: Further Reading (4)


Chapter 8: Further Reading (5)


Chapter 9
Monoids
Chapter 9.1
Motivation
Types

...equipped with an associative operation and a left-unit and a right-unit like

- **lists** with concatenation (++) and unit []
  
  \[(xs ++ ys) ++ zs = xs ++ (ys ++ zs)\] (associative)
  
  \[[] ++ xs = xs\] (left-unit)
  
  \[xs ++ [] = xs\] (right-unit)

- **Bool** with conjunction (&&) and unit 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
The Type Class Monoid

**Type Class Monoid**

class Monoid m where
  mempty :: m
  mappend :: m -> m -> m
  mconcat :: [m] -> m

-- Default implementation
mconcat = foldr mappend mempty

...monoids are instances of the type class Monoid (and hence types), which obey the monoid laws:

**Monoid Laws**

\[
\begin{align*}
\text{mempty} \ 'mappend' \ x & = x \quad (\text{MoL1}) \\
\text{x} \ 'mappend' \ \text{mempty} & = x \quad (\text{MoL2}) \\
(x \ 'mappend' \ y) \ 'mappend' \ z & = \\
\text{x} \ 'mappend' \ (y \ 'mappend' \ z) \\
\end{align*}
\]
Intuitively

Monoids are types which provide

- a binary operation `mappend`, a value `mempty`, and a function `mconcat`.

The monoid laws

- MoL1 and MoL2 require that `mempty` is a left-unit and a right-unit of `mappend`.
- MoL3 requires that `mappend` is associative.
- The function `mconcat` takes a list of monoid values and reduces them to a single monoid value by using `mappend`.

Note: It is a programmer obligation to 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.
Chapter 9.3

Monoid Examples
Chapter 9.3.1
Lists as Monoid
Lists as Monoid

...making \([a]\) an instance of the type class \texttt{Monoid}:

\[
\begin{align*}
\text{instance } \texttt{Monoid} \ [a] \ &\text{ where} \\
\text{mempty} &\ = \ [] \\
\text{mappend} &\ = \ (++)
\end{align*}
\]

Lemma 9.3.1.1 (Monoid Laws for \([a]\))

For every instance of type \(a\), the instance \([a]\) of \texttt{Monoid} satisfies the three monoid laws \texttt{MoL1}, \texttt{MoL2}, and \texttt{MoL3}, and is hence a monoid, the so-called \texttt{list monoid}.
Examples

...evaluating some terms for illustration:

\[ [1,2,3] \text{`mappend`} [4,5,6] \rightarrow [1,2,3,4,5,6] \]
\[ [1,2,3] \text{`mappend`} \text{mempty} \rightarrow [1,2,3] \]
\[ \text{mempty} \rightarrow [] \]

"Advanced " `mappend` "Functional " `mappend` "Programming"

\rightarrow "Advanced Functional Programming"

"Advanced " `mappend` ("Functional " `mappend` "Programming"

\rightarrow "Advanced Functional Programming")

("Advanced " `mappend` "Functional ") `mappend`

"Programming"

\rightarrow "Advanced Functional Programming"
...commutativity of \texttt{mappend} is not required by the monoid laws. E.g.:

```
"Semester" \texttt{mappend} "Holiday"
\rightarrow "Semester Holiday"
```

is different from

```
"Holiday" \texttt{mappend} "Semester"
\rightarrow "Holiday Semester"
```
Chapter 9.3.2
Numerical Types as Monoids
Numerical Types (and Boolean) as Monoids

Numerical types (as well as the Boolean type) are equipped with more than one operation that behave as required for the monoid operation \texttt{mappend}. E.g.:

\begin{itemize}
  \item [\checkmark] \texttt{*} and \texttt{+} for numerical types
  \item [\checkmark] \texttt{||} and \texttt{&&} for \texttt{Bool}
\end{itemize}

Hence, we will make use of \texttt{newtype} declarations for types of

\begin{itemize}
  \item numerical and Boolean values
\end{itemize}

to allow more than one monoid instance for them.

Moreover, we will use

\begin{itemize}
  \item record syntax
\end{itemize}

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)

...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)
Lemma 9.3.2.1 (Monoid Laws for Sum and Product)

For every numerical instance of type `a`, the instances `(Sum a)` and `(Product a)` of `Monoid` satisfy the three monoid laws `MoL1`, `MoL2`, and `MoL3`, and are hence monoids, the so-called product and sum monoids.
Examples

...evaluating some terms for illustration:

```haskell
getProduct $ Product 3 `mappend` Product 7 \[\rightarrow\] 21
getSum $ Sum 17 `mappend` Sum 4 \[\rightarrow\] 21

getProduct $ Product 3 `mappend` Product 7
  `mappend` Product 11 \[\rightarrow\] 231
getSum $ Sum 3 `mappend` Sum 7 `mappend` Sum 11
  \[\rightarrow\] 21

getProduct . mconcat . map Product $ \[3,7,11\] \[\rightarrow\] 231
getSum . mconcat . map Sum $ \[3,7,11\] \[\rightarrow\] 21

Product 3 `mappend` mempty \[\rightarrow\] Product 3
getSum $ mempty `mappend` Sum 3 \[\rightarrow\] 3
```
Chapter 9.3.3

Bool as Monoid
The All and Any Monoids (1)

...the **all** monoid of **Bool**:

```haskell
class Monoid a where
  mempty :: a
  mappend :: a -> a -> a

instance Monoid All where
  mempty = All True
  mappend x y = All (x && y)
  -- 'All' because True if every argument is true.
```

...the **any** monoid of **Bool**:

```haskell
class Monoid a where
  mempty :: a
  mappend :: a -> a -> a

instance Monoid Any where
  mempty = Any False
  mappend x y = Any (x || y)
  -- 'Any' because True if some argument is true.
```
The All and Any Monoids (2)

Lemma 9.3.3.1 (Monoid Laws for All and Any)

The instances `All` and `Any` of class `Monoid` satisfy the three monoid laws `MoL1`, `MoL2`, and `MoL3`, and are hence monoids, the so-called all and any monoids.
Examples

...evaluating some terms for illustration:

\[
\begin{align*}
\text{getAny} \ $ \ \text{Any True} & \ \\ 
\text{getAll} \ $ \ \text{All True} & \\
\text{getAny} \ $ \ \text{mempty} & \\
\text{getAny} \ . \ \text{mconcat} \ . \ \text{map Any} \ $ & \\
\text{getAll} \ . \ \text{mconcat} \ . \ \text{map All} \ $ & \\
\end{align*}
\]

\[
\begin{align*}
\text{mappend} \ Any \ False & \rightarrow \ True \\
\text{mappend} \ All \ False & \rightarrow \ False \\
\text{mappend} \ Any \ False & \rightarrow \ False \\
\text{mappend} \ \text{mempty} & \rightarrow \ True \\
[False,True,False,False] \rightarrow \ True \\
[False,True,True,False] \rightarrow \ False \\
\end{align*}
\]
Remarks on Numerical and Boolean Monoids

Note:

- For the monoids \((\text{Product } a)\), \((\text{Sum } a)\), \text{Any}, and \text{All}\ the monoid operation \text{mappend} is both \text{associative} and \text{commutative}.

- For most instances of the type class \text{Monoid}, however, this does not hold (and need not to hold). Two such examples are the list monoid \([a]\) and the ordering monoid \text{Ordering} considered next.
Chapter 9.3.4
Ordering as Monoid
Ordering as Monoid (1)

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

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

Note:

- The definition of the operation `mappend` induces an ‘alphabetical’ comparison of two list arguments.
- The operation `mappend` fails to be commutative for the ordering monoid `Ordering`:

```haskell
    LT  'mappend'  GT  ->>  LT
    GT  'mappend'  LT  ->>  GT
```
Lemma 9.3.4.1 (Monoid Laws for Ordering)

The instance `Ordering` of class `Monoid` satisfies the three monoid laws `MoL1`, `MoL2`, and `MoL3`, and is hence a monoid, the so-called ordering monoid.
Examples (1)

...showing some useful applications of mappend.

Note, the two definitions of `lengthCompare` w/ and w/out mappend:

```haskell
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 as can be verified by means of the properties of the monoid operation mappend.
Examples (2)

...as expected both versions of lengthCompare yield:

\[
\text{lengthCompare "his" "ants" \rightarrow LT}
\]

(since string “his” is shorter than string “ants”) and

\[
\text{lengthCompare "his" "ant" \rightarrow GT}
\]

(since string “his” is lexicographically larger than “ant”).
Examples (3)

...further comparison criteria can easily be added and prioritized.

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

```haskell
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 expected we get:

```
lengthCompareExt "songs" "abba" ->> GT
lengthCompareExt "song" "abba" ->> LT
lengthCompareExt "sono" "abba" ->> GT
lengthCompareExt "sono" "sono" ->> EQ
```
Chapter 9.4

Summary and Looking ahead
Summary

Monoids are most useful for defining folds over values of various data structures since folding requires an associative operation.

While for

- lists

folding seems obvious, it is possible for the values of many other data structures, too, e.g.

- trees

This generality 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 (1)

The **Type Constructor Class** `Foldable`:

```haskell
class Foldable f where
  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`. Hence, `f` and `t` are (1-ary) type constructors, not types.
- `Foldable` is thus a **type constructor class**, not just a type class.
- The operations `foldl` and `foldr` of `Foldable` generalize folding of lists to folding of values of other ‘foldable’ data structures.
Looking ahead: Type Constructor Classes (2)

...the type constructor `[]` for lists 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` as 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**, **Monad**, and **Arrow** (cf. Chapters 10, 11, and 12).
Chapter 9.5
References, Further Reading
Chapter 9: Further Reading


Chapter 10
Functors
Types

...whose values can be mapped over compositionally, with a neutral element, like

- **lists with mapL and id**
  
  \[ g \rightarrow a \rightarrow b, \quad h : b \rightarrow c \]
  
  \[
  \text{mapL } g \quad [] = [] \\
  \text{mapL } g \quad (x:x) = (g x) : \text{mapL } g \quad x
  \\
  \text{mapL } (h \ . \ g) \quad x = \text{mapL } h \quad (\text{mapL } g \quad x) \quad \text{(compositional)}
  \\
  \text{mapL } \text{id} \quad x = x \quad \text{(neutral element)}
  \\
  
  - **trees with mapT and id**
    
    \[ g \rightarrow a \rightarrow b, \quad h : b \rightarrow c \]
    
    data Tree a = Leaf a | Node a (Tree a) (Tree a)
    
    \[
    \text{mapT } g \quad (\text{Leaf } v) = \text{Leaf } (g v) \\
    \text{mapT } g \quad (\text{Node } v \quad l \quad r) = \text{Node } (g v) \quad (\text{mapT } g \quad l) \quad (\text{mapT } g \quad r)
    \\
    \text{mapT } (h \ . \ g) \quad t = \text{mapT } h \quad (\text{mapT } g \quad t) \quad \text{(compositional)}
    \\
    \text{mapT } \text{id} \quad t = t \quad \text{(neutral element)}
    \\
    
    should be made an instance of type constructor class Functor.
Chapter 10.1

Motivation
Mapping

...over values is a typical and recurring task. Recall:

[●] Lists

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

[●] Trees

```haskell
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 similarity of tasks performed by functions like

淹没 mapL
淹没 mapT

suggests bundling all types whose values can be mapped over in a unique type class offering an (over-loaded) function

淹没 mapGeneric

which covers mapL, mapT, and many more:

淹没 Type class Functor

Note that Functor is a representative of a new kind of type classes, higher-order type classes or
淹没 type (constructor) classes.
Chapter 10.2
The Type Constructor Class Functor
The Type Constructor Class Functor

Type Constructor Class Functor

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

...functors are instances of the type constructor class Functor (and hence 1-ary type constructors), which obey the functor laws:

Functor Laws

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

Note: It is a programmer obligation to prove that their instances of Functor satisfy the functor laws.
Note

...argument $f$ of Functor is applied to type variables. Hence:

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

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

The functor laws ensure:

- $\text{fmap}$ preserves the “shape of the container type.”
- $\text{fmap}$ does not regroup the contents of the container.
Class Functor, Functor Laws in more Detail

...with added type information:

Class Functor

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

Functor Laws

```
fmap id :: a -> a
    :: a -> a

fmap (h . g) :: c -> b
    :: a -> c
    :: a -> b

fmap id = id (FL1)
```

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

812/1927
Curried and Uncurried View of fmap

Curried view: \texttt{fmap} takes
\begin{itemize}
  \item a polymorphic function \( g :: a \to b \) and yields a polymorphic function \( g' :: f a \to f b \).
\end{itemize}

Example:
\begin{verbatim}
newtype Month a = M a
instance Functor Month where
  fmap g (M v) = M (g v)
\end{verbatim}

\begin{verbatim}
g :: Int \to String
g 1 = "January"
\ldots
g 12 = "December"
\end{verbatim}

\begin{verbatim}
fmap g \rightarrow g'
\end{verbatim}
\begin{itemize}
  \item \( g' :: Month Int \to Month String \)
  \item \( g' (M 1) = M "January" \)
  \item \( g' (M 12) = M "December" \)
\end{itemize}

Uncurried view: \texttt{fmap} takes
\begin{itemize}
  \item a polymorphic function \( g :: a \to b \) and a functor value \( va :: f a \) and yields a new functor value \( vb :: f b \).
\end{itemize}

Example: \begin{verbatim}
\texttt{fmap g (M 8)} \rightarrow \texttt{fmap (M (g 8))} \rightarrow M "August"
\end{verbatim}
Type Classes vs. Type Constructor Classes (1)

Recall the definition of the type class Monoid to compare it with the type constructor class Functor:

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

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 (2)

Type classes and type constructor classes are conceptually equal. They differ in the type of their members:

- **Type constructor classes** (Foldable, Functor, Monad, Arrow,...) have
  - type constructors (e.g., Tree, [], (,), (->),...) as members.

- **Type classes** (Eq, Ord, Num, Monoid,...) have
  - types (e.g., Tree a, [] a, (,) a a, (->) a a,...) as members.

Type constructors are

- maps, which construct new types from given types.

**Examples:** Tuple constructors (,), (,,), (,,,) ; list constructor []; map constructor (->) ; input/output constructor IO, ...
The List and Tree Functors $[]$ and Tree (1)

...making the 1-ary type constructors $[]$ and Tree for lists and trees, respectively, instances of the type constructor class Functor:

```haskell
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:

1. The symbol $[]$ is used above in two roles (over-loaded), as a type constructor in: instance Functor [] where...
2. value of some list type in: fmap g [] = [].
3. 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.
The List and Tree Functors \([\text{Array}]\) and \(\text{Tree}\) (2)

**Lemma 10.2.1 (Functor Laws for \([\text{Array}]\) and \(\text{Tree}\))**

The instances \([\text{Array}]\) and \(\text{Tree}\) of the type constructor class \(\text{Functor}\) satisfy the two functor laws \(\text{FL1}\) and \(\text{FL2}\), respectively, and hence, are functors, the so-called list and tree functor.
The List and Tree Functors \([\ ]\) and \textbf{Tree} (3)

The instance declarations for \([\ ]\) and \textbf{Tree} could have been equivalently but more concisely given as follows:

\begin{verbatim}
instance Functor \([\ ]\) where
  fmap = mapL -- user-defined \texttt{mapL}

instance Functor \([\ ]\) where
  fmap = map -- predefined \texttt{map}

instance Functor \textbf{Tree} where
  fmap = mapT -- user-defined \texttt{mapT}
\end{verbatim}
The List and Tree Functors \[\texttt{[]}\] and \texttt{Tree (4)}

Examples:

\[
\begin{align*}
ms &= [1..5] \\
\text{fmap} (*2) ms &\rightarrow [2,4,6,8,10] \\
\text{fmap} (^3) ms &\rightarrow [1,8,27,64,125] \\
\text{fmap} (3^) ms &\rightarrow [3,9,27,81,243]
\end{align*}
\]

\[
\begin{align*}
t &= \text{Node 2 (Node 3 (Leaf 5) (Leaf 7)) (Leaf 11)} \\
\text{fmap} (*2) t &\rightarrow \text{Node 4 (Node 6 (Leaf 10) (Leaf 14)) (Leaf 22)} \\
\text{fmap} (^3) t &\rightarrow \text{Node 8 (Node 27 (Leaf 125) (Leaf 343)) (Leaf 1331)} \\
\text{fmap} (3^) t &\rightarrow \text{Node 9 (Node 27 (Leaf 243) (Leaf 2187)) (Leaf 177147)}
\end{align*}
\]
Note

...the operation fmap of the type constructor class Functor is
  ▶ the (over-loaded) generic map mapGeneric
that we were looking and striving for.

Members of the type constructor class Functor can be
  ▶ pre-defined
  ▶ user-defined

1-ary type constructors.
Examples of Predefined Type Constructors

...of different arity:

- 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 be also 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.
Notational Remark

Recall, the following notations 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

\[
\text{fac :: } \text{Int} \rightarrow \text{Int} \\
\text{list2pair :: } [a] \rightarrow (a,a)
\]

can equivalently be written in the form:

\[
\text{fac :: } (\rightarrow) \text{Int} \text{Int} \\
\text{list2pair :: } [] \text{a} \rightarrow (a,a) \\
\text{list2pair :: } [a] \rightarrow (,\, ) \text{a} \text{a} \\
\text{list2pair :: } (\rightarrow) \text{[a]} \text{(a,a)} \\
\text{list2pair :: } [] \text{a} \rightarrow (,\, ) \text{a} \text{a} \\
\ldots \\
\text{list2pair :: } (\rightarrow) ([] \text{a}) (,\, ) \text{a} \text{a}
\]

However, we are more familiar with the ‘classical’ forms, which may thus appear more easily comprehensible.
Chapter 10.3

Predefined Functors
Chapter 10.3.1

The Identity Functor
The Identity Functor

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

```haskell
newtype Id a = Id a

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

Lemma 10.3.1.1 (Functor Laws for Id)

The instance `Id` of the type constructor class `Functor` satisfies the two functor laws `FL1` and `FL2`, and hence, is a functor, the so-called identity functor.
Chapter 10.3.2
The Maybe Functor
The Maybe Functor

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

```haskell
data Maybe a = Nothing | Just a

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

**Lemma 10.3.2.1 (Functor Laws for Maybe)**

The instance `Maybe` of the type constructor class `Functor` satisfies the two functor laws `FL1` and `FL2`, and hence, is a functor, the so-called maybe functor.
Examples

\[
\text{fmap} \ (\umat{++} \ "Programming") \ (\text{Just} \ "Functional")
\]
\[
\quad \rightarrow \ \text{Just} \ "Functional \ Programming"
\]

\[
\text{fmap} \ (\umat{++} \ "Programming") \ \text{Nothing}
\]
\[
\quad \rightarrow \ \text{Nothing}
\]
Anti-Example: Invalid Functor Instance (1)

...consider the 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`:

```haskell
data Maybe_with_counter a
    = Nothing_wc | Just_wc Int a deriving Show
```

...making `Maybe_with_counter` an instance of `Functor`:

```haskell
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 will show: The `Maybe_with_counter` instance of `Functor` violates functor law FL1.

Hence, `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 "fun prog"

fmap (++) "prog") (fmap (++) " "") (Just_wc 0 "fun")(Just_wc 0 "fun")
    ->> Just_wc 2 "fun prog"

...while everything is absolutely fine with these examples...
Anti-Example: Invalid Functor Instance (3)

...evaluating the expressions

\( \text{fmap id (Just_wc 0 "fun")} \)

and

\( \text{id (Just_wc 0 "fun")} \)

yield different values:

\[ \text{fmap id (Just_wc 0 "fun")} \rightarrow \text{Just_wc 1 "fun"} \]
\[ \text{id (Just_wc 0 "fun")} \rightarrow \text{Just_wc 0 "fun"} \]

Hence, functor law FL1 is violated: Equality \( \text{fmap id = id} \)
does not hold for the \text{Maybe with counter} instance. Thus:

**Corollary 10.3.2.2 (Invalid Instance)**

\text{Maybe with counter} is not a valid instance of \text{Functor}. 
Chapter 10.3.3
The List Functor
The List Functor

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

\[
\begin{align*}
\text{instance } & \text{Functor } [\:] \text{ where } \\
& \text{fmap } g [\:] = [] \\
& \text{fmap } g (l:ls) = g l : \text{fmap } g ls
\end{align*}
\]

Lemma 10.3.3.1 (Functor Laws for [\:]

The instance [\:] of the type constructor class Functor satisfies the two functor laws FL1 and FL2, and hence, is a functor, the so-called list functor.
Chapter 10.3.4
The Input/Output Functor
The Input/Output Functor

...making the 1-ary type constructor \( \text{IO} \) for input/output an instance of the type constructor class \textbf{Functor}:  

\[
\text{instance Functor IO where}
\]
\[
\text{fmap } g \text{ action} = \text{do result } <- \text{ action}
\]
\[
\text{return } (g \text{ result})
\]

**Lemma 10.3.4.1** (Functor Laws for \( \text{IO} \))

The instance \( \text{IO} \) of the type constructor class \textbf{Functor} satisfies the two functor laws \text{FL1} and \text{FL2}, and hence, is a functor, the so-called input/output functor.
Examples (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.
import Data.Char
import Data.List

The expressions

\[
\text{do } \text{line} \gets \text{fmap} \ (\text{intersperse} \ '-' \ . \ \text{reverse} \ . \ \text{map} \ \text{toUpper}) \ \text{getLine} \\
\text{putStrLn} \ \text{line}
\]

and

\[
(\\text{xs} \rightarrow \text{intersperse} \ '-' \ (\text{reverse} \ (\text{map} \ \text{toUpper} \ \text{xs})))
\]

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 10.3.5
The Either Functor
The Either Functor

...making the 1-ary type constructor \((\text{Either } a)\) an instance of the type constructor class \text{Functor}:

\[
\text{data Either } a \ b = \text{Left } a \mid \text{Right } b
\]

\[
\text{instance Functor } (\text{Either } a) \text{ where}
\]

\[
\text{fmap } g \ (\text{Right } x) = \text{Right } (g \ x)
\]

\[
\text{fmap } g \ (\text{Left } x) = \text{Left } x
\]

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

\textbf{Lemma 10.3.5.1 (Functor Laws for } (\text{Either } a)\text{)}

The instance \((\text{Either } a)\) of the type constructor class \text{Functor} satisfies the two functor laws FL1 and FL2, and hence, is a functor, the so-called \text{either functor}. 
Examples

\[
\text{fmap length (Right "Programming")} \\
\quad \rightarrow \text{ Right 11}
\]

\[
\text{fmap length (Left "Programming")} \\
\quad \rightarrow \text{ Left "Programming"}
\]
Consider the following instance declaration for \((\text{ Either } a)\):

\[
\text{data } \text{Either } a \text{ b } = \text{ Left } a \mid \text{ Right } b
\]

\[
\text{instance Functor (Either a) where}
\]
\[
\text{fmap } g \ (\text{Right } x) = \text{ Right } (g \ x)
\]
\[
\text{fmap } g \ (\text{Left } x) = \text{ Left } (g \ x)
\]

Would this instance declaration be meaningful?

Think about the constraints the above instance declaration imposes on the types which are eligible for \(a\) and \(b\).
Chapter 10.3.6
The Map Functor
The Map Functor

...making the 1-ary type constructor \((\rightarrow) d\) an instance of the type constructor class \textit{Functor}:

\[
\text{instance \textit{Functor} \((\rightarrow) d\) where \hspace{2cm} \text{\textmd{d reminding}}}
\]
\[
\quad \text{fmap} \ g \ h = (\lambda x \rightarrow g \ (h \ x)) \hspace{2cm} \text{\textmd{to domain}}
\]

\textbf{Note:} \textit{Either} and \((\rightarrow)\) are both 2-ary type constructors, i.e., have two arguments. Hence, only the partially evaluated type constructors \((\textit{Either} \ a)\) and \((\rightarrow) d\) can be made instances of \textit{Functor}, since they are 1-ary type constructors.

\textbf{Lemma 10.3.6.1 (Functor Laws for \((\rightarrow) d\))}

The instance \((\rightarrow) d\) of the type constructor class \textit{Functor} satisfies the two functor laws \textit{FL1} and \textit{FL2}, and hence, is a functor, the so-called \textit{map functor}. 
The Map Functor in more Detail

...with added type information:

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

```haskell
instance Functor ((->) d) where
    fmap g h = (
        x -> g (h x)
    )
```

Note: `fmap` defined (as above) by

```haskell
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:

```haskell
instance Functor ((->) d) where
    fmap = (.)
```
Notes on the Map Functor

...for the map functor \((\rightarrow) d\) the type of the generic operation \(\text{fmap}\) of the type constructor class \(\text{Functor}\)

\[
\text{fmap} :: (\text{Functor } f) \Rightarrow (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b
\]
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 \(\text{fmap}\) can be implemented by:

\[
\text{fmap } g\ h = (g \ . \ h)\ \\
\begin{array}{cccc}
:: a \rightarrow b & :: d \rightarrow a & :: a \rightarrow b & :: d \rightarrow a \\
:: (a \rightarrow b) \rightarrow (d \rightarrow a) \rightarrow (d \rightarrow b) & :: d \rightarrow b
\end{array}
\]
Examples (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).
Examples (2)

...recalling the generic type of fmap:

\[
\text{fmap :: (Functor } f) \Rightarrow (a \to b) \to f a \to f b
\]

we get:

Main>\texttt{:t fmap (*2)}
\texttt{fmap \ (*2) :: (Num a, Functor } f) \Rightarrow f a \to f a

Main>\texttt{:t fmap (replicate 3)}
\texttt{fmap (replicate 3) :: (Functor } f) \Rightarrow f a \to f [a]

where

\[
\text{replicate :: Int } \to a \to [a]
\]

\[
\text{replicate } n \ x
\]

\[
\begin{cases} 
  n \leq 0 & \Rightarrow [] \\
  \text{otherwise} & \Rightarrow x : \text{replicate} \ (n-1) \ x 
\end{cases}
\]
Examples (3)

```haskell
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"
```
Examples (4)

Applying \texttt{fmap} to \textit{n-ary maps} (e.g., \texttt{(*), (++)}, $\backslash x \ y \ z \rightarrow \ldots$, \ldots) instead of \textit{1-ary maps} (e.g., \texttt{replicate 3, (*3), (+100)}, \ldots) as so far, we get:

\begin{align*}
\texttt{fmap} \ (\ast) \ \text{(Just 3)} \ &\rightarrow>> \ \text{Just ((\ast) 3)} \\
\texttt{fmap} \ (\ast) \ \text{(Just "fun")} &\rightarrow> \text{Maybe ([Char] -> [Char])} \\
\texttt{fmap} \ \text{compare} \ \text{(Just 'a')} &\rightarrow> \text{Maybe (Char -> Ordering)} \\
\texttt{fmap} \ \text{compare} \ "A list of chars" &\rightarrow> \text{[Char -> Ordering]} \\
\texttt{fmap} \ (\backslash x \ y \ z \rightarrow x + y / z) \ [3,4,5,6] &\rightarrow> \text{(Fractional a) => [a -> a -> a]} \\
\end{align*}

\begin{align*}
a & = \text{\texttt{fmap}} \ (\ast) \ [1,2,3,4] \rightarrow> \text{[Int -> Int]} \\
\texttt{fmap} \ (\backslash f \rightarrow f \ 9) \ a & \rightarrow>> \ [9,18,27,36]
\end{align*}
Note

...some of the previous examples showed
   ▶ lifting
of a map of type
   ▶ \((a \rightarrow b)\)
to type
   ▶ \((f a \rightarrow f b)\)
by \textbf{fmap}. This again shows that \textbf{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 \(\leadsto\) \textbf{curried view}.
2. a functor value \(v :: f a\) and maps \(g\) over \(v\) \(\leadsto\) \textbf{uncurried view}.
Homework

Following the example of the map functor, provide (most general) type information for the following instance declarations of Functor:

1. Identity
2. Maybe
3. List
4. Input/Output
5. Either
6. Tree (cf. Chapter 10.2)
Chapter 10.4
The Type Constructor Class Applicative
The Type Constructor Class Applicative

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

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.
The Applicative Laws

...applicatives are instances of the type constructor class \textbf{Applicative} (and hence 1-ary functors), which obey the applicative laws:

\textbf{Applicative Laws}

\begin{align*}
\text{pure } \text{id} & \triact v = v \quad \text{(AL1)} \\
\text{pure } (\cdot) & \triact u \triact v \triact w = u \triact (v \triact w) \quad \text{(AL2)} \\
\text{pure } g & \triact \text{pure } x = \text{pure } (g \ x) \quad \text{(AL3)} \\
\text{u} & \triact \text{pure } y = \text{pure } (\$ \ y) \triact u \quad \text{(AL4)}
\end{align*}

\textbf{Note:} It is a programmer obligation to prove that their instances of \textbf{Applicative} satisfy the applicative laws.
Class Applicative and Appl. Laws in 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

```
pure id <*> v = v  \hspace{1cm} (AL1)

pure g <*> pure x = pure (g x)  \hspace{1cm} (AL3)
```
An Infix Operator <$> as Alias for fmap

...for a more compelling usage in operation sequences involving both fmap and (<*>).

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

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

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

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

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

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

...or its infix variant ‘fmap’:

```haskell
((++) ‘fmap’ Just "Functional ") <*> Just "Programming"
   ==> Just "Functional Programming"
```
...that defining (\$\$\$\$) by

\[
\text{(<$>$)} :: \text{(Functor } f) \Rightarrow (a \rightarrow b) \rightarrow f \ a \rightarrow f \ b \\
f \ <\$\$\$\$> x = \text{fmap } f \ x
\]

would be valid, too, since the context allows to decide if \( f \) is used as type constructor (\( f \)) or as an argument (\( f \)).
Utility Maps for Applicatives

\[ \text{liftA2} :: (\text{Applicative } f) \Rightarrow (a \rightarrow b \rightarrow c) \rightarrow f\ a \rightarrow f\ b \rightarrow f\ c \]
\[ \text{liftA2 } g\ a\ b = g \ <\$\ >\ a \ <\>*\ >\ b \]

\[ \text{sequenceA} :: (\text{Applicative } f) \Rightarrow [f\ a] \rightarrow f\ [a] \]
\[ \text{sequenceA} \ [] = \text{pure} \ [] \]
\[ \text{sequenceA} \ (x:x:s) = (:) \ <\$\ >\ x \ <\>*\ >\ \text{sequenceA} \ x:s \]

\[ \text{sequenceA} :: (\text{Applicative } f) \Rightarrow [f\ a] \rightarrow f\ [a] \]
\[ \text{sequenceA} = \text{foldr} \ (\text{liftA2} \ (:)) \ (\text{pure} \ []) \]

Examples:

\[ \text{fmap} \ (x \rightarrow [x]) \ (\text{Just\ }4) \rightarrow\text{Just\ }[4] \]
\[ \text{liftA2} \ (:) \ (\text{Just\ }3) \ (\text{Just\ }[4]) \rightarrow\text{Just\ }[3,4] \]
\[ (:) \ <\$\ >\ \text{Just\ }3 \ <\>*\ >\ \text{Just\ }4 \rightarrow\text{Just\ }[3,4] \]
Homework

Provide (most general) type information for the applicative laws AL2 and AL4:

\[
\text{pure (.) <*> u <*> v <*> w = u <*> (v <*> w)} \quad (\text{AL2})
\]

\[
u <*> \text{pure } y = \text{pure ($ y) <*> u} \quad (\text{AL4})
\]
Chapter 10.5
Predefined Applicatives
Chapter 10.5.1

The Identity Applicative
The Identity Applicative

...making the 1-ary type constructor \texttt{Id} an instance of the type constructor class \texttt{Applicative} (conceptually the simplest applicative):

\begin{verbatim}
newtype Id a = Id a

instance Applicative Id where
    pure            = Id
    Id g <*> (Id x) = Id (g x)
\end{verbatim}

\textbf{Note:} \texttt{g} plays the role of the applicative functor.

\textbf{Lemma 10.5.1.1 (Applicative Laws for Id)}

The instance \texttt{Id} of the type constructor class \texttt{Applicative} satisfies the four applicative laws \texttt{AL1}, \texttt{AL2}, \texttt{AL3}, and \texttt{AL4}, and hence, is an applicative, the so-called \texttt{identity} applicative.
The Identity Applicative in more Detail

...with added type information:

\[
\begin{align*}
\text{pure} & : (\text{Applicative } f) \Rightarrow a \rightarrow f a \\
(\langle*\rangle) & : (\text{Applicative } f) \Rightarrow f (a \rightarrow b) \rightarrow f a \rightarrow f b
\end{align*}
\]

\[
\text{instance Applicative Id where} \\
\text{pure} \quad = \quad \text{Id} \\
:: a \rightarrow \text{Id a} \\
:: a \rightarrow \text{Id a}
\]

\[
\begin{align*}
\text{Id } g \quad & \langle*\rangle \quad \text{Id } x \\
:: (a \rightarrow b) \quad & :: a \\
\text{Id } (a \rightarrow b) \quad & \text{Id } a \\
:: \text{Id b} \\
:: \text{Id b}
\end{align*}
\]

\[
\begin{align*}
\text{Id } (g \quad x) \\
:: a \rightarrow b \\
:: a \\
:: b
\end{align*}
\]
Chapter 10.5.2

The Maybe Applicative
The Maybe Applicative

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

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

Note: g plays the role of the applicative functor.

Lemma 10.5.2.1 (Applicative Laws for Maybe)

The instance Maybe of the type constructor class Applicative satisfies the four applicative laws AL1, AL2, AL3, and AL4, and hence, is an applicative, the so-called maybe applicative.
The Maybe Applicative in more Detail

...with added type information:

\[
\begin{align*}
\text{pure} & : (\text{Applicative } f) \Rightarrow a \rightarrow f a \\
(\text{<*>}) & : (\text{Applicative } f) \Rightarrow f (a \rightarrow b) \rightarrow f a \rightarrow f b \\
\text{fmap} & : (\text{Functor } f) \Rightarrow (a \rightarrow b) \rightarrow f a \rightarrow f b
\end{align*}
\]

\[
\text{instance Applicative Maybe where}
\]

\[
\begin{align*}
\text{pure} & = \text{Just} \\
:: a \rightarrow \text{Maybe } a & :: a \rightarrow \text{Maybe } a \\
\text{Nothing} \text{<*> } _ & = \text{Nothing} \\
:: \text{Maybe } (a \rightarrow b) & :: \text{Maybe } a & :: \text{Maybe } b \\
& :: \text{Maybe } b \\
(\text{Just } g) \text{<*> } \text{something} & = \text{fmap } g \text{ something} \\
:: \text{Maybe } (a \rightarrow b) & :: \text{Maybe } a & :: a \rightarrow b & :: \text{Maybe } a \\
& :: \text{Maybe } b & :: \text{Maybe } b
\end{align*}
\]
Examples (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
Examples (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
```
Examples (3)

\[
\text{pure (+) } \gg\gg \text{Nothing } \gg\gg \text{Just 5} \\
\quad \rightarrow\rightarrow \text{Just (+) } \gg\gg \text{Nothing } \gg\gg \text{Just 5} \\
\quad \rightarrow\rightarrow (\text{fmap (+) Nothing}) \gg\gg \text{Just 5} \\
\quad \rightarrow\rightarrow \text{Nothing } \gg\gg \text{Just 5} \\
\quad \rightarrow\rightarrow \text{Nothing}
\]

\textbf{Note:} The operator \((\gg\gg)\) is left-associative, i.e.:

\[
\text{pure (+) } \gg\gg \text{Just 3 } \gg\gg \text{Just 5} = \\
(\text{pure (+) } \gg\gg \text{Just 3}) \gg\gg \text{Just 5}
\]
Chapter 10.5.3
The List Applicative
The List Applicative

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

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

Lemma 10.5.3.1 (Applicative Laws for `[]`)

The instance `[]` of the type constructor class Applicative satisfies the four applicative laws AL1, AL2, AL3, and AL4, and hence, is an applicative, the so-called list applicative.
The List Applicative in more Detail

...with added type information:

```haskell
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 ]
```

Examples (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]
```

875/1927
Examples (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!"]
Examples (3)

\[ \text{filter (}>50)\ (\ast)\ \langle\langle\ [2,5,10] \ \langle\ast\rangle\ [8,10,11]\]

\[-\rightarrow\ \text{filter (}>50)\ (\text{fmap}\ (\ast)\ [2,5,10])\ \langle\ast\rangle\ [8,10,11]\]

\[-\rightarrow\ \text{filter (}>50)\ [(2\ast),(5\ast),(10\ast)]\ \langle\ast\rangle\ [8,10,11]\]

\[-\rightarrow\ \text{filter (}>50)\ [f\ x\ |\ f\ <-\ [(2\ast),(5\ast),(10\ast)],\ x\ <-\ [8,10,11]]\]

\[-\rightarrow\ \text{filter (}>50)\ [16,20,22,40,50,55,80,100,110]\]

\[-\rightarrow\ \text{filter (}>50)\ [16,20,22,40,50,55,80,100,110]\]

\[-\rightarrow\ [55,80,100,110]\]
Examples (4)

...the previous example shows that expressions using list comprehension

\[ [x*y \mid x \leftarrow [2,5,10], y \leftarrow [8,10,11]] \]
\[ \rightarrow [16,20,22,40,50,55,80,100,110] \]

...can alternatively be written using (\(<$>\)) and \(<*>\):

\((*) \ <$> [2,5,10] \ <*> [8,10,11] \)
\[ \rightarrow [16,20,22,40,50,55,80,100,110] \]
Chapter 10.5.4
The Input/Output Applicative
The Input/Output Applicative

...making the 1-ary type constructor \texttt{IO} an instance of the \texttt{type}
constructor class \texttt{Applicative}:

\begin{verbatim}
instance Applicative IO where
  pure = return
  a <*> b = do g <- a
              x <- b
              return (g x)
\end{verbatim}

Lemma 10.5.4.1 (Applicative Laws for \texttt{IO})

The instance \texttt{IO} of the type constructor class \texttt{Applicative}
satisfies the four applicative laws \texttt{AL1}, \texttt{AL2}, \texttt{AL3}, and \texttt{AL4},
and hence, is an applicative, the so-called input/output applicative.
The Input/Output Applicative in more Detail

...with added type information:

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

instance Applicative IO where

pure = return

:: a -> IO a

(:=) = do

:: IO (a -> b)

:: IO a

:: a -> b

:: IO (a -> b)

:: a

:: b

:: IO a

:: a

:: b

:: IO b
```

881/1927
Examples

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

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

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

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

```haskell
myAction' :: IO ()
myAction' =
  do a <- (++) <$> getLine <*> getLine
     putStrLn $ "Concatenation yields: " ++ a
```
Chapter 10.5.5
The Either Applicative
1. Make type constructor \((\text{Either} \ a)\) an instance of \textbf{Applicative}.

2. Provide (most general) type information for the defining equations of the applicative operations \texttt{pure} and \texttt{(<>)} of \((\text{Either} \ a)\).

3. Prove that \((\text{Either} \ a)\) satisfies the applicative laws.
Chapter 10.5.6

The Map Applicative
The Map Applicative

...making the 1-ary type constructor \((\rightarrow) d\) an instance of the \textbf{type constructor class Applicative}:

\begin{verbatim}
instance Applicative ((->) d) where
    pure x = (\_ -> x)
    g <*> h = \x -> g x (h x)
\end{verbatim}

\textbf{Lemma 10.5.6.1 (Applicative Laws for \((\rightarrow) d\))}

The instance \((\rightarrow) d\) of the type constructor class Applicative satisfies the four applicative laws AL1, AL2, AL3, and AL4, and hence, is an applicative, the so-called \textbf{map applicative}. 

886/1927
The Map Applicative in more Detail

...with added type information:

\[
\text{pure} :: (\text{Applicative } f) \Rightarrow a \to f a \\
(\langle*\rangle) :: (\text{Applicative } f) \Rightarrow f (a \to b) \to f a \to f b
\]

\[
\text{instance Applicative } (\to) \text{ where} \\
\text{pure } x = (\_ \to x) \\
\quad :: a \quad :: d :: a \\
\quad :: ((\to) d) a
\]

\[
g \langle*\rangle h = \_ \to g \_ (h \_)
\]

\[
\quad :: ((\to) d) (a \to b) \\
\quad :: d \to (a \to b)
\]

\[
\quad :: ((\to) d) a \\
\quad :: d \to a
\]

\[
\quad :: d :: d :: d \\
\quad :: a \\
\quad :: b \\
\quad :: d \to b \\
\quad :: ((\to) d) b)
\]
Examples

pure 3 "Hello"
->> (pure 3) "Hello"  \hspace{1cm} \textit{(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
...dealing with the map applicative.

Complete the stepwise evaluation of the below example:

$$\begin{align*}
  & (\lambda x \ y \ z \to [x,y,z]) \ <$$(+3)<*>(*2)<*>(/2) $ 5 \\
  \rightarrow & (fmap \ (\lambda x \ y \ z \to [x,y,z]) \ (+3)) \ <$*)(2)<*>(/2) $ 5 \\
  \rightarrow & ((\lambda x \ y \ z \to [x,y,z]) \ . \ (+3)) \ <$*)(2)<*>(/2) $ 5 \\
  \rightarrow & \ldots \\
  \rightarrow & [8.0,10.0,2.5]
\end{align*}$$
Chapter 10.5.7

The Ziplist Applicative
The Ziplist Applicative (1)

...making the 1-ary type constructor `ZipList` an instance of the `type constructor class` `Applicative`:

```haskell
newtype ZipList a = ZL [a]
    -- the newtype declaration is required since []
    -- can not be made a second 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)
```

Intuitively: `<>` applies the first function to the first value, the second function to the second value, and so on.
The Ziplist Applicative (2)

Recall:

\[
\text{repeat} :: a \rightarrow [a]
\]
\[
\text{repeat } x = x : \text{repeat } x \quad \text{-- generates stream } [x,.x,..]
\]
\[
\text{zipWith} :: (a \rightarrow b \rightarrow c) \rightarrow [a] \rightarrow [b] \rightarrow [c]
\]
\[
\text{zipWith } _ _ [] _ = []
\]
\[
\text{zipWith } _ _ _ [] = []
\]
\[
\text{zipWith } f (x:xs) (y:ys) = f x y : \text{zipWith } f xs ys
\]

Lemma 10.5.7.1 (Applicative Laws for ZipList)

The instance \text{ZipList} of the type constructor class \text{Applicative} satisfies the four applicative laws \text{AL1}, \text{AL2}, \text{AL3}, and \text{AL4}, and hence, is an applicative, the so-called ziplist applicative.
The Ziplist Applicative in more Detail

...with added type information:

\[
\begin{align*}
\text{pure} & : (\text{Applicative } f) \Rightarrow a \rightarrow f a \\
(\lt\ast\gt) & : (\text{Applicative } f) \Rightarrow f (a \rightarrow b) \rightarrow f a \rightarrow f b
\end{align*}
\]

instance Applicative ZipList where

\[
\begin{align*}
pure \ x & \equiv \text{ZL} (\text{repeat } x) \\
\text{ZL} \ gs \lt\ast\gt\text{ZL} \ xs & \equiv \text{ZL} (\text{zipWith}\ (\lambda g \ x \rightarrow g x) \ gs \ xs)
\end{align*}
\]
Examples

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')]

894/1927
Chapter 10.6
Kinds of Types and Type Constructors
Kinds of Types and Type Constructors

Like values, also

- types
- type constructors

have types themselves, so-called

- kinds.
Kinds of Types

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

Examples:

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

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

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

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

where * (read as “star” or as “type”) indicates that the type is ‘concrete’ or ‘final’, i.e., a type accepting no type arguments.
Type Constructors

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

Examples:

The type constructors Maybe, Either, and Tree

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

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

```haskell
Maybe Int :: * -- a concrete type
Either Int String :: * -- a concrete type
Tree Int :: * -- a concrete type
```
Kinds of Type Constructors

Like concrete types, type constructors have types, called kinds, too.

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 :: * -> *  -- like Maybe.

ghci> :k (->)
(->) :: * -> * -> *  -- like Either.
Kinds of Partially Evaluated Type Constructors

Like functions, also type constructors can be partially evaluated.

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.
Recalling the definition of the type constructor class `Functor`

```haskell
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`.
Chapter 10.7
References, Further Reading
Chapter 10: Further Reading (1)


Chapter 10: Further Reading (2)


Chapter 11

Monads
Monads: A Suisse Knife for Programming

Monadic programming is well suited for problems involving:

- Global state
  - Updating data during computation is often simpler than making all data dependencies explicit (state monad).

- Huge data structures
  - No need for replicating a data structure that is not needed otherwise.

- Exception and error handling
  - Maybe monad

- Side-effects, explicit sequencing and evaluation orders
  - Canonical scenario: Input/output operations (IO monad).
Chapter 11.1

Motivation
Motivation

...monads, a mundane approach for

- functional composition, for linking and sequencing functions!

The monad approach succeeds in

- linking and sequencing functions

whose types are incompatible and thus not amenable to

- ordinary functional composition (.)
Chapter 11.1.1

Functional Composition Reconsidered
Functional Composition

...means specifying the sequence of applications of functions:

\[(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)\]

\[(g \circ f) x = g \circ (f \ x)\]

If \(f\) and \(g\) are two functions of type:

\[f :: a \rightarrow b\]
\[g :: b \rightarrow c\]

then their composition is a function of type:

\[(g \circ f) :: a \rightarrow c\]

and applying \((g \circ f)\) to some argument \(x\) means: Applying \(f\) to \(x\) first, applying second \(g\) to the result of \(f\) for \(x\):

\[(g \circ f) x = g \circ (f \ x) = \begin{align*}
&\text{let } f\_result = f \ x \\
&g\_result = g \ f\_result \\
&\text{in } g\_result
\end{align*}\]
R2L, L2R Sequencing of Function Applications

Sequencing from right to left (R2L):

(·) :: (b → c) → (a → b) → (a → c)

(g · f) x = g (f v)

= let f_result = f x
    g_result = g f_result
    in g_result

...enables R2L application sequences of the form:

(k · (⋯ · (h · (g · f))⋯))

Sequencing from left to right (L2R):

(;) :: (a → b) → (b → c) → (a → c)

(f ; g) = (g · f)

...enables L2R application sequences of the form:

(((⋯((f ; g) ; h) ; ⋯) ; k)
R2L, L2R Sequencing: Two Derived Variants

The L2R sequencing variant (suggested by `( ; `)):

\[
(\ggg; ;) :: a \rightarrow (a \rightarrow b) \rightarrow b
\]

\[
x \ggg; f = f \ x
\]

...enables L2R application sequences of the form:

\[
(\ldots(((x \ggg; f) \ggg; g) \ggg; h) \ggg; \ldots \ k)
\]

\[
= x \ggg; f \ggg; g \ggg; h \ggg; \ldots \ k
\]

The R2L sequencing variant (suggested by `( . `)):

\[
(\lll<<) :: a \rightarrow (a \rightarrow b) \rightarrow b
\]

\[
f \lll<< x = f \ x
\]

-- Note: `( .<<)` = `( $ )`

\[
( = f \ $ x )
\]

...enables R2L application sequences of the form:

\[
(k \ldots \lll<< (h \lll<< (g \lll<< (f \lll<< x)))\ldots)
\]

\[
= k \ldots \lll<< h \lll<< g \lll<< f \lll<< x
\]
Putting things together: It’s all on Notation

...right-to-left (R2L) sequencing:

\[(g \cdot f) x\]  -- canonical sequencing notation

\[= g (f x)\]

\[= \text{let } f\_\text{result} = f x\]
\[\text{g\_result} = g f\_\text{result}\]
\[\text{in } g\_\text{result}\]

\[= g .<< f .<< x\]  -- notational variant

...left-to-right (L2R) sequencing:

\[(f ; g) x\]  -- derived not. variant

\[= (g \cdot f) x\]

\[= g (f x)\]

\[= \text{let } f\_\text{result} = f x\]
\[\text{g\_result} = g f\_\text{result}\]
\[\text{in } g\_\text{result}\]

\[= x >>; f >>; g\]  -- convenient not. variant
One more derived Sequencing Operation

...for left-to-right (L2R) sequencing:

\[(>;) :: a \rightarrow b \rightarrow b\]
\[x >; y = x >>; \_ \rightarrow y\]
\[(= y)\]

\[(>;)\] enables L2R application sequences of the form:

\[(\ldots (((x >; u) >; v) >; w) >; \ldots z)\]
\[= x >; u >; v >; w >; \ldots z\]
\[-\rightarrow z\]

which seems quite useless (and a notational overkill for just saying ‘forget and drop the first argument’) (but not so its monadic counterpart \((>>)\)!)

Keep in mind

The monadic sequencing operations:

\[
\text{(||=)} :: m \ a \rightarrow (a \rightarrow m \ b) \rightarrow m \ b \\
\text{(||)} :: m \ a \rightarrow m \ b \rightarrow m \ b
\]

...are the counterparts of:

\[
\text{(||;)} :: a \rightarrow (a \rightarrow b) \rightarrow b \\
\text{(|;)} :: a \rightarrow b \rightarrow b
\]
On Commonalities and Differences (1)

...of the monadic sequencing operations:

\[(\ggg=) \::\ m \ a \rightarrow (a \rightarrow m \ b) \rightarrow m \ b\]
\[c \ggg= f = \ldots\] -- Needs an m-specific implementation

\[(\ggg) \::\ m \ a \rightarrow m \ b \rightarrow m \ b\]
\[c \gg k = c \ggg= \_ \rightarrow k\]

...and their counterparts:

\[(\ggg;)\] \::\ a \rightarrow (a \rightarrow b) \rightarrow b
\[x \ggg; f = f \ x\] -- One implementation -- fits all types

\[(>;)\] \::\ a \rightarrow b \rightarrow b
\[x >; y = x \ggg; \_ \rightarrow y\]
On Commonalities and Differences (2)

\[(\text{>>;}) : a \rightarrow (a \rightarrow b) \rightarrow b\]
\[
x \text{ >>; } f = f \ x
\]
\[
:: a \quad :: a \rightarrow b \quad :: b
\]

\[(\text{>;} ) : a \rightarrow b \rightarrow b\]
\[
x \text{ >; } y = x \text{ >>; } \_ \rightarrow y
\]
\[
:: a \quad :: b \quad :: a \quad :: a \rightarrow b
\]
\[
\quad :: b \quad :: b
\]

\[(\text{>>=} ) : m \ a \rightarrow (a \rightarrow m \ b) \rightarrow m \ b\]
\[
c \text{ >>=} f = \ldots
\]
\[
:: m \ a \quad :: a \rightarrow m \ b \quad :: m \ b
\]

\[(\text{>>}) : m \ a \rightarrow m \ b \rightarrow m \ b\]
\[
c \text{ >> } k = c \text{ >>=} \_ \rightarrow k
\]
\[
:: m \ a \quad :: m \ b \quad :: m \ a \quad :: a \rightarrow m \ b
\]
\[
\quad :: m \ b \quad :: m \ b
\]
Note

...(>>=) is of type \((m a \rightarrow (a \rightarrow m b) \rightarrow m b)\), not of type \((m a \rightarrow (m a \rightarrow m b) \rightarrow m b)\)!

A sequencing operation \((>>>=)\):

\[
(>>>=) \::\ m a \rightarrow (m a \rightarrow m b) \rightarrow m b \\
c >>>= f = f \ c
\]

could be implemented once and for all fitting all types just as the implementation of \((>>;\) fits all types:

\[
(>>;\) \::\ a \rightarrow (a \rightarrow b) \rightarrow b \\
x >>; f = f \ x
\]

Often, however, we are lacking functions of type \((m a \rightarrow m b)\) but have functions of type \((a \rightarrow m b)\) instead.
Chapter 11.1.2
Example: Debug Information
**Objective**

...enhance two functions \( f : : a \rightarrow b, \ g : : b \rightarrow c \) such that they collect and output debug information during computation:

- type \( \text{Debug\_Info} = \text{String} \)

To this end, replace \( f \) and \( g \) by functions \( f' \) and \( g' \), which are as \( f \) and \( g \) but yield additionally to the results of \( f \) and \( g \) a piece of debug information:

\[
\begin{align*}
f' & : : a \rightarrow (b, \text{Debug\_Info}) \\
g' & : : b \rightarrow (c, \text{Debug\_Info})
\end{align*}
\]

Note: \( f' \) and \( g' \) can not be linked and sequenced using \( (\cdot) \) since their argument and result types are incompatible and do not fit to each other.
Ad hoc Sequencing

...to overcome this problem, we could define a new function \( h \) whose implementation realizes the linking \( g \) and \( f \), i.e., of sequentially composing them:

\[
-- \ 'h = f \ link \ g' \ w/ \ the \ meaning: \ first \ f, \ then \ g \\
\begin{aligned}
h &: a \rightarrow c \\
h \ x &= \ \text{let} \ (f\_\text{result}, f'\_\text{info}) = f' \ x \\
& \quad (g\_\text{result}, g'\_\text{info}) = g' \ f\_\text{result} \\
& \quad \text{in} \ (g\_\text{result}, g'\_\text{info} ++ f'\_\text{info})
\end{aligned}
\]

Though working this were impractical as it continuously required implementing new functions (like \( h \)) which realize the sequencing of a pair of functions (like \( f' \) and \( g' \)).
A new Sequencing Operator \texttt{link dbg}

...more conveniently sequencing could be handled by introducing a function \texttt{link dbg} for linking functions like $f'$ and $g'$:

\[
\text{link dbg} :: (a, \text{Debug_Info}) \rightarrow (a \rightarrow (b, \text{Debug_Info}))
\rightarrow (b, \text{Debug_Info})
\]

\[
\text{link dbg} (x, s) g = \text{let } (g\_result, g\_info) = g x
\text{ in } (g\_result, s ++ g\_info)
\]

Note, \texttt{link dbg} allows us to sequence $f'$ and $g'$ comfortably:

\[
f' x \ '\text{link dbg}' \ g' \ ( = \ h x )
\]
Example: Sequencing with link_dbg

Let:

\[
\begin{align*}
  f :: a \rightarrow b & \quad f' :: a \rightarrow (b, \text{Debug\_Info}) \\
  f \; x &= \ldots & f' \; x &= (f \; x, "f \; called, ") \\
  g :: b \rightarrow c & \quad g' :: b \rightarrow (c, \text{Debug\_Info}) \\
  g \; y &= \ldots & g' \; y &= (g \; y, "g \; called, ")
\end{align*}
\]

Then:

\[
\begin{align*}
  f' \; x \; '\text{link\_dbg}' \; g' \; '\text{link\_dbg}' \; (\forall z \rightarrow (z, "done.")) \\
  \quad \rightarrow \rightarrow (f \; x,"f \; called, ") \; '\text{link\_dbg}' \; g' \; '\text{link\_dbg}' \\
  \quad \quad \quad \quad (\forall z \rightarrow (z, "done.")) \\
  \quad \rightarrow \rightarrow (g \; (f \; x),"f \; called, \; g \; called, ") \; '\text{link\_dbg}' \\
  \quad \quad \quad \quad (\forall z \rightarrow (z, "done.")) \\
  \quad \rightarrow \rightarrow (g \; (f \; x),"f \; called, \; g \; called, \; done."))
\end{align*}
\]
Chapter 11.1.3

Example: Random Numbers
Objective

The library Data.Random provides a function

random :: StdGen -> (a,StdGen)

for computing (pseudo) random numbers.

All functions \( f :: a \rightarrow b \) can use random numbers, if they can (additionally) manage a value of type StdGen to be used by the next function to generate a random number:

\[
\begin{align*}
    f & : : a \rightarrow \text{StdGen} \rightarrow (b,\text{StdGen}) \\
    g & : : b \rightarrow \text{StdGen} \rightarrow (c,\text{StdGen})\end{align*}
\]

Note, \( f \) and \( g \) can not be sequenced using ordinary functional composition \(( . )\).
Ad hoc Sequencing

...similarly to the ‘debug’ example, we could define a new function \( h \), whose implementation realizes the sequential composition:

\[
\text{-- 'k = f link g' w/ the meaning: first f, then g}
\]
\[
k :: a \to \text{StdGen} \to (c,\text{StdGen})
\]
\[
k \ x \ \text{gen} = \text{let} \ (f\_\text{result},f\_\text{gen}) = f \ x \ \text{gen}
\]
\[
\quad \text{result} = g \ f\_\text{result} \ f\_\text{gen}
\]
\[
in \ \text{result}
\]

Again, this works but were impractical as it continuously required implementing new functions (like \( k \)) which realize the sequencing of a pair of functions (like \( f \) and \( g \)).
A new Sequencing Operator `link_rdm`

...more conveniently sequencing could be handled by introducing a function `link_rdm` for linking functions like `f` and `g`:

```haskell
link_rdm :: (StdGen -> (a,StdGen)) ->
       (a -> StdGen -> (b,StdGen)) ->
       StdGen -> (b,StdGen)

link_rdm :: f g gen = let (y,f_gen) = f gen
                       (z,g_gen) = g y f_gen
                       result = (z,g_gen)
                       in result
```

Note, `link_rdm` allow us to sequence `f` and `g` conveniently:

```
f x `link_rdm` g ( = k x )
```
Example: Sequencing with `link_rdm`

Let:

```
seed = ... :: StdGen
new_StdGen :: StdGen -> StdGen
new_StdGen gen = ...

f :: String -> StdGen -> (Int,StdGen)
f s gen = (length s, new_StdGen gen)
```

```
g :: Int -> StdGen -> (Bool,StdGen)
g n gen = (mod n 2 == 0, new_StdGen gen)
```

Then:

```
(f "Fun" 'link_rdm' g) seed
```

```
-\(\rightarrow\) let (m, f_gen) = (f "Fun") seed
  (b, g_gen) = g m f_gen
  result = (b, g_gen)
  in result
```

```
-\(\rightarrow\) let (m, f_gen) = (length "Fun", new_StdGen seed)
  (b, g_gen) = (mod m 2 == 0, new_StdGen f_gen)
  in (b, g_gen)
```

```
-\(\rightarrow\) let (m, f_gen) = (3, new_StdGen seed)
  (b, g_gen) = (mod 3 2 == 0, new_StdGen f_gen)
  in (b, g_gen)
```

```
-\(\rightarrow\) (False, new_StdGen (new_StdGen seed))
```
Chapter 11.1.4

Findings, Looking ahead
Finding

...the examples of the two case studies enjoy a

- common structure.

This common structure can be encapsulated in a new

- type constructor class.

This type class is the type (constructor) class

- Monad.
Outlook: The Type Constructor Class Monad

```haskell
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b -- link
  return :: a -> m a -- Value ‘lifting:’ Make an
           -- (m a)-value; unit wrt (>>=)
```

...defining debug information and random numbers as new types allows to make them monads, i.e., instances of Monad:

```haskell
newtype Dbg a = D (a,String)
newtype Rdm a = R (StdGen -> (a,StdGen))
```

such that:

```haskell
(>>=) :: Monad Dbg => Dbg a -> (a -> Dbg b) -> Dbg b
return :: Monad Dbg => a -> Dbg a
(>>=) :: Monad Rdm => Rdm a -> (a -> Rdm b) -> Rdm b
return :: Monad Rdm => a -> Rdm a
```
Outlook: The Instance Declarations

...for 1) the type constructor Debug:

newtype Dbg a = D (a, String)

instance Monad Dbg where
  (D (x, s)) >>= k = let D (x', s') = k (x, s)
                       in D (x', s ++ s')

  return x = D (x, "")

...for 2) the type constructor Random:

newtype Rdm a = R (StdGen -> (a, StdGen))

instance Monad Rdm where
  (R f) >>= k = R \gen -> (let (x, gen') = f gen
                             (R b) = k x
                             in b gen')

  return x = R \gen -> (x, gen)
Chapter 11.1.5

Excursus on Functional Composition
Functors, Applicatives, Monads – Intuition (1)

...note the similarity of the signature patterns:

$(\cdot) :: (a \to b) \to a \to b$
$g \cdot x = g \ x$

$fmap :: \text{(Functor } f\text{)} \Rightarrow (a \to b) \to f \ a \to f \ b$
$fmap \ g \ c = \ldots$

$(\lt>*\gt) :: \text{(Applicative } f\text{)} \Rightarrow f \ (a \to b) \to f \ a \to f \ b$
$(\lt>*\gt) \ k \ c = \ldots$

$(\gg>=) :: \text{(Monad } m\text{)} \Rightarrow m \ a \to (a \to m \ b) \to m \ b$
$(\gg>=) \ c \ k = \ldots$

$(\cdot) :: (b \to c) \to (a \to b) \to (a \to c)$
$(f \cdot g) \ x = f \ (g \ x)$
Functors, Applicatives, Monads – Intuition (2)

...in more detail with added type information:

($) :: (a -> b) -> a -> b

\( g \ $ \ x = g \ x \)

:: a -> b :: a :: b

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

fmap g \ c = \ ... \ -- w/ ... specific for f

:: a -> b :: f a :: f b

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

(<*>) \ k \ c = \ ... \ -- w/ ... specific for f

:: f (a -> b) :: f a :: f b

(>>=) :: (Monad m) => m a -> (a -> m b) -> m b

(>>=) \ c \ k = \ ... \ -- w/ ... specific for m

:: m a :: a -> m b) :: m b)

(.) :: (b -> c) -> (a -> b) -> (a -> c)

(f . g) x = f (g x)

:: a :: c
Composing Functions: (.) and (;) (1)

...by default, function composition (or sequencing) in Haskell is from “right to left,” just as in mathematics:

\[(.) :: (b -> c) -> (a -> b) -> (a -> c)\]
\[(f \cdot g) x = f (g x)\]

\[\Rightarrow \text{First } g \text{ is applied, then } f \text{ (application is “right to left!”)}\]

We complement “right to left” function composition (.) with “left to right” function composition (;):

\[(; ) :: (a -> b) -> (b -> c) -> (a -> c)\]
\[(f ; g) x = g (f x)\]

-- equivalently pointfree:
\[(f ; g) = g \cdot f\]

\[\Rightarrow \text{First } f \text{ is applied, then } g \text{ (application is “left to right!”)}\]
Composing Functions: (.) and (;) (2)

**Sequencing w/ (.)**: Functions are taken from “right to left:"

\[(fn \ldots f3 . f2 . f1 . f) \ x\]

\[\rightarrow \rightarrow (fn \ldots f3 . f2 . f1) (f \ x)\]

\[\rightarrow \rightarrow (fn \ldots f3 . f2) (f1 (f \ x))\]

\[\rightarrow \rightarrow (fn \ldots f3) (f2 (f1 (f \ x)))\]

\[\rightarrow \rightarrow \ldots\]

\[\rightarrow \rightarrow fn (\ldots (f3 (f2 (f1 x)))\ldots)\]

**Sequencing w/ (;)**: Functions are taken from “left to right:"

\[(f ; f1 ; f2 ; f3 ; \ldots ; fn) \ x\]

\[\rightarrow \rightarrow (f1; f2 ; f3 ; \ldots ; fn) (f \ x)\]

\[\rightarrow \rightarrow (f2 ; f3 ; \ldots ; fn) (f1 (f \ x))\]

\[\rightarrow \rightarrow (f3 ; \ldots ; fn) (f2 (f1 (f \ x)))\]

\[\rightarrow \rightarrow \ldots\]

\[\rightarrow \rightarrow fn (\ldots (f3 (f2 (f1 x)))\ldots)\]
Relationship of (.) and (;) (1)

If \( f, f_1, f_2, f_3, \ldots, f_n \) are functions and \( x \) a value of fitting types we have the following equalities:

\[
(((f_n \ldots . f_3) . f_2) . f_1) . f =
\]

\[
(f ; (f_1 ; (f_2 ; (f_3 ; \ldots ; f_n))))
\]

\[
(((f_n \ldots . f_3) . f_2) . f_1) . f) \ x =
\]

\[
(f ; (f_1 ; (f_2 ; (f_3 ; \ldots ; f_n)))) \ x
\]

938/192
Relationship (.) and (;) (2)

Both (.) and (;) are associative. Hence, parentheses can be dropped yielding:

\[ fn \ldots f_3 \ldots f_2 \ldots f_1 \ldots f = f \; f_1 \; f_2 \; f_3 \; \ldots \; fn \]

\[ (fn \ldots f_3 \ldots f_2 \ldots f_1 \ldots f) \; x = (f \; f_1 \; f_2 \; f_3 \; \ldots \; fn) \; x \]

Note:

- Both (.) and (;) specify explicitly the order, in which the functions are to be applied!
- This holds for monadic composition (>>=), too.
- Specifying sequencing precisely and explicitly is thus not a feature which is unique for monadic composition.
Sequencing for Monadic and Non-M. Types

In analogy to the monadic sequencing operator (\(\triangleright\triangleright=\)) for monads:

\[
\begin{align*}
    (\triangleright\triangleright=) &:: (\text{Monad } m) \Rightarrow m a \rightarrow (a \rightarrow m b) \rightarrow m b \\
    c \triangleright\triangleright= k &= \ldots :: m b \\
    (\text{dc } x) \triangleright\triangleright= k &= k \ x :: m b \\
    \quad \text{-- with dc some data constructor of type constructor } m, \\
    \quad \text{-- and with } x \text{ some value of type } a, \ i.e, \ x :: a
\end{align*}
\]

...we introduce a sequencing operator (\(\triangleright\triangleright;\)) inspired by (\(\triangleright\triangleright=\)) and (\(;\)) for non-monadic types:

\[
\begin{align*}
    (\triangleright\triangleright;)&:: a \rightarrow (a \rightarrow b) \rightarrow b \\
    x \triangleright\triangleright; f &= f \ x :: b
\end{align*}
\]
Sequencing Functions w/ (;) and (>>;)

The operators (;) and (>>;) are closely related:

\[
(f ; f_1 ; f_2 ; f_3 ; \ldots ; f_n) x =
\]
\[
x >>; f >>; f_1 >>; f_2 >>; f_3 >>; \ldots >>; f_n
\]

( ; ) : function application left to right but argument on the right.

\[
(f ; f_1 ; f_2 ; f_3 ; \ldots ; f_n) x
\]
\[
-\rightarrow (f_1 ; f_2 ; f_3 ; \ldots ; f_n) (f x)
\]
\[
-\rightarrow (f_2 ; f_3 ; \ldots ; f_n) (f_1 (f x))
\]
\[
-\rightarrow \ldots
\]
\[
-\rightarrow f_n ( \ldots ( f_3 ( f_2 ( f_1 x)))\ldots )
\]

( >>; ) : function application left to right and argument on the left!

\[
x >>; f >>; f_1 >>; f_2 >>; f_3 >>; \ldots >>; f_n
\]
\[
-\rightarrow (f x) >>; f_1 >>; f_2 >>; f_3 >>; \ldots >>; f_n
\]
\[
-\rightarrow (f_1 (f x)) >>; f_2 >>; f_3 >>; \ldots >>; f_n
\]
\[
-\rightarrow \ldots
\]
\[
-\rightarrow f_n ( \ldots ( f_3 ( f_2 ( f_1 x)))\ldots )
\]
Non-Monadic Function Sequencing: (>>; ) (1)

\[
\begin{align*}
x &\gg; \ f &\gg; \ f1 &\gg; \ f2 &\gg; \ f3 &\gg; \ f4 &\gg; \ g \\
&\gg; \ a &\gg; \ a \to b &\gg; \ b \to c &\gg; \ c \to d &\gg; \ d \to e &\gg; \ e \to g
\end{align*}
\]

\[
\begin{align*}
\text{id } x &\gg; \ f \\
&\gg; \ a &\gg; \ a \to b \\
&\gg; \ b &\gg; \ b \to c \\
&\gg; \ c &\gg; \ c \to d \\
&\gg; \ d &\gg; \ d \to e \\
&\gg; \ e &\gg; \ e \to g \\
x5 &\gg; \ g
\end{align*}
\]
Non-Monadic Function Sequencing: $(>>;)(2)$

The same but (most) types dropped and parentheses added for clarity:

$\(((((x \gg; f) \gg; f1) \gg; f2) \gg; f3) \gg; f4) :: g\)

$\vdash a \vdash a \to b \vdash b \to c \vdash c \to d \vdash d \to e \vdash e \to g$

$\(((((x \gg; f) \gg; f1) \gg; f2) \gg; f3) \gg; f4) \rightarrow (((((x1 \gg; f1) \gg; f2) \gg; f3) \gg; f4) \rightarrow (((((x2 \gg; f2) \gg; f3) \gg; f4) \rightarrow (((((x3 \gg; f3) \gg; f4) \rightarrow (((((x4 \gg; f4) \gg; x5) :: g)$
Non-Monadic Function Sequencing: $(\ggg;\ ) (3)$

The same but (most) types and parentheses dropped:

\[
x \ggg; f \ggg; f_1 \ggg; f_2 \ggg; f_3 \ggg; f_4 :: g
\]
\[
:: a :: a \rightarrow b :: b \rightarrow c :: c \rightarrow d :: d \rightarrow e :: e \rightarrow g
\]

\[
x \ggg; f \ggg; f_1 \ggg; f_2 \ggg; f_3 \ggg; f_4 \\
-\ggg \ x_1 \ggg; f_1 \ggg; f_2 \ggg; f_3 \ggg; f_4 \\
-\ggg \ x_2 \ggg; f_2 \ggg; f_3 \ggg; f_4 \\
-\ggg \ x_3 \ggg; f_3 \ggg; f_4 \\
-\ggg \ x_4 \ggg; f_4 \\
-\ggg \ x_5 :: g
\]

Note: The operators $(\ggg;\ )$ are applied from left to right and the argument is forwarded from left to right, too. This gets lost if $(\ggg;\ )$ is used as prefix operator (cf. next slide).
Non-Monadic Function Sequencing: (>>;)(4)

Infix usage of (>>;):

\[
x \quad >>; \quad f \quad >>; \quad f1 \quad >>; \quad f2 \quad >>; \quad f3 \quad >>; \quad f4 \quad :: \quad g
\]

\[
x_1
\]
\[
x_2
\]
\[
x_3
\]
\[
x_4
\]
\[
x_5
\]

...vs. prefix usage of (>>;):

\[
( >>; ) ( ( >>; ) ( ( >>; ) ( ( >>; ) x \ f ) \ f1 ) \ f2 ) \ f3 ) \ f4 \ :: \ g
\]

\[
x_1
\]
\[
x_2
\]
\[
x_3
\]
\[
x_4
\]
\[
x_5
\]
Monadic Function Sequencing: \((\ggg=)\) (1)

\[
c \ggg f \ggg f1 \ggg f2 \ggg f3 \ggg f4 : : m g
\]

\[
\begin{align*}
c & \ggg f \\
& \ggg f1 \\
& \ggg f2 \\
& \ggg f3 \\
& \ggg f4 \\
\end{align*}
\]

\[
\begin{align*}
c & : : m a \\
f & : : a \rightarrow m b \\
f1 & : : m b \\
f1 & : : b \rightarrow m c \\
f2 & : : m c \\
f2 & : : c \rightarrow m d \\
f3 & : : m d \\
f3 & : : d \rightarrow m e \\
f4 & : : m e \\
f4 & : : e \rightarrow m g \\
\end{align*}
\]

\[
\begin{align*}
c0 & \ggg f \\
c0 & : : a \\
c0 & : : m a \\
c0 & : : a \rightarrow m b \\
c1 & : : m b \\
c1 & : : b \rightarrow m c \\
c1 & : : m c \\
c1 & : : c \rightarrow m d \\
c2 & : : m d \\
c2 & : : d \rightarrow m e \\
c2 & : : m e \\
c2 & : : e \rightarrow m g \\
c3 & : : m g \\
c3 & : : m g \\
c3 & : : m g \\
c3 & : : m g \\
c4 & : : m g \\
c4 & : : m g \\
c4 & : : m g \\
c4 & : : m g \\
c5 & : : m g \\
c5 & : : m g \\
c5 & : : m g \\
c5 & : : m g \\
\end{align*}
\]
Monadic Function Sequencing:: (>>=) (2)

The same but (most) types dropped and parentheses added for clarity:

\[
\begin{align*}
\text{c} & \implies f \implies f_1 \implies f_2 \implies f_3 \implies f_4 :: m \ g \\
& :: m \ a \\
& :: a \rightarrow m \ b \\
& :: b \rightarrow m \ c \\
& :: c \rightarrow m \ d \\
& :: d \rightarrow m \ e \\
& :: e \rightarrow m \ g
\end{align*}
\]

\[
\begin{align*}
(((((c \implies f) \implies f_1) \implies f_2) \implies f_3) \implies f_4) \\
\implies (((c1 \implies f_1) \implies f_2) \implies f_3) \implies f_4) \\
\implies (((c2 \implies f_2) \implies f_3) \implies f_4) \\
\implies ((c3 \implies f_3) \implies f_4) \\
\implies (c4 \implies f_4) \\
\implies c5 :: m \ g
\end{align*}
\]
Monadic Function Sequencing:: (>>=) (3)

The same but (almost all) types and parentheses dropped:

\[
c >>= f >>= f1 >>= f2 >>= f3 >>= f4 :: m\ g
\]

\[
c >>= f >>= f1 >>= f2 >>= f3 >>= f4
\]

\[
-\gg\ gg c1 >>= f1 >>= f2 >>= f3 >>= f4
\]

\[
-\gg\ gg c2 >>= f2 >>= f3 >>= f4
\]

\[
-\gg\ gg c3 >>= f3 >>= f4
\]

\[
-\gg\ gg c4 >>= f4
\]

\[
-\gg\ gg c5 :: m\ g
\]

Note: The operators (>>=) are applied from left to right and the argument is forwarded from left to right, too. This gets lost if (>>=) is used as prefix operator (cf. next slide).
Monadic Function Sequencing: (>>=) (4)

Infix usage of (>>=):

\[
c >>= f >>= f1 >>= f2 >>= f3 >>= f4 :: m g
\]

...vs. prefix usage of (>>=):

\[
( >>=) (( >>=) (( >>=) (( >>=) c f) f1) f2) f3) f4 :: m g
\]
Monadic Function Sequencing via do-Not. (1)

\[
v \triangleright\triangleright= f \triangleright\triangleright= f1 \triangleright\triangleright= f2 \triangleright\triangleright= f3 \triangleright\triangleright= f4 :: m \ g
\]

\[
\begin{align*}
\text{:: m a} & \quad \text{:: a -> m b} & \quad \text{:: b -> m c} & \quad \text{:: c -> m d} & \quad \text{:: d -> m e} & \quad \text{:: e -> m g}
\end{align*}
\]

do \ v0' \leftarrow \text{return v0} \quad \text{-- Note: return v0 \triangleright\triangleright v}

\[
\begin{align*}
\text{:: a} & \\
\text{:: m a} & \\
\text{:: a} & \\
\text{:: a} & \quad \text{:: m a} \\
v1 \leftarrow f \ v0' & \\
\text{:: b} & \quad \text{:: m b} \\
v2 \leftarrow f1 \ v1 & \\
\text{:: c} & \quad \text{:: m c} \\
v3 \leftarrow f2 \ v2 & \\
\text{:: d} & \quad \text{:: m d} \\
v4 \leftarrow f3 \ v3 & \\
\text{:: e} & \quad \text{:: m e} \\
v5 \leftarrow f4 \ v4 & \\
\text{:: g} & \quad \text{:: m g}
\end{align*}
\]

\text{return v5}

\[
\begin{align*}
\text{:: m g}
\end{align*}
\]
Monadic Function Sequencing via do-Not. (2)

The expression

\[ (((((v >>= f) >>= f1) >>= f2) >>= f3) >>= f4) :: m g \]

\[ :: m a :: a -> m b :: b -> m c :: c -> m d :: d -> m e :: e -> m g \]

...in standard notation using (>>=) and parentheses for order specification can equivalently be written using the syntactic sugar of the do-notation

\[
\text{do } v0' :: a <- \text{return } v0 :: m a \quad \text{-- Note: }
\]

\[
v1 :: b <- f \ v0' :: m b \quad \text{-- return } v0 \rightarrow>> v
\]

\[
v2 :: c <- f1 \ v1 :: m c
\]

\[
v3 :: d <- f2 \ v2 :: m d
\]

\[
v4 :: e <- f3 \ v3 :: m e
\]

\[
v5 :: g <- f4 \ v4 :: m g
\]

\[
\text{return } v5 :: m g
\]

...with an implicit ordering specification by data dependencies.
Monadic Function Sequencing via do-Not. (3)

The same but (most) types dropped...

The expression

\[
(((v >>= f) >>= f1) >>= f2) >>= f3) >>= f4 \quad :: \quad m \, g
\]

\[
\quad :: \quad m \, a \quad :: \quad a \rightarrow m \, b \quad :: \quad b \rightarrow m \, c \quad :: \quad c \rightarrow m \, d \quad :: \quad d \rightarrow m \, e \quad :: \quad e \rightarrow m \, g
\]

...is equivalent to the do-expression:

```
do v0' <- return v0       -- Note: return v0 ->> v
    v1 <- f   v0'
    v2 <- f1  v1
    v3 <- f2  v2
    v4 <- f3  v3
    v5 <- f4  v4
return  v5
```
Compare: Monadic vs. Non-M. Operations (1)

A non-monadic application example:

"Functional Programming" >>; length >>; odd >>; f

where

\[
\begin{align*}
f &:: \text{Bool} \to \text{Char} \\
f \text{ True} & = 'H' \quad -- \text{reminding to High} \\
f \text{ False} & = 'L' \quad -- \text{reminding to Low}
\end{align*}
\]

...stepwise evaluated:

"Functional Programming" >>; length >>; odd >>; f
Compare: Monadic vs. Non-M. Operations (2)

...and its monadic counterpart:

\[
\text{Id } "\text{Functional Programming}\" \triangleright\triangleright= \text{length}_m \triangleright\triangleright= \text{odd}_m \triangleright\triangleright= \text{f}_m
\]

where

\[
\begin{align*}
\text{length}_m & : \text{String} \to \text{Id} \ \text{Int} \\
\text{length}_m \ s &= \text{Id} (\text{length} \ s) \\
\text{odd}_m & : \text{Int} \to \text{Id} \ \text{Bool} \\
\text{odd}_m \ n &= \text{Id} (\text{odd} \ n) \\
\text{f}_m & : \text{Bool} \to \text{Id} \ \text{Char} \\
\text{f}_m \ b &= \text{Id} (\text{f} \ b)
\end{align*}
\]

...stepwise evaluated:

\[
\text{Id } "\text{Functional Programming}\" \triangleright\triangleright= \text{length}_m \triangleright\triangleright= \text{odd}_m \triangleright\triangleright= \text{f}_m
\]

\[
\text{Id} \ 'L' \triangleright\triangleright= \text{Id} \ 'L' \triangleright\triangleright= \text{Id} \ 'L' \triangleright\triangleright= \text{Id} \ 'L'
\]
### Compare: Monadic vs. Non-M. Operations (3)

**Monadic operations**

\[
(\gg\gg) :: (\text{Monad } m) \Rightarrow m a \rightarrow (a \rightarrow m b) \rightarrow m b \\
(c \gg\gg k) = k x :: m b
\]

\[
(\text{dc } x) \quad -- w/ \text{ dc } a \text{ data constructor of type constructor } m, \text{ and } w/ \text{ x a value of type } a, \text{ i.e., } x :: a
\]

\[
\text{return} :: (\text{Monad } m) \Rightarrow a \rightarrow m a \\
\text{return } v = m v :: m a
\]

\[
\text{fail} :: (\text{Monad } m) \Rightarrow \text{String} \rightarrow m a \\
\text{fail } s = \text{error } s :: m a
\]

\[
(\gg) :: (\text{Monad } m) \Rightarrow m a \rightarrow m b \rightarrow m b \\
c \gg k = c \gg \_ \rightarrow k :: m b
\]

...and their non-monadic counterparts:

\[
(\gg;) :: a \rightarrow (a \rightarrow b) \rightarrow b \\
x \gg; f = f x :: b
\]

\[
\text{id} :: a \rightarrow a \\
\text{id } x \rightarrow x :: a
\]

\[
\text{fail} :: \text{String} \rightarrow a \\
\text{fail } s = \text{error } s :: a
\]

\[
(\;>) :: a \rightarrow b \rightarrow b \\
x >; y = x \gg; \_ \rightarrow y :: b \quad -- \text{i.e.: } x >; y = y :: b
\]
Why Introducing Monads at All? (1)

...generality, flexibility, and re-use!

Note, just staying with

\((\gg;\,) : : a \to (a \to b) \to b\)

\(v \gg; f = f \ v\)

means to stay

- with only one implementation of \((\gg;\,)\) for all types \(a\) and \(b\)
- which must be used and work for all types \(a\) and \(b\)
- which thus can not be particularly “type specific” since nothing can be assumed about \(a\) and \(b\) by the implementation of \((\gg;\,)\)
Why Introducing Monads at All? (2)

Note, (>>; ) does not allow to cope with the debug-example.

\[
\begin{align*}
f : : \text{String} & \rightarrow \text{Int} & \quad g : : \text{Int} & \rightarrow \text{Bool} \\
f &= \text{length} & \quad g &= \text{odd} \\
(g \cdot f) &= f \; ; \; g & \quad -- \text{composition of } f \text{ and } g \text{ works!} \\
(g \cdot f) \; s &= (f \; ; \; g) \; s = g(f(s)) & \quad -- \text{works for all values} \\
\quad &= s \; >>; \; f \; >>; \; g & \quad -- s \text{ of type String!}
\end{align*}
\]

While composition works fine for \( f \) & \( g \), it does not for \( f' \) & \( g' \):

\[
\begin{align*}
f' : : \text{String} & \rightarrow (\text{Int, String}) & \quad g' : : \text{Int} & \rightarrow (\text{Bool, String}) \\
f' \; s &= (f \; s, "f \text{ called, } \), & \quad g' \; n &= (g \; n, "g \text{ called, } \) \\
(g' \cdot f') &= f' \; ; \; g' & \quad -- \text{does not work: types of } g' \\
\quad & \quad -- \text{and } f' \text{ do not fit!} \\
(g' \cdot f') \; s &= (f' \; ; \; g') \; s = g'(f'(s)) & \quad -- \text{does not work:} \\
\quad &= s \; >>; \; f' \; >>; \; g' & \quad -- \text{type-specific implementations of (>>; ), (>>; )} \\
\quad & \quad -- \text{are required!}
\end{align*}
\]
Why Introducing Monads at All? (3)

Introduce a new data type `Debug a`:

```haskell
newtype Debug a = D (a,String)
```

Make the constructor `Debug` an instance of class `Monad`:

```haskell
instance Monad Debug where
    (D (v,s)) >>= f = let D (v',s') =
        f (v,s) in D (v',s++s')
    return x = D (x,"")
```

Note that `Debug Int` and `Debug Bool` are both instances of type `Debug a`. This allows us to switch from `f', g'` to `f_m, g_m`:

```haskell
f_m :: String -> Debug Int  g_m :: Int -> Debug Bool
f_m s = D (f s,"f called. ")  g_m n = D (g n,"g called. ")
```

```haskell
D (s,t) >>= f_m >>= g_m -- works for all values s, t of type String!
```

Hence, we got the desired type-awareness of `(>>=)` with just one instance declaration!
Why Introducing Monads at All? (4)

In fact, introducing the type constructor class `Monad`

```haskell
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
```

allows as many implementations of `(>>=)` for a type as needed. It only requires to hide the type behind a distinct new type constructor to allow another implementation of `(>>=)` for it:

```haskell
data Id a = ...; instance Monad Id where...
data [] a = ...; instance Monad [] where...
data Maybe a = ...; instance Monad Maybe where...
data Tree a = ...; instance Monad Tree where...
data IO a = ...; instance Monad IO where...
...
data Id' a = ...; instance Monad Id' where...
data Maybe' a = ...; instance Monad Maybe' where...
...
```
Why Introducing Monads at All? (5)

...where (the values of) the data types

```
data Id' a    = Id' a
data Maybe' a = Nothing' | Just' a
data List' a  = Empty' | Cons' a (List' a)
```

... equal their “unprimed” counterparts but allow us to implement a different behaviour for (>>=) and the other monadic operations.
Why Introducing Monads at All? (6)

All in all, this also allows to interleave applications of (\texttt{>>>=}) and (\texttt{>>>;}) and to change the monad in the course of the computation, e.g., from \texttt{Id} to \texttt{Id'}:

\[
\begin{align*}
\text{id2id'} &: \text{Id } a \rightarrow \text{Id'} a \quad \text{id'2id} &: \text{Id'} a \rightarrow \text{Id } a \\
\text{id2id'} (\text{Id } v) &= \text{Id'} v \quad \text{id'2id} (\text{Id'} v) &= \text{Id } v
\end{align*}
\]

\[
\text{s} = \text{Id } "\text{Fun}" :\text{Id String}
\]

\[
\text{f, g} :\text{String } \rightarrow \text{Id String}
\text{f', g'} :\text{String } \rightarrow \text{Id'} \text{ String}
\]

\[
\begin{align*}
\text{monad change: } &\text{Id2Id'} \quad &\text{monad change: } &\text{Id'}2\text{Id} \\
\text{s } \text{>>> } \text{f } \text{>>>; } \text{id2id'} \text{ } \text{>>> } \text{f'} \text{ } \text{>>> } \text{g'} \text{ } \text{>>>; } \text{id'2id} \text{ } \text{>>> } \text{g}
\end{align*}
\]
Last but not least (1)

If we had been prepared to change both domain and range of functions (instead of their range only), ordinary composition would have been sufficient for the debug-example:

While

\[ f' :: \text{String} \to (\text{Int, String}) \quad g' :: \text{Int} \to (\text{Bool, String}) \]

\[ f' s = (f s,"f called, ") \quad g' n = (g n,"g called, ") \]

\[(g' . f') = f' ; g' -- does not work: types of g' and f'
\]

\[ -- do not fit! \]

\[(g' . f') s = (f' ; g') s = g'(f'(s)) -- does not work: \]

\[= s >>; f' >>; g' -- type-specific implementations are required! \]

does not work, the following does work:

\[ f'' :: = (\text{String,String}) \to (\text{Int, String}) \]

\[ f'' (s,t) = (f s , t ++ "f called, ") \]

\[ g'' :: (\text{Int, String}) \to (\text{Bool, String}) \]

\[ g'' (n,t) = (g n , t ++ "g called, ") \]

\[(g'' . f'') s = (f'' ; g'') s = g''(f''(s)) = (s,"") >>; f'' >>; g'' \]
Last but not least (2)

Compare the monadic-free implementation of the debug-example:

```haskell
f'' :: (String,String) -> (Int,String)  -- Note: Concatenation of
f'' (s,t) = (f s , t ++ "f called, ")  -- Strings handled by

g'' :: (Int,String) -> (Bool,String)  -- f'' and g'', not by (>>;)
g'' (n,t) = (g n , t ++ "g called, ")

("Fun",""") >>; f'' >>; g''
->>> (3,"f called, ") >>; g'' ->>> (True,"f called, g called, ")
```

...with its monadic counterpart:

```haskell
newtype Debug a = D (a,String)
instance Monad Debug where
  (D (v,s)) >>= f = let D (v',s') =
                 f (v,s) in D (v',s ++ s')  -- by (>>=), not
  return x = D (x,""")  -- by fm and gm

fm :: String -> Debug Int  -- Note: Concatenation
  fm s = D (f s,"f called, ")

gm :: Int -> Debug Bool
  gm n = D (g n,"g called, ")

D (s,""") >>= fm >>= gm
->>> D (3,"f called, ") >>; gm ->>> D (True,"f called, g called, ")
```

Quite similar, aren’t they?
Chapter 11.2
The Type Constructor Class Monad
The Type Constructor Class Monad

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

...monads are instances of the type constructor class Monad (hence 1-ary type constructors), which obey the monad laws:

Monad Laws

   return x >>= f = f x (ML1)
c >>= return = c (ML2)
c >>= (\x -> (f x) >>= g) = (c >>= f) >>= g (ML3)
Type Constructor Class Monad in more Detail

class Monad m where

   -- ‘Primary’ functions (relevant for every monad)
   return :: a -> m a
       -- Value ‘lifting:’ Making 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:
                :: String
                :: String
       -- Outputting s as an error message
       :: m a
       :: m a
   c >> k    = c >>= \_ -> k
       :: m a
       :: m b
       :: m b
       :: m b
       :: a -> m b


The Monad Laws in more Detail

...with added type information:

\[
\text{return } x \gg= f = f x \quad (\text{ML1})
\]

\[
\begin{align*}
& \text{return } x \gg= f = f x \\
& \text{:: } a \rightarrow m a \quad \text{:: } a \rightarrow m b \\
& \quad \text{:: } m a \\
& \quad \text{:: } m b
\end{align*}
\]

\[
\begin{align*}
& c \gg= \text{return } = c \\
& \text{:: } m a \\
& \quad \text{:: } a \rightarrow m a \\
& \quad \text{:: } m a
\end{align*}
\]

\[
\text{(ML2)}
\]
Homework

Provide (most general) type information for the monad law ML3:

\[ c >>= (\lambda x \rightarrow (f \ x) >>= g) = (c >>= f) >>= g \quad (\text{ML3}) \]
Note

...the monad laws require from (proper) monad instances:

▶ `return` passes its argument without any other effect, i.e., `return` is unit of `(>>=)` (see also function `pure` of class `Applicative`) (ML1, ML2).

▶ `(>>=)` is associative, i.e., sequencings given by `(>>=)` must not depend on how they are bracketed (ML3).

Proof obligation:

▶ It is a programmer obligation to prove that their instances of `Monad` satisfy the monad laws.

Note: Sequence operator `(>>=)`: Read as `bind` (Paul Hudak) or `then` (Simon Thompson). Sequence operator `(>>)`: Derived from `(>>=)`, read as `sequence` (Paul Hudak).
Associativity of \((\gg\gg)\)

**Lemma 11.2.1 (Associativity of \((\gg\gg)\))**

If \((\gg\gg=)\) of some monad \(m\) is associative, then also the default implementation of \((\gg\gg)\) is associative, i.e.:

\[
c_1 \gg (c_2 \gg c_3) = (c_1 \gg c_2) \gg c_3
\]
The Operator (>@>)

Note, the formulation of associativity for (>>>):  
\[ c >>> (\lambda x \to (f \ x) >>> g) = (c >>> f) >>> g \]

...is less appealing than the one for (>>):  
\[ c1 >> (c2 >> c3) = (c1 >> c2) >> c3 \]

The operator (>@>) derived from (>>>=) and defined by:
\[ (>@>) :: Monad m => (a -> m b) -> (b -> m c) \to (a -> m c) \]
\[ f >@> g = \lambda x \to (f \ x) >>> g \]

...improves on this: For (>@>), the monad laws, especially the associativity requirement, become as natural and obvious as for (>>).
The Monad Laws in Terms of (>@>)

Lemma 11.2.2
If (>>=) and return of some monad m are associative and unit of (>>=), respectively, then we have:

\[
\begin{align*}
\text{return} >@> f &= f \quad (\text{ML1'}) \\
f >@> \text{return} &= f \quad (\text{ML2'}) \\
(f >@> g) >@> h &= f >@> (g >@> h) \quad (\text{ML3'})
\end{align*}
\]

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:

\[
\text{fmap } g \ x s \ = \ x s \gg= \ \text{return} \ . \ g \\
( = \ do \ x \leftarrow \ x s ; \ \text{return} \ (g \ x) )
\]
Selected Utility Functions for Monads (1)

\[(=<<)=: \text{Monad } m \Rightarrow (a \rightarrow m b) \rightarrow m a \rightarrow m b\]

\[f =<< x = x >>= f\]

\[\text{sequence} :: \text{Monad } m \Rightarrow [m a] \rightarrow m [a]\]
\[\text{sequence} = \text{foldr mcons (return [])}\]
\[\quad \text{where mcons p q = do } l \leftarrow p\]
\[\quad \quad ls \leftarrow q\]
\[\quad \quad \text{return } (l:ls)\]

\[\text{sequence}_\_ :: \text{Monad } m \Rightarrow [m a] \rightarrow m ()\]
\[\text{sequence}_\_ = \text{foldr } (>>>) (\text{return } ())\]

\[\text{mapM} :: \text{Monad } m \Rightarrow (a \rightarrow m b) \rightarrow [a] \rightarrow m [b]\]
\[\text{mapM } f \text{ as } = \text{sequence } (\text{map } f \text{ as})\]

\[\text{mapM}_\_ :: \text{Monad } m \Rightarrow (a \rightarrow m b) \rightarrow [a] \rightarrow m ()\]
\[\text{mapM}_\_ f \text{ as } = \text{sequence}_\_ (\text{map } f \text{ as})\]
Selected Utility Functions for Monads (2)

\[
\begin{align*}
\text{mapF} & \quad : \text{Monad } m \Rightarrow (a \rightarrow b) \rightarrow m \; [a] \rightarrow m \; [b] \\
\text{mapF } f \; x & \quad = \quad \text{do } v \leftarrow x; \; \text{return} \; (f \; v) \\
& \quad \text{-- equals map on lists, i.e., for picking } [] \text{ as } m \\
\text{joinM} & \quad : \text{Monad } m \Rightarrow m \; (m \; a) \rightarrow m \; a \\
\text{joinM } x & \quad = \quad \text{do } v \leftarrow x; \; v \\
& \quad \text{-- equals concat on lists, i.e., for picking } [] \text{ as } m 
\end{align*}
\]

...and many more (see e.g., library Monad).

Lemma 11.2.3

1. \( \text{mapF } (f \; . \; g) = \text{mapF } \; . \; \text{mapF } g \)
2. \( \text{joinM return } = \text{joinM } \; . \; \text{mapF } \; \text{return} \)
3. \( \text{joinM return } = \text{id} \)
Homework

1. Prove Lemma 11.2.3.

2. Do the functor and monad laws imply law FML? Provide a proof or a counter-example.

3. Provide (most general) type information for
   3.1 the defining equation of $(>@>)$:
      \[
      (>@>) :: \text{Monad } m \Rightarrow (a \rightarrow m b) \rightarrow (b \rightarrow m c) \\
      \rightarrow (a \rightarrow m c)
      \]
      \[
      f >@> g = \lambda x \rightarrow (f x) \gg= g
      \]
   3.2 the statement of Lemma 11.2.1:
      \[
      c1 \gg (c2 \gg c3) = (c1 \gg c2) \gg c3
      \]
   3.3 the statements of Lemma 11.2.2:
      \[
      \text{return } >@> f = f \quad (\text{ML1'}\\n      f >@> \text{return } = f \quad (\text{ML2'}\\n      (f >@> g) >@> h = f >@> (g >@> h) \quad (\text{ML3'})
      \]
Chapter 11.3
Syntactic Sugar: The do-Notation
The do-Notation

...the monadic operations \((\ggg\ggg)\) and \((\ggg)\) allow very much as functional composition \((.)\)

▶ to specify the sequencing of (fitting) operations explicitly.

Both functional and monadic sequencing introduce

▶ an imperative flavour into functional programming.

Using the so-called

▶ do-notation

as syntactic sugar expresses this flavour for monadic sequencing in a syntactically more appealing and concise fashion.
Relating Monadic Operations and do-Notation

...four conversion rules allow the conversion of sequences of monadic operations composed of

- $(>>=)$ and $(>>)$

into equivalent ('<=>') sequences of

- do-blocks

and vice versa.
Intuitively

Recall:

\[(\ggg) :: m a \rightarrow m b \rightarrow m b\]

Then:

\[
dc \ v \ggg f \rightarrow f \ v
\]

\[
:: m a \quad :: (a \rightarrow m b) \quad :: m b
\]

"\(\Leftarrow\) do x \leftarrow dc \ v; y \leftarrow f \ x; return \ y" 

\[
:: a \quad :: m a \quad :: b \quad :: m b \quad :: m b
\]

\[
dc \ v \gg dc' \ v' \rightarrow dc \ v \ggg _ \rightarrow dc' \ v'
\]

\[
:: m a \quad :: m b \quad :: m a \quad :: (a \rightarrow m b)
\]

"\(\Leftarrow\) do _ \leftarrow dc \ v; y \leftarrow dc' \ v'; return \ y" 

\[
:: a \quad :: m a \quad :: b \quad :: m b \quad :: m b
\]

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 that are 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 `(>>)`:

\[
\text{return } a \triangleright= f = f \ a \quad \text{(ML1)} \\
c \triangleright= \text{return } = c \quad \text{(ML2)} \\
c \triangleright= (\lambda x \rightarrow (f \ x) \triangleright= g) = (c \triangleright= f) \triangleright= g \quad \text{(ML3)}
\]

**B)** The **Monad laws** using **do-notation**:

\[
\text{do } x \leftarrow \text{return } a; f \ x = f \ a \quad \text{(ML1)} \\
\text{do } x \leftarrow c; \text{return } x = c \quad \text{(ML2)} \\
\text{do } x \leftarrow c; y \leftarrow f \ x; g \ y = \\
\quad \quad \text{do } y \leftarrow (\text{do } x \leftarrow c; f \ x); g \ y \quad \text{(ML3)}
\]
Semicolons vs. Linebreaks in do-Notation

B) do-notation in ‘one’ line (w/ ‘;’, no linebreaks):

\[
\begin{align*}
\text{do } x & \leftarrow \text{return } a; \ f \ x \quad = \ f \ a \quad \text{(ML1)} \\
\text{do } x & \leftarrow c; \ \text{return } x \quad = \ c \quad \text{(ML2)} \\
\text{do } x & \leftarrow c; \ y \leftarrow f \ x; \ g \ y \ = \\
& \quad \quad \text{do } y \leftarrow (\text{do } x \leftarrow c; \ f \ x); \ g \ y \quad \text{(ML3)}
\end{align*}
\]

C) do-notation in ‘several’ lines (w/ linebreaks, no ‘;’):

\[
\begin{align*}
\text{do } x & \leftarrow \text{return } a \\
& \quad f \ x \quad = \ f \ a \quad \text{(ML1)} \\
\text{do } x & \leftarrow c \\
& \quad \text{return } x \quad = \ c \quad \text{(ML2)} \\
\text{do } x & \leftarrow c \\
& \quad y \leftarrow f \ x \\
& \quad g \ y \quad = \quad \text{do } y \leftarrow (\text{do } x \leftarrow c \\
& \quad \quad f \ x) \\
& \quad g \ y \quad \text{(ML3)}
\end{align*}
\]
Chapter 11.4
Predefined Monads
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:

▶ \((\ggg\ggg)\) 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.

▶ \((\ggg)\) and `fail` are usually not to be implemented afresh. Usually, their default implementations provided in `Monad` are just fine.

If the default implementations of \((\ggg)\) and `fail` are used, this means for

▶ \((\ggg)\): the first argument is evaluated and dropped, the second argument is evaluated and returned as result.

▶ `fail`: the computation stops by calling `error` with some appropriate error message.
Chapter 11.4.1
The Identity Monad
The Identity Monad

...the identity 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., \( \text{Id} \ a \) denotes a type.
- **Id**: 1-ary data (or value) constructor, i.e., \( \text{Id} \ v, v :: a \), denotes a value: \( \text{Id} \ v :: \text{Id} \ a \).
- **(>>=)** and **fail** are implicitly defined by their default implementations.
- **(>>=)** : : \( \text{Id} \ a \) \( \rightarrow \) (\( a \rightarrow \text{Id} \ b \) \( \rightarrow \) \( \text{Id} \ b \)
- **return** : : \( a \rightarrow \text{Id} \ a \)
- **(>>)** : : \( \text{Id} \ a \) \( \rightarrow \) \( \text{Id} \ b \) \( \rightarrow \) \( \text{Id} \ b \)
Notes on the Identity Monad (1)

The monad operations recalled:

\[
(\gg\gg) \quad :: \quad (\text{Monad } m) \Rightarrow m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b \\
v \gg\gg k = \ldots \quad :: \quad m\ b \\
\text{return} \quad :: \quad (\text{Monad } m) \Rightarrow a \rightarrow m\ a \\
\text{return} v = \ldots \quad :: \quad m\ a
\]

The instance declaration for \textit{Id} with added type information:

\[
\text{instance Monad Id where} \\
\text{Id } x \quad \gg\gg \quad f = \quad f\ x \quad -- \quad \text{yields an (Id } b)\text{-value} \\
:: \quad \text{Id } a \quad :: \quad a \rightarrow \text{Id } b \quad :: \quad \text{Id } b \\
\text{return } x = \quad \text{Id } x \quad -- \quad \text{yields an (Id } a)\text{-value} \\
:: \quad a \quad :: \quad \text{Id } a
\]

Recall the overloading of \textit{Id} (\textit{newtype Id } a = Id a):

\begin{itemize}
  \item \textit{Id} followed by \textit{x}: \textit{Id} is data (or value) constructor.
  \item \textit{Id} followed by \textit{a} or \textit{b}: \textit{Id} is type constructor.
\end{itemize}
Notes on the Identity Monad (2)

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 11.5.1)
- The operation \( (>\circ>) \) becomes for the identity monad forward composition of functions \( (>\cdot>) \) (\( = (>\circ>;) \)): \( (>\cdot>) :: (a \to b) \to (b \to c) \to (a \to c) \)
  \[ g >\cdot> f = f \cdot g \]
- Forward composition of functions \( (>\cdot>) \) is associative with unit element \( id \).

Lemma 11.4.1.1 (Monad Laws)

Instance \( Id \) of \( Monad \) satisfies the three monad laws \( ML1 \), \( ML2 \), and \( ML3 \).
Chapter 11.4.2

The List Monad
The List Monad

...the list monad:

instance Monad [] where
    xs >>= f = concat (map f xs)
    return x = [x]
    fail s   = []

Note:

- `concat` and `map` are from the Standard Prelude.
- `[]`: 1-ary type constructor, i.e., `[a]` denotes a type.
- `[]`: 1-ary data (or value) constructor, i.e., `[x], x :: a`, denotes a value: `[x] :: [a]; in particular, `[]` denotes a value, the empty list.
- `(>>>)` is implicitly defined by its default implementation; the default implementation of `fail` is overwritten.
- `(>>>=) :: [] a -> (a -> [] b) -> [] b`
  - `return :: a -> [] a`
  - `(>>>) :: [] a -> [] b -> [] b`
Monad Laws for []

Lemma 11.4.2.1 (Monad Laws)

Instance [] of Monad satisfies the three monad laws ML1, ML2, and ML3.

For convenience, we recall from the Standard Prelude:

\[
\begin{align*}
\text{concat} & \quad : \quad [\,[a]\,] \quad \rightarrow \quad [a] \\
\text{concat lss} & \quad = \quad \text{foldr} \ (++) \ ([] \ lss) \\
\text{concat} \ [[1,2,3],[4],[5,6]] & \quad \rightarrow\rightarrow \quad [1,2,3,4,5,6]
\end{align*}
\]
Notes on the List Monad

The monad operations recalled:

\[(>>=) :: (\text{Monad } m) \Rightarrow m \text{ a} \rightarrow (\text{a} \rightarrow m \text{ b}) \rightarrow m \text{ b}\]

\[v >>= k = \ldots :: m \text{ b}\]

\[\text{return} :: (\text{Monad } m) \Rightarrow \text{ a} \rightarrow m \text{ a}\]

\[\text{return } v = \ldots :: m \text{ a}\]

\[\text{fail} :: (\text{Monad } m) \Rightarrow \text{ String} \rightarrow m \text{ a}\]

\[\text{fail } s = \ldots :: m \text{ a}\]

The instance declaration for \([\ ]\) with added type information:

\[
\text{instance Monad } []\text{ where}
\]

\[xs >>= f = \text{concat (map } f \text{ xs)} -- \text{yields a [b]-list}\]

\[:: [] \text{ a} :: \text{a} \rightarrow [] \text{ b}\]

\[:: [] \text{ ([]} \text{ b})\]

\[:: [] \text{ b}\]

\[\text{return } x = [x] -- \text{yields the singleton list } [x]\]

\[:: \text{a}\]

\[:: [] \text{ a}\]

\[\text{fail } s = [] -- \text{yields the empty list } []\]

\[:: \text{String}\]

\[:: [] \text{ a}\]
Using the Type Constructor \([\ ]\) as a Monad (1)

Examples:

\[
ls = [1,2,3] :: [] \text{ Int}
\]
\[
f = \lambda n \rightarrow [(n,\text{odd}(n))] :: \text{Int} \rightarrow [] (\text{Int},\text{Bool})
\]
\[
g = \lambda n \rightarrow [x\times n \mid x \leftarrow [1.5,2.5,3.5]] :: \text{Int} \rightarrow [] \text{Float}
\]
\[
h = \lambda n \rightarrow [1..n] :: \text{Int} \rightarrow [] \text{Int}
\]

\[
h \ 3 \ >> \ f
\]
\[
\quad \rightarrow \ ls \ >> \ f
\]
\[
\quad \rightarrow \ \text{concat} \ [ \ [(1,True)], \ [(2,False)], \ [(3,True)] \ ]
\]
\[
\quad \rightarrow \ [(1,True),(2,False),(3,True)] :: [] (\text{Int},\text{Bool})
\]

\[
h \ 3 \ >> \ g
\]
\[
\quad \rightarrow \ ls \ >> \ g
\]
\[
\quad \rightarrow \ \text{concat} \ [ \ [x\times n \mid x \leftarrow [1.5,2.5,3.5]] \mid n \leftarrow [1,2,3] \ ]
\]
\[
\quad \rightarrow \ \text{concat} \ [ \ [1.5*1,2.5*1,3.5*1], \ [1.5*2,2.5*2,3.5*2], \ [1.5*3,2.5*3,3.5*3] \ ]
\]
\[
\quad \rightarrow \ \text{concat} \ [ \ [1.5,2.5,3.5], \ [3.0,5.0,7.0], \ [4.5,7.5,10.5] \ ]
\]
\[
\quad \rightarrow \ [1.5,2.5,3.5,3.0,5.0,7.0,4.5,7.5,10.5] :: [] \text{Float}
\]
Using the Type Constructor \([\ ]\) as a Monad (2)

The monad operations recalled:

\[
(\gg\gg) :: (\text{Monad } m) \Rightarrow m a \to (a \to m b) \to m b
\]

\[
v \gg\gg k = \ldots :: m b
\]

\[
\text{return} :: (\text{Monad } m) \Rightarrow a \to m a
\]

\[
\text{return } v = \ldots :: m a
\]

\[
\text{fail} :: (\text{Monad } m) \Rightarrow \text{String} \to m a
\]

\[
\text{fail } s = \ldots :: m a
\]

The instance declaration for \([\ ]\) with added type information:

\[
\text{instance Monad } [\ ] \text{ where}
\]

\[
xs \gg\gg f = \text{concat } (\text{map } f xs) \quad -- \text{yields a } [b]-\text{list}
\]

\[
:: [] a :: a \to [] b
\]

\[
\:: [] ([] b)
\]

\[
:: [] b
\]

\[
\text{return } x = [x] \quad -- \text{yields the singleton list } [x]
\]

\[
:: a
\]

\[
:: [] a
\]

\[
\text{fail } s = [] \quad -- \text{yields the empty list } []
\]

\[
:: \text{String}
\]

\[
:: [] a
\]

Examples:

\[
\text{ls} = [1,2,3] :: [] \text{Int}
\]

\[
f = \lambda n \to [(n, \text{odd}(n))] :: \text{Int} \to [] (\text{Int}, \text{Bool})
\]

\[
g = \lambda n \to [x*n \mid x \leftarrow [1.5,2.5,3.5]] :: \text{Int} \to [] \text{Float}
\]

\[
h = \lambda n \to [1..n] :: \text{Int} \to [] \text{Int}
\]

\[
h \ 3 \gg\gg f \rightarrow> \text{ls} \gg\gg f \rightarrow> \text{concat} \ [\ [(1,\text{True})], \ [(2,\text{False})], \ [(3,\text{True})] \ ]
\rightarrow> [(1,\text{True}),(2,\text{False}),(3,\text{True})] :: [] (\text{Int},\text{Bool})
\]

\[
h \ 3 \gg\gg g \rightarrow> \text{ls} \gg\gg g \rightarrow> \text{concat} \ [\ x*n \mid x \leftarrow [1.5,2.5,3.5] \ ] \mid n \leftarrow [1,2,3] \]
\rightarrow> \text{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] \ ]
\rightarrow> \text{concat} \ [\ [1.5,2.5,3.5], [3.0,5.0,7.0], [4.5,7.5,10.5] \ ]
\rightarrow> [1.5,2.5,3.5,3.0,5.0,7.0,4.5,7.5,10.5] :: [] \text{Float}
\]
The List Monad Reconsidered

...the list monad can equivalently be defined by:

```haskell
instance Monad [] where
  (x:xs) >>= f = f x ++ (xs >>= f)
  [] >>= f = []
  return x = [x]
  fail s = []
```

Note: For the list monad the operations (>>=) and return have the types:

- ` >>= :: [a] -> (a -> [b]) -> [b]`
- `return :: a -> [a]`
List Monad and List Comprehension

...the list monad and list comprehension are closely related:

\[
\text{do } x \leftarrow [1,2,3] \\
y \leftarrow [4,5,6] \\
\text{return } (x,y) \\
\rightarrow> [(1,4),(1,5),(1,6), \\
(2,4),(2,5),(2,6), \\
(3,4),(3,5),(3,6)]
\]

In fact, the following expressions are equivalent:

Proposition 11.4.2.2

\[
[(x,y) | x \leftarrow [1,2,3], y \leftarrow [4,5,6]] \leftrightarrow \\
\text{do } x \leftarrow [1,2,3] \\
y \leftarrow [4,5,6] \\
\text{return } (x,y)
\]

...list comprehension is syntactic sugar for monadic syntax!
List comprehension

...as syntactic sugar for monadic syntax.

We have:

**Lemma 11.4.2.3**

\[ f \ x \mid x \gets xs \] \<=>\ \texttt{do} \ x \gets xs; \texttt{return} (f x)

**Lemma 11.4.2.4**

\[ a \mid a \gets as, p a \] \<=>\ 
\texttt{do} \ a \gets as; \texttt{if} (p a) \texttt{then} \texttt{return} a \texttt{else} \texttt{fail} ""
Homework

Prove by stepwise evaluation the equivalences stated in:

1. Proposition 11.4.2.2
2. Lemma 11.4.2.3
3. Lemma 11.4.2.4
Chapter 11.4.3
The Maybe Monad
The Maybe Monad

...the Maybe 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.
Monad Laws for Maybe

Lemma 11.4.3.1 (Monad Laws)

Instance **Maybe** of **Monad** satisfies the three monad laws **ML1**, **ML2**, and **ML3**.

Recall that **Maybe** is also a predefined instance of **Functor**:

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

Lemma 11.4.3.2 (Monad/Functor Laws)

Instance **Maybe** of **Monad** and **Functor** satisfies law **MFL** (of Chap. 11.2).
Notes on the Maybe Monad

The monad operations recalled:

\[
(\gg\gg) :: (\text{Monad } m) \Rightarrow m a \rightarrow (a \rightarrow m b) \rightarrow m b
\]

\[
v \gg\gg k = \ldots :: m b
\]

\[
\text{return} :: (\text{Monad } m) \Rightarrow a \rightarrow m a
\]

\[
\text{return } v = \ldots :: m a
\]

\[
\text{fail} :: (\text{Monad } m) \Rightarrow \text{String} \rightarrow m a
\]

\[
\text{fail } s = \ldots :: m a
\]

The instance declaration for Maybe with added type information:

\[
\text{instance Monad Maybe where}
\]

\[
\text{Just } x \gg\gg k = k x \quad -- \text{yields a Just-value}
\]

\[
:: \text{Maybe} a :: a \rightarrow \text{Maybe} b :: \text{Maybe} b
\]

\[
\text{Nothing} \gg\gg k = \text{Nothing} \quad -- \text{yields the Nothing-value}
\]

\[
:: \text{Maybe} a :: a \rightarrow \text{Maybe} b :: \text{Maybe} b
\]

\[
\text{return } x :: \text{a} = \text{Just } x \quad -- \text{yields the Just-value}
\]

\[
:: \text{Maybe} a :: \text{String}
\]

\[
\text{fail } s :: \text{Maybe} a = \text{Nothing} \quad -- \text{yields the empty list}
\]
Using the Maybe Monad: Error Handling (1)

...or: How to compose functions with monadic value ranges.

Let $f'$ and $g'$ be two functions of type:

$$
\begin{align*}
    f' &: a \rightarrow b \\
    g' &: b \rightarrow c
\end{align*}
$$

Obviously, composing $f'$ and $g'$ sequentially is straightforward:

$$
\begin{align*}
    h' &: a \rightarrow c \\
    h' &= (g' \cdot f') \\
    h' \, x &\rightarrow (g' \cdot f') \, x \rightarrow g' \, (f' \, x)
\end{align*}
$$
Using the Maybe Monad: 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:

\[
\begin{align*}
f & : : a \rightarrow \text{Maybe } b \quad \text{-- } f \text{ replaces } f' \\
g & : : b \rightarrow \text{Maybe } c \quad \text{-- } g \text{ replaces } g'
\end{align*}
\]

Unlike \( f' \) and \( g' \), however, \( f \) and \( g \) can not straightforwardly be sequentially composed:

\[
\begin{align*}
h & : : a \rightarrow \text{Maybe } c \\
h \ x = \text{case } (f \ x) \text{ of} \\
& \quad \text{-- } "h = (g \ . \ f)" : \\
& \quad \text{Nothing } \rightarrow \text{Nothing} \\
& \quad \text{Just } y \rightarrow \text{case } (g \ y) \text{ of} \\
& \quad \quad \text{Nothing } \rightarrow \text{Nothing} \\
& \quad \quad \text{Just } z \rightarrow \text{Just } z
\end{align*}
\]

Though possible, the explicit nesting of cases to sequentially compose \( f \) and \( g \) is inconvenient and tedious.
Using the Maybe Monad: 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:

\[
\begin{align*}
  h :: a \to \text{Maybe } c & \quad \text{-- "} h = (g \ . \ f) '' \\
  h x = (f x) \gg= \ y \to (g y) \gg= \ z \to \text{return } z
\end{align*}
\]

or, equivalently, using the do notation:

\[
\begin{align*}
  h :: a \to \text{Maybe } c & \quad \text{-- "} h = (g \ . \ f) '' \\
  h x = \text{do } y \leftarrow f x \\
  \quad z \leftarrow g y \\
  \quad \text{return } z
\end{align*}
\]

...the ‘nasty’ error checks are now hidden in the implementation of the bind operation (\( \gg=} \)) of the Maybe monad.
Using the Maybe Monad: Error Handling (4)

Step 2: Hiding the bind operation \( (\gg\gg=) \).

Note that the sequence of monad operations:

\[
f \ x \ \gg\gg\ y \rightarrow g \ y \ \gg\gg\ z \rightarrow return \ z
\]

can be simplified to:

\[
\begin{align*}
f \ x \ \gg\gg\ \ y \rightarrow g \ y \ \gg\gg\ z \rightarrow return \ z \\
& \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \q
Using the Maybe Monad: Error Handling (5)

...making use of this observation and introducing function:

\[
\text{composeM :: Monad } m \Rightarrow (b \to m c) \to \\
(a \to m b) \to (a \to m c)
\]

\[(g \ '\text{composeM}' \ f) \ x = f \ x \gg= \ g\]

allows an even more pleasing notation for composing \(f\) and \(g\):

\[
h :: a \to \text{Maybe } c \\
h = (g \ '\text{composeM}' \ f)
\]

Hence, we obtain:

\[(g \ \text{composeM} \ f)\]

as the monadic notational counterpart of sequentially composing \(f'\) and \(g'\):

\[(g' \ . \ f')\]
Using the Maybe Monad: Error Handling (6)

Overall: Using monadic sequencing

\[ f \ x \ >>=} \ g \ \ (\text{or equivalently: } (g \ 'composeM' \ f) \ x) \]

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:

\[ (g \ composeM \ f) \ x = f \ x \ >>=} \ g \]
Chapter 11.4.4

The Either Monad
Homework

1. Make 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.
Chapter 11.4.5
The Map Monad
The Map Monad

...the map monad:

```haskell
instance Monad ((->) d) where
    h >>= f = \x -> f (h x) x
    return x = \_ -> x
```

Note:

```
(>>>=) :: ((->) d) a -> (a -> ((->) d) b) -> ((->) d) b
return :: a -> ((->) d) a
(>>>) :: ((->) d) a -> ((->) d) b -> ((->) d) b
```

Lemma 11.4.5.1 (Monad Laws)

Instance ((->) d) of Monad satisfies the three monad laws
ML1, ML2, and ML3.
Example \((w/ \text{String for } d, \text{Int for } a, (\text{Bool, String}) \text{ for } b)\) (1)

\[
(\text{>>=} : ((\to) d) a \to (a \to ((\to) d) b) \to ((\to) d) b) \\
(\equiv (\text{>>=} : (d \to a) \to (a \to (d \to b)) \to (d \to b)) )
\]

\[
h \text{>>=} f = \lambda x \to f (h \ x) \ x
\]

\[
h \text{length} : ((\to) \text{String}) \text{Int}
(\equiv h \text{length} : \text{String} \to \text{Int})
\]

\[
h \text{length} = \text{length}
\]

\[
f \_\_\_\_\_\_p : \text{Int} \to ((\to) \text{String})((,)\text{Bool, String})
(\equiv f \_\_\_\_\_\_p : \text{Int} \to (\text{String} \to (\text{Bool, String}) )
\]

\[
f \_\_\_\_\_\_p n s = (,)((\text{mod } n \text{ 2} == 1))((\text{copy } n \ s)
\]

where \[
\text{copy} n s = \text{if } n > 0 \text{ then } s ++ " " ++ \text{copy} (n-1) s \text{ else ""
\]

\[
g : ((\to) \text{String})((,)\text{Bool, String})
(\equiv g : \text{String} \to (\text{Bool, String}) )
\]

\[
g = \lambda s \to f \_\_\_\_\_\_p (h \text{length} s) s
(\equiv g s = ((\text{mod } (\text{length } s) \text{ 2} == 1,\text{copy } (\text{length } s) s))
\]

\[
h \text{length} \text{>>=} f \_\_\_\_\_\_p
\]

\[
\to ((\lambda x \to f \_\_\_\_\_\_p (h \text{length} x) x) \ (\equiv g) \)
\]

\[
(h \text{length} \text{>>=} f \_\_\_\_\_\_p) \ \text{"Fun"}
\]

\[
\to \ldots \to (\text{True,} \text{"Fun Fun Fun")}
\]
Example \( (w/ \ String \ for \ d, \ Int \ for \ a, \ (Bool,String) \ for \ b) \) (2)

...in more detail:

\[
\text{h_length} \gg= \ f_{cp_p} \\
\quad \rightarrow (\lambda x \rightarrow f_{cp_p} (\text{h_length} \ x) \ x) \\
\quad = \ g \quad (:: \ String \rightarrow (Bool,String))
\]

\[(\text{h_length} \gg= f_{cp_p}) \ "Fun" \]
\[
\quad \rightarrow (\lambda x \rightarrow f_{cp_p} (\text{h_length} \ x) \ x) \ "Fun" \\
\quad = \ g \ "Fun" \\
\quad \rightarrow (\text{mod} (\text{length} \ "Fun") \ 2 = 1, \text{copy} (\text{length} \ "Fun") \ "Fun") \\
\quad \rightarrow (\text{mod} \ 3 \ 2 = 1, \text{copy} \ 3 \ "Fun") \\
\quad \rightarrow (\text{True}, "Fun Fun Fun") \quad (:: (Bool,String))
\]
Example (w/ String for d, Int for a, (Bool,String) for b) (3)

\[
(\gg\gg) \;::\; ((\rightarrow) \;d) \;a \rightarrow (a \rightarrow ((\rightarrow) \;d) \;b) \rightarrow ((\rightarrow) \;d) \;b
\]

\[
h \gg\gg f = \lambda x \rightarrow f (h \;x) \;x
\]

\[
\text{return} \;::\; a \rightarrow ((\rightarrow) \;d) \;a \quad \overset{\text{≡}}{=} \quad \text{return} \;::\; \text{Int} \rightarrow ((\rightarrow) \;\text{String}) \;\text{Int}
\]

\[
\text{return} \;x = \lambda _{-} \rightarrow x \quad \overset{\text{≡}}{=} \quad \text{return} \;::\; \text{Int} \rightarrow (\text{String} \rightarrow \text{Int})
\]

\[
\text{return} \;0 = \lambda _{-} \rightarrow 0 \quad (\overset{\text{≡}}{=} \quad \text{String} \rightarrow \text{Int})
\]

\[
\text{return} \;0 \gg\gg f_{\text{cp\_p}}
\]

\[
\rightarrow\rightarrow \;\lambda x \rightarrow f_{\text{cp\_p}} ((\text{return} \;0) \;x) \;x
\]

\[
\rightarrow\rightarrow \;\lambda x \rightarrow f_{\text{cp\_p}} (\lambda _{-} \rightarrow 0) \;x \;x \quad (\overset{\text{≡}}{=} \quad \text{String} \rightarrow (\text{Bool},\text{String})
\]

\[
(\text{return} \;0 \gg\gg f_{\text{cp\_p}}) \;"Fun"
\]

\[
\rightarrow\rightarrow ((\lambda x \rightarrow f_{\text{cp\_p}} ((\text{return} \;0) \;x) \;x) \;"Fun"
\]

\[
\rightarrow\rightarrow f_{\text{cp\_p}} ((\text{return} \;0) \;"Fun") \;"Fun"
\]

\[
\rightarrow\rightarrow f_{\text{cp\_p}} ((\lambda _{-} \rightarrow 0) \;"Fun") \;"Fun"
\]

\[
\rightarrow\rightarrow f_{\text{cp\_p}} 0 \;"Fun"
\]

\[
\rightarrow\rightarrow (\text{mod} \;0 \;2 \;==\; 1, \text{copy} \;0 \;"Fun")
\]

\[
\rightarrow\rightarrow (\text{False,""}) \quad (\overset{\text{≡}}{=} \quad (\text{Bool},\text{String})
\]

\[
(\text{return} \;1 \gg\gg f_{\text{cp\_p}}) \;"Fun" \rightarrow\rightarrow ... \rightarrow\rightarrow (\text{True,"Fun"})
\]

\[
(\text{return} \;2 \gg\gg f_{\text{cp\_p}}) \;"Fun" \rightarrow\rightarrow ... \rightarrow\rightarrow (\text{False,"Fun Fun"})
\]

\[
(\text{return} \;3 \gg\gg f_{\text{cp\_p}}) \;"Fun" \rightarrow\rightarrow ... \rightarrow\rightarrow (\text{True,"Fun Fun Fun"})
\]
Example (w/ String for d, Int for a) (4)

\[
(\leqtriangleright\leqtriangleright) :: (\rightarrow) d \rightarrow (a \rightarrow ((\rightarrow) d) b) \rightarrow ((\rightarrow) d) b
\]

\[
h \leqtriangleright\leqtriangleright f = \lambda x \rightarrow f (h x) x
\]

\[
return :: a \rightarrow ((\rightarrow) d) a \quad \cong \quad return :: Int \rightarrow ((\rightarrow) String) Int
\]

\[
return x = \lambda _\rightarrow x \quad \cong \quad return :: Int \rightarrow (String \rightarrow Int)
\]

\[
return 3 = \lambda _\rightarrow 3 \quad (:: String \rightarrow Int)
\]

\[
h_{\text{length}} \leqtriangleright\leqtriangleright \text{return}
\]

\[
\rightarrow f \rightarrow \text{return} (h_{\text{length}} x) x
\]

\[
\rightarrow f \rightarrow \text{return} (\text{length} x) x
\]

\[
\rightarrow f \rightarrow (\lambda _\rightarrow \text{length} x) x \quad (:: String \rightarrow Int)
\]

\[
(h_{\text{length}} \leqtriangleright\leqtriangleright \text{return}) \quad \text{"Fun"}
\]

\[
\rightarrow f \rightarrow (\text{return} (h_{\text{length}} x) x) \quad \text{"Fun"}
\]

\[
\rightarrow f \rightarrow \text{return} (h_{\text{length}} \quad \text{"Fun"}) \quad \text{"Fun"}
\]

\[
\rightarrow f \rightarrow \text{return} (\text{length} \quad \text{"Fun"}) \quad \text{"Fun"}
\]

\[
\rightarrow f \rightarrow \text{return} 3 \quad \text{"Fun"}
\]

\[
\rightarrow f \rightarrow (\lambda _\rightarrow 3) \quad \text{"Fun"}
\]

\[
\rightarrow f \rightarrow 3 \quad (:: Int)
\]
Homework

1. Recall the monad operations:
   \[(\ggg) :: (\text{Monad } m) \Rightarrow m a \rightarrow (a \rightarrow m b) \rightarrow m b\]
   \[v \ggg k = \ldots :: m b\]
   \[\text{return} :: (\text{Monad } m) \Rightarrow a \rightarrow m a\]
   \[\text{return } v = \ldots :: m a\]

   Add (most general) type information for the instance declaration of \(((\rightarrow) d)\):
   \[
   \text{instance Monad } ((\rightarrow) d) \text{ where}
   \[
   h \ggg f = \lambda x \rightarrow f (h x) x
   
   \text{return } x = \lambda _\rightarrow x
   \]

2. Evaluate stepwise:
   2.1 \((\text{return } 2 \ggg f_{\text{cp}_p})\) "Fun"
   2.2 \((h_{\text{length}} \ggg \text{return})\) "Fun Prog"
   2.3 \((h_{\text{length}} \ggg \text{return} \ggg f_{\text{cp}_p})\) "Fun"
Chapter 11.4.6
The State Monad
Objective

...modelling global state and 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.
The State Monad

...the state monad:

\[
\text{newtype State \( st \) \( a \) = St \((st \rightarrow (st,a))\)}
\]

\[
\text{instance Monad \((\text{State \( st\)})\) where}
\]

\[
\text{\((\text{St} \ h) \gg= f = \text{St} \ \left(\lambda s \rightarrow \text{let } (s',x) = h \ s \quad \text{in } f' = f \ x \quad \text{in } f' \ s'\right)\)}
\]

\[
\text{:: (st,b)}
\]

-- Applying map \( h :: (st \rightarrow (st,a)) \) to state \( s :: st \)
-- yields a pair \((s',x) :: (st,a)\) onto whose 2nd component \( x :: a \) a map \( f :: a \rightarrow (\text{State \( st\)}) \( b \) \) is applied.
-- This yields a state value \( \text{St} \ f' :: (\text{State \( st\)}) \( b \),
-- whose map value \( f' :: st \rightarrow (st,b) \) is applied to
-- \( s' :: st \) yielding a pair \( f' \ s' :: (st,b) \) as required.

\[
\text{return } x = \text{St} \ \left(\lambda s \rightarrow (s,x)\right)
\]

\[
\text{:: a} \quad \text{:: st} \quad \text{:: (st,a)}
\]

-- \( x :: a \) and every state \( s :: st \) are identically mapped.
Monad Laws for (State st)

Note: For the state monad (State st) the monad operations (>>=) and return have the types:

\[
(\text{>>=}) :: (\text{State st}) a \to (a \to (\text{State st}) b) \to (\text{State st}) b
\]

\[
\text{return} :: a \to (\text{State st}) a
\]

\[
(\text{>>=}) :: (\text{State st}) a \to (\text{State st}) b \to (\text{State st}) b
\]

Lemma 11.4.6.1 (Monad Laws)

Instance (State st) of Monad satisfies the three monad laws ML1, ML2, and ML3.
Notes on the State Monad

The monad operations recalled:

\[
(\gg\gg) \::\; ((\text{Monad} \; m)) \Rightarrow m \; a \rightarrow (a \rightarrow m \; b) \rightarrow m \; b
\]
\[
c \gg\gg k = \ldots \::\; m \; b
\]
\[
\text{return} \::\; ((\text{Monad} \; m)) \Rightarrow a \rightarrow m \; a
\]
\[
\text{return} \; x = \ldots \::\; m \; a
\]

The instance declaration for \((\text{State} \; st)\) with added type information:

\[
\text{instance} \; \text{Monad} \; (\text{State} \; st) \; \text{where}
\]
\[
\text{St} \; h \gg\gg f
\]
\[
\::\; (\text{State} \; st) \; a \quad ::\; a \rightarrow (\text{State} \; st) \; b
\]
\[
= \text{St} \; (\\lambda s \rightarrow \text{let} \ldots \text{in} \; f' \; s') \quad -- \text{constructing}
\]
\[
::\; st \quad ::\; (st,b)
\]
\[
::\; st \rightarrow (st,b) \quad -- \text{a proper state}
\]
\[
::\; (\text{State} \; st) \; b \quad -- \text{value using } m
\]

\[
\text{return} \; x = \text{St} \; (\\lambda s \rightarrow (s,x)) \quad -- \text{constructing a proper}
\]
\[
::\; a \quad ::\; (\text{State} \; st) \; a
\]
\[
-- \text{state value using } x
\]
\[
-- \text{in the simplest way.}
\]
Intuitively

...state transformers

- are mappings $m$ of some type $(\text{st} \rightarrow (\text{st}, \text{a}))$, i.e., $m :: \text{st} \rightarrow (\text{st}, \text{a})$.
- map (or transform) global (internal program) states of type $\text{st}$ into (possibly modified) new states of type $\text{st}$ while additionally computing a result of type $\text{a}$.
- map an argument state $s$ of some type $\text{st}$ to a pair of a (possibly modified) result state $s'$ of type $\text{st}$ and a value $v$ of some type $\text{a}$, i.e., $m \ s \Rightarrow (s', v), \ s :: \text{st}, \ s' :: \text{st}, \ v :: \text{a}$.
The State Monad

...specialized for some concrete state (component) type.

Let $\text{CStT}$ (reminding to ‘Concrete State Type’) be some concrete type (e.g., $\text{Int}$, $[\text{String}]$, ...):

```haskell
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')

  return x = St' (\cs -> (cs,x))
                        :: a
                        :: CStT :: (CStT,a)
```

Note: $\text{State'}$ is a 1-ary type constructor while $\text{State}$ is a 2-ary type constructor.
Monad Laws for (State')

Note: For the state monad State' the monad operations (>>=) and return have the types:

\[
(>>=) :: \text{State}' a \to (a \to \text{State}' b) \to \text{State}' b \\
\text{return} :: a \to \text{State}' a
\]

Lemma 11.4.6.2 (Monad Laws)

Instance State' of Monad satisfies the three monad laws ML1, ML2, and ML3.
The State Monad Reconsidered (1)

...sometimes renaming objects helps getting things clear(er).

Think about \texttt{st\_otw} as a type variable where the values of appropriate type instances of \texttt{st\_otw} describe or model the

- State of the World (\texttt{St\_otW}).

The sequencing operation (\texttt{>>=}) of the state monad (\texttt{State st\_otw}) allows then to transform a current state of the world into a new state 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.

Intuitively, this suggests for a state transformer:

\texttt{state\_transformer :: st\_otw -> st\_otw}

Note: (\texttt{State st\_otw}), (\texttt{>>=}) make this a bit more complex.
The State Monad Reconsidered (2)

newtype (State \( \text{stotw} \)) \( \text{a} \) = \( \text{St} \) (\( \text{stotw} \rightarrow (\text{stotw},\text{a}) \))

instance Monad (State \( \text{stotw} \)) where
\( \text{St} \) \( h \) >>= \( f \)
= \( \text{St} \) (\( \text{current\_state} \rightarrow \)

let (intermediate\_state,\( \text{x} \)) = \( h \) \( \text{current\_state} \)

\( \text{St} \) \( g \) = \( f \) \( \text{x} \)

(\( \text{new\_state},\text{z} \)) = \( g \) intermediate\_state

in (\( \text{new\_state},\text{z} \))

return \( \text{x} \) = \( \text{St} \) (\( \text{current\_state} \rightarrow (\text{current\_state},\text{x}) \))

Note resp. compare:

\[ (\gg=) :: (\text{State} \text{stotw}) \text{a} \rightarrow (\text{a} \rightarrow (\text{State} \text{stotw}) \text{b}) \rightarrow (\text{State} \text{stotw}) \text{b} \]

\[ \text{return} :: \text{a} \rightarrow (\text{State} \text{stotw}) \text{a} \]

\[ (g \ . \ f) = (f; g) = \text{x} \rightarrow \text{let intermediate} = f \ \text{x} \]
\[ y = g \ \text{intermediate} \]
\[ \text{in} \ y \quad -- \ y = g \ (f \ \text{x}) \]
Chapter 11.4.7

The Input/Output Monad
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).
- **(>>>=)**: If \( p, q \) are commands, then \( p \ >>>= \ q \) 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.
Illustrating the Nature of Commands

**Command** $\texttt{cmd} :: \texttt{IO a}$

- **IO Operation**
  - Component with **‘procedural’** behaviour: \( \texttt{IO operation} \) generates irreversible side effect.

  **Command** $\texttt{cmd} :: \texttt{IO a}$

**Command yielding function** $\texttt{f_cmd} :: \texttt{a -> IO b}$

- **IO Operation**
  - Component with **‘procedural’** behaviour: \( \texttt{IO operation} \) generates irreversible side effect (depending possibly on the \( \texttt{a-value} \)).

  **Command yielding function** $\texttt{f_cmd} :: \texttt{a -> IO b}$

- **b**
  - Component with **‘functional’** behaviour: Computes \( \texttt{b value} \) as result of the command (depending possibly on the \( \texttt{a value} \)).
...the operational meaning of \((\text{cmd} \gg= \text{f\_cmd})\):

\[
\text{cmd} :: \text{IO a} \quad \text{f\_cmd} :: \text{a \to IO b}
\]

\[
\text{cmd} \gg= \text{f\_cmd} \equiv \text{cmd} \gg= \lambda x \to \text{f\_cmd} x
\]
Illustrating

...the operational meaning of \((\text{cmd} \gg \text{cmd}')\):

\[
\text{cmd} :: \text{IO } a \\
\text{cmd}' :: \text{IO } b
\]

\[
\text{cmd} \gg \text{cmd}' \equiv \text{cmd} \gg \_ \rightarrow \text{cmd}'
\]
Illustrating

...the operational meaning of \( \text{return} \):

Component with ‘procedural’ behaviour: ‘empty’; no IO operation, no side effect.

Component with ‘functional’ return behaviour: Forwards the a-value as the result of the command.

Command \( \text{return} :: a \rightarrow \text{IO} \ a \)
The Type

...of all read commands is

▶ \((\text{IO } a)\) (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

▶ \((\text{IO } ())\), 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.

Lemma 11.4.7.1 (Monad Laws)

Instance \(\text{IO}\) of \(\text{Monad}\) satisfies the three monad laws \(\text{ML1}\), \(\text{ML2}\), and \(\text{ML3}\).
Input/Output and 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:

![Diagram showing input, Haskell program, and output]


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)

by 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:

Note that input/output operations are a major source for side effects: read statements e.g. will yield different values for every call which directly causes 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?"
     putStrLnLine ("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
Examples: Simple IO Programs (3)

...note the relationship of do-notation

```
  do writeFile "testFile.txt" "Hello File System!"
      putStr "Hello World!"
```

and (canonic) monadic operations:

```
  writeFile "testFile.txt" "Hello File System!" >>
  putStr "Hello World!"
```

Note also sometimes (subtle) difference in result types:

```
Main>putStr ('a':('b':('c':[]))))
      putStrLn (head ['x','y','z'])

  -->> abc :: IO ()  -->> x :: IO ()
```

but

```
Main>('a':('b':('c':[])))
     putStrLn ['x','y','z']

  -->> "abc" :: [Char]  -->> 'x' :: Char
```

```
Main>print "abc"
     putStrLn 'x'

  -->> "abc" :: IO ()  -->> 'a' :: IO ()
```
Examples: Simple IO Programs (4)

...the sequence of output commands

```
    do writeFile "testFile.txt" "Hello File System!"
    putStrLn "Hello World!"
```

is equivalent to:

```
    writeFile "testFile.txt" "Hello File System!" >>
    putStrLn "Hello World!"
```
Examples: Simple IO Programs (5)

...the sequence of input/output commands with local declarations within a do-construct

```haskell
reverse2lines :: IO ()
reverse2lines = do line1 <- getLine
                 line2 <- getLine
                 let rev1 = reverse line1
                 let rev2 = reverse line2
                 putStrLn rev2
                 putStrLn rev1
```

is equivalent to the following one without:

```haskell
reverse2lines :: IO ()
reverse2lines = do line1 <- getLine
                 line2 <- getLine
                 putStrLn (reverse line2)
                 putStrLn (reverse line1)
```

In Closing (1)

...monadic input/output in Haskell allows us to conceptually think of a Haskell program as consisting of a

- a purely functional computational core and
- a procedural-like interaction shell.

In Closing (2)

...the monad concept of Haskell allows to

▸ conceptually separate functions belonging to the
  ▸ computational core (pure functions)
  ▸ interaction shell (impure functions, i.e., performing
    input/output operations causing side effects).

by assigning different types to them:

≈ Int, Real, String,... vs. IO Int, IO Real, IO String,...

with type constructor IO a pre-defined monad.

▸ precisely specify the evaluation order of functions of the
interaction shell (i.e., of basic input/output primitives
provided by Haskell) by using the monadic sequencing
operations.

...see e.g. lecture notes of LVA 185.A03 Funktionale Programmierung for further details and examples.
Chapter 11.5
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.
Chapter 11.5.1

Folding Trees
The Setting (1)

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:
The Setting (2)

Means:

Two functional approaches

- w/out monads
- w/ monads

respectively, for comparison.
1st Approach: Straightforward w/out Monads

...using a recursive function:

\[
\begin{align*}
\text{sum} & : \text{Tree Int} \rightarrow \text{Int} \\
\text{sum Nil} & = 0 \\
\text{sum (Node n t1 t2)} & = n + \text{sum t1 + sum t2}
\end{align*}
\]

**Note:**

- The evaluation order of the right-hand term of the (non-trivial) defining equation of \text{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:

\[
\begin{align*}
\text{sum (Node n t1 t2)} & = n + \text{sum t1 + sum t2} \\
\text{sum (Node n t1 t2)} & = \text{sum t2 + n + sum t1} \\
\ldots \\
\text{sum (Node n t1 t2)} & = \text{sum t2 + sum t1 + n}
\end{align*}
\]
2nd Approach: Using the Identity Monad

...using the identity monad $\text{Id}$:

```haskell
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 $n :: \text{Int}$ to $\text{num}$
     s1 <- sum' t1   -- Evaluating left subtree
  return (s2+num+s1) -- Yielding $\text{Id}(\text{num}+s1+s2) :: \text{Id Int}$ 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 $s2+\text{num}+s1$. 
The Identity Monad

Recall the identity monad \( \text{Id} \):

\[
\text{newtype } \text{Id} \ a = \text{Id} \ a \\
\text{instance Monad } \text{Id} \text{ where} \\
(\text{Id} \ x) >>= f = f \ x \\
\text{return} = \text{Id}
\]

Note:

- \( \text{Id} \): 1-ary type constructor, i.e., \( \text{Id} \ a \) denotes a type.
- \( \text{Id} \): 1-ary data (or value) constructor, i.e., \( \text{Id} \ v, v :: a \), denotes a value: \( \text{Id} \ v :: \text{Id} \ a \).
Illustrating the Imperative Flavour of sum’

...unlike \texttt{sum}, \texttt{sum’} enjoys an ‘imperative’ flavour quite similar to sequentially sequencing assignment statements of some imperative programming language:

\begin{center}
\begin{tabular}{ll}
\textbf{Imperative} & \textbf{Monadic} \\
s2 & \texttt{do s2 <- sumTree t2} \\
s1 & \texttt{s1 <- sumTree t1} \\
num & \texttt{num <- return n} \\
return (s2+s1+num); & \texttt{return (s2+s1+num)}
\end{tabular}
\end{center}
3rd Approach: Using `extract` and Monad `Id`

...using an extraction function to allow a function \( \text{sum}'' \) of type 
\( \text{(Tree Int -> Int)} \):

\[
\text{extract :: Id a -> a}
\]
\[
\text{extract (Id x) = x}
\]

This enables:

\[
\text{sum}'' :: \text{Tree Int -> Int}
\]
\[
\text{sum}'' = \text{extract . sum}'
\]

Example:

\[
t = (\text{Node 5 (Node 3 Nil Nil) (Node 7 Nil Nil)})
\]
\[
\text{sum}'' \ t \ ->> (\text{extract . sum}') \ t
\]
\[
\quad ->> \text{extract (sum' \ t)}
\]
\[
\quad ->> \text{extract (Id 15)}
\]
\[
\quad ->> 15
\]
Chapter 11.5.2
Numbering Tree Labels
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 \( t :: \text{Tree Char} \):

\[
t = \text{Branch} \left( \text{Branch} \left( \text{Leaf} \ 'a' \right) \left( \text{Leaf} \ 'b' \right) \right) \\
\left( \text{Branch} \left( \text{Leaf} \ 'b' \right) \left( \text{Leaf} \ 'c' \right) \right)
\]

shall be transformed into the tree value \( t' :: \text{Tree Int} \):

\[
t' = \text{Branch} \left( \text{Branch} \left( \text{Leaf} \ 0 \right) \left( \text{Leaf} \ 1 \right) \right) \\
\left( \text{Branch} \left( \text{Leaf} \ 2 \right) \left( \text{Leaf} \ 3 \right) \right)
\]
The Setting (2)

Means:

Two functional approaches

- w/out monads
- w/ monads

respectively, for comparison.
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 \( \text{lab} \) is tedious and intricate.
2nd Approach: Using the State Monad (1)

...using the pattern of the 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 dropped by bracketing expr. \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 State Monad (2)

...the renaming of labels can now be achieved as follows:

```haskell
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 State Monad (3)

Example: Applying $\text{label}'$ to tree value $t$:

$$t = \text{Branch} \ (\text{Branch} \ (\text{Leaf} \ 'a') \ (\text{Leaf} \ 'b')) \ (\text{Branch} \ (\text{Leaf} \ 'b') \ (\text{Leaf} \ 'c')))$$

we get as desired:

$$\text{label}' \ t \rightarrow \text{Branch} \ (\text{Branch} \ (\text{Leaf} \ 0) \ (\text{Leaf} \ 1)) \ (\text{Branch} \ (\text{Leaf} \ 2) \ (\text{Leaf} \ 3)) \equiv t'$$
Homework

Provide (most general) type information for the defining equations of

1. the operations
   1.1 (>>=)
   1.2 `return`

   of the state instance declaration of `Label`.

2. the functions
   2.1 `label'`
   2.2 `lab'`
   2.3 `get_label`

   of the monadic solution of the numbering problem.
Chapter 11.5.3
Renaming Tree Labels
The Setting

Given:

\[
data \text{ Tree } a = \text{Nil} | \text{Node } a (\text{Tree } a) (\text{Tree } a)
\]

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 a Monadic Approach (1)

We start defining:

\[
\text{number_tree} :: \text{Eq a} \Rightarrow \text{Tree a} \rightarrow \text{State a (Tree Int)}
\]
\[
\text{number_tree } \text{Nil} = \text{return } \text{Nil}
\]
\[
\text{number_tree } (\text{Node } x \ t1 \ t2) =
\]
\[
= \text{do } \text{num } \leftarrow \text{number_node } x
\]
\[
\text{nt1 } \leftarrow \text{number_tree } t1
\]
\[
\text{nt2 } \leftarrow \text{number_tree } t2
\]
\[
\text{return } (\text{Node } \text{num nt1 nt2})
\]

...post-poning the implementation of \text{number_node}.
Towards a Monadic Approach (2)

Additionally, we introduce a table type

type Table a = [a]

for storing pairs of the form

\((\text{<string>}, \text{<number of occurrences>})\)

In particular, the list (or table) value

\([\text{True}, \text{False}]\)

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`:

```haskell
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`:

```haskell
number_node :: Eq a => a -> (State a) Int
number_node x = St (num_node x)
```

```haskell
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.
```

```haskell
lookup :: Eq a => a -> Table a -> Int
lookup x table = ... -- Homework: Completing the im-
-- plementation of lookup.
```
Mon. Approach: Using the State Monad (3)

Putting the pieces together, \(\text{number\_tree}\) is fully defined:

\[
\text{number\_tree} :: \text{Eq } a \Rightarrow \text{Tree } a \rightarrow \text{State } a (\text{Tree } \text{Int})
\]

\[
\text{number\_tree } \text{Nil} = \text{return } \text{Nil}
\]

\[
\text{number\_tree} (\text{Node } x \ t1 \ t2)
\]

\[
= \text{do } \text{num } \leftarrow \text{number\_node } x
\]

\[
\text{nt1 } \leftarrow \text{number\_tree } t1
\]

\[
\text{nt2 } \leftarrow \text{number\_tree } t2
\]

\[
\text{return } (\text{Node } \text{num } \text{nt1 } \text{nt2})
\]

Note, for every value \(t :: \text{Eq } a \Rightarrow \text{Tree } a\), e.g., the tree of the illustrating example, we can conclude (functional and hence) type correctness:

\[
\text{number\_tree } t :: \text{State } a (\text{Tree } \text{Int})
\]

\[
\equiv \text{(State } a \text{) (Tree } \text{Int})
\]

\[
\equiv \text{((State } a \text{) (Tree } \text{Int))}
\]
...introducing an **extraction function**:

```haskell
extract :: State a b -> b
extract (St st) = snd (st [])
```

we get the implementation of the initially envisioned function **number**:

```haskell
number :: Eq a => Tree a -> Tree Int
number = extract . number_tree
```
Homework

Provide (most general) type information for the defining equations of

1. the operations
   1.1 (>>=)
   1.2 return

   of the state instance declaration of \( \text{State a} \).

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.
Chapter 11.6
MonadPlus
Chapter 11.6.1

The Type Constructor Class MonadPlus
The Type Constructor Class MonadPlus

...monads with an appropriate ‘zero’ element and ‘plus’ operation can be instances of the type constructor class MonadPlus.

Type Constructor Class MonadPlus

class Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a

MonadPlus Laws

m >>= (\x -> mzero) = mzero \hspace{1cm} (MPL1)
mzero >>= m = mzero \hspace{1cm} (MPL2)
m `mplus` mzero = m \hspace{1cm} (MPL3)
mzero `mplus` m = m \hspace{1cm} (MPL4)
Proper Instances of the Type Class MonadPlus

...must satisfy additionally to all monad laws the MonadPlus laws, i.e., the laws for the ‘zero’ element and the ‘plus’ operation, which, intuitively, mean:

- \texttt{mzero} is left-zero and right-zero for (>>=).
- \texttt{mzero} is left-unit and right-unit for \texttt{mplus}.

Proof obligation: As usual for type class instances, it is the programmer’s obligation to prove that their instances of MonadPlus satisfy all monad and MonadPlus laws.

Note: The \texttt{IO} monad can not be made an instance of MonadPlus because of the lack of an appropriate ‘zero’ element.
Chapter 11.6.2
The Maybe MonadPlus
The Maybe Instance of MonadPlus

...the Maybe instance of the type constructor class MonadPlus:

```haskell
instance MonadPlus Maybe where
  mzero = Nothing
  Nothing 'mplus' ys = ys
  xs 'mplus' ys = xs
```

Lemma 11.6.2.1 (Maybe Instance of MonadPlus)
Instance Maybe of MonadPlus satisfies all monad and monadPlus laws.
Chapter 11.6.3

The List MonadPlus
The \texttt{[]} Instance of MonadPlus

...the \texttt{list} instance of the type constructor class \texttt{MonadPlus}:

\begin{verbatim}
instance MonadPlus [] where
  mzero = []
  mplus = (++)
\end{verbatim}

Lemma 11.6.3.1 (List Instance of MonadPlus)

Instance \texttt{[]} of \texttt{MonadPlus} satisfies all monad and \texttt{MonadPlus} laws.
Homework

1. Provide (most general) type information for
   1.1 the monadPlus 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 [] instance
      of MonadPlus.

2. Which of the other monads considered in Chapter 11.4 (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 MonadPlus operations.
   2.2 Prove that all instances satisfy the MonadPlus laws.
Chapter 11.7

Summary
Summary

**Monads** (i.e., instances of type constructor class **Monad**) combine features of

- **functors and functional composition:**
  \[(\ggg) \colon \text{m a} \rightarrow (\text{a} \rightarrow \text{m b}) \rightarrow \text{m b}\]
  \[c \ggg k \ggg k' \ggg k'' \ggg \ldots\]

**Monads** are thus well-suited for

- structuring and sequencing evaluation steps

because they

- allow to specify sequential program parts systematically.
- offer an adequately high abstraction by decoupling the data type forming a monad (instance) from the structure of computation.
- support equational reasoning, e.g., by applying the **monad laws**.
On the Origins of the Notion Monad: Monads in Philosophy

...monad, derived from Greek *monas* meaning unit(y) (in German: Eins, Einheit).

Gottfried Wilhelm Leibniz (*1646 in Leipzig; †1716 in Hannover) used monad as a counterpart of

▶ ‘atom’ denoting like atom ‘something indivisible’

to ‘solve’ (possibly more accurate: tackle) the so-called

▶ body-soul problem (in German: Leib-Seele-Problem)

evolving from the body-soul dualism in 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/used the monad notion into

- category theory

and

- programming languages theory

as a means for describing the

- semantics of programming languages:

Monads in Functional Programming

...later on, the monad notion became particularly popular (w/out the background from philosophy and category theory) in the field of functional programming (Philip Wadler, 1992), especially because monads as in the sense of Haskell e.g.

- allow to introduce some useful aspects of imperative programming such as sequencing into functional programming,

- are well suited for smoothly integrating 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!
On the other Hand

...the origin and connection of the monad notion to (often difficult considered) fields like

- philosophy, category theory, programming languages theory, programming languages semantics

might be responsible for awarding the monad notion an aura of something

- mystically, wondrously that is difficult to grasp (‘once I will have understood monads, I will have understood functional programming’ letting monads appear the Holy Grail of functional programming).

Overall, this gives monads a mythical flavour.
On Constituting the Mythical Aura

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 $C$ is a triple $(T, \eta, \mu)$, where $T : C \to C$ is a functor, $\eta : Id_C \to T$ and $\mu : T^2 \to T$ are natural transformations and the following equations hold:

$$\mu_TA; \mu_A = T(\mu_A); \mu_A$$
$$\eta_TA; \mu_A = id_{TA} = T(\eta_A); \mu_A$$

... “a monad is a monoid in the category of endofunctors.”
...the monad notion in functional programming (applying to Haskell, too) lost its connection to the monad notion in philosophy and category theory (almost) completely, and hence, everything which might be or which might be considered a mystery or a 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 ‘Functional Gap’ and its Closing (1)

...smashing the myth behind functional programming monads.

► Function application (‘mapping over’):

($) :: (a -> b) -> a -> b

\[ g \$ x = g \ x \]

► Special case (\( m \ a \) for \( a \), \( m \ b \) for \( b \)):

($) :: (m a -> m b) -> m a -> m b

\[ g \$ x = g \ x \]

► Sequential function composition (‘sequencing’):

(\.) :: (b -> c) -> (a -> b) -> (a -> c)

\[ (f \ . \ g) x = f \ (g \ x) \]

► Special case (\( m \ a \) for \( a \), \( m \ b \) for \( b \), \( m \ c \) for \( c \)):

(\.) :: (m b -> m c) -> (m a -> m b) -> (m a -> m c)

\[ (f \ . \ g) x = f \ (g \ x) \]

...one implementation fits all types: Parametric polymorphism
On the ‘Functional Gap’ and its Closing (2)

- **Functorial mapping (‘mapping over’):**
  \[ \text{fmap} :: (\text{Functor } f) \Rightarrow (a \to b) \to f \ a \to f \ b \]
  \[ \text{fmap } g \ c = \ldots \ ‘(\text{unpack, map, pack})’ \]

- **Functorial mapping 2&3 ( + sequencing):**
  \[ \text{(fmap2)} :: (\text{Functor2 } f) \Rightarrow (a \to f \ b) \to f \ a \to f \ b \]
  \[ \text{(fmap2)} \ k \ c = \ldots \ ‘(\text{unpack, map})’ \]

- **Functorial mapping 3 ( + sequencing):**
  \[ \text{(fmap3)} :: (\text{Functor3 } f) \Rightarrow (f \ a \to b) \to f \ a \to f \ b \]
  \[ \text{(fmap3)} \ k \ c = \ldots \ ‘(\text{map, pack})’ \]

- **(Monadic) sequencing (+ mapping): Changing Focus**
  \[ \text{(||=)} :: (\text{Monad } m) \Rightarrow m \ a \to (a \to m \ b) \to m \ b \]
  \[ \text{(||=)} \ c \ k = \text{fmap2} \ k \ c \ ‘(\text{unpack, map, repeat } ||=)’ \]

...type-specific implementations fit every 1-ary type constructor:

*Ad hoc* polymorphism
Overall

...AdvancedSequencing, AdvancedComposing (or something alike) instead of Monad might have been a better choice for the class name for avoiding suggesting any mysticism:

```haskell
class AdvancedSequencing as where
  (>>=) :: as a -> (a -> as b) -> as b
  return :: a -> as a
  (>>) :: as a -> as b -> as b
  fail :: String -> as a

  c >>= k = c >>= \_ -> k
  fail s = error s

class AdvancedComposing ac where...
```

instead of the actually provided type constructor class Monad:

```haskell
class Monad m where...
```
Chapter 11.8

References, Further Reading
Chapter 11: Further Reading (1)

Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 17, Monaden)


Ernst-Erich Doberkat. *Haskell: Eine Einführung für Objektorientierte*. Oldenbourg Verlag, 2012. (Kapitel 5, Ein-/Ausgabe; Kapitel 7, Monaden)

Chapter 11: Further Reading (2)


Chapter 11: Further Reading (3)


Chapter 11: 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)


Chapter 11: Further Reading (5)


Chapter 11: Further Reading (6)


Chapter 11: Further Reading (7)


Chapter 11: Further Reading (8)


A (Reasonably) Comprehensive List of Tutorials on Monads: haskell.org/haskellwiki/Monad_tutorials.
Chapter 11: Monads - Background Reading (9)


Chapter 12

Arrows
Chapter 12.1

Motivation
Motivation

The higher-order type constructor class Arrow

- complements the type class Monad

with a complementary mechanism for

- function composition

which is amenable for 2-ary type constructors and useful e.g. for

- functional reactive programming (cf. Chapter 15).
Chapter 12.2
The Type Constructor Class Arrow
The Type Constructor Class Arrow

Arrows are 2-ary type constructors, which are instances of the type constructor class **Arrows** obeying the arrow laws:

```haskell
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

  ```haskell
  firstfun f = \((x,y)\) \to (f x, y)
  ```
The Arrow Laws

Proper instances of the type constructor class `Arrow` must satisfy the following nine arrow laws:

### Arrow Laws

1. **Identity**: \[ \text{pure } \text{id} \implies f = f \]  
   \[ (\text{AL1}): \text{identity} \]

2. **Identity**: \[ f \implies \text{pure } \text{id} = f \]  
   \[ (\text{AL2}): \text{identity} \]

3. **Associativity**: \[ (f \implies g) \implies h = f \implies (g \implies h) \]  
   \[ (\text{AL3}): \text{associativity} \]

4. **Functor Composition**: \[ \text{pure } (g \cdot f) = \text{pure } f \implies \text{pure } g \]  
   \[ (\text{AL4}): \text{functor composition} \]

5. **Extension**: \[ \text{first } (\text{pure } f) = \text{pure } (f \times \text{id}) \]  
   \[ (\text{AL5}): \text{extension} \]

6. **Functor**: \[ \text{first } (f \implies g) = \text{first } f \implies \text{first } g \]  
   \[ (\text{AL6}): \text{functor} \]

7. **Exchange**: \[ \text{first } f \implies \text{pure } (\text{id} \times g) = \text{pure } (\text{id} \times g) \implies \text{first } f \]  
   \[ (\text{AL7}): \text{exchange} \]

8. **Unit**: \[ \text{first } f \implies \text{pure } \text{fst} = \text{pure } \text{fst} \implies f \]  
   \[ (\text{AL8}): \text{unit} \]

9. **Association**: \[ \text{first } (\text{first } f) \implies \text{pure } \text{assoc} = \text{pure } \text{assoc} \implies \text{first } f \]  
   \[ (\text{AL9}): \text{association} \]
Instance (\(\rightarrow\)) of Class Arrow (1)

...making the type constructor (\(\rightarrow\)) an instance of the type constructor class Arrow:

```
instance Arrow (\(\rightarrow\)) where
  pure f = f
  f >>> g = g . f
  first f = f × id
```

where

\[
(\times) :: (b \rightarrow c) \rightarrow (d \rightarrow e) \rightarrow (b,d) \rightarrow (c,e)
\]

\[
(f \times g) \sim (bv, dv) = (f \ bv, g \ dv) :: (c,e)
\]

Note: Defining `first` by `first f = \(\lambda (b,d) \rightarrow (f \ b, d)\)` would have been equivalent.
Instance (\(\rightarrow\)) of Class Arrow (2)

...in more detail with added type information:

```haskell
class Arrow a where
  pure :: ((\(\rightarrow\)) b c) -> a b c
  (>>>) :: a b c -> a c d -> a b d
  first :: a b c -> a (b,d) (c,d)
```

...making (\(\rightarrow\)) an instance of Arrow means constructor a equals (\(\rightarrow\)):

```haskell
instance Arrow (\(\rightarrow\)) where
  pure f = f
  f >>> g = g . f
  first f = f \times id
```

Recall: Defining `first` by `first f = \((b,d) \rightarrow (f b, d)\)` would have been equivalent.
Utility Functions (1)

The product map $\times$ (recalled):

$$(\times) :: (a -> a') -> (b -> b') -> (a,b) -> (a',b')$$

$$(f \times g)\sim(a,b) = (f\ a,\ g\ b)$$

Regrouping arguments via assoc, unassoc, and swap:

$$assoc :: ((a,b),c) -> (a,(b,c))$$

$$assoc\sim((x,y),z) = (x,(y,z))$$

$$unassoc :: (a,(b,c)) -> ((a,b),c)$$

$$unassoc\sim(x,((y,z)) = ((x,y),z))$$

$$swap :: (a,b) -> (b,a)$$

$$swap\sim(x,y) = (y,x)$$

The dual analogue to the map first, the map second:

$$second :: Arrow\ a => a\ b\ c -> a\ (d,b)\ (d,c)$$

$$second\ f = pure\ swap >>> first\ f >>> pure\ swap$$
Utility Functions (2)

...derived operators for the type constructor class Arrow:

\[(***) \:: \text{Arrow} \ a \Rightarrow \text{a} \ b \ c \Rightarrow \text{a} \ b' \ c' \Rightarrow \text{a} \ (b,b') \ (c,c')\]

\[f *** g = \text{first} \ f >>> \text{second} \ g\]

\[(&&&) \:: \text{Arrow} \ a \Rightarrow \text{a} \ b \ c \Rightarrow \text{a} \ b \ c' \Rightarrow \text{a} \ b \ (c,c')\]

\[f &&& g = \text{pure} \ (\_ \Rightarrow (b,b)) >>> (f *** g)\]

\[\text{idA} :: \text{Arrow} \ a \Rightarrow \text{a} \ b \ b\]

\[\text{idA} = \text{pure} \ \text{id}\]
Application: Modelling Circuits (1)

The map add introduces a notion of computation:

\[
\text{add} :: (b \to \text{Int}) \to (b \to \text{Int}) \to (b \to \text{Int})
\]
\[
\text{add} \ f \ g \ z = f \ z + g \ z
\]

...which can be generalized in various ways.
Application: Modelling Circuits (2)

First, generalizing `add` to state transformers:

```haskell
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)
```

Illustration:
Second, generalizing `add` to non-determinism:

```haskell
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 ]
```
Application: Modelling Circuits (4)

Third, generalizing \texttt{add} to \texttt{map} transformers:

\begin{verbatim}
  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
\end{verbatim}
Application: Modelling Circuits (5)

Fourth, generalizing `add` to simple automata:

```haskell
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, it allows us

▶ modelling of synchronous circuits (with feedback loops).
Application: Modelling Circuits (6)

- Functions and programs often contain components that are ‘function-like’ ‘w/out being just functions.’
- **Arrows** define a common interface for coping with the “notion 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.
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.
Back to the Application

...next we are going to implement the previously introduced types as instances of the type constructor class Arrow. To this end, we reintroduce them as new types using newtype:

```haskell
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))
```
Instance `(State s)` of Class `Arrow (1)`

...making state transformers an instance of `Arrow`:

```haskell
newtype State s i o = ST ((s,i) -> (s,o))

instance Arrow (State s) where
    pure f = ST (id \times f)
    ST f >>> ST g = ST (g \cdot f)
    first (ST f) = ST (assoc \cdot (f \times id) \cdot unassoc)
```
Instance \((\text{State } s)\) of Class Arrow (2)

...in more detail with added type information:

```haskell
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 \((\text{State } s)\) an instance of Arrow means type constructor variable \(a\) is set to \((\text{State } s)\):

```haskell
newtype State s i o = ST ((s,i) -> (s,o))
```

```haskell
instance Arrow (State s) where
  pure f = ST (id \times f)
  ST f <<< ST g = ST (g . f)
  first (ST f) = ST (assoc . (f \times id) . unassoc)
```

...making “non-determinism” 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])
Instance NonDet of Class Arrow (2)

...in more detail with added type information:

```haskell
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])
```

```haskell
instance Arrow NonDet where
    pure f = ND (\b -> [f b])
    (>>>) :: (->) b c -> NonDet b c
    ND f >>> ND g = ND (\b -> [d | c <- f b, d <- g c])
    :: NonDet b c -> NonDet c d
    first (ND f) = ND (\(b,d) -> [(c,d) | c <- f b])
    :: NonDet b c -> (b,d) -> (c,d)
```

Instance (MapTrans s) of Class Arrow (1)

...Making map transformers 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)
Instance \((\text{MapTrans } s)\) of Class Arrow (2)

...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 \((\text{MapTrans } s)\) an instance of Arrow means type constructor variable \(a\) is set to \((\text{MapTrans } s)\):

```
MapTrans s i o = MT ((s -> i) -> (s -> o))
```

```
instance Arrow (MapTrans s) where
    pure f = MT (f .)
    (>>>): (MapTrans s) b c -> (MapTrans s) c d -> (MapTrans s) b d
    first (MT f) = MT (zipMap . (f x id) . unzipMap)
```

```
:: (->) b c :: (MapTrans s) b c :: (MapTrans s) b c
:: (MapTrans s) b c :: (MapTrans s) c d :: (MapTrans s) b d
:: (MapTrans s) b c :: (MapTrans s) (b,d) (c,d)
```
Instance Auto of Class Arrow (1)

...Making simple automata an instance of Arrow:

```haskell
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'))
```
Instance Auto of Class Arrow (2)

...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 :: (->) b c
        = A (\b -> (f b, pure f))
    A f >>> A g :: Auto b c
        = A (\b -> let (c,f’) = f b
                  (d,g’) = g c
                  in (d, f’ >>> g’)))

first (A f) :: Auto b c
define
    first (A f) :: Auto b c
        = A (\(b,d) -> let (c,f’) = f b
                  in ((c,d),first f’))

References

Appendix A
...generalization:

Consider the general combinator:

```haskell
addA :: Arrow a => a b Int -> a b Int -> a b Int
addA f g = f &&& g >>> pure (uncurry (+))
```

**Note:** Each of the considered variants of `add` results as a specialization of `addA` with the corresponding `arrow`-type.
Summing up

- **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.
Last but not least (1)

...compare (same color means “correspond to each other”):

\[(.) :: (b \to c) \to (a \to b) \to (a \to c)\]
\[f \cdot g \, v = f \, (g \, v)\]

\[(); :: (a \to b) \to (b \to c) \to (a \to c)\]
\[f ; g = g \cdot f\] -- pointfree

\[(>>;); :: a \to (a \to b) \to b\]
\[v >>; f = f \, v\]

\[();<<) :: (a \to b) \to a \to b\]
\[f ;< v = v >>; f\] -- Non-monadic operations

\[=<<) :: Monad m \Rightarrow (a \to m \, b) \to m \, a \to m \, b\] -- Monadic op.
\[f =<< x = x >>= f\]

\[(>>=) :: Monad m \Rightarrow m \, a \to (a \to m \, b) \to m \, b\]
\[m >>= k = k \, v...\] -- "m = dc \, v"

\[(@(>) :: Monad m \Rightarrow (a \to m \, b) \to (b \to m \, c) \to (a \to m \, c)\]
\[f @> g = \lambda x \to (f \, x) >>= g\]

\[(<@<) :: Monad m \Rightarrow (b \to m \, c) \to (a \to m \, b) \to (a \to m \, c)\]
\[f <@< g = g @> f\] -- pointfree
Last but not least (2)

\[ (\gggg) :: \text{Arrow } a \Rightarrow a \ b \ c \rightarrow a \ c \ d \rightarrow a \ b \ d \]

...introduces composition for 2-ary type constructors.

Reconsider now instance \((\rightarrow)\) of class \text{Arrow}:

```haskell
instance Arrow (\rightarrow) where
    pure f = f
    f >>> g = g \ . \ f
    first f = f \times id
```

This means: For \((\rightarrow)\) as \text{Arrow} instance

- arrow composition boils down to ordinary function composition, i.e.: \((\gggg) = (.)\)
Chapter 12.3
A Fresh Look at the Haskell Class Hierarchy
Monoids, Monads, Functors, Arrows,…

…as part of the Haskell’98 type class hierarchy:

- **Eq**
  - `(==)` `/=`
- **Ord**
  - `compare` `()` `()` `()` `()` `()`
  - `max` `min`
- **Num**
  - `+` `-` `*`
  - `negate` `abs` `signum` `fromInteger`
- **Enum**
  - `succe` `pred` `toEnum` `fromEnum` `enumFrom` `enumFromThen` `enumFromThenTo`
- **Monoid**
  - `mempty` `mappend` `mconcat`
- **Functor**
  - `fmap`
- **Monad**
  - `(>>=)`
  - `(>>>)` `return` `fail`
- **MonadPlus**
  - `mZero` `mPlus`
- **Applicative**
  - `pure` `(<*>)`
- **Arrow**
  - `pure` `(:>>:``

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:

- **Eq**
  - `(==) (/=)

- **Ord**
  - `compare` (`, (<=) (>=) (>)
  - `max min

- **Num**
  - `+(→) (*)
  - `negate abs signum fromInteger

- **Ix**
  - `range index inRange rangeSize

- **Real**
  - `toRational recip fromRational fromDouble

- **Enum**
  - `suc pred toEnum fromEnum enumFrom enumFromThen enumFromTo enumFromThenTo

- **Integral**
  - `quot rem div mod quotRem divMod even odd toInteger

- **Show**
  - `showsPrec show showList

- **ShowPrec**
  - `mempty mconcat

- **Functor**
  - `fmap

- **Applicative**
  - `pure (<**>)

- **Monad**
  - `(>>)=) (>>) return fail

- **Arrow**
  - `pure (>>>)


- **Floating**
  - `pi exp log sqrt (***) logBase sin cos tan sinh cosh tanh

- **Integral**
  - `quotRem divMod even odd toInteger

- **Read**
  - `readsPrec readList

- **ReadPrec**
  - `readList

- **Functor**
  - `fmap

- **Applicative**
  - `pure (<**>)

- **Monad**
  - `(>>)=) (>>) return fail

- **Arrow**
  - `pure (>>>)


References:

Fethi Rabhi, Guy Lalalme. *Algorithms*. Addison-Wesley, 1999, Figure 2.4, p.46 (extended)
Type Classes and Type Class Instances

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

```
Eq
  All except IO, (→)
Ord
  All except (→), IO, IOError
Num
  Int, Integer, Float, Double
Show
  All Prelude Types
Real
  Int, Integer, Float, Double
RealFrac
  Float, Double
Integral
  Int, Integer
Enum
  (), Bool, Char, Ordering, Int, Integer, Float, Double
Ix
  Int, Integer, Char, Bool, Tuples of Ix types
Fractional
  Float, Double
Eq ⏅ Show ⏅ Monoid
  [], Ordering
Functor
  IO, [], Maybe
Monad
  IO, [], Maybe
MonadPlus
  IO, [], Maybe
Applicative
  IO, [], Maybe, ((→) d)
MonoidPlus
  IO, [], Maybe
Monoid
  All Prelude Types
Arrow
  (→)
Eq ⏅ Show ⏅ Monoid ⏅ Functor ⏅ Applicative ⏅ MonoidPlus ⏅ Arrow
```

Paul Hudak. The Haskell School of Expression. Cambridge University Press, 2000, p.156 (extended)
### '98 Type Class Memberships of Selected Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Instance of</th>
<th>Derivation</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>Read</td>
<td>Eq Ord Enum Bounded</td>
</tr>
<tr>
<td>[a]</td>
<td>Read</td>
<td>Eq Ord</td>
</tr>
<tr>
<td>[]</td>
<td>Functor Applicative, Monad, MonadPlus</td>
<td></td>
</tr>
<tr>
<td>(a,b)</td>
<td>Read</td>
<td>Eq Ord Bounded</td>
</tr>
<tr>
<td>((-&gt;) d)</td>
<td>Functor Applicative</td>
<td></td>
</tr>
<tr>
<td>(-&gt;)</td>
<td>Arrow</td>
<td></td>
</tr>
<tr>
<td>Array</td>
<td>Functor, Eq Ord, Read</td>
<td>Eq Ord Enum Read Bounded</td>
</tr>
<tr>
<td>Bool</td>
<td></td>
<td>Eq Ord Enum Read Bounded</td>
</tr>
<tr>
<td>Char</td>
<td></td>
<td>Eq Ord Enum Read Bounded</td>
</tr>
<tr>
<td>Complex</td>
<td>Floating, Read</td>
<td></td>
</tr>
<tr>
<td>Double</td>
<td>RealFloat, Read</td>
<td></td>
</tr>
<tr>
<td>Either</td>
<td></td>
<td>Eq Ord Read</td>
</tr>
<tr>
<td>Float</td>
<td>RealFloat</td>
<td></td>
</tr>
<tr>
<td>Int</td>
<td>Integral, Bounded Ix, Read</td>
<td></td>
</tr>
<tr>
<td>Integer</td>
<td>Integral, Ix, Read</td>
<td></td>
</tr>
<tr>
<td>IO</td>
<td>Functor Applicative, Monad, MonadPlus</td>
<td></td>
</tr>
<tr>
<td>IOError</td>
<td>Eq</td>
<td></td>
</tr>
<tr>
<td>Maybe</td>
<td>Functor Applicative, Monad, MonadPlus</td>
<td>Eq Ord Read</td>
</tr>
<tr>
<td>Ordering</td>
<td>Monoid</td>
<td>Eq Ord Enum Read Bounded</td>
</tr>
<tr>
<td>Ratio</td>
<td>RealFrac, Read</td>
<td></td>
</tr>
</tbody>
</table>

Fethi Rabhi, Guy Lapalme. *Algorithms.* Addison–Wesley, 1999, Table 2.4, p. 47 (extended)
Update on the Haskell Type Class Hierarchy

...Haskell is a research vehicle and, hence, a moving target:

Haskell’98

Functor
- fmap

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

Applicative
- pure
- (<$>)

Arrow
- first

Haskell’98 Onwards

Functor
- fmap

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

Applicative
- pure
- (<$>)

Arrow
- first

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

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

where 'arr' is a two-ary type variable

...for more information, check out:

https://wiki.haskell.org/Typeclassopedia
Chapter 12.4
References, Further Reading
Chapter 12: Further Reading


Es ist nicht genug zu wissen, 
man muss auch anwenden.

Johann Wolfgang von Goethe (1749-1832)
dt. Dichter und Naturforscher

Part V
Applications
Chapter 13
Parsing
Parsing: Lexical and Syntactical Analysis

Parsing

- a common term for the \textit{lexical and syntactical analysis} of the structure of text, e.g., \textit{source code text of programs}.
- \textit{an(other) application often used for demonstrating the power and elegance of functional programming.}
- enjoys a long history, see e.g.

as an example of an early text book concerned with parsing.
Functional Approaches for Parsing

...two different but conceptually related approaches are:

- Combinator parsing

- Monadic parsing

which are both well-suited for building recursive descent parsers.
Chapter 13.1
Motivation
Informally

...the parsing problem is as follows:

- Read a sequence of objects of some type \( a \).
- Yield an object or a sequence of objects of some type \( b \).

Example: Reading a sequence of objects of type Char:

\[
\langle \text{if n mod = 0 then 2*n else 2*n+1 fi} \rangle
\]

Yielding a sequence of objects of (enriched symbol) tokens:

\[
\langle \text{if}_\text{symb},",",(\text{var}_\text{symb},"n"),(\text{op}_\text{symb},"mod"),
(\text{rel}_\text{symb},"="),(\text{cst}_\text{symb},"0"),(\text{then}_\text{symb},""),(\text{cst}_\text{symb},"2"),(\text{op}_\text{symb},"*"),(\text{var}_\text{symb},"n"),
(\text{else}_\text{symb},""),\ldots,(\text{fi}_\text{symb},"")) \rangle
\]
Parsing Arithmetic Expressions

...a more complex parsing problem: Write a parser \( p \), which

- reads a string \( s \) representing a well-formed arithmetic expression (e.g., \( s = "((2+b)*5)" \))
- yields the value of type \( \text{Exp} \) represented by \( s \) with:

```haskell
data Exp = Lit Int | Var Char | Op Ops Exp Exp
data Ops = Add | Sub | Mul | Div | Mod
```

Applied to string "((2+b)*5)", e.g., parser \( p \) shall deliver the \( \text{Exp} \)-value:

\[
\text{Op Mul (Op Add (Lit 2) (Var 'b')) (Lit 5)}
\]

Note: \( p \) can be considered the reverse of the \texttt{show} function. It is also similar to the automatically derived \texttt{read} function for \texttt{Expr}: \( p \) and \texttt{read}, however, differ in the arguments they accept: strings of the compact form "((2+b)*5)" vs. strings of the form "Op Mul (Add (Lit 2) (Var 'b')) (Lit 5)".
Towards the Type of Parser Functions (1)

...characterizing parsing as the

- reading of sequences \( s \) of objects of some type \( a \)
- yielding objects or lists of objects of some type \( b \)

suggests naively for the type of parser functions:

\[
\text{type } \text{NaiveParse } a \ b = \ [a] \rightarrow b
\]

This, however, raises some questions. Assume, \texttt{bracket} and \texttt{number} are parser functions for detecting brackets and numbers, respectively:

<table>
<thead>
<tr>
<th>Parser</th>
<th>Input</th>
<th>What shall be the output?</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{bracket}</td>
<td>&quot;(xyz&quot; →&gt;</td>
<td>‘(‘? If so, what to do w/ &quot;xyz&quot;?</td>
</tr>
<tr>
<td>\texttt{number}</td>
<td>&quot;234&quot; →&gt;</td>
<td>2? Or 23? Or 234?</td>
</tr>
<tr>
<td>\texttt{bracket}</td>
<td>&quot;234&quot; →&gt;</td>
<td>No result? Failure?</td>
</tr>
</tbody>
</table>
Towards the Type of a Parser Function (2)

In detail: How shall a parser function behave if

- the input is not completely read?
- there are multiple results?
- there is a failure?

Answering the latter two questions first suggests to refine the type of parser functions to:

```
type RefinedParse a b = [a] -> [b]
```

which allows the following parsing output for the previous example:

<table>
<thead>
<tr>
<th>Parser</th>
<th>Input</th>
<th>Expected Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>bracket</td>
<td>&quot;(xyz&quot;</td>
<td>[‘(‘]</td>
</tr>
<tr>
<td>number</td>
<td>&quot;234&quot;</td>
<td>[2,23,234]</td>
</tr>
<tr>
<td>bracket</td>
<td>&quot;234&quot;</td>
<td>[]</td>
</tr>
</tbody>
</table>
Towards the Type of a Parser Function (3)

The first question, however, has still to be answered:

▶ What shall a parser function do with the part of the input that has not been read?

Answering it leads to the definite definition of the type of parser functions:

\[
\text{type } \text{Parse} \ a \ b = [a] \to [(b, [a])]\]

...enabling the parsing output:

<table>
<thead>
<tr>
<th>Parser</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>bracket</td>
<td>&quot;(xyz&quot;</td>
<td>[&quot;('&quot;, &quot;xyz&quot;)&quot;]</td>
</tr>
<tr>
<td>number</td>
<td>&quot;234&quot;</td>
<td>[(2, &quot;34&quot;), (23, &quot;4&quot;), (234, &quot;&quot;) ]</td>
</tr>
<tr>
<td>bracket</td>
<td>&quot;234&quot;</td>
<td>[]</td>
</tr>
</tbody>
</table>
Intuitively

...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 still needs to be analyzed.

Note, using lists for enabling the delivery of multiple results

▶ is known as the so-called list of successes technique (Philip Wadler, 1985).
▶ enables parsers to analyze also ambiguous grammars.
Note

...the following presentation is based on:


Chapter 13.2
Combinator Parsing
Objective

...developing a combinator library for parsing composed of

▶ Four primitive parser functions
  ▶ Two of which are input-independent (none, succeed)
  ▶ Two of which are input-dependent (token, spot)

▶ Three parser combinators for
  ▶ Alternatives (alt)
  ▶ Sequencing ((>>>)
  ▶ 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 13.2.1
Primitive Parsers
The two Input-independent Primitive Parsers

Recall:

\[
\text{type } \text{Parse } a \ b = \left[ a \right] \rightarrow \left[ (b, [a]) \right]
\]

- **none**, the always failing parser
  \[
  \text{none} :: \text{Parse } a \ b
  \text{none } _ = \left[ \right]
  \]

- **succeed**, the always succeeding parser
  \[
  \text{succeed} :: b \rightarrow \text{Parse } a \ b
  \text{succeed } \text{val} \ \text{inp} = \left[ (\text{val}, \text{inp}) \right]
  \]

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 $\varepsilon$ representing the empty word.
The two Input-dependent Primitive Parsers

- **token**, the parser recognizing single objects (so-called tokens):

  \[
  \text{token} :: \text{Eq a} \Rightarrow a \rightarrow \text{Parse a a} \\
  \text{token } t (x:xs) = \begin{cases} \\
  [(t,xs)] & | t == x \\
  [] & | \text{otherwise} \\
  \end{cases} \\
  \text{token } t [] = []
  \]

- **spot**, the parser recognizing single objects enjoying some property:

  \[
  \text{spot} :: \text{(a \rightarrow Bool)} \rightarrow \text{Parse a a} \\
  \text{spot } p (x:xs) = \begin{cases} \\
  [(x,xs)] & | p x \\
  [] & | \text{otherwise} \\
  \end{cases} \\
  \text{spot } p [] = []
  \]
Example: Using the Primitive Parsers

...for constructing parsers for simple parsing problems:

\[
\begin{align*}
\text{bracket} &= \text{token} \ ('(' \\
\text{dig} &= \text{spot \ isDigit} \\
\text{isDigit} &: \text{Char} \rightarrow \text{Bool} \\
\text{isDigit \ ch} &= ('0' \leq ch) \&\& (ch \leq '9')
\end{align*}
\]

Note: The parser functions \text{token} and \text{bracket} could also be defined using \text{spot}:

\[
\begin{align*}
\text{token} &: \text{Eq \ a} \Rightarrow \text{a} \rightarrow \text{Parse \ a \ a} \\
\text{token \ t} &= \text{spot \ (== \ t)} \\
\text{bracket} &: \text{Char} \rightarrow \text{Parse \ Char \ Char} \\
\text{bracket} &= \text{spot \ (== \ '(':)}
\end{align*}
\]
Chapter 13.2.2
Parser Combinators
...to write more complex and powerful parser functions, we need in addition to primitive parsers
down with parser-combining functions (or parser combinators)
which are re-usable higher-order polymorphic functions.
The Parser Combinator for Comb. Alternatives

Combining parsers as alternatives:

- alt, the parser combining parsers as alternatives:
  
  ```haskell
  alt :: Parse a b -> Parse a b -> Parse a b
  alt p1 p2 inp = p1 inp ++ p2 inp
  ```

  Intuitively: `alt` combines the results of the parses of `p1` and `p2`. The success of either of them is a success of the combination.

Example:

```plaintext
(bracket 'alt' dig) "234" ->> [] ++ [(2,"34")]
  ->> [(2,"34")]
```

More generally, an expression is either a literal, or a variable or an operator expression:

```plaintext
(lit 'alt' var 'alt' opexp) "(234+7)" ->> ...
```
The Parser Combinator for Sequential Comp.

Combining parsers sequentially:

▶ \((\star \star)\), the parser combining parsers sequentially:

\[
\text{infixr 5} \ \star \star \\
(\star \star) :: \text{Parse a b} -> \text{Parse a c} -> \text{Parse a (b,c)} \\
(\star \star) \ p1 \ p2 \ \text{inp} \\
\quad = [[[y,z],\text{rem2}] \mid (y,\text{rem1}) <- p1 \ \text{inp}, \ (z,\text{rem2}) <- p2 \ \text{rem1}] 
\]

Intuitively

▶ The values \((y,\text{rem1})\) run through the results of parser \(p1\) applied to \(\text{inp}\). For each of them, parser \(p2\) is applied to \(\text{rem1}\), the unconsumed part of the input by \(p1\) in that particular case. The results of the two successful parses, \(y\) and \(z\), are returned as a pair.

E.g., an operator expression starts with a bracket (detected by parser \(\text{bracket}\)) followed by a number (detected by parser \(\text{number}\)).
Example for 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(":

\[
\begin{align*}
\text{(number} & \text{ >***} \text{bracket) } "24("
\implies [((y,z),\text{rem2}) | (y,\text{rem1}) \leftarrow [(2,"4"), (24,"(")], \\
& (z,\text{rem2}) \leftarrow \text{bracket rem1 }] \\
\implies [((2,z),\text{rem2}) | (z,\text{rem2}) \leftarrow \text{bracket } "4(" ] ++ \\
& [((24,z),\text{rem2}) | (z,\text{rem2}) \leftarrow \text{bracket } "(" ] \\
\implies [((24,z),\text{rem2}) | (z,\text{rem2}) \leftarrow \text{bracket } "(" ] \\
\implies [((24,z),\text{rem2}) | (z,\text{rem2}) \leftarrow [\langle '\(',"\"\rangle]] \\
\implies [((24,\langle '\(',"\"\rangle]]
\end{align*}
\]
The Parser Combinator for Transformations

Combining a parser with a map transforming the parse results:

- **build**, the parser transforming obtained parse results:
  
  ```haskell
  build :: Parse a b -> (b -> c) -> Parse a c
  build p f inp = [(f x,rem) | (x,rem) <- p inp]
  ```

**Intuitively**: The map argument `f` of `build` transforms the items returned by its parser argument: It builds something from it.
Example for Transforming Parse Results

...the parser \texttt{digList} is assumed to return a \texttt{list} of \texttt{digit lists}, whose elements are transformed by \texttt{digsToNum} into the numbers whose values they represent:

\begin{verbatim}
(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")]
->> [(\texttt{2},"1a3"),(\texttt{21},"a3")]  
\end{verbatim}

\[
1168/1927
\]
Chapter 13.2.3

Universal Combinator Parser Basis
Universal Combinator Parser Basis

...together, the four primitive parsers

- none, succeed, token, and spot

and the three parser combinators

- alt, (>*>), and build

form a universal combinator parser basis, i.e., they allow us to build any parser we might be interested in.
The Universal Parser Basis at a Glance (1)

The priority of the sequencing operator:

\texttt{infixr 5 >>}

The type of parser functions:

\texttt{type Parse a b = [a] -> [(b,[a])]}

Two input-independent primitive parser functions:

\begin{itemize}
  \item The \textbf{always failing parser} function:
    \begin{verbatim}
    none :: Parse a b
    none _ = []
    \end{verbatim}
  \item The \textbf{always succeeding parser} function:
    \begin{verbatim}
    succeed :: b -> Parse a b
    succeed val inp = [(val,inp)]
    \end{verbatim}
\end{itemize}
The Universal Parser Basis at a Glance (2)

Two input-dependent primitive parser functions:

- The parser for recognizing single objects:
  \[
  \text{token} :: \text{Eq} \ a \Rightarrow a \rightarrow \text{Parse} \ a \ a \\
  \text{token} \ t = \text{spot} \ (==t)
  \]

- The parser for recognizing single objects satisfying some property:
  \[
  \text{spot} :: (a \rightarrow \text{Bool}) \rightarrow \text{Parse} \ a \ a \\
  \text{spot} \ p \ (x:xs) \\
  \quad | \ p \ x \quad = \quad [(x,xs)] \\
  \quad | \ \text{otherwise} \quad = \quad [] \\
  \text{spot} \ p \ [] \quad = \quad []
  \]
The Universal Parser Basis at a Glance (3)

Three parser combinators:

- **Alternatives**
  
  `alt :: Parse a b -> Parse a b -> Parse a b`  
  `alt p1 p2 inp = p1 inp ++ p2 inp`

- **Sequencing**
  
  `(>*>) :: Parse a b -> Parse a c -> Parse a (b,c)`  
  `(>*>) p1 p2 inp = [((y,z),rem2) | (y,rem1) <- p1 inp,  
  (z,rem2) <- p2 rem1]`

- **Transformation**
  
  `build :: Parse a b -> (b -> c) -> Parse a c`  
  `build p f inp = [(f x, rem) | (x,rem) <- p inp]`
Chapter 13.2.4
Structure of Combinator Parsers
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  
alts :: 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 \dash \dash\text{see Exam. 2, Chap. 13.2.5}
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.
Chapter 13.2.5
Writing Combinator Parsers: Examples
Using the Parser Basis

...for constructing (more) complex parser functions.

A parser

- recognizing a list of objects (example 1).
- 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:

\[
\text{list :: Parse a b} \rightarrow \text{Parse a [b]}
\]

\[
\text{list p} = (\text{succeed []}) \ 'alt' \\
\quad ((p >*> \text{list p}) \ 'build' \ (\text{uncurry (,:)})
\]

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 String Expressions (1)

...back to the initial example: Parsing string expressions like "(234+\sim42)*b", we shall construct the corresponding value of the algebraic data type:

```haskell
data Expr = Lit Int | Var Char | Op Ops Expr Expr
data Ops = Add | Sub | Mul | Div | Mod
```

Parsing "(234+\sim42)*b", e.g., shall yield the \texttt{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, \sim89, etc., where \sim 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 String 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 String Expressions (3)

Parsing variable names:

```haskell
ameParse :: Parse Char Expr
nameParse = spot isName 'build' Var

isName :: Char -> Bool
isName x = ('a' <= x && x <= 'z') -- A variable name
          -- must be a lower case character
```

Parsing literals (numerals):

```haskell
litParse :: Parse Char Expr
litParse
          -- A literal starts
          -- optionally with ‘~‘
          -- followed by a non-
          -- ‘build‘ (charlistToExpr . uncurry (++)
          -- empty
          -- list of digits
```

```haskell
= ((optional (token '~')) >>=
   (neList (spot isDigit))
   'build' (charlistToExpr . uncurry (++)
   )
```

Example 2: Parsing String Expressions (4)

Parsing fully bracketed operator expressions:

```haskell
optExpParse :: Parse Char Expr
opExpParse = (token '(' >>= parser >> opExpParse >> parser >>= token ')') -- A non-trivial expression
               >>= spot isOp >>= parser >> parser >>= token ')') -- must end with a closing bracket.

'build' makeExpr
```
Example 2: Parsing String Expressions (4)

...required supporting parser functions:

\[
\begin{align*}
\text{neList} & : \text{Parse } a \ b \rightarrow \text{Parse } a \ [b] \\
\text{optional} & : \text{Parse } a \ b \rightarrow \text{Parse } a \ [b]
\end{align*}
\]

where

- \text{neList } p \text{ recognizes a non-empty list of the objects recognized by } p.
- \text{optional } p \text{ recognizes an object recognized by } p \text{ or succeeds immediately.}

Note: \text{neList}, \text{optional}, and some other supporting functions including

\begin{align*}
\text{isOp} \\
\text{charlistToExpr}
\end{align*}

are still be defined, which is left here as homework.
Example 2: Parsing String Expressions (5)

...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:

```haskell
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)"` ->>
  
  $\text{Op Mul (Op Add (Lit 234) (Lit -42)) (Var 'b')}$
Chapter 13.3

Monadic Parsing
Monadic Parsing

...complements the concept of combining forms underlying combinator parsing with the one of monads.

For rendering this possible, the type of parser functions needs to be adjusted in order to make it a 1-ary type constructor which is eligible as an instance of type class Monad:

```haskell
newtype Parser a = Parse (String -> [(a,String)])
```

while re-using 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 13.3.1
Parser as Monads
Making Parser an Instance of Monad

Recalling the definition of type class Monad:

```haskell
class Monad m where
    (>>=) :: m a -> (a -> m b) -> m b -- (>>), failure are not
    return :: a -> m a -- not needed: Their de-
    -- fault implement. apply.
```

...Parser, a 1-ary type constructor, is made an instance of Monad as follows:

```haskell
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

```haskell
parse :: (Parser a) -> (String -> [(a,String)])
p.parse (Parse p) = p
```
Remarks on Parser as an Instance of Monad

```haskell
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 \(\text{return } a\) succeeds without consuming any of the argument string, and returns the single value \(a\).
- \(\text{parse}\) denotes a deconstructor function for parsers defined by \(\text{parse } (\text{Parse } p) = p\).
- The parser sequence \(p >>= f\) applies first parser \((\text{parse } p)\) 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 \((\text{parse } (f a))\) 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 Obligations for Parser as a Monad Inst.

...we can prove that Parser satisfies the monad laws and is thus a valid instance of Monad:

Lemma 13.3.1.1 (Monad Laws)

\[
\begin{align*}
\text{return } a & \implies f = f \ a \\
p & \implies \text{return} = p \\
p & \implies (\lambda a \rightarrow (f \ a \implies g)) = (p \implies (\lambda a \rightarrow f \ a)) \implies g
\end{align*}
\]

Note:

- \((\implies)\) being associative allows suppression of parentheses when parsers are applied sequentially.
- \text{return} being left-unit and right-unit for \((\implies)\) allows some parser definitions to be simplified.
Chapter 13.3.2

Parsers by Type Class Instantiations
Note

...having made Parser an instance of Monad provides us with two important parser functions, a primitive parser and a (monadic) parser combinator:

- return, the always succeeding parser
- (>>=), a combinator for sequentially combining parsers

which are the monadic counterparts of the parser combinators

- succeed
- (>>*)

of Chapter 13.2.1 and 13.2.2, respectively.

Making Parser an instance of MonadPlus will provide us with two further parser functions...
Making Parser an Instance of MonadPlus

...where `MonadPlus` is defined by (cf. Chapter 11.6):

```haskell
class Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

will provide us with the parser functions:

- `mzero`, the always failing parser
- `mplus` (via `(++)`), the parser for alternatives (or non-deterministic choice)

which are the monadic counterparts of the parser combinators

- `none`
- `alt`

of Chapter 13.2.1 and 13.2.2, respectively.
The Instance Decl. of Parser for MonadPlus

...yields the new parser functions \texttt{mzero} and \texttt{mplus}:

\begin{verbatim}
instance MonadPlus Parser where
    -- The \texttt{always failing} parser
    mzero = Parse (\cs -> [])

    -- The parser combinator for \texttt{alternatives}:
    p `mplus` q = Parse (\cs -> parse p cs ++ parse q cs)
\end{verbatim}

\textbf{Note}: \texttt{mplus} can yield more than one result; the value of
(\texttt{parse p cs ++ parse q cs}) can be a list of any length.
In this sense \texttt{mplus} is considered to explore parsers \texttt{altematively} (or, in this sense, \texttt{non-deterministically}).
Proof Obligat. for Parser as MonadPlus Inst.

...we can prove that Parser satisfies the MonadPlus laws:

Lemma 13.3.2.1 (MonadPlus Laws)

\[ p >>= (\_ \rightarrow \text{mzero}) = \text{mzero} \]
\[ \text{mzero} >>= p = \text{mzero} \]
\[ \text{mzero} \text{'} \text{mplus'} p = p \]
\[ p \text{'} \text{mplus'} \text{mzero} = p \]

Intuitively, 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 13.3.2.2**

\[
p \text{ 'mplus' } (q \text{ 'mplus' } r) = (p \text{ 'mplus' } q) \text{ 'mplus' } r
\]

\[
(p \text{ 'mplus' } q) \gg= f = (p \gg= f) \text{ 'mplus' } (q \gg= f)
\]

\[
p \gg= (\lambda a \rightarrow f a \text{ 'mplus' } g a) = (p \gg= f) \text{ 'mplus' } (p \gg= g)
\]

Intuitively, this means:

- mplus is associative.
- (\gg=) distributes through mplus.
Chapter 13.3.3
Universal Monadic Parser Basis
...in order to arrive at a universal monadic parser basis as in Chapter 13.2.3 we are left with defining monadic counterparts of the

- primitive parsers `token` and `spot`.
- parser combinator `build`.
The Monadic Counterpart of Parser spot

...parser sat recognizing single characters satisfying a given property:

\[
\text{sat} :: (\text{Char} \rightarrow \text{Bool}) \rightarrow \text{Parser} \text{ Char} \\
\text{sat} \ p = \\
\text{do} \ \{c <- \text{item}; \text{if } p \ c \ \text{then return } c \ \text{else zero}\}
\]

is the monadic counterpart of the parser function token of Chapter 13.2.1.
The Monadic Counterpart of Parser token

...parser `char` recognizing **single characters** defined in terms of parser `sat`:

```haskell
char :: Char -> Parser Char
char c = sat (== c)
```

is the **monadic** counterpart of the parser function `token` of Chapter 13.2.1.
The Universal Monadic Parser Basis (1)

The type of parser functions:

\[
\text{newtype } \text{Parser } a = \text{Parse} \ (\text{String} \rightarrow [(a, \text{String})])
\]

Two input-independent primitive parser functions:

- The always failing parser function:
  \[
  \text{mzero} :: \text{Parser } a \\
  \text{mzero} = \text{Parse} \ (\text{\textbackslash cs} \rightarrow [])
  \]

- The always succeeding parser function:
  \[
  \text{return} :: a \rightarrow \text{Parser } a \\
  \text{return } a = \text{Parse} \ (\text{\textbackslash cs} \rightarrow [(a, \text{cs})])
  \]
The Universal Monadic Parser Basis (2)

Two input-dependent primitive parser functions:

- The parser for recognizing single objects:
  ```haskell
  char :: Char -> Parser Char
  char c = sat (== c)
  ```

- The parser for recognizing single objects satisfying some property:
  ```haskell
  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:

- **Alternatives**

  \[
  \text{mplus :: Parser } a \rightarrow \text{ Parser } a \rightarrow \text{ Parser } a \]

  \[
  \text{p } \text{\textquoteleft mplus\textquoteleft } q =
  \text{Parse } (\text{\textbackslash cs } \rightarrow \text{ parse } p \text{ cs } ++ \text{ parse } q \text{ cs})
  \]

- **Sequencing**

  \[
  (\ggg \ggg) :: \text{Parser } a \rightarrow (a \rightarrow \text{Parser } b) \rightarrow \text{Parser } b
  \]

  \[
  \text{p } \ggg \ggg f =
  \text{Parse } (\text{\textbackslash cs } \rightarrow \text{ concat } [(\text{parse } (f a)) \text{ cs'} | \text{ (a,cs') } \leftarrow (\text{parse } p \text{ cs})])
  \]

- **Transformation**

  \[
  \text{mbuild :: Parser } a \rightarrow (a \rightarrow b) \rightarrow \text{Parser } b
  \]

  \[
  \text{mbuild } p \ f \ \text{inp } = \ldots \text{ (completion left as homework)}
  \]
Chapter 13.3.4

Utility Parsers
Utility Parsers (1)

Consuming the first character of an input string, if it is non-empty, and failing otherwise:

\[
\text{item} :: \text{Parser Char} \\
\text{item} = \text{Parse} (\text{\backslash cs} \rightarrow \text{case cs of} \\
\quad "" \rightarrow [] \\
\quad (c:cs) \rightarrow [(c,cs)] )
\]

Parsing a specific string:

\[
\text{string} :: \text{String} \rightarrow \text{Parser String} \\
\text{string} "" = \text{return} "" \\
\text{string} (c:cs) = \text{do char} c; \text{string} cs; \text{return} (c:cs)
\]
Utility Parsers (2)

The deterministically selecting parser:

\[
( +++ ) :: \text{Parser } a \to \text{Parser } a \to \text{Parser } a
\]

\[
p +++ q = \text{Parse } (\lambda cs \to \text{case parse } (p \ 'mplus' \ q) cs \text{ of } \\
\quad [] \to [] \\
\quad (x:xs) \to [x])
\]

Note:

- \( (+++) \) shows the same behavior as \texttt{mplus}, but yields at most one result (in this sense ‘deterministically’), whereas \texttt{mplus} can yield several ones (in this sense ‘non-deterministically’)
- \( (+++) \) satisfies all of the previously listed properties of \texttt{mplus}.  

Utility Parsers (3)

Applying a parser `p` repeatedly:

-- zero or more applications of `p`
\[
\text{many} :: \text{Parser} \ a \rightarrow \text{Parser} \ [a]
\text{many} \ p = \text{many1} \ p \ +++ \ \text{return} \ []
\]

-- one or more applications of `p`
\[
\text{many1} :: \text{Parser} \ a \rightarrow \text{Parser} \ [a]
\text{many1} \ p = \text{do} \ a \leftarrow p; \ as \leftarrow \text{many} \ p; \ \text{return} \ (a:as)
\]

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:

```haskell
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 \( \text{chainl} \) and \( \text{chainl1} \):

\[
\begin{align*}
\text{chainl} & : \text{Parser } a \rightarrow \text{Parser } (a \rightarrow a \rightarrow a) \\
& \rightarrow a \rightarrow \text{Parser } a \\
\text{chainl} \ p \ op \ a & = (p \ ' \text{chainl1}' \ op) +++ \text{return } a \\

\text{chainl1} & : \text{Parser } a \rightarrow \text{Parser } (a \rightarrow a \rightarrow a) \\
& \rightarrow \text{Parser } a \\
\( p \ ' \text{chainl1}' \ op \) & = \text{do } a \leftarrow p; \text{rest } a \\
& \quad \text{where rest } a = (\text{do } f \leftarrow op \\
& \quad \quad b \leftarrow p \\
& \quad \quad \text{rest } (f \ a \ b)) \\
& \quad +++ \text{return } a
\end{align*}
\]
Utility Parsers (6)

Handling white space, tabs, newlines, etc.

- Parsing a string with blanks, tabs, and newlines:
  ```hs
  space :: Parser String
  space = many (sat isSpace)
  ```

- Parsing a token by means of a parser p skipping any ‘trailing’ space:
  ```hs
  token :: Parser a -> Parser a
  token p = do {a <- p; space; return a}
  ```

- Parsing a symbolic token:
  ```hs
  symb :: String -> Parser String
  symb cs = token (string cs)
  ```

- Applying a parser p and throwing away any leading space:
  ```hs
  apply :: Parser a -> String -> [(a, String)]
  apply p = parse (do {space; p})
  ```
...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 13.3.5
Structure of a Monadic Parser
The Typical Structure of a Monadic Parser

...using the sequencing operator (>>=) or the syntactically sugared do-notation:

\[ p_1 >>= \lambda a_1 \rightarrow \text{do } a_1 \leftarrow p_1 \]
\[ p_2 >>= \lambda a_2 \rightarrow a_2 \leftarrow p_2 \]
\[ \ldots \]
\[ p_n >>= \lambda a_n \rightarrow a_n \leftarrow p_n \]
\[ f \ a_1 \ a_2 \ldots a_n \]

...the latter one equivalently expressed in just one line, if so desired:

\[ \text{do } \{ a_1 \leftarrow p_1; a_2 \leftarrow p_2; \ldots; a_n \leftarrow p_n; f\ a_1\ a_2\ldots a_n \} \]

Recall: The expressions \( a_i \leftarrow p_i \) are called generators (since they generate values for the variables \( a_i \)). Generators of the form \( a_i \leftarrow p_i \) can be replaced by \( p_i \), if the generated value will not be used afterwards.
...the intuitive, natural operational reading of such a monadic parser:

- Apply parser $p_1$ and call its result value $a_1$.
- Apply subsequently parser $p_2$ and call its result value $a_2$.
- ...
- Apply subsequently parser $p_n$ and call its result value $a_n$.
- Combine finally the intermediate results by applying an appropriate function $f$.

Note, most typically $f = \text{return} (g \ a_1 \ a_2 \ldots \ a_n)$; for an exception see parser $\text{chainl1}$ in Chapter 13.3.4, which needs to parse ‘more of the argument string’ before it can return a result.
Chapter 13.3.6

Writing Monadic Parsers: Examples
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:

\[
p :: Parser (Char,Char) \\
p = \text{do} \ c \gets \text{item}; \ \text{item}; \ d \gets \text{item}; \ \text{return} \ (c,d)
\]
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 \texttt{chainl1} combinator of Chapter 13.3.4 to implement the left-recursive production rules for \texttt{expr} and \texttt{term}.
Example 2: Parsing Arithm. Expressions (3)

The implementation of the parser `expr`:

```haskell
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)}
```

1220/1927
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 13.4

Summary
In conclusion

...non-monadic 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:

<table>
<thead>
<tr>
<th></th>
<th>Combinator Parsing</th>
<th>Monadic Parsing</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Primitive Parsers</strong></td>
<td>none</td>
<td>mzero</td>
</tr>
<tr>
<td></td>
<td>succeed</td>
<td>return</td>
</tr>
<tr>
<td></td>
<td>token</td>
<td>char</td>
</tr>
<tr>
<td></td>
<td>spot</td>
<td>sat</td>
</tr>
<tr>
<td><strong>Parser Combinators</strong></td>
<td>alt</td>
<td>mplus</td>
</tr>
<tr>
<td></td>
<td>(&gt;**)</td>
<td>(&gt;&gt;=)</td>
</tr>
<tr>
<td></td>
<td>build</td>
<td>mbuild</td>
</tr>
</tbody>
</table>
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 13.5
References, Further Reading
Chapter 13.2: Further Reading (1)


Jan van Eijck, Christina Unger. *Computational Semantics with Functional Programming*. Cambridge University Press, 2010. (Chapter 9, Parsing)

Chapter 13.2: Further Reading (2)


Chapter 13.2: Further Reading (3)


Chapter 13.2: Further Reading (4)


Chapter 13.2: Further Reading (5)


Chapter 13.3: Further Reading (6)

- Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer Verlag, 2011. (Kapitel 19.10.5, \(\lambda\)-Parser)


Chapter 13.3: Further Reading (7)


[legacy.cs.uu.nl/daan/parsec.html](http://legacy.cs.uu.nl/daan/parsec.html)

Chapter 13.3: Further Reading (8)


Chapter 14
Logic Programming Functionally
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


...part of it is summarized in Chapter 14.1.
Chapter 14.1

Motivation
Chapter 14.1.1

On the Evolution of Programming Languages
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 14.1.2

Functional vs. Logic Languages
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


Functional Logic Languages (2)

➤ **TOY**

Francisco J. López-Fraguas, Jaime Sánchez-Hernández. 

➤ **Mercury**

Zoltan Somogyi, Fergus Henderson, Thomas Conway. 

See also: [The Mercury Programming Language](http://www.mercurylang.org)
Chapter 14.1.2
A Curry Appetizer
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

```haskell
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`:
  \[
  \text{sem } \text{abstar} \rightarrow \text{non-deterministically} \Rightarrow \text{"a", "ab", "abb"} \\
  \text{where } \text{abstar} = \text{Conc (Lit 'a') (Star (Lit 'b'))}
  \]

- Checking whether some word `w` is in the language of a regular expression `re`:
  \[
  \text{sem } \text{re} =:= \text{w}
  \]

- Checking whether some string `s` contains a word generated by a regular expression `re` (similar to Unix's grep utility):
  \[
  \text{x} \text{s ++ sem } \text{re} ++ \text{y} \text{s =:= s} \\
  \text{Note: } \text{x} \text{s and } \text{y} \text{s are free!}
  \]
Chapter 14.1.4

Outline
...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.
...we follow the last approach as proposed by Michael Spivey and Silvija Seres in:


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 14.2
The Combinator Approach
Chapter 14.2.1

Introduction
Three Key Problems

...are to be solved in the course of developing this approach:

Modelling

1. logic programs yielding (possibly) multiple answers
   $\leadsto$ using the lists of successes technique
2. the evaluation/search strategy inherent to logic programs
   $\leadsto$ encapsulating the search strategy in ‘search monads’
3. logical variables (no distinction between input and output variables)
   $\leadsto$ 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 \(a \to b\) can be replaced by a function of type \(a \to [b]\).
- 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.:

<table>
<thead>
<tr>
<th>Integer</th>
<th>Factor pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>(1,24), (2,12), (3,8), (4,6), ..., (24,1)</td>
</tr>
</tbody>
</table>

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:
  - \( r \times s = n \Rightarrow r \leq n \land s \leq n \)

  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:

\[ [1..] \times [1..] \]
Illustrating the Search Space $[1..] \times [1..]$
Back to the Running Example

...adapting function \texttt{factor} straightforward to the infinite search space \([1..]\times[1..]\) yields:

\begin{verbatim}
  factor :: Int \rightarrow [(Int,Int)]
  factor n = [(r,s) | r<-\([1..]\), s<-\([1..]\), r*s == n]
\end{verbatim}

Applying \texttt{factor} to the argument \(24\) yields:

\begin{verbatim}
  factor 24
  \rightarrow [\((1,24)\)
\end{verbatim}

...followed by an infinite wait.

This is \texttt{useless} and of no practical value!
## The Problem: Unfair Depth-first Search

...the two-dimensional space is searched in a **depth-first order**:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(1,1)</td>
<td>(1,2)</td>
<td>(1,3)</td>
<td>(1,4)</td>
<td>(1,5)</td>
<td>(1,6)</td>
<td>(1,7)</td>
<td>(1,8)</td>
<td>(1,9)</td>
<td>...</td>
</tr>
<tr>
<td>2</td>
<td>(2,1)</td>
<td>(2,2)</td>
<td>(2,3)</td>
<td>(2,4)</td>
<td>(2,5)</td>
<td>(2,6)</td>
<td>(2,7)</td>
<td>(2,8)</td>
<td>(2,9)</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>(3,1)</td>
<td>(3,2)</td>
<td>(3,3)</td>
<td>(3,4)</td>
<td>(3,5)</td>
<td>(3,6)</td>
<td>(3,7)</td>
<td>(3,8)</td>
<td>(3,9)</td>
<td>...</td>
</tr>
<tr>
<td>4</td>
<td>(4,1)</td>
<td>(4,2)</td>
<td>(4,3)</td>
<td>(4,4)</td>
<td>(4,5)</td>
<td>(4,6)</td>
<td>(4,7)</td>
<td>(4,8)</td>
<td>(4,9)</td>
<td>...</td>
</tr>
<tr>
<td>5</td>
<td>(5,1)</td>
<td>(5,2)</td>
<td>(5,3)</td>
<td>(5,4)</td>
<td>(5,5)</td>
<td>(5,6)</td>
<td>(5,7)</td>
<td>(5,8)</td>
<td>(5,9)</td>
<td>...</td>
</tr>
<tr>
<td>6</td>
<td>(6,1)</td>
<td>(6,2)</td>
<td>(6,3)</td>
<td>(6,4)</td>
<td>(6,5)</td>
<td>(6,6)</td>
<td>(6,7)</td>
<td>(6,8)</td>
<td>(6,9)</td>
<td>...</td>
</tr>
<tr>
<td>7</td>
<td>(7,1)</td>
<td>(7,2)</td>
<td>(7,3)</td>
<td>(7,4)</td>
<td>(7,5)</td>
<td>(7,6)</td>
<td>(7,7)</td>
<td>(7,8)</td>
<td>(7,9)</td>
<td>...</td>
</tr>
<tr>
<td>8</td>
<td>(8,1)</td>
<td>(8,2)</td>
<td>(8,3)</td>
<td>(8,4)</td>
<td>(8,5)</td>
<td>(8,6)</td>
<td>(8,7)</td>
<td>(8,8)</td>
<td>(8,9)</td>
<td>...</td>
</tr>
<tr>
<td>9</td>
<td>(9,1)</td>
<td>(9,2)</td>
<td>(9,3)</td>
<td>(9,4)</td>
<td>(9,5)</td>
<td>(9,6)</td>
<td>(9,7)</td>
<td>(9,8)</td>
<td>(9,9)</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

This **search order is unfair**: Pairs in **rows 2 onwards** will never be reached and considered for being a factor pair.
Chapter 14.2.2

Diagonalization
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:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(1,1)</td>
<td>(1,2)</td>
<td>(1,3)</td>
<td>(1,4)</td>
<td>(1,5)</td>
<td>(1,6)</td>
<td>(1,7)</td>
<td>(1,8)</td>
<td>(1,9)</td>
<td>...</td>
</tr>
<tr>
<td>2</td>
<td>(2,1)</td>
<td>(2,2)</td>
<td>(2,3)</td>
<td>(2,4)</td>
<td>(2,5)</td>
<td>(2,6)</td>
<td>(2,7)</td>
<td>(2,8)</td>
<td>(2,9)</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>(3,1)</td>
<td>(3,2)</td>
<td>(3,3)</td>
<td>(3,4)</td>
<td>(3,5)</td>
<td>(3,6)</td>
<td>(3,7)</td>
<td>(3,8)</td>
<td>(3,9)</td>
<td>...</td>
</tr>
<tr>
<td>4</td>
<td>(4,1)</td>
<td>(4,2)</td>
<td>(4,3)</td>
<td>(4,4)</td>
<td>(4,5)</td>
<td>(4,6)</td>
<td>(4,7)</td>
<td>(4,8)</td>
<td>(4,9)</td>
<td>...</td>
</tr>
<tr>
<td>5</td>
<td>(5,1)</td>
<td>(5,2)</td>
<td>(5,3)</td>
<td>(5,4)</td>
<td>(5,5)</td>
<td>(5,6)</td>
<td>(5,7)</td>
<td>(5,8)</td>
<td>(5,9)</td>
<td>...</td>
</tr>
<tr>
<td>6</td>
<td>(6,1)</td>
<td>(6,2)</td>
<td>(6,3)</td>
<td>(6,4)</td>
<td>(6,5)</td>
<td>(6,6)</td>
<td>(6,7)</td>
<td>(6,8)</td>
<td>(6,9)</td>
<td>...</td>
</tr>
<tr>
<td>7</td>
<td>(7,1)</td>
<td>(7,2)</td>
<td>(7,3)</td>
<td>(7,4)</td>
<td>(7,5)</td>
<td>(7,6)</td>
<td>(7,7)</td>
<td>(7,8)</td>
<td>(7,9)</td>
<td>...</td>
</tr>
<tr>
<td>8</td>
<td>(8,1)</td>
<td>(8,2)</td>
<td>(8,3)</td>
<td>(8,4)</td>
<td>(8,5)</td>
<td>(8,6)</td>
<td>(8,7)</td>
<td>(8,8)</td>
<td>(8,9)</td>
<td>...</td>
</tr>
<tr>
<td>9</td>
<td>(9,1)</td>
<td>(9,2)</td>
<td>(9,3)</td>
<td>(9,4)</td>
<td>(9,5)</td>
<td>(9,6)</td>
<td>(9,7)</td>
<td>(9,8)</td>
<td>(9,9)</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- 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)]
- ...
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:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>((1,1))</td>
<td>((1,2))</td>
<td>((1,3))</td>
<td>((1,4))</td>
<td>((1,5))</td>
<td>((1,6))</td>
<td>((1,7))</td>
<td>...</td>
</tr>
<tr>
<td>2</td>
<td>((2,1))</td>
<td>((2,2))</td>
<td>((2,3))</td>
<td>((2,4))</td>
<td>((2,5))</td>
<td>((2,6))</td>
<td>((2,7))</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>((3,1))</td>
<td>((3,2))</td>
<td>((3,3))</td>
<td>((3,4))</td>
<td>((3,5))</td>
<td>((3,6))</td>
<td>((3,7))</td>
<td>...</td>
</tr>
<tr>
<td>4</td>
<td>((4,1))</td>
<td>((4,2))</td>
<td>((4,3))</td>
<td>((4,4))</td>
<td>((4,5))</td>
<td>((4,6))</td>
<td>((4,7))</td>
<td>...</td>
</tr>
<tr>
<td>5</td>
<td>((5,1))</td>
<td>((5,2))</td>
<td>((5,3))</td>
<td>((5,4))</td>
<td>((5,5))</td>
<td>((5,6))</td>
<td>((5,7))</td>
<td>...</td>
</tr>
<tr>
<td>6</td>
<td>((6,1))</td>
<td>((6,2))</td>
<td>((6,3))</td>
<td>((6,4))</td>
<td>((6,5))</td>
<td>((6,6))</td>
<td>((6,7))</td>
<td>...</td>
</tr>
<tr>
<td>7</td>
<td>((7,1))</td>
<td>((7,2))</td>
<td>((7,3))</td>
<td>((7,4))</td>
<td>((7,5))</td>
<td>((7,6))</td>
<td>((7,7))</td>
<td>...</td>
</tr>
<tr>
<td>8</td>
<td>((8,1))</td>
<td>((8,2))</td>
<td>((8,3))</td>
<td>((8,4))</td>
<td>((8,5))</td>
<td>((8,6))</td>
<td>((8,7))</td>
<td>...</td>
</tr>
<tr>
<td>9</td>
<td>((9,1))</td>
<td>((9,2))</td>
<td>((9,3))</td>
<td>((9,4))</td>
<td>((9,5))</td>
<td>((9,6))</td>
<td>((9,7))</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Homework

The previous figure illustrates that there is a bijective map

\[ m : \mathbb{IN} \rightarrow \mathbb{IN} \times \mathbb{IN} \]

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:

\[
\text{diagprod} :: [a] \to [b] \to [(a,b)]
\]
\[
\text{diagprod} \ \text{xs} \ \text{ys} =
\[(\text{xs}!!i, \text{ys}!!(n-i)) \mid n<-\{0..\}, \ i<-\{0..n\}\]
\]

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:

\[
[(1,1), (1,2), (2,1), (1,3), (2,2), (3,1), (1,4), (2,3), (3,2), (4,1), (1,5), (2,4), (3,3), (4,2), (5,1), (1,6), (2,5), \ldots, (6,1), (1,7), (2,6), \ldots (7,1), \ldots
\]
## Implementing Diagonalization (2)

The function `diagprod` is defined as:

```haskell
diagprod :: [a] -> [b] -> [(a,b)]
diagprod xs ys = [(xs!!i, ys!!(n-i)) | n<-[0..], i<-[0..n]]
```

The table below illustrates the diagonalization process:

<table>
<thead>
<tr>
<th>n</th>
<th>i</th>
<th>n-i</th>
<th>(xs!!i, ys!!(n-i))</th>
<th>([1..]!!i, [1..]!!(n-i))</th>
<th>#</th>
<th>Diag. #</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>(xs!!0,ys!!0)</td>
<td>(1,1)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>(xs!!0,ys!!1)</td>
<td>(1,2)</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>(xs!!1,ys!!0)</td>
<td>(2,1)</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>(xs!!0,ys!!2)</td>
<td>(1,3)</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>(xs!!1,ys!!1)</td>
<td>(2,2)</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td>(xs!!2,ys!!0)</td>
<td>(3,1)</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
<td>(xs!!0,ys!!3)</td>
<td>(1,4)</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>(xs!!1,ys!!2)</td>
<td>(2,3)</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>(xs!!2,ys!!1)</td>
<td>(3,2)</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0</td>
<td>(xs!!3,ys!!0)</td>
<td>(4,1)</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>4</td>
<td>(xs!!0,ys!!4)</td>
<td>(1,5)</td>
<td>11</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>3</td>
<td>(xs!!1,ys!!3)</td>
<td>(2,4)</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
<td>(xs!!2,ys!!2)</td>
<td>(3,3)</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
<td>(xs!!3,ys!!1)</td>
<td>(4,2)</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0</td>
<td>(xs!!4,ys!!0)</td>
<td>(5,1)</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
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:

\[
\text{factor} :: \text{Int} \to [(\text{Int}, \text{Int})] \\
\text{factor } n = [ (r, s) | (r, s) \leftarrow \text{diagprod } [1..] [1..], r \times s == n ] \\
\]

Applying now factor to the argument 24, we obtain:

\[
\text{factor } 24 \rightarrow [(4, 6), (6, 4), (3, 8), (8, 3), (2, 12), (12, 2), (1, 24), (24, 1)] \\
\]

...i.e., we obtain all results, followed by an infinite wait.

Of course, this is not surprising, since the search space is infinite.
Chapter 14.2.3

Diagonalization with Monads
Finite Lists, Infinite Streams, Monads

...in the following we conceptually distinguish between:

- \([a]\): Finite lists.
- \(\text{Stream } a\): Infinite lists defined as type alias by:
  
  \[
  \text{type Stream } a = [a]
  \]

Note: Distinguishing between \((\text{Stream } a)\) for infinite lists and \([a]\) for finite lists is conceptually and notationally only as is made explicit by defining \((\text{Stream } a)\) 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:

\[
\text{class Monad } m \text{ where }
\]

\[
\begin{align*}
\text{return } & : a \to m a \\
(\gg\gg=) & : m a \to (a \to m b) \to m b
\end{align*}
\]
Making Stream an Instance of 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 11.4.2:

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 \texttt{return} and \texttt{(>>=)} for lists and streams allow us to avoid or replace list comprehension:

\textit{E.g.}, the expression

\[
[(x, y) \mid x \leftarrow [1..], y \leftarrow [10..]]
\]

using list comprehension is equivalent to the expression

\[
[1..] \ggg (\langle x \rightarrow [10..] \ggg (\langle y \rightarrow \texttt{return} (x, y)))
\]

using monad operations; this is is made explicit by stepwise unfolding the monadic expression yielding first the equivalent expression:

\[
\text{concat} \ (\text{map} \ (\langle x \rightarrow [(x, y) \mid y \leftarrow [10..]])[1..])
\]

and second the equivalent expression:

\[
\text{concat} \ (\text{map} \ (\langle x \rightarrow \text{concat} \ (\text{map} \ (\langle y \rightarrow [(x, y)])[10..])))[1..])
\]
Notational Benefit (2)

By exploiting the general rule that

```haskell
  do x1 <- e1; x2 <- e2; ... ; xn <- en; e
```

is a shorthand for

```haskell
  e1 >>= (\x1 -> e2 >>= (\x2 -> ... >>= (\xn -> e)...))
```

...Haskell’s do-notation allows an even more compact equivalent representation:

```haskell
  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 (>>=)

...idea: Embedding diagonalization into (>>=).

To this end, we introduce a new polymorphic type \texttt{Diag}:

\[
\text{newtype } \texttt{Diag} \ a = \text{MkDiag} \ (\text{Stream} \ a) \ 
\text{deriving Show}
\]

together with a utility function for \texttt{stripping off} the data constructor \texttt{MkDiag}:

\[
\text{unDiag} :: \text{Diag} \ a \to a
\]

\[
\text{unDiag} \ (\text{MkDiag} \ \texttt{xs}) = \texttt{xs}
\]
Diagonalization with Monads

...making `Diag` an instance of the type constructor class `Monad`:

```haskell
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:

```haskell
diag :: Stream (Stream a) -> Stream [a]
diag [] = []
diag (xs:xss) =
    lzw (++) [ [x] | x <- xs] ([] : diag xss)
```
Making Diag an Instance of Monad

...using itself the utility function \( \text{lzw} \) (‘like zipWith.’):

\[
\text{lzw} :: (a \rightarrow a \rightarrow a) \rightarrow \text{Stream } a \rightarrow \text{Stream } a \rightarrow \text{Stream } a
\]

\[
\text{lzw } f \ [\] \ ys = ys
\]

\[
\text{lzw } f \ xs \ [\] = xs
\]

\[
\text{lzw } f \ (x:xs) \ (y:ys) = (f \ x \ y) : (\text{lzw } f \ xs \ ys)
\]

**Note:** \( \text{lzw} \) equals \( \text{zipWith} \) except that the non-empty remainder of a non-empty argument list is attached, if one of the argument lists gets empty.
...for monad \texttt{Diag} holds:

- \texttt{return} yields the singleton list.
- \texttt{undia} strips off the constructor added by the function \( f :: a \to \texttt{Diag} \ b \).
- \texttt{dia} arranges the elements of the list into a \textit{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 \textbf{Bunch}. 
Chapter 14.2.4
Filtering with Conditions
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:

```haskell
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 14.2.5 onwards.
Filtering with Conditions using test

Using `zero`, the function `test`, which might not look useful at first sight, yields the key for filtering:

```haskell
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:

```haskell
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:

\[
\begin{align*}
\text{do } & \quad \text{x} \leftarrow [1..]; \\
\quad & \quad :: \text{Int} \quad :: \text{[]} \text{Int} \\
\quad & \quad () \leftarrow \text{test} (\text{x} \mod 3 == 0); \\
\quad & \quad :: () \quad [()] :: \text{[]} () \text{, if true} \\
\quad & \quad \text{[]} :: \text{[]} () \text{, if false} \\
\text{return } & \quad \text{x} \\
\quad & \quad :: \text{[]} \text{Int}
\end{align*}
\]

...if \text{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 \text{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:

```haskell
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:

```haskell
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:

\[
\text{do } y \leftarrow \text{MkDiag } [1..]; \text{ test}(x*y==24); \text{ return } (x,y)
\]

Intuitively

- This expression yields for a given value of \( x \) all values of \( y \) with \( x \times y = 24 \).
- 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 \( 5 \times y = 24 \).
Why is that? (2)

...the deeper reason for this (undesired) behaviour:

The bind operation (>>=) of Diag is not associative, i.e.,

\[ \text{xm} >>= (\lambda x \to f \ x >>= g) = (\text{xm} >>= f) >>= g \]

...does not hold! Or, equivalently expressed using do:

\[
\begin{align*}
do \ x \leftarrow \text{xm}; \ y \leftarrow f \ x; \ g \ y \\
&= \text{xm} >>= (\lambda x \to f \ x >>= (\lambda y \to g \ y)) \\
&= \text{xm} >>= (\lambda x \to f \ x >>= g) \\
&= (\text{xm} >>= f) >>= g \\
&= (\text{xm} >>= (\lambda x \to f \ x)) >>= (\lambda y \to g \ y) \\
&= do \ y \leftarrow (do \ x \leftarrow \text{xm}; \ f \ x); \ g \ y
\end{align*}
\]

...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:

\[
\text{do (x,y) <- (do u <- MkDiag [1..]; v <- MkDiag [1..]; return (u,v))}
\text{test (x*y==24); return (x,y)}
\]

\[
\rightarrow> \text{MkDiag [(4,6),(6,4),(3,8),(8,3),(2,12),(12,2), (1,24),(24,1)}
\]

...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.
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 14.2.5
Indicating Search Progress
Indicating Search Progress

...to this end, we introduce a new type Matrix together with a cost-guided diagonalization search, a true breadth search.

Intuitively

- 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 \texttt{Matrix}:

\begin{verbatim}
  newtype Matrix a =
    MkMatrix (Stream [a]) deriving Show
\end{verbatim}

...and a utility function for \texttt{stripping off} the data constructor:

\begin{verbatim}
  unMatrix :: Matrix a -> a
  unMatrix (MkMatrix xm) = xm
\end{verbatim}
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:

```haskell
instance Monad Matrix where
  return x = MkMatrix [[x]]
  (MkMatrix xm) >>= f = MkMatrix (bindm xm (unMatrix . f))
```

```haskell
instance Bunch Matrix where
  zero = MkMatrix []
  alt (MkMatrix xm) (MkMatrix ym) =
    MkMatrix (lzw (++) xm ym)
  wrap (MkMatrix xm) =
    MkMatrix ([]:xm)
    -- ‘wrap xm’ yields a matrix w/
    -- 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:

```r
do r <- intMat; s <- intMat; test(r*s==24); return (r,s)
```

->> `MkMatrix` 
```r
[(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 $!(\cdot, \cdot)!$:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(1,1)</td>
<td>(1,2)</td>
<td>(1,3)</td>
<td>(1,4)</td>
<td>(1,5)</td>
<td>(1,6)</td>
<td>(1,7)</td>
<td>(1,8)</td>
<td>(1,9)</td>
</tr>
<tr>
<td>2</td>
<td>(2,1)</td>
<td>(2,2)</td>
<td>(2,3)</td>
<td>(2,4)</td>
<td>(2,5)</td>
<td>(2,6)</td>
<td>(2,7)</td>
<td>(2,8)</td>
<td>(2,9)</td>
</tr>
<tr>
<td>3</td>
<td>(3,1)</td>
<td>(3,2)</td>
<td>(3,3)</td>
<td>(3,4)</td>
<td>(3,5)</td>
<td>(3,6)</td>
<td>(3,7)</td>
<td>(3,8)</td>
<td>(3,9)</td>
</tr>
<tr>
<td>4</td>
<td>(4,1)</td>
<td>(4,2)</td>
<td>(4,3)</td>
<td>(4,4)</td>
<td>(4,5)</td>
<td>(4,6)</td>
<td>(4,7)</td>
<td>(4,8)</td>
<td>(4,9)</td>
</tr>
<tr>
<td>5</td>
<td>(5,1)</td>
<td>(5,2)</td>
<td>(5,3)</td>
<td>(5,4)</td>
<td>(5,5)</td>
<td>(5,6)</td>
<td>(5,7)</td>
<td>(5,8)</td>
<td>(5,9)</td>
</tr>
<tr>
<td>6</td>
<td>(6,1)</td>
<td>(6,2)</td>
<td>(6,3)</td>
<td>(6,4)</td>
<td>(6,5)</td>
<td>(6,6)</td>
<td>(6,7)</td>
<td>(6,8)</td>
<td>(6,9)</td>
</tr>
<tr>
<td>7</td>
<td>(7,1)</td>
<td>(7,2)</td>
<td>(7,3)</td>
<td>(7,4)</td>
<td>(7,5)</td>
<td>(7,6)</td>
<td>(7,7)</td>
<td>(7,8)</td>
<td>(7,9)</td>
</tr>
<tr>
<td>8</td>
<td>(8,1)</td>
<td>(8,2)</td>
<td>(8,3)</td>
<td>(8,4)</td>
<td>(8,5)</td>
<td>(8,6)</td>
<td>(8,7)</td>
<td>(8,8)</td>
<td>(8,9)</td>
</tr>
<tr>
<td>9</td>
<td>(9,1)</td>
<td>(9,2)</td>
<td>(9,3)</td>
<td>(9,4)</td>
<td>(9,5)</td>
<td>(9,6)</td>
<td>(9,7)</td>
<td>(9,8)</td>
<td>(9,9)</td>
</tr>
</tbody>
</table>

...
Chapter 14.2.6

Selecting a Search Strategy
An Array of Search Strategies

...is now at our disposal, namely

1. Depth search \([1..]\)
2. Diagonalization \((\text{MkDiag} \, [[n] \mid n \leftarrow [1..]])\)
3. Breadth search \((\text{MkMatrix} \, [[n] \mid n \leftarrow [1..]])\)

...and we can choose each of them at the very last moment, just by picking the right monad when calling a function:

```haskell
-- Picking the desired search strategy by choosing 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)
```

...is now at our disposal, namely
Picking a Search Strategy at Call Time

...specifying the result type of \texttt{factor} when calling it selects the search monad and thus the search strategy applied.

Illustrated in terms of our running example:

\begin{itemize}
\item \textbf{-- Depth Search: Picking Stream}
\texttt{factor 24 :: Stream (Int,Int)} \rightarrow [(1,24)]
\item \textbf{-- Diagonalization Search: Picking Diag}
\texttt{factor 24 :: Diag (Int, Int)} \rightarrow \texttt{MkDiag [(4,6),(6,4),(3,8),(8,3),(2,12),(12,2),(1,24),(24,1)]}
\item \textbf{-- Breadth Search w/ Progress Indication: Picking Matrix}
\texttt{factor 24 :: Matrix (Int, Int)} \rightarrow \texttt{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!
...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 14.2.7

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.
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:

\[
\text{unify} :: (\text{Term}, \text{Term}) \rightarrow \text{Subst} \rightarrow \text{Maybe Subst}
\]

\[
\text{unify} \ (t,u) \ s = \\
\text{case (deref } s \ t, \text{deref } s \ u) \text{ of }
\]

\[
(\text{Nil, Nil}) \rightarrow \text{Just } s \\
(\text{Cons } x \ xs, \text{Cons } y \ ys) \rightarrow \\
\quad \text{unify } (x,y) \ s \gg= \text{unify } (xs, ys)
\]

\[
(\text{Int } n, \text{Int } m) \mid (n==m) \rightarrow \text{Just } s \\\n(\text{Var } x, \text{Var } y) \mid (x==y) \rightarrow \text{Just } s \\
(\text{Var } x, t) \rightarrow \text{if occurs } x \ t \ s \text{ then Nothing} \text{ else Just (extend } x \ t \ s) \\
(t, \text{Var } x) \rightarrow \text{if occurs } x \ t \ s \text{ then Nothing} \text{ else Just (extend } x \ t \ s) \\
(\_,\_) \rightarrow \text{Nothing}
\]
Term Unification (2)

where

\[
\text{occurs} :: \text{Variable} \to \text{Term} \to \text{Subst} \to \text{Bool} \\
\text{occurs } x t s = \\
\quad \text{case } \text{deref } s t \text{ of} \\
\quad \quad \text{Var } y \quad \to \quad x == y \\
\quad \quad \text{Cons } y \ ys \quad \to \quad \text{occurs } x \ y \ s \ || \ \text{occurs } x \ ys \ s \\
\quad \quad _ \quad \to \quad \text{False}
\]
Predicates: Modelling Logic Programs (1)

...in our scenario \( m \) is of type \texttt{bunch}.

Logic programs are of type:

\[
\text{type Pred } m = \text{Answer} \rightarrow m \text{ Answer}
\]

...intuitively, applied to an \texttt{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:

\[
\text{newtype Answer = MkAnswer (Subst,Int)}
\]

...intuitively, the \texttt{substitution} carries the information about the values of variables; the \texttt{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:

```hs
initial :: Answer
initial = MkAnswer (idsubst, 0)
```

Logical program run: Predicate \( p \) as query is applied to the initial ‘input’ answer:

```hs
run :: Bunch m => Pred m -> m Answer
run p = p initial
```

Example: Choosing \texttt{Stream} for \( m \) allows evaluating the predicate \texttt{append} (defined later):

```hs
run (append (list [1,2],list [3,4],var "z"))
   :: Stream Answer
->> [{z=[1,2,3,4]}]   -- an appropriate show function is assumed
```
Chapter 14.2.8
Combinators for Logic Programs
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

1314/1927
Combinator (|||): Disjunction

...combinator (|||) allows us to connect predicates disjunctively, e.g.:

\[
\text{run (var "x" =:= Int 3 ||| var "x" =:= Int 4)} \\
\quad :\!\!: \text{Stream Answer} \\
\rightarrow \rightarrow \quad [{x=3, x=4}]
\]

Implementation of (|||) by means of the alt operation of monad bunch:

\[
(|||) :\!\!\!: \text{Bunch } m \Rightarrow \text{Pred } m \rightarrow \text{Pred } m \rightarrow \text{Pred } m \\
(p ||| q) s = \text{alt} (p s) (q s)
\]
Assigning Priorities to (=:=), (&&&), (|||)

...is done as follows:

```haskell
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

\[
\text{exists} :: \text{Bunch } m \Rightarrow (\text{Term} \rightarrow \text{Pred } m) \rightarrow \text{Pred } m
\]
\[
\text{exists } p \ (\text{MkAnswer} \ (s, n)) =
\]
\[
p \ (\text{Var} \ (\text{Generated} \ n)) \ (\text{MkAnswer} \ (s, n+1))
\]

Note:

- The term \( \text{exists} (\lambda x \rightarrow \ldots x \ldots) \) has the same meaning as the predicate \( \ldots x \ldots \) but with \( x \) denoting a fresh variable which is different from all the other variables used by the program; \( n+1 \) in \( \text{MkAnswer} \ (s, n+1) \) ensures that never the same variable is introduced by nested calls of \text{exists}.

- The function \text{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:

```haskell
step :: Bunch m => Pred m -> Pred m
step p s = wrap (p s)
```

Illustrating the usage and effect of step:

```haskell
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 14.2.9
Writing Logic Programs: Two Examples
Writing Logic Programs: Two Examples

We consider two examples:

Example 1: List Concatenation (1)

...implementing a predicate \texttt{append (a,b,c)}, where \(a, b\) denote lists and \(c\) the concatenation of \(a\) and \(b\).

The implementation of predicate \texttt{append}:

\begin{verbatim}
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))))))
\end{verbatim}
Example 1: List Concatenation (2)

...in more detail:

\[
\text{append} :: \text{Bunch } m \Rightarrow (\text{Term, Term, Term}) \rightarrow \text{Pred } m \\
\text{append } (p,q,r) = \\
\quad -- \text{Case 1} \\
\quad \text{step } (p =:= \text{Nil } &&& q =:= r \\
\quad \quad \quad \quad \quad \quad ||| \\
\quad \quad \quad \quad \quad \quad -- \text{Case 2} \\
\quad \quad \text{exists } (\lambda x \rightarrow \text{exists } (\lambda a \rightarrow \text{exists } (\lambda b \rightarrow \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{p } =:= \text{Cons } x \ a &&& \text{r } =:= \text{Cons } x \ b &&& \text{append } (a,q,b)))))
\]

Intuitively

- **Case 1**: If \(p\) is \(\text{Nil}\), then \(r\) must be the same as \(q\).
- **Case 2**: If \(p\) has the form \(\text{Cons } x \ a\), then \(r\) must have the form \(\text{Cons } x \ b\), 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 \text{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:

\[
\text{run (append (list [1,2], list [3,4], \text{var } \text{"z"}))}
\]

\[:: \text{Stream Answer} \]

\[->> [\{z=[1,2,3,4]\}] \]

-- 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:

\[->> \text{Cons 1 (Cons 2 (Cons 3 (Cons 4 Nil)))} \]
Example 1: List Concatenation (4)

Using `append` for computing the set of lists which equal a given list

b) ...when concatenated:

```plaintext
run (append (var "x", var "y", list [1,2,3])) :: Stream Answer
```

```plaintext
->> [{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:

```plaintext
run (append (var "x", list [2,3], list [1,2,3])) :: Stream Answer
```

```plaintext
->> [{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 14.2.9.1 (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 14.2.8.1(2) does not hold: the sequence $[11100] = [1]++[11]++[00]$, e.g., is bad.
Example 2: ‘Good’ Sequences (4)

The implementation of predicate good:

```haskell
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:

```prolog
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:

\[
\text{run (good (var "s"))} :: \text{Stream Answer} \\
\rightarrow [\{s=[0]\}, \{s=[1,0,0]\}, \{s=[1,0,1,0,0]\}, \{s=[1,0,1,0,1,0,0]\}, \ldots]
\]

...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]},
   {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.’
Remarks on Missing Code / Homework

Note, code for

- pretty printing terms and answers
- making the types `Term`, `Subst`, and `Answer` instances of the type class `Show`

is missing and must be provided by a user of the approach.
Chapter 14.3

Summary
Summing up

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 14.4

References, Further Reading
Chapter 14: Further Reading (1)


Chapter 14: Further Reading (2)


Chapter 14: Further Reading (3)

www.curry-language.org/  
Vers. 0.8.3, September 11, 2012:  
http://www.informatik.uni-kiel.de/~curry/report.html


Chapter 14: Further Reading (4)


Chapter 14: Further Reading (5)


Chapter 14: Further Reading (6)

Chapter 14: Further Reading (7)


Chapter 14: Further Reading (8)


Chapter 15
Pretty Printing
Chapter 15.1
Motivation
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 ifself 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 ifself ‘pretty.’
The Prettier Printer

...presented in this chapter has been proposed by Philip Wadler in:


which has been designed to improve (cf. *Chapter 15.5*) on a pretty printer proposed by John Hughes which is widely recognized as a standard:

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 $\text{Doc}$.

The

- **simple pretty printer** (cf. Chapter 15.2)
  - implements $\text{Doc}$ as strings.
  - supports for every document only **one possible layout**, in particular, no attempt is made to compress structure onto a single line.

- **prettier printer** (cf. Chapter 15.3)
  - implements $\text{Doc}$ in terms of suittable algebraic sum data types.
  - allows **multiple layouts** of a document and to pick a best one out of them for printing a document.
Chapter 15.2

The Simple Pretty Printer
Chapter 15.2.1
Basic Document Operators
The Simple Pretty Printer

...(as well as the prettier printer later on) relies on six basic document operators:

Associative operator for concatenating documents:

\[
(\langle\rangle) :: \text{Doc} \to \text{Doc} \to \text{Doc}
\]

The empty document being a right and left unit for (\langle\rangle):

\[
nil :: \text{Doc}
\]

Converting a string into a document (arguments of function \text{text} shall not contain newline characters):

\[
\text{text} :: \text{String} \to \text{Doc}
\]

The document representing a line break:

\[
\text{line} :: \text{Doc}
\]

Adding indentation to a document:

\[
\text{nest} :: \text{Int} \to \text{Doc} \to \text{Doc}
\]

Layouting a document as a string:

\[
\text{layout} :: \text{Doc} \to \text{String}
\]
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.
...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 15.5).
Chapter 15.2.2
Normal Forms of String Documents
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 "bbbbb" <> 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 "bbbbb[" <>
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 15.2.2.1 (Associativity of Doc. Concatenat.)**

$(<>)$ is associative with unit $\text{nil}$.

...as well as the collection of basic operator laws compiled in Lemma 15.2.2.2.
Basic Operators Laws

Lemma 15.2.2.2 (Basic Operator Laws)

1. Operator text is a homomorphism from string to document concatenation:

   \[
   \text{text} \ (s \ ++ \ t) \ = \ \text{text} \ s \ <> \ \text{text} \ t
   \]
   \[
   \text{text} \ "" \qquad = \ qquad \text{nil}
   \]

2. Opr. nest is a homomorph. from addition to composition:

   \[
   \text{nest} \ (i+j) \ x \quad = \quad \text{nest} \ i \ (\text{nest} \ j \ x)
   \]
   \[
   \text{nest} \ 0 \ x \quad = \quad x
   \]

3. Opr. nest distributes through document concatenation:

   \[
   \text{nest} \ i \ (x \ <> \ y) \ = \ \text{nest} \ i \ x \ <> \ \text{nest} \ i \ y
   \]
   \[
   \text{nest} \ i \ \text{nil} \quad = \quad \text{nil}
   \]

4. Nesting is absorbed by text (differently to the pretty printer of Hughes):

   \[
   \text{nest} \ i \ (\text{text} \ s) \ = \ \text{text} \ s
   \]
Note

...the laws compiled in Lemma 15.2.2.1 and 15.2.2.2

- 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 15.2.2.3 (Layout Operator Laws)

1. Operator layout is a homomorphism from document to string concatenation:

\[
\text{layout} (x <> y) = \text{layout} x ++ \text{layout} y \\
\text{layout} \text{nil} = ""
\]

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:

\[
\text{layout} (\text{nest} i \text{ line}) = \text{"\n'} : \text{copy} i \text{'} \\
\]
Chapter 15.2.3

Printing Trees
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[bbbbb[ccc, 
   dd], 
   eee, 
   ffff[gg, 
       hhh, 
       ii]]

aaa[
   bbbbbb[
       ccc, 
       dd 
   ], 
   eee, 
   ffff[
       gg, 
       hhh, 
       ii 
   ]
]

where $t = \text{Node } \text{"aaa"}$

[Node "bbbbb" [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.

\[
\text{aaa[bbbbb[ccc, } \\
\text{          dd], } \\
\text{       eee, } \\
\text{      ffff[gg, } \\
\text{         hhh, } \\
\text{        ii]]} \\
\text{aaa[} \\
\text{      bbbbbb[} \\
\text{          ccc, } \\
\text{          dd} \\
\text{          ],} \\
\text{      eee, } \\
\text{      ffff[} \\
\text{          gg, } \\
\text{          hhh, } \\
\text{          ii} \\
\text{      ]} \\
\text{]} \\
\]
Implementing the ‘Left’ Layout Strategy

...by means of a utility function \texttt{showTree} converting a tree into a string document according to the ‘left’ layout strategy:

\begin{verbatim}
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
\end{verbatim}
Implementing the ‘Right’ Layout Strategy

...by means of a utility function \texttt{showTree’} converting a tree into a string document according to the ‘right’ layout strategy:

\begin{verbatim}

\textbf{type Doc = String}
\textbf{data Tree = Node String [Tree]}

\textbf{showTree’ :: Tree -> Doc}
\textbf{showTree’ (Node s ts) = text s <> showBracket’ ts}

\textbf{showBracket’ :: [Tree] -> Doc}
\textbf{showBracket’ [] = nil}
\textbf{showBracket’ ts =}
\textbf{    text "[" <> nest 2 (line <> showTrees’ ts) <> line
\textbf{                       <> text "]")}

\textbf{showTrees’ :: [Tree] -> Doc}
\textbf{showTrees’ [t] = showTree t}
\textbf{showTrees’ (t:ts) =}
\textbf{    showTree t <> text "," <> line <> showTrees ts}

\end{verbatim}
Chapter 15.3
The Prettier Printer
Chapter 15.3.1
Algebraic Documents
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:

\[
\text{data Doc } = \begin{array}{c}
\text{Nil} \\
\mid \text{String 'Text' Doc} \\
\mid \text{Int 'Line' Doc}
\end{array}
\]

Note, the data constructors \text{Nil}, \text{Text}, and \text{Line} of \text{Doc} relate to the basic document operators \text{nil}, \text{text}, and \text{line} of the simple pretty printer as follows:

(1) \text{Nil} \equiv \text{nil}
(2) \text{s 'Text' x} \equiv \text{text s <> x}
(3) \text{i 'Line' x} \equiv \text{nest i line <> x}

...the normal form representation of the string document considered in Chapter 15.2.2:

```plaintext
text "bbbbb[" <>
nest 2 line <> text "ccc," <>
nest 2 line <> text "dd" <>
nest 0 line <> text "]
```

...is represented by the algebraic Doc-value:

```plaintext
"bbbbb[" 'Text' ( 2 'Line' ("ccc," 'Text' ( 2 'Line' ("dd," 'Text' ( 0 'Line' ("]," 'Text' Nil))))))
```
Chapter 15.3.2
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 15.3.1:

\[
\begin{align*}
\text{nil} & = \text{Nil} \\
\text{text } s & = s \text{ ‘Text’ Nil} \\
\text{line} & = 0 \text{ ‘Line’ Nil} \\
\text{Nil }\leftrightarrow y & = y \\
(s \text{ ‘Text’ } x) &\leftrightarrow y = s \text{ ‘Text’ } (x \leftrightarrow y) \\
(i \text{ ‘Line’ } x) &\leftrightarrow y = i \text{ ‘Line’ } (x \leftrightarrow y) \\
\text{nest } i \text{ Nil} & = \text{Nil} \\
\text{nest } i (s \text{ ‘Text’ } x) & = s \text{ ‘Text’ nest } i x \\
\text{nest } i (j \text{ ‘Line’ } x) & = (i+j) \text{ ‘Line’ nest } i x \\
\text{layout } \text{Nil} & = "$"
\text{layout } (s \text{ ‘Text’ } x) & = s \text{ ++ layout } x \\
\text{layout } (i \text{ ‘Line’ } x) & = '\n': \text{copy } i \text{ ‘ ‘ ++ layout } x
\end{align*}
\]
Justification

...for the derived definitions can be given using **equational reasoning**, e.g.:

**Proposition 15.3.2.1**

\[(s \text{ 'Text' } x) <> y = s \text{ 'Text' } (x <> y)\]

**Proof** by equational reasoning.

\[
\begin{align*}
(s \text{ 'Text' } x) <> y &= \{ \text{Definition of Text, equ. (2)} \} \\
(text s <> x) <> y &= \{ \text{Associativity of <>} \} \\
text s <> (x <> y) &= \{ \text{Definition of Text, equ. (2)} \} \\
s \text{ 'Text' } (x <> y) &\quad \square
\end{align*}
\]

...similarly, **correctness** of the other equations from the previous slide can be shown.
Chapter 15.3.3
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:

```haskell
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:

\[
\text{pretty} :: \text{Int} \rightarrow \text{Doc} \rightarrow \text{String}
\]

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 15.2.3:

```haskell
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`:

```haskell
showTree (Node s ts) =
    group (text s <> nest (length s) (showBracket ts))
```

will ensure that

- trees are fit onto one line where possible (≤ \textit{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:

\[(<|>) \::\ Doc \rightarrow Doc \rightarrow Doc\]

Flattening operator, replacing each line break (and its associated indentation) by a single space:

\[\text{flatten} \::\ Doc \rightarrow Doc\]

Note: The operators <|> and flatten will not directly exposed to the user but only via group and the operators fillwords and fill defined in Chapter 15.3.6.
Required Invariant for (<|>)

...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 (<|>) is required:

▶ Invariant: In (x <|> y) all layouts of x and y flatten to the same layout.

...this invariant must be ensured when creating a union (<|>).
Distribution Laws

...required for the implementations of \((<|>)\) and \texttt{flatten}.

Each operator on simple documents extends pointwise through union:

**Distributive Laws for \((<|>)\)**

1. \((x <|> y) <|> z = (x <|> z) <|> (y <|> z)\)
2. \(x <|> (y <|> z) = (x <|> y) <|> (x <|> z)\)
3. \(\text{nest } i (x <|> y) = \text{nest } i x <|> \text{nest } i y\)

Since flattening gives the same result for each element of a set, the distribution law for \texttt{flatten} is simpler:

**Distributive Law for \texttt{flatten}**

\[
\texttt{flatten } (x <|> y) = \texttt{flatten } x
\]
Interaction Laws

...required for the implementation of flatten.

Concerning the interaction of flatten with other document operators:

**Interaction Laws for flatten**

1. \( \text{flatten} (x <> y) = \text{flatten} x <> \text{flatten} y \)
2. \( \text{flatten} \text{ nil} = \text{nil} \)
3. \( \text{flatten} \text{ (text s)} = \text{text s} \)
4. \( \text{flatten} \text{ line} = \text{text " "} \)
5. \( \text{flatten} \text{ (nest i x)} = \text{flatten x} \)

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 `( <| > )`:

```haskell
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 15.3.4

Normal Forms of Algebraic Documents
Normal Form Representations

...due to the laws for flattening (\texttt{flatten}) and union (\texttt{(<|>)}) every document can be reduced to a representation in normal form of the form:

\[ x_1 <|> \ldots <|> x_n \]

where every \texttt{x}_j is in the normal form of simple documents (cf. Chapter 15.2.2).
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 15.3.1 is extended by a new data value constructor Union representing the union of two documents:

```haskell
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 $\equiv$ nil
2. s 'Text' x $\equiv$ text s <> x
3. i 'Line' x $\equiv$ nest i line <> x
4. x 'Union' y $\equiv$ 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.
...of pretty printing is improved by applying the **distributive law** for `Union` giving

\[(s \text{ 'Text' } (x \text{ 'Union' } y))\]

preference to the equivalent

\[((s \text{ 'Text' } x) \text{ 'Union' } (s \text{ 'Text' } y))\]
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 d c b a
d c b
```

1395/1927
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, $(<>)$ and \texttt{nest} must be extended to specify how they interact with \texttt{Union}:

\begin{align*}
(x \texttt{ 'Union' } y) <> z &= (x <> z) \texttt{ 'Union' } (y <> z) \tag{1} \\
nest k (x \texttt{ 'Union' } y) &= nest k x \texttt{ 'Union' } nest k y \tag{2}
\end{align*}

while the definitions of \texttt{nil}, \texttt{text}, \texttt{line}, $(<>)$, and \texttt{nest} remain unchanged.

\textbf{Note}, (1) and (2) follow from the distributive laws. In particular, they preserve Invariant 2 required by \texttt{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 reason-ning, e.g.:

Proposition 15.3.4.1

group (i 'Line' x) =
(" " 'Text' flatten x) 'Union' (i 'Line' x)

Proof by equational reasoning.

group (i 'Line' x)
= {Definition of Line, equ. (3)}
group (nest i line <> x)
= {Definition of group}
    flatten (nest i line <> x) <|> (nest i line s <> x)
= {Definition of flatten}
    (text " " <> flatten x) <|> (nest i line <> x)
= {Definition of Text, Union, Line, equ. (2), (4), (3)}
    (" " 'Text' flatten x) 'Union' (i 'Line' x)
\[\square\]
Justification (2)

Proposition 15.2.4.5

\[
\text{group } (s \ '\text{Text}' \ x) = s \ '\text{Text}' \ \text{group } x
\]

Proof by equational reasoning.

\[
\begin{align*}
\text{group } (s \ '\text{Text}' \ x) & = \{\text{Definition of Text, equ. (2)}\} \text{group } (\text{text } s \ <> \ x) \\
& = \{\text{Definition of group}\} \text{flatten } (\text{text } s \ <> \ x) \ <|> \ (\text{text } s \ <> \ x) \\
& = \{\text{Definition of flatten}\} (\text{text } s \ <> \ \text{flatten } x) \ <|> \ (\text{text } s \ <> \ x) \\
& = \{(<>\) \text{ distributes through } (<>\}) \text{text } s \ <> \ (\text{flatten } x \ <|> \ x) \\
& = \{\text{Definition of group}\} \text{text } s \ <> \ \text{group } x \\
& = \{\text{Definition of Text, equ. (2)}\} s \ '\text{Text}' \ \text{group } x
\end{align*}
\]

\[\square\]
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 $w$:

- $\text{fits } w \times x | w<0 = \text{False} \quad -- \text{cannot fit}$
- $\text{fits } w \times \text{Nil} = \text{True} \quad -- \text{fits trivially}$
- $\text{fits } w \times (s \times \text{Text} \times x)$
  $= \text{fits } (w - \text{length } s) \times x \quad -- \text{fits if } x \text{ fits into}$
  $\quad -- \text{the remaining space}$
  $\quad -- \text{after placing } s$

- $\text{fits } w \times (i \times \text{Line} \times x) = \text{True} \quad -- \text{yes, it fits}$

Last but not least, the output routine: Pick the best layout and convert it to a string:

$\text{pretty } w \times x = \text{layout } (\text{best } w \times 0 \times x)$
Chapter 15.3.5
Improving Performance
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:

\[
(\ldots((\text{text } s_0 \<> \text{text } s_1) <> \ldots) <> \text{text } s_n
\]

...assuming each string has length one, this may require time \(O(n^2)\) to process (instead of \(O(n)\) as hoped for).

2. Nesting of documents adds a layer of processing to increment the indentation of the inner document:

\[
\text{nest } i_o (\text{text } s_0 \<> \text{nest } i_1 (\text{text } s_1 <> \ldots <> \text{nest } i_n (\text{text } s_n)\ldots))
\]

...even if we assume document concatenation associates to the right.

...assuming again each string has length one, this may also require time \(O(n^2)\) to process (instead of \(O(n)\) 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:

```haskell
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:

\[
\begin{align*}
nil &= \text{NIL} \\
x <> y &= x :<> y \\
nest i x &= \text{NEST} i x \\
text s &= \text{TEXT} s \\
\text{line} &= \text{LINE}
\end{align*}
\]

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:

\[
\begin{align*}
group x &= \text{flatten } x :<> x \\
\text{flatten } \text{NIL} &= \text{NIL} \\
\text{flatten } (x :<> y) &= \text{flatten } x :<> \text{flatten } y \\
\text{flatten } \text{NEST } i x &= \text{NEST } i (\text{flatten } x) \\
\text{flatten } \text{TEXT } s &= \text{TEXT } s \\
\text{flatten } \text{LINE} &= \text{TEXT } " " \\
\text{flatten } (x :<> y) &= \text{flatten } x
\end{align*}
\]

...the definitions follow immediately from the equations given before.
The Representation Function \( \text{rep} \)

...maps a list of indentation-document pairs into the corresponding document:

\[
\text{rep } z = \text{fold } (<> ) \text{ nil } [\text{nest } i \text{ x } | \ (i,x) \leftarrow z]
\]
Finding the ‘best’ Layout

...the operation best of Chapter 15.3.4 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:

\[
be \ w \ k \ z = \ best \ w \ k \ (rep \ z) \quad \text{(hypothesis)}
\]

The new definition is directly derived from the old one:

\[
\begin{align*}
best \ w \ k \ x & = be \ w \ k \ [(0,x)] \\
be \ w \ k \ [] & = \text{Nil} \\
be \ w \ k \ ((i,\text{NIL}):z) & = be \ w \ k \ z \\
be \ w \ k \ ((i,x :<> y):z) & = be \ w \ k \ ((i,x):(i,y):z) \\
be \ w \ k \ ((i,\text{NEST}\ j \ x):z) & = be \ w \ k \ ((i+j),x):z) \\
be \ w \ k \ ((i,\text{TEXT}\ s):z) & = s \ '\text{Text}' \ be \ w \ (k,+\text{length}\ s) \ z \\
be \ w \ k \ ((i,\text{LINE}):z) & = i \ '\text{Line}' \ be \ w \ i \ z \\
be \ w \ k \ ((i.x :\text{|}| y):z) & = \\
& \text{better} \ w \ k \ (be \ w \ k \ ((i.x):z)) \ (be \ w \ k \ (i,y):z))
\end{align*}
\]
Correctness

...of the equations of the previous slide can be shown by equational reasoning, e.g.:

**Proposition 15.3.5.1**

\[
\text{best } w \ k \ x = \text{be } w \ k \ [(0, x)]
\]

**Proof** by equational reasoning.

\[
\begin{align*}
\text{best } w \ k \ x \\
= \{0 \text{ is unit for nest}\} \\
\text{best } w \ k \ (\text{nest } 0 \ x) \\
= \{\text{nil is unit for } <>\} \\
\text{best } w \ k \ (\text{nest } 0 \ x \ <> \ \text{nil}) \\
= \{\text{Definition of rep, hypothesis}\} \\
\text{be } w \ k \ [(0, x)]
\end{align*}
\]

□
...while the argument to \texttt{best} is represented using
\begin{itemize}
  \item \texttt{DOC}
\end{itemize}
its result is represented using the formerly introduced type
\begin{itemize}
  \item \texttt{Doc}
\end{itemize}
Hence, \texttt{pretty} can be defined as in Chapter 15.3.4:
\begin{verbatim}
pretty w x = layout (best w 0 x)
\end{verbatim}
The functions \texttt{layout}, \texttt{better}, and \texttt{fits}, finally, remain un-
changed.
Chapter 15.3.6

Utility Functions
Utility Functions (1)

...for *recurring*ly occurring tasks, e.g.:

- **Separating** two documents by inserting a *space*:
  \[ x \mathbin{\text{+}} y = x \mathbin{\text{<> text " " <>}} y \]

- **Separating** two documents by inserting a *line break*:
  \[ x \mathbin{\text{/>}} y = x \mathbin{\text{<> line <>}} y \]

- **Folding** a document:
  \[
  \begin{align*}
  \text{folddoc } f \; [] &= \text{nil} \\
  \text{folddoc } f \; [x] &= x \\
  \text{folddoc } f \; (x:xs) &= f \; x \; (\text{folddoc } f \; xs)
  \end{align*}
  \]

- **Advanced document folding**:
  \[
  \begin{align*}
  \text{spread} &= \text{folddoc } (\text{+}) \\
  \text{stack} &= \text{folddoc } (\text{/>})
  \end{align*}
  \]
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`:
  
  ```haskell
  bracket l x r = group (text l <>
  nest 2 (line <> x) <>
  line <> text r)
  ```

- The ‘right’ layout strategy for trees of Chapter 15.2.3, abbreviated by `showBracket'`:
  
  ```haskell
  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`:
  
  ```haskell
  x <+/> y = x <> (text " " :<|> line) <> y
  fillwords = folddoc (+/>). map text . words
  ```
...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`:

```haskell
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 15.3.7

Printing XML-like Documents
Printing XML Documents

...enjoying a simplified XML syntax with elements, attributes, and text defined by:

```haskell
data XML = Elt String [Att] [XML]
    | Txt String

data Att = Att String String
```

Utility Functions (1)

...for printing XML documents:

- **Showing documents:**
  \[
  \text{showXML} \; x = \text{folddoc} (\langle\rangle) (\text{showXMLs} \; x)
  \]

- **Showing elements:**
  \[
  \begin{align*}
  \text{showXMLs} \; (\text{Elt} \; n \; a \; []) &= \\
  &\quad \text{[text } \langle\rangle \text{ showTag } n \; a \text{ text } \rangle\rangle \\
  \text{showXMLs} \; (\text{Elt} \; n \; a \; c) &= \\
  &\quad \text{[text } \langle\rangle \text{ showTag } n \; a \text{ text } \rangle\rangle \text{ <> showFill showXMLs } c \text{ <> text } \langle\rangle \text{ text } n \text{ text } \rangle\rangle
  \end{align*}
  \]

- **Showing text:**
  \[
  \text{showXMLs} \; (\text{Txt} \; s) = \text{map} \; \text{text} \; (\text{words} \; s)
  \]

- **Showing attributes:**
  \[
  \begin{align*}
  \text{showAtts} \; (\text{Att} \; n \; v) &= \\
  &\quad \text{[text } n \text{ <> text } \langle\rangle \text{ text } \rangle\rangle \text{ <> text } (\text{quoted} \; v)]
  \end{align*}
  \]
Utility Functions (2)

...for printing XML documents (cont’d):

► Adding quotes:

\[ \text{quoted } s = "" + s + "" \]

► Showing tags:

\[ \text{showTag } n \ a = \text{text } n \ <> \ \text{showFill} \ \text{showAtts} \ a \]

► Filling lines:

\[ \text{showFill } f \ [ ] = \text{nil} \]
\[ \text{showFill } f \ xs = \]
\[ \quad \text{bracket } "" \ (\text{fill } (\text{concat } (\text{map } f \ xs)))\ "" \]

...for printing XML documents (cont’d):

▶ Adding quotes:

\[ \text{quoted } s = "" + s + "" \]

▶ Showing tags:

\[ \text{showTag } n \ a = \text{text } n \ <> \ \text{showFill} \ \text{showAtts} \ a \]

▶ Filling lines:

\[ \text{showFill } f \ [ ] = \text{nil} \]
\[ \text{showFill } f \ xs = \]
\[ \quad \text{bracket } "" \ (\text{fill } (\text{concat } (\text{map } f \ xs)))\ "" \]
Example: 1st Layout of an XML Document

...for a maximum line width of 30 characters:

```xml
<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:

```xml
<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: 3rd Layout of an XML Document

...dropping the two occurrences of flatten in fill (cf. Chapter 15.3.6) leads to the following output:

```xml
<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 15.4
The Prettier Printer Code Library
A Summary

...of the code of the

- performance-improved fully-fledged prettier printer.
- tree example.
- XML-documents example.

according to:

Chapter 15.4.1

The Prettier Printer
The Prettier Printer (1)

Defining operator priorities

```haskell
infixr 5:<|>
infixr 6:<>
infixr 6 <>
```

Defining algebraic document types

```haskell
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

\[
\begin{align*}
nil &= NIL \\
x <> y &= x :<> y \\
nest i x &= NEST i x \\
text s &= TEXT s \\
line &= LINE
\end{align*}
\]

Layouting normal form documents

\[
\begin{align*}
\text{layout Nil} &= ""
\text{layout (s ‘Text‘ x)} &= s ++ \text{layout x} \\
\text{layout (i ‘Line‘ x)} &= ‘\n’: \text{copy i ‘ ‘ ++ layout x} \\
\text{copy i x} &= [x \mid _ <- [1..i]]
\end{align*}
\]
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 \( w \ k \ x \) = \( \text{be} \ w \ k \ [(0,x)] \)

\( \text{be} \ w \ k \ [] \) = Nil

\( \text{be} \ w \ k \ ((i,\text{NIL}):z) \) = \( \text{be} \ w \ k \ z \)

\( \text{be} \ w \ k \ ((i,x:<>y):z) \) = \( \text{be} \ w \ k \ ((i,x):(i,y):z) \)

\( \text{be} \ w \ k \ ((i,\text{NEST} \ j \ x):z) \) = \( \text{be} \ w \ k \ ((i+j),x):z) \)

\( \text{be} \ w \ k \ ((i,\text{TEXT} \ s):z) \) = \( s \text{ 'Text' be} \ w \ (k+\text{length} \ s) \ z \)

\( \text{be} \ w \ k \ ((i,\text{LINE}):z) \) = \( i \text{ 'Line' be} \ w \ i \ z \)

\( \text{be} \ w \ k \ ((i.x:<>y):z) \) =

  better \( w \ k \ (\text{be} \ w \ k \ ((i.x):z)) \) (\( \text{be} \ w \ k \ (i,y):z) \)

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{ 'Text' x}) \) = fits (\( w - \text{length} \ s \)) \ x

fits \ w \ (i \text{ 'Line' x}) \ = \ True
The Prettier Printer (5)

Printing documents prettily

```haskell
pretty w x = layout (best w 0 x)
```

Defining utility functions

```haskell
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)

```haskell
fillwords = folddoc (<+/>). map text . words
fill []   = nil
fill [x]  = x
fill (x:y:zs) =
    (flatten x <+> fill (flatten y : zs))
        :<|> (x </> fill (y : zs))
```

1433/1927
Chapter 15.4.2
The Tree Example
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

```haskell
  tree = Node "aaa" [ Node "bbbb" [ Node "ccc" [],
             Node "dd" []
         ],
       Node "eee" [],
       Node "ffff" [ Node "gg" [],
             Node "hhh" [],
             Node "ii" []
         ]
]
```

Defining two testing environments

```haskell
  testtree w = putStr (pretty w (showTree tree))
  testtree' w = putStr (pretty w (showTree' tree))
```
Chapter 15.4.3

The XML Example
The XML Example (1)

Defining the XML-like document format

```haskell
data XML = Elt String [Att] [XML] | Txte String

data Att = Att String String
```

Defining utility functions

```haskell
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 (Txte s) = map text (words s)
```
The XML Example (2)

Defining utility functions (cont’d)

```haskell
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

\[
\text{xml} = \\
\text{Elt "p"[Att "color" "red",} \\
\text{Att "font" "Times",} \\
\text{Att "size" "10"} \\
\text{][Txt "Here is some",} \\
\text{Elt "em" [] [Txt "emphasized"],} \\
\text{Txt "text."}, \\
\text{Txt "Here is a",} \\
\text{Elt "a" [Att "href" "http://www.eg.com/"}] \\
\text{[Txt "link" ],} \\
\text{Txt "elsewhere."} \\
\text{]} \\
\]

Defining a testing environment

\[
\text{testXML w = putStrLn (pretty w (showXML xml))} \\
\]

Chapter 15.5
Summary
Summary

...the pretty printer library proposed by John Hughes is widely recognized as a standard:


...a variant of it is implemented in the Glasgow Haskell Compiler:

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

...two notes on an early work on an imperative pretty printer by:


...and a functional realization of it by:

Chapter 15.6

References, Further Reading
Chapter 15: Further Reading (1)


Chapter 15: Further Reading (2)


Chapter 15: Further Reading (3)


Chapter 16
Functional Reactive Programming
Chapter 16.1

Motivation
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 16.2 (using monads):


...for Chapter 16.3 (using arrows):


Note: Chapter 16.2 and 16.3 are independent and do not build upon each other.
Chapter 16.2

An Imperative Robot Language
Chapter 16.2.1
The Robot’s World
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) \texttt{drawSquare} =
    \begin{align*}
    & \text{do penDown} \\
    & \text{move} \\
    & \text{turnRight} \\
    & \text{move} \\
    & \text{turnRight} \\
    & \text{move}
    \end{align*}

(2) \texttt{moveToWall} =
    \begin{align*}
    & \text{while (isnt blocked)} \\
    & \text{do move}
    \end{align*}

(3) \texttt{getCoinsToWall} =
    \begin{align*}
    & \text{while (isnt blocked)} \$
    & \text{do move} \\
    & \text{checkAndPickCoin}
    \end{align*}
In more detail

...assuming that `Robot` is a monad:

```haskell
newtype Robot a = Rob...
instance Monad Robot where...

drawSquare =
do penDown      (penDown :: Robot () / pen ready to write)
  move          (move :: Robot () / moving one space forward)
  turnRight     (turnRight: Robot () / turn 90 degrees clockwise)
  move
  turnRight
  move

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 16.2.2
Modelling the Robot’s World
Modelling the World

...the robots live and act in a 2-dimensional grid.

**Positions** are given by their *x* and *y* coordinates:

```haskell
type Position = (Int, Int)
```

**Directions** a robot can face or head to:

```haskell
data Direction = North | East | South | West
deriving (Eq, Show, Enum)
```

**World**, a two-dimensional grid as **Array**-type:

```haskell
type Grid = Array Position [Direction]
```
Chapter 16.2.3
Modelling Robots
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:

```haskell
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:

```haskell
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:

```haskell
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

```plaintext
s1 = RState { position = (0,0)
, facing   = East
, pen      = True
, color    = Green
, treasure = [(2,3),(7,9),(12,42)]
, pocket   = 2
} :: RobotState
```

is equivalent to:

```plaintext
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

\[ s3 = s2 \{ \text{position} = (22,43), \text{pen} = \text{False} \} \]

\[ \implies RState \{ \text{position} = (22,43) \]
\[ , \text{facing} = \text{East} \]
\[ , \text{pen} = \text{False} \]
\[ , \text{color} = \text{Green} \]
\[ , \text{treasure} = [(2,3),(7,9),(12,42)] \]
\[ , \text{pocket} = 2 \]
\[ \} :: \text{RobotState} \]

Example 3: Accessing field values

\[ \text{position s1} \implies (0,0) \]
\[ \text{treasure s3} \implies [(2,3),(7,9),(12,42)] \]
\[ \text{color s3} \implies \text{Green} \]

Example 4: Using field names in patterns

\[ \text{jump (RState \{ \text{position} = (x,y) \})} = (x+2,y+1) \]
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:

```python
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 \textit{s1}.
Chapter 16.2.4
Modelling Robot Commands as State Monad
Modelling Robot Commands

...by Robot, a 1-ary type constructor, defined by:

```haskell
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):

```haskell
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:

```haskell
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

```haskell
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 16.2.5
The Imperative Robot Language
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
Recalling the Usage of Type Class Enum

The following ‘equalities’ hold:

\[
\begin{align*}
\text{enumFrom } n & \quad \triangleq \quad [n..] \\
\text{enumFromThen } n \; n' & \quad \triangleq \quad [n,n'..] \\
\text{enumFromTo } n \; m & \quad \triangleq \quad [n..m] \\
\text{enumFromThenTo } n \; n' \; m & \quad \triangleq \quad [n,n'..m]
\end{align*}
\]

Example:

```haskell
data Color = Red | Orange | Yellow | Green \\
| Blue | Indigo | Violet deriving Enum

[Red..Green]   \rightarrow\rightarrow [Red, Orange, Yellow, Green] \\
[Red, Yellow..] \rightarrow\rightarrow [Red, Yellow, Blue, Violet] \\
fromEnum Blue   \rightarrow\rightarrow 4 \\
toEnum 3        \rightarrow\rightarrow \text{Green}
```

16/1927
IRL Commands for Robot Orientation

...by updating the internal robot state.

- **Turn right:**
  
  ```hs
  turnLeft :: Robot ()
  turnLeft =
    updateState (\s -> s {facing = left (facing s)})
  ```

- **Turn left:**
  
  ```hs
  turnRight :: Robot ()
  turnRight =
    updateState (\s -> s {facing = right (facing s)})
  ```

- **Turn to:**
  
  ```hs
  turnTo :: Direction -> Robot ()
  turnTo d = updateState (\s -> s {facing = d})
  ```

- **Facing what direction?**
  
  ```hs
  direction :: Robot Direction
  direction = queryState facing
  ```
IRL Command for Blockade Checking

▶ Motion blocked in direction currently facing?

```haskell
blocked :: Robot Bool
blocked =
    Rob $ \ s \ g \_ \rightarrow
        return (s, facing s 'notElem' (g 'at' position s))

with \texttt{notElem} from the \texttt{Standard Prelude}.
```
IRL Commands for Motion

- Moving forward one space if not blocked:

```haskell
move :: Robot ()
move =
  cond1 (isnt blocked)
  (Rob $ \s \rightarrow \text{do}
    \text{let newPos = movePos (position } s \text{) (facing } s\text{)}
    \text{graphicsMove } w \ s \ \text{newPos}
    \text{return } (s \ {\text{position = newPos}}, ())
  )
```

- Moving forward one space in direction of:

```haskell
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:
  
  ```haskell
  setPenColor :: Color -> Robot ()
  setPenColor c = updateState (\s -> s {color = c})
  ```

- Pen down to start writing:
  
  ```haskell
  penDown :: Robot ()
  penDown = updateState (\s -> s {pen = True})
  ```

- Pen up to stop writing:
  
  ```haskell
  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 = queryStateState (\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?
  
  ```haskell
  coins :: Robot Int
  coins = queryState pocket
  ```

- Drop coin, if there is at least one in the pocket:
  
  ```haskell
  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:**
  
  \[
  \text{cond} :: \text{Robot Bool} \rightarrow \text{Robot a} \\
  \quad \rightarrow \text{Robot a} \rightarrow \text{Robot a}
  \]
  
  \[
  \text{cond} \ p \ c \ a = \text{do} \ pred <- p \\
  \quad \text{if} \ pred \ \text{then} \ c \ \text{else} \ a
  \]
  
  \[
  \text{cond1} \ p \ c = \text{cond} \ p \ c \ (\text{return} ()
  \]

- **Performing commands while some condition is met:**
  
  \[
  \text{while} :: \text{Robot Bool} \rightarrow \text{Robot ()} \rightarrow \text{Robot ()}
  \]
  
  \[
  \text{while} \ p \ b = \text{cond1} \ p \ (b \ >> \ \text{while} \ p \ b)
  \]

- **Connecting commands ‘disjunctively:’**
  
  \[
  (||*) :: \text{Robot Bool} \rightarrow \text{Robot Bool} \rightarrow \text{Robot Bool}
  \]
  
  \[
  \text{b1} \ ||* \ \text{b2} = \text{do} \ p <- \ b1 \\
  \quad \text{if} \ p \ \text{then} \ \text{return} \ True \\
  \quad \text{else} \ \text{b2}
  \]
Utility Functions for Logic and Control (2)

- Connecting commands ‘conjunctively:’
  \[
  (\&\&*) :: \text{Robot Bool} \rightarrow \text{Robot Bool} \rightarrow \text{Robot Bool} \\
  b1 \&\&* b2 = \text{do } p \leftarrow b1 \\
  \quad \text{if } p \text{ then } b2 \\
  \quad \text{else return False}
  \]

- Lifting negation to commands:
  \[
  \text{isnt} :: \text{Robot Bool} \rightarrow \text{Robot Bool} \\
  \text{isnt} = \text{liftM not}
  \]

- Lifting comparisons to commands:
  \[
  (>*) :: \text{Robot Int} \rightarrow \text{Robot Int} \rightarrow \text{Robot Bool} \\
  (>*) = \text{liftM2 (＞)}
  \]

  \[
  (<>*) :: \text{Robot Int} \rightarrow \text{Robot Int} \rightarrow \text{Robot Bool} \\
  (<>*) = \text{liftM2 (＜)}
  \]
Recalling the Definitions of the Lift Operators

...the higher-order lift operations \texttt{liftM} and \texttt{liftM2} are defined in the library \texttt{Monad} (as well as \texttt{liftM3}, \texttt{liftM4}, and \texttt{liftM5}):

\begin{verbatim}
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')
\end{verbatim}
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:**
  ```haskell```
  ```
  evade :: Robot ()
  evade = cond blocked
          (do turnRight
             move)
  move
  ```

- **cond1:**
  ```haskell```
  ```
  evade' :: Robot ()
  evade' = do cond1 blocked turnRight
             move
  ```
Moving in a Spiral

...an example of an advanced IRL program:

```haskell
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..]
```
Chapter 16.2.6
Defining a Robot’s World
The Robot’s World: Preliminary Definitions

The robot’s world is a grid of type `Array`:

```haskell
  type Grid = Array Position [Direction]
```

Grid points can be accessed using:

```haskell
  at :: Grid -> Position -> [Direction]
  at = (!)
```
Defining the Initial World $g_0$ (1)

The size of the initial grid world $g_0$ is given by:

```haskell
size :: Int
size = 20
```

with the grid world’s

- centre at: $(0,0)$
- corners at: $(-size, size)$ $$(size, size)$$ $$(0, -size)$$ $$(-size, 0)$$ $$(size, -size)$$
Defining the Initial World $g_0$ (2)

Inner, border, and corner points of world $g_0$ are characterized by the directions of motion they allow:

- **Inner points** of $g_0$ allow moving toward:
  \[ \text{interior} = [\text{North, South, East, West}] \]

- **Border points** at the north, east, south, and west border allow moving toward:
  \[ \text{nb} = [\text{South, East, West}] \quad (\text{nb: north border}) \]
  \[ \text{eb} = [\text{North, South, West}] \]
  \[ \text{sb} = [\text{North, East, West}] \]
  \[ \text{wb} = [\text{North, South, East}] \quad (\text{wb: west border}) \]

- **Corner points** at the northwest, northeast, southeast, and southwest corner allow moving toward:
  \[ \text{nwc} = [\text{South, East}] \quad (\text{nwc: northwest corner}) \]
  \[ \text{nec} = [\text{South, West}] \]
  \[ \text{sec} = [\text{North, West}] \]
  \[ \text{swc} = [\text{North, East}] \quad (\text{swc: southwest corner}) \]
Defining the Initial World $g_0$ (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 $g_0$ as follows:

$$
g_0 :: \text{Grid}\\
g_0 = \text{array} \left((-\text{size}, -\text{size}), (\text{size}, \text{size})\right)\\
\left(\left[[((i, \text{size}), \text{nb}) \mid i < - \text{r} \right] ++\\
\left[[((i, -\text{size}), \text{sb}) \mid i < - \text{r} \right] ++\\
\left[[((\text{size}, i), \text{eb}) \mid i < - \text{r} \right] ++\\
\left[[((-\text{size}, i), \text{wb}) \mid i < - \text{r} \right] ++\\
\left[[((\text{size}, i), \text{eb}) \mid i < - \text{r} \right] ++\\
\left[[((i,j), \text{interior}) \mid i < - \text{r}, j < - \text{r} \right] ++\\
\left[[((\text{size}, \text{size}), \text{nec}), ((\text{size}, -\text{size}), \text{sec}),\\
((-\text{size}, \text{size}), \text{nwc}),\\
((-\text{size}, -\text{size}), \text{swc})]\\
\right)\\
where \text{r} = [1-\text{size}..\text{size}-1]\)
Building World $g_1$ from World $g_0$

...by erecting a west/east-oriented wall leading from $(-5,10)$ to $(5,10)$:

$$g_1 :: \text{Grid}$$
$$g_1 = g_0 // \text{mkHorWall} (-5) 5 10$$

where $\text{Array}$ is the $\text{Array}$ library function (cf. Chapter 7.2):

$$\text{Array} :: \text{Ix} a \Rightarrow \text{Array} a b \rightarrow [(a,b)] \rightarrow \text{Array} a b$$
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.
Type `Color` is defined as in the Graphics library:

```haskell
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

```haskell
  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 \((x_2,y)\)):

\[
\text{mkHorWall} :: \text{Int} \to \text{Int} \to \text{Int} \to [\text{Position, [Direction]}]
\]

\[
\text{mkHorWall} \ x1 \ x2 \ y = \\
\quad [((x,y), \text{nb}) | x <- [x1..x2]] ++ \\
\quad [((x,y+1), \text{sb}) | x <- [x1..x2]]
\]

Building walls vertically (north/south-oriented, leading from \((x,y_1)\) to \((x,y_2)\)):

\[
\text{mkVerWall} :: \text{Int} \to \text{Int} \to \text{Int} \to [\text{Position, [Direction]}]
\]

\[
\text{mkVerWall} \ y1 \ y2 \ x = \\
\quad [((x,y), \text{eb}) | y <- [y1..y2]] ++ \\
\quad [((x+1,y), \text{wb}) | y <- [y1..y2]]
\]
Utility Functions for Building Rooms

...naively, rooms could be built using `mkHorWall` and `mkVerWall` straightforwardly:

```haskell
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:

\[
\text{accum :: (Ix a) => (b -> c -> b)} \\
\rightarrow \text{Array a b} \\
\rightarrow [(a,c)]} \\
\rightarrow \text{Array a b}
\]

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.:

  \[
  \text{[South, East, West] ‘intersect’} \\
  \text{[North, South, West] -> [South, West]}
  \]

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:

\[
g2 :: \text{Grid} \\
g2 = \text{accum intersect g0 (mkBox (-15,8) (2,17))}
\]

using \text{accum}, \text{intersect}, and \text{mkBox}.
Building World $g_3$ from World $g_2$

...by adding a **door** (to the middle of the top wall of the room)

$$g_3 :\text{ Grid}$$

$$g_3 = \text{accum union } g_2 \left[ ((-7,17), \text{ interior}), \right.$$  
$$\left. ((-7,18), \text{ interior}) \right]$$

using **accum**, **union**, and **interior**.
Chapter 16.2.7
Robot Graphics: Animation in Action
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`:

\[
\text{drawInWindowNow} :: \text{Window} \rightarrow \text{Color} \\
\quad \rightarrow \text{Point} \rightarrow \text{Point} \rightarrow \text{IO (})
\]

which draws the updated graphics immediately after any changes, and can be used, e.g., for drawing lines:

\[
\text{drawLine} :: \text{Window} \rightarrow \text{Color} \\
\quad \rightarrow \text{Point} \rightarrow \text{Point} \rightarrow \text{IO (})
\]

\[
\text{drawLine w c p1 p2 = drawInWindowNow w (withColor c (line p1 p2))}
\]
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:

\[ d :: \text{Int} \]
\[ d = 5 \]

Color of walls and coins:

\[ wc, cc :: \text{Color} \]
\[ wc = \text{Blue} \]
\[ cc = \text{Yellow} \]

Window size:

\[ xWin, yWin :: \text{Int} \]
\[ xWin = 600 \]
\[ yWin = 500 \]
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]]
Defining Utility Functions (2)

Used by `drawGrid`:

```haskell
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:

```haskell
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)))
Defining Utility Functions (4)

Drawing robot moves:

```haskell
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

```haskell
getWindowTick :: Window -> IO ()
```
Running IRL Programs: The Top-level Prg. (1)

...putting it all together.

Running an IRL program:

```haskell
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
```
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 $g_0$ through $g_3$ can now be used to run IRL programs with.

1) Fixing $s_0$ as a suitable starting state:

$$s_0 :: \text{RobotState}$$

$$s_0 = \text{RobotState} \begin{cases} \text{position} = (0,0) \\
, \text{pen} = \text{False} \\
, \text{color} = \text{Red} \\
, \text{facing} = \text{North} \\
, \text{treasure} = \text{tr} \\
, \text{pocket} = 0 \\
\end{cases}$$

2) Placing 'treasure' (all coins are placed inside the room in grid $g_3$):

$$\text{tr} :: [\text{Position}]$$

$$\text{tr} = [(x,y) | x \leftarrow [-13,-11..1], y \leftarrow [9,11..15]]$$
Animation in Action (2)

3) Running the ‘spiral’ program with s0, g0:

```haskell
main = runRobot spiral s0 g0
```

...leads to the ‘spiral’ example shown for illustration at the beginning of this chapter:
Chapter 16.3
Robots on Wheels
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 16.3.1
The Setting
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 at

http://www.haskell.org/yampa

In the following we will consider essential code snippets.
Chapter 16.3.2

Modelling the Robots’ World
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*:

```haskell
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

```haskell
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
```
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, otPos :: Position2, otHdng :: Heading -- Simbot A robot
  | OTSimbotB otRId :: RobotId, otPos :: Position2, otHdng :: Heading -- Simbot B robot
Chapter 16.3.3
Classes of Robots
Types of Robots

...usually, there are different types of robots

▶ differring 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:

```haskell
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 ()
```
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)

```haskell
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:

```haskell
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 16.3.4
Robot Simulation in Action
Typical Structure of a Robot Control Program

```haskell
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 = ...
```
Robot Simulation in Action

Running a robot simulation:

```haskell
runcSim :: Maybe WorldTemplate -> SimbotController -> SimbotController -> IO ()
```

Simbot controllers:

```haskell
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 16.3.5
Examples
Robot Actions: Control Programs (1)

A stationary robot:

\[
\text{rcStop :: SimbotController} \\
\text{rcStop \_ = constant (mrFinalize ddBrake)}
\]

A blind robot moving at constant speed:

\[
\text{rcBlind1 \_ =} \\
\text{constant (mrFinalize \$ ddVelDiff 10 10)}
\]

A blind robot moving at half the maximum speed:

\[
\text{rcBlind2 rps =} \\
\text{let max = rpWSMax rps} \\
\text{in constant (mrFinalize \$} \\
\text{ddVelDiff (max/2) (max/2))}
\]
Robot Actions: Control Programs (2)

A robot rotating at a pre-given speed:

```haskell
rcTurn :: Velocity -> SimbotController
rcTurn vel rps =
  let vMax = rpWSMax rps
      rMax = 2 * (vMax - vel) / rpDiameter rps
  in constant (mrFinalize $ ddVelTR vel rMax)
```
Chapter 16.4

Summary
The Origins

...of functional reactive programming (FRP) can be traced back to functional reactive animation (FRAn):


Seminal Works

...on functional reactive programming (FRP):


Applications of FRP (1)

...on Functional Reactive Robotics (FRob):


Applications of FRP (2)

...on Functional Animation Languages (FAL):

...on Functional Vision Systems (FVision):

...on Functional Reactive User Interfaces (FRUIt):
Applications of FRP (3)

...towards **Real-Time FRP (RT-FRP)**:


...towards **Event-Driven FRP (ED-FRP)**:

Chapter 16.5
References, Further Reading
Chapter 16: Further Reading (1)


Chapter 16: Further Reading (2)


Chapter 16: Further Reading (3)


Chapter 16: Further Reading (4)


Chapter 16: Further Reading (6)


Chapter 16: Further Reading (7)


Part VI

Extensions, Perspectives
Chapter 17
Extensions to Parallel and ‘Real World’ Functional Programming
Chapter 17.1

Parallelism in Functional Languages
Motivation

...recall:


...adopting a functional programming style could make your programs more robust, more compact, and more easily parallelizable.

Reading for this chapter:

Parallelism in Programming Languages

Predominant in imperative languages:

- Libraries (PVM, MPI) ↦ 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 $f(e_1, \ldots, e_n)$ 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
The Setting

...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:

```haskell
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:

  \[
  \leadsto \text{parallelization pattern Farm.}
  \]
Parallel Map on Distributed Lists

Illustration:

\[
\begin{align*}
\text{f } [a_1,\ldots, a_k, a_{k+1},\ldots, a_m, a_{m+1},\ldots, a_m] \\
\Downarrow \\
\text{Decomposition} \\
\text{f } [a_1,\ldots, a_k] & \quad \text{f } [a_{k+1},\ldots, a_m] & \quad \text{f } [a_{m+1},\ldots, a_m] \\
\Downarrow & & \Downarrow \\
\text{Parallel} & \text{Computation} & \text{Composition} \\
[b_1,\ldots, b_k] & [b_{k+1},\ldots, b_m] & [b_{m+1},\ldots, b_m] \\
\Downarrow & & \\
[b_1,\ldots, b_k, b_{k+1},\ldots, b_m, b_{m+1},\ldots, b_m]
\end{align*}
\]

On the Implementation

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: Supposes 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.
Programming of 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 $p_2$:

\[
\begin{array}{cccccc}
\text{a1} & \text{ak} & \text{ak+1} & \text{am} & \text{am+1} & \text{am} \\
\hline
p_0 & & & & \hline
p_1 & & & \hline
p_2 & & & & \hline
\end{array}
\]

Example (2)

...illlustrating a list cover with overlapping elements.

General Structure

...of a cover:

\[
\text{Cover} = \{ \\
\text{Type } S a & \quad \text{-- Whole object} \\
\quad C b & \quad \text{-- Cover} \\
\quad U c & \quad \text{-- Local sub-objects} \\
\}
\]

\[
\text{split :: } S a \rightarrow C (U a) \quad \text{-- Decomposing the} \\
\quad \text{-- original object} \\
\text{glue :: } C (U a) \rightarrow S a \quad \text{-- Composing the} \\
\quad \text{-- original object}
\]

where it must hold: glue \ . \ \text{split} = \text{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 17.2

Haskell for ‘Real World’ Programming
‘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
- ...

1580/1927
‘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

since 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.
Chapter 17.3

References, Further Reading
Chapter 17.1: Further Reading (1)


Chapter 17.1: Further Reading (2)


Chapter 17.1: Further Reading (3)


Chapter 17.1: Further Reading (4)


Chapter 17.1: Further Reading (5)


Chapter 17.1: Further Reading (6)


Chapter 17.2: Further Reading (6)


Chapter 17.2: 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)

Chapter 17.2: Further Reading (8)

- Peter Pepper, Petra Hofstedt. *Funktionale Programmierung*. Springer-V., 2006. (Kapitel 19, Agenten und Prozesse; Kapitel 20, Graphische Schnittstellen (GUIs))


- “Haskell community.” *Haskell wiki*. haskell.org/haskellwiki/Applications_and_libraries
Chapter 18

Conclusions and Perspectives
Chapter 18.1
Research Venues, Research Topics, and More
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:

Relevant Archival Journals

For functional programming:


For programming in general:

- ACM Transactions on Programming Languages and Systems (TOPLAS), since 1979.
Summer Schools

Focused on functional programming:

Hot Research Topics (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 (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.”

Hot Research Topics (3)

...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 (4)

- **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 (5)

- **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 18.2
Programming Contest
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.
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.)...

The 22nd Programming Contest at ICFP 2019

In 2019, the **programming contest** will start 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](https://icfpcontest2019.github.io)

- Stay tuned for news on the
  - programming contest series at the ICFP conf. series: [https://www.icfpconference.org/contest.html](https://www.icfpconference.org/contest.html)
Chapter 18.3

In Conclusion
Functional Programming

...certainly arrived in the real world:


- Haskell in Industry and Open Source: www.haskell.org/haskellwiki/Haskell_in_industry
A Plea for Functional Programming

...even though


might suggest the opposite, which, however, is actually not true, and Philip Wadler’s apparent lamentation is 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 the real world:


and brand-new:

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
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
Chapter 18.4
References, Further Reading
Chapter 18: Further Reading (1)


Chapter 18: Further Reading (2)


Chapter 18: Further Reading (3)


“Haskell community.” *Haskell in Industry and Open Source.*

www.haskell.org/haskellwiki/Haskell_in_industry
References
Reading

...for deepened and independent studies.

- I Textbooks
- II Monographs
- III Volumes
- IV Articles
- V Haskell 98 – Language Definition
- V The History of Haskell
# Textbooks (1)


I Textbooks (2)


I Textbooks (3)


I Textbooks (4)


I Textbooks (5)


I Textbooks (6)


I Textbooks (7)


I Textbooks (8)


I Textbooks (9)


Mark P. Jones, Alastair Reid et al. (Eds.). *The Hugs98 User Manual*. www.haskell.org/hugs


I Textbooks (10)


I Textbooks (11)


Textbooks (12)


I Textbooks (13)


I Textbooks (14)


I Textbooks (15)


I Textbooks (16)


II Monographs (1)


II Monographs (2)


III Volumes (1)


III Volumes (2)


IV Articles (2)


IV Articles (3)


IV Articles (4)


IV Articles (5)


IV Articles (6)


IV Articles (7)


IV Articles (8)


IV Articles (9)


IV Articles (10)


IV Articles (11)


IV Articles (12)


IV Articles (13)


Jeremy Gibbons. *Functional Pearls – An Editor’s Perspective*. www.cs.ox.ac.uk/people/jeremy.gibbons/pearls/


IV Articles (14)


IV Articles (15)


IV Articles (16)


IV Articles (17)


IV Articles (18)


IV Articles (19)


IV Articles (20)


IV Articles (21)


IV Articles (22)


IV Articles (23)


IV Articles (24)


IV Articles (25)


IV Articles (26)


IV Articles (27)


IV Articles (28)


IV Articles (30)


IV Articles (31)


IV Articles (32)


IV Articles (33)


IV Articles (34)


IV Articles (35)


IV Articles (36)


IV Articles (37)


IV Articles (38)


IV Articles (39)


IV Articles (40)


IV Articles (41)


IV Articles (42)


IV Articles (43)


IV Articles (44)


IV Articles (45)


IV Articles (46)


IV Articles (47)


IV Articles (48)


IV Articles (49)


V Haskell 98 – Language Definition


VI The History of Haskell

research.microsoft.com/users/simonpj/papers/haskell-retrospective/

Appendix
Appendix A

Mathematical Foundations
A.1 Relations
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 $M_1, \ldots, M_k$, i.e., $R \subseteq M_1 \times \ldots \times M_k$ is a subset of the cartesian product of the sets $M_i, 1 \leq i \leq k$.

**Examples**

- $\emptyset$ is the smallest relation on $M_1 \times \ldots \times M_k$.
- $M_1 \times \ldots \times M_k$ is the biggest relation on $M_1 \times \ldots \times M_k$. 
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 $(m, n) \in R$. 

1697/1927
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: $(M, R)$. 
Definition A.1.4 (Domain and Range)

Let $R$ be a relation from $M$ to $N$.

The sets

- $\text{dom}(R) = \{ m \mid \exists n \in N. (m, n) \in R \}$
- $\text{ran}(R) = \{ n \mid \exists m \in M. (m, n) \in R \}$

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

- **reflexive** iff $\forall m \in M. \ m R m$
- **irreflexive** iff $\forall m \in M. \ \neg m R m$
- **transitive** iff $\forall m, n, p \in M. \ m R n \land n R p \Rightarrow m R p$
- **intransitive** iff $\forall m, n, p \in M. \ m R n \land n R p \Rightarrow \neg m R p$
- **symmetric** iff $\forall m, n \in M. \ m R n \iff n R m$
- **antisymmetric** iff $\forall m, n \in M. \ m R n \land n R m \Rightarrow m = n$
- **asymmetric** iff $\forall m, n \in M. \ m R n \Rightarrow \neg n R m$
- **linear** iff $\forall m, n \in M. \ m R n \lor n R m \lor m = n$
- **total** iff $\forall m, n \in M. \ m R n \lor n R m$
(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 intransive, 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 ‘· divides ·’ (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
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 $=, \leq \text{ and } \geq$ on $\mathbb{IN}$.
- The relation $m \mid n \ (m \text{ is a divisor of } n)$ on $\mathbb{IN}$.

**Strict partial order** (asymmetric, transitive)
- The relations $<$ and $>$ on $\mathbb{IN}$.
- 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 $\mathbb{IN}$.
- 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 Ordering

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

$$\forall p, q \in P. \quad p \sqsubset q \iff p \sqsubseteq q \land p \neq q$$

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 \( \sqsubseteq \) be a strict partial order (asymmetric, transitive) on \( P \).

**Lemma A.2.1.6**
The relation \( \sqsubseteq \) is irreflexive.

**Lemma A.2.1.7**
The pair \((P, \sqsubseteq)\), where \( \sqsubseteq \) is defined by
\[
\forall p, q \in P. \ p \sqsubset q \iff_{df} p \sqsubseteq q \lor p = q
\]
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

\[
\forall q, r \in Q. \; q \sqsubseteq_Q r \iff q \sqsubseteq_P r
\]

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. $5 \mid 35$.

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 ‘· is an immediate predecessor of ·’ defined by

$$p R q \iff p \sqsubseteq q \land \forall r \in P. \ p \sqsubseteq r \sqsubseteq q$$

of a partial order $(P, \sqsubseteq)$, where $\sqsubseteq$ is the strict part of $\sqsubseteq$. 

1714/1927
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:
  - \( p \sqsubseteq q \land p = q \) (reflexivity): trivially represented (just without an explicit link)
  - \( p \sqsubseteq q \land p \neq q \) (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?

a) \{ \} 

b) 1

c) 1

d) 1 2

e) 2

f) 3

g) 3

h) 2 3

i) 4 3

j) 6 5

2 4

1
Exercise A.2.2.2

Let \( \mid \) 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. \( 5 \mid 35 \).

Draw an expressive section of the Hasse diagram of the divisibility relation \( \mid \) 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

- lower bound of \(Q'\), in signs: \(q \sqsubseteq Q'\), if \(\forall q' \in Q'. q \sqsubseteq q'\)
- upper bound of \(Q'\), in signs: \(Q' \sqsubseteq q\), if \(\forall q' \in Q'. q' \sqsubseteq q\)
- 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: \(\hat{q} \sqsubseteq q\).
- 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

- \textbf{minimal element} of \(Q'\), if there is no \(q' \in Q'\) with \(q' \sqsubset q\).
- \textbf{maximal element} of \(Q'\), if there is no \(q' \in Q'\) with \(q \sqsubset q'\).
- \textbf{least (or minimum) element} of \(Q'\), if \(q \sqsubseteq Q'\).
- \textbf{greatest (or maximum) element} of \(Q'\), if \(Q' \sqsubseteq q\).

\textbf{Note:} Least and greatest elements of \(Q\) itself are usually denoted by \(\bot\) 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 \(\bigvee\) and \(\bigwedge\) partial maps \(\bigvee, \bigwedge : \mathcal{P}(P) \to P\) 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 \(\bot\) and \(\top\))**

The least element \(\bot\) 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.,

\[
\bot = \bigcup \emptyset = \bigcap P \quad \text{and} \quad \top = \bigcap \emptyset = \bigcup P
\]

if they exist.
Lower and Upper Bound Sets

Considering $\sqcup$ and $\sqcap$ partial functions $\sqcup, \sqcap : \mathcal{P}(P) \to P$ on the powerset of a partial order $(P, \sqsubseteq)$ suggests introducing two further maps $LB, UB : \mathcal{P}(P) \to \mathcal{P}(P)$ on $\mathcal{P}(P)$:

**Definition A.2.3.6 (Lower and Upper Bound Sets)**

Let $(P, \sqsubseteq)$ be a partial order. Then:

$L,B, UB : \mathcal{P}(P) \to \mathcal{P}(P)$ denote two maps, which map a subset $Q \subseteq P$ to the set of its lower bounds and upper bounds, respectively:

1. $\forall Q \subseteq P. \quad LB(Q) =_{df} \{ lb \in P \mid lb \sqsubseteq Q \}$
2. $\forall Q \subseteq P. \quad UB(Q) =_{df} \{ ub \in P \mid Q \sqsubseteq ub \}$
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:

\[
\bigvee Q = \bigcap UB(Q) \quad \text{and} \quad \bigwedge Q = \bigcup LB(Q)
\]

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. \(Q_1 \subseteq Q_2 \Rightarrow LB(Q_1) \supseteq LB(Q_2) \land UB(Q_1) \supseteq UB(Q_2)\)
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?

a) \{ \}  
b) 1  
c) 1  
d) 1  
e) 2  
f) 3  
g) 1  
h) 2  
i) 3  
j) 1  

\begin{align*}
\text{\{ \}} & \quad 1 & \quad 2 \\
\text{\{ \}} & \quad 2 \\
\text{\{ \}} & \quad 3 \\
\text{\{ \}} & \quad 4 \\
\text{\{ \}} & \quad 5 \\
\text{\{ \}} & \quad 6 \\
\end{align*}
Exercise A.2.3.10

Let $\mid$ denote the divisibility relation on the set of natural numbers $\mathbb{IN}_0$, i.e., the relation ‘· divides ·’ (w/out remainder), e.g. $5 \mid 35$.

Write down the sets of elements of $\mathbb{IN}_0$, which are

1. minimal
2. maximal
3. least
4. greatest

wrt the divisibility relation $\mid$ on $\mathbb{IN}_0$. 
A.2.4

Noetherian and Artinian Orders
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 \((\mathbb{N}, \sqsubseteq)\) be a Noetherian order, let \(\mathbb{N}_{\text{min}} \subseteq \mathbb{N}\) be the set of minimal elements of \(\mathbb{N}\), and let \(\phi : \mathbb{N} \to \mathbb{IB}\) be a predicate on \(\mathbb{N}\). Then:

If

1. \(\forall n \in \mathbb{N}_{\text{min}}. \phi(n)\)\text{ (Induction base)}
2. \(\forall n \in \mathbb{N} \setminus \mathbb{N}_{\text{min}}. (\forall m \sqsubseteq n. \phi(m)) \Rightarrow \phi(n)\) \text{ (Induction step)}

then:

\(\forall n \in \mathbb{N}. \phi(n)\)
A.2.5 Chains
Chains, Antichains

Let \((P, \sqsubset)\) 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., \(\forall c_1, c_2 \in C.\ c_1 \sqsubseteq c_2 \lor c_2 \sqsubseteq c_1\).

**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

- $C = \{c_0 \sqsubseteq c_1 \sqsubseteq c_2 \sqsubseteq \ldots\}$
- $C = \{c_0 \sqsupseteq c_1 \sqsupseteq c_2 \sqsupseteq \ldots\}$

is called an ascending chain and descending chain, respectively.
Examples of Chains

The set

- $S =_{df} \{ n \in \text{IN} \mid n \text{ even} \}$ is a chain in \text{IN}.
- $S =_{df} \{ z \in \mathbb{Z} \mid z \text{ odd} \}$ is a chain in \mathbb{Z}.
- $S =_{df} \{ \{ k \in \text{IN} \mid k < n \} \mid n \in \text{IN} \}$ is a chain in the powerset $\mathcal{P}(\text{IN})$ of \text{IN}.

Note: A chain can always be given in the form of an ascending or descending chain.

- $\{0 \leq 2 \leq 4 \leq 6 \leq \ldots\}$: \text{IN} as ascending chain.
- $\{\ldots \geq 6 \geq 4 \geq 2 \geq 0\}$: \text{IN} as descending chain.
- $\{\ldots \leq -3 \leq -1 \leq 1 \leq 3 \leq \ldots\}$: \mathbb{Z} as ascending chain.
- $\{\ldots \geq 3 \geq 1 \geq -1 \geq -3 \geq \ldots\}$: \mathbb{Z} as descending chain.
- $\ldots$
Eventually Stationary Sequences

Definition A.2.5.5 (Stationary Sequence)

1. An ascending sequence of the form

\[ p_0 \sqsubseteq p_1 \sqsubseteq p_2 \sqsubseteq \ldots \]

is called \textit{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

\[ p_0 \supseteq p_1 \supseteq p_2 \supseteq \ldots \]

is called \textit{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

\[ p_0 \sqsubseteq p_1 \sqsubseteq p_2 \sqsubseteq \ldots \quad \text{or} \quad p_0 \sqsupseteq p_1 \sqsupseteq p_2 \sqsupseteq \ldots \]

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

\[ \{ p_0 \sqsubseteq p_1 \sqsubseteq p_2 \sqsubseteq \ldots \} \quad \text{or} \quad \{ p_0 \sqsupseteq p_1 \sqsupseteq p_2 \sqsupseteq \ldots \} \]

and in terms of sequences

\[ p_0 \sqsubseteq p_1 \sqsubseteq p_2 \sqsubseteq \ldots \quad \text{or} \quad p_0 \sqsupseteq p_1 \sqsupseteq p_2 \sqsupseteq \ldots \]

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 $p_1 \sqsubseteq p_2 \sqsubseteq \ldots \sqsubseteq p_n \sqsubseteq \ldots$ 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 $p_1 \sqsupseteq p_2 \sqsupseteq \ldots \sqsupseteq p_n \sqsupseteq \ldots$ 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 statements are equivalent:

1. \((P, \sqsubseteq)\) is a Noetherian order.
2. \((P, \sqsubseteq)\) satisfies the descending chain condition.
3. Every chain of the form
   \[ p_0 \supseteq p_1 \supseteq p_2 \supseteq \ldots \]
   is eventually stationary, i.e.:
   \[ \exists n \in \mathbb{IN}. \ \forall j \in \mathbb{IN}. \ p_{n+j} = p_n. \]
4. Every chain of the form
   \[ p_0 \supseteq p_1 \supseteq p_2 \supseteq \ldots \]
   is finite.
Chains and Artinian Orders

Let \((P, \sqsubseteq)\) be a partial order.

**Lemma A.2.5.9 (Artinian Order)**

The following statements are equivalent:

1. \((P, \sqsubseteq)\) is an Artinian order.
2. \((P, \sqsubseteq)\) satisfies the ascending chain condition.
3. Every chain of the form

   \[ p_0 \sqsubseteq p_1 \sqsubseteq p_2 \sqsubseteq \ldots \]

   is eventually stationary, i.e.: \(\exists n \in \mathbb{IN}. \forall j \in \mathbb{IN}. \ p_{n+j} = p_n\).
4. Every chain of the form

   \[ p_0 \sqsubset p_1 \sqsubset p_2 \sqsubset \ldots \]

   is finite.
Chains and Noetherian, Artinian Orders

Let \((P, \sqsubseteq)\) be a partial order.

**Lemma A.2.5.10 (Noetherian and Artinian Order)**

The following statements are equivalent:

1. \((P, \sqsubseteq)\) is a Noetherian and an Artinian order.
2. \((P, \sqsubseteq)\) satisfies the descending and the ascending chain condition.
3. Every chain \(C \subseteq P\) is finite.
A.2.6
Directed Sets
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 statement of Corollary A.2.6.4 does usually not hold.
Strongly Directed Sets

Let \((P, \sqsubseteq)\) be a partial order with least element \(\bot\), 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. \(\bot \in D\)
2. \(\forall d, e \in D. \exists f \in D. f = \bigvee\{d, e\}\), i.e., for any two elements \(d\) and \(e\) the supremum \(\bigvee\{d, e\}\) of \(d\) and \(e\) exists in \(D\).
Properties of Strongly Directed Sets

Let \((P, \sqsubseteq)\) be a partial order with least element \(\bot\), 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 statement of Lemma A.2.6.7 does usually not hold, if \(D\) is infinite.
Directed Sets, Strongly Directed Sets, Chains

Let \((P, \sqsubseteq)\) be a partial order with least element \(\bot\).

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 \(\bot \in D\) and \(D\) is a chain.

Corollary A.2.6.9

Let \(\emptyset \neq D \subseteq P\) be a non-empty subset of \(P\). Then:

\(\bot \in D \land D\) 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)  
\[ \begin{align*}
1 & \rightarrow 2 \\
3 & \rightarrow 4
\end{align*} \]

b)  
\[ \begin{align*}
1 & \rightarrow 2 \\
3 & \rightarrow 4
\end{align*} \]

c)  
\[ \begin{align*}
1 & \rightarrow 2 \\
3 & \rightarrow 4
\end{align*} \]

d)  
\[ \begin{align*}
2 & \rightarrow 1 \\
2 & \rightarrow 3 \\
4 & \rightarrow 3
\end{align*} \]

\[ \begin{align*}
1 & \\
3 & \\
4 & \\
\end{align*} \]

e)  
\[ \begin{align*}
2 & \rightarrow 4 \\
1 & \rightarrow 3
\end{align*} \]

f)  
\[ \begin{align*}
2 & \rightarrow 4 \\
1 & \rightarrow 3
\end{align*} \]

g)  
\[ \begin{align*}
2 & \rightarrow 4 \\
1 & \rightarrow 5
\end{align*} \]
Exercise A.2.6.11

Which of the below partial orders are (strongly) directed sets? Which of their subsets are (strongly) directed sets?

a)  
\[ \begin{array}{ccc}
4 & 5 \\
2 & 3 \\
1 & \\
\end{array} \]

b)  
\[ \begin{array}{ccc}
4 & 5 \\
2 & 3 \\
1 & \\
\end{array} \]

c)  
\[ \begin{array}{ccc}
4 & 5 \\
2 & 3 \\
1 & \\
\end{array} \]

d)  
\[ \begin{array}{ccc}
4 & 5 \\
2 & 3 \\
1 & \\
\end{array} \]

e)  
\[ \begin{array}{ccc}
6 & \\
4 & 5 \\
2 & 3 \\
1 & \\
\end{array} \]

f)  
\[ \begin{array}{ccc}
6 & \\
4 & 5 \\
2 & 3 \\
1 & \\
\end{array} \]

g)  
\[ \begin{array}{ccc}
6 & \\
4 & 5 \\
2 & 3 \\
1 & \\
\end{array} \]

h)  
\[ \begin{array}{ccc}
6 & \\
4 & 5 \\
2 & 3 \\
1 & \\
\end{array} \]
Exercise A.2.6.12

Let \((\mathbb{N}_0, \sqsubseteq)\) be the partial order with \(\sqsubseteq = \text{df } |\), where \(|\) denotes the divisibility relation on the natural numbers \(\mathbb{N}_0\), i.e., the relation ‘· divides ·’ (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 \(f \in [C \to D]\) 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 \(f \in [C \to C]\) 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

- expanding (or inflationary) for \(\hat{c}\) iff \(\hat{c} \sqsubseteq f(\hat{c})\)
- expanding (or inflationary) iff \(\forall c \in C. c \sqsubseteq f(c)\)

Definition A.2.7.4 (Contracting Maps on POs)

\(f\) is called

- contracting (or deflationary) for \(\hat{c}\) iff \(f(\hat{c}) \sqsubseteq \hat{c}\)
- 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 \(f \in [P \to R]\) 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
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., \( \bigsqcup C \) 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., \( \bigsqcup C \) 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., \(\bigsqcup D\) exists \(\in P\).
2. pointed directedly complete partial order (DCPO), if it is a pre-DCPO and has a least element \(\bot\).

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 \(\bot\), which is given by the supremum of the empty chain or set, i.e.: \(\bot = \bigcup \emptyset\).

**Corollary A.3.1.4 (Non-Emptyness 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 ⊑ be a relation on $P$.

**Lemma A.3.1.5 (Fin. POs, pre-CCPOs, pre-DCPOs)**

The following statements 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 statements are equivalent:

1. $(P, \sqsubseteq)$ is a partial order.
2. $(P, \sqsubseteq)$ is a CCPO.
3. $(P, \sqsubseteq)$ is a DCPO.
Theorem A.3.1.7 (Equivalence)

Let \((P, \sqsubseteq)\) be a partial order. Then the following statements 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 \(\bigcup C\) of \(C\) chain \(\subseteq \mathcal{P}(\mathbb{N})\): \(\bigcup_{C' \in C} C'\)

- The set of finite and infinite strings \(S\) partially ordered by the prefix relation \(\sqsubseteq_{pfx}\) defined by
  \[
  \forall s, s'' \in S. \ s \sqsubseteq_{pfx} s'' \iff df
  \]
  \[
  s = s'' \lor (s \text{ finite } \land \exists s' \in S. \ s + s' = s'')
  \]
  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 $\sqsubseteq_{\text{lex}}$ defined by

\[
\forall s, t \in S. \ s \sqsubseteq_{\text{lex}} t \iff df
\]

\[
s = t \lor (\exists p \text{ finite}, s', t' \in S. \ s = p + + s' \land t = p + + t' \land (s' = \varepsilon \lor s'_1 < t'_1))
\]

where $\varepsilon$ denotes the empty string, $w_{1}$ 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

- \((\text{IN}, \leq)\) is not a CPO (i.e., not a CCPO and DCPO).
- The set of finite strings \(S_{\text{fin}}\) partially ordered by the
  - prefix relation \(\sqsubseteq_{\text{prefix}}\) defined by
    \[
    \forall s, s' \in S_{\text{fin}}.\; s \sqsubseteq_{\text{prefix}} s' \iff \exists s'' \in S_{\text{fin}}.\; s + + s'' = s'
    \]
    is not a CPO (i.e., not a CCPO and DCPO).
- lexicographical order \(\sqsubseteq_{\text{lex}}\) defined by
  \[
  \forall s, t \in S_{\text{fin}}.\; s \sqsubseteq_{\text{lex}} t \iff \exists p, s', t' \in S_{\text{fin}}.\; s = p + + s' \land t = p + + t' \land
  (s' = \varepsilon \lor s'_{\downarrow 1} < t'_{\downarrow 1})
  \]
  where \(\varepsilon\) 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).
- \((\mathcal{P}_{\text{fin}}(\text{IN}), \subseteq)\) 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)  

Which of the partial orders given by the below Hasse diagrams are (pre-) CCPOs? Which ones are (pre-) DCPOs?
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 \((\mathbb{IN}_0, \sqsubseteq)\) be the partial order with \(\sqsubseteq = \text{df} \mid\), where \(\mid\) denotes the divisibility relation on the natural numbers \(\mathbb{IN}_0\), i.e., the relation ‘\(\cdot\) divides \(\cdot\)’ (w/out remainder), e.g. \(5 \mid 35\).

Prove or disprove: \((\mathbb{IN}_0, \sqsubseteq)\) 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 \(f \in [C \to D]\) 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

\[
\forall C' \neq \emptyset \text{ chain } \subseteq C. \quad f(\bigsqcup_C C') = D \bigsqcup_D f(C')
\]

(Preservation of least upper bounds)

**Note:** \(\forall S \subseteq C. \quad f(S) =_{df} \{ f(s) \mid s \in S \}\)
Let \((D, \sqsubseteq_D)\) and \((E, \sqsubseteq_E)\) be DCPOs, and let \(f \in [D \to E]\) be a map from \(D\) to \(E\).

**Definition A.3.2.2 (Continuous Maps on DCPOs)**

\(f\) is called **continuous** iff

\[
\forall D' \neq \emptyset \text{ directed set } \subseteq D. f(D') \text{ directed set } \subseteq E \land \quad f(\bigsqcup_D D') =_E \bigsqcup_E f(D')
\]

(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
   
   \[\text{iff } \forall C' \neq \emptyset \text{ chain } \subseteq C. \quad f(C') \text{ chain } \subseteq D \land f(\bigsqcup_C C') \sqsubseteq_D \bigsqcup_D f(C')\]

2. \(g : E \rightarrow F\) is monotonic
   
   \[\text{iff } \forall E' \neq \emptyset \text{ directed set } \subseteq E. \quad g(E') \text{ directed set } \subseteq F \land g(\bigsqcup_E E') \sqsubseteq_F \bigsqcup_F g(E')\]
Strict Maps on CCPOs and DCPOs

Let \((C, \sqsubseteq_C), (D, \sqsubseteq_D)\) be CCPOs with least elements \(\bot_C\) and \(\bot_D\), respectively, let \((E, \sqsubseteq_E), (F, \sqsubseteq_F)\) be DCPOs with least elements \(\bot_E\) and \(\bot_F\), respectively, and let \(f \in [C \cong D]\) and \(g \in [E \cong F]\) be continuous maps.

**Definition A.3.2.4 (Strict Functions on CPOs)**

\(f\) and \(g\) are called **strict**, if the equalities
\[
\begin{align*}
\forall C' = \emptyset, E' = \emptyset: \\
\bigvee C C' & =_D \bigvee_D f(C'), & \bigvee E E' & =_F \bigvee_F g(E')
\end{align*}
\]
also hold for \(C' = \emptyset\) and \(E' = \emptyset\), i.e., if the equalities
\[
\begin{align*}
\forall C = \emptyset, E = \emptyset: \\
\bigvee C \emptyset & =_C \bot_C, & f(\bot_C) & =_D \bot_D, & \bigvee D \emptyset & =_D \bot_D \\
\bigvee E \emptyset & =_E \bot_E, & g(\bot_E) & =_F \bot_F, & \bigvee F \emptyset & =_F \bot_F
\end{align*}
\]
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 CPOs

Lemma A.3.3.1 (Flat CPO Construction)

Let $C$ be a set. Then:

$$(C \cup \{\bot\}, \sqsubseteq_{\text{flat}})$$

with $\sqsubseteq_{\text{flat}}$ defined by

$$\forall c, d \in C \cup \{\bot\}. \ c \sqsubseteq_{\text{flat}} d \iff_{df} c = \bot \lor c = d$$

is a CPO, a so-called flat CPO.
Common CPO Constructions: Flat pre-CPOs

Lemma A.3.3.2 (Flat Pre-CPO Construction)

Let $D$ be a set. Then:

$$(D \cup \{\top\}, \sqsubseteq_{flat})$$

with $\sqsubseteq_{flat}$ defined by

$$\forall d, e \in D \cup \{\top\}. d \sqsubseteq_{flat} e \iff_{df} e = \top \lor d = e$$

is a pre-CPO, a so-called flat pre-CPO.
Common CPO Constructions: Products (1)

Lemma A.3.3.3 (Non-strict Product Construction)

Let \((P_1, \sqsubseteq_1), (P_2, \sqsubseteq_2), \ldots, (P_n, \sqsubseteq_n)\) be CPOs. Then:

The non-strict product \((\times P_i, \sqsubseteq_\times)\), where

- \(\times P_i =_{df} P_1 \times P_2 \times \ldots \times P_n\) is the cartesian product of all \(P_i, 1 \leq i \leq n\)
- \(\sqsubseteq_\times\) is defined pointwise by

\[
\forall (p_1, \ldots, p_n), (q_1, \ldots, q_n) \in \times P_i.
\]

\[\!
(p_1, \ldots, p_n) \sqsubseteq_\times (q_1, \ldots, q_n) \iff_{df} \forall i \in \{1, \ldots, n\}. \ p_i \sqsubseteq_i q_i
\]

is a CPO.
Lemma A.3.3.4 (Strict Product Construction)

Let \((P_1, \sqsubseteq_1), (P_2, \sqsubseteq_2), \ldots, (P_n, \sqsubseteq_n)\) be CPOs. Then:

The strict (or smash) product \(\bigotimes P_i, \sqsubseteq_{\bigotimes}\), where

\[
\bigotimes P_i =_{df} \times P_i \text{ is the the cartesian product of all } P_i
\]

\[
\sqsubseteq_{\bigotimes} =_{df} \sqsubseteq_{\times} \text{ defined pointwise with the additional setting}
\]

\[
(p_1, \ldots, p_n) = \bot \iff_{df} \exists i \in \{1, \ldots, n\}. p_i = \bot_i
\]

is a CPO.
Common CPO Constructions: Sums (1)

Lemma A.3.3.5 (Separated Sum Construction)
Let \((P_1, \sqsubseteq_1), (P_2, \sqsubseteq_2), \ldots, (P_n, \sqsubseteq_n)\) be CPOs. Then:

The separated (or direct) sum \((\bigoplus_{\bot} P_i, \sqsubseteq_{\bigoplus_{\bot}})\), where

\begin{itemize}
  \item \(\bigoplus_{\bot} P_i =_{df} P_1 \cup P_2 \cup \ldots \cup P_n \cup \{\bot\}\) is the disjoint union of all \(P_i, 1 \leq i \leq n\), and a fresh bottom element \(\bot\)
  \item \(\sqsubseteq_{\bigoplus_{\bot}}\) is defined by
    \[\forall p, q \in \bigoplus_{\bot} P_i. \ p \sqsubseteq_{\bigoplus_{\bot}} q \iff_{df} p = \bot \lor (\exists i \in \{1, \ldots, n\}. p, q \in P_i \land p \sqsubseteq_i q)\]
\end{itemize}

is a CPO.
Common CPO Constructions: Sums (2)

Lemma A.3.3.6 (Coalesced Sum Construction)

Let \((P_1, \sqsubseteq_1), (P_2, \sqsubseteq_2), \ldots, (P_n, \sqsubseteq_n)\) be CPOs. Then:

The **coalesced sum** \((\bigoplus \lor_i P_i, \sqsubseteq_{\bigoplus \lor})\), where

\[
\bigoplus \lor P_i =_{df} P_1 \setminus \{\bot_1\} \lor P_2 \setminus \{\bot_2\} \lor \ldots \lor P_n \setminus \{\bot_n\} \lor \{\bot\}
\]

is the disjoint union of all \(P_i, 1 \leq i \leq n\), and a fresh bottom element \(\bot\), which is identified with and replaces the least elements \(\bot_i\) of the sets \(P_i\), i.e., \(\bot =_{df} \bot_i\), \(i \in \{1, \ldots, n\}\)

\[
\sqsubseteq_{\bigoplus \lor}
\]

is defined by

\[
\forall p, q \in \bigoplus \lor P_i. \quad p \sqsubseteq_{\bigoplus \lor} q \iff_{df} p = \bot \lor \left( \exists i \in \{1, \ldots, n\}. \quad p, q \in P_i \land p \sqsubseteq_i q \right)
\]

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 \([C \xrightarrow{\text{con}} D], \sqsubseteq_{cfs}\), where

- \([C \xrightarrow{\text{con}} D]\) is the set of continuous maps from \(C\) to \(D\)
- \(\sqsubseteq_{cfs}\) is defined pointwise by

\[
\forall f, g \in [C \xrightarrow{\text{con}} D].\ f \sqsubseteq_{cfs} g \iff \forall c \in C.\ f(c) \sqsubseteq_D g(c)
\]

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
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 \(\bot\), satisfying:
   \[ \bot = \bigsqcup \emptyset = \bigcap P. \]
2. a greatest element in \(P\), denoted by \(\top\), satisfying:
   \[ \top = \bigcap \emptyset = \bigsqcup P. \]

Lemma A.4.1.4 (Characterization Lemma)
Let \((P, \sqsubseteq)\) be a partial order. Then the following statements 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
   \[\forall \emptyset \neq S \subseteq P. \bigcup S \text{ exists } \in P.\]

2. **meet semi-lattice** (in German: Schnitthalbverband) iff
   \[\forall \emptyset \neq S \subseteq P. \bigcap S \text{ exists } \in P.\]
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 \(\bigvee P \text{ exists } \in P\) (whereas \(\bigvee \emptyset (\equiv \bot)\) does usually not exist in \(P\)).

2. meet semi-lattice, then \(\bigwedge P \text{ exists } \in P\) (whereas \(\bigwedge \emptyset (\equiv \top)\) 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:

\[ \bigvee P \text{ exists } \in P \text{ and is the (unique) greatest element in } P \]
that is usually denoted by \(\top\), i.e., \(\top = \bigvee P\).

Lemma A.4.1.10 (Ex. least El. in C. Meet Semi-Lat.)

Let \((P, \sqsubseteq)\) be a complete meet semi-lattice. Then:

\[ \bigwedge P \text{ exists } \in P \text{ and is the (unique) least element in } P \] that is usually denoted by \(\bot\), i.e., \(\bot = \bigwedge P\).
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 \(\bigsqcap Q\) exists \(\in P\) satisfying
   \[
   \bigsqcap Q = \bigsqcup \{p \in P \mid p \sqsubseteq Q\}
   \]

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
   \[
   \bigsqcup Q = \bigsqcap \{p \in P \mid Q \sqsubseteq p\}
   \]

If \((P, \sqsubseteq)\) is a complete

1. join semi-lattice and \(\bigcup \emptyset \in P\), then \(\bigcup \emptyset\) is the (unique) least element in \(P\), denoted by \(\bot\), i.e., \(\bot = \bigcup \emptyset\).

2. meet semi-lattice and \(\bigwedge \emptyset \in P\), then \(\bigwedge \emptyset\) is the (unique) greatest element in \(P\), denoted by \(\top\), i.e., \(\top = \bigwedge \emptyset\).
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 $\bigvee \emptyset \in P$
2. meet semi-lattice and $\bigwedge \emptyset \in P$

then $(P, \sqsubseteq)$ is a complete lattice.
Exercise A.4.1.14

Prove or disprove:

If \((P, \sqsubseteq)\) is a complete lattice, then

1. \((P\setminus\perp, \sqsubseteq\perp)\) is a complete join semi-lattice.
2. \((P\setminus\top, \sqsubseteq\top)\) is a complete meet semi-lattice.

where \(\sqsubseteq\perp\) and \(\sqsubseteq\top\) 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

a) \{a,b,c\}
   \{a,b\} \{a,c\} \{b,c\}
   \{a\} \{b\} \{c\}
   \{

b) True
   False

c) d)
(Anti-) Examples

- The partial order \((P, \sqsubseteq)\) given by the below Hasse diagram is not a lattice (whereas it is a CPO).

- \((P_{\text{fin}}(\text{IN}), \subseteq)\) 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) 1  
c) 1 2  
d) 2  
e) 3  
f) 3  
g) 2 3  
h) 4  
i) 6  

1803/19
Exercise A.4.1.18

Let $(\mathbb{IN}_0, \sqsubseteq)$ be the partial order with $\sqsubseteq =_{df} |$, where $|$ denotes the divisibility relation on the natural numbers $\mathbb{IN}_0$, i.e., the relation ‘· divides ·’ (w/out remainder), e.g. $5 | 35$.

Prove or disprove: $(\mathbb{IN}_0, \sqsubseteq)$ 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) \lor
$(P, \sqsubseteq)$ complete join semi-lattice and
\[ \bigsqcup \emptyset \text{ exists } \in P \text{ (L. A.4.1.13(1))} \lor \]
$(P, \sqsubseteq)$ complete meet semi-lattice and
\[ \bigsqcap \emptyset \text{ exists } \in P \text{ (L. A.4.1.13(2))} \]
\[ \Rightarrow (P, \sqsubseteq) \text{ complete lattice} \]

(D. A.4.1.2 and
L. A.4.1.14) \Rightarrow (P, \sqsubseteq) \text{ lattice and complete partial order}

(D. A.4.1.1 and
D. A.3.1.1/2) \Rightarrow (P, \sqsubseteq) \text{ partial order}

(D. A.2.1.2) \Rightarrow (P, \sqsubseteq) \text{ pre-order}
Excercise A.4.1.20

Let

\[ \text{QO, PO, L, CPO, CL, FL, CJSL, CJSL}_\perp, \text{CMSL, CMSL}^\top \]

denote the sets of all quasi-orders QO, partial orders PO, lattices L, complete partial orders CPO, complete lattices CL, finite lattices 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, 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 \(f \in [P \to P]\) be a map on \(P\).

**Definition A.4.2.1 (Distributive, Additive Map)**

\(f\) is called

- **distributive** (or \(\sqcap\)-continuous) iff
  \[
  \forall \emptyset \neq P' \subseteq P. \ f(\sqcap P') = \sqcap f(P')
  \]
  (Preservation of greatest lower bounds)

- **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:

\[
\begin{align*}
\text{f is monotonic} & \iff \forall P' \subseteq P. \ f(\bigcap P') \subseteq \bigcap f(P') \\
& \iff \forall P' \subseteq P. \ f(\bigcup P') \supseteq \bigcup f(P')
\end{align*}
\]

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 $f \in [P \to P]$ be a map on $P$.

**Lemma A.4.2.3**

\[ f \text{ is distributive iff } f \text{ is additive.} \]

**Lemma A.4.2.4**

\[ f \text{ is monotonic, if } f \text{ is distributive (or additive).} \]

(i.e., distributivity (or additivity) implies monotonicity)
A.4.3

Lattice Homomorphisms, Lattice Isomorphisms
Lattice Homomorphisms, Lattice Isomorphisms

Let \((P, \sqsubseteq_P)\) and \((R, \sqsubseteq_R)\) be two lattices, and let \(f \in [P \to R]\) 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**

\[
f \in [P^{\text{hom}} \rightarrow R] \Rightarrow f \in [P^{\text{mon}} \rightarrow R]
\]

The reverse implication of **Lemma A.4.3.3** does not hold, however, the following weaker relation holds:

**Lemma A.4.3.4**

\[
f \in [P^{\text{mon}} \rightarrow R] \Rightarrow
\forall p, q \in P.\ f(p \sqcup_P q) \sqsubseteq_Q f(p) \sqcup_Q f(q) \land
f(p \sqcap_P q) \sqsubseteq_Q f(p) \sqcap_Q f(q)
\]
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**

\[
f \in [P \xrightarrow{iso} R] \implies f^{-1} \in [R \xrightarrow{iso} P]
\]

**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

\[ \forall p, q, r \in P. \ p \sqsubseteq r \Rightarrow p \sqcup (q \sqcap r) = (p \sqcup q) \sqcap r \]
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:

\[ \begin{array}{c}
\text{a} \\
\text{b} \\
\text{c} \\
\text{d} \\
\text{e}
\end{array} \]
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 statements are equivalent:

1. \( \forall p, q, r \in P. \ p \cup (q \cap r) = (p \cup q) \cap (p \cup r) \)

2. \( \forall p, q, r \in P. \ p \cap (q \cup r) = (p \cap q) \cup (p \cap r) \)

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) \[
\begin{array}{ccc}
    & e & \\
    a & b & \\
    c & & d
\end{array}
\]

b) \[
\begin{array}{ccc}
    e & & \\
    b & c & d
\end{array}
\]

**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 \(\bot\), 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 \(p \sqcap q = \bot\).
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 \(\bot \neq \top\).

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. \(\bar{\bar{p}} = p\) \hspace{1cm} \text{(Involution Law)}
2. \(\bar{p} \sqcup q = \bar{p} \cap \bar{q}, \quad \bar{p} \cap q = \bar{p} \sqcup \bar{q}\) \hspace{1cm} \text{(De Morgan Laws)}
3. \(p \sqsubseteq q \iff \bar{p} \cup q = \top \iff p \cap \bar{q} = \bot\)
4. \(p \sqsubseteq q \sqcup r \iff p \cap \bar{q} \sqsubseteq r \iff \bar{q} \sqsubseteq \bar{p} \sqcup r\)
Boolean Lat. Homomorphisms/Isomorphisms

Let \((P, \sqsubseteq_P)\) and \((Q, \sqsubseteq_Q)\) be two Boolean lattices, and let \(f \in [P \to Q]\) 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**

\[ f(\bot) = \bot \land f(\top) = \top \]

**Lemma A.4.4.15**

\(f\) is a Boolean lattice isomorphism iff \(f(\bot) = \bot \land f(\top) = \top\)
Summary, Overview

Corollary A.4.4.16

Let \( P \neq \emptyset \) be a non-empty set, and \( \sqsubseteq \) a relation on \( P \). Then:

\[
(P, \sqsubseteq) \text{ Boolean lattice}
\]

(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

\[
\text{QO} \supset \text{PO} \supset \text{L} \supset \text{ML} \supset \text{DL} \supset \text{BL}
\]

where all inclusions are proper and \( \text{QO}, \text{PO}, \text{L}, \text{ML}, \text{DL} \), and \( \text{BL} \) 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 \(\sqsubseteq = \text{df} \mid\), where \(|\) denotes the divisibility relation on the natural numbers \(\mathbb{N}_0\), i.e., the relation ‘\(\cdot \) divides \(\cdot\)’ (w/out remainder), e.g. \(5 \mid 35\).

Prove or disprove: \((\mathbb{N}_0, \sqsubseteq)\) 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:

$$(C \cup \{\bot, \top\}, \sqsubseteq_{\text{flat}})$$ with $\sqsubseteq_{\text{flat}}$ defined by

$$\forall c, d \in C \cup \{\bot, \top\}. c \sqsubseteq_{\text{flat}} d \iff_{df} c = \bot \lor c = d \lor d = \top$$

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.
- \(\sqcap, \sqcup : P \times P \to P\) 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 \(\sqcap, \sqcup : P \times P \to P\) satisfy the following laws:

▶ Idempotency Laws:
\[
\begin{align*}
 p \sqcap p &= p \\
 p \sqcup p &= p
\end{align*}
\]

Lemma A.4.6.3
For all \(p, q \in P\), the maps \(\sqcap, \sqcup : P \times P \to P\) 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

\[
\forall p, q \in P. \quad p \sqsubseteq q \iff p \sqcap q = p
\]

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 \((P, \cap, \cup)\) be an algebraic lattice, and let \(\sqsubseteq\) be the induced partial order of \((P, \cap, \cup)\).

**Lemma A.4.6.6**

For all \(p, q \in P\), the infimum (\(\hat{\cap}\) greatest lower bound) and the supremum (\(\hat{\cup}\) least upper bound) of the set \(\{p, q\}\) exist and are given by the images of \(\cap\) and \(\cup\) applied to \(p\) and \(q\), respectively, i.e.:

\[
\forall p, q \in P. \quad \cap \{p, q\} = p \cap q \land \cup \{p, q\} = p \cup q
\]

**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:

\[
\exists \text{glb}, \text{lub} \in P. \quad \text{glb} = \cap Q \land \text{lub} = \cup Q
\]
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. \(\forall p, q \in P. \quad p \sqcap q =_{df} \bigsqcap\{p, q\}\)
2. \(\forall p, q \in P. \quad p \sqcup q =_{df} \bigsqcup\{p, q\}\)
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 statements 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:

- **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:**
  \[ (p \sqcap q) \sqcup p = p \]
  \[ (p \sqcup q) \sqcap p = p \]

- **Idempotency Laws:**
  \[ p \sqcap p = p \]
  \[ p \sqcup p = p \]
Order-theoretic Lattices Algebraically

Corollary A.4.6.12 (From \((P, \subseteq)\) to \((P, \sqcap, \sqcup)\))

Let \((P, \subseteq)\) be an order-theoretic lattice. Then:

\((P, \sqcap, \sqcup)\), where \(\sqcap\) and \(\sqcup\) are the induced maps of \((P, \subseteq)\), 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 $\bot$ and top $\top$ with
  - Bottom $\bot \in P$: $\bot = \bigcup \emptyset$
  - Top $\top \in P$: $\top = \bigcap \emptyset$

- **algebraically** as $(P, \sqcap, \sqcup)$, it is appropriate to think of its least and greatest element in terms of Zero $0$ and One $1$, where $(P, \sqcap, \sqcup)$ is said to have a (if existent, uniquely determined)
  - Zero $0 \in P$: $\forall p \in P. \ p \sqcup 0 = p$
  - One $1 \in P$: $\forall p \in P. \ p \sqcap 1 = p$
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 \( 0 \), and in that case:
   \[
   \left( \bigcup \emptyset = \right) \perp = 0
   \]

2. \((P, \sqsubseteq)\) has a top element \( \top \) iff \((P, \sqcap, \sqcup)\) has a one element \( 1 \), and in that case:
   \[
   \left( \bigcap \emptyset = \right) \top = 1
   \]
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 \(\mid\) denotes the divisibility relation on the natural numbers \(\mathbb{N}_0\), i.e., the relation ‘\(\cdot\) divides \(\cdot\)’ (w/out remainder), e.g. \(5 \mid 35\).

Provide the definition of \((\mathbb{N}_0, \land, \lor)\), 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. \(\land : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0\)
2. \(\lor : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0\)

What is the

1. zero element \(0\)
2. one element \(1\)

of \((\mathbb{N}_0, \land, \lor)\)?
A.5

Fixed Point Theorems
A.5.1

Fixed Points, Towers
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 \(f \in [P \to P]\) be a function on \(P\), and let \(p\) be a fixed point of \(f\), i.e., \(f(p) = p\). Then:

\(p\) is called the

- **least fixed point** of \(f\), denoted by \(\mu f\),
  iff \(\forall q \in P. \ f(q) = q \implies p \sqsubseteq q\)

- **greatest fixed point** of \(f\), denoted by \(\nu f\),
  iff \(\forall q \in P. \ f(q) = q \implies q \sqsubseteq p\)
Towers in Chain Complete Partial Orders

Definition A.5.1.3 ($f$-Tower in $C$)

Let $(C, \sqsubseteq)$ be a CCPO, let $f \in [C \to C]$ 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. $\bot \in T$.
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

$$I = \text{df} \bigcap \{T \mid T \text{ $f$-tower in } C\}$$

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 \(\mu f\), 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 \(f \in [C \xrightarrow{con} C]\) 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
\(\{\bot, f(\bot), f^2(\bot), \ldots\}\), i.e.:

\[
\mu f = \bigsqcup_{i \in \mathbb{N}_0} f^i(\bot) = \bigsqcup_{i \in \mathbb{N}_0}\{\bot, f(\bot), f^2(\bot), \ldots\}
\]

Note: \(f^0 = df \text{Id}_C; f^i = df f \circ f^{i-1}, \ i > 0.\)
Proof of Fixed Point Theorem A.5.2.3 (1)

We have to prove:

$$\mu f = \bigsqcup_{i \in \mathbb{N}_0} f^i(\bot) = \bigsqcup \{ f^i(\bot) \mid i \geq 0 \}$$

1. exists,
2. is a fixed point of $f$,
3. is the least fixed point of $f$. 

1857/1927
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 $\{f^i(\perp) \mid i \geq 0\}$ exists.

\[
\bigsqcup\{f^i(\perp) \mid i \geq 0\} = \bigsqcup_{i \in \mathbb{N}_0} f^i(\perp) \text{ exists.}
\]
Proof of Fixed Point Theorem A.5.2.3 (3)

2. Fixed point property

\[ f(\bigsqcup_{i \in \mathbb{N}_0} f^i(\bot)) \]

\[
(f \text{ continuous}) = \bigsqcup_{i \in \mathbb{N}_0} f(f^i(\bot))
\]

\[
= \bigsqcup_{i \in \mathbb{N}_1} f^i(\bot)
\]

\[
(C' =_{df} \{ f^i \bot \mid i \geq 1 \} \text{ is a chain} \Rightarrow \\
\bigsqcup C' \text{ exists} = \bot \sqcup \bigsqcup C') = \bot \sqcup \bigsqcup_{i \in \mathbb{N}_1} f^i(\bot)
\]

\[
(f^0(\bot) =_{df} \bot) = \bigsqcup_{i \in \mathbb{N}_0} f^i(\bot)
\]
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: $\bot \sqsubseteq c$.
- Since $f$ is continuous and hence monotonic, we obtain by means of (natural) induction:
  $\forall i \in \mathbb{N}_0. \ f^i(\bot) \sqsubseteq f^i(c) \ (= c)$.
- Since $c$ is a fixed point of $f$, this implies:
  $\forall i \in \mathbb{N}_0. \ f^i(\bot) \sqsubseteq c \ (= f^i(c))$.
- Thus, $c$ is an upper bound of the set $\{f^i(\bot) \mid i \in \mathbb{N}_0\}$.
- Since $\{f^i(\bot) \mid i \in \mathbb{N}_0\}$ is a chain, and $\bigsqcup_{i \in \mathbb{N}_0} f^i(\bot)$ is by definition the least upper bound of this chain, we obtain the desired inclusion

$$\bigsqcup_{i \in \mathbb{N}_0} f^i(\bot) \sqsubseteq c.$$
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.:

\[
\forall x \in C.\ f(x) = x \land d \sqsubseteq x \Rightarrow c_d \sqsubseteq x
\]
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 \(\{d, f(d), f^2(d), \ldots\}\), i.e.:

\[
\mu f_d = \bigsqcup_{i \in \mathbb{N}_0} f^i(d) = \bigsqcup \{d, f(d), f^2(d), \ldots\}
\]
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 \(i \in \mathbb{IN}\) with \(f^i(\bot) = f^{i+1}(\bot)\), then we have: \(\mu f = f^i(\bot)\).

**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 \(i \in \mathbb{IN}\) 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

- finite iff the set \(Q =_{df} \{q \in P \mid q \sqsubseteq p\}\) does not contain an infinite chain.

- 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 $f \in [P \stackrel{\text{mon}}{\rightarrow} P]$ be a monotonic function on $P$. Then:

1. $f$ has a unique least fixed point $\mu f \in P$, which is given by $\mu f = \prod \{p \in P \mid f(p) \sqsubseteq p\}$.

2. $f$ has a unique greatest fixed point $\nu f \in P$, which is given by $\nu f = \bigsqcup \{p \in P \mid p \sqsubseteq f(p)\}$.

Characterization Theorem A.5.3.2 (Davis)

Let $(P, \sqsubseteq)$ be a lattice. Then:

$(P, \sqsubseteq)$ is complete iff every $f \in [P \stackrel{\text{mon}}{\rightarrow} P]$ 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 \(f \in [P \rightarrow P]^{mon}\) be a monotonic function on \(P\), and let \(\text{Fix}(f) = \{ p \in P \mid f(p) = p \}\) 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 \(f \in [P \rightarrow P]^{mon}\) 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 \(f \in [P \to P]\) be a function on \(P\). Then: \(f\) has a unique

1. least fixed point \(\mu f \in P\) given by \(\mu f = \bigsqcup_{i \in \mathbb{N}_0} f^i(\bot)\), if \(f\) is additive, i.e., \(f \in [P^{\text{add}} \to P]\).

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., \(f \in [P^{\text{dis}} \to P]\).

Recall: \(f^0 =_{df} \text{Id}_C\); \(f^i =_{df} f \circ f^{i-1}\), \(i > 0\).
A.6

Fixed Point Induction
Admissible Predicates

Fixed point induction allows proving properties of fixed points. Essential is the notion of admissible predicates:

Definition A.6.1 (Admissible Predicate)

Let \((P, \sqsubseteq)\) be a complete lattice, and let \(\phi : P \rightarrow \mathbb{IB}\) be a predicate on \(P\). Then:

\(\phi\) is called admissible (or \(\sqsubseteq\)-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 \(\phi : P \rightarrow \mathbb{IB}\) be an admissible predicate on \(P\). Then: \(\phi(\bot) = \text{true}\).

Proof. The admissibility of \(\phi\) implies \(\phi(\bigsqcup \emptyset) = \text{true}\). Moreover, we have \(\bot = \bigsqcup \emptyset\), which completes the proof.
Sufficient Conditions for Admissibility

Theorem A.6.3 (Admissibility Condition 1)

Let \((P, \sqsubseteq)\) be a complete lattice, and let \(\phi : P \to \text{IB}\) be a predicate on \(P\). Then:

\(\phi\) is admissible, if there is a complete lattice \((Q, \sqsubseteq_Q)\) and two additive functions \(f, g \in [P^{\text{add}} \to Q]\), 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 \(\phi, \psi : P \to \text{IB}\) be two admissible predicates on \(P\). Then:

The conjunction of \(\phi\) and \(\psi\), the predicate \(\phi \land \psi\) defined by

\[
\forall p \in P. \ (\phi \land \psi)(p) =_{df} \phi(p) \land \psi(p)
\]

is admissible.
Fixed Point Induction on Complete Lattices

**Theorem A.6.5 (Fixed Point Induction on C. Lat.)**

Let \((P, \sqsubseteq)\) be a complete lattice, let \(f \in [P^{\text{add}}] P\) be an additive function on \(P\), and let \(\phi : P \rightarrow \mathbb{I} \mathbb{B}\) be an admissible predicate on \(P\). Then:

The validity of

\[
\forall p \in P. \ \phi(p) \Rightarrow \phi(f(p))
\]

(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 \(\phi\) (cf. Lemma A.6.2) and proved when verifying the admissibility of \(\phi\).
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 \(f \in [C^{\text{mon}} \to C]\) be a monotonic function on \(C\), and let \(\phi : C \to IB\) be an admissible predicate on \(C\). Then:

The validity of
\[\forall c \in C. \phi(c) \Rightarrow \phi(f(c))\]  
(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
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: \( q \in D \implies p \in D \).
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.

![Hasse diagram]

Then, e.g.:

1. \(\emptyset, P \in \mathcal{D}(P), \forall q \in P. \{p \in P \mid p \sqsubseteq q\} \in \mathcal{D}(P)\)
2. \(\{1, 3\}, \{1, 2, 3\}, \{1, 2, 3, 4\} \in \mathcal{D}(P)\)
3. \(\{2, 3\}, \{2, 4, 5\}, \{1, 2, 4, 5\} \notin \mathcal{D}(P)\)
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 statements 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:

\[ D(P) = \{ \downarrow Q \mid Q \subseteq P \} \]

Corollary A.7.1.5

Let \((P, \sqsubseteq)\) be a partial order, let \(D \in D(P)\), 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. \( \downarrow q \equiv_{df} \{ p \in P \mid p \sqsubseteq q \} \) 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 \((P, \sqsubseteq)\):

\[
\begin{array}{c}
6 \\
4 & 5 \\
2 & 3 \\
1
\end{array}
\]

\[
\begin{align*}
\forall p \in P. & \quad \downarrow p =_{df} \{ r \mid r \sqsubseteq p \} \text{ directed } \in \mathcal{D}(P). \\
\forall p \in P \setminus \{6\}. & \quad \downarrow p \text{ strongly directed } \in \mathcal{D}(P). \\
\downarrow 6 =_{df} \{ r \mid r \sqsubseteq 6 \} = \{1, 2, 3, 4, 5, 6\} = P \in \mathcal{D}(P) \text{ is a downset of } P, \text{ however, it is not strongly directed, since its subsets } \{2, 3\}, \{1, 2, 3\} \subseteq \downarrow 6 \text{ do not have a least upper bound in } \downarrow 6 = P \text{ (though upper bounds: } 4, 5, 6). 
\end{align*}
\]
Example 2: Consider the below lattice \((\mathbb{Z}, \leq)\):

\[
\begin{array}{c}
\vdots \\
2 \\
1 \\
0 \\
-1 \\
-2 \\
\vdots \\
\end{array}
\]

\[
\mathcal{D}(\mathbb{Z}) = \emptyset \cup \mathcal{PD}(\mathbb{Z}) \cup \mathbb{Z} = \emptyset \cup \{ \downarrow z =_{df} \{ r \in \mathbb{Z} \mid r \leq z \} \mid z \in \mathbb{Z} \} \cup \mathbb{Z}
\]

\[
\forall S \in \mathcal{D}(\mathbb{Z}). \ S \ \text{directed} \ \text{but not strongly directed} \ (\text{since it lacks a least element}).
\]
Downsets, Directed Sets (3)

...arbitrary downsets even of complete lattices are usually not strongly directed, though directed.

Example 3: Consider the below complete lattice \((P, \sqsubseteq)\):

- \(\downarrow\{4, 5\} =_{df} \{ r \mid r \sqsubseteq 4 \lor r \sqsubseteq 5 \} = \{1, 2, 3, 4, 5\} \in \mathcal{D}(P)\)
- \(\downarrow\{3, 4\} =_{df} \{ r \mid r \sqsubseteq 3 \lor r \sqsubseteq 4 \} = \{1, 2, 3, 4\} \in \mathcal{D}(P)\)

E.g., the downsets

- \(\downarrow\{4, 5\}\) 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:
   - \(p, q \in I \Rightarrow p \sqcup q \in I\).
   - \(q \in I \Rightarrow p \sqcap q \in I\).

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 \(\bot\), and \(I \in \mathcal{I}(P)\). Then:

1. \(\bot \in I\).
2. \(\{\bot\} \in \mathcal{I}(P)\) 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:

\[
I \in \mathcal{I}(P) \iff \forall p, q \in P. \ p, q \in I \iff p \sqcup q \in I
\]
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 \(\bot\), and let \(\sqsubseteq_I\) be the following ordering relation on the set \(\mathcal{I}(P)\) of ideals of \(P\):

\[
\forall I, J \in \mathcal{I}(P). \ I \sqsubseteq_I J \text{ iff } I \subseteq J
\]

Then: \((\mathcal{I}(P), \sqsubseteq_I)\) is a complete lattice, the so-called lattice of ideals of \(P\), with join operation \(\sqcup_I\) defined by

\[
\forall I, J \in \mathcal{I}(P). \ I \sqcup_I J =_{df} \{ p \in P | \exists i \in I, j \in J. \ p \sqsubseteq i \sqcup j \}
\]

and meet operation \(\sqcap_I\) defined by

\[
\forall I, J \in \mathcal{I}(P). \ I \sqcap_I J =_{df} I \cap J
\]

and with least element \(\{\bot\}\) 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. \(\mathcal{P}\mathcal{I}(P) =_{df} \{\downarrow q \mid q \in P\}\) 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:

\[\forall q, r \in P. \quad \downarrow q \cap_{\mathcal{I}} \downarrow r = \downarrow (q \cap r) \land \downarrow q \cup_{\mathcal{I}} \downarrow r = \downarrow (q \cup r)\]
The Sublattice of Principal Ideals

Theorem A.7.2.11 (Sublattice of Principal Ideals)

Let \((P, \sqsubseteq)\) be a lattice with least element, let \((\mathcal{I}(P), \sqsubseteq_I)\) be the complete lattice of ideals of \(P\), let \(\mathcal{P}_I(P)\) be the set of the principal ideals of \(P\), and let \(\sqsubseteq_{PI}\) be the restriction of \(\sqsubseteq_I\) onto \(\mathcal{P}_I(P)\). Then:

\[
(\mathcal{P}_I(P), \sqsubseteq_{PI}) \text{ is a sublattice of } (\mathcal{I}(P), \sqsubseteq_I).
\]

Note: The sublattice \((\mathcal{P}_I(P), \sqsubseteq_{PI})\) of \((\mathcal{I}(P), \sqsubseteq_I)\) is usually not complete, not even if \((P, \sqsubseteq)\) is complete.

(The lattice \((\mathbb{Z}, \leq)\), e.g., enriched with a least element \(\bot\) and a greatest element \(\top\) is complete, while the lattice of its principal ideals \((\mathcal{P}_I(\mathbb{Z}), \subseteq_{PI})\) 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

$$e_{\mathcal{I}} : P \to \mathcal{PI}(P)$$

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

\[\blacktriangleright\]
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
Cuts

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. \(C(P)\) 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. \(LB(\{q\}) = \downarrow q = df \{p \in P \mid p \sqsubseteq q\} \in C(P)\)
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 \(P\):
   \[
   \forall q \in P. \langle q \rangle = df \{p \in P \mid p \sqsubseteq q\} = LB(\{q\}) \in C(P)
   \]
   (or: \(\forall Q \subseteq P. \ Q \in \mathcal{P}\mathcal{I}(P) \Rightarrow Q \in C(P)\))
2. Lemma A.7.3.2(2) characterizes the principal ideals of \(P\) in terms of the function composition \(LB \circ UB\).
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. \(\downarrow q =_{df} \text{LB}(\text{UB}(\{q\}))\) 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 \(C(P)\) of cuts of \(P\):

\[
\forall C, D \in C(P). \ C \sqsubseteq_C D \iff C \subseteq D
\]

Then: \((C(P), \sqsubseteq_C)\) is a complete lattice, the so-called lattice of cuts of \(P\), with join operation \(\sqcup_C\) defined by

\[
\forall C, D \in C(P). \ C \sqcup_C D =_{df} \bigcap \{E \in C(P) \mid C \cup D \subseteq E\}
\]

and meet operation \(\sqcap_C\) defined by

\[
\forall C, D \in C(P). \ C \sqcap_C D =_{df} C \cap D
\]

and with least element \(\{\bot\}\) 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 $(C(P), \sqsubseteq_C)$ be the complete lattice of its cuts. Then:

The map

$$e_C : P \to \mathcal{PC}(P)$$

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 $(C(P), \sqsubseteq_C)$ be the complete lattice of its cuts, and let $e_C : P \rightarrow \mathcal{PC}(P)$ be the map of Theorem A.7.3.7. Then:

$(\mathcal{PC}(P), \sqsubseteq_{\mathcal{PC}})$ is a sublattice of $(C(P), \sqsubseteq)$ and $e_C$ 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 \((P, \sqsubseteq)\) 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 \(e_C : P \to \mathcal{P}C(P)\) (of Theorem A.7.3.7) defined by

\[
\forall p \in P. \ e_C(p) = \text{def} \ \text{LB}(\text{UB}({p}))
\]

is an order isomorphism between \(P\) and the partial order \((\mathcal{P}C(P), \subseteq)\) of the principal cuts of \(P\), or, equivalently, the map \(e_C : P \to \mathcal{D}(P)\) defined as above is a partial order embedding of \((\mathcal{P}C(P), \subseteq)\) 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 \((\mathcal{P}C(P), \sqsubseteq_{\mathcal{P}C})\) 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

$$\downarrow p = \text{df } \{ r \mid r \subseteq p \} \in \mathcal{PD}(P)$$

yields an

- embedding of $P$ into $\mathcal{PD}(P) = \text{df } \{ \downarrow q \mid q \in P \}$, i.e., a function $e : P \rightarrow \mathcal{PD}(P)$ with

$$\forall p, q \in P. \ p \subseteq q \iff \downarrow p \subseteq \downarrow q$$
From Monotonic to Continuous Functions

...completion is the key to Theorem A.7.5.1:

Let \((P, ⊑_P)\) be a partial order, let \(\downarrow q =_{df} \{p ∈ P \mid p ⊑ q\}\) for \(q ∈ P\), let \(\mathcal{PD}(P) =_{df} \{\downarrow q \mid q ∈ P\}\), and let \((C, ⊑_C)\) be a CPO.

**Theorem A.7.5.1 (From Monotonicity to Continuity)**

A monotonic function \(f ∈ [P^{mon} \rightarrow C]\) can uniquely be extended to a continuous function \(\hat{f} ∈ [\mathcal{PD}(P)^{con} \rightarrow C]\).
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 ‘· is a prefix of ·’ on \( L \) defined by

\[
\forall l, l'' \in L. \, l \sqsubseteq_{pfx} l'' \iff df \ l = l'' \lor (l \text{ finite } \land \exists l' \in L. \, l + + l' = l'')
\]

Then: \((L, \sqsubseteq_{pfx})\) 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. \( \downarrow l =_{df} \{ l' \in L \mid l' \sqsubseteq_{pfx} l \} \) 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
Appendix A: Further Reading (1)


Appendix A: Further Reading (2)


Appendix A: Further Reading (3)


- 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)


George Grätzer. *General Lattice Theory*. Birkhäuser, 2nd edition, 1998. (Chapter 1, First Concepts; Chapter 2, Distributive Lattices; Chapter 3, Congruences and Ideals; Chapter 5, Varieties of Lattices)

Appendix A: Further Reading (5)

- Paul R. Halmos. *Naive Set Theory*. Springer-V., Reprint, 2001. (Chapter 6, Ordered Pairs; Chapter 7, Relations; Chapter 8, Functions)
Appendix A: Further Reading (6)


Appendix A: Further Reading (7)


Appendix A: Further Reading (8)


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)